Assignment 5

Justin Kennelly

University Of Arizona

CYBV 454 MALWARE THREATS & ANALYSIS

Professor Galde

March 1, 2022

# LAB 5-1

File Examined: Lab05-01.dll

## LAB 5-1 Question 1

## What is the address of DllMain?

Text

Description automatically generated with low confidence

*Figure 1: IDA Pro after opening Lab05-01.dll*

Using IDA Pro ver. 5.0 to open Lab05-01.dll, IDA Pro analyzes the binary and the text view starts at DllMain. The address found for DllMain is found at 0x1000D02E.Using the space bar to switch from the graph view to text view shows more clearly the address of the function seen highlighted in yellow in Figure 1.

## LAB 5-1 Question 2

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

Table

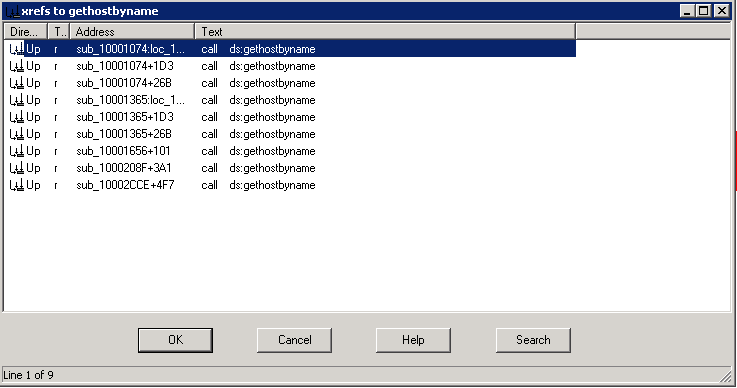
Description automatically generated

*Figure 2: Imports View of Lab05-01.dll in IDA Pro*

I was able to click in the left pane within the names window. Here I went to the tool bar and clicked search. I searched for gethostbyname and the search function quickly found the import I was looking for. The import is located at the address 0x0100163CC as seen in Figure 2.

## LAB 5-1 Question 3

## How many functions call gethostbyname?



*Figure 3: Viewing cross references of the gethostbyname import*

A picture containing table

Description automatically generated

*Figure 4: Highlighted addressing of the gethostbyname import in Xres*

A picture containing timeline

Description automatically generated

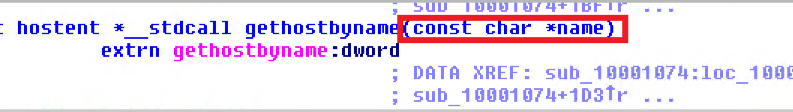
*Figure 5: Cross refs graph of gethostbyname*

Once I located the import, gethostbyname, I double clicked the import within the Names Window to open it in the text view in the main window. Here I right-clicked on the name of the import and selected “jump to xref to operand”. Figure 3 shows the result of that operand and the window that is displayed. Here I can see that in total there are 9 calls to gethostbyname, however, by taking a closer looked I could see that several of these calls were made by functions at the same address. In Figure 4, you can see that the first three calls are from the function at sub\_10001074. Also the next three calls are all made from sub\_10001365. After that the calls are unique. I opened a cross reference graph for the import to get a better understanding as well (Figure 5). So it appears that although there are 9 total calls by functions, only 5 functions are unique, therefore, five functions make calls to gethostbyname.

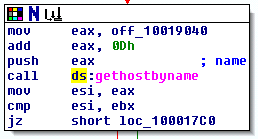
## LAB 5-1 Question 4

## Focusing on the call to gethostbyname located at 0x10001757, can you figure out which DNS

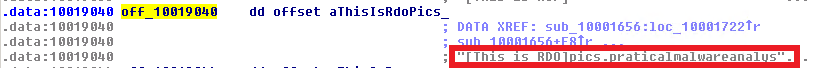
## request will be made?



