A Case of Vidar Infostealer - Part 1 (Unpacking)

xer0xE9 blog

March 27, 2022

Mar 27, 2022

Hi, in this post, I'll be unpacking and analyzing Vidar infostealer from my **BSides Islamabad 2021** talk. Initial stage sample comes as .xll file which is Excel Add-in file extension. It allows third party applications to add extra functionality to Excel using Excel-DNA, a tool or library that is used to write .NET Excel add-ins. In this case, xll file embeds malicious downloader dll which further drops packed Vidar infostealer executable on victim machine, investigating whole infection chain is out of scope for this post, however I'll be digging deep the dropped executable (Packed Vidar) in Part1 of this blogpost and final infostealer payload in Part2.

SHA256: 5cd0759c1e566b6e74ef3f29a49a34a08ded2dc44408fccd41b5a9845573a34c

Technical Analysis

I usually start unpacking general malware packers/loaders by looking it first into basic static analysis tools, then opening it into IDA and taking a bird's eye view of different sections for variables with possible encrypted strings, keys, imports or other global variables containing important information, checking if it has any crypto signatures identified and then start debugging it. After loading it into x64dbg, I first put breakpoint on memory allocation APIs such as LocalAlloc, GlobalAlloc, VirtualAlloc and memory protection API: VirtualProtect, and hit run button to see if any of the breakpoints hits. If yes, then it is fairly simple to unpack it and extract next stage payload, otherwise it might require in-depth static and dynamic analysis. Let's hit run button to see where it takes us next.

Shellcode Extraction

Here we go, the first breakpoint hits in this case, is **VirtualProtect**, being called on a **stack** memory region of size **0x28A** to grant it **E**xecute **R**ead **W**rite (0x40) protection, strange enough right!

