# A Case of Vidar Infostealer - Part 1 (Unpacking)

Published: 2022-03-27 · Archived: 2026-04-05 19:04:55 UTC

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!

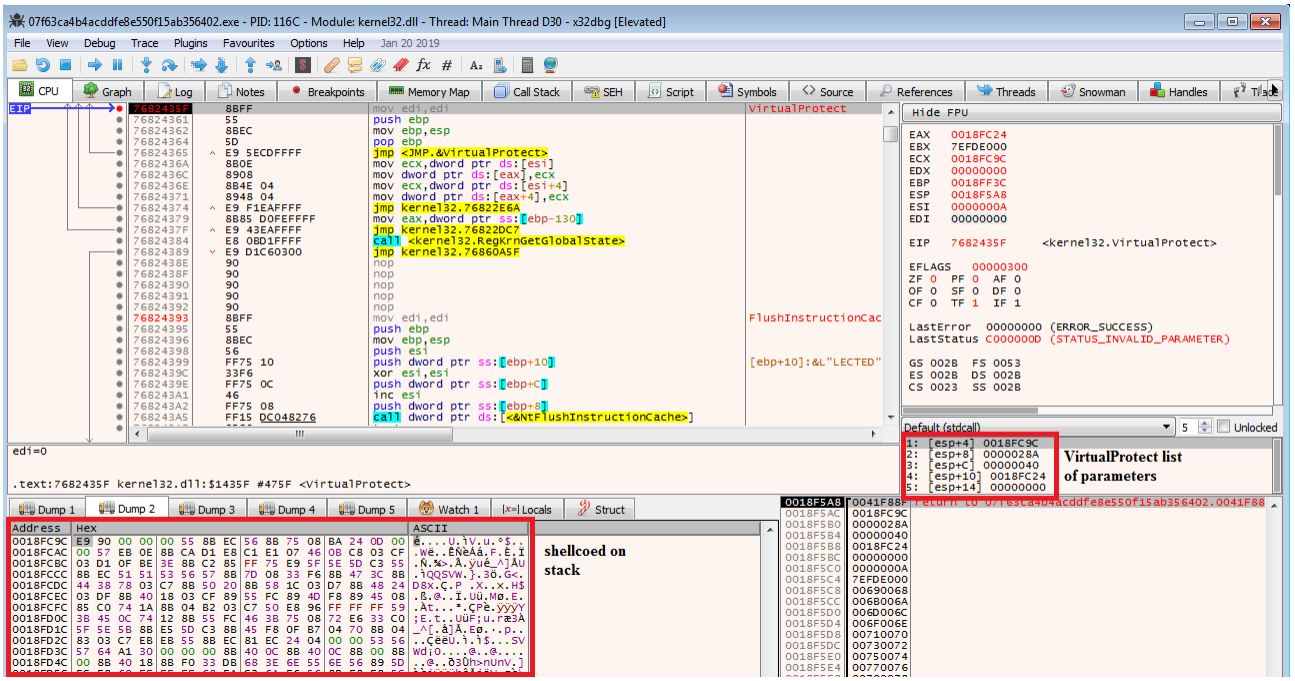
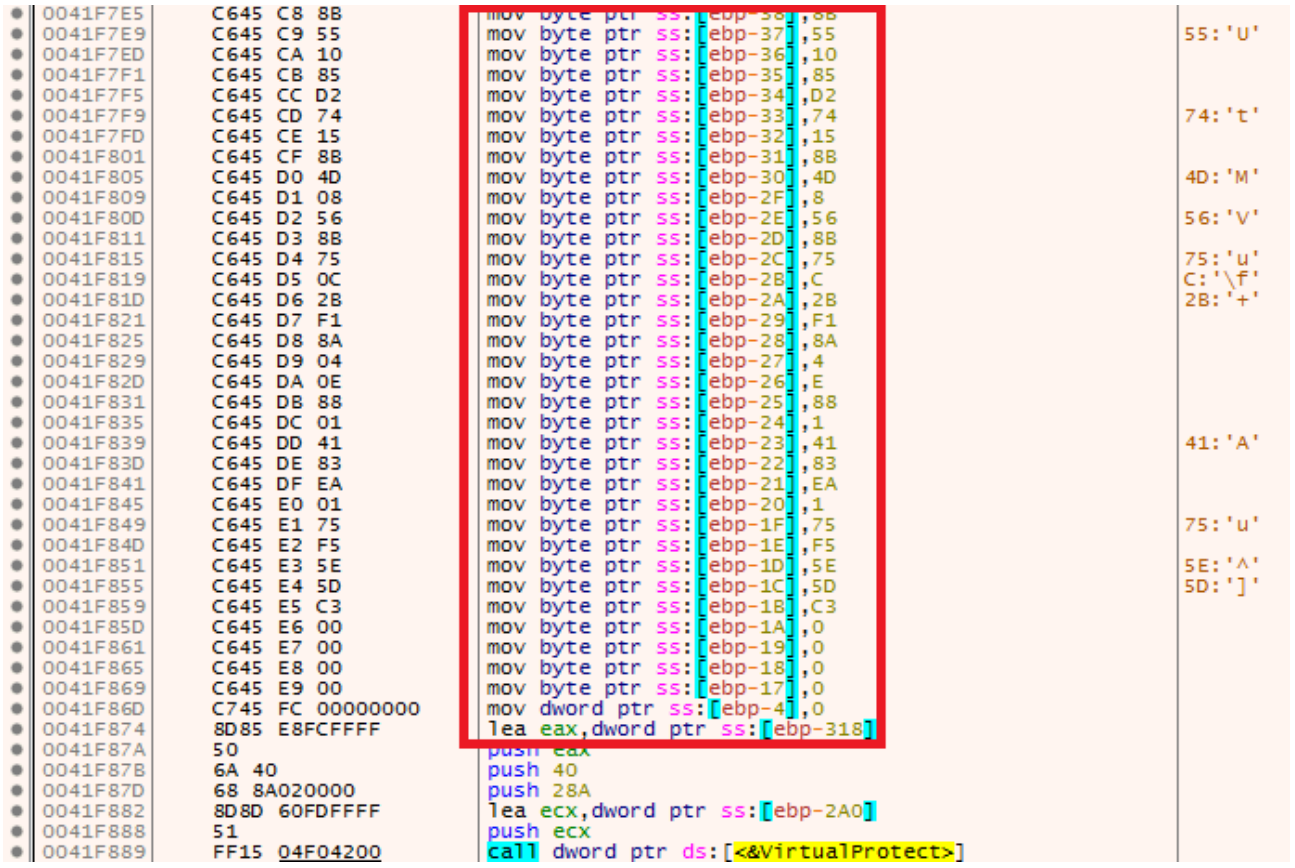
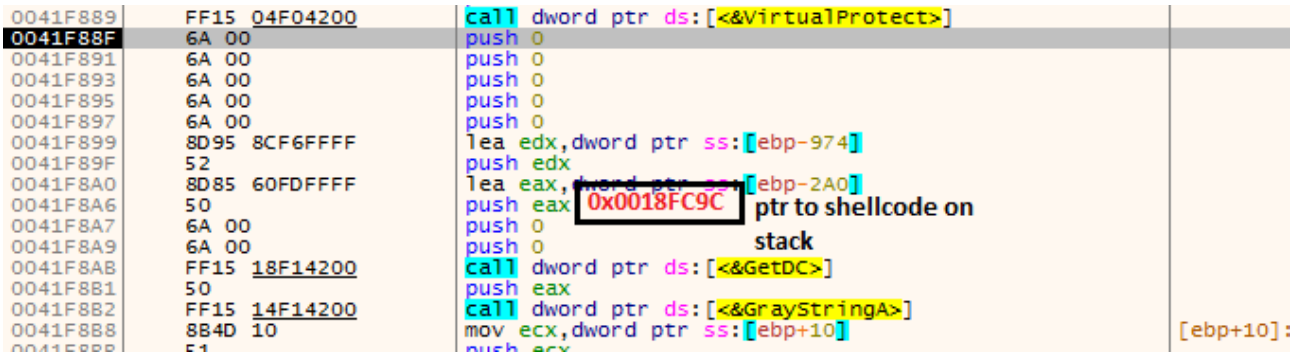



