Web Security and Malware Analysis

Answers for Assignment 6

SILVIA LUCIA SANNA – 70/90/00053

Task 1 – PE structure

In this task I have to analyse two different exe files and 1 dll.

- Dos header and stub, differences exe1 and dll1: in the DOS header the difference between the files is e_lfanew that in the exe1 is 000000E8 while in the dll is 000000E0 (as they are in little endian). This value stands for the point where the PE starts and from this, we can have the length of the stub: in exe1 is 84 words, while in dll1 is 80 words.

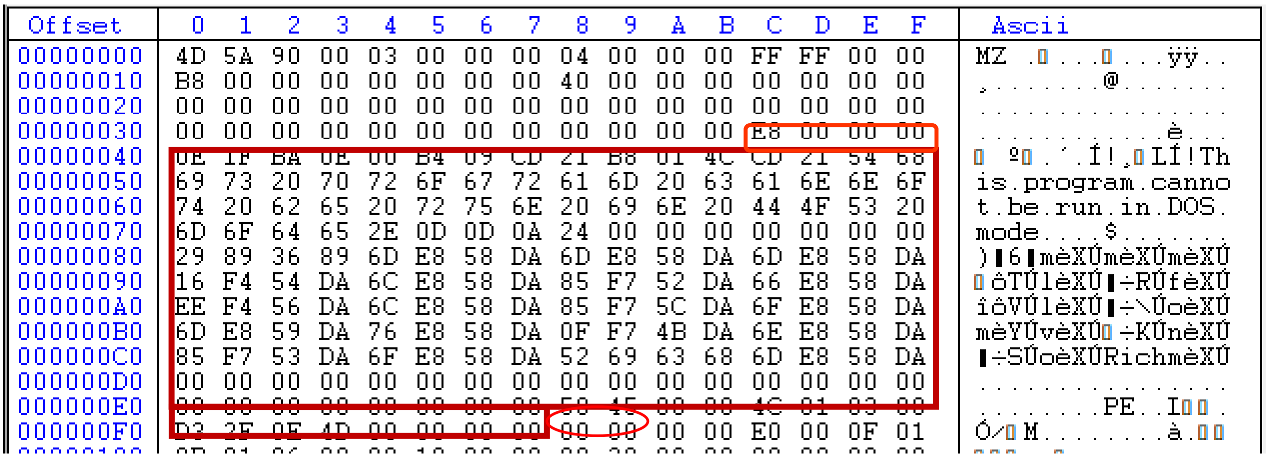


Figura 1: exe01 header and stub

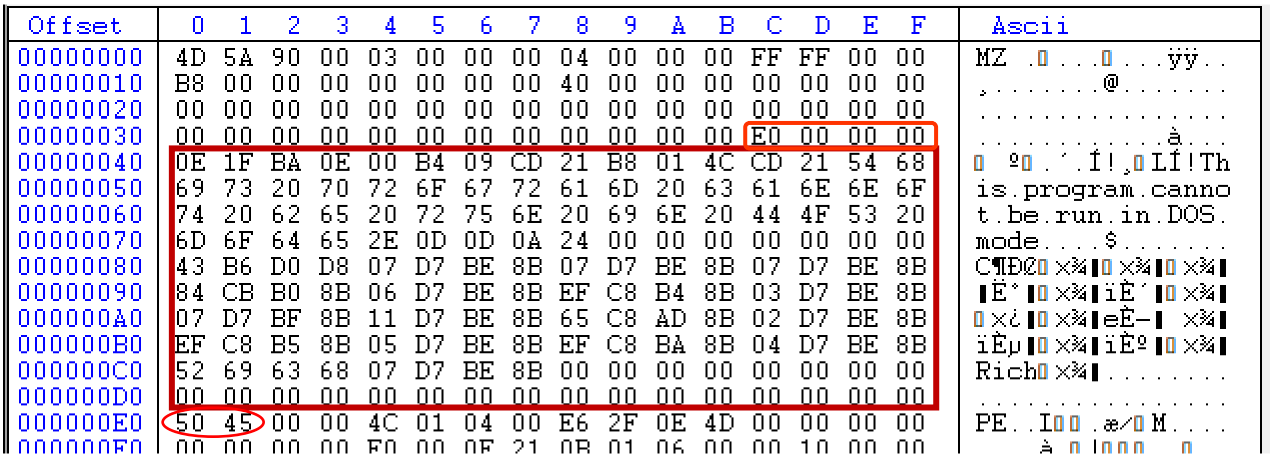


Figura 2: dll01 header and stub

- File and Optional headers, parts of these headers critical for the execution:

Lab01-01.exe | Lab01-01.dll | Lab01-03.exe

Member	Offset	Size	Value	Meaning
Machine	000000EC	Word	014C	Intel 386
NumberOfSections	000000EE	Word	0003	
TimeDateStamp	000000F0	Dword	4D0E2FD3	
PointerToSymbolTable	000000F4	Dword	00000000	
NumberOfSymbols	000000F8	Dword	00000000	
SizeOfOptionalHeader	000000FC	Word	00E0	
Characteristics	000000FE	Word	010F	Click here

Figura 3: File header exe01

Lab01-01.exe | **Lab01-01.dll** | Lab01-03.exe

Member	Offset	Size	Value	Meaning
Machine	000000E4	Word	014C	Intel 386
NumberOfSections	000000E6	Word	0004	
TimeDateStamp	000000E8	Dword	4D0E2FE6	
PointerToSymbolTable	000000EC	Dword	00000000	
NumberOfSymbols	000000F0	Dword	00000000	
SizeOfOptionalHeader	000000F4	Word	00E0	
Characteristics	000000F6	Word	210E	Click here

Figura 4: File header dll01

We can easily see that the offsets of the exe are all shifted by 8 with respect to offsets in dll file. This is because of the difference in the Stubs' length. From these headers, we can have very important info about the program and some of them in my opinion are critical in the sense of important.

In the file header the critical parameters are:

- Machine which stands for the type of target machine, in our case is Intel386 or later.
- Pointer to symbol table containing the file offset of COFF symbol table that, as in our case, if it is 0 means that the file is an image (executable file).

- Size of optional header is important because it is essential for executable files but not for object files for which it should be 0 value. In our case is 00E0 and means that aligns data in a boundary of 8192 bytes.