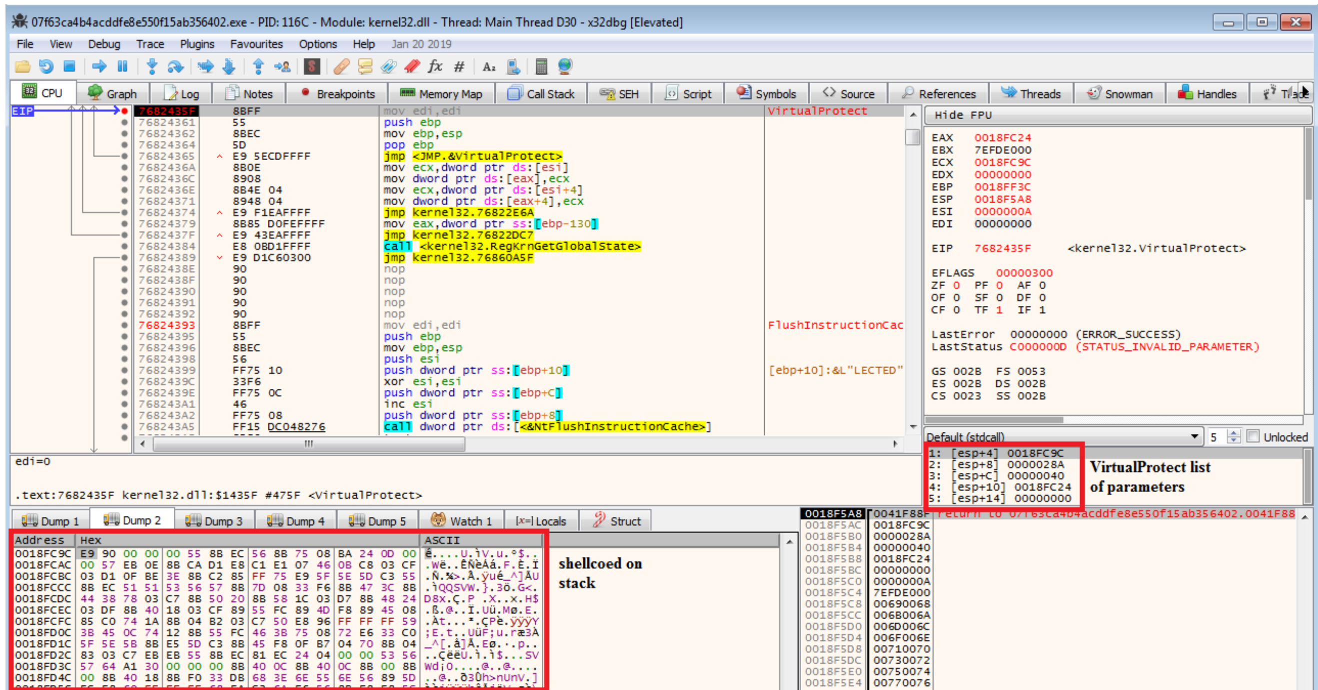


Figure1

first few opcodes **E9**, **55**, **8B** in dumped data on stack correspond to **jmp**, **push** and **mov** instructions respectively, so it can be assumed it is shellcode being pushed on stack and then granted Execute protection to later execute it, If I hit execute till return button on VirtualProtect and trace back from it into disassembler, I can see shellcode stored as **stack strings** right before VirtualProtect call and list of arguments are pushed as shown in the figure below

```
0041F7E5    C645 C8 8B        mov byte ptr ss:[ebp-38],8B
0041F7E9    C645 C9 55        mov byte ptr ss:[ebp-37],55          55:'U'
0041F7ED    C645 CA 10        mov byte ptr ss:[ebp-36],10
0041F7F1    C645 CB 85        mov byte ptr ss:[ebp-35],85
0041F7F5    C645 CC D2        mov byte ptr ss:[ebp-34],D2
0041F7F9    C645 CD 74        mov byte ptr ss:[ebp-33],74          74:'t'
0041F7FD    C645 CE 15        mov byte ptr ss:[ebp-32],15
0041F801    C645 CF 8B        mov byte ptr ss:[ebp-31],8B
0041F805    C645 D0 4D        mov byte ptr ss:[ebp-30],4D          4D:'M'
0041F809    C645 D1 08        mov byte ptr ss:[ebp-2F],8
0041F80D    C645 D2 56        mov byte ptr ss:[ebp-2E],56          56:'V'
0041F811    C645 D3 8B        mov byte ptr ss:[ebp-2D],8B
0041F815    C645 D4 75        mov byte ptr ss:[ebp-2C],75          75:'u'
0041F819    C645 D5 0C        mov byte ptr ss:[ebp-2B],C           C:'\f'
0041F81D    C645 D6 2B        mov byte ptr ss:[ebp-2A],2B          2B:'+'
0041F821    C645 D7 F1        mov byte ptr ss:[ebp-29],F1
0041F825    C645 D8 8A        mov byte ptr ss:[ebp-28],8A
0041F829    C645 D9 04        mov byte ptr ss:[ebp-27],4
0041F82D    C645 DA 0E        mov byte ptr ss:[ebp-26],E
0041F831    C645 DB 88        mov byte ptr ss:[ebp-25],88
0041F835    C645 DC 01        mov byte ptr ss:[ebp-24],1
0041F839    C645 DD 41        mov byte ptr ss:[ebp-23],41          41:'A'
0041F83D    C645 DE 83        mov byte ptr ss:[ebp-22],83
0041F841    C645 DF EA        mov byte ptr ss:[ebp-21],EA
0041F845    C645 E0 01        mov byte ptr ss:[ebp-20],1
0041F849    C645 E1 75        mov byte ptr ss:[ebp-1F],75          75:'u'
0041F84D    C645 E2 F5        mov byte ptr ss:[ebp-1E],F5
0041F851    C645 E3 5E        mov byte ptr ss:[ebp-1D],5E          5E:'^'
0041F855    C645 E4 5D        mov byte ptr ss:[ebp-1C],5D          5D:']'
0041F859    C645 E5 C3        mov byte ptr ss:[ebp-1B],C3
0041F85D    C645 E6 00        mov byte ptr ss:[ebp-1A],0
0041F861    C645 E7 00        mov byte ptr ss:[ebp-19],0
0041F865    C645 E8 00        mov byte ptr ss:[ebp-18],0
0041F869    C645 E9 00        mov byte ptr ss:[ebp-17],0
0041F86D    C745 FC 00000000  mov dword ptr ss:[ebp-4],0
0041F874    8D85 E8FCFFFF     lea eax,dword ptr ss:[ebp-318]
0041F87A    50                push eax
0041F87B    6A 40             push 40
0041F87D    68 8A020000       push 28A
0041F882    8D8D 60FDFFFF     lea ecx,dword ptr ss:[ebp-2A0]
0041F888    51                push ecx
0041F889    FF15 04F04200     call dword ptr ds:[<&VirtualProtect>]
```

