Assignment 3

Adam Livingston

University Of Arizona

CYBV 454 MALWARE THREATS & ANALYSIS

Professor Galde

7 Mar 2023

CYBV 454 Assignment 3 LIVINGSTON

LAB 5-1

- Lab05-01.dll: 1a9fd80174aafecd9a52fd908cb82637 (Figure 2)

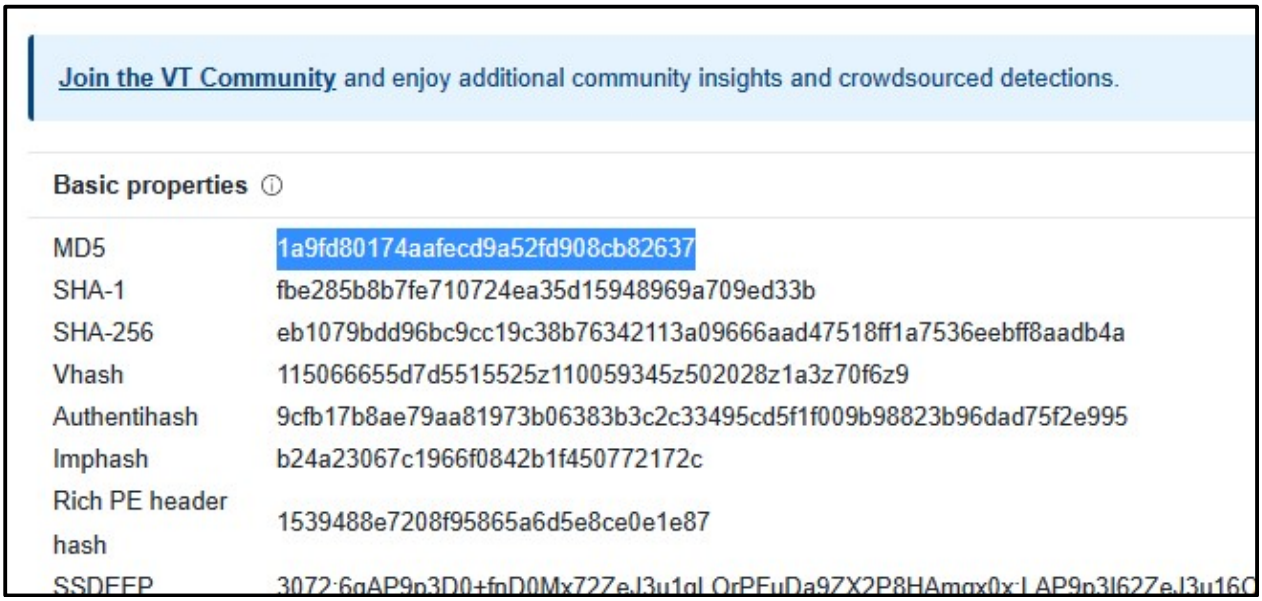


Figure 1: Virus Total MD5 Hash for Lab05-01.dll

Virus Total found 50 matching signatures for backdoor Trojan malware for the file Lab05-01.dll (Figure 3). It has a compilation timestamp of 09 Jun 2008 at 12:49:29 UTC (Figure 2).

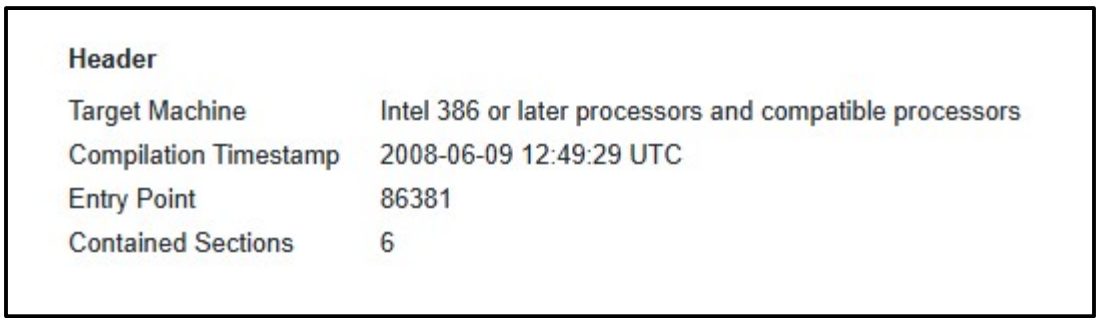


Figure 2: Virus Total Compilation Timestamp for Lab05-01.dll

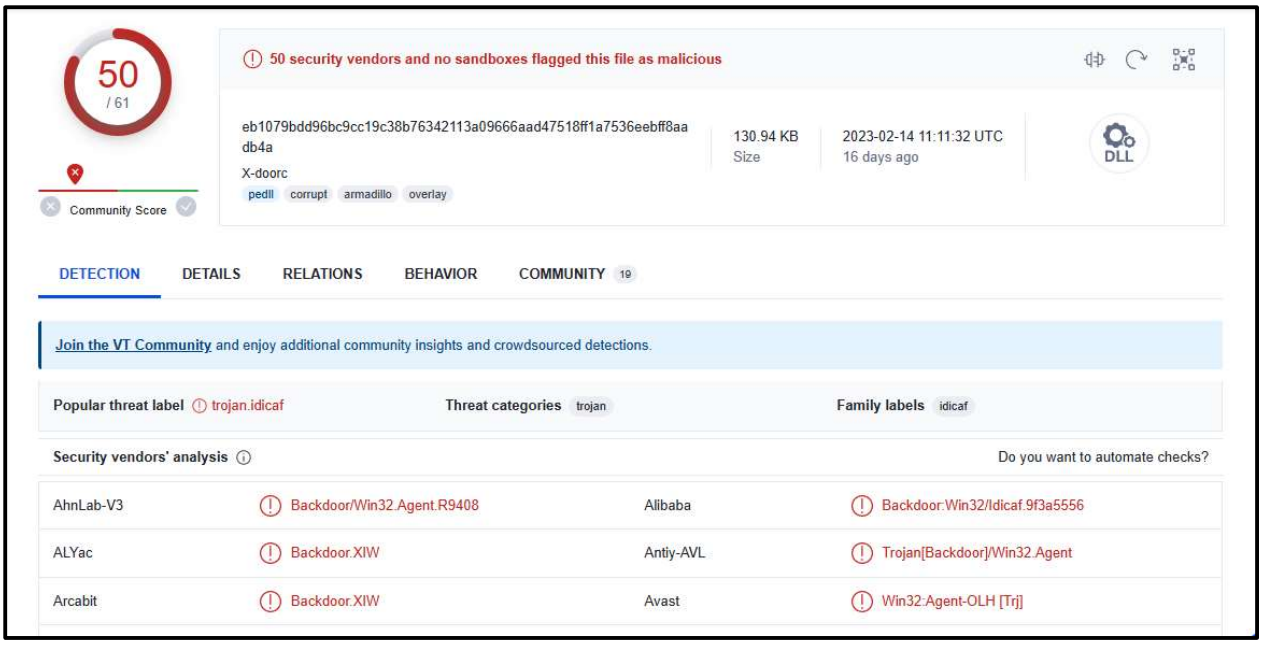


Figure 3: Virus Total Findings for file Lab05-01.dll.

It appears to import many dynamic linked libraries: iphlpapi.dll, winmm.dll, msvfw32.dll, gdi32.dll, kernel32.dll, MSVCRT.dll, oleaut32.dll, advapi32.dll, PSAPI.DLL, WS2_32.dll, ole32.dll, and user32.dll (Figure 4). Immediately, I notice MSVCRT.dll which indicates that this file was written in C++.

Kernel32 indicates that it has the capability to access and modify the core OS functions. user32.dll suggests it will manipulate the user interface in some fashion in conjunction with gdi32.dll, which contains functions for displaying and manipulating graphics. These graphics and UI .dll files are most likely used in conjunction with msvfw32.dll, which is the Microsoft Video for Windows dynamic linked library. advapi32.dll indicates that core Windows components will be altered, such as the Service Manager and Registry. Ws2_32.dll hints that it will perform some network-related tasks, most likely being the connection calling out to establish the backdoor that the security vendors' naming convention suggests. The malware potentially alters the

functionality of Windows programs using ole32.dll and oleaut32.dll. Winmm.dll will allow the file to access essential Windows OS system files and driver functions.

The last two .dll imports are Application Programming Interface (API) dynamic linked libraries: PSAPI.dll and iphlpapi.dll. PSAPI is the Process Status Application Programming Interface that most likely will help the malware obtain information about processes and device drivers. It is possible that if it detects some sort of monitoring software running on the machine, then it will ensure to obfuscate itself. Or it can use the information it collects about device drivers and rewrite the driver registry contents for its own purposes. Then we notice iphlpapi.dll which is the IP Helper API. This import is most likely used to call functions in the C++ code the file is written in to establish Internet Protocol settings, most likely a means to establish and configure the backdoor.

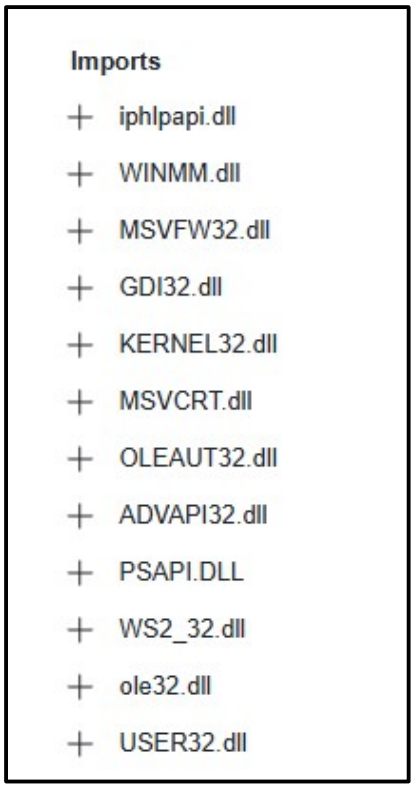


Figure 4: Virus Total imports for Lab05-01.dll

Under the “Behavior” tab, the malware reportedly has indicators of persistence, privilege escalation, and defense evasion (Figures 5-7). Many of the behavioral details match the capabilities of the file’s imports, such as modifying/starting/stopping services and modifying the registry.

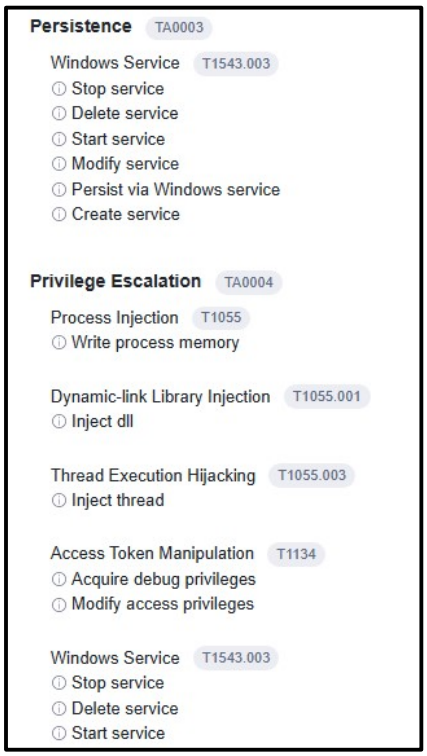


Figure 5: Virus Total behavior for Lab05-01.dll

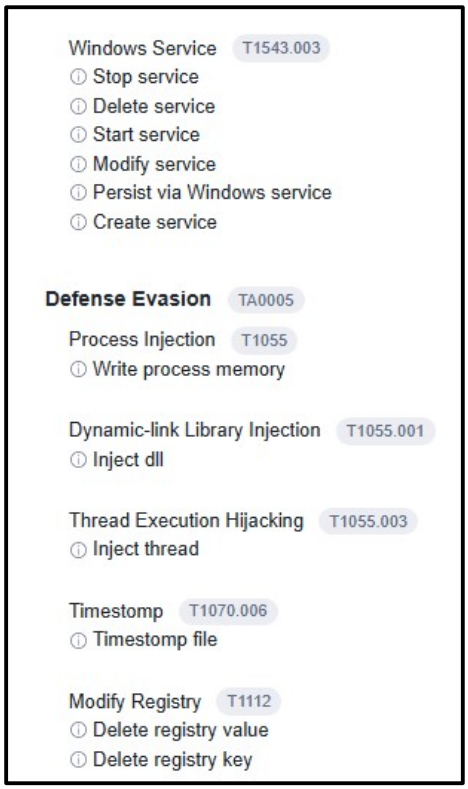


Figure 6: (Cont.) Virus Total behavior for Lab05-01.dll

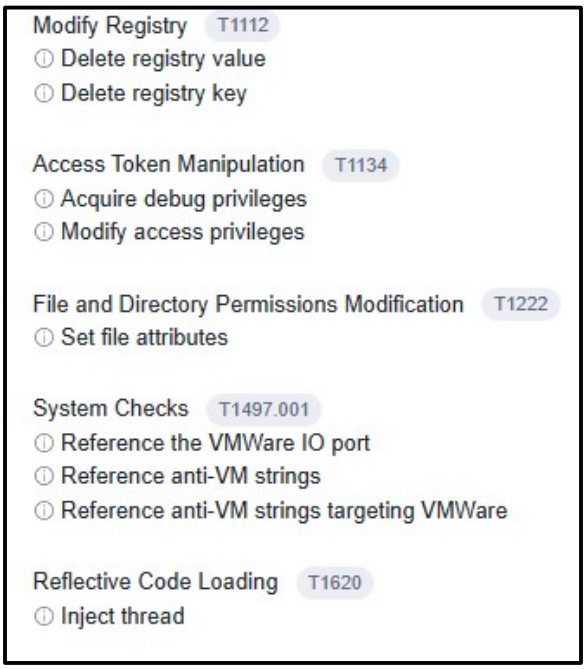


Figure 7: (Cont.) Virus Total behavior for Lab05-01.dll

It is also reported that it has the capability to perform screen captures and the ability to download and write a file (Figure 8). This type of behavior is consistent with backdoor malware, but in Figure 9, VirusTotal states that there were no found Dropped Files or network communications. Because this file was provided from an educational textbook, it most likely is a neutered backdoor without any true network capabilities but rather the appearance of having them. The use of the networking-based imports indicates that some sort of network-related code will be found when examined with IDA Pro.

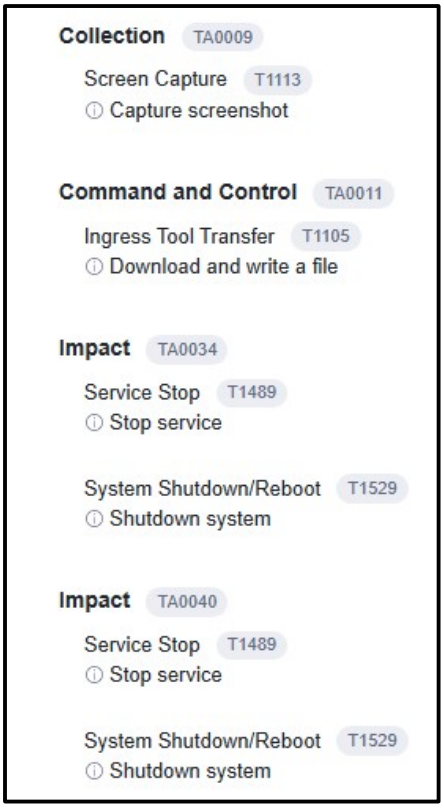


Figure 8: (Cont.) Virus Total behavior for Lab05-01.dll

Figure 8: Virus Total found no dropped files or network comms for Lab05-01.dll

LAB 5-1

LAB 5-1 Question 1

What is the address of DllMain?

