

A Case of Vidar Infostealer - Part 1 (Unpacking)

[0x00-0x7f.github.io/A-Case-of-Vidar-Infostealer-Part-1-\(-Unpacking-\)/](https://0x00-0x7f.github.io/A-Case-of-Vidar-Infostealer-Part-1-(-Unpacking-)/)

0x00-0x7F 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 **Execute Read Write (0x40)** protection, strange enough right!

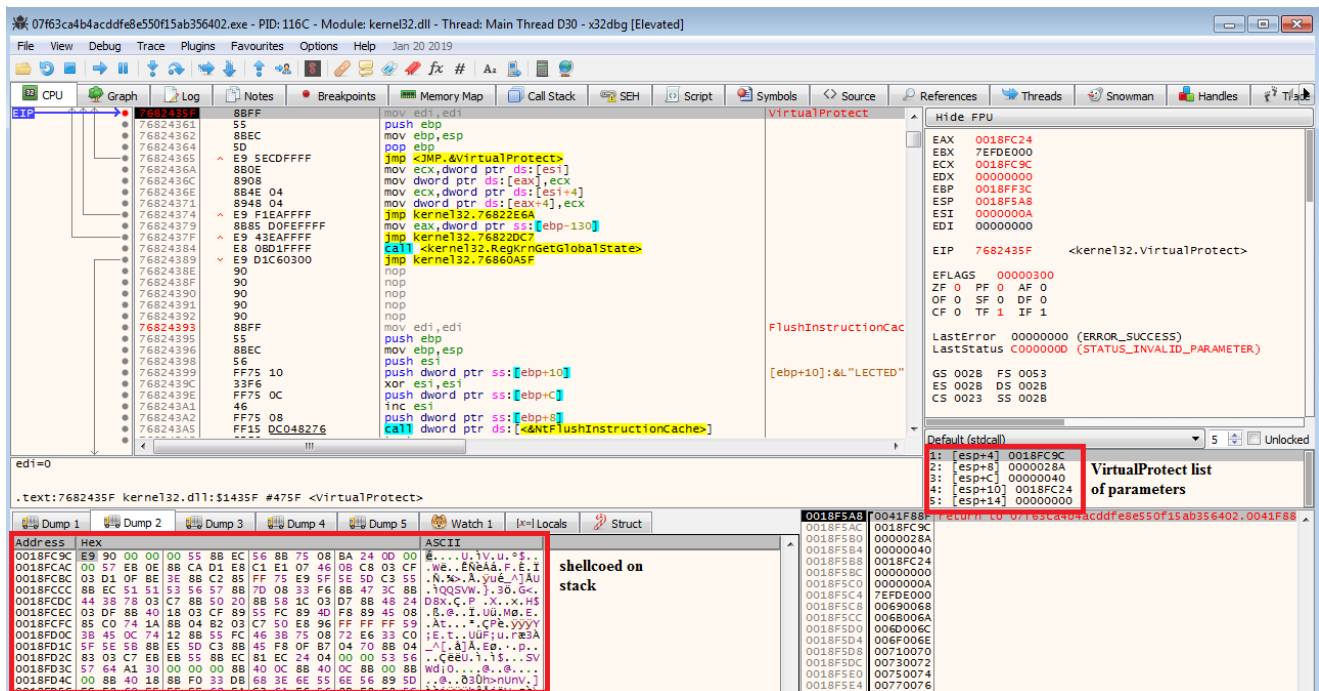


Figure1

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 88	mov byte ptr ss:[ebp-38],88	
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 88	mov byte ptr ss:[ebp-31],88	
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: '\'
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>]	

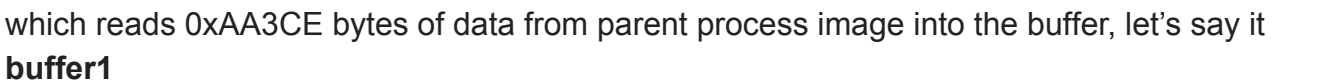
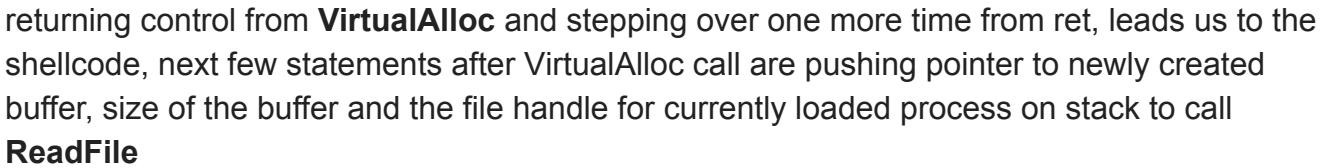
following few statements are preparing to execute shellcode on stack by retrieving a handle to a device context (DC) object and passing this handle to GrayStringA to execute shellcode from stack (ptr value in eax taken from Figure1)

