A detailed analysis of ELMER Backdoor used by APT16

cybergeeks.tech/a-detailed-analysis-of-elmer-backdoor-used-by-apt16/

Summary

In this blog post, we're presenting a detailed analysis of a backdoor known as ELMER that was used by the Chinese actor identified as APT16. This group targeted Japanese and Taiwanese organizations in industries such as high-tech, government services, media and financial services.

The malware is encrypted with a custom algorithm and it's written in Delphi. This sample is capable of detecting proxy settings on the local machine and exfiltrating information such as the hostname and IP address of the machine to the Command and Control server. The process uses a custom decryption algorithm that consists of AND, XOR, and ADD operations in order to decrypt relevant strings during runtime. It implements 8 different commands depending on the response from the C2 server, including: file uploads and downloads, process execution, exfiltration of file names/sizes and directory names, exfiltration of processes/process IDs. Data exfiltration is performed using an HTML document that contains the information encoded using the NOT operator.

This sample is using a custom encryption algorithm, that we will describe below. For this analysis, we have also created a python script that can be used to facilitate the decryption process, which can be found at https://github.com/Rackedydig/string_decode_algorithm_apt16.

Technical analysis

SHA256:
BED00A7B59EF2BD703098DA6D523A498C8FDA05DCE931F028E8F16FF434DC89E

It's important to mention that a part of the malicious code is encrypted, and we'll explain using a step-by-step approach how to decrypt it. The process is scanning the memory in order to find the magic number "MZ" which corresponds to EXEs (DLLs), and then it's extracting the first word of the PE header and compares it with "PE" as follows:

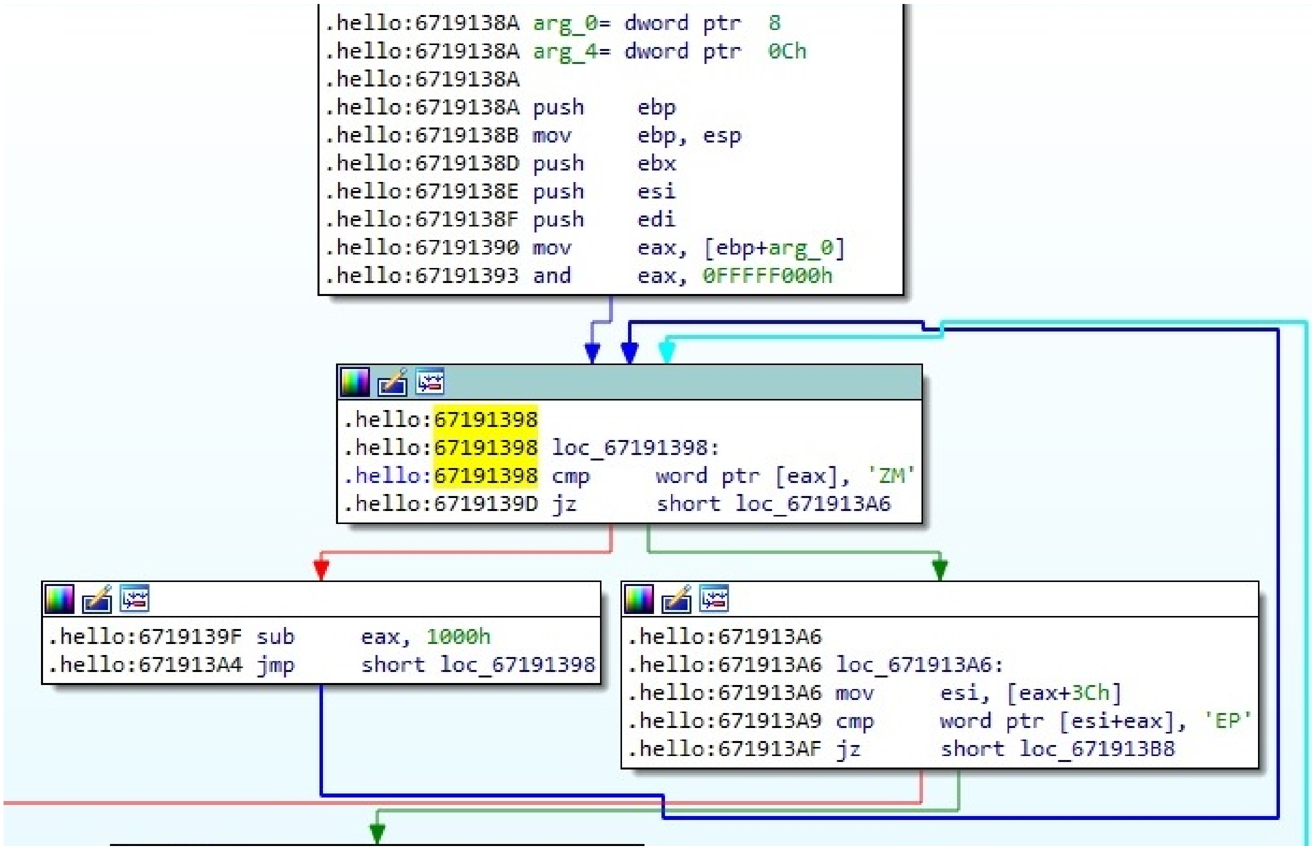

Figure 1

The following picture contains a part of the bytes that will be transformed as we'll see in the next paragraphs:

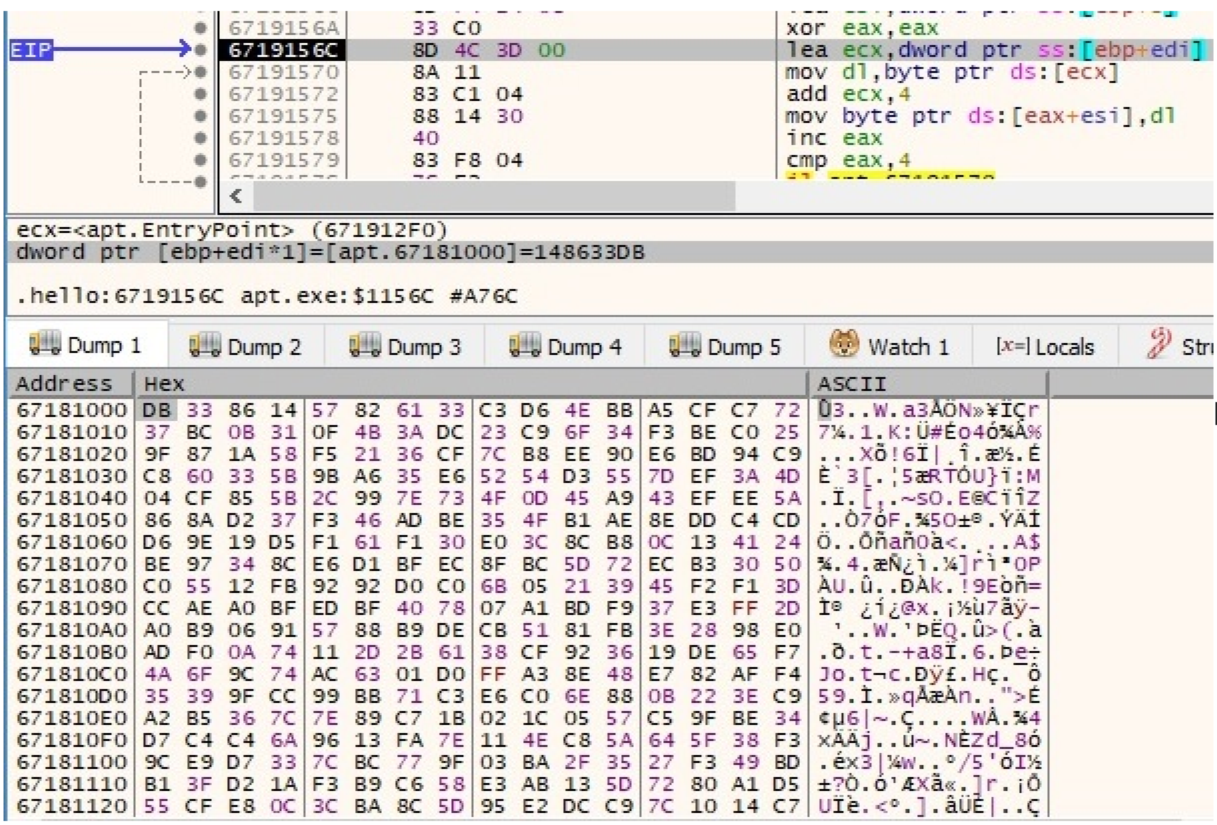

Figure 2

The first 16 bytes are reordered as follows: [byte1, byte5, byte9, byte13], [byte2, byte6, byte10, byte14], [byte3, byte7, byte11, byte15], [byte4, byte8, byte12, byte16]:

Address	Hex				ASCII
0019FEE4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FEF4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FF04	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FF14	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FF24	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FF34	00 00 00 00	00 10 18 67	00 00 00 00	94 FF 19 00	.......g.....ÿ..
0019FF44	DB 57 C3 A5	33 82 D6 CF	86 61 4E C7	14 33 BB 72	ÛWÃ¥3.ÖÏ.aNÇ.3»r

Figure 3

Now there is a buffer of 16 bytes, which represents a “key” in the upcoming operations:

Address	Hex				ASCII
671911AC	D0 C9 E1 B6	14 EE 3F 63	F9 25 0C 0C	A8 89 C8 A6	ÐÉá¶.î?cù%..¨.È¦
671911BC	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
671911CC	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
671911DC	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....

Figure 4

An XOR operation is performed between the corresponding positions of the 2 buffers mentioned above:

Address	Hex				ASCII
0019FEE4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FEF4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FF04	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FF14	00 00 00 00	00 00 00 00	04 00 00 00	54 FF 19 00	............Tÿ..
0019FF24	00 10 18 67	D4 81 00 00	96 15 19 67	44 FF 19 00	...gÔ......gDÿ..
0019FF34	AC 11 19 67	00 10 18 67	00 00 00 00	94 FF 19 00	¬..g...g.....ÿ..
0019FF44	0B 9E 22 13	27 6C E9 AC	7F 44 42 CB	BC BA 73 D4	..".'lé¬.DBË¼°sÔ

Figure 5

The first 4 bytes of the buffer remain in their current positions, however, the last 12 bytes are reordered, as shown in figure 6:

Address	Hex				ASCII
0019FEE4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FEF4	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FF04	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	....
0019FF14	00 00 00 00	00 00 00 00	04 00 00 00	54 FF 19 00	............Tÿ..
0019FF24	00 10 18 67	04 00 00 00	9C 11 19 67	A8 15 19 67	...g.......g¨..g
0019FF34	BA 73 D4 BC	00 10 18 67	00 00 00 00	94 FF 19 00	°sÔ¼...g.....ÿ..
0019FF44	0B 9E 22 13	AC 27 6C E9	42 CB 7F 44	BA 73 D4 BC	..".¬'lèBË.D°sÔ¼

Figure 6

Each byte is replaced by a byte that can be found at the position 0x671911EC+current_byte, as explained in the next figure:

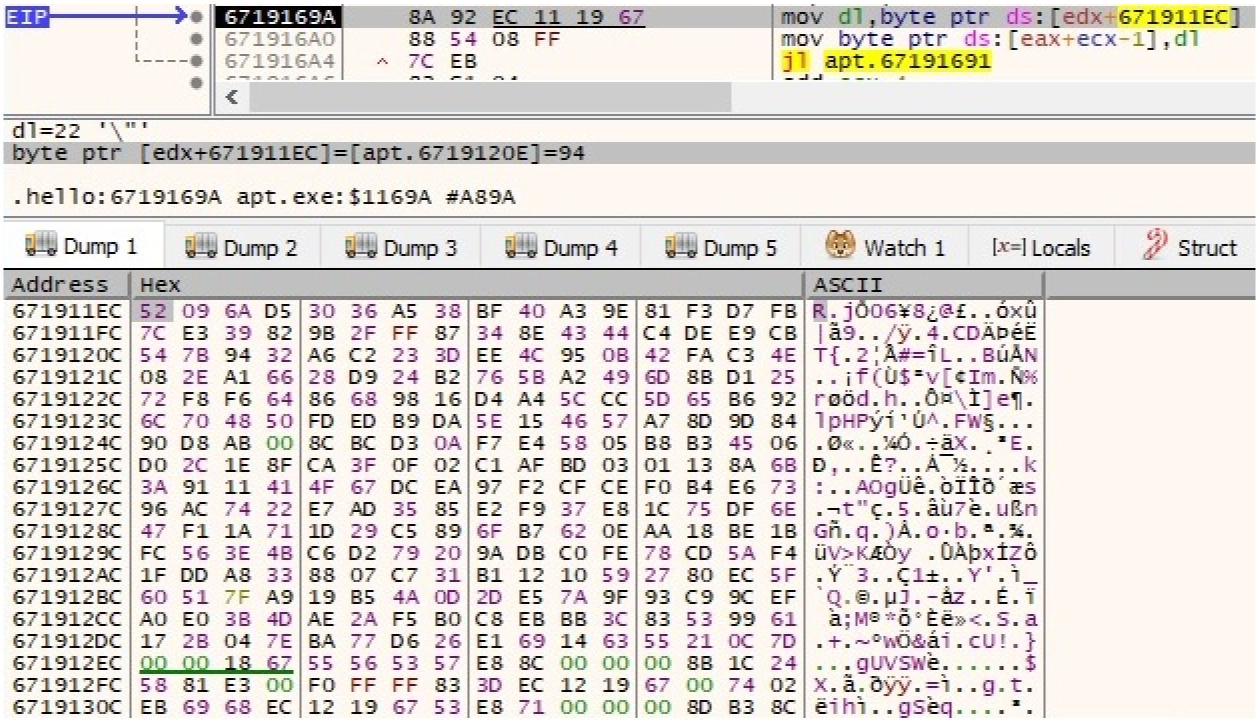

Figure 7

After this transformation, the buffer becomes the following one:

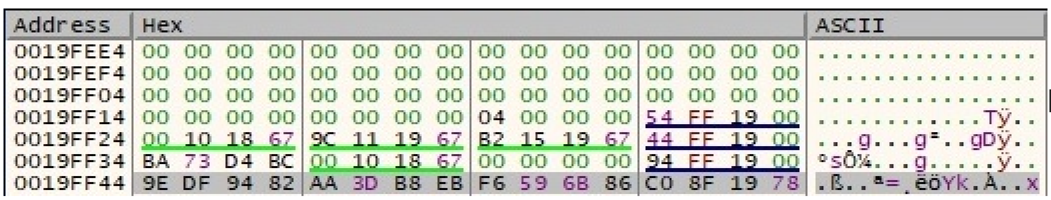

Figure 8

There is a second XOR decryption step, but this time the key is changing:

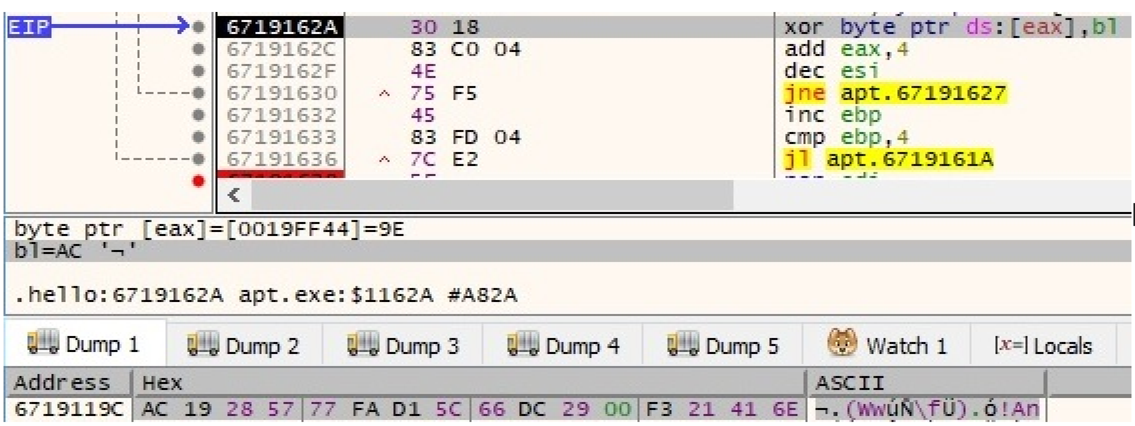

Figure 9

After the XOR operation is complete, the current buffer has been changed, as shown below:

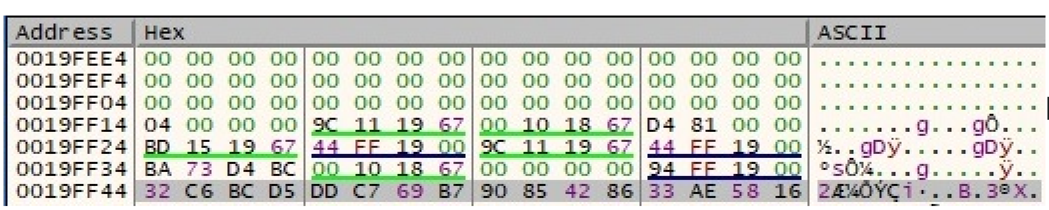

Figure 10

A few more operations will be performed, including shl cl, 1 (shift left by 1) and xor cl, 1B (xor with 0x1B). Let’s take, for example, byte 0x90 from the buffer which is left shifted by 1 (0x20) and then XORed with 0x1B -> 0x3B. Byte 0x3B is left shifted by 1 and becomes 0x76 (no XOR is performed) and one more time, 0x76 is left shifted by 1 and becomes 0xEC. The confirmation that all of these operations are accurate:

```
Address   Hex                                              ASCII
0019FEEC  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0019FEFC  CE 81 00 00 DD FF 19 00 23 17 19 67 0D 00 00 00  Î...Ýÿ..#..g....
0019FF0C  90 3B 76 EC 0B 00 00 00 DD A1 59 B2 04 00 00 00  .;vì....Ý¡Y²....
```

Figure 11

Now the values from this buffer are XORed together (0x90 XOR 0x76) XOR 0xEC and then the result (0xa) is XORed with other results from similar operations. After all operations are done, the buffer will be the following:

```
Address   Hex                                              ASCII
0019FEE4  00 00 00 00 00 00 00 00 DE 81 00 00 02 00 00 C4  ........Þ......Ä
0019FEF4  4E 17 19 67 0E 00 00 00 16 2C 58 B0 09 00 00 00  N..g.....,X°....
0019FF04  86 17 2E 5C 0D 00 00 00 B7 75 EA CF 0B 00 00 00  ...\....·uêÏ....
0019FF14  D5 B1 79 F2 04 00 00 00 9C 11 19 67 00 10 18 67  Õ±yò.......g...g
0019FF24  D4 81 00 00 48 FF 19 00 00 00 00 00 D2 15 19 67  Ô...Hÿ......Ò..g
0019FF34  D5 B7 86 16 00 10 18 67 00 00 00 00 94 FF 19 00  Õ·.....g.....ÿ..
0019FF44  53 B2 35 65 C8 9D 16 44 69 A5 4E C9 BE A0 A2 1A  S²5eÈ..Di¥NÉ¾ ¢.
```

Figure 12

The sample performs the steps presented above 10 times, and the buffer looks like in the next figure:

```
Address   Hex                                              ASCII
0019FEE4  00 00 00 00 00 00 00 00 D1 81 00 00 02 00 00 15  ........Ñ.......
0019FEF4  4E 17 19 67 0E 00 00 00 8E 07 0E 1C 09 00 00 00  N..g............
0019FF04  D0 BB 6D DA 0D 00 00 00 78 F0 FB ED 0B 00 00 00  Ð»mÚ....xðûí....
0019FF14  04 00 00 00 0C 11 19 67 00 10 18 67 D4 81 00 00  .......g...gÔ...
0019FF24  BD 15 19 67 44 FF 19 00 0C 11 19 67 44 FF 19 00  ½..gDÿ.....gDÿ..
0019FF34  A3 8B C4 2E 00 10 18 67 00 00 00 00 94 FF 19 00  £.Ä....g.....ÿ..
0019FF44  04 02 61 00 10 04 72 00 18 43 01 00 67 68 00 FF  ..a...r..C..gh.ÿ
```

Figure 13

The buffer is reordered and copied in the location displayed in figure 2, as follows:

```
Address   Hex                                              ASCII
67181000  04 10 18 67 02 04 43 68 61 72 01 00 00 00 00 FF  ...g..Char.....ÿ
67181010  37 BC 0B 31 0F 4B 3A DC 23 C9 6F 34 F3 BE C0 25  7¼.1.K:Ü#Éo4ó¾À%
67181020  9F 87 1A 58 F5 21 36 CF 7C B8 EE 90 E6 BD 94 C9  ...Xõ!6Ï|¸î.æ½.É
67181030  C8 60 33 5B 9B A6 35 E6 52 54 D3 55 7D EF 3A 4D  È`3[.¦5æRTÓU}ï:M
67181040  04 CF 85 5B 2C 99 7E 73 4F 0D 45 A9 43 EF EE 5A  .Ï.[,.~sO.E©CïîZ
67181050  86 8A D2 37 F3 46 AD BE 35 4F B1 AE 8E DD C4 CD  ..Ò7óF.¾5O±®.ÝÄÍ
67181060  D6 9E 19 D5 F1 61 F1 30 E0 3C 8C B8 0C 13 41 24  Ö..Õñañ0à<.¸..A$
67181070  BE 97 34 8C E6 D1 BF EC 8F BC 5D 72 EC B3 30 50  ¾.4.æÑ¿ì.¼]rì³0P
67181080  C0 55 12 FB 92 92 D0 C0 6B 05 21 39 45 F2 F1 3D  ÀU.û..ÐÀk.!9Eòñ=
67181090  CC AE A0 BF ED BF 40 78 07 A1 BD F9 37 E3 FF 2D  Ì® ¿í¿@x.¡½ù7ãÿ-
671810A0  A0 B9 06 91 57 88 B9 DE CB 51 81 FB 3E 28 98 E0   ¹..W.¹ÞËQ.û>(.à
671810B0  AD F0 0A 74 11 2D 2B 61 38 CF 92 36 19 DE 65 F7  .ð.t.-+a8Ï.6.Þe÷
671810C0  4A 6F 9C 74 AC 63 01 D0 FF A3 8E 48 E7 82 AF F4  Jo.t¬c.Ðÿ£.Hç.¯ô
671810D0  35 39 9F CC 99 BB 71 C3 E6 C0 6E 88 0B 22 3E C9  59.Ì.»qÃæÀn.">É
671810E0  A2 B5 36 7C 7E 89 C7 1B 02 1C 05 57 C5 9F BE 34  ¢µ6|~.Ç....WÅ.¾4
671810F0  D7 C4 C4 6A 96 13 FA 7E 11 4E C8 5A 64 5F 38 F3  ×ÄÄj..ú~.NÈZd_8ó
67181100  9C E9 D7 33 7C BC 77 9F 03 BA 2F 35 27 F3 49 BD  .é×3|¼w..º/5'óI½
67181110  B1 3F D2 1A F3 B9 C6 58 E3 AB 13 5D 72 80 A1 D5  ±?Ò.ó¹ÆXã«.]r.¡Õ
67181120  55 CF E8 0C 3C BA 8C 5D 95 E2 DC C9 7C 10 14 C7  UÏè.<º.].âÜÉ|..Ç
```