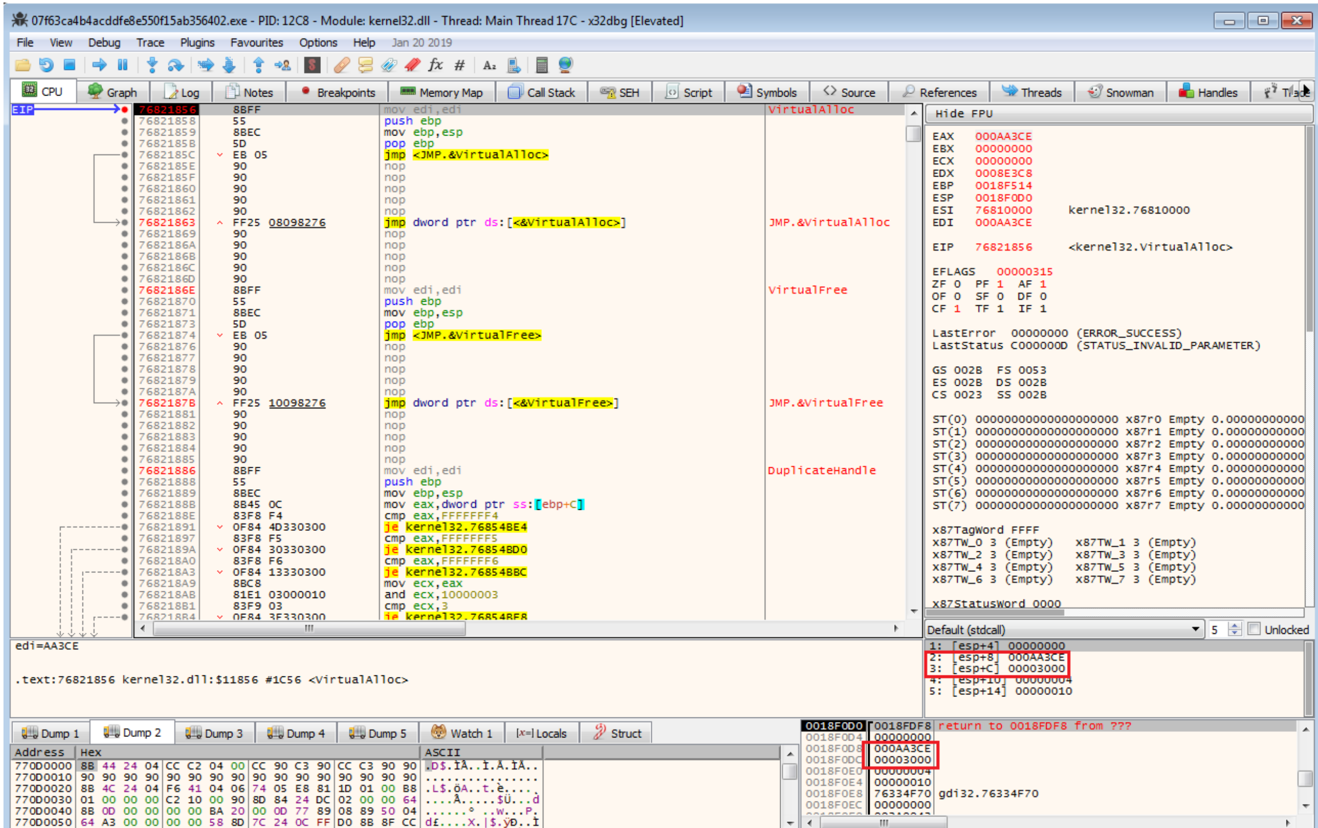
following few statements are preparing to execute shellcode on stack by retrieving a handle to a device context (DC) object and passing this handle to GrayStringA to execute shellcode from stack (ptr value in eax taken from Figure1)

```
0041F889    FF15 04F04200     call dword ptr ds:[<&VirtualProtect>]
0041F88F    6A 00             push 0
0041F891    6A 00             push 0
0041F893    6A 00             push 0
0041F895    6A 00             push 0
0041F897    6A 00             push 0
0041F899    8D95 8CF6FFFF     lea edx,dword ptr ss:[ebp-974]
0041F89F    52                push edx
0041F8A0    8D85 60FDFFFF     lea eax,dword ptr ss:[ebp-2A0]
0041F8A6    50                push eax    0x0018FC9C  ptr to shellcode on stack
0041F8A7    6A 00             push 0
0041F8A9    6A 00             push 0
0041F8AB    FF15 18F14200     call dword ptr ds:[<&GetDC>]
0041F8B1    50                push eax
0041F8B2    FF15 14F14200     call dword ptr ds:[<&GrayStringA>]
0041F8B8    8B4D 10           mov ecx,dword ptr ss:[ebp+10]        [ebp+10]:
0041F8BB    51                push ecx
```

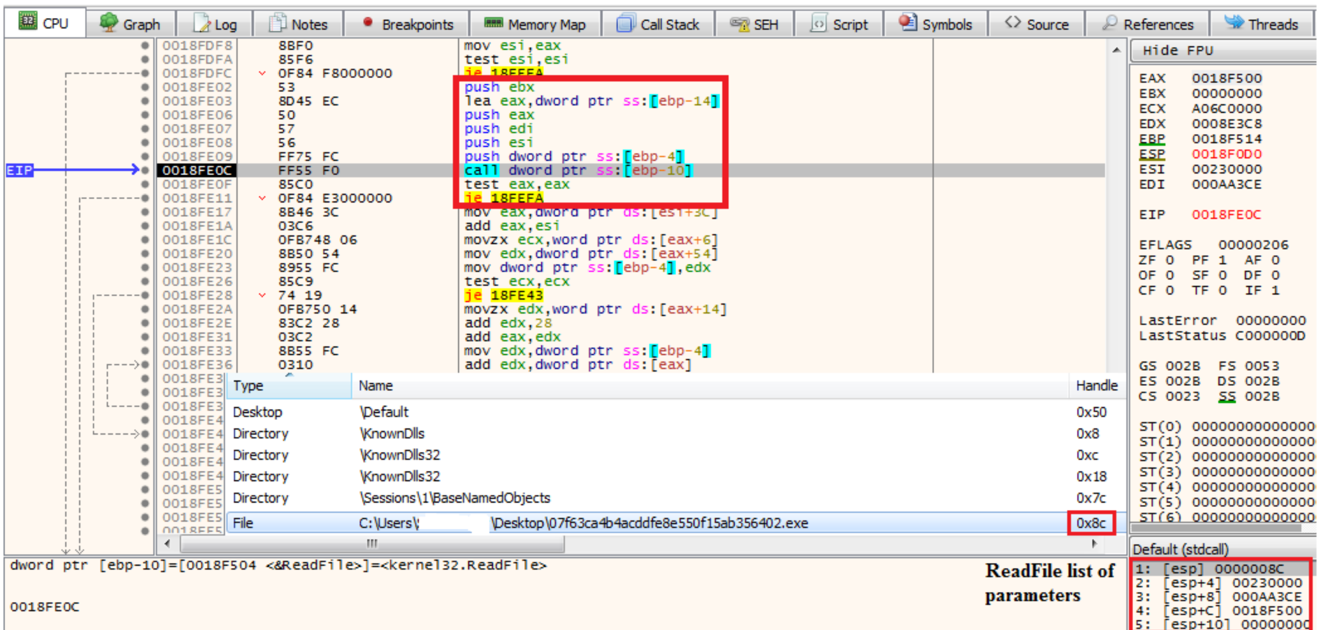
let’s now start exploring the shellcode.