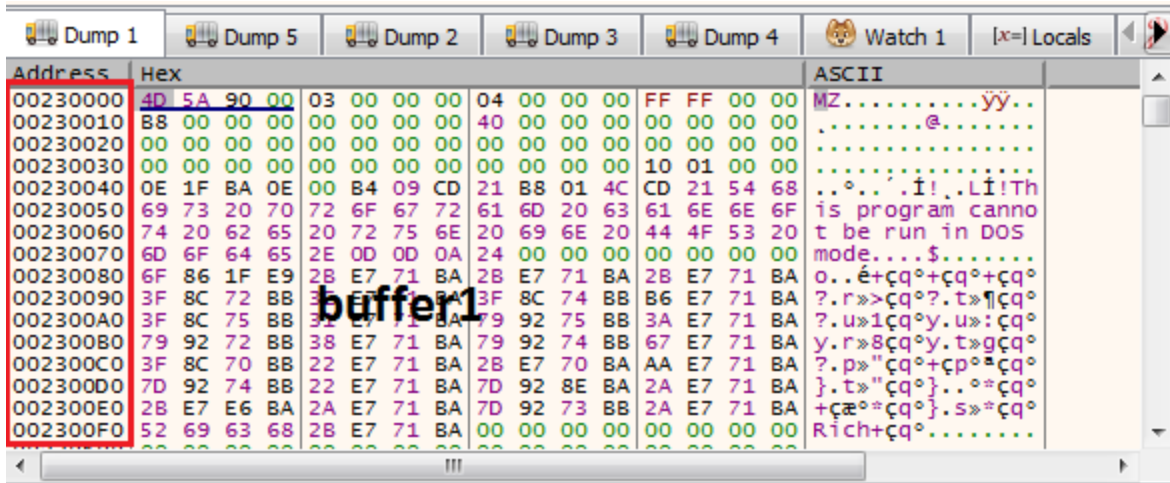
0041F889	FF15 04F04200	call dword ptr ds:[<&VirtualProtect>]	
0041F88F	6A 00	push 0	
0041F891	6A 00	push 0	
0041F893	6A 00	push 0	
0041F895	6A 00	push 0	
0041F897	6A 00	push 0	
0041F899	8D95 8CF6FFFF	lea edx,dword ptr ss:[ebp-974]	
0041F89F	52	push edx	
0041F8A0	8D85 60FDFFFF	lea eax,dword ptr ss:[ebp-2A0]	
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]:
0041F8BD	51	push ecx	

let's now start exploring the shellcode.

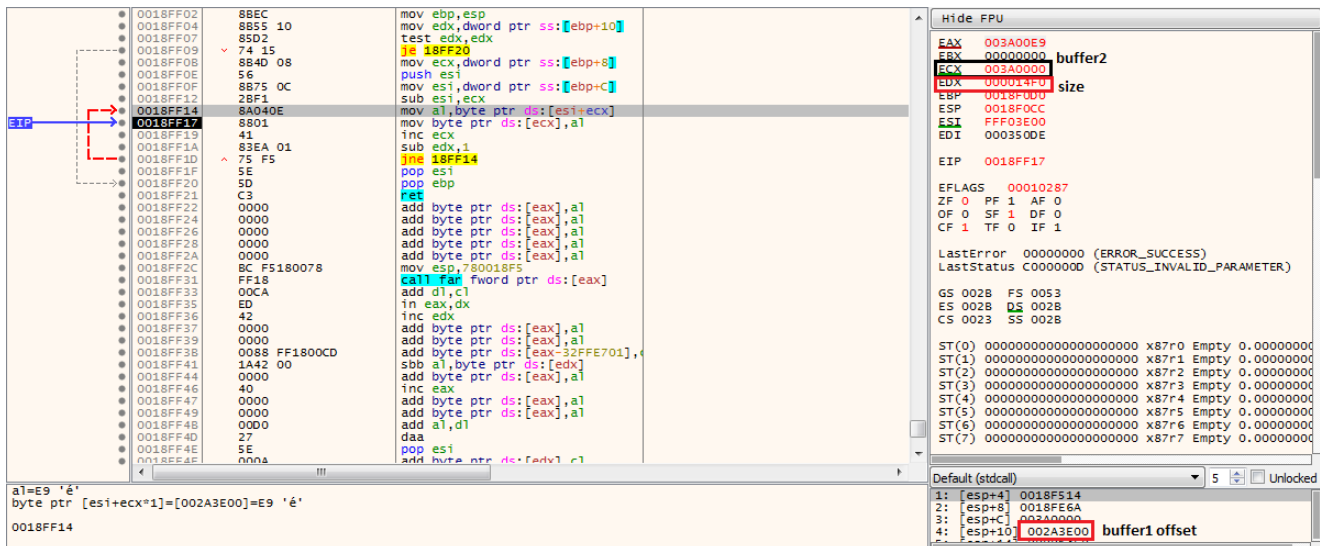
Debugging shellcode to extract final payload