*Figure1*

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



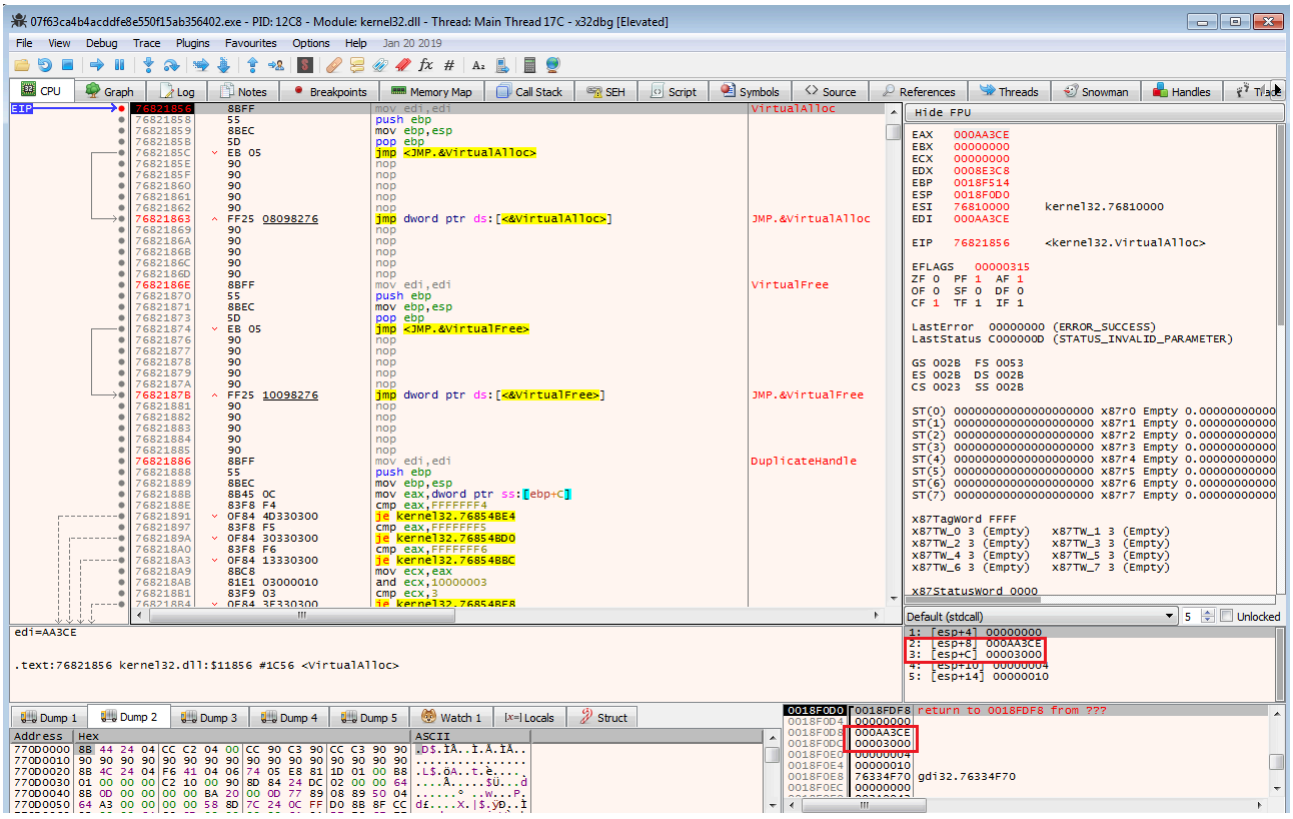
following few statements are preparing to execute shellcode on stack by retrieving a handle to a device context (DC) object and passing this handle to GrayStringA to execute shellcode from stack (ptr value in eax taken from Figure1)



