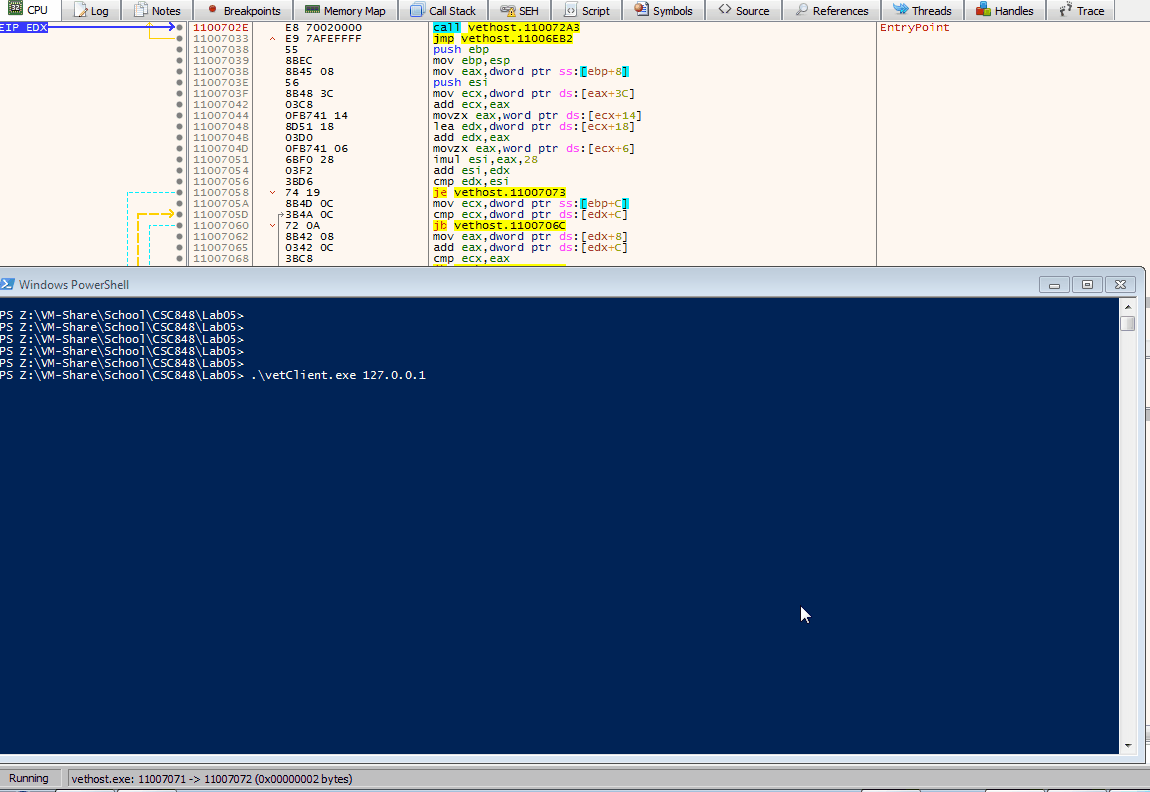
Lab 05 – ROP with Vet APP

**Part 1 - Finding the Exploit**



I began looking for the vulnerability by running the vetHost.exe under x32dbg and then the vetClient.exe from PowerShell. My initial hunch was that the exploit would be accessible from the host, otherwise, why would we be given a task with two stages; I could be wrong, but it looks like I was not.

After experimenting with the options from the client, I noticed an EXCEPTION\_ACCESS\_VIOLATION was generated bc of a specific chain of inputs via the “Q: Quick Add/Delete…”. Assuming the following series of inputs:

1. Input ‘Q’
2. Animals… X; Clinics… Y
3. Animals delete… -1; Clinics delete… -1
   1. Skips deletion at this stage…
4. Input ‘Q’
5. Animals… 0; Clinics… 0
6. Animals delete… 0, -1; Clinics delete… 0,-1
7. EXCEPTION\_ACCESS\_VIOLATION
8. ….

For reference watch the looped .GIF above.



The reason being both EAX and ECX store the value ‘FEEEFEEE’ - which is used by HeapFree() to indicate that heap memory has been freed. So, in this instance the program is trying to reach something that has probably been freed or delinked by our deletion of the index [0] for Animals/Clinics.

This will probably be useful later as buffer-overflow… notably a payload/buffer offset of 106 for Animal[0][‘Name’] will place the following 4 bytes into EAX and ECX.

