Hancitor's packer demystified

uperesia.com/hancitor-packer-demystified

HANCITOR'S PACKER DEMYSTIFIED

Posted by Felix Weyne, May 2019.
Author contact: Twitter | LinkedIn
Tags: Hancitor, Chanitor, packer, unpacking, spaghetti code, shellcode, control flow obfuscation, import table reconstruction, reflective PE loading, YARA

It has been a while since I have written a blog - I have been working on some tools and other projects instead - so I decided to have another go at it 😊. A while ago, the Twitter users 0verfl0w_ and Vitali published some nice blogs on the Hancitor malware. This made me curious to also have a look at the malware family.

The Hancitor malware family has been around for a while and its core job is to download and execute additional malware. In order to succeed at its job, the malware must succeed in being run undetected on the machine and thus effectively stay under the radar of security software such as an antivirus. One of Hancitor's endeavors to bypass antivirus is by making use of a booby trapped Office document and to instruct Office to inject the Hancitor binary in a legitimate Windows process. This method has been documented well by the Airbus security team and has been used untill approximately the summer of 2018. Around that time, the Hancitor crew has shifted its infection mechanism by making their spammed Office documents download a packed executable to disk. An executable written to disk usually gets inspected/scanned by antivirus, yet the Hancitor malware has been reasonably successful in evading being detected (initially) as malicious.

Hancitor's evasive success can be partly attributed to the packer/crypter being used. **In this blog I will do a (technical) deep dive into Hancitor's packer, which has not changed much since the summer of 2018. I will discuss how the packer protects its payload and how it tries to thwart analysis. At the end of this blog, I'll demonstrate how this packer has also been used by many other malware families in the past.**

The packer

The below image gives an overview of the sample which I'll discuss in this blog. Although I will be discussing a specific packed Hancitor sample, the information in this blog is applicable to many other packed Hancitor samples, as the packer has not changed much between the many SPAM campaigns (particularly the first layer of the packer has been very consistent). In this archive (password=infected) a collection of many packed Hancitor samples can be found (many thanks to Brad and James for sharing the samples on Twitter!).

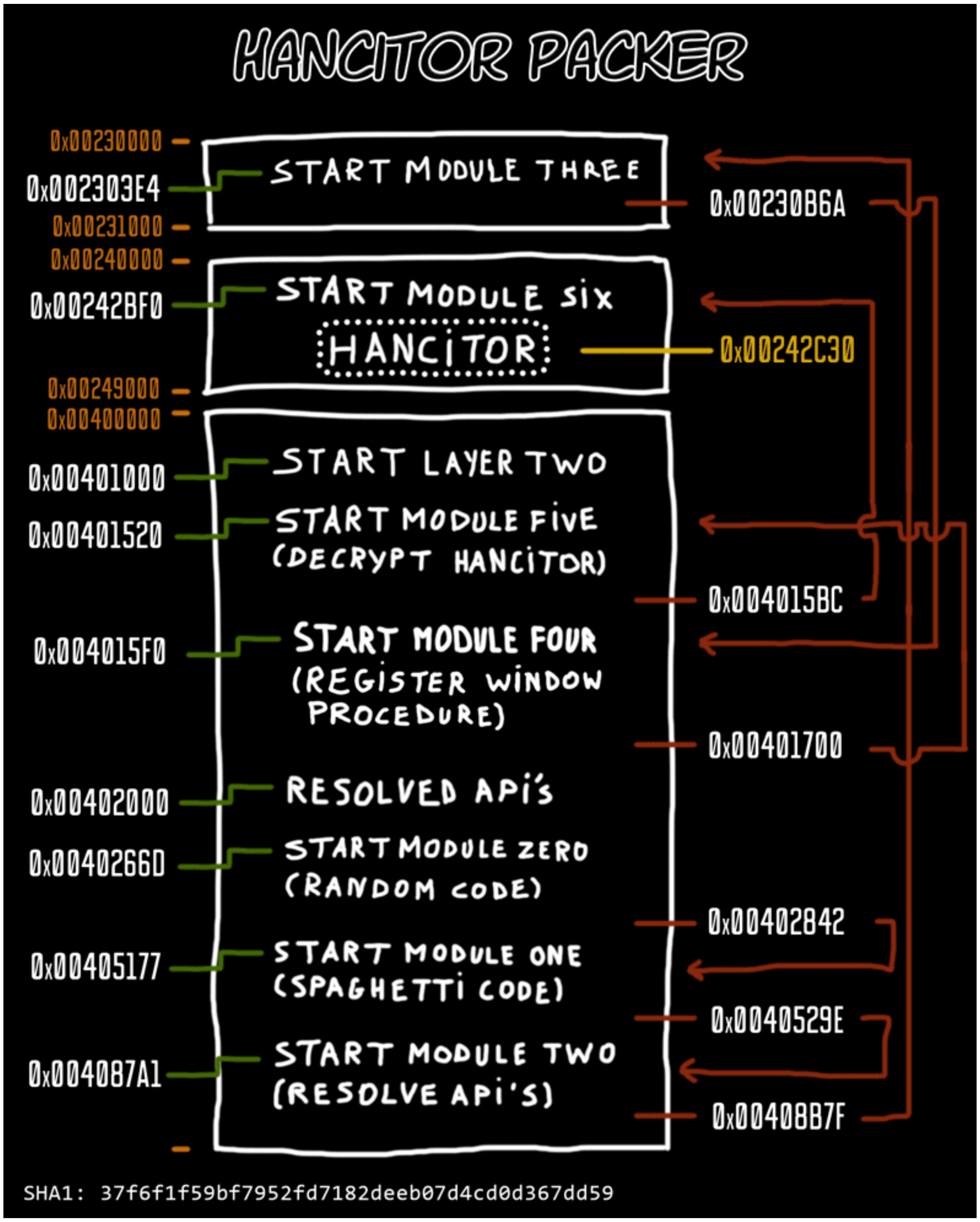


Image one: Overview of the packed Hancitor sample

In order to keep the analysis organized, I have a divided the packed sample into "modules" (pieces) based on functionality. For each module I have added the address of the first and last relevant assembly instruction, such that interested readers can use this blog as a reference when unpacking the sample themselves in a debugger. For those who are interested in the disassembled code, but don't want to plow through the entire sample in a debugger, I have added a commented assembly output per module. Lastly, for the malware hunters among us, I have added a YARA rule for the packer in the blog's addendum.

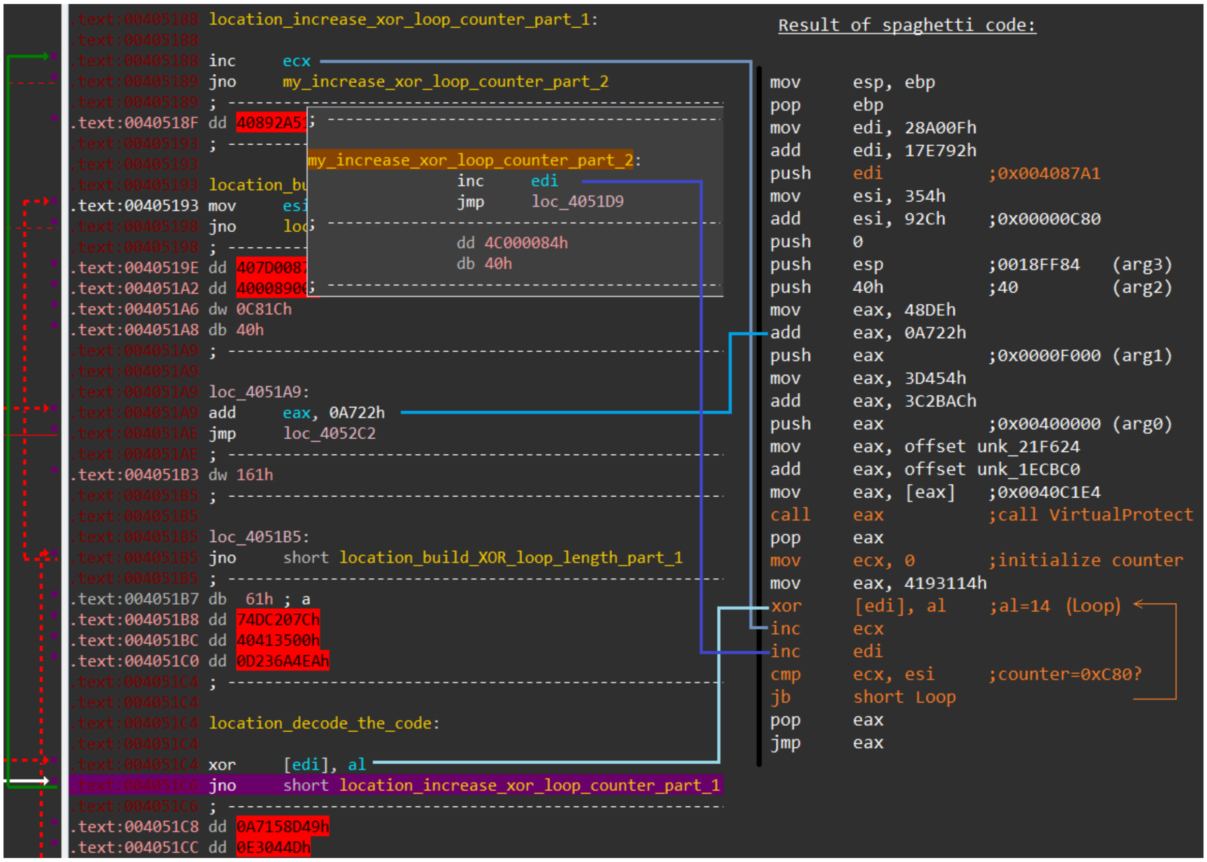
- Module 0: link to commented disassembled code (start address: 0x0040266D)
- Module 1: link to commented disassembled code (start address: 0X00405177)
- Module 2: link to commented disassembled code (start address: 0X004087A1)

- Module 3: link to commented disassembled code (start address: start_mem_region+0x3E4)
- Module 4: link to commented disassembled code (start address: 0X004015F0)
- Module 5: link to commented disassembled code (start address: 0X00401520)
- Module 6: link to commented disassembled code (start address: start_mem_region+0x2BF0)

Spaghetti code

The packed Hancitor executables always start by executing random, non-dodgy functions. We will define this code region as module zero (disassembled code). Putting random code near the executables' entrypoint makes them look unique, that is to say, for security products which (understandably) only parse/emulate executables partially because of performance reasons. The random code ends by jumping to the next module, module one (disassembled code).

