CNIT 58100 CFM: CYBERFORENSICS OF MALWARE – LAB 5 (PART 2)

Ibrahim Waziri Jr

PhD in Information Security (CERIAS)

Lab 5 – Part 2

Due on: September 17th, 2014 (Week 3)

Instructor: **Associate Prof Sam Liles**

Purdue University

2014

Abstract

This lab covers the skill discussed in chapter 5 of the text. The practice covered in this lab is all based on malware analysis and the Interactive Disassembler Professional (IDA Pro) software. The malware files used are provided as an extension of the text for practical purposes.

The lab consists of multiple questions that require short answers. Throughout this lab we used a special tool known as IDA Pro for the malware analysis.

This paper provides answers to Chapter 5 lab. The lab uses the file *Lab05-01.dll.* This file is a malwares and therefore could be harmful if used for non-training purposes.

The goal of this lab is to give a hands-on experience with IDA Pro.

Lab 5 -1

Analyze the malware found in the file Lab05-01.dll using only IDA Pro. The goal of this lab is to give you hands-on experience with IDA Pro. If you've already worked with IDA Pro, you may choose to ignore these questions and focus on reverse-engineering the malware.

Questions

Q1. What is the address of DllMain?

Q2. Use the Imports window to browse to *gethostbyname*. Where is the import located?

Q3. How many functions call *gethostbyname*?

Q4. Focusing on the call to *gethostbyname* located at 0x10001757, can you figure out which DNS request will be made?

Q5. How many local variables has IDA Pro recognized for the subroutine at 0x10001656?

Q6. How many parameters has IDA Pro recognized for the subroutine at 0x10001656?

Q7. Use the Strings window to locate the string \cmd.exe /c in the disassembly. Where is it located?

Q8. What is happening in the area of code that references \cmd.exe /c?

Q9. In the same area, at 0x100101C8, it looks like dword_1008E5C4 is a global variable that helps decide which path to take. How does the malware set dword_1008E5C4? (Hint: Use dword_1008E5C4's cross-references.)

Q10. A few hundred lines into the subroutine at 0x1000FF58, a series of comparisons use memcmp to compare strings. What happens if the string comparison to robotwork is successful (when memcmp returns 0)?

Q11. What does the export PSLIST do?

Q12. Use the graph mode to graph the cross-references from sub_10004E79. Which API functions could be called by entering this function? Based on the API functions alone, what could you rename this function?

Q13. How many Windows API functions does DllMain call directly? How many at a depth of 2?

Q14. At 0x10001358, there is a call to Sleep (an API function that takes one parameter containing the number of milliseconds to sleep). Looking backward through the code, how long will the program sleep if this code executes?

Q15. At 0x10001701 is a call to socket. What are the three parameters?

Q16. Using the MSDN page for socket and the named symbolic constants functionality in IDA Pro, can you make the parameters more meaningful? What are the parameters after you apply changes?

Q17. Search for usage of the in instruction (opcode 0xED). This instruction is used with a magic string VMXh to perform VMware detection. Is that in use in this malware? Using the cross-references to the function that executes the in instruction, is there further evidence of VMware detection?

Q18. Jump your cursor to 0x1001D988. What do you find?

Q19. If you have the IDA Python plug-in installed (included with the commercial version of IDA Pro), run Lab05-01.py, an IDA Pro Python script provided with the malware for this book. (Make sure the cursor is at 0x1001D988.) What happens after you run the script?

Q20. With the cursor in the same location, how do you turn this data into a single ASCII string?

Q21. Open the script with a text editor. How does it work?

Answers:

1: DllMain is at 0x1000D02E

As shown in Figure 1 below. We load the malicious file Lab05-01.dll, and then go to **Options** – **General** - **and** check **Line Prefixes.**

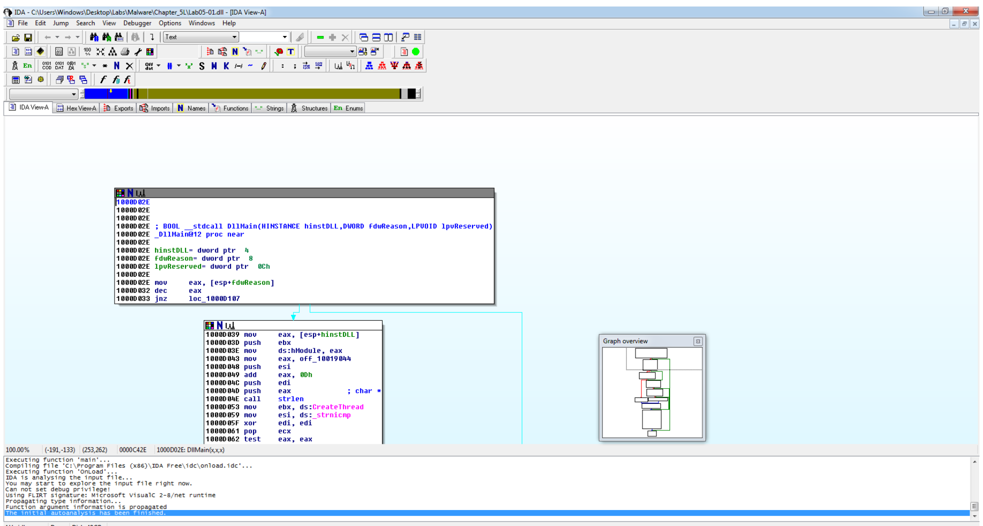

Figure 1: DllMain

2: *gethostbyname* is located at 0x1000D02E

View the imports using **View** – **Open Subviews** – **Import** or by clicking on the Imports tab! Browsing through the imports we can see that *gethostbyname*. Double-clicking it to see it in the disassembly. The *gethostbyname* import is located at 0x100163CC of the .idata section as shown in Figure 2 below.