Debugging shellcode to extract final payload

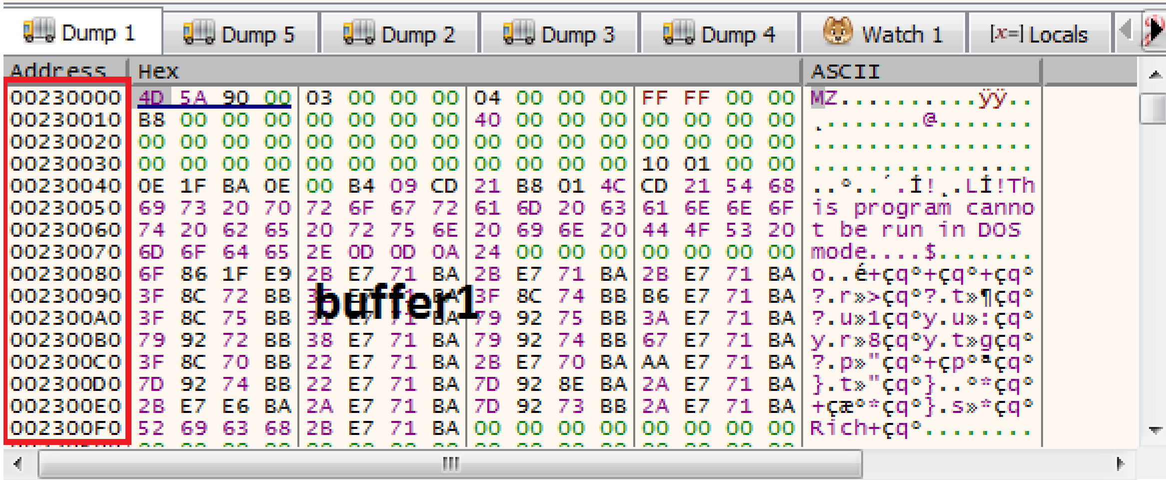
As soon as, **GrayStringA** executes, it hits on **VirtualAlloc** breakpoint set in the debugger, which is being called to reserver/commit 0xAA3CE size of memory with **MEM_COMMIT | MEM_RESERVE** (0x3000) memory allocation type



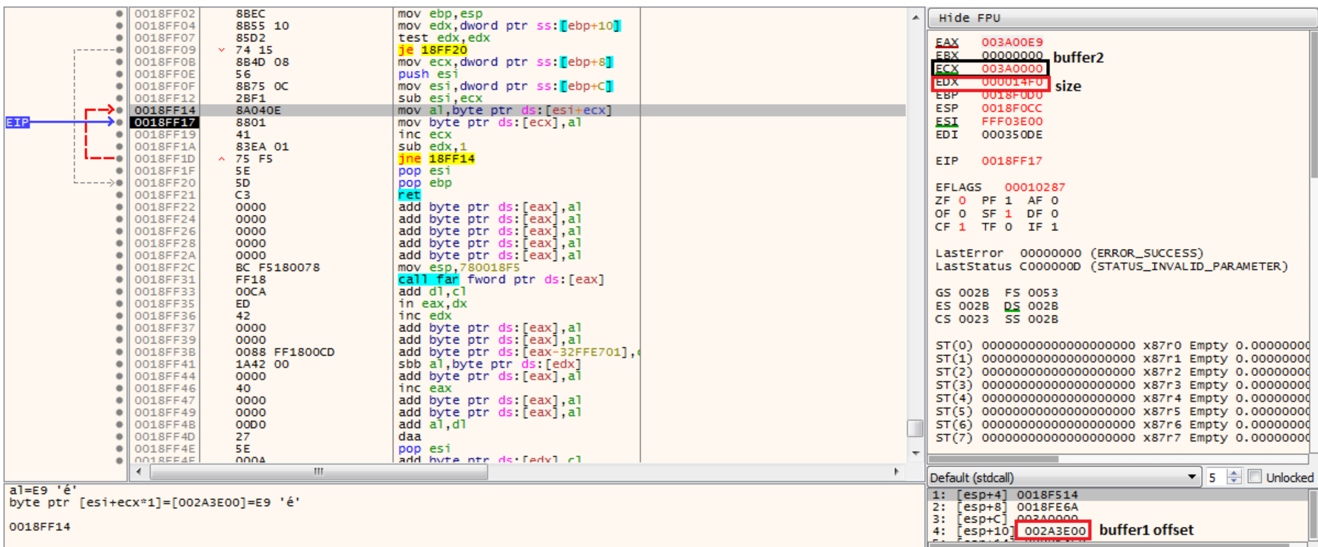
returning control from **VirtualAlloc** and stepping over one more time from ret, leads us to the shellcode, next few statements after VirtualAlloc call are pushing pointer to newly created buffer, size of the buffer and the file handle for currently loaded process on stack to call **ReadFile**



which reads 0xAA3CE bytes of data from parent process image into the buffer, let's say it **buffer1**



further execution again hits at **VirtualAlloc** breakpoint, this time allocating **0x14F0** bytes of memory, I'll now put a write breakpoint in the memory region reserved/committed by second VirtualAlloc API call to see what and how data gets dumped into second buffer, **buffer2**. Hitting Run button once more will break at instruction shown in the figure below



