Memory Corruption 101 Dino Dai Zovi ddz@theta44.org

Memory Corruption

- Memory corruption is when a programming error causes a program to change memory in an invalid way
 - Overwriting memory reserved for a different variable
 - Overwriting memory reserved for programming language runtime control structures
- When memory corruption may allow an attacker to take control of a program, it is a security vulnerability

Memory Corruption Classes

- Buffer overflows (Stack, Heap, Data segment, etc)
- Format string injection
- Out-of-bounds array accesses
- Integer overflows (can lead to buffer overflows or outof-bounds array access)
- Uninitialized memory use
- Dangling/stale pointers

Memory Corruption Exploits

- Usually the goal is to inject a machine code payload ("shellcode") and get the target program to run it
 - Usually we just want it to give us a remote or higherprivileged shell (/bin/sh or cmd.exe)
 - Not all exploits will use a payload that runs a shell
- Not all memory corruption exploits execute shellcode

Solaris TTYPROMPT Bug

% telnet
telnet> environ define TTYPROMPT abcdef
telnet> o localhost

SunOS 5.8

Vulnerability Analysis

- A program crashes, is it repeatable and reproducible?
- Memory is corrupted, is it controllable?
- Memory corruption can be controlled, is it exploitable?
- Some tools are available to help
 - !exploitable (WinDbg)
 - Crash Wrangler (Mac OS X)

Exploit Development

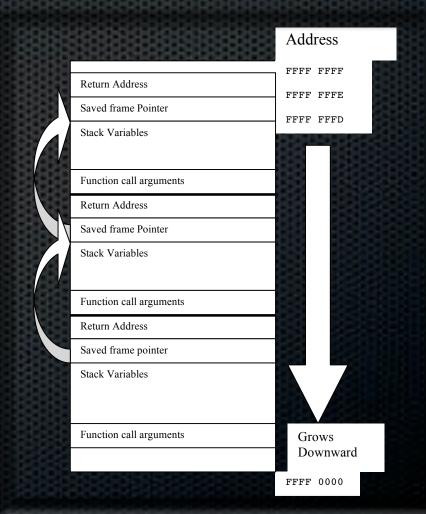
- Identify methods of controlling memory corruption
- Leverage controlled memory corruption to affect the program's behavior in a way that would give an attacker more privileges, capabilities, or access to the system
- Ideally, we would like to make it execute our payload
- Everyone loves a remote root/SYSTEM shell

Stack Buffer Overflows

- The canonical, simplest type of memory corruption to understand and exploit
- First publicly used by Robert Morris worm in 1988
 - Used a stack buffer overflow in VAX BSD in.fingerd
- Are *still* exploitable on many systems today
 - Many operating systems and compilers include defenses against these now (more on this later)

The Stack

- Stack grows downward
- Memory writes go upward
- Stack variables can overflow into saved frame pointer and return address



Smashing the Stack and controlling EIP

D'C/Documents and Settings\ddr\M	y Documents'/Visual Studie Projects'/En	where the leave \1	trankMease" stack	Windigstate 28
File Edit View Debug Window Help				
S ABR NEMM	0001000000		1 101 AA 🛠	
Disassembly		E Registers		(21) (2)
Offset:	Presca Nest	Customize		Raw args Func info
	210 220 177	Reg V edi 0	alue .	Source Addrs Headings Nonvolatile regs
41414143 ??	177	esi a	28	
	122		00016111	Frame nums Source args
41414146 77 7	177	edx 1 ecx 4	09068	More Less
	122	eax 0)	
	22		211c0	WARNING: Frame I
	177		b	0x41414141
41414140 77	177	ef1 1	10206	0x41414141 0x41414141
		1 4	Hear +	0x41414141
Command			01	0x41414141 0x41414141
Executable search path 15				0xa987bc00
ModLoad 00400000 00405000	CrashMe.exe		-	0x12ffc8 0x8053d14f
ModLoad: 77f50000 77ff7000 ModLoad: 77e60000 77ff6000	ntdll.dll C:\WINDOWS\system32\kernel	22.411	10	Oxffffffff
(fbc.620): Access violation	a - code c0000005 (first char	ce)		kernel32!SetThre kernel32!OpenCon
First chance exceptions are	reported before any excepti	on handling		
This exception may be expected as *00000000 ebx*7ffdf000 e	cx=00409068 edx=00000001 esi	-00000a28 m	di=000000000	
eip+41414141 esp+0012feec e	sbp=0012ffc0 topl=0	w up ei pl i	nz na po nc	
	es=0023 fs=0038 gs=0000	e	£1=00010206	
	d not be found. Defaulted t	o export sys	aboin for C	
			-	
			20	
0:000>				1 2
i de la companya de la compa	lo 19 Cel 1	Set D clocala D	or 000 fbc Thyd 000	620 AGH JOWN CAPS IN
	len sy ou s	have be required by	and statement in the state	Owner Departs Lingued Landon Rand

Stack Buffer Overflow

- Stack variable overflows, overwriting the return address
- The attacker writes a memory address in the stack for the return address
- The subroutine returns into payload on stack



Let's see a real (fake) one...