The disassembled output of the module one section is hard to interpret. **The packer's author has broken the linear sequence of assembly instructions by reordering the instructions and connecting them to each other via JUMP instructions, as can be seen in image two. Additionally, between each instruction random instructions - which will never be executed - are placed.**



```
.text:00405188 location_increase_xor_loop_counter_part_1:
.text:00405188
.text:00405188 inc     ecx
.text:00405189 jno     my_increase_xor_loop_counter_part_2
.text:00405189 ; ---------------------------------------------------------------
.text:0040518F dd 40892A51
.text:00405193 ; --------
.text:00405193
.text:00405193 location_bu
.text:00405193 mov     esi
.text:00405198 jno     loc
.text:00405198 ; --------
.text:0040519E dd 407D0087
.text:004051A2 dd 40008900
.text:004051A6 dw 0C81Ch
.text:004051A8 db 40h
.text:004051A9 ; ---------------------------------------------------------------
.text:004051A9
.text:004051A9 loc_4051A9:
.text:004051A9 add     eax, 0A722h
.text:004051AE jmp     loc_4052C2
.text:004051AE ; ---------------------------------------------------------------
.text:004051B3 dw 161h
.text:004051B5 ; ---------------------------------------------------------------
.text:004051B5
.text:004051B5 loc_4051B5:
.text:004051B5 jno     short location_build_XOR_loop_length_part_1
.text:004051B5 ; ---------------------------------------------------------------
.text:004051B7 db  61h ; a
.text:004051B8 dd 74DC207Ch
.text:004051BC dd 40413500h
.text:004051C0 dd 0D236A4EAh
.text:004051C4 ; ---------------------------------------------------------------
.text:004051C4
.text:004051C4 location_decode_the_code:
.text:004051C4
.text:004051C4 xor     [edi], al
.text:004051C6 jno     short location_increase_xor_loop_counter_part_1
.text:004051C6 ; ---------------------------------------------------------------
.text:004051C8 dd 0A7158D49h
.text:004051CC dd 0E3044Dh
```

```
; ---------------------------------------------------------------

my_increase_xor_loop_counter_part_2:
                inc     edi
                jmp     loc_4051D9
; ---------------------------------------------------------------

                dd 4C000084h
                db 40h
; ---------------------------------------------------------------
```

Result of spaghetti code:

```
mov     esp, ebp
pop     ebp
mov     edi, 28A00Fh
add     edi, 17E792h
push    edi             ;0x004087A1
mov     esi, 354h
add     esi, 92Ch       ;0x00000C80
push    0
push    esp             ;0018FF84   (arg3)
push    40h             ;40         (arg2)
mov     eax, 48DEh
add     eax, 0A722h
push    eax             ;0x0000F000 (arg1)
mov     eax, 3D454h
add     eax, 3C2BACh
push    eax             ;0x00400000 (arg0)
mov     eax, offset unk_21F624
add     eax, offset unk_1ECBC0
mov     eax, [eax]      ;0x0040C1E4
call    eax             ;call VirtualProtect
pop     eax
mov     ecx, 0          ;initialize counter
mov     eax, 4193114h
xor     [edi], al       ;al=14  (Loop) <-
inc     ecx
inc     edi
cmp     ecx, esi        ;counter=0xC80?
jb      short Loop
pop     eax
jmp     eax
```

Image two: Spaghetti code which decrypts the next module

This technique, known as spaghetti code, bypasses static detection techniques which rely on the malicious instructions being placed consecutively on each other. The goal of the spaghetti code is to change the memory protection of a part of the executable (to which we will referrer as module two) and then to decrypt said part via a simple XOR loop. Once the relevant part is decrypted, the code execution is transferred to that part via a simple JMP EAX instruction.

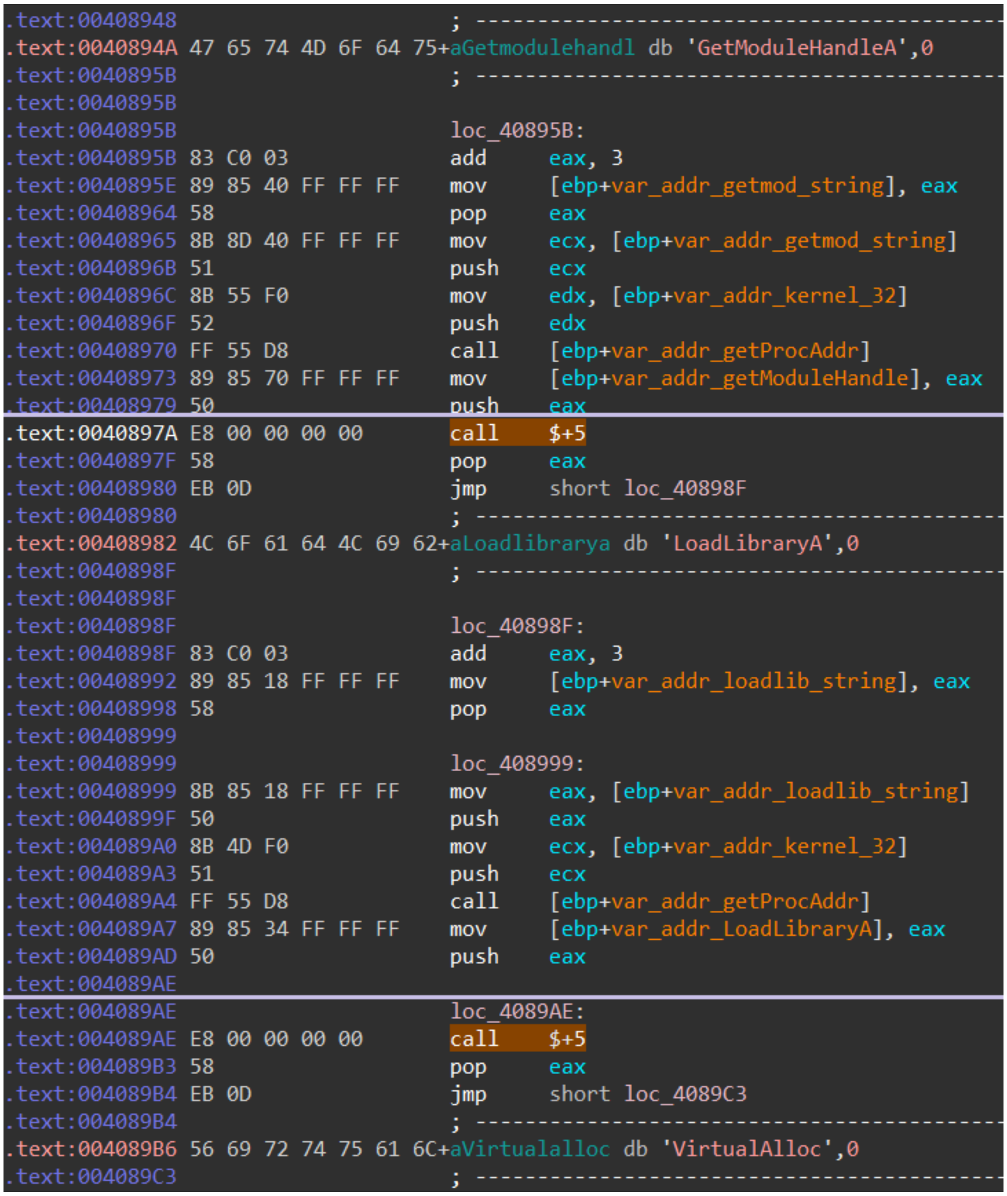
Resolving APIs

Module two (disassembled code) has three tasks: resolve the addresses of APIs which will be used in the next module, map itself and the next module in a newly allocated memory region and hunt for the start of the next module in the new memory region (delimited by the 70C5BA88 byte marker).

I will not discuss how the API addresses are resolved, as the packer will use a similar technique in a later module, at which point I'll discuss the technique in depth (see paragraph: reconstruct import table). The most important part of the API resolving code is the list of APIs which are resolved:

- kernel32_GetProcAddress
- kernel32_GetModuleHandleA
- kernel32_LoadLibraryA
- kernel32_VirtualAlloc
- kernel32_VirtualFree
- kernel32_OutputDebugStringA
- ntdll_memset
- ntdll_memcpy

The APIs in the list will be used to map DLLs into the packer's process memory, to resolve additional API addresses and to allocate and free memory regions. **The thing in module two that stands out the most is the way (API) strings are embedded inline with the assembly code, as can be seen on image three.**



```
.text:00408948                     ; -------------------------------------------
.text:0040894A 47 65 74 4D 6F 64 75+aGetmodulehandl db 'GetModuleHandleA',0
.text:0040895B                     ; -------------------------------------------
.text:0040895B
.text:0040895B                     loc_40895B:
.text:0040895B 83 C0 03            add     eax, 3
.text:0040895E 89 85 40 FF FF FF   mov     [ebp+var_addr_getmod_string], eax
.text:00408964 58                  pop     eax
.text:00408965 8B 8D 40 FF FF FF   mov     ecx, [ebp+var_addr_getmod_string]
.text:0040896B 51                  push    ecx
.text:0040896C 8B 55 F0            mov     edx, [ebp+var_addr_kernel_32]
.text:0040896F 52                  push    edx
.text:00408970 FF 55 D8            call    [ebp+var_addr_getProcAddr]
.text:00408973 89 85 70 FF FF FF   mov     [ebp+var_addr_getModuleHandle], eax
.text:00408979 50                  push    eax
.text:0040897A E8 00 00 00 00      call    $+5
.text:0040897F 58                  pop     eax
.text:00408980 EB 0D               jmp     short loc_40898F
.text:00408980                     ; -------------------------------------------
.text:00408982 4C 6F 61 64 4C 69 62+aLoadlibrarya db 'LoadLibraryA',0
.text:0040898F                     ; -------------------------------------------
.text:0040898F
.text:0040898F                     loc_40898F:
.text:0040898F 83 C0 03            add     eax, 3
.text:00408992 89 85 18 FF FF FF   mov     [ebp+var_addr_loadlib_string], eax
.text:00408998 58                  pop     eax
.text:00408999
.text:00408999                     loc_408999:
.text:00408999 8B 85 18 FF FF FF   mov     eax, [ebp+var_addr_loadlib_string]
.text:0040899F 50                  push    eax
.text:004089A0 8B 4D F0            mov     ecx, [ebp+var_addr_kernel_32]
.text:004089A3 51                  push    ecx
.text:004089A4 FF 55 D8            call    [ebp+var_addr_getProcAddr]
.text:004089A7 89 85 34 FF FF FF   mov     [ebp+var_addr_LoadLibraryA], eax
.text:004089AD 50                  push    eax
.text:004089AE
.text:004089AE                     loc_4089AE:
.text:004089AE E8 00 00 00 00      call    $+5
.text:004089B3 58                  pop     eax
.text:004089B4 EB 0D               jmp     short loc_4089C3
.text:004089B4                     ; -------------------------------------------
.text:004089B6 56 69 72 74 75 61 6C+aVirtualalloc db 'VirtualAlloc',0
.text:004089C3                     ; -------------------------------------------
```

