The Lazarus Injector

norfolkinfosec.com/the-lazarus-injector/

norfolk July 22, 2019

In May and June, two files were submitted to VirusTotal that were signed with the same digital certificate and were connected to the SWIFT-heist wing of the DPRK. One file is re-themed version of the fake resume creating tool used in the Redbanc and Pakistan attacks. The second file is a tool used to inject and run payloads inside of explorer.exe.

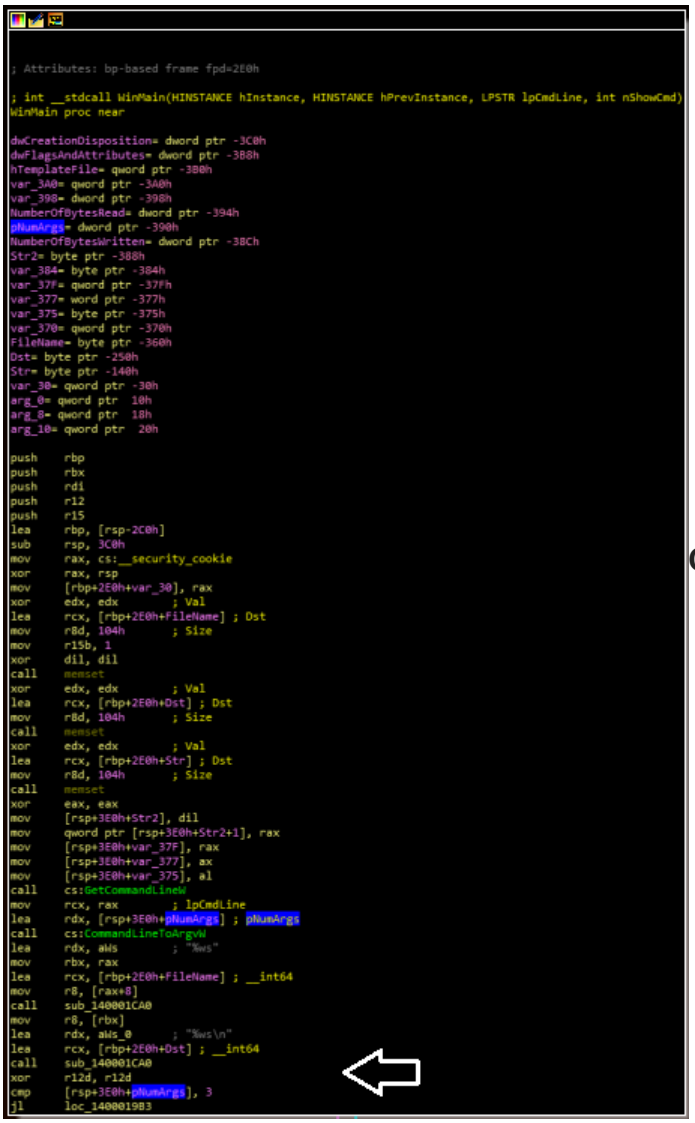
This brief post documents the capabilities of this second tool.

MD5: b9ad0cc2a2e0f513ce716cdf037da907
SHA1: 1a50a7ea5ca105df504c33af1c0329d36f03715b
SAH256: db0f102af2d350aa1a63772e6ee9b211d78aa962a34f75c8702e71ccd261243e

Parameter Check

The malware expects at least one parameter: a file path (pointing towards the injected payload) to be passed to it during execution.