Figure 14

The algorithm applied for the first 16 bytes is repeated 2078 times. The new buffer is the decrypted version of the first one:

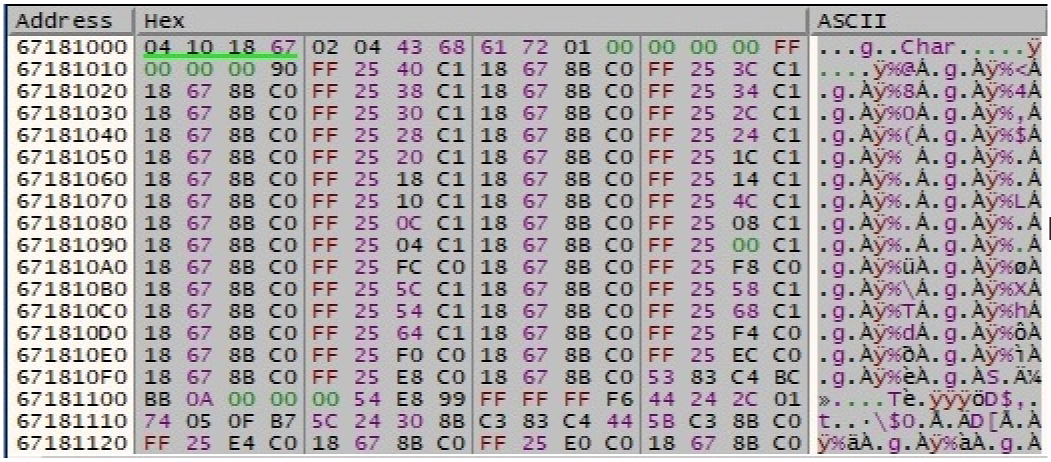

Figure 15

The malicious process loads multiple DLLs and retrieves the address of export functions using LoadLibraryA and GetProcAddress APIs:

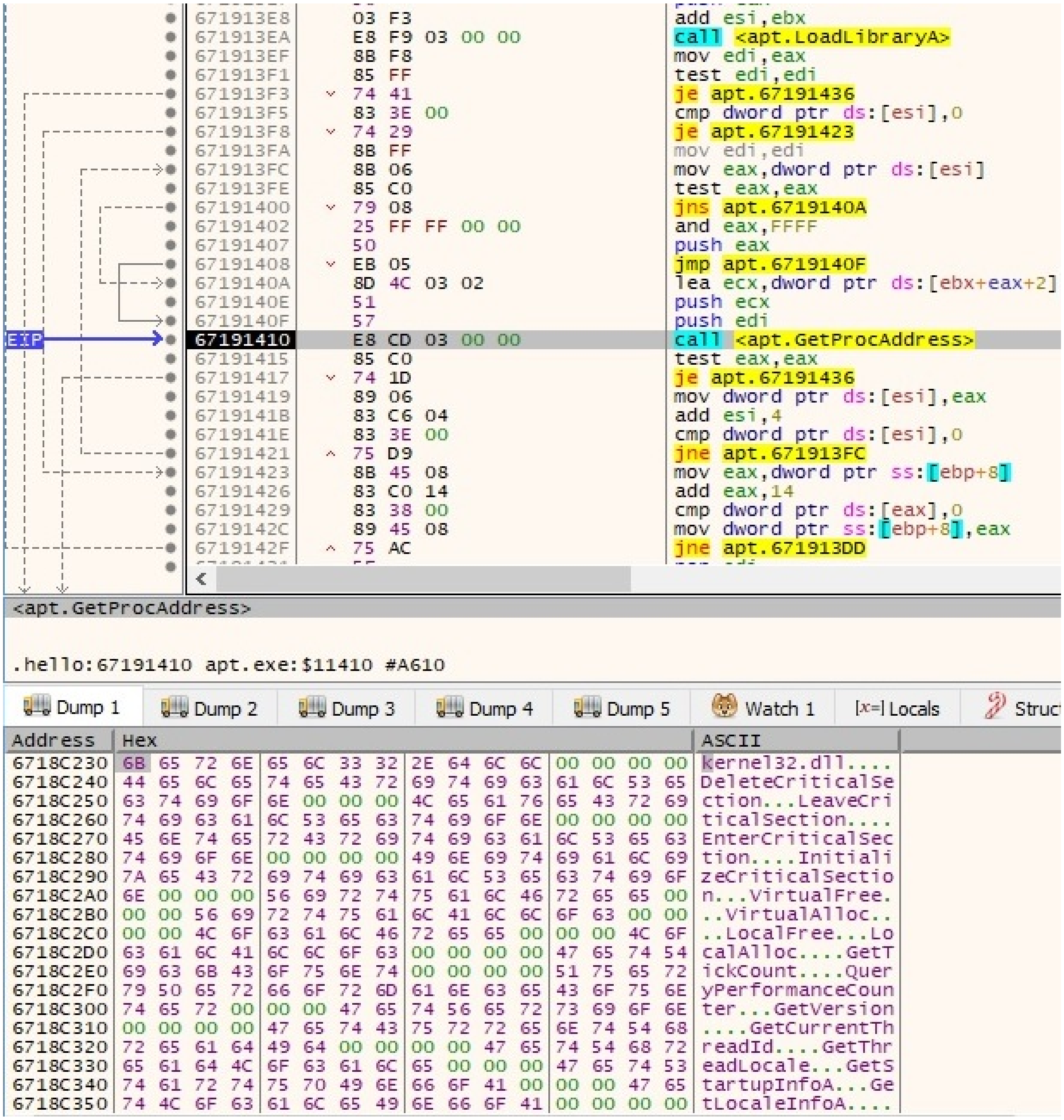

Figure 16

The list of DLLs to be loaded + the export functions:

kernel32.dll

DeleteCriticalSection, LeaveCriticalSection, EnterCriticalSection, InitializeCriticalSection, VirtualFree, VirtualAlloc, LocalFree, LocalAlloc, GetTickCount, QueryPerformanceCounter, GetVersion, , GetCurrentThreadId, GetThreadLocale, GetStartupInfoA, GetLocaleInfoA, GetLastError, GetCommandLineA, FreeLibrary, ExitProcess, WriteFile, UnhandledExceptionFilter, SetEndOfFile, RtlUnwind, RaiseException, GetStdHandle, GetFileSize, GetFileType, CreateFileA, CloseHandle, TlsSetValue, TlsGetValue, GetModuleHandleA, lstrcmpiA, WaitForSingleObject, Sleep, SetFilePointer, ReadFile, GetProcAddress, GetModuleFileNameA, GetFileAttributesA, GetCurrentDirectoryA, FindNextFileA, FindFirstFileA, FindClose, FileTimeToLocalFileTime, CreateThread, CreateProcessA

user32.dll

GetKeyboardType, MessageBoxA

advapi32.dll

RegQueryValueExA, RegOpenKeyExA, RegCloseKey

oleaut32.dll

SysFreeString, SysReAllocStringLen

ws2_32.dll

WSAGetLastError, gethostname, gethostbyname, socket, setsockopt, send, recv, inet_ntoa, inet_addr, htons, connect, closesocket, WSACleanup, WSAStartup

dnsapi.dll

DnsRecordListFree, DnsQuery_A

The process passes the execution flow to the unencrypted code as illustrated in the next figure:

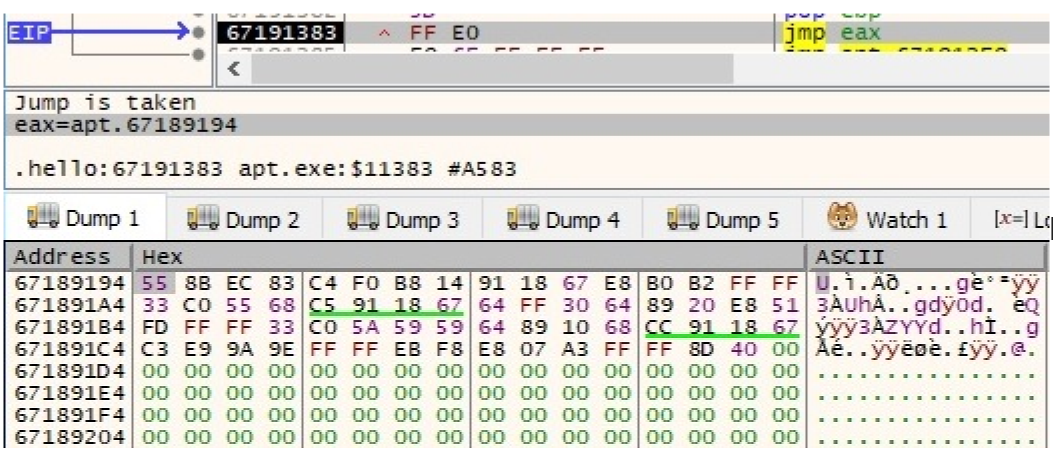

Figure 17

In order to also perform static analysis on the binary, we have to dump the memory of this process using OllyDumpEx plugin of x32dbg debugger:

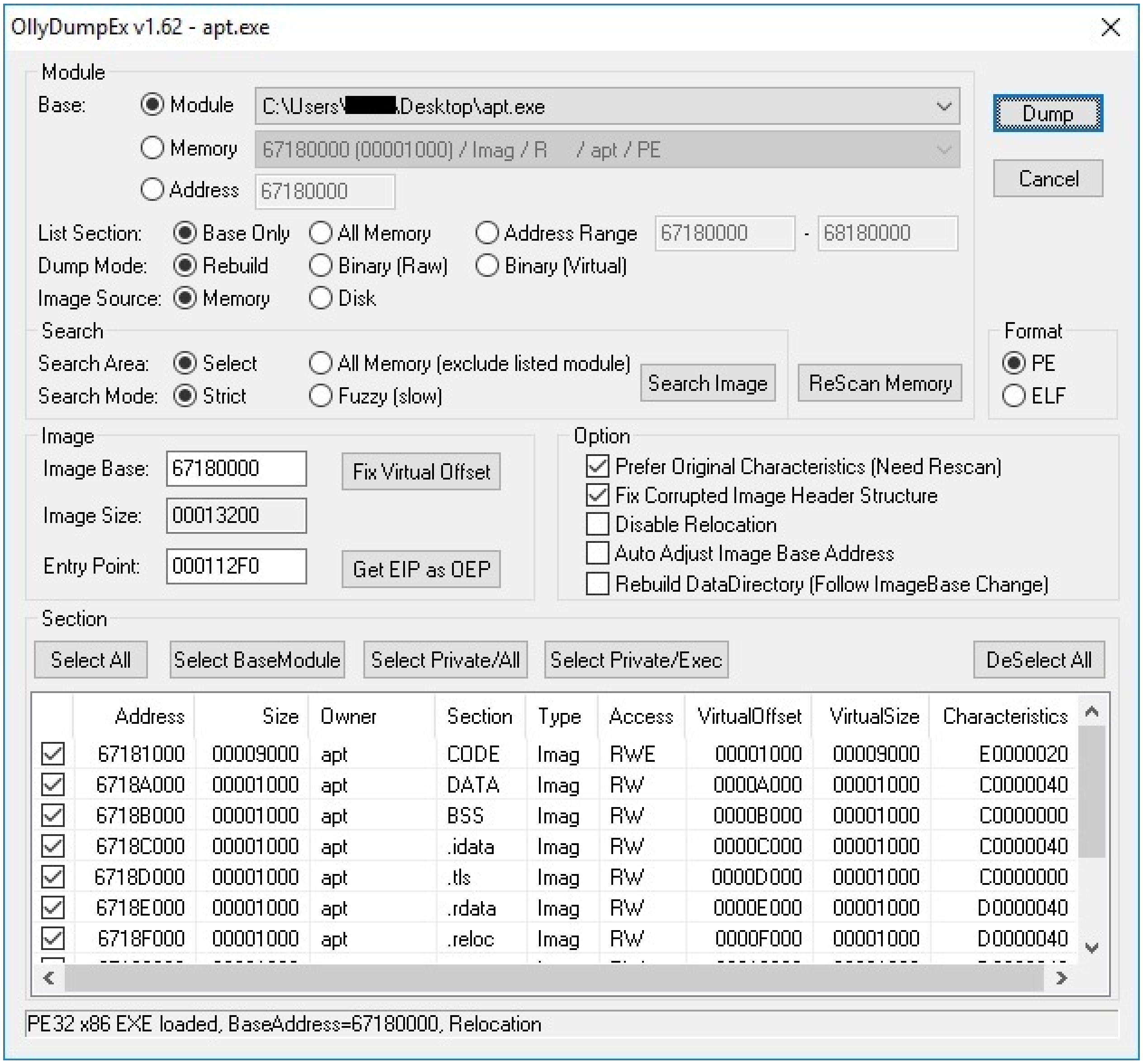

Figure 18

The problem is that the IAT (Import address table) hasn't been populated as expected and contains only 2 functions that were also present in the original binary:

Figure 19

We have to use another plugin of x32dbg called Scylla. This plugin is used to find the IAT entries in the process memory, and then it can fix our dropped binary:

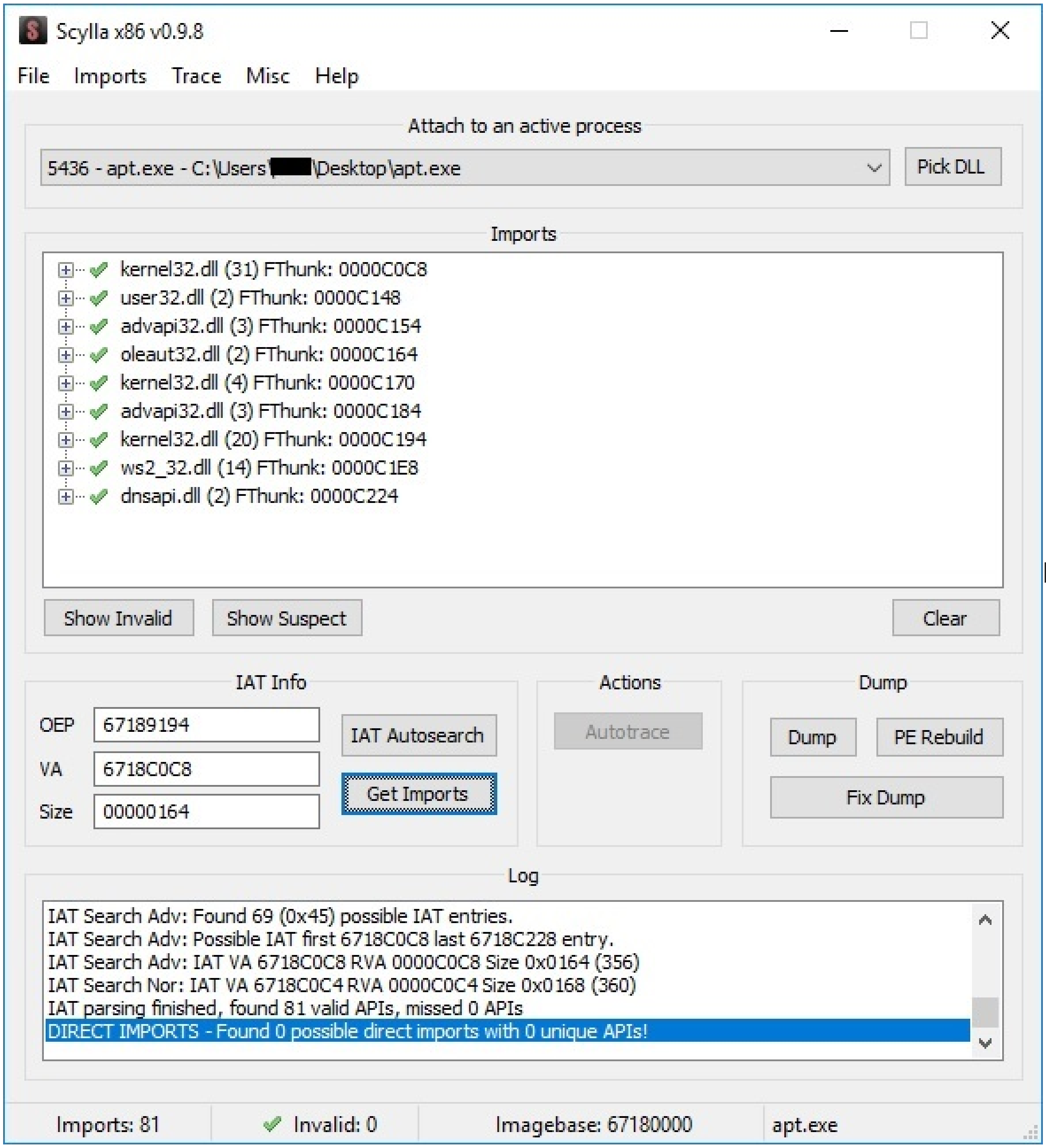

Figure 20

We've successfully fixed the IAT in our dropped binary, and this operation is useful because it reveals different API calls which have to be analyzed:

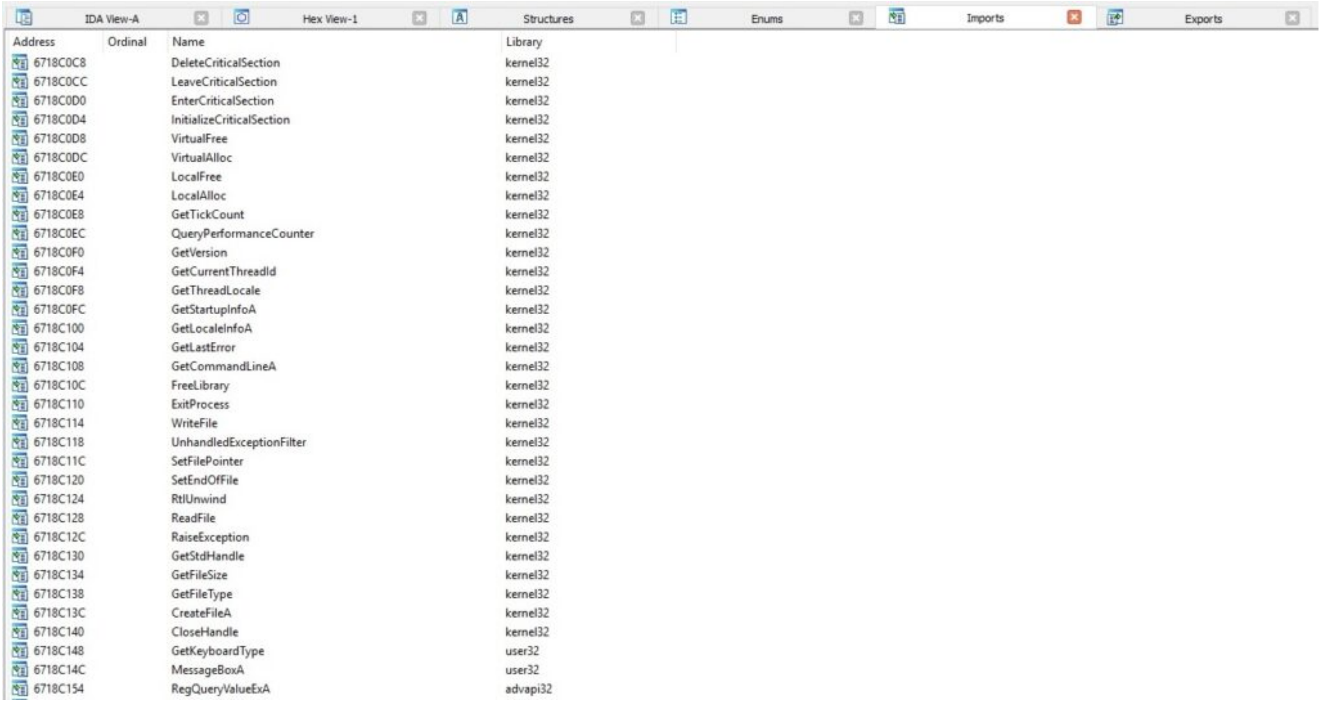

Figure 21

Now we will analyze the decrypted binary. It initiates the use of Winsock DLL by calling the WSAStartup function:

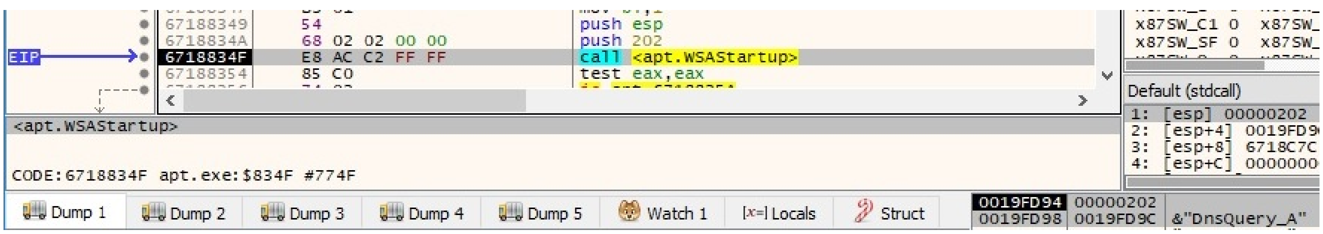

Figure 22

During the entire execution, the process decrypts relevant strings by using a custom algorithm that can be described shortly: If m is the encrypted buffer and key is the decryption key, the result of the algorithm is (m[i] AND 0xF) XOR (key[i] AND 0xF) + (m[i] AND 0xF0), as presented below:

