SEC660 | ADVANCED PENETRATION TESTING, EXPLOIT WRITING, AND ETHICAL HACKING

660.5

Exploiting Windows for Penetration Testers

SANS

SEC660.5

Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

SANS

Exploiting Windows for Penetration Testers

Exploiting Windows f r Penetration Testers - 660.5

In this section, we focus primarily on the Windows OS, performing many of the same attack styles we performed on Linux. There are many differences when exploiting the Windows OS, which become quite evident as we progress through the different techniques.

Table of Contents	PAGE
Introduction to Windows Exploitation	04
Windows OS Protections and Compile-Time Controls	36
Windows Overfows	62
EXERCISE: Basic S t Overfow	63
EXERCISE: SEH Ove r ite	103
Defeat ng Hardware DEP with ROP	131
Demonstration: De f ating Hardware D E Prior to Windows 7	140
EXERCISE: Using ROP to Disable DEP on Windows 7/8 10	167
Building a Me t sploit Module	201
Windows Shellcode	222
Bootcamp	235

Table of Contents

This slide serves as the Table of Contents f r 660.5.

A Word about This Section

- You do not have to write your own exploits to take full advantage of this material. We will learn to:
  - Repair broken Metasploit scripts with invalid code reuse addresses (trampolines)
  - Understand how to use return-oriented programming (ROP) and exploitation to defeat Windows 7/8/10 controls
  - Determine how to f nd bad characters
  - Search for stack over f ow vulnerabilities
  - Understand modern OS controls such as Control Flow Guard (CFG) and Address Space Layout Randomization (ASLR)
  - Understand the internals of the Windows OS and how it di f ers f om Linux

A Word about This Section

A quick note about this section's material and 660.4's, for that matter: Many penetration testers have the mindset that they will never be responsible for writing custom exploits or per f rming 0-day bug hunting. Aside f om the f ct that it is a great skill to have and you may f nd yoursel f in that role in the f uture, penetration testers are o f en f ced with scripts that do not work and OS or compile-time protections thwarting your attack f om being successful. If you cannot repair a script to utilize ROP techniques to bypass DEP, someone else will succeed. It is not uncommon to run a Metasploit script against a seemingly vulnerable system, only to f nd that the exploit fails. Do you stop here and assume the system is safe? No! You must research f rther to see if your attack is f iling due to OS and compile-time controls, or bad memory addresses being chosen for code reuse attacks, such as with trampolines. Keep this in mind as we progress through the day.

Course Roadmap

- Network Attacks f r Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux f r Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Ove r ows

Exercise: Basic S t ck Ove r low

Exercise: SEH Ove r rite

Defeating Hardware DEP with ROP

Demonstration: Defeating Hardware DEP Prior to Windows 7

Exercise: Using ROP to Disable DEP on Windows 7/8/10

Building a Metasploit Module

Windows Shellcode

Bootcamp

SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 4

Introduction to Windows Exploitation

In this module, we take a look at the linking and loading process on Windows, the Windows API, and various inte r als, such as the Process Environment Block (PEB), Thread In f rmation Block (TIB), and Structured Exception Handling (SEH). Attackers use these areas within a process to per f rm such things as locating desired variables and addresses within memory and overwriting critical pointers, and to lea r the overall structure of a process. Penetration testers must have the same knowledge and skill set to determine if a process is vulnerable.

Objectives

- Our objective for this module is to understand:
  - Windows Overview
  - Linkers and Loaders
    - PE/COFF
  - Windows API
  - Thread Information Block (TIE)
  - Process Environment Block (PEE)
  - Structured Exception Handling (SEH)

Objectives

This module introduces Windows OS exploitation topics and concepts. We f rst focus on some of the di f erences between Windows and Linux from an operational perspective. This includes the linking and loading process on Windows, the Windows API, and other Windows-speci f c areas, such as the Thread In f rmation Block (TIB), Process Environment Block (PEB), and Structured Exception Handling (SEH). This module enables us to lay out the foundation needed to look for exploitation opportunities on Windows.

CPU Modes/Processor Access Modes

- Windows has two access modes:
  - Kernel mode - core operating system components, drivers
  - User mode - application code, drivers
- Kernel memory is shared between processes
- 32-bit Windows provides 2GB of virtual memo r to the kernel and 2GB to the user; however, there is an optional /3GB f ag 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, but is plenty f r now

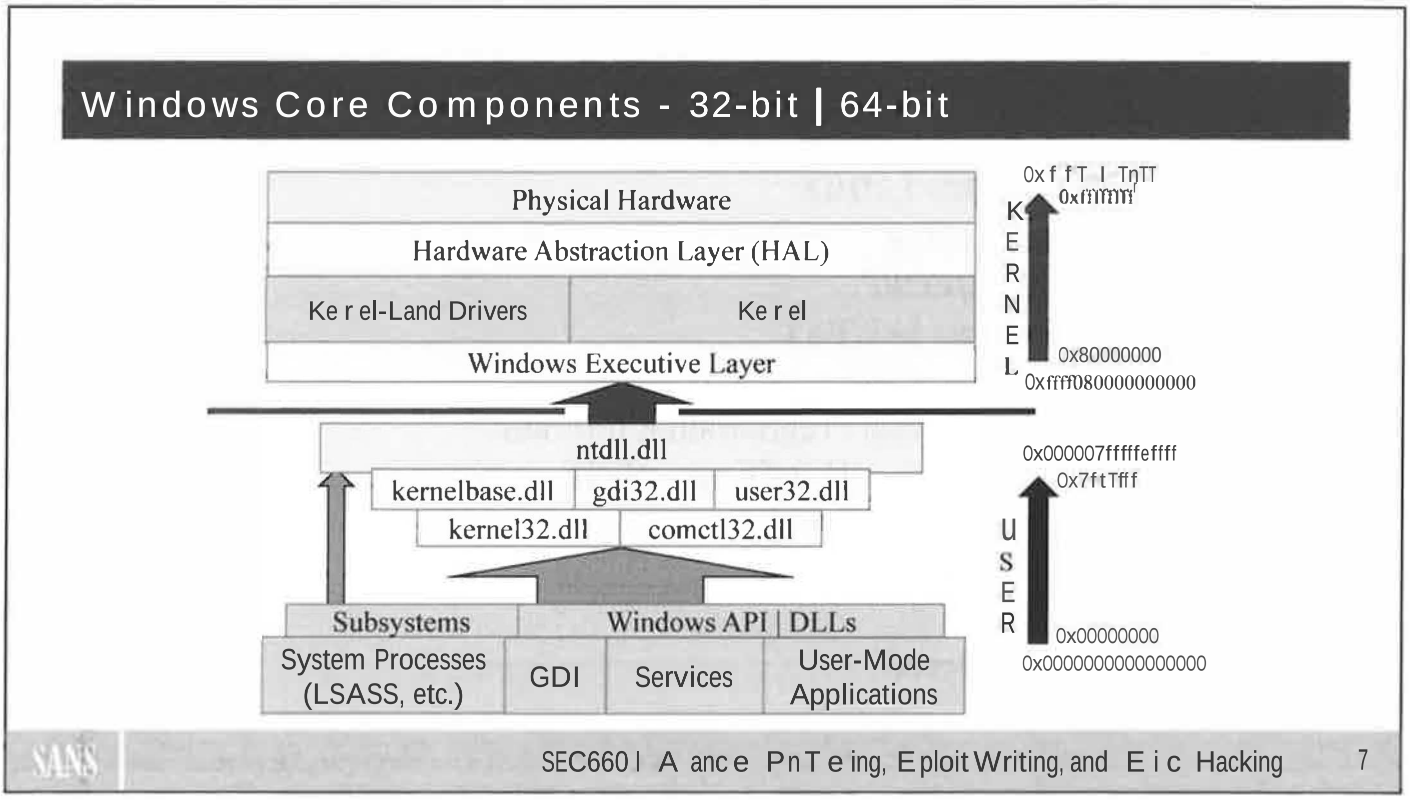
SANS SEC660 | Advanced Pen Testing, Exploit W riting, and Ethical Hacking 6

CPU Modes/Processor Access Modes

The majority of operating systems support a two-ring model. Ring 0, also known as kernel mode or kernel land, contains the kernel code that performs core operating system f nctionality such as the synchronization of multiple processors, access to hardware, and the handling of interrupts. The kernel uses shared memory and must be protected f om user applications. User mode, or Ring 3, is where user-mode applications run. If an exception occurs in user mode, it should be handled, even if it results in application termination. An exception in kernel mode can be catastrophic and lead to a system crash, requiring a reboot to recover.

Kernel memory shares a single address space, unlike user-mode applications. On 32-bit systems, the kernel can easily access all memory on all processes with unlimited control. On 64-bit systems, Ke r el Mode Code Signing (KMCS) was introduced, requiring a Certificate Authority (CA) to vouch for a driver. Drivers' code runs in kernel mode and exploitable vulnerabilities have historically resulted in full system control.

On Windows, 32-bit applications receive 4GBs of virtual memory. Memory address range 0x00000000 to 0x7f f ffff is f r user mode, and 0x80000000 to 0xffffffff is f r kernel mode. On 64-bit applications, 7TBs or 8TBs are given to the user and the kernel, depending on the architecture. User mode runs from 0x00000000000 to 0x7fff f'ffff Note that ITB is 2/40. Memory address range 0x80000000000 through 2 / 64 is a large block. Addressing is handled differently on the various architectures.



Windows Core Components - 32-bit | 64-bit

On this slide is a high-level view of the core components of the Windows OS. To the right are the address ranges f r both user mode and kernel mode. The larger address ranges are f r 64-bit applications on 64-bit versions of Windows, and the smaller address ranges are f r 32-bit applications. This diagram lacks much of the granularity of each component; however, it serves as a good overview. You can see various service and application types in the user space on the bottom, each required to go through various DLLs and APIs to access ke r el resources. The ke r el is protected in ring 0. There are multiple layers in ring 0, such as the Windows Executive, ke r el drivers, the kernel or micro-ke r el 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.

Windows Overview

- Windows versus Linux
  - Linking and loading
    - ELF versus PE/COFF
    - GOT/PLT versus IAT/EAT
  - Windows API
    - Windows Application Programming Inter f ce
  - Structured Exception Handling - SEH
    - Global exception handler
    - Try and Except/Catch blocks
  - Threading
    - fork() versus threads

SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking B

Windows Overview

Windows versus Linux

We have already covered how the linking and loading process works on Linux. In this section, we cover how the linking and loading process works on Windows. As Linux uses the ELF object f le format, Windows uses the Portable Executable/Common Object File Format (PE/COFF). They serve the same type of f nction, but of course, the implementation is different. The equivalent to Linux's use of the Procedure Linkage Table (PLT) and the Global Offset Table (GOT) on Windows is the Import Address Table (IAT) and the Export Address Table (EAT). Windows also suppotis a f rm of "Lazy Binding" with delayed symbol resolution. We cover this in the next section.

Many of this section's modules f cus on the differences between Windows and what we've already covered on Linux. Many of the principles and concepts are similar, but the execution is different. Much of this di f erence is due to the Windows Application Programming Interface (API) and various process or thread-specific constructs. With Linux, we use system calls to access f nctions within kernel address space. Linux system calls can be compared to the Windows APIs, where you have a collection of functions that allow f r privileged access outside of user address space. The Windows API is a collection of Dynamically Linked Libraries (DLLs) and f nctions that allow a programmer to write programs and utilize services on the Windows OS. More details on this soon.

Exception handling operates much more dynamically on the Windows OS. An event on Linux like a segmentation fault sends a signal SIGSEGV to the kernel, informing it of the invalid memory reference. The do_page Jault() funclioll is lhtn calltd to determine if the referenced address is within the process's address space. The result is usually to terminate the process. Windows uses a global exception handler, which walks through a set of one or more Try blocks, each containing one or more Catch blocks. If a proper handler is found, its code for that handler is executed. If no exception handler is f und, the de f ult handler will be called to terminate the process. We'll look at some examples of Windows Structured Exception Handling (SEH) coming up.