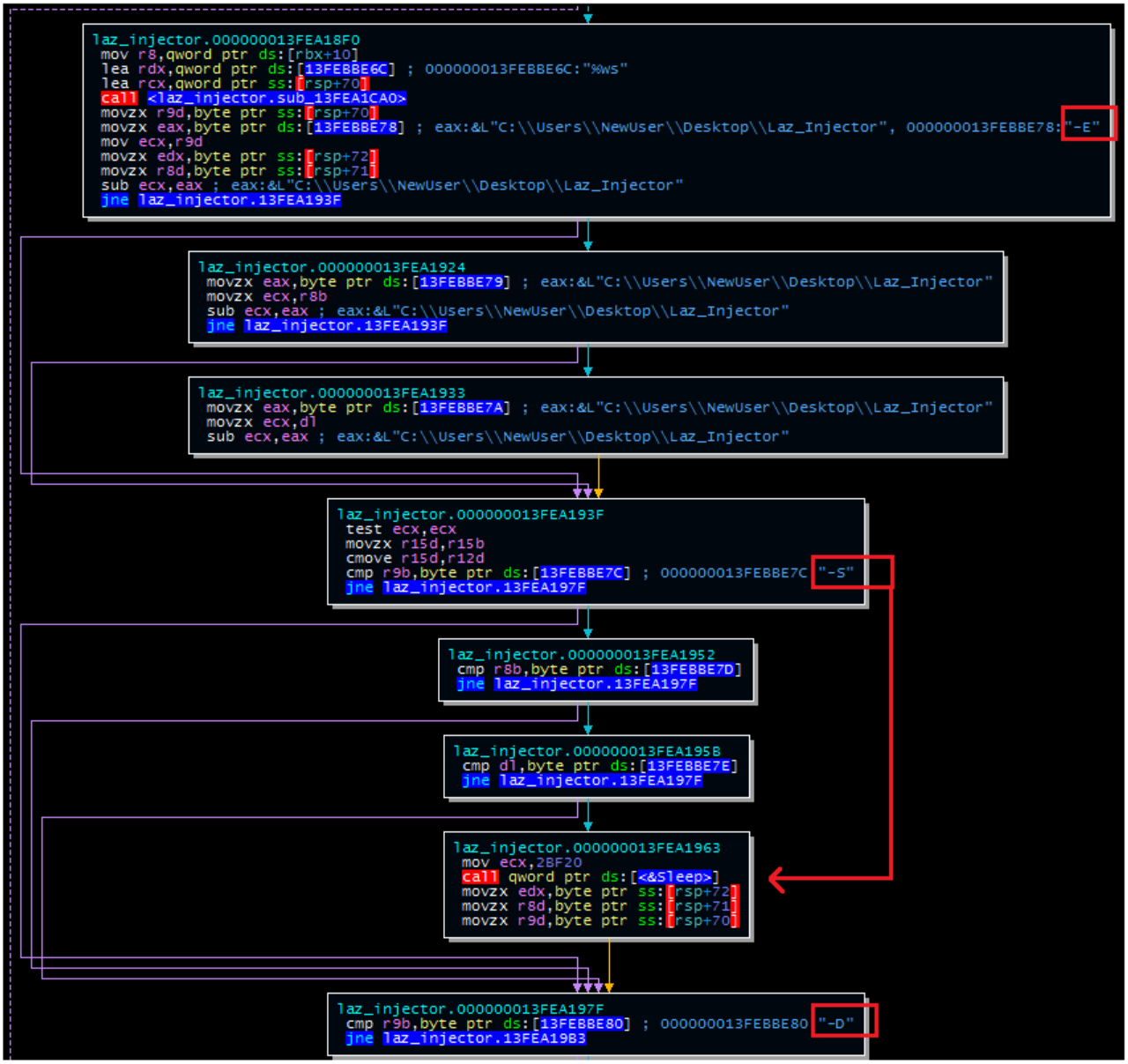
The majority of the injector's workflow takes place within two functions. In the first function, the injector checks for any arguments set during execution (coincidentally, similar to a previous post on this blog). If this number is *less than* 3, the malware will jump to a "create file check," to be discussed shortly:



```
; Attributes: bp-based frame fpd=2E0h

; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
WinMain proc near

dwCreationDisposition= dword ptr -3C0h
dwFlagsAndAttributes= dword ptr -3B8h
hTemplateFile= qword ptr -3B0h
var_3A0= qword ptr -3A0h
var_398= dword ptr -398h
NumberOfBytesRead= dword ptr -394h
pNumArgs= dword ptr -390h
NumberOfBytesWritten= dword ptr -38Ch
Str2= byte ptr -388h
var_384= byte ptr -384h
var_37F= qword ptr -37Fh
var_377= word ptr -377h
var_375= byte ptr -375h
var_370= qword ptr -370h
FileName= byte ptr -360h
Dst= byte ptr -250h
Str= byte ptr -140h
var_30= qword ptr -30h
arg_0= qword ptr  10h
arg_8= qword ptr  18h
arg_10= qword ptr  20h

push    rbp
push    rbx
push    rdi
push    r12
push    r15
lea     rbp, [rsp-2C0h]
sub     rsp, 3C0h
mov     rax, cs:__security_cookie
xor     rax, rsp
mov     [rbp+2E0h+var_30], rax
xor     edx, edx        ; Val
lea     rcx, [rbp+2E0h+FileName] ; Dst
mov     r8d, 104h       ; Size
mov     r15b, 1
xor     dil, dil
call    memset
xor     edx, edx        ; Val
lea     rcx, [rbp+2E0h+Dst] ; Dst
mov     r8d, 104h       ; Size
call    memset
xor     edx, edx        ; Val
lea     rcx, [rbp+2E0h+Str] ; Dst
mov     r8d, 104h       ; Size
call    memset
xor     eax, eax
mov     [rsp+3E0h+Str2], dil
mov     qword ptr [rsp+3E0h+Str2+1], rax
mov     [rsp+3E0h+var_37F], rax
mov     [rsp+3E0h+var_377], ax
mov     [rsp+3E0h+var_375], al
call    cs:GetCommandLineW
mov     rcx, rax        ; lpCmdLine
lea     rdx, [rsp+3E0h+pNumArgs] ; pNumArgs
call    cs:CommandLineToArgvW
lea     rdx, aWs        ; "%ws"
mov     rbx, rax
lea     rcx, [rbp+2E0h+FileName] ; __int64
mov     r8, [rax+8]
call    sub_140001CA0
mov     r8, [rbx]
lea     rdx, aWs_0      ; "%ws\n"
lea     rcx, [rbp+2E0h+Dst] ; __int64
call    sub_140001CA0
xor     r12d, r12d
cmp     [rsp+3E0h+pNumArgs], 3
jl      loc_1400019B3
```

