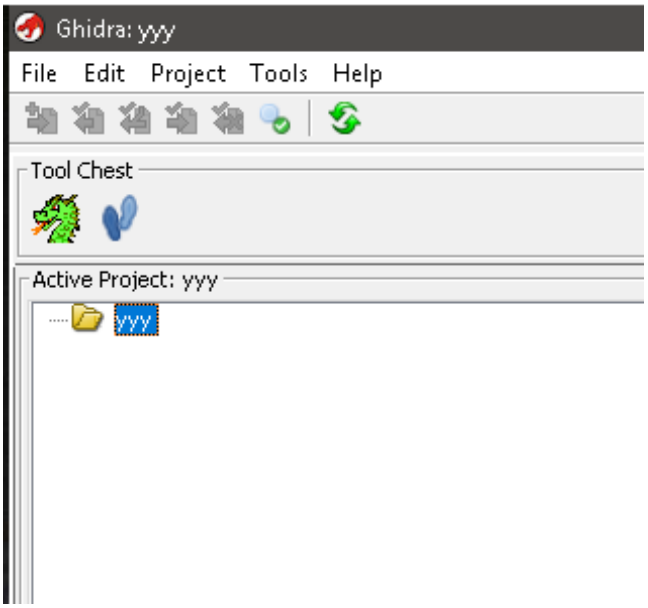
Quick Analysis of a Trickbot Sample with NSA's Ghidra SRE Framework

Archived: 2026-04-05 20:41:53 UTC

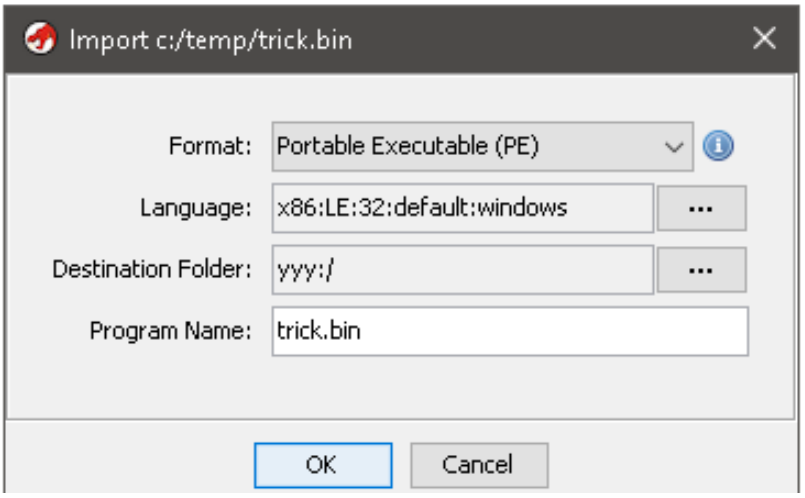
This post is not a deep analysis of TrickBot. Here, I did a quick analysis of a TrickBot sample from early 2019 by using the Ghidra Software Reverse Engineering (SRE) Framework, developed by the NSA, that was released some hours ago. This is not a deep analysis of TrickBot, I only wanted to learn a bit about Ghidra and I used this framework to find some interesting parts of the code of TrickBot that were introduced in the newer versions of the malware. Hope you enjoy it!

Starting with Ghidra Framework

About Ghidra, when you start the framework, you should create a project and a workspace:

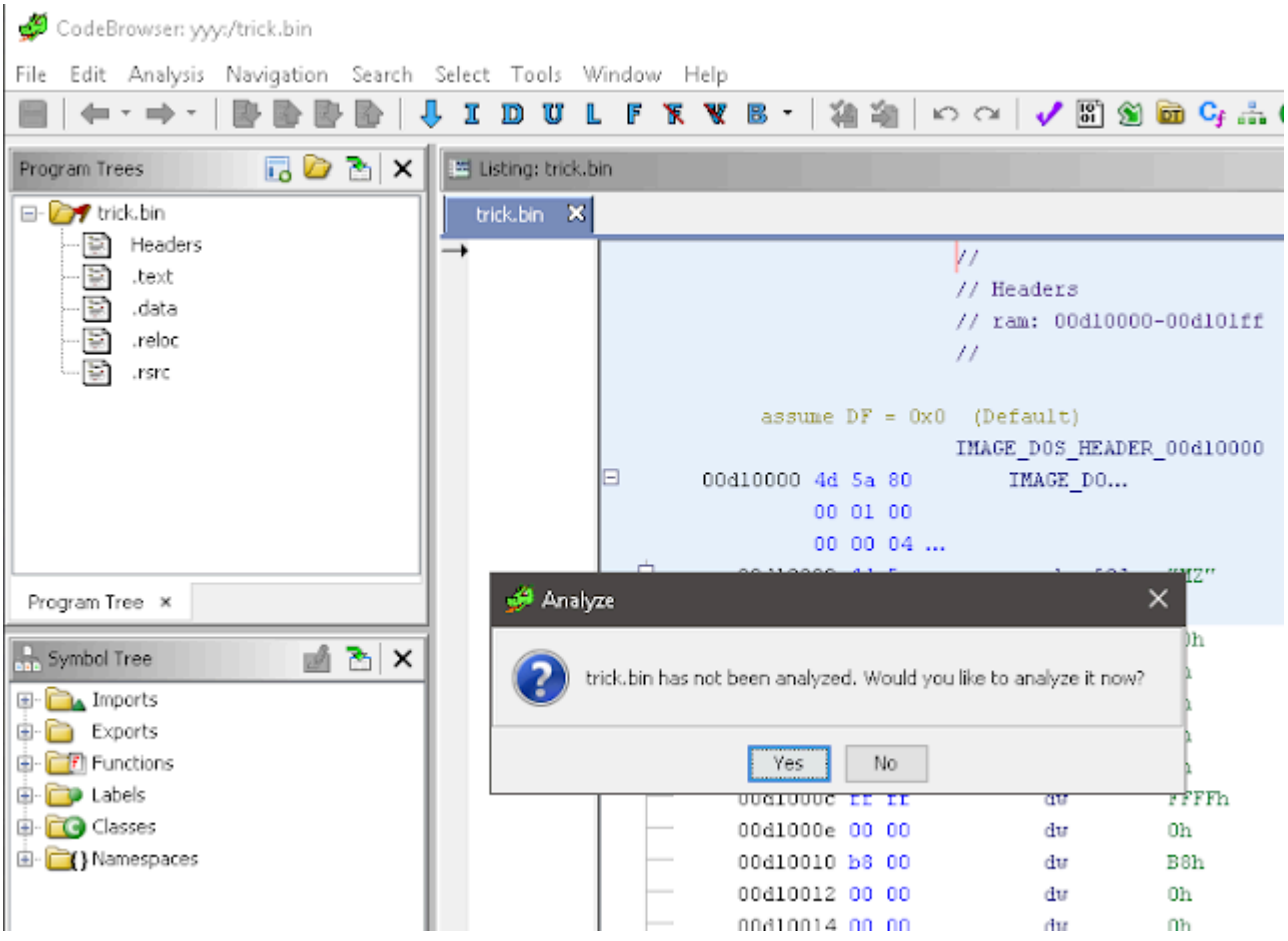


Then, we can import files, for example PE files:



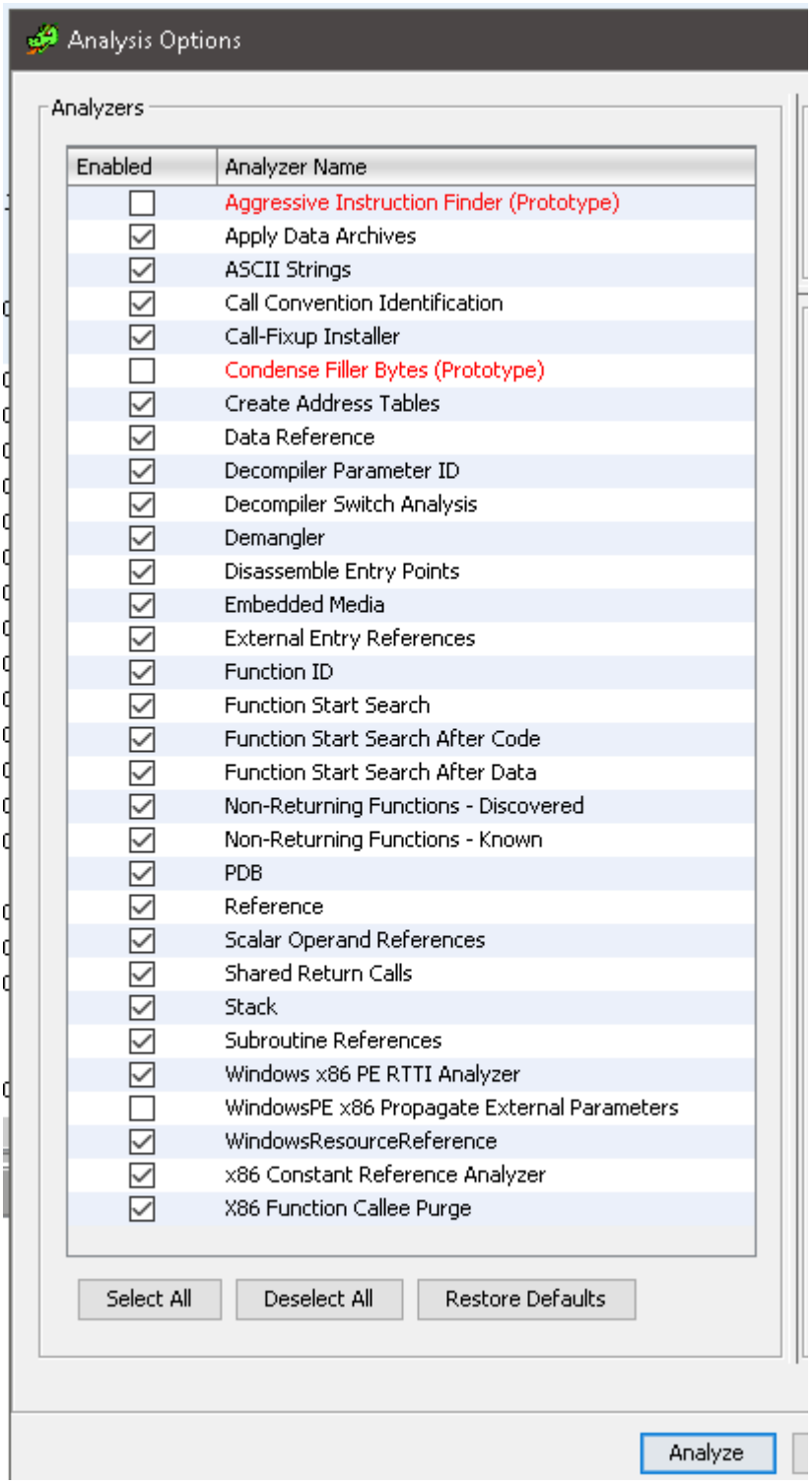
Ghidra CodeBrowser

Once PE file is imported, CodeBrowser can be launched:

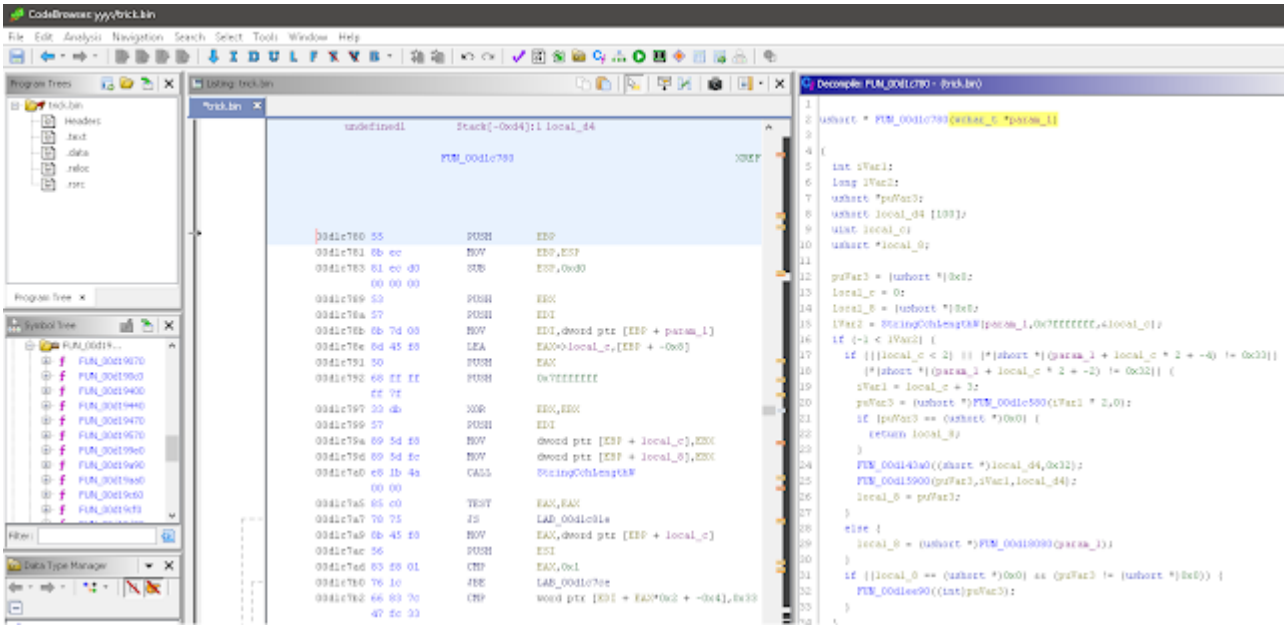


Initially, PE headers are parsed but code is not analyzed, the framework asks you if analyzers should be launched, and what analyzers should be launched. This is the list of analyzers (they are marked the analyzers that are marked

by default):



