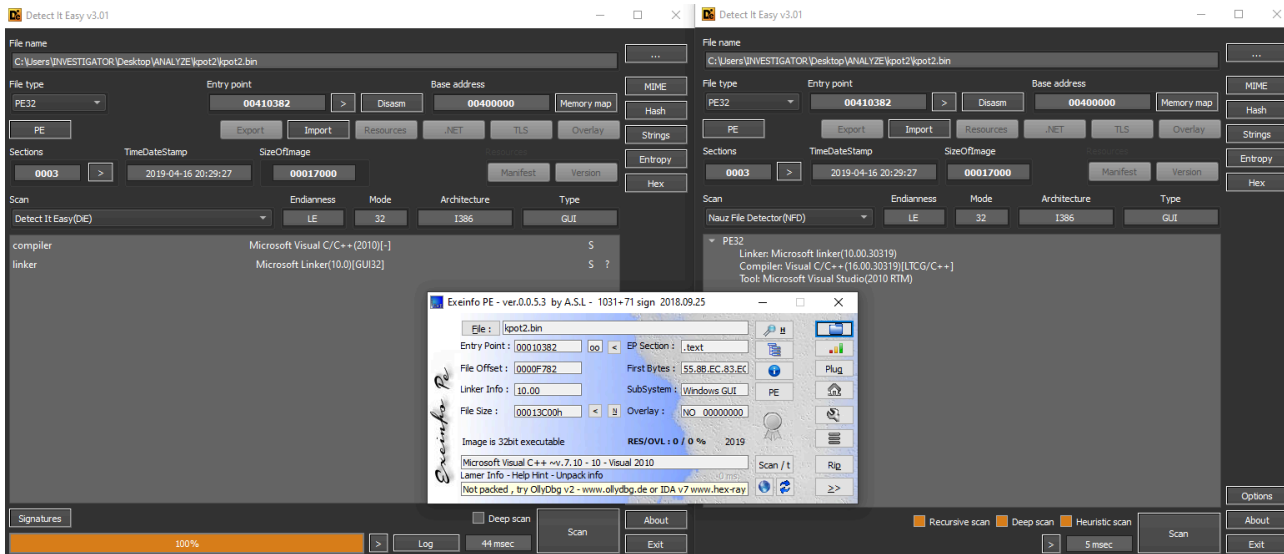


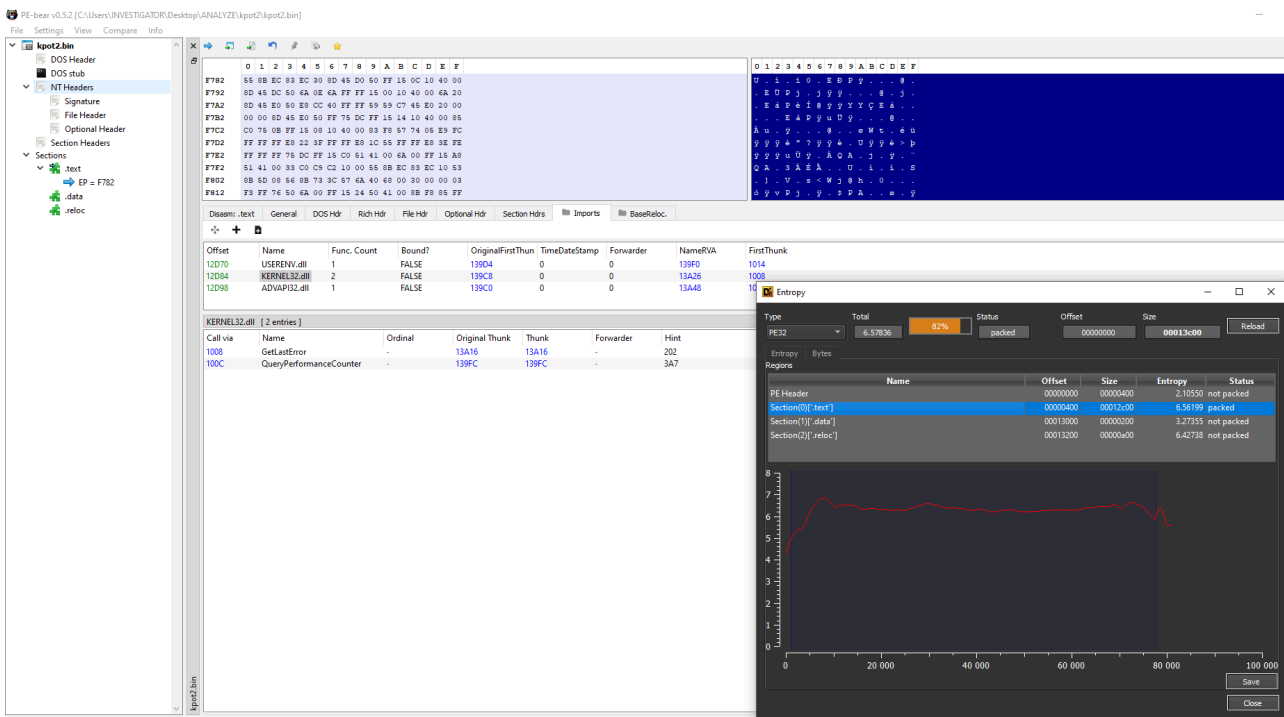


I would not be covering the whole – not so interesting static analysis of file, but only focusing on the IAT of the sample and entropy which usually unhide that the sample is packed.

Well in this case it looks like deterministic signatures cannot identify some well-known packer.



Let's try something what works almost every time. Another picture is more than words.



You can see that the sample has only 4 imports and the entropy of the .text code section is too high – packed.

So for now we know that we have to deal with sample which is some kind of stealer and it is probably encrypted or packed.

**Let's start Reversing !!!**

After throwing the sample to IDA, we can clearly see that in the start (entrypoint) there are 4 functions which should be in our interest.

```
.text:00410382 start:
.text:00410382 push ebp
.text:00410383 mov ebp, esp
.text:00410385 sub esp, 30h
.text:00410388 lea eax, [ebp-30h]
.text:00410388 push eax
.text:0041038C call ds:QueryPerformanceCounter
.text:00410392 lea eax, [ebp-24h]
.text:00410395 push eax
.text:00410396 push 0Eh
.text:00410398 push 0FFFFFFFh
.text:0041039A call ds:OpenProcessToken
.text:004103A0 push 20h
.text:004103A2 lea eax, [ebp-20h]
.text:004103A5 push eax
.text:004103A6 call sub_404477
.text:004103A8 pop ecx
.text:004103AC pop ecx
.text:004103AD mov dword ptr [ebp-20h], 20h
.text:004103B0 lea eax, [ebp-20h]
.text:004103B7 push eax
.text:004103B8 push dword ptr [ebp-24h]
.text:004103BA call ds:LoadUserProfileW
.text:004103C1 test eax, eax
.text:004103C3 jnz short loc_4103D0
.text:004103C5 call ds:GetLastError
.text:004103C8 cmp eax, 57h
.text:004103CE jz short loc_4103D5
.text:004103D0 loc_4103D0:
.text:004103D0 ; CODE XREF: .text:004103C3↑j
.text:004103D0 ; .text:loc_4103D0↑j
.text:004103D0 jmp near ptr loc_4103D0+1
.text:004103D5 loc_4103D5:
.text:004103D5 ; CODE XREF: .text:004103CE↑j
.text:004103D5 call sub_4042FC
.text:004103D7 call sub_4058FB
.text:004103D9 call sub_410222
.text:004103DB push dword ptr [ebp-24h]
.text:004103DD call dword_4151C0
.text:004103DF push 0
.text:004103E1 call dword_4151A8
.text:004103E3 xor eax, eax
.text:004103E5 leave
.text:004103E7 retn 10h
```

These 4 functions will be point of interest.

Some unresolved function.

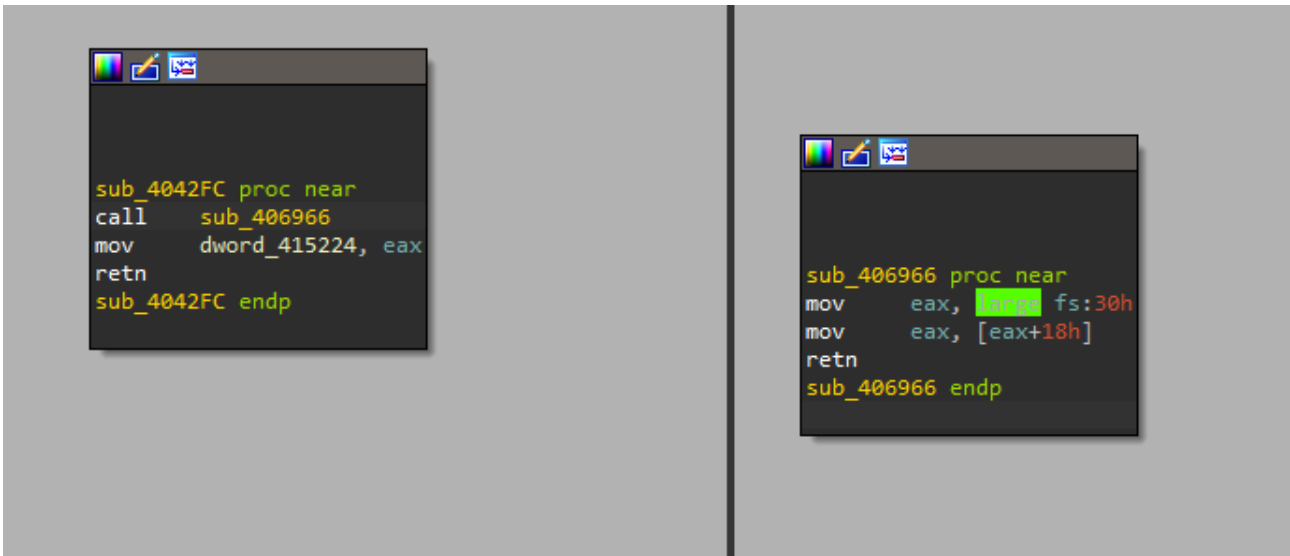
You can see also unresolved calls like “call dword\_4151C0” – these calls are pointing to some location in .data section which is now empty and probably gets filled with addresses later.

```
.data:004151B8 dword_4151B8 dd ? ; DATA XREF: sub_405827+10↑r
.data:004151B8 ; sub_4058FB+55E↑o
.data:004151BC dword_4151BC dd ? ; DATA XREF: sub_404B3F+AE↑r
.data:004151BC ; sub_4058FB+9FC↑o
.data:004151C0 dword_4151C0 dd ? ; DATA XREF: sub_403F10+5F↑r
.data:004151C0 ; sub_403FA4+77↑r ...
.data:004151C4 dword_4151C4 dd ? ; DATA XREF: sub_4058FB+A5A↑o
.data:004151C4 ; sub_4116A3+14D↑r
.data:004151C8 dword_4151C8 dd ? ; DATA XREF: sub_4058FB+A46↑o
.data:004151C8 ; sub_4116A3+7C↑r
.data:004151CC dword_4151CC dd ? ; DATA XREF: sub_4058FB+D27↑o
.data:004151CC ; sub_40FB6F+19↑r
.data:004151D0 dword_4151D0 dd ? ; DATA XREF: sub_4058FB+6DA↑o
.data:004151D0 ; sub_4101A0+5↑r
```

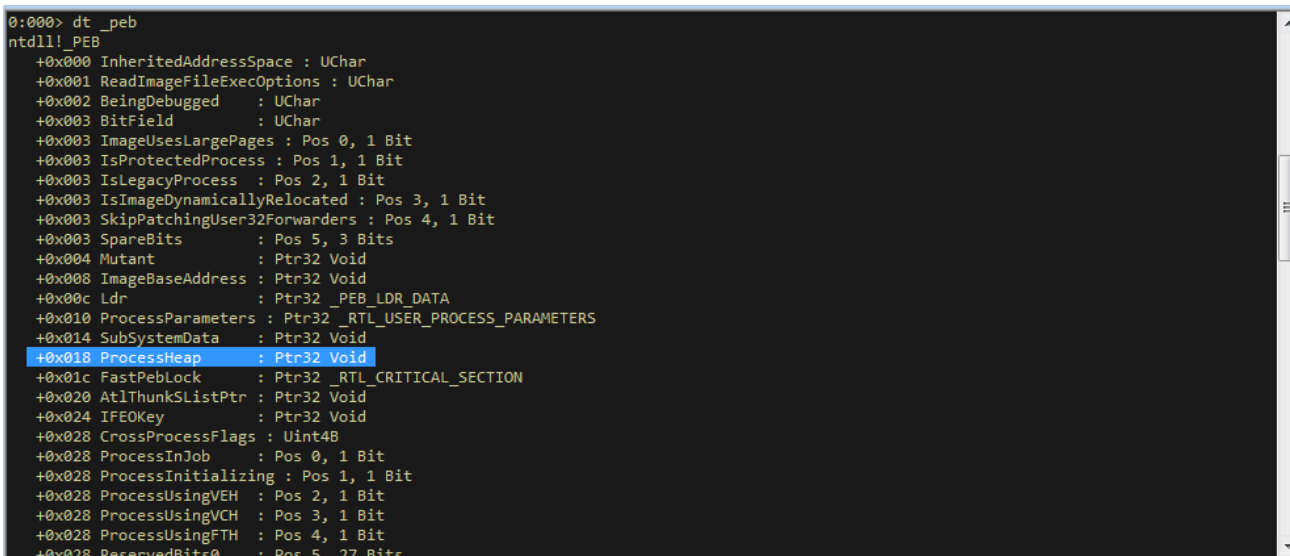
So we have almost no imports and plenty of unresolved calls. Let’s start with the 4 interesting functions mentioned before.

First function is sub\_404477 – this function is not interesting at all. It is only clearing 20 bytes in memory for call LoadUserProfileW.

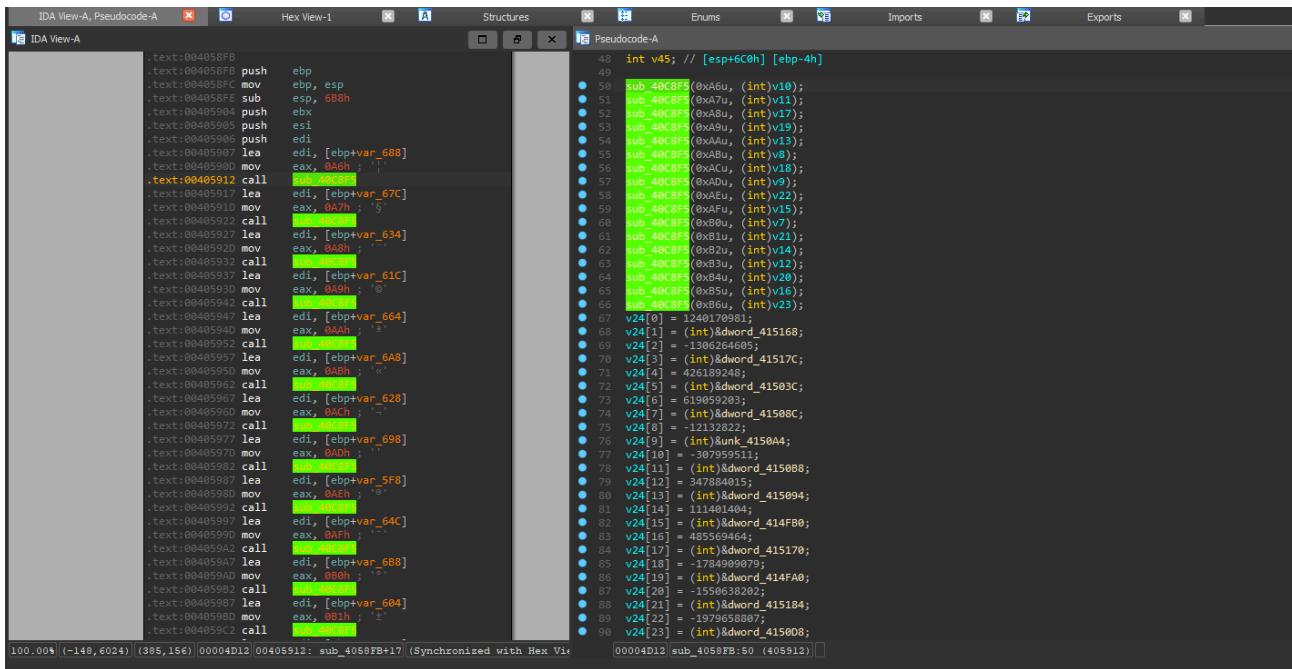
So let’s continue to another call sub\_4042FC. This function is locating PEB exactly ProcessHeap and saving it to location dword\_415224.