*Figure 6: Text view of the gethostbyname*



*Figure 7: Graph view of gethostbyname at address 0x010001757*

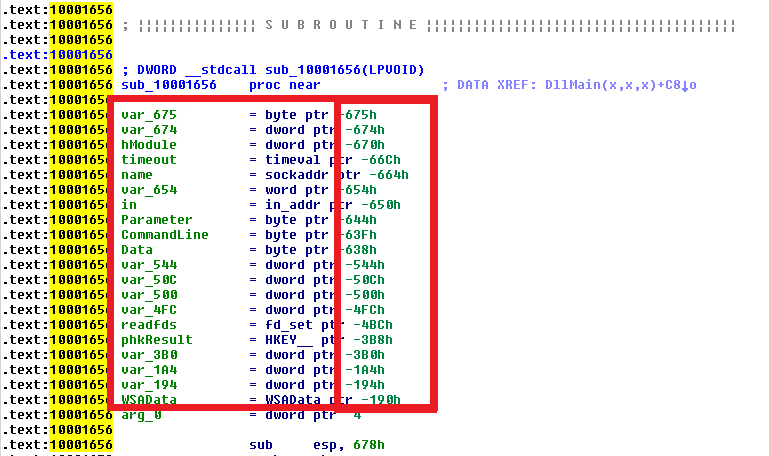


*Figure 8: Text view of off\_10019040*

I went to the text view of the binary and pressed G to jump to the address 0x01001757. Here I could see that gethostbyname takes a parameter as a name pointer, Figure 6. I highlighted gethostbyname and pressed space to switch to the graph view. Here I could see more of the operations that were happening at the specified address. Figure 7 shows the graph node where gethostbyname is called at the address 0x01001757. We can see that we are moving off\_10019040 into the EAX. Then EAX gets extra bytes added and finally EAX is then pushed as a parameter to gethostbyname (Figure 7). Since I already discovered the parameter existed for gethostbyname I then determined that I should work backwards and look at off\_10019040. Double-clicking on off\_10019040 took me straight to the text view. I could see at the address that the offset contains the data “[This is RDO]pics.practicalmalwareanalysis” (Figure 8). The DNS request being made is to the URL pics.practicalmalwareanalysis.

## LAB 5-1 Question 5

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



*Figure 9: Text view of the address pointing to the subroutine 0x10001656*

Given the address, I used G again to jump to the address 0x10001656. I took a look at the text for the subroutine seen in Figure 9. As I have learned the negative offsets point to the variables and the positive offsets point to the arguments used here in Figure 9. Therefore, in the subroutine there are 20 local variables.

## LAB 5-1 Question 6

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

Text

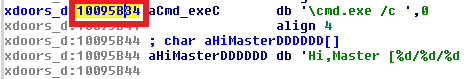
Description automatically generated

*Figure 10: Graph node of the subroutine*

I was already looking at the subroutine at the address 0x1000656 from the previous question and had already determined how the offsets signify what is a variable and what is an argument. In Figure 10 the argument is highlighted where the positive offset is assigned. There is only one parameter passed to this subroutine.

## LAB 5-1 Question 7

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

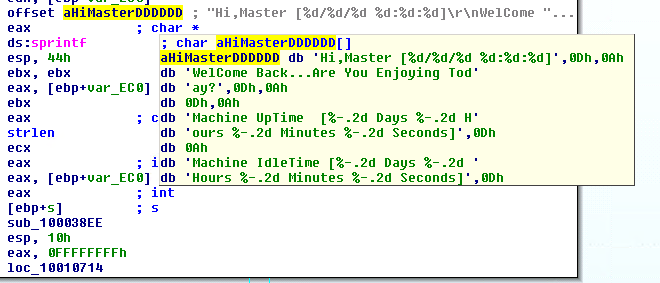


*Figure 11: Text view of the string found at address 0x10095B34*

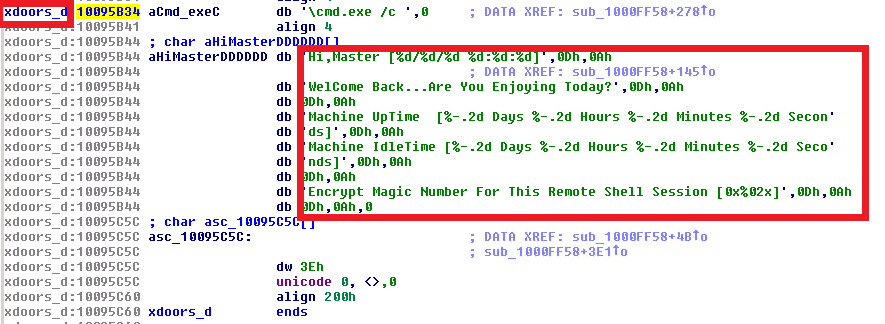
To find the specific string, “\cmd.exe /c” I simply clicked in the strings windowpane on the bottom right side of the IDA Pro dashboard. Once selected I clicked in the tool bar under search and searched the exact string, \cmd.exe /c. This took me directly to the string. I was then able to double click the string to view in in text view within the main window (Figure 11). Here you can see that the string is located at the address 0x10095B34.