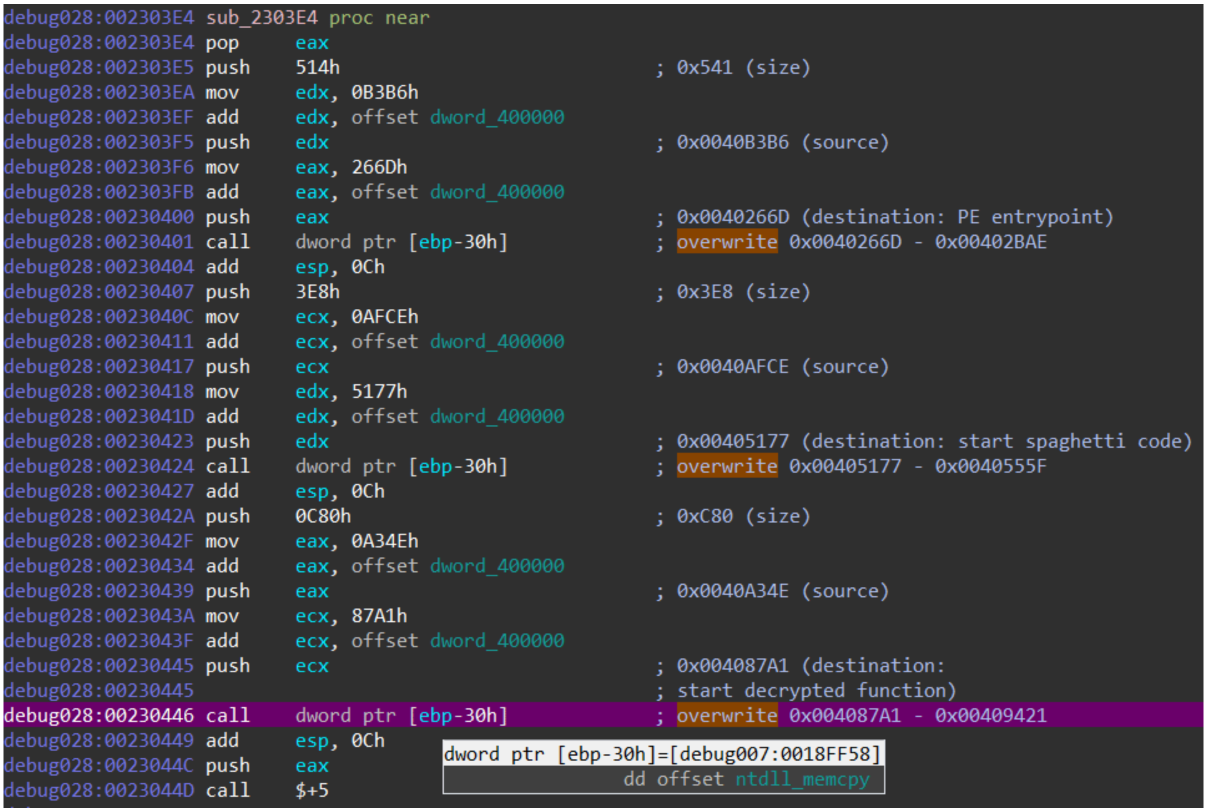
Image three: Data (API names) inline with the assembly code

Most compilers will place strings in a region which is different from the region where the assembly code resides. **To get the memory address of the inline string, the assembly code makes use of a simple trick: it will execute a CALL $+5 instruction (a procedure call where the destination is the subsequent instruction).**

Executing a CALL instruction will result in the return address (i.e. the address of the instruction that follows the call instruction) being pushed on the stack. The return address is immediately retrieved by executing a POP EAX instruction (pop the top of the stack into the EAX register). The return address is thus pointing to the location of the POP instruction. Because the assembly is interested in the start address of the inline placed string, three bytes needs to be added to return address (skip the POP and JMP short instructions). We can see the assembly code performing this action as follows: ADD EAX, 3. It is useful to remember this little trick in your short-term memory, because it will also be used in the next module.

Decrypt next layer

Module three (disassembled code) starts by overwriting code at three locations, as can be seen on image four. These locations correspond with the packed executable's entrypoint (module zero), the start of the spaghetti code (module one) and the start of module two (the addresses are described on image one).

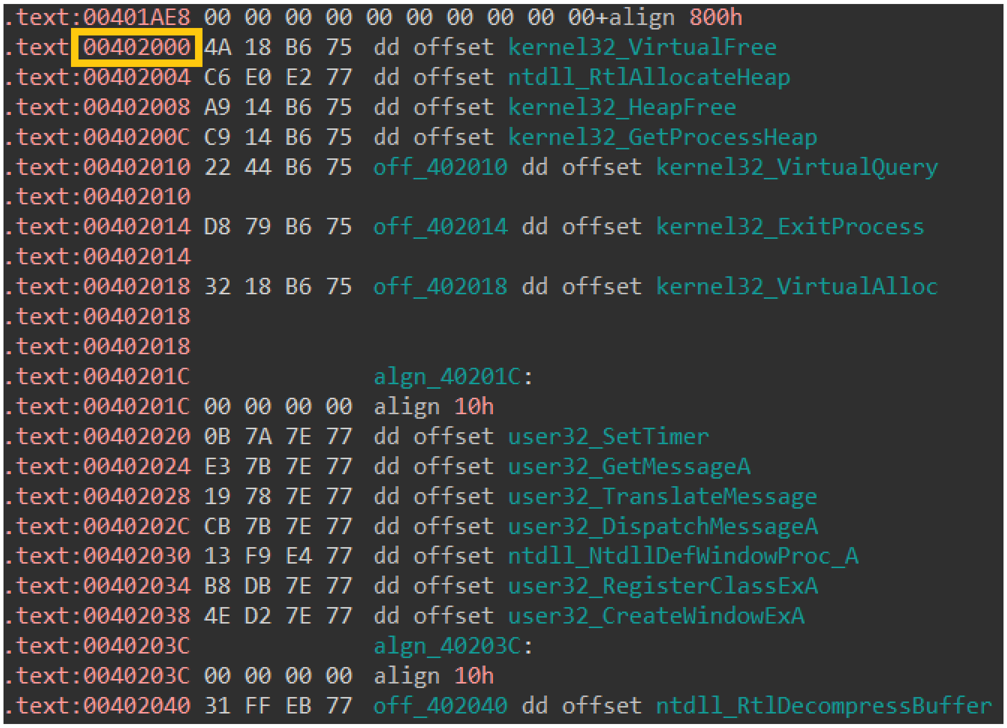


```
debug028:002303E4 sub_2303E4 proc near
debug028:002303E4 pop     eax
debug028:002303E5 push    514h                          ; 0x541 (size)
debug028:002303EA mov     edx, 0B3B6h
debug028:002303EF add     edx, offset dword_400000
debug028:002303F5 push    edx                           ; 0x0040B3B6 (source)
debug028:002303F6 mov     eax, 266Dh
debug028:002303FB add     eax, offset dword_400000
debug028:00230400 push    eax                           ; 0x0040266D (destination: PE entrypoint)
debug028:00230401 call    dword ptr [ebp-30h]           ; overwrite 0x0040266D - 0x00402BAE
debug028:00230404 add     esp, 0Ch
debug028:00230407 push    3E8h                          ; 0x3E8 (size)
debug028:0023040C mov     ecx, 0AFCEh
debug028:00230411 add     ecx, offset dword_400000
debug028:00230417 push    ecx                           ; 0x0040AFCE (source)
debug028:00230418 mov     edx, 5177h
debug028:0023041D add     edx, offset dword_400000
debug028:00230423 push    edx                           ; 0x00405177 (destination: start spaghetti code)
debug028:00230424 call    dword ptr [ebp-30h]           ; overwrite 0x00405177 - 0x0040555F
debug028:00230427 add     esp, 0Ch
debug028:0023042A push    0C80h                         ; 0xC80 (size)
debug028:0023042F mov     eax, 0A34Eh
debug028:00230434 add     eax, offset dword_400000
debug028:00230439 push    eax                           ; 0x0040A34E (source)
debug028:0023043A mov     ecx, 87A1h
debug028:0023043F add     ecx, offset dword_400000
debug028:00230445 push    ecx                           ; 0x004087A1 (destination:
debug028:00230445                                       ; start decrypted function)
debug028:00230446 call    dword ptr [ebp-30h]           ; overwrite 0x004087A1 - 0x00409421
debug028:00230449 add     esp, 0Ch
debug028:0023044C push    eax
debug028:0023044D call    $+5
```

```
dword ptr [ebp-30h]=[debug007:0018FF58]
                 dd offset ntdll_memcpy
```