BLUF: 0x1000D02E

Once a file is loaded into IDA Pro, it automatically presents the graph view of the main entry point of the program and is normally labeled as “DllMain”. However, in this case we see code calling for “DllEntryPoint” at the .text address of 0x1001516D (Figure 9). This was also confirmed in the “Exports” subview (Figure 10).

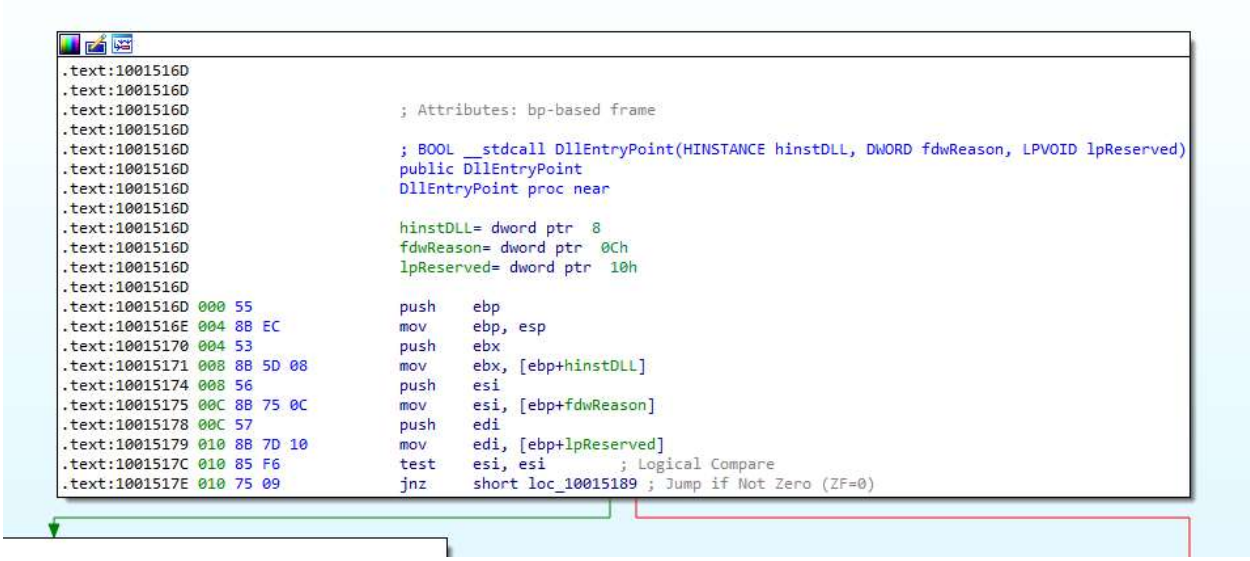


Figure 9: DllEntryPoint .text address in graph view.

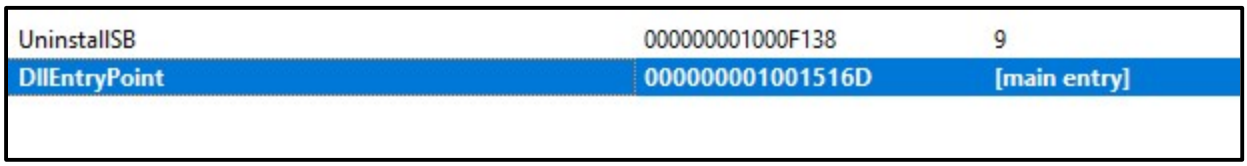
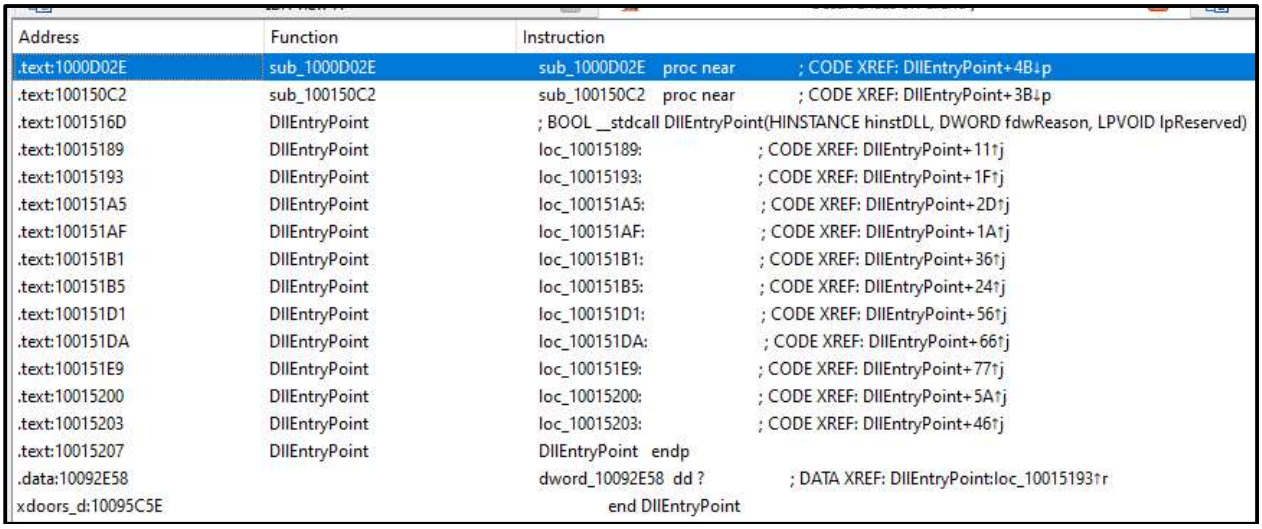


Figure 10: DllEntryPoint address in Exports subview.

Although there are multiple lines with the address of 0x1001516D in Figure 9, we know that the first command called by DllEntryPoint is “push ebp”, which saves the base pointer address of the

stack until it is popped off again, returning to the next line of assembly code after the function is called. To see what other occurrences of this function are within the code, a search of "DllEntry" was conducted and 17 instances were found (Figure 11).



Address	Function	Instruction
.text:1000D02E	sub_1000D02E	sub_1000D02E proc near ; CODE XREF: DllEntryPoint+4B↓p
.text:100150C2	sub_100150C2	sub_100150C2 proc near ; CODE XREF: DllEntryPoint+3B↓p
.text:1001516D	DllEntryPoint	; BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
.text:10015189	DllEntryPoint	loc_10015189: ; CODE XREF: DllEntryPoint+11↑j
.text:10015193	DllEntryPoint	loc_10015193: ; CODE XREF: DllEntryPoint+1F↑j
.text:100151A5	DllEntryPoint	loc_100151A5: ; CODE XREF: DllEntryPoint+2D↑j
.text:100151AF	DllEntryPoint	loc_100151AF: ; CODE XREF: DllEntryPoint+1A↑j
.text:100151B1	DllEntryPoint	loc_100151B1: ; CODE XREF: DllEntryPoint+36↑j
.text:100151B5	DllEntryPoint	loc_100151B5: ; CODE XREF: DllEntryPoint+24↑j
.text:100151D1	DllEntryPoint	loc_100151D1: ; CODE XREF: DllEntryPoint+56↑j
.text:100151DA	DllEntryPoint	loc_100151DA: ; CODE XREF: DllEntryPoint+66↑j
.text:100151E9	DllEntryPoint	loc_100151E9: ; CODE XREF: DllEntryPoint+77↑j
.text:10015200	DllEntryPoint	loc_10015200: ; CODE XREF: DllEntryPoint+5A↑j
.text:10015203	DllEntryPoint	loc_10015203: ; CODE XREF: DllEntryPoint+46↑j
.text:10015207	DllEntryPoint	DllEntryPoint endp
.data:10092E58		dword_10092E58 dd ? ; DATA XREF: DllEntryPoint:loc_10015193↑r
xdoors_d:10095C5E		end DllEntryPoint

Figure 11: Occurrences of DllEntryPoint.

Interestingly, we see the address of 0x1001516D found in figures 9 and 10 on the third line of this search. But there are no instances of DllMain that were found within IDA Pro (Figure 12).

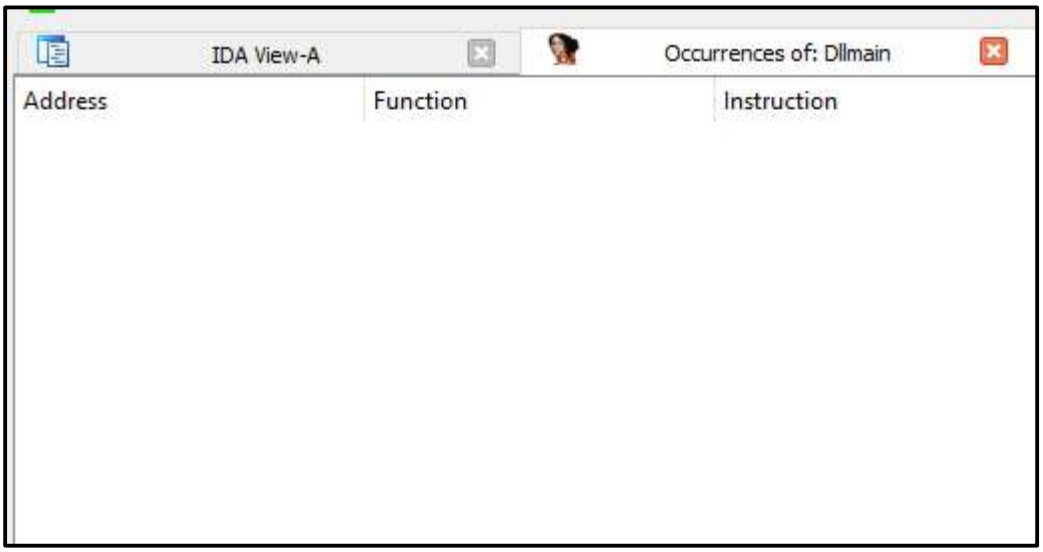


Figure 11: No occurrences of DllMain.

Upon further investigation, we can confirm that the address of 0x1001516D is the address for DllEntryPoint and SHOULD be the address for DllMain as it is the first code that executes

within the function hierarchy and calls upon multiple subroutines within the assembly code, confirmed by looking at the code through the Proximity Browser (Figure 12). Although the book states that the DllMain is at the address of 0x1000D02E, all indicators and evidence point to the contrary with the true address being at 0x1001516D. Lecture 6 slides state on slide 10 that IDA Pro is initially positioned on the DLLMain function, which was confirmed in Figure 9. Additionally, page 495 of Sikorsky states that although "we [should be] taken directly to DllMain at 0x1000D02E…once we load the malicious DLL into IDA Pro," we were taken to DllEntryPoint instead.

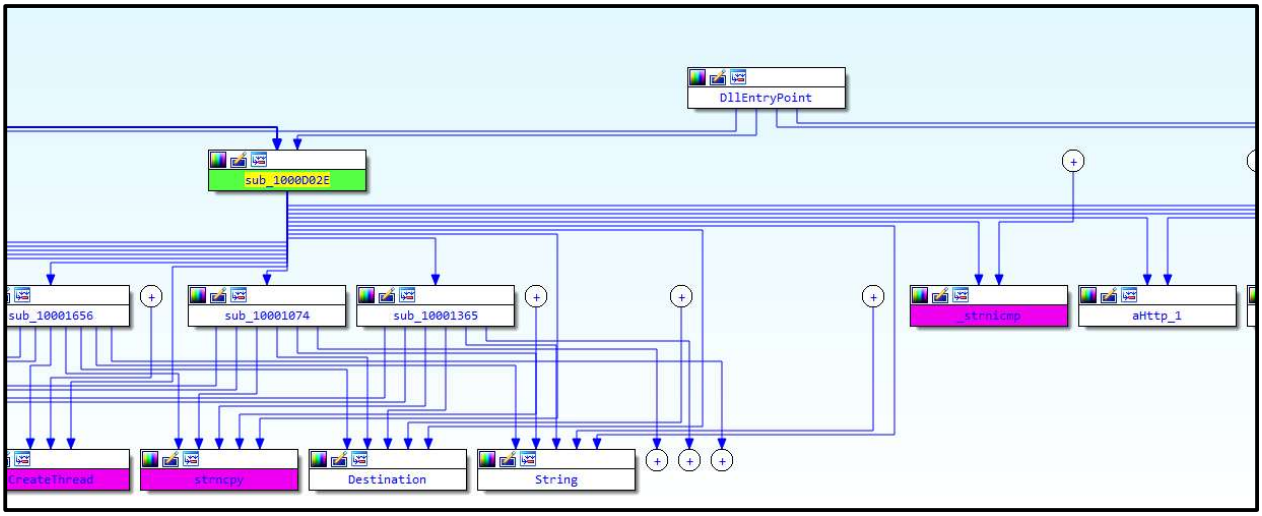


Figure 12: Proximity browser view showing DllEntryPoint should be DllMain.

However, when the malware is loaded onto the Windows XP environment on the VLE, we can see that indeed, we are taken to the proper address of 0x1000D02E and IDA has properly labeled the section as DllMain (Figure 13).

```
.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]
.text:1000D032                 dec     eax
.text:1000D033                 jnz     loc_1000D107
.text:1000D039                 mov     eax, [esp+hinstDLL]
.text:1000D03D                 push    ebx
.text:1000D03E                 mov     ds:hModule, eax
.text:1000D043                 mov     eax, off_10019044
.text:1000D048                 push    esi
.text:1000D049                 add     eax, 0Dh
.text:1000D04C                 push    edi
.text:1000D04D                 push    eax             ; char *
.text:1000D04E                 call    strlen
.text:1000D053                 mov     ebx, ds:CreateThread
.text:1000D059                 mov     esi, ds:_strnicmp
.text:1000D05F                 xor     edi, edi
.text:1000D061                 pop     ecx
.text:1000D062                 test    eax, eax
.text:1000D064                 jz      short loc_1000D089
```

Figure 13: Windows XP IDA Pro showing address of DllMain.

LAB 5-1 Question 2

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

BLUF: 0x100163CC

When organizing the "Imports" window alphabetically, we notice that it sorts the imports in alphabetical order based off of the decimal value of the ASCII letter, placing the lowercase letters further down on the list than the uppercase letters (Figure 14). We find the import "gethostbyname" at the memory address of 0x100163CC with an ordinal of 52. It also is imported from the WS2_32 library located in the WS2_32.dll (Figure 15).

