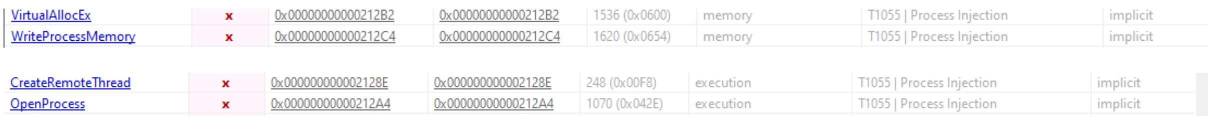
Detecting Dll Injection and Dumping the Dll Shellcode

In this section, we will reverse engineer the dll injection process.

First, let's analyze the .exe file in pestudio:

Under the imports tab, we see the following functions:

1. VirtualAllocEx
2. WriteProcessMemory
3. OpenProcess
4. CreateRemoteThread



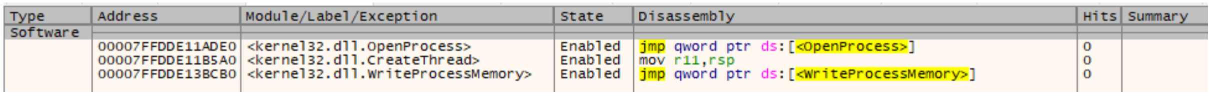
VirtualAllocEx	x	0x00000000000212B2	0x00000000000212B2	1536 (0x0600)	memory	T1055 | Process Injection	implicit
WriteProcessMemory	x	0x00000000000212C4	0x00000000000212C4	1620 (0x0654)	memory	T1055 | Process Injection	implicit
CreateRemoteThread	x	0x000000000002128E	0x000000000002128E	248 (0x00F8)	execution	T1055 | Process Injection	implicit
OpenProcess	x	0x00000000000212A4	0x00000000000212A4	1070 (0x042E)	execution	T1055 | Process Injection	implicit

These all processes are even flagged by Pestudio, by seeing all these API functions, we can see that it is either a process injection or a dll injection.

So, we will be putting breakpoints in our debugger, and analyzing the program.

So, to analyze the program, we will be putting the breakpoints at the following points:

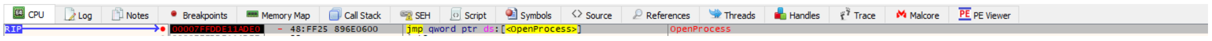
1. OpenProcess
2. WriteProcessMemory
3. CreateThread



Type	Address	Module/Label/Exception	State	Disassembly	Hits	Summary
Software						
	00007FFDDE11ADE0	<kernel32.dll.OpenProcess>	Enabled	jmp qword ptr ds:[<OpenProcess>]	0	
	00007FFDDE11B5A0	<kernel32.dll.CreateThread>	Enabled	mov r11,rsp	0	
	00007FFDDE13BCB0	<kernel32.dll.WriteProcessMemory>	Enabled	jmp qword ptr ds:[<WriteProcessMemory>]	0	

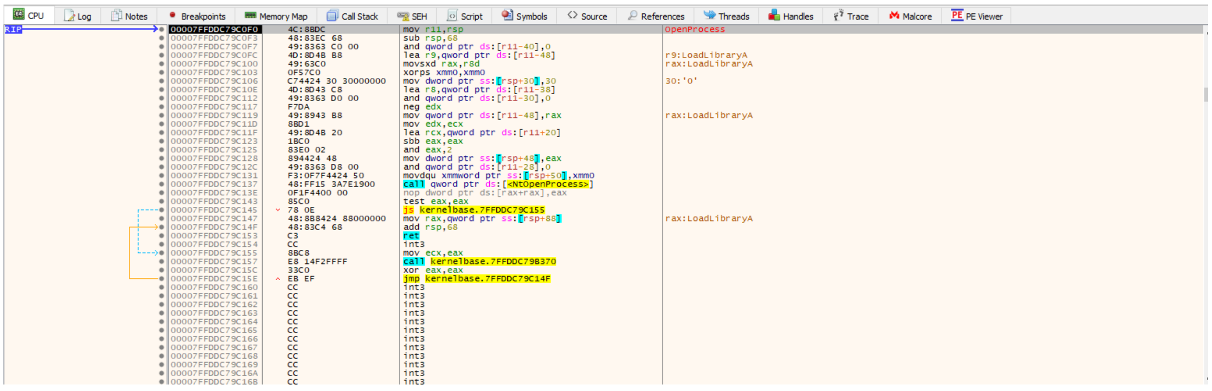
Now we can run the program

And we get a hit at the first breakpoint: OpenProcess



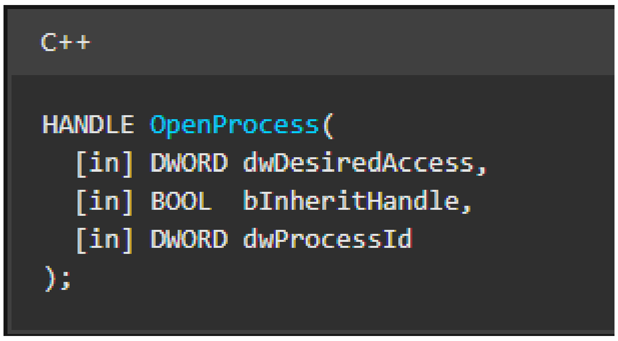
```
RIP  00007FFDDE11ADE0  - 48:FF25 896E0600  jmp qword ptr ds:[<OpenProcess>]    OpenProcess
```

And then step down, and you will be taken to the OpenProcess:



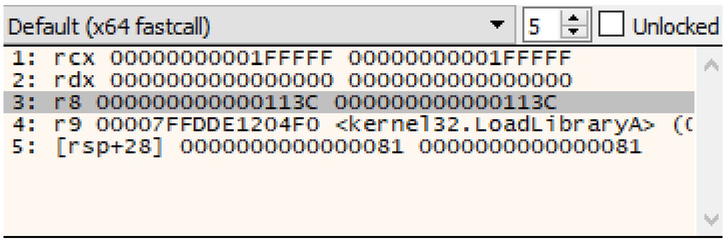
```
RIP  00007FFDDC79C0F0    4C:8BDC              mov r11,rsp                              OpenProcess
     00007FFDDC79C0F3    48:83EC 68           sub rsp,68
     00007FFDDC79C0F7    49:8363 C0 00        and qword ptr ds:[r11-40],0
     00007FFDDC79C0FC    4D:8D4B B8           lea r9,qword ptr ds:[r11-48]             r9:LoadLibraryA
     00007FFDDC79C100    49:63C0              movsxd rax,r8d                           rax:LoadLibraryA
     00007FFDDC79C103    0F57C0               xorps xmm0,xmm0
     00007FFDDC79C106    C74424 30 30000000   mov dword ptr ss:[rsp+30],30             30:'0'
     00007FFDDC79C10E    4D:8D43 C8           lea r8,qword ptr ds:[r11-38]
     00007FFDDC79C112    49:8363 D0 00        and qword ptr ds:[r11-30],0
     00007FFDDC79C117    F7DA                 neg edx
     00007FFDDC79C119    49:8943 B8           mov qword ptr ds:[r11-48],rax            rax:LoadLibraryA
     00007FFDDC79C11D    8BD1                 mov edx,ecx
     00007FFDDC79C11F    49:8D4B 20           lea rcx,qword ptr ds:[r11+20]
     00007FFDDC79C123    1BC0                 sbb eax,eax
     00007FFDDC79C125    83E0 02              and eax,2
     00007FFDDC79C128    894424 48            mov dword ptr ss:[rsp+48],eax
     00007FFDDC79C12C    49:8363 D8 00        and qword ptr ds:[r11-28],0
     00007FFDDC79C131    F3:0F7F4424 50       movdqu xmmword ptr ss:[rsp+50],xmm0
     00007FFDDC79C137    48:FF15 3A7E1900     call qword ptr ds:[<NtOpenProcess>]
     00007FFDDC79C13E    0F1F4400 00          nop dword ptr ds:[rax+rax],eax
     00007FFDDC79C143    85C0                 test eax,eax
     00007FFDDC79C145  v 78 0E                js kernelbase.7FFDDC79C155
     00007FFDDC79C147    48:8B8424 88000000   mov rax,qword ptr ss:[rsp+88]            rax:LoadLibraryA
     00007FFDDC79C14F    48:83C4 68           add rsp,68
     00007FFDDC79C153    C3                   ret
     00007FFDDC79C154    CC                   int3
     00007FFDDC79C155    8BC8                 mov ecx,eax
     00007FFDDC79C157    E8 14F2FFFF          call kernelbase.7FFDDC79B370
     00007FFDDC79C15C    33C0                 xor eax,eax
     00007FFDDC79C15E  ^ EB EF                jmp kernelbase.7FFDDC79C14F
     00007FFDDC79C160    CC                   int3
     00007FFDDC79C161    CC                   int3
     00007FFDDC79C162    CC                   int3
     00007FFDDC79C163    CC                   int3
     00007FFDDC79C164    CC                   int3
     00007FFDDC79C165    CC                   int3
     00007FFDDC79C166    CC                   int3
     00007FFDDC79C167    CC                   int3
     00007FFDDC79C168    CC                   int3
     00007FFDDC79C169    CC                   int3
     00007FFDDC79C16A    CC                   int3
     00007FFDDC79C16B    CC                   int3
```