Heap Metadata Corruption

Pid 2344 - WinDbgr6.6.0003.5

_ 8 ×

File Edit View Debug Window Help

Y 181 A. 🛠 is assembly 611 23 티비 **Offset**: Previous. Next Raw args Func info Source Addrs Customize... 7715819e e8d6ad0200 call ntd11+Rt1pUpdateIndes Headings Nonvolable regs Rect Value . 77f581a3 8d4608 1ea eax.[esi+0x8] 0 77f581a6 898538ffffff [ebp=0xc8].eax 95 8.077 Source args Frame nume 38 77f581ac 8b08 ecx. [eax] ts. 3.077 [ebp-0xcc].ecx 23 77158lae 898d34111111 INCO/ 00 More Less 58154 854004 BOT max.[max+0x4] ds 23 \$81b7 898530ffffff ROW [ebo-0xd0].eax edi [enx].ecx 3.077 de: 3207d8 ntd111Rt1A11ocateHeap+0 651 581bf 894104 ecx+0x41.eax NOVE 320000 eba CrashMe!_heap_alloc+0x4 77f581c2 8x4605 5.077 al.[esi+0x5] CrashMe!_nh_malloc+0x10 edx 3207d8 77(581c5 8845a3 [ebo-0x5d].al ROVE CrashMe!malloc+0xf ecik 41322f90 77f581c8 0fb706 XZVOR eax word ptr [esi] CrashMe!corrupt_heap+0x (55X) 41414141 771581cb 8b7de4 edi.[ebo-0x1c] BOVF. CrashMe!main+0x51 12ef 00 ebp 774581cm 294728 Ren [B(whithm] weibs CrashMe!sainCRTStartup+ 41 kernel32/BaseProcessSta 1 A Command Executable search path is . ModLoad: 00400000 00405000 C:\Documents and Settings\ddz\My Documents KodLoad : C:\WINDOWS\System32\atdl1 d11 77150000 77117000 ModLoad. 77e60000 77146000 C:\WINDOWS\systes32\kernel32.dll 77440000 KodLoad : 77eSd000 C:\WINDOWS\system32\ADVAPI32_DIL ModLoad 78000000 78087000 C:\WINDOWS\system32\RPCRT4.dll (928.f00): Access violation - code c0000005 (111 second chance 111) ecx+41322f90 edx+003207d8 esi=003207d8 edi=000 141 ebx+00320000 eip=77f581bd esp=0012ecdc ebp=0012ef00 icpl=0 ny up ei pl zr na es=0023 fs=0038 gs=0000 cs+001b ss+0023 ds+0023 ef1=000 ntd11/Rt1411ocateMeap+0x60f 77f581bd 8908 [eax].ecx du:0023:41414141=?? BON? 4 0:000> 4 Ln D, Col D Sys D: <Local> Proc 000:928 Thyd 000:400 ASH CHD CLES 11 M

What is going on here?

🗊 Pid 996 - WinDbg:6.6.0003.5					_ (#) ×
File Edit Vew Debug Window Help C2 1 Bb IPI bit bit without	000000000000		101 A. 4	*	
Wrtusk cox	Previous Display format: Dyte		Genisters Customize		51
Unable to retrieve informa	tion. Vin32 error 30: The s	system cannot r	gs fs es ds edi esi ebx edx ecx	Value 0 38 23 23 0 320e80 7ffdf000 320e84 407210	*
*I		بر م		12ffc0	1 1
eax=00000000 ebx=7ffdf000 eip=0040118d esp=0012fedc cs=001b ss=0023 ds=0023 ••• WARNING: Unable to ver CrashMe!main=0xcd: 0040118d a100000000	C:\VINDOVS\System32\ntd C:\VINDOVS\system32\ADV C:\VINDOVS\system32\ADV C:\VINDOVS\system32\ADV C:\VINDOVS\system32\RPC ecx=00407210 edx=00320e84 (ebp=0012ffc0 iop1=0 es=0023 fs=0038 gs=0000 ify checksum for C:\Document	11.d11 he132.d11 API32.DIL RT4.d11 ond chance !!!) esi=00320e80 edi=0 nv up ei pl zr n ef1=0	0000000 a po nc 0000246 idz\My Dec		al Stu
0.000>					1
10.00031	Ln 65, Co	1 Sys 0: <1.ocal> Proc 000	1:3e4 Thyd 00	00:876 AGH (OVD.	CAPS (N.H.