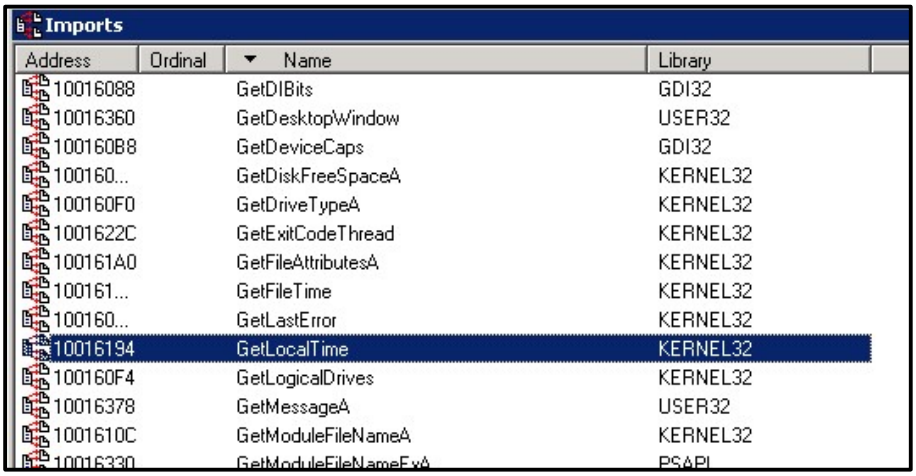


Figure 14: Windows XP IDA Pro showing decimal order of sorting by name.

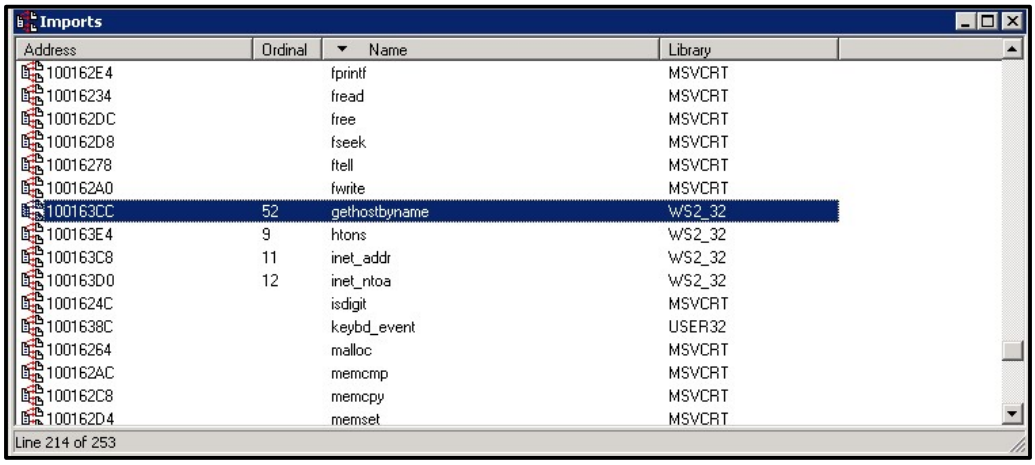


Figure 15: IDA Pro showing address of import 'gethostbyname.'

LAB 5-1 Question 3

How many functions call gethostbyname?

BLUF: Nine.

To view the functions that call gethostbyname, we simply first double-click on gethostbyname within the imports window to be taken to the memory address of 0x100163CC (Figure 16).

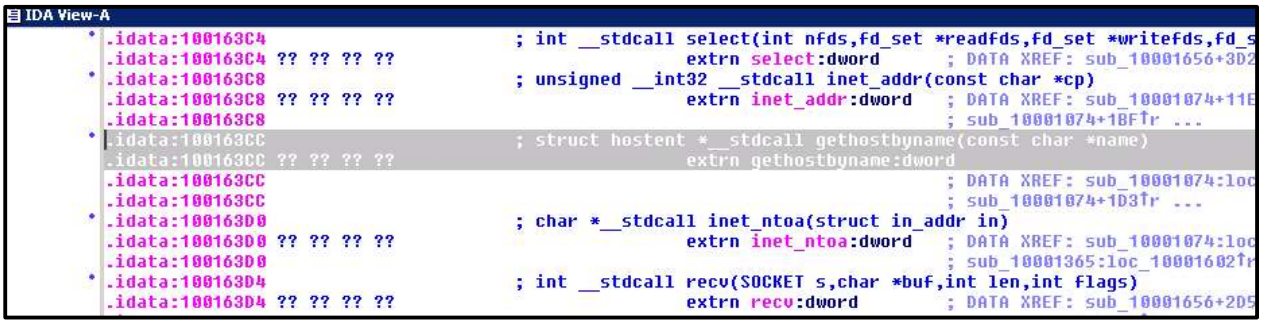


Figure 16: gethostbyname memory address code.

Below the highlighted code in Figure 16, we see a comment indicating a 'DATA XREF' followed by multiple memory addresses. These are cross-references to the to functions that call gethostbyname. IDA Pro has a built-in shortcut of 'Ctrl+X' to view the code cross-references. The code cross-references provides information as to where the function is called from and which functions can access the data. When we use this shortcut, the '_xrefs to gethostbyname' window appears, showing nine functions that access the gethostbyname function (Figure 17). We also see a column labeled, "type". Each instance of the gethostbyname cross reference has a type of 'r' which is a "read" reference.

Additionally, we see that there are five extremely-similar memory addresses that call gethostbyname: 0x10001074, 0x10001365, 0x10001656, 0x1000208F, and 0x10002CCE. This indicates that there are five functions that call gethostbyname with some calling it multiple times.

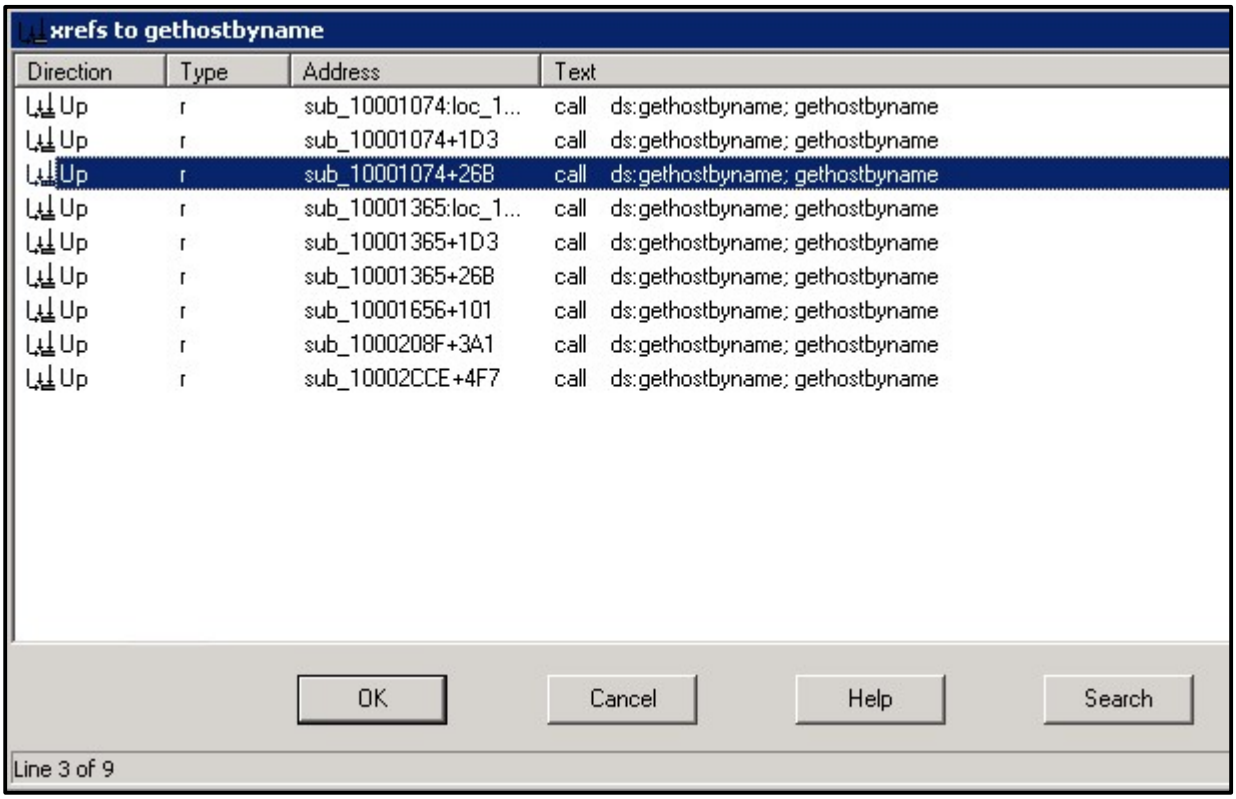


Figure 17: Cross references to gethostbyname.

LAB 5-1 Question 4

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

BLUF: pics.practicalmalwareanalysis.com.

Another useful shortcut for IDA Pro is “Jump to Address” invoked by simply pressing “G” within the ‘IDA View’ window. When we enter the address of 0x10001757 within the ‘Jump to Address’ window, we see the following code: (Figure 18).

```
.text:10001742 688 39 1D CC E5 08 10          cmp     dword_1008E5CC, ebx ; Compare Two Operands
.text:10001748 688 0F 85 9F 00 00 00          jnz     loc_100017ED    ; Jump if Not Zero (ZF=0)
.text:1000174E 688 A1 40 90 01 10             mov     eax, off_10019040
.text:10001753 688 83 C0 0D                   add     eax, 0Dh        ; Add
.text:10001756 688 50                         push    eax             ; name
.text:10001757 68C FF 15 CC 63 01 10          call    ds:gethostbyname ; Indirect Call Near Procedure
.text:1000175D 688 8B F0                      mov     esi, eax
.text:1000175F 688 3B F3                      cmp     esi, ebx        ; Compare Two Operands
.text:10001761 688 74 5D                      jz      short loc_100017C0 ; Jump if Zero (ZF=1)
.text:10001763 688 0F BF 46 0A                movsx   eax, word ptr [esi+0Ah] ; Move with Sign-Extend
.text:10001767 688 50                         push    eax             ; size_t
.text:10001768 68C 8B 46 0C                   mov     eax, [esi+0Ch]
.text:1000176B 68C FF 30                      push    dword ptr [eax] ; void *
.text:1000176D 690 8D 44 24 40                lea     eax, [esp+690h+in] ; Load Effective Address
.text:10001771 690 50                         push    eax             ; void *
.text:10001772 694 E8 C9 37 01 00             call    memcpy          ; Call Procedure
```

Figure 17: Code at memory address 0x10001757.

In order to figure out which DNS request will be made, we need to understand what parameters the function takes as arguments. According to Microsoft’s documentation on ‘gethostbyname’, we find that the function is part of the winsock2.h library (WS2_32.dll). This matches what we found when analyzing the imports for gethostbyname in Figure 15. The official documentation also states that there is one argument that the function takes: a const char pointer (Figure 18). (https://learn.microsoft.com/en-us/windows/win32/api/winsock2/nf-winsock2-gethostbyname)

Figure 18: Arguments for gethostbyname function from Microsoft.com.

Since it only takes one argument, we can conclude that from the assembly code in Figure 17 that prior to gethostbyname is called, the EAX register will store the const char argument. We can also observe that a jnz instruction occurs just prior to a value being moved into EAX, meaning that value just after the jnz instruction is the argument for the gethostbyname function. Then 0xD bytes (decimal value of 13) are added to EAX and EAX is moved onto the stack with the 'push' instruction. When we double-click on the value moved into EAX, 'off_10019040, we can see a partial domain name within the comment, '"[This is RDO]pics.practicalmalwareanalys"…' (Figure 19).

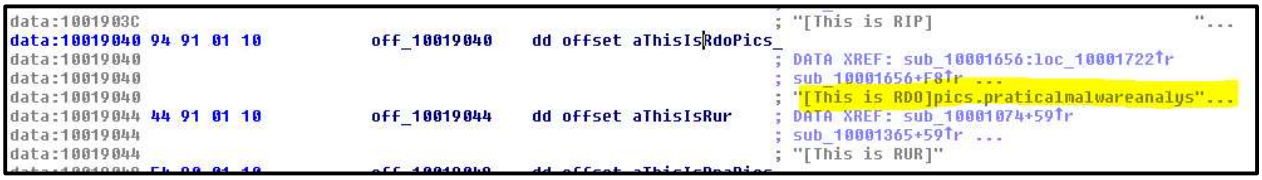


Figure 19: Partial domain name at address 0x10019040.

When we double-click again on the text, 'aThisIsRdoPics_' in Figure 19, we can see the full domain name of 'pics.practicalmalwareanalysis.com' (Figure 20). We know we can discount the "[This is RDO]" portion of the domain name since we had previously added 0xD bytes to EAX prior to pushing it onto the stack. "[This is RDO]" contains 13 characters and 0xD has a

decimal value of 13. We can also see that the variable ‘aThisIsRdoPics_’ is stored in the .data section at memory address 0x10019194 (Figure 20).

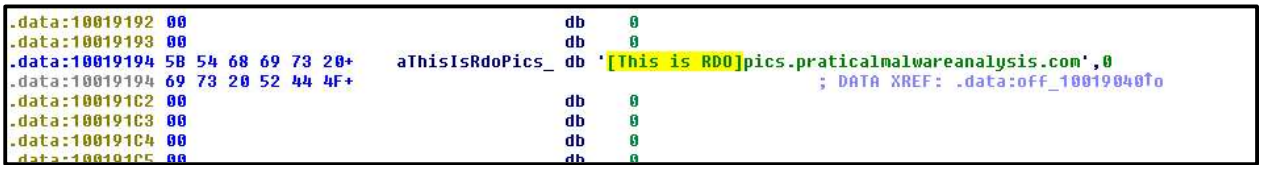


Figure 20: Full domain name.

LAB 5-1 Question 5

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

BLUF: Twenty (20).

Once again, we can use the Jump to Address shortcut to get to 0x10001656. Initially, we don't see any local variables but rather a subtraction instruction where 678h is deducted from the ESP stack pointer (the top of the stack) (Figure 21).

```
.text:10001656 000 81 EC 78 06 00 00                sub     esp, 678h       ; Integer Subtraction
.text:1000165C 678 53                               push    ebx
.text:1000165D 67C 55                               push    ebp
.text:1000165E 680 56                               push    esi
.text:1000165F 684 57                               push    edi
.text:10001660 688 E8 9B F9 FF FF                   call    sub_10001000    ; Call Procedure
.text:10001665 688 85 C0                            test    eax, eax        ; Logical Compare
.text:10001667 688 75 53                            jnz     short loc_100016BC ; Jump if Not Zero (ZF=
```

Figure 21: Instruction at 0x10001656.