Where many *NIX OSs perform f rking, spawning a whole new process, Windows uses threading. A process on Windows can have multiple threads of execution within itself. These threads share the same address space as the parent process and can inherit attributes of the parent marked f r inheritance. Take an application that f rks a new process each time a user connects. For each instance, the entire process is copied to the new process, a process ID (PID) is assigned, and each process receives its own memory space. Threading allows f r the sharing of the process ID, addressing, and memory segments. The only exception is that each thread gets its own stack and Thread In f rmation Block (TIB). This makes f r much more eff cient use of memory and system resources.

Linking and Loading - PE/COFF (1)

- Windows object file format
- Two primary image file types:
  - Executable Format
  - Dynamic Link Libraries (DLL)
- Import Address Table
- Export Address Table
- .reloc section
  - Fix-ups
  - Relocation not common, although DLLs mapped into a process could conflict

SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 10

Linking and Loading - PE/COFF (1)

The object file format used by Microsoft Windows is PE/COFF. It is based on the original COFF format used by UNIX systems after the a.out format and prior to the ELF format still used today. The format is optimized to work in environments using paging and can be mapped directly into memory due to fixed sizing. The PE/COFF format has two primary image file types, the Executable Format and DLL format. The Microsoft equivalent of Shared Object files is Dynamic Link Libraries (DLLs).

PE/COFF files utilize an Import Address Table (IAT) and an Export Address Table (EAT). The IAT holds symbols that require resolution upon program runtime, and the EAT makes available the functions local to the program or library that may be used by other executable files or shared libraries. The IAT lists the symbols needing resolution from external DLL files contained on the system in which the program is run. The function name is included in the IAT, as well as the DLL file, which should contain the requested function. For example, if the program requires the function LoadLibraryA(), it includes the function name as well as the DLL file, kernel32.dll. The requested function is often obtained by using an ordinal value assigned based on its location within the table. For example, inside kernel32.dll is the function GetCurrentThreadId(), which is bound to the address 0x7C809737 on certain versions of Windows. The imports table holds this information and also references that function by an ordinal value such as 318.

Executable files do not typically export any symbols, although they may be visible. DLL files export the symbol information for each function they contain. A binary lookup is used to determine if the DLL file contains the requested function. The EAT makes available the relative virtual address of the requested function. The relative address must then be added to the load address to obtain the full 32-bit virtual memory address, or 64-bit address, respectively.

The PE/COFF header includes a COFF section that describes the contents of the PE f le. The relocation (.reloc) section file contains "f x-ups," holding in f r ation about which sections require relocation in the event there is a con f ict with addressing. Fix-ups a e not o f en necessary with modem Windows systems; however, if there are multiple DLLs loaded into memory that request the same address space, they will require relocating. During program runtime, the PE is loaded into memory. At this point the PE is called a module, and the location of the start address is called the HMODULE. The "H" stands f r Handle and is now synonymous with HINSTANCE.

PE/COFF (2)

- PE/COFF Primary Sections
  - DOS executable f le
    - MZ header - "4D SA"
    - Mark Zbikowski
    - Stub area
      - "This program cannot be run in DOS mode"
  - Signature
    - PE Signature - PE\0\0

PE/COFF (2)

PE/COFF Primary Sections

Similar to the ELF format, PE/COFF has multiple headers and sections that are read and loaded into memory during load-time.

DOS Executable File and Signature

The first item inside a PE/COFF object f le is the DOS MZ header. You will most commonly see the hex values 40 5A as the f rst value. The magic number 40 5A translates f om hex to ASCII as MZ, which stands for Mark Zbikowski, one of the original DOS developers. This header also has a stub area. An example of when the code in this stub area is executed is when a user attempts to run the f le under DOS. The following message would display in that case: "This program cannot be run in DOS mode". Also included in this header is a f eld that points to the PE signature. The PE signature is a 4-byte value that is always PE\0\0.

PE/COFF (3)

- PE/COFF Primary Sections, cont.
  - File header
    - 0x014c - Intel 386 requirement
  - Optional header
    - 0x010b - PE32 f rmat | 0x020b - PE64 f rmat
    - Image size
    - RVA o f set
    - Stack and heap requirements

PE/COFF (3)

File Header and Optional Header

When it has been veri f ed that the program is a valid executable that can run on the system, execution moves through the f le header. The f le header consists of a COFF section and optional header. You will most o f en find the value 0x014c, which tells the system that a minimum of an Intel 386 processor is needed to run the executable. Other data in this header includes a timestamp and relative virtual addressing info n ation.

The optional header always starts with the magic number 0x010b f r PE32 f rmat or 0x020b f r PE64 format. You can usually use this value to know where the beginning of the optional header is located and use the information for proper parsing of 32-bit or 64-bit f les. The optional header contains in f rmation about the size of the image, as well as the RVA o f set to where execution of the program should begin. O f set 20 f om the start of the optional header is a 4-byte value that shows the relative o f set of where the code section of the image will be loaded into memory, relative to the image base. This can change depending on the version. The optional header even speci f es how much space to reserve for the stack and heap.

PE/COFF (4)

- PE/COFF Primary Sections - cont.
  - Section table
    - ASCII section names:
      - .text, .idata, .rsrc, and so on
    - Memo r location of sections
    - Bounda r aligned
- Lazy linking
  - Similar to PLT and GOT relationship
  - Symbols may not be resolved until f rst call

SANS SEC660 I Advanced Pen Testing, Exploit Writing, and Ethical Hacking 1-

PE/COFF (4)

Section Table or Header

The section table or header gives us the ASCII name of the sections included in the executable, such as .text, .idata, .rsrc, and others, and provides us with infonnation as to where in memory they will be located. Sections are mapped into memory using a boundary alignment speci f ed in the optional header at offset 32, de f ulting to the architecture-speci f c page size.

Lazy Linking

Similar to the way in which ELF uses the PLT and GOT, Windows also allows for a form of lazy linking a f er program loading. Instead of including the entries in the .idata section to be automatically resolved during runtime, they can be loaded into a delay-load import table. This section holds the libraries and functions that will not be called until the f rst call to that f nction. A process can also utilize functions, such as LoadLibrary() and GetProcAddress(), to obtain a symbol's linear address when requested a f er program runtime has initiated. This is a common goal of an attacker's shellcode a f er obtaining the location of kernel32.dll. These f nctions can be used to load any desired library into memory.

Tool: OllyDbg

- So f ware debugger for Windows
- Author: Oleh Yuschuk
- Shareware!
- Binary code analysis
- Register contents, procedures, API calls, patching and more!

Tool: OllyDbg

OllyDbg is a so f ware debugger f r Windows. The tool was created by Oleh Yuschuk and is f eely available. Using the tool f r commercial use requires you to register.

OllyDbg has many features and also allows you to write your own plugins. When the source code of a program is not available, you must have a way to disassemble the code and understand what the code is doing. OllyDbg allows you to analyze and modi f the register contents, such as EAX, EIP, EDI, and so on. The disassembler pane allows you to inspect and modi f the assembly code of the compiled binary. API calls may be monitored and intercepted. OllyDbg even attempts to tell you the path of execution a program is going to take.

Tool: Immuni t Debugger

- Immunity - founded by Dave Aitel
  - https:// w .immunityinc.com/products/debugger/
  - Free debugger based o f of OllyDbg
- Extensive development work focused on reverse engineering and exploit development
- Combines command line and GUI
- Supports Python scripting

SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 16

Tool: Immunity Debugger

Immunity Debugger is another debugger option available. The tool is based on OllyDbg, so the layout should look familiar, along with many of the functions. The tool is maintained by Dave Aitel and his employees at Immunity. Immunity can be f und online at https://www.immunityinc.com/products/debugger/. The debugger is f ee and combines the GUI layout of OllyDbg with command-line support similar to WinDbg and ODS. Python scripting is also supported f r extensibility. The tool is aimed at reverse engineering and exploit development, claiming to cut down on exploit development time by 50%. A benef t to using Immunity Debugger is the easy ability to import debugging symbols when necessary. OllyDbg can be quite troublesome when trying to connect to the Microso f Symbol Store or to a local symbol store.

Tool: PEdump

- PEdump for PE f le examination
- Author: Matt Pietrek
- Freeware!
- Display PE header data
- Display section tables
- Display symbol tables

Tool: PEdump

PEdump is a freeware tool created by Matt Pietrek that displays Portable Executable structure data in an easy-to-read format. The tool enables you to view all headers within a PE f le, as well as the section tables, contents, RVA information, symbol tables, relocation information, and more. There are many tools that perform a similar f nction, such as Dumpbin, but PEdump is an e f icient one to use when collecting object file in f rmation on Windows.

NOTE

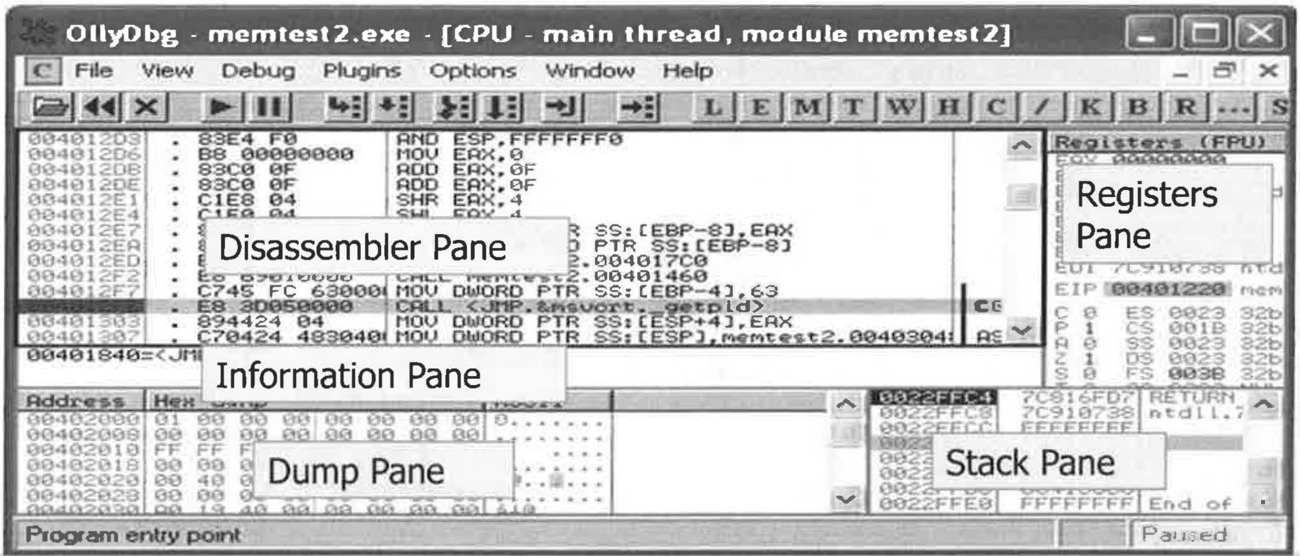
Windows memory addressing on your system may be entirely different. This is due to the many different service packs and patch levels available.

SANS SEC 6 | A d nc e P n T e ng. E loit W r n a n Ethical Hackng 1e

NOTE

You need to understand that unlike our exercises on Linux, the memory addressing in Windows may vary from what you see on the slides. Windows constantly updates its DLLs and APIs resulting in changes to their location in memory when loaded. If the addressing on your system is di ferent than what is shown on the slides, this is completely normal and all techniques and labs still work. You may sometimes be required to search for a particular construct or opcode in memory, which is typical. Ask your instructor if you have any trouble f nding something.