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





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 again 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

Address	Disassembly	Comment
0018FE58	0375 FC	add esi,dword ptr ss:[ebp-4]
0018FE5B	68 F0140000	push 14F0
0018FE5D	58	push esi
0018FE61	50	push eax
0018FE62	8945 F0	mov dword ptr ss:[ebp-10],eax
0018FE65	E8 97000000	call 18FF01
0018FE6D	6A 40	push 40
0018FE6F	68 00300000	push 3000
0018FE74	57	push edi
0018FE75	53	push ebx
0018FE76	FF55 F8	call dword ptr ss:[ebp-8]
0018FE79	57	push edi
0018FE7A	8D8E F0140000	lea ecx,dword ptr ds:[esi+14F0]
0018FE80	8945 F4	mov dword ptr ss:[ebp-C],eax
0018FE83	51	push ecx
0018FE84	50	push eax
0018FE85	E8 77000000	call 18FF01 copy
0018FE8A	8B55 F0	mov edx,dword ptr ss:[ebp-10]
0018FE8D	83C4 0C	add esp,C
0018FE90	8A0413	mov al,byte ptr ds:[ebx+edx]
0018FE93	B1 43	mov cl,43
0018FE95	34 88	xor al,88
0018FE97	F6D0	not al
0018FE99	2AC3	sub al,b1
0018FE9B	F6D0	not al
0018FE9D	C0C0 03	rol al,3
0018FEA0	34 57	xor al,57
0018FEA2	02C3	add al,b1
0018FEA4	34 84	xor al,84
0018FEA6	02C3	add al,b1
0018FEA8	F6D0	not al
0018FEAA	02C3	add al,b1
0018FEAC	34 E3	xor al,E3
0018FEAE	C0C8 02	ror al,2
0018FEB1	2AC3	sub al,b1
0018FEB3	34 84	xor al,84
0018FEB5	2C 24	sub al,24
0018FEB7	32C3	xor al,b1
0018FEB9	2C 49	sub al,49
0018FEBB	C0C8 02	ror al,2
0018FEBE	32C3	xor al,b1
0018FEC0	D0C0	rol al,1
0018FEC2	2AC8	sub cl,a1
0018FEC4	8AC3	mov al,b1
0018FEC6	32C8	xor cl,b1
0018FEC8	D0C1	rol cl,1
0018FECB	FEC1	inc cl
0018FECF	02C8	add cl,b1
0018FED2	32C8	xor cl,b1
0018FED3	80E9 49	sub cl,49
0018FED5	F6D1	not cl
0018FED7	2AC8	sub cl,b1
0018FED9	C0C1 03	rol cl,3
0018FEDA	80F1 41	xor cl,41
0018FEDD	F6D1	not cl
0018FEDF	2AC8	sub cl,b1
0018FEE1	80F1 67	xor cl,67
0018FEE4	2AC1	sub al,c1
0018FEE6	32C3	xor al,b1
0018FEE8	880413	mov byte ptr ds:[ebx+edx],al
0018FEEB	43	inc ebx
0018FEEC	81F8 F0140000	cmp ebx,F0140000
0018FEE7	72 8C	jnb 18FF00
0018FEE9	FF75 F4	push dword ptr ds:[ebp-4]
0018FEEF	FD2	call edx
0018FEF0	EA	ret