Once analyzers finish, CodeBrowser interface is like this:

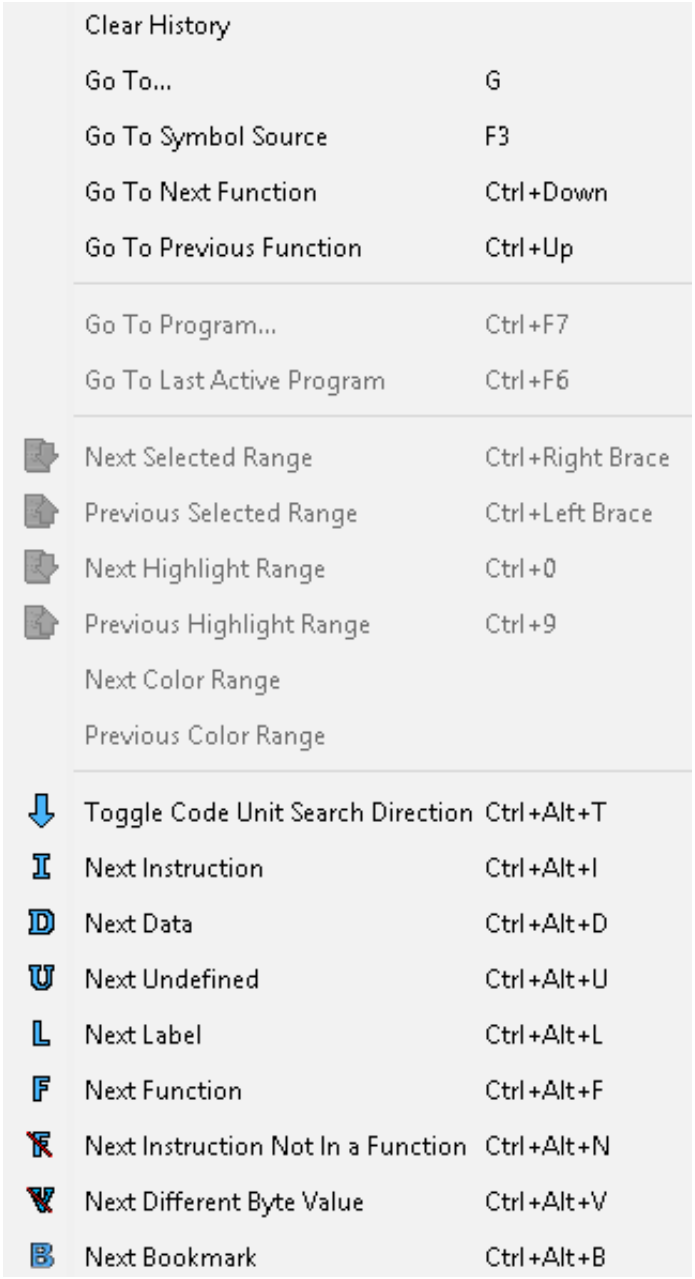


Code is fully decompiled and while you browse each function, the decompiled code is showed in the right window.