PE/COFF Walkthrough (I)



SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 19

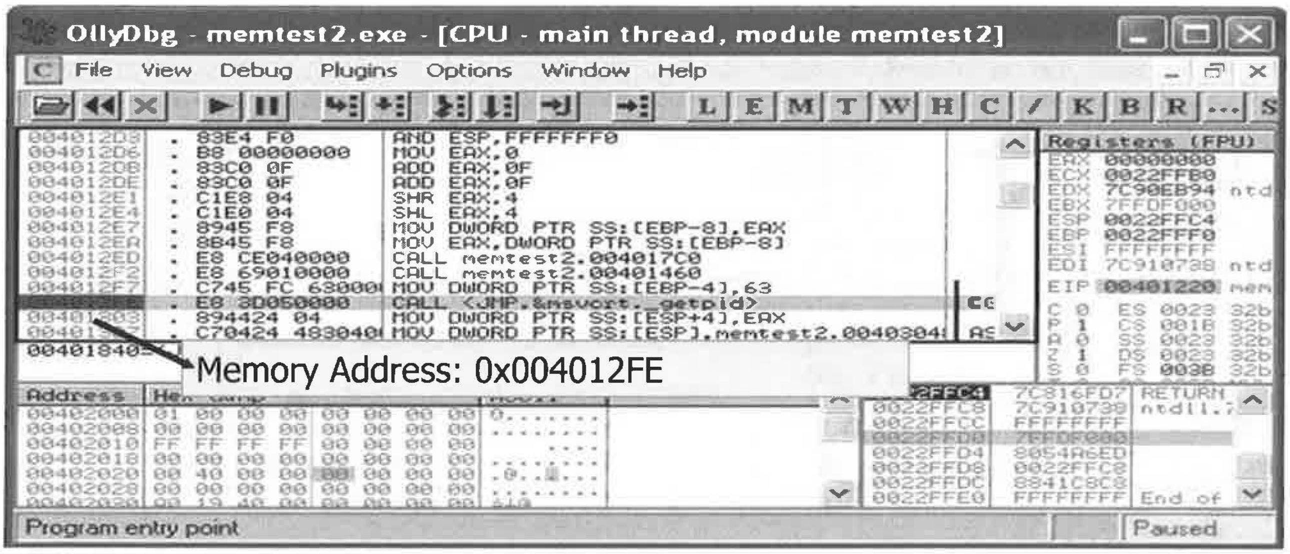
PE/COFF Walkthrough (1)
Let's now go through the symbol resolution on a Windows system using the PE/COFF file format. We track the resolution of the function getpid(), which is located in msvcrt.dll. The getpid() function is used by the same memtest application we used on Linux. A compiled version of this program for Windows exists on the USB and is named memtest2.exe.

In this example, OllyDbg is used to open the program. You can perform the same with Immunity Debugger. Spend some time getting familiar with the different panes within OllyDbg and Immunity Debugger, and notice the five primary sections within the CPU window:

- Disassembler Pane: This pane shows the memory locations and assembly instructions of the loaded or attached program.
- Information Pane: The information pane decodes arguments.
- Dump Pane: The dump pane shows the contents of memory.
- Registers Pane: The registers pane shows the contents of a large number of CPU registers. For example, the EIP register holds the address of the instruction currently being executed.
- Stack Pane: The stack pane shows the stack and the location where ESP is currently pointing.

Before moving to the next slide, see if you can locate where the call to the getpid() function is located. If you find it, click once on the memory address of that instruction on the left, and press the F2 key. F2 sets a breakpoint. The highlighted memory address should turn red at this point to show that a breakpoint has been set. By setting a breakpoint, you are telling the debugger to pause program execution when a call to that function has been reached. If you were unable to locate the call to the getpid() function, turn to the next slide for the location.

PE/COFF Walkthrough (2)

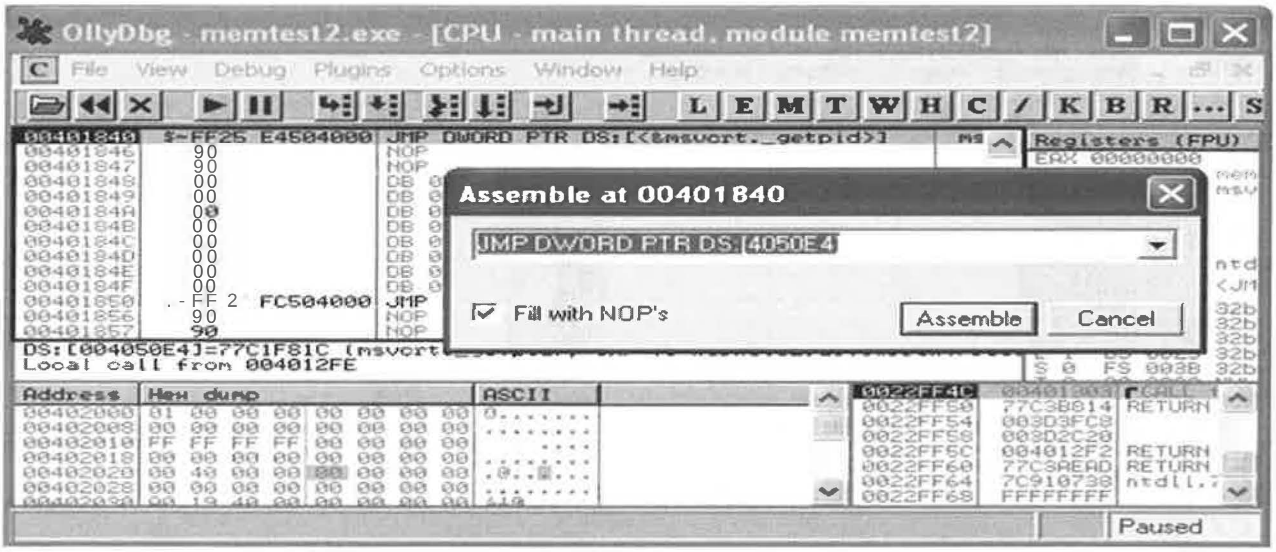


SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 20

PE/COFF Walkthrough (2)

This slide shows our breakpoint set at the memory address 0x004012FE, which is the location of our program's call to the getpid() function on the OS used for the presentation. At this point, you should click the blue Play button toward the top le f of the CPU window. Pressing F9 per f rms the same function and simply means Run. If you set up the breakpoint properly, the call to getpid() should be highlighted, and the program should show as Paused on the bottom right of the screen. Press F7 once to step into the next instruction located at the destination call address. F7 is the command to per f rm a single step of one instruction.

PE/COFF Walkthrough (3)



SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 21

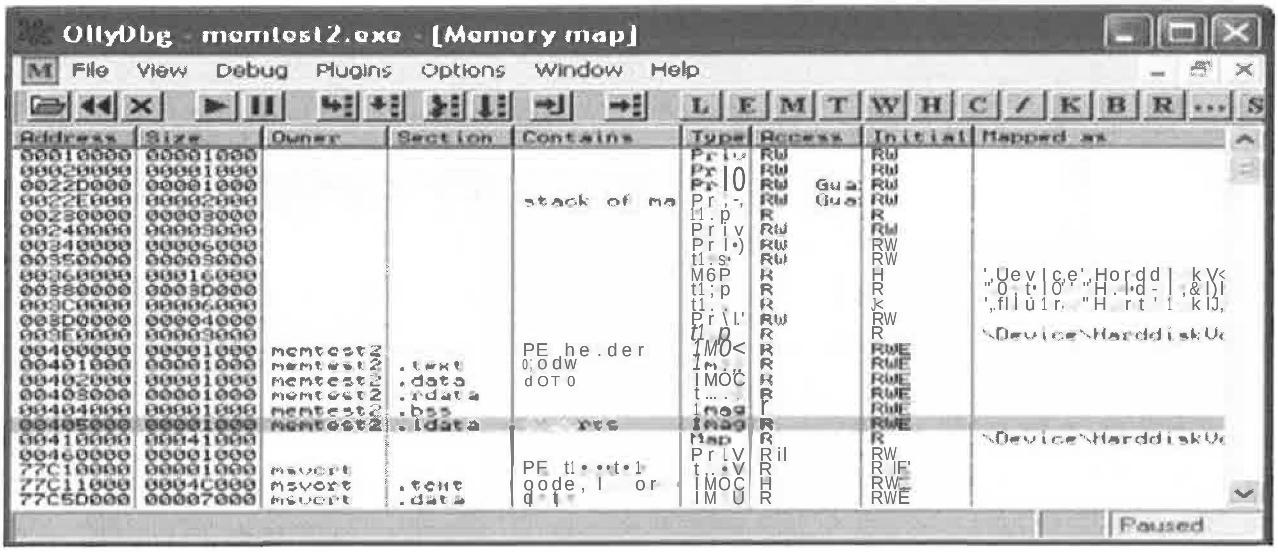
PE/COFF Walkthrough (3)

Notice that we have now jumped to the instruction at the address 0x00401840. Can you figure out where in memory we have jumped? Try to determine where execution may have taken us before moving on. The next paragraph provides you with the answer. You can also try clicking the M button on the main Ribbon bar. It takes you to the memory map of the program, where you can easily identify addressing ranges. Click C at any time to go back to the CPU window.

You may have noticed that we're still within the code or text segment. This section of the code segment is a table of indirect jumps covering each entry in the Import Address Table (IAT), as shown in the idata section. To the calling function, the address provided is the location of the desired function and is transparent. Double-click the instruction JMP DWORD PTR OS:[<&msvcrt._getpid>] and you should get the pop-up box shown on the slide. The address 0x004050E4 shows up and will be the destination of this jump instruction. However, this is a pointer to the next address toward our desired function, getpid(). If you take a look in the information pane, you can notice that it says, DS:[004050E4]=77C1F81C, which should be the true address of the resolved symbol. But wait! Where did that address come from?

Click the Cancel button and press F7 once. We'll look up the source of this address on the next slide.

PE/COFF Walkthrough (4)



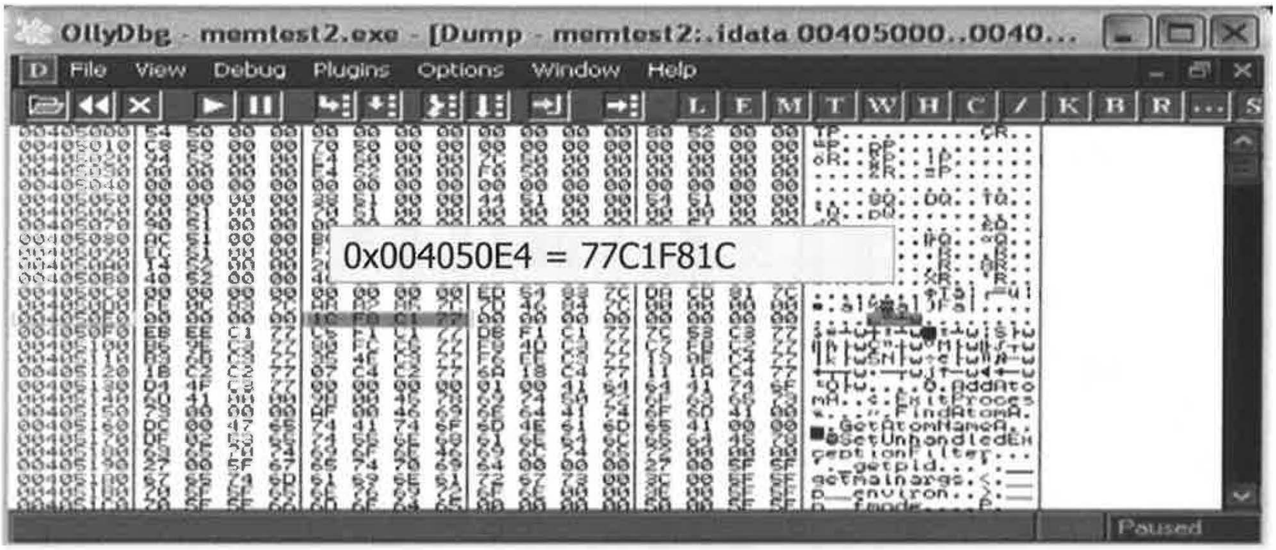
SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 2

PE/COFF Walkthrough (4)

In the last step, we discovered that the pointer was taking us to the address 0x004050E4. Again, your OS may be slightly di ferent and you may have to compensate for that in your research. To determine what resides at this memory location, we must navigate to the appropriate section. This address is obviously out of bounds for the code segment. From the main CPU window in OllyDbg or Immunity Debugger, click the View menu option and select Memory from the list of choices. You should see the same window as shown on this slide. Locate the base address where 0x004050E4 will fall. What section does our pointer take us to?

