Reverse engineering KPOT v2.0 Stealer

github.com/Dump-GUY/Malware-analysis-and-Reverse-engineering/blob/main/kpot2/KPOT.md

Dump-GUY

Dump-GUY/**Malware-analysis-and-Reverse-...**

Some of my publicly available Malware analysis and Reverse engineering.

1 Contributor   0 Issues   410 Stars   80 Forks

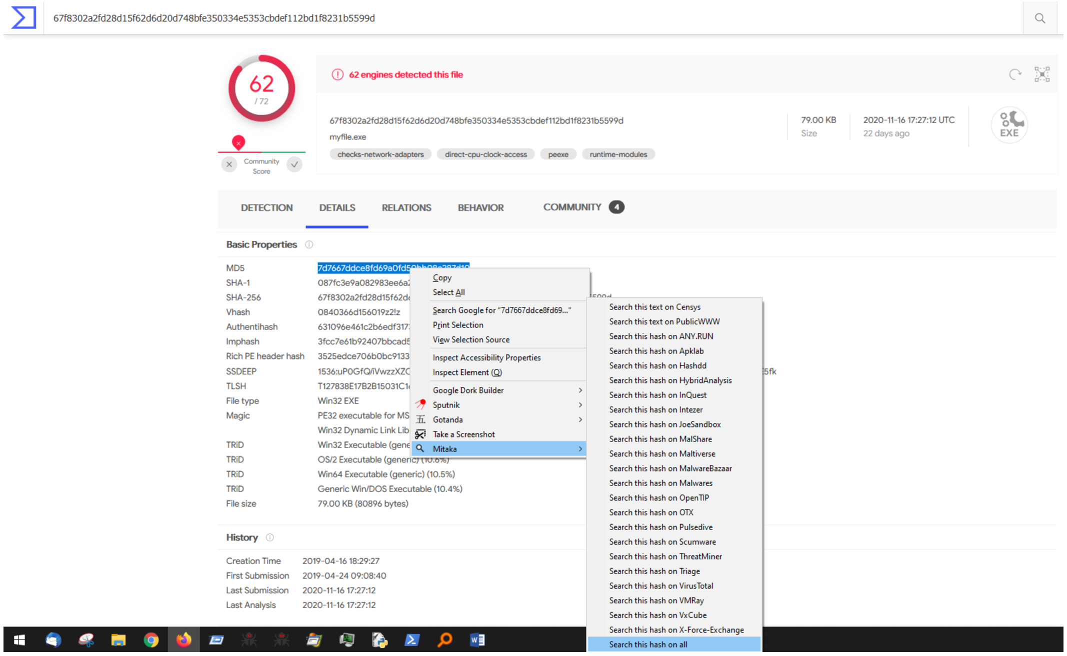
main

Malware-analysis-and-Reverse-engineering/kpot2/KPOT.md

Cannot retrieve contributors at this time

KPOT Stealer is a “stealer” malware that focuses on exfiltrating account information and other data from web browsers, instant messengers, email, VPN, RDP, FTP, cryptocurrency, and gaming software.
Sample:[Virustotal]

At first it is usually good to start with a little recon about this sample. For this purpose, I usually use browser extension called “Mitaka” [https://github.com/ninoseki/mitaka]. This is very useful browser extension for IOC OSINT search.



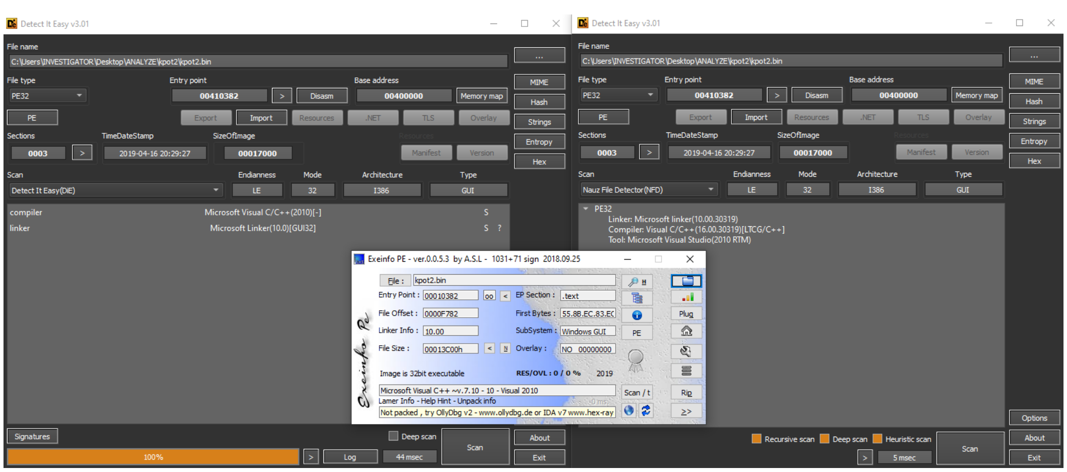
To be more sure about first assumption that it could be a “kpot” stealer, it is also good to perform a YARA scanning on this sample. I prefer YARA rules from Malpedia. [https://malpedia.caad.fkie.fraunhofer.de/]

```
C:\Users\INVESTIGATOR\Desktop\MALWARE_TOOLS\OFFLINE_SCANNERS\yara-v4.0.1-1323-win64>yara64.exe -w merged.yar kpot2
win_kpot_stealer_auto kpot2
```

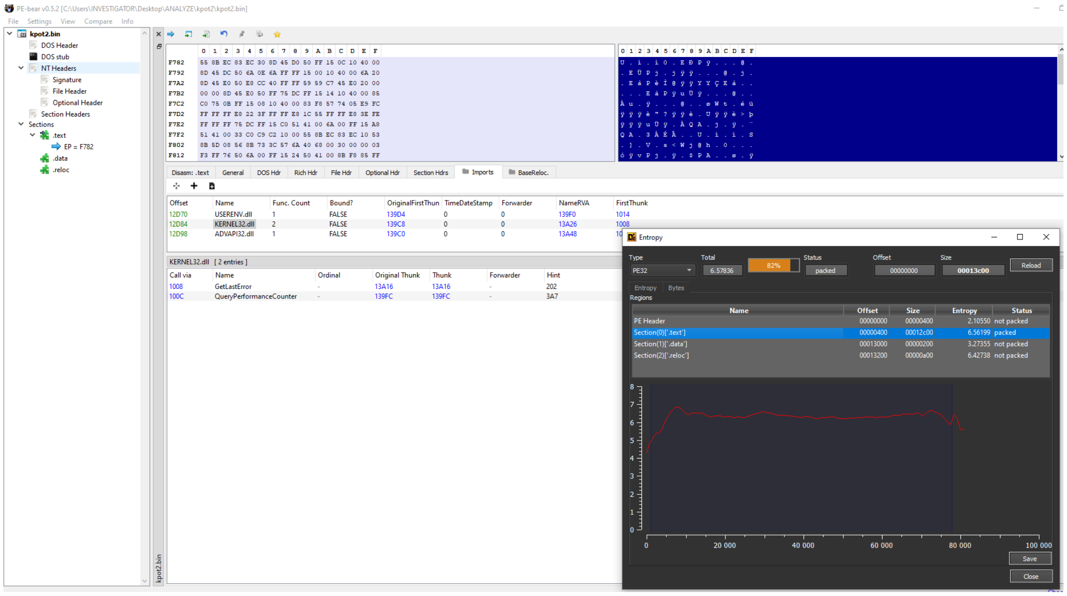
So where to start? Usually one of my first questions is: “Is it packed or somehow encrypted?”

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.

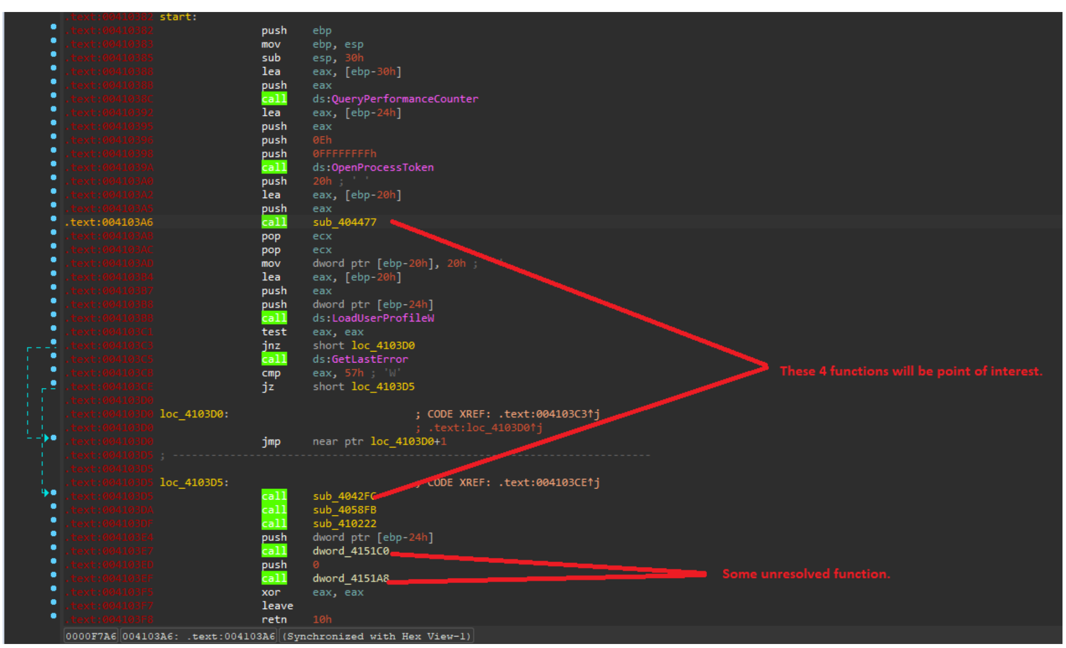


You can see that the sample has only 4 imports and the entropy of the .text code section is too high – packed.

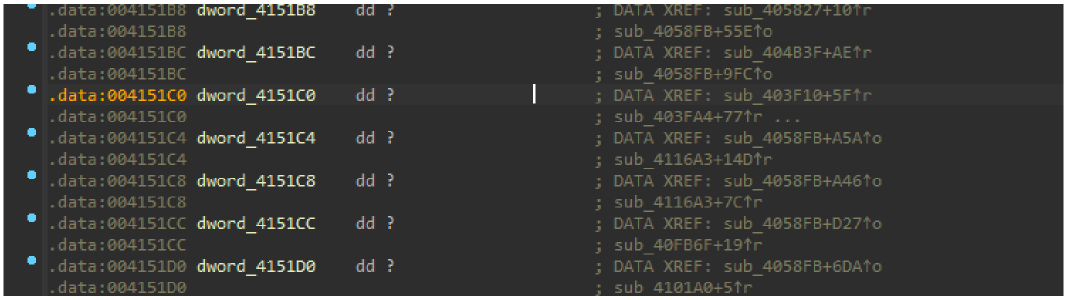
So for now we know that we have to deal with sample which is some kind of stealer and it is probably encrypted or packed.

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.



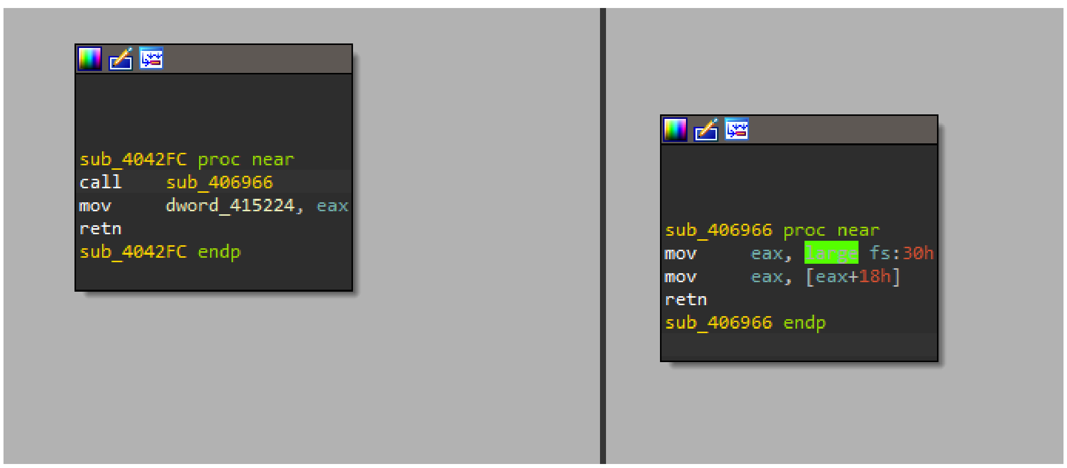
You can see also unresolved calls like “call dword_4151C0” – these calls are pointing to some location in .data section which is now empty and probably gets filled with addresses later.