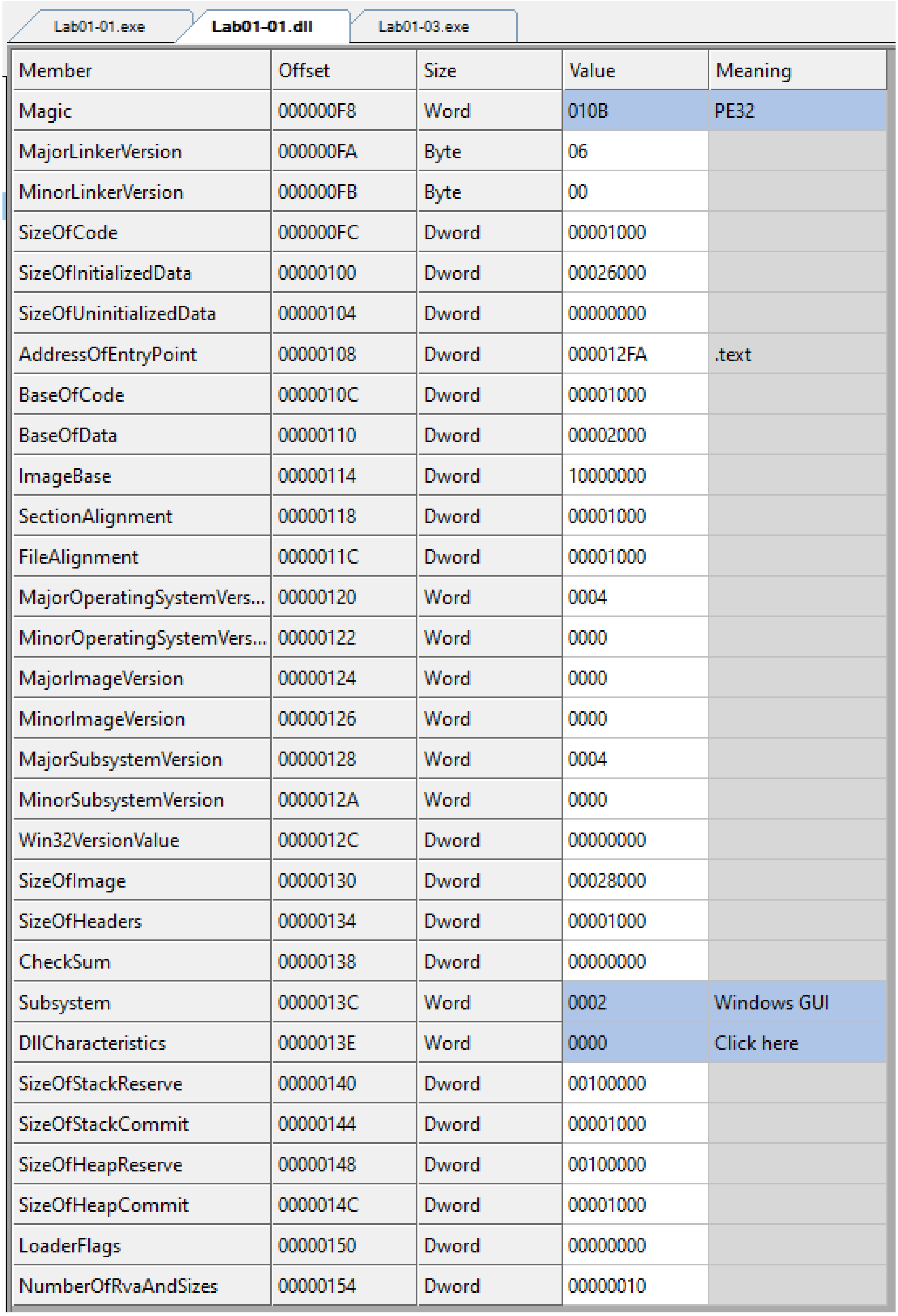
While, in the optional header critical parameters in my opinion are:

- Size of code, uninitialized and initialized data because express how much code and data there are in the program.
- Base of code and base of data to know where code and data section begin. In this field also file and section alignment are important because express the alignment factor used to align sections or raw data of sections when loaded in memory.
- Size stack reserve and size heap reserve in my opinion are both important and critical in the dangerous sense, even if they stand for stack and heap which are two different memory area, but we can apply same line of reasoning. Important because says how much stack/heap space will this code use but dangerous because maybe can help the buffer overflow attack.

Lab01-01.exe | Lab01-01.dll | Lab01-03.exe

Member	Offset	Size	Value	Meaning
Magic	00000100	Word	010B	PE32
MajorLinkerVersion	00000102	Byte	06	
MinorLinkerVersion	00000103	Byte	00	
SizeOfCode	00000104	Dword	00001000	
SizeOfInitializedData	00000108	Dword	00002000	
SizeOfUninitializedData	0000010C	Dword	00000000	
AddressOfEntryPoint	00000110	Dword	00001820	.text
BaseOfCode	00000114	Dword	00001000	
BaseOfData	00000118	Dword	00002000	
ImageBase	0000011C	Dword	00400000	
SectionAlignment	00000120	Dword	00001000	
FileAlignment	00000124	Dword	00001000	
MajorOperatingSystemVers...	00000128	Word	0004	
MinorOperatingSystemVers...	0000012A	Word	0000	
MajorImageVersion	0000012C	Word	0000	
MinorImageVersion	0000012E	Word	0000	
MajorSubsystemVersion	00000130	Word	0004	
MinorSubsystemVersion	00000132	Word	0000	
Win32VersionValue	00000134	Dword	00000000	
SizeOfImage	00000138	Dword	00004000	
SizeOfHeaders	0000013C	Dword	00001000	
CheckSum	00000140	Dword	00000000	
Subsystem	00000144	Word	0003	Windows Console
DllCharacteristics	00000146	Word	0000	Click here
SizeOfStackReserve	00000148	Dword	00100000	
SizeOfStackCommit	0000014C	Dword	00001000	
SizeOfHeapReserve	00000150	Dword	00100000	
SizeOfHeapCommit	00000154	Dword	00001000	
LoaderFlags	00000158	Dword	00000000	
NumberOfRvaAndSizes	0000015C	Dword	00000010	

Figura 5: Optional header in exe01



Lab01-01.exe | **Lab01-01.dll** | Lab01-03.exe

Member	Offset	Size	Value	Meaning
Magic	000000F8	Word	010B	PE32
MajorLinkerVersion	000000FA	Byte	06	
MinorLinkerVersion	000000FB	Byte	00	
SizeOfCode	000000FC	Dword	00001000	
SizeOfInitializedData	00000100	Dword	00026000	
SizeOfUninitializedData	00000104	Dword	00000000	
AddressOfEntryPoint	00000108	Dword	000012FA	.text
BaseOfCode	0000010C	Dword	00001000	
BaseOfData	00000110	Dword	00002000	
ImageBase	00000114	Dword	10000000	
SectionAlignment	00000118	Dword	00001000	
FileAlignment	0000011C	Dword	00001000	
MajorOperatingSystemVers...	00000120	Word	0004	
MinorOperatingSystemVers...	00000122	Word	0000	
MajorImageVersion	00000124	Word	0000	
MinorImageVersion	00000126	Word	0000	
MajorSubsystemVersion	00000128	Word	0004	
MinorSubsystemVersion	0000012A	Word	0000	
Win32VersionValue	0000012C	Dword	00000000	
SizeOfImage	00000130	Dword	00028000	
SizeOfHeaders	00000134	Dword	00001000	
CheckSum	00000138	Dword	00000000	
Subsystem	0000013C	Word	0002	Windows GUI
DllCharacteristics	0000013E	Word	0000	Click here
SizeOfStackReserve	00000140	Dword	00100000	
SizeOfStackCommit	00000144	Dword	00001000	
SizeOfHeapReserve	00000148	Dword	00100000	
SizeOfHeapCommit	0000014C	Dword	00001000	
LoaderFlags	00000150	Dword	00000000	
NumberOfRvaAndSizes	00000154	Dword	00000010	