However, simply scrolling up just a bit shows us all of the local variables for this subroutine since local variables are always stored at the beginning of a function (Figure 22).

```
A
.text:10001656
.text:10001656                          ; DWORD __stdcall sub_10001656(LPVOID)
.text:10001656                          sub_10001656    proc near               ; DATA XREF: DllMain(x,x,x)+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                          in              = in_addr ptr -650h
.text:10001656                          Parameter       = byte ptr -644h
.text:10001656                          CommandLine     = byte ptr -63Fh
.text:10001656                          Data            = byte ptr -638h
.text:10001656                          var_544         = dword ptr -544h
.text:10001656                          var_50C         = dword ptr -50Ch
.text:10001656                          var_500         = dword ptr -500h
.text:10001656                          var_4FC         = dword ptr -4FCh
.text:10001656                          readfds         = fd_set ptr -4BCh
.text:10001656                          phkResult       = HKEY__ 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                          arg_0           = dword ptr  4
.text:10001656
.text:10001656 000 81 EC 78 06 00 00                sub     esp, 678h       ; Integer Subtraction
.text:1000165C 678 53                               push    ebx
```

Figure 22: Instruction at 0x10001656.

When we copy and paste all of these lines from var_675 to arg_0 into Notepad++, we see that there are 21 lines (Figure 23). However, local variables will be annotated with negative offsets and we see that arg_0 does not have one. Therefore, there are 20 variables that are stored in 0x675 bytes (decimal 1,653 bytes). It is possible that there are even more local variables within this function since IDA Pro Free limits itself to only 20 variables.

new 1

```
1  .text:10001656  var_675         = byte ptr -675h
2  .text:10001656  var_674         = dword ptr -674h
3  .text:10001656  hModule         = dword ptr -670h
4  .text:10001656  timeout         = timeval ptr -66Ch
5  .text:10001656  name            = sockaddr ptr -664h
6  .text:10001656  var_654         = word ptr -654h
7  .text:10001656  in              = in_addr ptr -650h
8  .text:10001656  Parameter       = byte ptr -644h
9  .text:10001656  CommandLine     = byte ptr -63Fh
10 .text:10001656  Data            = byte ptr -638h
11 .text:10001656  var_544         = dword ptr -544h
12 .text:10001656  var_50C         = dword ptr -50Ch
13 .text:10001656  var_500         = dword ptr -500h
14 .text:10001656  var_4FC         = dword ptr -4FCh
15 .text:10001656  readfds         = fd_set ptr -4BCh
16 .text:10001656  phkResult       = HKEY__ ptr -3B8h
17 .text:10001656  var_3B0         = dword ptr -3B0h
18 .text:10001656  var_1A4         = dword ptr -1A4h
19 .text:10001656  var_194         = dword ptr -194h
20 .text:10001656  WSAData         = WSAData ptr -190h
21 .text:10001656  arg_0           = dword ptr  4
```

Figure 22: Instruction at 0x10001656

LAB 5-1 Question 6

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

BLUF: One.

Since we know that malware is often written in C and from our knowledge of C language a typical main function is defined as “int main (int argc, char* argv[])”, we can anticipate IDA to label such arguments with a variation of “arg”. Immediately, the ‘arg_0’ in Figure 22 stands out as the likely candidate. When double-clicking on ‘arg_0’, the Stack Frame window opens up and defines the stack variables. From this window, we can see that arg_0 is an 8-byte variable (type double) and comes immediately after the 0x00 stack frame address, indicating that this is the only argument for the subroutine at memory address 0x10001656 (Figure 23).

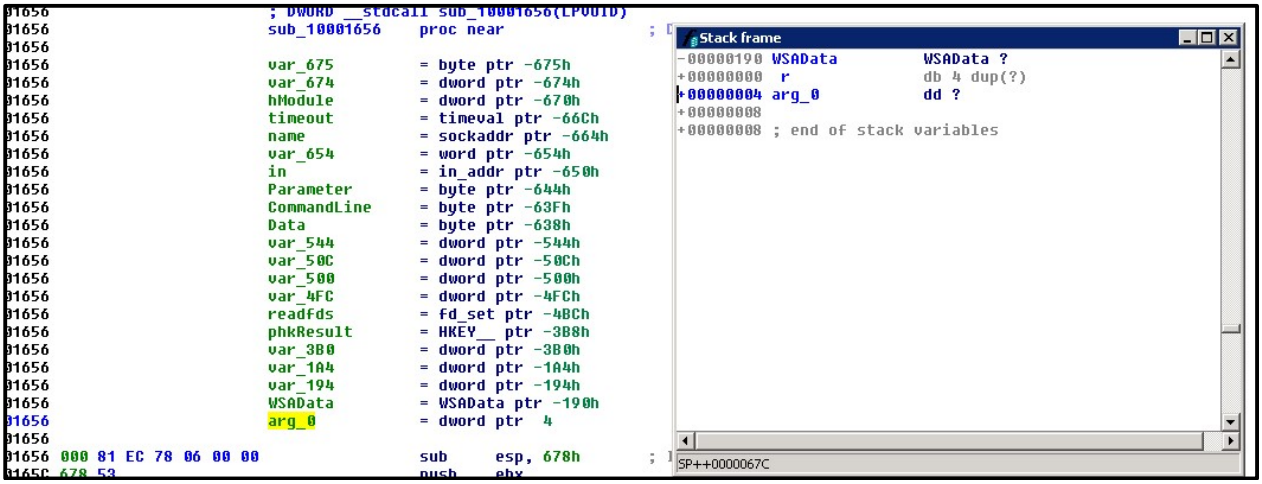


Figure 23: ‘arg_0’ is the only argument for the subroutine.

LAB 5-1 Question 7

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

BLUF: 0x10095B34.

When opening up the Strings window, we organize by the string name and find \cmd.exe /c. We initially see its address located at “xdoors_d: 0x10095B34” (Figure 24).

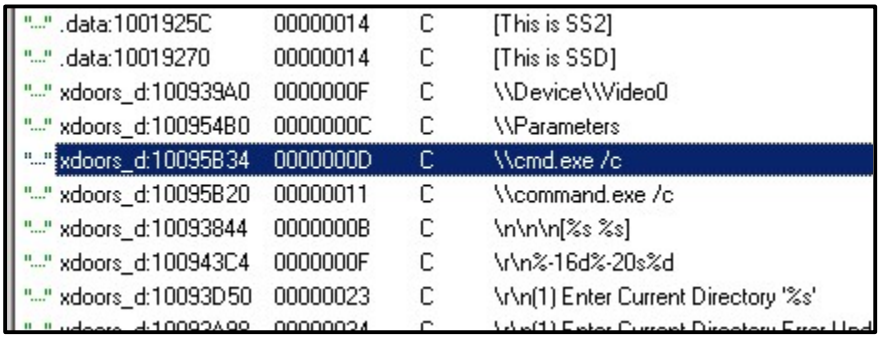


Figure 24: Potential address for \cmd.exe /c.

Simply double-clicking on the line in the Strings window brings us to the command in IDA View and confirms that the memory address is indeed 0x10095B34 (Figure 25). We also see that it is stored within the variable ‘aCmd_exeC’. If we take the string ‘cmd.exe /c’ we can anticipate that the command prompt will be called with some command(s) that follow the /c flag. The command prompt will run the command(s) and then terminate itself due to the /c flag being set.

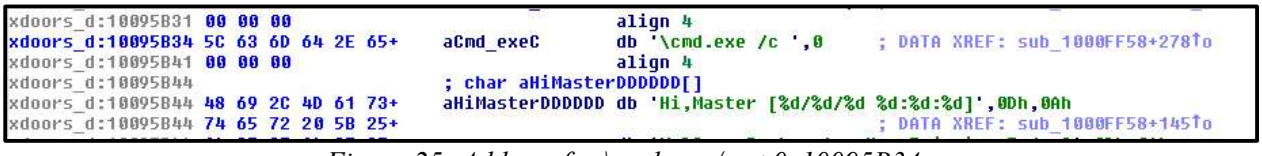


Figure 25: Address for \cmd.exe /c at 0x10095B34.

LAB 5-1 Question 8

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

BLUF: The variable ‘aCmd_exeC’ containing the string ‘\cmd.exe /c’ is pushed onto the stack.

In Figure 25, we saw a “DATA XREF” to ‘\cmd.exe /c’. Using the Ctrl+X shortcut, we see a single cross-reference to the code where it appears it is pushed onto the stack (Figure 26).

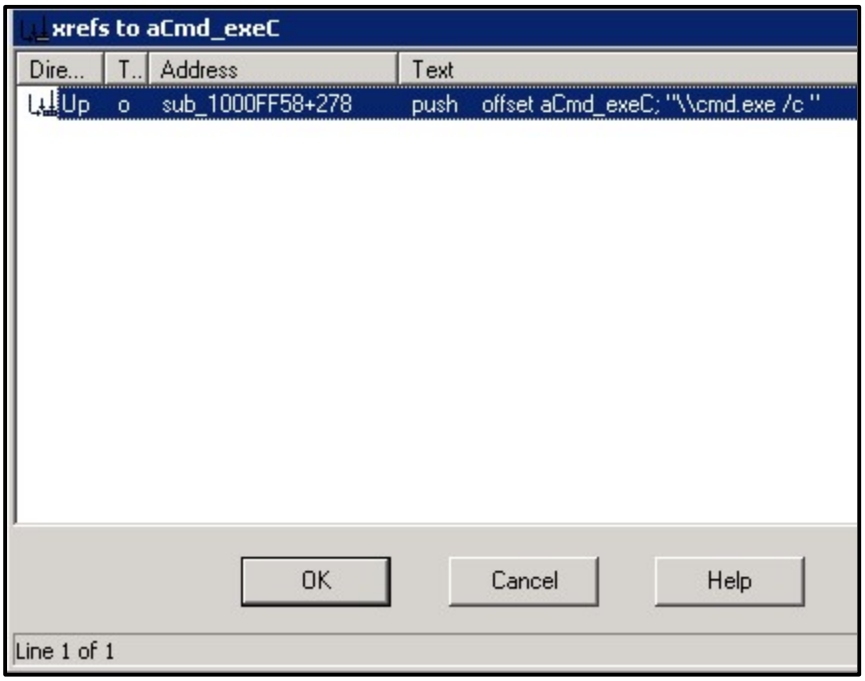


Figure 26: One cross reference to \cmd.exe /c.

When we open this cross-reference in IDA View, we confirm that the variable ‘aCmd_exeC’ has a ‘push offset’ command at memory address 0x100101D0, pushing it onto the stack. Following this, it jumps to the memory address of 0x100101DC (Figures 27 and 28).

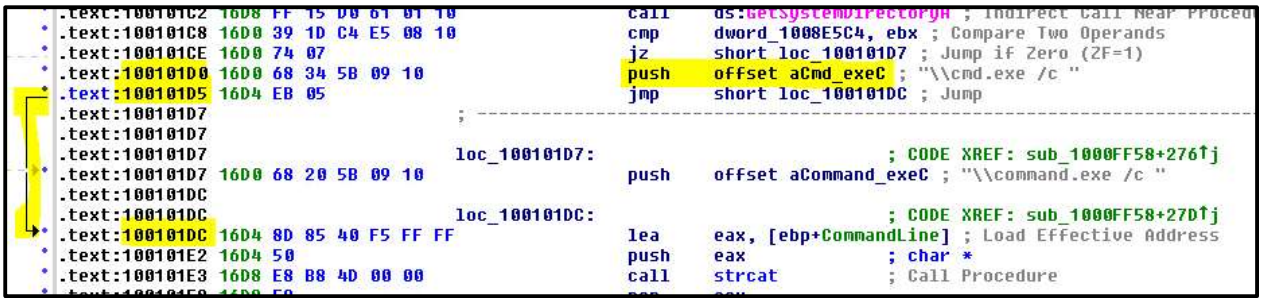


Figure 27: Text view.

Since the variable 'aCmd_exeC' contains the string '\cmd.exe /c' and has been pushed onto the stack, we can anticipate that this string will eventually have commands concatenated to it following further instructions. We can also see that in Figure 28 (same as Figure 27, just the graph view) the effective address of the stack base pointer, EBP, with the addition of "Command Line" is loaded into EAX. Therefore, whatever was string is stored inside the Command Line variable will be concatenated to '\cmd.exe /c' and execute.

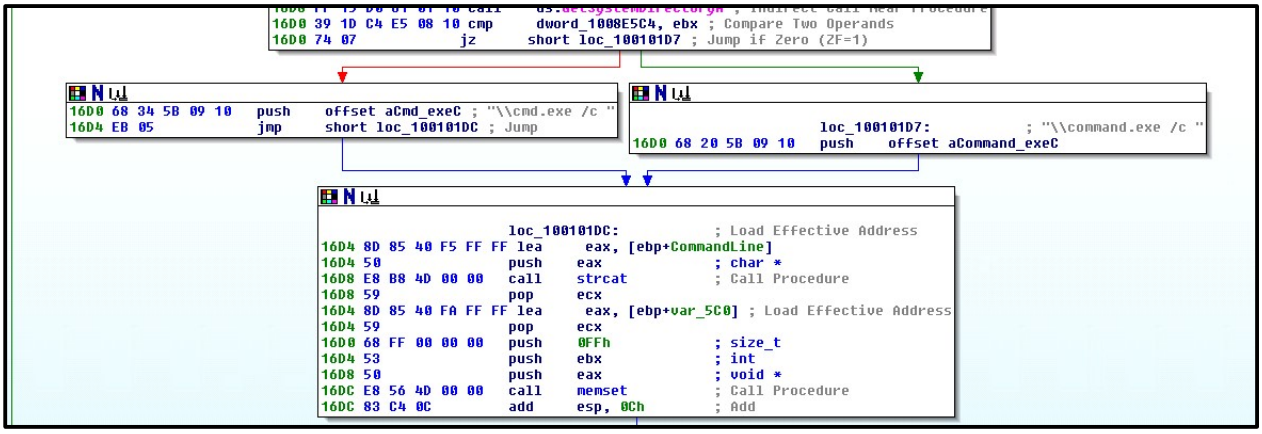


Figure 28: Graph view.

We also notice that this section of code is part of a much larger function when examining the graph overview window (Figure 29). This makes since because Figure 26 shows the beginning of the function at 0x1000FF58 and we navigated to a memory address 278 bytes beyond the start of the function to memory address 0x100101D0 (Figure 30).