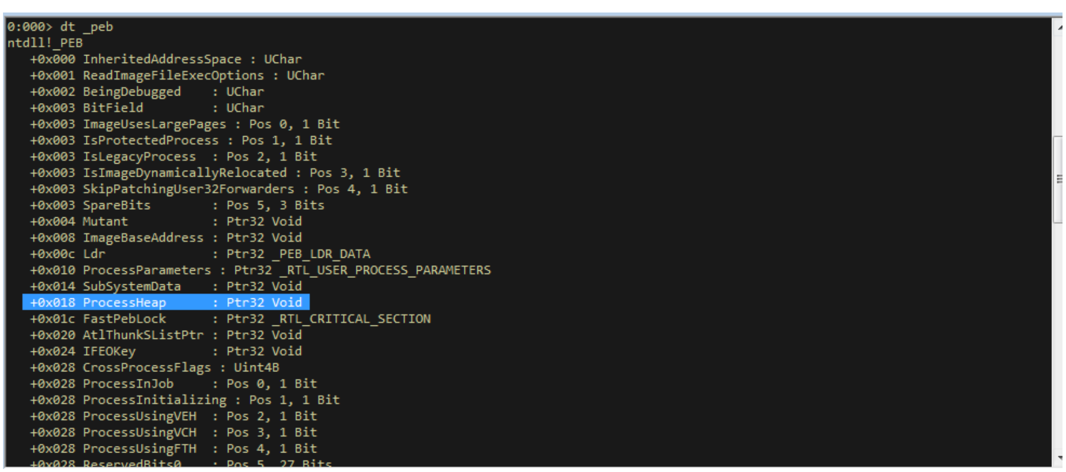
So we have almost no imports and plenty of unresolved calls. Let’s start with the 4 interesting functions mentioned before.

First function is sub_404477 – this function is not interesting at all. It is only clearing 20 bytes in memory for call LoadUserProfileW.

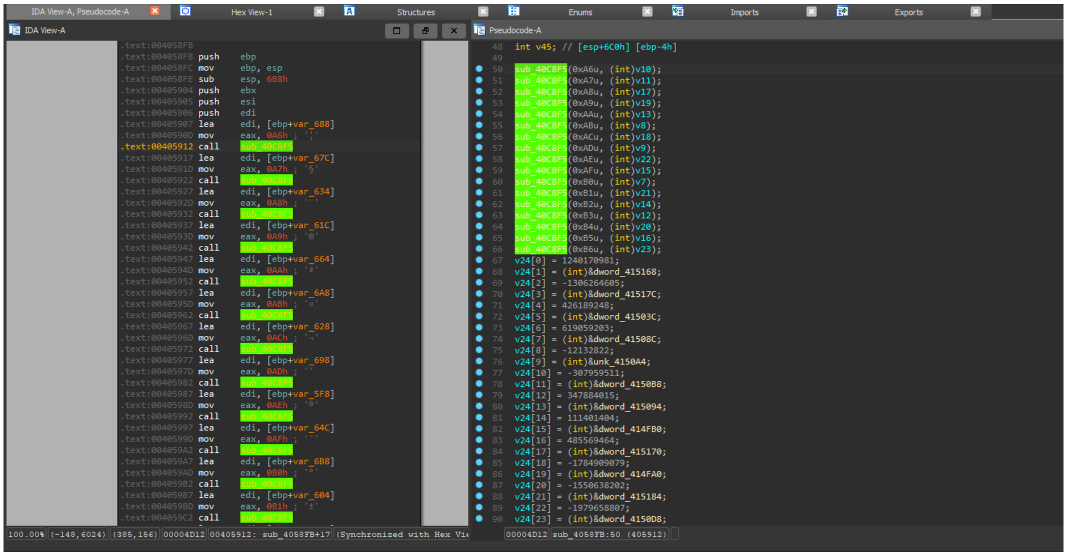
So let’s continue to another call sub_4042FC. This function is locating PEB exactly ProcessHeap and saving it to location dword_415224.



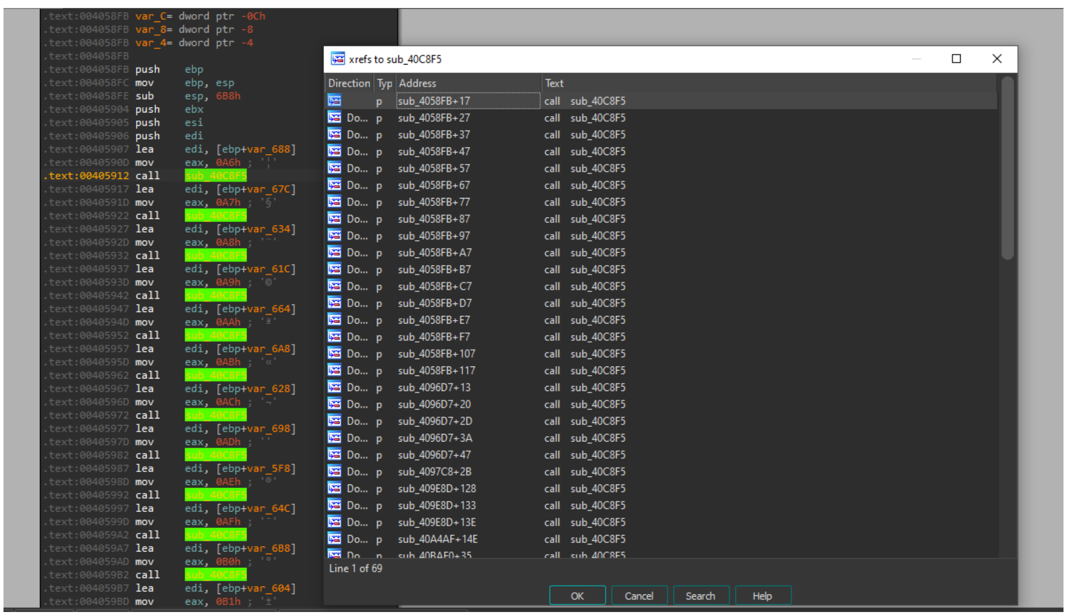
We can confirm it in windbg where we can easily parse PEB structure.



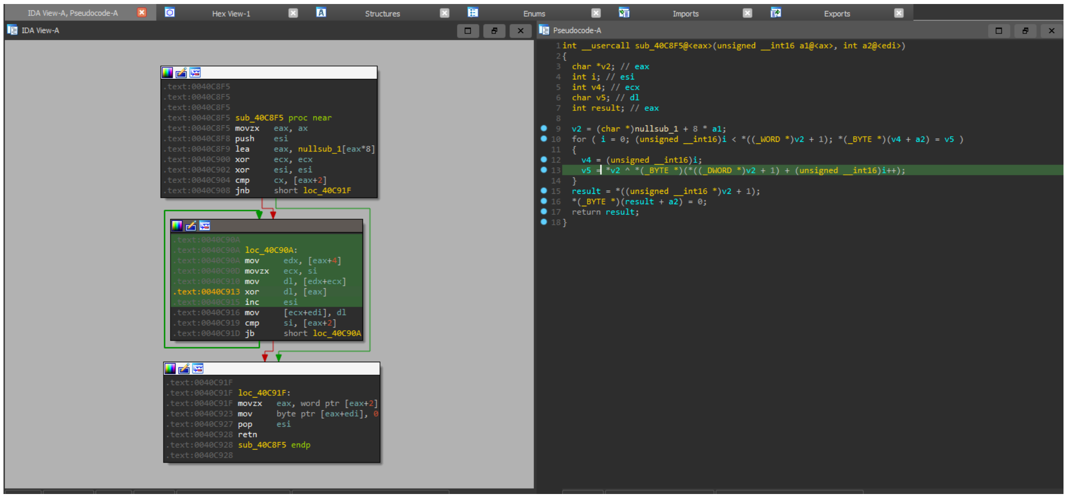
Move to the next function sub_4058FB. This function is the most interesting where string decryption and API resolving happens.



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.

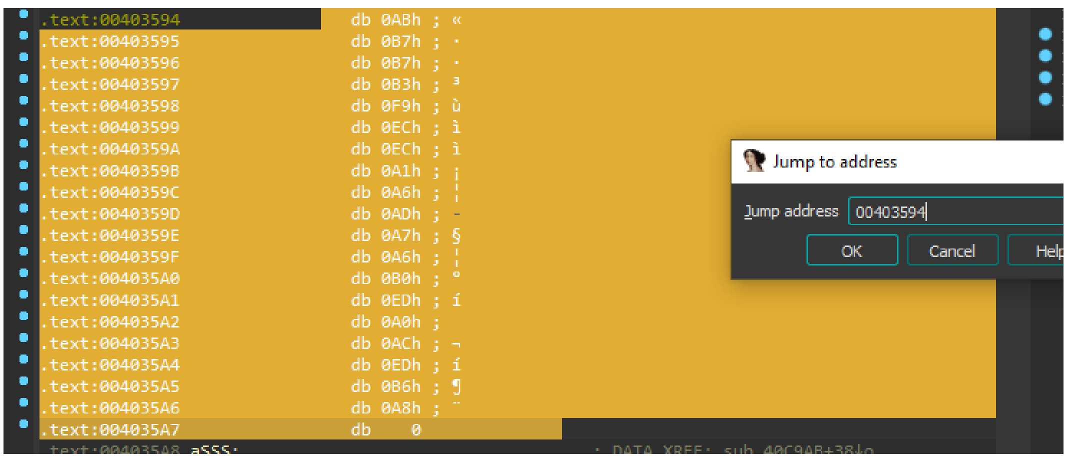
From the decompiler view we can see that the second argument is actually pointer to BYTE. If we set the types and names of variables correctly we can see better but not the best results.

```
int __usercall String_Decrypt1@<eax>(unsigned __int8 INDEX@<al>, BYTE *RETVAL@<edi>)
{
  char *v2; // eax
  int i; // esi
  int v4; // ecx
  BYTE v5; // dl
  int result; // eax

  v2 = nullsub_1 + 8 * INDEX;
  for ( i = 0; i < *(v2 + 1); RETVAL[v4] = v5 )
  {
    v4 = i;
    v5 = *v2 ^ *(*(v2 + 1) + i++);
  }
  result = *(v2 + 1);
  RETVAL[result] = 0;
  return result;
}
```

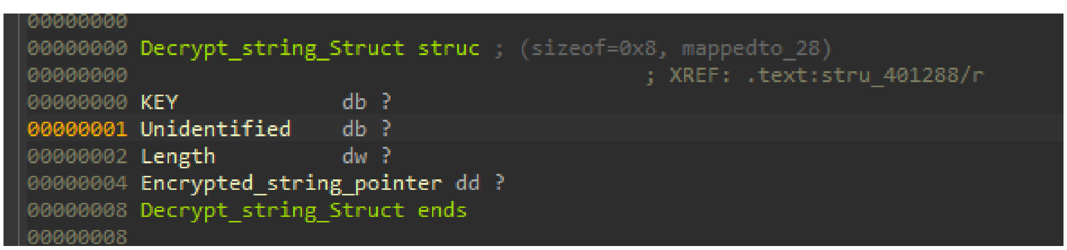
For better results, we must check also the nullsub_1 which is not a function but address to array of structures. Let’s undefine the nullsub_1 firstly.

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).