this loop is copying 0x14F0 bytes of data from a certain offset of buffer1 into buffer2, next few statements are agaian calling VirtualAlloc to allocate another 0x350DE bytes of memory say **buffer3**, pushing returned buffer address along with an offset from buffer1 on stack to copy 0x350DE bytes of data from buffer1 into buffer3

loop in the following figure is decrypting data copied to buffer2, next push instruction is pushing the buffer3 pointer on stack as an argument of the routine being called from buffer2 address in edx which is supposed to process buffer3 contents

figure below is showing final buffer2 decrypted contents

```
Hex                                                  ASCII
E9 0A 80 60 DC E3 11 68 0E 36 C3 44 19 E5 C5 2D     é..`Üã.h.6ÃD.åÅ-
11 E0 62 15 A7 DD 93 E9 19 65 68 76 01 46 E6 23     .àb.§Ý.é.ehv.Fæ#
7D 59 9F E1 2D 9C B1 C7 29 FD 79 14 04 0A ED 08     }Y.á-.±Ç)ýy...í.
00 22 C4 44 E6 2D 0D 28 9C 4B 1A BF 36 06 02 FE     ."ÄDæ-.(.K.¿6..þ
87 45 9E A8 E9 B7 41 6C 5B B5 8B 0B 90 3B 53 C8     .E.¨é·Al[µ...;SÈ
48 BA C0 1D 15 43 25 39 2D 7E 13 09 FC B9 1D 39     H°À..C%9-~..ü¹.9
85 A2 FB E5 BF D5 43 AF F4 E9 96 85 C8 14 AE 95     .¢ûå¿ÕC¯ôé..È.®.
2F DE 4B B5 8C 58 52 28 44 4B 4C 22 39 FA 7E 92     /ÞKµ.XR(DKL"9ú~.
7E EE 21 4E AF 82 9D 19 38 BA 0D AB DD 6E B0 6A     ~î!N¯...8º.«Ýn°j
1F 2E 8C D3 26 12 C1 2F CC F4 1E EF B6 8E 45 97     ...Ó&.Á/Ìô.ï¶.E.
D7 9C D7 67 2F C2 8D C1 7F AF 9D C5 26 8C C3 6C     ×.×g/Â.Á.¯.Å&.Ãl
AC 35 D6 AB 61 09 5D 2A 38 D5 83 70 C7 4C 96 5F     ¬5Ö«a.]*8Õ.pÇL._
26 E3 C5 EB D1 55 C2 72 75 28 62 F9 FE 67 43 18     &ãÅëÑUÂru(bùþgC.
C6 03 C3 EF A7 9D 3F 35 E1 F8 12 22 53 2C 5E 22     Æ.Ãï§.?5áø."S,^"
E7 54 92 A1 BA 1E 44 40 F6 84 10 1B 02 7F 1B 35     çT.¡º.D@ö......5
C6 F1 C1 AF C6 58 53 AE 57 40 69 DD CB 82 87 69     ÆñÁ¯ÆXS®W@iÝË..i
46 E4 63 20 0C CF F4 1D 47 89 E9 EE 51 37 6E 0F     Fäc .Ïô.G.éîQ7n.
8A 8D 62 6E 7A 1E 64 B2 33 C0 3B EF 3D 2C 63 35     ..bnz.d²3À;ï=,c5
4C 0A 33 DC A2 9C 95 5C 61 BE 62 18 0F 95 2C 72     L.3Ü¢..\a¾b...,r
75 7F D1 BC 6A 13 EB C8 52 D4 B1 B6 33 83 A0 2D     u.Ñ¼j.ëÈRÔ±¶3. -
6A 06 2B 99 22 D8 05 A2 DE A0 7D FF FE 00 B9 AE     j.+."Ø.¢Þ }ÿþ.¹®
```