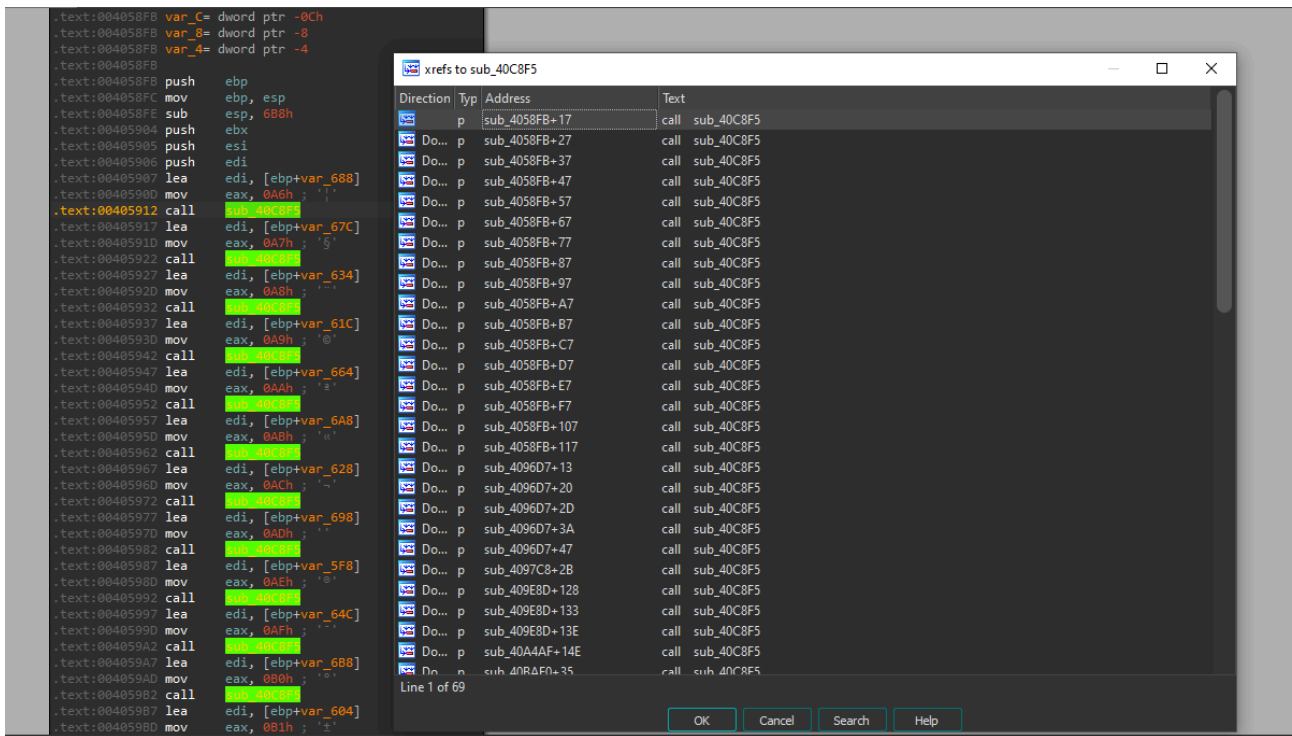
We can confirm it in windbg where we can easily parse PEB structure.



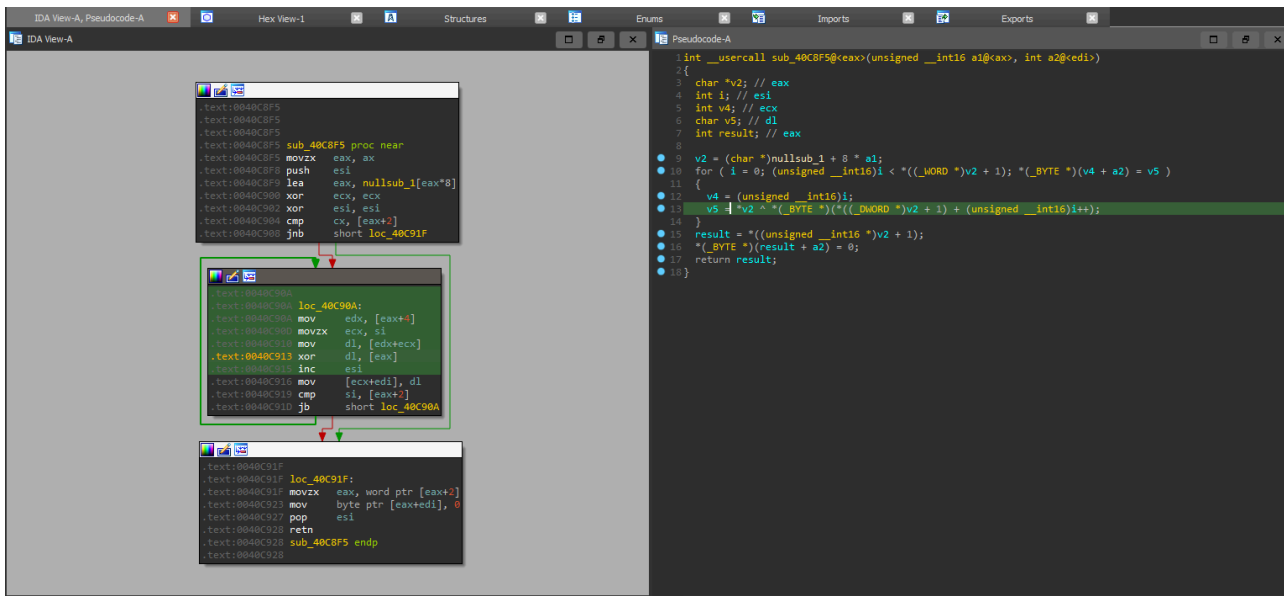
Move to the next function `sub_4058FB`. This function is the most interesting where string decryption and API resolving happens.



At first, we will focus on the function sub\_40C8F5 which you can see is referenced from 69 locations.



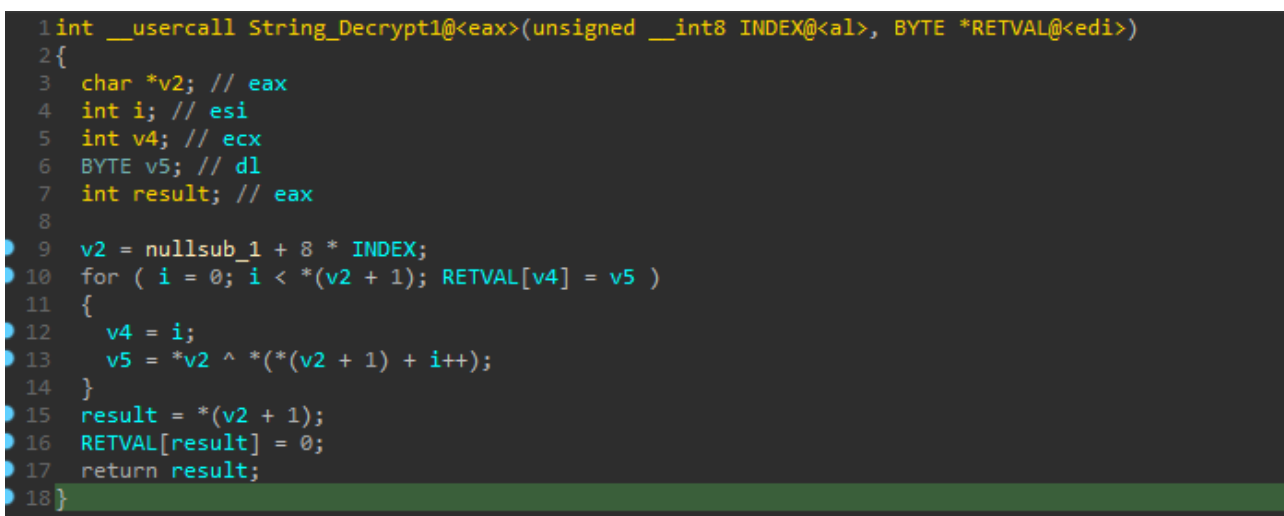
We can see this function (sub\_40C8F5) in the picture below. It looks like some basic xor cipher. It also looks like that decompiler has some hard time to produce us more pretty code so we help him.



So first of all, we check the arguments to this function and retype it correctly. Function sub\_40C8F5 takes 2 arguments, where the first one is some hardcoded unsigned \_\_int8 which looks like some kind of index and the second one is a pointer to stack address.



From the decompiler view we can see that the second argument is actually pointer to BYTE. If we set the types and names of variables correctly we can see better but not the best results.



For better results, we must check also the nullsub\_1 which is not a function but address to array of structures. Let's undefine the nullsub\_1 firstly.

```

.text:00401280      ; sub_40831C; loc_40835D; loc_40835E
.text:00401288 unk_401288  db 0C3h ; Å ; DATA_XREF: String_Decrypt1+410
.text:00401289      db 0 ; sub_40C929+310
.text:0040128A      db 13h
.text:0040128B      db 0
.text:0040128C      db 94h ; " OFF32 SEGDEF [_.text,403594]
.text:0040128D      db 35h ; 5
.text:0040128E      db 40h ; @
.text:0040128F      db 0
.text:00401290      db 0A6h ; !
.text:00401291      db 0
.text:00401292      db 11h
.text:00401293      db 0
.text:00401294      db 80h ; € OFF32 SEGDEF [_.text,403580]
.text:00401295      db 35h ; 5
.text:00401296      db 40h ; @
.text:00401297      db 0

```

```

6 BYTE v5; // dl
7 int result; // eax
8
9 v2 = unk_401288 + 8 * INDEX;
10 for ( i = 0; i < *(v2 + 1); RETVAL[v4] = v5 )
11 {
12     v4 = i;
13     v5 = *v2 ^ (*(v2 + 1) + i++);
14 }
15 result = *(v2 + 1);
16 RETVAL[result] = 0;
17 return result;
18 }

```

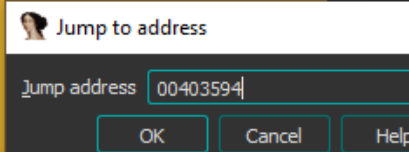
You can see that the index variable is used for pointing to the specific structure which would be probably 8bytes in size. We can confirm it when we check the address .text:00401288 where we can see another 183 structures – 8 bytes in size.

When we check the address .text:00401288, it looks like the first BYTE value “C3” is used as xor key, second BYTE value could be unidentified (undefined), the WORD “0013” looks like length of string which will be xored and the last DWORD (00403594) is the address where our encrypted string is located. Let’s check that address (403594) if our assumption is correct and if there is some kind of encrypted string with length 13h (19).

```

.text:00403594      db 0ABh ; «
.text:00403595      db 0B7h ; ·
.text:00403596      db 0B7h ; ·
.text:00403597      db 0B3h ; ¸
.text:00403598      db 0F9h ; ù
.text:00403599      db 0ECh ; ì
.text:0040359A      db 0ECh ; ì
.text:0040359B      db 0A1h ; ;
.text:0040359C      db 0A6h ; !
.text:0040359D      db 0ADh ; -
.text:0040359E      db 0A7h ; §
.text:0040359F      db 0A6h ; !
.text:004035A0      db 0B0h ; °
.text:004035A1      db 0EDh ; í
.text:004035A2      db 0A0h ; ¨
.text:004035A3      db 0ACh ; ~
.text:004035A4      db 0EDh ; í
.text:004035A5      db 0B6h ; ¶
.text:004035A6      db 0A8h ; ¨
.text:004035A7      db 0

```



Our first assumption was correct so let’s create a structure and apply it as array of structures.

```

00000000
00000000 Decrypt_string_Struct struc ; (sizeof=0x8, mappedto_28)
00000000                                     ; XREF: .text:stru_401288/r
00000000 KEY          db ?
00000001 Unidentified db ?
00000002 Length      dw ?
00000004 Encrypted_string_pointer dd ?
00000008 Decrypt_string_Struct ends
00000008

```

To apply our created structure “Decrypt\_string\_Struct” simply navigate to location 00401288 and press ALT+Q and choose newly created structure.

```

.text:00401280 ; sub_40831C:loc_40835D↓o ...
.text:00401288 ; Decrypt_string_Struct stru_401288[]
• .text:00401288 stru_401288 | Decrypt_string_Struct <0C3h, 0, 13h, 403594h>
.text:00401288 ; DATA XREF: String_Decrypt1+4↓o
.text:00401288 ; sub_40C929+3↓o
• .text:00401290 db 0A6h ; |
• .text:00401291 db 0
• .text:00401292 db 11h
• .text:00401293 db 0
• .text:00401294 db 80h ; € OFF32 SEGDEF [_text,403580]
• .text:00401295 db 35h ; 5
• .text:00401296 db 40h ; @

```

Convert the structure to array with array size = 183.

```

.text:00401288 ; Decrypt_string_Struct stru_401288[]
• .text:00401288 stru_401288 | Decrypt_string_Struct <0C3h, 0, 13h, 403594h>; 0
.text:00401288 ; DATA XREF: String_Decrypt1+4↓o
.text:00401288 ; sub_40C929+3↓o
.text:00401288 Decrypt_string_Struct <0A6h, 0, 11h, 403580h>; 1
.text:00401288 Decrypt_string_Struct <0C3h, 0, 10h, 40356Ch>; 2
.text:00401288 Decrypt_string_Struct <79h, 0, 0Fh, 40355Ch>; 3
.text:00401288 Decrypt_string_Struct <84h, 0, 12h, 403548h>; 4
.text:00401288 Decrypt_string_Struct <0A8h, 0, 13h, 403534h>; 5
.text:00401288 Decrypt_string_Struct <70h, 0, 13h, 403520h>; 6
.text:00401288 Decrypt_string_Struct <8Fh, 0, 13h, 40350Ch>; 7
.text:00401288 Decrypt_string_Struct <3Eh, 0, 1Bh, 4034F0h>; 8
.text:00401288 Decrypt_string_Struct <7, 0, 1Bh, 4034D4h>; 9
.text:00401288 Decrypt_string_Struct <0FAh, 0, 13h, 4034C0h>; 10
.text:00401288 Decrypt_string_Struct <8Ah, 0, 13h, 4034ACh>; 11
.text:00401288 Decrypt_string_Struct <76h, 0, 19h, 403490h>; 12
.text:00401288 Decrypt_string_Struct <0CBh, 0, 0Fh, 403480h>; 13
.text:00401288 Decrypt_string_Struct <67h, 0, 0Bh, 403474h>; 14
.text:00401288 Decrypt_string_Struct <11h, 0, 0Eh, 403464h>; 15
.text:00401288 Decrypt_string_Struct <0D2h, 0, 4, 40345Ch>; 16
.text:00401288 Decrypt_string_Struct <2Dh, 0, 6, 403454h>; 17
.text:00401288 Decrypt_string_Struct <18h, 0, 4, 40344Ch>; 18
.text:00401288 Decrypt_string_Struct <0D2h, 0, 4, 403444h>; 19
.text:00401288 Decrypt_string_Struct <0EAh, 0, 0Dh, 403434h>; 20
.text:00401288 Decrypt_string_Struct <9Fh, 0, 0Eh, 403424h>; 21
.text:00401288 Decrypt_string_Struct <0CBh, 0, 8, 403418h>; 22
.text:00401288 Decrypt_string_Struct <1Fh, 0, 8, 40340Ch>; 23
.text:00401288 Decrypt_string_Struct <20h, 0, 8, 403400h>; 24
.text:00401288 Decrypt_string_Struct <40h, 0, 4, 4033F8h>; 25
.text:00401288 Decrypt_string_Struct <1Fh, 0, 5, 4033F0h>; 26
.text:00401288 Decrypt_string_Struct <10h, 0, 4, 4033E8h>; 27
.text:00401288 Decrypt_string_Struct <5Dh, 0, 8, 4033DCh>; 28
.text:00401288 Decrypt_string_Struct <3Eh, 0, 7, 4033D4h>; 29
.text:00401288 Decrypt_string_Struct <85h, 0, 13h, 4033C0h>; 30
.text:00401288 Decrypt_string_Struct <0D3h, 0, 0Bh, 4033B4h>; 31
.text:00401288 Decrypt_string_Struct <76h, 0, 0Bh, 4033A8h>; 32
.text:00401288 Decrypt_string_Struct <4Ch, 0, 8, 40339Ch>; 33

```

And now we are ready to check our better decompiled function String\_Decrypt1. Below is comparing of decompiled function String\_Decrypt1 before and after modification.

```

1 int __usercall String_Decrypt1@eax(unsigned __int8 INDEX@cal, BYTE *RETVAL@edi)
2 {
3     Decrypt_string_Struct *str_current; // eax
4     int i; // esi
5     int v4; // ecx
6     BYTE v5; // dl
7     int result; // eax
8
9     str_current = &stru_401288[INDEX];
10    for ( i = 0; i < str_current->Length; RETVAL[v4] = v5 )
11    {
12        v4 = i;
13        v5 = str_current->KEY ^ *(str_current->Encrypted_string_pointer + i++);
14    }
15    result = str_current->Length;
16    RETVAL[result] = 0;
17    return result;
18 }

```

AFTER

```

1 int __usercall sub_40C8F5@eax(unsigned __int16 a1@eax, int a2@edi)
2 {
3     char *v2; // eax
4     int i; // esi
5     int v4; // ecx
6     char v5; // dl
7     int result; // eax
8
9     v2 = nullsub_1 + 8 * a1;
10    for ( i = 0; i < *(v2 + 1); *(v4 + a2) = v5 )
11    {
12        v4 = i;
13        v5 = *v2 ^ (*(v2 + 1) + i++);
14    }
15    result = *(v2 + 1);
16    *(result + a2) = 0;
17    return result;
18 }

```

BEFORE

So this algorithm is very basic: First argument to this function is index of the structure in array and second argument is location on stack where the decrypted string is saved.