Figura 6: Optional header dll01

- Differences between bases of codes and bases of data: base of code and base of data of exe and dll contain the same values but what differs is the offset: offset of base of code and base of data of the exe are equal to the offset of image base and section alignment of the dll. Exe base of code and dll Image base have same offset value because the beginning of code section (base of code) and the first byte of the image (image base) can mean the same thing. While the beginning of data section (base of data) and alignment of all sections loaded in memory (section alignment) can also mean same thing if the first elements loaded in memory are data sections. This reasoning encounters a contradiction in offset sizeOfCode in exe and offset sizeOfUninitializedData in dll because they have same offset (104) but no relation. So, we can easily see that exe offsets are shifted from dll offset and some of them have a relationship but not all of them.
- Image bases: are completely different values because the image base stands for the address of the first byte of the image when loaded in memory and as dll and exe files are completely different and loaded in memory in two different areas, of course these values are different.

- Section header:

Lab01-01.exe | Lab01-01.dll | Lab01-03.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	00000970	00001000	00001000	00001000	00000000	00000000	0000	0000	60000020
.rdata	000002B2	00002000	00001000	00002000	00000000	00000000	0000	0000	40000040
.data	000000FC	00003000	00001000	00003000	00000000	00000000	0000	0000	C0000040

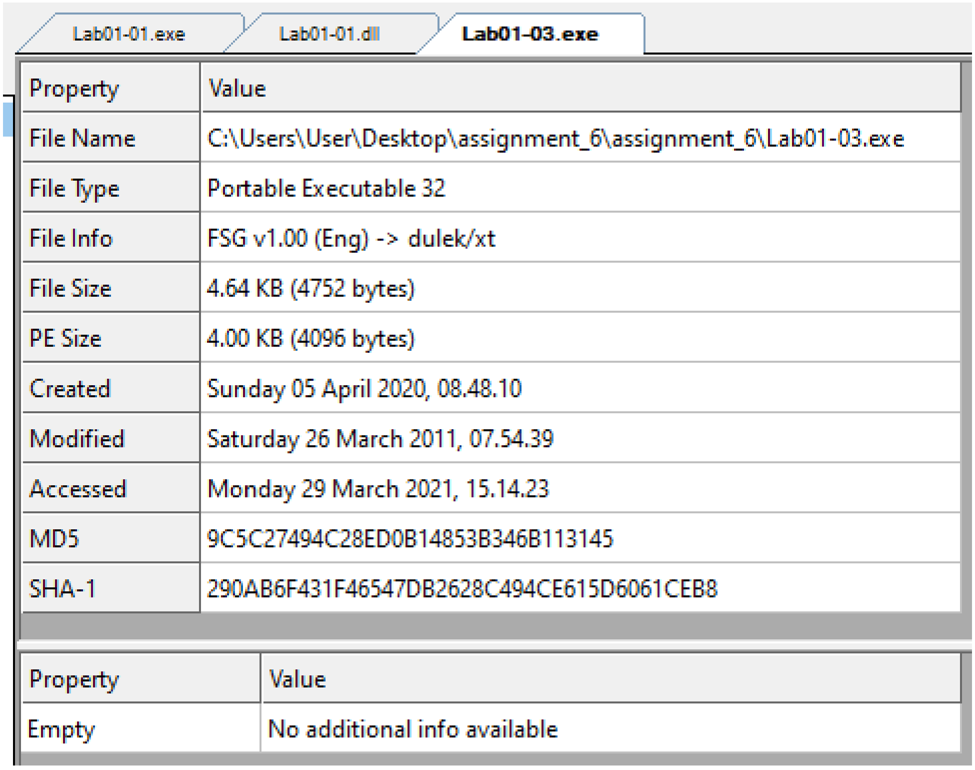
Figura 7: section header exe01

Lab01-01.exe | Lab01-01.dll | Lab01-03.exe

Name	Virtual Size	Virtual Address	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Characteristics
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dword
.text	0000039E	00001000	00001000	00001000	00000000	00000000	0000	0000	60000020
.rdata	00023FC6	00002000	00024000	00002000	00000000	00000000	0000	0000	40000040
.data	0000006C	00026000	00001000	00026000	00000000	00000000	0000	0000	C0000040
.reloc	00000204	00027000	00001000	00027000	00000000	00000000	0000	0000	42000040

Figura 8: section header dll01

As we can see from the images above, text, rdata and data of exe and dll have equal values but different virtual size, virtual address and so on; between them there is no constant difference that makes think about a specific relation. What catches the eye is the reloc field present in the dll but not in the exe and this because reloc stands for library relocation: this means that in the dll there are different libraries to be used and a process of relocation is implemented (a space of memory is used to readdress a specific function if its preferred address is already used by another one).



Property	Value
File Name	C:\Users\User\Desktop\assignment_6\assignment_6\Lab01-03.exe
File Type	Portable Executable 32
File Info	FSG v1.00 (Eng) -> dulek/xt
File Size	4.64 KB (4752 bytes)
PE Size	4.00 KB (4096 bytes)
Created	Sunday 05 April 2020, 08.48.10
Modified	Saturday 26 March 2011, 07.54.39
Accessed	Monday 29 March 2021, 15.14.23
MD5	9C5C27494C28ED0B14853B346B113145
SHA-1	290AB6F431F46547DB2628C494CE615D6061CEB8

Property	Value
Empty	No additional info available

Figura 9: section header exe03

The section header of the 03exe file has something strange, very different from the previous one: its fields are like what you see when you make “properties/info” in a file, we do not have the classical info about the section and in fact we have no reloc field. I have no explanation about why this uncommon structure is encountered here.