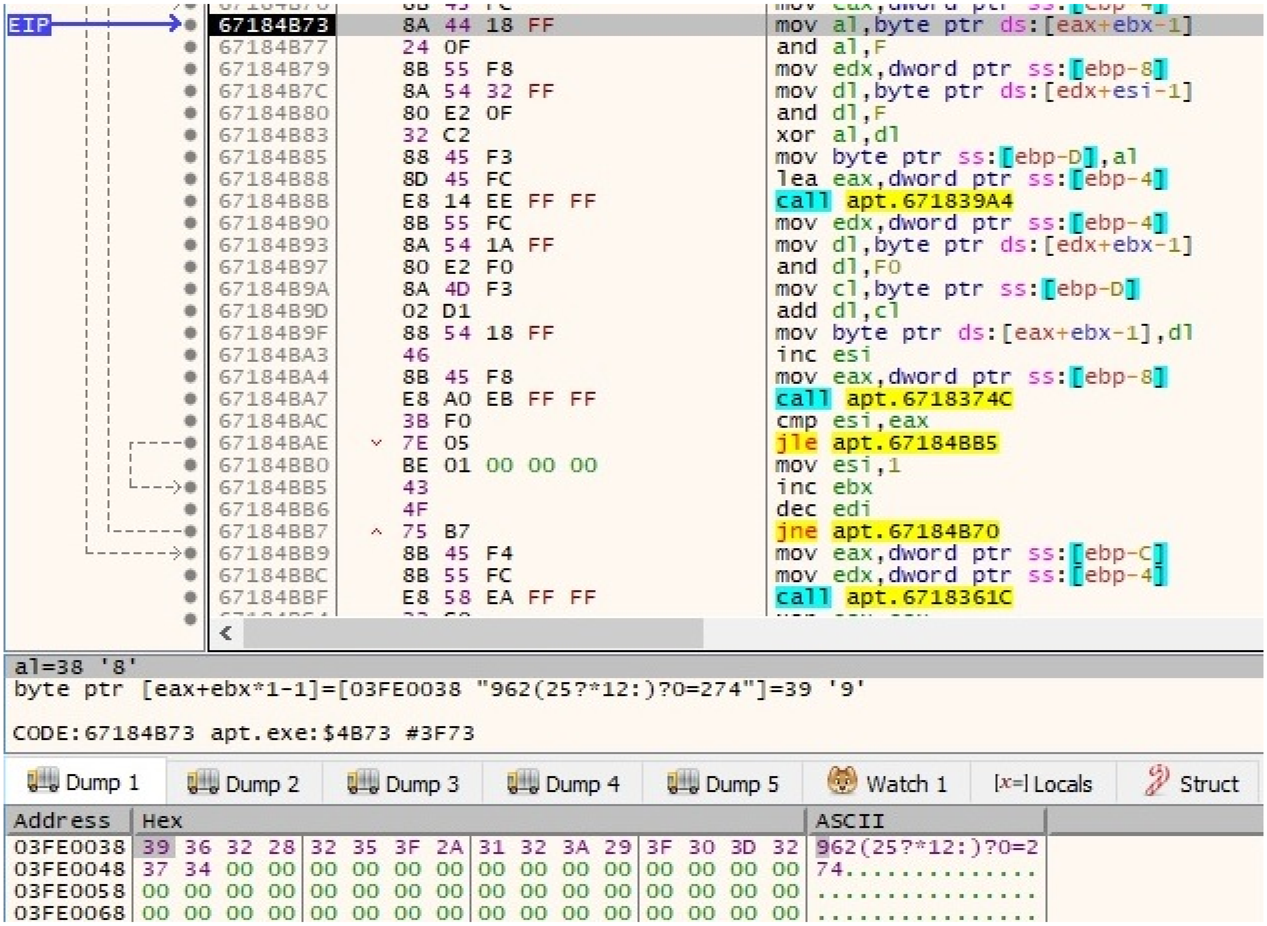

Figure 23

After these operations are finished, the result represents the C2 server and the corresponding port number:

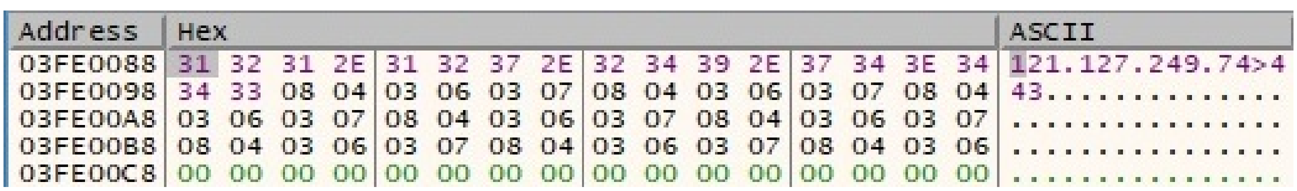

Figure 24

The malware opens the “Software\Microsoft\Windows\CurrentVersion\Internet Settings” registry key by calling the RegOpenKeyExA API:

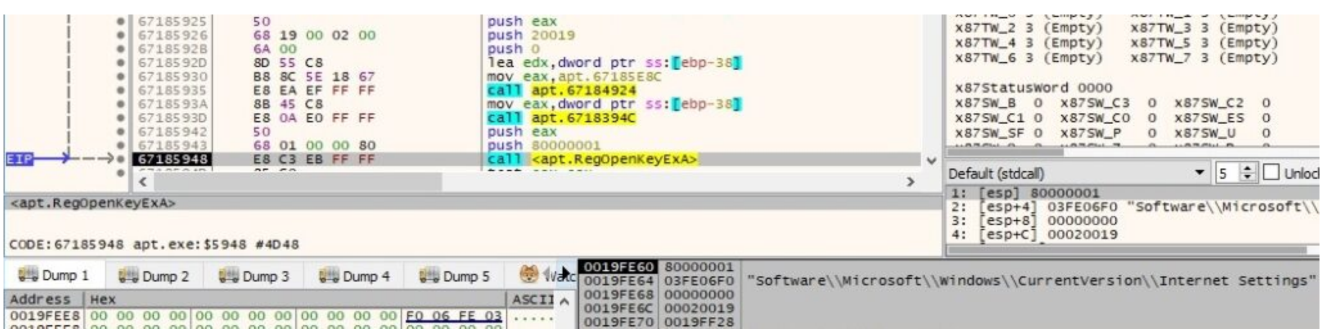

Figure 25

The “ProxyEnable” value is extracted using the RegQueryValueExA function, and it’s compared with 1. This action has the purpose of verifying if the current machine is using a proxy for network communications:

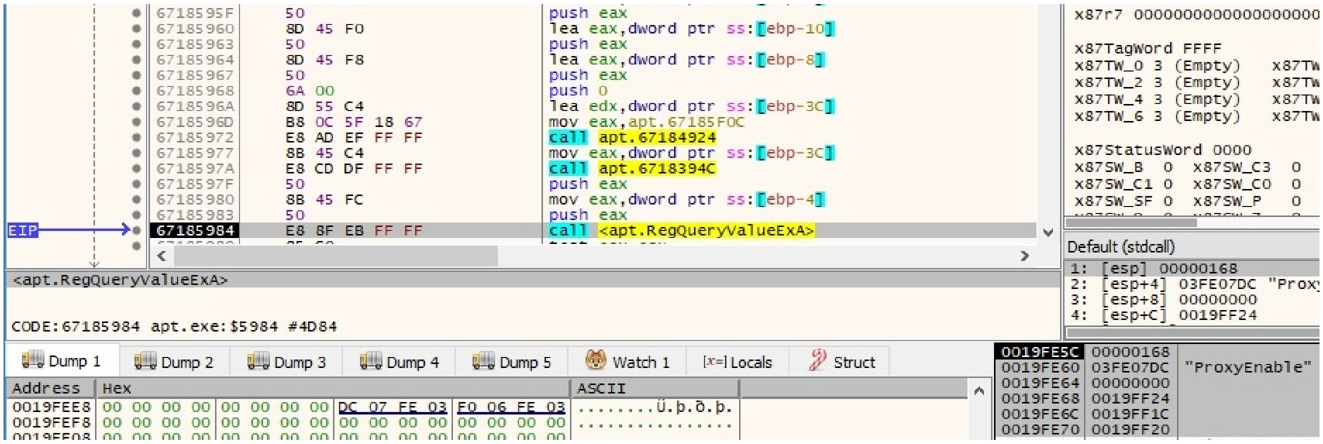

Figure 26

If “ProxyEnable” is equal to 1, the malware proceeds and extracts the value of “ProxyServer” (hostnames/IPs of the proxy server on the network), as displayed in the next figure:

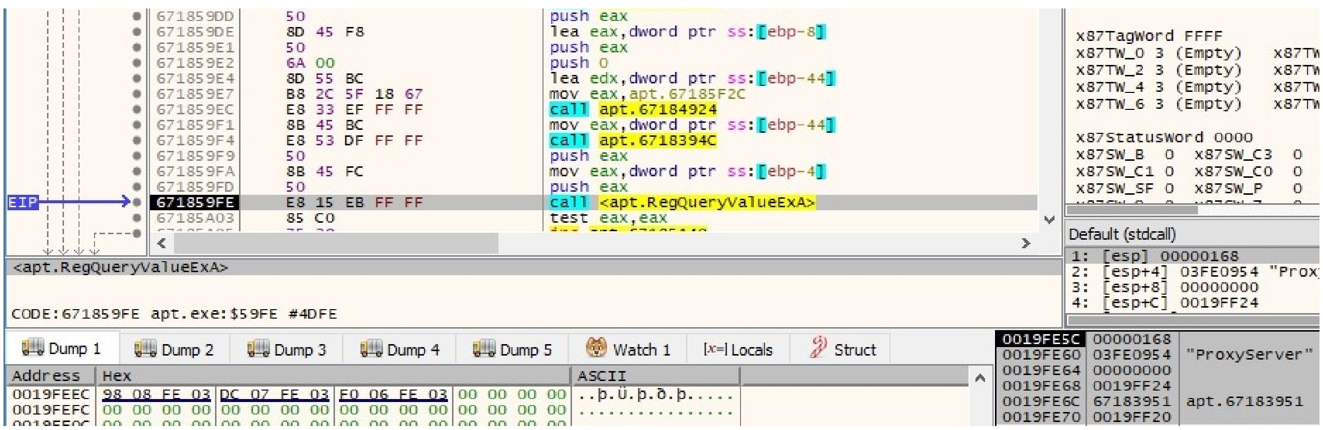

Figure 27

The gethostname function is used to retrieve the host name for the local machine:

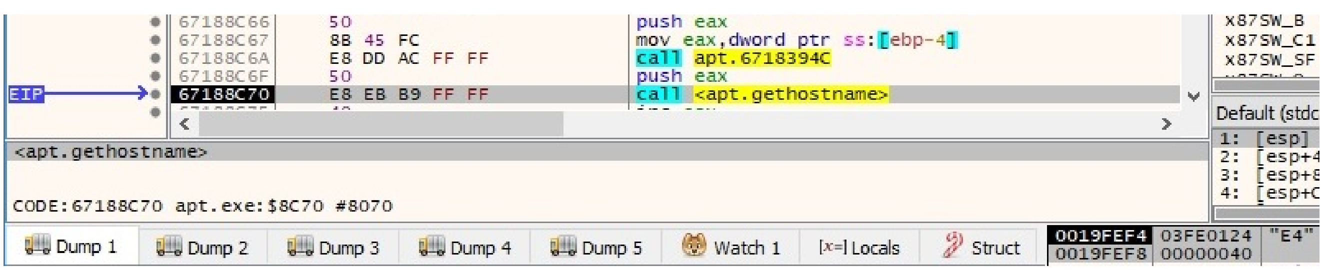

Figure 28

The function result from above is used as a parameter for the gethostbyname function, which can be used to retrieve host information corresponding to the local machine, as shown in figure 29:

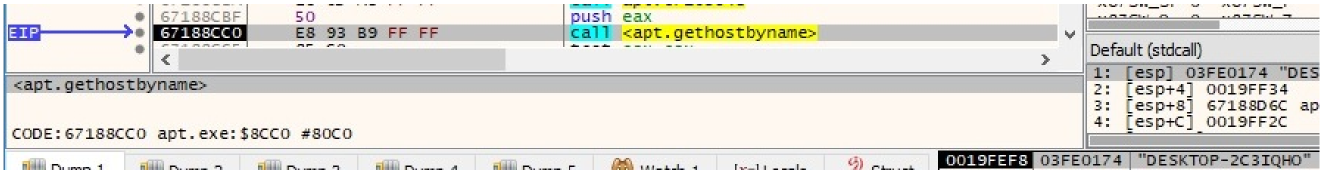

Figure 29

The inet_ntoa function is utilized to convert the IP address of the host into an ASCII string (dotted-decimal format):

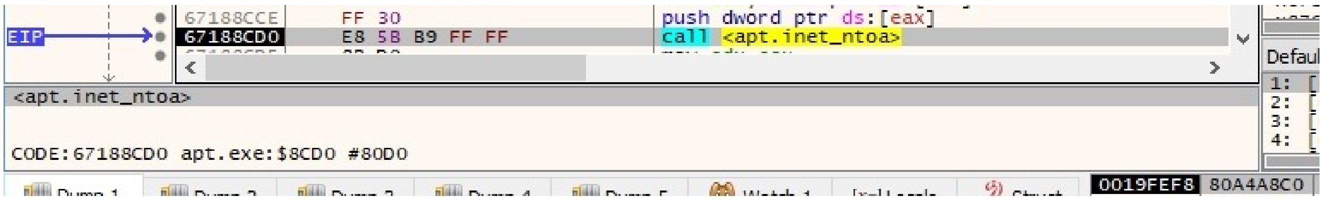

Figure 30

There is some sort of reverse operation done by the malware because it's using the inet_addr function to convert the string representation of the IP address into a proper address for the IN_ADDR structure:

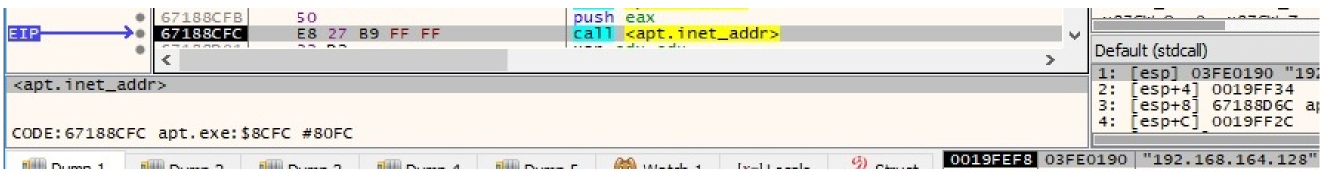

Figure 31

The hostname and the IP address of the machine represented as a decimal number are combined into a string that will be used in the upcoming network communications with the C2 server:

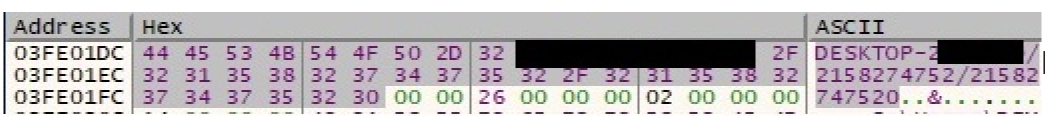

Figure 32

The malicious process uses the same decryption algorithm described before in order to decrypt important strings. The function is highlighted in the next picture:

```
CODE:671852F9 call    sub_67184BF0
CODE:671852FE push    [ebp+var_8]
CODE:67185301 push    ds:dword_6718B684
CODE:67185307 push    offset _str__.Text
CODE:6718530C push    ds:dword_6718B688
CODE:67185312 lea     ecx, [ebp+var_C]
CODE:67185315 mov     edx, offset _str__dhg.Text
CODE:6718531A mov     eax, offset _str__gp_g__.Text
CODE:6718531F call    sub_67184BF0
CODE:67185324 push    [ebp+var_C]
CODE:67185327 push    [ebp+var_4]
CODE:6718532A lea     ecx, [ebp+var_10]
CODE:6718532D mov     edx, offset _str__dhg.Text
CODE:67185332 mov     eax, offset _str__mfck__wft_OZPX.Text
CODE:67185337 call    sub_67184BF0
CODE:6718533C push    [ebp+var_10]
CODE:6718533F push    offset _str____0.Text
CODE:67185344 lea     edx, [ebp+var_14]
CODE:67185347 mov     eax, offset _str_u0_J.Text
CODE:6718534C call    sub_67184D28
CODE:67185351 push    [ebp+var_14]
CODE:67185354 push    offset _str____0.Text
CODE:67185359 lea     ecx, [ebp+var_18]
CODE:6718535C mov     edx, offset _str_f_dg.Text
CODE:67185361 mov     eax, offset _str_Gmgbvz_Kg_crgia.Text
CODE:67185366 call    sub_67184BF0
CODE:6718536B push    [ebp+var_18]
CODE:6718536E push    offset _str____0.Text
CODE:67185373 lea     ecx, [ebp+var_1C]
CODE:67185376 mov     edx, offset _str_f_dg.Text
CODE:6718537B mov     eax, offset _str_Naws__.Text
CODE:67185380 call    sub_67184BF0
CODE:67185385 push    [ebp+var_1C]
CODE:67185388 push    ds:dword_6718B684
CODE:6718538E push    offset _str__.Text
CODE:67185393 push    ds:dword_6718B688
CODE:67185399 push    offset _str____0.Text
CODE:6718539E lea     ecx, [ebp+var_20]
CODE:671853A1 mov     edx, offset _str_f_dg.Text
CODE:671853A6 mov     eax, offset _str_V_e_ko__ha_dgml.Text
CODE:671853AB call    sub_67184BF0
CODE:671853B0 push    [ebp+var_20]
CODE:671853B3 push    offset _str____0.Text
CODE:671853B8 lea     edx, [ebp+var_24]
```

Figure 33

An example of how the algorithm performs is displayed below, where EAX represents the encrypted string and the key is moved into the EDX register:

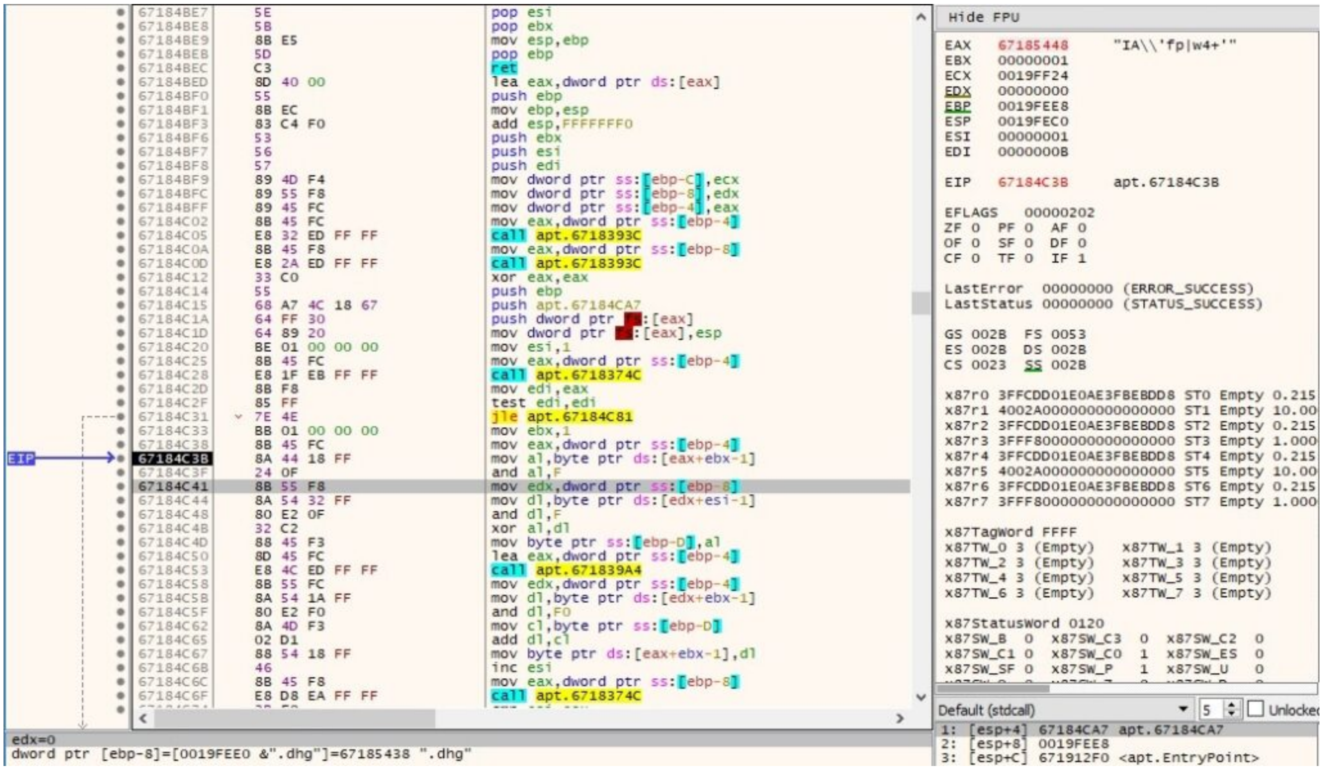

Figure 34

By placing a breakpoint after the operation is supposed to end, we can observe that the string was successfully decrypted:

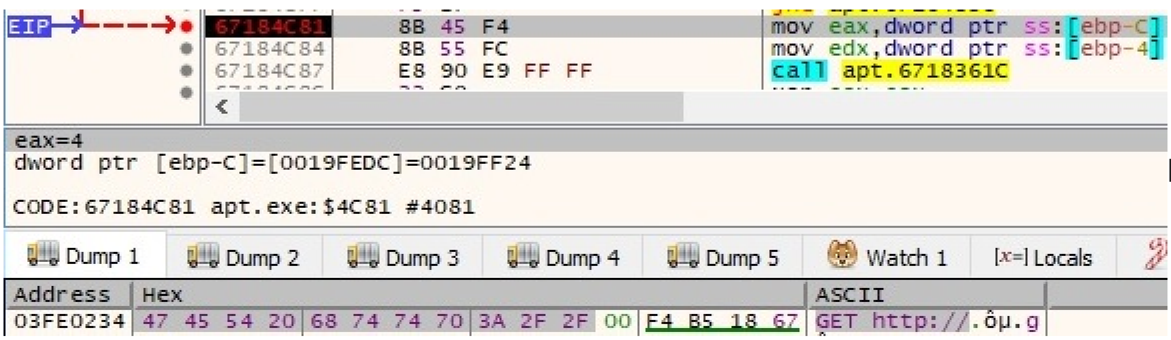

Figure 35

After a few more operations are performed, we can distinguish other interesting strings, like the User Agent that will be used in the communications with the Command and Control server:

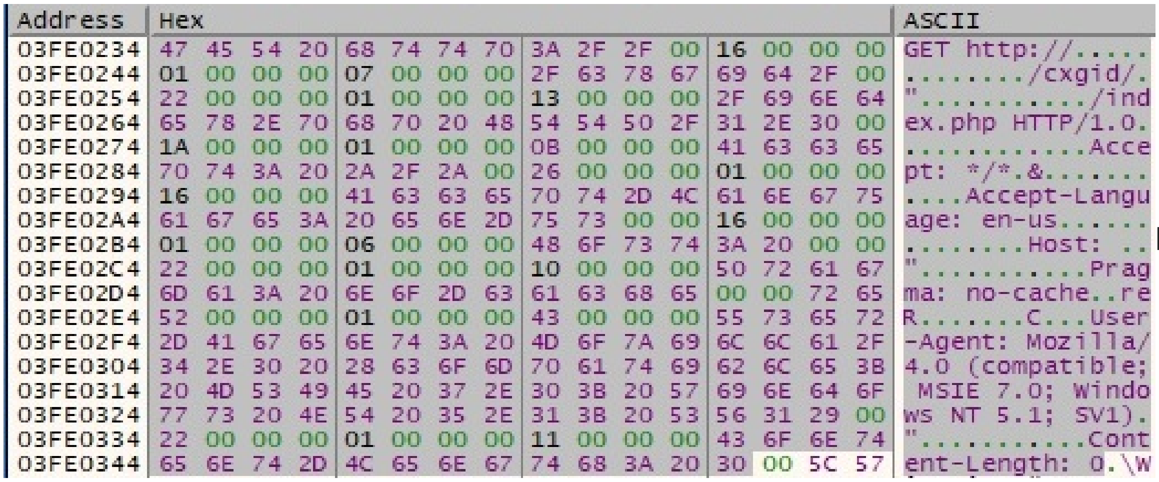

Figure 36

The sample builds an HTML document that contains the infected hostname and the IP address corresponding to the local machine. This form will be used in a POST request as we'll see later on:

