SANS ISC: Jumping into Shellcode - SANS Internet Storm Center SANS Site Network Current Site SANS Internet Storm Center Other SANS Sites Help Graduate Degree Programs Security Training Security Certification Security Awareness Training Penetration Testing Industrial Control Systems Cyber Defense Foundations DFIR Software Security Government OnSite Training SANS ISC InfoSec Forums

isc.sans.edu/forums/diary/Jumping+into+Shellcode/27256/

Malware analysis is exciting because you never know what you will find. In previous diaries[1], I already explained why it's important to have a look at groups of interesting Windows API call to detect some behaviors. The classic example is code injection. Usually, it is based on something like this:

1. You allocate some memory
2. You get a shellcode (downloaded, extracted from a specific location like a section, a resource, ...)
3. You copy the shellcode in the newly allocated memory region
4. You create a new threat to execute it.

But it's not always like this! Last week, I worked on an incident involving a malicious DLL that I analyzed. The technique used to execute the shellcode was slightly different and therefore interesting to describe it here.

The DLL was delivered on the target system with an RTF document. This file contained the shellcode:

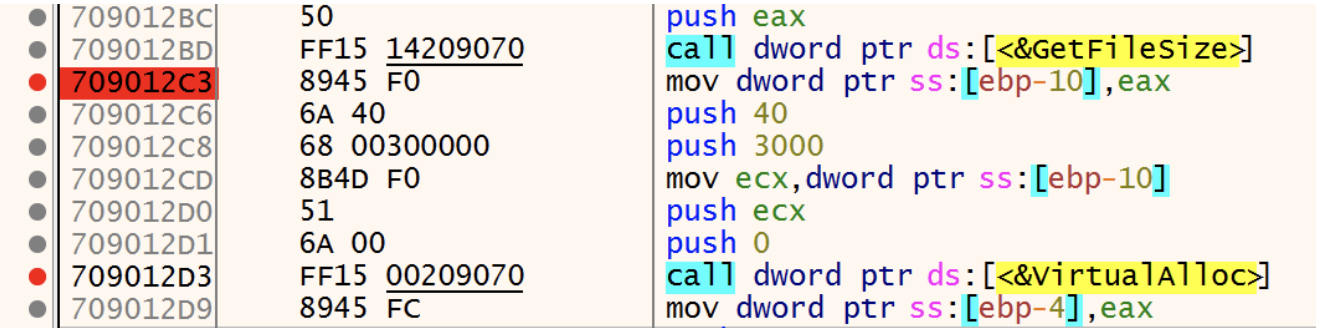
```
remnux@remnux:/MalwareZoo/20210318$ rtfdump.py suspicious.rtf
    1 Level  1        c=    3 p=00000000 l=    1619 h=     143;       5 b=       0
u=     539 \rtf1
    2  Level  2       c=    2 p=00000028 l=      91 h=       8;       2 b=       0
u=      16 \fonttbl
    3   Level  3      c=    0 p=00000031 l=      35 h=       3;       2 b=       0
u=       5 \f0
    4   Level  3      c=    0 p=00000056 l=      44 h=       5;       2 b=       0
u=      11 \f1
    5  Level  2       c=    0 p=00000087 l=      33 h=       0;       4 b=       0
u=       2 \colortbl
    6  Level  2       c=    0 p=000000ac l=      32 h=      13;       5 b=       0
u=       5 \*\generator
    7 Remainder       c=    0 p=00000655 l=  208396 h=   17913;       5 b=       0
u=  182176
      Whitespace = 4878  NULL bytes = 838  Left curly braces = 832  Right curly
braces = 818
```