If you said the .idata section, you are correct! As mentioned earlier, the .idata section maps symbol names we need to the appropriate memory addresses by working with the Import Address Table (IAT). Right-click on the .idata section and select dump from the options. Proceed to the next slide.

PE/COFF Walkthrough (5)



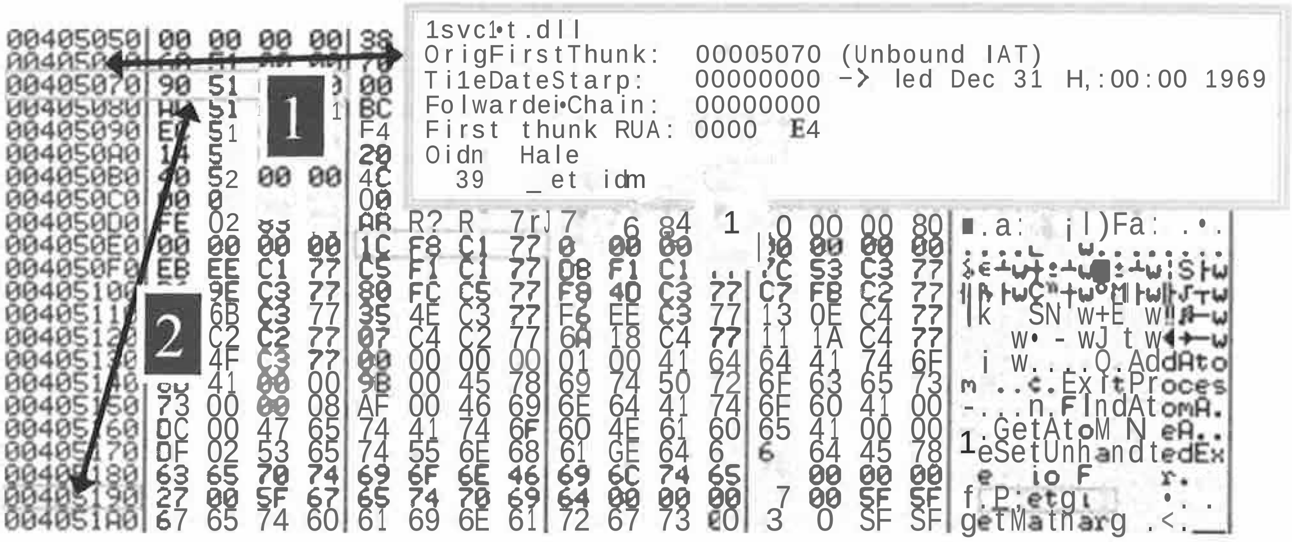
SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 2

PE/COFF Walkthrough (5)

You should have a window that matches this slide. Locate the memory address from our pointer at 0x004050E4. You should have the address 77C1F81C or similar held in that location, depending on your version of Windows. Remember that x86 uses little endian format. This address should be the location of the getpid() f nction. By viewing the in f rmation pane within the main CPU window, we can see that at the memory location 0x77C1F81C, we are actually forwarded to the address 0x7C809920, which lies within kernel32.dll. It so happens that the getpid() function residing in msvcrt.dll f rwards us to the f nction getcurrentprocessid() residing in kernel32.dll.

On the next slide, we take a look at the IAT and how the table re f rences each other.

PE/COFF Walkthrough (6)



SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 24

PE/COFF Walkthrough (6)

On this slide, we see two separate screenshots. The screenshot outlined in red is information displayed by the PEdump tool. This tool is located on your tools CD. You should create a folder in the Program Files directory called PEdump. Unzip the pedump.zip file from your tools CD, and extract all files to your newly created PEdump folder on your system.

From command line, navigate into your C:\Program Files\PEdump\ directory and type the command:

```
pedump /S <path to the memtest2.exe program>
```

Scroll down until you see the Imports Table. You should now locate the data from this screenshot. The larger screenshot is from the same OllyDbg dump as the last slide.

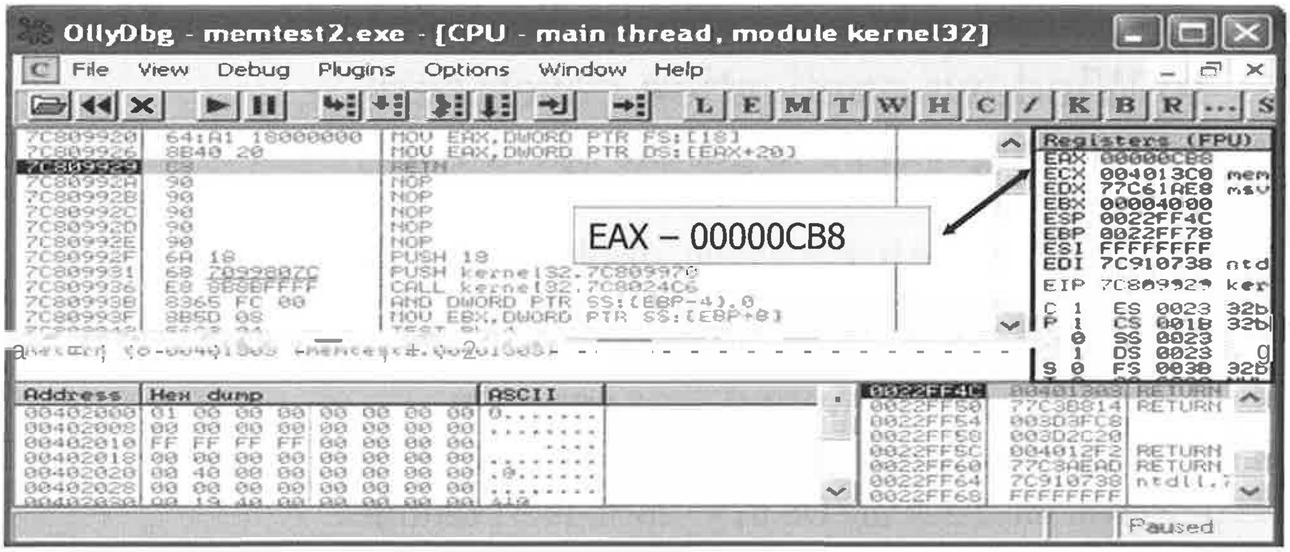
Box 1: Here we see that the unbound IAT entry in the Imports Table created by the original linker is pointing us to the relative address 0x5070. At 0x5070 we see another relative memory address of 0x5190.

Box 2: At the relative location of 0x5190 we see the symbol name _getpid decoded for us on the right.

Box 3: In box 3 we see the address from the First thunk RVA entry in the Imports Table points us to the relative address 0x50E4. If we look at 0x50E4 in the OllyDbg dump of that .idata memory space, we see that it points us to 0x77C1F81C, the location of getpid() in msvcrt.dll.

Proceed back to the main CPU window in OllyDbg, and press F7 three times until you hit the RETN instruction. You will be taken to the address shown on the next slide.

PE/COFF Walkthrough (7)



PE/COFF Walkthrough (7)

We have now been taken to the memory address 0x7C809920. Again, this may be a di f erent address on your system. We can see that by stepping through the instructions at 0x7C809920, EAX results in the value 0x00000CB8. This value in decimal is 3256, which is the process ID retu r ed f om the getcurrentprocessid() f nction. **Note: This process ID will be different each time the program is run.**

Also note the first three lines of disassembly at the top of the disassembler pane:

```
MOV EA , DWORD PTR FS: (18]
MOV EA , DWORD PTR DS: [E A +20]
RETN
```

The FS segment register points to the Thread In f rmation Block (TIB). In the f rst instruction, we dere f rence o fset 0x18 from the TIB into EAX. At this location is a self-referencing pointer, or simply the full linear address of the TIB. In the second line, we are dereferencing offset 0x20, which is the location in the TIB that holds the PIO f r the process. FS o fset 0x0 holds a pointer to the SEH chain, and FS offset 0x30 holds a pointer to the PEB. To view the TIB, simply note the address being moved into EAX during the first instruction, go to the memory map, and double-click the address.

We have now walked a function call on a Windows system using the PE/COFF object f le format.

The Windows API

- Set of compiled functions and services provided to Windows application developers
  - Makes it possible to get the operating system to do something
    - That is, write to the screen, display a menu, open a window, open a port, and so on
  - Provides services such as network services, Registry access, and command-line services
  - Windows is entirely closed-source
    - You must ask the OS to perform most routines

SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 26

The Windows API

The Windows API is the interface used to provide access to system resources. Functions are grouped together into Dynamically Linked Libraries (DLLs) and compiled. To get Windows to pretty much do anything, you must program an application to the rules of interacting with the many APIs. This includes services such as opening a window, accessing drop-down menus, opening up network ports, accessing command-line utilities, graphics utilities, Registry access, and pretty much anything else. If you want to do something simple such as open a file on a Windows system, you cannot do this without using the appropriate Windows API and having the operating system do it for you.

A compiler that has access to Windows symbol information is needed to create a proper Import Address Table and resolve names during linking and loading time. The functionality of the Windows API and dynamic linking enables Microsoft Windows developers to modify underlying functions within the DLLs without affecting application functionality. This also means that the location of functions when DLLs are modified may change, and they often do. As long as you have the symbol information, your application will still work properly on the various versions and service packs of Windows. From the opposite side, exploit code referencing static addresses such as functions inside of kernel32.dll will often fail because the address of this library and function offsets often changes. There are ways to write shellcode that do not rely on static addressing, and we will get to that a bit later.

Thread Information Block or Thread Environment Block

- Thread Information Block (TIB) / Thread Environment Block (TEB)
  - Stores information about the current thread
    - FS:[0x00] - Pointer to SEH chain
    - FS:[0x30] - Address of PEB
    - FS:[0x18] - Address of TIB
  - Takes away the requirement to make an API call to get structural data
  - Each thread has a TIB

SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 27

Thread Information Block or Thread Environment Block

You see the Thread Information Block (TIB), also referenced as the Thread Environment Block (TES). They are synonymous. The TIB is a structure of data that stores information about the current thread, and every thread has one. The FS segment register holds the location of the TIB. Rather than calling a function such as getprocessid(), the PIO of the process can be found at FS:[0x20] within the TIB. These values can be found at DWORD boundaries from offset FS:[0x00].

The TIB holds a large amount of information about the current thread; however, some common offsets from FS:[0x00] that we are interested in follow:

FS:[0x00] - Pointer to Structured Exception Handling (SEH) chain. When an event occurs that requires the global exception handler to be called, this pointer is pushed onto the stack to begin the exception handling process.

FS:[0x30] - Address of Process Environment Block (PEB).

FS:[0x18] - Address of Thread Information Block (TIB). This is a self-referencing pointer.

Process Environment Block

- Process Environment Block (PEB)
  - Structure of data with process-specific information
    - Image base address
    - Heap address
    - Imported modules
      - kernel32.dll is always loaded
      - ntdll.dll is always loaded
  - Overwriting the pointer to RTL_CRITICAL_SECTION was historically a common attack
    - The PEB was located at 0x7FFDF000
    - 0x7FFDF020 held the FastPebLock Pointer
    - 0x7FFDF024 held the FastPebUnlock Pointer

Process Environment Block

The Process Environment Block (PEB) is a structure of data in a process user address space that holds information about the process. This information includes items such as the base address of the loaded module (HMODULE), the start of the heap, imported DLLs, and much more. A pointer to the PEB can be found at FS:[0x30]. Because the PEB has modifiable attributes, you can imagine that it is a common place for attacks. Windows shellcode often takes advantage of the PEB because it stores the address of modules such as kernel32.dll. If the shellcode can find kernel32.dll's address in memory, it often gets the location of the function GetProcAddress() and uses that to locate the address of desired functions.