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



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.

```
int __usercall String_Decrypt1@<eax>(unsigned __int8 INDEX@<al>, BYTE *RETVAL@<edi>)
{
  Decrypt_string_Struct *str_current; // eax
  int i; // esi
  int v4; // ecx
  BYTE v5; // dl
  int result; // eax

  str_current = &stru_401288[INDEX];
  for ( i = 0; i < str_current->Length; RETVAL[v4] = v5 )
  {
    v4 = i;
    v5 = str_current->KEY ^ *(str_current->Encrypted_string_pointer + i++);
  }
  result = str_current->Length;
  RETVAL[result] = 0;
  return result;
}
```

AFTER

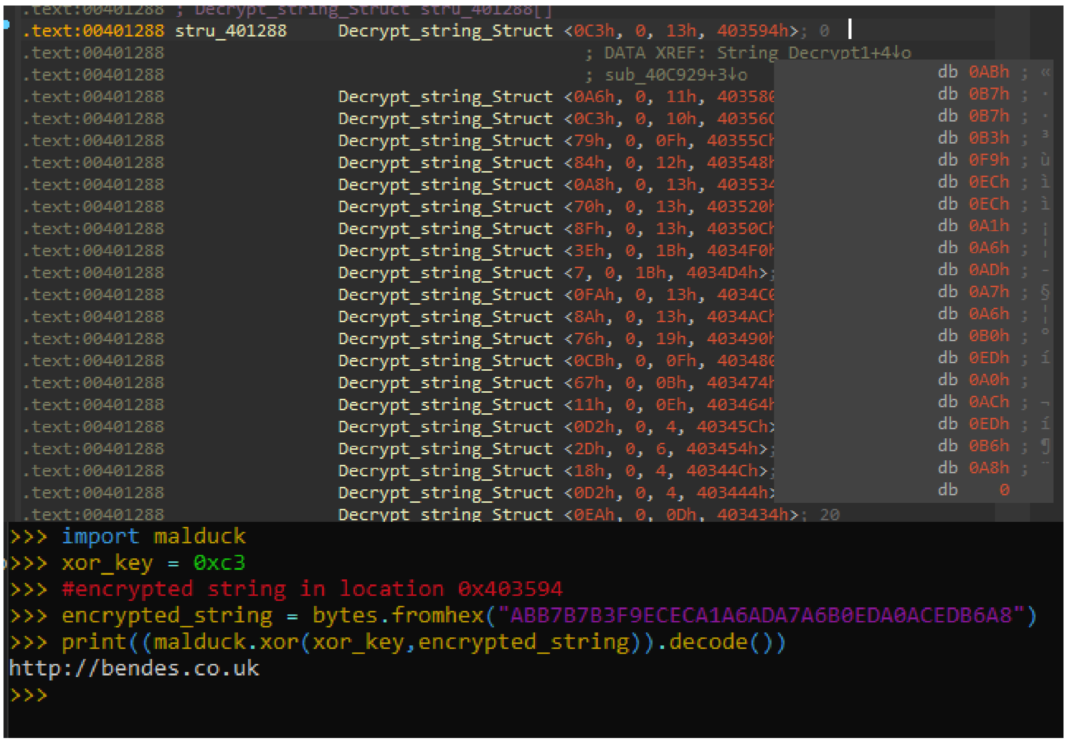
```
int __usercall sub_40C8F5@<eax>(unsigned __int16 a1@<ax>, int a2@<edi>)
{
  char *v2; // eax
  int i; // esi
  int v4; // ecx
  char v5; // dl
  int result; // eax

  v2 = nullsub_1 + 8 * a1;
  for ( i = 0; i < *(v2 + 1); *(v4 + a2) = v5 )
  {
    v4 = i;
    v5 = *v2 ^ *(*(v2 + 1) + i++);
  }
  result = *(v2 + 1);
  *(result + a2) = 0;
  return result;
}
```

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.

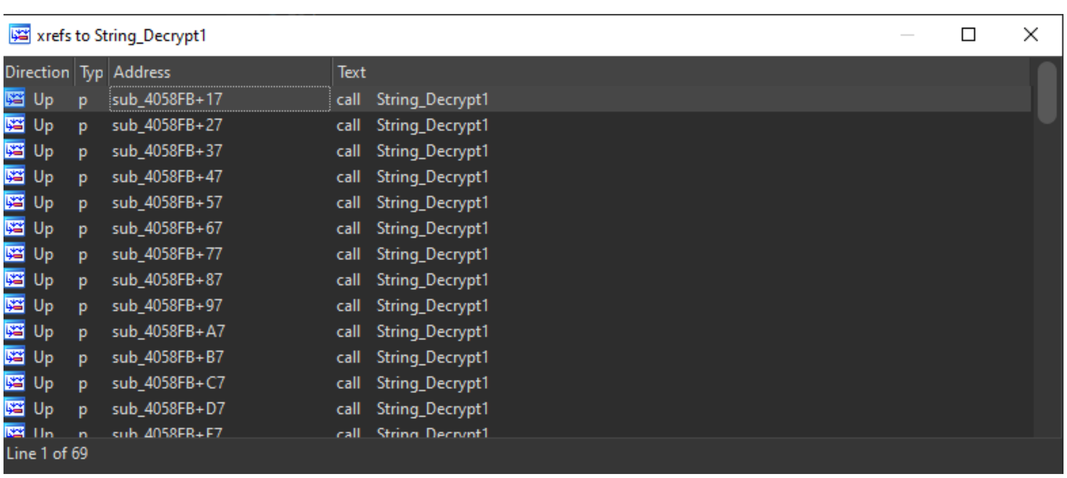
Key (BYTE) from the structure is xored with each BYTE in the location (Encrypted_string_pointer) from our indexed structure, till it reaches the length of encrypted string.

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

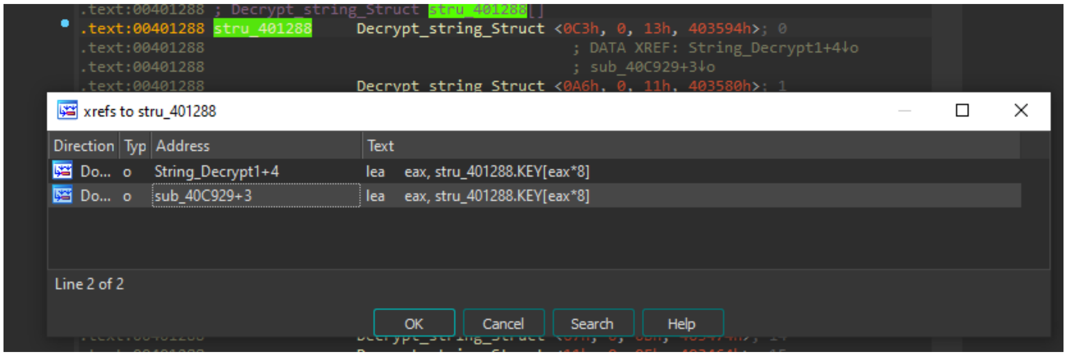


We were correct and obtained our first IOC.

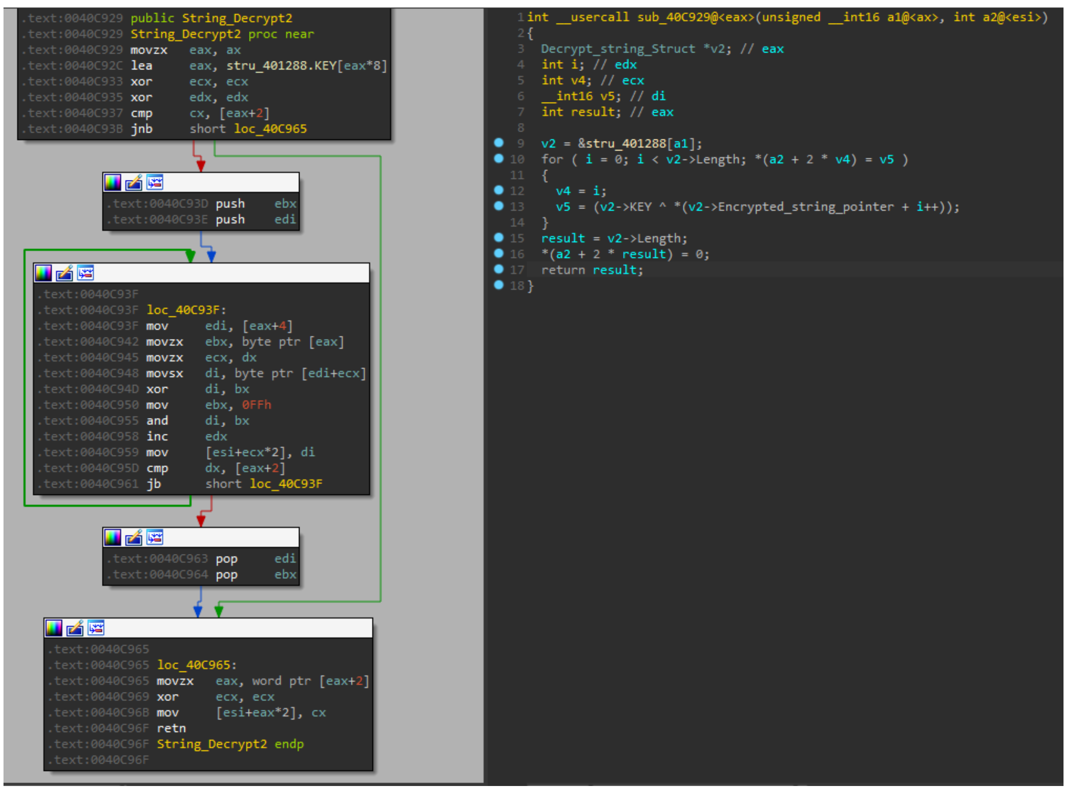
Before jumping to IDAPython we forgot something. If you remember the function String_Decrypt1 was referenced from 69 locations but our array of structures contains 183 members.



So we could check Xreferences to our array of structures if we could find another String_DecryptX function.



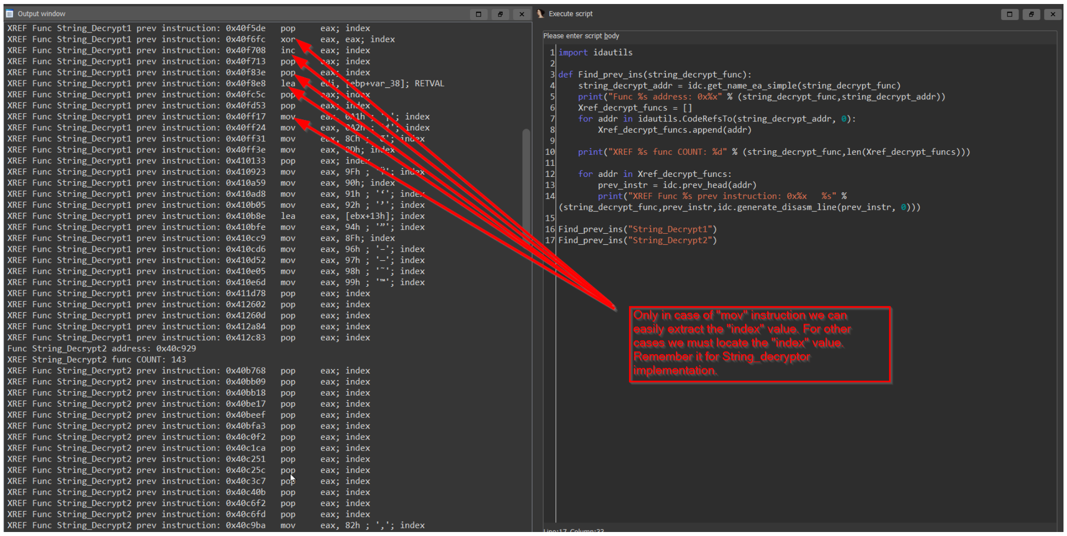
We were right, there is another one. Quick checking that function (sub_40C929) revealed that it is basically the same as function String_Decrypt1. So we rename it to String_Decrypt2.



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

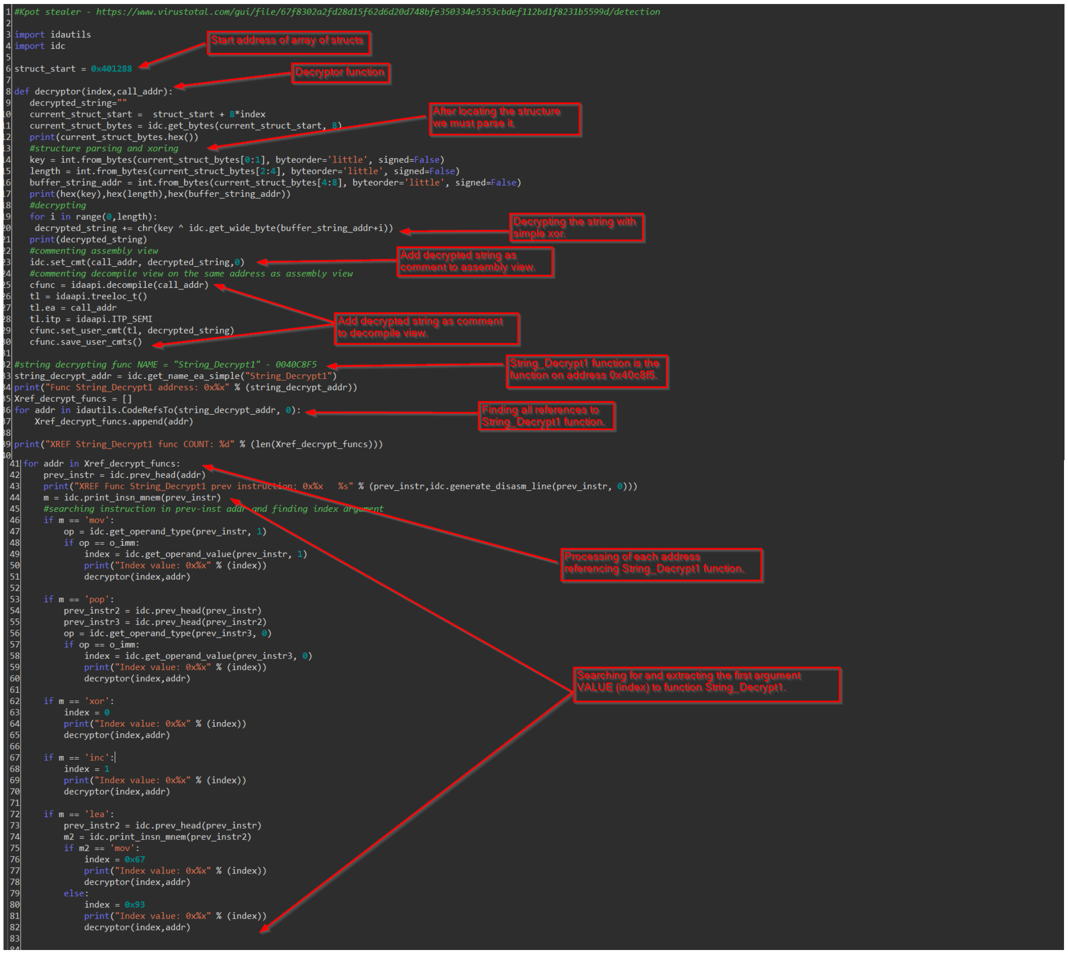
The final decryptor could be something, what will find all location from where our 2 string-decrypting functions (String_Decrypt1, String_Decrypt2) are called. After it finds these locations it will grab the first argument as our "INDEX" to structure, find and parse the structure[index]. This will serve 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].