Key (BYTE) from the structure is xored with each BYTE in the location (Encrypted\_string\_pointer) from our indexed structure, till it reaches the length of encrypted string.

Let's quickly confirm it for the first structure in array with python.

```

.text:00401288 ; Decrypt_string_Struct stru_401288]
.text:00401288 stru_401288 Decrypt_string_Struct <0C3h, 0, 13h, 403594h>; 0 |
.text:00401288 ; DATA XREF: String_Decrypt1+4lo
.text:00401288 ; sub_40C929+3lo
.text:00401288 Decrypt_string_Struct <0A6h, 0, 11h, 403580h>; 0
.text:00401288 Decrypt_string_Struct <0C3h, 0, 10h, 403560h>; 0
.text:00401288 Decrypt_string_Struct <79h, 0, 0Fh, 40355Ch>; 0
.text:00401288 Decrypt_string_Struct <84h, 0, 12h, 403548h>; 0
.text:00401288 Decrypt_string_Struct <0A8h, 0, 13h, 403534h>; 0
.text:00401288 Decrypt_string_Struct <70h, 0, 13h, 403520h>; 0
.text:00401288 Decrypt_string_Struct <8Fh, 0, 13h, 40350Ch>; 0
.text:00401288 Decrypt_string_Struct <3Eh, 0, 1Bh, 4034F0h>; 0
.text:00401288 Decrypt_string_Struct <7, 0, 1Bh, 4034D4h>; 0
.text:00401288 Decrypt_string_Struct <0FAh, 0, 13h, 4034C0h>; 0
.text:00401288 Decrypt_string_Struct <8Ah, 0, 13h, 4034ACh>; 0
.text:00401288 Decrypt_string_Struct <76h, 0, 19h, 403490h>; 0
.text:00401288 Decrypt_string_Struct <0CBh, 0, 0Fh, 403480h>; 0
.text:00401288 Decrypt_string_Struct <67h, 0, 0Bh, 403474h>; 0
.text:00401288 Decrypt_string_Struct <11h, 0, 0Eh, 403464h>; 0
.text:00401288 Decrypt_string_Struct <0D2h, 0, 4, 40345Ch>; 0
.text:00401288 Decrypt_string_Struct <2Dh, 0, 6, 403454h>; 0
.text:00401288 Decrypt_string_Struct <18h, 0, 4, 40344Ch>; 0
.text:00401288 Decrypt_string_Struct <0D2h, 0, 4, 403444h>; 0
.text:00401288 Decrypt_string_Struct <0EAh, 0, 0Dh, 403434h>; 20

```

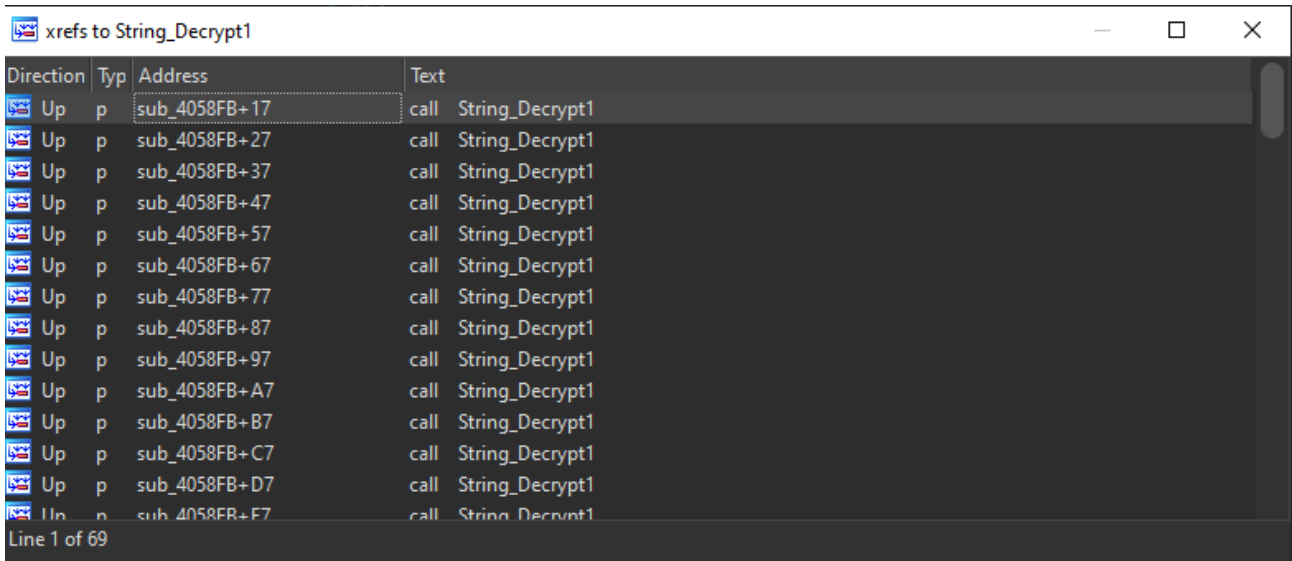
```

>>> import malduck
>>> xor_key = 0xc3
>>> #encrypted string in location 0x403594
>>> encrypted_string = bytes.fromhex("ABB7B7B3F9ECECA1A6ADA7A6B0EDA0ACEDB6A8")
>>> print((malduck.xor(xor_key, encrypted_string)).decode())
http://bendes.co.uk
>>>

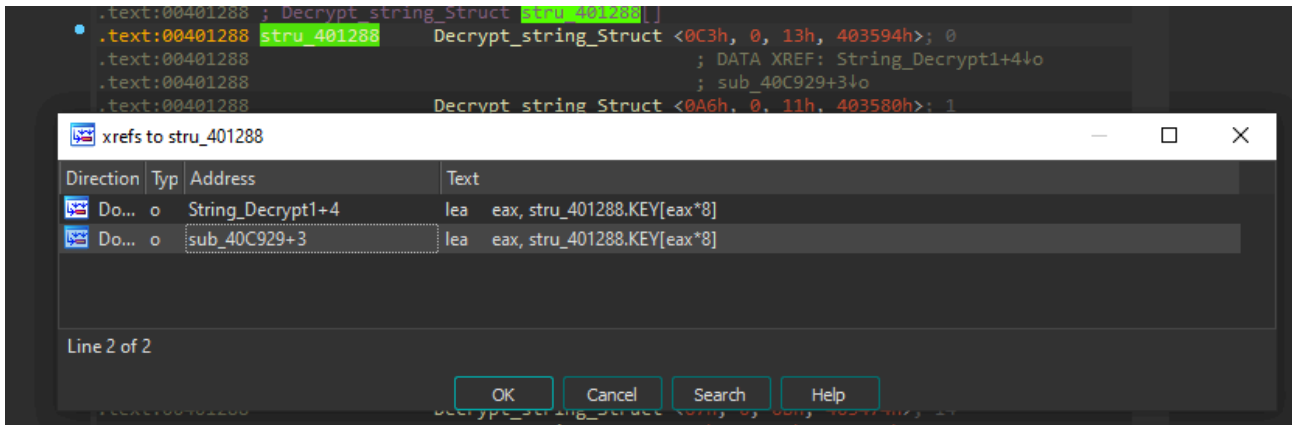
```

We were correct and obtained our first IOC.

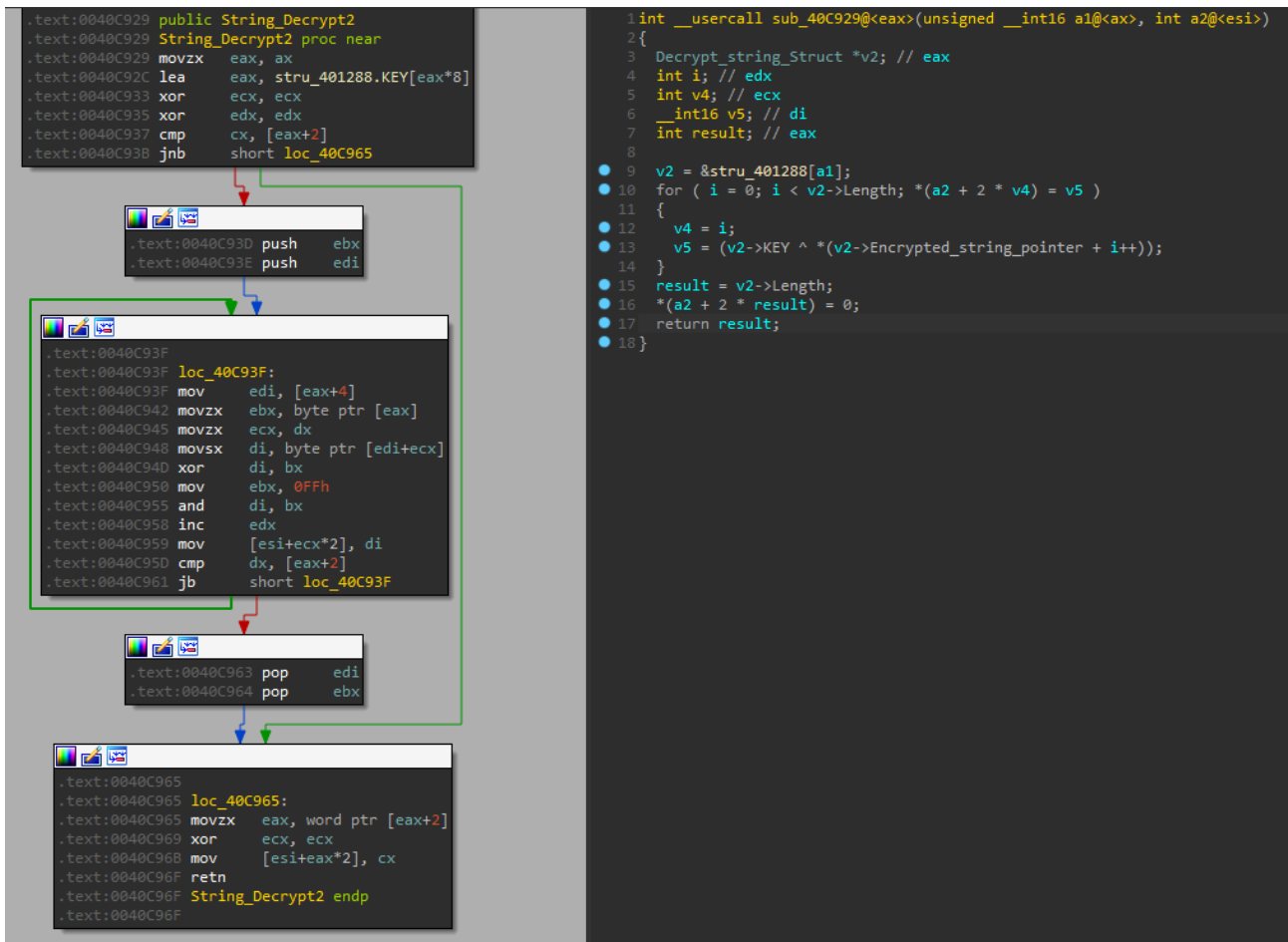
Before jumping to IDAPython we forgot something. If you remember the function String\_Decrypt1 was referenced from 69 locations but our array of structures contains 183 members.



So we could check Xreferences to our array of structures if we could find another String\_DecryptX function.



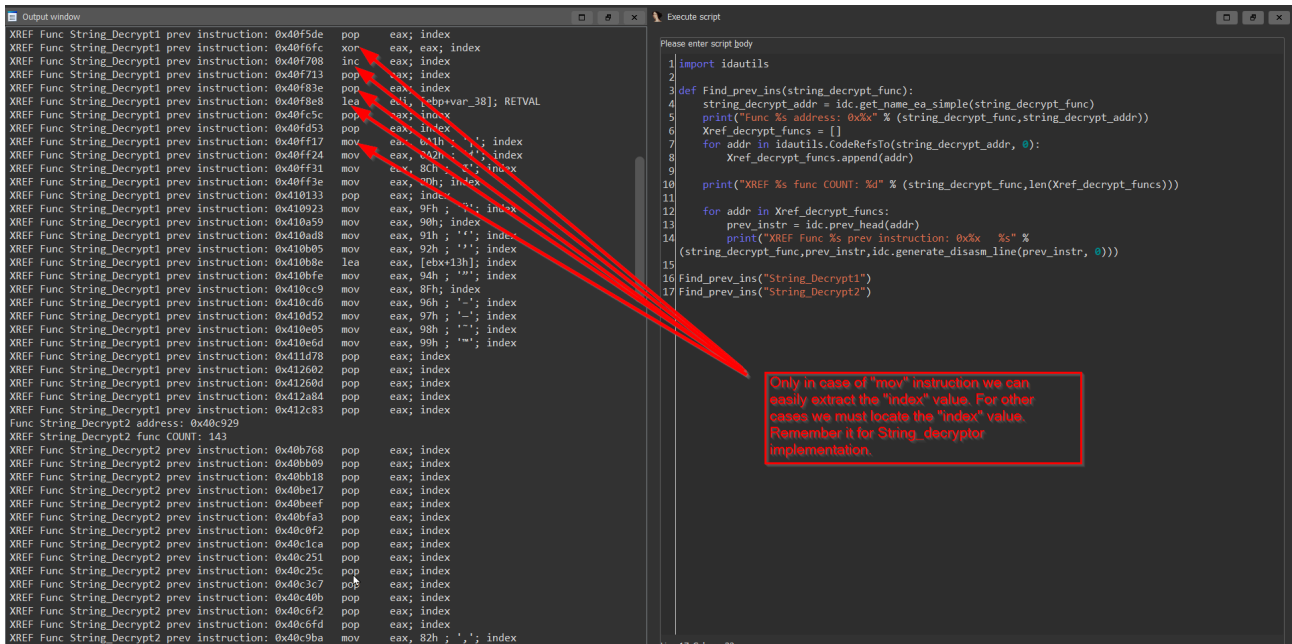
We were right, there is another one. Quick checking that function (sub\_40C929) revealed that it is basically the same as function String\_Decrypt1. So we rename it to String\_Decrypt2.



Now when we found both functions referencing our array of structures, we can jump to IDAPython and write a decryptor.

The final decryptor could be something, what will find all location from where our 2 string-decrypting functions (String\_Decrypt1, String\_Decrypt2) are called. After it finds these locations it will grab the first argument as our “INDEX” to structure, find and parse the structure[index]. This will serve us for decrypting the current string so we could insert a comment to location from where the string-decrypt function was called.

During the creating of decryptor, I found one quite tricky problem with locating the first argument value “INDEX” for our (String\_Decrypt1, String\_Decrypt2) functions. You can see it on the picture below where I let IDA with little help from IDAPython to print assembly line for all previous instruction before our functions (String\_Decrypt1, String\_Decrypt2) get called. The script part is self-explanatory.



You can find script “Find\_previous\_instruction.py” here [[Find previous instruction.py](#)].