Image four: Overwriting three previous modules

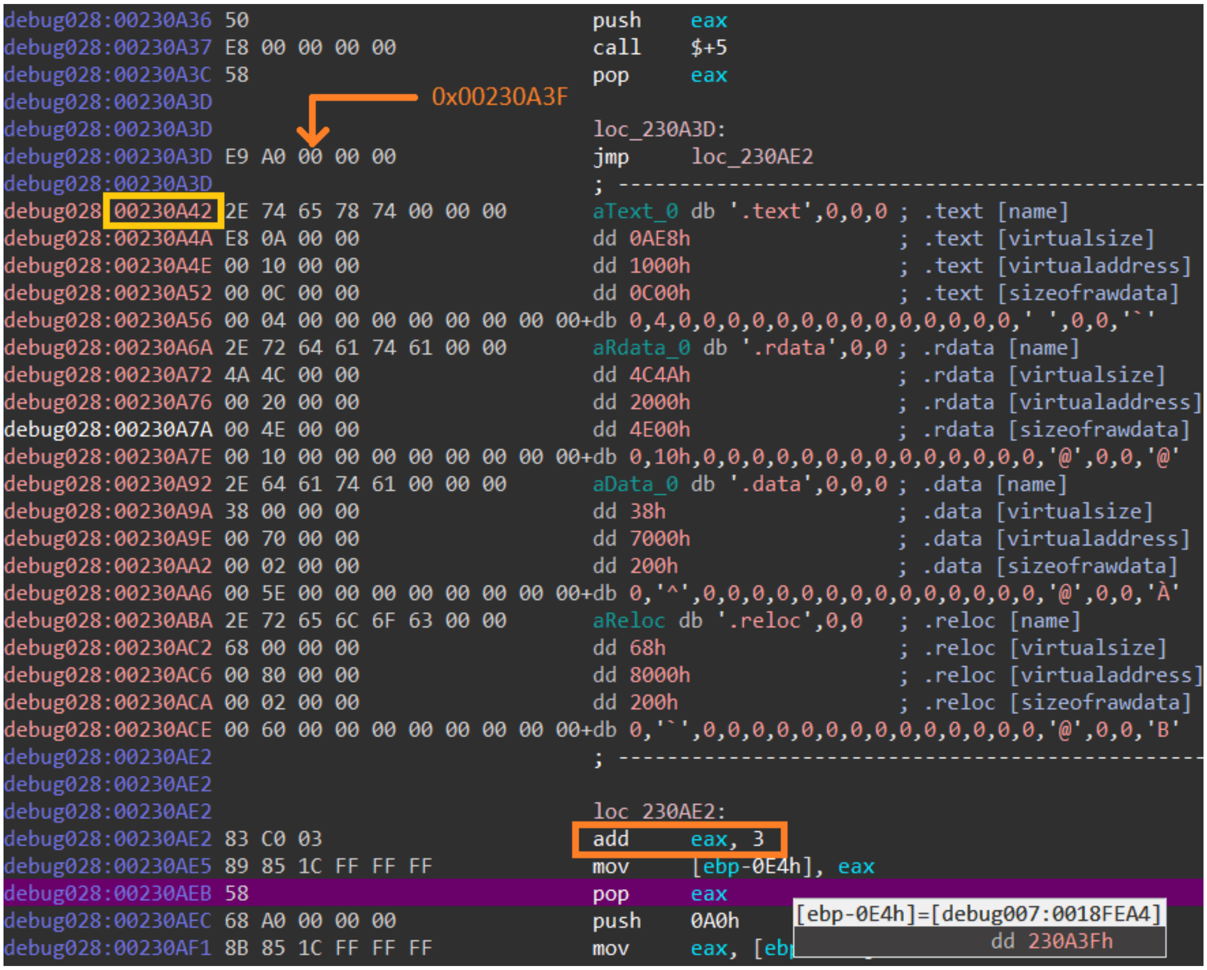
The code then continues by decrypting the next layer (the next modules), by making use of the APIs listed in the previous paragraph. Once the next layer has been decrypted, the module resolves the addresses of the APIs which will be used in the next layer (image five), to which we will refer as layer two.



```
.text:00401AE8 00 00 00 00 00 00 00 00 00 00+align 800h
.text 00402000 4A 18 B6 75  dd offset kernel32_VirtualFree
.text:00402004 C6 E0 E2 77  dd offset ntdll_RtlAllocateHeap
.text:00402008 A9 14 B6 75  dd offset kernel32_HeapFree
.text:0040200C C9 14 B6 75  dd offset kernel32_GetProcessHeap
.text:00402010 22 44 B6 75  off_402010 dd offset kernel32_VirtualQuery
.text:00402010
.text:00402014 D8 79 B6 75  off_402014 dd offset kernel32_ExitProcess
.text:00402014
.text:00402018 32 18 B6 75  off_402018 dd offset kernel32_VirtualAlloc
.text:00402018
.text:00402018
.text:0040201C              algn_40201C:
.text:0040201C 00 00 00 00  align 10h
.text:00402020 0B 7A 7E 77  dd offset user32_SetTimer
.text:00402024 E3 7B 7E 77  dd offset user32_GetMessageA
.text:00402028 19 78 7E 77  dd offset user32_TranslateMessage
.text:0040202C CB 7B 7E 77  dd offset user32_DispatchMessageA
.text:00402030 13 F9 E4 77  dd offset ntdll_NtdllDefWindowProc_A
.text:00402034 B8 DB 7E 77  dd offset user32_RegisterClassExA
.text:00402038 4E D2 7E 77  dd offset user32_CreateWindowExA
.text:0040203C              algn_40203C:
.text:0040203C 00 00 00 00  align 10h
.text:00402040 31 FF EB 77  off_402040 dd offset ntdll_RtlDecompressBuffer
```

Image five: addresses of resolved APIs in memory

After having resolved the API addresses, the code does something somewhat odd: it patches values in the PE header and it overwrites the section header. This action doesn't make much sense to me, because I believe these values are of no use once the executable has been mapped into memory 🤔? Nevertheless, this action helps us in our efforts to dump the second layer executable from memory, as it seems like we have the correct PE header as well as the decrypted code.



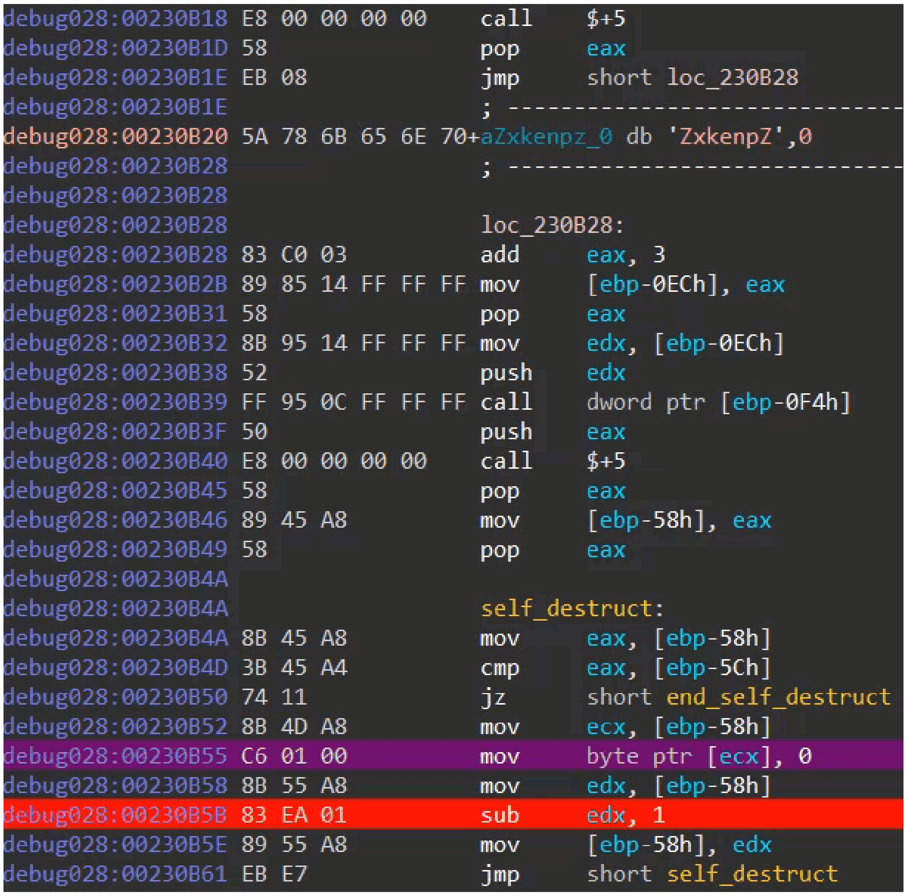
```
debug028:00230A36 50                               push    eax
debug028:00230A37 E8 00 00 00 00                   call    $+5
debug028:00230A3C 58                               pop     eax
debug028:00230A3D
debug028:00230A3D                                  loc_230A3D:
debug028:00230A3D E9 A0 00 00 00                   jmp     loc_230AE2
debug028:00230A3D                                  ; ----------------------------------------------
debug028 00230A42 2E 74 65 78 74 00 00 00          aText_0 db '.text',0,0,0 ; .text [name]
debug028:00230A4A E8 0A 00 00                      dd 0AE8h                 ; .text [virtualsize]
debug028:00230A4E 00 10 00 00                      dd 1000h                 ; .text [virtualaddress]
debug028:00230A52 00 0C 00 00                      dd 0C00h                 ; .text [sizeofrawdata]
debug028:00230A56 00 04 00 00 00 00 00 00 00 00+db 0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,' ',0,0,'`'
debug028:00230A6A 2E 72 64 61 74 61 00 00          aRdata_0 db '.rdata',0,0 ; .rdata [name]
debug028:00230A72 4A 4C 00 00                      dd 4C4Ah                 ; .rdata [virtualsize]
debug028:00230A76 00 20 00 00                      dd 2000h                 ; .rdata [virtualaddress]
debug028:00230A7A 00 4E 00 00                      dd 4E00h                 ; .rdata [sizeofrawdata]
debug028:00230A7E 00 10 00 00 00 00 00 00 00 00+db 0,10h,0,0,0,0,0,0,0,0,0,0,0,0,0,0,'@',0,0,'@'
debug028:00230A92 2E 64 61 74 61 00 00 00          aData_0 db '.data',0,0,0 ; .data [name]
debug028:00230A9A 38 00 00 00                      dd 38h                   ; .data [virtualsize]
debug028:00230A9E 00 70 00 00                      dd 7000h                 ; .data [virtualaddress]
debug028:00230AA2 00 02 00 00                      dd 200h                  ; .data [sizeofrawdata]
debug028:00230AA6 00 5E 00 00 00 00 00 00 00 00+db 0,'^',0,0,0,0,0,0,0,0,0,0,0,0,0,0,'@',0,0,'À'
debug028:00230ABA 2E 72 65 6C 6F 63 00 00          aReloc db '.reloc',0,0   ; .reloc [name]
debug028:00230AC2 68 00 00 00                      dd 68h                   ; .reloc [virtualsize]
debug028:00230AC6 00 80 00 00                      dd 8000h                 ; .reloc [virtualaddress]
debug028:00230ACA 00 02 00 00                      dd 200h                  ; .reloc [sizeofrawdata]
debug028:00230ACE 00 60 00 00 00 00 00 00 00 00+db 0,'`',0,0,0,0,0,0,0,0,0,0,0,0,0,0,'@',0,0,'B'
debug028:00230AE2                                  ; ----------------------------------------------
debug028:00230AE2
debug028:00230AE2                                  loc_230AE2:
debug028:00230AE2 83 C0 03                         add     eax, 3
debug028:00230AE5 89 85 1C FF FF FF                mov     [ebp-0E4h], eax
debug028:00230AEB 58                               pop     eax
debug028:00230AEC 68 A0 00 00 00                   push    0A0h     [ebp-0E4h]=[debug007:0018FEA4]
debug028:00230AF1 8B 85 1C FF FF FF                mov     eax, [eb                 dd 230A3Fh
```

Image six: overwriting section headers bug

Upon inspecting the dumped second layer executable, I noticed that the section headers were shifted. **When we look at the code responsible for overwriting the section headers, we can notice an interesting bug in the packer.** Remember the inline data trick I discussed in the previous paragraph? It looks like the packer's author made a small mistake while using it to overwrite the section header 😅.