VirtualAlloc

43: 'C'

loop decrypting buffer2 contents

execute buffer2

Hide FPU

EAX 003B0000 buffer3

EBX 00000000

ECX 002A52F0 buffer1 offset

EDX 000E3C8

EBP 0018F514

ESP 0018F008

ESI 002A5F00

EDI 0003500E size of data being copied

EIP 0018FE85

EFLAGS 00000246

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 0028 FS 0053

ES 0028 DS 0028

CS 0023 SS 0028

ST(0) 000000000000000000000000 x87r0 Empty 0.00000000

ST(1) 000000000000000000000000 x87r1 Empty 0.00000000

ST(2) 000000000000000000000000 x87r2 Empty 0.00000000

ST(3) 000000000000000000000000 x87r3 Empty 0.00000000

ST(4) 000000000000000000000000 x87r4 Empty 0.00000000

ST(5) 000000000000000000000000 x87r5 Empty 0.00000000

ST(6) 000000000000000000000000 x87r6 Empty 0.00000000

ST(7) 000000000000000000000000 x87r7 Empty 0.00000000

Default (stdcall) 5 Unlocked

1: [esp] 003B0000

2: [esp+4] 002A52F0

3: [esp+8] 0003500E

4: [esp+C] 00000010

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

Address	Disassembly	Comment
0018FE85	E8 77000000	call 18FF01
0018FE8A	8B55 F0	mov edx,dword ptr ss:[ebp-10]
0018FE8D	83C4 0C	add esp,C
0018FE90	8A0413	mov al,byte ptr ds:[ebx+edx]
0018FE93	B1 43	mov cl,43
0018FE95	34 88	xor al,88
0018FE97	F6D0	not al
0018FE99	2AC3	sub al,b1
0018FE9B	F6D0	not al
0018FE9D	C0C0 03	rol al,3
0018FEA0	34 57	xor al,57
0018FEA2	02C3	add al,b1
0018FEA4	34 84	xor al,84
0018FEA6	02C3	add al,b1
0018FEA8	F6D0	not al
0018FEAA	02C3	add al,b1
0018FEAC	34 E3	xor al,E3
0018FEAE	C0C8 02	ror al,2
0018FEB1	2AC3	sub al,b1
0018FEB3	34 84	xor al,84
0018FEB5	2C 24	sub al,24
0018FEB7	32C3	xor al,b1
0018FEB9	2C 49	sub al,49
0018FEBB	C0C8 02	ror al,2
0018FEBE	32C3	xor al,b1
0018FEC0	D0C0	rol al,1
0018FEC2	2AC8	sub cl,a1
0018FEC4	8AC3	mov al,b1
0018FEC6	32C8	xor cl,b1
0018FEC8	D0C1	rol cl,1
0018FECB	FEC1	inc cl
0018FECF	02C8	add cl,b1
0018FED2	32C8	xor cl,b1
0018FED3	80E9 49	sub cl,49
0018FED5	F6D1	not cl
0018FED7	2AC8	sub cl,b1
0018FED9	C0C1 03	rol cl,3
0018FEDA	80F1 41	xor cl,41
0018FEDD	F6D1	not cl
0018FEDF	2AC8	sub cl,b1
0018FEE1	80F1 67	xor cl,67
0018FEE4	2AC1	sub al,c1
0018FEE6	32C3	xor al,b1
0018FEE8	880413	mov byte ptr ds:[ebx+edx],al
0018FEEB	43	inc ebx
0018FEEC	81F8 F0140000	cmp ebx,F0140000
0018FEE7	72 8C	jnb 18FF00
0018FEE9	FF75 F4	push dword ptr ds:[ebp-4]
0018FEEF	FD2	call edx
0018FEE0	EA	ret

loop decrypting buffer2 contents

execute buffer2

Hide FPU

EAX 003B000F

EBX 00000012

ECX 003E50E5

EDX 003A0000 buffer2

ESP 0018F0E4

ESI 002A5F00

EDI 0003500E size

EIP 0018FEE8

EFLAGS 00000304

ZF 0 PF 1 AF 0

OF 0 SF 0 DF 0

CF 0 TF 1 IF 1

LastError 00000000 (ERROR_SUCCESS)

LastStatus C000000D (STATUS_INVALID_PARAMETER)

GS 0028 FS 0053

ES 0028 DS 0028

CS 0023 SS 0028

ST(0) 000000000000000000000000 x87r0 Empty 0.00000000

ST(1) 000000000000000000000000 x87r1 Empty 0.00000000

ST(2) 000000000000000000000000 x87r2 Empty 0.00000000

ST(3) 000000000000000000000000 x87r3 Empty 0.00000000

ST(4) 000000000000000000000000 x87r4 Empty 0.00000000

ST(5) 000000000000000000000000 x87r5 Empty 0.00000000

ST(6) 000000000000000000000000 x87r6 Empty 0.00000000

ST(7) 000000000000000000000000 x87r7 Empty 0.00000000

X87tagWord FFFF

X87TW_0 3 (Empty) X87TW_1 3 (Empty)