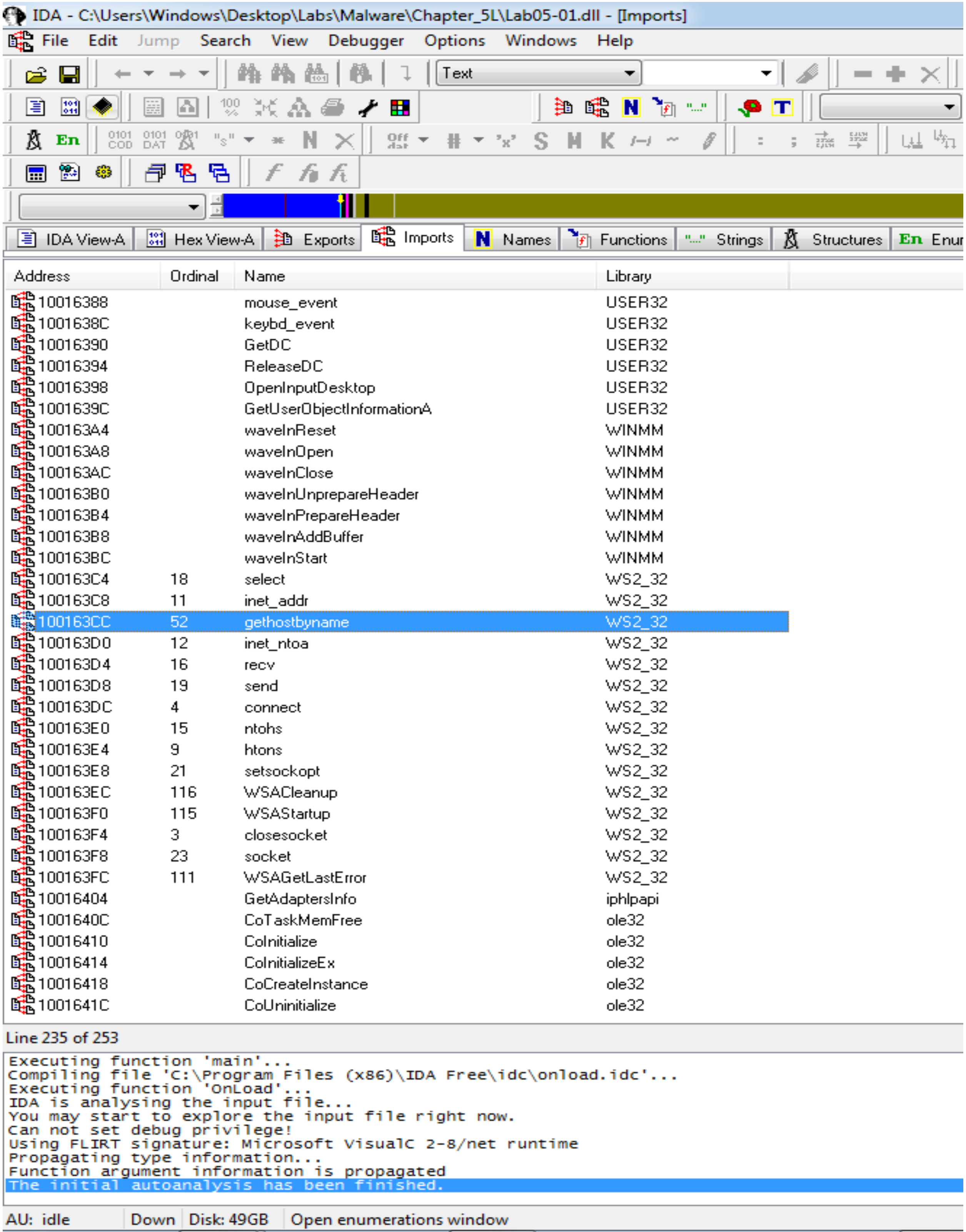

Figure 2: *gethostbyname* location

3: *gethostbyname* import is called nine time by five different functions.
Double-clicking on *gethostbyname* of figure 2 above shows the number of functions that call *gethostbyname* as shown in figure 3 below:

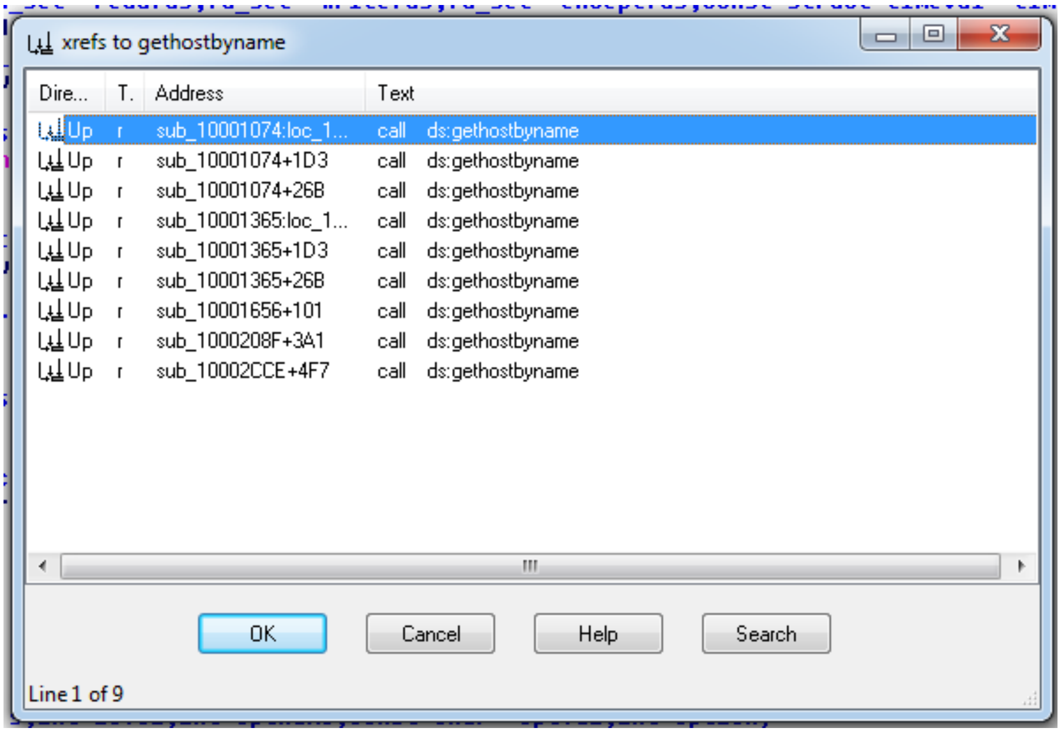

Figure 3: *gethostbyname* import calls

4: A DNS request for pics.practicalmalware analysis will be made as shown in Figure 4B below:
By navigating and click on the **IDAViewA** tab and pressing **G** keyboard to search for 0x10001757 (as referenced in the question). We can see the code which calls *gethostbyname* as shown in Figure 4A:

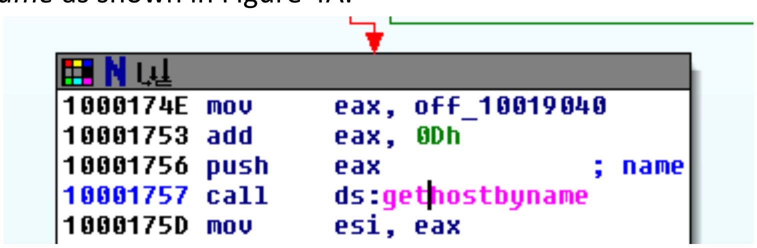

Figure 4A: *gethostbyname* call

Double clicking on *gethostbyname* of Figure 4A above shows the DNS request that will be made as shown in Figure 4B, which is **"[This is RDO]pics.practicalmalwareanalysis.com"**

```
.data:10019040 off_10019040    dd offset aThisIsRdoPics_
.data:10019040                                         ; DATA XREF: sub_10001656:loc_10001722↑r
.data:10019040                                         ; sub_10001656+F8↑r ...
.data:10019040                                         ; "[This is RDO]pics.praticalmalwareanalys"...
.data:10019044 off_10019044    dd offset aThisIsRur    ; DATA XREF: sub_10001074+59↑r
```

Figure 4B: *gethostbyname* DNS request.

5: 20 local variables for the function 0x10001656 were recognized by IDA Pro.

Pressing **G** on the keyboard and navigating to 0x10001656 shows the result shown in Figure 5 below: The result shows the number of variables and parameters for the function. The negative offsets refer to the local variables. Counting all the negative offsets proves that there are 20 local variables.

```
10001656 sub_10001656 proc near
10001656
10001656 var_675= byte ptr -675h
10001656 var_674= dword ptr -674h
10001656 hModule= dword ptr -670h
10001656 timeout= timeval ptr -66Ch
10001656 name= sockaddr ptr -664h
10001656 var_654= word ptr -654h
10001656 in= in_addr ptr -650h
10001656 Parameter= byte ptr -644h
10001656 CommandLine= byte ptr -63Fh
10001656 Data= byte ptr -638h
10001656 var_544= dword ptr -544h
10001656 var_50C= dword ptr -50Ch
10001656 var_500= dword ptr -500h
10001656 var_4FC= dword ptr -4FCh
10001656 readfds= fd_set ptr -4BCh
10001656 phkResult= HKEY__ ptr -3B8h
10001656 var_3B0= dword ptr -3B0h
10001656 var_1A4= dword ptr -1A4h
10001656 var_194= dword ptr -194h
10001656 WSAData= WSAData ptr -190h
```

Figure 5: 0x10001656 local variables

6: 1 parameter for the function 0x10001656 is recognized by IDA Pro.

As explained in 5 above, the negative offset represent the parameter of the function. As shown below:

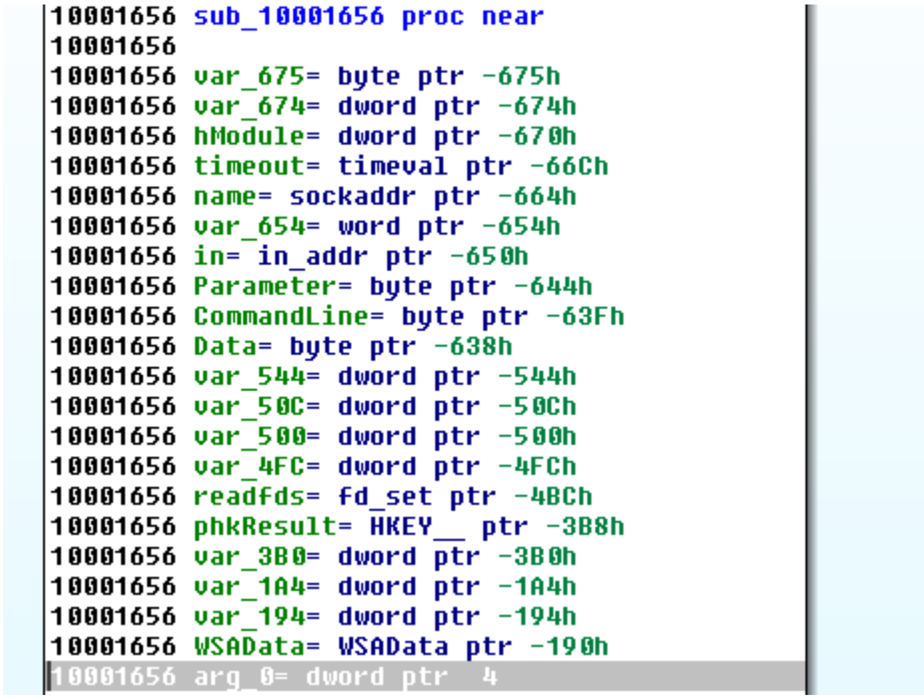

Fig 6: 0x10001656 parameter.

Q7: The string \cmd.exe/c is located at 0x10095B34.

To view the string \cmd.exe/c we begin by viewing the complete strings by clicking on **View – Open Subviews – Strings** or clicking on the **string** tab. doing so shows the result shown in Figure 7A below. Double-clicking *\cmd.exe/ c* reveals its location which is *xdoors_d* section of the PE file at 0x10095B34 as shown in figure 7B below:

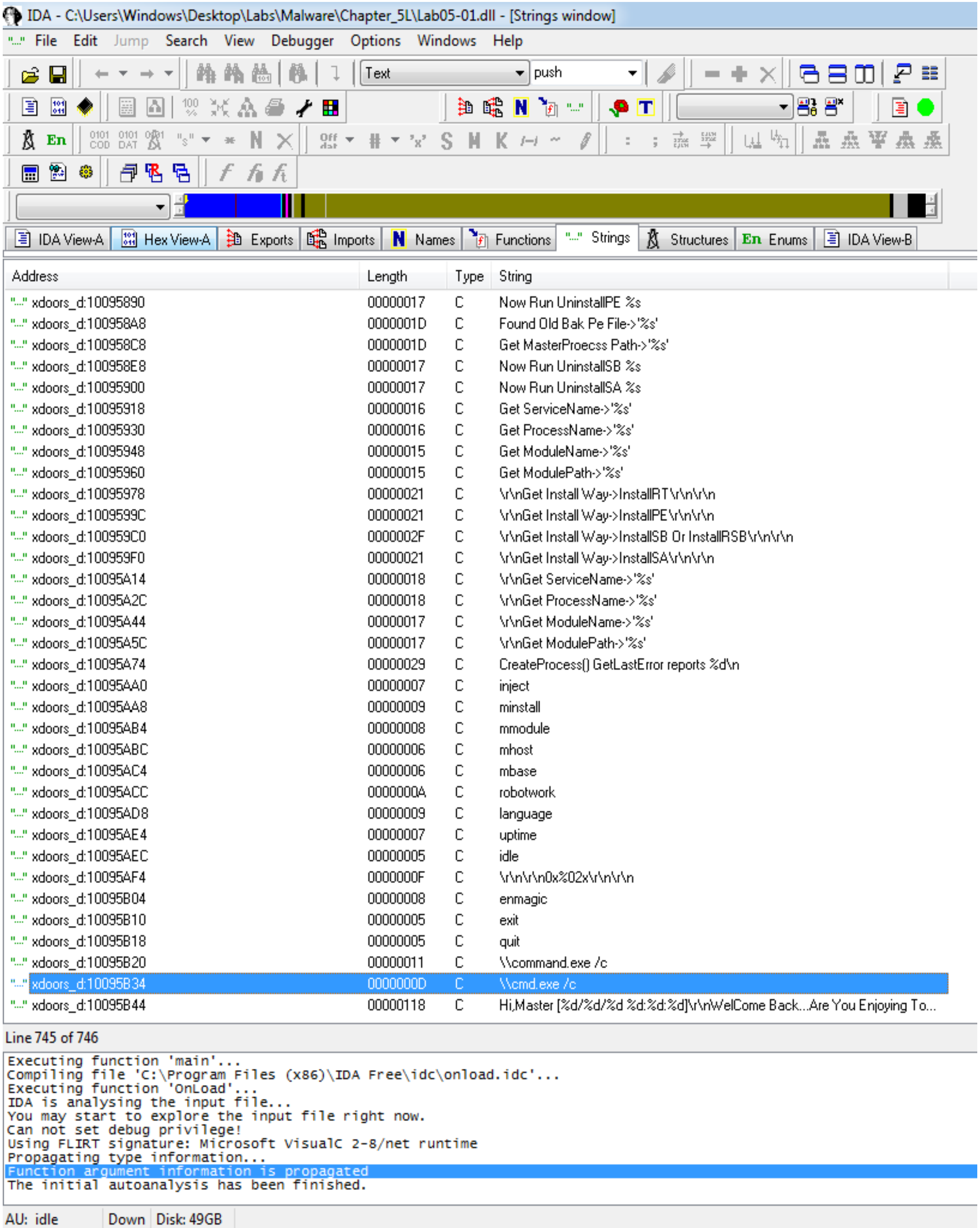

Figure 7A: Lab05-01.dll Strings

```
xdoors_d:10095B34 aCmd_exeC       db '\cmd.exe /c ',0      ; DATA XREF: sub_1000FF58+278↑o
```

Figure 7B: \cmd.exe/c location.