A common legacy attack on the PEB was to overwrite the pointer to RTL_CRITICAL_SECTION. This technique has been documented several times. Critical Sections typically ensure that only one thread is accessing a protected area or service at once. It allows access only for a fixed time to ensure other threads can have equal access to variables or resources monitored by the Critical Section.

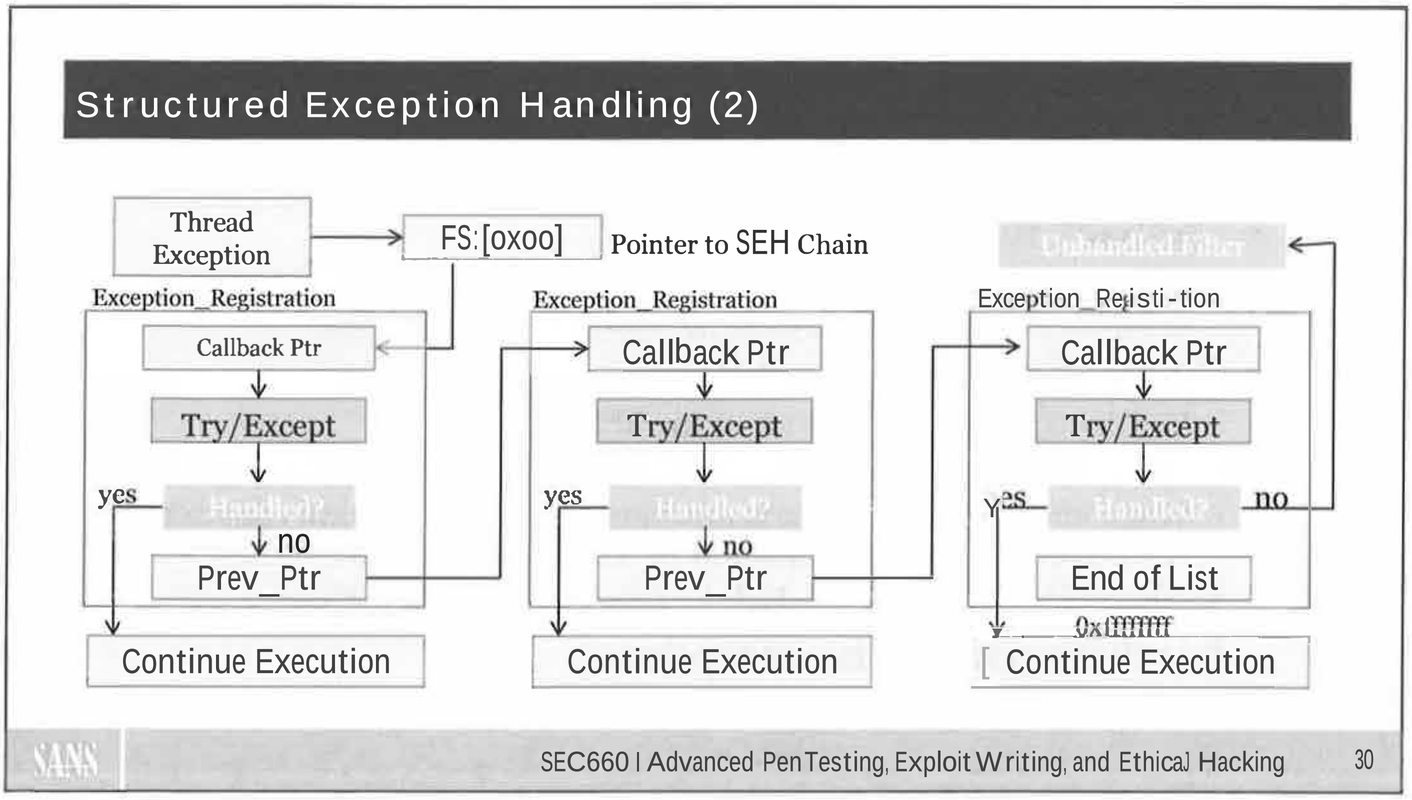
Structured Exception Handling (I)

- Structured Exception Handling (SEH)
  - Callback function
    - Allows the programmer to define what happens in the event of an exception such as print a message and exit or fix the issue
  - Chain of exception handlers
    - FS:[0x00] points to the start of the SEH chain
    - List of structures is walked until finding one to handle the exception
    - When one is found, the list is unwound, and the exception registration structure at FS:[0x0] points only to the callback handler
  - UnhandledExceptionFilter is called if no other handlers handle the exception
    - Terminates the process

Structured Exception Handling (I)

Exception handling in Windows can be more complex than on Linux. The pointer stored at FS:[0x00] inside the TIB points to an EXCEPTION_REGISTRATION structure that is part of a linked list of exception handlers and structures. If an exception occurs within a programmer's code, the Windows operating system uses a callback function to allow the program the chance to handle the exception. If the first structure can handle the exception, a value is returned indicating the result of the handling function. If the result is a continue execution value, the processor may attempt to retry the set of instructions that caused the exception to occur. If the handler declines the request to handle the exception, a pointer to the next exception handling structure is used.

Programmers can define their own exception handling within a program and choose to terminate the process, print out an error, perform some sort of action, or pretty much anything else you can do with a program. If these programmer-defined handlers or compiler handlers do not handle the exception, the default handler picks up the exception and terminates the program as stated. The image on the next slide helps to visualize the layout in memory.



Structured Exception Handling (2)

This diagram provides a visual representation of the layout of the SEH chain in memory. First, an exception must occur within a thread. Each thread has its own TIB and there are its own exception handling structure. When an exception occurs, the operating system needs to know where to obtain the callback function address. This is achieved by accessing offset FS:[0x00] within the thread's TIB. The address held here gives us the first exception registration structure to call. Inside this structure is a callback pointer to a handler. If the code is handled by the handler, a continue_execution value is returned and execution continues. If the exception is not handled, a pointer to the next structure in the SEH chain is cailed. Following this same process, the SEH chain unwinds until a handler handles the exception or the end is reached. If the end is reached, the Windows Unhandled_Exception_Handler handles the exception, terminating the process or giving the option to debug when applicable.

WOW64

- Windows 32-bit on Windows 64-bit
  - Many applications are still 32-bit
  - Emulator/subsystem that supports 32-bit applications on 64-bit systems
  - Supported by the majority of Windows 64-bit OSs
  - Set of user-mode DLLs to handle calls to and f om 32-bit processes

SANS SEC660 | Advanced Pen Testing, Exploit Writing, a d Ethical Hacking 31

WOW64

The majority of developers and vendors are still catching up to 64-bit systems. Microso f has ported and recoded its entire operating system to run on 64-bit processors in 64-bit mode natively. Large vendors such as Adobe have also completed, or are well on their way to fully supporting, 64-bit systems natively with their applications. However, many vendors are still far from converting over, and many companies would not simply run out to purchase a new license just because it's written f r 64-bit systems. This being the case, there must be support for 32-bit applications on 64-bit systems. On 64-bit Microso f OSs, the f ature to accomplish this requirement is Windows 32-bit On Windows 64-bit (WOW).

WOW is a collection of Dynamic Link Libraries (DLLs) that run within a 32-bit process and f lly emulate all requirements f r the 32-bit application. DLLs such as WoW64.dll run within the 32-bit process to intercept and translate calls. All required 32-bit DLLs are loaded into the application as needed to support f ll f nctionality.

A good paper and reference f r this slide on WOW64 can be found at http://msdn.microso f .com/enus/windows/hardware/gg46305 l .

Module Summary

- Some important differences between Linux and Windows
- The PE/Common Object File Format
- The Windows API is a complex set of libraries and functions
- TIB/TEB and PEB structures
- Exception handling with SEH

Module Summary

In this module, we took a high-level look at some of the differences between Windows and Linux. These differences will become more apparent as we go deeper into each area from an exploitation and security perspective. Notably, the use of the Windows API is a big difference from having the ability to directly access kernel resources through system calls within a process, as on Linux. There are many structures holding metadata and maintaining sanity within the process that must be protected.

Review Questions

1. What is stored at FS segment register offset FS:[0x30]?
2. What DLLs are almost always loaded as modules for a program?

Review Questions

1) What is stored at FS segment register o f set FS:[0x30]?

2) What DLLs are almost always loaded as modules f r a program?

Answers

1. The Process Envronment Block (PEB)
2. kernel32.dll, kernelbase.dll, and ntdll.dll

Answers

1) The Process Environment Block (PEB): The PEB is stored at location FS:[0x30] from within the Thread In f rmation Block {TIB).

2) kernel32.dll, kernelbase.dll, and ntdll.dll: These DLLs are critical to the Windows Subsystem and are almost always loaded.

Recommended Reading

- *Linkers and Loaders*, by John R. Levine, 2000
- *Assembly Language for Intel-Based Computers*, 5th Edition, by Kip R. Irvine, 2007
- "A Crash Course on the Depths of Win32 Structured Exception Handling," by Matt Pietrek: http://bytepointer.com/resources/pietrek_crash_course_depths_of_win32_seh.htm
- "The Forger's Win32 API Programming Tutorial," by Brook Miles (Forgey): http://www.winprog.org/tutorial/

Recommended Reading

Linkers & Loaders, by John R. Levine, 2000

Assembly Language for Intel-Based Computers, 5th Edition, by Kip R. Irvine, 2007

"A Crash Course on the Depths of Win32 Structured Exception Handling," by Matt Pietrek:
http://bytepointer.com/resources/pietrek_crash_course_depths_of_win32_seh.htm

"The Forger's Win32 API Programming Tutorial," by Brook Miles (Forgey): http://www.winprog.org/tutorial/

Course Roadmap

- Network Attacks for Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux for Penetration Testers
- Exploiting Windows for Penetration Testers
- Capture the Flag Challenge

Day 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

- Exercise: Basic Stack Overflow
- Exercise: SEH Overwrite

Defeating Hardware DEP with ROP

- Demonstration: Defeating Hardware DEP Prior to Windows 7
- Exercise: Using ROP to Disable DEP on Windows 7/8/10

Building a Metasploit Module

Windows Shellcode

Bootcamp

SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 36

Windows OS Protections and Compile-Time Controls

In this module, we walk through protection mechanisms added to the various Windows operating systems over the years. It is important to understand each of these protections to better understand what you are up against when attempting to defeat or circumvent them. Some of the protections can be defeated, and others can simply be bypassed or disabled. When performing penetration testing, an exploit may fail against a system that should be vulnerable. This may be due to one or more protections that can potentially be defeated. Each possible situation should be ruled out.

Objectives

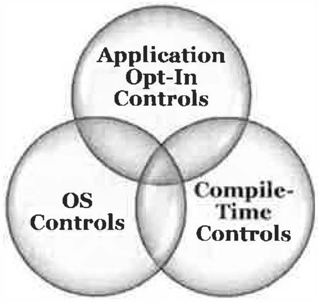
- Our objective for this module is to understand:
  - Data Execution Protection (DEP) and W^X
  - Security cookies and stack canaries
  - PEB randomization
  - Heap cookies
  - Safe unlinking
  - Windows Low Fragmentation Heap (LFH)
  - ASLR on Vista, 7/8/10 and Server 2008/2012/2016
  - ...and many others

Objectives

Our objective for this module is to understand many of the modem exploit mitigations that you are likely to come across when performing penetration testing and exploitation.