We must deal with locating the first argument during the string-decryptor implementation. In the picture below is the string-decryptor script in IDAPython for the “String\_Decrypt1” function.

```

1 #Kpot_stedler - https://www.virustotal.com/gui/file/e7f8302a2fd28d15f62bd20d48bfe30334e5353cbdef112ba1f9231b5599d/detection
2
3 import idutils
4 import idc
5
6 struct_start = 0x401288
7
8 def decryptor(index, call_addr):
9     decrypted_string=""
10    current_struct_start = struct_start + #*index
11    current_struct_bytes = idc.get_bytes(current_struct_start, 8)
12    print(current_struct_bytes.hex())
13    #structure parsing and xorring
14    key = int.from_bytes(current_struct_bytes[0:1], byteorder='little', signed=False)
15    length = int.from_bytes(current_struct_bytes[2:4], byteorder='little', signed=False)
16    buffer_string_addr = int.from_bytes(current_struct_bytes[4:8], byteorder='little', signed=False)
17    print(hex(key),hex(length),hex(buffer_string_addr))
18    #decrypting
19    for i in range(0,length):
20        decrypted_string += chr(key ^ idc.get_wide_byte(buffer_string_addr+i))
21    print(decrypted_string)
22    #commenting assembly view
23    idc.set_cmt(call_addr, decrypted_string,0)
24    #commenting decompile view on the same address as assembly view
25    cfunc = idaapi.decompile(call_addr)
26    t1 = idaapi.treeloc_t()
27    t1.ea = call_addr
28    t1.itp = idaapi.ITP_SEMI
29    cfunc.set_user_cmt(t1, decrypted_string)
30    cfunc.save_user_cmts()
31
32    #string decrypting func NAME = "String_Decrypt1" - 0040C8F5
33    string_decrypt_addr = idc.get_name_ea_simple("String_Decrypt1")
34    print("Func String_Decrypt1 address: 0x%x" % (string_decrypt_addr))
35    Xref_decrypt_funcs = []
36    for addr in idutils.CodeRefsTo(string_decrypt_addr, 0):
37        Xref_decrypt_funcs.append(addr)
38
39    print("XREF String_Decrypt1 func COUNT: %d" % (len(Xref_decrypt_funcs)))
40

```

```

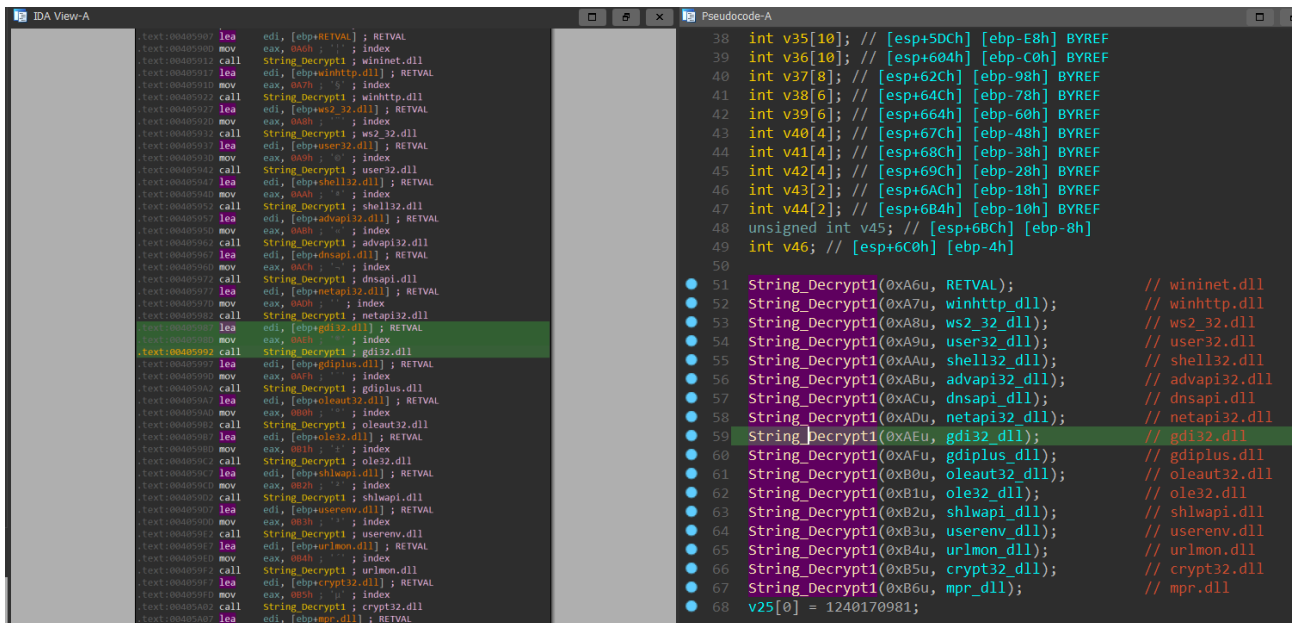
41 for addr in Xref_decrypt_funcs:
42     prev_instr = idc.prev_head(addr)
43     print("XREF Func String_Decrypt1 prev instruction: 0x%x %s" % (prev_instr, idc.generate_disasm_line(prev_instr, 0)))
44     m = idc.print_insn_mnem(prev_instr)
45     #searching instruction in prev-instr addr and finding index argument
46     if m == 'mov':
47         op = idc.get_operand_type(prev_instr, 1)
48         if op == 0: imm:
49             index = idc.get_operand_value(prev_instr, 1)
50             print("Index value: 0x%x" % (index))
51             decryptor(index, addr)
52
53     if m == 'pop':
54         prev_instr2 = idc.prev_head(prev_instr)
55         prev_instr3 = idc.prev_head(prev_instr2)
56         op = idc.get_operand_type(prev_instr3, 0)
57         if op == 0: imm:
58             index = idc.get_operand_value(prev_instr3, 0)
59             print("Index value: 0x%x" % (index))
60             decryptor(index, addr)
61
62     if m == 'xor':
63         index = 0
64         print("Index value: 0x%x" % (index))
65         decryptor(index, addr)
66
67     if m == 'inc':|
68         index = 1
69         print("Index value: 0x%x" % (index))
70         decryptor(index, addr)
71
72     if m == 'leq':
73         prev_instr2 = idc.prev_head(prev_instr)
74         m2 = idc.print_insn_mnem(prev_instr2)
75         if m2 == 'mov':
76             index = 0x57
77             print("Index value: 0x%x" % (index))
78             decryptor(index, addr)
79         else:
80             index = 0x93
81             print("Index value: 0x%x" % (index))
82             decryptor(index, addr)
83
84

```

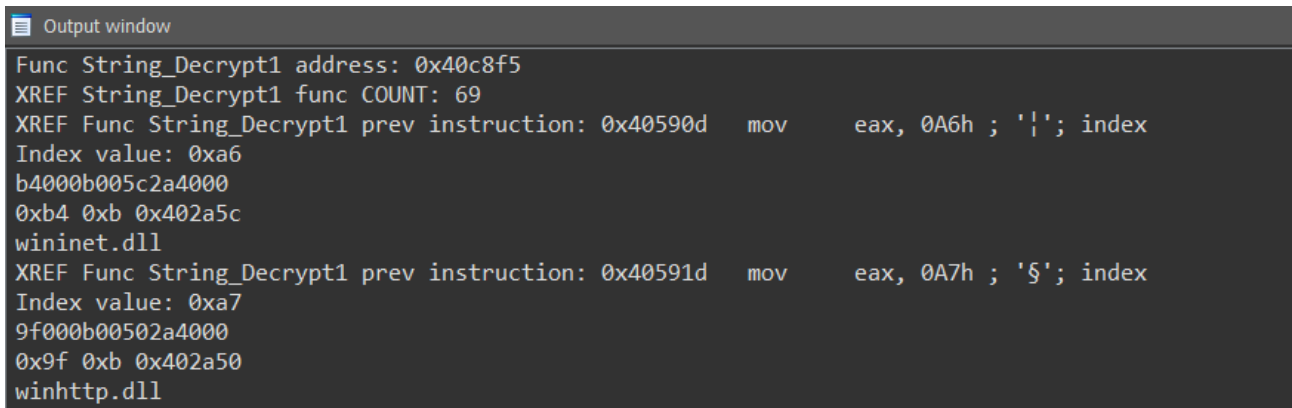
String-decryptor script for the “String\_Decrypt2” function is little different only in area of searching and extracting the first argument VALUE (index) to function String\_Decrypt2.

You can find both scripts for decrypting functions (String\_Decrypt1, String\_Decrypt2) here [\[Decrypt\\_KPOT\\_Strings1.py, Decrypt\\_KPOT\\_Strings2.py\]](#).

After running these scripts, we get commented all location from where (String\_Decrypt1, String\_Decrypt2) are called with decrypted strings in both assembly view and decompile view.



In Output window we could see some information like: String\_Decrypt1 function address, count of references and for each processed reference is shown - current index value, current structure in hex, current xor KEY, length of encrypted string, address where the encrypted string is located and finally decrypted string.



As we are now able to see decrypted strings we are getting some ideas about functionality of this sample. As you can see we were able to get 211 locations with decrypted strings. Some of them are referencing the same string. We can clearly say that this sample is some kind of credential, cryptocurrency stealer...

Address	Instruction/Data	Comment
0X40CB3B	N call String_Decrypt2; Software	Software
0X40CB48	N call String_Decrypt2; wallet.dat	wallet.dat
0X40CBEB	N call String_Decrypt2; Software	Software
0X40CBF6	N call String_Decrypt2; monero-project	monero-project
0X40CC03	N call String_Decrypt2; wallet_path	wallet_path
0X40CC10	N call String_Decrypt2; Crypto	Crypto
0X40CC1D	N call String_Decrypt2; wallet.dat	wallet.dat
0X40CD22	N call String_Decrypt2; com.libertyjvaxx\IndexedDB\file_0\indexddb.leveldb\000003.log	com.libertyjvaxx\IndexedDB\file_0\indexddb.leveldb\000003.log
0X40CD2F	N call String_Decrypt2; Crypto	Crypto
0X40CD98	N call String_Decrypt2; Exodus	Exodus
0X40CDFB	N call String_Decrypt2; wallet.dat	wallet.dat
0X40D009	N call String_Decrypt2; 0123456789ABCDEF	0123456789ABCDEF
0X40D10D	N call String_Decrypt2; connections	connections
0X40D118	N call String_Decrypt2; GHISLER\wcv_ftp.ini	GHISLER\wcv_ftp.ini
0X40D123	N call String_Decrypt2; Host	Host
0X40D12E	N call String_Decrypt2; Username	Username
0X40D139	N call String_Decrypt2; Password	Password
0X40D147	N call String_Decrypt2; 1\TotalCommander\%s\%s\%s	1\TotalCommander\%s\%s\%s
0X40D2E6	N call String_Decrypt2; recentservers	recentservers
0X40D2F4	N call String_Decrypt2; sitemanager	sitemanager
0X40D302	N call String_Decrypt2; FileZilla	FileZilla
0X40D30D	N call String_Decrypt2; Host	Host
0X40D318	N call String_Decrypt2; User	User
0X40D323	N call String_Decrypt2; Port	Port
0X40D32E	N call String_Decrypt2; Pass	Pass
0X40D339	N call String_Decrypt2; encoding	encoding
0X40D347	N call String_Decrypt2; 1\FileZilla\%s\%s\%s\%S	1\FileZilla\%s\%s\%s\%S
0X40D5C9	N call String_Decrypt2; Software	Software
0X40D5D7	N call String_Decrypt2; Martin Prikyr\WinSCP 2\Sessions	Martin Prikyr\WinSCP 2\Sessions
0X40D5E2	N call String_Decrypt2; HostName	HostName
0X40D5ED	N call String_Decrypt2; UserName	UserName
0X40D5F8	N call String_Decrypt2; Password	Password
0X40D606	N call String_Decrypt2; 1\WinSCP\%s\%s\%s	1\WinSCP\%s\%s\%s
0X40D77E	N call String_Decrypt2; Ipswitch\WS_FTP\Sites\wcv_ftp.ini	Ipswitch\WS_FTP\Sites\wcv_ftp.ini
0X40D789	N call String_Decrypt2; Hostname	Hostname
0X40D794	N call String_Decrypt2; UID	UID
0X40D79F	N call String_Decrypt2; PWD	PWD
0X40D7AA	N call String_Decrypt2; 1\WS_FTP\%s\%s\%S	1\WS_FTP\%s\%s\%S

So for now strings are decrypted and we can continue to resolve API calls.

We will continue with our string-decrypting and API resolving function sub\_4058FB to see what is going on next. We can see that there will be probably some kind of API name hashing which after matching hash of API name, the address of the API function will be saved to the hardcoded memory location. In the picture below we can see the stack preparation for the API name hashing and resolving.

```

.text:00405980 mov     eax, 0A0h
.text:00405982 call   String_Decrypt1; netapi32.dll
.text:00405984 lea   edi, [ebp+gd32.d1]
.text:00405986 mov     eax, 0A0h
.text:00405988 call   String_Decrypt1; gd32.dll
.text:0040598A lea   edi, [ebp+gdplus.d1]
.text:0040598C mov     eax, 0A0h
.text:0040598E call   String_Decrypt1; gdplus.d1
.text:00405990 lea   edi, [ebp+oleaut32.d1]
.text:00405992 mov     eax, 0B0h
.text:00405994 call   String_Decrypt1; oleaut32.dll
.text:00405996 lea   edi, [ebp+ole32.d1]
.text:00405998 mov     eax, 0B0h
.text:0040599A call   String_Decrypt1; ole32.dll
.text:0040599C lea   edi, [ebp+shlwapi.d1]
.text:0040599E mov     eax, 0B0h
.text:004059A0 call   String_Decrypt1; shlwapi.dll
.text:004059A2 lea   edi, [ebp+userenv.d1]
.text:004059A4 mov     eax, 0B0h
.text:004059A6 call   String_Decrypt1; userenv.dll
.text:004059A8 lea   edi, [ebp+urlmon.d1]
.text:004059AA mov     eax, 0B0h
.text:004059AC call   String_Decrypt1; urlmon.dll
.text:004059AE lea   edi, [ebp+crypt32.d1]
.text:004059B0 mov     eax, 0B0h
.text:004059B2 call   String_Decrypt1; crypt32.dll
.text:004059B4 mov     eax, 0B0h
.text:004059B6 call   String_Decrypt1; mpr.dll
.text:004059B8 mov     [ebp+var_5C4], 40391386h
.text:004059BA mov     [ebp+var_5C0], offset_dword_41517C
.text:004059BC mov     [ebp+var_5D0], offset_dword_41593C
.text:004059BE mov     [ebp+var_5C8], 24E61803h
.text:004059C0 mov     [ebp+var_5C4], offset_dword_41508C
.text:004059C2 mov     [ebp+var_5C4], 0F960924h
.text:004059C4 mov     [ebp+var_5C0], offset_dword_415944
.text:004059C6 mov     [ebp+var_58C], 0E0A4E92h
.text:004059C8 mov     [ebp+var_588], offset_dword_415088
.text:004059CA mov     [ebp+var_584], 148C70E9h
.text:004059CC mov     [ebp+var_580], offset_dword_415094
.text:004059CE mov     [ebp+var_5AC], 6A3D98Ch
.text:004059D0 mov     [ebp+var_5A8], offset_dword_414FB0
.text:004059D2 mov     [ebp+var_5A4], 1C913386h
.text:004059D4 mov     [ebp+var_5A0], offset_dword_415170
.text:004059D6 mov     [ebp+var_59C], 95A72E9Bh
.text:004059D8 mov     [ebp+var_598], offset_dword_414FA0
.text:004059DA mov     [ebp+var_594], 0A391286h
.text:004059DC mov     [ebp+var_590], offset_dword_415184
.text:004059DE mov     [ebp+var_58C], 0898DC9h
.text:004059E0 mov     [ebp+var_588], offset_dword_415008
.text:004059E2 mov     [ebp+var_584], 7A8FF0D4h
.text:004059E4 mov     [ebp+var_580], offset_dword_414F74
.text:004059E6 mov     [ebp+var_57C], 61302400h
    
```

```

35 int v33[12]; // [esp+97Ch] [ebp-148h] BYREF
36 int v33[12]; // [esp+95ACh] [ebp-118h] BYREF
37 int v34[10]; // [esp+950Ch] [ebp-E8h] BYREF
38 int v35[10]; // [esp+884h] [ebp-C0h] BYREF
39 int v36[8]; // [esp+62Ch] [ebp-90h] BYREF
40 int v37[6]; // [esp+64Ch] [ebp-78h] BYREF
41 int v38[6]; // [esp+664h] [ebp-60h] BYREF
42 int v39[4]; // [esp+67Ch] [ebp-48h] BYREF
43 int v40[4]; // [esp+68Ch] [ebp-38h] BYREF
44 int v41[4]; // [esp+69Ch] [ebp-28h] BYREF
45 int v42[2]; // [esp+6ACh] [ebp-18h] BYREF
46 int v43[2]; // [esp+6B4h] [ebp-10h] BYREF
47 int v44; // [esp+6BC] [ebp-8h]
48 int v45; // [esp+6C0] [ebp-4h]
49
50 String_Decrypt1(0xA6u, wininet.dll); // wininet.dll
51 String_Decrypt1(0xA7u, winhttp.dll); // winhttp.dll
52 String_Decrypt1(0xA8u, ws2_32.dll); // ws2_32.dll
53 String_Decrypt1(0xA9u, user32.dll); // user32.dll
54 String_Decrypt1(0xAau, shell32.dll); // shell32.dll
55 String_Decrypt1(0xABu, advapi32.dll); // advapi32.dll
56 String_Decrypt1(0xACu, dnsapi.dll); // dnsapi.dll
57 String_Decrypt1(0xADu, netapi32.dll); // netapi32.dll
58 String_Decrypt1(0xAEu, gd32.dll); // gd32.dll
59 String_Decrypt1(0xAFu, gdplus.d1); // gdplus.d1
60 String_Decrypt1(0xB0u, oleaut32.dll); // oleaut32.dll
61 String_Decrypt1(0xB1u, ole32.dll); // ole32.dll
62 String_Decrypt1(0xB2u, shlwapi.dll); // shlwapi.dll
63 String_Decrypt1(0xB3u, userenv.dll); // userenv.dll
64 String_Decrypt1(0xB4u, urlmon.dll); // urlmon.dll
65 String_Decrypt1(0xB5u, crypt32.dll); // crypt32.dll
66 String_Decrypt1(0xB6u, mpr.dll); // mpr.dll
67 v24[0] = 124017091;
68 v24[1] = &word_415168;
69 v24[2] = -130625405;
70 v24[3] = &word_41517C;
71 v24[4] = 426189248;
72 v24[5] = &word_415083;
73 v24[6] = &word_415929;
74 v24[7] = &word_41508C;
75 v24[8] = -12132822;
76 v24[9] = &unk_415044;
77 v24[10] = -30795911;
78 v24[11] = &word_415088;
79 v24[12] = 347884015;
80 v24[13] = &word_415094;
81 v24[14] = 111461404;
82 v24[15] = &word_414FB0;
83 v24[16] = 485568464;
84 v24[17] = &word_415170;
85 v24[18] = -178490979;
86 v24[19] = &word_414FA0;
87 v24[20] = -153663202;
88 v24[21] = &word_415184;
89 v24[22] = -197965807;
    
```