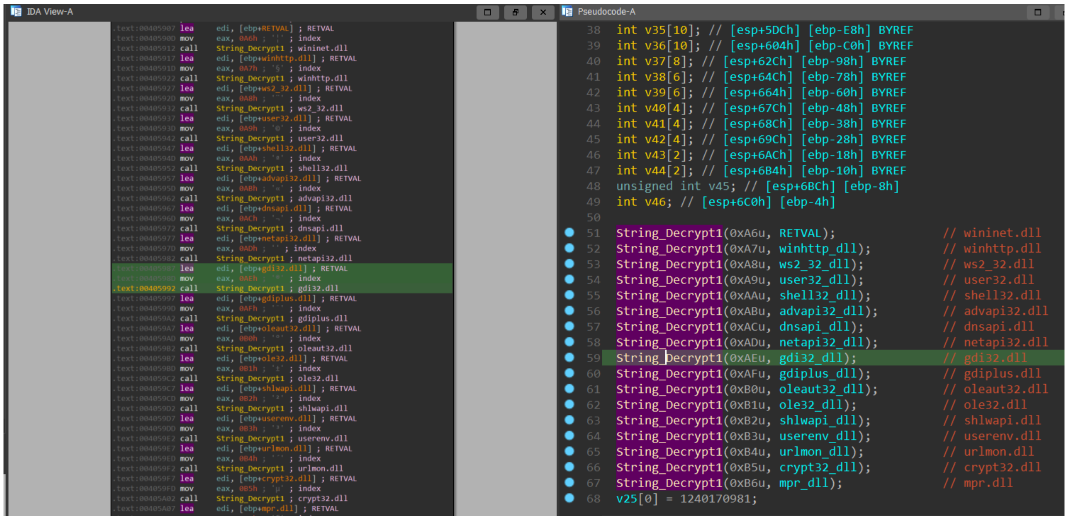
We must deal with locating the first argument during the string-decryptor implementation. In the picture below is the string-decryptor script in IDAPython for the “String_Decrypt1” function.



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.

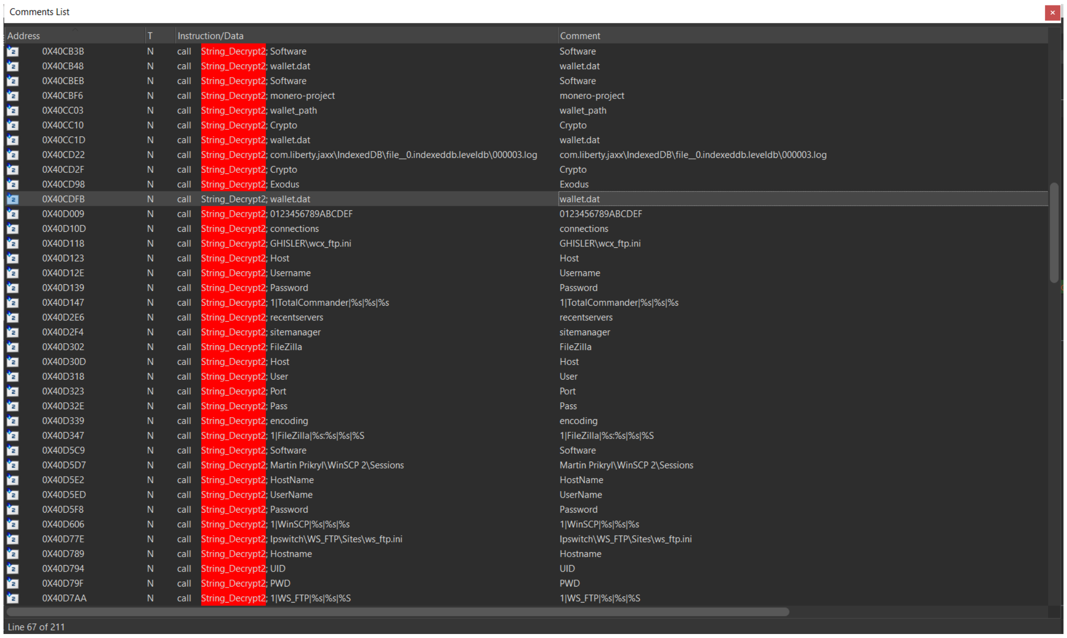


In Output window we could see some information like: String_Decrypt1 function address, count of references and for each processed reference is shown - current index value, current structure in hex, current xor KEY, length of encrypted string, address where the encrypted string is located and finally decrypted string.

Output window

```
Func String_Decrypt1 address: 0x40c8f5
XREF String_Decrypt1 func COUNT: 69
XREF Func String_Decrypt1 prev instruction: 0x40590d   mov     eax, 0A6h ; '¦'; index
Index value: 0xa6
b4000b005c2a4000
0xb4 0xb 0x402a5c
wininet.dll
XREF Func String_Decrypt1 prev instruction: 0x40591d   mov     eax, 0A7h ; '§'; index
Index value: 0xa7
9f000b00502a4000
0x9f 0xb 0x402a50
winhttp.dll
```

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…



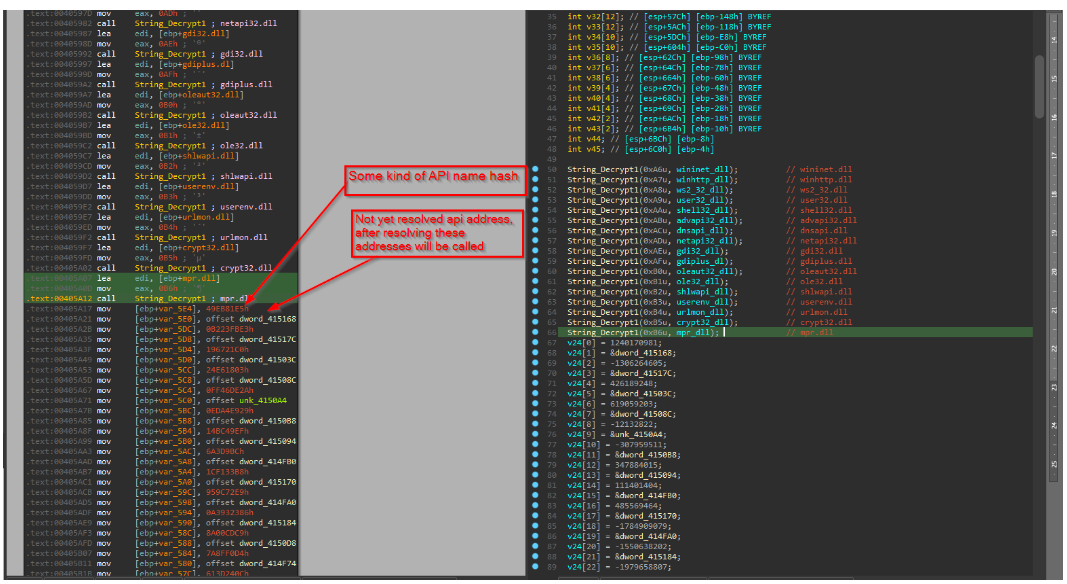
Comments List

Address	T	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.liberty.jaxx\IndexedDB\file__0.indexeddb.leveldb\000003.log	com.liberty.jaxx\IndexedDB\file__0.indexeddb.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\wcx_ftp.ini	GHISLER\wcx_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 Prikryl\WinSCP 2\Sessions	Martin Prikryl\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\ws_ftp.ini	Ipswitch\WS_FTP\Sites\ws_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

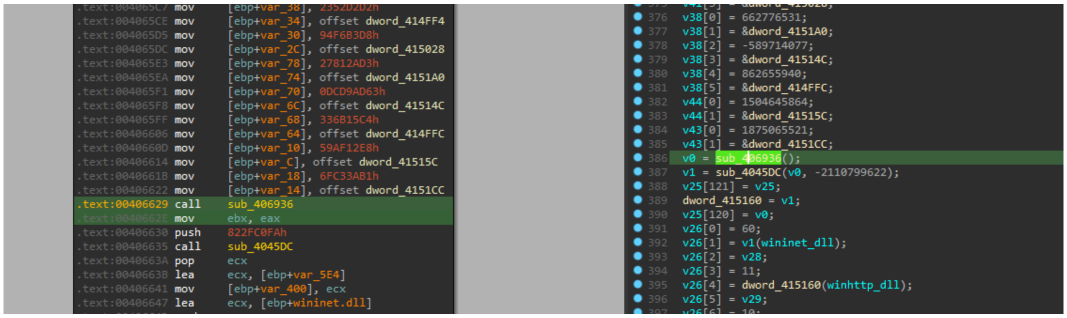
Line 67 of 211

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

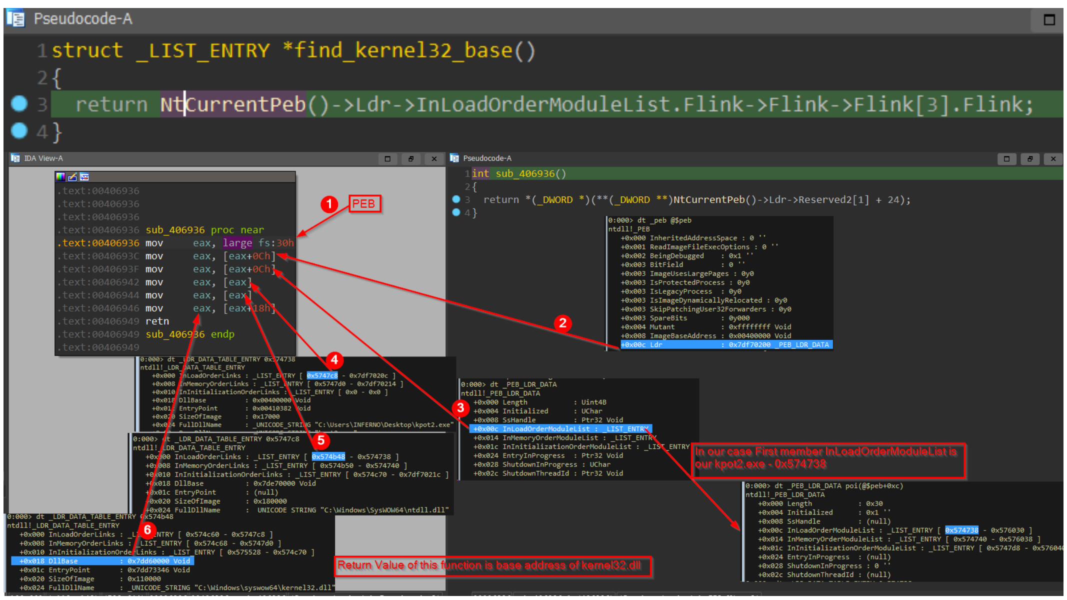
We will continue with our string-decrypting and API resolving function sub_4058FB to see what is going on next. We can see that there will be probably some kind of API name hashing which after matching hash of API name, the address of the API function will be saved to the hardcoded memory location. In the picture below we can see the stack preparation for the API name hashing and resolving.



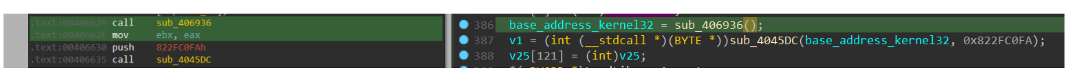
After the stack is prepared two functions get called. Let's check the first function sub_406936.



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.