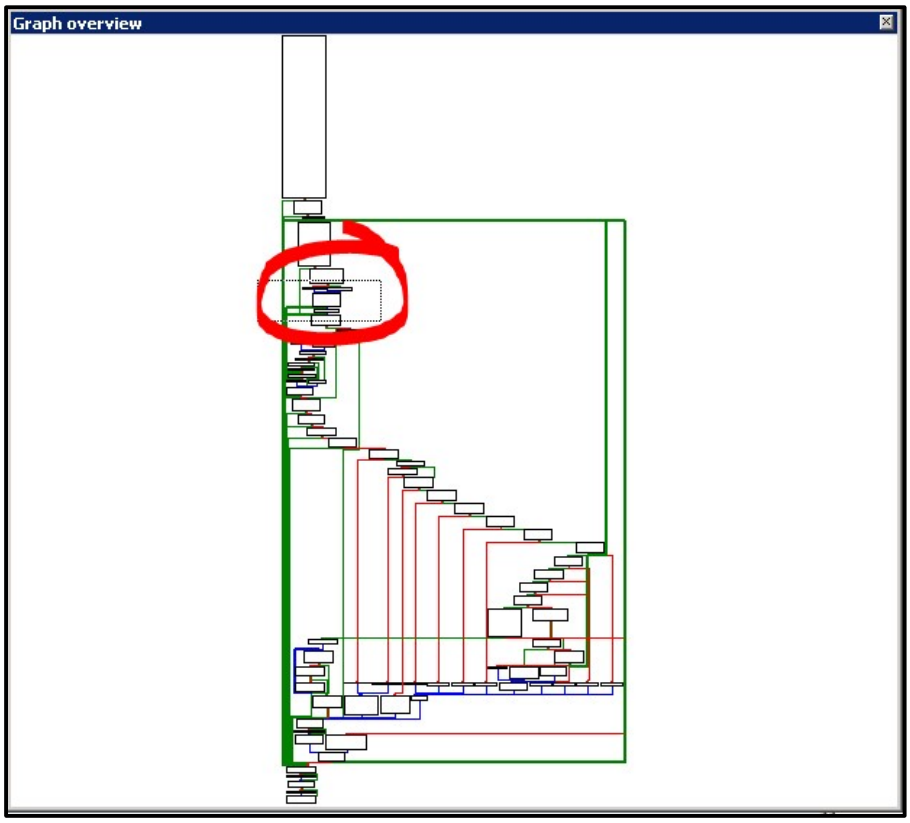


Figure 29: Graph view of function at 0x1000FF58. Red circle indicates where Figures 27 and 28 were located.

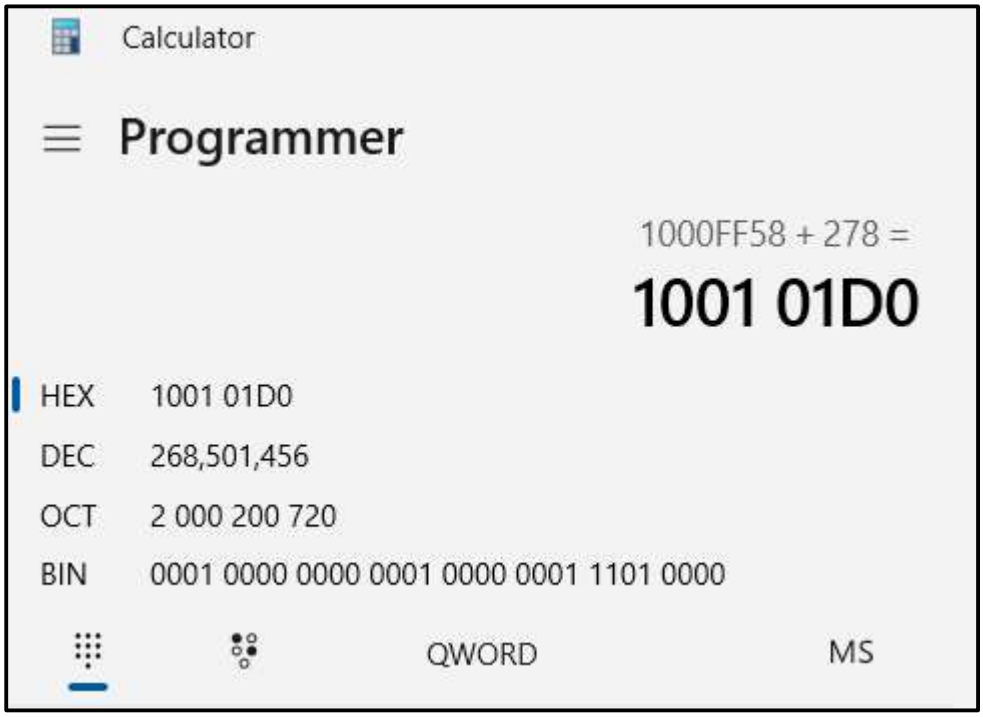


Figure 30: Math to confirm byte reading from top of function to 0x100101D.

As we navigate down from the top of the function starting at 0x1000FF58, we notice multiple external functions within the code that are indicated by pink text. We see the functions GetCurrentDirectoryA, GetTickCount, and GetLocalTime (Figure 31).

```
text:1000FF97 16D8 FF 15 14 61 01 10          call    ds:GetCurrentDirectoryA ; Indirect C
text:1000FF9D 16D0 8D 85 40 FE FF FF          lea     eax, [ebp+Buffer] ; Load Effective A
text:1000FFA3 16D0 68 5C 5C 09 10             push    offset asc_10095C5C ; ">"
text:1000FFA8 16D4 50                         push    eax             ; char *
text:1000FFA9 16D8 E8 F2 4F 00 00             call    strcat          ; Call Procedure
text:1000FFAE 16D8 59                         pop     ecx
text:1000FFAF 16D4 33 C0                      xor     eax, eax        ; Logical Exclusive
text:1000FFB1 16D4 59                         pop     ecx
text:1000FFB2 16D0 8D BD 41 F1 FF FF          lea     edi, [ebp+var_EC0+1] ; Load Effectiv
text:1000FFB8 16D0 B9 FF 00 00 00             mov     ecx, 0FFh
text:1000FFBD 16D0 88 9D 40 F1 FF FF          mov     byte ptr [ebp+var_EC0], bl
text:1000FFC3 16D0 F3 AB                      rep stosd               ; Store String
text:1000FFC5 16D0 66 AB                      stosw                   ; Store String
text:1000FFC7 16D0 AA                         stosb                   ; Store String
text:1000FFC8 16D0 FF D6                      call    esi ; GetTickCount ; Indirect Call N
text:1000FFCA 16D0 89 45 F8                   mov     [ebp+NumberOfBytesRead], eax
text:1000FFCD 16D0 8D 45 DC                   lea     eax, [ebp+SystemTime] ; Load Effecti
text:1000FFD0 16D0 50                         push    eax             ; lpSystemTime
text:1000FFD1 16D4 FF 15 94 61 01 10          call    ds:GetLocalTime ; Indirect Call Near
text:1000FFD7 16D0 E8 79 35 FF FF             call    sub_10003555    ; Call Procedure
```

Figure 31: External function calls.

We also see an sprint call where it appears that another variable labeled “aHiMasterDDDDDD” (hereby referred to as HiMaster) is pushed onto the stack (Figure 32). Due since the function sprintf prints data to the screen and we notice multiple integer format specifiers (%d) within the comments.

```
ext:10010097 170C 8D 85 40 F1 FF FF           lea     eax, [ebp+var_EC0] ; Load Effective Address
ext:1001009D 170C 68 44 5B 09 10              push    offset aHiMasterDDDDDD ; "Hi,Master [%d/%d/%d %d:%d:%d]\r\nWelCome "...
ext:100100A2 1710 50                          push    eax             ; char *
ext:100100A3 1714 FF 15 F4 62 01 10           call    ds:sprintf      ; Indirect Call Near Procedure
ext:100100A9 1714 83 C4 44                    add     esp, 44h        ; Add
ext:100100AC 16D0 33 DB                       xor     ebx, ebx        ; Logical Exclusive OR
ext:100100AE 16D0 8D 85 40 F1 FF FF           lea     eax, [ebp+var_EC0] ; Load Effective Address
```

Figure 32: sprintf function call with HiMaster variable.

Double-clicking on the HiMaster variable, we are taken to the address within the xdoors section at 0x10095B4 and can see that the variable contains multiple strings that appear to greet and provide information to some user (Figure 33). It provides details regarding the machine uptime and idle time, most likely placed into the strings by the functions highlighted in Figure 31. Most

interestingly, it also gives a string that states “Encrypt Magic Number for this Remote Shell Session”.

```
xdoors_d:10095B20                       ; char aCommand_exeC[]
xdoors_d:10095B20 5C 63 6F 6D 6D 61+    aCommand_exeC   db '\command.exe /c ',0 ; DATA XREF: sub_1000FF58:loc_100101D7↑o
xdoors_d:10095B31 00 00 00                              align 4
xdoors_d:10095B34 5C 63 6D 64 2E 65+    aCmd_exeC       db '\cmd.exe /c ',0     ; DATA XREF: sub_1000FF58+278↑o
xdoors_d:10095B41 00 00 00                              align 4
xdoors_d:10095B44                       ; char aHiMasterDDDDDD[]
xdoors_d:10095B44 48 69 2C 4D 61 73+    aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah
xdoors_d:10095B44 74 65 72 20 5B 25+                                    ; DATA XREF: sub_1000FF58+145↑o
xdoors_d:10095B44 64 2F 25 64 2F 25+                    db 'WelCome Back...Are You Enjoying Today?',0Dh,0Ah
xdoors_d:10095B44 64 20 25 64 3A 25+                    db 0Dh,0Ah
xdoors_d:10095B44 64 3A 25 64 5D 0D+                    db 'Machine UpTime  [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Secon'
xdoors_d:10095B44 0A 57 65 6C 43 6F+                    db 'ds]',0Dh,0Ah
xdoors_d:10095B44 6D 65 20 42 61 63+                    db 'Machine IdleTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Seco'
xdoors_d:10095B44 6B 2E 2E 2E 41 72+                    db 'nds]',0Dh,0Ah
xdoors_d:10095B44 65 20 59 6F 75 20+                    db 0Dh,0Ah
xdoors_d:10095B44 45 6E 6A 6F 79 69+                    db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',0Dh,0Ah
xdoors_d:10095B44 6E 67 20 54 6F 64+                    db 0Dh,0Ah,0
xdoors_d:10095C5C                       ; char asc_10095C5C[]
xdoors_d:10095C5C                       asc_10095C5C:                   ; DATA XREF: sub_1000FF58+4B↑o
xdoors_d:10095C5C                                                       ; sub_1000FF58+3E1↑o
```

Figure 33: HiMaster variable contents.

We can therefore deduce that the function beginning at 0x1000FF58 will produce a terminal using aCmd_exeC with messages in the HiMaster variable. The main purpose of this function is to create a remote shell session for the user that receives this messages. This is further supported by calls to external functions such as the CreatePipe function (part of namedpipeapi.h) at 0x10010183 (Figure 34) and the recv function (part of winsock.h and used to create sockets) at 0x1001020B (Figure 35),

```
.text:1001017C 16E0 C7 45 D8 01 00 00+  mov     [ebp+PipeAttributes.bInheritHandle], 1
.text:10010183 16E0 FF 15 F8 61 01 10   call    ds:CreatePipe   ; Indirect Call Near Procedure
.text:10010189 16D0 85 C0               test    eax, eax        ; Logical Compare
.text:1001018B 16D0 0F 84 83 05 00 00   jz      loc_10010714    ; Jump if Zero (ZF=1)
.text:10010191 16D0 8D 45 8C            lea     eax, [ebp+StartupInfo] ; Load Effective Address
```

Figure 34: CreatePipe function.

```
.text:10010208 16DC FF 75 08            push    [ebp+s]         ; s
.text:1001020B 16E0 FF 15 D4 63 01 10   call    ds:recv         ; Indirect Call Near Procedure
.text:10010211 16D0 83 F8 FF            cmp     eax, 0FFFFFFFFh ; Compare Two Operands
.text:10010214 16D0 0F 84 FA 04 00 00   jz      loc_10010714    ; Jump if Zero (ZF=1)
```

Figure 35: recv function.

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.)

BLUF: It uses the subroutine 'sub_10003695' and sets it to a 1 or 0 depending on the operating system the program is running on.

We can see in Figures 27 and 28, we can see the command at memory address 0x100101C8 as, 'cmp dword_1008E5C4'. There are three cross-references to 'dword_1008E5C4' (Figure 36). We can also confirm that dword_1008E5C4 is a global variable as it is located with the .data section at the memory address 0x1008E5C4 (Figure 37).

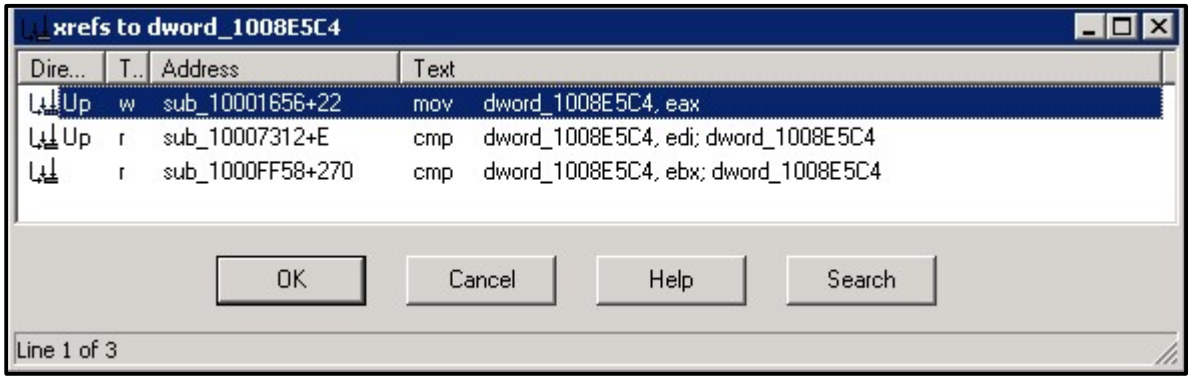


Figure 36: X-Refs to dword_1008E5C4.

Figure 37: dword_1008E5C4 is a global variable.

Looking closer at the xrefs window in Figure 36, we can determine that there is only one instance where the variable is modified with the 'mov dword_1008E5C4, eax'. Therefore, the variable is set with whatever value was placed into the EAX register prior to this command. We

also notice that this variable was modified within a separate function than the one analyzed in Question 8. Figure 38 shows the graph overview of the function and the red circle highlights where the ‘mov’ command is.

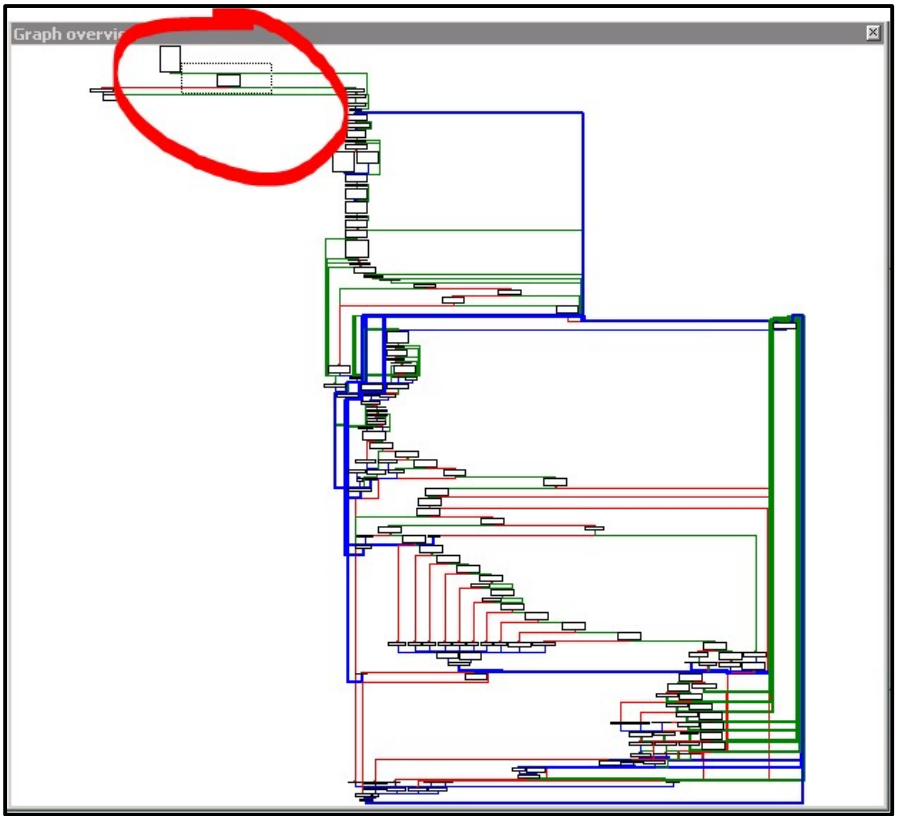


Figure 38: Graph overview where dword_1008E5C4 is set.