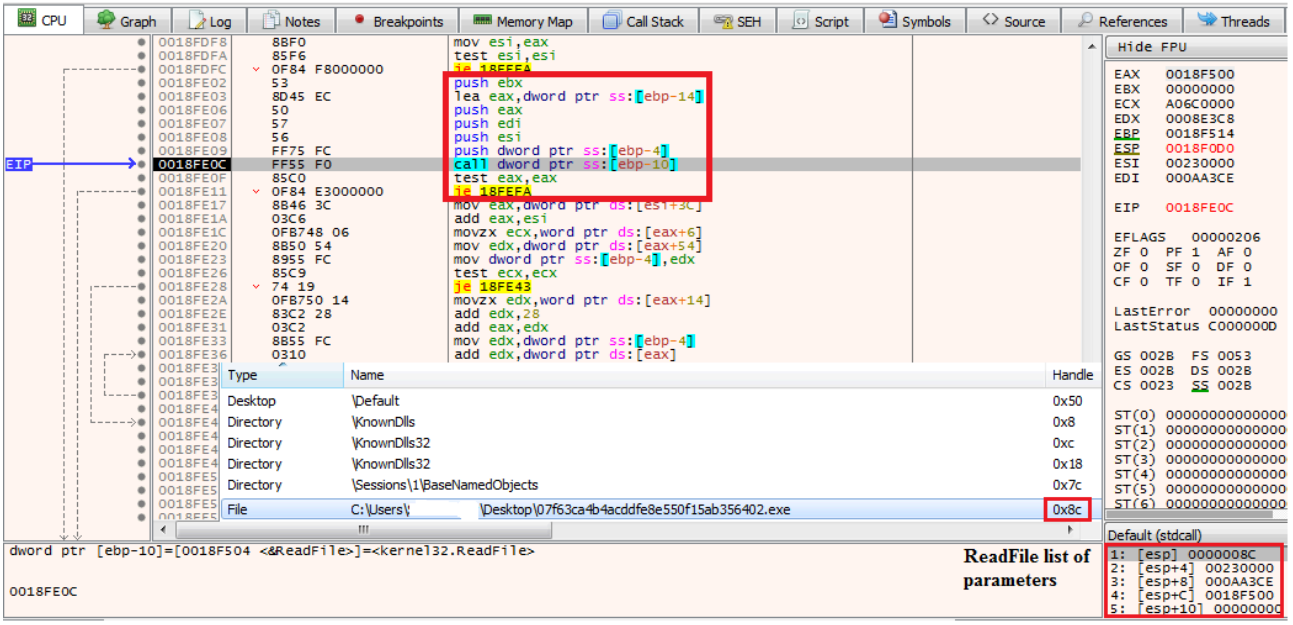

let's now start exploring the shellcode.

**Debugging shellcode to extract final payload**

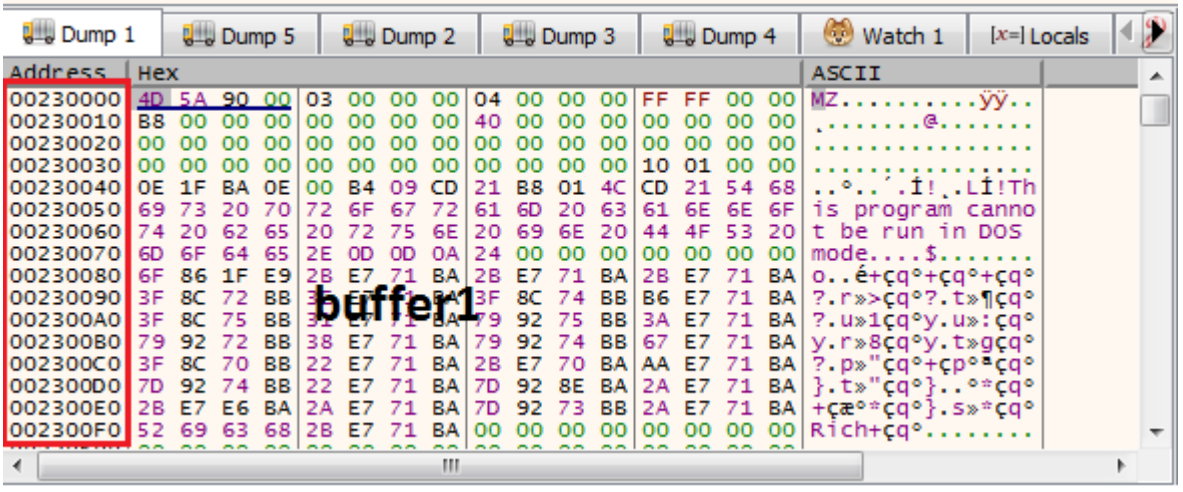
As soon as, **GrayStringA** executes, it hits on **VirtualAlloc** breakpoint set in the debugger, which is being called to reserver/commit 0xAA3CE size of memory with **MEM_COMMIT | MEM_RESERVE** (0x3000) memory allocation type