Bonus task 1

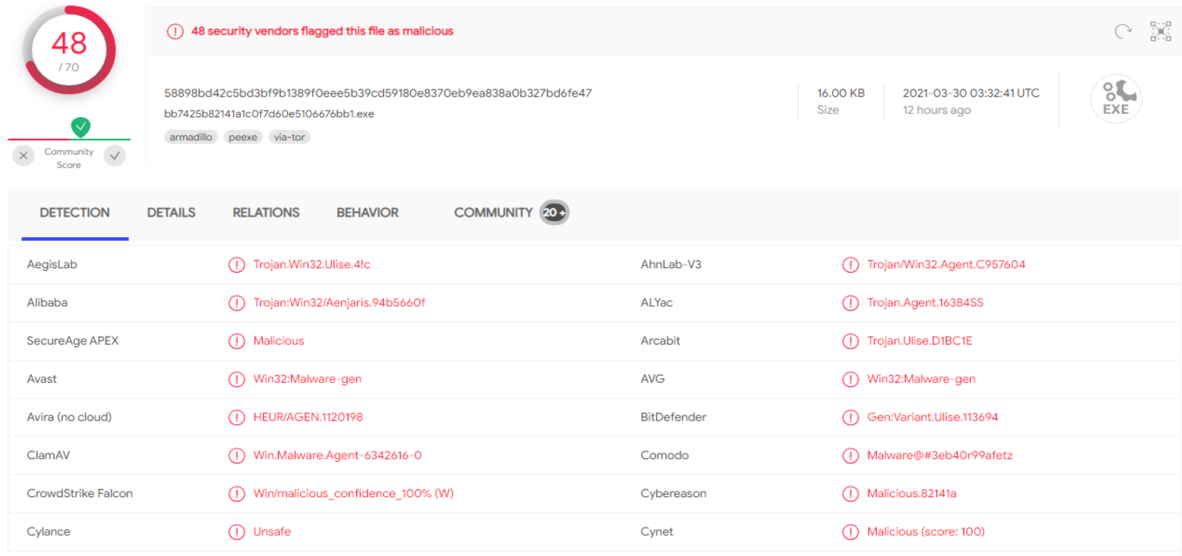


Figura 10: virustotal exe01

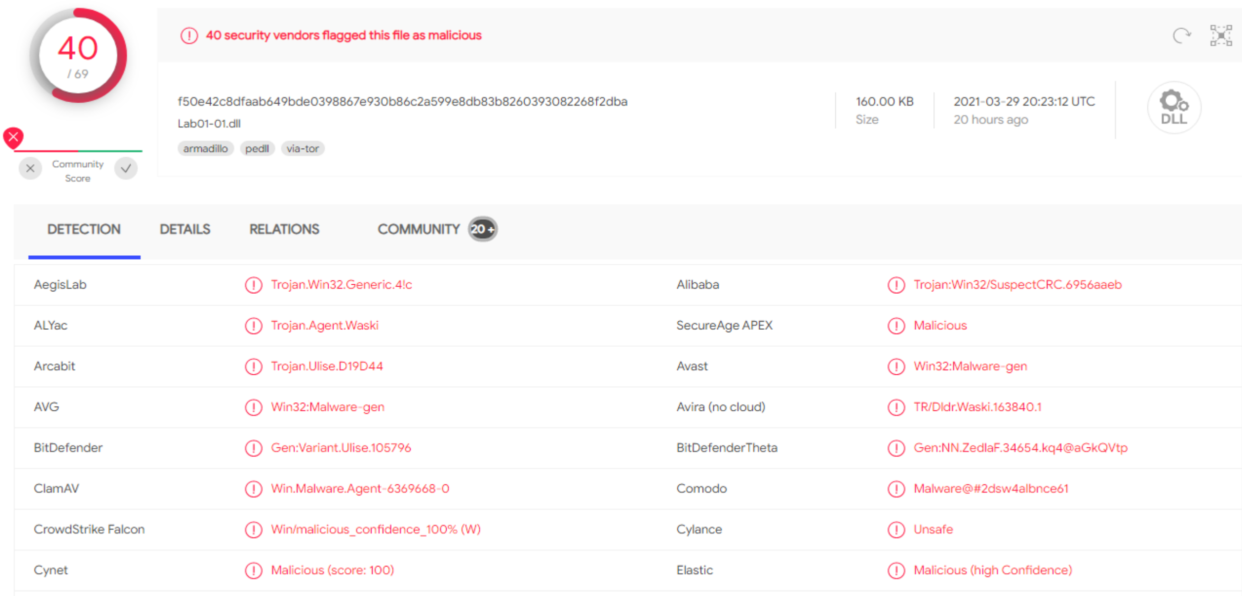


Figura 11: virustotal dll01

By analysing exe01 and dll01 with VirusTotal I can easily understand that they are recognised as something malicious and above all as a Trojan for Windows 32. I think that also analysing the files with idaPRO I should see a particular structure regarding how trojan horse works. I expected to find like 2 processes running at the same time like in parallel or running one after the other seeing specific callings from the first to the second (malicious one) but I didn’t see anything, or maybe I didn’t recognise some instructions.

Task 2 – IDA PRO Practice

1) The address of DllMain is 1000D02E, the value in left column where the DllMain is encountered.

```
.text:1000D02E
.text:1000D02E ; =============== S U B R O U T I N E =======================================
.text:1000D02E
.text:1000D02E
.text:1000D02E ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
.text:1000D02E _DllMain@12     proc near               ; CODE XREF: DllEntryPoint+4B↓p
.text:1000D02E                                         ; DATA XREF: sub_100110FF+2D↓o
.text:1000D02E
.text:1000D02E hinstDLL        = dword ptr  4
.text:1000D02E fdwReason       = dword ptr  8
.text:1000D02E lpvReserved     = dword ptr  0Ch
.text:1000D02E
.text:1000D02E                 mov     eax, [esp+fdwReason]
```