**Note: The above results were a good start but irrelevant towards actual exploitation :(**

But this is not the memory disclosure that we need…

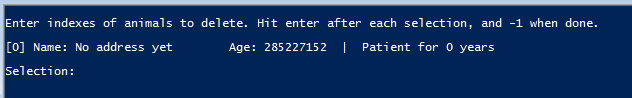


**Note: I’d played with the binary attempting to find a memory disclosure, but the disclosure would not print until I’d used the above command you mentioned before. Is this only supposed to work if hpa and ust have been enabled… ?**

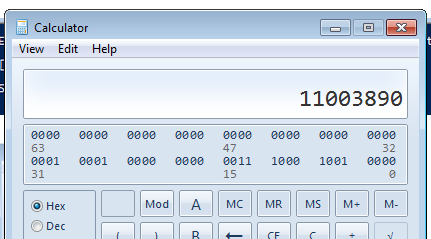
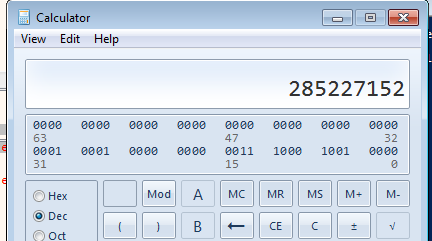
Otherwise… After playing around some more with the binary, I found that the following steps would produce a memory disclosure:

1. ‘A’ → Animals
2. ‘D’ → Delete Animals index[0]
3. ‘Q’ → Quick add [1,1]
4. Memory disclosure produced
5. Quick delete nothing [-1, -1]

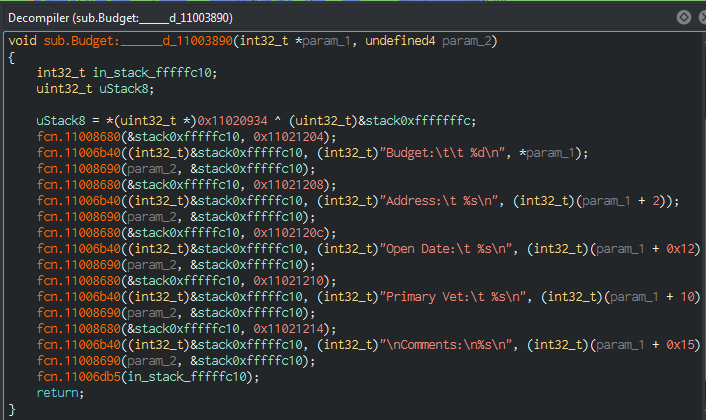
The output of which should appear as…



We can then take that value and convert it from the decimal format to the hex address

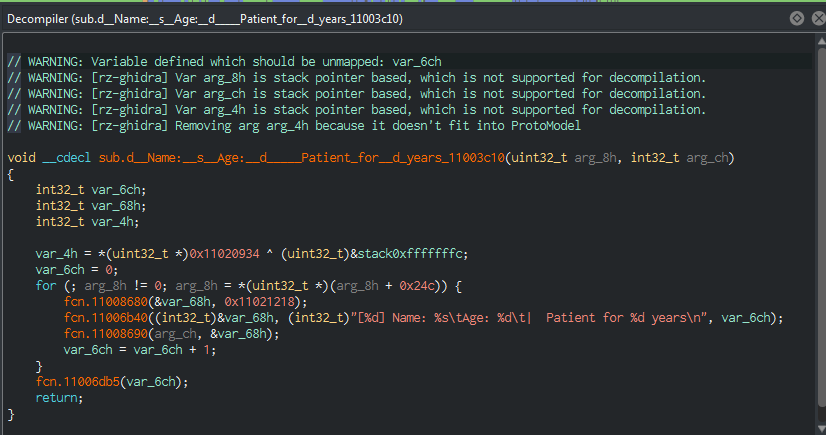


Not gonna lie - this part confused me for several hours wondering why my disclosure was absolute garbage output, and then it only turned out to be decimal format and not hex.