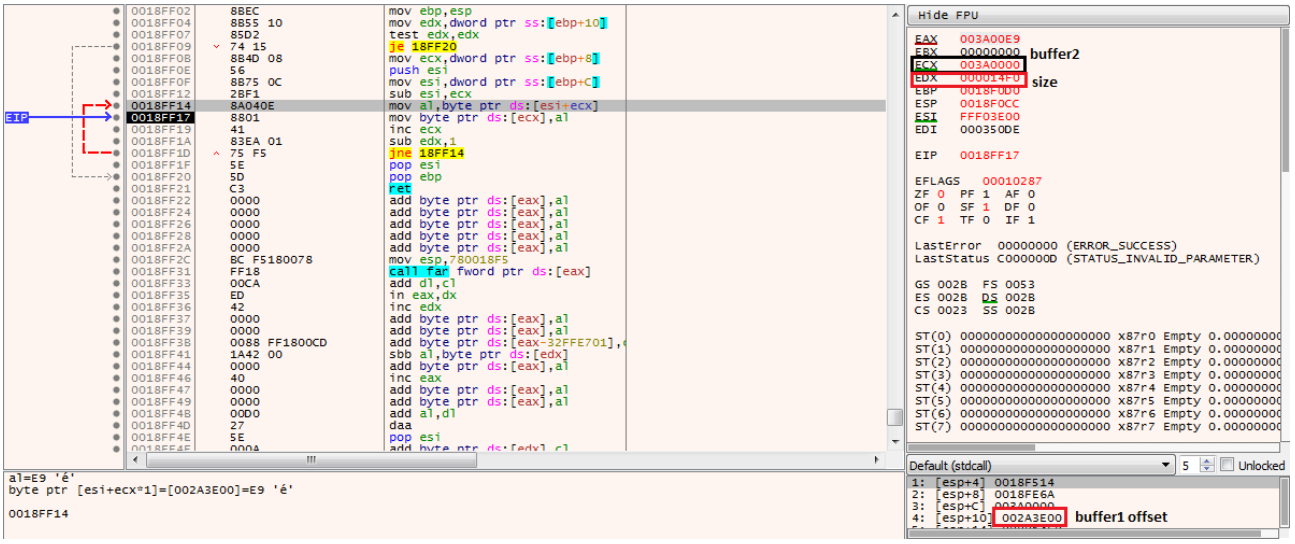
returning control from **VirtualAlloc** and stepping over one more time from ret, leads us to the shellcode, next few statements after VirtualAlloc call are pushing pointer to newly created buffer, size of the buffer and the file handle for currently loaded process on stack to call **ReadFile**



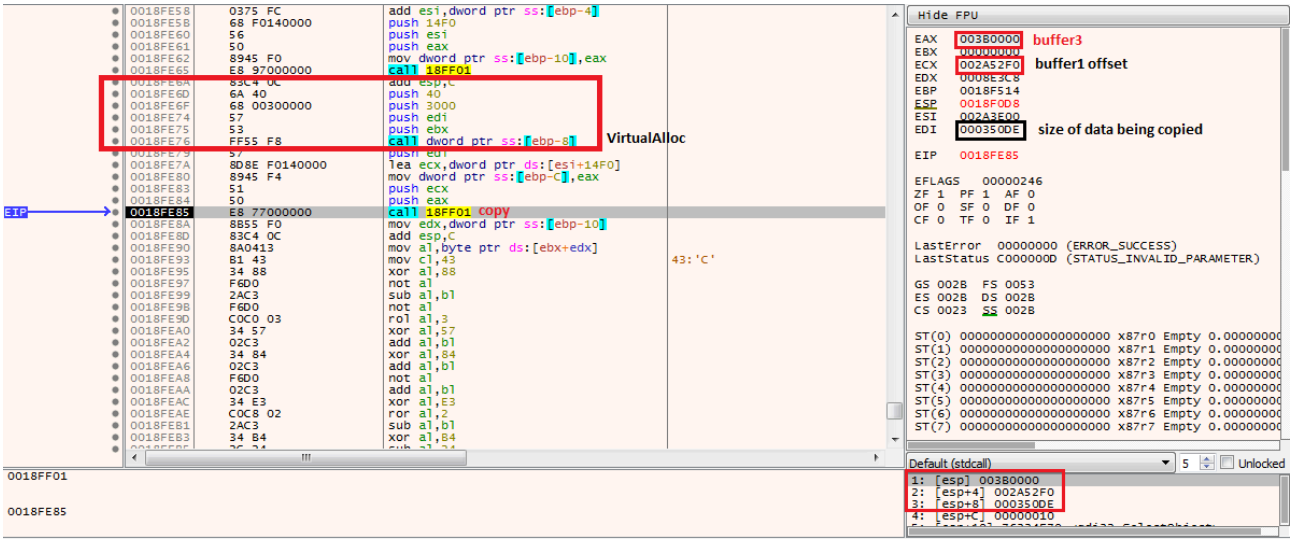

which reads 0xAA3CE bytes of data from parent process image into the buffer, let's say it **buffer1**