Because the JMP instruction following the POP EAX instruction in module three consists of five bytes (it consisted of only three bytes in module two), the start address of the section header data is off by three bytes (image six). Instead of adding three bytes to the EAX register, the code should've added six bytes. If we correct this mistake while debugging, we get a correct dump of layer two (which I have added here).

Module three ends by destroying its own code. The destruction is performed via a simple loop which overwrites every address in the module with zero valued bytes (image seven).



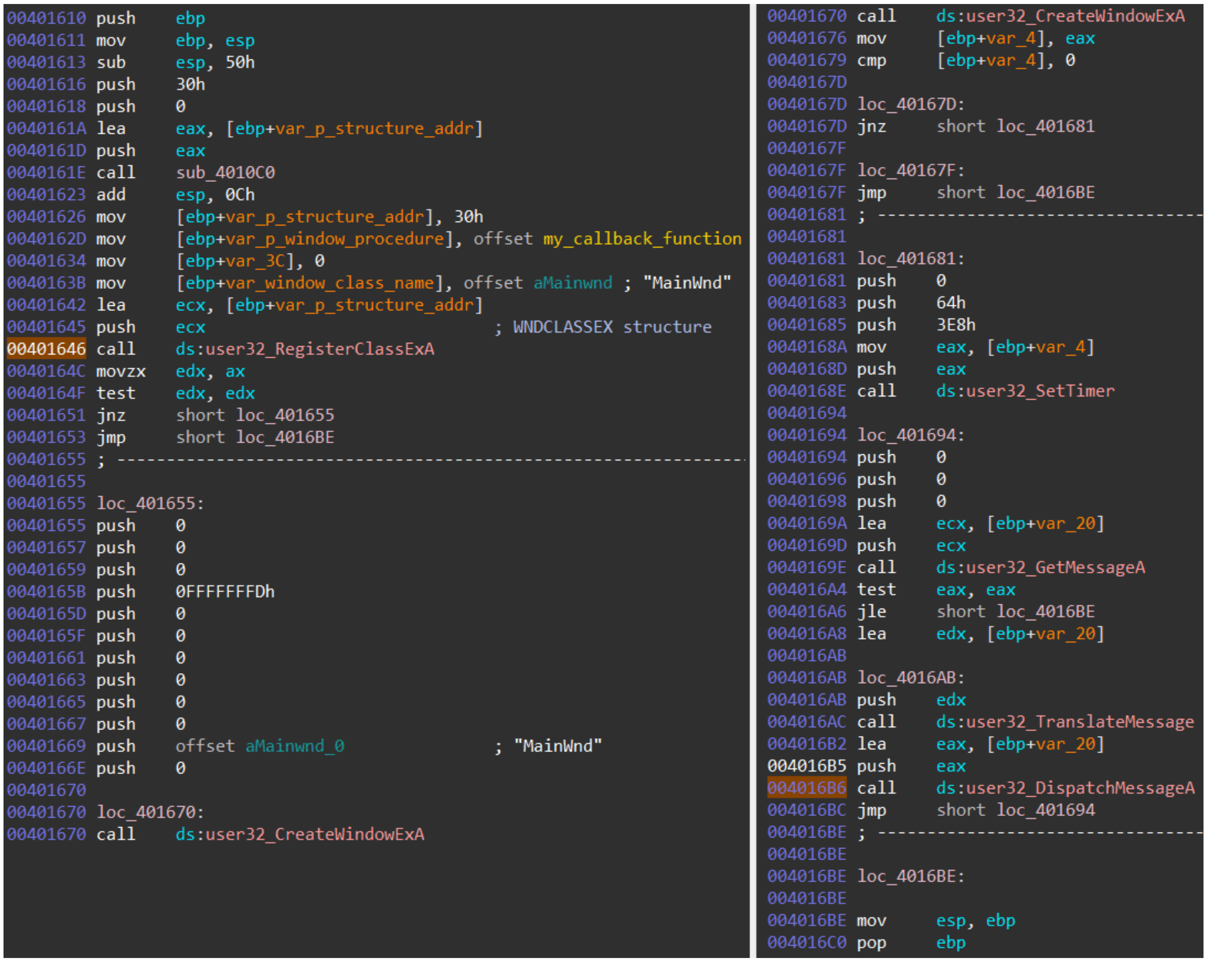
```
debug028:00230B18 E8 00 00 00 00    call    $+5
debug028:00230B1D 58                pop     eax
debug028:00230B1E EB 08             jmp     short loc_230B28
debug028:00230B1E                   ; -----------------------------
debug028:00230B20 5A 78 6B 65 6E 70+aZxkenpz_0 db 'ZxkenpZ',0
debug028:00230B28                   ; -----------------------------
debug028:00230B28
debug028:00230B28                   loc_230B28:
debug028:00230B28 83 C0 03          add     eax, 3
debug028:00230B2B 89 85 14 FF FF FF mov     [ebp-0ECh], eax
debug028:00230B31 58                pop     eax
debug028:00230B32 8B 95 14 FF FF FF mov     edx, [ebp-0ECh]
debug028:00230B38 52                push    edx
debug028:00230B39 FF 95 0C FF FF FF call    dword ptr [ebp-0F4h]
debug028:00230B3F 50                push    eax
debug028:00230B40 E8 00 00 00 00    call    $+5
debug028:00230B45 58                pop     eax
debug028:00230B46 89 45 A8          mov     [ebp-58h], eax
debug028:00230B49 58                pop     eax
debug028:00230B4A
debug028:00230B4A                   self_destruct:
debug028:00230B4A 8B 45 A8          mov     eax, [ebp-58h]
debug028:00230B4D 3B 45 A4          cmp     eax, [ebp-5Ch]
debug028:00230B50 74 11             jz      short end_self_destruct
debug028:00230B52 8B 4D A8          mov     ecx, [ebp-58h]
debug028:00230B55 C6 01 00          mov     byte ptr [ecx], 0
debug028:00230B58 8B 55 A8          mov     edx, [ebp-58h]
debug028:00230B5B 83 EA 01          sub     edx, 1
debug028:00230B5E 89 55 A8          mov     [ebp-58h], edx
debug028:00230B61 EB E7             jmp     short self_destruct
```

Image seven: self destruction code in action (as seen via IDA debugger)

Given the fact that the module is mapped in a newly allocated memory region (image one), one can only guess why the packer's author didn't just free the region. Maybe (s)he wanted to avoid analysis techniques which dump code by hooking VirtualFree calls? Maybe (s)he wanted to keep the modules nicely separated (VirtualFree can not be called before execution is transferred to another region/module, as a VirtualFree call would destroy the code responsible for said execution transferring)? After destroying everything, a jump is made to the entrypoint of the second layer executable, to which I will refer as module four.

Decrypt Hancitor binary

Module four (disassembled code) contains a debug-thwarting trick which can be confusing if you are not aware of what is happening. The module makes use of a technique called control flow obfuscation. **The goal of the trick is to make use of a Windows API call in such a way that the main code flow does not continue on the code following the API call. Instead the main code flow is transferred to a callback function which is executed during the API call. If you are not aware of this trick, you would probably jump over each instruction in module four which would result in loosing control over the execution, since no debugger points are set in the registered callback function.** Image eight shows how the Hancitor packer makes use of this technique.



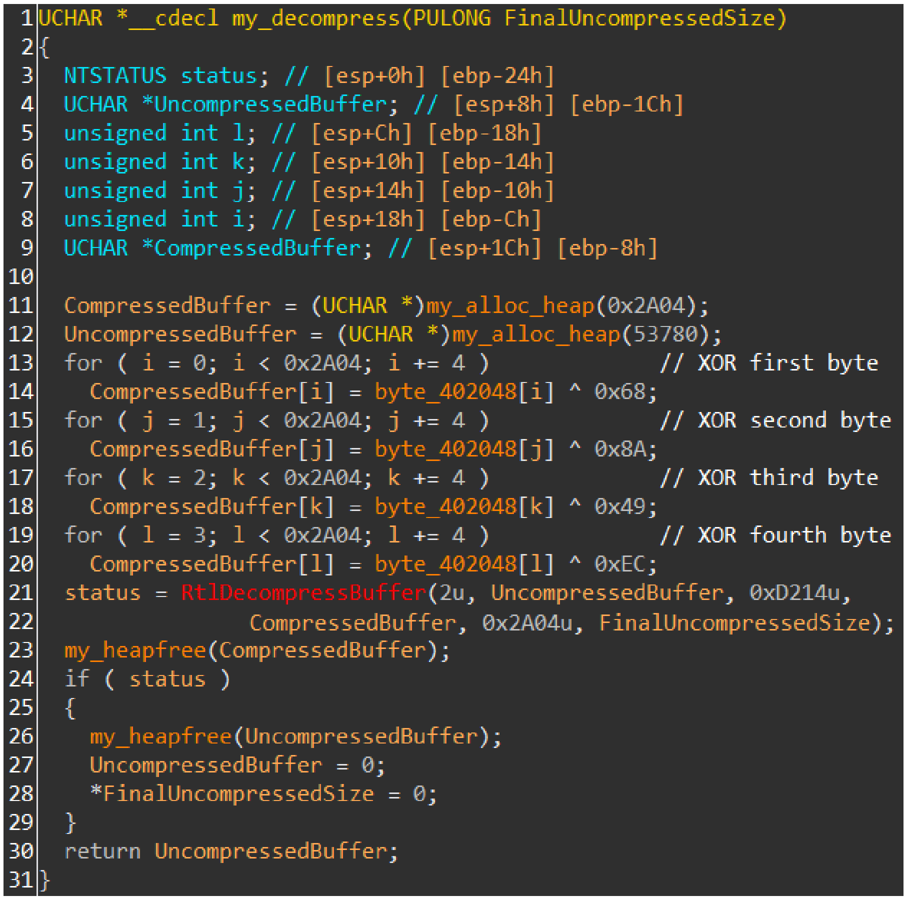
```
00401610 push    ebp
00401611 mov     ebp, esp
00401613 sub     esp, 50h
00401616 push    30h
00401618 push    0
0040161A lea     eax, [ebp+var_p_structure_addr]
0040161D push    eax
0040161E call    sub_4010C0
00401623 add     esp, 0Ch
00401626 mov     [ebp+var_p_structure_addr], 30h
0040162D mov     [ebp+var_p_window_procedure], offset my_callback_function
00401634 mov     [ebp+var_3C], 0
0040163B mov     [ebp+var_window_class_name], offset aMainwnd ; "MainWnd"
00401642 lea     ecx, [ebp+var_p_structure_addr]
00401645 push    ecx                             ; WNDCLASSEX structure
00401646 call    ds:user32_RegisterClassExA
0040164C movzx   edx, ax
0040164F test    edx, edx
00401651 jnz     short loc_401655
00401653 jmp     short loc_4016BE
00401655 ; ---------------------------------------------------------------
00401655
00401655 loc_401655:
00401655 push    0
00401657 push    0
00401659 push    0
0040165B push    0FFFFFFFDh
0040165D push    0
0040165F push    0
00401661 push    0
00401663 push    0
00401665 push    0
00401667 push    0
00401669 push    offset aMainwnd_0               ; "MainWnd"
0040166E push    0
00401670
00401670 loc_401670:
00401670 call    ds:user32_CreateWindowExA
```

```
00401670 call    ds:user32_CreateWindowExA
00401676 mov     [ebp+var_4], eax
00401679 cmp     [ebp+var_4], 0
0040167D
0040167D loc_40167D:
0040167D jnz     short loc_401681
0040167F
0040167F loc_40167F:
0040167F jmp     short loc_4016BE
00401681 ; --------------------------------
00401681
00401681 loc_401681:
00401681 push    0
00401683 push    64h
00401685 push    3E8h
0040168A mov     eax, [ebp+var_4]
0040168D push    eax
0040168E call    ds:user32_SetTimer
00401694
00401694 loc_401694:
00401694 push    0
00401696 push    0
00401698 push    0
0040169A lea     ecx, [ebp+var_20]
0040169D push    ecx
0040169E call    ds:user32_GetMessageA
004016A4 test    eax, eax
004016A6 jle     short loc_4016BE
004016A8 lea     edx, [ebp+var_20]
004016AB
004016AB loc_4016AB:
004016AB push    edx
004016AC call    ds:user32_TranslateMessage
004016B2 lea     eax, [ebp+var_20]
004016B5 push    eax
004016B6 call    ds:user32_DispatchMessageA
004016BC jmp     short loc_401694
004016BE ; --------------------------------
004016BE
004016BE loc_4016BE:
004016BE
004016BE mov     esp, ebp
004016C0 pop     ebp
```