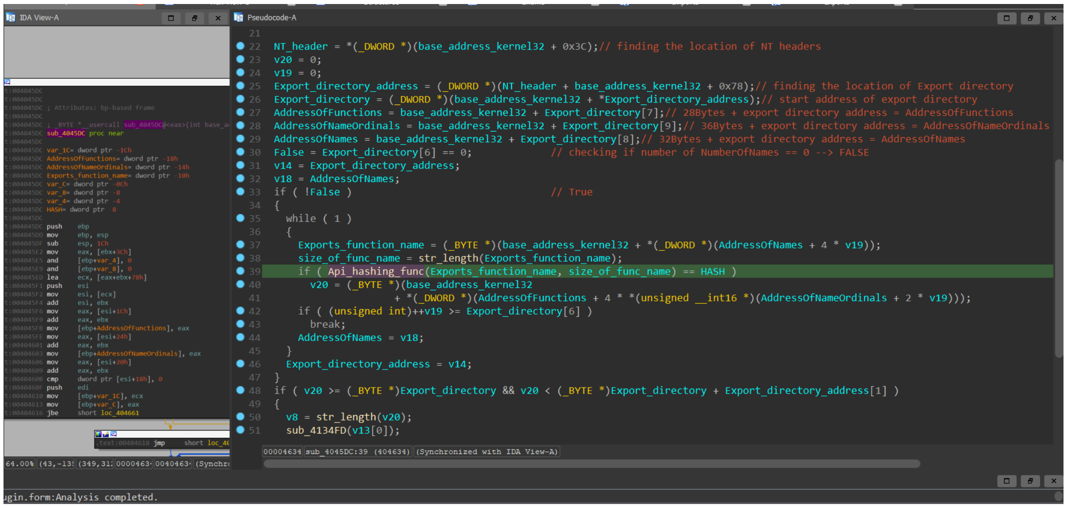
We can move to the next function sub_4045DC which is responsible for finding address of LoadLibraryA API function from kernel32.dll module.



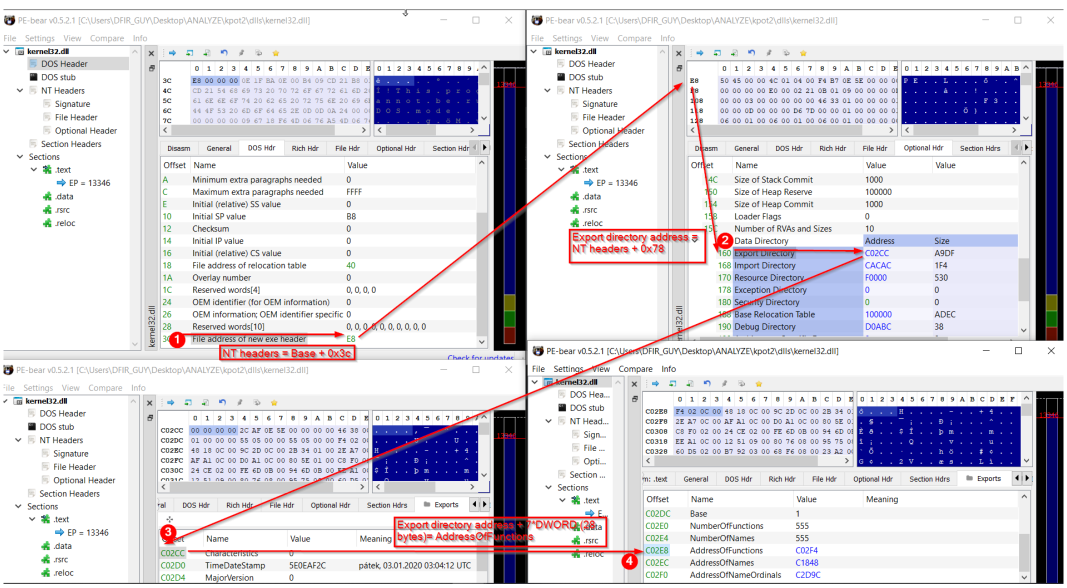
This function (sub_4045DC) is not responsible only for finding address of LoadLibraryA but it is able to find API address via hash value of its name and base address of module as arguments.

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.



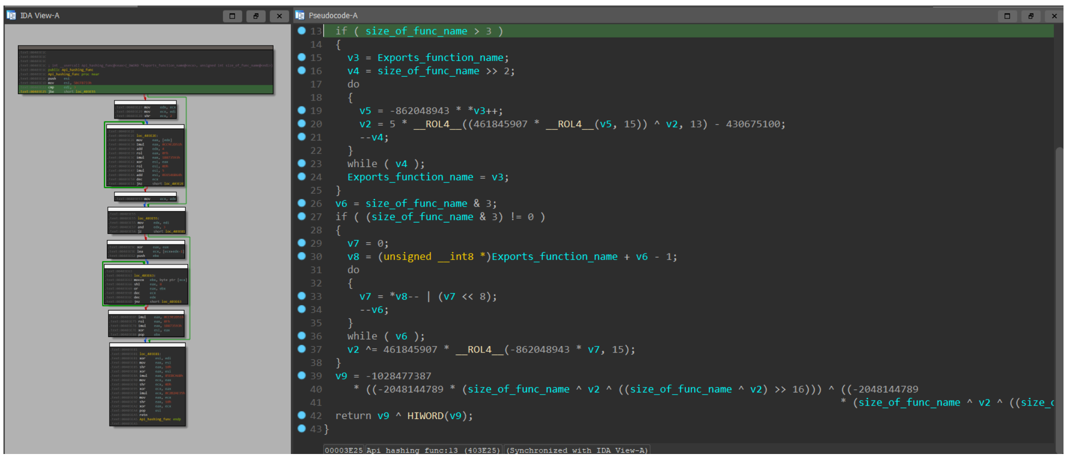
We can focus more on the function Api_hashing_func later.



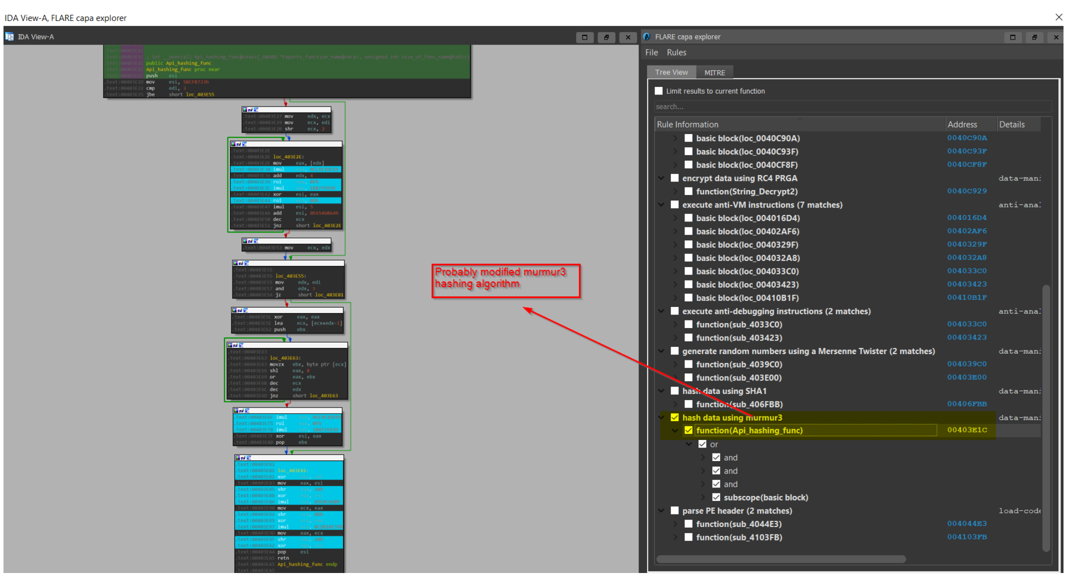
Of course we can save some time and let IDA help you with defaultly defined structs for PE. But I personally think that it is a needed skill to understand and be able to parse PE manually.

```
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 )
```

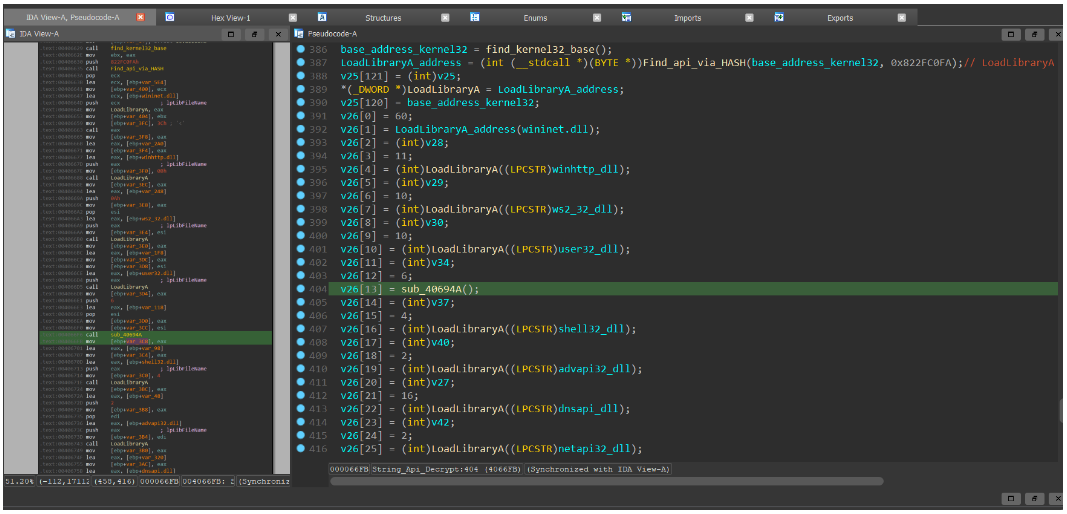
So let´s jump to the function Api_hashing_func (0x403E1C) which you could see in the picture below is implementing some probably modified version of well-known hashing algorithm.



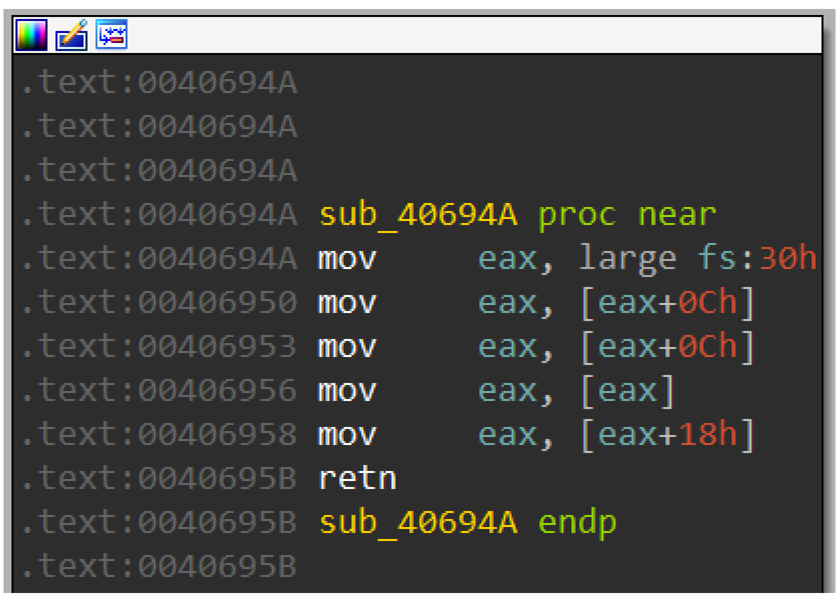
We could use a little help to find out what hash algorithm is implemented from another excellent tool Capa [https://github.com/fireeye/capa]. This gives us a hint that it could be hashing algorithm of type murmur3. We will come back to this hashing algorithm later.



So for now, we have more information and can come back and continue with function sub_4058FB - picture below which I populated with all known info. You can see that some another dlls are loaded and also another function sub_40694A is called.

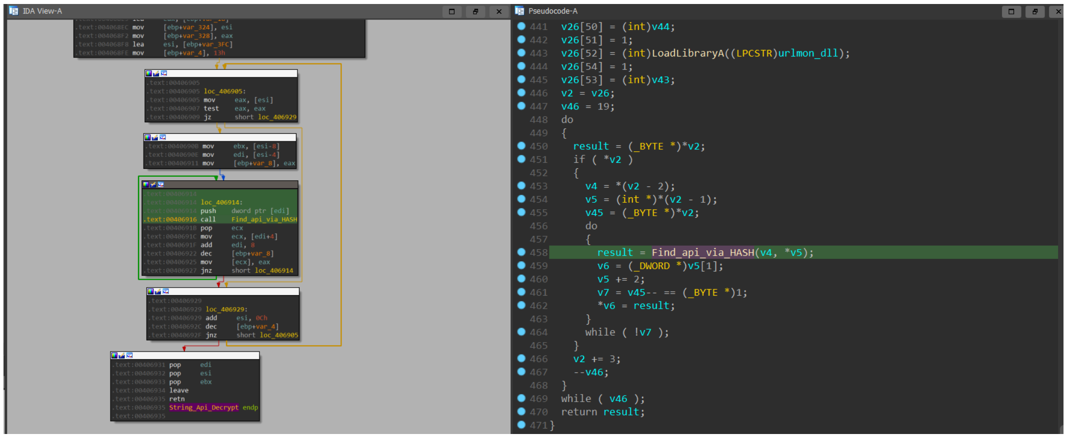


Function sub_40694A is parsing PEB where it returns ntdll.dll base address.