This file is completely valid from an RTF format point of view, will open successfully, and render a fake document. But the attacker appended the shellcode at the end of the file (have a look at stream 7 which has a larger size and a lot of unexpected characters ("u="). Let's try to have a look at the shellcode:

```
remnux@remnux:/MalwareZoo/20210318$ rtfdump.py suspicious.rtf -s 7 | head -20
00000000: 0D 0A 00 6E 07 5D A7 5E  66 D2 97 1F 65 31 FD 7E  ...n.].^f...e1.~
00000010: D9 8E 9A C4 1C FC 73 79  F0 0B DA EA 6E 06 C3 03  ......sy....n...
00000020: 27 7C BD D7 23 84 0B BD  73 0C 0F 8D F9 DF CC E7  '|..#...s.......
00000030: 88 B9 97 06 A2 F9 4D 8C  91 D1 5E 39 A2 F5 9A 7E  ......M...^9...~
00000040: 4C D6 C8 A2 2D 88 D0 C4  16 E6 2B 1C DA 7B DD F7  L...-.....+..{..
00000050: C4 FB 61 34 A6 BE 8E 2F  9D 7D 96 A8 7E 00 E2 E8  ..a4.../.}..~...
00000060: BB A2 D9 53 1C F3 49 81  77 93 30 16 11 9D 88 93  ...S..I.w.0.....
00000070: D2 6C 9D 56 60 36 66 BA  29 3E 73 45 CE 1A BE E3  .l.V`6f.)>sE....
00000080: 5A C7 96 63 E0 D7 DF C9  21 2F 56 81 BD 84 6C 2D  Z..c....!/V...l-
00000090: CF 4C 4E BE 90 23 47 DC  A7 A9 8E A2 C3 A3 2E D1  .LN..#G.........
```

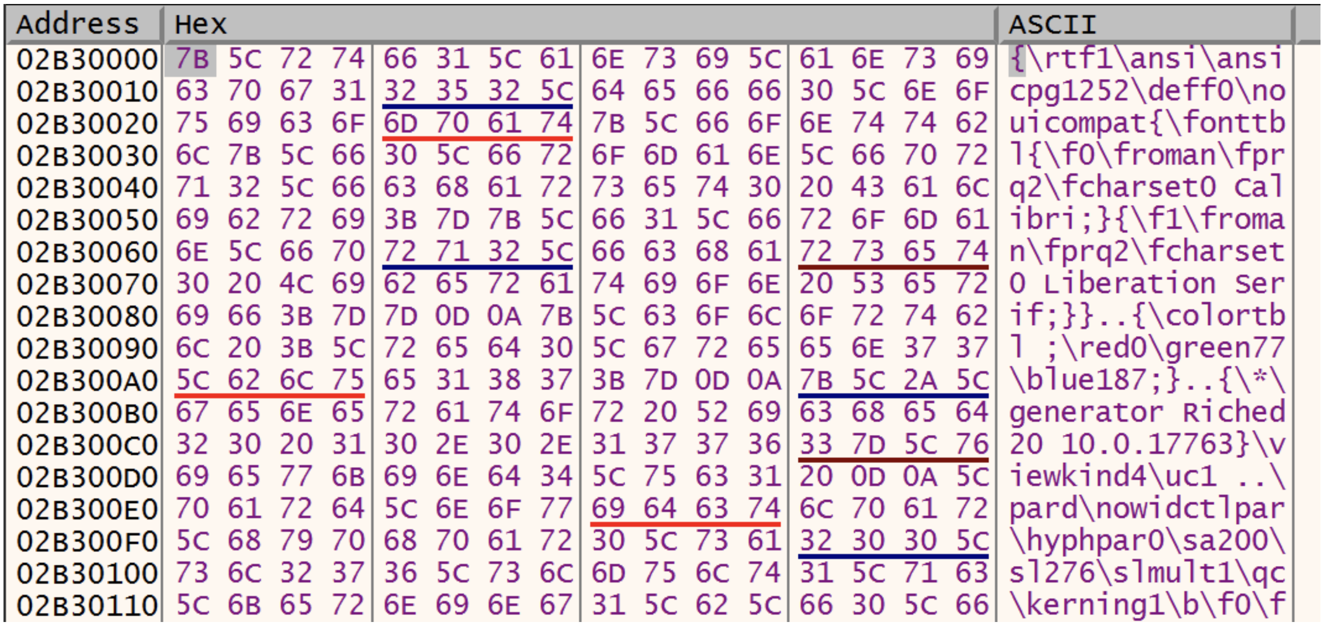
It looks encrypted and a brute force of a single XOR encoding was not successful. Let's see how it works in a debugger.