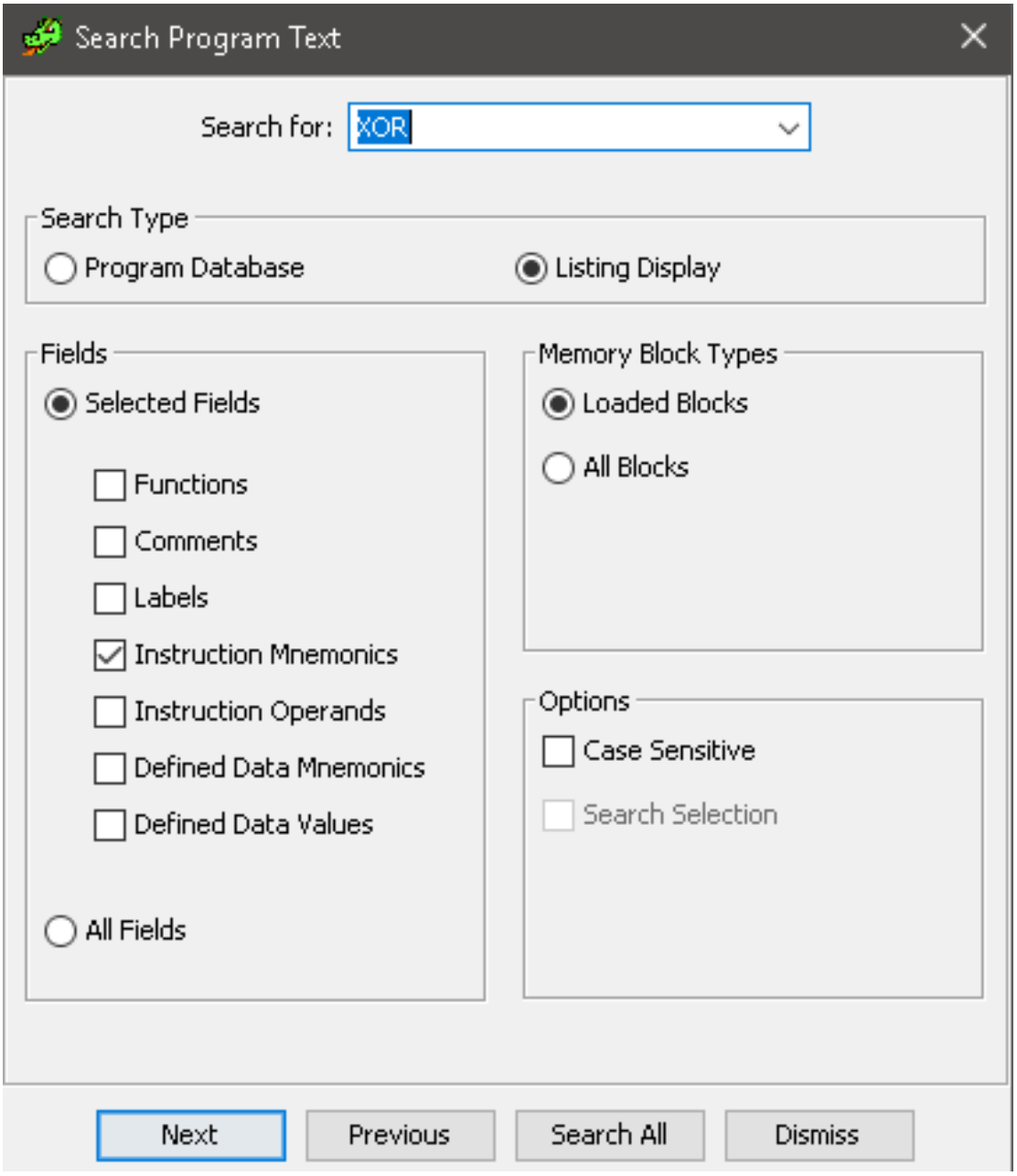
Browsing Code

Browsing code is similar to IDA, you can double-click a name to jump there (for example double-clicking the destination of a call <destination>, would take you to the destination function). You can move easily to the previous location with Alt+left (equivalent to Esc in IDA) and next location with Alt+right (equivalent to Ctrl+Enter in IDA).

Other navigation options:

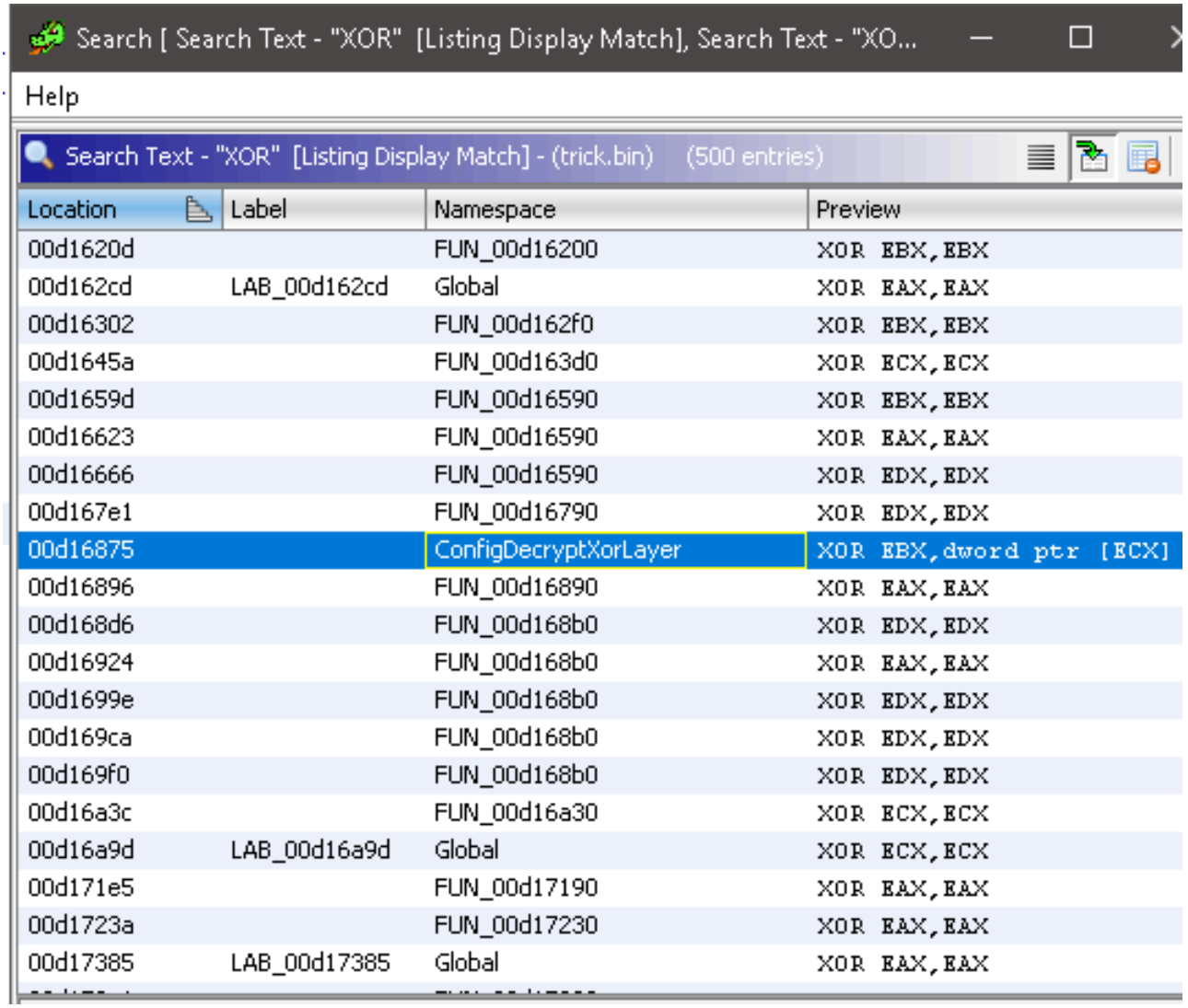


You can search for text, like IDA Alt+t, however (and I found this an interesting characteristic), you can select where do you want the text is going to be searched:



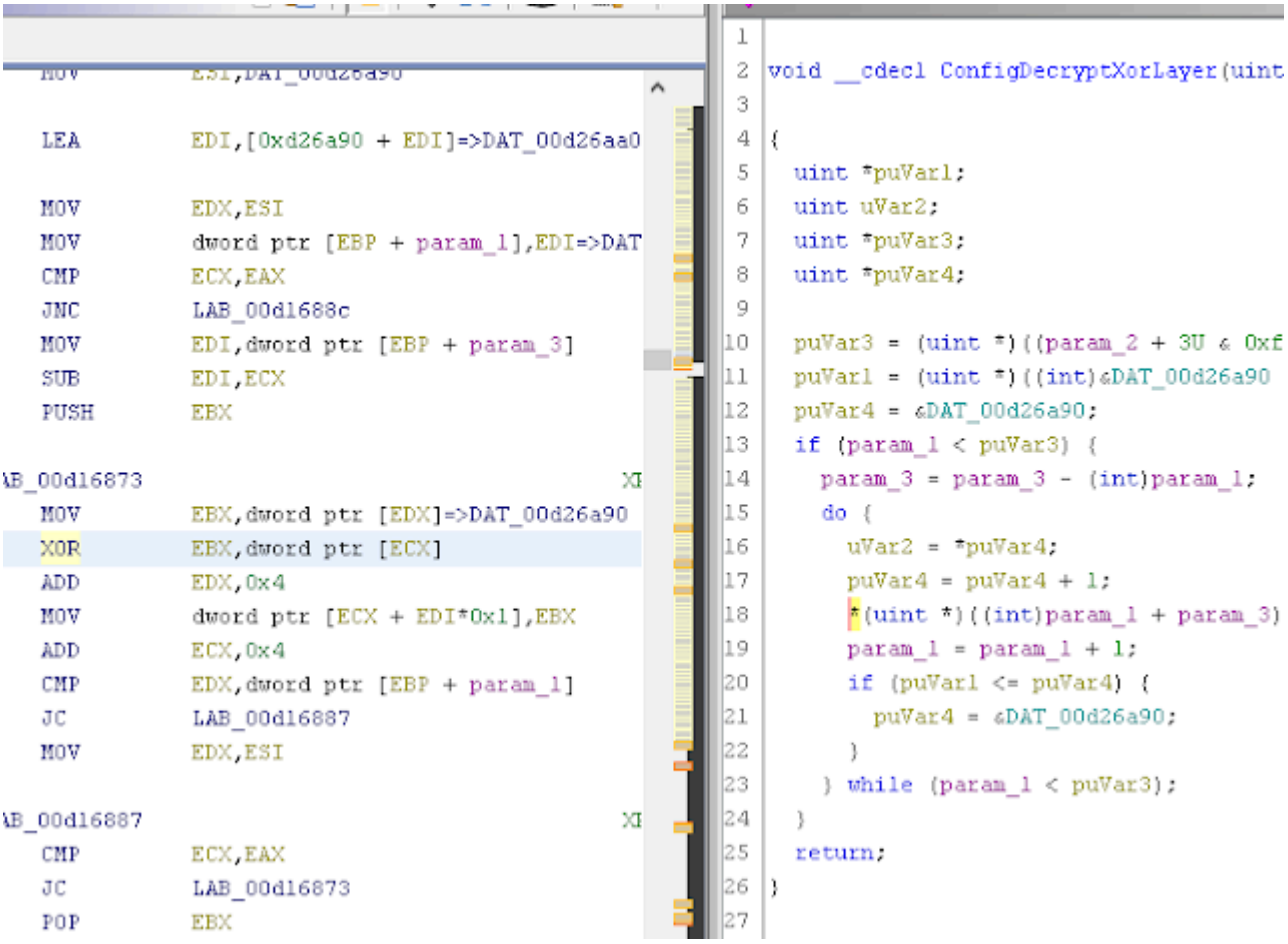
Find TrickBot Config Xor-layer Decryptor

For example, we can try to search for XOR instructions, and we get a list of matches:



Location	Label	Namespace	Preview
00d1620d		FUN_00d16200	XOR EBX,EBX
00d162cd	LAB_00d162cd	Global	XOR EAX,EAX
00d16302		FUN_00d162f0	XOR EBX,EBX
00d1645a		FUN_00d163d0	XOR ECX,ECX
00d1659d		FUN_00d16590	XOR EBX,EBX
00d16623		FUN_00d16590	XOR EAX,EAX
00d16666		FUN_00d16590	XOR EDX,EDX
00d167e1		FUN_00d16790	XOR EDX,EDX
00d16875		ConfigDecryptXorLayer	XOR EBX,dword ptr [ECX]
00d16896		FUN_00d16890	XOR EAX,EAX
00d168d6		FUN_00d168b0	XOR EDX,EDX
00d16924		FUN_00d168b0	XOR EAX,EAX
00d1699e		FUN_00d168b0	XOR EDX,EDX
00d169ca		FUN_00d168b0	XOR EDX,EDX
00d169f0		FUN_00d168b0	XOR EDX,EDX
00d16a3c		FUN_00d16a30	XOR ECX,ECX
00d16a9d	LAB_00d16a9d	Global	XOR ECX,ECX
00d171e5		FUN_00d17190	XOR EAX,EAX
00d1723a		FUN_00d17230	XOR EAX,EAX
00d17385	LAB_00d17385	Global	XOR EAX,EAX

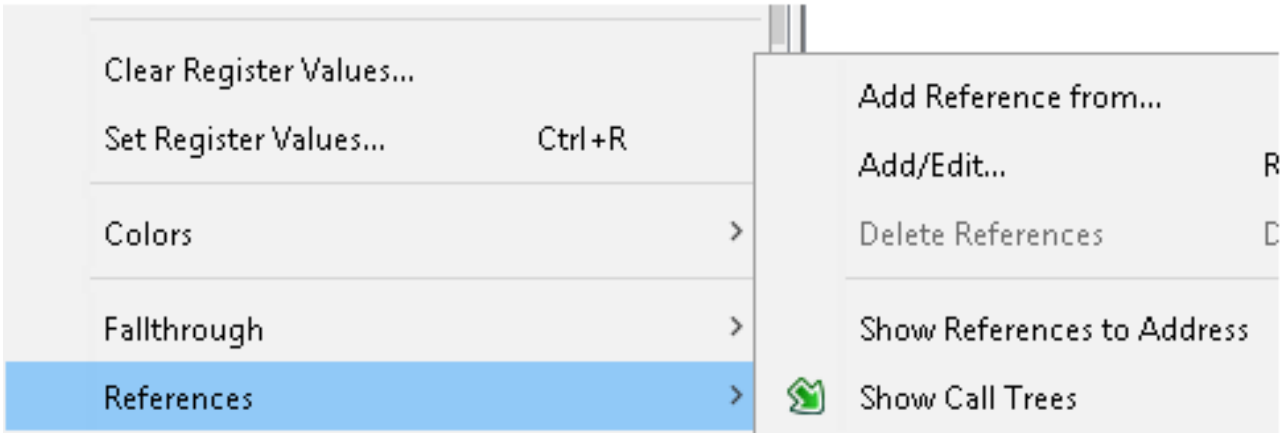
In the analyzed sample (a trickbot from early 2019), if we look for XOR instructions, we can find easily some XOR instructions modifying memory, and one of them belongs to the function that decrypts the XOR layer of the trickbot config:



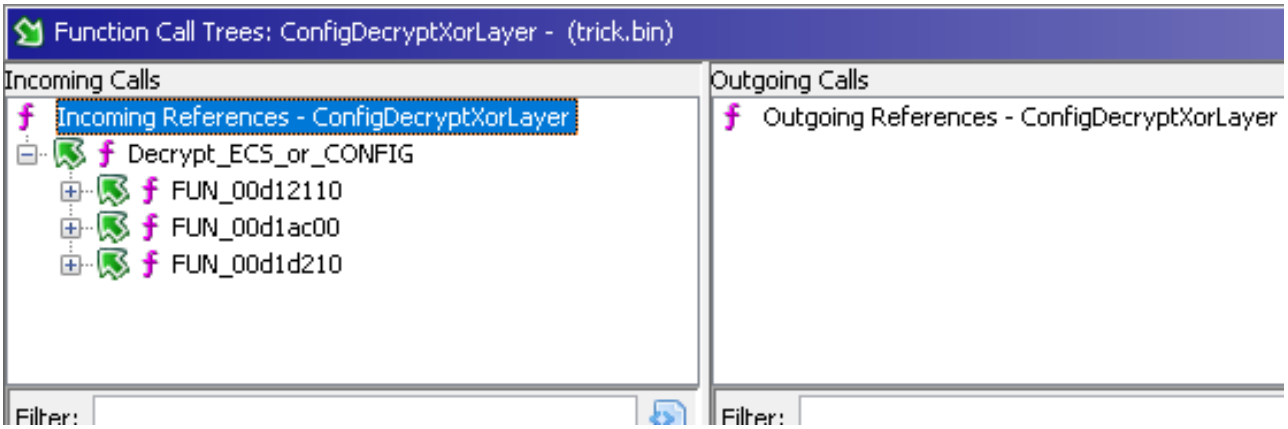
(Btw, as we can see in the image, when you select with the mouse a line in the disassembly window, the equivalent line is highlighted in the decompiled window).

Using references to find more interesting parts of the code

Once you have located an interesting point in the code, you can show a tree of calls to that point:



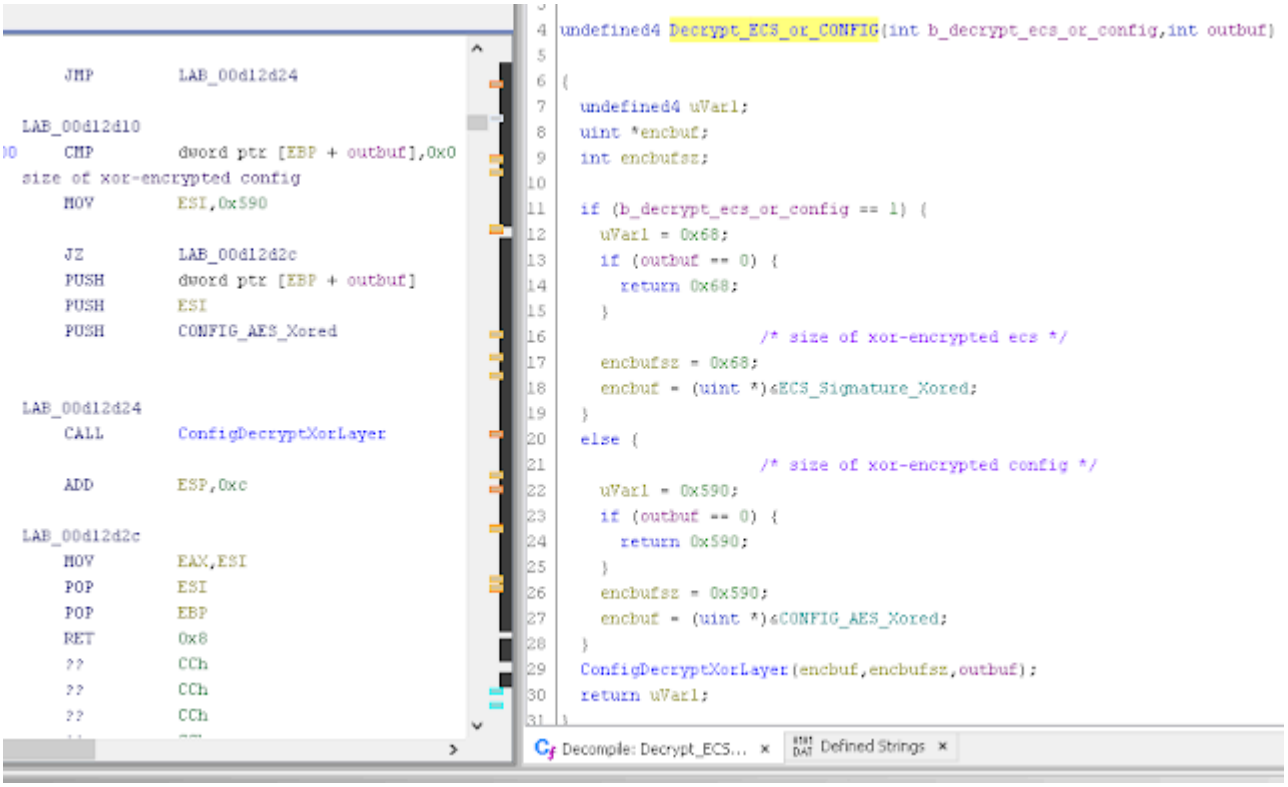
The tree makes easy to follow the incoming or outgoing references to the interesting function:



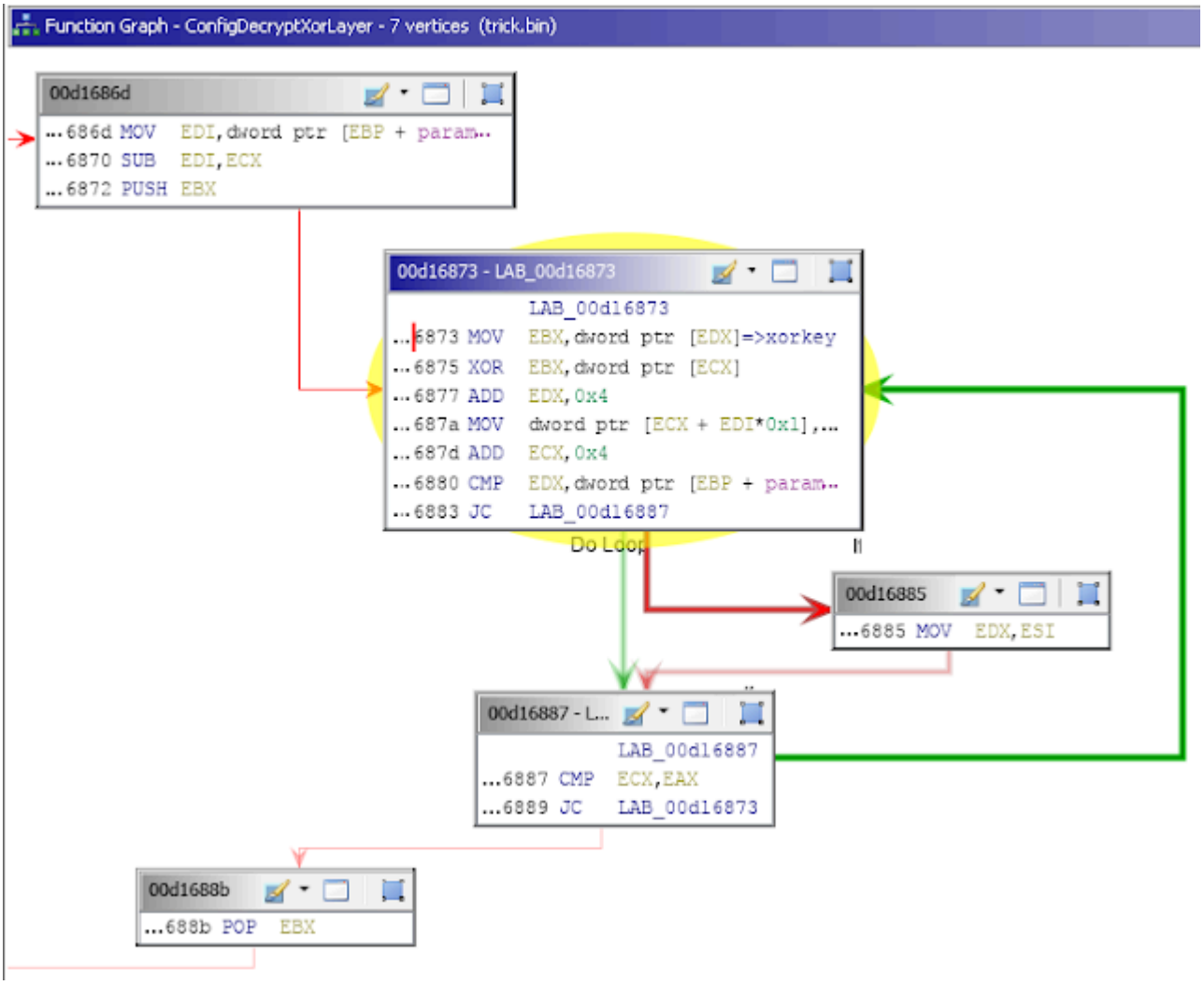
Additionally, you could highlight (select) back or forward refs to an address in the disassembly and decompiled windows.

TrickBot ECS signature and Config Xor Decryptor

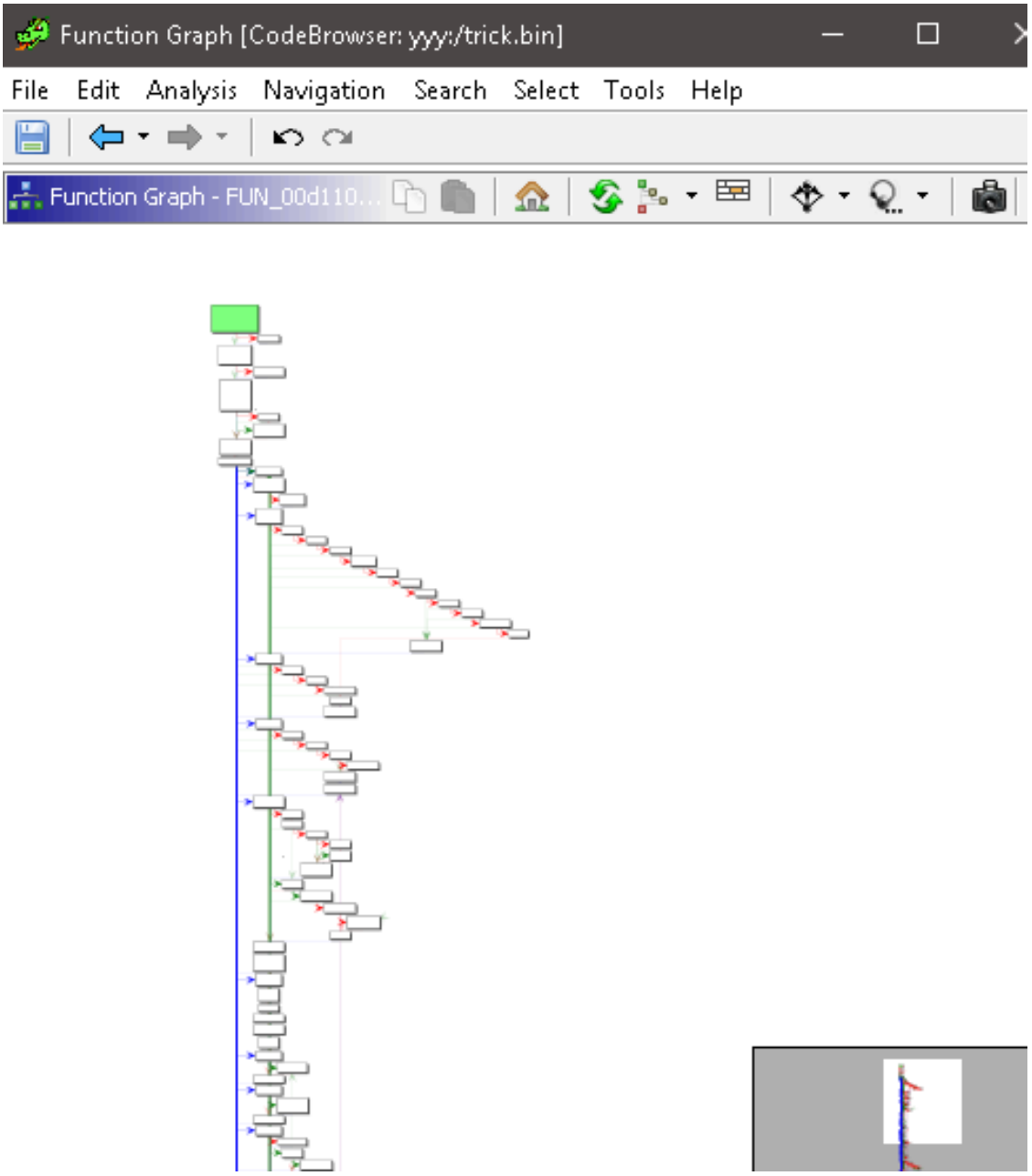
By using the call trees, we can find easily the functions that decrypts the XOR layer of the elliptic curve signature or the XOR layer of the TrickBot Config:



In addition, you can open a function graph window, similar to IDA graphs. Here is the XOR decryptor loop of TrickBot:



You can move easily on the graph, and zoom in/out with the mouse wheel:



TrickBot Strings Decryptor

About strings.. All the strings used by the newer versions of TrickBot are encrypted. While IDA was able to construct a nice table of strings that makes easy to find the decryptor:

```
data:00D25410 51 62 53 32 51 62 74 43 36 75+aQbs2qbtc6ubs3w db 'Qb52QbtC6uBS3wqp3bBSgE9NQbIt',0
data:00D25410 42 53 33 77 71 70 33 62 42 53+                                ; DATA XREF: sub_D1FE50:StringsDecryptor
data:00D2542D 59 72 46 32 51 62 53 38 48 50+aYrf2qbs8hpb2ge db 'YrF2QbS8HPB2ge',0
data:00D2543C 59 72 46 43 33 50 52 38 48 50+aYrfc3pr8hp28   db 'YrFC3PR8HP28',0
data:00D25449 51 72 46 43 48 50 32 6A 59 77+aQrfchp2jywrbhp db 'QrFCHP2jYwRBHPIyRj',0
data:00D2545C 59 77 69 53 33 50 53 53 7A 50+aYwis3psszp2jhp db 'YwiS3PSSzP2jHPi83T',0
data:00D2546F 33 72 32 32 7A 73 6B 32 36 50+a3r22zsk26pbs3n db '3r22zsk26PBS3n2jHPi83T',0
data:00D25486 67 45 6B 50 59 72 69 74 7A 77+aGekpyritzw2jhp db 'gEkPYritzw2jHPi83T',0
data:00D25499 59 72 65 4E 51 77 42 42 36 45+aYreNqwBB6ElxHP db 'YreNQwBB6ElxHPB2ge',0
data:00D254AC 51 72 46 43 48 50 32 6A 59 77+aQrfchp2jywrbhp_0 db 'QrFCHP2jYwRBHPIyRj',0
data:00D254BF 51 72 46 43 48 50 32 6A 48 30+aQrfchp2jh0ic   db 'QrFCHP2jH0ic',0
data:00D254CC 59 77 6B 32 33 30 54 4E 33 77+aYwk230tn3ww    db 'Ywk230TN3wW',0
data:00D254D8 67 45 67 45 48 50 5A 42 52 72+aGegehpzbrrsdrr db 'gEgEHPZBRrSDRrlNQw4C6uBx3bD',0
data:00D254F4 48 45 46 4B 51 77 32 4E 00    aHefkqw2n       db 'HEFKQw2N',0
data:00D254FD 48 62 32 6A 00                aHb2j           db 'Hb2j',0
data:00D25502 48 45 6C 53 67 6A 00          aHelsgj         db 'HElSgj',0
data:00D25509 48 45 6B 32 7A 73 54 00       aHek2zst        db 'HEk2zsT',0
data:00D25511 48 31 49 50 33 45 6C 74 51 72+aH1ip3eltqrtign db 'H1IP3EltQrTIgnoage',0
data:00D25524 7A 50 6F 4E 48 30 69 6A 51 77+aZponh0ijqwzvqr db 'zPoNH0ijQwZVQro1HPIyRj',0
data:00D2553B 51 62 6C 4B 48 50 71 63 67 72+aQblkhpqcgri2qr db 'QblKHPqcgri2QrTN3El0',0
```

Ghidra were not able to identify all the strings and construct a nice table, it is much lesser intuitive:

```
00d25410 51              undefined1  51h

                    DAT_00d25411                                    XREF[2]:     StringsDecryptor:00d1fe6f(R),
                                                                                 StringsDecryptor:00d1fe7c(*)
00d25411 62              undefined1  62h

                    s_S2QbtC6uBS3wqp3bBSgE9NQbIt_00d25412           XREF[2]:     StringsDecryptor:00d1fe90(*),
                                                                                 StringsDecryptor:00d1fe9c(*)
00d25412 53 32 51        ds          "S2QbtC6uBS3wqp3bBSgE9NQbIt"
         62 74 43
         36 75 42 ...
00d2542d 59              ??          59h    Y
00d2542e 72              ??          72h    r
00d2542f 46              ??          46h    F
00d25430 32              ??          32h    2
00d25431 51              ??          51h    Q
00d25432 62              ??          62h    b
00d25433 53              ??          53h    S
```

Maybe I missed something with Ghidra, but I selected the option Analysis->One shot->Ascii Strings, and these are the results. This makes difficult, for example, to find strings' decryptors.

```
  MOV         dword ptr [EBP + param_1],ECX
  MOV         EDX,dword ptr [EBP + local_c]
  ADD         EDX,0x1
  MOV         dword ptr [EBP + local_c],EDX=>s_S2QbtC6u
  JMP         LAB_00d1fe66

B_00d1fe95                                          XREF[1]:
  MOV         EAX,dword ptr [EBP + param_2]
  PUSH        EAX
  MOV         ECX,dword ptr [EBP + local_c]
  PUSH        ECX=>s_S2QbtC6uBS3wqp3bBSgE9NQbIt_00d2541
  CALL        StringsDecryptorSub

  ADD         ESP,0x8
  MOV         dword ptr [EBP + local_8],EAX
  MOV         EAX,dword ptr [EBP + local_8]
  MOV         ESP,EBP
  POP         EBP
  RET
```

```
2  undefined * __cdecl StringsDecryptor(int param_1,i
3
4  {
5    undefined *puVar1;
6    char *local_c;
7
8    local_c = &StringsTable;
9    param_1 = param_1 + -1;
10   while (param_1 != 0) {
11     while (*local_c != 0) {
12       local_c = local_c + 1;
13     }
14     param_1 = param_1 + -1;
15     local_c = local_c + 1;
16   }
17   puVar1 = StringsDecryptorSub(local_c,param_2);
18   return puVar1;
19 }
20
```

Conclusion

in spite of the fact that I really love IDA (and WinDbg), I liked this framework, and I will continue using it.