After the stack is prepared two functions get called. Let's check the first function sub\_406936.

```

.text:004065C7 mov [ebp+var_30], 4952020h
.text:004065CE mov [ebp+var_34], offset dword_414FF4
.text:004065D5 mov [ebp+var_30], 94F693D8h
.text:004065DC mov [ebp+var_2C], offset dword_415028
.text:004065E3 mov [ebp+var_78], 27812AD3h
.text:004065EA mov [ebp+var_74], offset dword_4151A0
.text:004065F1 mov [ebp+var_70], 00CD9AD63h
.text:004065F8 mov [ebp+var_6C], offset dword_41514C
.text:004065FF mov [ebp+var_68], 336815C4h
.text:00406606 mov [ebp+var_64], offset dword_414FFC
.text:0040660D mov [ebp+var_10], 59AF12E8h
.text:00406614 mov [ebp+var_C], offset dword_41515C
.text:0040661B mov [ebp+var_18], 6FC33AB1h
.text:00406622 mov [ebp+var_14], offset dword_4151CC
.text:00406629 call sub_406936
.text:00406632 mov ebx, eax
.text:00406636 push 822FC0FAh
.text:00406639 call sub_4045DC
.text:0040663A pop ecx
.text:0040663B lea ecx, [ebp+var_5E4]
.text:00406641 mov [ebp+var_400], ecx
.text:00406647 lea ecx, [ebp+wininet.dll]

```

The function sub\_406936 is basically parsing PEB structure and loading base address of the kernel32.dll module. You can easily confirm it with help of IDA \_PEB struct or windbg as in the pictures below. It is finding the PEB structure, \_PEB\_LDR\_DATA where it finds first member in InLoadOrderModuleList which is our sample kpot2.exe. After that, it finds a location of the third loaded module (kernel32.dll) and extracts the base address. This base address of kernel32.dll is passed to the next function sub\_4045DC so it will be used to find addresses of export functions.

```

Pseudocode-A
1 struct _LIST_ENTRY *find_kernel32_base()
2 {
3   return NtCurrentPeb()->Ldr->InLoadOrderModuleList.Flink->Flink->Flink[3].Flink;
4 }

```

The screenshot shows the assembly code for sub\_406936 and its corresponding PEB structure. Red arrows and boxes highlight key elements:

- 1. PEB structure
- 2. InLoadOrderModuleList
- 3. First member (kpot2.exe)
- 4. InMemoryOrderModuleList
- 5. Third member (kernel32.dll)
- 6. Return value of find\_kernel32\_base()

In our case First member InLoadOrderModuleList is our kpot2.exe - 0x574738

Return Value of this function is base address of kernel32.dll

We can move to the next function sub\_4045DC which is responsible for finding address of LoadLibraryA API function from kernel32.dll module.

```

.text:00406629 call sub_406936
.text:00406632 mov ebx, eax
.text:00406636 push 822FC0FAh
.text:00406639 call sub_4045DC

```

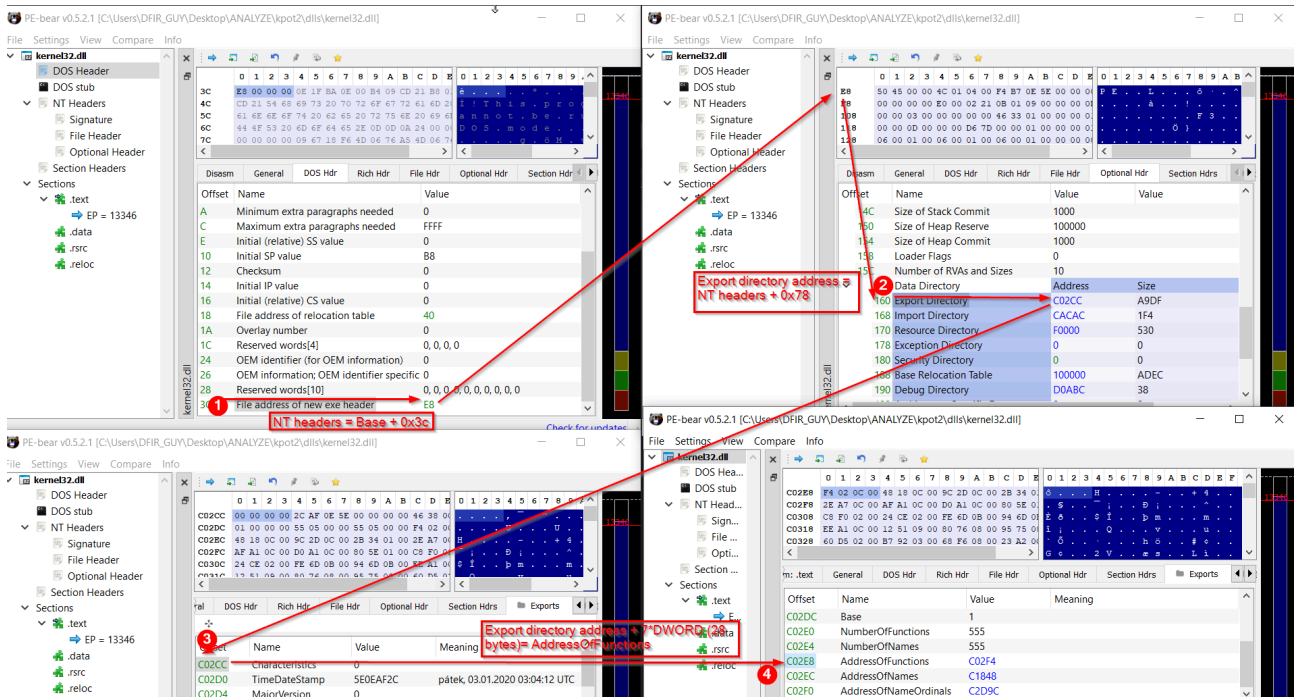
This function (sub\_4045DC) is not responsible only for finding address of LoadLibraryA but it is able to find API address via hash value of its name and base address of module as arguments.

So we can clearly rename it as function “Find\_api\_via\_HASH”. With a little help with tool like PEbear [https://github.com/hasherezade/pe-bear-releases] we could properly annotate the function sub\_4045DC - “Find\_api\_via\_HASH”. In this case where arguments to the function are kernel32.dll base address and API name hash 0x822FC0FA (LoadLibraryA), it is parsing kernel32.dll and searching for export function name which hash is 0x822FC0FA.

```

21  NT_header = *(_DWORD*)(base_address_kernel32 + 0x3C); // finding the location of NT headers
22  v20 = 0;
23  v19 = 0;
24  Export_directory_address = (_DWORD*)(NT_header + base_address_kernel32 + 0x78); // finding the location of Export directory
25  Export_directory = (_DWORD*)(base_address_kernel32 + *Export_directory_address); // start address of export directory
26  AddressOfFunctions = base_address_kernel32 + Export_directory[7]; // 28Bytes + export_directory address = AddressOfFunctions
27  AddressOfNameOrdinals = base_address_kernel32 + Export_directory[9]; // 36Bytes + export_directory address = AddressOfNameOrdinals
28  AddressOfNames = base_address_kernel32 + Export_directory[8]; // 32Bytes + export_directory address = AddressOfNames
29  False = Export_directory[6] == 0; // checking if number of NumberOfNames == 0 -> FALSE
30  v14 = Export_directory_address;
31  v18 = AddressOfNames;
32  if ( !False ) // True
33  {
34  while ( 1 )
35  {
36  Exports_function_name = (_BYTE*)(base_address_kernel32 + *(DWORD*)(AddressOfNames + 4 * v19));
37  size_of_func_name = str_length(Exports_function_name);
38  if ( Api_hashing_func(Exports_function_name, size_of_func_name) == HASH )
39  {
40  v20 = (_BYTE*)(base_address_kernel32
41  + *(DWORD*)(AddressOfFunctions + 4 * *(unsigned_int16*)(AddressOfNameOrdinals + 2 * v19));
42  if ( (unsigned_int)v19 >= Export_directory[6] )
43  break;
44  AddressOfNames = v18;
45  }
46  Export_directory_address = v14;
47  }
48  if ( v20 >= (_BYTE*)Export_directory && v20 < (_BYTE*)Export_directory + Export_directory_address[1] )
49  {
50  v8 = str_length(v20);
51  sub_4134FD(v13[0]);
52  jmp short loc_404661
    
```

We can focus more on the function Api\_hashing\_func later.



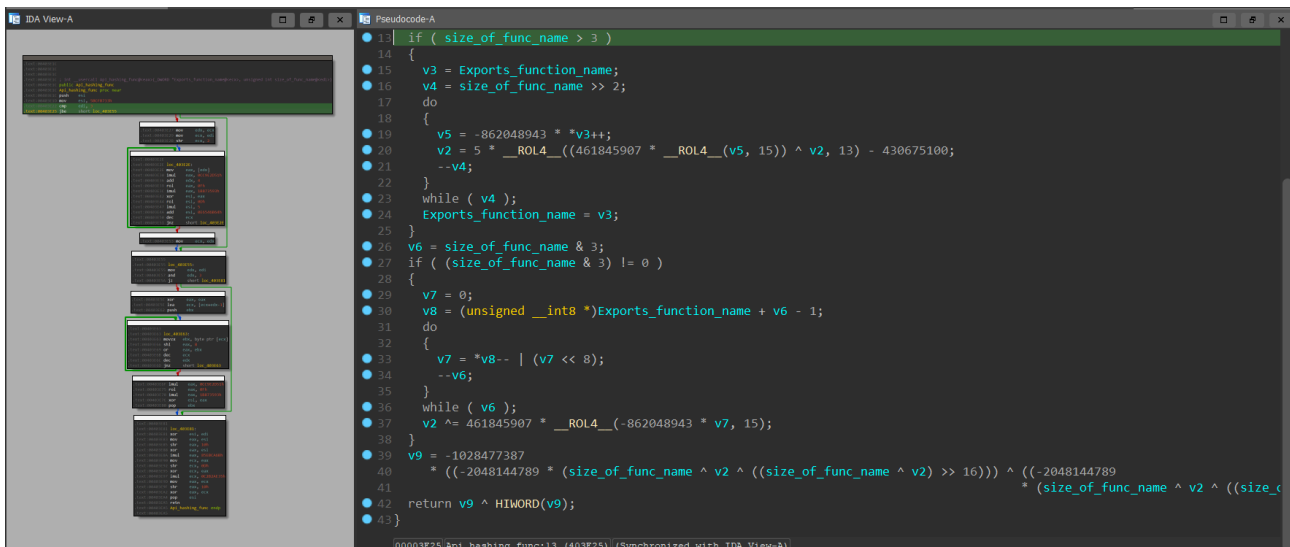
Of course we can save some time and let IDA help you with defaultly defined structs for PE. But I personally think that it is a needed skill to understand and be able to parse PE manually.

```

IMAGE_NT_HEADERS = *(IMAGE_NT_HEADERS **)(base_address + 0x3C);
v19 = 0;
v18 = 0;
v3 = (IMAGE_EXPORT_DIRECTORY *)((char *)IMAGE_NT_HEADERS->OptionalHeader.DataDirectory + base_address);
v4 = (IMAGE_EXPORT_DIRECTORY *)(base_address + v3->Characteristics);
AddressOfFunctions = base_address + v4->AddressOfFunctions;
AddressOfNameOrdinals = base_address + v4->AddressOfNameOrdinals;
AddressOfNames = base_address + v4->AddressOfNames;
False = v4->NumberOfNames == 0;
v13 = v3;
v17 = AddressOfNames;
if ( !False )
{
    while ( 1 )
    {

```

So let's jump to the function Api\_hashing\_func (0x403E1C) which you could see in the picture below is implementing some probably modified version of well-known hashing algorithm.



We could use a little help to find out what hash algorithm is implemented from another excellent tool Capa [<https://github.com/fireeye/capa>]. This gives us a hint that it could be hashing algorithm of type murmur3. We will come back to this hashing algorithm later.



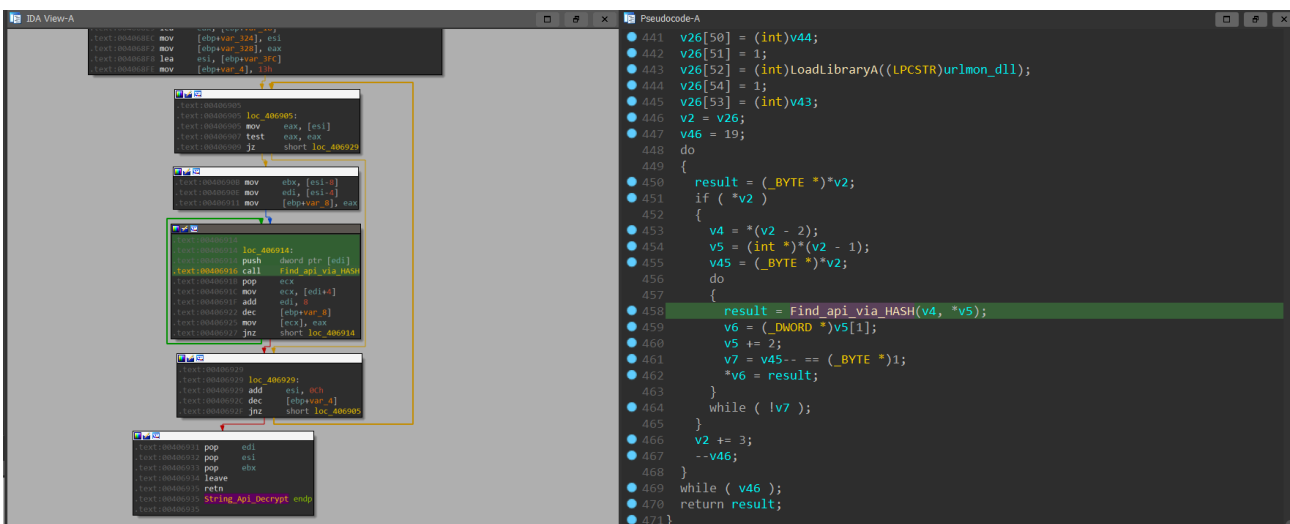
```

.text:0040694A
.text:0040694A
.text:0040694A
.text:0040694A sub_40694A proc near
.text:0040694A mov     eax, large fs:30h
.text:00406950 mov     eax, [eax+0Ch]
.text:00406953 mov     eax, [eax+0Ch]
.text:00406956 mov     eax, [eax]
.text:00406958 mov     eax, [eax+18h]
.text:0040695B retn
.text:0040695B sub_40694A endp
.text:0040695B

```

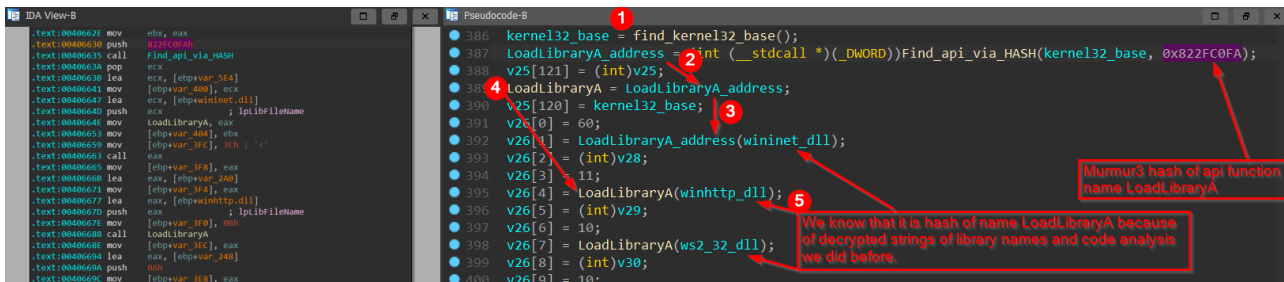
So we can continue and finally reach the interesting part.

In the picture below, we can see the last part of sub\_4058FB which we can clearly rename now as “String\_Api\_Decrypt”. This last part as you can see is responsible for resolving all API functions and saving them to .data section in memory. All these resolved API functions addresses are later in code referenced. You can see that there is a loop which is looping through all API name hashes saved on stack before and calling Find\_api\_via\_HASH.