encrypted buffer2

```
Hex                                                  ASCII
E9 B0 0A 00 00 55 8B EC 83 EC 40 53 56 57 83 65     é°...U.ì.ì@SVW.e
F0 00 0F 57 C0 66 0F 13 45 E0 0F 57 C0 66 0F 13     ð..WÀf..Eà.WÀf..
45 E8 83 65 F8 00 C7 45 FC 28 00 00 00 83 65 F4     Eè.eø.ÇEü(....eô
00 FF 75 0C FF 75 10 8D 45 F8 50 E8 FD 00 00 00     .ÿu.ÿu..EøPèý...
89 45 D8 89 55 DC FF 75 0C FF 75 10 8D 45 F8 50     .EØ.UÜÿu.ÿu..EøP
E8 E8 00 00 00 89 45 D0 89 55 D4 FF 75 0C FF 75     èè....EÐ.UÔÿu.ÿu
10 8D 45 F8 50 E8 D3 00 00 00 89 45 C8 89 55 CC     ..EøPèÓ....EÈ.UÌ
FF 75 0C FF 75 10 8D 45 F8 50 E8 BE 00 00 00 89     ÿu.ÿu..EøPè¾....
45 C0 89 55 C4 83 7D 10 04 76 3A 6A 08 58 6B C0     EÀ.UÄ.}..v:j.XkÀ
03 03 45 0C 99 89 45 E0 89 55 E4 8B 45 10 83 E8     ..E...Eà.Uä.E..è
04 33 C9 89 45 E8 89 4D EC 8B 45 E8 8B 4D FC 8D     .3É.Eè.Mì.Eè.Mü.
04 C1 33 D2 6A 10 59 F7 F1 8B 45 E8 03 55 FC 8D     .Á3Òj.Y÷ñ.Eè.Uü.
04 C2 89 45 FC 57 56 89 65 F4 83 E4 F0 6A 33 E8     .Â.EüWV.eô.äðj3è
00 00 00 00 83 04 24 05 CB 2B 65 FC FF 75 D8 59     ......$.Ë+eüÿuØY
FF 75 D0 5A FF 75 C8 41 58 FF 75 C0 41 59 FF 75     ÿuÐZÿuÈAXÿuÀAYÿu
E0 5F FF 75 E8 5E 85 F6 74 10 67 48 8B 0C F7 67     à_ÿuè^.öt.gH..÷g
48 89 4C F4 20 83 EE 01 75 F0 FF 75 D8 41 5A 8B     H.Lô .î.uðÿuØAZ.
45 08 0F 05 89 45 F0 03 65 FC E8 00 00 00 00 C7     E....Eð.eüè....Ç
44 24 04 23 00 00 00 83 04 24 0D CB 8B 65 F4 5E     D$.#.....$.Ë.eô^
5F 8B 45 F0 5F 5E 5B 8B E5 5D C2 0C 00 55 8B EC     _.Eð_^[.å]Â..U.ì
51 51 0F 57 C0 66 0F 13 45 F8 8B 45 08 8B 00 3B     QQ.WÀf..Eø.E...;
```

decrypted buffer2

stepping into **edx** starts executing buffer2 contents, where it seems to push stack strings for kernel32.dll first and then retrieves kernel32.dll handle by parsing PEB (Process Environment Block) structure

```
003A0AA7    8B45 F0             mov eax,dword ptr ss:[ebp-10]
003A0AAA    0FB70470            movzx eax,word ptr ds:[eax+esi*2]
003A0AAE    8B0483              mov eax,dword ptr ds:[ebx+eax*4]
003A0AB1    03C7                add eax,edi
003A0AB3  ^ EB EB               jmp 3A0AA0
003A0AB5    55                  push ebp
003A0AB6    8BEC                mov ebp,esp
003A0AB8    83EC 50             sub esp,50
003A0ABB    6A 53               push 53
003A0ABD    58                  pop eax
003A0ABE    66:8945 D8          mov word ptr ss:[ebp-28],ax
003A0AC2    6A 68               push 68
003A0AC4    58                  pop eax
003A0AC5    66:8945 DA          mov word ptr ss:[ebp-26],ax
003A0AC9    6A 6C               push 6C
003A0ACB    58                  pop eax
003A0ACC    66:8945 DC          mov word ptr ss:[ebp-24],ax
003A0AD0    6A 77               push 77
003A0AD2    58                  pop eax
003A0AD3    66:8945 DE          mov word ptr ss:[ebp-22],ax
003A0AD7    6A 61               push 61
003A0AD9    58                  pop eax
003A0ADA    66:8945 E0          mov word ptr ss:[ebp-20],ax
003A0ADE    6A 70               push 70
003A0AE0    58                  pop eax
003A0AE1    66:8945 E2          mov word ptr ss:[ebp-1E],ax
003A0AE5    6A 69               push 69
003A0AE7    58                  pop eax
003A0AE8    66:8945 E4          mov word ptr ss:[ebp-1C],ax
003A0AEC    6A 2E               push 2E
003A0AEE    58                  pop eax
003A0AEF    66:8945 E6          mov word ptr ss:[ebp-1A],ax
003A0AF3    6A 64               push 64
003A0AF5    58                  pop eax
003A0AF6    66:8945 E8          mov word ptr ss:[ebp-18],ax
003A0AFA    6A 6C               push 6C
003A0AFC    58                  pop eax
003A0AFD    66:8945 EA          mov word ptr ss:[ebp-16],ax
003A0B01    6A 6C               push 6C
003A0B03    58                  pop eax
003A0B04    66:8945 EC          mov word ptr ss:[ebp-14],ax
003A0B08    33C0                xor eax,eax
003A0B0A    66:8945 EE          mov word ptr ss:[ebp-12],ax
003A0B0E    C745 F8 CF500300    mov dword ptr ss:[ebp-8],350CF
003A0B15    E8 85FEFFFF         call <kernel32_handle>
003A0B1A    8945 FC             mov dword ptr ss:[ebp-4
```

parsing PEB structure

```
mov eax,dword ptr fs:[30]
mov eax,dword ptr ds:[eax+C]
mov eax,dword ptr ds:[eax+C]
mov eax,dword ptr ds:[eax]
mov eax,dword ptr ds:[eax]
mov eax,dword ptr ds:[eax+18]
ret
```