And here?

🗊 Pid 1056 - WinDbg:6.6.0003.5					_ 8 ×
File Edit Were Debug Window Help			1		
· · · · · · · · · · · · · · · · · · ·	MU Q D D D D D D D D D D D D D D D D D D		*		
Memory Wrtust cost	Display format: Byte	Previous	Nex	Registers	
the second se	and the second s	and approximately a second of	and a desired and and	Customize	
Unable to retrieve information, W	inj2 error j0. The system o	cannot read from	the s	Reg	Valu *
				gs ts	20
				es	23
				ds	23
				eds ess	0 3814
				ebx	7ttc.
				edx	0
				ecx eax	4050
			-	ebp	1211-
		2	1		14
Command					24
ModLoad: 7c900000 7c9b0000 C:\W ModLoad: 7c800000 7c8f4000 C:\W ModLoad: 77d0000 77e6b000 C:\W ModLoad: 77e70000 77f01000 C:\W (420.aac): Access violation - cod eax=00000000 ebx=7ffdf000 ecx=004 eip=00402370 esp=0012fed9 ebp=001	050fc edx=00000000 emi=0038 2ffc0 iopl=0 nv up 3 fs=003b gs=0000 cksum for C:\Documents and	11 11 314a8 edi=0000000 ei pl zr na po n ef1=0000024	0 c 6 Documents		
1					2
0:000>					
	Ln 0, Col 0 Sys D: <	Local> Proc 000:420 Th	10 000 bac	SPI COVIL	CAPS MUM

Exploit By Numbers

- 1. Trigger the vulnerability
- 2. Identify usable characters for attack string
- 3. Identify offsets and significant elements in attack string
- 4. Fill in jump addresses, readable/writable addresses, etc
- 5. Identify amount of usable space for the payload
- 6. Drop in payload

Trigger the Vulnerability

- Write a network client to talk to the server
- Create a malformed file that gets opened by the app
 - Document (.doc, .ppt, .pdf)
 - Media file (.mp3, .mov, .wmv)
- Create a malicious web page that is viewed by browser
- Cause the target application to crash

Identify Usable Characters

- The attack string is the part of the input that triggers the vulnerability and contains values for overwritten memory (and possibly the payload also)
- Certain characters in the attack string may cause the application to parse the input differently and not trigger the vulnerability ("bad bytes")
 - NULL bytes (any ASCII string)
 - Whitespace (\t\n\r)

Identify Offsets

- Use a pattern string to identify offsets into your attack string of data placed into registers or written to memory
- We are going to use Metasploit's pattern_create.rb
 - % pattern_create.rb 32
 Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab
 - % pattern_offset.rb 0x41366141
 18

Fill in Memory Addresses

- For an exploit to function, certain parts of the attack string may need to readable, writable, or executable memory addresses
 - In particular, we want to overwrite the return address with the memory address of executable code
 - This memory address will redirect execution into our attack string
 - Spend quality time in your target's address space

Identify Usable Space

- We need to know how much room we have for our payload
- We will size it out by placing increasingly large numbers of NOPs followed by a debug interrupt (int 3)
- If the target generates a breakpoint exception, we have that much usable space
- If the target crashes in another way, we may need to shrink the payload space

Drop in Payload

- The payload must also not use any bad bytes or else it may get truncated and not execute properly
- For simple payloads and vulnerabilities, avoiding NULL bytes in the instruction encodings may be enough
- For more complex payloads and vulnerabilities, a payload decoder may be used to decode the payload before executing

Exploiting Windows 2000

- There are many aspects of Windows 2000 and the x86 processor that make exploitation of memory corruption vulnerabilities possible and even *easy*
 - Libraries are always loaded at same place in memory
 - Executable page protection permissions are ignored
 - There are no alignment requirements
 - There are no issues with cache coherency

Live Demo Time...

Exploitation Mitigation

Exploitation Mitigation

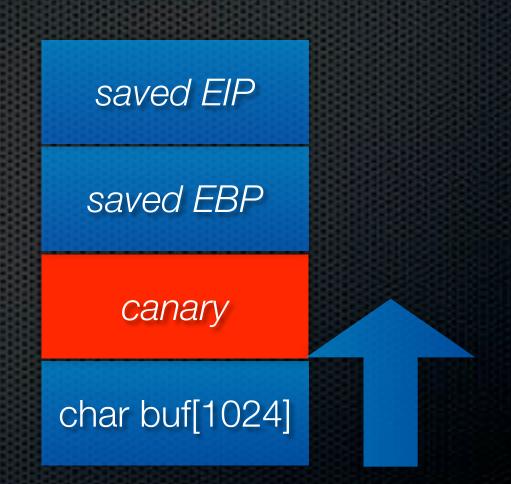
- Finding and fixing every vulnerability is impossible
- It is possible to make exploitation more difficult through:
 - Memory page protection
 - Run-time validation
 - Obfuscation and Randomization
- Making every vulnerability non-exploitable is impossible