First, the RTF file is opened to get a handle and its size is fetched with `GetFileSize()`. Then, a classic `VirtualAlloc()` is used to allocate a memory space equal to the size of the file. Note the "push 40" which means that the memory will contain executable code (PAGE_EXECUTE_READWRITE):



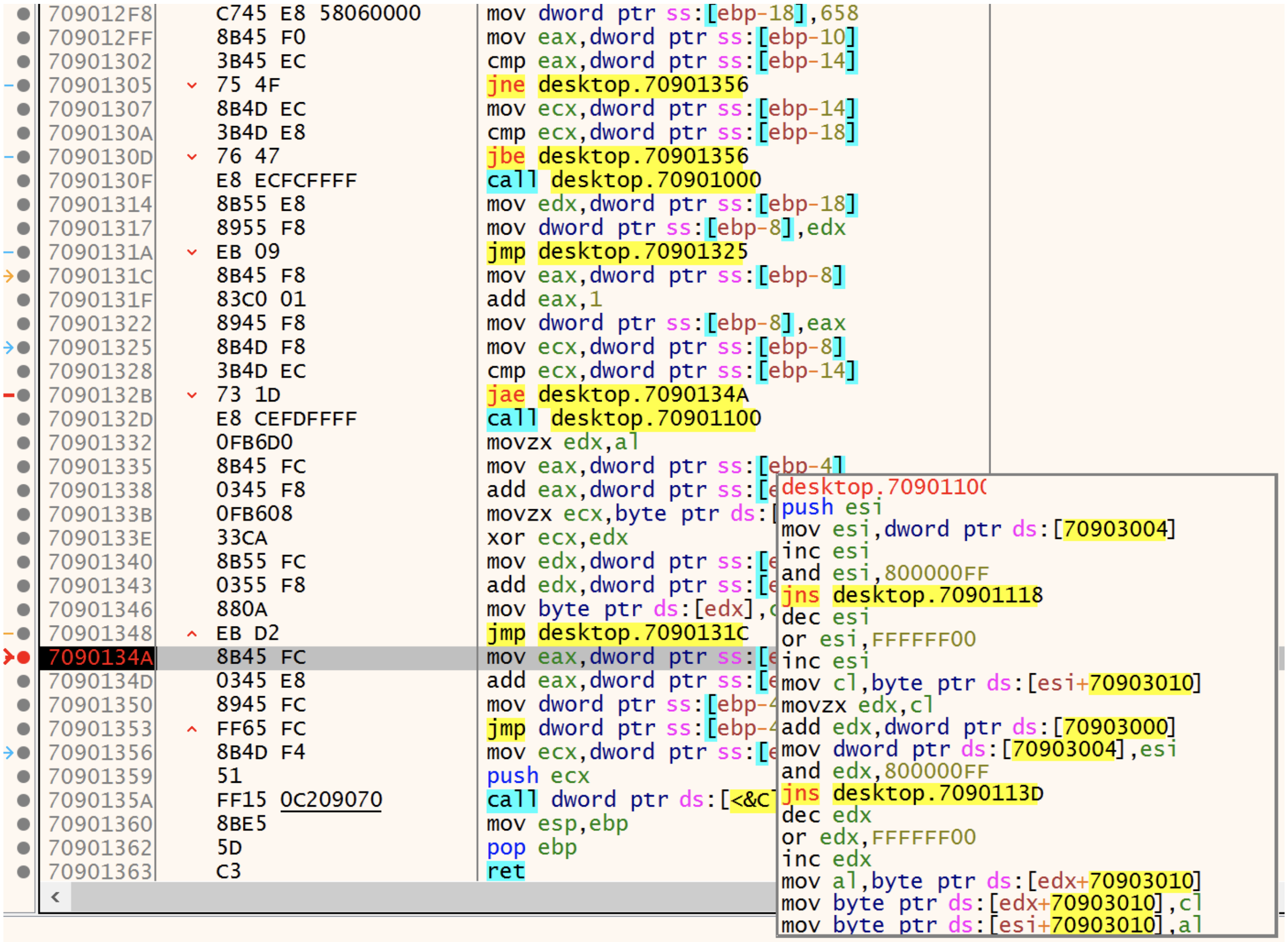
```
709012BC    50               push eax
709012BD    FF15 14209070    call dword ptr ds:[<&GetFileSize>]
709012C3    8945 F0          mov dword ptr ss:[ebp-10],eax
709012C6    6A 40            push 40
709012C8    68 00300000      push 3000
709012CD    8B4D F0          mov ecx,dword ptr ss:[ebp-10]
709012D0    51               push ecx
709012D1    6A 00            push 0
709012D3    FF15 00209070    call dword ptr ds:[<&VirtualAlloc>]
709012D9    8945 FC          mov dword ptr ss:[ebp-4],eax
```