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

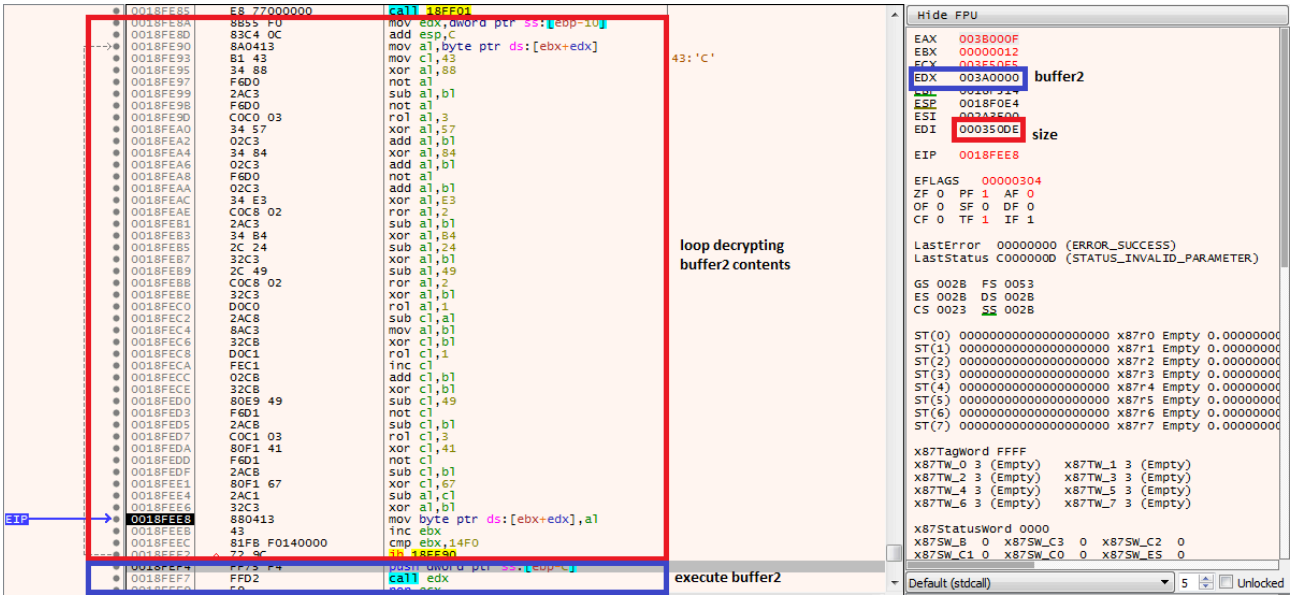
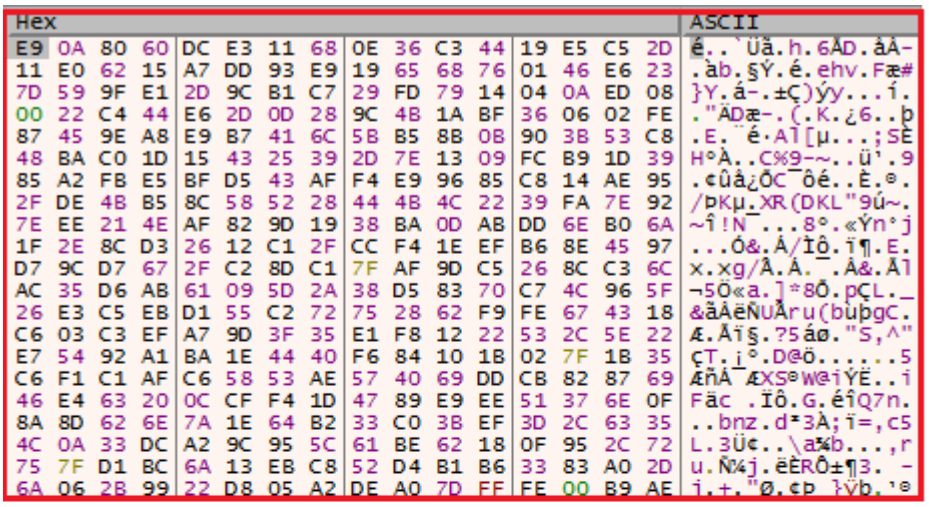
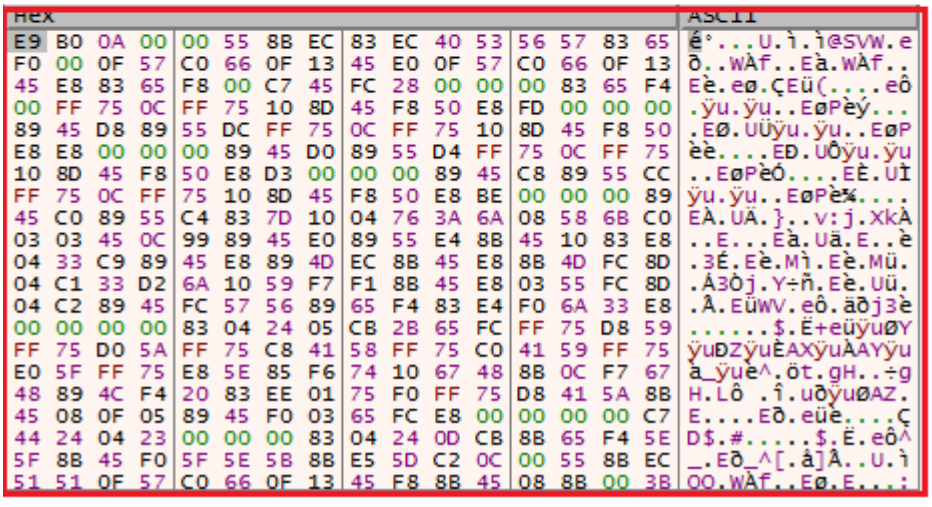