retrieved kernel32.dll handle is passed to next call along with another argument with constant **FF7F721A** value, a quick Google search for this constant results in some public sandbox links but not clear what is this exactly about. Let's dig into it further, stepping over this routine **0x0A4E** results in **GetModuleFileNameW** API's resolved address from Kernel32.dll stored in eax which means this routine is meant to resolve hashed APIs

```
003A0AF6    66:8945 E8          mov word ptr ss:[ebp-18],ax
003A0AFA    6A 6C               push 6C
003A0AFC    58                  pop eax
003A0AFD    66:8945 EA          mov word ptr ss:[ebp-16],ax
003A0B01    6A 6C               push 6C
003A0B03    58                  pop eax
003A0B04    66:8945 EC          mov word ptr ss:[ebp-14],ax
003A0B08    33C0                xor eax,eax
003A0B0A    66:8945 EE          mov word ptr ss:[ebp-12],ax
003A0B0E    C745 F8 CF500300    mov dword ptr ss:[ebp-8],350CF
003A0B15    E8 85FEFFFF         call <kernel32_handle>
003A0B1A    8945 FC             mov dword ptr ss:[ebp-4],eax
003A0B1D    BA 1A727FFF         mov edx,FF7F721A
003A0B22    8B4D FC             mov ecx,dword ptr ss:[ebp-4]
003A0B25    E8 24FFFFFF         call 3A0A4E
003A0B2A    8945 F0             mov dword ptr ss:[ebp-10],eax
003A0B2D    BA 78A0917F         mov edx,7F91A078
003A0B32    8B4D FC             mov ecx,dword ptr ss:[ebp-4]
003A0B35    E8 14FFFFFF         call 3A0A4E
```

```
Hide FPU

EAX     76824950     <kernel32.GetModuleFileNameW>
EBX     000014F0
ECX     BFFFD198
EDX     FF7F721A
EBP     0018F0D8
ESP     0018F088
ESI     002A3E00
EDI     000350DE

EIP     003A0B2A

EFLAGS    00000206
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1
```

similarly second call resolves **7F91A078** hash value to **ExitProcess** API, wrapper routine **0x0A4E** iterates over library exports and routine **0x097A** is computing hash against input export name parameter. Shellcode seems to be using a custom algorithm to hash API, computed hash value is retuned back into **eax** which is compared to the input hash value stored at [ebp-4], if both hash values are equal, API is resolved and its address is stored in eax

next few instructions write some junk data on stack followed by pushing pointer to buffer3 and total size of buffer3 contents (0x350C0) on stack and execute routine **0x0BE9** for decryption - this custom decryption scheme works by processing each byte from buffer3 using repetitive neg, sub, add, sar, shl, not, or and xor set of instructions with hard-coded values in multiple layers, intermediate result is stored in [ebp-1]

```
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
xor eax,74
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
sar eax,2
movzx ecx,byte ptr ss:[ebp-1]
shl ecx,6
or eax,ecx
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
neg eax
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
sub eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
not eax
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
add eax,8D
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
xor eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
sub eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
neg eax
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
xor eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
add eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
neg eax
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
add eax,F5
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
sar eax,6
movzx ecx,byte ptr ss:[ebp-1]
shl ecx,2
or eax,ecx
mov byte ptr ss:[ebp-1],al
movzx eax,byte ptr ss:[ebp-1]
sub eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],al
mov eax,dword ptr ss:[ebp+8]
add eax,dword ptr ss:[ebp-8]
mov cl,byte ptr ss:[ebp-1]
mov byte ptr ds:[eax],cl
jmp E0BE8
```

and final value overwrites the corresponding buffer3 value at [eax] offset

```
Hex                                                  ASCII
11 97 82 7F C4 52 0C 37 97 0E 0D A7 53 8E E6 CB  ....ÄR.7...§S.æË
FE 74 CE A3 EC 90 B1 C2 F1 B8 37 22 C0 74 59 99  þtÎ£ì.±Âñ¸7"ÀtY.
D2 F7 B4 FC 5D E2 13 B9 0E 1B 0C 8B 53 04 6E 4C  Ò÷´ü]â.¹....S.nL
18 7A 49 83 5D B1 85 2F 91 85 3C 3C 4C E9 6F A8  .zI.]±./..<<Léo¨
53 6C E3 B7 79 FB 27 D9 7C 29 68 3B 30 64 35 50  Slã·yû'Ù|)h;0d5P
2D B6 38 BA C9 EA 1D A0 15 FA BF 9C 19 DF 17 2A  -¶8ºÉê. .ú¿..ß.*
E0 01 6B 4E 85 AB EA 87 9B 22 EE A8 09 A3 8A FE  à.kN.«ê.."î¨.£.þ
A0 1B 68 3E 1D CE 3F ED F9 3D 57 0C 6F 22 88 31   .h>.Î?íù=W.o".1
35 43 61 AC E3 8D E4 9A 5D B1 91 84 EA DD 49 EC  5Ca¬ã.ä.]±..êÝIì
34 01 D4 14 A7 71 A8 46 74 C0 E5 5D F0 F5 19 6A  4.Ô.§q¨FtÀå]ðõ.j
32 4F FC 65 39 FD 41 E9 7D B1 A5 D5 A0 50 DB 0F  2Oüe9ýAé}±¥Õ PÛ.
1C 86 08 77 BC AC 66 DB 34 03 E2 E8 6F 8E FB 02  ...w¼¬fÛ4.âèo.û.
0D 58 D0 AA 0B CC D7 CD C4 6E D0 2D 4A E6 17 9C  .XÐª.Ì×ÍÄnÐ-Jæ..
B4 78 19 2C 0C F1 12 59 C2 EC 84 5F A3 73 EF 2C  ´x.,.ñ.YÂì._£sï,
BC 98 15 19 8B CB 99 07 F4 15 FC 52 7D F3 3D 49  ¼....Ë..ô.üR}ó=I
B2 50 AF D1 4B 27 2C 11 15 EE A8 0B C9 AD 50 F9  ²P¯ÑK',..î¨.É.Pù
BE AC 4C 7F 98 52 04 4E 81 F5 3A A7 8D 4C 15 CB  ¾¬L..R.N.õ:§.L.Ë
CD C6 CE A3 EC 90 B1 C2 66 FA 37 22 C0 C2 59 99  ÍÆÎ£ì.±Âfú7"ÀÂY.
D2 9C 9E FC 5D E2 B5 B9 0E 47 0C 8B 53 BC 6E 4C  Ò..ü]âµ¹.G..S¼nL
```