Usually, the shellcode is extracted from the file by reading the exact amount of bytes. The malware jumps to the position of the shellcode start in the file and reads bytes until the EOF. In this case, the complete RTF file is read then copied into the newly allocated memory:



```
Address   Hex                                              ASCII
02B30000  7B 5C 72 74 66 31 5C 61 6E 73 69 5C 61 6E 73 69  {\rtf1\ansi\ansi
02B30010  63 70 67 31 32 35 32 5C 64 65 66 66 30 5C 6E 6F  cpg1252\deff0\no
02B30020  75 69 63 6F 6D 70 61 74 7B 5C 66 6F 6E 74 74 62  uicompat{\fonttb
02B30030  6C 7B 5C 66 30 5C 66 72 6F 6D 61 6E 5C 66 70 72  l{\f0\froman\fpr
02B30040  71 32 5C 66 63 68 61 72 73 65 74 30 20 43 61 6C  q2\fcharset0 Cal
02B30050  69 62 72 69 3B 7D 7B 5C 66 31 5C 66 72 6F 6D 61  ibri;}{\f1\froma
02B30060  6E 5C 66 70 72 71 32 5C 66 63 68 61 72 73 65 74  n\fprq2\fcharset
02B30070  30 20 4C 69 62 65 72 61 74 69 6F 6E 20 53 65 72  0 Liberation Ser
02B30080  69 66 3B 7D 7D 0D 0A 7B 5C 63 6F 6C 6F 72 74 62  if;}}..{\colortb
02B30090  6C 20 3B 5C 72 65 64 30 5C 67 72 65 65 6E 37 37  l ;\red0\green77
02B300A0  5C 62 6C 75 65 31 38 37 3B 7D 0D 0A 7B 5C 2A 5C  \blue187;}..{\*\
02B300B0  67 65 6E 65 72 61 74 6F 72 20 52 69 63 68 65 64  generator Riched
02B300C0  32 30 20 31 30 2E 30 2E 31 37 37 36 33 7D 5C 76  20 10.0.17763}\v
02B300D0  69 65 77 6B 69 6E 64 34 5C 75 63 31 20 0D 0A 5C  iewkind4\uc1 ..\
02B300E0  70 61 72 64 5C 6E 6F 77 69 64 63 74 6C 70 61 72  pard\nowidctlpar
02B300F0  5C 68 79 70 68 70 61 72 30 5C 73 61 32 30 30 5C  \hyphpar0\sa200\
02B30100  73 6C 32 37 36 5C 73 6C 6D 75 6C 74 31 5C 71 63  sl276\slmult1\qc
02B30110  5C 6B 65 72 6E 69 6E 67 31 5C 62 5C 66 30 5C 66  \kerning1\b\f0\f
```