figure below is showing final buffer2 decrypted contents



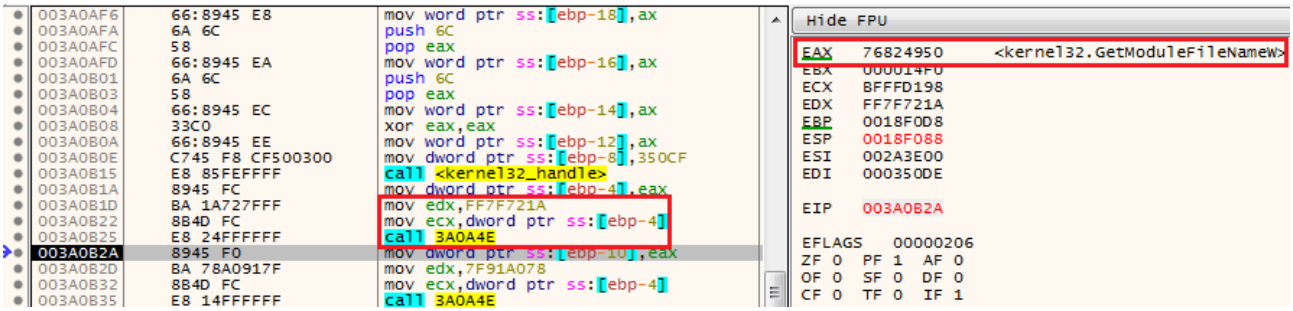
encrypted buffer2



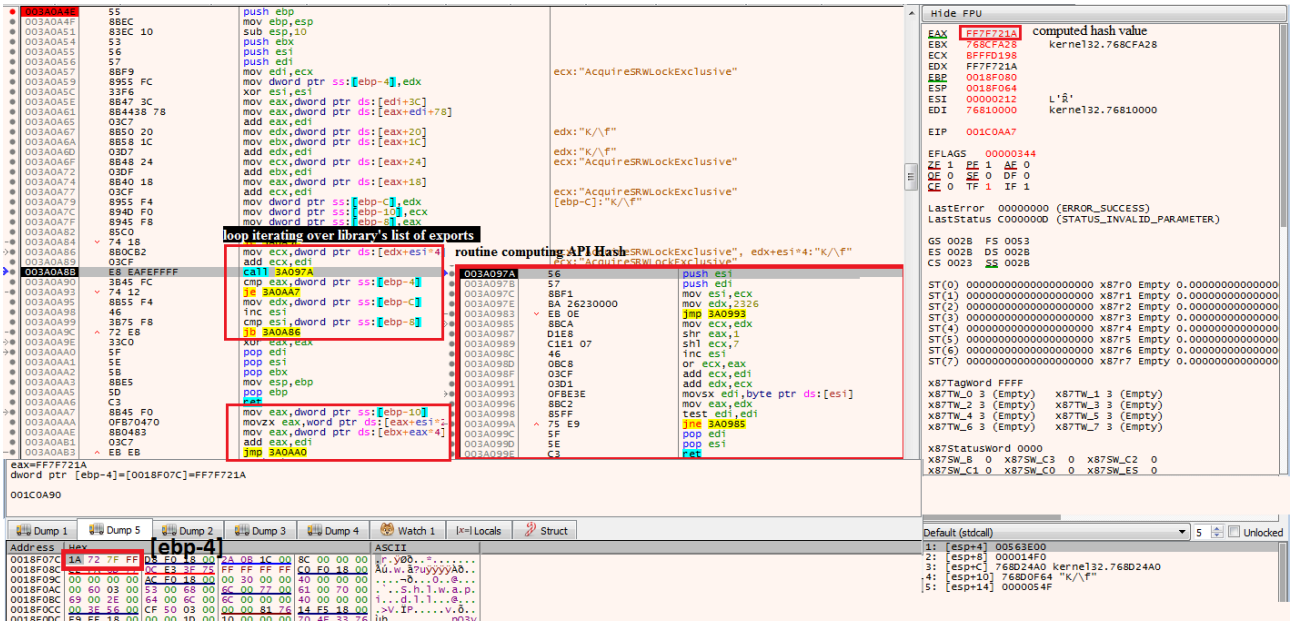
decrypted buffer2

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

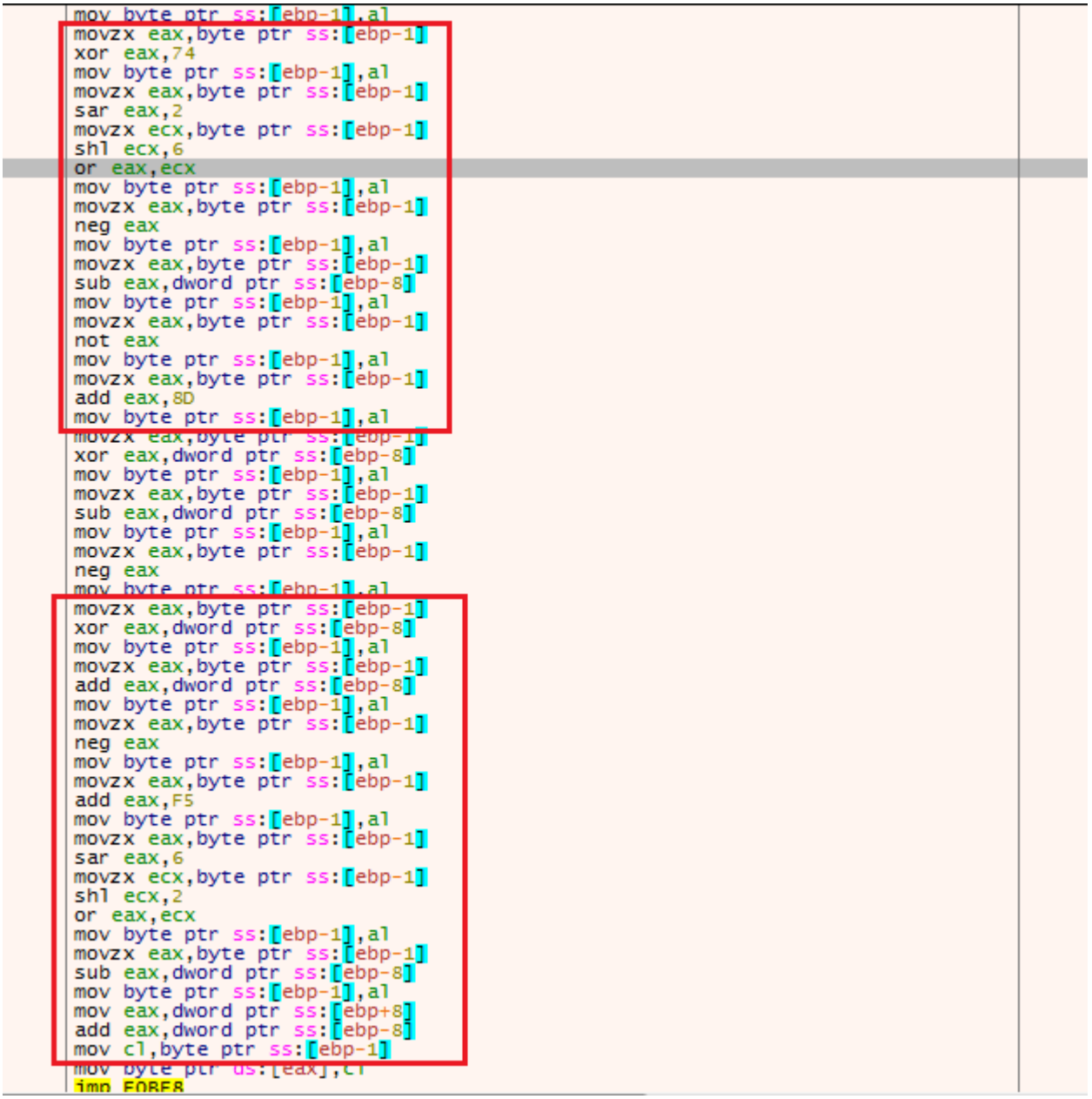

retrieved kernel32.dll handle is passed to next call along with another argument with constant **FF7F721A** value, a quick Google search for this constant results in some public sandbox links but not clear what is this exactly about. Let's dig