X87TW_2 3 (Empty) X87TW_3 3 (Empty)

X87TW_4 3 (Empty) X87TW_5 3 (Empty)

X87TW_6 3 (Empty) X87TW_7 3 (Empty)

X87StatusWord 0000

X87SW_B 0 X87SW_C3 0 X87SW_C2 0

X87SW_C1 0 X87SW_C0 0 X87SW_E5 0

Default (stdcall) 5 Unlocked

figure below is showing final buffer2 decrypted contents

Hex	ASCII
E9 0A 80 60 DC E3 11 68 OE 36 C3 44 19 E5 C5 2D	é...Uä.h.6AD.äA-
11 E0 62 15 A7 DD 93 E9 19 65 68 76 01 46 E6 23	.ab.šý.é.ehv.Fæ#
7D 59 9F E1 2D 9C B1 C7 29 FD 79 14 04 0A ED 08	}Y.ä-.±Ç)ýý...í.
00 22 C4 44 E6 2D 0D 28 9C 4B 1A BF 36 06 02 FE	."ADæ-.(.K.¿6..p
87 45 9E A8 E9 B7 41 6C 5B B5 8B 0B 90 38 53 C8	.E."é.AI[µ...;SÈ
48 BA C0 1D 15 43 25 39 2D 7E 13 09 FC 89 1D 39	H°A..C%9-~..ü'.9
85 A2 FB E5 BF D5 43 AF F4 E9 96 85 C8 14 AE 95	.cûà¿ÖC-ôé..È.º.
2F DE 4B B5 8C 58 52 28 44 4B 4C 22 39 FA 7E 92	/pKµ.XR(DKL"9ú~.
7E EE 21 4E AF 82 9D 19 38 BA 0D AB DD 6E B0 6A	~î!N"...8º.«Yn°j
1F 2E 8C D3 26 12 C1 2F CC F4 1E EF B6 8E 45 97	...Ó&.Á/IÖ.î¶.E.
D7 9C D7 67 2F C2 8D C1 7F AF 9D C5 26 8C C3 6C	x.xg/Ä.Ä..Ä&.ÄI
AC 35 D6 AB 61 09 5D 2A 38 D5 83 70 C7 4C 96 5F	~50«a.]*80.pçL._
26 E3 C5 EB D1 55 C2 72 75 28 62 F9 FE 67 43 18	&äÄeNUÄru(búpçC.
C6 03 C3 EF A7 9D 3F 35 E1 F8 12 22 53 2C 5E 22	Æ.Äiç.75äø."S,^"
E7 54 92 A1 BA 1E 44 40 F6 84 10 1B 02 7F 1B 35	çT.i°.D@ö.....5
C6 F1 C1 AF C6 58 53 AE 57 40 69 DD CB 82 87 69	ÄñA ÄX5°w@iYÈ..i
46 E4 63 20 0C CF F4 1D 47 89 E9 EE 51 37 6E 0F	Fäc .IÖ.G.éiQ7n.
8A 8D 62 6E 7A 1E 64 B2 33 C0 3B EF 3D 2C 63 35	..bnz.d*3A;î=,c5
4C 0A 33 DC A2 9C 95 5C 61 BE 62 18 0F 95 2C 72	L.3Üc..\\a%b...r
75 7F D1 BC 6A 13 EB C8 52 D4 B1 B6 33 83 A0 2D	u.N4j.ëERO±¶3. -
6A 06 2B 99 22 D8 05 A2 DE A0 7D FF FE 00 B9 AE	i.+. "ø.çp }yb.'ø