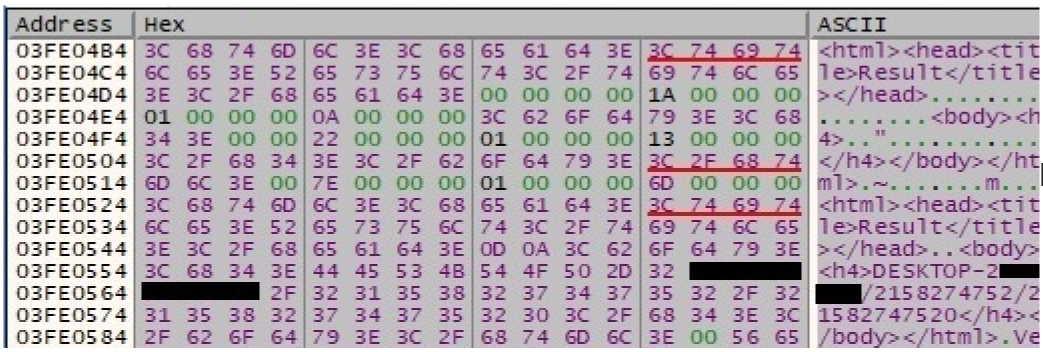

Figure 37

The socket function is used to create a socket, and the following parameters are passed to the function call: 0x2 (**AF_INET** – IPv4 address family), 0x1 (**SOCK_STREAM** – provides sequenced, reliable, two-way streams with an OOB data transmission mechanism) and 0 (the protocol is not specified). The function call is shown below:

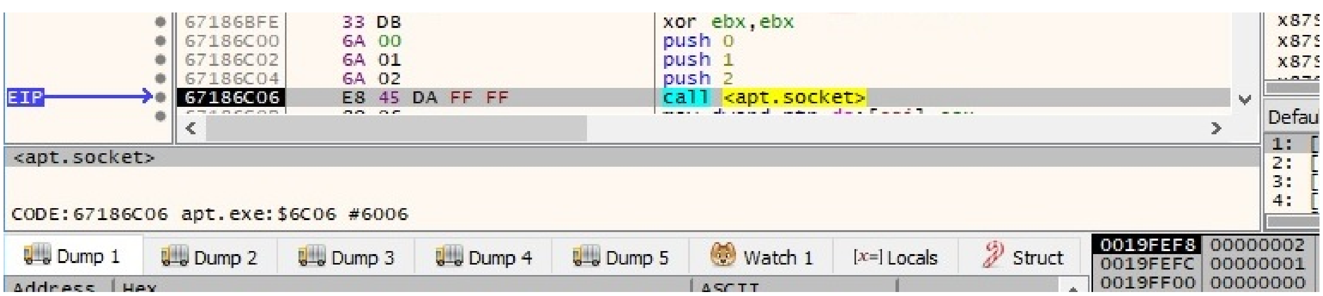

Figure 38

The setsockopt API is used to set a socket option. The following parameters can be highlighted – 0xFFFF (**SOL_SOCKET** – socket layer), 0x8 (**SO_KEEPALIVE** – enable keep-alive packets for a socket connection):

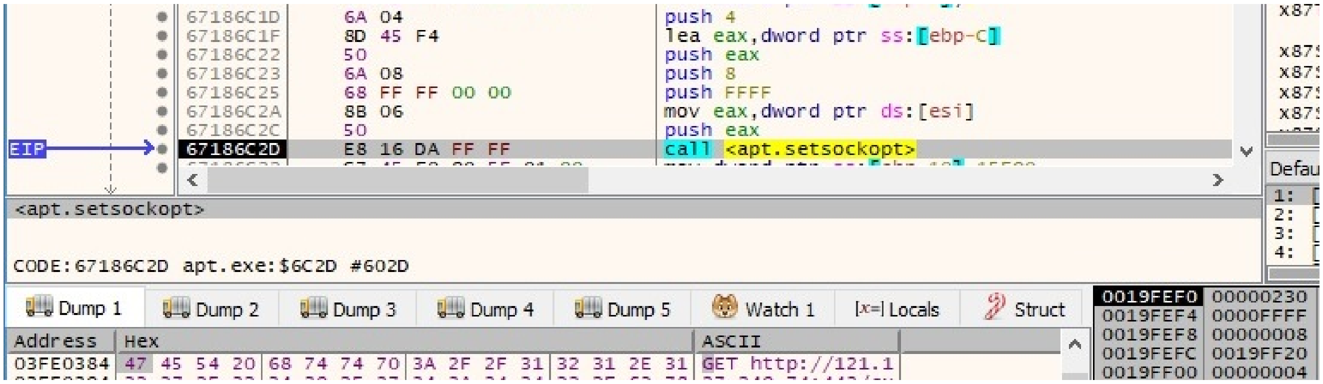

Figure 39

The second setsockopt call has different parameters – 0xFFFF (**SOL_SOCKET** – socket layer), 0x1006 (**SO_RCVTIMEO** – receive timeout), 0x15f90 = 90000ms = 90s (optval parameter):

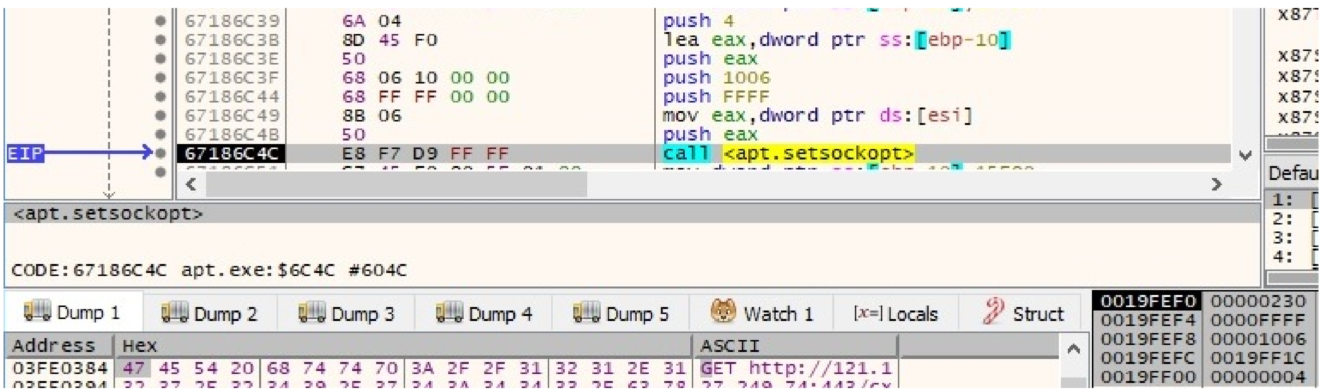

Figure 40

The third setsockopt call is different than the second one because it sets the send timeout to 90 seconds:

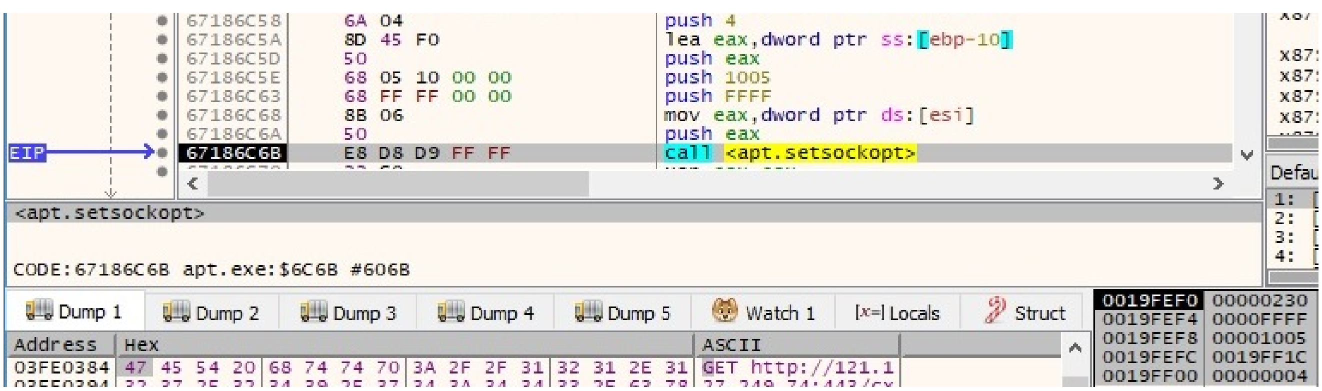

Figure 41

The port number 0x1BB is converted from TCP/IP network byte order to host byte order (little-endian on Intel processors) by using a ntohs function call:

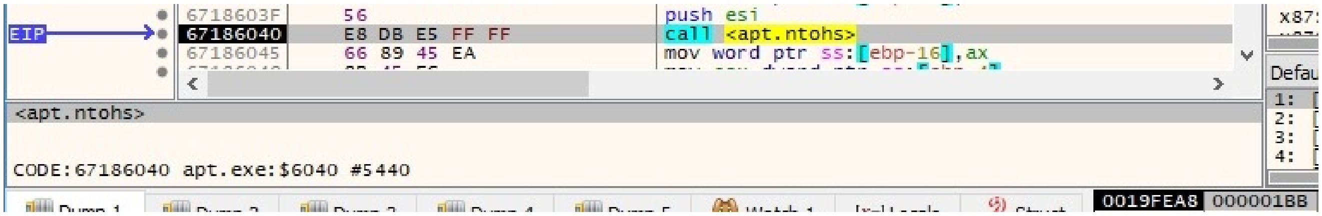

Figure 42

The malware is using the inet_addr function to transform the C2 IP address into a proper address for the IN_ADDR structure:

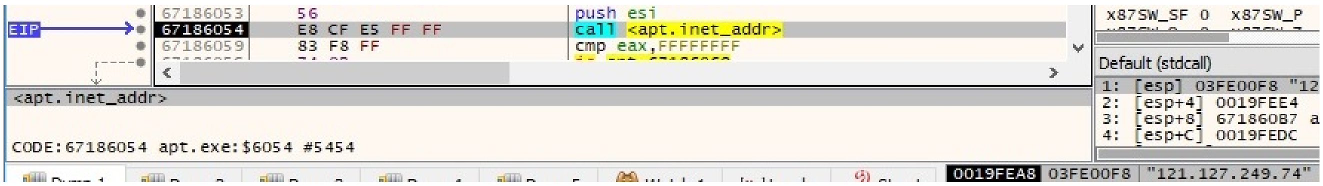

Figure 43

There is a network connection established to the C2 server using the connect function. The following elements can be highlighted in the sockaddr structure: 0x2 (**AF_INET** – IPv4 address family), 0x1BB = 443 (port number), 0x797FF94A (the C2 server represented as a hex value). The function call is represented in the next figure:

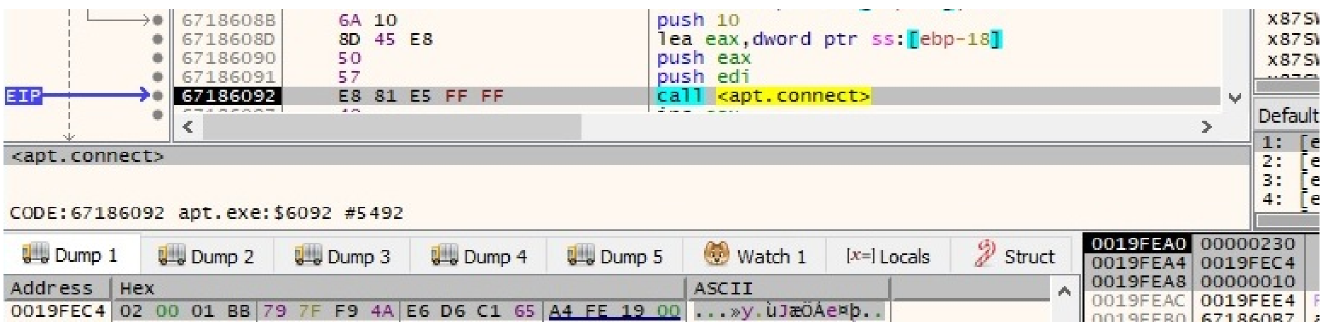

Figure 44

The sample performs a GET request to the C2 server with the user agent that was decrypted earlier: “User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; SV1)”. The data is sent using the send function:

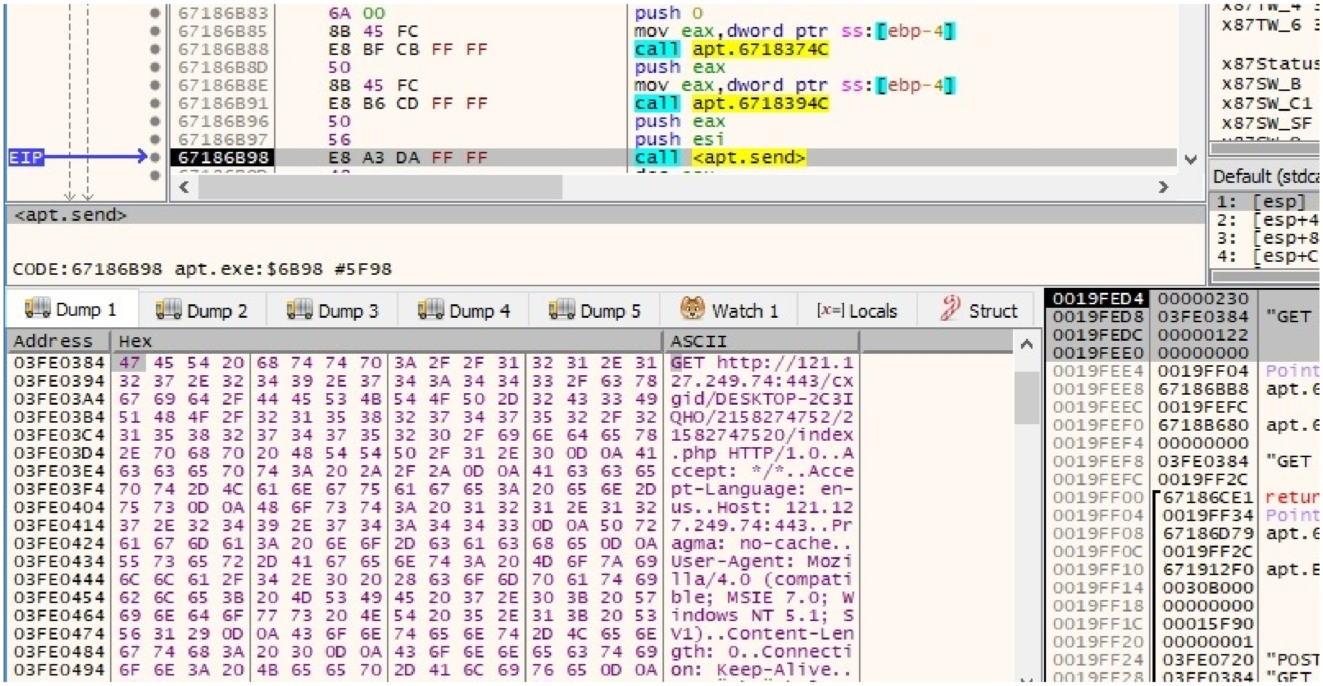
Figure 45

The malware reads the response from the server using the recv function, byte-by-byte (the length parameter is 1). It stops when the result contains "\x0d\x0a\x0d\x0a" (2 new lines characters in Windows) and it checks to see if the response contains "200 OK", which means that the connection was successfully established:

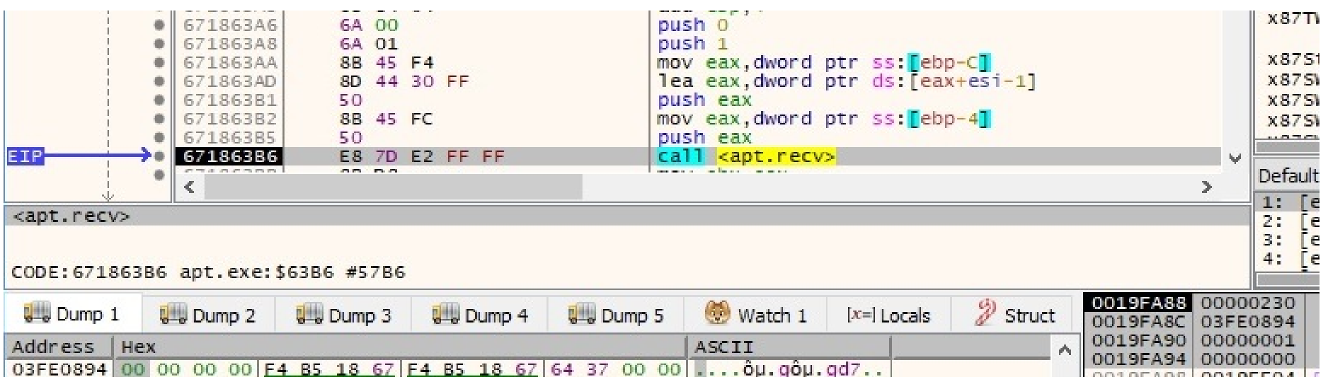
Figure 46

There is also a second comparison between the response and the "!!" string (if the result doesn't contain "!!", then the process performs a closesocket API call):

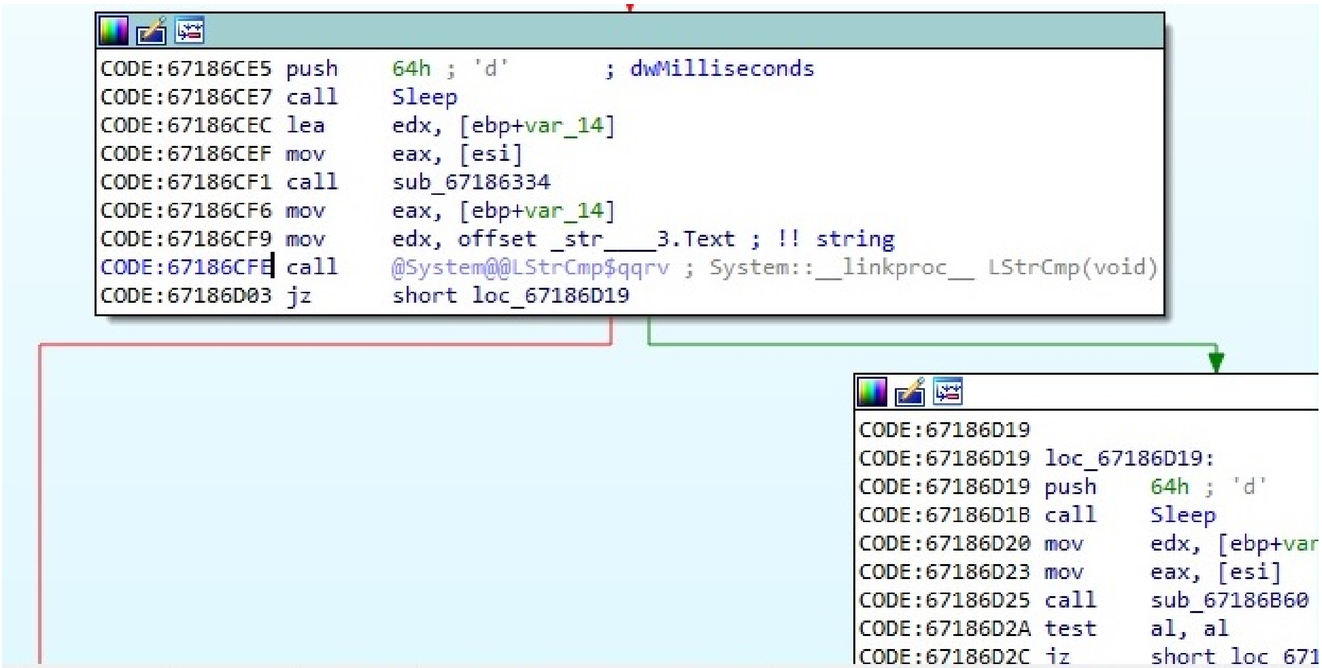

Figure 47

The hostname and the IP address of the local machine are exfiltrated to the C2 server using a POST request. The SessionID parameter is randomly generated:

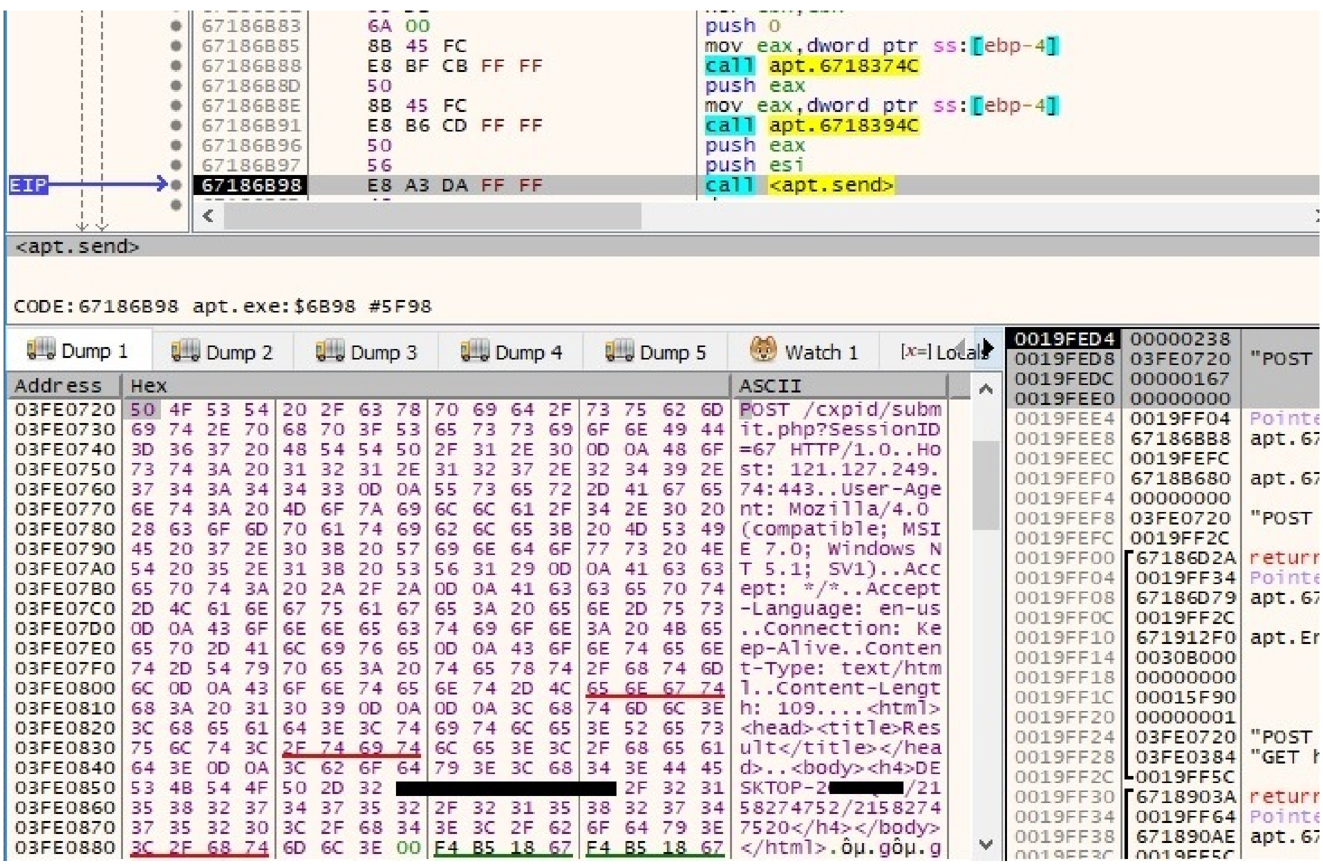

Figure 48

As before, there are multiple recv function calls following the POST request, and the process expects the response to contain “200 OK” and “Success”. If it doesn’t, then there is a Sleep call for 90 seconds and it tries again. A new thread is created using the CreateThread function:

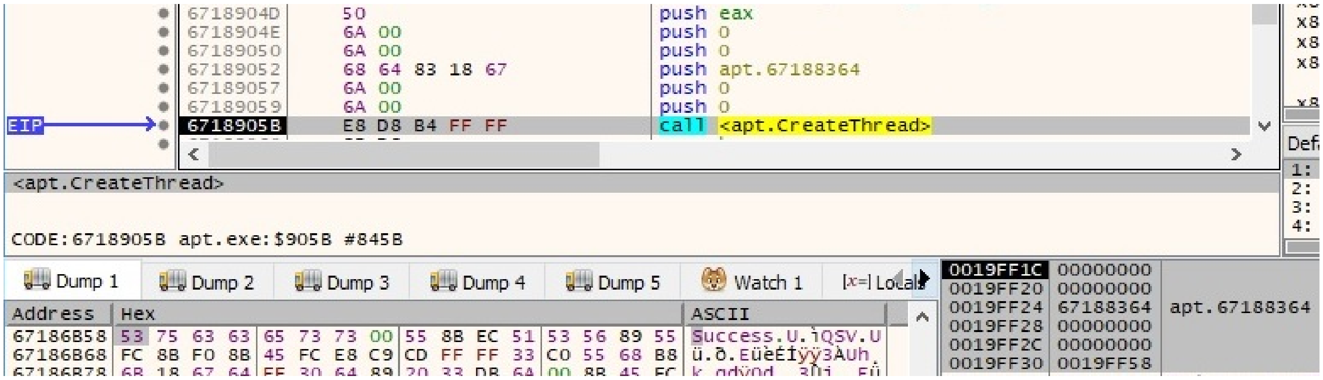

Figure 49
Thread activity

Some parameters used in the network communications like “id” and “SessionID” are generated by a function called “Randomize”:

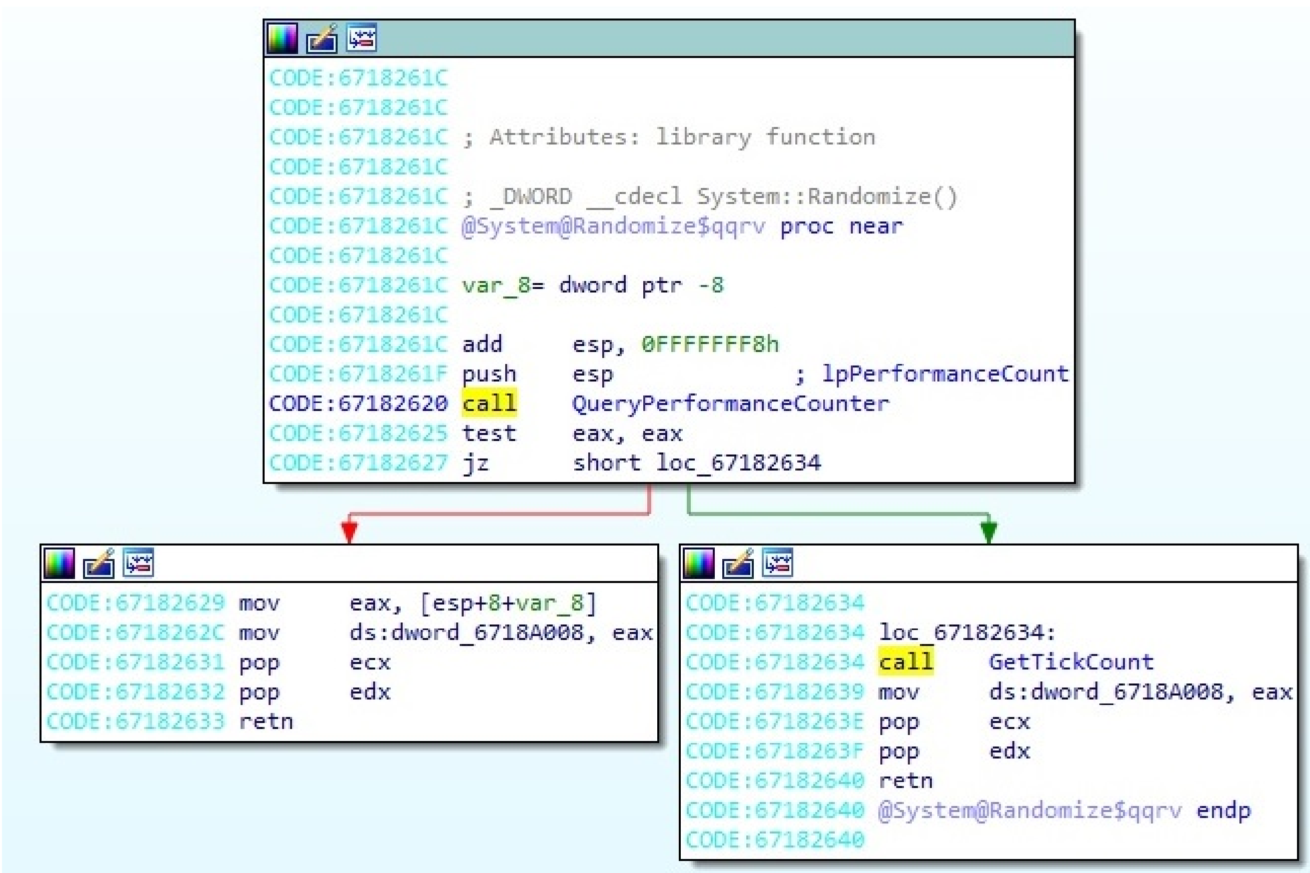

Figure 50
It’s important to mention that some HTTP headers are just decrypted before the network communication is performed using the algorithm described in the first paragraphs. The sample performs another GET request using the send function:

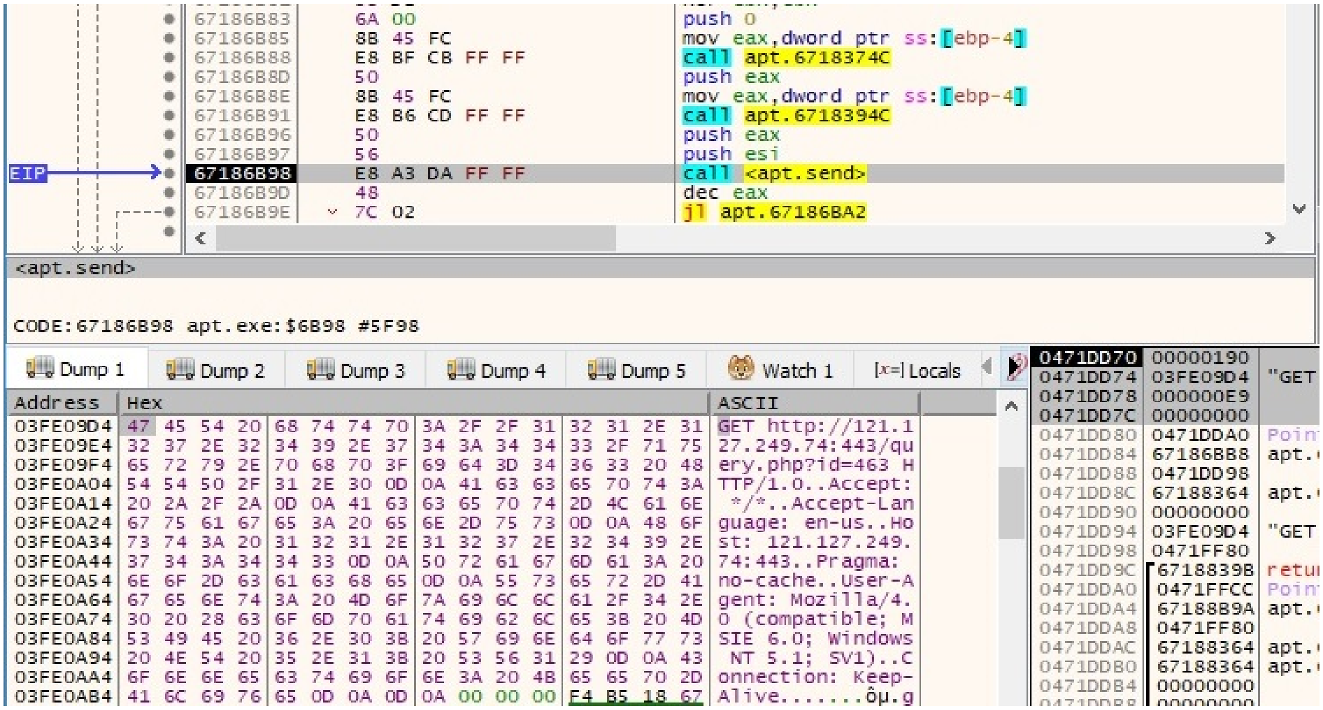

Figure 51

The file reads the response from the server using the recv function, byte-by-byte. It expects again a “200 OK” string and as opposed to before, it expects the response not to contain “!!” (if it does, the malware exits):

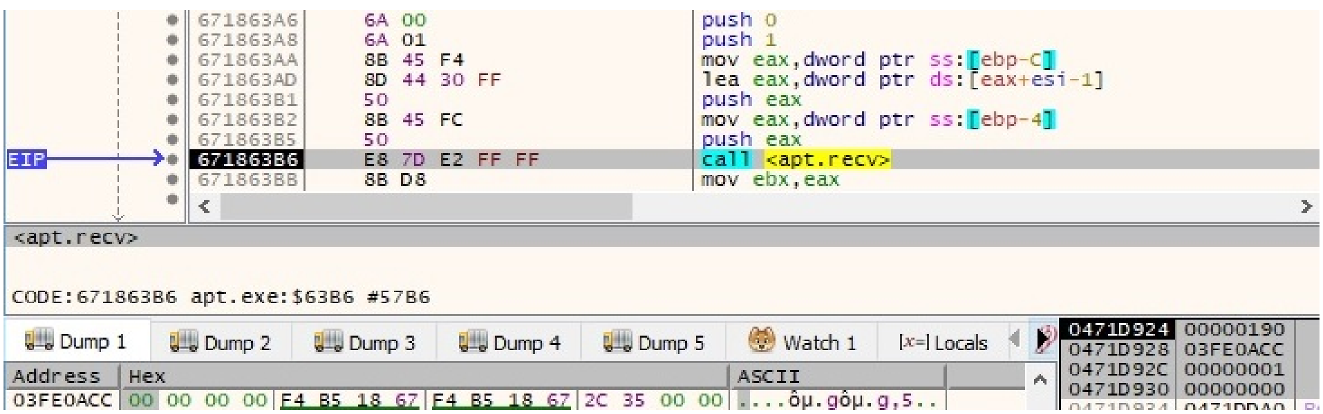

Figure 52

The process parses the response from the C2 server for an integer corresponding to a command that has to be executed. It implements 8 different commands, as shown in figure 53:

```
CODE:6718845F call    unknown_libname_66 ; BDS 2005-2007 and Delphi6-7 Visual Component Library
CODE:67188464 mov     ebx, eax
CODE:67188466 lea     eax, [ebp+var_24]
CODE:67188469 push    eax
CODE:6718846A mov     ecx, ebx
CODE:6718846C dec     ecx
CODE:6718846D mov     edx, 1
CODE:67188472 mov     eax, [ebp+var_30]
CODE:67188475 call    @System@@LStrCopy$qqrv ; System::__linkproc__ LStrCopy(void)
CODE:6718847A lea     eax, [ebp+var_30]
CODE:6718847D mov     ecx, ebx
CODE:6718847F mov     edx, 1
CODE:67188484 call    @System@@LStrDelete$qqrv ; System::__linkproc__ LStrDelete(void)
CODE:67188489 mov     edx, [ebp+var_30]
CODE:6718848C mov     eax, offset _str___10.Text
CODE:67188491 call    unknown_libname_66 ; BDS 2005-2007 and Delphi6-7 Visual Component Library
CODE:67188496 mov     ebx, eax
CODE:67188498 lea     eax, [ebp+var_28]
CODE:6718849B push    eax
CODE:6718849C mov     ecx, ebx
CODE:6718849E dec     ecx
CODE:6718849F mov     edx, 1
CODE:671884A4 mov     eax, [ebp+var_30]
CODE:671884A7 call    @System@@LStrCopy$qqrv ; System::__linkproc__ LStrCopy(void)
CODE:671884AC mov     eax, [ebp+var_1C]
CODE:671884AF call    unknown_libname_75 ; BDS 2005-2007 and Delphi6-7 Visual Component Library
CODE:671884B4 cmp     eax, 7          ; switch 8 cases
CODE:671884B7 ja      def_671884BD    ; jumptable 671884BD default case
```

Figure 53

Case 1 – EAX = 0

The process sends a POST request to the server that contains a similar HTML document, however the exfiltrated information is different. The following bytes can be highlighted: CF 83 CD 83 CF 83, on which we can apply a NOT operation and obtain 30 7C 32 7C 30 7C (0|2|0|):

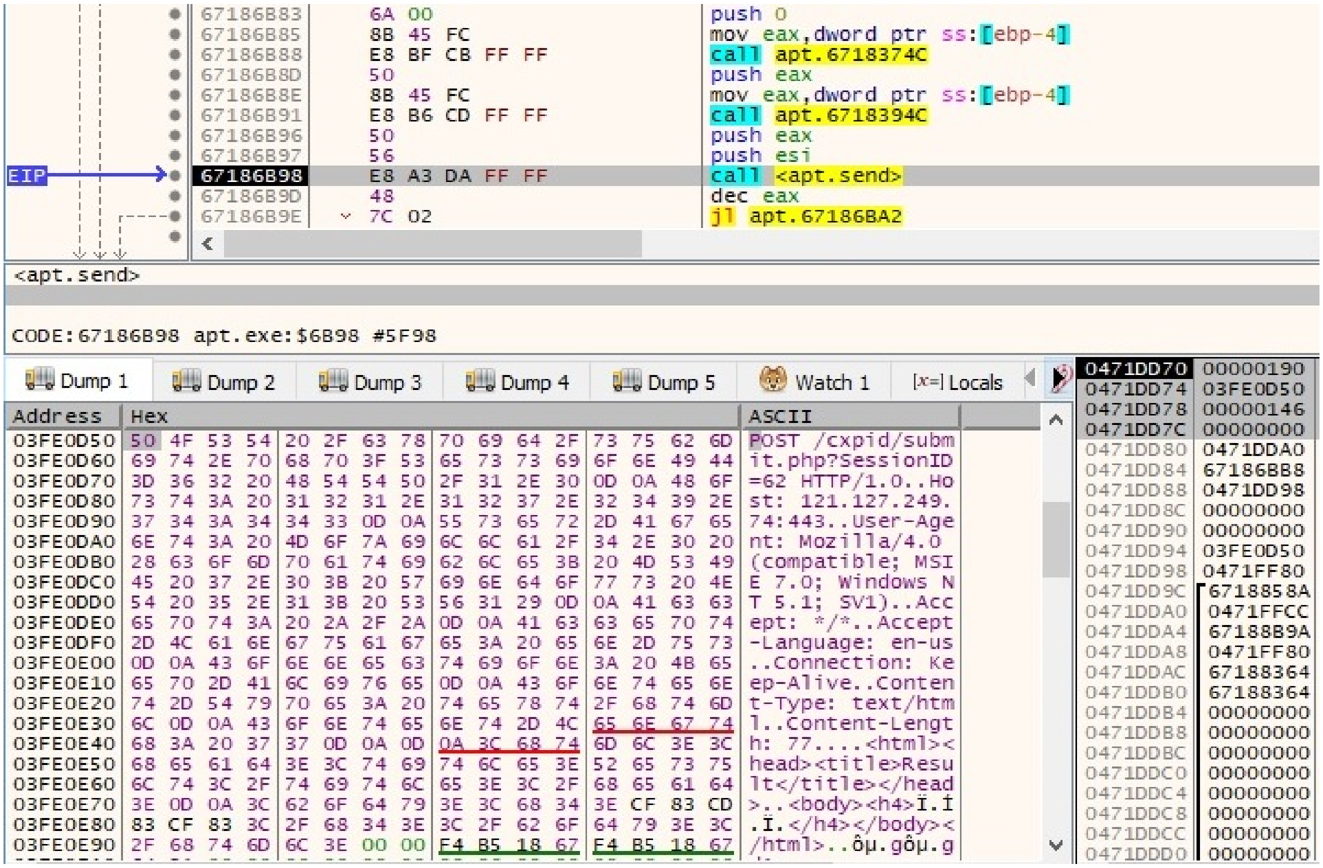

Figure 54

The reponse from the server is received using the recv function. If the connection was successful, the process expects a “200 OK” string and also “Success”, as shown below:

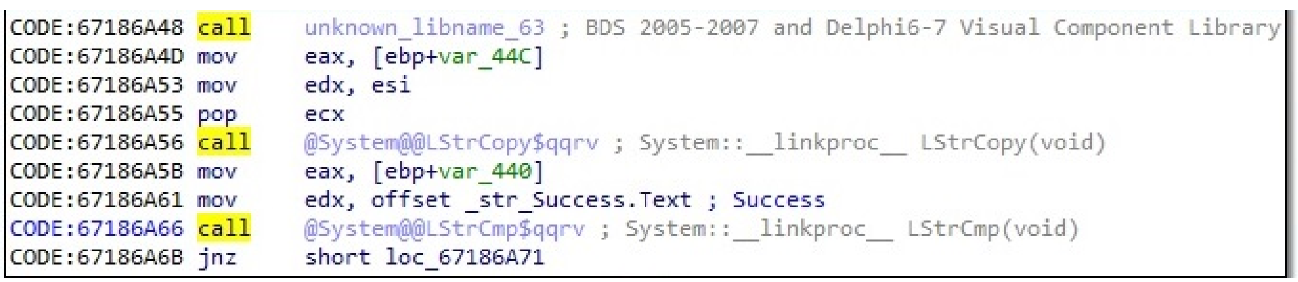

Figure 55

There is another GET request to the CnC server performed by the malicious process:

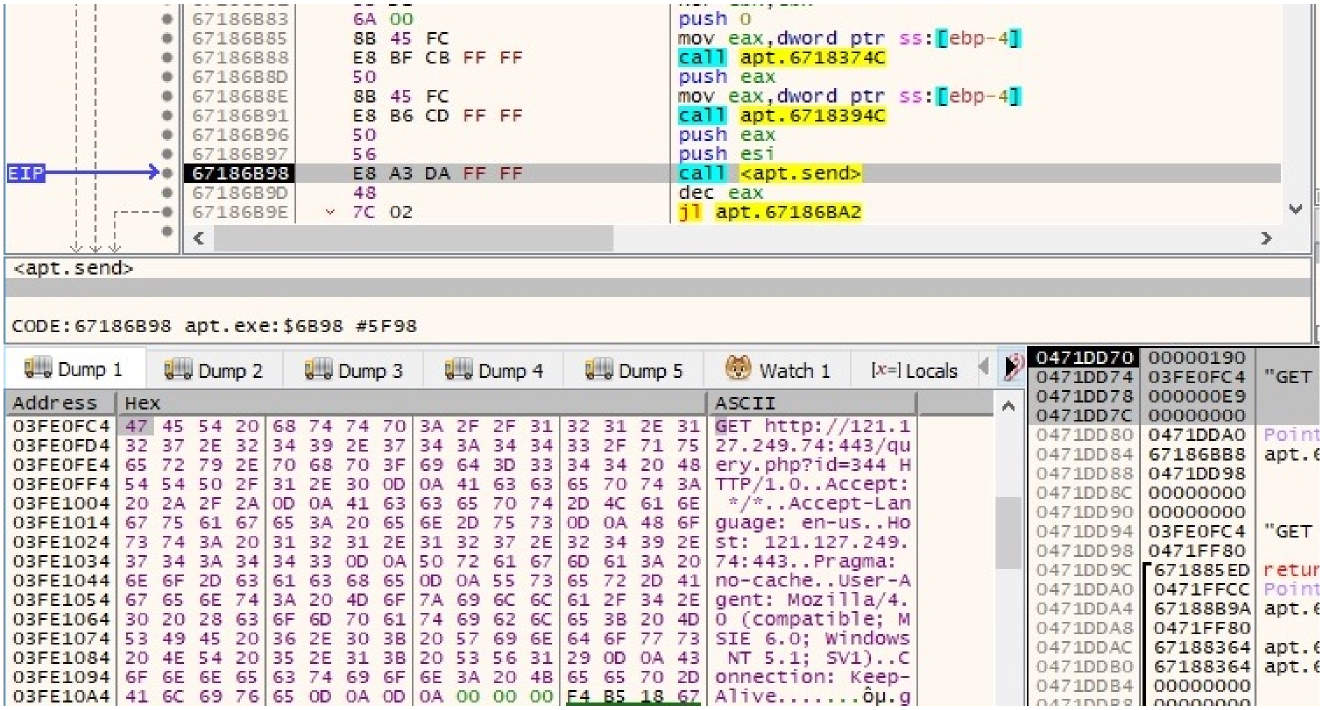

Figure 56

The response from the server is expected to be larger this time (0x1000 = 4096 bytes):