Timeline of Mitigations

- Windows 1.0 Windows XP SP1
 - Corruption of stack and heap metadata is possible
- Windows 2003
 - Operating System is compiled with stack cookies
- Windows XP SP 2
 - Stack/heap cookies, SafeSEH, Software/Hardware DEP
- Windows Vista
 - Address Space Layout Randomization

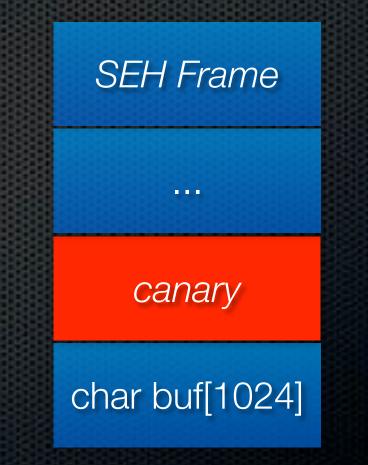
Visual Studio /GS Flag

- Place a random
 "cookie" in stack
 frame before frame
 pointer and return
 address
- Check cookie before using saved frame pointer and return address



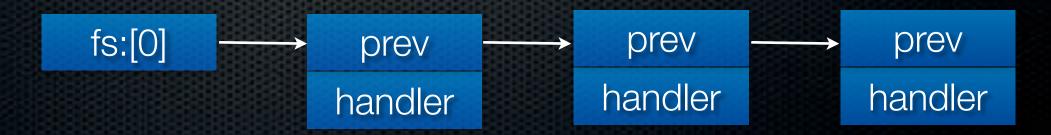
Structured Exception Handling

- Supports __try/ __except blocks in C and C++ exceptions
- Nested SEH frames are stored on stack
- Contain pointer to next frame and exception filter *function pointer*



SEH Frame Overwrite Attack

- Overwrite an exception handler function pointer in SEH frame and cause an exception before any of the overwritten stack cookies are detected
 - i.e. run data off the top of the stack
- David Litchfield, "Defeating the Stack Based Buffer Overflow Protection Mechanism of Microsoft Windows 2003 Server"



Visual Studio /SafeSEH

- Pre-registers all exception handlers in the DLL or EXE
- When an exception occurs, Windows will examine the pre-registered table and only call the handler if it exists in the table
- What if one DLL wasn't compiled w/ SafeSEH?
 - Windows will allow any address in that module as an SEH handler
 - This allows an attacker to still gain full control

RTL Heap Safe Unlinking

- Corrupting the next/prev linked list pointers of a heap block on the free list allows an attacker to write a chosen value to a chosen location when that block is removed from the free list
 - i.e. Overwrite the global UnhandledExceptionFilter
- Safe Unlinking adds a 16-bit cookie to heap header, which is checked before the block is removed

Data Execution Prevention

Software DEP

 Makes sure that SEH exception handlers point to non-writable memory (weak)

Hardware DEP

- Enforces that processor does not execute instructions from data memory pages (stack, heap)
- Make page permission bits meaningful (R !=> X)

Bypassing DEP

- Return-to-libc / code reuse
 - Return into the beginning of a library function
 - Function arguments come from attacker-controlled stack
 - Can be chained to call multiple functions in a row
- On XP SP2 and Windows 2003, attacker could return to a particular place in NTDLL and disable DEP for the entire process

Return-Oriented Programming

- Return into useful instruction sequences followed by return instructions
- Chain useful sequences together to form useful operations ("gadgets")
 - "store X at memory address Y"
 - "add X to value stored at memory address Y"
- Academics have built "compilers" for return-oriented "programs" in C-like languages

Address Space Layout Randomization

- Almost all exploits require hard-coding memory addresses
- If those addresses are impossible to predict, those exploits would not be possible
- ASLR moves around code (executable and libraries), data (stacks, heaps, and other memory regions)
- Windows Vista randomizes DLLs at boot-time, everything else at run-time

Bypassing ASLR

Poor entropy

- Sometimes the randomization isn't random enough or the attacker may try as many times as needed
- Memory address disclosure
 - Some vulnerabilities or other tricks can be used to reveal memory addresses in the target process
 - One address may be enough to build your exploit

Exploit Payloads

Local Unix Shellcode

- The oldest buffer overflow exploits were local privilege escalation exploits against setuid executables
 - Just a small bit of machine code to run a shell
 - execve("/bin/sh", NULL, NULL)
 - Shell runs with higher privilege
- Easy to write for any OS/Architecture if you know the architecture's assembly language

execve("/bin/sh", NULL, NULL)