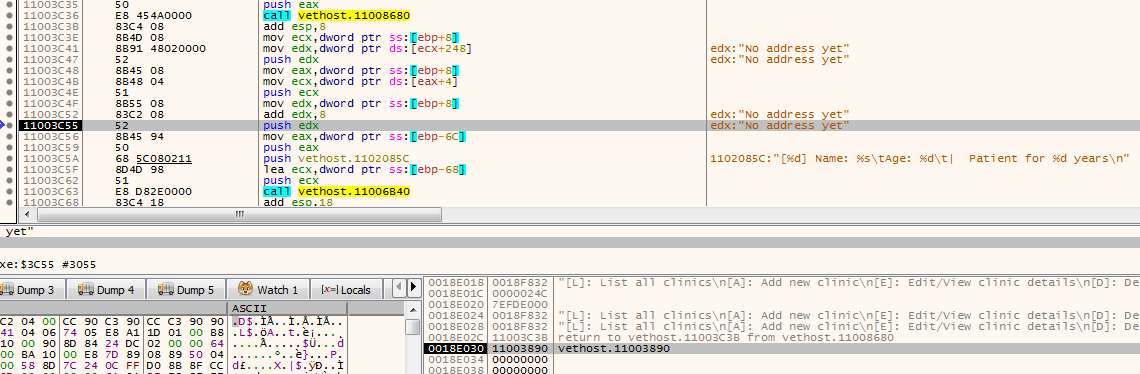
The disclosure is the address for a call to print clinic information.

The offset is then 11000000 (Module Base) - 11003890 (Virtual Address) = 0x3890 or 14480



Above is the vulnerable function - this will attempt to access a record we previously deleted before visiting the Quick Add/Delete menu; assuming the record is deleted, the field Animal[“Age”] will contain the decimal representation of a print function for Clinic records.

Here we can see the function actually on the stack… after the call to vethost.11008680 (and several stack manipulations)…

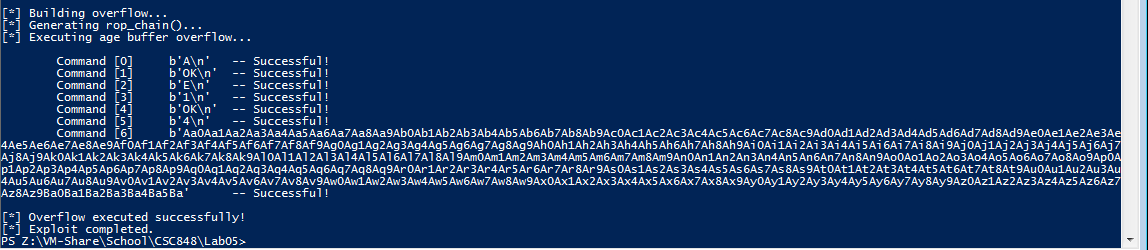


So - after playing with the editing functions of Animal we know that the Age value has a buffer overflow. We can figure out the exact offset to control the stack by feeding a buffer overflow register offset string and then matching the byte pattern to find the exact offset.

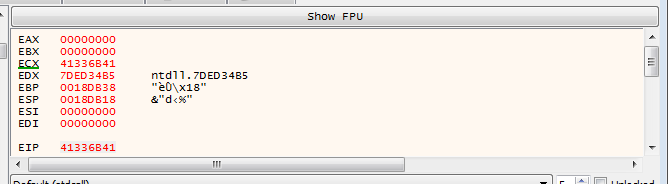
Here is a partial look at the string used…. The total length is 400 bytes.



Our output in the terminal shows that the exploit ran successfully…



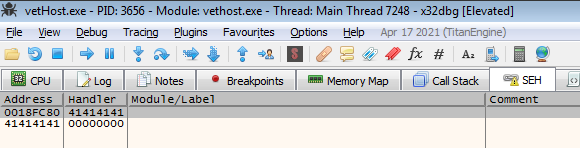
If we look at the debugger then, we can see an exception has been generated because the register EIP was overrun with the provided string. Specifically…



By taking the contents of EIP, 0x41336B41 we can then match the pattern to the supplied payload and find the exact offset needed to control EIP.

In this instance, the offset needed is 309 bytes from Animal[1][Age].

From here we need only find our correct offsets for SEH which we can easily do within x32dbg

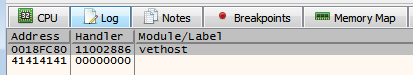


SEH is overwritten @ 0x18FC80 and EAX @ 0x18FC98

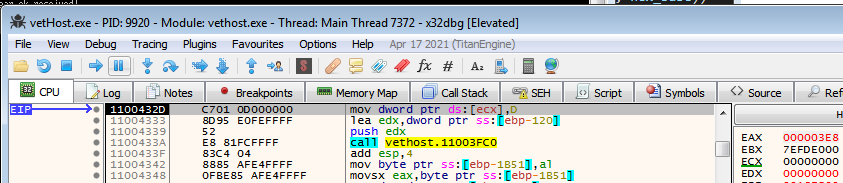
This gets replaced by our stackpivot at the address…

**0x11002886 : # ADD ESP,2188 # RETN \*\* [vetHost.exe] \*\* | {PAGE\_EXECUTE\_READ}**

This navigates back from the generated exception handler to our convenient mona rop\_chain.