## LAB 5-1 Question 8

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



*Figure 12: Text view of the addressing near the cmd.exe string*

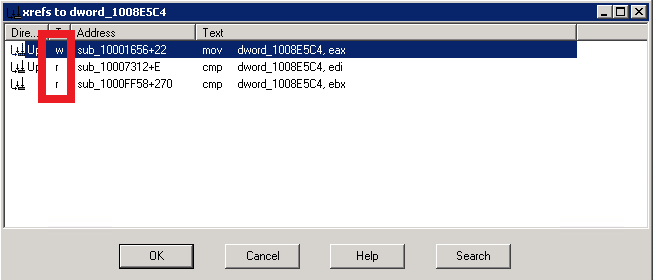


*Figure 13: Additional text view of the addressing near the cmd.exe string*

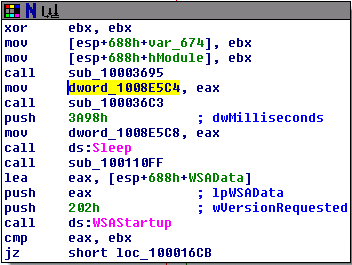
Just under the string we found in question 7, I was able to look at the surrounding code. I noticed some things right away such as the strings that are printed. Strings like, “Hi Master”, “Machine UpTime”, “Machine IdleTime” (Figure 12). Additionally, as seen in Figure 13, I can see that these strings are all within the section titled, xdoors\_d and the last set of strings mention a encrypting a magic number for a remote shell session (Figure 13). This gives me a pretty good idea that this is put in the malware to access the target machine by an attacker using a remote shell.

## LAB 5-1 Question 9

## 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.)



*Figure 14: Cross ref view of dword at the address 0x1008E5C4*



*Figure 15: Graph node view of dword*

Text, timeline

Description automatically generated with medium confidence

*Figure 16: Graph node view of subroutine at the address 0x100036C3*

*Graphical user interface, text, application

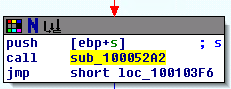
Description automatically generated*

*Figure 17: Graph node view of subroutine at the address 0x10003695*

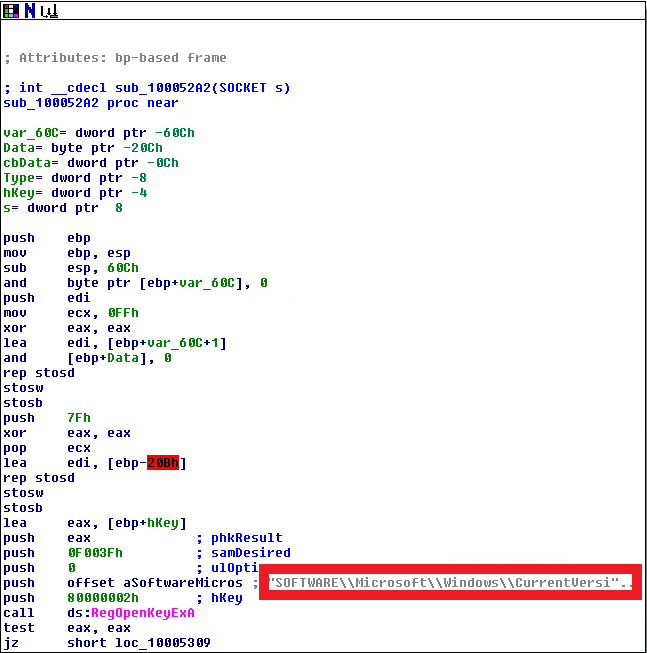
I used G to jump to dword\_1008E5C4. From there I right-clicked and selected the xref operand. In Figure 14, we can see that there are three references to this dword. Two of which are read only and one is write (Figure 14). Since we are determining how the dword is set I decided that looking at the address relating to the write operand made more sense. Double-clicking on that address to me straight to the graph node view seen in Figure 15. Here I notice that EAX is moved into dword and then it is used as a parameter in the next call for sub\_100036C3 (Figure 16). Taking a step back though I realized the subroutine prior to dword, sub\_10003695 is actually called and returned with a value (Figure 17). I believe the value returned from sub\_1003695 is on the EAX register and then moved onto the dword where it is then passed to sub\_100036C3. Both subroutines call the function GetVersionEXA. In sub\_1003695 the operating system is determined with the GetVersionEXA function and then return where it is set to dword and then passed to sub\_10036C3 where it will determine what path it will take. The dword global variable will either be set to 1 or it will return its’ set value.