Looking at the code prior to dword_1008E5C4 being set, we notice a function call to ‘sub_10003695’ at memory address 0x10001673 (Figure 39). Whatever the return value of this function is will be stored into EAX and then moved into dword_1008E5C4.

```
.text:1000166F 688 89 5C 24 18                 mov     [esp+688h+hModule], ebx
.text:10001673 688 E8 1D 20 00 00              call    sub_10003695    ; Call Procedure
.text:10001678 688 A3 C4 E5 08 10              mov     dword_1008E5C4, eax
.text:1000167D 688 E8 41 20 00 00              call    sub_100036C3    ; Call Procedure
```

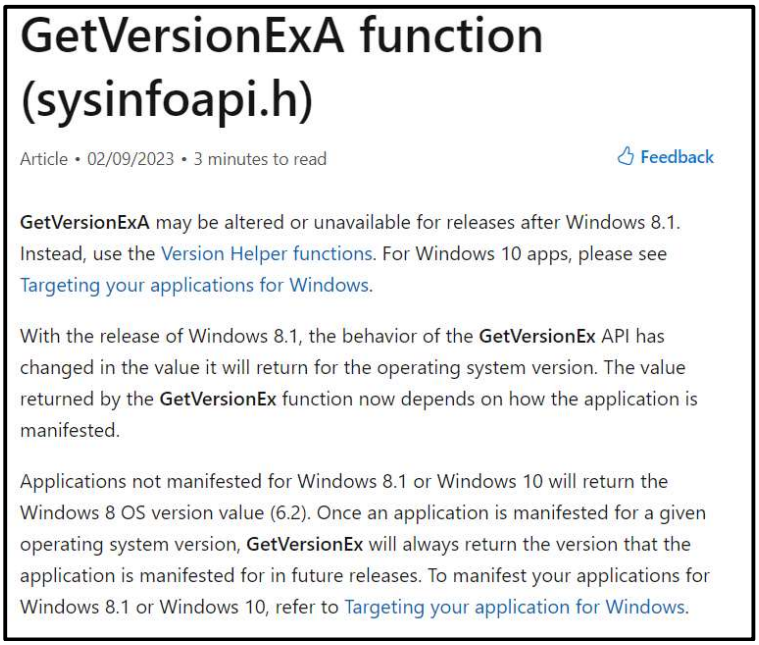
Figure 39: Function call prior to dword_1008E5C4 being set.

To determine what the sub_10003695 function returns, we double click on it and are taken to memory address 0x10003695 in .text and see the code for a subroutine (Figure 40).

```
.text:10003695
.text:10003695                         ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
.text:10003695
.text:10003695                         ; Attributes: bp-based frame
.text:10003695
.text:10003695                         sub_10003695    proc near               ; CODE XREF: sub_10001656+1D↑p
.text:10003695                                                                 ; sub_10003B75+7↓p ...
.text:10003695
.text:10003695                         VersionInformation= _OSVERSIONINFOA ptr -94h
.text:10003695
.text:10003695 000 55                                  push    ebp
.text:10003696 004 8B EC                               mov     ebp, esp
.text:10003698 004 81 EC 94 00 00 00                   sub     esp, 94h        ; Integer Subtraction
.text:1000369E 098 8D 85 6C FF FF FF                   lea     eax, [ebp+VersionInformation] ; Load Effective Address
.text:100036A4 098 C7 85 6C FF FF FF+                  mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
.text:100036AE 098 50                                  push    eax             ; lpVersionInformation
.text:100036AF 09C FF 15 D4 60 01 10                   call    ds:GetVersionExA ; Get extended information about the
.text:100036AF 09C                                                             ; version of the operating system
.text:100036B5 098 33 C0                               xor     eax, eax        ; Logical Exclusive OR
.text:100036B7 098 83 BD 7C FF FF FF+                  cmp     [ebp+VersionInformation.dwPlatformId], 2 ; Compare Two Operands
.text:100036BE 098 0F 94 C0                            setz    al              ; Set Byte if Zero (ZF=1)
.text:100036C1 098 C9                                  leave                   ; High Level Procedure Exit
.text:100036C2 000 C3                                  retn                    ; Return Near from Procedure
.text:100036C2                         sub_10003695    endp
.text:100036C2
```

Figure 40: Subroutine that sets value of dword_1008E5C4.

Immediately we see an external function call of 'GetVersionExA' which is part of the sysinfoapi.h library which will return the operating system version (Figure 41).



GetVersionExA function (sysinfoapi.h)

Article • 02/09/2023 • 3 minutes to read

Feedback

GetVersionExA may be altered or unavailable for releases after Windows 8.1. Instead, use the Version Helper functions. For Windows 10 apps, please see Targeting your applications for Windows.

With the release of Windows 8.1, the behavior of the **GetVersionEx** API has changed in the value it will return for the operating system version. The value returned by the **GetVersionEx** function now depends on how the application is manifested.

Applications not manifested for Windows 8.1 or Windows 10 will return the Windows 8 OS version value (6.2). Once an application is manifested for a given operating system version, **GetVersionEx** will always return the version that the application is manifested for in future releases. To manifest your applications for Windows 8.1 or Windows 10, refer to Targeting your application for Windows.

Figure 41: GetVersionExA documentation from Microsoft.com.

We then see in Figure 40 the next lines of code being the xor-ing of the EAX register and then comparing 'EBP + VersionInformation.dwPlaformID' to 2. This will determine how to set the AL register (the lowest 8-bits of the 32-bit EAX register) with the 'setz' command following the 'cmp' command. This will check to see if the current OS that the program is running on is a

Windows 2000 version or higher. Since there aren't any Boolean data types within assembly but true/false values can be represented by either a 1 or 0, we can therefore determine that once the AL register will be a 1 or 0 and when returned is placed within dword_1008E5C4 for future checks.

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)?

BLUF: Values for registry keys ‘WorkTime’ and ‘WorkTimes’ are sent over the shell.

Scrolling down through the code of 0x1000FF58, we find the section of code that references ‘robotwork’ beginning at memory address 0x1001044 (Figure 42). We also see that many other decisions were made prior to enetering this section of code and is very far down within the code at 0x1000FF58 (Figure 43). Going back to Figure 42, we see that robotwork is pushed to the stack at 0x1001044C, followed by EAX being pushed to the stack after it was loaded with the effective address of EBP+var_5C0. Then the call to ‘memcmp’ occurs at 0x10010452. If the comparison is unsuccessful (the string does not match robotwork), it will jump to the subroutine at 0x10010468. If the comparison is successful, then it will call sub_100052A2 at 0x100052A2.

```
.text:10010444                         ; ---------------------------------------------------------------------------
.text:10010444
.text:10010444                         loc_10010444:                           ; CODE XREF: sub_1000FF58+4E0↑j
.text:10010444 16D0 6A 09                              push    9               ; size_t
.text:10010446 16D4 8D 85 40 FA FF FF                  lea     eax, [ebp+var_5C0] ; Load Effective Address
.text:1001044C 16D4 68 CC 5A 09 10                     push    offset aRobotwork ; "robotwork"
.text:10010451 16D8 50                                 push    eax             ; void *
.text:10010452 16DC E8 01 4B 00 00                     call    memcmp          ; Call Procedure
.text:10010457 16DC 83 C4 0C                           add     esp, 0Ch        ; Add
.text:1001045A 16D0 85 C0                              test    eax, eax        ; Logical Compare
.text:1001045C 16D0 75 0A                              jnz     short loc_10010468 ; Jump if Not Zero (ZF=0)
.text:1001045E 16D0 FF 75 08                           push    [ebp+s]         ; s
.text:10010461 16D4 E8 3C 4E FF FF                     call    sub_100052A2    ; Call Procedure
.text:10010466 16D4 EB 8E                              jmp     short loc_100103F6 ; Jump
.text:10010468                         ; ---------------------------------------------------------------------------
```

Figure 42: Block of code where robotwork and memcmp are.

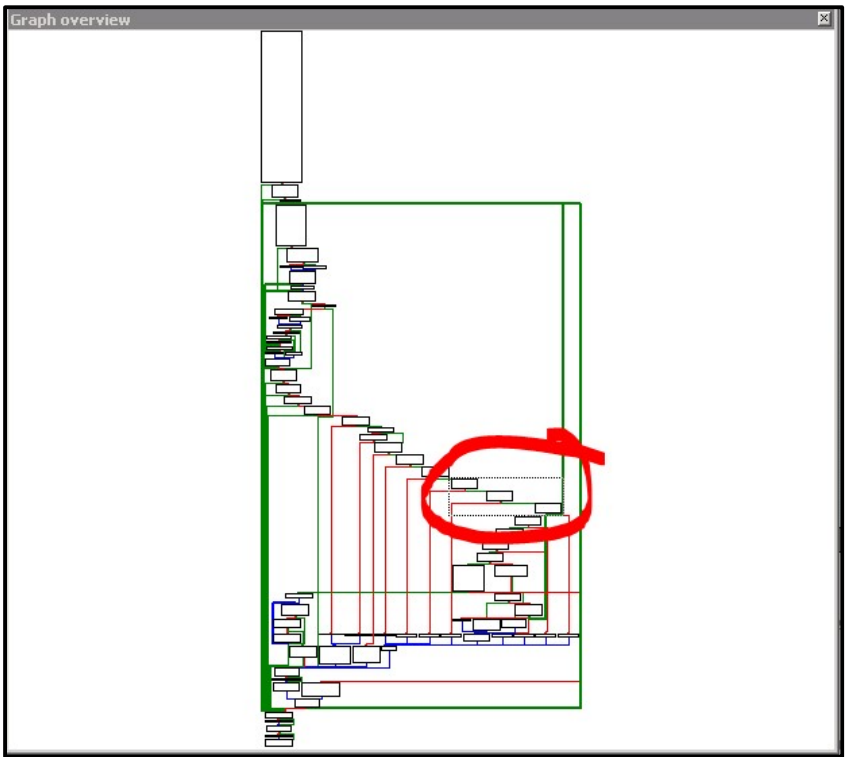


Figure 43: Graph overview of where robotwork and memcmp are.

Note: the variable ‘aRobotwork’ is defined in the xdoors section at memory address 0x10095ACC and is defined as the string ‘robotwork’ (Figure 44).

```
xdoors_d:10095ACA 00 00                                   align 4
xdoors_d:10095ACC                       ; void aRobotwork
xdoors_d:10095ACC 72 6F 62 6F 74 77+    aRobotwork        db 'robotwork',0        ; DATA XREF: sub_1000FF58+4F4↑o
xdoors_d:10095AD6 00 00                                   align 4
xdoors_d:10095AD8                       ; void aLanguage
```

Figure 44: aRobotwork definition.

To determine what value is attempting to be compared to the string ‘robotwork’, we can analyze what happens if the cmp instruction is unsuccessful (returning a value that is not 0 and bypassing the jnz instruction). As previously stated, if the comparison is unsuccessful then the function will call sub_100052A2. We notice in the subroutine located at 0x10052A2 that there is a push instruction of the variable ‘aSoftwareMicros’ followed by the external function call of ‘RegOpenKeyExA’ (Figure 45).

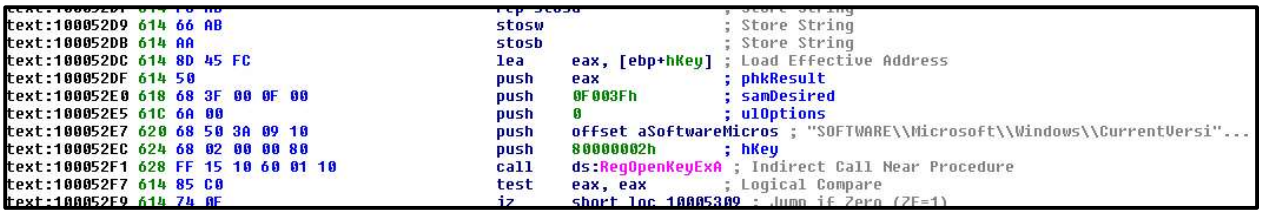


Figure 45: Code in sub_100052A2.

aSoftwareMicros is a variable defined in xdoors address 0x10093A50 and is the string of 'SOFTWARE\Microsoft\Windows\CurrentVersion', a registry key folder within the HKLM hive (Figure 46).

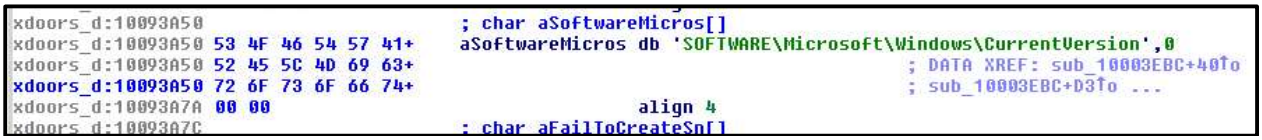


Figure 46: aSoftwareMicros definition.

We also see that in addition to the aSoftwareMicros variable string is pushed onto the stack, there are multiple other values pushed prior to the RegOpenKeyExA function is called, which opens a specified registry key. It does a logical comparison of EAX to itself and jumps to 0x1005309 if it successfully opens HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion.