Exploit Mitigation Controls

- Controls to mitigate the successful exploitation of a software vulnerability
- Three primary categories:
  - Application Opt-In Controls – /dynamicbase, DEP
  - OS Controls – ASLR, DEP
  - Compile-Time Controls – Canaries, SafeSEH
- Often have strict requirements to be effective:
  - One bad module can break the whole protection
  - Better security when using multiple categories

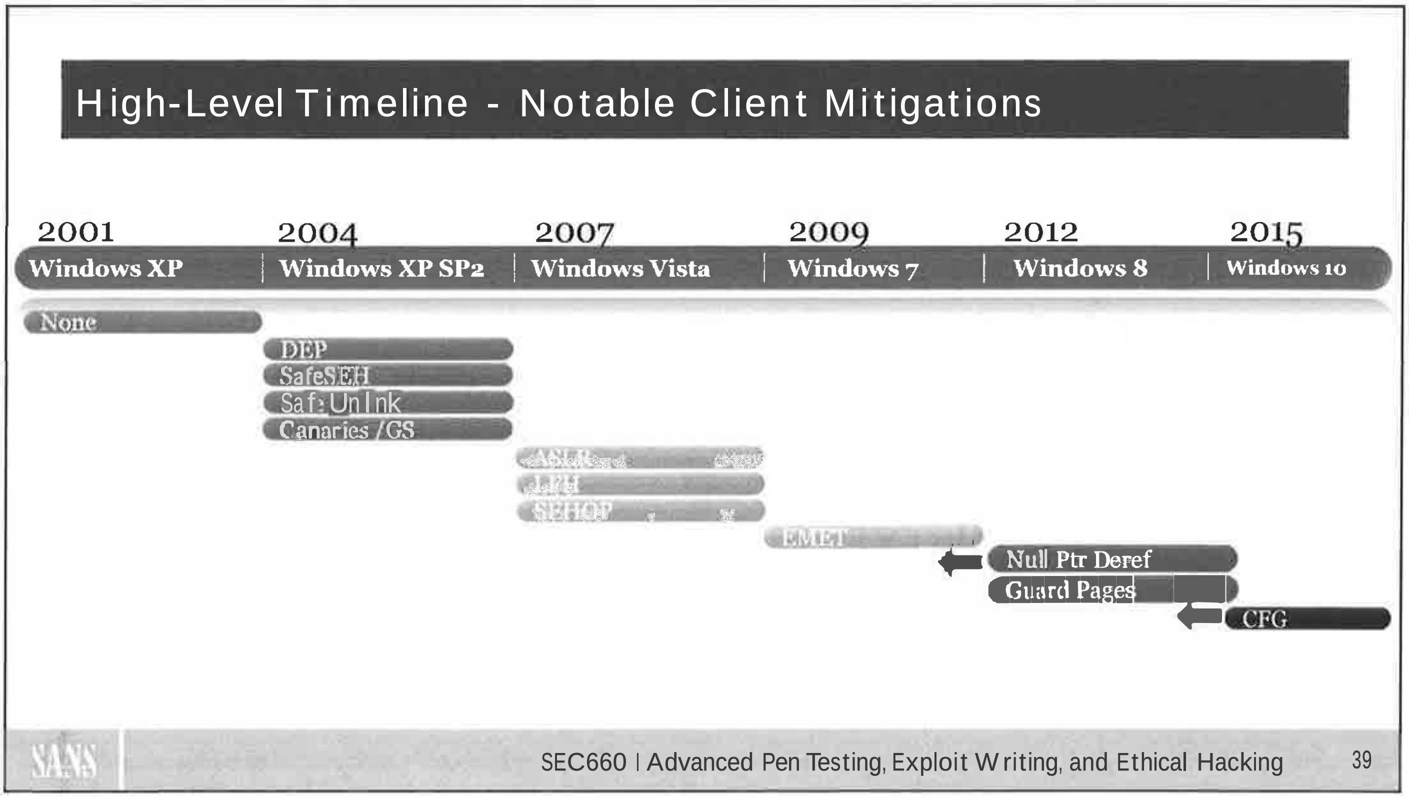


Exploit Mitigation Controls

The Venn diagram on this slide shows the three primary categories of exploit mitigation controls.

1) Application Opt-In Controls: This category of exploit mitigation allows the developer of an application or module to determine if it will participate in a given exploit mitigation control supported by the OS. These controls include ASLR participation, DEP participation, and others.
2) OS Controls: Regardless of a developer's choice to compile a program to participate in a control such as ASLR, the OS must have support f r the control. These controls include ASLR, hardware DEP, and several others.
3) Compile-Time Controls: This category of exploit mitigation describes controls that are added in during compile time. These controls include canaries or security cookies, application ASLR, Sa f SEH, and several others.

When the circles overlap, more controls are enforced, improving the security. When you get to the center, where all circles overlap, security is at the highest.



High-Level Timeline - Notable Client Mitigations

This slide shows a high-level timeline of exploit mitigations added or made available over the years. This is not exhaustive by any means, and we address each one in this section.

Linux Write XOR Execute (W^X)

- Marks areas in memory as writable or executable
  - Code segments are executable
  - Data segments are writable
  - Cannot be both
- Some ret2libc style attacks still successful
  - For example, passing arguments to a desired function
- No Execute (NX) bit – AMD
- eXecute Disable (XD) bit – Intel

SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 40

Linux Write XOR Execute (W"X)

It is uncommon, if not unheard of, f r a program to require code execution on the stack or heap. You certainly wouldn't want to accept executable code f om a user. A simple way to protect these memory segments f om holding executable content is to mark them as writable, but not executable. Code segments are typically executable and not writable. This being the case, segments in memory that are writable can be set as non-executable, and segments in memory that are executable can be set as non-writable. W"X, first implemented by OpenBSD, marks every page as either writable or executable, but never both. Many attacks are prevented by adding this protection. For example, if one places shellcode into a bu f er and attempts to return to it, the pages in memory holding that data are marked as non-executable, and as such, the attack f ils. There are still some ret2libc style attacks that may still be successful if W"X is used.

NX bit and XD/ED Bit

The NX bit used by AMO 64-bit processors and the XD (also known as ED) bit used by Intel processors provide protection through a form of W"X. NX and XD are built into the hardware, unlike the original W"X so f ware-based method. There are multiple methods that may be used to bypass or defeat this protection. If code you are looking to execute already resides within the applications code segment, you may simply return to the address holding the instructions you want to execute. If you have the ability to write to an area of memory in which you control the permissions, you may also retu r to that area holding your shellcode. On some implementations of W"X, it is possible to disable the feature. Each implementation and OS holds this capability in di f erent locations.

Data Execution Prevention

- Data Execution Prevention (DEP)
  - Started with Windows XP SP2 and 2003 Server
  - Marks pages as non-executable
    - For example: Stack, heap
    - Raises an exception if execution is attempted
  - Hardware based by setting the Execute Disable (XD) bit on Intel
    - AMD uses the No Execute (NX) bit
  - Can be manually disabled in system properties
  - So f ware DEP is supported even if hardware DEP is not supported
    - Software DEP prevents only SEH attacks with SafeSEH

Data Execution Prevention

Data Execution Prevention (DEP) is primarily a hardware-based security feature that is a take on the W^X control on Linux. The idea is that no code execution should ever take place on areas such as the stack and heap. Only pages explicitly marked f r code execution, such as the code segment, may do so. Any attempt to execute code in areas marked as non-executable will cause an exception, and the code will not be permitted to run. DEP is not supported in versions of Windows be f re XP SP2 and 2003 Server. You can also manually tu r DEP on or o f through system properties. If you go to Start, Run, type in sysdm.cpl, and press Enter, you pull up the System Properties menu. From there, you click the Advanced tab on the top of the panel and then the Settings option under Per f rmance. Then click the Data Execution Prevention tab on the top of the screen. You now have the option to turn DEP on f r essential Windows programs and services only, or you can turn it on for all programs and services, except f r the one you explicitly list.

As mentioned previously, Intel calls the bit that is set to mark all non-executable pages the Execute Disable (XO) bit. AMO calls this bit the No Execute (NX) bit. Both are hardware-based implementations of DEP in which the processor marks memory pages with a f ag as they are allocated by the processor. So f ware DEP provides only SafeSEH protection, which we discuss shortly.

SafeSEH

- SafeSEH
  - Builds a table of trusted exception handlers during compile time
  - Will not pass control to an address that is not in the table
  - Almost 100% of Windows DLLs and programs have been recompiled with this feature
  - To secure the program, all input f les must support the feature
  - Third-party programs and DLLs may cause a problem

SANS SEC6 6 I Ad v nced PenTesting, Exploit W riting, and E ical Hacking -2

SafeSEH

Starting with Windows XP SP2, the SafeSEH compiler option was added to provide protection against common attacks on SEH overwrites. When this f ag is used during compile time, the linker builds a table of good exception handlers that may be used. If the exception handler is ove r ritten and the address is not listed in the table as a valid handler, the program terminates and control is not passed to the unknown address. Most Windows DLLs and programs have been recompiled using the /SAFESEH flag, but it depends on the OS version.

The main problem with SafeSEH is that many third-party programs are not compiled with the /SAFESEH flag. O f en during program runtime, the Windows DLLs used by the program are protected by Sa f SEH, but the program itself has its own DLLs or code that is not protected. This gives an opportunity to the attacker to exploit the unprotected pieces loaded into the program's memory space. We will go into this attack later.

More in f rmation can be found by visiting Microso f at: https://msdn.microso f .com/en-us/library/9a89h429.aspx.

SEHOP

- Structured Exception Handling Over f ow Protection (SEHOP)
  - Started with Vista SP1 and Server 2008
  - Adds a validation frame at the bottom of the stack
  - Prior to an exception handler being called, the SEH chain is walked to verify that the validation frame is at the end
  - Defeating this control would require one to create a fake validation frame in the attack, which points to ntdll!FinalExceptionHandler

SEHOP

Matt Miller (Skape) outlined a control that could be implemented to stop the simple attack to overwrite the SE handlers on the stack of a given thread. This paper is titled, "Preventing the Exploitation of SEH Overwrites" and is available here: http://www.unin f rmed.org/?v=5&a=2&t=txt.

Microso f added support f r this control, starting with Vista SPI and Server 2008. The idea behind the control is to walk the list of handlers on the stack to confirm that the end of the list can be reached. The end of the list should contain a DWORD of 0x f ffffff, f llowed by a pointer to ntdll!FinalExceptionHandler. It is commonly managed using Microso f 's Enhanced Mitigation Experience Toolkit (EMET), which is "end of life" as of July 2018. Starting with the Fall 2017 Creators Update of Windows 10, Microso f moved the majority of EMET controls into Windows De f nder Exploit Guard.

A paper was released by Stefan Le Serre and Damien Cauquil on a technique to de f at the control. The technique involves creating a fake validation f ame as part of your exploit, tricking the control into thinking that the SEH chain is intact. The paper is available at http://dl.packetstormsecurity.net/papers/general/sehop_en.pdf.

Also, see: https://blogs.technet.microso f .com/srd/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/

Visual C++ /GS Check

- /GS security check supported on MS Visual C++ compiler
  - Pushes a security cookie onto the stack to protect return addresses
    - Cookie == Canary
    - Also protects exception handlers during unwind, unless handler is called prior to epilogue
  - Is enabled by default and is much more aggressive with newer versions of Visual Studio, such as Visual Studio 2017
  - Cookie is XORed against EBP or RSP to o fer additional security

Visual C++ /GS Check

The /GS option has been available on Microso f's Visual Studio C++ compiler since 2002. More recent versions provide extra security, such as on Visual Studio 2017, where there is protection f r vulnerable parameters on the stack by moving them below the security cookie. The /GS f ature pushes a 32- or 64-bit security cookie onto the stack if determined it is vulnerable. The cookie is XORed against EBP during the function prologue f r 32-bit applications and against RSP for 64-bit applications. This provides additional security. There are exceptions to the protection of f nctions, including whether the function includes a string buffer, bu f ers smaller than 5 bytes, and others; although this can be set to be more aggressive and depends on the version of Visual Studio. /GS-compiled executables and DLLs can be detected through signature analysis.

The /GS f ature is enabled by default, and of course, anything that was compiled without using MS Visual Studio would need to be recompiled with such to include the protection. Similar to Stack Smashing Protection (SSP) on Ubuntu and other variants, the security cookie is pushed onto the stack when a f nction is called. Upon function retu r , the cookie is checked against the master cookie to validate its integrity. If the check f ils, a handler takes over and terminates the process.