## LAB 5-1 Question 10

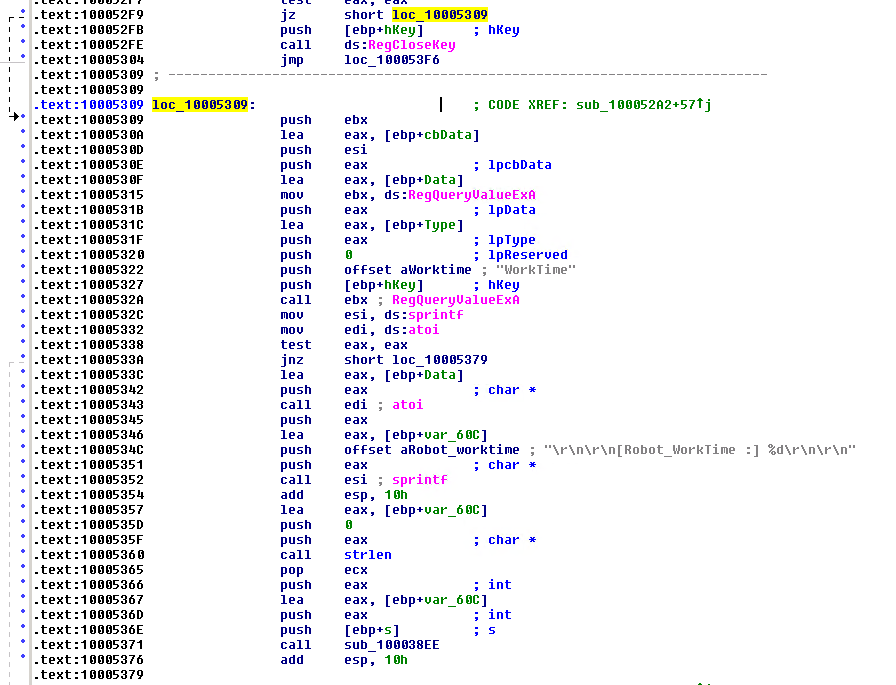
## A few hundred lines into the subroutine at 0x1000FF58, a series of comparisons use memcmp to compare strings. What happens if the string comparison to robotwork is successful (when memcmp returns 0)?



*Figure 18:*



*Figure 19: Text view of subroutine 10052A2*



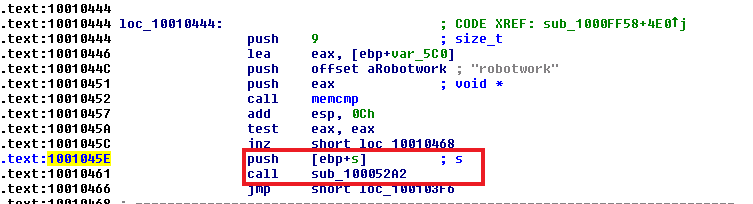
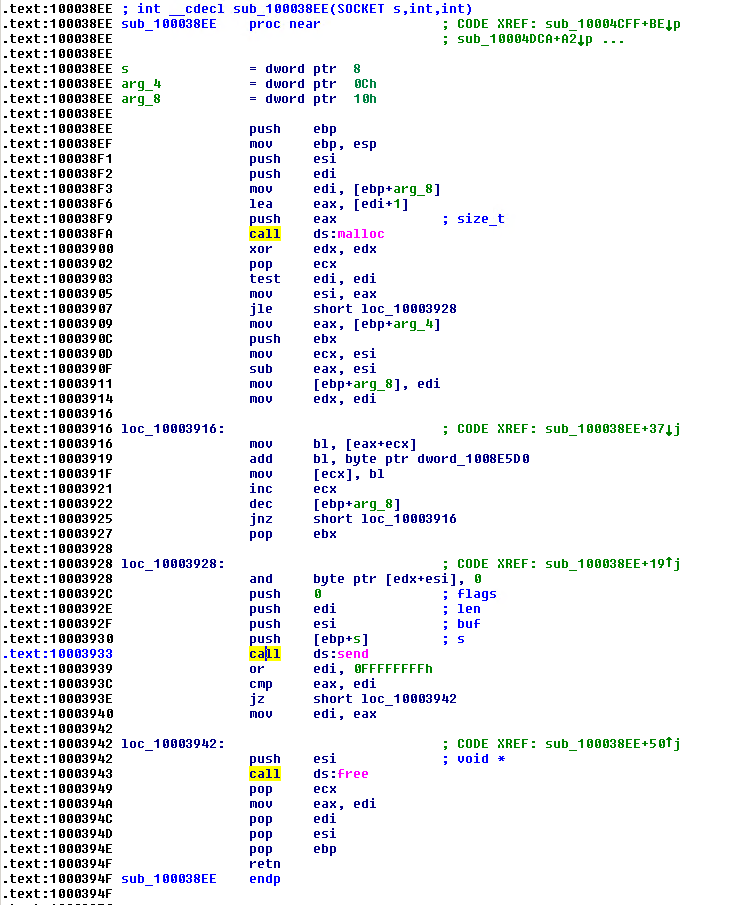