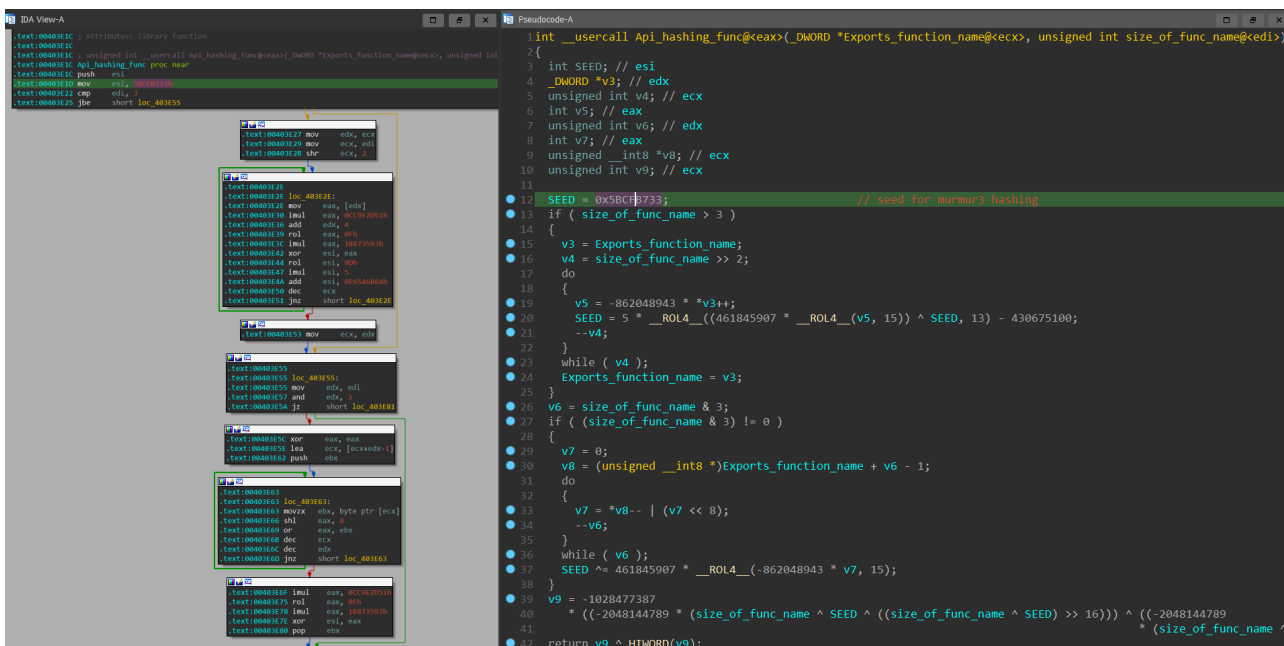


So now we have more options to obtain and populate all resolved API functions in our code. One of the option is to implement murmur3 hashing algorithm and with help of IDAPython, find all API function name hashes to process it with our algorithm. As we did some IDAPython scripting before and I want to show you different methods you can only see that our assumption about murmur3 hashing algorithm is right in the pictures below:

According to our annotated code – the hash of API function name LoadLibraryA is 0x822FC0FA



We are also able to find out that murmur3 is using Seed value 0x5BCFB733 by examining the code in function Api\_hashing\_func (0x403E1C).



To verify that it is really murmur3 hashing algorithm with seed 0x5BCFB733:

```
>>> #LoadLibraryA Hash --> 0x822FC0FA
>>> import mmh3
>>> api_name = 'LoadLibraryA'
>>> seed = 0x5BCFB733
>>> hex(mmh3.hash(api_name, seed, signed=False))
'0x822fc0fa'
>>>
>>>
```

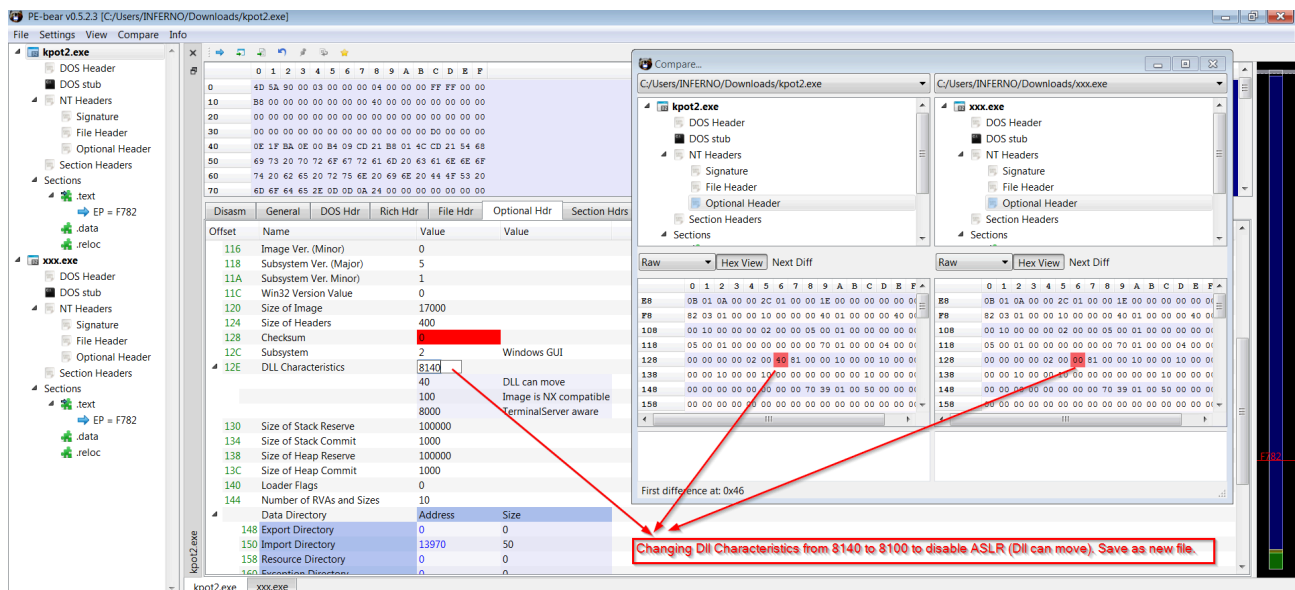
Our assumption about hashing algorithm is right so move next.

The another option to obtain and populate all resolved API functions in our code is to debug the sample kpot2 and after API functions addresses get resolved, apply plugin Scylla to reconstruct IAT – this sometimes does not work well. Option we will use and which I am finding more interesting and in this case perfectly suitable is to use tool “apiscout” [https://github.com/danielplohmann/apiscout]. This tool is extremely useful in situation like this.

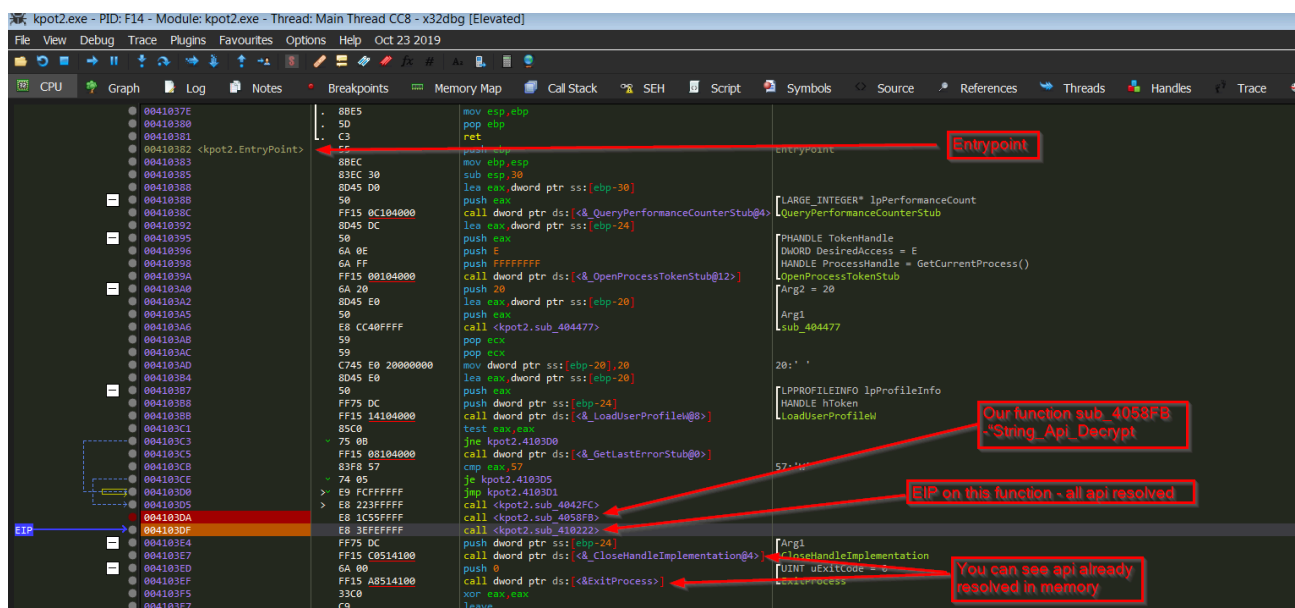
When we have all information about how the API resolving works, we could let the sample populate the resolved API function addresses in debugger, dump the process from memory and after that, we need something what is able to find in our dumped memory all populated API function addresses and annotate it for us. This is the time when apiscout comes to save the situation.

One of the feature of apiscout is creating of database of all API functions (exports of module). We can let the apiscout build the database from all dlls on our system or we can select only some of them. It is basically parsing all modules exports and creating database with information like name of API function, VA, ASLR offset etc...

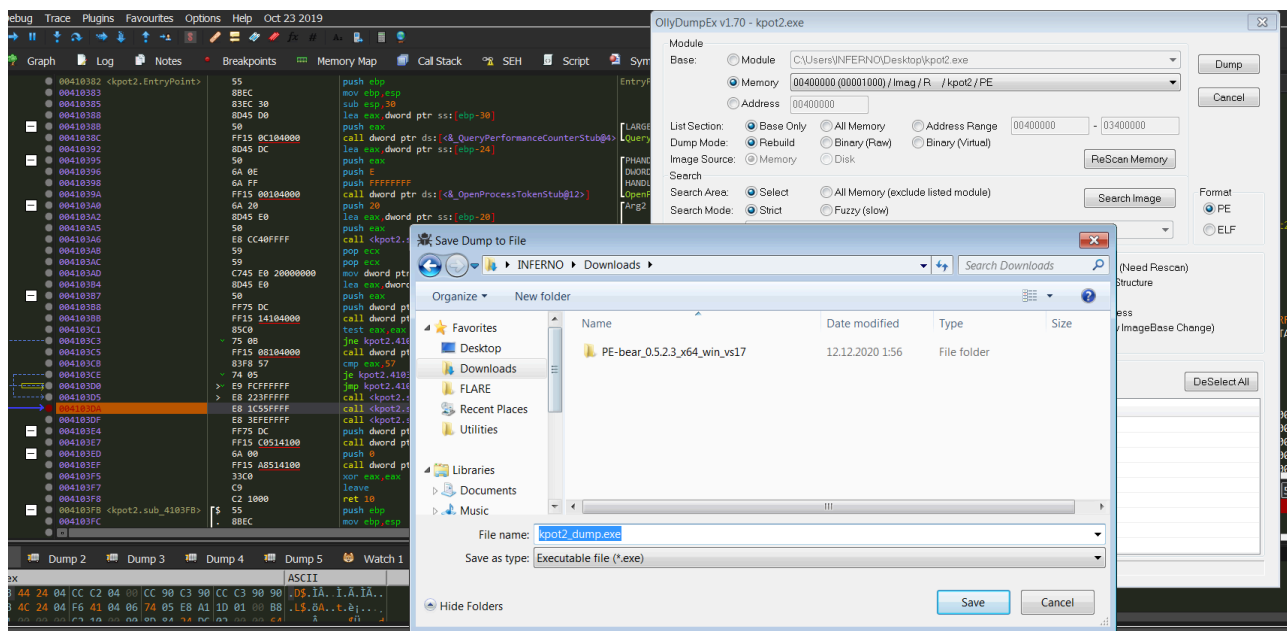
Let's start with dumping our kpot2.exe process from memory in debugger like x64dbg after it populates the resolved API function addresses. We put a breakpoint after the call sub\_4058FB - "String\_Api\_Decrypt" and dump the process. To find location of this function in debugger easily, do not forget to disable ASLR in the optional header of kpot2.exe.



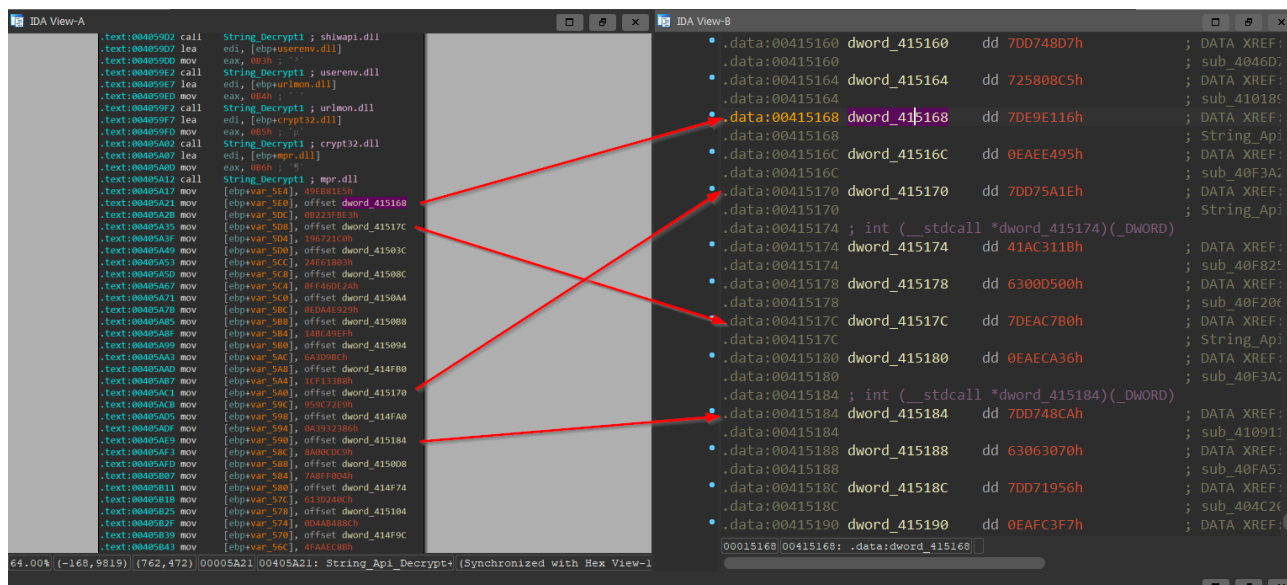
Locating our sub\_4058FB - "String\_Api\_Decrypt function.



Dumping the kpot2.exe process from memory with plugin OllyDumpEx.



Confirmation in IDA that all referenced API addresses are already populated in our kpot2 process dump “kpot2\_dump.bin”:



Apiscout is able to work also on system with ASLR enabled but in case we want to choose apiscout option to ignore ASLR, we must disable the ASLR before we perform the process dump of kpot2.exe – find registry key:

[HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management]

Create a new dword value: “MoveImages” = dword:00000000 (without quote)

Restart system.

If we do not want to create database of all dlls from our system, first of all we should find and copy to some location all dlls which is our sample kpot2.exe loading and processing:



```

C:\Users\INFERNO\Desktop\kpot2\apiscout-master\apiscout\adb_builder>py -3 DatabaseBuilder.py
usage: DatabaseBuilder.py [-h] [--filter] [--auto] [--paths P [P ...]]
                        [--outfile OUTPUT_FILE] [--ignore_aslr]
                        [--aslr_check]

Build a database to be used by apiscout.

optional arguments:
  -h, --help            show this help message and exit
  --filter              (optional) filter DLLs by name (see config.py)
  --auto               Use default configuration (filtered DLLs from
                        preconfigured paths (see config.py) and extract ASLR
                        offsets.
  --paths P [P ...]   the paths to recursively crawl for DLLs (None -> use
                        default, see config.py).
  --outfile OUTPUT_FILE
                        (optional) filepath where to put the resulting API DB
                        file.
  --ignore_aslr        Do not perform extraction of ASLR offsets.
  --aslr_check         Only show ASLR offset.

C:\Users\INFERNO\Desktop\kpot2\apiscout-master\apiscout\adb_builder>py -3 DatabaseBuilder.py --paths C:\Users\INFERNO\Desktop\kpot2\dlls --ignore_aslr --outfile kpot2_DB.json
2020-12-13 02:50:49,699 processing: C:\Users\INFERNO\Desktop\kpot2\dlls advapi32.dll
2020-12-13 02:50:50,355 APIs: 807
2020-12-13 02:50:50,355 processing: C:\Users\INFERNO\Desktop\kpot2\dlls api-ms-win-downlevel-advapi32-11-1-0.dll
2020-12-13 02:50:50,386 APIs: 145
2020-12-13 02:50:50,386 processing: C:\Users\INFERNO\Desktop\kpot2\dlls api-ms-win-downlevel-advapi32-12-1-0.dll
2020-12-13 02:50:50,386 APIs: 14
2020-12-13 02:50:50,386 processing: C:\Users\INFERNO\Desktop\kpot2\dlls api-ms-win-downlevel-normaliz-11-1-0.dll
2020-12-13 02:50:50,402 APIs: 2
...
2020-12-13 02:51:29,808 processing: C:\Users\INFERNO\Desktop\kpot2\dlls WSHTCPIP.DLL
2020-12-13 02:51:29,826 APIs: 16
2020-12-13 02:51:29,840 PEs examined: 77 (0 duplicates, 0 skipped)
2020-12-13 02:51:29,840 Successfully evaluated 77 DLLs with 17609 APIs
    
```