Checking for the number of passed paramters

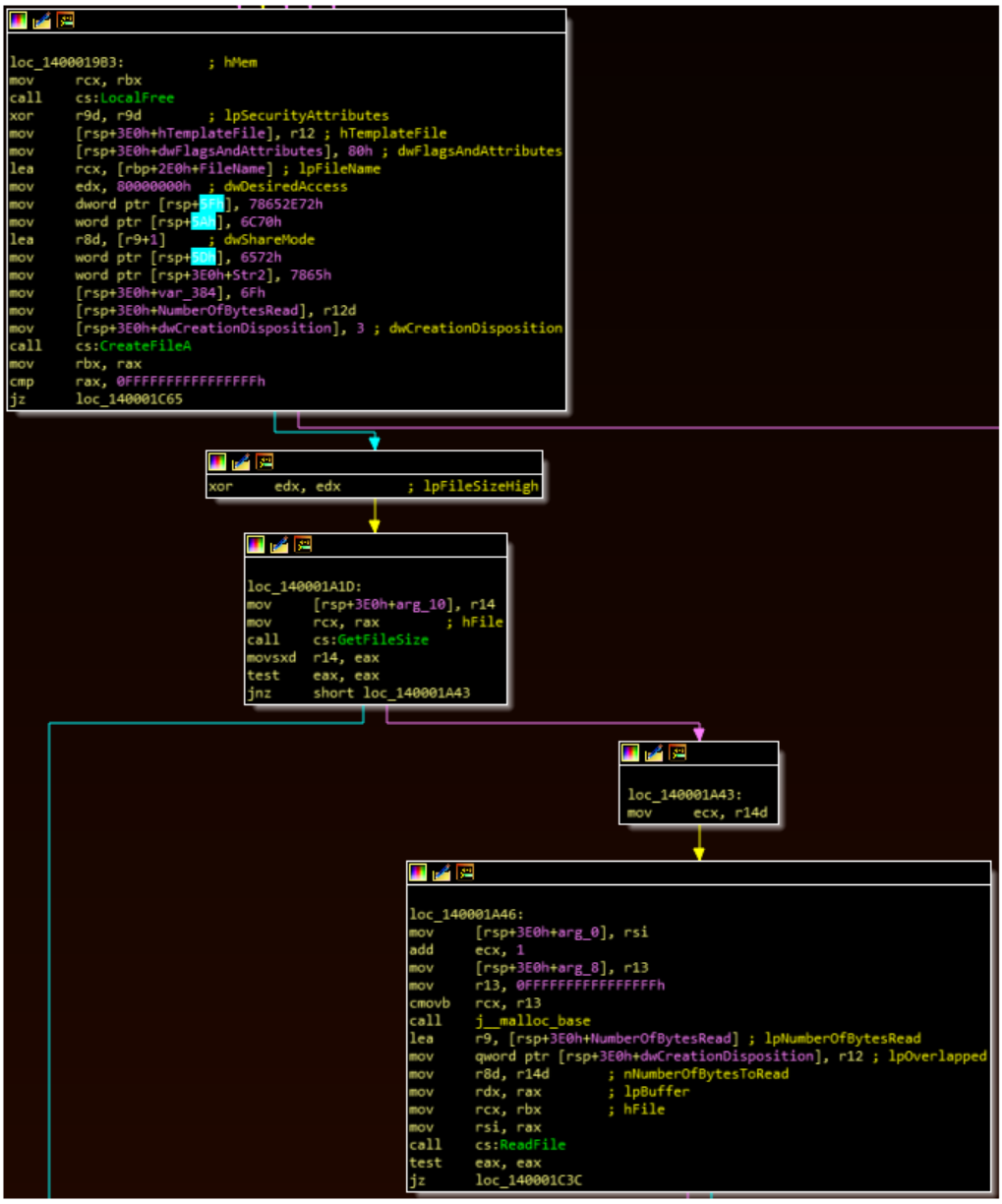
If, however, this number is *great than or equal to* 3, the malware will begin checking for execution parameters. These can be seen in clear-text during debugging. Accepted parameters include -S, -E, and -D. Of these, only -S has an immediately discernible purpose: it causes the malware to sleep. These parameters (and the sleep function) are shown below:



Argument checks and sleep function

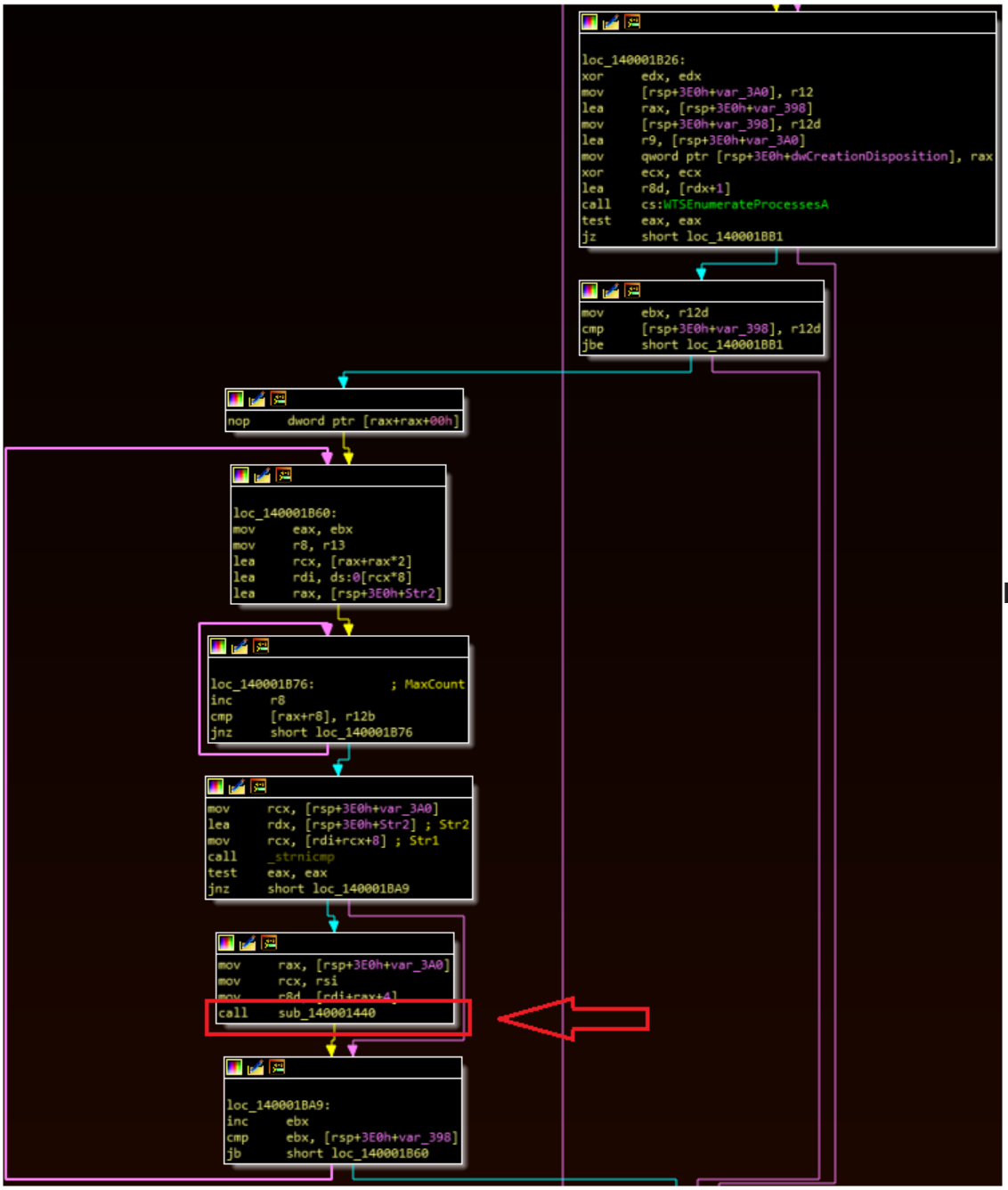
Payload Checks

After checking for these parameters, the malware performs an additional check: it uses the passed filepath and attempts to open a handle at this location with CreateFile. If this is unsuccessful, the malware will exit this workflow and terminate. In addition, the malware makes two additional checks: one to GetFileSize and one to ReadFile. Each is followed by a “test EAX EAX” instruction. In practical terms, this ensures that the file in question has a size greater than zero (i.e. isn’t empty) and can be read by the malware.



File access and filesize checks

Next, the malware calls WTSEnumerateProcessesA to list the running processes. It cycles through these until it identifies the process for Explorer.exe, and which point it enters the subroutine boxed below:



Process Enumeration

Injection Routine

This routine is the parent function for decoding and dynamically resolving several API calls related to process resolution.

```
64bit_thing2.000000014000148A
 mov qword ptr ss:[rsp+C8],rbx
 mov ecx,C ; C:'\f'
 mov qword ptr ss:[rsp+A0],rsi
 mov qword ptr ss:[rsp+98],rdi
 mov qword ptr ss:[rsp+90],r13
 mov qword ptr ss:[rsp+88],r14
 movsxd r14,dword ptr ds:[r12+3C]
 add r14,r12
 mov dword ptr ss:[rbp+7],83460960
 mov dword ptr ss:[rbp+B],7848E341
