Lab 03 – ROP with WAVreader

**Note: I should have noticed earlier… but I’ve been using my own Win7 environment because the Ialab is too laggy for me. Any scripts I submit from now on will be tailored to limit any obstacles created by the “different” environments.**

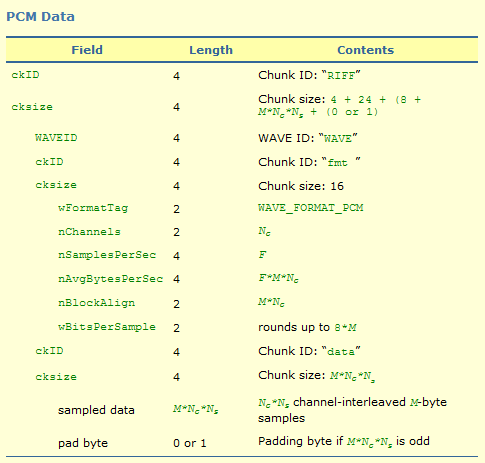
Starting the program and just doing the basics of seeing how everything flows from execution to termination… and with a valid .WAV file it doesn’t work? I’ll dig into this later with a disassembler.



Presumably, whatever implementation this is doesn’t follow the file standards for .WAV. For reference, this is what the file structure should look like:

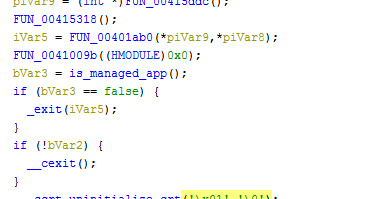
<http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html>

<https://github.com/cr/a1541/blob/2f8c5f1b6b90403d811387c42da6f3ad76057d27/doc/formats/WAV.TXT>

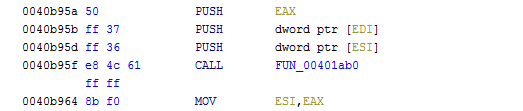


I have a very strong hunch that the above formatting info will be important later on…

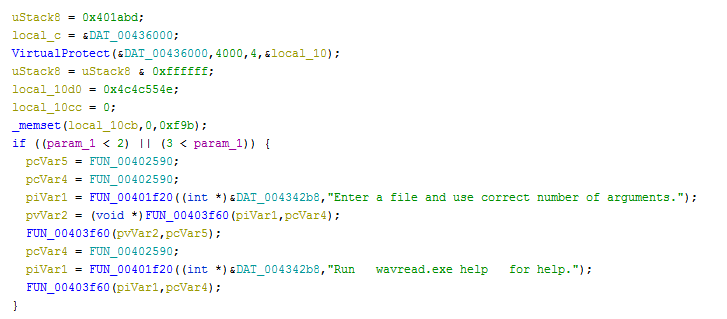
Jumping through the program… We go from entry() to the exit calls…



See the exit() call above… Then just look for the instructions for MOV XXX, EAX immediately succeeding a call to somewhere else unlabeled in the program.



Here we have main() - we know this is it because we see xrefs to strings shown during runtime



At this point I really just want to see why our properly formatted .WAV file fails to read and returns the output ‘Invalid Format.’ - I know I have interpreted the instructions for this executable correctly, so this is the first step towards understanding core functionality.

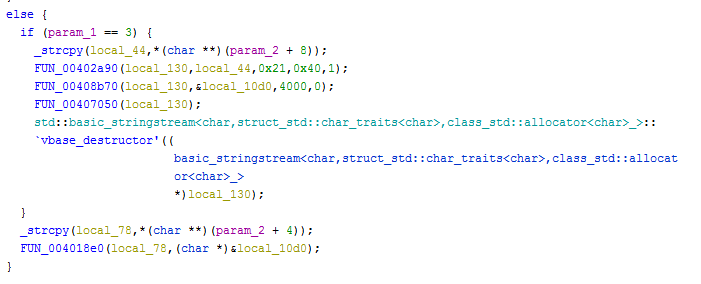
Without getting to nitty gritty into the details - some key logic/calls we see in main() are:



 == ‘NULL’



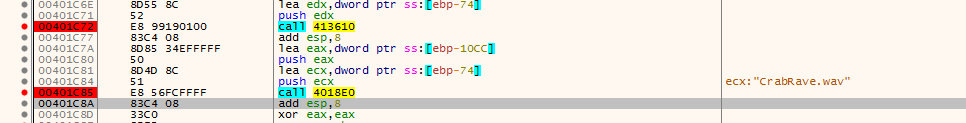
And then the code block that is most interesting is here… If you walk through the majority of main() it is just making sure that the correct number of args is provided, or that you’re not asking for ‘help’.



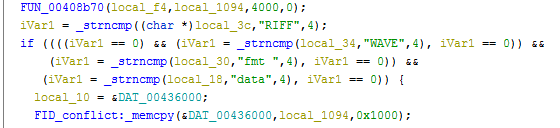
The first call to \_strcpy is retrieving the contents of the provided ‘command.txt’ - at this moment, the contents are just ‘test’.

However, I am most interested in the FUN\_004018e0 because this is where the text for our error is generated. We should also note that it reads the contents of the provided wave file before making the call through \_strcpy(param\_2 + 4).

Using x32dbg we set the breakpoints and then run until breaked…



Referring back to Ghidra… if we look at the FUN\_004018e0 we should see some references to the file format structure for .WAV.



So pretty obvious it is checking the first 3-4 fields represented in the WAV file format.

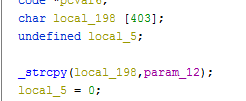
* ChunkID: RIFF
* [Skipped] ChunkSize
* WaveID: WAVE
* ChunkID: fmt
* [Skipped] ChunkSize
* ChunkID: data

This buffer is all stored @ DAT\_00436000.

Continuing further into the function it will enter this function…Whether the parameter controlling the command has been specified via file or by prompt. We are going the file route because I am assuming it will be easier to exploit with more file I/O.



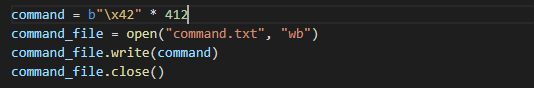
The interesting part here is the call to FUN\_00401740 - this references the contents of the provided command file…



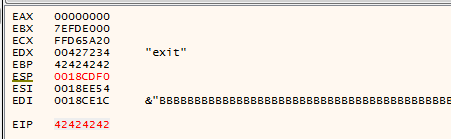
Immediately we see this call to \_strcpy from EBP + 52 into a local buffer of 0x198 or 403 bytes.

My assumption at the moment is that this might provide some control over the stack via the provided command file.

This can be tested by generating the necessary payload and then watching the stack…



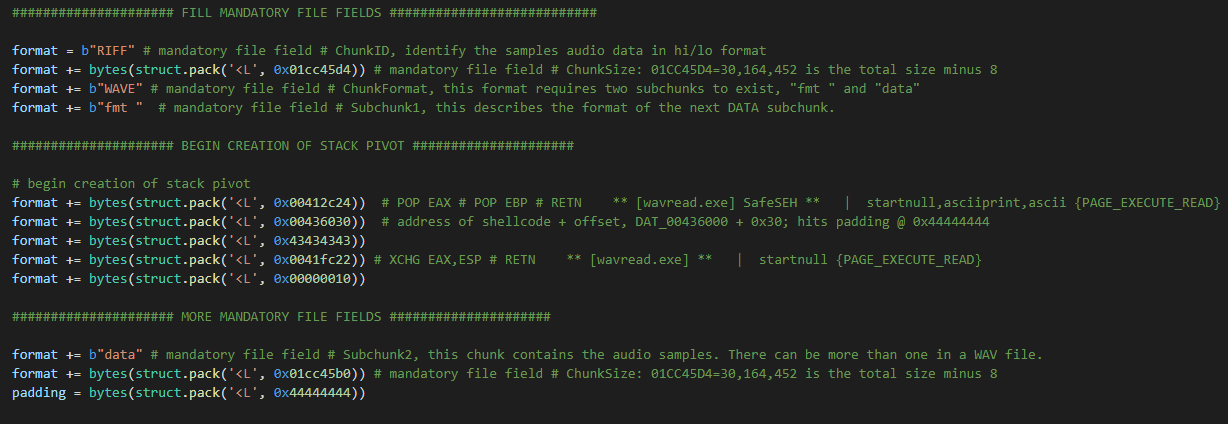
Running the executable until the ret() from FUN\_00401740 definitely proves that we have some control of EIP and afterwards…