We know that it takes 3 parameters, and in the third parameter, it takes the pid of the program



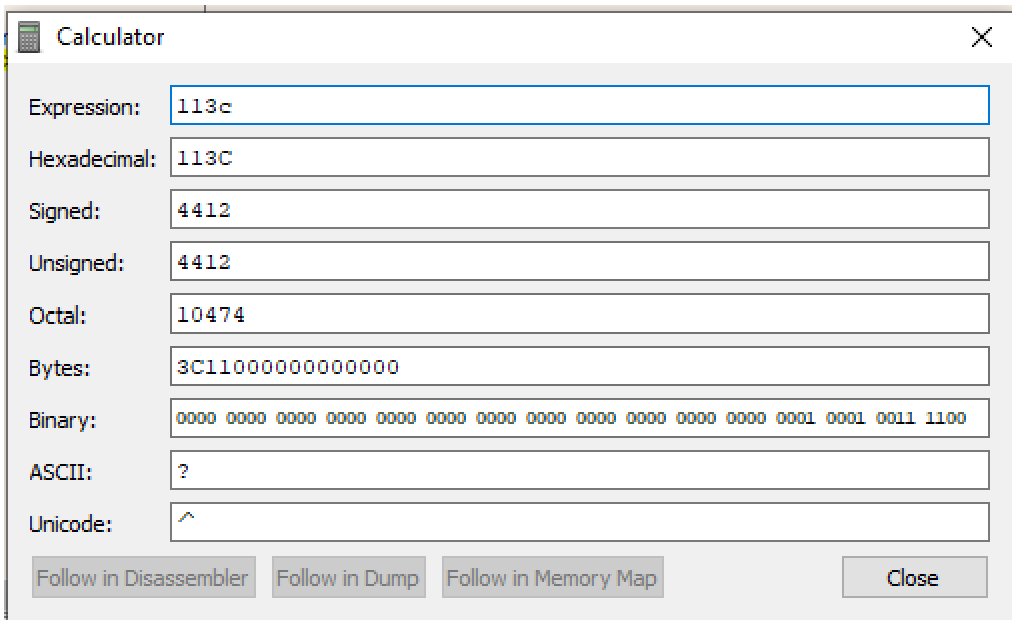
```C++
HANDLE OpenProcess(
  [in] DWORD dwDesiredAccess,
  [in] BOOL  bInheritHandle,
  [in] DWORD dwProcessId
);
```

So, analyze the 3rd parameter in the xdbg.



```
Default (x64 fastcall)   5   Unlocked
1: rcx 00000000001FFFFF 00000000001FFFFF
2: rdx 0000000000000000 0000000000000000
3: r8 000000000000113C 000000000000113C
4: r9 00007FFDDE1204F0 <kernel32.LoadLibraryA> (0
5: [rsp+28] 0000000000000081 0000000000000081
```

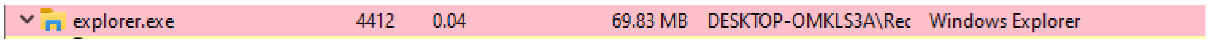
In HEX it is given 113C, so let's analyze it in the calculator.



Calculator

Expression:	113c
Hexadecimal:	113C
Signed:	4412
Unsigned:	4412
Octal:	10474
Bytes:	3C11000000000000
Binary:	0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 0001 0011 1100
ASCII:	?
Unicode:	^

We see that we get 4412 as the pid.

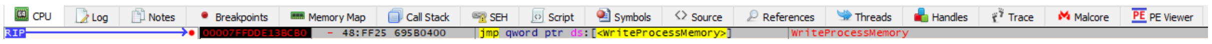


```
explorer.exe    4412    0.04    69.83 MB    DESKTOP-OMKLS3A\Rec    Windows Explorer
```

We can see that in process Hacker, we get the pid of explorer.exe to be 4412, so it is trying to inject in explorer.exe

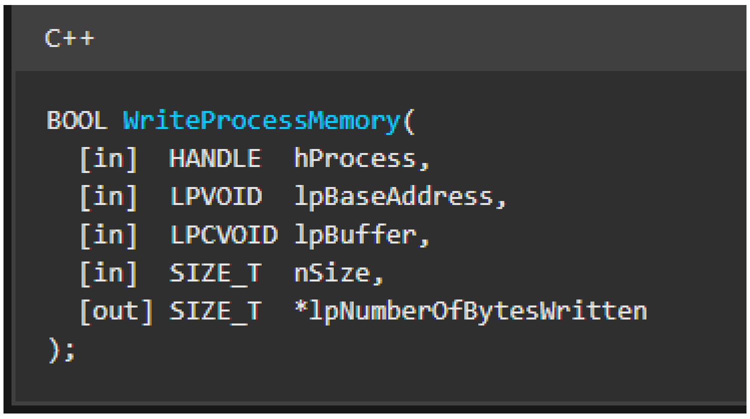
Double-click on the explorer, then open the memory tab, but here we don't know at which address it will inject, so for that we need to rely on the next process WriteProcessMemory.

So, run the program, and we another breakpoint: WriteProcessMemory



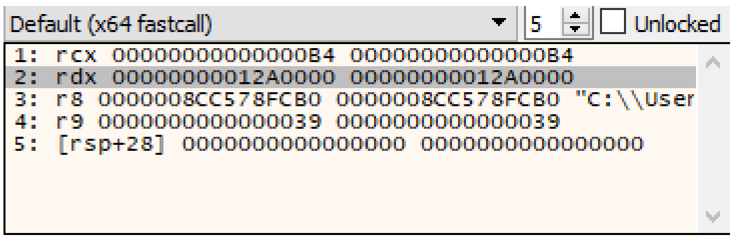
```
RIP  00007FFDDE13BCB0  - 48:FF25 695B0400    jmp qword ptr ds:[<WriteProcessMemory>]    WriteProcessMemory
```

Press Step Over, then we know that WriteProcessMemory takes 5 parameters



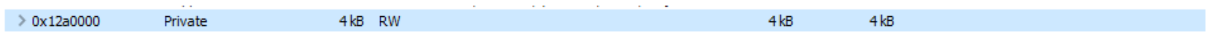
```C++
BOOL WriteProcessMemory(
  [in]  HANDLE  hProcess,
  [in]  LPVOID  lpBaseAddress,
  [in]  LPCVOID lpBuffer,
  [in]  SIZE_T  nSize,
  [out] SIZE_T  *lpNumberOfBytesWritten
);
```

So, we are interested in the 2nd parameter, because it will give us the address, of where it is going to write in the memory.