encrypted buffer3

```
Hex                                                  ASCII
4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ¸.......@.......
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 F0 00 00 00  ............ð...
0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!Th
69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
74 20 62 65 20 72 75 6E 9B 22 EE A8 09 A3 8A FE  t be run."î¨.£.þ
A0 1B 68 3E 1D CE 3F ED F9 3D 57 0C 6F 22 88 31   .h>.Î?íù=W.o".1
35 43 61 AC E3 8D E4 9A 5D B1 91 84 EA DD 49 EC  5Ca¬ã.ä.]±..êÝIì
34 01 D4 14 A7 71 A8 46 74 C0 E5 5D F0 F5 19 6A  4.Ô.§q¨FtÀå]ðõ.j
32 4F FC 65 39 FD 41 E9 7D B1 A5 D5 A0 50 DB 0F  2Oüe9ýAé}±¥Õ PÛ.
1C 86 08 77 BC AC 66 DB 34 03 E2 E8 6F 8E FB 02  ...w¼¬fÛ4.âèo.û.
0D 58 D0 AA 0B CC D7 CD C4 6E D0 2D 4A E6 17 9C  .XÐª.Ì×ÍÄnÐ-Jæ..
B4 78 19 2C 0C F1 12 59 C2 EC 84 5F A3 73 EF 2C  ´x.,.ñ.YÂì._£sï,
BC 98 15 19 8B CB 99 07 F4 15 FC 52 7D F3 3D 49  ¼....Ë..ô.üR}ó=I
B2 50 AF D1 4B 27 2C 11 15 EE A8 0B C9 AD 50 F9  ²P¯ÑK',..î¨.É.Pù
```

buffer3 in processing

once buffer3 contents are decrypted, it continues to resolve other important APIs in next routine **0x0FB6**

```
mov dword ptr ss:[ebp-C],eax
mov edx,FF7F721A   -> GetModuleFileNameW
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-78],eax
mov edx,7FE2736C   -> CreateProcessW
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-80],eax
mov edx,7FA1F993  -> GetThreadContext
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-84],eax
mov edx,7FA3EF6E   -> ReadProcessMemory
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-88],eax
mov edx,7FE1F1FB  -> CloseHandle
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-1C],eax
mov edx,FF31BF16  -> Wow64SetThreadContext
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-90],eax
mov edx,7FB6C905   -> GetCommandLineW
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-7C],eax
mov edx,7FE7F9C0   -> TerminateProcess
mov ecx,dword ptr ss:[ebp-C]
call 230A4E
mov dword ptr ss:[ebp-94],eax
```

I wrote a simple POC python script for hashing algorithm implemented by decrypted shellcode which can be found here

```
In [22]: apis = ["CreateProcessW", "ReadProcessMemory", "GetCommandLineW"]

In [23]: for api in apis:
    ...:     seed = 0x2326
    ...:     for c in api:
    ...:         shr = seed >> 1
    ...:         shl = seed << 7
    ...:         bitwiseor = shr|shl
    ...:         add_char = bitwiseor + ord(c)
    ...:         new_seed = add_char+seed
    ...:         seed = new_seed
    ...:     hash = hex(seed)
    ...:     hash = hash[:-1]
    ...:     hash =  hash[-8:]
    ...:     print hash
    ...:
7fe2736c
7fa3ef6e
7fb6c905
```

after all required APIs have been resolved, it proceeds to create a new process

```
and dword ptr ss:[ebp-70],0
mov eax,dword ptr ss:[ebp-70]
mov dword ptr ss:[ebp-8C],eax
push 103
lea eax,dword ptr ss:[ebp-7BC]
push eax
push 0
call dword ptr ss:[ebp-78]        GetModuleFileNameW
test eax,eax
jne F1168
xor eax,eax
inc eax
jmp F14E7
mov dword ptr ss:[ebp-6C],1
lea eax,dword ptr ss:[ebp-34]
push eax
lea eax,dword ptr ss:[ebp-E0]
push eax
push 0
push 0
push 8000004
push 0
push 0
push 0
call dword ptr ss:[ebp-7C]        GetCommandLineW
push eax
lea eax,dword ptr ss:[ebp-7BC]
push eax
call dword ptr ss:[ebp-80]        CreateProcessW
test eax,eax
jne F11A0
jmp F149B
```

using **CreateProcessW** in suspended mode

and then final payload is injected into newly created process using SetThreadContext API, **CONTEXT** structure for remote thread is set up with ContextFlag and required memory buffers and **SetThreadContext** API is called with current thread handle and remote thread CONTEXT structure for code injection

main process terminates right after launching this process, we can now take a dump of this process to extract final payload.

That's it for unpacking! see you soon in the next blogpost covering detailed analysis of Vidar infostealer.