Path to your extracted dlls used by kpot2

Only when you have on your SYSTEM ASLR disabled

Now when we have build our kpot2\_DB.json, before we apply it to our previously created process dump file in IDA “kpot2\_dump.bin”, we can verify that apiscout is able to find all API functions in our dump according to kpot2\_DB.json. For this purpose, we use apiscout tool “scout.py” as you can see in the picture below.

```

C:\Users\DFIR_GUY\Desktop\ANALYZE\kpot2\apiscout-master\apiscout-master_latest_errors_patched_BY_ME>py -3 scout.py kpot2_dump.bin kpot2_DB.json
Using base address 0x0 to infer reference counts.
2020-12-13 15:30:07,150 loaded 17609 exports from 77 DLLs (Windows 7 Service Pack 1 (AMD64)) with 8 potential collisions.
Using
  kpot2_DB.json
to analyze
  kpot2_dump.bin.
Buffer size is 94208 bytes, 16633 APIs loaded.

Results for API DB: Windows 7 Service Pack 1 (AMD64)
idx: offset ; VA ; IT?; #ref;DLL ; API
1: 0x00001000; 0x77c74234; yes; 2; advapi32.dll_0x77c60000 (32bit) ; OpenProcessToken
2: 0x00001008; 0x7dd711c0; yes; 2; kernel32.dll_0x7dd60000 (32bit) ; GetLastError
3: 0x0000100c; 0x7dd716f5; yes; 2; kernel32.dll_0x7dd60000 (32bit) ; QueryPerformanceCounter
4: 0x00001014; 0x406a1ab4; yes; 2; userenv.dll_0x406a0000 (32bit) ; LoadUserProfileW
-----
5: 0x00014f64; 0x73813c39; no; 3; shell32.dll_0x73800000 (32bit) ; ShellExecuteW
6: 0x00014f68; 0x77c70d57; no; 2; advapi32.dll_0x77c60000 (32bit) ; GetSidSubAuthority
-----
160: 0x000151d4; 0x6fc400c0; no; 1; oleaut32.dll_0x6fc30000 (32bit) ; SafeArrayUnaccessData
161: 0x000151d8; 0x6fc34757; no; 3; oleaut32.dll_0x6fc30000 (32bit) ; SysAllocString
162: 0x000151dc; 0x7dc74053; no; 3; user32.dll_0x7dc50000 (32bit) ; GetKeyboardLayoutList
163: 0x000151e0; 0x41ac3c5f; no; 2; ws2_32.dll_0x41ac0000 (32bit) ; WSACleanup
DLLs: 19, APIs: 161, references: 284

WinApi1024 Vector Results:
Windows 7 Service Pack 1 (AMD64): 135 / 159 (84.91%) APIs covered in WinApi1024 vector.
Vector: A19BAAGa3gABA7E45EA1A10CAgA6IQA5CA4G3IAACAIAIMACAQA5gABAakINQAAIAQIB_gAABAEAAQEAFAAGAEYqQAKIWIip,ACMkKqyF**AWnRIGU+CoychvJc
confidence: 88.61067572167505
    
```

Previously created database with "DatabaseBuilder.py"

Previously created process dump of kpot2

What is WinApi1024 vector?

We can see that apiscout was successful and there is more – something called “WinApi1024 vector”. Basically speaking it is something like ImpHash on steroids. You can read more about Apivector here: <https://byte-atlas.blogspot.com/2018/04/apivectors.html>. As we get WinApi1024 vector of our kpot2\_dump.bin calculated, we can use it against big database maintained on Malpedia which is covering big amount of well-known malware families <https://malpedia.caad.fkie.fraunhofer.de/apiqr/>. We can see that our WinApi1024 vector is matched 100% with family “win.kpot\_stealer” below.



Inventory Statistics Usage ApiVector Login

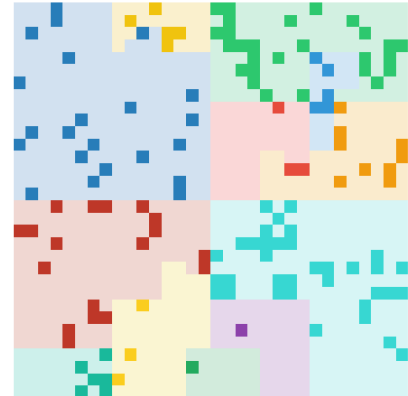
### ApiQR

You are looking at the results for:

A19BAAgA3gABA7E5EAIA10CAgA6IQA5CA4GA3IAACAAIAMCAQA5gABAaKIQAAIAQIB\_gAABAEAAQEAAFAAgAEMyqQAKI  
IwIp,ACMkKqyF^\*AWnRIgU+C0ychvjC

Input another ApiVector. Process

On the right side you can see a visualization of this ApiVector as generated by the **ApiScout** library. Solid squares indicate the presence of the respective Windows API function in the vector.



The table below shows the results of matching against the ApiVectors of all currently dumped samples in Malpedia.

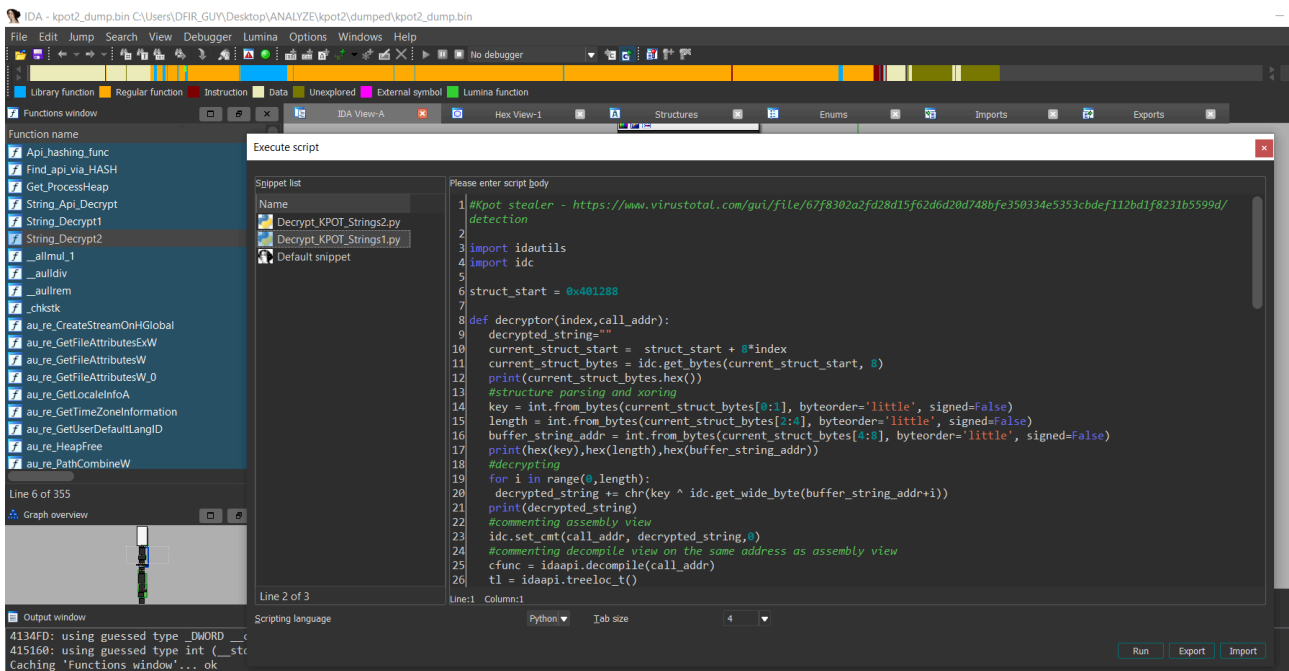
### Match Results:

Top 10 Family Matches:

Family	Score
win.kpot_stealer	100.00%
win.azorult	50.93%
win.raccoon	28.15%

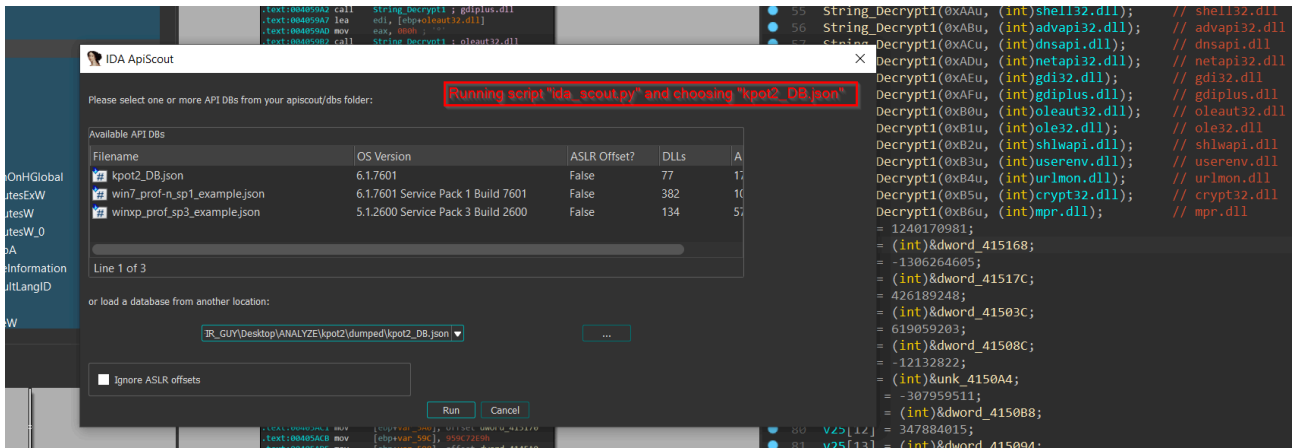
To apply all previously annotated names of functions from previous IDA database file to our newly created kpot2 process dump “kpot2\_dump.bin”, we could use IDA plugin called “rizzo” [<https://github.com/tacnetsol/ida/tree/master/plugins/rizzo>].

After that, previously created IDAPython scripts for decrypting strings must be run again (Decrypt\_KPOT\_Strings1.py, Decrypt\_KPOT\_Strings2.py) [[View here](#)]

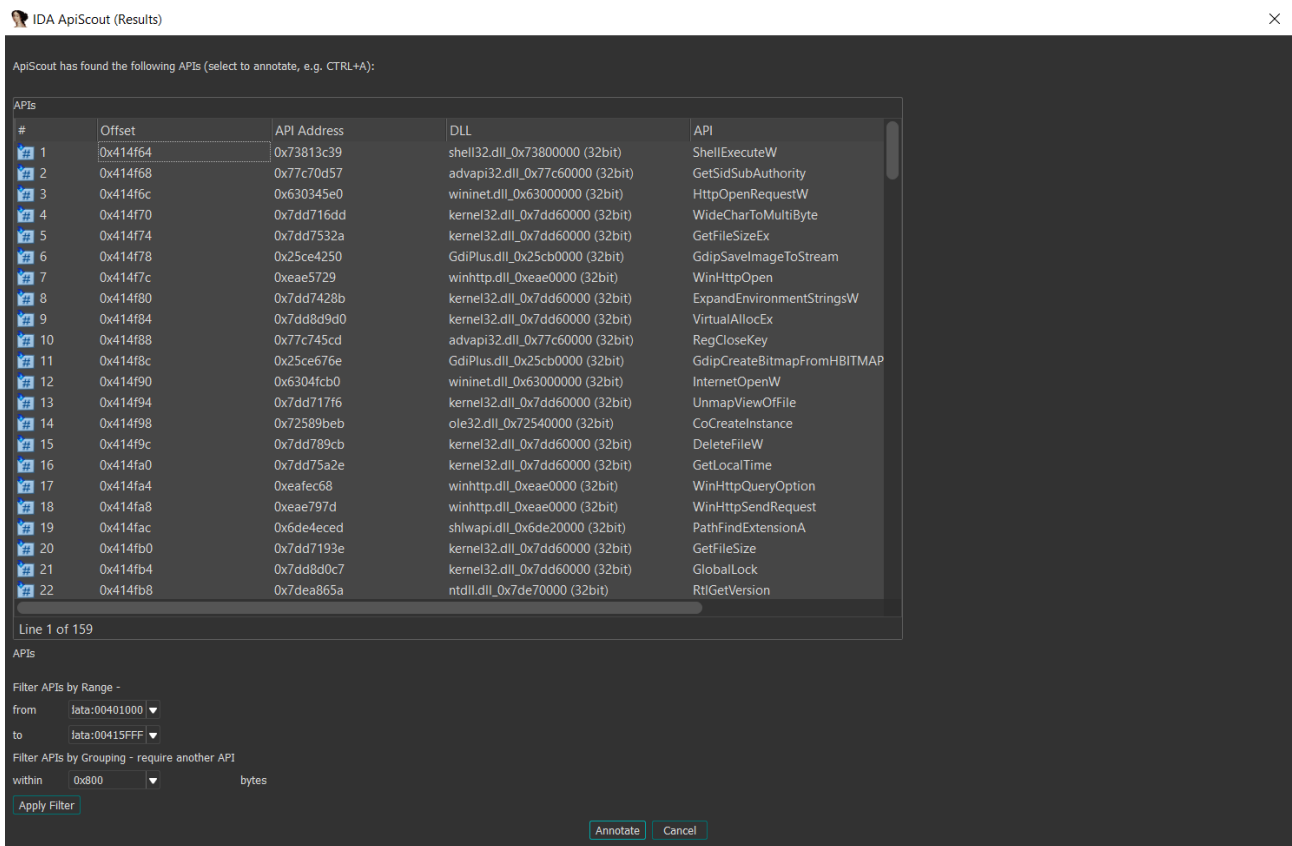


Now we are almost in the same state with “kpot2\_dump.bin” as we were in the original sample.

Let's continue to apply our created database kpot2\_DB.json to process dump kpot2\_dump.bin in context of IDA. We will use apiscout IDAPython script "ida\_scout.py" for that.



In the next window choose all of the found APIs and click "Annotate".



After apiscout is done we can check the results – all referenced API addresses are annotated with their names and type.