Image eight: Control flow obfuscation by making use of Window Procedures (RegisterClassExA & CreateWindowExA)

The callback function is registered as part of a Windows Class Ex structure, which is passed as an argument to the RegisterClassExA API call. When a call is made to the DispatchMessageA API, the callback function gets executed. The callback function contains a jump to the fifth module.

Module five (disassembled code) does not contain many interesting functions. The most important function is a function which decrypts and decompresses the Hancitor executable (if you are still reading at this point, you probably wondered when we would ever get to this stage 😊). The encrypted executable is stored as data inside layer two, the decryption is performed by three simple XOR loops, as can be seen on the decompiled function code on image nine.



```
UCHAR *__cdecl my_decompress(PULONG FinalUncompressedSize)
{
  NTSTATUS status; // [esp+0h] [ebp-24h]
  UCHAR *UncompressedBuffer; // [esp+8h] [ebp-1Ch]
  unsigned int l; // [esp+Ch] [ebp-18h]
  unsigned int k; // [esp+10h] [ebp-14h]
  unsigned int j; // [esp+14h] [ebp-10h]
  unsigned int i; // [esp+18h] [ebp-Ch]
  UCHAR *CompressedBuffer; // [esp+1Ch] [ebp-8h]

  CompressedBuffer = (UCHAR *)my_alloc_heap(0x2A04);
  UncompressedBuffer = (UCHAR *)my_alloc_heap(53780);
  for ( i = 0; i < 0x2A04; i += 4 )                 // XOR first byte
    CompressedBuffer[i] = byte_402048[i] ^ 0x68;
  for ( j = 1; j < 0x2A04; j += 4 )                 // XOR second byte
    CompressedBuffer[j] = byte_402048[j] ^ 0x8A;
  for ( k = 2; k < 0x2A04; k += 4 )                 // XOR third byte
    CompressedBuffer[k] = byte_402048[k] ^ 0x49;
  for ( l = 3; l < 0x2A04; l += 4 )                 // XOR fourth byte
    CompressedBuffer[l] = byte_402048[l] ^ 0xEC;
  status = RtlDecompressBuffer(2u, UncompressedBuffer, 0xD214u,
              CompressedBuffer, 0x2A04u, FinalUncompressedSize);
  my_heapfree(CompressedBuffer);
  if ( status )
  {
    my_heapfree(UncompressedBuffer);
    UncompressedBuffer = 0;
    *FinalUncompressedSize = 0;
  }
  return UncompressedBuffer;
}
```

Image nine: decompiled decryption code

The decompression is performed via a function call to RtlDecompressBuffer (note that the address of this API was resolved in module three, the puzzle pieces are starting to come together!). The decrypted executable is mapped into a newly allocated memory region, to which we will refer to as module six.

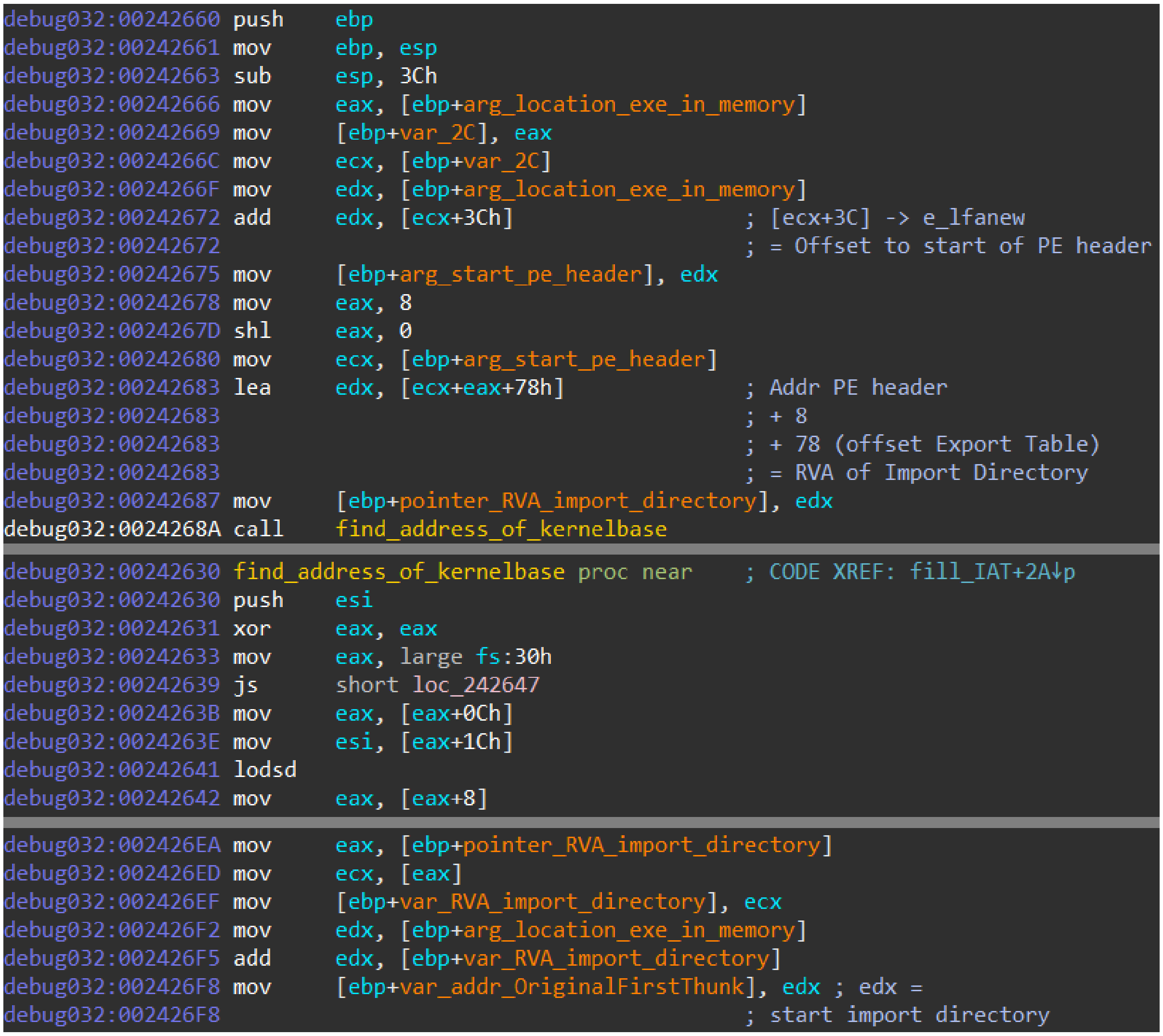
Reconstruct import table

Module six (disassembled code) contains the last functionality of the packer. **The goal of the module is to emulate behavior which normally is performed by the Windows Loader: map libraries (DLLs) into the process' address space, resolve the addresses of APIs and store those addresses in the executable's Import Address Table (IAT).** This behavior needs to be emulated by the packer because it has loaded the Hancitor executable directly into memory. If the Hancitor executable were to have been loaded from disk, the Windows Loader would have done its job. Obviously, loading the malware from disk is not feasible, as it would be detected quickly by security products. Code similar to the code in this module is frequently present in malware and greyhat tools which load an executable reflectively. As the reader will notice, the reverse engineered code discussed below for example looks very similar to a leaked Gozi/IFSB code part (mirror) which is described by the author as: 'a routine used to create, initialize and execute [a] PE-image without a file'.

I am *not* a suitable person to write referral material about PE structures 🤐. However, for the sake of giving some background information on the actions which are performed in module six, I'll try to briefly write down some pointers about the PE's import tables.

The IAT is a table of pointers to function (API) addresses which is used as a lookup table when an application is calling a function. The addresses of functions inside a library (DLL) are not static but change when updated versions of the DLL are released, so applications cannot be built using hardcoded function addresses. In order for the Windows Loader to know which libraries and functions it needs to import, they obviously need to be defined inside the executable. This is where the Import Directory Table (IDT) comes into play.