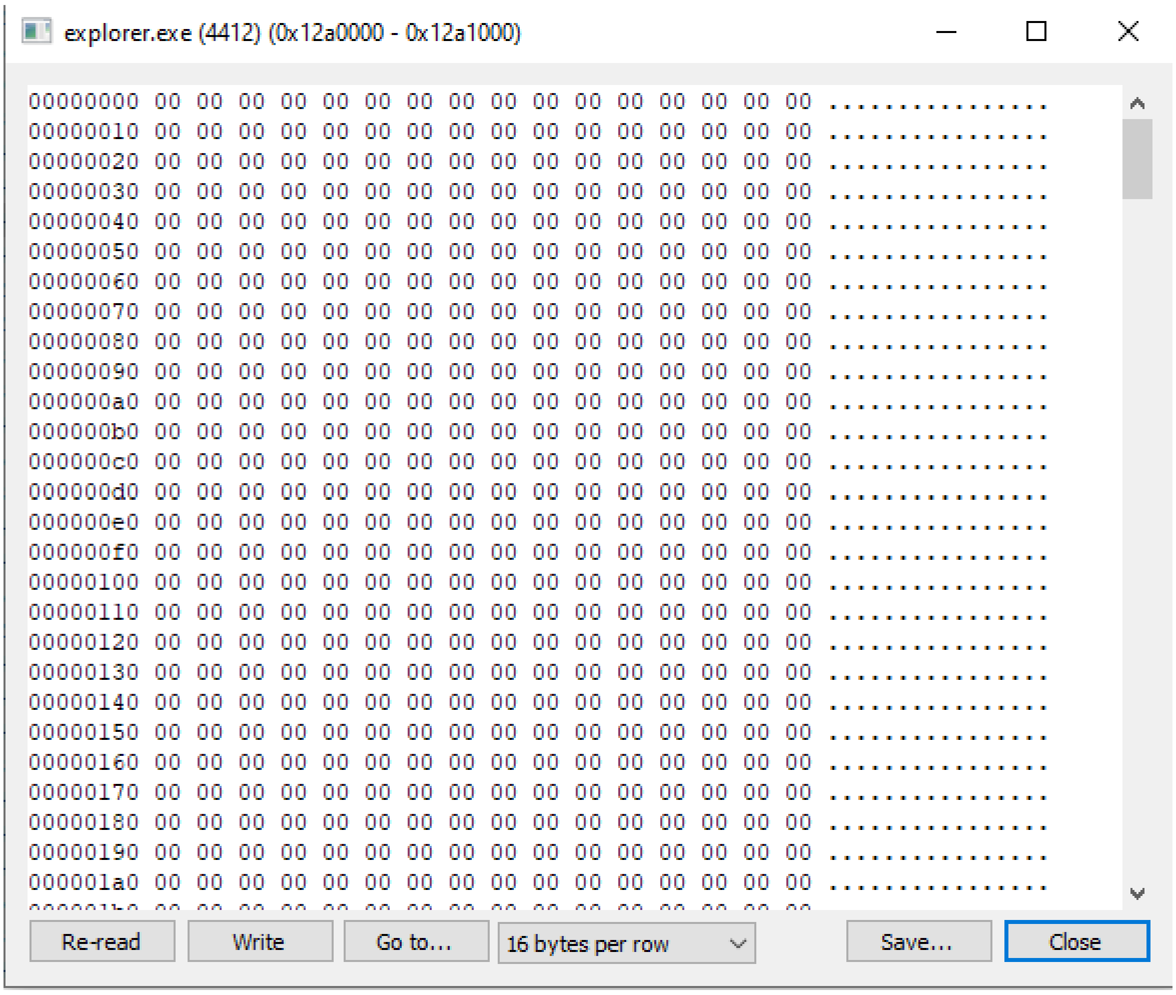
```
Default (x64 fastcall)          5   Unlocked
1: rcx 00000000000000B4 00000000000000B4
2: rdx 00000000012A0000 00000000012A0000
3: r8 0000008CC578FCB0 0000008CC578FCB0 "C:\\User
4: r9 0000000000000039 0000000000000039
5: [rsp+28] 0000000000000000 0000000000000000
```

We can see here in the process hacker in the memory tab of explorer.exe's:



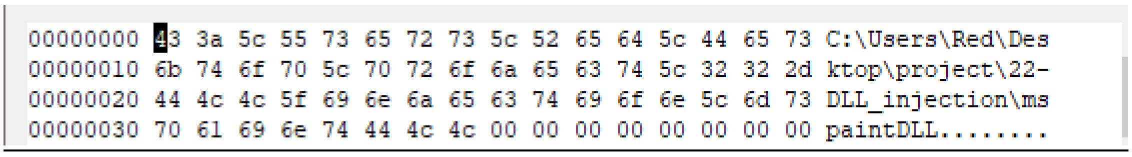
```
> 0x12a0000    Private    4 kB   RW    4 kB    4 kB
```

If we double-click on it, and open the address, we can see that nothing has been written yet:



```
00000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000050 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000000f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000110 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000120 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000140 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000150 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000160 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000170 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

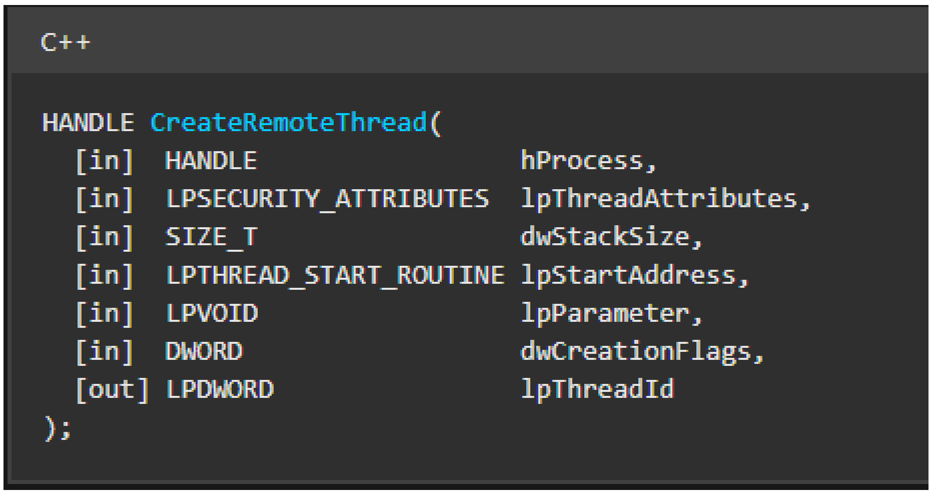
So, we need to see what will be filled.



```
00000000 43 3a 5c 55 73 65 72 73 5c 52 65 64 5c 44 65 73 C:\Users\Red\Des
00000010 6b 74 6f 70 5c 70 72 6f 6a 65 63 74 5c 32 32 2d ktop\project\22-
00000020 44 4c 4c 5f 69 6e 6a 65 63 74 69 6f 6e 5c 6d 73 DLL_injection\ms
00000030 70 61 69 6e 74 44 4c 4c 00 00 00 00 00 00 00 00 paintDLL........
```

We can see that it is filled with some dll files, and its location is mentioned here.

So, we can go to its location, and check what is the dll file, or if we want to check what happens in the CreateThread process, we can see that it takes 7 parameters, and we will see the 5th parameter. And the 4th parameter is used to load the libraries.