 mov dword ptr ss:[rbp+F],17642F66
 mov dword ptr ss:[rbp+13],FDFDFDFD
 mov byte ptr ss:[rbp+17],5C ; 5C:'\\'
 call <64bit_thing2.sub_140003BFC>
 mov r8,rax
 lea rdx,qword ptr ss:[rbp+7]
 mov rsi,rax
 call <64bit_thing2.Decoder> ; Decodes kernel32.dll
 mov ecx,C ; C:'\f'
 mov byte ptr ds:[rsi+C],0
 mov dword ptr ss:[rbp+7],9946055D
 mov dword ptr ss:[rbp+B],B17EE51
 mov dword ptr ss:[rbp+F],18672724
 mov dword ptr ss:[rbp+13],FDFD2553
 mov word ptr ss:[rbp+17],FDFD
 call <64bit_thing2.sub_140003BFC>
 mov r8,rax
 lea rdx,qword ptr ss:[rbp+7]
 mov rdi,rax
 call <64bit_thing2.Decoder> ; Decodes VirtualAllocEx
 mov ecx,C ; C:'\f'
 mov byte ptr ds:[rdi+E],0
 mov dword ptr ss:[rbp+7],995D1E5C
 mov dword ptr ss:[rbp+B],2509DF41
 mov dword ptr ss:[rbp+F],87B2E2B
 mov dword ptr ss:[rbp+13],927F385B
 mov dword ptr ss:[rbp+17],FDFD0DFD
 mov word ptr ss:[rbp+1B],FDFD
 call <64bit_thing2.sub_140003BFC>
 mov r8,rax
 lea rdx,qword ptr ss:[rbp+7]
 mov rbx,rax
 call <64bit_thing2.Decoder> ; Decodes WriteProcessMemory
 mov rcx,rsi
 mov byte ptr ds:[rbx+12],0
 call qword ptr ds:[<&GetModuleHandleA>]
 mov rcx,rax
 mov rdx,rdi
 call qword ptr ds:[<&GetProcAddress>]
 mov rcx,rsi
 mov rdi,rax
 call qword ptr ds:[<&GetModuleHandleA>]
 mov rcx,rax
 mov rdx,rbx
 call qword ptr ds:[<&GetProcAddress>]
 mov r13,rax
 test rdi,rdi
 je 64bit_thing2.1400016FC
```