My ideas going forward now is to use stack pivoting and SEH to bypass DEP, and then ROP to execute my shellcode.

From what I remember reading bc of the last assignment, in order for stack pivoting to work we need a buffer of memory where we can load our ROP + Shellcode and is accessible w/o causing access violations.

See here (stack pivoting): <https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/#pivot>

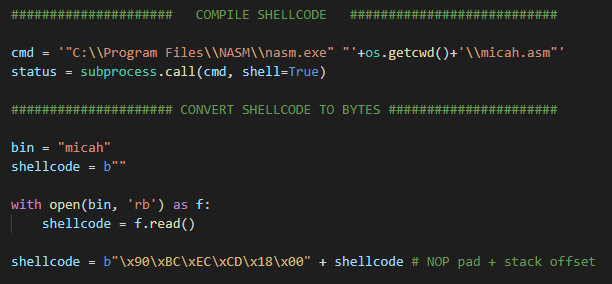


To bypass this we have created a python script that will generate two files…

* Exploit.wav → houses the actual stack pivot, rop gadgets, and payload…
* Command.txt → Kick starts the program with control of EIP…

Exploit.wav mimics the .WAV file structure - at least enough to bypass the logic shown earlier when matching file format field contents…. RIFF/WAV/fmt/data/etc…

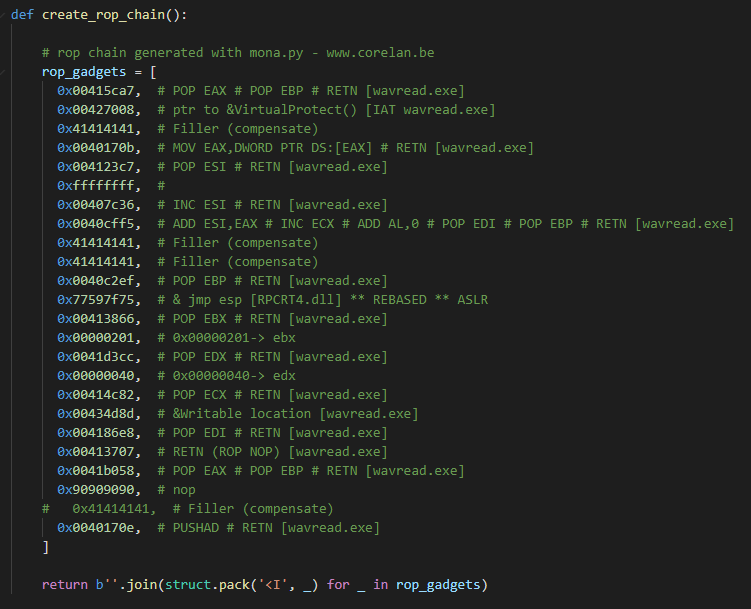
And Command is a txt file that contains enough padding to control EIP.



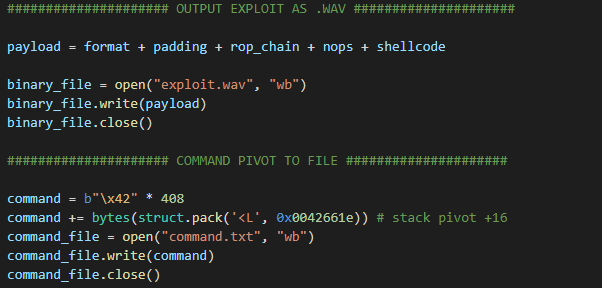
These two blocks of code are used to compile everything at runtime - it minimizes the steps needed to showcase the exploit script.