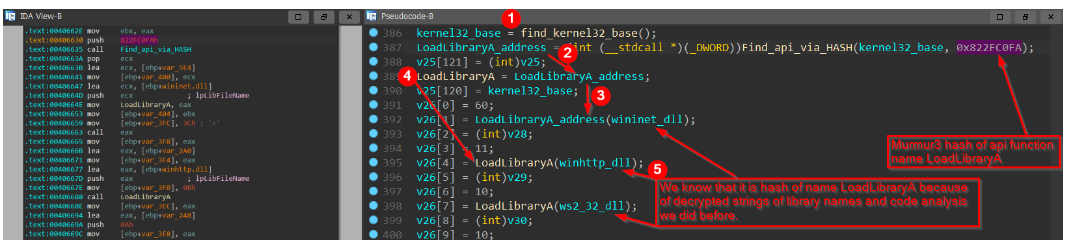
So we can continue and finally reach the interesting part.

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.

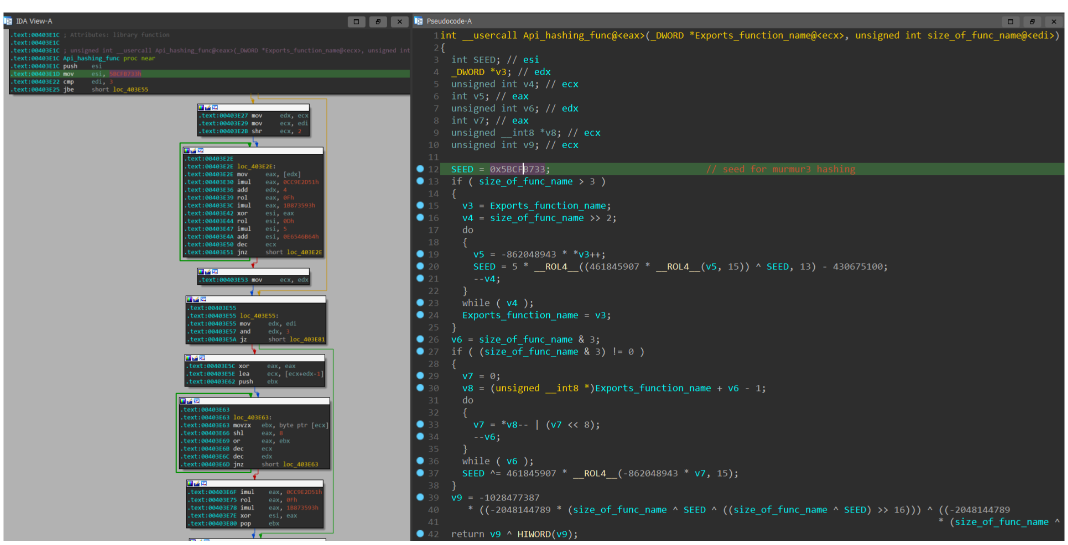


So now we have more options to obtain and populate all resolved API functions in our code. One of the option is to implement murmur3 hashing algorithm and with help of IDAPython, find all API function name hashes to process it with our algorithm. As we did some IDAPython scripting before and I want to 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'
>>>
>>>
```

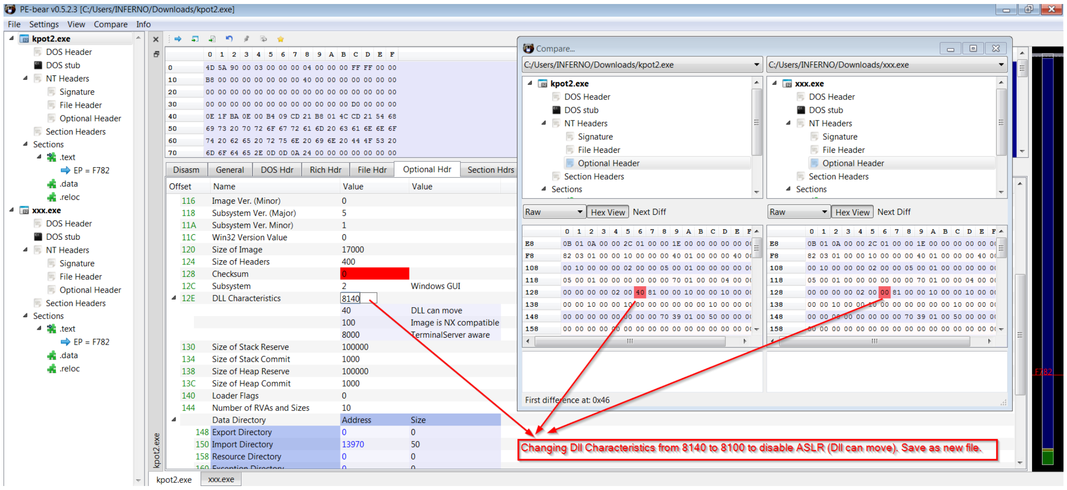
Our assumption about hashing algorithm is right so move next.

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.

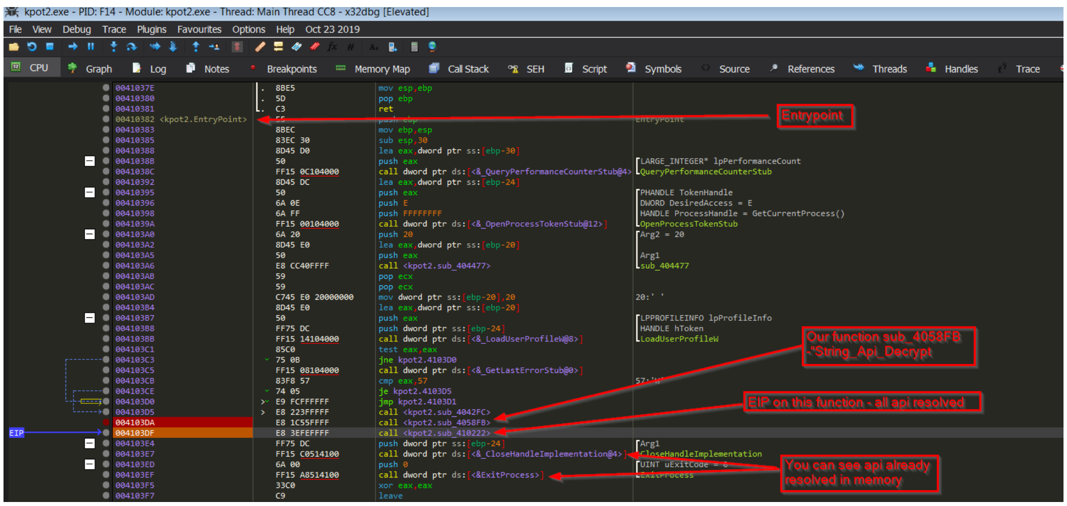
When we have all information about how the API resolving works, we could let the sample populate the resolved API function addresses in debugger, dump the process from memory and after that, we need something what is able to find in our dumped memory all populated API function addresses and annotate 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…

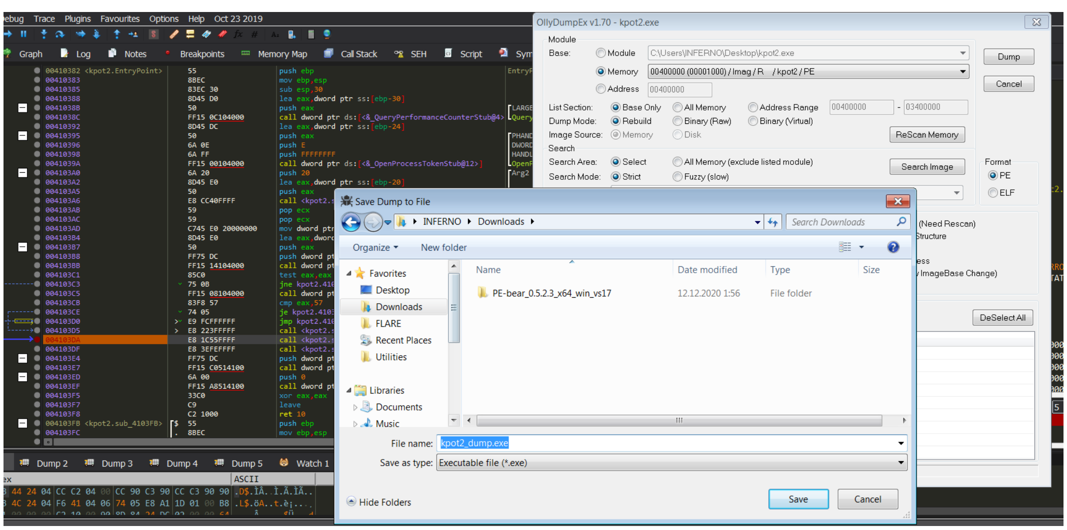
Let's start with dumping our kpot2.exe process from memory in debugger like x64dbg after it populates the resolved API function addresses. We put a breakpoint after the call sub_4058FB - "String_Api_Decrypt" and dump the process. To find location of this function in debugger easily, do not forget to disable ASLR in the optional header of kpot2.exe.