Within that memory location of 0x1005309, we see multiple external function calls and data being passed into them. In one section, we see the string of "WorkTime" being pushed onto the stack prior to a call to the external function 'RegQueryValueExA' (Figure 47). This function will retrieve the type and data for the specified value name with an open registry key (Microsoft.com). This tells us that the program is querying the registry key of HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime and retrieving the data. This data is then placed within the variable 'aRobot_worktime' at 0x1000534C, containing the string, "\r\n\r\n[Robot_WorkTime :] %d\r\n\r\n".

```
text:1000531B 620 50                      push    eax             ; lpData
text:1000531C 624 8D 45 F8                lea     eax, [ebp+Type] ; Load Effective Address
text:1000531F 624 50                      push    eax             ; lpType
text:10005320 628 6A 00                   push    0               ; lpReserved
text:10005322 62C 68 E0 40 09 10          push    offset aWorktime ; "WorkTime"
text:10005327 630 FF 75 FC                push    [ebp+hKey]      ; hKey
text:1000532A 634 FF D3                   call    ebx ; RegQueryValueExA ; Indirect Call Near Procedure
text:1000532C 61C 8B 35 F4 62 01 10       mov     esi, ds:sprintf
text:10005332 61C 8B 3D B4 62 01 10       mov     edi, ds:atoi
text:10005338 61C 85 C0                   test    eax, eax        ; Logical Compare
text:1000533A 61C 75 3D                   jnz     short loc_10005379 ; Jump if Not Zero (ZF=0)
text:1000533C 61C 8D 85 F4 FD FF FF       lea     eax, [ebp+Data] ; Load Effective Address
text:10005342 61C 50                      push    eax             ; char *
text:10005343 620 FF D7                   call    edi ; atoi      ; Indirect Call Near Procedure
text:10005345 620 50                      push    eax
text:10005346 624 8D 85 F4 F9 FF FF       lea     eax, [ebp+var_60C] ; Load Effective Address
text:1000534C 624 68 C0 40 09 10          push    offset aRobot_worktime ; "\r\n\r\n[Robot_WorkTime :] %d\r\n\r\n"
text:10005351 628 50                      push    eax             ; char *
```

Figure 47: aSoftwareMicros definition.

After the value is placed within the string, we see a call to sub_100038EE (Figure 48).

```
.text:10005343 620 FF D7                  call    edi ; atoi      ; Indirect Call Near Procedure
.text:10005345 620 50                     push    eax
.text:10005346 624 8D 85 F4 F9 FF FF      lea     eax, [ebp+var_60C] ; Load Effective Address
.text:1000534C 624 68 C0 40 09 10         push    offset aRobot_worktime ; "\r\n\r\n[Robot_WorkTime :] %d\r\n\r\n"
.text:10005351 628 50                     push    eax             ; char *
.text:10005352 62C FF D6                  call    esi ; sprintf   ; Indirect Call Near Procedure
.text:10005354 62C 83 C4 10               add     esp, 10h        ; Add
.text:10005357 61C 8D 85 F4 F9 FF FF      lea     eax, [ebp+var_60C] ; Load Effective Address
.text:1000535D 61C 6A 00                  push    0
.text:1000535F 620 50                     push    eax             ; char *
.text:10005360 624 E8 E7 FB 00 00         call    strlen          ; Call Procedure
.text:10005365 624 59                     pop     ecx
.text:10005366 620 50                     push    eax             ; int
.text:10005367 624 8D 85 F4 F9 FF FF      lea     eax, [ebp+var_60C] ; Load Effective Address
.text:1000536D 624 50                     push    eax             ; int
.text:1000536E 628 FF 75 08               push    [ebp+s]         ; s
.text:10005371 62C E8 78 E5 FF FF         call    sub_100038EE    ; Call Procedure
.text:10005376 62C 83 C4 10               add     esp, 10h        ; Add
```

Figure 48: Call to sub_100038EE.

Within this subroutine at 0x100038EE, we see the parameters defined as “(SOCKET s, int, int)” (Figure 49). Based on the push commands prior to this subroutine being called in Figure 48, we know that the string within ‘aRobot_worktime’ was passed into this subroutine. Within this subroutine, we see a call to the function ‘send’ of the winsock2.h library at 0x10003933 (Figure 50). This means that the value of WorkTime returned by RegQueryValueExA and placed in the aRobot_worktime variable will be sent over the socket to the receiving end of the remote shell. The function then returns to the next line of code after its original call, returning us to 0x10005376 (Figure 48).

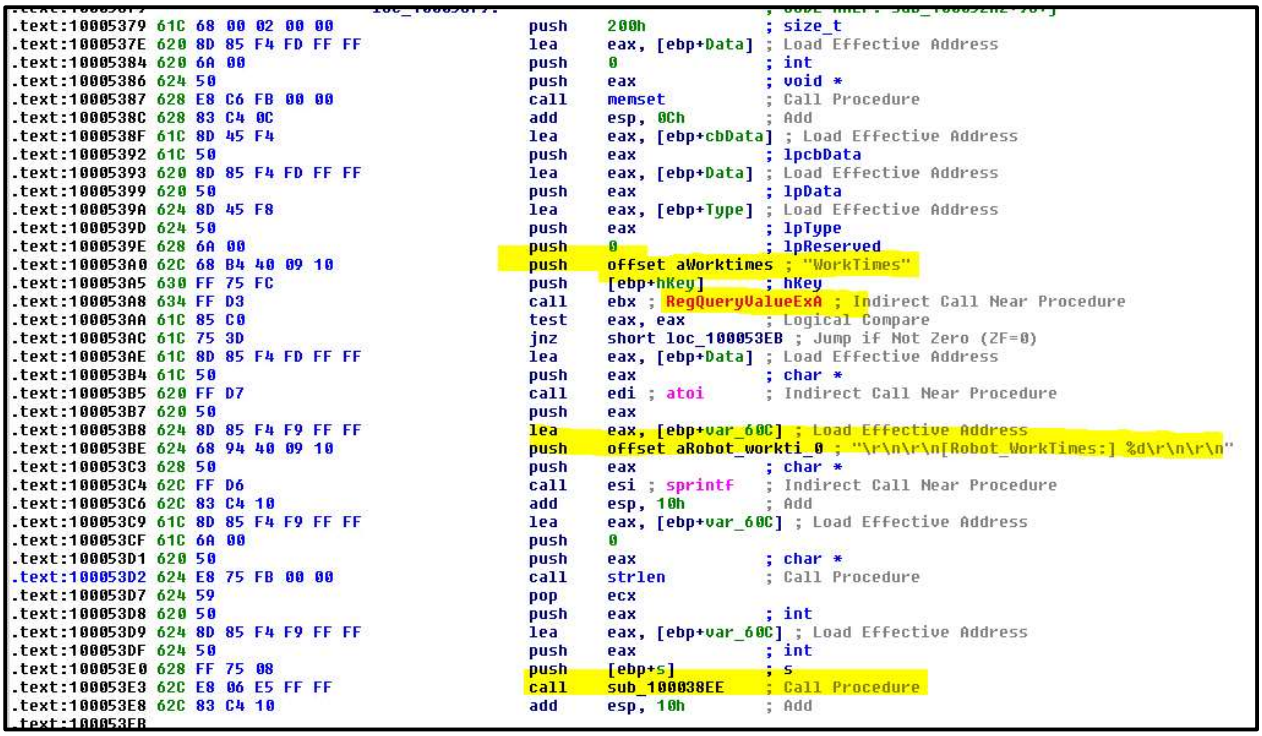
```
.text:100038ED                  sub_100038BB    endp
.text:100038ED
.text:100038EE
.text:100038EE                  ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
.text:100038EE
.text:100038EE                  ; Attributes: bp-based frame
.text:100038EE
.text:100038EE                  ; int __cdecl sub_100038EE(SOCKET s,int,int)
.text:100038EE                  sub_100038EE    proc near               ; CODE XREF: sub_10004CFF+BE↓p
.text:100038EE                                                          ; sub_10004DCA+A2↓p ...
.text:100038EE
.text:100038EE                  s               = dword ptr  8
.text:100038EE                  arg_4           = dword ptr  0Ch
.text:100038EE                  arg_8           = dword ptr  10h
```

Figure 49: sub_100038EE parameters.

```
.text:10003928
.text:10003928                          loc_10003928:                           ; CODE XREF: sub_100038EE+
.text:10003928 00C 80 24 32 00                  and     byte ptr [edx+esi], 0 ; Logical AND
.text:1000392C 00C 6A 00                        push    0                       ; flags
.text:1000392E 010 57                           push    edi                     ; len
.text:1000392F 014 56                           push    esi                     ; buf
.text:10003930 018 FF 75 08                     push    [ebp+s]                 ; s
.text:10003933 01C FF 15 D8 63 01 10            call    ds:send                 ; Indirect Call Near Proce
.text:10003939 00C 83 CF FF                     or      edi, 0FFFFFFFFh ; Logical Inclusive OR
.text:1000393C 00C 3B C7                        cmp     eax, edi                ; Compare Two Operands
.text:1000393E 00C 74 02                        jz      short loc_10003942 ; Jump if Zero (ZF=1)
.text:10003940 00C 8B F8                        mov     edi, eax
```

Figure 50: sub_100038EE sending data over the socket.

Once the function returns, we notice another block of code that is nearly-identical, but the registry key queried is ‘WorkTimes’. The call to sub_100038EE is then called and performs the same actions as it did for ‘WorkTime’ detailed above (Figure 51).



```
.text:10005379 61C 68 00 02 00 00                  push    200h            ; size_t
.text:1000537E 620 8D 85 F4 FD FF FF               lea     eax, [ebp+Data] ; Load Effective Address
.text:10005384 620 6A 00                           push    0               ; int
.text:10005386 624 50                              push    eax             ; void *
.text:10005387 628 E8 C6 FB 00 00                  call    memset          ; Call Procedure
.text:1000538C 628 83 C4 0C                        add     esp, 0Ch        ; Add
.text:1000538F 61C 8D 45 F4                        lea     eax, [ebp+cbData] ; Load Effective Address
.text:10005392 61C 50                              push    eax             ; lpcbData
.text:10005393 620 8D 85 F4 FD FF FF               lea     eax, [ebp+Data] ; Load Effective Address
.text:10005399 620 50                              push    eax             ; lpData
.text:1000539A 624 8D 45 F8                        lea     eax, [ebp+Type] ; Load Effective Address
.text:1000539D 624 50                              push    eax             ; lpType
.text:1000539E 628 6A 00                           push    0               ; lpReserved
.text:100053A0 62C 68 B4 40 09 10                  push    offset aWorktimes ; "WorkTimes"
.text:100053A5 630 FF 75 FC                        push    [ebp+hKey]      ; hKey
.text:100053A8 634 FF D3                           call    ebx ; RegQueryValueExA ; Indirect Call Near Procedure
.text:100053AA 61C 85 C0                           test    eax, eax        ; Logical Compare
.text:100053AC 61C 75 3D                           jnz     short loc_100053EB ; Jump if Not Zero (ZF=0)
.text:100053AE 61C 8D 85 F4 FD FF FF               lea     eax, [ebp+Data] ; Load Effective Address
.text:100053B4 61C 50                              push    eax             ; char *
.text:100053B5 620 FF D7                           call    edi ; atoi      ; Indirect Call Near Procedure
.text:100053B7 620 50                              push    eax
.text:100053B8 624 8D 85 F4 F9 FF FF               lea     eax, [ebp+var_60C] ; Load Effective Address
.text:100053BE 624 68 94 40 09 10                  push    offset aRobot_workti_0 ; "\r\n\r\n[Robot_WorkTimes:] %d\r\n\r\n"
.text:100053C3 628 50                              push    eax             ; char *
.text:100053C4 62C FF D6                           call    esi ; sprintf   ; Indirect Call Near Procedure
.text:100053C6 62C 83 C4 10                        add     esp, 10h        ; Add
.text:100053C9 61C 8D 85 F4 F9 FF FF               lea     eax, [ebp+var_60C] ; Load Effective Address
.text:100053CF 61C 6A 00                           push    0
.text:100053D1 620 50                              push    eax             ; char *
.text:100053D2 624 E8 75 FB 00 00                  call    strlen          ; Call Procedure
.text:100053D7 624 59                              pop     ecx
.text:100053D8 620 50                              push    eax             ; int
.text:100053D9 624 8D 85 F4 F9 FF FF               lea     eax, [ebp+var_60C] ; Load Effective Address
.text:100053DF 624 50                              push    eax             ; int
.text:100053E0 628 FF 75 08                        push    [ebp+s]         ; s
.text:100053E3 62C E8 06 E5 FF FF                  call    sub_100038EE    ; Call Procedure
.text:100053E8 62C 83 C4 10                        add     esp, 10h        ; Add
.text:100053EB
```

Figure 51: S*ame code for 'WorkTimes' key as for 'WorkTime' key.*

LAB 5-1 Question 11

What does the export PSLIST do?

BLUF: Returns process information to the remote shell user.

Double-clicking on the PSLIST export within the Exports window, we are taken to 0x10007025. We notice three subroutines: sub_100036C3 at 0x1000702F, sub_10006518 at 0x10007047, and sub_1000664C at 0x10007054 (Figure 52).

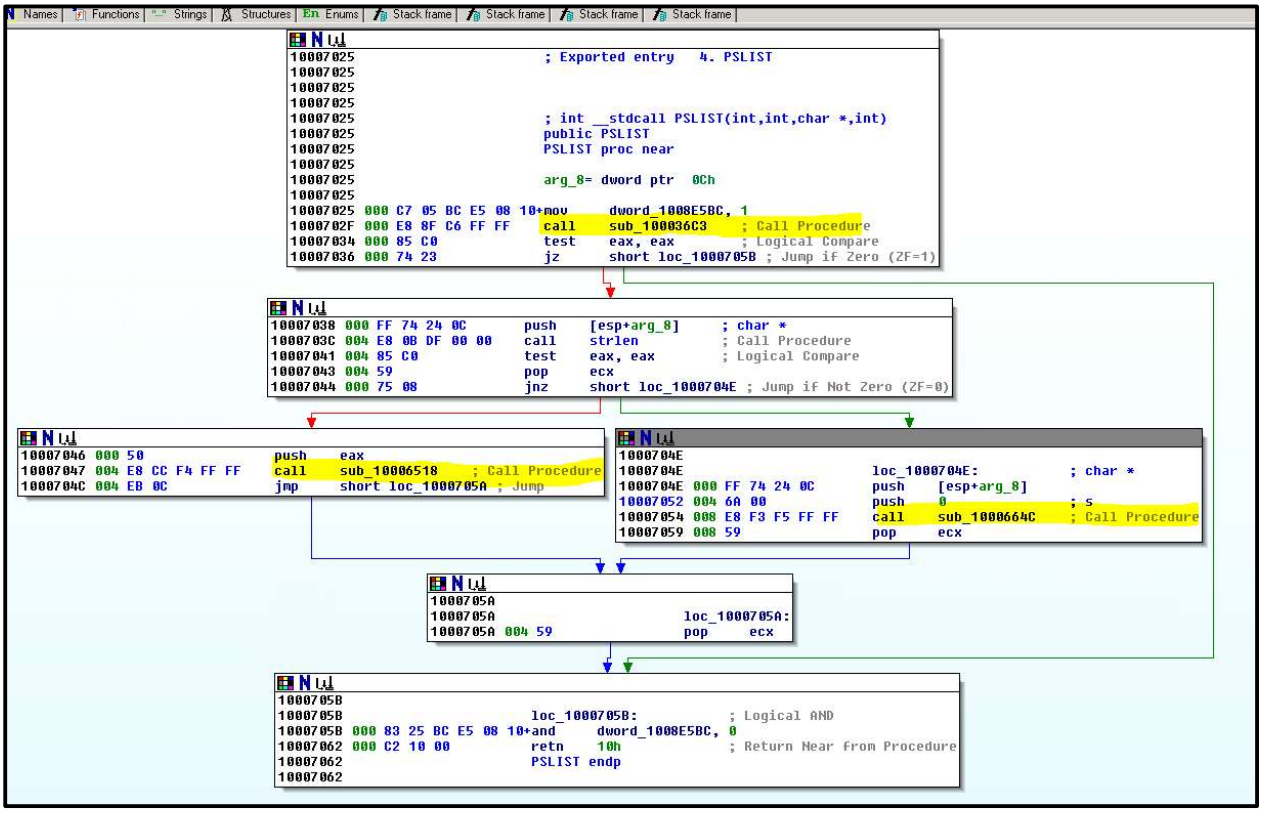


Figure 52: Code for PSLIST.

sub_100036C3 is the first to be called and simply gets the OS version, similar to sub_10003695 as evaluated in question 9 (Figure 53). However, we also see at 0x100036EC a cmp instruction for dwMajorVersion 5. This is the major version number of the operating system. A table provided by Microsoft about these versions can be found here. OS-es from Windows 200 to

Windows Server 2003 R2 are major version 5. If the major version is less than 5, then it pushes 1 and pops EAX. If not, then is XOR's EAX and returns to the next line of code after the subroutine was called.