PEB Randomization

- PEB randomization
  - Introduced on Windows XP SP2
    - Pre-SP2 the PEB is always at 0x7FFDF000
  - The PEE had 16 possible locations with the initial implementation of PEE randomization:
    - 0x7FFD0000, 0x7FFD1000, ..., ..., 0x7FFDF000
    - Symantec research showed that a single guess has a 25% chance of success
  - On Windows 10, the PEE is randomized with greater entropy; however, it is still reachable by dereferencing FS:[0x30] while in user-mode context

SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking -s

PEB Randomization

Prior to Windows XP SP2, the Process Environment Block (PEB) is always f und at the address 0x7FFDF000. The PEB is a structure within each Windows process that holds process-speci f c information such as image and library load addressing. The static address made it possible for attacks such as overwriting RtlCriticalSection pointers to be called upon program exit. With the initial PEB randomization, the location of the PEB in memory would not always be loaded at the address 0x7FFDF000. It o fered 16 possible locations for it to be loaded, starting at 0x7FFD000 up to 0x7FFDF000, aligned on 4096-byte boundaries. Symantec's research showed that an attacker has a 25% chance of guessing the right PEB location on the f rst try. This is due to some inconsistency in the randomization that seems to favor certain load addresses. This research can be found at: http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf

PEB randomization is much greater on Windows 10.

Heap Cookies

- Heap cookies
  - 8 bits in length (256 possible values)
  - There is a 1/256 chance of guessing the right value on the first try
  - Introduced on XP SP2 and Windows 2003 Server
  - Placed directly after the Previous Chunk Size field

Heap Cookies

Heap cookies were introduced in Windows XP SP2 and Windows 2003 Server. They are 8 bits in length, providing up to 256 different keys that protect a block of memory. In theory, if you test an application that allows multiple attempts at corrupting the heap, you have a 1/256 chance on the first try. Heap cookies may be defeated through brute force or by memory leaks in vulnerabilities such as format string bugs. Heap cookies are placed directly after the Previous Chunk Size field in the header data. They are also validated only under certain instances. More will be discussed later.

Safe Unlinking

- Safe unlinking
  - Added to XP SP2 and 2003 Server
  - Similar to the update to early GLIBC unlink() usage on Linux; for example, dlmalloc
  - Much better protection than 8-bit cookies
  - Combined with cookies and PEB randomization, exploitation is di f cult
    - (B->Flink)->Blink == B && (B->Blink)->Flink == B

SANS SEC660 | Advanced Pen Te s ng, Exploit W riting, and Ethical Hacking 47

Sa f Unlinking

Safe unlinking was introduced in Windows XP SP2 and Windows 2003 Server. It is similar to how the modi f ed version of unlink() is used by the GNU C Library on Linux. Basically, the pointers are tested to make sure they are properly pointing to the chunk about to be f eed prior to unlinking. This is a much stronger protection than the 8-bit security cookies used for heap protection. Safe unlinking can be defeated in certain situations; however, the combination of cookies, safe unlinking, PEB randomization, ASLR, and other controls increase the di f culty of exploitation.

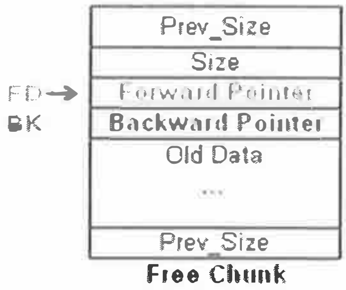
The f llowing is the code snippet used to safely unlink chunks of memory to be coalesced:

```
(8->Flink)->Blink == B && (B->Blink)->Flink == B
```

The code says that the next chunk's backward pointer should point to the current chunk and (&&) that the previous chunk's forward pointer should also point to the current chunk.

Linux Unlink() without Checks

```
#define unlink(P, BK, FD) { \
    FD = P->fd' \
    /* FD = the pointer stored at chunk +8 */
    BK = P->bk! \
    /* BK = the pointer stored at chunk +12 */
    FD->bk = BK; \
    /* At FD +12 write BK to set new bk pointer */
    BK->fd = FD; \
    /* At BK +8 write FD to set new fd pointer */
}
```



Linux UnlinkO without Checks - Recap f r Comparison to Windows

Here is the original source for the unlink() macro with added comments:

```
#define unlink(P, BK, FD) { \
            FD = P->fd; \
            /* FD = the pointer stored at chunk +8 */
            BK = P->bk; \
            /* BK = the pointer stored at chunk +12 */
            FD->bk = BK; \
            /* At FD +12 write BK to set new bk pointer */
            BK->fd = FD; \
            /* At BK +8 write FD to set new fd pointer */
}
```

Linux Unlink() with Checks

```
#define unlink(P, BK, FD) { \
    FD = P->fd; \
    BK = P->bk; \
    if (_builtin_expect (FD->bk != P || BK->fd != P, 0)) \
        malloc_printerr (check_action, "corrupted double-linked list", P); \
    else { \
        FD->bk = BK; \
        BK->fd = FD; \
    } \
}
```


Linux Unlink() with Checks - Recap for Comparison to Windows

Checks are now made to ensure the pointers have not been corrupted. Here is the code:

```
#define unlink(P, BK, FD) { \
            FD = P->fd; \
            BK = P->bk; \
            if (_builtin_expect (FD->bk != P || BK->fd != P, 0)) \
                        malloc_printerr (check_action, "corrupted double-
linked list", P); \
            else { \
                        FD->bk = BK; \
                        BK->fd = FD; \
            } \
}
```

Now we are simply adding a check to make sure that the FD's bk pointer is pointing to our current chunk, and that BK's fd pointer is also pointing to our current chunk. If it is != we print out the error, "corrupted double-linked list." The Windows safe unlinking technique works in the same manner.

Low Fragmentation Heap

- Low Fragmentation Heap (LFH)
  - 32-bit chunk encoding
  - Primarily began with Vista and replaced the Lookaside Lists as the frontend heap allocator in user mode
  - Can allocate blocks up to 16KB per Microsoft
    - >16 KB uses the standard heap
  - Allocates blocks in predetermined size ranges by putting blocks into buckets
    - 128 buckets total
    - Blocks are 8 bytes each
  - Is triggered after 18 consecutive allocations of the same size

Low Fragmentation Heap

The Low Fragmentation Heap (LFH) was introduced in Windows XP SP2 and Windows Server 2003, although it was not used unless explicitly configured and compiled to run with an application. It is a replacement to the frontend heap manager (lookaside list) in Windows Vista and later. LFH adds a great deal of security to the heaps it manages. When allocating blocks out of buckets, 32-bit chunk header encoding is used to perform a strong integrity check, acting as a security cookie. This is a more secure cookie than the 8-bit cookie protecting standard heaps on XP SP2 and Server 2003. LFH can allocate blocks greater than 8 bytes, but not larger than 16KB. Allocations >16KB use the standard heap, so the 32-bit cookie is not used.

Allocations are performed using predetermined chunk sizes arranged in 128 buckets. There are various groupings of buckets with each grouping sharing the same granularity. Detailed information about the block sizes stored in each bucket can be found at: http://msdn.microsoft.com/en-us/library/aa366750(VS.85).aspx.

Address Space Layout Randomization (ASLR)

- Began with Windows Vista
- Randomization for 32-bit applications is much less than on 64-bit applications
  - 32-bit DLLs are randomized 2^12, only ofering 4,096 unique load addresses
  - 64-bit applications and DLLs compiled with the /HIGHE N ROP Y A Visual Studio compiler fag take advantage of much greater entropy
- The stack and heap are always randomized, versus DLLs which must be compiled with the /D Y AMICBASE fag

SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 51

Address Space Layout Randomization (ASLR)

Microsof Vista was the first OS to support Address Space Layout Randomization (ASLR). This support then required that you compile the program with the /DYNAMICBASE fag, starting with MS Visual Studio 2005. Any program compiled on an earlier or different compiler requires recompilation. ASLR on a 32-bit application cannot support as much entropy as a 64-bit application compiled with the /HIGHENTROPYVA fag. There are differences between the different structures that are randomized, such as the stack and heap. ASLR has greatly improved since Vista, as well as kernel and driver ASLR.

A great presentation by Matt Miller and David Weston fom Microsof on Windows 10 security improvements can be found here: https://www.blackhat.com/docs/us-16/materials/us-16-Weston-Windows-10-Mitigation-Improvements.pdf

Defeating ASLR

- Defeating ASLR on Windows
  - Targeted attacks still possible
    - How much randomness exists?
    - How many times can an attacker try?
  - Some systems not running 7/8/10
    - Native ASLR does not exist in XP SP3 and prior
  - Format String attacks can leak memory
    - Location of stack and heap can be determined
  - Browser-based weaknesses allow for the creation of memory segments not participating in ASLR
    - Some modules, especially third-party, do not participate in ASLR

Defeating ASLR

ASLR on Vista and later provides a nice increase in security. Successfully defeating ASLR often requires the brute-forcing of a program. If there are a possible 256 locations for a desired variable in memory to be located and randomness is even, you have a 1 in 256 chance of successfully guessing the correct location in memory. If an application allows you to repeatedly hack at it, success is imminent. If there are a possible 65,536 locations for a variable to exist in memory, success is much less likely without crashing the process. Format String attacks may also allow for memory to be leaked, resulting in the discovery of addressing. When combining multiple defenses such as ASLR, DEP, PEB randomization, and others, attacks to take over control of a process become quite difficult. Much is still dependent on the application itself. If you have an application that creates many threads and allows for many connections such as IIS, successful exploitation can be more likely under certain conditions. If an attacker has selected a specific target and repeated attempts are permitted, exploitation is still possible against an ASLR-protected program. You must also remember that many systems are not running Vista or later and do not support native ASLR. Therefore, exploitation still continues with ease. There are also third-party modules that are loaded in by applications that do not participate in ASLR.

Enhanced Mitigation Experience Toolkit (EMET)

- Enhanced Mitigation Experience Toolkit (EMET)
  - Adds additional or stricter exploit mitigation controls to the Windows OS
  - Greatly increases the di f culty of exploiting a vulnerability, especially with versions 5.5+
  - Low adoption by most organizations
  - Flexibility to push EMET to only the applications desired
  - Introduces controls such as Mandatory ASLR
  - Heavy focus on stopping ROP-oriented attacks
  - End of life as of July 2018, but incorporated into Windows Defender Exploit Guard starting with Windows 10 Fall 2017 Creators Update

Enhanced Mitigation Experience Toolkit (EMET)

The idea behind EMET is to enforce stricter exploit mitigation controls to protect against attacks. An example of the type of control EMET can add is to f rce the use of ASLR to modules or programs that were not compiled to participate in the control. There are concerns around whether applications can handle the use of EMET. Microso f gives you multiple choices with EMET as to what programs should receive the additional protection. Many of the mitigations are aimed at preventing exploit techniques such as ROP.

The best link to understand the controls and features of EMET is at: https://support.microso f .com/en-us/help/2458544/the-enhanced-mitigation-experience-toolkit

Information on Windows Defender Exploit Guard can be f und here: https://docs.microso f .com/en-us/windows/threat-protection/windows-defender-exploit-guard/windows-defender-exploit-guard

Isolated Heaps and New IE Protections

- In June and July 2014, Microsoft pushed out patches that affected IE security
  - The June patch added Isolated Heaps for DOM objects to make the replacement of freed objects unlikely
  - The July patch added a protection called "Deferred Free" to help protect the freeing of objects, holding on to them before releasing them
- The primary goal is to mitigate Use-After-Free (UAF) exploitation

SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking 54

Isolated Heaps and New IE Protections

In June and July 2014, Microsoft added some new Internet Explorer protections as part of the Patch Tuesday updates, aimed at mitigating Use-After-Free (UAF) exploitation. In June, the patch added Isolated Heaps. This control protects it so object allocations are not made as part of the standard process heap. Instead, they are isolated, making the replacement of freed objects much more difficult. The July patch added a series of memory protections focused on the release of objects when freed. Instead of immediately freeing the objects when they are no longer needed, they are held on to and not released until a threshold is met. Even then, they are apparently not all let go at once.