```
C++

HANDLE CreateRemoteThread(
  [in]  HANDLE                 hProcess,
  [in]  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  [in]  SIZE_T                 dwStackSize,
  [in]  LPTHREAD_START_ROUTINE lpStartAddress,
  [in]  LPVOID                 lpParameter,
  [in]  DWORD                  dwCreationFlags,
  [out] LPDWORD                lpThreadId
);
```

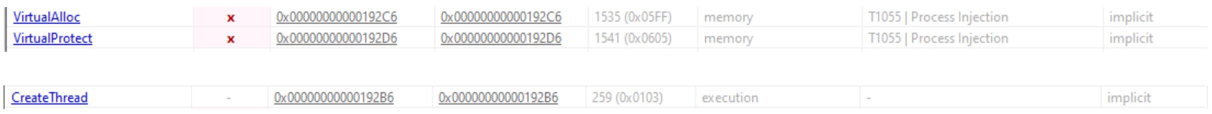
Because in the 5th parameter, is the return address, which is the same as the address, that we saw in Process Hacker in the memory tab. So, it is confirmed that it is using the CreateThread process to load the dll files.

Now we can open that dll in xdbg, and analyze it,

Let's first analyze it in the pestudio:

We can see the following functions:

1. VirtualAlloc
2. VirtualProtect
3. CreateThread

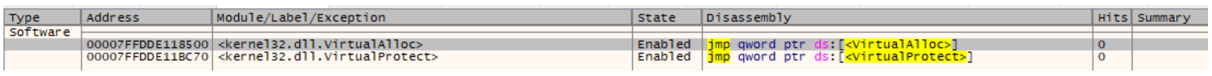


VirtualAlloc	x	0x00000000000192C6	0x00000000000192C6	1535 (0x05FF)	memory	T1055 | Process Injection	implicit
VirtualProtect	x	0x00000000000192D6	0x00000000000192D6	1541 (0x0605)	memory	T1055 | Process Injection	implicit
CreateThread	-	0x00000000000192B6	0x00000000000192B6	259 (0x0103)	execution	-	implicit

So, we can say that it is trying to execute the shell code.

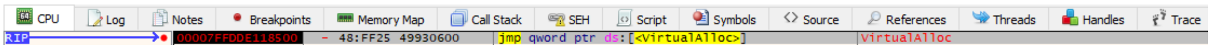
Now open the dll in xdbg.

Make sure to put the breakpoint at both VirtualAlloc, and at VirtualProtect, and now run the program.



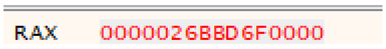
```
Type     | Address          | Module/Label/Exception          | State   | Disassembly                          | Hits | Summary
Software |                  |                                 |         |                                      |      |
         | 00007FFDDE118500 | <kernel32.dll.VirtualAlloc>     | Enabled | jmp qword ptr ds:[<VirtualAlloc>]    | 0    |
         | 00007FFDDE11BC70 | <kernel32.dll.VirtualProtect>   | Enabled | jmp qword ptr ds:[<VirtualProtect>]  | 0    |
```

We hit our first breakpoint at VirtualAlloc



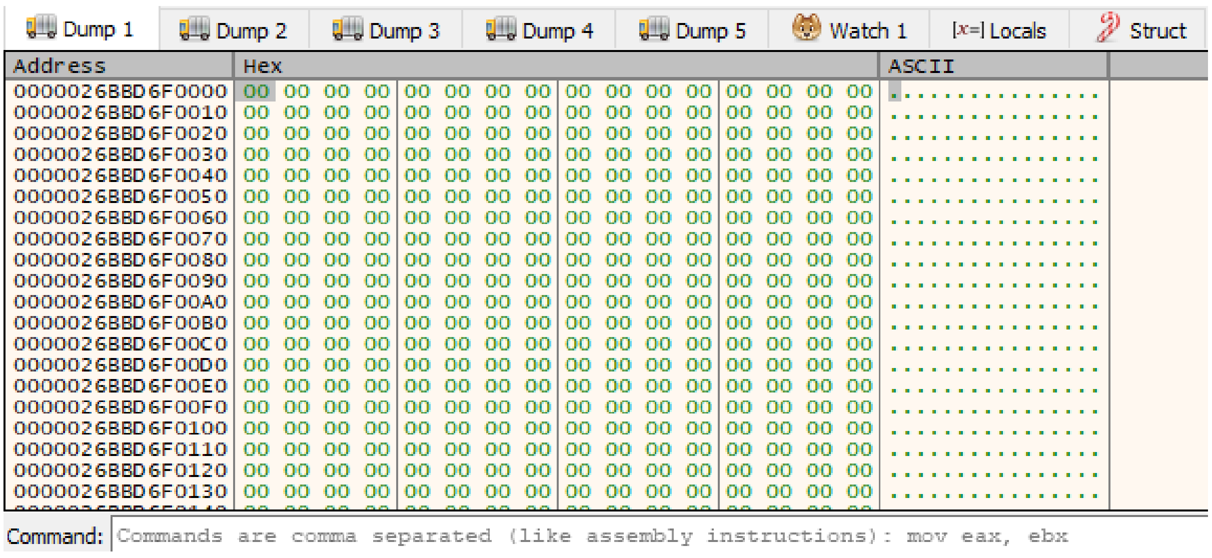
```
RIP ──────> ● 00007FFDDE118500 - 48:FF25 49930600   jmp qword ptr ds:[<VirtualAlloc>]    VirtualAlloc
```

Now to check what address has been allocated, we can run through the user code, and see what's the address.



```
RAX    0000026BBD6F0000
```

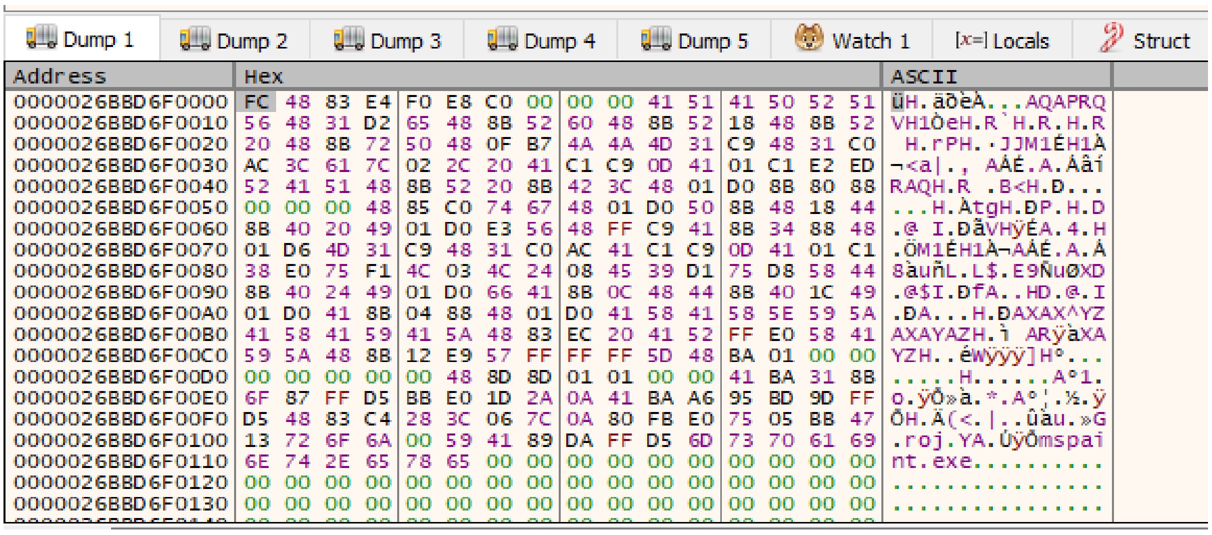
Follow this address in the dump:



```
Address           Hex                                               ASCII
0000026BBD6F0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F00A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F00B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F00C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F00D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F00E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F00F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
Command: Commands are comma separated (like assembly instructions): mov eax, ebx
```

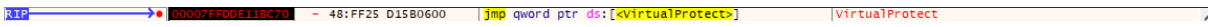
Now run the program once again:

We hit the breakpoint at VirtualProtect, and at the same time, we see that the shellcode is injected into the allocated memory.



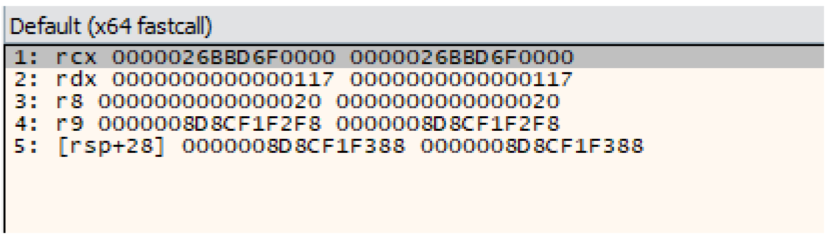
```
Address           Hex                                               ASCII
0000026BBD6F0000  FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51  üH.äðèÀ...AQAPRQ
0000026BBD6F0010  56 48 31 D2 65 48 8B 52 60 48 8B 52 18 48 8B 52  VH1ÒeH.R`H.R.H.R
0000026BBD6F0020  20 48 8B 72 50 48 0F B7 4A 4A 4D 31 C9 48 31 C0   H.rPH.·JJM1ÉH1À
0000026BBD6F0030  AC 3C 61 7C 02 2C 20 41 C1 C9 0D 41 01 C1 E2 ED  ¬<a|., AÁÉ.A.Áâí
0000026BBD6F0040  52 41 51 48 8B 52 20 8B 42 3C 48 01 D0 8B 80 88  RAQH.R .B<H.Ð...
0000026BBD6F0050  00 00 00 48 85 C0 74 67 48 01 D0 50 8B 48 18 44  ...H.ÀtgH.ÐP.H.D
0000026BBD6F0060  8B 40 20 49 01 D0 E3 56 48 FF C9 41 8B 34 88 48  .@ I.ÐãVHÿÉA.4.H
0000026BBD6F0070  01 D6 4D 31 C9 48 31 C0 AC 41 C1 C9 0D 41 01 C1  .ÖM1ÉH1À¬AÁÉ.A.Á
0000026BBD6F0080  38 E0 75 F1 4C 03 4C 24 08 45 39 D1 75 D8 58 44  8àuñL.L$.E9ÑuØXD
0000026BBD6F0090  8B 40 24 49 01 D0 66 41 8B 0C 48 44 8B 40 1C 49  .@$I.ÐfA..HD.@.I
0000026BBD6F00A0  01 D0 41 8B 04 88 48 01 D0 41 58 41 58 5E 59 5A  .ÐA...H.ÐAXAX^YZ
0000026BBD6F00B0  41 58 41 59 41 5A 48 83 EC 20 41 52 FF E0 58 41  AXAYAZH.ì ARÿàXA
0000026BBD6F00C0  59 5A 48 8B 12 E9 57 FF FF FF 5D 48 BA 01 00 00  YZH..éWÿÿÿ]Hº...
0000026BBD6F00D0  00 00 00 00 00 48 8D 8D 01 01 00 00 41 BA 31 8B  .....H......Aº1.
0000026BBD6F00E0  6F 87 FF D5 BB E0 1D 2A 0A 41 BA A6 95 BD 9D FF  o.ÿÕ»à.*.Aº¦.½.ÿ
0000026BBD6F00F0  D5 48 83 C4 28 3C 06 7C 0A 80 FB E0 75 05 BB 47  ÕH.Ä(<.|..ûàu.»G
0000026BBD6F0100  13 72 6F 6A 00 59 41 89 DA FF D5 6D 73 70 61 69  .roj.YA.ÚÿÕmspai
0000026BBD6F0110  6E 74 2E 65 78 65 00 00 00 00 00 00 00 00 00 00  nt.exe..........
0000026BBD6F0120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Breakpoint at VirtualProtect:



```
RIP ──→● 00007FFDDE11BC70  - 48:FF25 D15B0600  jmp qword ptr ds:[<VirtualProtect>]   VirtualProtect
```

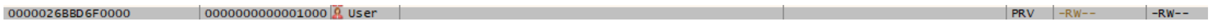
And if see the 1st parameter, it shows us where it might alter the permission:



```
Default (x64 fastcall)
1: rcx 0000026BBD6F0000 0000026BBD6F0000
2: rdx 0000000000000117 0000000000000117
3: r8 0000000000000020 0000000000000020
4: r9 0000008D8CF1F2F8 0000008D8CF1F2F8
5: [rsp+28] 0000008D8CF1F388 0000008D8CF1F388
```

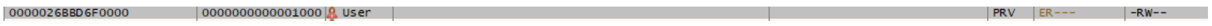
It is the same location, where the shellcode has been injected.

If we follow it in the Memory Map, we can see the permission: Read and Write.



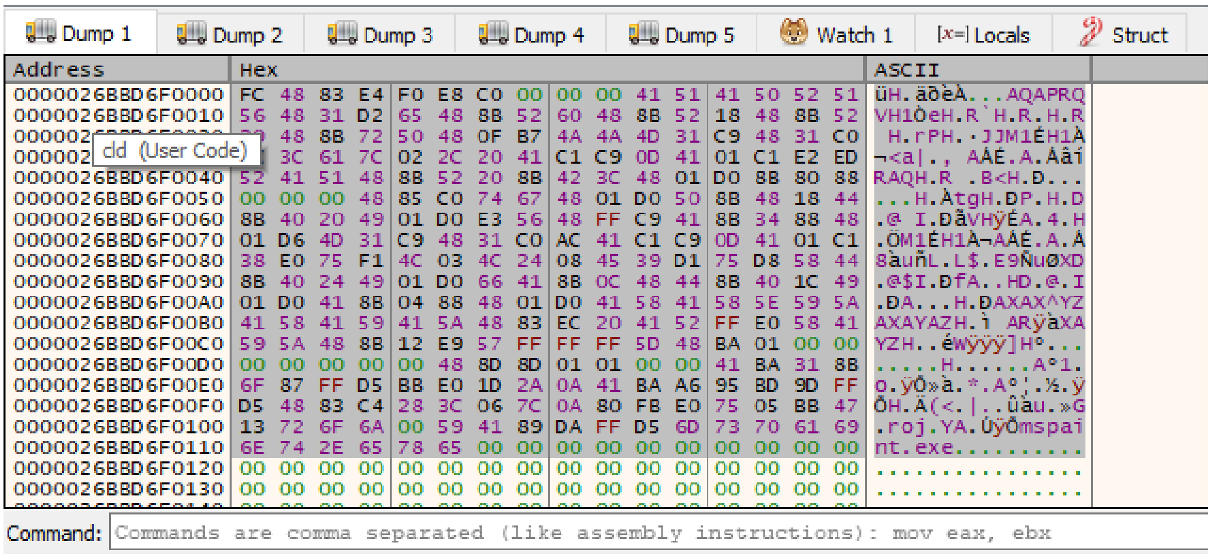
```
0000026BBD6F0000 | 0000000000001000 | User | | | PRV | -RW-- | -RW--
```

Now if run the user code, and see the permission, we can see that the permission has been changed to: Execute and Read



```
0000026BBD6F0000 | 0000000000001000 | User | | | PRV | ER--- | -RW--
```

Now we can just extract the shellcode, and check what it is doing:

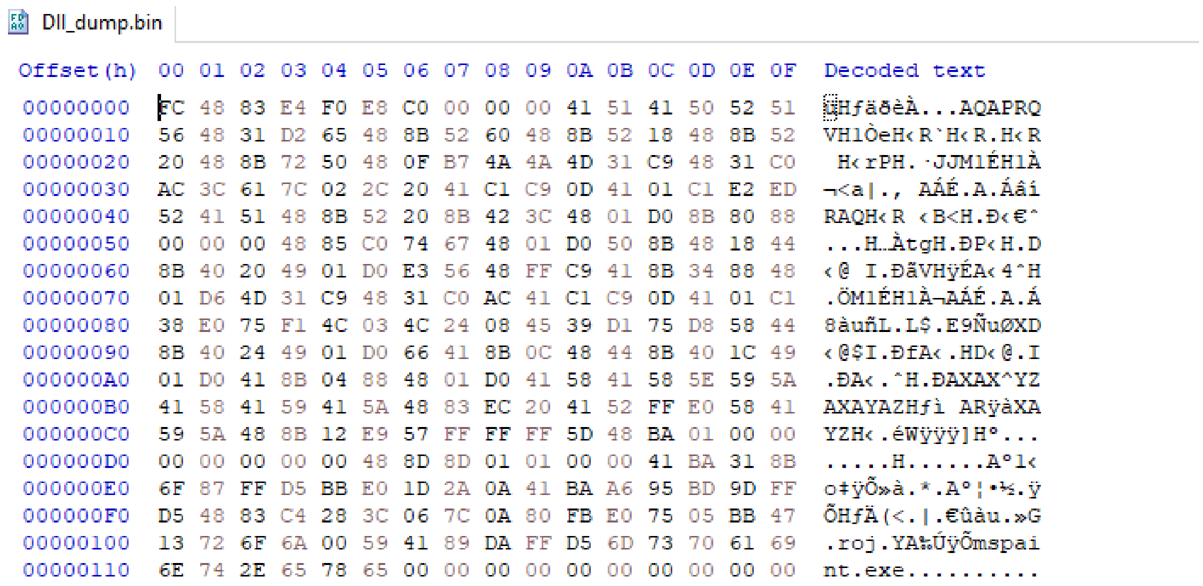


```
Address           Hex                                               ASCII
0000026BBD6F0000  FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51  üH.äðèÀ...AQAPRQ
0000026BBD6F0010  56 48 31 D2 65 48 8B 52 60 48 8B 52 18 48 8B 52  VH1ÒeH.R`H.R.H.R
000002 cld (User Code) 48 8B 72 50 48 0F B7 4A 4A 4D 31 C9 48 31 C0   H.rPH.·JJM1ÉH1À
000002               3C 61 7C 02 2C 20 41 C1 C9 0D 41 01 C1 E2 ED  ¬<a|., AÁÉ.A.Áâí
0000026BBD6F0040  52 41 51 48 8B 52 20 8B 42 3C 48 01 D0 8B 80 88  RAQH.R .B<H.Ð...
0000026BBD6F0050  00 00 00 48 85 C0 74 67 48 01 D0 50 8B 48 18 44  ...H.ÀtgH.ÐP.H.D
0000026BBD6F0060  8B 40 20 49 01 D0 E3 56 48 FF C9 41 8B 34 88 48  .@ I.ÐãVHÿÉA.4.H
0000026BBD6F0070  01 D6 4D 31 C9 48 31 C0 AC 41 C1 C9 0D 41 01 C1  .ÖM1ÉH1À¬AÁÉ.A.Á
0000026BBD6F0080  38 E0 75 F1 4C 03 4C 24 08 45 39 D1 75 D8 58 44  8àuñL.L$.E9ÑuØXD
0000026BBD6F0090  8B 40 24 49 01 D0 66 41 8B 0C 48 44 8B 40 1C 49  .@$I.ÐfA..HD.@.I
0000026BBD6F00A0  01 D0 41 8B 04 88 48 01 D0 41 58 41 58 5E 59 5A  .ÐA...H.ÐAXAX^YZ
0000026BBD6F00B0  41 58 41 59 41 5A 48 83 EC 20 41 52 FF E0 58 41  AXAYAZH.ì ARÿàXA
0000026BBD6F00C0  59 5A 48 8B 12 E9 57 FF FF FF 5D 48 BA 01 00 00  YZH..éWÿÿÿ]H°...
0000026BBD6F00D0  00 00 00 00 00 48 8D 8D 01 01 00 00 41 BA 31 8B  .....H......A°1.
0000026BBD6F00E0  6F 87 FF D5 BB E0 1D 2A 0A 41 BA A6 95 BD 9D FF  o.ÿÕ»à.*.A°¦.½.ÿ
0000026BBD6F00F0  D5 48 83 C4 28 3C 06 7C 0A 80 FB E0 75 05 BB 47  ÕH.Ä(<.|..ûàu.»G
0000026BBD6F0100  13 72 6F 6A 00 59 41 89 DA FF D5 6D 73 70 61 69  .roj.YA.ÚÿÕmspai
0000026BBD6F0110  6E 74 2E 65 78 65 00 00 00 00 00 00 00 00 00 00  nt.exe..........
0000026BBD6F0120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0000026BBD6F0130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Command: Commands are comma separated (like assembly instructions): mov eax, ebx

Now let’s see what’s the shellcode doing:

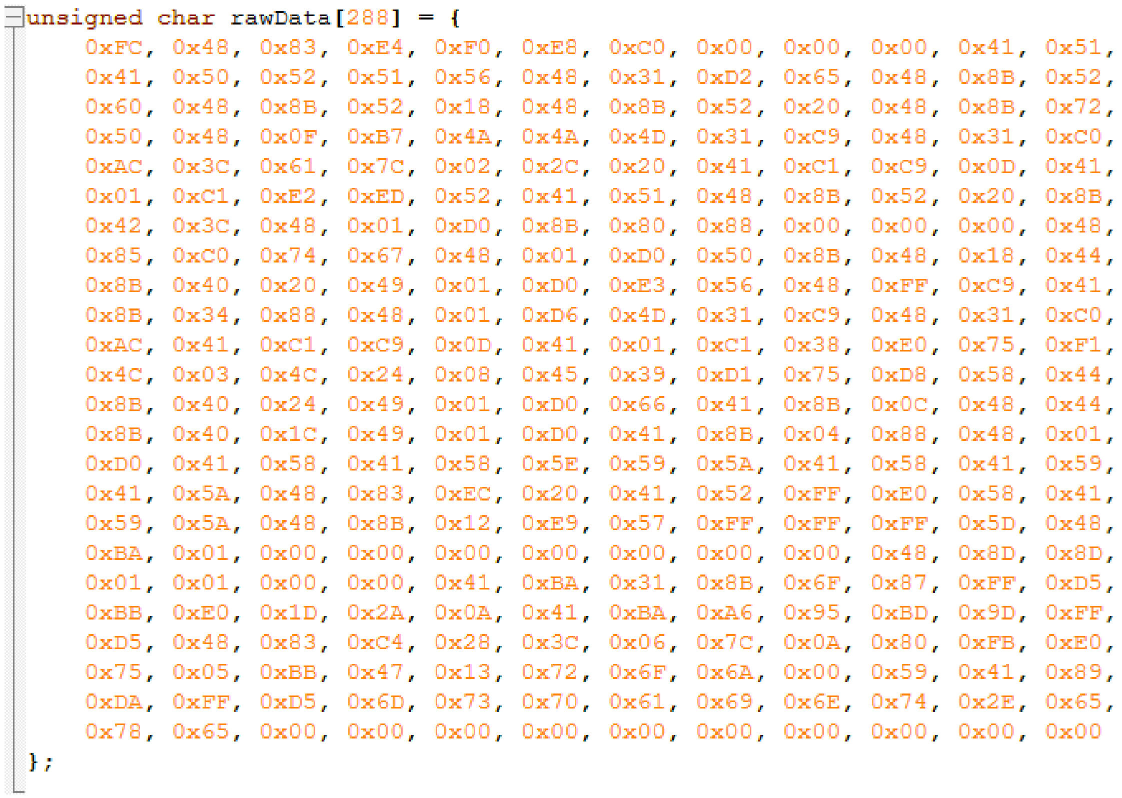
Open it in the Hexeditor and then extract it to a .c file, and then put the shellcode in the shellcode runner, and see what’s happening.



Dll_dump.bin

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  FC 48 83 E4 F0 E8 C0 00 00 00 41 51 41 50 52 51  üHƒäðèÀ...AQAPRQ
00000010  56 48 31 D2 65 48 8B 52 60 48 8B 52 18 48 8B 52  VH1ÒeH‹R`H‹R.H‹R
00000020  20 48 8B 72 50 48 0F B7 4A 4A 4D 31 C9 48 31 C0   H‹rPH.·JJM1ÉH1À
00000030  AC 3C 61 7C 02 2C 20 41 C1 C9 0D 41 01 C1 E2 ED  ¬<a|., AÁÉ.A.Áâí
00000040  52 41 51 48 8B 52 20 8B 42 3C 48 01 D0 8B 80 88  RAQH‹R ‹B<H.Ð‹€ˆ
00000050  00 00 00 48 85 C0 74 67 48 01 D0 50 8B 48 18 44  ...H…ÀtgH.ÐP‹H.D
00000060  8B 40 20 49 01 D0 E3 56 48 FF C9 41 8B 34 88 48  ‹@ I.ÐãVHÿÉA‹4ˆH
00000070  01 D6 4D 31 C9 48 31 C0 AC 41 C1 C9 0D 41 01 C1  .ÖM1ÉH1À¬AÁÉ.A.Á
00000080  38 E0 75 F1 4C 03 4C 24 08 45 39 D1 75 D8 58 44  8àuñL.L$.E9ÑuØXD
00000090  8B 40 24 49 01 D0 66 41 8B 0C 48 44 8B 40 1C 49  ‹@$I.ÐfA‹.HD‹@.I
000000A0  01 D0 41 8B 04 88 48 01 D0 41 58 41 58 5E 59 5A  .ÐA‹.ˆH.ÐAXAX^YZ
000000B0  41 58 41 59 41 5A 48 83 EC 20 41 52 FF E0 58 41  AXAYAZHƒì ARÿàXA
000000C0  59 5A 48 8B 12 E9 57 FF FF FF 5D 48 BA 01 00 00  YZH‹.éWÿÿÿ]H°...
000000D0  00 00 00 00 00 48 8D 8D 01 01 00 00 41 BA 31 8B  .....H......A°1‹
000000E0  6F 87 FF D5 BB E0 1D 2A 0A 41 BA A6 95 BD 9D FF  o‡ÿÕ»à.*.A°¦•½.ÿ
000000F0  D5 48 83 C4 28 3C 06 7C 0A 80 FB E0 75 05 BB 47  ÕHƒÄ(<.|.€ûàu.»G
00000100  13 72 6F 6A 00 59 41 89 DA FF D5 6D 73 70 61 69  .roj.YA‰ÚÿÕmspai
00000110  6E 74 2E 65 78 65 00 00 00 00 00 00 00 00 00 00  nt.exe..........
```