Figure 20: Text view of the subroutine section for calling sub\_100052A2

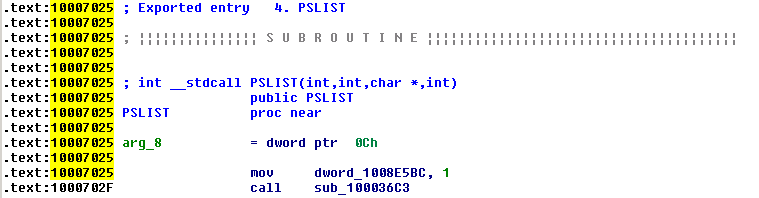


*Figure 21: Text view of the socket function*

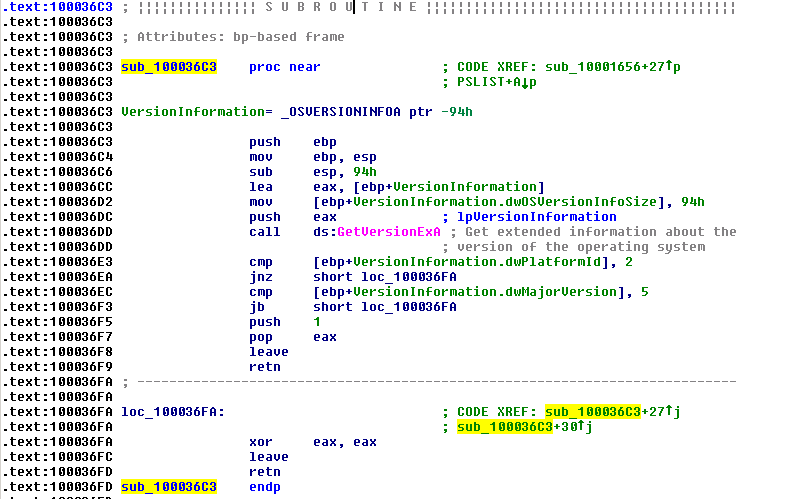
I used G to jump to the address 0x1000FF58. From the I switched to the graph view and followed the process until I found the ‘memcpy robotwork’ node. Looking at the jnz command I notice that if it does not equal zero it will route to the subroutine 0x100052A2 seen in Figure 20. Looking at the subroutine we see that a registry is called at HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime. Going back to the text view I see that the subroutine is a socket function (Figure 21).Therefore I have determined that the registry information is pushed in Figure 20 and called in Figure 21.