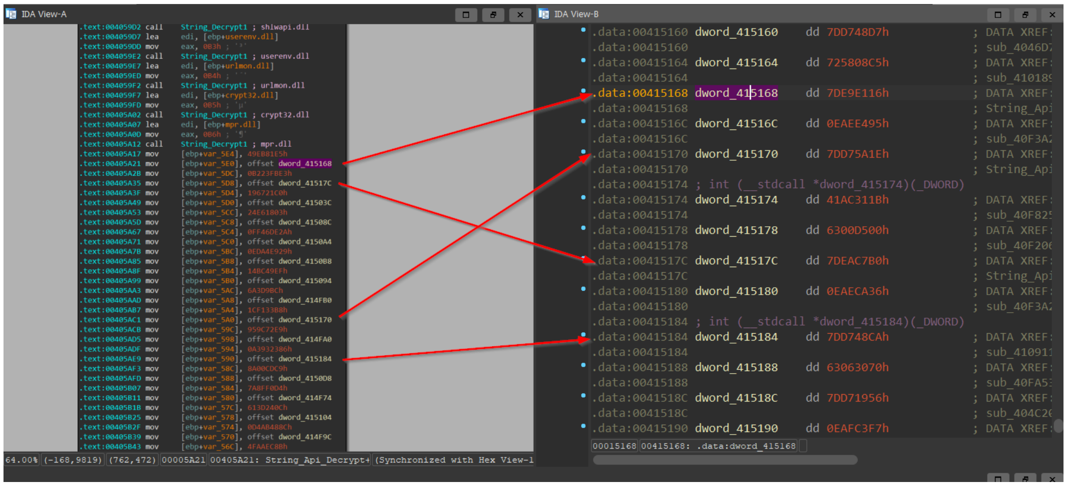
Locating our sub_4058FB - “String_Api_Decrypt function.



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:

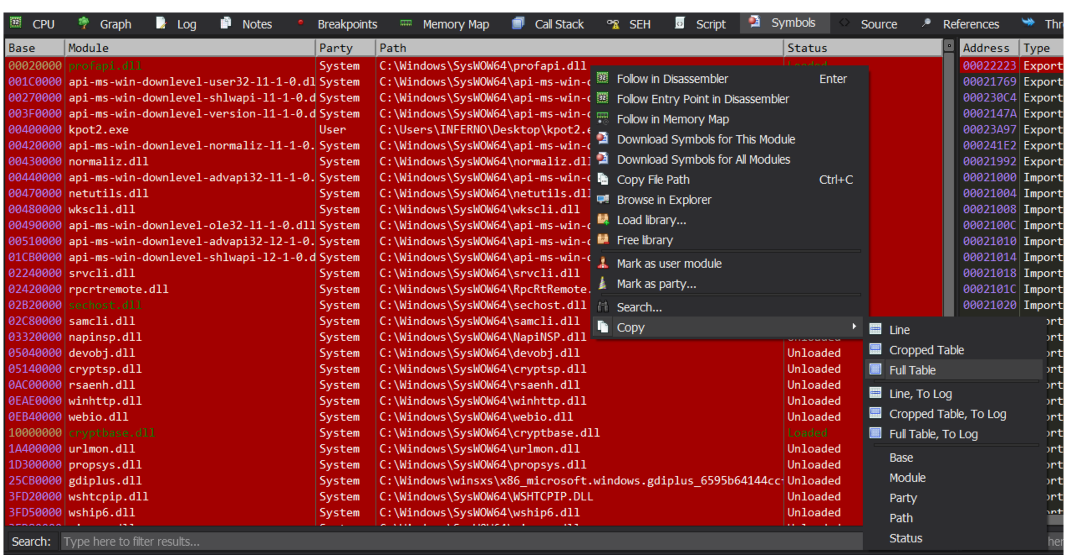
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management]

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

Restart system.

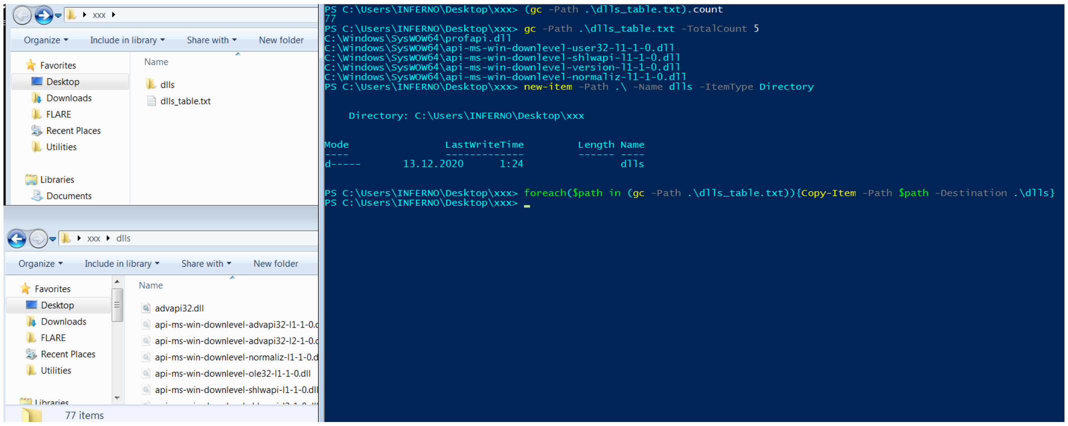
If we do not want to create database of all dlls from our system, first of all we should find and copy to some location all dlls which is our sample kpot2.exe loading and processing:

We can see this information in debugger from where we can copy the whole table to .txt file:

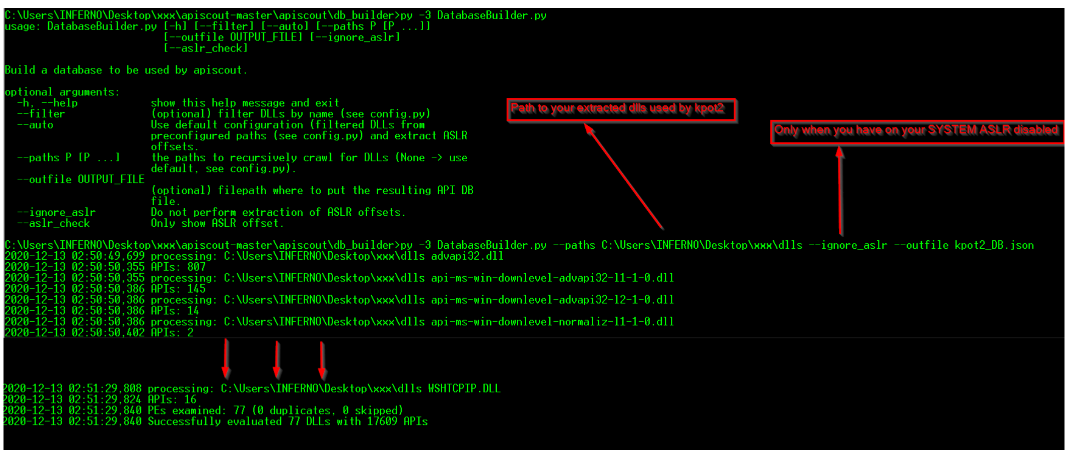


Extract dlls path with some regex, editors etc…

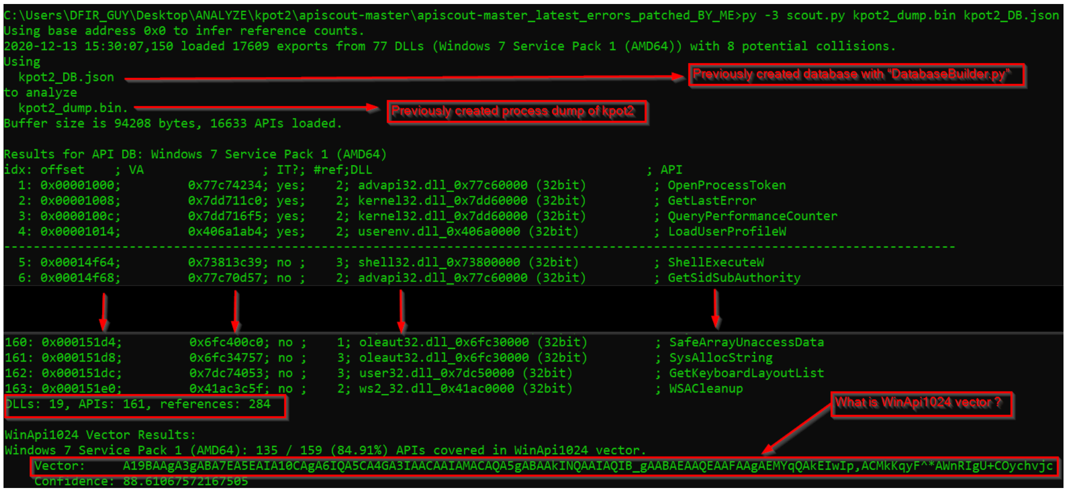
To copy all dlls from provided paths with powershell:



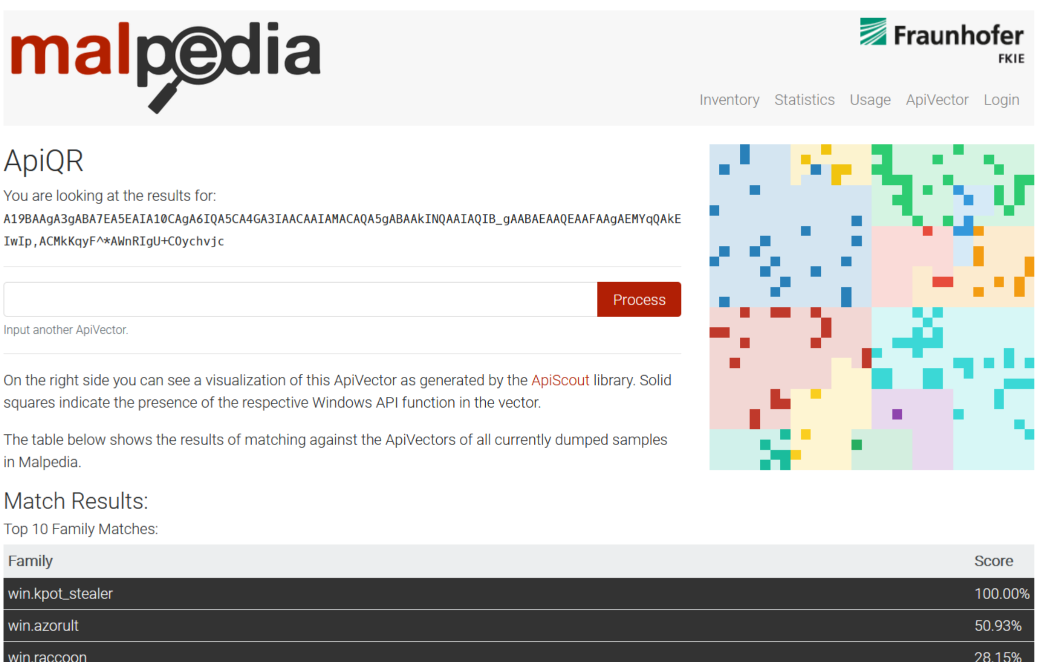
Now when we have all our needed dlls we start with apiscout – “DatabaseBuilder.py” to create our database.



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.

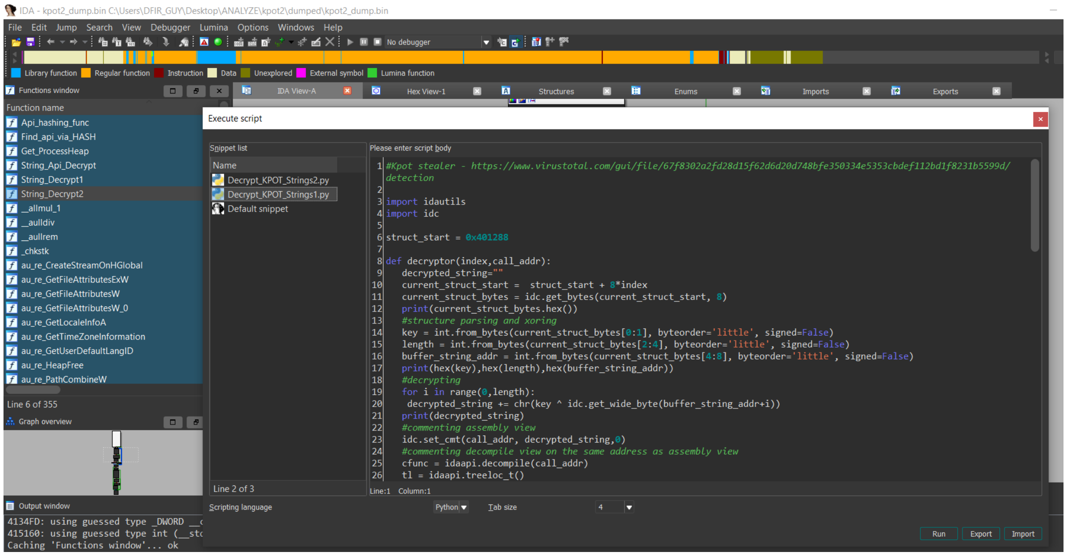


We can see that apiscout was successful and there is more – something called "WinApi1024 vector". Basically speaking it is something like ImpHash on steroids. You can read more about Apivector here: [https://byte-atlas.blogspot.com/2018/04/apivectors.html]. As we get WinApi1024 vector of our kpot2_dump.bin 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.



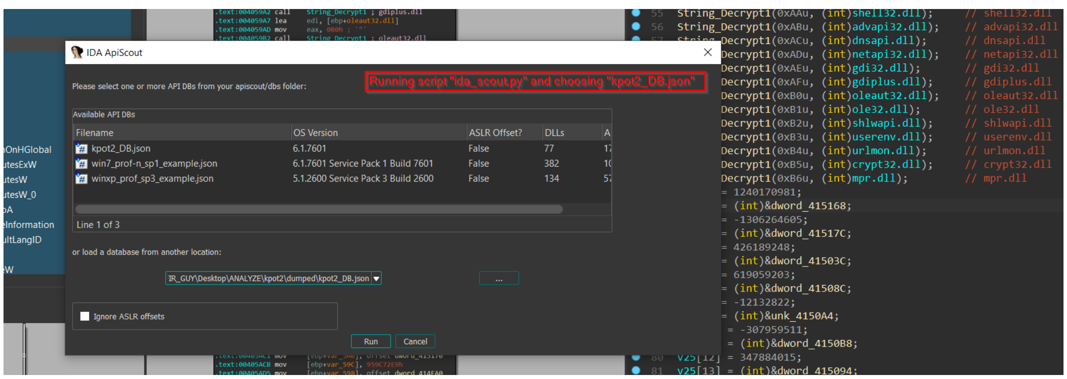