The steps are essentially…

1. Generate shellcode from .ASM using NASM
2. Read generated shellcode into variable as bytes
3. Add preceding NOP offset byte and stack offset address @ 0x18CDECBC



The rop chain used is the same one provided by !mona rop for the target binary… It didn’t need any modifications other than a borrowed line from RPCRT4.dll for a JMP ESP.

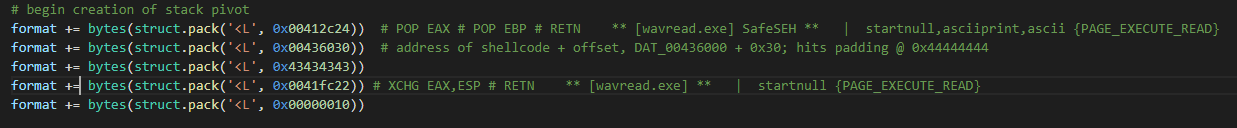


Here we output the two files and can see the structure of the payloads…

* Exploit.wav → Format + Padding + ROP + NOPs + Shellcode
* Command.txt → Padding + EIP Address @ 0042661E

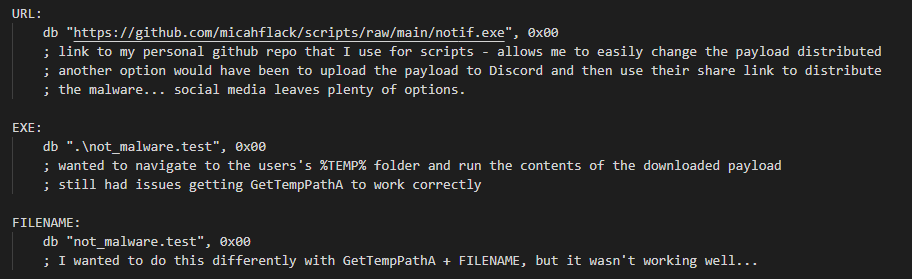
What this should look like then altogether is…

1. Wavread.exe loads the two files Exploit.wav & Command.txt
2. Command.txt overwrites the first buffer telling EIP to start @ 42661E
3. After removing 16 bytes we are inside our file format stack pivot



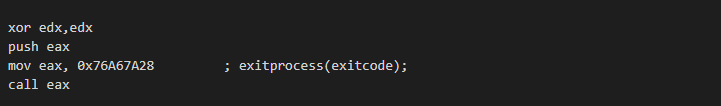
* 1. POP EAX → EAX == 0x00436030
  2. POP EBP → EBP == 0x43434343
  3. RETN → 0x0041FC22
     1. Swap EAX and ESP, then return
     2. EAX == 00436030

1. We return then to the offset that holds our shellcode at 0x00436030
   1. Being 48 bytes from the start, this is the beginning to our ROP chain
2. Execute ROP chain for VirtualProtect()
3. NOP Sled…
4. 🎉 Begin executing personal shellcode

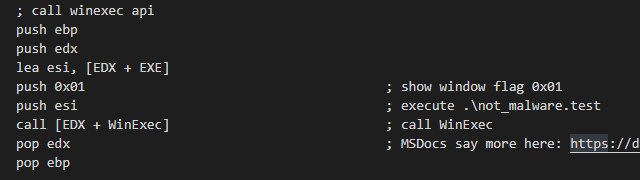


Some minor changes were made to the shellcode…

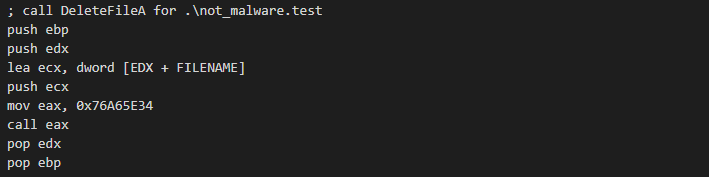
* New payload is pulled from the same github repo ‘scripts’
* File drops to same location as it was ran instead of %TEMP%
* Proper exit(), no more APPCRASH errors after detonating with payload



I documented it in the script - but no hashing was used for the call to exitprocess - none of the hashes generated would work and it kept breaking.