8: The area code that referenced \cmd.exe/c appears to be making calls and moving sessions for the attacker as shown in Figure 8 below:
This could be achieved by pressing **G** on the keyboard and narrowing down the search.

```
10001673 call    sub_10003695
10001678 mov     dword_1008E5C4, eax
1000167D call    sub_100036C3
```

Figure 8: \cmd.exe/ c area code.

9: Operating System version information is stored in the dword_1008E5C4
Using *dword_1008E5c4* to cross-reference. We press **G** and look at sub_10003695 by double-clicking it and looking at the disassembly. We can see that the sub_10003695 function contains a call to *GetVersionEx* as shown in figure 9 below:

```
100036AF call    ds:GetVersionExA ; Get extended information about the
100036AF                          ; version of the operating system
100036B5 xor     eax, eax
100036B7 cmp     [ebp+VersionInformation.dwPlatformId], 2
100036BE setz    al
```

Figure 9: *sub_10003695* function

As seen above, this function contains information about the operating system.

10: Referenced to the question: subroutine at 0x1000FF58 contains strings such as *memcp* and *robotwork* as shown in Figure 10A below:

```
100052A2
100052A2
100052A2 ; Attributes: bp-based frame
100052A2
100052A2 ; int __cdecl sub_100052A2(SOCKET s)
100052A2 sub_100052A2 proc near
100052A2
100052A2 var_60C= dword ptr -60Ch
100052A2 Data= byte ptr -20Ch
100052A2 cbData= dword ptr -0Ch
100052A2 Type= dword ptr -8
100052A2 hKey= dword ptr -4
100052A2 s= dword ptr  8
100052A2
100052A2 push    ebp
100052A3 mov     ebp, esp
100052A5 sub     esp, 60Ch
100052AB and     byte ptr [ebp+var_60C], 0
100052B2 push    edi
100052B3 mov     ecx, 0FFh
100052B8 xor     eax, eax
100052BA lea     edi, [ebp+var_60C+1]
100052C0 and     [ebp+Data], 0
100052C7 rep stosd
100052C9 stosw
100052CB stosb
100052CC push    7Fh
100052CE xor     eax, eax
100052D0 pop     ecx
100052D1 lea     edi, [ebp-20Bh]
100052D7 rep stosd
100052D9 stosw
100052DB stosb
100052DC lea     eax, [ebp+hKey]
100052DF push    eax             ; phkResult
100052E0 push    0F003Fh         ; samDesired
100052E5 push    0               ; ulOptions
100052E7 push    offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVersi"...
100052EC push    80000002h       ; hKey
100052F1 call    ds:RegOpenKeyExA
100052F7 test    eax, eax
100052F9 jz      short loc_10005309
```

Figure 10A: 0x1000FF58 strings.

```
10010444
10010444 loc_10010444:           ; size_t
10010444 push    9
10010446 lea     eax, [ebp+var_5C0]
1001044C push    offset aRobotwork ; "robotwork"
10010451 push    eax             ; void *
10010452 call    memcmp
10010457 add     esp, 0Ch
1001045A test    eax, eax
1001045C jnz     short loc_10010468
```

Figure 10B: 0x1000FF58 strings

Taking a close look at figure 10B, we can see that a call at push will be made. Examining sub_100052A2, we can see that it queries the registry at SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime as shown in10A above.

11: The PSLIST finds a process name and information over a network.
To do this, we view the exports as shown in Figure 11A below:

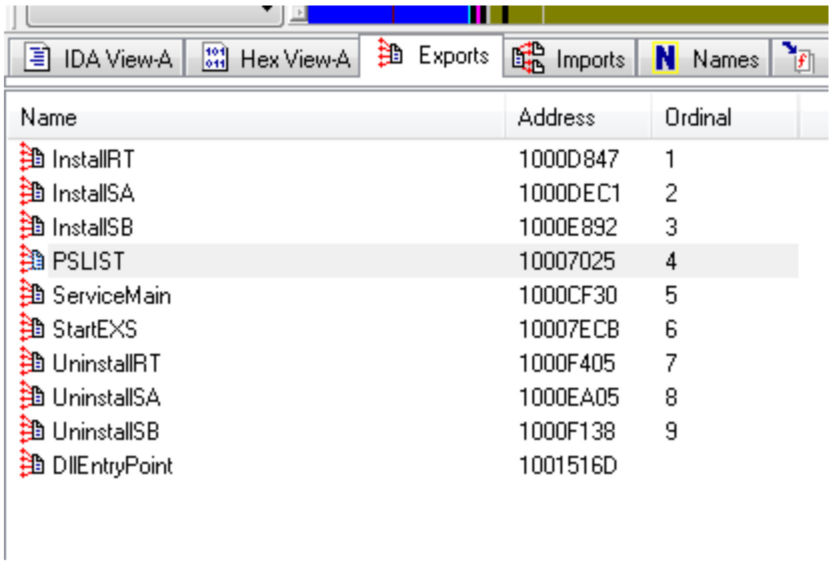

Figure 11A: Exports.