encrypted buffer2

Hex	ASCII
E9 B0 0A 00 00 55 8B EC 83 EC 40 53 56 57 83 65	é°...U.î.î@SVW.e
F0 00 0F 57 C0 66 0F 13 45 E0 0F 57 C0 66 0F 13	ð..wÄf...Eä.wÄf..
45 E8 83 65 F8 00 C7 45 FC 28 00 00 00 83 65 F4	Eè.eø.ÇEü(....eð
00 FF 75 0C FF 75 10 8D 45 F8 50 E8 FD 00 00 00	.ýü.ýü..EøPéý...
89 45 D8 89 55 DC FF 75 0C FF 75 10 8D 45 F8 50	.Eø.UÜýü.ýü..EøP
E8 E8 00 00 00 89 45 D0 89 55 D4 FF 75 0C FF 75	èè....ED.Uöýü.ýü
10 8D 45 F8 50 E8 D3 00 00 00 89 45 C8 89 55 CC	..EøPéö....EÈ.UI
FF 75 0C FF 75 10 8D 45 F8 50 E8 BE 00 00 00 89	ýü.ýü..EøPè%....
45 C0 89 55 C4 83 7D 10 04 76 3A 6A 08 58 6B C0	EÄ.UÄ.}.v:j.XkÄ
03 03 45 0C 99 89 45 E0 89 55 E4 8B 45 10 83 E8	..E...Eä.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	.Ä3Öj.Y÷ñ.Eè.Uü.
04 C2 89 45 FC 57 56 89 65 F4 83 E4 F0 6A 33 E8	.Ä.Eüwv.eð.äðj3è
00 00 00 00 83 04 24 05 CB 2B 65 FC FF 75 D8 59\$.È+eüýüøY
FF 75 D0 5A FF 75 C8 41 58 FF 75 C0 41 59 FF 75	ýüðZýüEAXýüAAYýü
E0 5F FF 75 E8 5E 85 F6 74 10 67 48 8B 0C F7 67	ä_ýüe^..öt.gH..÷g
48 89 4C F4 20 83 EE 01 75 F0 FF 75 D8 41 5A 8B	H.Lö .î.uöýüøAZ.
45 08 0F 05 89 45 F0 03 65 FC E8 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\$.#.....\$.È.eð^
5F 8B 45 F0 5F 5E 5B 8B E5 5D C2 0C 00 55 8B EC	_.Eð_^[.ä]Ä..U.î
51 51 0F 57 C0 66 0F 13 45 F8 8B 45 08 8B 00 3B	00.wÄf...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	880483	mov eax,dword ptr ds:[ebx+eax*4]
003A0AB1	03C7	add eax,edi
003A0AB3	EB EB	jmp 3A0AA0
003A0AB5	55	push ebp
003A0AB6	88EC	mov ebp,esp
003A0AB8	83EC 50	sub esp,50
003A0AB8	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],eax

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 24FFFFF	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 14FFFFF	call 3A0A4E

Hide FPU

EAX	76824950	<kernel32.GetModuleFileNameW>			
EBX	000014F0				
ECX	8FFFD198				
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 returned 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**

The screenshot displays a debugger interface with three main panels:

- Assembly Window:** Shows assembly instructions. A loop is highlighted with a red box, labeled "loop iterating over library's list of exports". The instructions include:
 - `mov ecx, dword ptr ds:[edx+esi*4]`
 - `call 3A097A`
 - `inc esi`
 - `cmp esi, dword ptr ss:[ebp-8]`
 - `jnb 3A0A86`
 - `mov eax, dword ptr ss:[ebp-10]`
 - `movzx ebx, word ptr ds:[ebx+esi*4]`
 - `add eax, edi`
 - `jnb 3A0A86`
- Registers Window:** Shows the state of registers. **EAX** is highlighted and contains the value **768CFA28**, labeled "computed hash value". Other registers like ECX, EDI, and EIP are also visible.
- Dump Window:** Shows the contents of memory. The address **[ebp-4]** is highlighted, showing a sequence of bytes: **72 7F FF 00 00 00 00 00 00 00 00 00 00 00 00 00 00**.