MemGC

- Replacement for Deferred Free starting with Microsoft Edge on Windows 10
- Aimed at mitigating UAF exploitation
- Goes beyond looking for object references on the stack and in registers by also checking managed objects
- Greatly increases difficulty in exploiting UAF bugs

SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hacking

MemGC

The MemGC protection was added into Microsoft Edge starting with Windows 10 and it is an improvement over the "Deferred Free" mitigation added in 2014. MemProtect checked only the stack and registers for object references prior to freeing. MemGC goes beyond that and checks any MemGC managed objects for references to any objects marked to be freed. An object marked to be free that still has a reference will cause an exception that is handled, preventing exploitation.

An excellent white paper on Windows 10 browser exploit mitigations by Mark Vincent Yason can be found at: https://www.blackhat.com/docs/us-15/materials/us-15-Yason-Understanding-The-Attack-Surface-And-Attack-Resilience-Of-Project-Spartans-New-EdgeHTML-Rendering-Engine-wp.pdf

Control Flow Guard (CFG)

- New control targeting ROP-based exploitation
- Compiler control supported by Windows 10 and Windows 8, update 3
- Creates a bitmap representing the start addressing of all functions
- If an indirect call (call eax) is going to an address that is not the start of a valid function, the application terminates

Control Flow Guard (CFG)

A new control added to Windows I0 and back-ported to Windows 8, Update 3 is Control Flow Guard (CFG). It is a compile-time control that identi f es addresses deemed sa f as a destination f r an indirect call. The in f rmation is stored in a complex bitmap and checked prior to these indirect calls.

Core Security released an interesting article on bypassing CFO by using JIT compilation of ActionScript in Flash objects: https://blog.coresecurity.com/2015/03/25/exploiting-cve-2015-0311-part-ii-bypassing-control-f ow-guard-on-windows-8-l-update-3/

Control Flow Integrity (CFI)

- Intel released a paper in June 2016 describing new controls to be added:
  - Shadow stacks
  - Indirect branch tracking
- The idea of shadow stacks has been around for well over a decade, such as "Stack Shield" released in 2000: http:// w w.angel f re.com/sk/stackshield/
- Shadow stacks allow only the CALL instruction to push a copy of the return pointer to protected memory
- The return address f om the primary stack is checked against the address stored on the shadow stack
- Indirect branch tracking performs edge validation

Control Flow Integrity (CFI)

In June 2016, Intel released some press releases and a detailed PDF on Control Flow Enforcement (CET), their new upcoming control to prevent code reuse attacks and retu r -oriented programming (ROP) techniques. https://so f ware.intel.com/sites/default/ f les/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

The primary controls introduced in this paper are shadow stacks and indirect branch tracking. Each of these ideas have been proposed in various forms for well over 10 years. An example is "Stack Shield" released back in 2000. http://www.angel f re.com/sk/stackshield/

If properly integrated into the processor architecture, each control could have a moderate impact on code reuse techniques. The grsecurity team released a short posting as to their opinion on how Intel's implementation plan of these controls is lacking. https://forums.grsecurity.net/viewtopic.php?f=7&t=4490#P9

Shadow stacks work by marking certain pages of memory as protected, allowing only the CALL instruction the ability to write a copy of the return addresses used in the call chain. The return pointer on the actual stack is tested against the copy stored on the shadow stack. If there is a mismatch, an exception is thrown.

Indirect branch tracking takes advantage of the new instruction "ENDBR32" f r 32-bit or "ENDBR64" for 64-bit. This instruction is inserted a f er each valid CALL or JMP instruction. If this is not the next instruction, an exception is thrown. The instruction has the same e f ect as an NOP and simply is used f r validation.

Windows Kernel Hardening

- On Windows 8, 10, and Server 2012/2016
  - 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

SANS SEC660 I Advanced Pen Testing, Exploit Writing, and Ethical Hacking se

Windows Kernel Hardening

The Windows kernel has received a lot of hardening over the past f w OS revisions. This is mainly due to the fact that the kernel became more of a target when the exploit mitigations in userland (Ring 3) became more prevalent. Because there were fewer mitigations in the kernel, it became a good target. More and more f nctionality was also pulled out of Ring 3 and put into Ring 0. Improvements include mapping the first 64KB of memory so that the null pointer dereference vulnerability class was removed. Guard pages were added to ke r el memory. If a block of memory is marked as a guard and that memory is hit with a write attempt, an exception occurs. ASLR was greatly improved in the kernel, where it had previously been lacking. Security cookies were added to kernel routines.

Additional enhancements that started with Windows 8, unrelated to the kernel, are C++ virtual f nction table protection, return-oriented programming (ROP) protection, mandatory ASLR (ForceASLR), and more aggressive cookies.

Module Summary

- There are many controls available on Windows
- Combining these controls greatly increases security
- Many companies have not yet moved to Windows 10
- Controls are not a silver bullet

Module Summary

In this module, we looked at some of the most important security controls added to the Microsof Windows operating system over the past few years. It is likely that these controls will continue to improve, as they have proven to be a significant inhibitor to exploitation techniques, especially when combined.

Review Questions

1) How many bits of a chunk are encoded if managed by LFH?
   A. 32 bits
   B. 16 bits
   C. 8 bits
   D. 24 bits
2) Data Execution Prevention (DEP) works by marking pages in physical memory as executable or non-executable. True or False?
3) ASLR was available to use natively on XP SP2 but had to be enabled. True or False?

Review Questions

1) How many bytes of a chunk are encoded if managed by LFH?
   A. 32 bits
   B. 16 bits
   C. 8 bits
   D. 24 bits

2) Data Execution Prevention (DEP) works by marking pages in physical memory as executable or non-executable. True or False?

3) ASLR was available to use natively on XP SP2 but had to be enabled. True or False?

Answers

1. A: 32-bit chunk encoding is used
2. True: DEP sets a bit to mark pages of memory during allocation
3. False: ASLR was not available natively on XP SP2

Answers

1) A: 32-bit chunk encoding is used
2) True: DEP sets a bit to mark pages of memory during allocation
3) False: ASLR was not available natively on XP SP2

Course Roadmap

- Network Attacks f r Penetration Testers
- Crypto and Post Exploitation
- Python, Scapy, and Fuzzing
- Exploiting Linux f r Penetration Testers
- Exploiting Windows f r Penetration Testers
- Capture the Flag Challenge

Day 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

Exercise: Basic S t ck Ove r low

Exercise: SEH Overwrite

Defeating Hardware DEP with ROP

Demonstration: De f ating Hardware DEP Prior to Windows 7

Exercise: Using ROP to Disable DEP on Windows 7/8/10

Building a Metasploit Module

Windows Shellcode

Bootcamp

SANS SEC660 | Ad v nced Pen Te s ng. E loit W r ng, and E t ical Hackng 62

Windows Over f ows

In this module, we walk through various techniques to exploit the stack and exception handling on Windows.

Exercise: Basic Stack Overf ow

- Target: BlazeVideo HDTV Player 6.6 Pro
  - An HDTV player f r Windows
  - Version 6.6 is vulnerable to a stack overf ow
  - Vulnerability discovered in earlier versions as f r back as 2008 by ThE g0bL!N, f o f ow, and others
  - Use your provided Windows 10 VM
- Goals:
  - To trigger a bu f er overf ow inside the program
  - Determine the bu f er size
  - Veri f control of the instruction pointer
  - Locate a trampoline and get around ASLR

Estimated duration: 4 s minutes

Your instructor will walk you through this up to a certain point prior to handing over control.

SANS SEC660 | Advanced Pen Testing, Exploit Writing, and Ethical Hackng 63

Exercise: Basic Stack Overf ow

In this exercise, you per f rm a basic stack overf ow against a Windows application. The application we are targeting is BlazeVideo HDTV Player 6.6 Pro, available at http://www.blazevideo.com/hdtv-player/. It is vulnerable to a stack overf ow. The vulnerability was discovered in previous versions of the program, as far back as version 3.5 (f r example, https://www.exploit-db.com/exploits/32129/) by f0 f0w and ThE g0bL!N.

Your goal is to generate a malicious playlist f le to cause the program to crash with a buffer overf ow. When this is completed, determine the size of the bu fer before getting control of the return pointer, veri f control of EIP, and locate a trampoline to redirect execution to your shellcode. You will use your SANS-provided Windows 10 VM.

Your instructor will walk you through this one prior to handing over control f r you to complete. You will use your SANS-supplied Windows 10 VM.

Install BlazeVideo HDTV Player 6.6 Pro

- Take a snapshot prior to continuing!
- In **C:\lab\day5** folder is a folder called **Blaze Files**
  - In it is the installer titled BlazeD T roSetup.exe
  - Double-click the installer and accept any defaults
  - You may get an error message about a missing f le. Simply click Ignore and the installer will f nish
  - When you f nish installing the program, continue to the next slide

Install BlazeVideo HDTV Player 6.6 Pro

First, take a snapshot of your Windows 10 VM so that you can revert back if and when desired. The program we are installing has a 14-day trial period. Next, install BlazeVideo HDTV Player 6.6 Pro. On your Windows VM in your C:\lab\day5\ folder is a folder called Blaze Files. Double-click the file BlazeDTVProSetup.exe from inside this folder to install the program. Accept any defaults and continue to the next slide. You may get an error about a missing GIF f le during installation. Simply click ignore and continue with installation.

If you are unable to take snapshots, the trial period can be reset by deleting the f le SysInf_6_6_p.dll f om C:\Windows\SysWOW64\. This file is created during installation.

3 {••,€qö ÈK}••Krqs]^rK]ÿiq€$^•^Ki

"•0¥u•‰K•u|•†r••egp•KuNrKTgV]|VC•7••u†q•†•g•r••Koo•
0qq†rg•)•VN†cKVŸ•VZµq•eVpou%•g¥€•
\^ IT1CC5@3?B5^GTH@1QTZ>^@5HVO^eIGZCI5TMI^ ^<C45M^
Q@DGG6@Q1HQZ 51T>>5OM,Q5T¥5^
[^ 7#ZQH?^ ^@HOI@ ^HOQQ.2@CH5@QOB@2T5M^TCC4@QBI5^
@HQ@QIOC^Z5^O^
[^?5H@ HOQ.@3OIOQJ@3HQ%^5C^G1HJ.Z=^IGZ^ITM^ ^^
<C45UM5OMI^+#MI>M@GY5O+GGHT@QIZ3+GTH@QZ^
51T>>5M+#Z +GG.H4O^
"•=*708)*%#T•T•|•Wq•K•ru•q•K•R†eV•ŒKRrEog•Tu‰••
‹Vr•KrK•'gqdVqu|• •

#„Z&ô;N ã8§Š–(Vå Ë`„¦–!•´"2g¼a 8f#@@ø š‡67E ø f—õòø

P Æ %ž•éÛbô Ü±•ô±‡b

= -Û Û Û Û Û 0;RÛÛ Û Û Û < 1 Û Û¢ Û Û a 9Û 0€Û ÛÛ
µ ÛÛ Ûß 0 Û 0 Û

U6ÛÛ ÛÛ ;R Û Û ÛÛÛ ;, Û Ã Û Û Û ÛÛ =0 Û -Û crÛÛ ÛÛ
Â Û0 Û 5 i 7ÛÛ Ûz ±Û -Û þ†Û‡„ Û o X zÛß Û Û;u YÛ
, Û „ÛÛ Û Û i ÛcƒÛ TÛËÛ^ Ã±Û Û Û7 6Û Û Û7;RÛ ÛÛÛ Û
ƒ ec Ûß ÛÛ Û "Û

… Û 0 Û Û ÛÛ Û ;š0 Û c ;ºÛ Û \Û)c>dÛ aÛ ÛCš˜ƒ HÛ{XÛM•\ Û
• s• -Û ƒs C 1s Û 0Û Û cÛ 9~Û BÛÛ Û Û Û ;ÛR Û
Û Û Û < f Û!Û

= µ -Û Û Û;ÛR Û™¾Û• ™• Z µ•Û

™ b