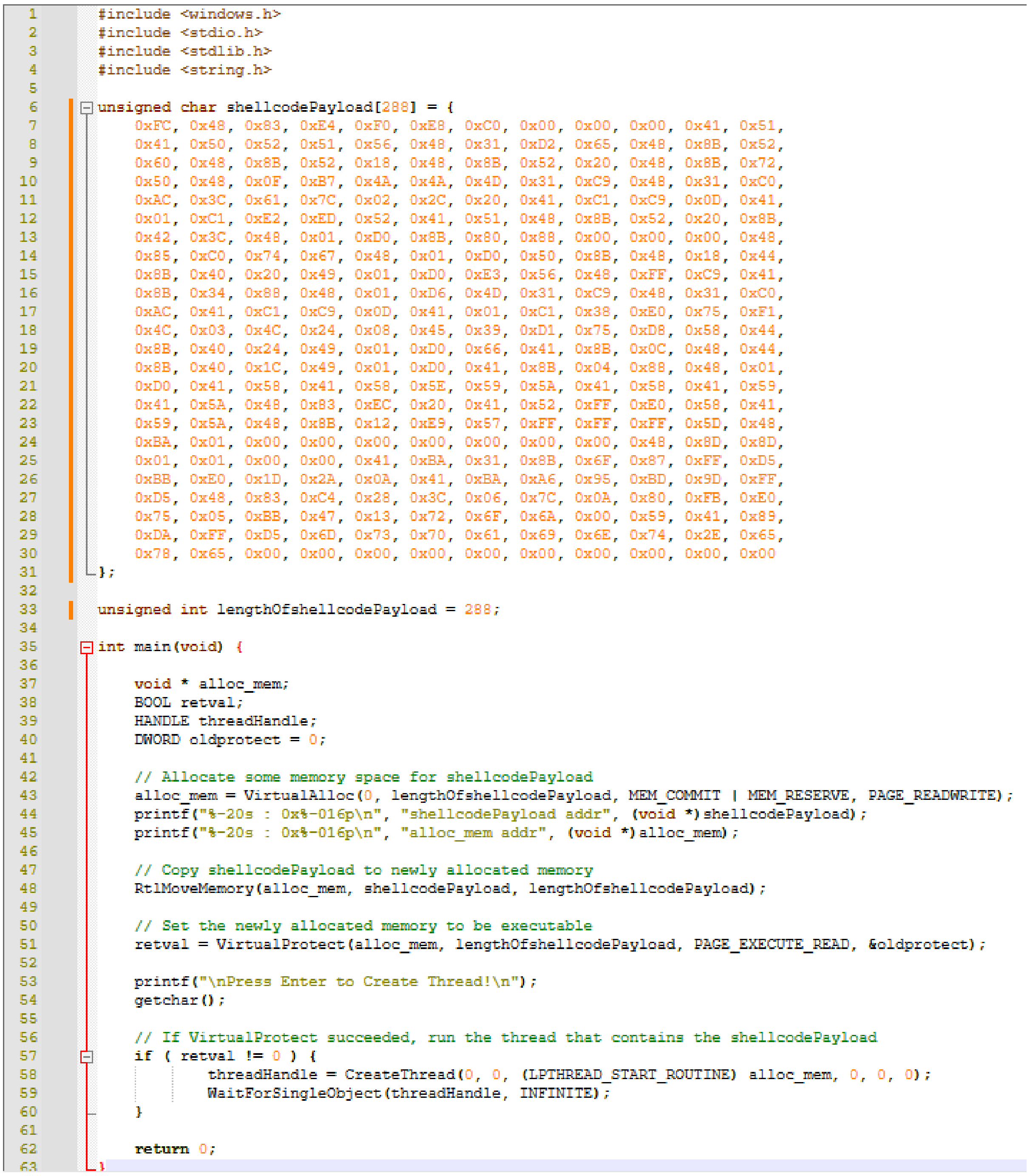
Here’s the .c file of the extracted file:



```
unsigned char rawData[288] = {
	0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xC0, 0x00, 0x00, 0x00, 0x41, 0x51,
	0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65, 0x48, 0x8B, 0x52,
	0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x48, 0x8B, 0x72,
	0x50, 0x48, 0x0F, 0xB7, 0x4A, 0x4A, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
	0xAC, 0x3C, 0x61, 0x7C, 0x02, 0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41,
	0x01, 0xC1, 0xE2, 0xED, 0x52, 0x41, 0x51, 0x48, 0x8B, 0x52, 0x20, 0x8B,
	0x42, 0x3C, 0x48, 0x01, 0xD0, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
	0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x50, 0x8B, 0x48, 0x18, 0x44,
	0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 0x41,
	0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
	0xAC, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75, 0xF1,
	0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
	0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,
	0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x41, 0x8B, 0x04, 0x88, 0x48, 0x01,
	0xD0, 0x41, 0x58, 0x41, 0x58, 0x5E, 0x59, 0x5A, 0x41, 0x58, 0x41, 0x59,
	0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20, 0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41,
	0x59, 0x5A, 0x48, 0x8B, 0x12, 0xE9, 0x57, 0xFF, 0xFF, 0xFF, 0x5D, 0x48,
	0xBA, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x8D,
	0x01, 0x01, 0x00, 0x00, 0x41, 0xBA, 0x31, 0x8B, 0x6F, 0x87, 0xFF, 0xD5,
	0xBB, 0xE0, 0x1D, 0x2A, 0x0A, 0x41, 0xBA, 0xA6, 0x95, 0xBD, 0x9D, 0xFF,
	0xD5, 0x48, 0x83, 0xC4, 0x28, 0x3C, 0x06, 0x7C, 0x0A, 0x80, 0xFB, 0xE0,
	0x75, 0x05, 0xBB, 0x47, 0x13, 0x72, 0x6F, 0x6A, 0x00, 0x59, 0x41, 0x89,
	0xDA, 0xFF, 0xD5, 0x6D, 0x73, 0x70, 0x61, 0x69, 0x6E, 0x74, 0x2E, 0x65,
	0x78, 0x65, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};
```