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 current 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],a1
movzx eax,byte ptr ss:[ebp-1]
xor eax,74
mov byte ptr ss:[ebp-1],a1
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],a1
movzx eax,byte ptr ss:[ebp-1]
neg eax
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
sub eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
not eax
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
add eax,80
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
xor eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
sub eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
neg eax
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
xor eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
add eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
neg eax
mov byte ptr ss:[ebp-1],a1
movzx eax,byte ptr ss:[ebp-1]
add eax,F5
mov byte ptr ss:[ebp-1],a1
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],a1
movzx eax,byte ptr ss:[ebp-1]
sub eax,dword ptr ss:[ebp-8]
mov byte ptr ss:[ebp-1],a1
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
imn E0RER

```

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	...AR.7...\$S.æE
FE 74 CE A3 EC 90 B1 C2 F1 88 37 22 C0 74 59 99	ptifî.±Añ.7"ÀtY.
D2 F7 84 FC 5D E2 13 89 0E 18 0C 8B 53 04 6E 4C	O="ü]ä.'...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	Slä.yû"U]h;Od5P
2D B6 38 BA C9 EA 1D A0 15 FA 8F 9C 19 DF 17 2A	-ŋ8°ÊÊ. .û. .ß.*
E0 01 68 4E 85 AB EA 87 9B 22 EE A8 09 A3 8A FE	à.kN.«è. "í .f.p
A0 1B 68 3E 1D CE 3F ED F9 3D 57 0C 6F 22 88 31	.h>.i?íu=w.o".1
35 43 61 AC E3 8D E4 9A 5D B1 91 84 EA DD 49 EC	5Ca-ä.ä.]±.ëYIì
34 01 D4 14 A7 71 A8 46 74 C0 E5 5D F0 F5 19 6A	4.ô.sq FtAâ]ôö.j
32 4F FC 65 39 FD 41 E9 7D B1 A5 D5 A0 50 DB 0F	20üe9yAé]±¥0 P0.
1C 86 08 77 BC AC 66 DB 34 03 E2 E8 6F 8E FB 02	...w%-f04.âeo.û.
0D 58 D0 AA 08 CC D7 CD C4 6E D0 2D 4A E6 17 9C	.XD*.îxîAnD-Jæ..
B4 78 19 2C 0C F1 12 59 C2 EC 84 5F A3 73 EF 2C	'x.,.ñ.YÄi. _fsî,
BC 98 15 19 8B CB 99 07 F4 15 FC 52 7D F3 3D 49	%....É...ô.ÜR}ó=I
B2 50 AF D1 4B 27 2C 11 15 EE A8 0B C9 AD 50 F9	*P`NK',...î .É.Pü
BE AC 4C 7F 98 52 04 4E 81 F5 3A A7 8D 4C 15 CB	%-L..R.N.ô:\$.L.É
CD C6 CE A3 EC 90 B1 C2 66 FA 37 22 C0 C2 59 99	iÄîî.±Afú7"ÀAY.
D2 9C 9E FC 5D E2 B5 B9 0E 47 0C 8B 53 BC 6E 4C	ô..ü]äµ'.G..S%NL

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 00e.....
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00ô...
0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..°...î!..Lî!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."í .f.p
A0 1B 68 3E 1D CE 3F ED F9 3D 57 0C 6F 22 88 31	.h>.i?íu=w.o".1
35 43 61 AC E3 8D E4 9A 5D B1 91 84 EA DD 49 EC	5Ca-ä.ä.]±.ëYIì
34 01 D4 14 A7 71 A8 46 74 C0 E5 5D F0 F5 19 6A	4.ô.sq FtAâ]ôö.j
32 4F FC 65 39 FD 41 E9 7D B1 A5 D5 A0 50 DB 0F	20üe9yAé]±¥0 P0.
1C 86 08 77 BC AC 66 DB 34 03 E2 E8 6F 8E FB 02	...w%-f04.âeo.û.
0D 58 D0 AA 08 CC D7 CD C4 6E D0 2D 4A E6 17 9C	.XD*.îxîAnD-Jæ..
B4 78 19 2C 0C F1 12 59 C2 EC 84 5F A3 73 EF 2C	'x.,.ñ.YÄi. _fsî,
BC 98 15 19 8B CB 99 07 F4 15 FC 52 7D F3 3D 49	%....É...ô.ÜR}ó=I
B2 50 AF D1 4B 27 2C 11 15 EE A8 0B C9 AD 50 F9	*P`NK',...î .É.Pü
BE AC 4C 7F 98 52 04 4E 81 F5 3A A7 8D 4C 15 CB	%-L..R.N.ô:\$.L.É

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 F1498
```

using **CreateProcessW** in suspended mode

x32dbg.exe	0.28	65,216 K	2348 x64dbg
07f63ca4b4acddfe8e550f15ab356402.exe	0.02	1,844 K	3348
07f63ca4b4acddfe8e550f15ab356402.exe	Susp...	372 K	4684

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

The screenshot shows the Windows Memory Dump tool (07f63ca4b4acddfe8e550f15ab356402.exe (4684) Properties) with the Memory tab selected. The memory dump shows a list of memory regions with their base addresses, types, sizes, and permissions. A red box highlights the memory region at 0x400000, which is mapped and has RWX permissions. Below this, a red box highlights the memory region at 0x400000, which is mapped and has RWX permissions. The memory dump also shows a list of strings, including 'base64 encrypted strings'.