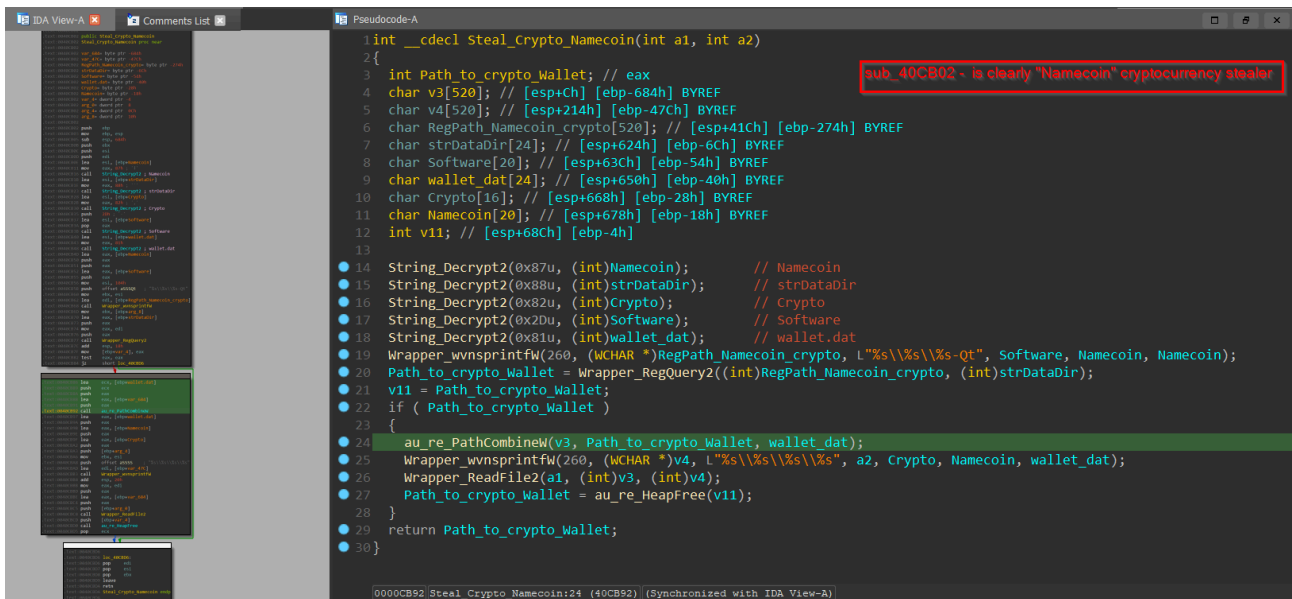
```
text:00406500 mov [ebp+var_24c], offset GetUserNames
text:00406517 mov [ebp+var_24d], offset GetTokenInformation
text:00406521 mov [ebp+var_18], offset wvsprintfA
text:00406537 mov [ebp+var_10], offset wvsprintfA
text:00406549 mov [ebp+var_0c], offset wvsprintfW
text:00406555 mov [ebp+var_08], offset StrToInt64ExA
text:00406570 mov [ebp+var_00], offset PathFindExtensionA
text:00406578 mov [ebp+var_04], offset PathFindExtensionA
text:0040658f mov [ebp+var_48], offset SHGetFolderPath
text:00406590 mov [ebp+var_04], offset SHGetFolderPath
text:004065a4 mov [ebp+var_3c], offset ShellExecuteW
text:004065a8 mov [ebp+var_28], offset DnsQuery_A
text:004065b7 mov [ebp+var_2c], offset DnsQuery_A
text:004065c1 mov [ebp+var_34], offset LoadUserProfile_0
text:004065d5 mov [ebp+var_30], offset UnloadUserProfile
text:004065e3 mov [ebp+var_2c], offset MmOpenEnumW
text:004065f1 mov [ebp+var_20], offset MmCloseEnum
text:00406600 mov [ebp+var_1c], offset CryptUnprotectData
text:00406618 mov [ebp+var_18], offset CryptUnprotectData
text:00406622 mov [ebp+var_14], offset IsValidURL
text:00406629 call Find_kernel32_base
text:00406632 mov ebx, eax
text:00406638 push 822fc0f8h
text:0040663a call Find_api_via_HASH
text:0040663c pop ecx
text:0040663e lea [ebp+var_564], ecx
text:00406641 mov [ebp+var_400], ecx
text:00406647 lea [ebp+var_01d1], ecx
text:0040664d push ; lpLibFileName
text:00406653 mov LoadLibraryA, eax
text:00406659 mov [ebp+var_400], ebx
text:00406663 call eax
text:00406667 mov [ebp+var_3f4], eax
text:00406677 lea [ebp+var_01d1], ecx
text:00406679 push ; lpLibFileName
text:0040667b mov [ebp+var_3f0], ebx
text:00406688 call LoadLibraryA
text:0040668e mov [ebp+var_3f0], eax
text:00406694 lea [ebp+var_240], ebx
text:00406696 push ebx
.data:00415134 ; BOOL __stdcall GetFileAttributesExW(LPCWSTR lpFileName, GET_FILEEX_INFO_1
.data:00415134 GetFileAttributesExW dd 7DD745E3h ; DATA XREF: au_re_GetFileAttribute
; String Api Decrypt+36Ato
.data:00415138 ; BOOL __stdcall GetTokenInformation(HANDLE TokenHandle, TOKEN_INFORMATION_
| ; DATA XREF: sub_4049A3+2Afr
; sub_4049AF+29fr ...
.data:00415138 GetTokenInformation dd 77C742ACh
.data:00415138 ; DWORD __stdcall NetUserEnum(LPCWSTR servername, DWORD level, DWORD filter
; sub_4049AF+29fr ...
.data:0041513C NetUserEnum dd 2C859CFh ; DATA XREF: String_Api_Decrypt+745
; sub_411101+11Bfr
.data:0041513C ; int (__stdcall *FindClose)(DWORD)
.data:00415140 FindClose dd 7DD74481h ; DATA XREF: sub_404156+123fr
; String_Api_Decrypt+446to ...
.data:00415140 ; int (__stdcall *GetPrivateProfileStringW)(DWORD, DWORD, DWORD, DWORD,
; String_Api_Decrypt+586to ...
.data:00415144 GetPrivateProfileStringW dd 7DD7EA48h ; DATA XREF: String_Api_Decrypt+586
; sub_40D0F7+F1fr ...
.data:00415144 ; HANDLE __stdcall CreateFileMappingW(HANDLE hFile, LPSECURITY_ATTRIBUTES l
; String_Api_Decrypt+482to ...
.data:00415148 CreateFileMappingW dd 7DD7718D9h ; DATA XREF: sub_403FA4+55fr
; String_Api_Decrypt+5C2to ...
.data:00415148 ; DWORD __stdcall WNetEnumResourceW(HANDLE hEnum, LPDWORD lpcCount, LPVOID
; String_Api_Decrypt+CFto ...
.data:0041514C WNetEnumResourceW dd 408C3058h ; DATA XREF: String_Api_Decrypt+CF
; sub_40D05E+BEfr
.data:00415150 ; int (__stdcall *BitBlt)(DWORD, DWORD, DWORD, DWORD, DWORD, _
; String_Api_Decrypt+A82to ...
.data:00415150 BitBlt dd 7DAC5EA5h ; DATA XREF: String_Api_Decrypt+A82
; sub_4116A3+93fr
.data:00415150 ; DWORD __stdcall GetTempPathW(DWORD nBufferLength, LPWSTR lpBuffer)
; String_Api_Decrypt+5C2to ...
.data:00415154 GetTempPathW dd 7DD8D4FCh ; DATA XREF: sub_4049C7+17fr
; String_Api_Decrypt+5C2to ...
.data:00415154 ; int (__stdcall *GetVolumeInformationW)(DWORD, DWORD, DWORD, _
; String_Api_Decrypt+30Eto ...
.data:00415158 GetVolumeInformationW dd 7DD8C888h ; DATA XREF: String_Api_Decrypt+30E
; sub_412A59+1Cfr
.data:0041515C ; int (__stdcall *CryptUnprotectData)(DWORD, DWORD, DWORD, _
; String_Api_Decrypt+30Eto ...
```

```
text:00406567 mov [ebp+var_00], offset PathCombineW
text:00406571 mov [ebp+var_04], offset PathFindExtensionA
text:00406578 mov [ebp+var_08], offset PathFindExtensionA
text:0040658f mov [ebp+var_48], offset SHGetFolderPath
text:00406590 mov [ebp+var_04], offset SHGetFolderPath
text:004065a4 mov [ebp+var_3c], offset ShellExecuteW
text:004065a8 mov [ebp+var_28], offset DnsQuery_A
text:004065b7 mov [ebp+var_2c], offset DnsQuery_A
text:004065c1 mov [ebp+var_34], offset LoadUserProfile_0
text:004065d5 mov [ebp+var_30], offset UnloadUserProfile
text:004065e3 mov [ebp+var_2c], offset MmOpenEnumW
text:004065f1 mov [ebp+var_20], offset MmCloseEnum
text:00406600 mov [ebp+var_1c], offset CryptUnprotectData
text:00406618 mov [ebp+var_18], offset CryptUnprotectData
text:00406622 mov [ebp+var_14], offset IsValidURL
text:00406629 call Find_kernel32_base
text:00406632 mov ebx, eax
text:00406638 push 822fc0f8h
text:0040663a call Find_api_via_HASH
text:0040663c pop ecx
text:0040663e lea [ebp+var_564], ecx
text:00406641 mov [ebp+var_400], ecx
text:00406647 lea [ebp+var_01d1], ecx
text:0040664d push ; lpLibFileName
text:00406653 mov LoadLibraryA, eax
text:00406659 mov [ebp+var_400], ebx
text:00406663 call eax
text:00406667 mov [ebp+var_3f4], eax
text:00406677 lea [ebp+var_01d1], ecx
text:00406679 push ; lpLibFileName
text:0040667b mov [ebp+var_3f0], ebx
text:00406688 call LoadLibraryA
text:0040668e mov [ebp+var_3f0], eax
text:00406694 lea [ebp+var_240], ebx
text:00406696 push ebx
368 v42[0] = -147146881;
369 v42[1] = (int)&DnsQuery_A;
370 v42[2] = -338324061;
371 v42[3] = (int)&DnsFree;
372 v41[0] = 592630482;
373 v41[1] = (int)LoadUserProfile_0;
374 v41[2] = -1795771432;
375 v41[3] = (int)UnloadUserProfile;
376 v38[0] = 662776531;
377 v38[1] = (int)WNetOpenEnumW;
378 v38[2] = -589714077;
379 v38[3] = (int)WNetEnumResourceW;
380 v38[4] = 862655940;
381 v38[5] = (int)WNetCloseEnum;
382 v44[0] = 1504645864;
383 v44[1] = (int)&CryptUnprotectData;
384 v43[0] = 1875065521;
385 v43[1] = (int)IsValidURL;
386 kernel32_base = find_kernel32_base();
387 LoadLibraryA_address = (int (__stdcall *) (DWORD))Find_api_via_HASH(kernel32_base, 0);
388 v25[121] = (int)v25;
389 LoadLibraryA = LoadLibraryA_address;
390 v25[120] = kernel32_base;
391 v26[0] = 60;
392 v26[1] = LoadLibraryA_address(wininet_dll);
393 v26[2] = (int)v26;
394 v26[3] = 11;
395 v26[4] = LoadLibraryA(winhttp_dll);
396 v26[5] = (int)v29;
397 v26[6] = 10;
398 v26[7] = LoadLibraryA(ws_32_dll);
```

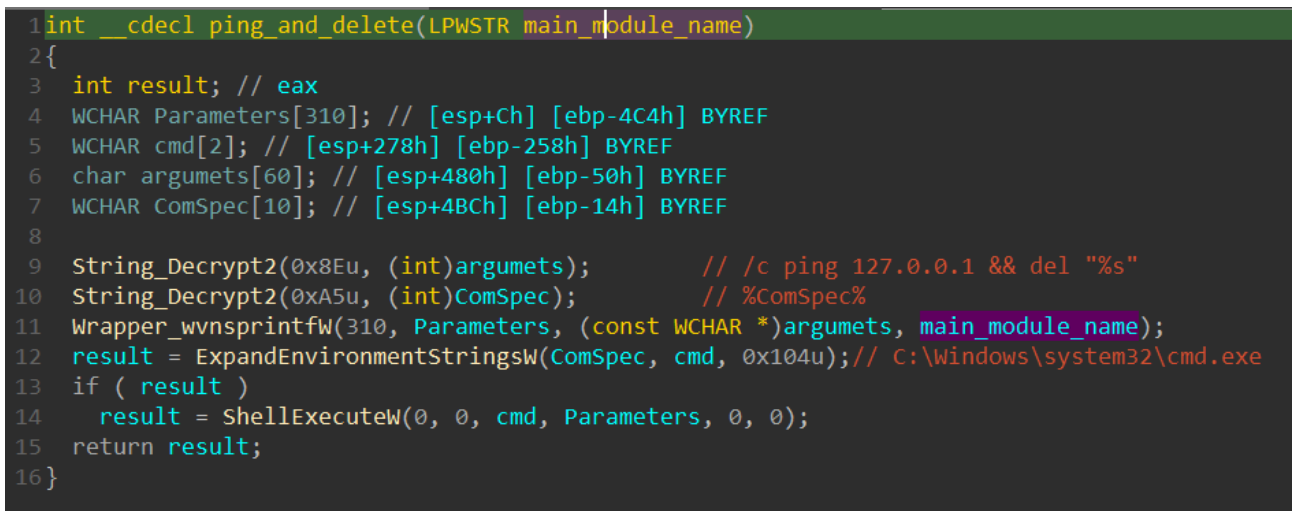
Now we are in state where we have all strings decrypted, all API function calls resolved and annotated so we are ready to benefit from it in analysis.

The analysis of the sample is now a simple task so for brevity, I will show only some of functions. Capabilities of the functions are now usually self-explanatory.

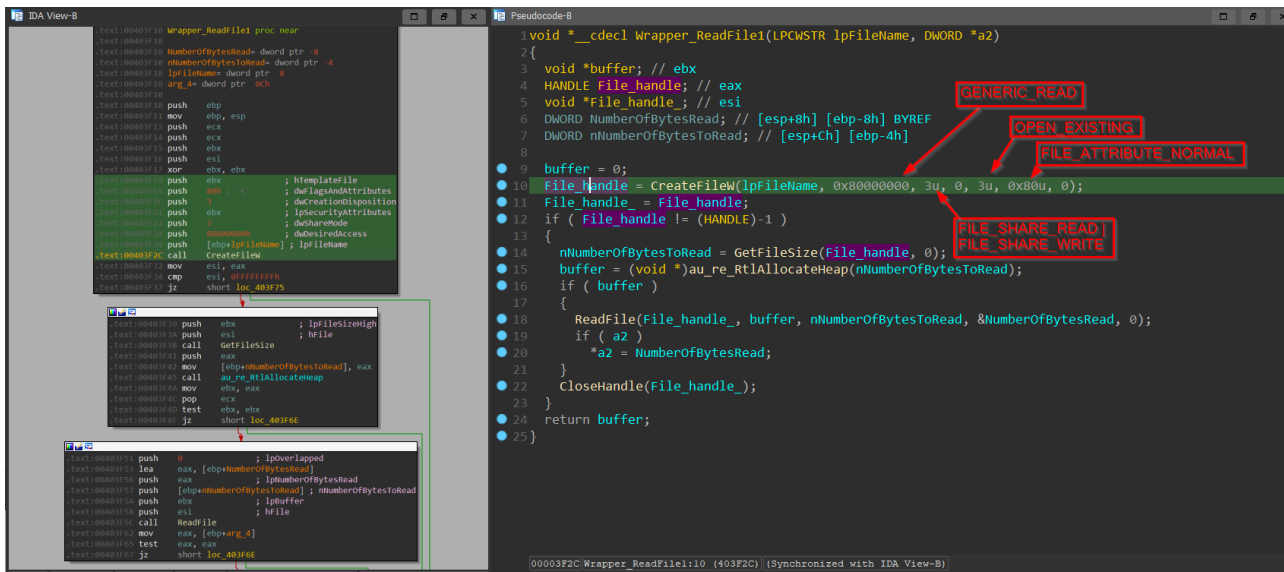
sub\_40CB02 - is clearly "Namecoin" cryptocurrency stealer:



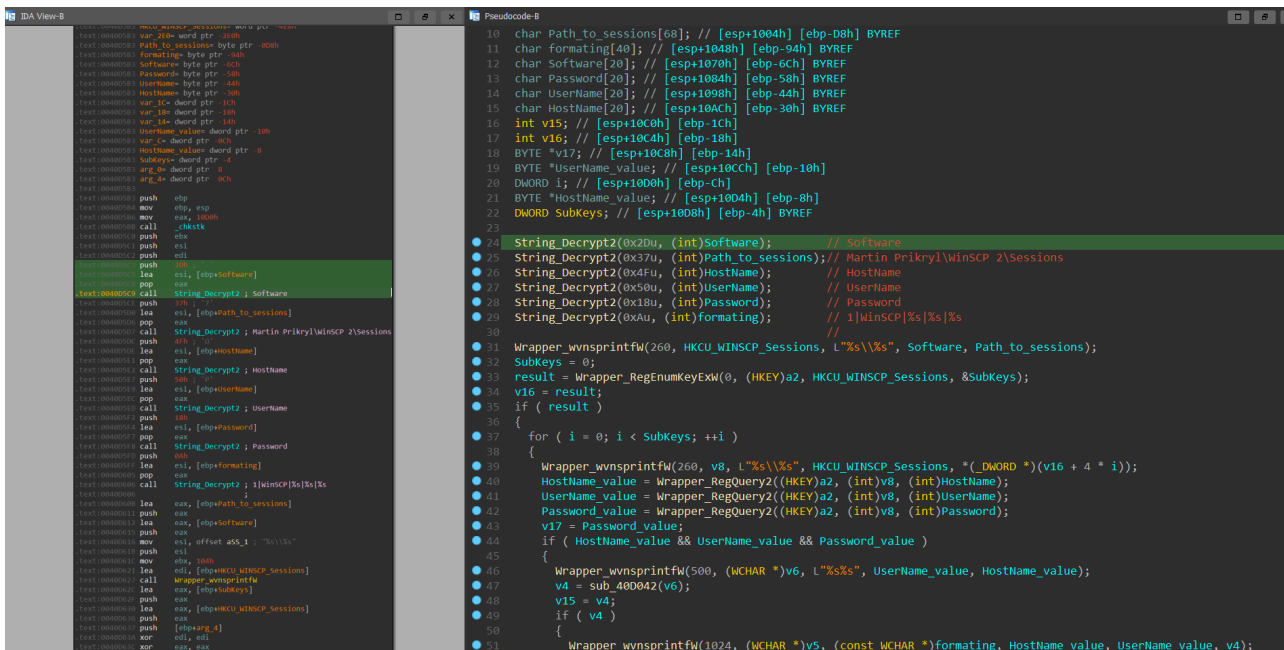
sub\_4101AB – ping + delete main module (kpot2.exe) always called before exit().



We can also easily rename wrapped functions when we have all API functions resolved:



### sub\_40D5B3 - WinSCP 2 sessions information stealer.



## Conclusion:

Kpot2 stealer is able to exfiltrate account information and other data from web browsers, instant messengers, email, VPN, RDP, FTP, cryptocurrency, and gaming software.

Most of them:

Firefox, Internet Explorer, cryptocurrency: (Ethereum, Electrum, Namecoin, Monero) Wallets - Jaxx Liberty, Exodus, TotalCommander FTP, FileZilla, WinSCP 2, Ipswitch ws\_ftp, Battle.net, Steam, Skype, Telegram, Discordapp, Pidgin, Psi, Outlook, RDP, NordVPN, EarthVPN.

It is almost impossible to cover all of stealing/exfiltrating functions here and it wasn't even my intention. I wanted to cover some tricky techniques during reversing and hope that anybody could find something from this analysis

useful or even interesting.

If you find it useful and want to share it on your blog or somewhere else, you can, just let me know if you would like to get it in better format for sharing.

Thank you to everybody who was able to read it to the end.

## **Author:**

[\[Twitter\]](#)

[\[Github\]](#)

## **Download:**

[\[Download PDF\]](#)

---

Source: <https://github.com/Dump-GUY/Malware-analysis-and-Reverse-engineering/blob/main/kpot2/KPOT.md>