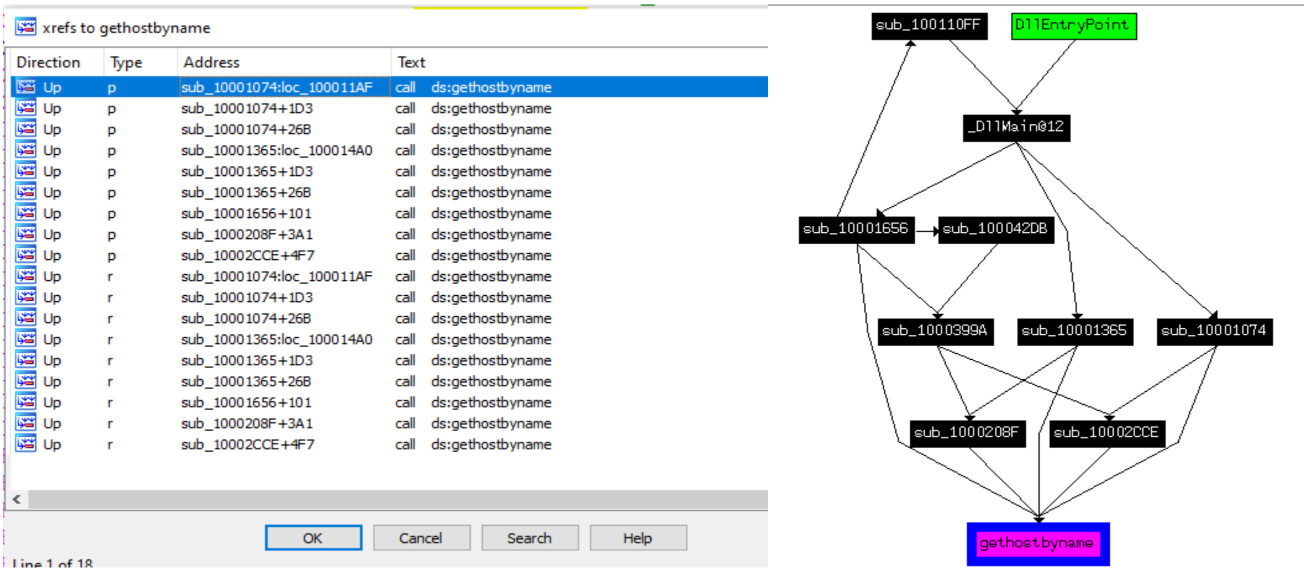
Figura 12: DllMain address

2) The import to gethostbyname is at the address 100163CC, we can see it in "Imports" window and scrolling down up to see gethostbyname".

100163CC	52	gethostbyname	WS2_32

Figura 13: gethostbyname imports

3) We can see how many functions call "gethostbyname" by double clicking on the name on the image above and then ctrl+X to list all cross-references. In this list we can identify some repeated results, so another way is to view "Xrefs graph to" with right click on gethostbyname. From this graph we can see 5 functions calling directly gethostbyname.


xrefs to gethostbyname

Direction	Type	Address	Text
Up	p	sub_10001074:loc_100011AF	call ds:gethostbyname
Up	p	sub_10001074+1D3	call ds:gethostbyname
Up	p	sub_10001074+26B	call ds:gethostbyname
Up	p	sub_10001365:loc_100014A0	call ds:gethostbyname
Up	p	sub_10001365+1D3	call ds:gethostbyname
Up	p	sub_10001365+26B	call ds:gethostbyname
Up	p	sub_10001656+101	call ds:gethostbyname
Up	p	sub_1000208F+3A1	call ds:gethostbyname
Up	p	sub_10002CCE+4F7	call ds:gethostbyname
Up	r	sub_10001074:loc_100011AF	call ds:gethostbyname
Up	r	sub_10001074+1D3	call ds:gethostbyname
Up	r	sub_10001074+26B	call ds:gethostbyname
Up	r	sub_10001365:loc_100014A0	call ds:gethostbyname
Up	r	sub_10001365+1D3	call ds:gethostbyname
Up	r	sub_10001365+26B	call ds:gethostbyname
Up	r	sub_10001656+101	call ds:gethostbyname
Up	r	sub_1000208F+3A1	call ds:gethostbyname
Up	r	sub_10002CCE+4F7	call ds:gethostbyname

Figura 14: cross-references for gethostbyname

4) As we can see from the first part of this block, this gethostbyname will make a request to a specific host which is set in eax (from value 13 up to the end in the string located at off_10019040, which is pics.practicalmalwareanalysis.com). Then will make a comparison with ebx value and if have equal values, makes some string copy, otherwise will make an ipconfig/flushdns.

```
.data:10019194 aThisIsRdoPicsP db '[This is RDO]pics.praticalmalwareanalysis.com',0
```

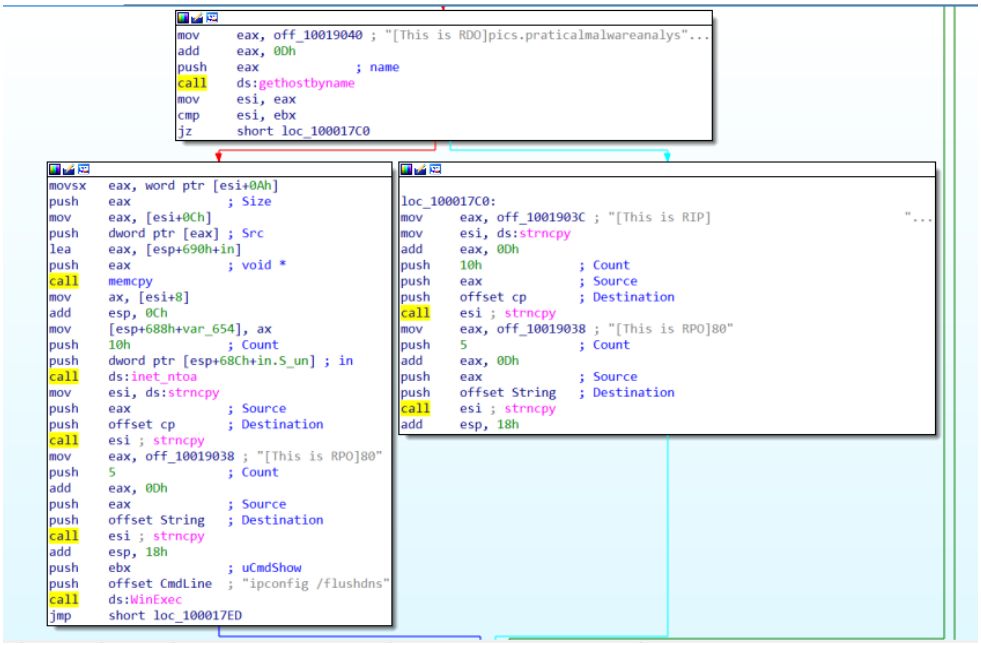


Figura 15: gethostbyname

5-6) For subroutine starting at 10001656 and ending at 10002089 Ida recognized 23 local variables only 1 parameter (lpThreadParameter, the one in input). Normally we can determine a parameter if it is like ebp+xxx and ebp-xxx for local variables.

```
.text:10001656
.text:10001656
.text:10001656 ; DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)
.text:10001656 sub_10001656     proc near               ; DATA XREF: sub_1000D02E+C8↓o
.text:10001656
.text:10001656 var_675          = byte ptr -675h
.text:10001656 var_674          = dword ptr -674h
.text:10001656 hModule          = dword ptr -670h
.text:10001656 timeout          = timeval ptr -66Ch
.text:10001656 name             = sockaddr ptr -664h
.text:10001656 var_654          = word ptr -654h
.text:10001656 Dst              = dword ptr -650h
.text:10001656 Str1             = byte ptr -644h
.text:10001656 var_640          = byte ptr -640h
.text:10001656 CommandLine      = byte ptr -63Fh
.text:10001656 Str              = byte ptr -63Dh
.text:10001656 var_638          = byte ptr -638h
.text:10001656 var_637          = byte ptr -637h
.text:10001656 var_544          = byte ptr -544h
.text:10001656 var_50C          = dword ptr -50Ch
.text:10001656 var_500          = byte ptr -500h
.text:10001656 Buf2             = byte ptr -4FCh
.text:10001656 readfds          = fd_set ptr -4BCh
.text:10001656 buf              = byte ptr -3B8h
.text:10001656 var_3B0          = dword ptr -3B0h
.text:10001656 var_1A4          = dword ptr -1A4h
.text:10001656 var_194          = dword ptr -194h
.text:10001656 WSAData          = WSAData ptr -190h
.text:10001656 lpThreadParameter= dword ptr  4
.text:10001656
.text:10001656                  sub     esp, 678h
```

This is a dll and we do not have ebp, so the way to check the parameters is to see inputs of the function.