Simple WinExec instead of the Powershell/WMI call used previously…

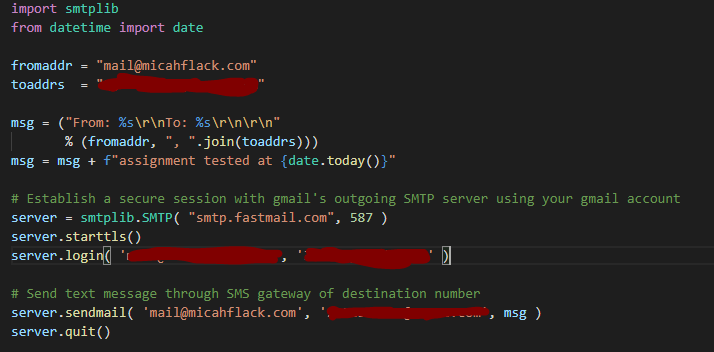


And then it deletes the second stage that was pulled from Github… .\not\_malware.test

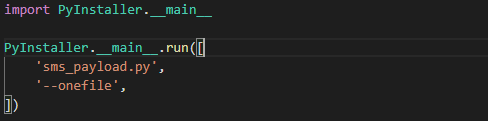
I tried messing with GetCurrentDirectoryA/GetTempPathA/GetClipboardData/VirtualAlloc/etc… but had consistent issues just getting them to correctly call… no idea why that was even happening.

But this works and I changed it up enough to be content ¯\\_(ツ)\_/¯

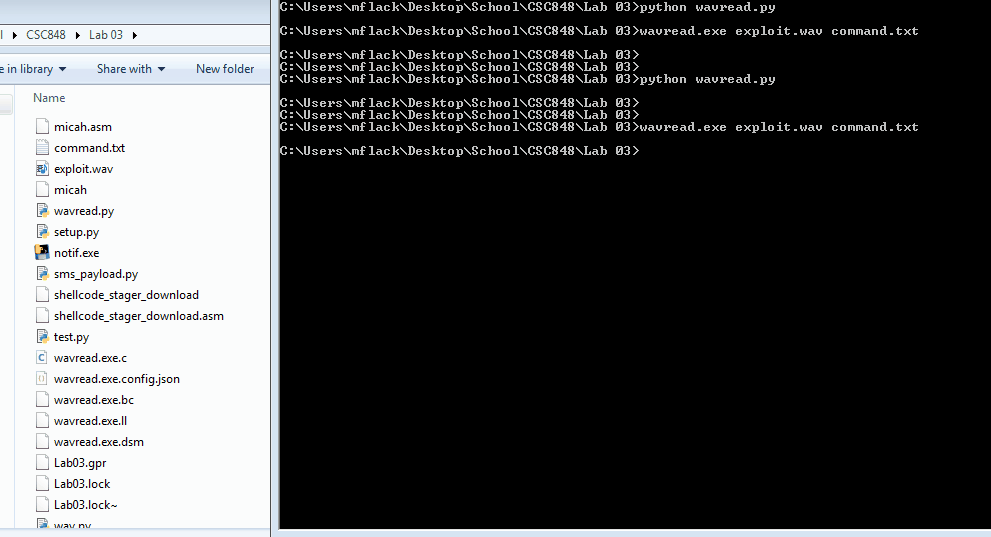
The payload pulled from Github this time is created with Pyinstaller and UPX - it has some minor “C2” communications; meaning, it literally just sends a timestamped email lol. I thought about changing the host to Discord thinking it’d be entertaining, but Github is easier for keeping track of things.



And then it was generated with…



This is just a .MOV to show the entire process from beginning to finish… Hopefully it works…



Files included with the report should be:

* micah.asm
* wavread.py
* notif.exe (if you don’t want to or can’t download successfully)