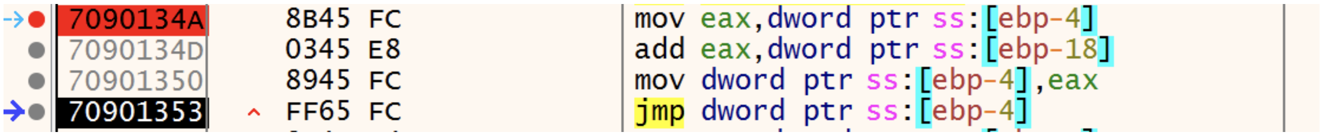
This is the interesting part of the code which processes the shellcode:



```
709012F8      C745 E8 58060000   mov dword ptr ss:[ebp-18],658
709012FF      8B45 F0            mov eax,dword ptr ss:[ebp-10]
70901302      3B45 EC            cmp eax,dword ptr ss:[ebp-14]
70901305    ˅ 75 4F              jne desktop.70901356
70901307      8B4D EC            mov ecx,dword ptr ss:[ebp-14]
7090130A      3B4D E8            cmp ecx,dword ptr ss:[ebp-18]
7090130D    ˅ 76 47              jbe desktop.70901356
7090130F      E8 ECFCFFFF        call desktop.70901000
70901314      8B55 E8            mov edx,dword ptr ss:[ebp-18]
70901317      8955 F8            mov dword ptr ss:[ebp-8],edx
7090131A    ˅ EB 09              jmp desktop.70901325
7090131C      8B45 F8            mov eax,dword ptr ss:[ebp-8]
7090131F      83C0 01            add eax,1
70901322      8945 F8            mov dword ptr ss:[ebp-8],eax
70901325      8B4D F8            mov ecx,dword ptr ss:[ebp-8]
70901328      3B4D EC            cmp ecx,dword ptr ss:[ebp-14]
7090132B    ˅ 73 1D              jae desktop.7090134A
7090132D      E8 CEFDFFFF        call desktop.70901100
70901332      0FB6D0             movzx edx,al
70901335      8B45 FC            mov eax,dword ptr ss:[ebp-4]
70901338      0345 F8            add eax,dword ptr ss:[e
7090133B      0FB608             movzx ecx,byte ptr ds:[
7090133E      33CA               xor ecx,edx
70901340      8B55 FC            mov edx,dword ptr ss:[e
70901343      0355 F8            add edx,dword ptr ss:[e
70901346      880A               mov byte ptr ds:[edx],c
70901348    ˄ EB D2              jmp desktop.7090131C
7090134A      8B45 FC            mov eax,dword ptr ss:[e
7090134D      0345 E8            add eax,dword ptr ss:[e
70901350      8945 FC            mov dword ptr ss:[ebp-4
70901353    ˄ FF65 FC            jmp dword ptr ss:[ebp-4
70901356      8B4D F4            mov ecx,dword ptr ss:[e
70901359      51                 push ecx
7090135A      FF15 0C209070      call dword ptr ds:[<&C
70901360      8BE5               mov esp,ebp
70901362      5D                 pop ebp
70901363      C3                 ret
```

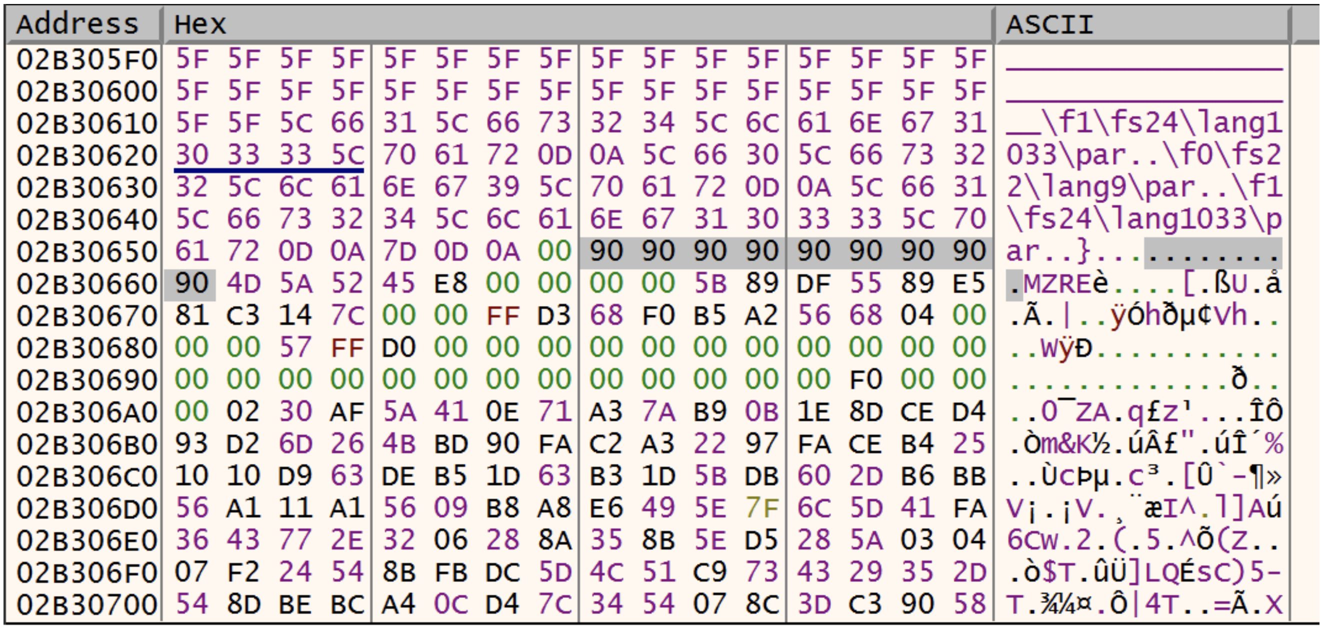
```
desktop.7090110(
push esi
mov esi,dword ptr ds:[70903004]
inc esi
and esi,800000FF
jns desktop.70901118
dec esi
or esi,FFFFFF00
inc esi
mov cl,byte ptr ds:[esi+70903010]
movzx edx,cl
add edx,dword ptr ds:[70903000]
mov dword ptr ds:[70903004],esi
and edx,800000FF
jns desktop.7090113D
dec edx
or edx,FFFFFF00
inc edx
mov al,byte ptr ds:[edx+70903010]
mov byte ptr ds:[edx+70903010],cl
mov byte ptr ds:[esi+70903010],al
```