Looking at PSLIST and double-clicking it shows figure 11B:

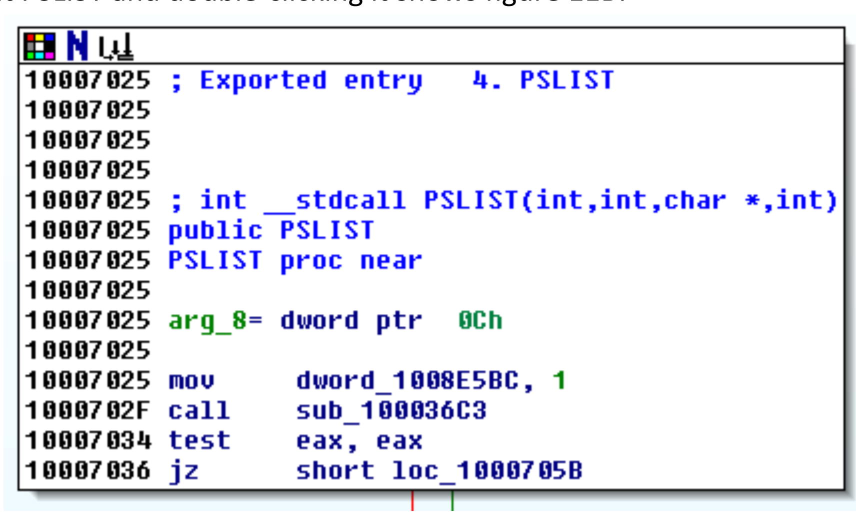

Figure 11B: PSLIST.

The function appears to take two paths as shown in Figure 11C below. Depending on each path, it checks to see the OS version.

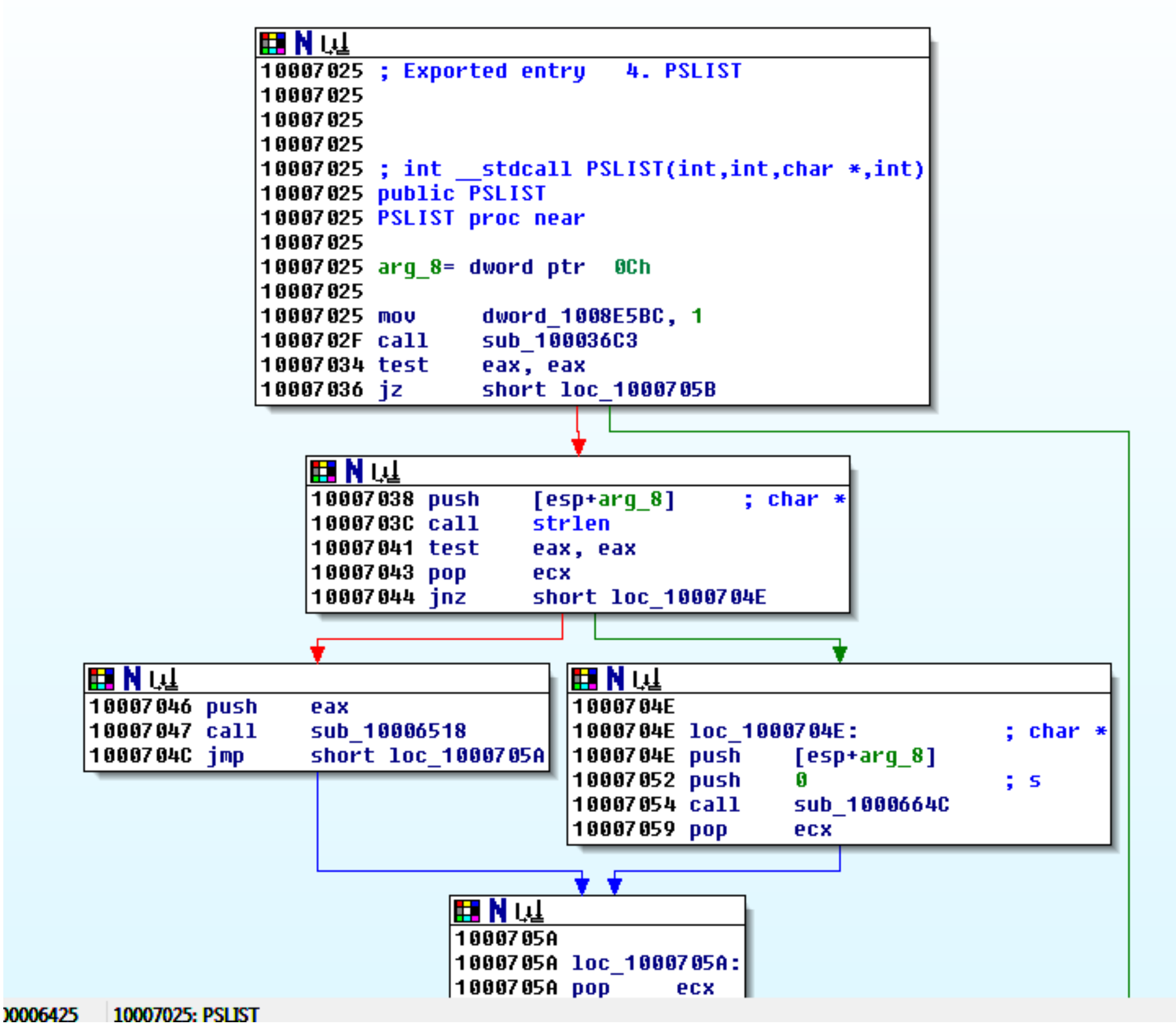

Figure 11C: PSLIST paths

12: *GetSystemDefaultLangID* and *send.* We can rename the function or SystemLang or GetLang or any meaningful thing.