## LAB 5-1 Question 11

## What does the export PSLIST do?



*Figure 22: Export text view of PLIST*

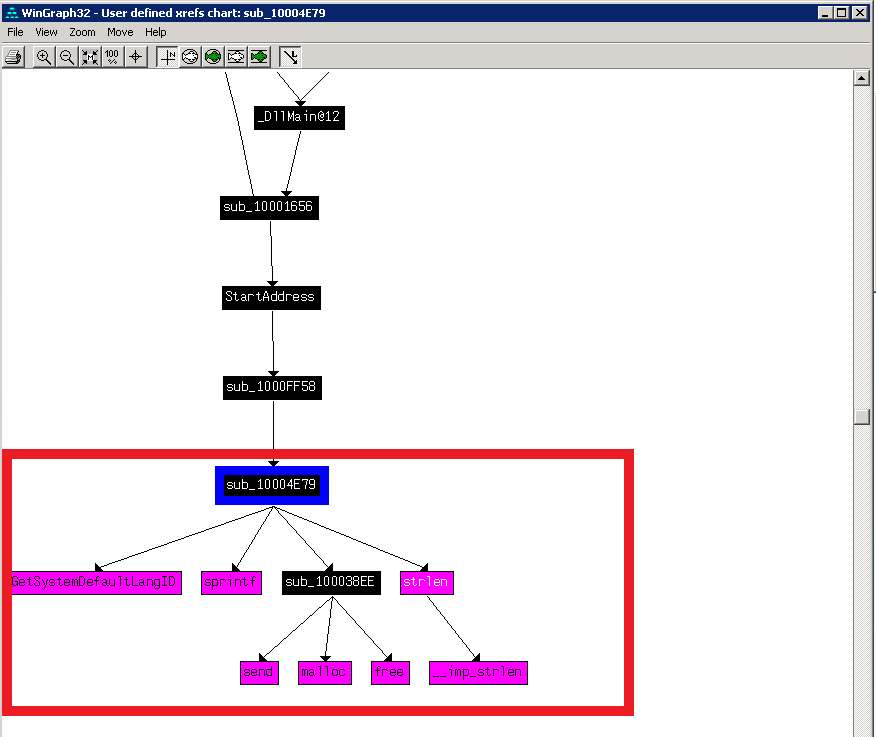


*Figure 22: Text view of sub\_100036C3*

I was able to find PLIST by selecting the exports window. Once I located it I double clicked on PLIST to view the export. A subroutine is called at address 0x100036C3 within PLIST function (Figure 22). Double clicking on the subroutine takes me to the text view (Figure 22). Here I see that we are checking the OS of the system once again. Depending on the OS PLIST will determine which path it will take.

## LAB 5-1 Question 12

## 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?

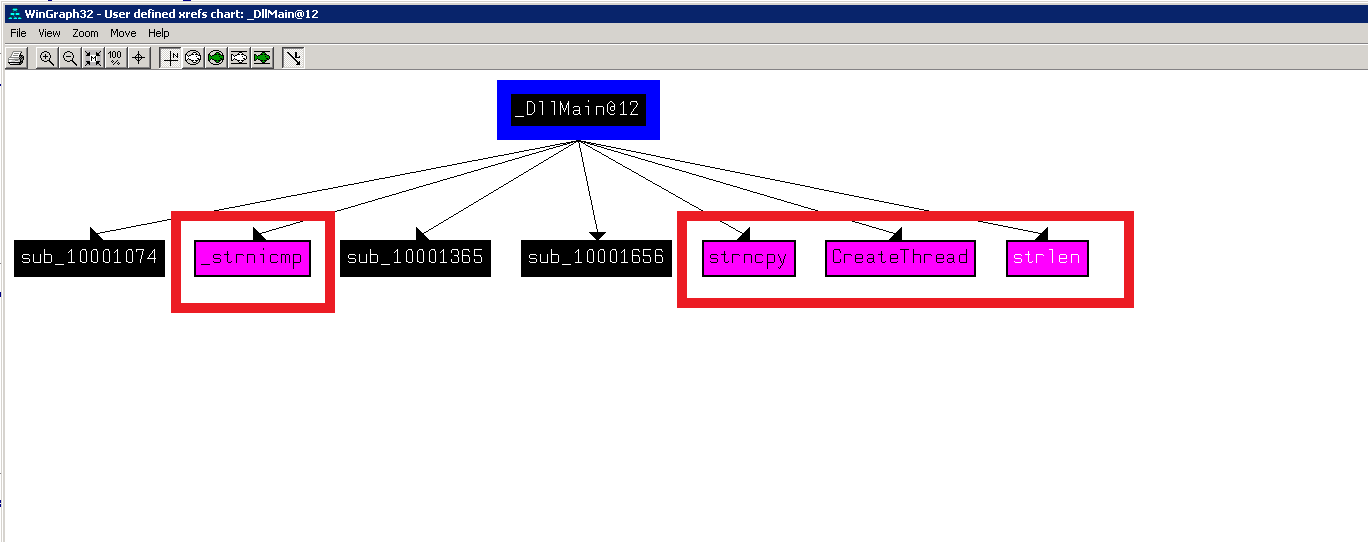


*Figure 23: Graph view of sub\_10004E79*

I used G to jump to the address 0x10004E79. From there I used the view tab to take a look at the API functions of the subroutine. There were seven functions: GetSystemDefaultLangID, Sprintf, strlen, send, malloc, free, and imp\_strlen. (Figure 23).