The first line " `mov word ptr ss:[ebp-18], 658` " defines where the shellcode starts in the memory map. In a loop, all characters are XOR'd with a key that is generated in the function `desktop.70901100` . The next step is to jump to the location of the decoded shellcode:



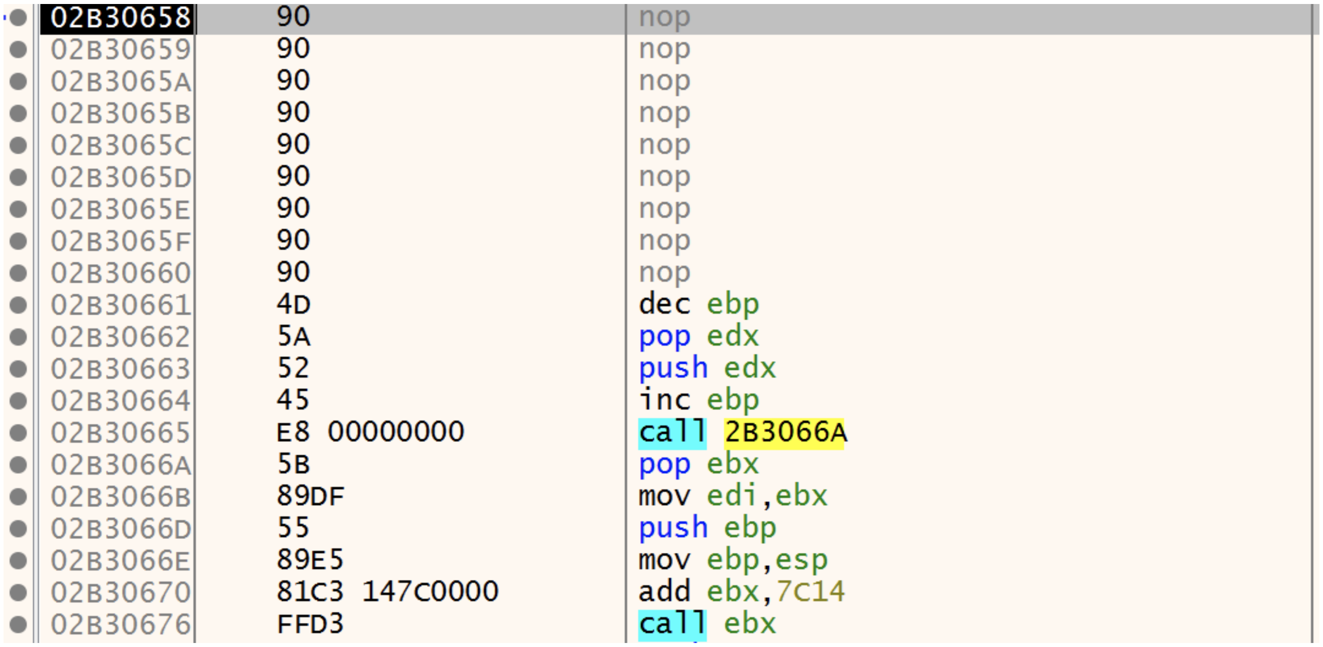
```
7090134A      8B45 FC       mov eax,dword ptr ss:[ebp-4]
7090134D      0345 E8       add eax,dword ptr ss:[ebp-18]
70901350      8945 FC       mov dword ptr ss:[ebp-4],eax
70901353    ˄ FF65 FC       jmp dword ptr ss:[ebp-4]
```