Moving values that are then passed through a decoding and API resolution routine.

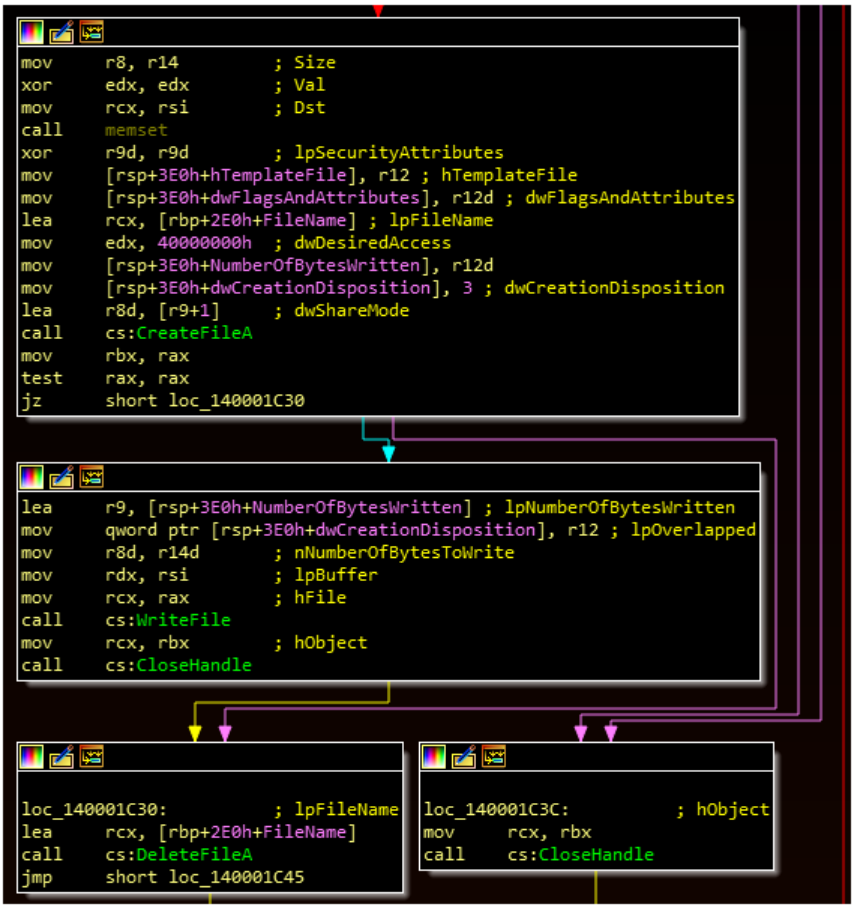
The file then allocates a section of memory to Explorer and writes the payload to this memory section (using the resolved APIs). It resolves the NTCreateThreadEx API and then creates and executes a thread at this location:

CreateThread resolution and execution

Cleaning Up

At this point, the malware returns to the original loop that was used to identify Explorer.exe as a running process. Curiously, the malware actually *continues* to run in this loop rather than breaking the loop once it is found.

Once this loop completes, the malware will exit this function and the loader will terminate. If the -D or -S parameters were specified, the malware will overwrite the original contents of the loaded payload and then delete this file from disk. If -E is specified, the malware will actually skip this step.



Code for deleting the payload from disk

At this stage, the payload is expected to be run in memory and the “loading” is complete.