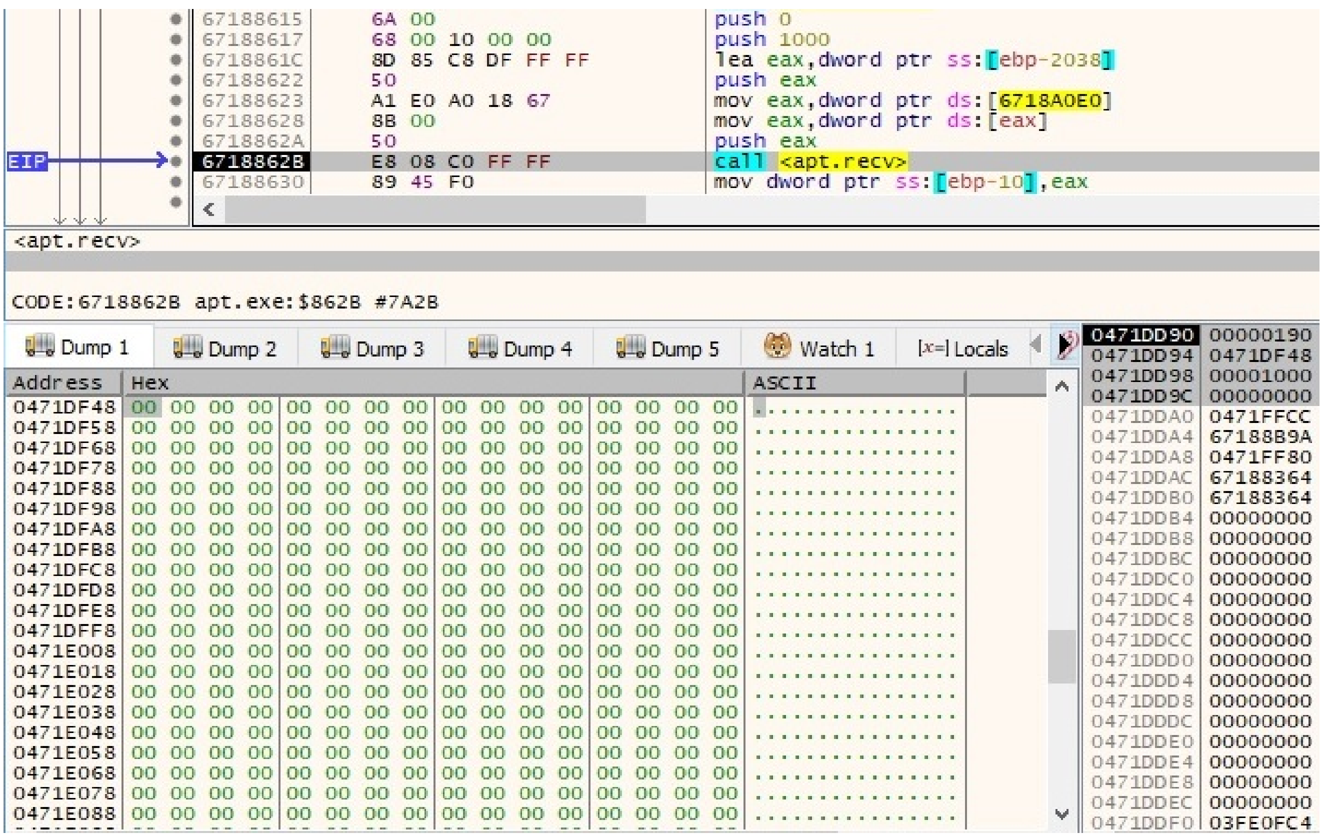

Figure 57

The response from the server is written to a file specified by a handle transmitted by the C2 server (in our case, this was 0 because we're trying to emulate the C2 server communications). The WriteFile API call is presented below:

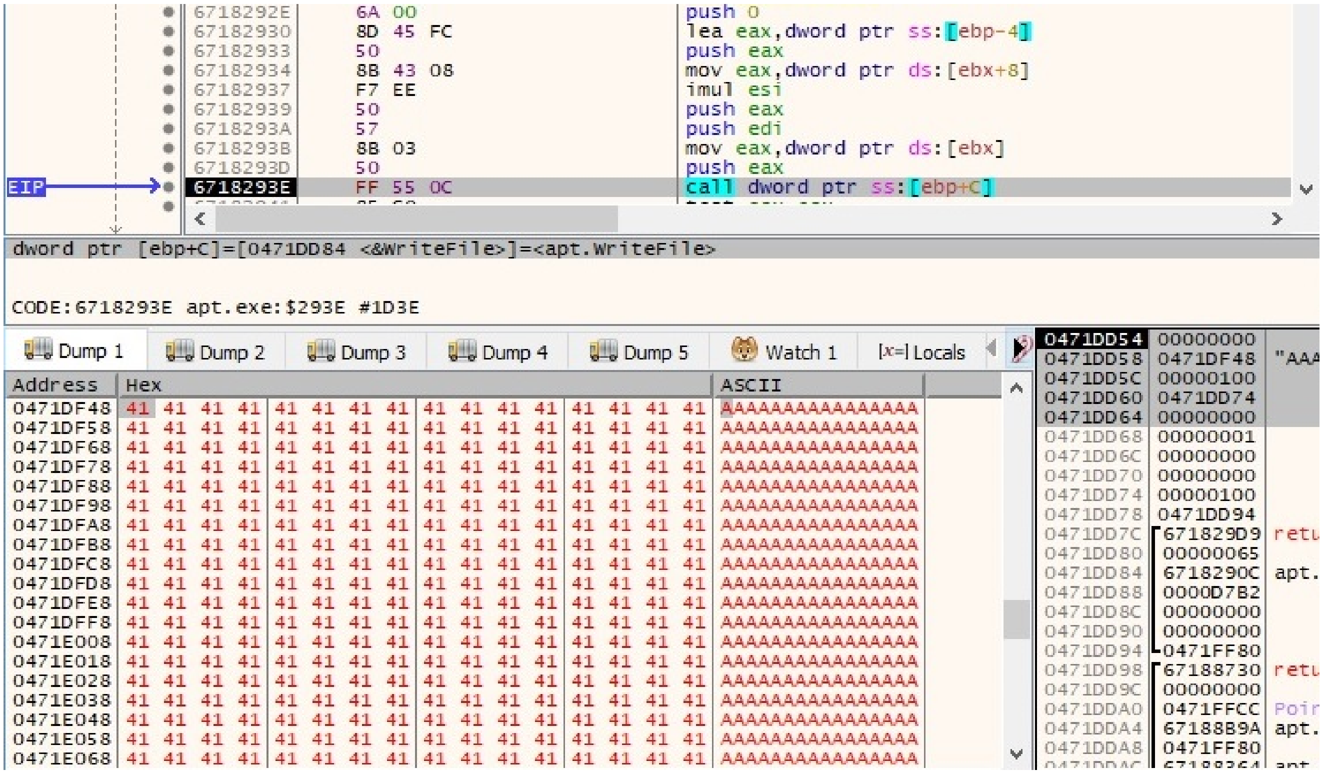

Figure 58

The process announces the C2 server that the write operation was successful by issuing a POST request (NOT (CF 83 CE 83 CF 83) = 30 7C 31 7C 30 7C = “0|1|0|”):

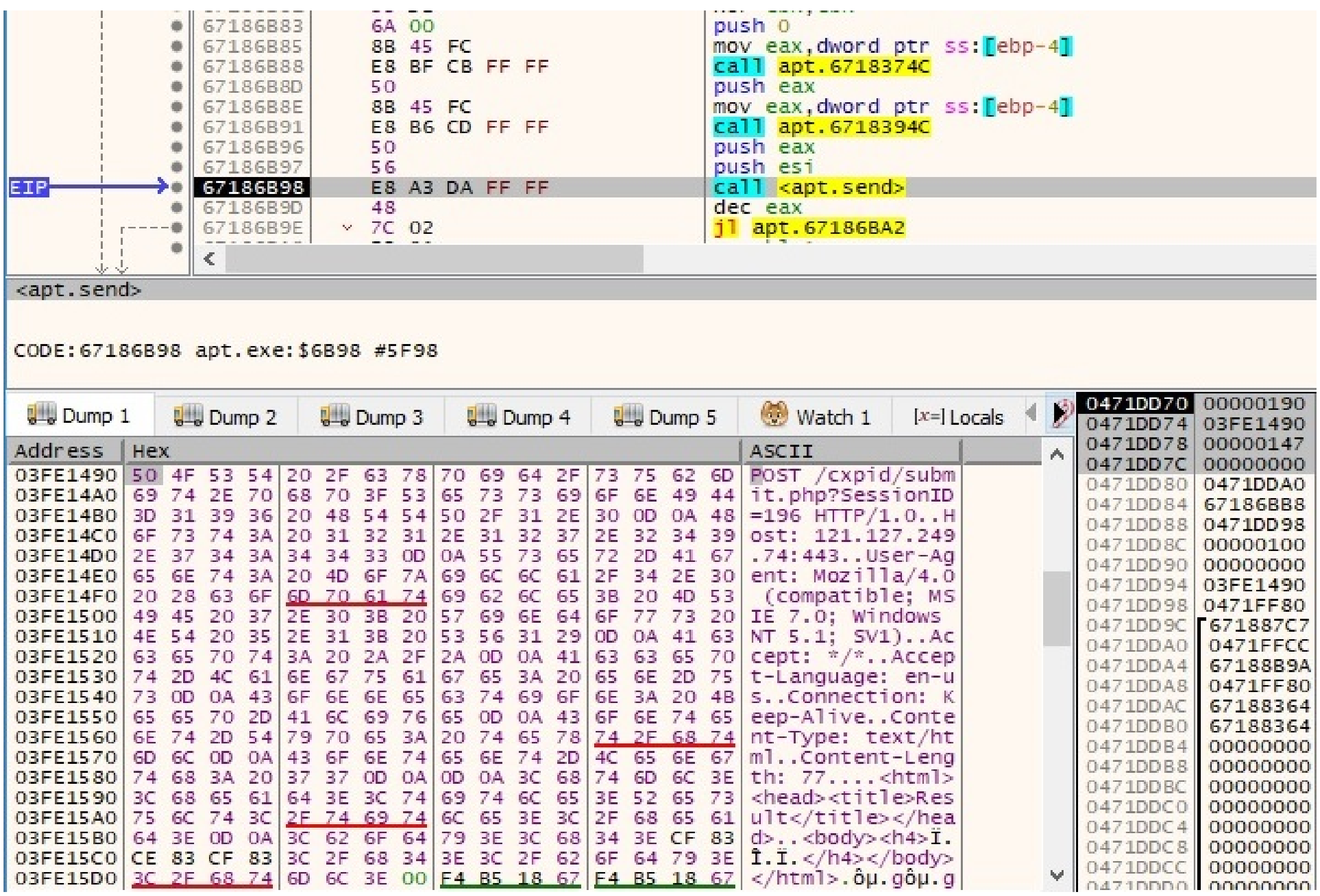

Figure 59

If the write operation failed, the request is changing (NOT (CF 83 CF 83 CF 83) = 30 7C 30 7C 30 7C = “0|0|0|”):

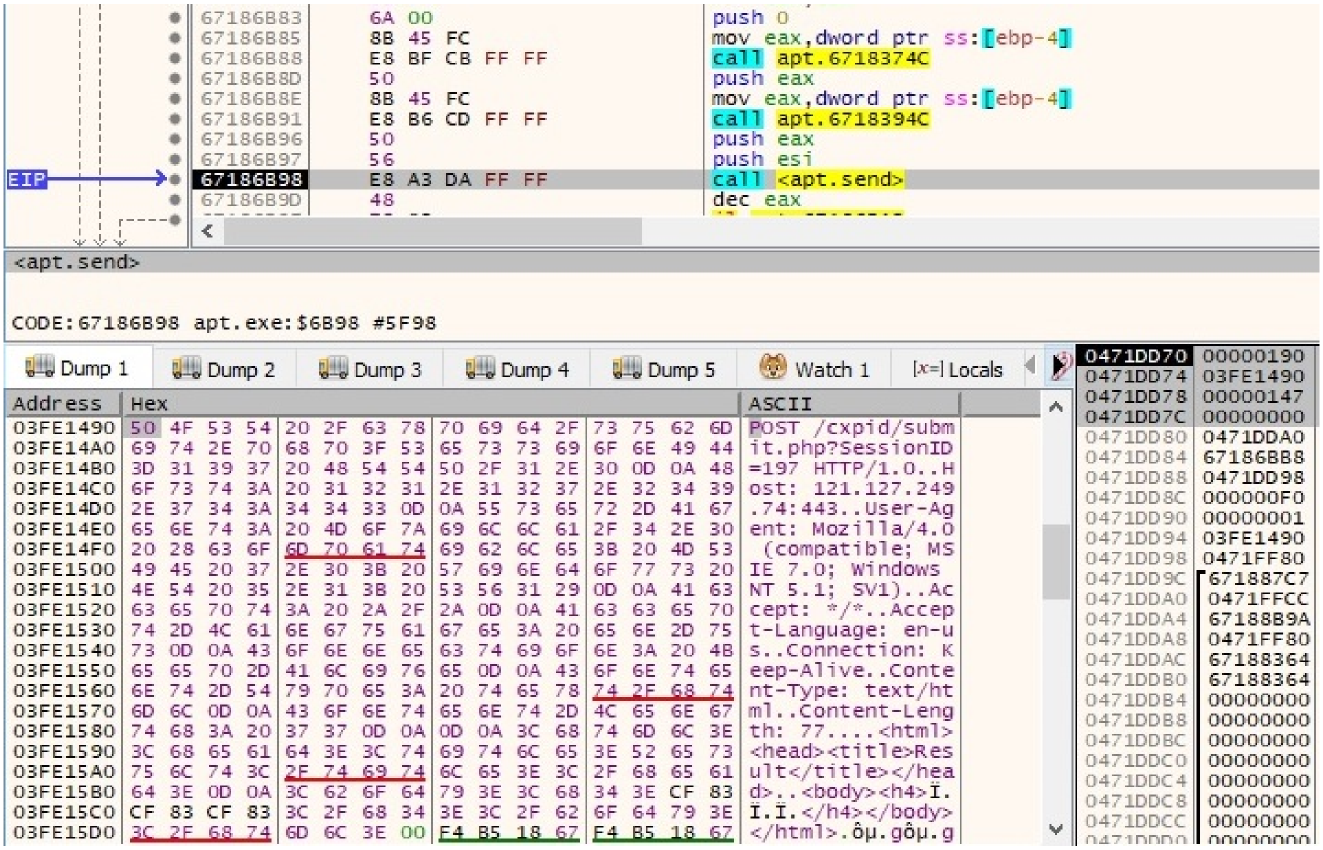

Figure 60

An identical GET request, as presented before, is sent to the server and the malware jumps back to the switch statement (this applies to each case).

Case 2 – EAX = 1

In this case, we have 2 subcases depending on the response from the server. In the first one, the only thing that is exfiltrated to the CnC server is the current directory, which can be obtained by applying a NOT operation:

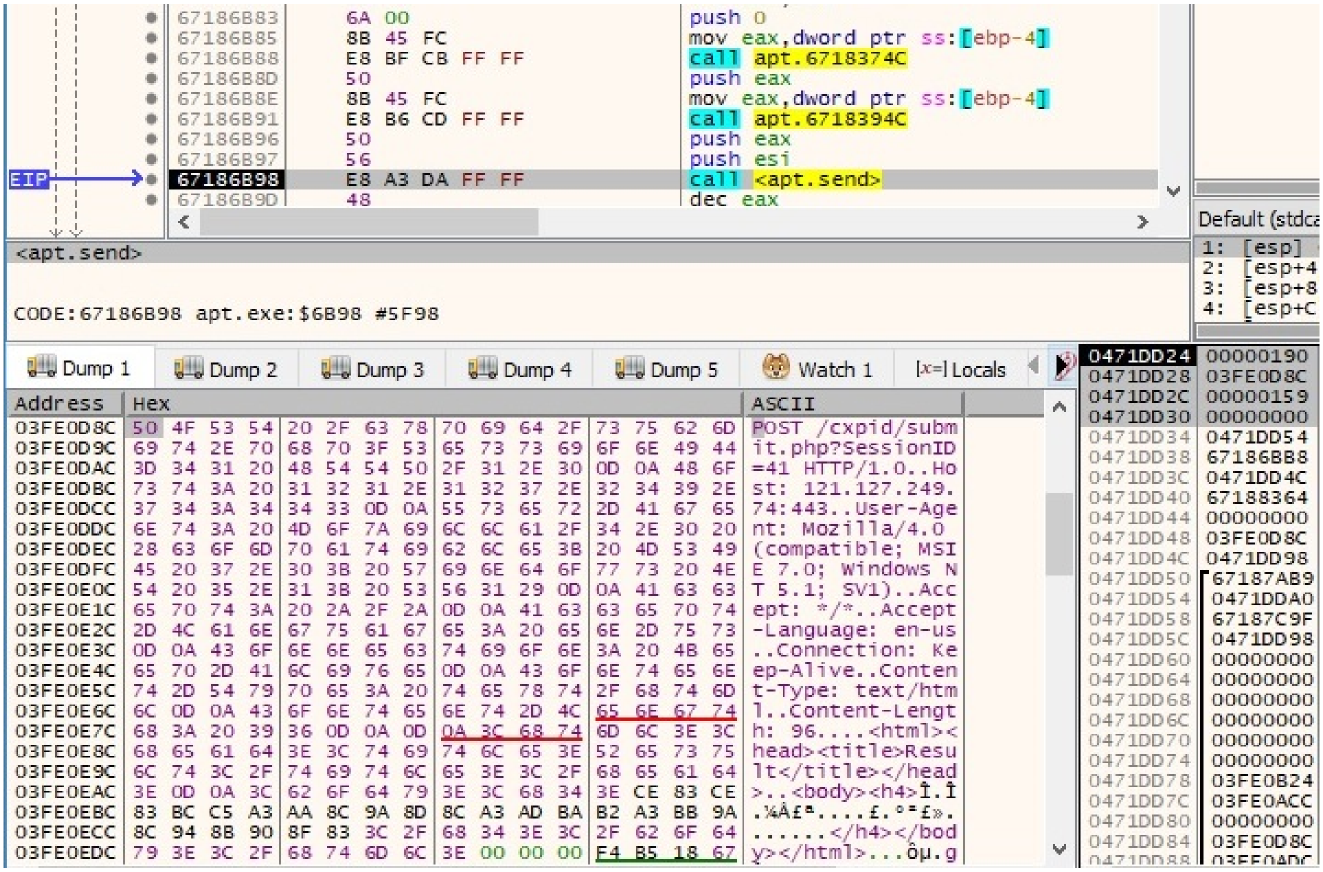

Figure 61

In the second subcase, the malware scans the current directory using the FindFirstFileA and FindNextFileA functions:

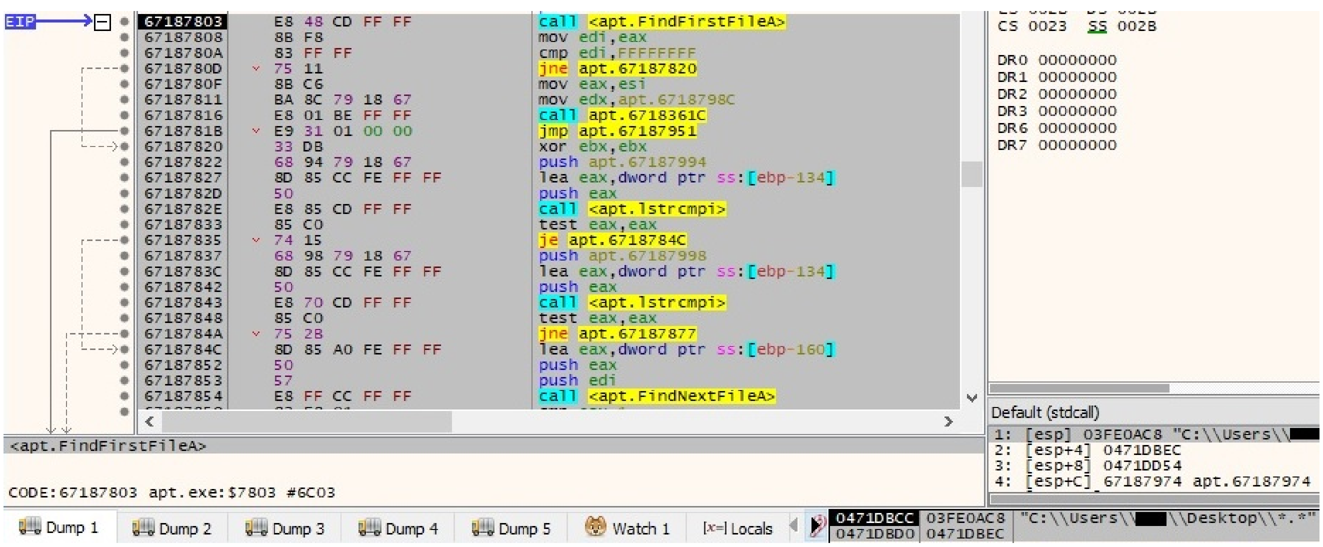

Figure 62

Each file time is extracted and converted to a local file time by using the FileTimeToLocalFileTime API:

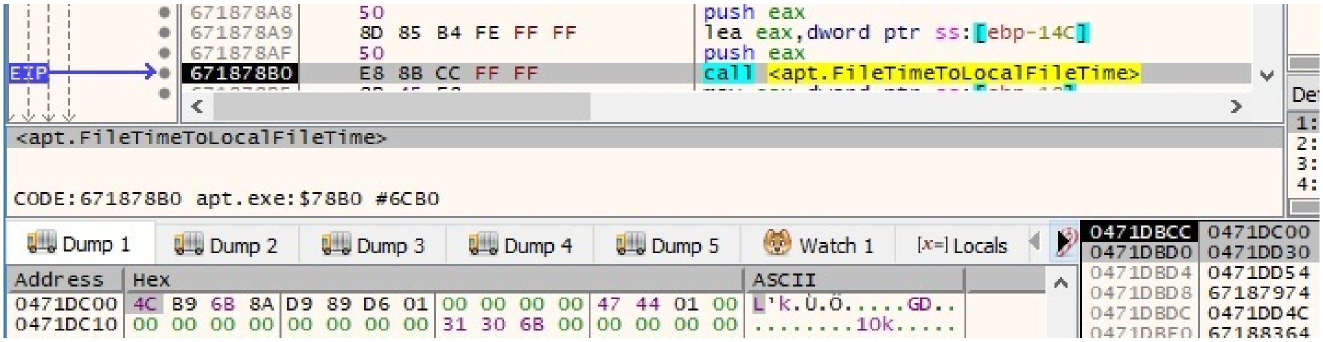

Figure 63

The process constructs the next buffer for every file: 1|File name|dwHighDateTime (high-order 32 bits of the file time) in decimal|File size in decimal|. An example of such buffer is presented in the next picture:

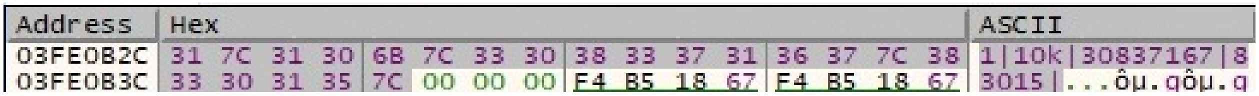

Figure 64

After the process succeeds in applying the algorithm for every file in the current directory, the final buffer looks like the following:

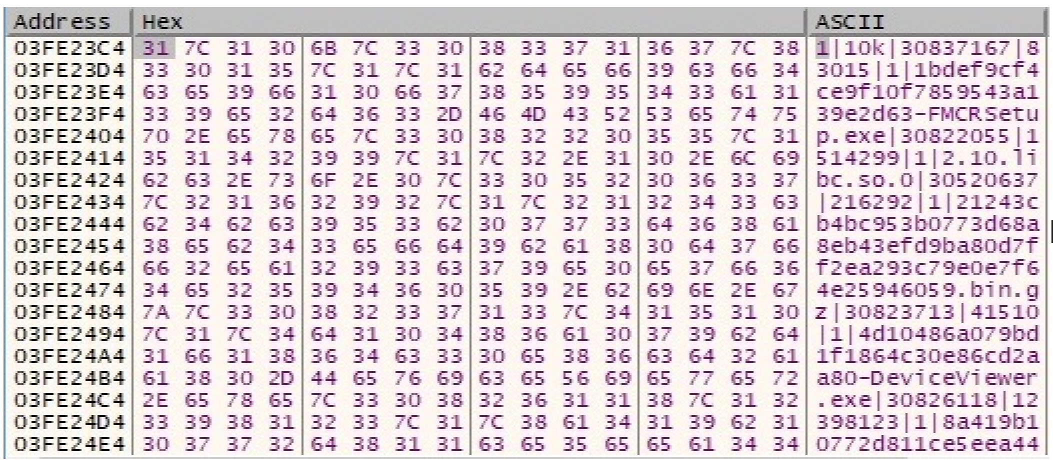

Figure 65

The buffer is encoded using the NOT operator and is exfiltrated to the C2 server via a POST request:

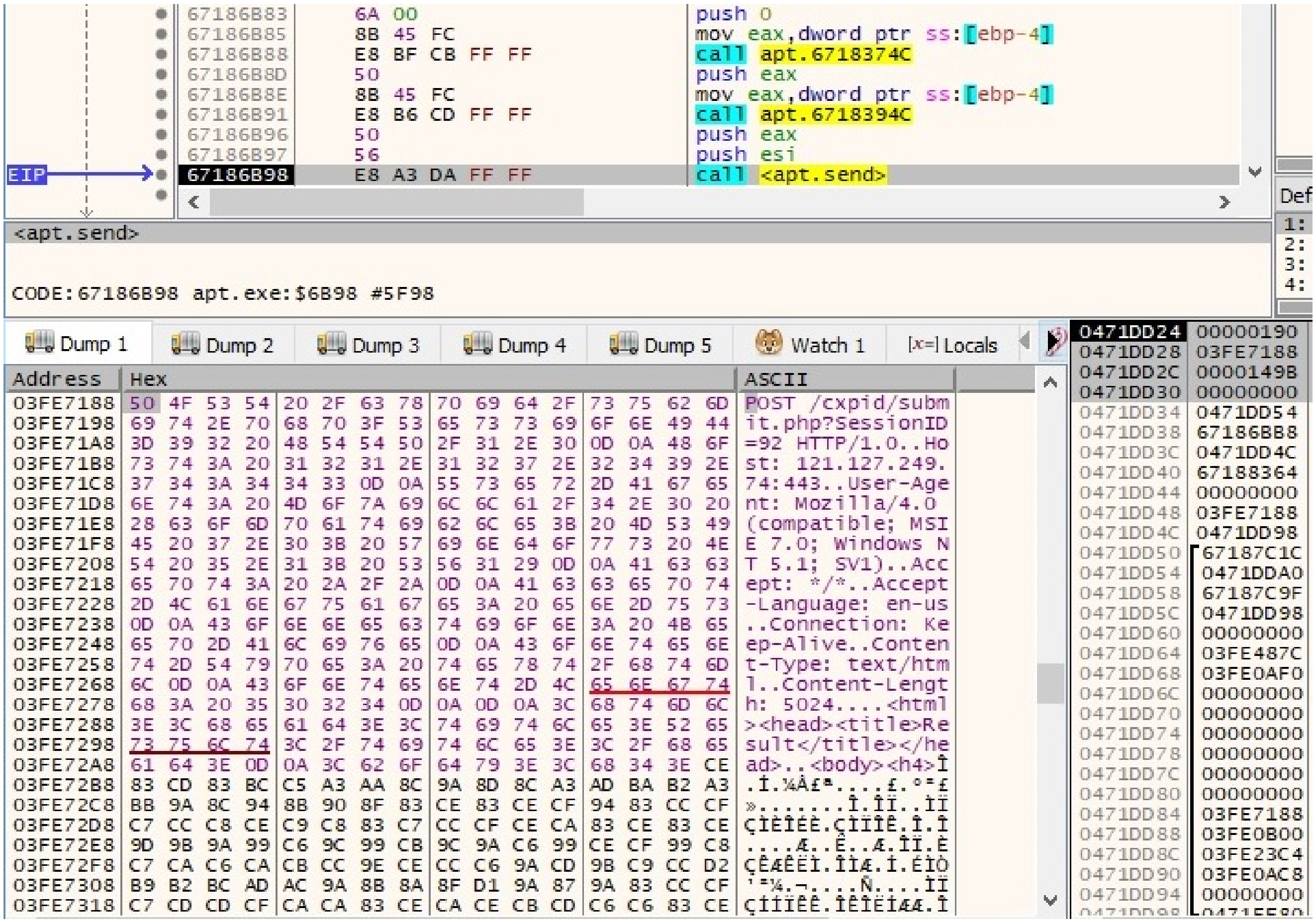

Figure 66

Case 3 – EAX = 2

By parsing the response from the server to obtain the command line to be executed, there is a new process created using the CreateProcessA function:

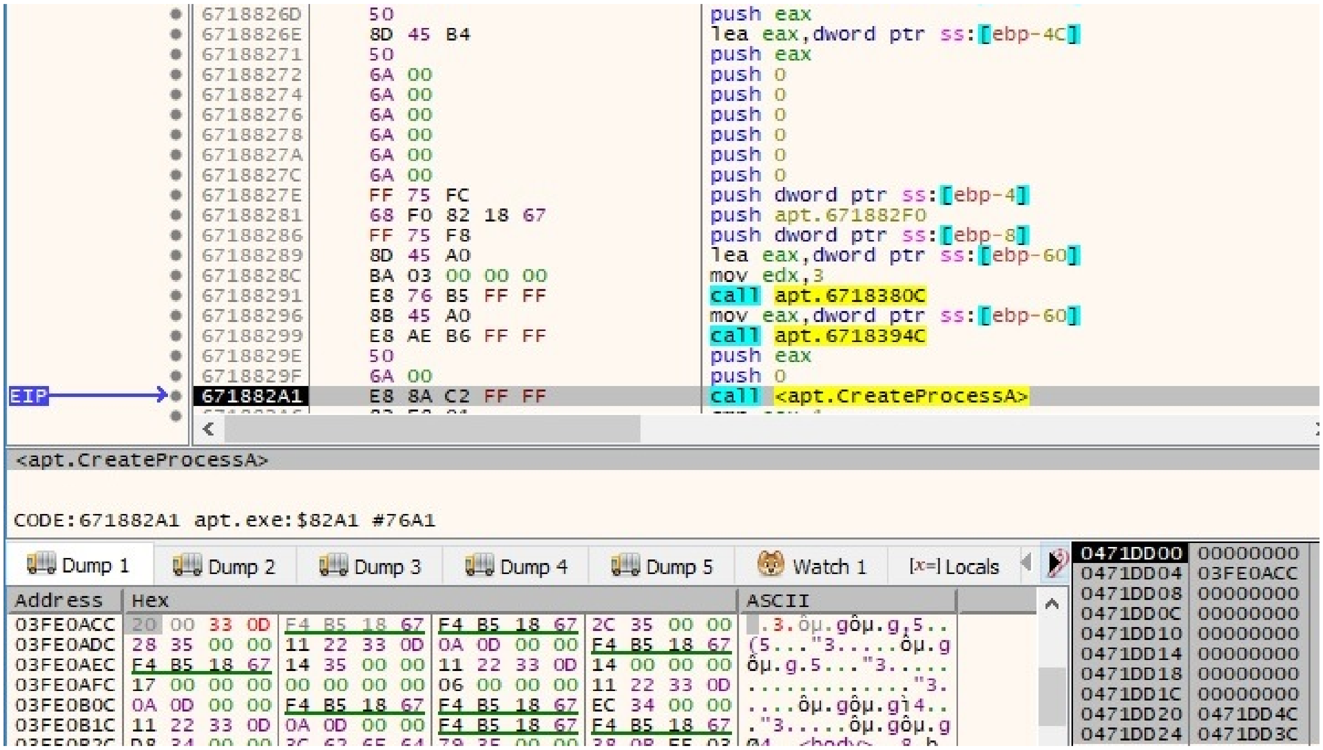

Figure 67

If the new process was successfully created, the following request is made to the CnC server (NOT (CD 83 CE 83 CF 83) = 32 7C 31 7C 30 7C = “2|1|0|”):

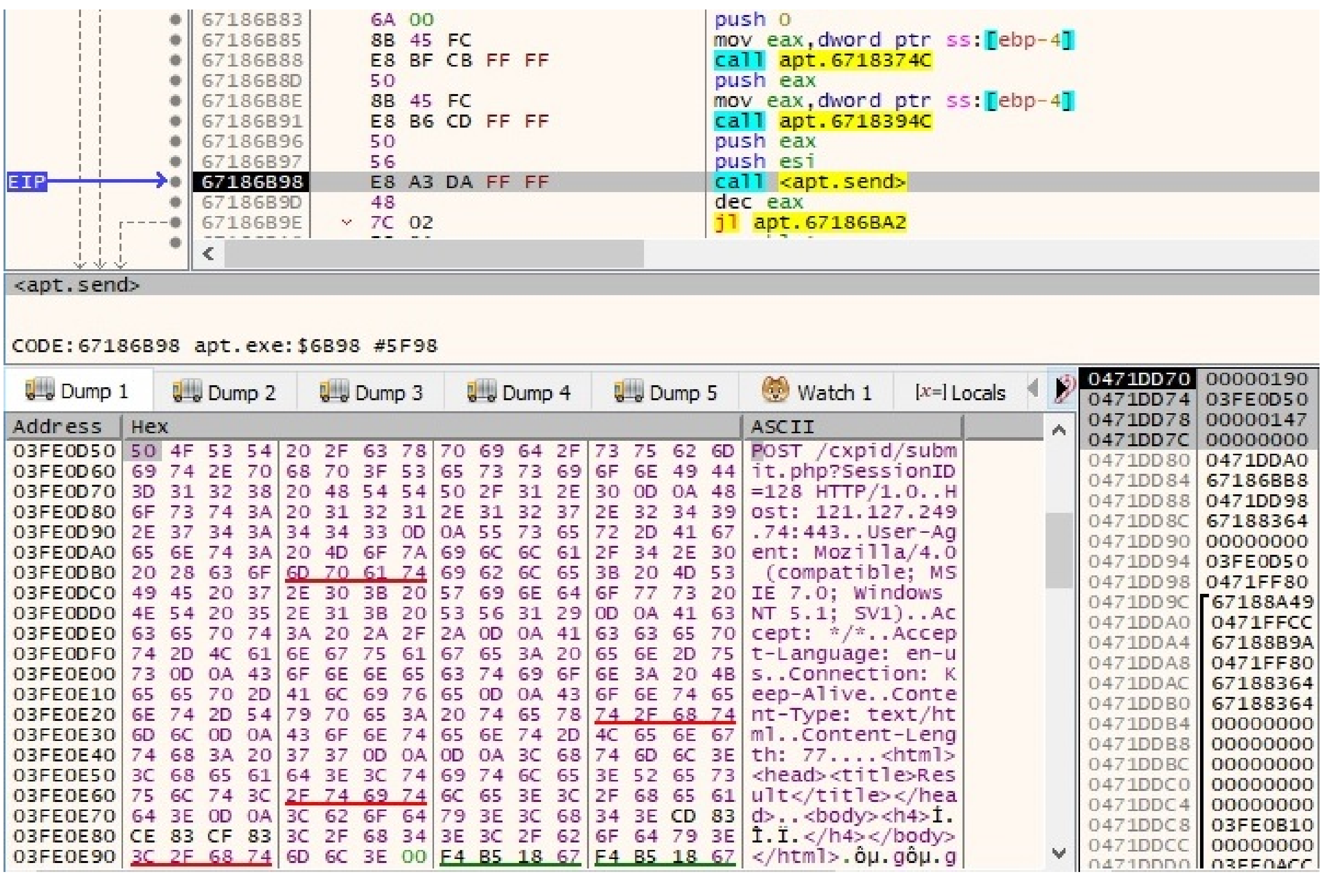

Figure 68

Whether any error occurred during the process creation, the POST request is different (NOT (CD 83 CF 83 CF 83) = 32 7C 30 7C 30 7C = “2|0|0|”):

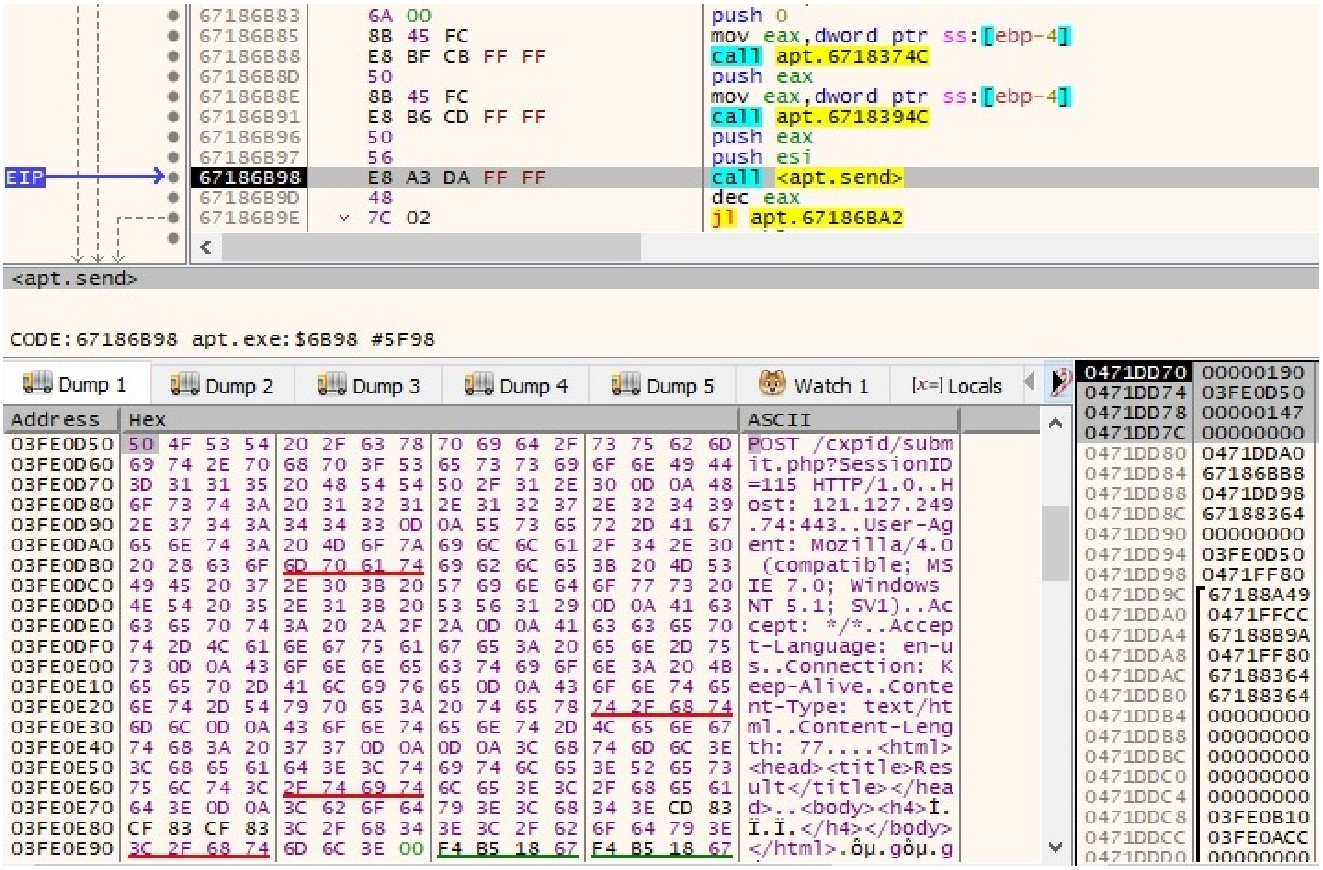

Figure 69

Case 4 – EAX = 3

We have only observed a POST request performed by the malware (NOT (CC 83 CE 83 CF 83) = 33 7C 31 7C 30 7C = “3|1|0|”):

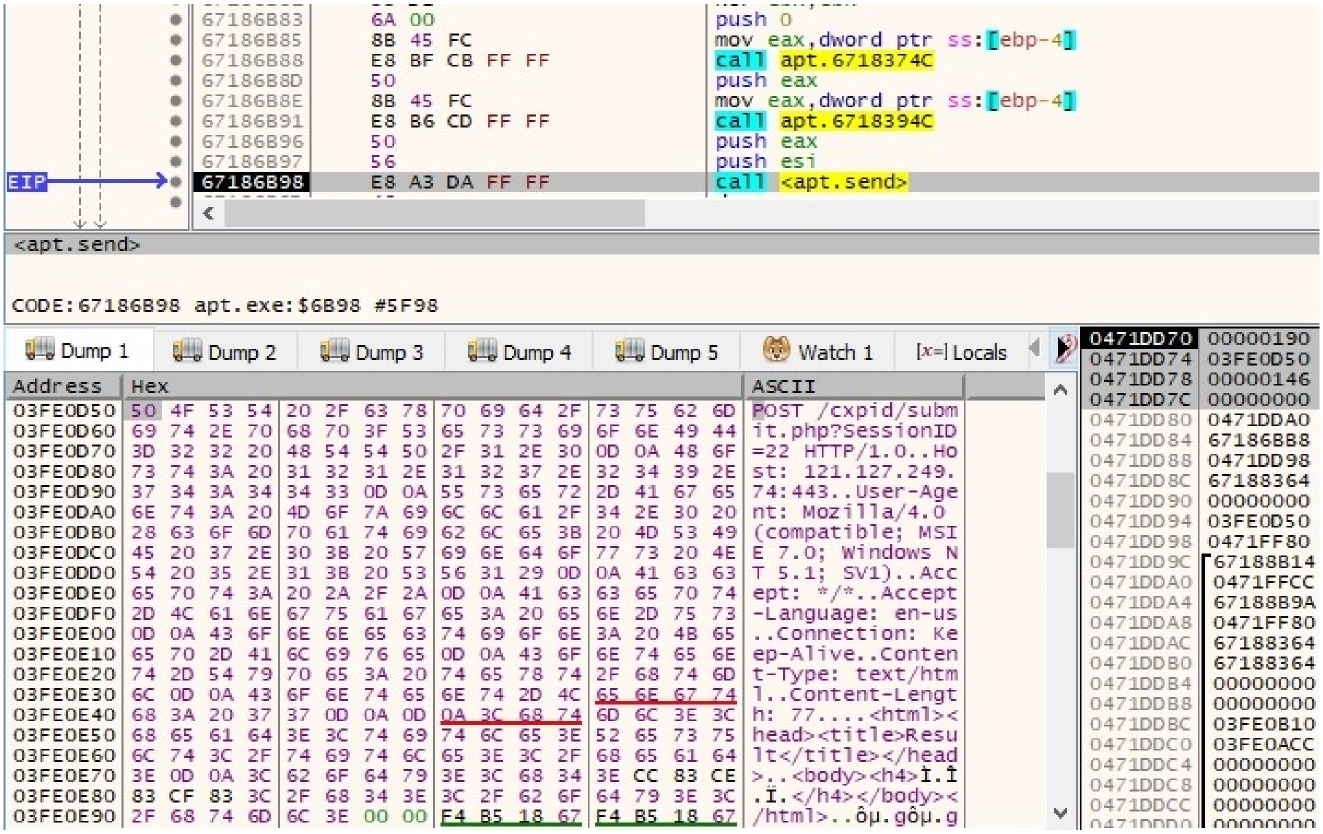

Figure 70

Case 5 – EAX = 4

The server provides a file name to be opened by the malicious process. This action might indicate that the attacker tries to exfiltrate the content of targeted files:

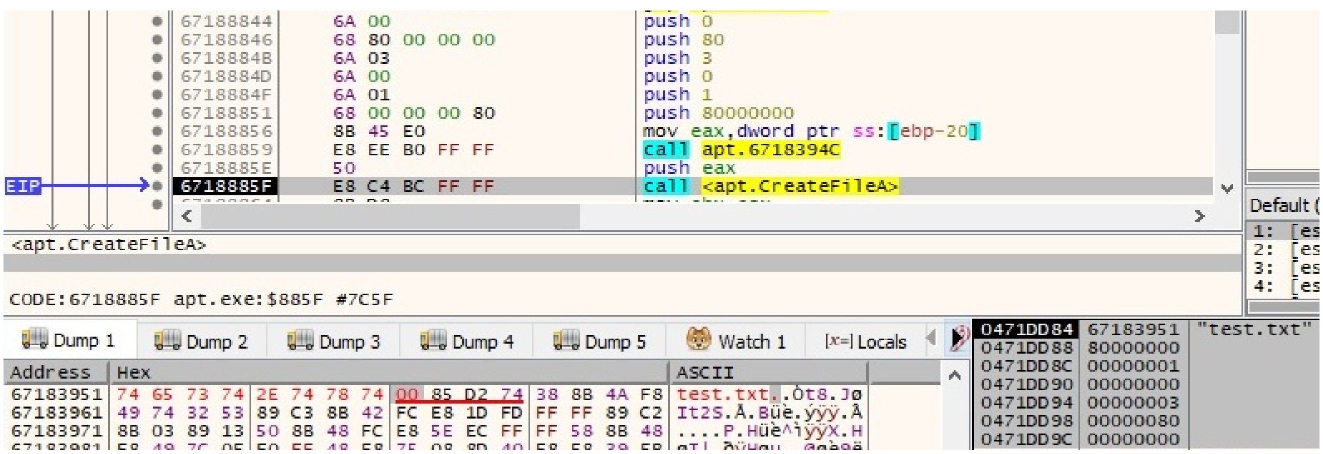

Figure 71

A POST request is performed by the file, the user agent is the same as in every network communication:

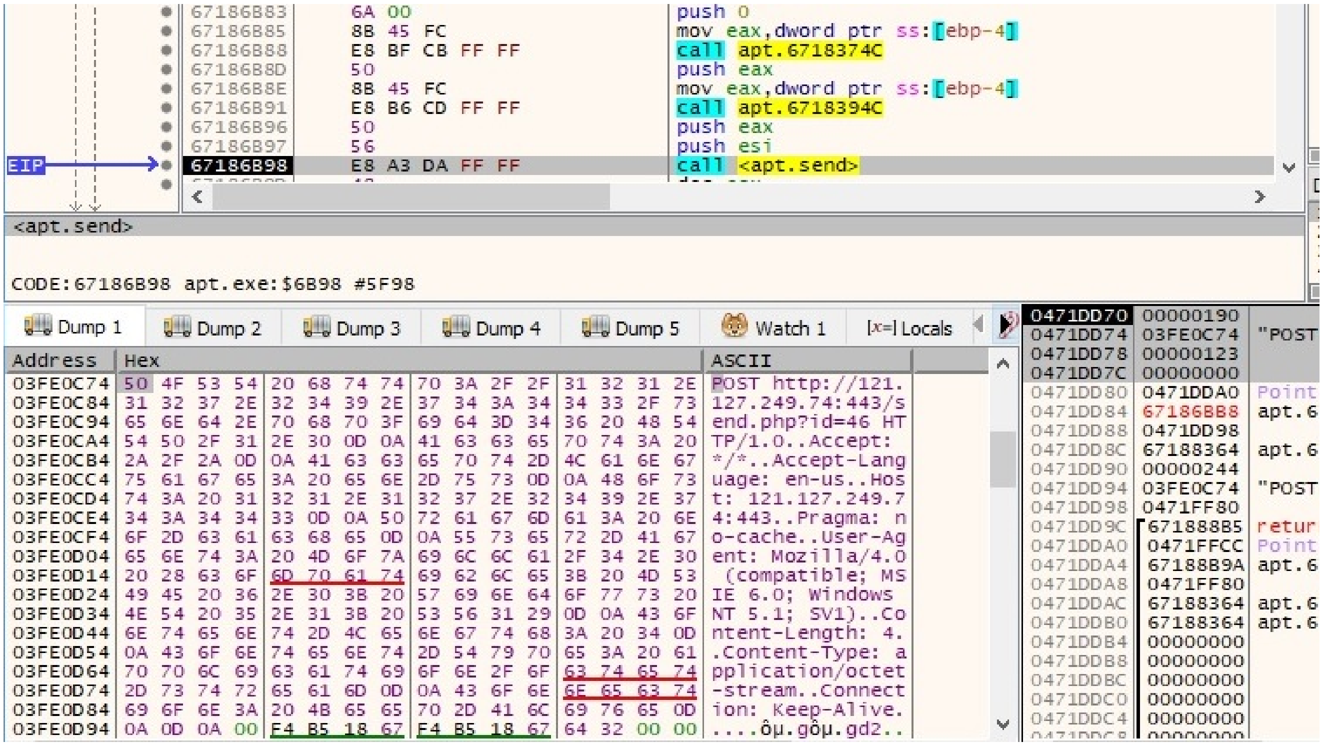

Figure 72

The process reads the content of the specified file by using a ReadFile function call:

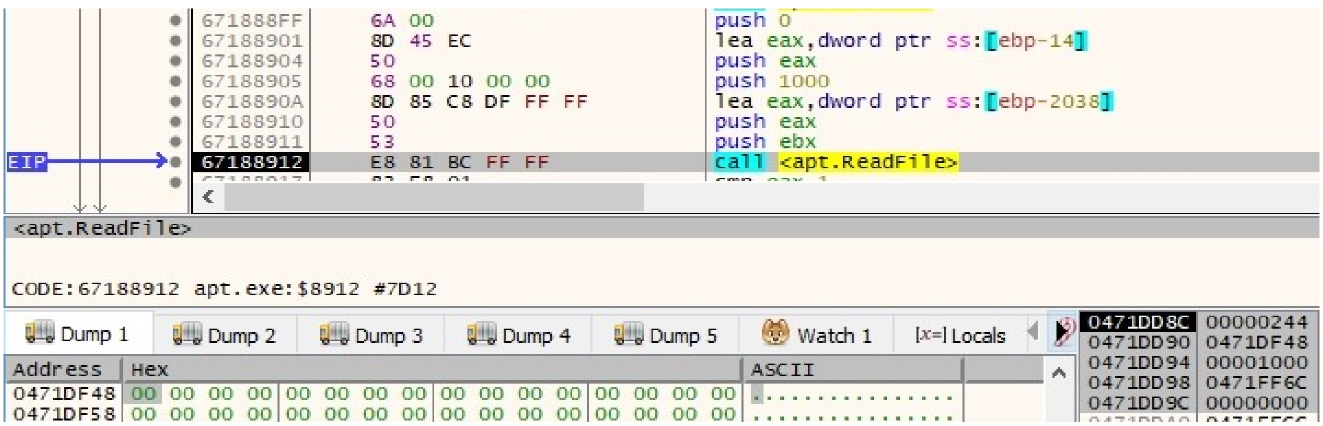

Figure 73

The content of the targeted file is exfiltrated to the CnC server using the send function:

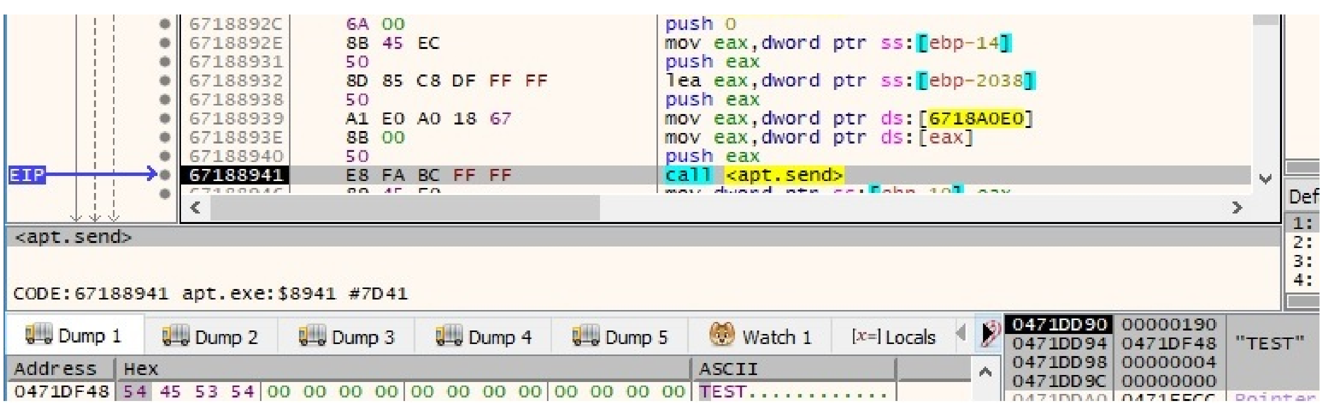

Figure 74

Case 6 – EAX = 5

We believe that this command is responsible for downloading other malware payloads. There is only a GET request to the same C2 server:

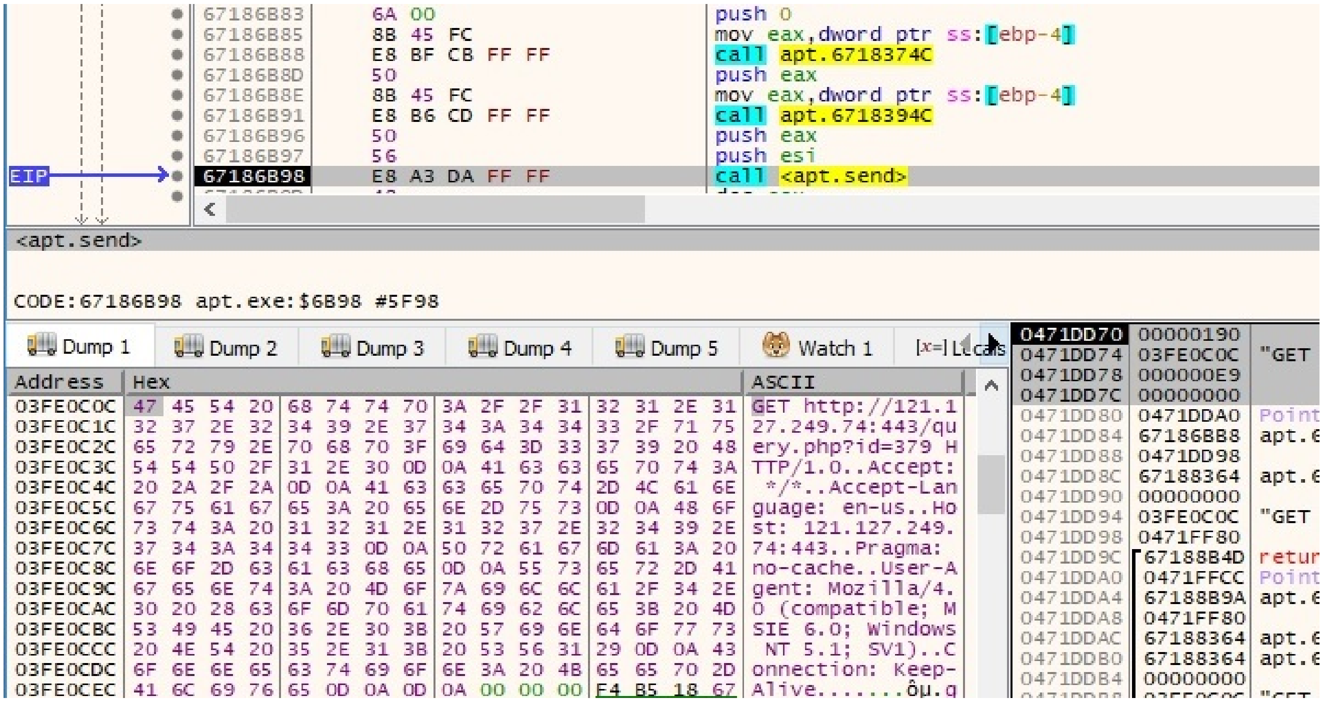

Figure 75

Case 7 – EAX = 6

The CreateToolhelp32Snapshot API is utilized to take a snapshot of the processes, the first parameter being 0x2 (**TH32CS_SNAPPROCESS** – all processes in the system):

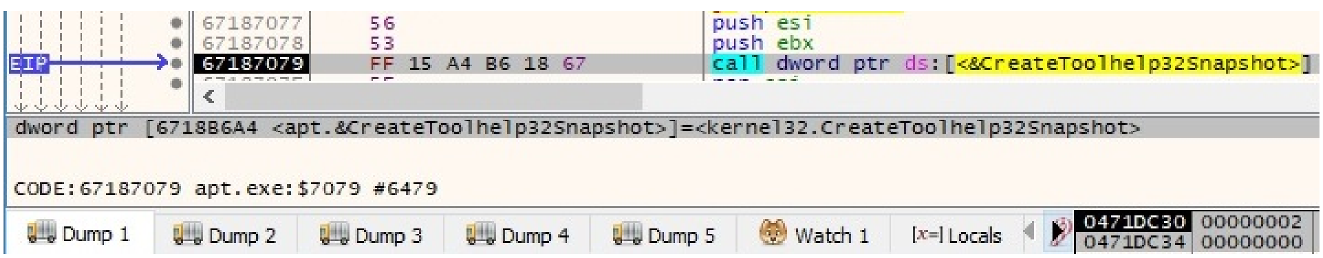

Figure 76

All running processes on the system are retrieved by using the Process32First and Process32Next functions:

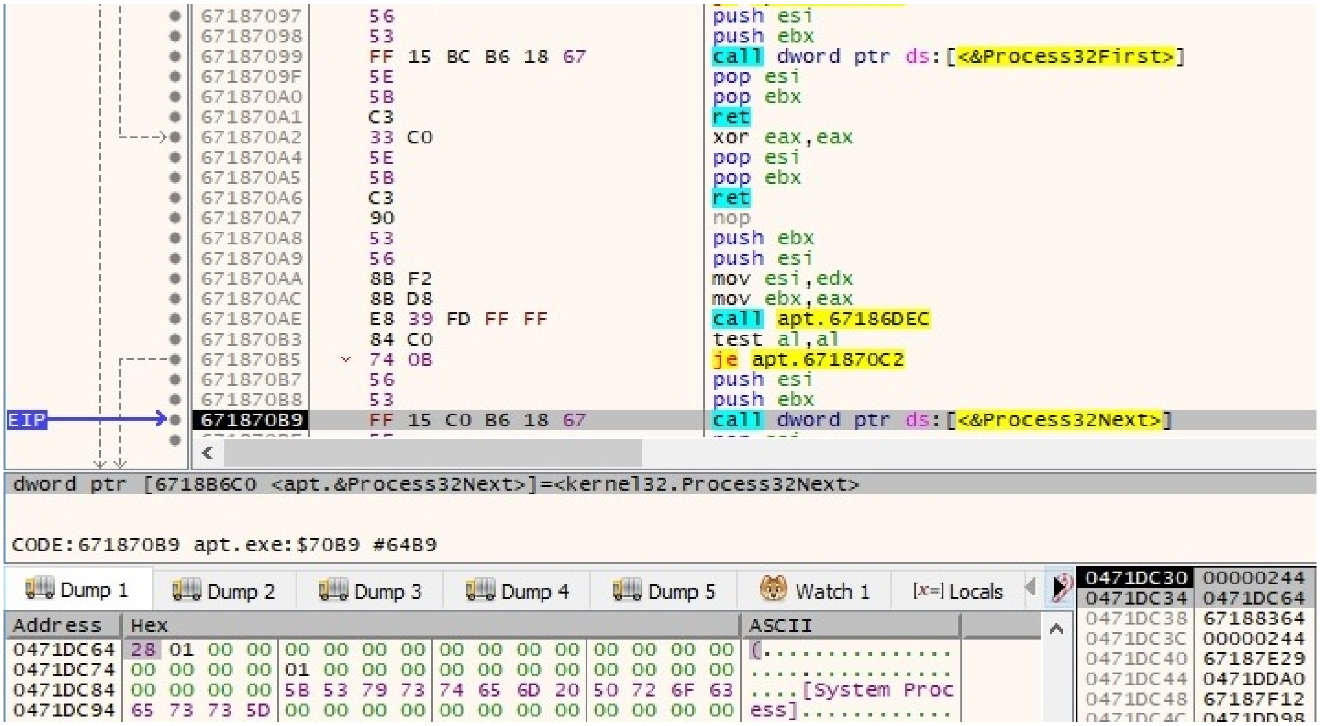

Figure 77

The list of processes is exfiltrated to the CnC server. By decoding the encoded information, we can observe the following string in the beginning "6|1|System Idle Process|0|System|4|smss.exe|500|csrss.exe|604|" (note the process name and the process ID in the buffer):

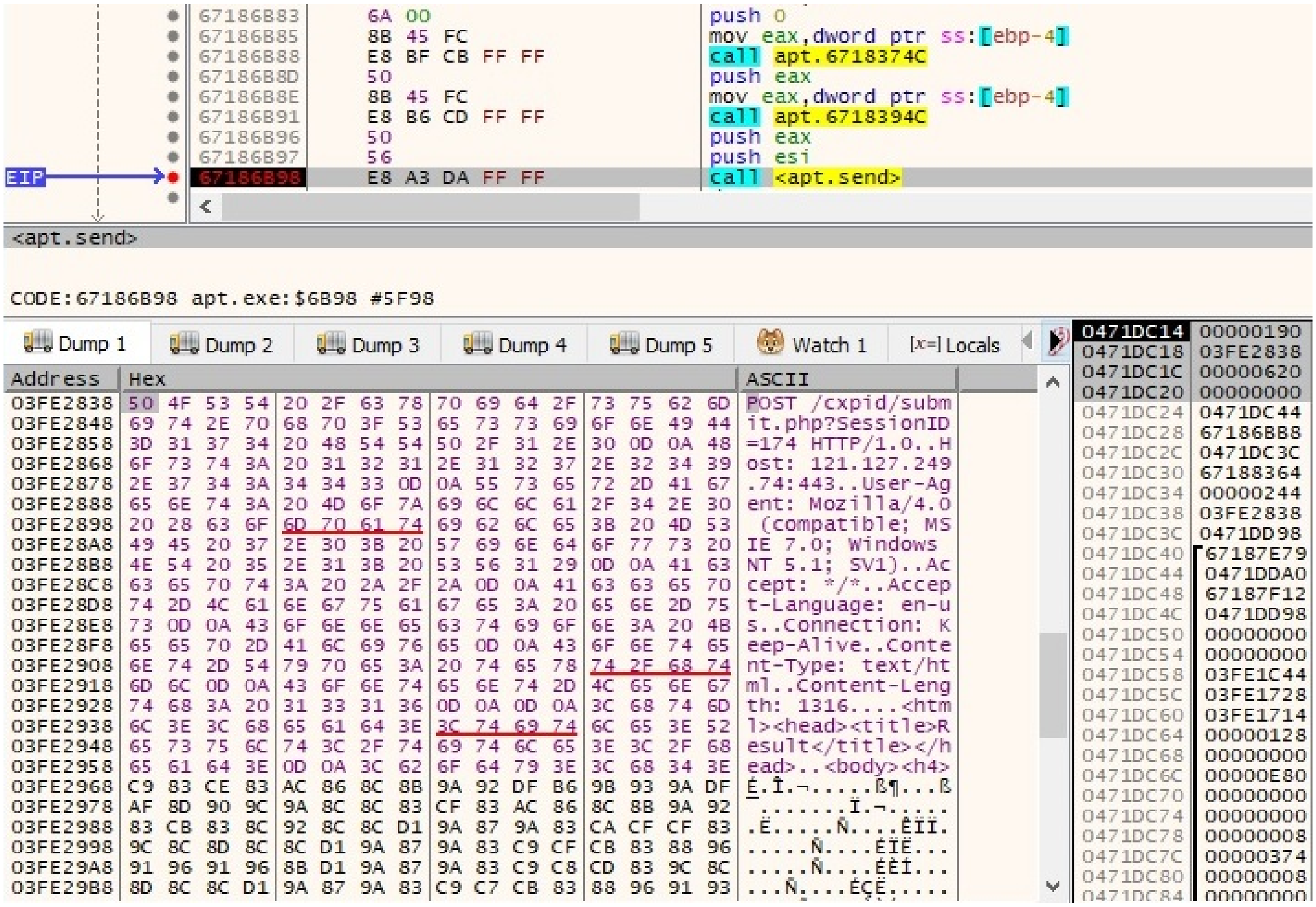

Figure 78

Case 8 – EAX = 7

The GetFileAttributesA API is used to retrieve file system attributes for the current directory, as shown in figure 79:

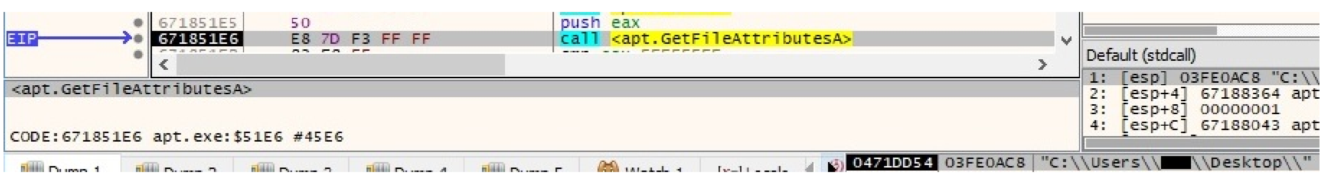

Figure 79

The current directory name is sent to the CnC server in the following form “7|1|Directory name|”:

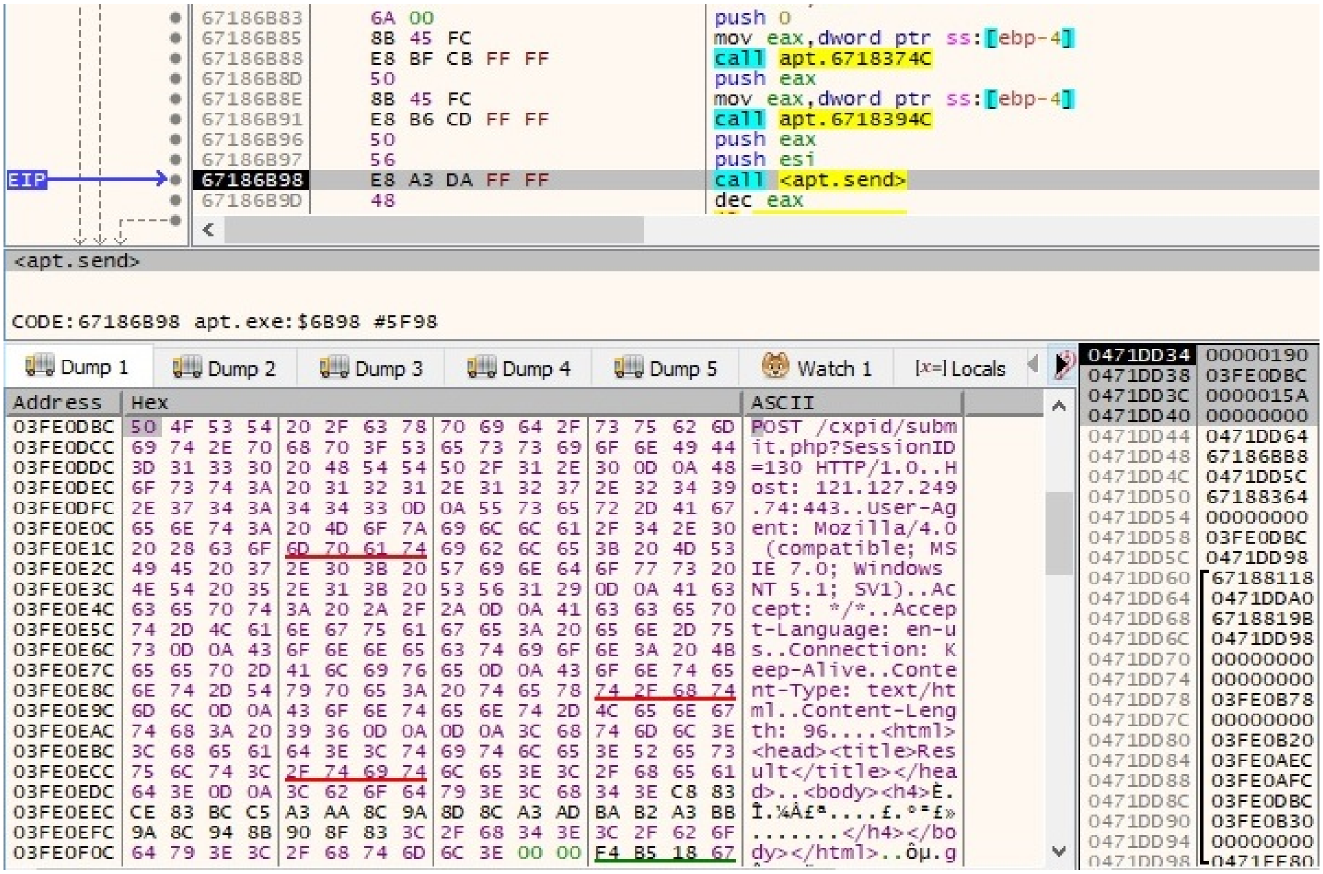

Figure 80

If EAX > 7, the process performs a few recv function calls and jumps back to the switch instruction.

References

Decryption algorithm: https://github.com/Rackedydig/string_decode_algorithm_apt16

FireEye APT groups: https://www.fireeye.com/current-threats/apt-groups.html

FireEye report: https://www.fireeye.com/blog/threat-research/2015/12/the-eps-awakens-part-two.html

MSDN: https://docs.microsoft.com/en-us/windows/win32/api/

Fakenet: https://github.com/fireeye/flare-fakenet-ng

VirusTotal: https://www.virustotal.com/gui/file/bed00a7b59ef2bd703098da6d523a498c8fda05dce931f028e8f16ff434dc89e/detection

INDICATORS OF COMPROMISE

C2 IP address: 121.127.249.74

SHA256:
BED00A7B59EF2BD703098DA6D523A498C8FDA05DCE931F028E8F16FF434DC89E

SHA256:
44DD6A777F50E22EC295FEAE2DDEFFFF1849F8307F50DA4435584200A2BA6AF0

URLs: https[:]//121.127.249.74/cxpid/submit.php?SessionID=<decimal number>

https[:]//121.127.249.74/send.php?id=<decimal number>

https[:]//121.127.249.74/query.php?id=<decimal number>

https[:]//121.127.249.74/cxgid/<Hostname>/<IP address in decimal>/<IP address in decimal>0/index.php

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)