To do this, we view the graph mode y **View** – **Graphs** – **Xrefs from**. Cross referencing sub_10004E79 by pressing **G** on the keyboard, we see the graph as shown in Figure 12 below.

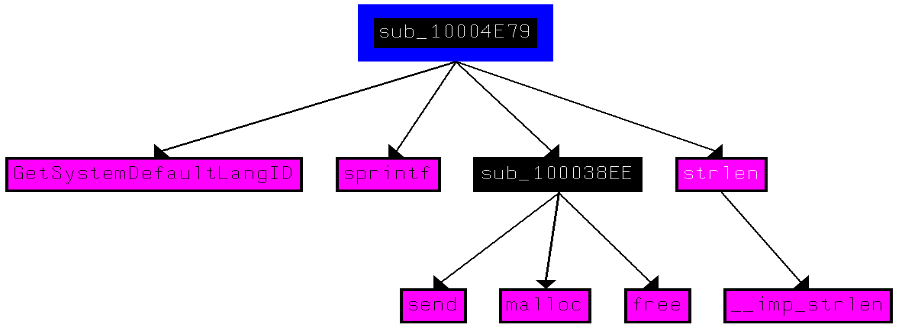

Figure 12: sub_10004E79 graph view.

13: As shown in Figure 13B. DllMain calls for API functions directly, and these are: *createThread, Strncpy, Strlen,* and *_strnicmp.* At a depth of 2 it calls a variety of API's, which shows a very large graph.

To view the DllMain calls for API function, we view the graph following the same guideline outlined in Question 12. But in this case we set a custom cross-reference graph, by using the settings as shown in Figure 13A below:

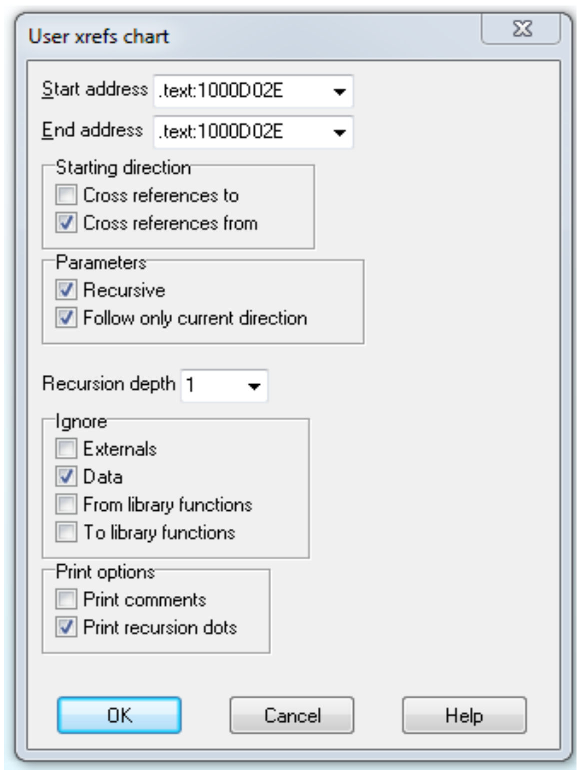

Figure 13A: Graph Setting

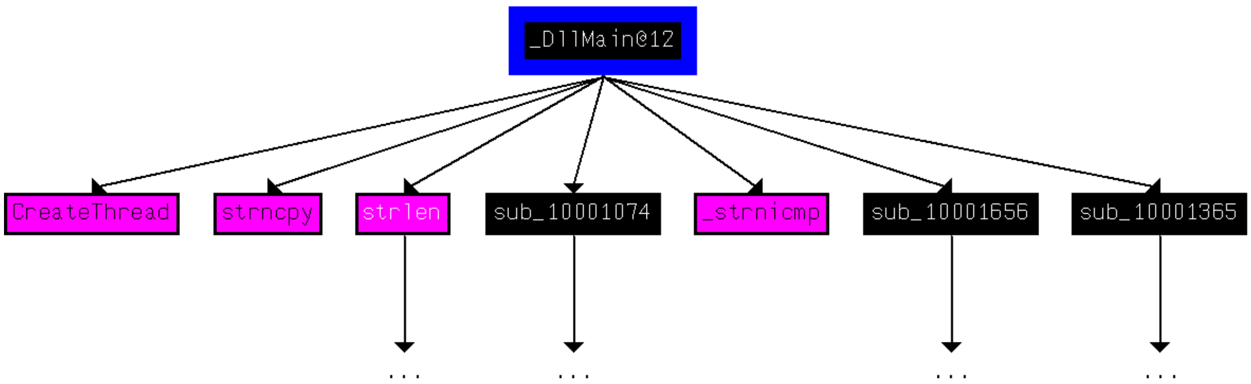

Figure 13B: _DllMain Graph Setting.

14: Approximately 30 seconds.