The address where to jump is based on the address of the newly allocated memory (0x2B30000) + the offset (658). Let's have a look at this location (0x2B30658):



```
Address   Hex                                               ASCII
02B305F0  5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F  ________________
02B30600  5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F 5F  ________________
02B30610  5F 5F 5C 66 31 5C 66 73 32 34 5C 6C 61 6E 67 31  __\f1\fs24\lang1
02B30620  30 33 33 5C 70 61 72 0D 0A 5C 66 30 5C 66 73 32  033\par..\f0\fs2
02B30630  32 5C 6C 61 6E 67 39 5C 70 61 72 0D 0A 5C 66 31  2\lang9\par..\f1
02B30640  5C 66 73 32 34 5C 6C 61 6E 67 31 30 33 33 5C 70  \fs24\lang1033\p
02B30650  61 72 0D 0A 7D 0D 0A 00 90 90 90 90 90 90 90 90  ar..}...........
02B30660  90 4D 5A 52 45 E8 00 00 00 00 5B 89 DF 55 89 E5  .MZREè....[.ßU.å
02B30670  81 C3 14 7C 00 00 FF D3 68 F0 B5 A2 56 68 04 00  .Ã.|..ÿÓhðµ¢Vh..
02B30680  00 00 57 FF D0 00 00 00 00 00 00 00 00 00 00 00  ..WÿÐ...........
02B30690  00 00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00  .............ð..
02B306A0  00 02 30 AF 5A 41 0E 71 A3 7A B9 0B 1E 8D CE D4  ..0¯ZA.q£z¹...ÎÔ
02B306B0  93 D2 6D 26 4B BD 90 FA C2 A3 22 97 FA CE B4 25  .Òm&K½.úÂ£".úÎ´%
02B306C0  10 10 D9 63 DE B5 1D 63 B3 1D 5B DB 60 2D B6 BB  ..ÙcÞµ.c³.[Û`-¶»
02B306D0  56 A1 11 A1 56 09 B8 A8 E6 49 5E 7F 6C 5D 41 FA  V¡.¡V.¸¨æI^.l]Aú
02B306E0  36 43 77 2E 32 06 28 8A 35 8B 5E D5 28 5A 03 04  6Cw.2.(.5.^Õ(Z..
02B306F0  07 F2 24 54 8B FB DC 5D 4C 51 C9 73 43 29 35 2D  .ò$T.ûÜ]LQÉsC)5-
02B30700  54 8D BE BC A4 0C D4 7C 34 54 07 8C 3D C3 90 58  T.¾¼¤.Ô|4T..=Ã.X
```

Sounds good, we have a NOP sled at this location + the string "MZ". Let's execute the unconditional JMP:



```
02B30658  90            nop
02B30659  90            nop
02B3065A  90            nop
02B3065B  90            nop
02B3065C  90            nop
02B3065D  90            nop
02B3065E  90            nop
02B3065F  90            nop
02B30660  90            nop
02B30661  4D            dec ebp
02B30662  5A            pop edx
02B30663  52            push edx
02B30664  45            inc ebp
02B30665  E8 00000000   call 2B3066A
02B3066A  5B            pop ebx
02B3066B  89DF          mov edi,ebx
02B3066D  55            push ebp
02B3066E  89E5          mov ebp,esp
02B30670  81C3 147C0000 add ebx,7C14
02B30676  FFD3          call ebx
```

We reached our shellcode! Note the NOP instructions and also the method used to get the EIP:

```
02B30665 | E8 00000000 | call 2B3066A | call $0
02B3066A | 5B          | pop ebx      |
```

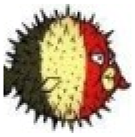
Now the shellcode will execute and perform the next stages of the infection...

[1] https://isc.sans.edu/forums/diary/Malware+Triage+with+FLOSS+API+Calls+Based+Behavior/26156

Xavier Mertens (@xme)
Senior ISC Handler - Freelance Cyber Security Consultant
PGP Key

I will be teaching next: Reverse-Engineering Malware: Malware Analysis Tools and Techniques - SANS London June 2022

Xme



687 Posts
ISC Handler
Mar 29th 2021