Base address	Type	Size	Protect...	Use	Total WS	Private WS	Shareable WS	Shared WS
0x10000	Private	128 kB	RW		12 kB	12 kB		
0x30000	Private	12 kB	RW		12 kB	12 kB		
0x40000	Image	4 kB	WCX	C:\Windows\System32\apisetschem...	4 kB		4 kB	4 kB
0x50000	Mapped	16 kB	R		16 kB		16 kB	16 kB
0x60000	Mapped	4 kB	R		4 kB		4 kB	4 kB
0x70000	Private	4 kB	RW		4 kB	4 kB		
0xa0000	Private	256 kB	RW	Stack (thread 4168)	8 kB	8 kB		
0x130000	Private	1,024 kB	RW	Stack 32-bit (thread 4168)	4 kB	4 kB		
0x400000	Mapped	224 kB	RWX		32 kB		32 kB	32 kB
0x400000	Mapped: Com...	224 kB	RWX		32 kB		32 kB	32 kB

07f63ca4b4acddfe8e550f15ab356402.exe (4684) (0x400000 - 0x438000)

```

0002b630 37 45 63 3d 00 00 00 00 66 37 64 4d 39 55 32 49 7Ec=...f7dM9U2I
0002b640 51 53 48 2b 6f 45 66 2f 7a 6f 67 49 6b 62 4d 3d QSH+oE/zoGIkbM=
0002b650 00 00 00 00 66 37 63 38 39 55 33 34 51 53 47 42 ...f7c89U34QSGb
0002b660 6f 45 65 61 7a 6f 68 74 6b 62 4d 3d 00 00 00 00 oEazohtkbM=...
0002b670 44 34 63 67 39 55 30 3d 00 00 00 00 48 70 6f 77 D4cg9U0=...Hpow
0002b680 67 47 58 6d 50 51 3d 3d 00 00 00 00 66 37 63 62 qGXmPQ=...f7cb
0002b690 39 55 30 3d 00 00 00 00 43 5a 77 6c 68 48 37 6d 9U0=...CZw1hH7m
0002b6a0 4e 6b 57 43 32 57 36 73 69 4a 35 64 78 37 69 4f NkWC2W6siJ5dx7iO
0002b6b0 7a 6f 7a 73 51 70 78 69 63 4b 52 41 36 32 49 4e zozsQpxicKRA62IN
0002b6c0 77 54 58 4f 34 55 38 72 47 46 65 2b 2f 78 63 78 wTXO4U8rGF+/xcx
0002b6d0 74 35 57 36 59 45 70 39 74 56 59 37 38 56 31 2f t5W6YEp9tVY78V1/
0002b6e0 77 67 3d 3d 00 00 00 00 48 72 6f 51 6f 45 58 47 wg=...HroQoEXG
0002b6f0 48 51 75 2f 36 45 59 3d 00 00 00 00 48 72 6f 51 HQu/6EY=...HroQ
0002b700 6f 45 58 47 48 52 4f 37 39 31 43 73 68 49 49 3d oEXGHR0791CshII=
0002b710 00 00 00 00 66 36 42 47 74 41 3d 3d 00 00 00 00 ...f6BGtA==...
0002b720 69 00 6d 00 61 00 67 00 65 00 2f 00 6a 00 70 00 i.m.a.g.e./j.p.
0002b730 65 00 67 00 00 00 00 00 73 00 63 00 72 00 65 00 e.g.....s.c.r.e.
0002b740 65 00 6e 00 73 00 68 00 6f 00 74 00 2e 00 6a 00 e.n.s.h.o.t...j.
0002b750 70 00 67 00 00 00 00 00 64 62 42 44 70 45 6a 55 p.g....dbBDpEjU
0002b760 44 79 36 33 36 55 2f 6c 78 4a 78 62 30 50 66 4e Dy636U/lxJxbOPFN
0002b770 33 76 43 57 4e 5a 42 2b 64 62 68 53 75 42 73 69 3vCWNZB+dbbSuBsi
0002b780 6f 6d 61 63 77 57 35 6c 51 31 4c 37 6f 6a 56 34 omacwW5lQ1L7ojV4
0002b790 2f 59 69 36 59 44 55 7a 2b 68 67 74 2f 56 56 6e /Yi6YDUz+hgt/VVn
0002b7a0 00 00 00 00 4f 62 34 48 2f 6b 7a 66 41 51 3d 3d ...Ob4H/kzfAQ==
0002b7b0 00 00 00 00 61 35 45 6d 6c 68 6e 6d 55 58 4b 63 ...a5Em1hnmUXKc
0002b7c0 77 42 4c 30 32 36 70 32 67 4f 48 66 2b 77 3d 3d wBL026p2gOHI+w==
0002b7d0 00 00 00 00 47 62 77 4e 70 45 7a 4a 45 47 69 61 ...GbwNpEzJEGia
0002b7e0 37 46 43 31 68 4a 30 62 77 4a 36 48 31 4f 71 51 75C1b70h+75H1C0

```

base64 encrypted strings

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.