Referenced in question 14, there us a call to sleep at 0x10001358 as shown in Figure 14 below:

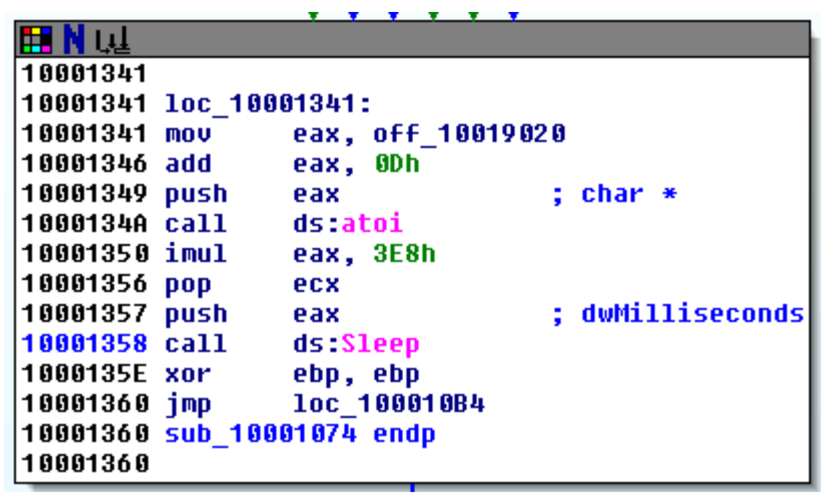

Figure 14: Call to sleep at 0x10001358.

This can be viewed by cross-referencing and pressing G on the keyboard to find 0x10001358. To know for how long the program will sleep if we execute the program, we 30 x 1000 = (30,000 milliseconds) or 30 seconds. Were 30 is the string number multiple by 1000.

15: The 3 parameters at the call to socket of 0x10001701 are 6,1, and 2 as shown in figure 15 below.

This can be viewed by pressing **G** on the keyboard and cross-referencing 0x10001701.

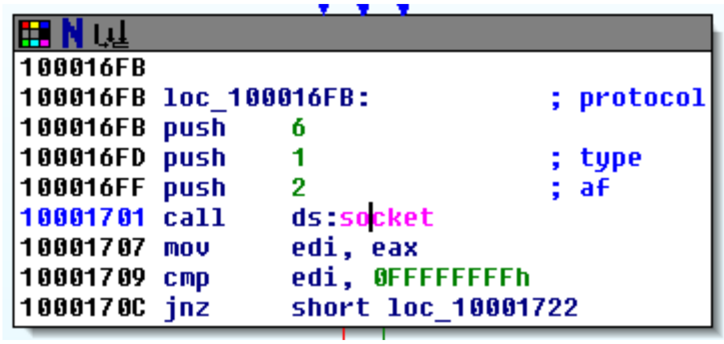

Figure 15: Parameter of a call socket.

16: The arguments correspond to IPPROTO_TCP, SOCK_STREAM, and AF_INET. Right-clicking on each number of the result shown in Fig 15 above reveals what the argument corresponds to.

17: Yes! The string 564D5868h which shows that the string found Virtual Machine in the caller function as shown in Figure 17B below: To find this, we search for the **in** instruction by selecting **Search – Text** and entering **in** and checking **Find All Occurrences** in the search dialog. As shown in Figure 17A, the result which is shown in Figure 17B is the basis of the result decision.

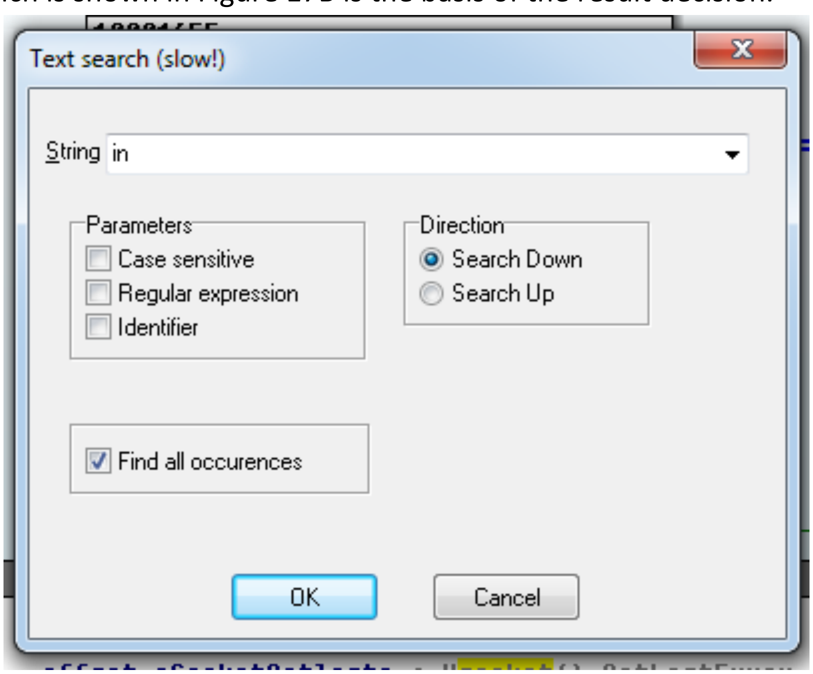

Figure 17A: Text Search

```
.text:100061C7        mov    eax, 564D5868h
.text:100061DB        in     eax, dx
```

Figure 17B: “**in”** search result.

18: At 0x1001D988 nothing appears to happen but a random data which doesn’t makes much sense as shown in Figure 18 below:
To find this, we press **G** on the keyboard and cross-reference with 0x1001D988.

```
.data:1001D987                db    0
.data:1001D988                db  2Dh ; -
.data:1001D989                db  31h ; 1
```

Figure 18: Cross-referencing 0x1001D988

Question 19, 20, and 21. Cannot be answered, IDA Pro responds with an error when *lab05-01.py* is loaded. This could be due to IDA Pro free version been used, and not including the Python plug-in installed.

Conclusion

This lab aims to provide a dynamic analysis of a malware using an advanced and sophisticated tool known as IDA Pro. The lab provides answers to what malware imports and strings are, it also discusses the basic characteristics of the malware.