And the SEH for vetHost.exe has been replaced with our above address.



Stepping within the debugger from this access violation, where the dereferenced PTR ECX is now 0, causes an access violation and triggers our overwritten SEH value.

From here the rop\_chain using VirtualProtect() is executed and our shellcode is ran.

If something about the process still hasn’t been explained well enough, the provided python script is fairly well commented and should fill any knowledge gaps.

**Part 3 – Bypassing DEP with ROP**

Initially, I tried seeing what ROP chains mona could generate from…

* vetHost.exe, rpcrt4.dll, and msvcrt.dll

Obviously there were some issues because not all of the generated chains were usable. Both vetHost.exe and RPCRT4 generated malformed chains containing gadgets with exploit crashing side effects, or they were straight up missing the necessary gadgets. However, MSVCRT did generate a usable chain and that’s ultimately what I ended up using ¯\\_(ツ)\_/¯

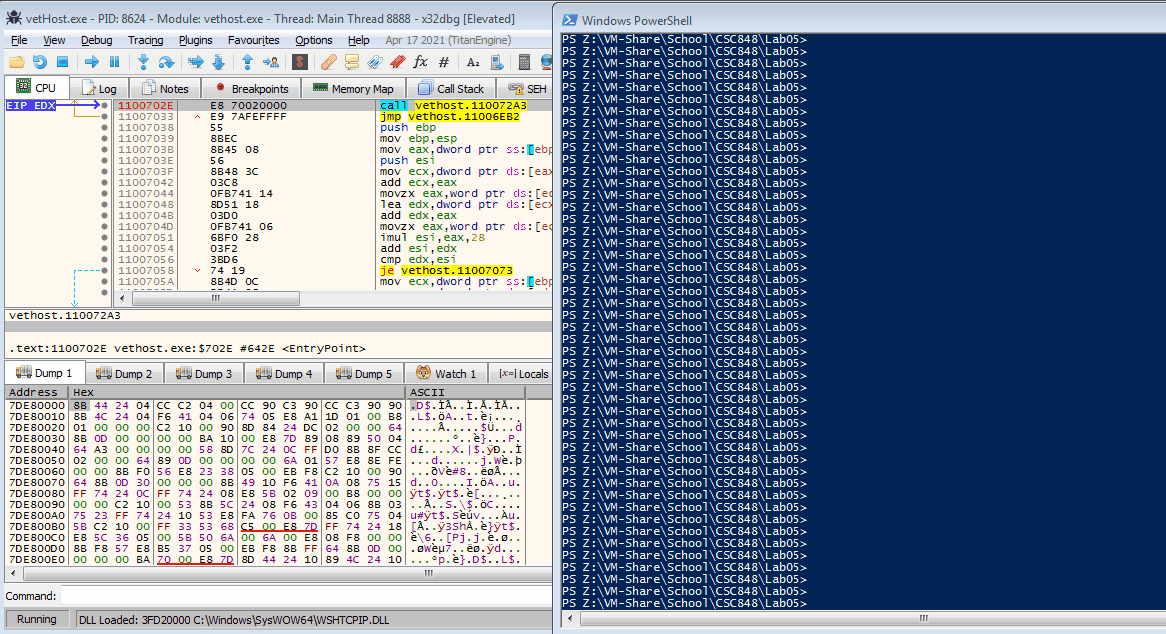
See here:



**Part 2 - Writing Shellcode**

**The shellcode is the same as before. So the overall process goes like:**

1. **Download second stage from Github as “not\_malware.test”**
   1. **It is saved to the same location the executable is ran**
2. **Execute the payload**
   1. **Sends email/sms notif to me that the payload was ran using pyinstaller executable generated from custom python script**
3. **Sleep/Wait for 5000ms**
4. **Delete the payload named “not\_malware.test”**
5. **ExitProcess() gracefully**

****

**You should be able to follow that process in the following GIF.**

**Files Included:**

* **micah (bin)**
* **micah.asm**
* **micah.py**