into it further, stepping over this routine **0x0A4E** results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



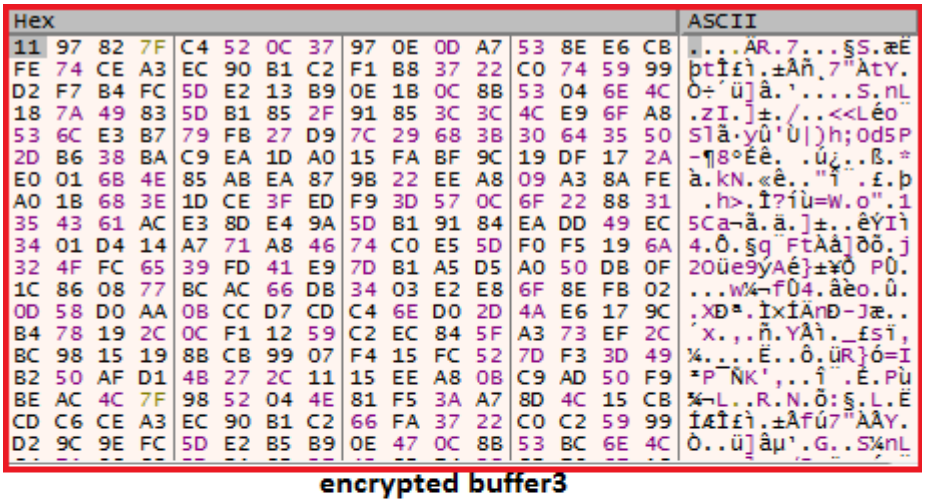
similarly second call resolves **7F91A078** hash value to **ExitProcess** API, wrapper routine **0x0A4E** iterates over library exports and routine **0x097A** is computing hash against input export name parameter. Shellcode seems to be using a custom algorithm to hash API, computed hash value is retuned 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]