Figura 16: variables and parameter for subroutine 10001656

7) First of all, I must open the String window with shift+F12. The string “\cmd.exe /c” is located at 10095B34.

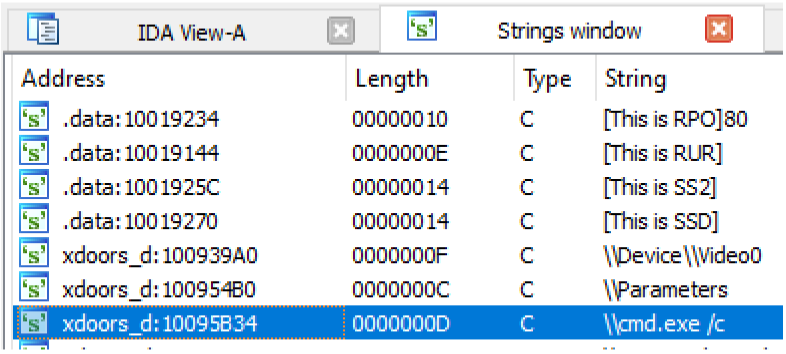


Figura 17: String window "\cmd.exe /c"

8) I think that in the area of code of \cmd.exe /c a shell has been opened and this prints the total time where the machine was up and where was idle and before closing asks for a number to encrypt this remote shell session.

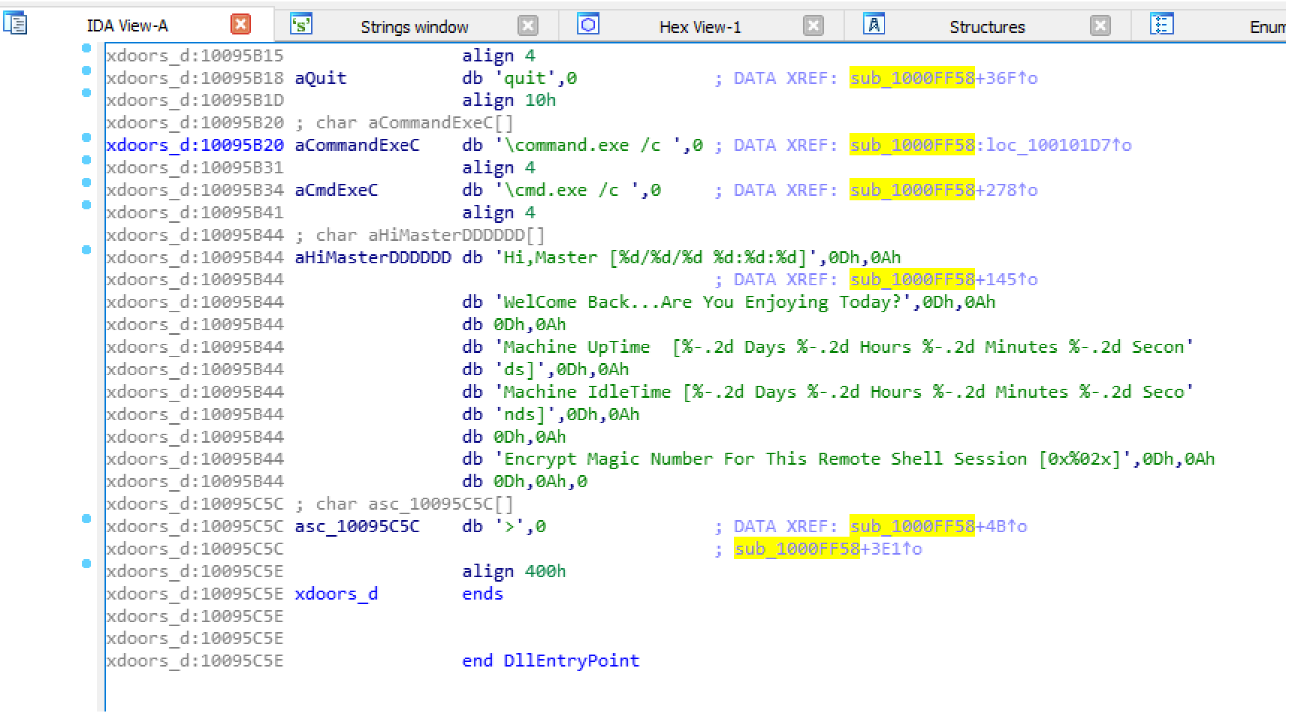


Figura 18: area of code of "\cmd.exe /c"

9) To know how the program sets “dword_1008E5C4”, we have to check its cross references: analysing the first one we see the instruction “mov dword_1008E5C4, eax”, this means that the variable is set with eax value, which can be the output of the previous instruction that is the call to function “sub_10003695”. Let’s try to analyse it by double clicking on its name: we can

easily see that this function will check OS version and will return the output in variable eax. So dword_1008E5C4 contains the version of the operating system where the program is running.