## LAB 5-1 Question 13

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

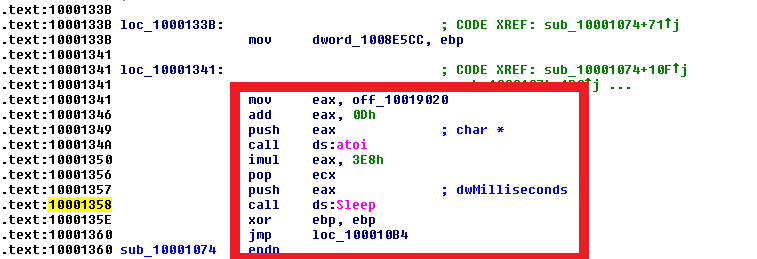


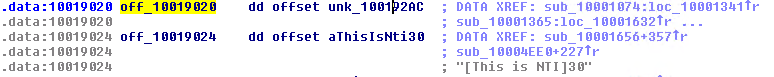
*Figure 24: Graph view of DllMain*

DllMain appears to call four API functions at the second depth (Figure 24). I was able to find this out by using G to jump to the address of DllMain found in question 1. Then I went to the view tab and opened a xrefs graph and selecting certain parameters. I set the depth to 1, and made it only track cross references from DllMain. I didn’t set the depth to 2 because I noticed the graph quickly got messy and the additional nodes were not in the scope of what I was looking for.

## LAB 5-1 Question 14

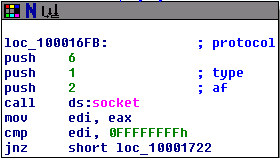
## At 0x10001358, there is a call to Sleep (an API function that takes one parameter containing the number of milliseconds to sleep). Looking backward through the code, how long will the program sleep if this code executes?





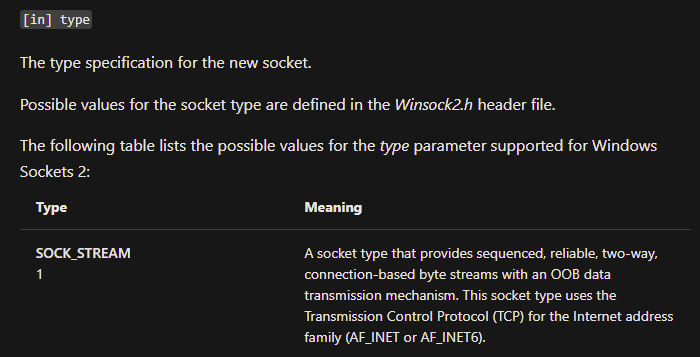
## LAB 5-1 Question 15

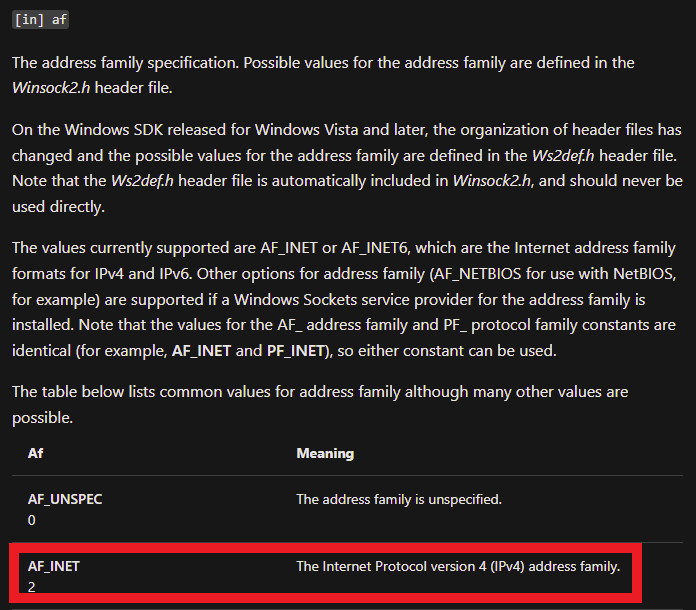
## At 0x10001701 is a call to socket. What are the three parameters?