To apply all previously annotated names of functions from previous IDA database file to our newly created kpot2 process dump "kpot2_dump.bin", we could use IDA plugin called "rizzo" [https://github.com/tacnetsol/ida/tree/master/plugins/rizzo].

After that, previously created IDAPython scripts for decrypting strings must be run again (Decrypt_KPOT_Strings1.py, Decrypt_KPOT_Strings2.py) [View here]

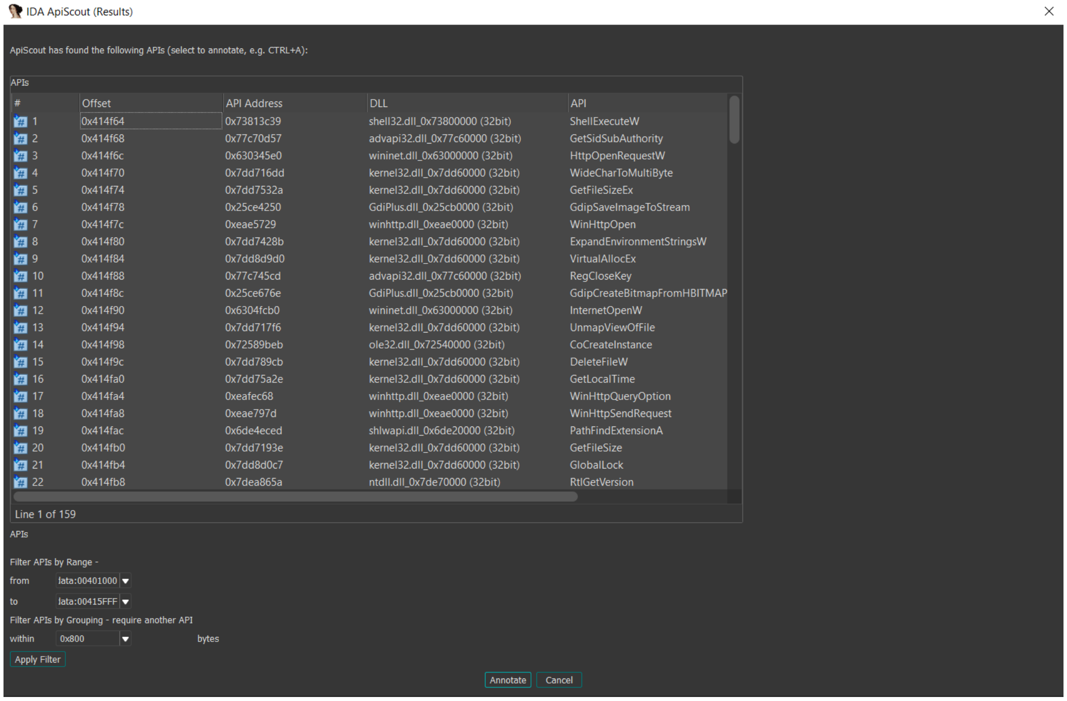


Now we are almost in the same state with “kpot2_dump.bin” as we were in the original sample.

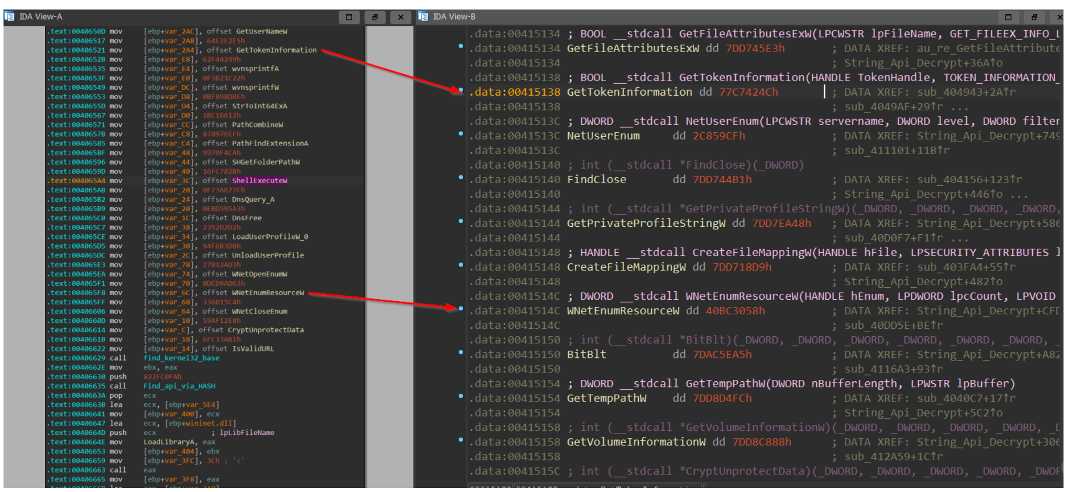
Let’s continue to apply our created database kpot2_DB.json to process dump kpot2_dump.bin in context of IDA. We will use apiscout IDAPython script “ida_scout.py” for that.

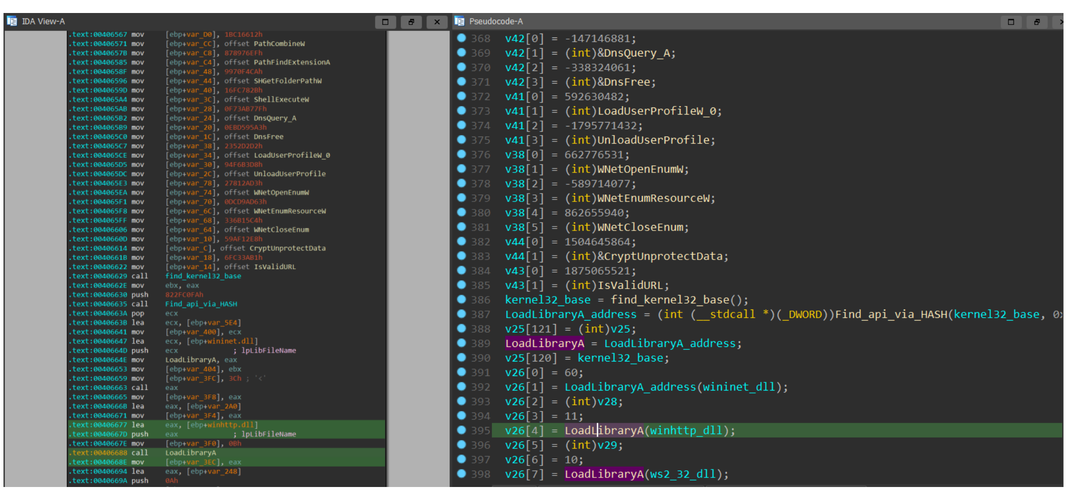


In the next window choose all of the found APIs and click “Annotate”.



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

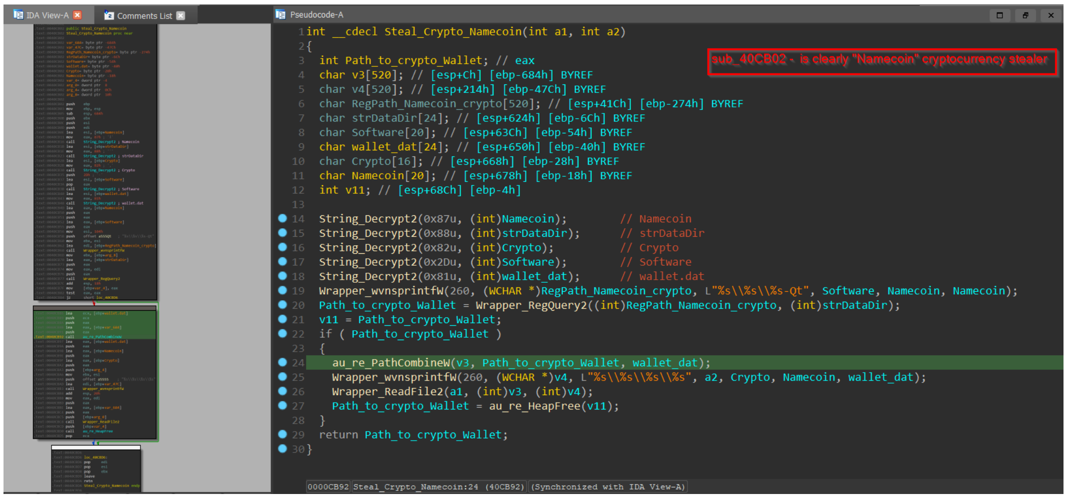




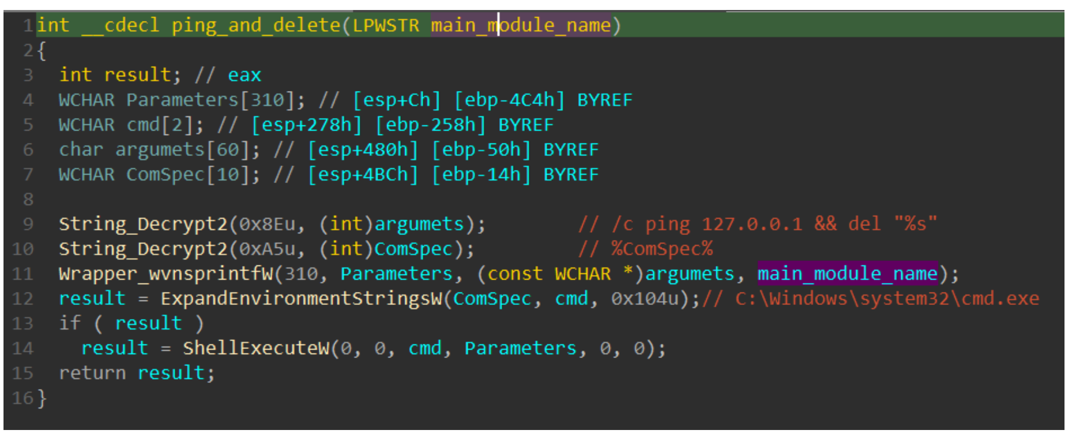
Now we are in state were we have all strings decrypted, all API function calls resolved and annotated so we are ready to benefit from it in analysis.

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

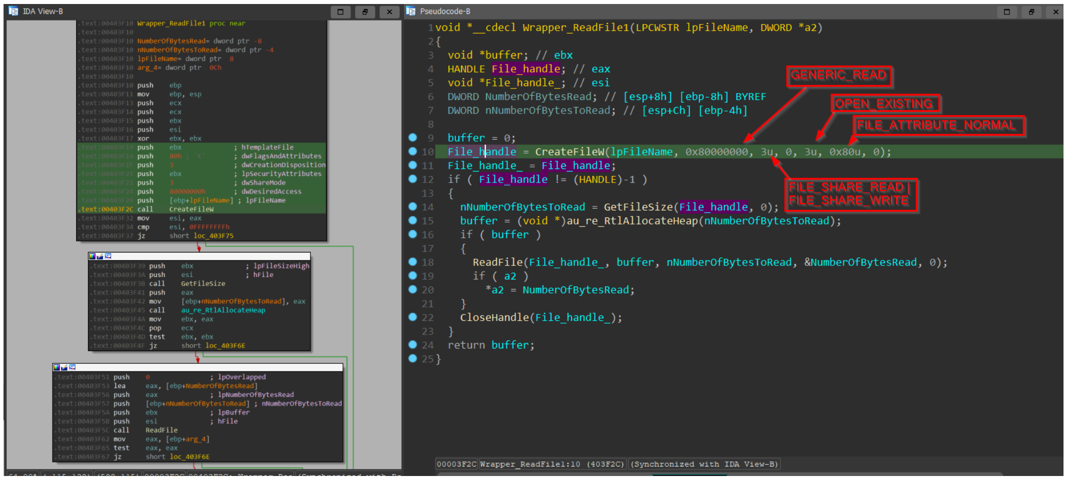
sub_40CB02 - is clearly "Namecoin" cryptocurrency stealer:



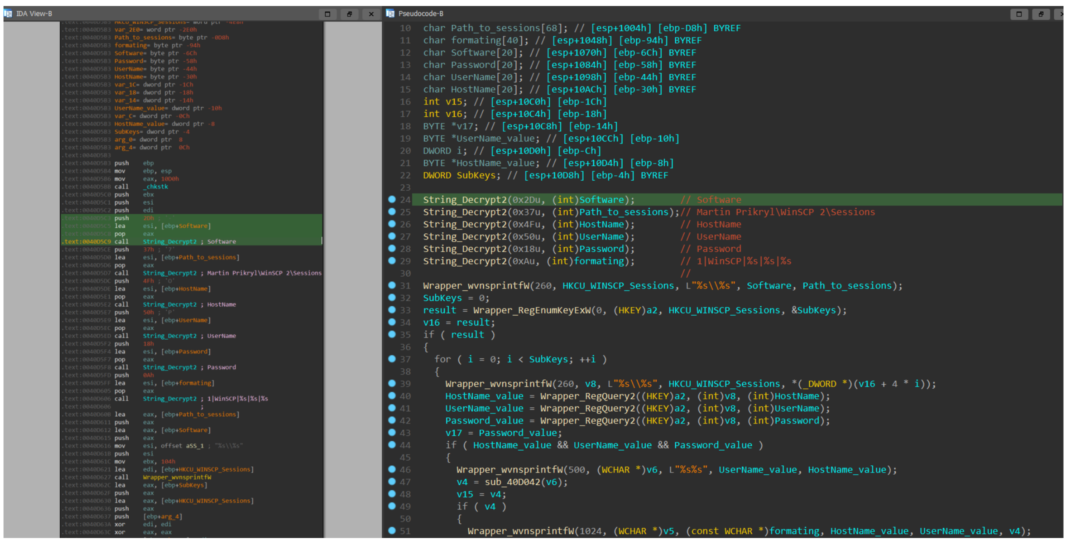
sub_4101AB – ping + delete main module (kpot2.exe) always called before exit().



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]