The IDT contains structures which contain information about a DLL which a PE file imports functions from. Two important fields in those structures are FirstThunk: a relative virtual address (RVA) inside the IAT, and OriginalFirstThunk: a RVA of the Import Lookup Table (ILT). The Import Lookup Table contains an array of RVAs, each RVA points to a hint/name table (source: PE format, Microsoft). As the name suggests, the hint/name table contains the name of a function which needs to be imported.
Module six starts by calculating the in-memory start address of the Import Directory Table. It calculates said address by parsing the PE header of the in-memory mapped Hancitor executable, as can be seen on image ten. First, the executable searches for the start offset of the PE header, a value which is stored at the e_lfanew field (ref: PE offsets). The module then jumps to a certain offset from the start of the PE header to locate a field whose value contains the RVA of the Import Directory. Because this value is a *relative* offset, the value needs to be added to the in-memory start of the mapped executable. This resulting calculation contains the in-memory start of the Import Directory Table.



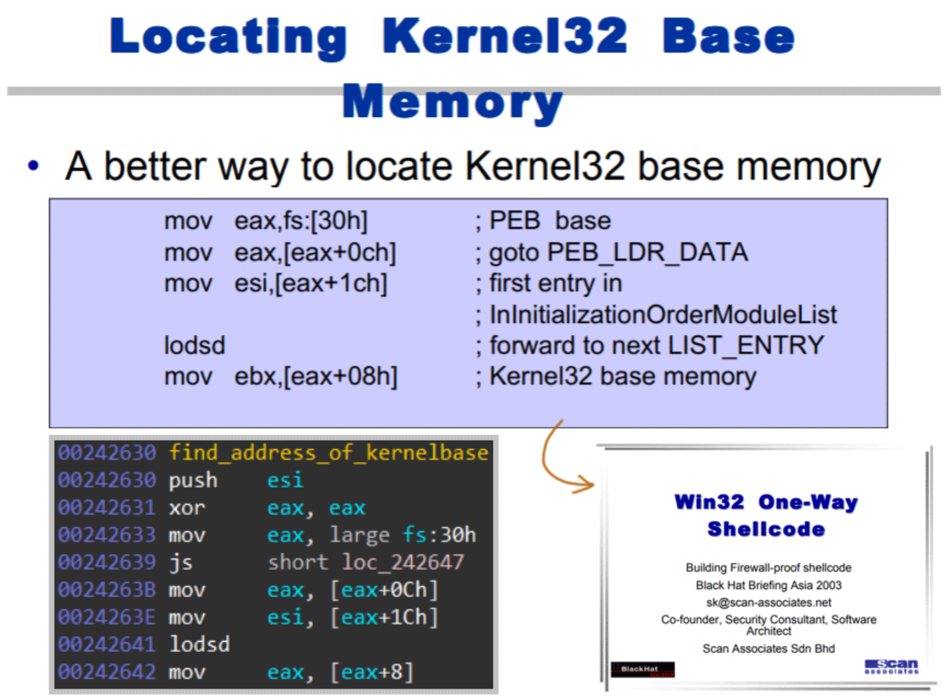
```
debug032:00242660 push    ebp
debug032:00242661 mov     ebp, esp
debug032:00242663 sub     esp, 3Ch
debug032:00242666 mov     eax, [ebp+arg_location_exe_in_memory]
debug032:00242669 mov     [ebp+var_2C], eax
debug032:0024266C mov     ecx, [ebp+var_2C]
debug032:0024266F mov     edx, [ebp+arg_location_exe_in_memory]
debug032:00242672 add     edx, [ecx+3Ch]                   ; [ecx+3C] -> e_lfanew
debug032:00242672                                          ; = Offset to start of PE header
debug032:00242675 mov     [ebp+arg_start_pe_header], edx
debug032:00242678 mov     eax, 8
debug032:0024267D shl     eax, 0
debug032:00242680 mov     ecx, [ebp+arg_start_pe_header]
debug032:00242683 lea     edx, [ecx+eax+78h]               ; Addr PE header
debug032:00242683                                          ; + 8
debug032:00242683                                          ; + 78 (offset Export Table)
debug032:00242683                                          ; = RVA of Import Directory
debug032:00242687 mov     [ebp+pointer_RVA_import_directory], edx
debug032:0024268A call    find_address_of_kernelbase
```

```
debug032:00242630 find_address_of_kernelbase proc near     ; CODE XREF: fill_IAT+2A↓p
debug032:00242630 push    esi
debug032:00242631 xor     eax, eax
debug032:00242633 mov     eax, large fs:30h
debug032:00242639 js      short loc_242647
debug032:0024263B mov     eax, [eax+0Ch]
debug032:0024263E mov     esi, [eax+1Ch]
debug032:00242641 lodsd
debug032:00242642 mov     eax, [eax+8]
```

```
debug032:002426EA mov     eax, [ebp+pointer_RVA_import_directory]
debug032:002426ED mov     ecx, [eax]
debug032:002426EF mov     [ebp+var_RVA_import_directory], ecx
debug032:002426F2 mov     edx, [ebp+arg_location_exe_in_memory]
debug032:002426F5 add     edx, [ebp+var_RVA_import_directory]
debug032:002426F8 mov     [ebp+var_addr_OriginalFirstThunk], edx ; edx =
debug032:002426F8                                          ; start import directory
```

Image ten: Resolve address of kernelbase & find address of import directory table

For module six to be able to map libraries (used by Hancitor) into the process' address space, it needs the memory location of kernel32's LoadLibrary and GetProcAddress functions. To retrieve the function addresses, the packer needs to figure out at which address (inside its own process address space) the kernel32 library is mapped. For this hunt the packer relies on a small piece of shellcode which reads the Process Environment Block (PEB). The below slide from a fifteen-years-old presentation about shellcode explains how the PEB is used to resolve kernel32's base address.



Locating Kernel32 Base Memory

- A better way to locate Kernel32 base memory

```
mov   eax,fs:[30h]          ; PEB  base
mov   eax,[eax+0ch]         ; goto PEB_LDR_DATA
mov   esi,[eax+1ch]         ; first entry in
                            ; InInitializationOrderModuleList
lodsd                       ; forward to next LIST_ENTRY
mov   ebx,[eax+08h]         ; Kernel32 base memory
```

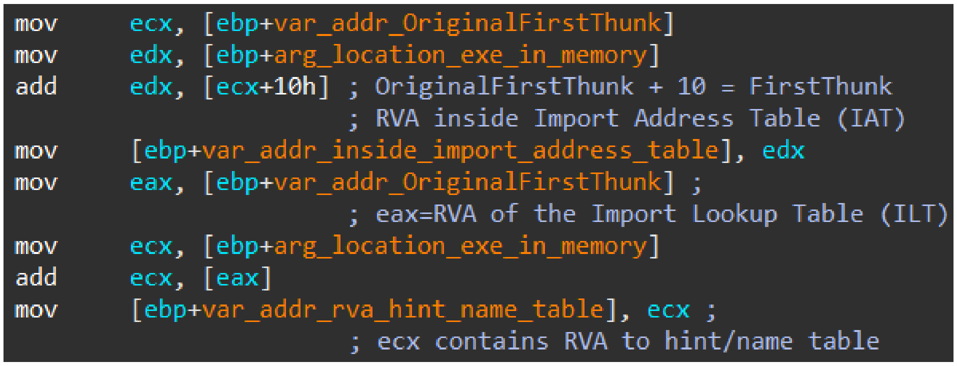
```
00242630 find_address_of_kernelbase
00242630 push    esi
00242631 xor     eax, eax
00242633 mov     eax, large fs:30h
00242639 js      short loc_242647
0024263B mov     eax, [eax+0Ch]
0024263E mov     esi, [eax+1Ch]
00242641 lodsd
00242642 mov     eax, [eax+8]
```

Win32 One-Way Shellcode

Building Firewall-proof shellcode
Black Hat Briefing Asia 2003
sk@scan-associates.net
Co-founder, Security Consultant, Software Architect
Scan Associates Sdn Bhd

Image eleven: Fifteen-year-old presentation discussing shellcode which retrieves the kernel32 base memory address

After having resolved the in-memory location of the LoadLibrary and GetProcAddress functions, module six reads the FirstThunk and the OriginalFirstThunk field values inside the Import Directory Table (image twelve, image thirteen).



```
mov     ecx, [ebp+var_addr_OriginalFirstThunk]
mov     edx, [ebp+arg_location_exe_in_memory]
add     edx, [ecx+10h] ; OriginalFirstThunk + 10 = FirstThunk
                       ; RVA inside Import Address Table (IAT)
mov     [ebp+var_addr_inside_import_address_table], edx
mov     eax, [ebp+var_addr_OriginalFirstThunk] ;
                       ; eax=RVA of the Import Lookup Table (ILT)
mov     ecx, [ebp+arg_location_exe_in_memory]
add     ecx, [eax]
mov     [ebp+var_addr_rva_hint_name_table], ecx ;
                       ; ecx contains RVA to hint/name table
```

Image twelve: Parsing Import Directory Table for OriginalFirstThunk & FirstThunk fields

By enumerating these fields, the module knows via the corresponding hint/name tables which functions need to be imported. The libraries are imported via calls to the LoadLibrary function, the function addresses are resolved via calls to the GetProcAddress function. The module writes the function addressess into Hancitor's Import Address Table. The result of this action can be seen on image fourteen (note that the Import Directory field values can be nicely visualised via Hasherezade's PE bear). A graphical overview of the relation between the fields and import tables discussed in this paragraph can be seen on image thirteen.

This action is the last action by the packer, the execution can now *finally* be transferred to Hancitor's code 🔅.