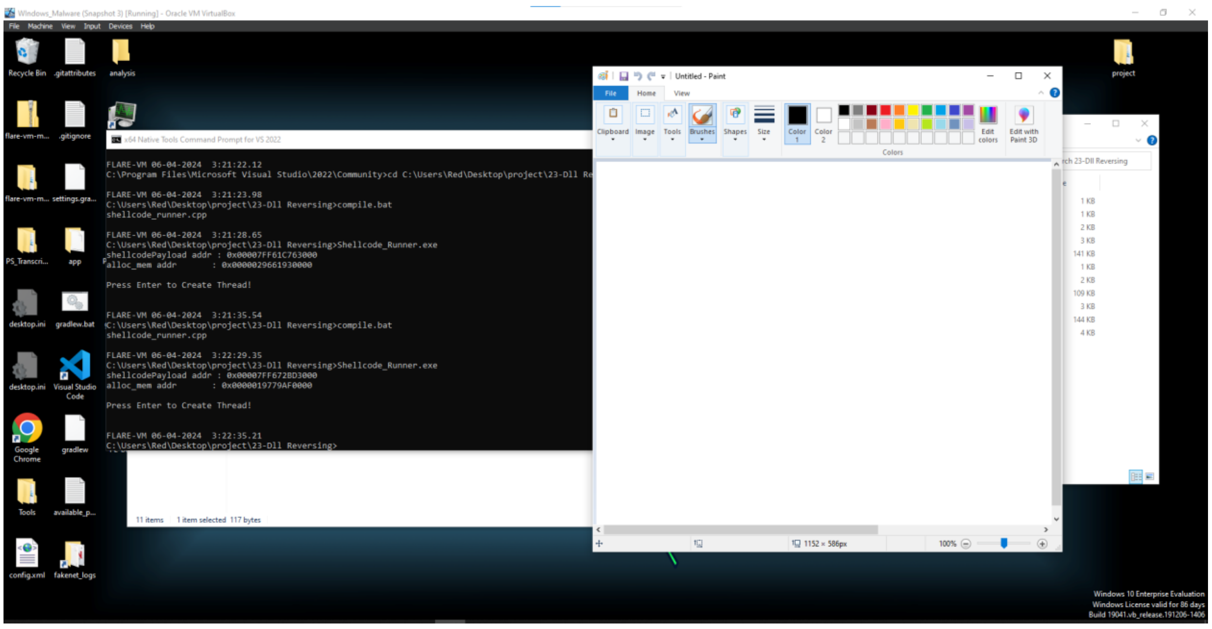
Paste this shellcode in the shellcode runner

Here’s the shellcode runner which we will use:



```c
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

unsigned char shellcodePayload[288] = {
    0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xC0, 0x00, 0x00, 0x00, 0x41, 0x51,
    0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xD2, 0x65, 0x48, 0x8B, 0x52,
    0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x48, 0x8B, 0x72,
    0x50, 0x48, 0x0F, 0xB7, 0x4A, 0x4A, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
    0xAC, 0x3C, 0x61, 0x7C, 0x02, 0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41,
    0x01, 0xC1, 0xE2, 0xED, 0x52, 0x41, 0x51, 0x48, 0x8B, 0x52, 0x20, 0x8B,
    0x42, 0x3C, 0x48, 0x01, 0xD0, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
    0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x50, 0x8B, 0x48, 0x18, 0x44,
    0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 0x41,
    0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,
    0xAC, 0x41, 0xC1, 0xC9, 0x0D, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75, 0xF1,
    0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,
    0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,
    0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x41, 0x8B, 0x04, 0x88, 0x48, 0x01,
    0xD0, 0x41, 0x58, 0x41, 0x58, 0x5E, 0x59, 0x5A, 0x41, 0x58, 0x41, 0x59,
    0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20, 0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41,
    0x59, 0x5A, 0x48, 0x8B, 0x12, 0xE9, 0x57, 0xFF, 0xFF, 0xFF, 0x5D, 0x48,
    0xBA, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8D, 0x8D,
    0x01, 0x01, 0x00, 0x00, 0x41, 0xBA, 0x31, 0x8B, 0x6F, 0x87, 0xFF, 0xD5,
    0xBB, 0xE0, 0x1D, 0x2A, 0x0A, 0x41, 0xBA, 0xA6, 0x95, 0xBD, 0x9D, 0xFF,
    0xD5, 0x48, 0x83, 0xC4, 0x28, 0x3C, 0x06, 0x7C, 0x0A, 0x80, 0xFB, 0xE0,
    0x75, 0x05, 0xBB, 0x47, 0x13, 0x72, 0x6F, 0x6A, 0x00, 0x59, 0x41, 0x89,
    0xDA, 0xFF, 0xD5, 0x6D, 0x73, 0x70, 0x61, 0x69, 0x6E, 0x74, 0x2E, 0x65,
    0x78, 0x65, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
};

unsigned int lengthOfshellcodePayload = 288;

int main(void) {

    void * alloc_mem;
    BOOL retval;
    HANDLE threadHandle;
    DWORD oldprotect = 0;

    // Allocate some memory space for shellcodePayload
    alloc_mem = VirtualAlloc(0, lengthOfshellcodePayload, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    printf("%-20s : 0x%-016p\n", "shellcodePayload addr", (void *)shellcodePayload);
    printf("%-20s : 0x%-016p\n", "alloc_mem addr", (void *)alloc_mem);

    // Copy shellcodePayload to newly allocated memory
    RtlMoveMemory(alloc_mem, shellcodePayload, lengthOfshellcodePayload);

    // Set the newly allocated memory to be executable
    retval = VirtualProtect(alloc_mem, lengthOfshellcodePayload, PAGE_EXECUTE_READ, &oldprotect);

    printf("\nPress Enter to Create Thread!\n");
    getchar();

    // If VirtualProtect succeeded, run the thread that contains the shellcodePayload
    if ( retval != 0 ) {
            threadHandle = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) alloc_mem, 0, 0, 0);
            WaitForSingleObject(threadHandle, INFINITE);
    }

    return 0;
}
```

Now just run the .bat file, then run the .exe file.



```
FLARE-VM 06-04-2024  3:21:22.12
C:\Program Files\Microsoft Visual Studio\2022\Community>cd C:\Users\Red\Desktop\project\23-Dll Re

FLARE-VM 06-04-2024  3:21:23.98
C:\Users\Red\Desktop\project\23-Dll Reversing>compile.bat
shellcode_runner.cpp

FLARE-VM 06-04-2024  3:21:28.65
C:\Users\Red\Desktop\project\23-Dll Reversing>Shellcode_Runner.exe
shellcodePayload addr : 0x00007FF61C763000
alloc_mem addr       : 0x0000029661930000

Press Enter to Create Thread!


FLARE-VM 06-04-2024  3:21:35.54
C:\Users\Red\Desktop\project\23-Dll Reversing>compile.bat
shellcode_runner.cpp

FLARE-VM 06-04-2024  3:22:29.35
C:\Users\Red\Desktop\project\23-Dll Reversing>Shellcode_Runner.exe
shellcodePayload addr : 0x00007FF672BD3000
alloc_mem addr       : 0x0000019779AF0000

Press Enter to Create Thread!


FLARE-VM 06-04-2024  3:22:35.21
C:\Users\Red\Desktop\project\23-Dll Reversing>
```

Windows_Malware (Snapshot 3) [Running] - Oracle VM VirtualBox
x64 Native Tools Command Prompt for VS 2022
Untitled - Paint
11 items 1 item selected 117 bytes
1152 × 586px
100%
Windows 10 Enterprise Evaluation
Windows License valid for 86 days
Build 19041.vb_release.191206-1406

Here we can see that it opens mspaint when we execute the .exe file.