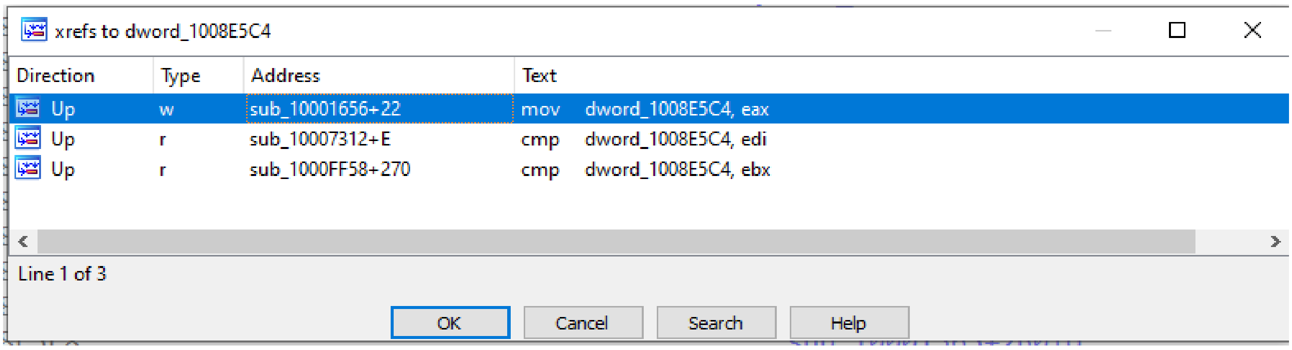


Figura 19: references of dword_1008E5C4

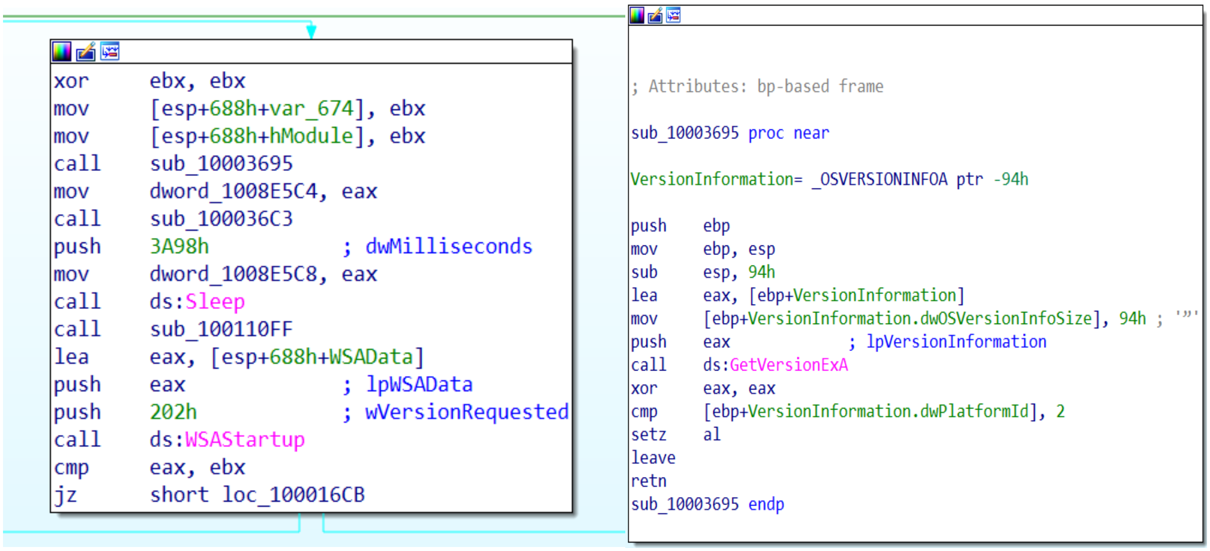


Figura 21: function where dword_1008E5C4 (left) and function sub_10003695 (right)

10) In loc_10010444, when memcmp returns zero, means that jnz is false (the comparison result is zero so, not zero is false). As the jnz is false, the program will not jump to mbase. In Ida, the red edge in the graphs, means that the jump is not taken: a jump is not taken when the condition is false, in this case the next instructions will be the right image composed by 3 different blocks run one after the other (line blu).

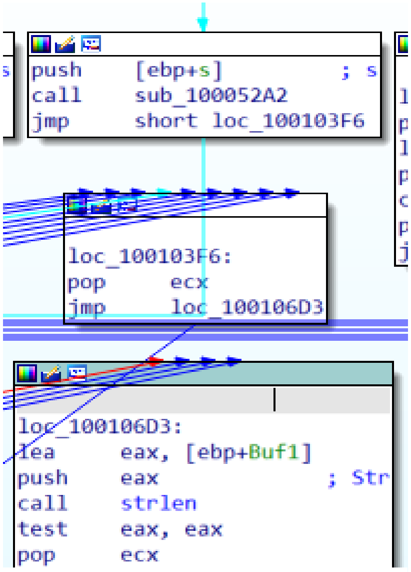


Figura 20: next 3 blocks executed when memcmp returns 0

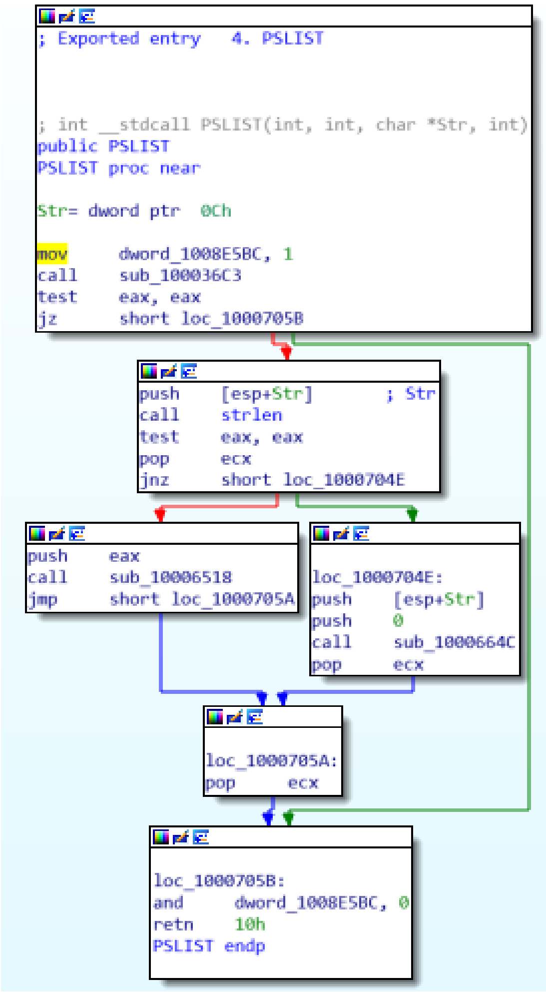


Figura 22: Graph of function PSList

11) In the first block of PSList there is the call to function sub_100036C3 which checks the OS version (we can see it by double clicking on its name and read its program) and if this is right makes a mask with the value of eax and ends. If the OS version is not the right one, checks the destination value and if this is the right one, pops ecx value and calls a certain process (sub_10006518) at which ends pops ecx value, makes a mask with eax and ends. If the destionation value is not the right one calls a different process (sub_1000664C) and as it ends, pops ecx, makes a mask with eax and ends.

12) We can see the APIs called by the function sub_10004E79 by right clicking on its name, both at ".text:10004E79" and ".idata:10016120", and then "Xrefs graph from". This will show the graph containing all APIs called by the function both directly and indirectly (called directly from its direct functions). Analysing these points, we can name this function as "Get System Language" which will get (GetSystemDefaultLangID a Windows API that will return the language id for the local system), print (sprint), save dynamically (malloc and free functions for dynamic allocation) and send (send function) this value.