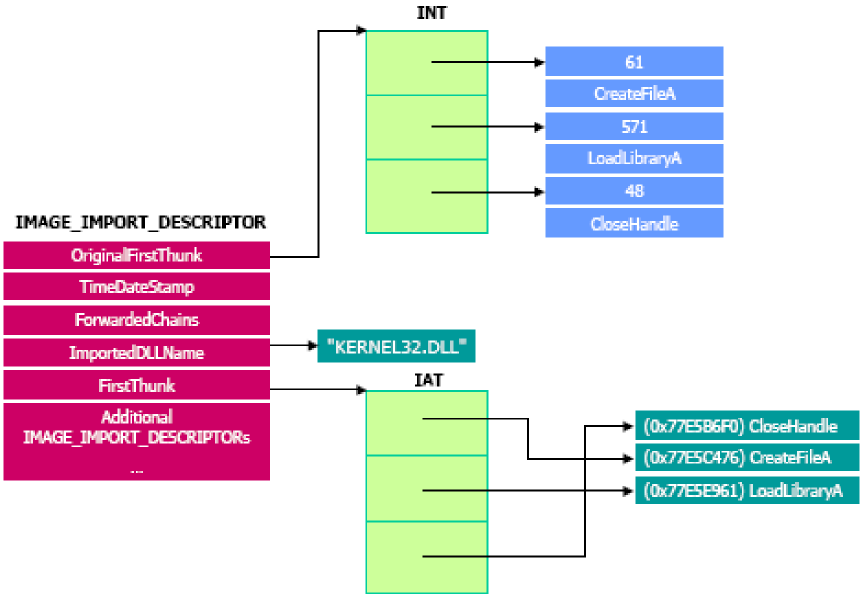
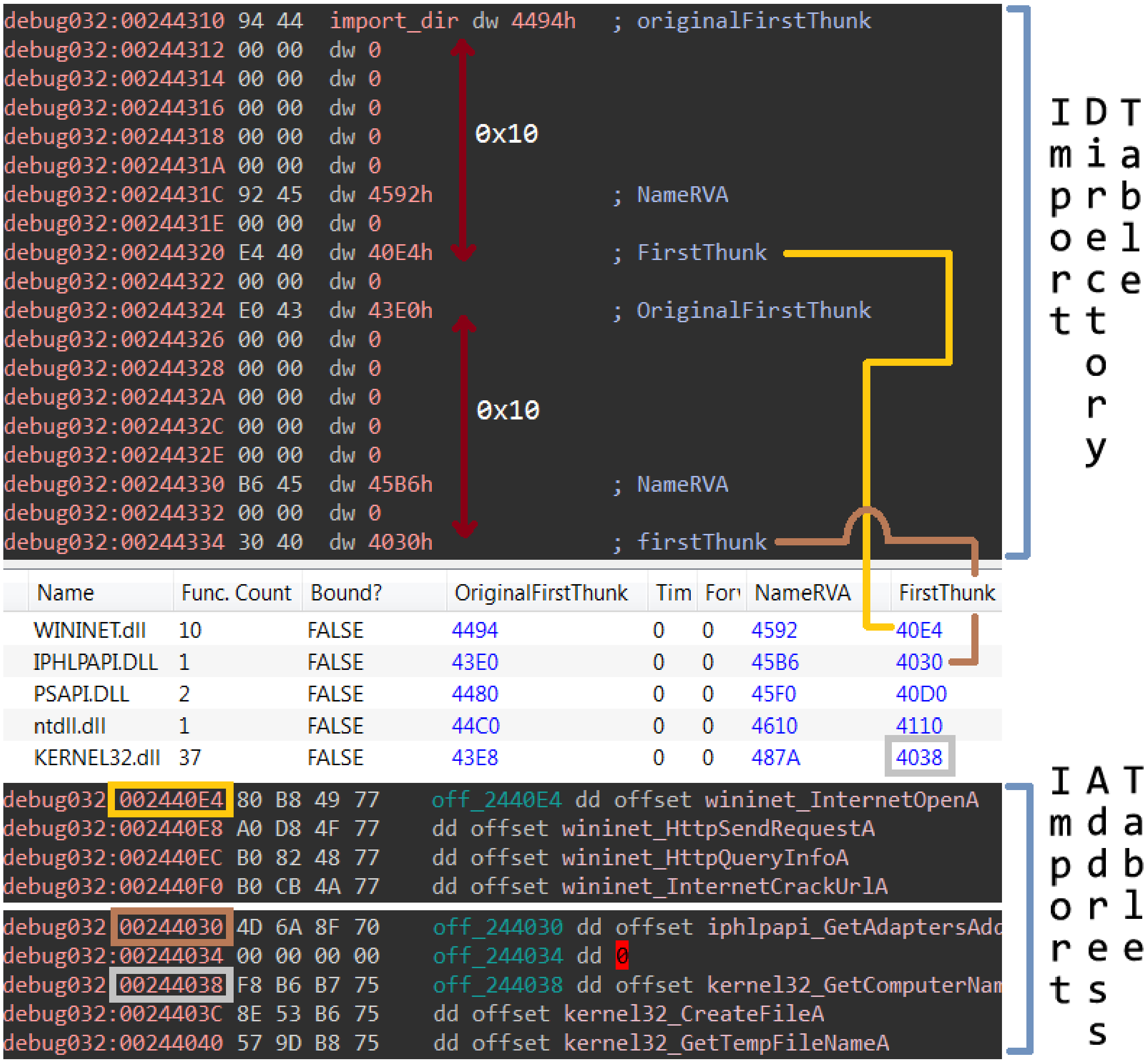


Image thirteen: Graphical overview of the relations between the import tables.
Source: dematte.org.



```
debug032:00244310 94 44  import_dir dw 4494h   ; originalFirstThunk
debug032:00244312 00 00  dw 0
debug032:00244314 00 00  dw 0
debug032:00244316 00 00  dw 0
debug032:00244318 00 00  dw 0
debug032:0024431A 00 00  dw 0
debug032:0024431C 92 45  dw 4592h              ; NameRVA
debug032:0024431E 00 00  dw 0
debug032:00244320 E4 40  dw 40E4h              ; FirstThunk
debug032:00244322 00 00  dw 0
debug032:00244324 E0 43  dw 43E0h              ; OriginalFirstThunk
debug032:00244326 00 00  dw 0
debug032:00244328 00 00  dw 0
debug032:0024432A 00 00  dw 0
debug032:0024432C 00 00  dw 0
debug032:0024432E 00 00  dw 0
debug032:00244330 B6 45  dw 45B6h              ; NameRVA
debug032:00244332 00 00  dw 0
debug032:00244334 30 40  dw 4030h              ; firstThunk
```

Name	Func. Count	Bound?	OriginalFirstThunk	Tim	For	NameRVA	FirstThunk
WININET.dll	10	FALSE	4494	0	0	4592	40E4
IPHLPAPI.DLL	1	FALSE	43E0	0	0	45B6	4030
PSAPI.DLL	2	FALSE	4480	0	0	45F0	40D0
ntdll.dll	1	FALSE	44C0	0	0	4610	4110
KERNEL32.dll	37	FALSE	43E8	0	0	487A	4038

```
debug032 002440E4 80 B8 49 77     off_2440E4 dd offset wininet_InternetOpenA
debug032:002440E8 A0 D8 4F 77     dd offset wininet_HttpSendRequestA
debug032:002440EC B0 82 48 77     dd offset wininet_HttpQueryInfoA
debug032:002440F0 B0 CB 4A 77     dd offset wininet_InternetCrackUrlA

debug032 00244030 4D 6A 8F 70     off_244030 dd offset iphlpapi_GetAdaptersAdd
debug032:00244034 00 00 00 00     off_244034 dd 0
debug032 00244038 F8 B6 B7 75     off_244038 dd offset kernel32_GetComputerNam
debug032:0024403C 8E 53 B6 75     dd offset kernel32_CreateFileA
debug032:00244040 57 9D B8 75     dd offset kernel32_GetTempFileNameA
```

Image fourteen: Parsing the Import Directory Table (IDT) with the ultimate goal of filling the Import Address Table (IAT)

Old packer, still does the job

During the hunt for additional packed Hancitor samples (using the below YARA rule), I noticed that some of the packed samples were protecting a malware family which didn't look like Hancitor at all 🧐. One sample protected some kind of Delphi malware which embedded the names of Turkish banks. The malware looked very similar to the ATMZombie malware, which Kaspersky blogged about (mirror). When we look at an ATMZombie sample which is explicitly mentioned in the Kaspersky blog, we can see that the packer of the mentioned sample is the same packer as the one which is discussed in this blog. Another packed sample which I noticed during my hunt protected a shellcode loader. The sample is mentioned in a Proofpoint blog (mirror) as a Metasploit Stager which in turn downloaded Cobalt Strike.

At this point it became clear to me that this packer has been around for a time, and that it isn't exclusively used by Hancitor. In fact, when I kept digging, I found many samples of (old) malware families which were packed by this packer. Some examples are: Zeus/Panda banker, Cryptowall, Ramnit, PoSeidon and Gootkit. All packed and unpacked malware samples can be found here (password=infected). When I launched a YARA search on parts of the encrypted module two bytes (there are 255 variations, as a single byte XOR key is used in the spaghetti code of module one), I found older versions of the packer. One example is a packed Qadars sample. The sample is mentioned as an IOC in an ESET article (mirror) from 2013. This suggests that the packer has been around for at least five years already.

Addendum: YARA Rule

```
import "pe"
rule hancitor_packer
{
  meta:
    author = "Felix Weyne, 2019"
    description = "Hancitor packer spaghetti code (loose match)"
    hash1= "37f6f1f59bf7952fd7182deeb07d4cd0d367dd59"
    hash2= "2508b3211b066022c2ab41725fbc400e8f3dec1e"
    hash3= "3855f6d9049936ddb29561d2ab4b2bf26df7a7ff"
    hash4= "e9ec4a4fb6f5d143b304df866bba4277cd473843"
  strings:
    //E9=JMP, EB=JMP SHORT, 71/0F=JNO
    $change_sp={89 EC (E9|EB|71|0F)}            //mov    esp,ebp
    $2={5D (E9|EB|71|0F)}                       //pop    ebp
    $3={BF ?? ?? ?? 00 (E9|EB|71|0F)}           //mov    edi, 274C67h
    $4={81 ?? ?? ?? ?? 00 (E9|EB|71|0F)}        //add    edi, 17E792h
    $5={57 (E9|EB|71|0F)}                       //push   edi
    $6={BE ?? ?? 00 00 (E9|EB|71|0F)}           //mov    esi, 88Bh
    $7={6A 00 (E9|EB|71|0F)}                    //push   0
    $8={54 (E9|EB|71|0F)}                       //push   esp
    $9={6A 40 (E9|EB|71|0F)}                    //push   40h
    $mov_eax={B8 ?? ?? ?? 00 (E9|EB|71|0F)}     //mov    eax, 5ADBh
    $add_eax={05 ?? ?? ?? 00 (E9|EB|71|0F)}     //add    eax, 0E525h
    $12={8B 00 (E9|EB|71|0F)}                   //mov    eax, [eax]
    $13={FF D0 (E9|EB|71|0F)}                   //call   eax
    $ecx_zero={B9 00 00 00 00 (E9|EB|71|0F)}    //mov    ecx, 0
    $xor={30 07 (E9|EB|71|0F)}                  //xor    [edi], al
    $18={41 (E9|EB|71|0F)}                      //inc    ecx
    $19={47 (E9|EB|71|0F)}                      //inc    edi
    $20={39 F1 (E9|EB|71|0F)}                   //cmp    ecx, esi
    $21={58 (E9|EB|71|0F)}                      //pop    eax
  condition:
    filesize < 110KB
    and pe.is_32bit()
    and #add_eax >= 3
    and #mov_eax >= 3
    and all of them
    and for any i in (1..#xor):($change_sp in (@xor[i][email protected][i]+400))
    and for any i in (1..#xor):($ecx_zero in (@xor[i][email protected][i]+300))
}
```