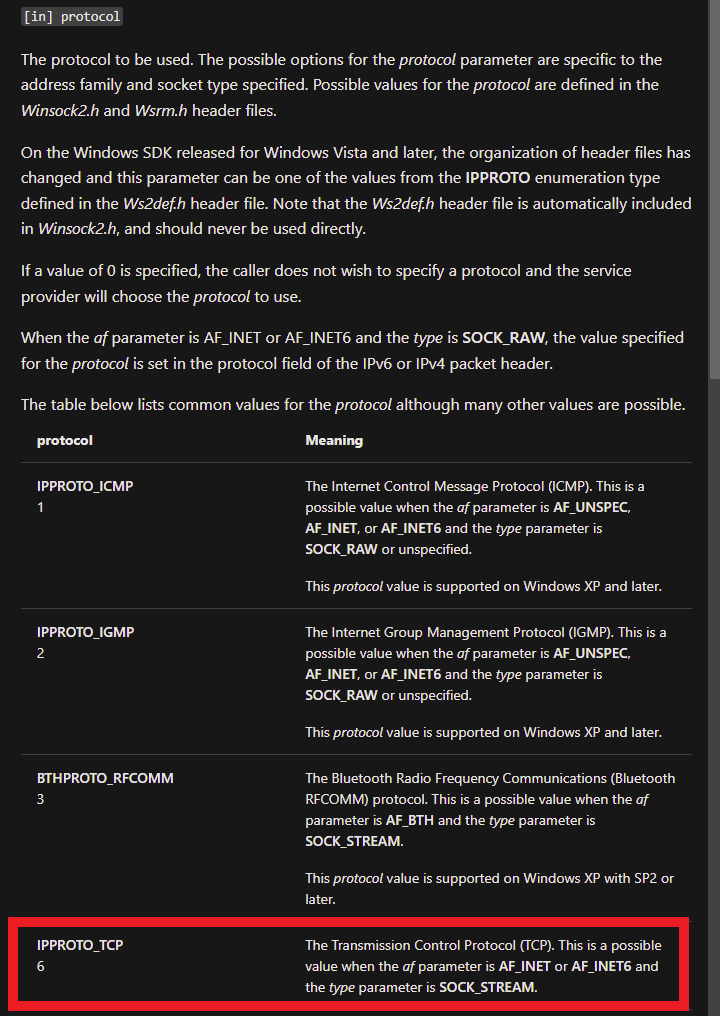


## LAB 5-1 Question 16

## 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?

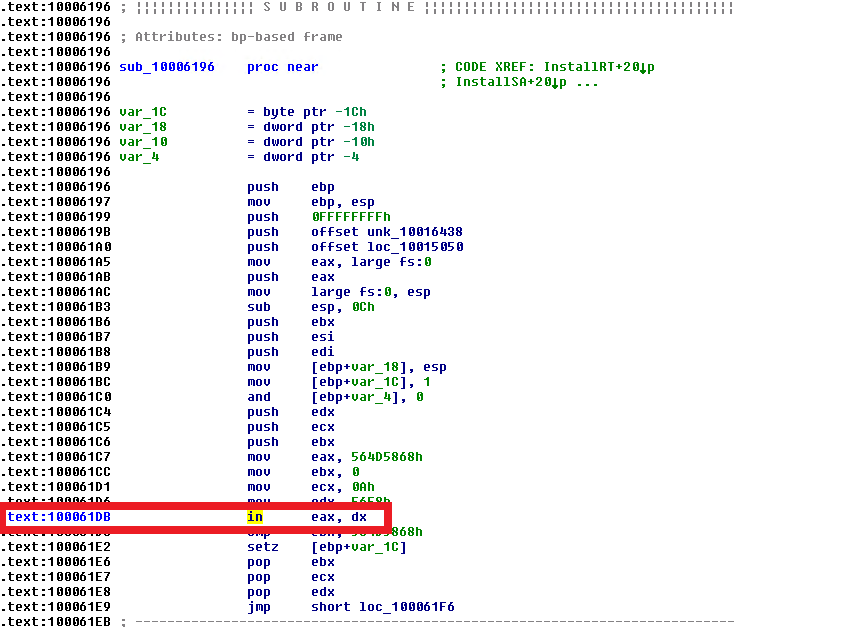






## LAB 5-1 Question 17

## 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?



## LAB 5-1 Question 18

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