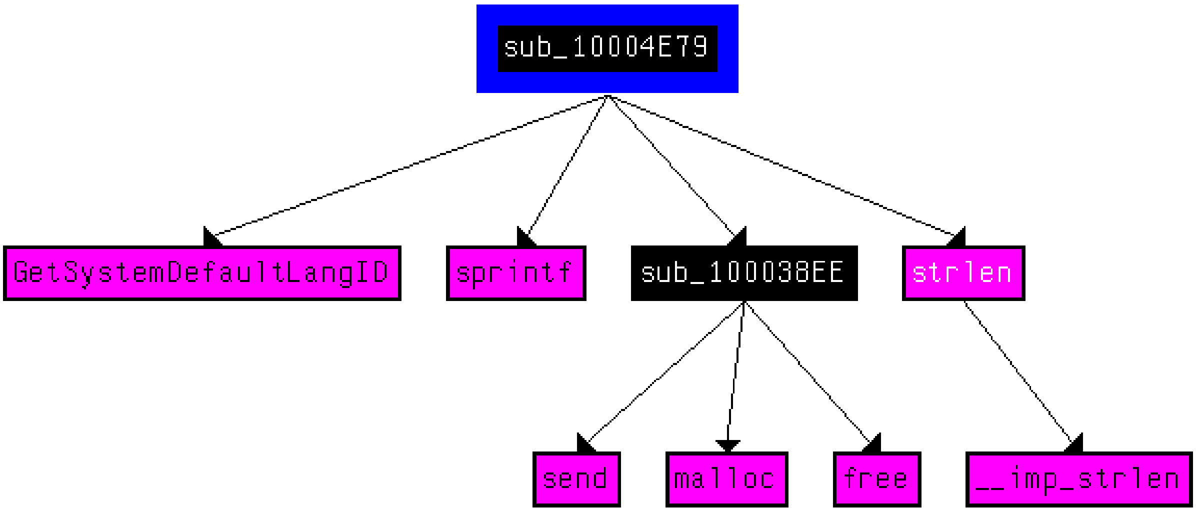


Figura 23: sub_10004E79 APIs

13) To analyse APIs of a certain function up to a specific level, we have to go in “View -> Graphs -> User Xrefs chart” and here we have to compile the table properly: as start and end address we put the address of the function we have to analyse (in our case is .text:10001074) and recursion depth is the level where we want to stop (for us is 2).

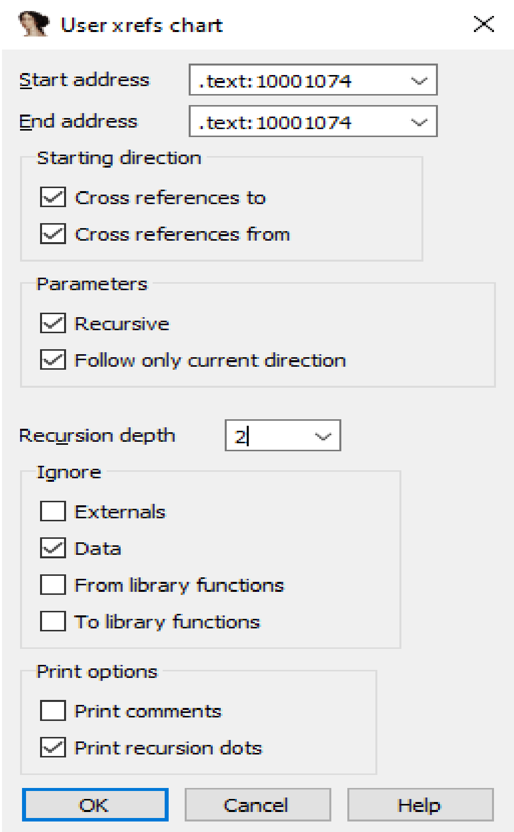


Figura 24: How to set depth level in Xrefs chart

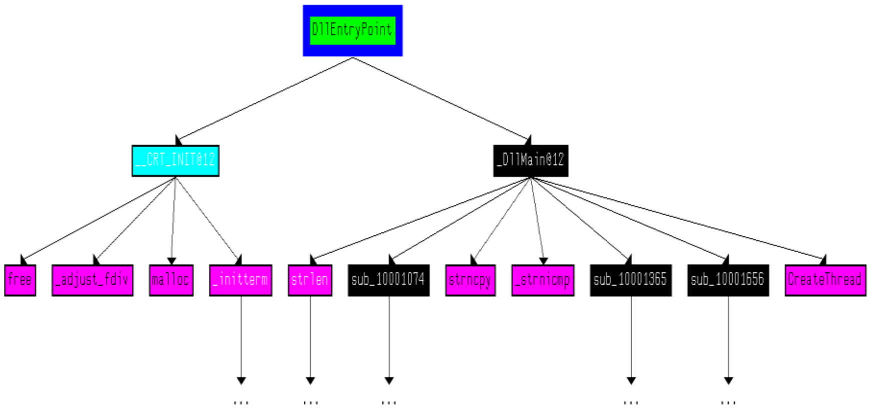


Figura 25: APIs up to level 2

14) To understand how many times will the program sleep, we can easily view it in the graph mode. In the following block there are all elements to understand all values. In the first

off_10019020 is loaded in eax at which, in the second line, is added 0D in hexadecimal format and then in the fifth line this value is multiplied with 3E8. When the function Sleep is called, the only parameter it receives is eax. So, we have to do these computations to know how many milliseconds it will sleep, and above all we need to know the value of off_10019020.

```
loc_10001341:
mov     eax, off_10019020
add     eax, 0Dh
push    eax                 ; Str
call    ds:atoi
imul    eax, 3E8h
pop     ecx
push    eax                 ; dwMilliseconds
call    ds:Sleep
xor     ebp, ebp
jmp     loc_100010B4
sub_10001074 endp
```

Figura 26: Sleep function block

The variable off_10019020 contains the offset of the string “[This is CTI]30”, and in the strings logic, the offset of a string is the offset of the first value, in our case character “[”. So, the addition 0D means to consider from character 13 up to the end of the string: this is “30”. Then a function “atoi” is called which will convert the string to integer, store again in eax and multiply with 1000. We can now easily say that the program will sleep 3 seconds.

```
.data:10019020 off_10019020    dd offset aThisIsCti30  ; DATA XREF: sub_10001074:loc_10001341↑r
.data:10019020                                         ; sub_10001365:loc_10001632↑r ...
.data:10019020                                         ; "[This is CTI]30"
.data:100192AC aThisIsCti30    db '[This is CTI]30',0  ; DATA XREF: .data:off_10019020↑o
```

Figura 27: offset and value of string "[This is CTI]30"

15) The function socket receives in input 3 parameters which are respectively: int af=2, int type=1, int protocol=6.

```
.text:100016FB     push    6           ; protocol
.text:100016FD     push    1           ; type
.text:100016FF     push    2           ; af
.text:10001701     call    ds:socket
.text:10001707     mov     edi, eax
.text:10001709     cmp     edi, 0FFFF; SOCKET __stdcall socket(int af, int type, int protocol)
.text:1000170C     jnz     short loc_          extrn socket:dword     ; CODE XREF: sub_10001656+AB↑p
.text:1000170E     call    ds:WSAGetL                                 ; sub_1000208F+3F4↑p ...
```

Figura 28: socket function inputs