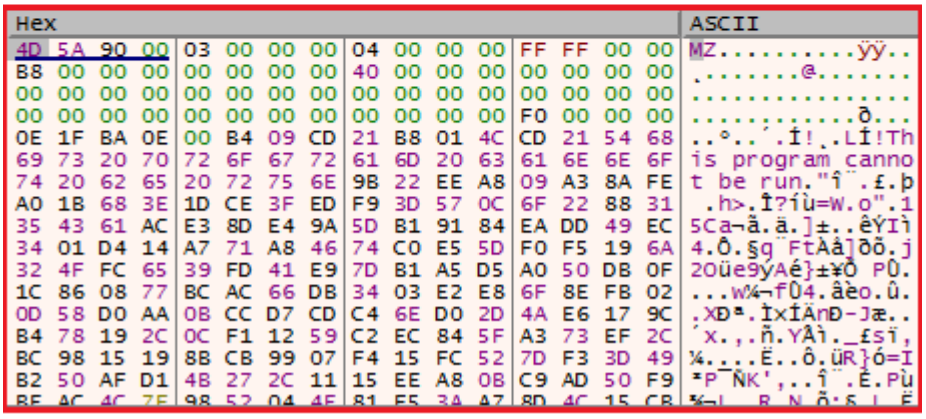



and final value overwrites the corresponding buffer3 value at [eax] offset



| 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\|);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.ÌxÍÄnÐ-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]âµ'.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.ÌxÍÄnÐ-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 |

**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
```

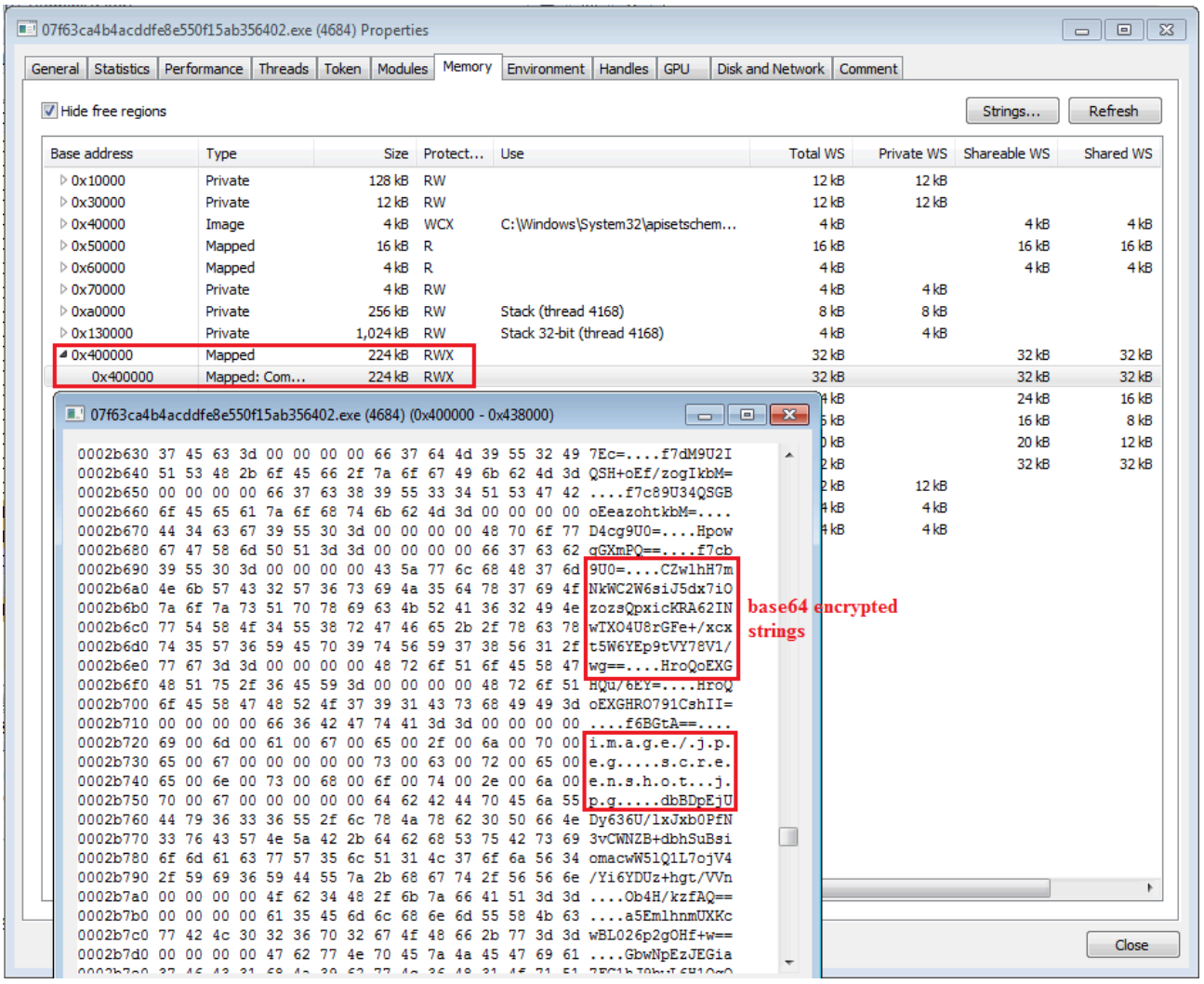
after all required APIs have been resolved, it proceeds to create a new process

```
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

| | | | |
|---|---|---|---|
| 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




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.

---

Source: https://xer0xe9.github.io/A-Case-of-Vidar-Infostealer-Part-1-(-Unpacking-)/