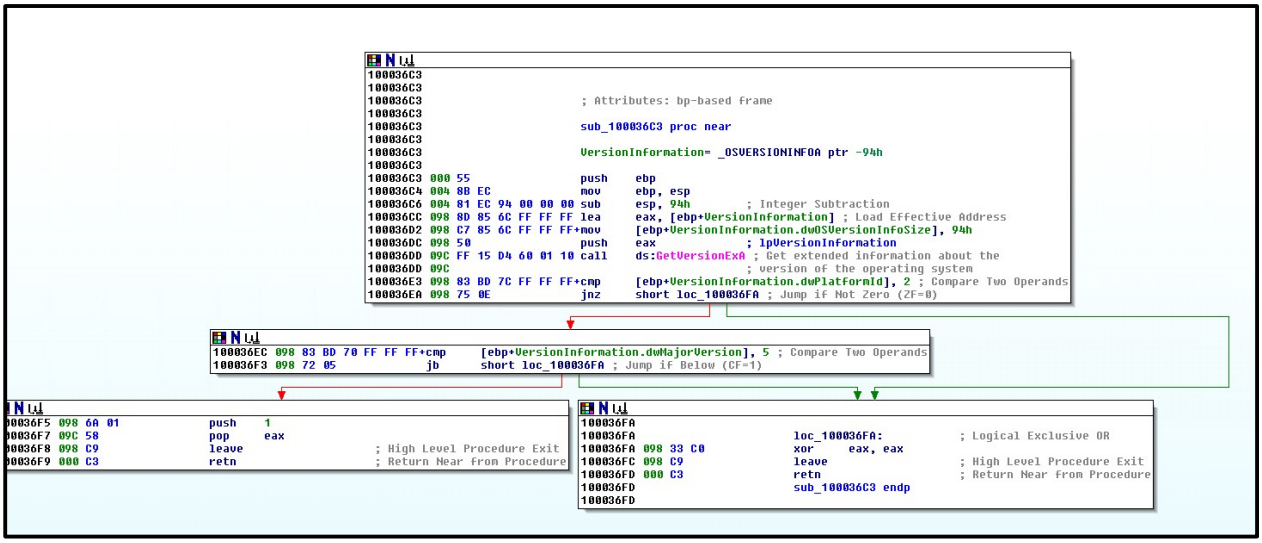


Figure 53: sub_100036C3 getting OS information.

In Figure 52, a logical comparison is conducted on EAX with itself. If the comparison is unsuccessful (the Zero-flag is set), the code goes to 0x1000705B and returns the hex value 10 (decimal 16). If the logical comparison is successful, meaning the major version of the OS that the program gathered is 5 or greater (gathered from sub_100036C3), then it continues to 0x10007038. It will then test the string length of the string the function was passed (the char * parameter). If the string IS NOT empty, then it will continue on the left path in Figure 52 and call sub_10006518. If the string IS empty, it will continue on the right path in Figure 52 and call sub_1000664C.

If the string IS NOT empty, sub_10006518 is called. We notice a call to the function 'CreateToolhelp32Snapshot' at 0x10006558. From the subsequent strings and API calls within

sub_10006518, we can determine that the return value from this subroutine is a string that displays the Process ID, Process Name, and ThreadNumber of each process it queries (Figure 54). We also know that from our previous analysis of other functions within this program that it is most likely sent over the socket to the remote shell user.

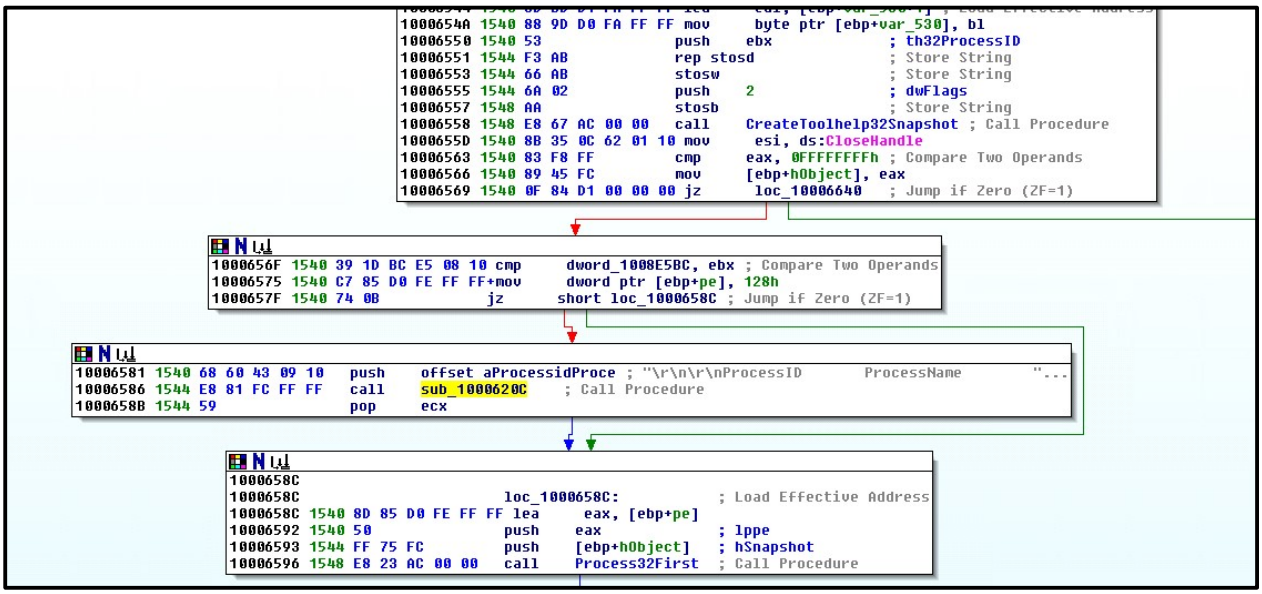


Figure 54: Portion of sub_10006518.

It also appears that sub_1000664C does the same thing, albeit in a different manner. We see the same call to 'CreateToolhelp32Snapshot' at 0x10006669F and the same string variable to store the process information at 0x100066DF (Figure 55).

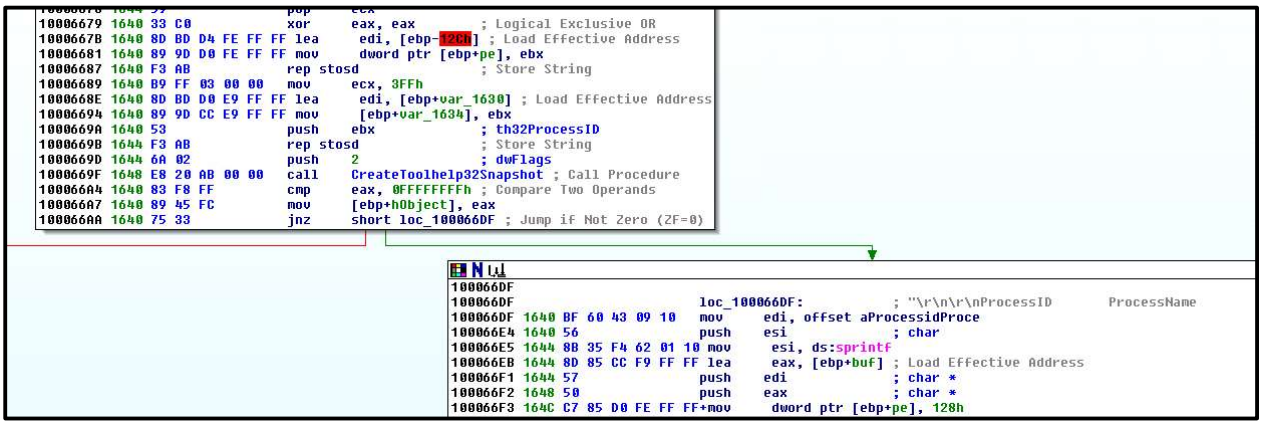


Figure 54: Portion of sub_1000664C.

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?

BLUF:

Navigating to sub_10004E79, we can get a high-level overview of the cross-references of the subroutine by going to View > Graphs > Xrefs From. We notice that there are four directly-called functions: GetSystemDefaultLangID, sprintf, sub_100038EE, and strlen. sub_100038EE calls send, malloc, and free. strlen calls __imp_strlen.

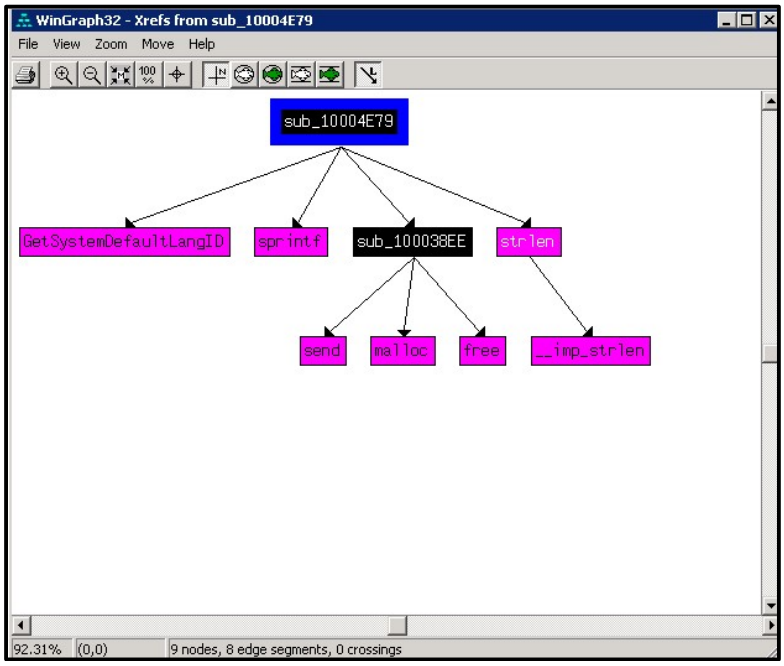


Figure 55: Cross-references from sub_10004E79.

From these calls, the two that stand out the most are 'GetSystemDefaultLangID' and 'send'. The GetSystemDefaultLangID function returns the language identifier of the system. We can interpret from this information that the program is getting the language identifier/setting of the system and sends it to the user at the other end of the secure shell. This can help the user, especially if they

are an attacker, determine if the system is configured in a language they are familiar with. If the attacker is not familiar with that language, then they need to make a determination if attacking this particular system is worth the time and resources to translate the language.

LAB 5-1 Question 13

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

BLUF: Four directly, 32 at depth 2.

We navigate back to 0x1000D02E and open the same XRef window as we did in Question 12. But we find that the depth of the chart is too broad to meaningfully analyze. So, we open the User Xrefs Chart and set the recursion depth to 1 and uncheck the 'Cross references to' box. We then can see the Windows API calls highlighted in pink: CreateThread, strncpy, strlen, and _strnicmp (Figure 56).

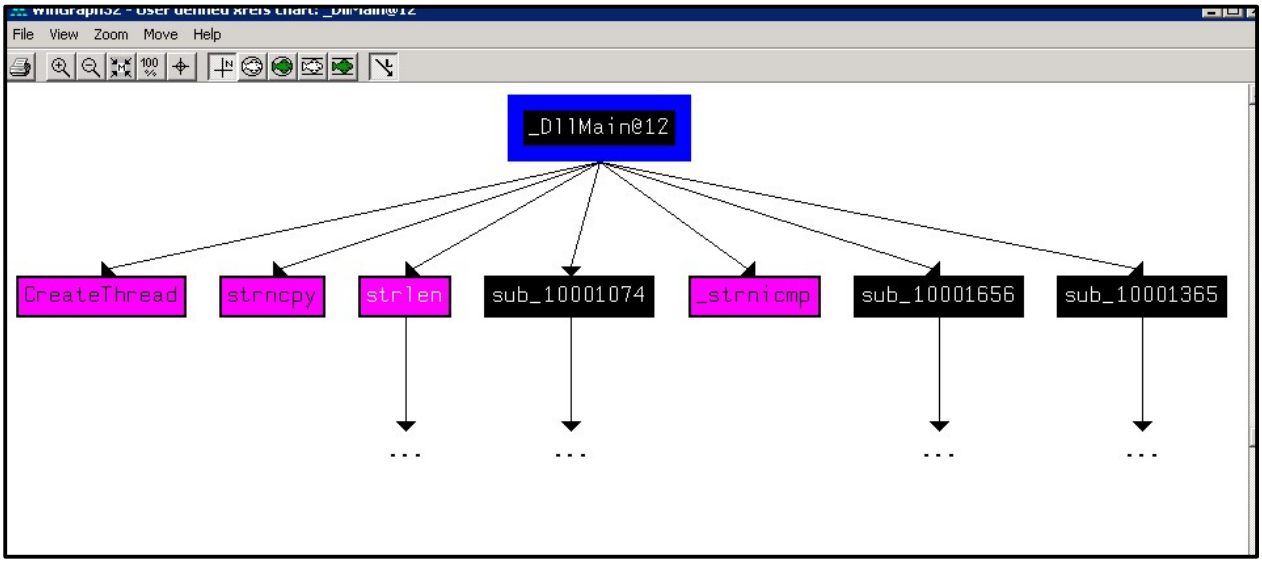


Figure 56: Depth of 1 Windows API calls in pink.

By opening up another window but setting our depth to 2, we get a much larger graph that is unwieldy to display effectively on this report (Figure 57). DllMain is circled in red and we can see that just to the upper-right of it there is a subroutine that connects to it, indicating recursiveness. Zooming in on the graph, we can gather the depth 2 API calls: strcpy, strncpy, strchr, memcpy, gethostbyname, inet_ntoa, WinExec, Sleep, __imp_strlen (a subset of the depth

1 strlen), CreateThread, closesocket, strncmp, FreeLibrary, memset, inet_addr, atoi, select, __imp_printf, connect, WSAGetLastError, recv, CloseHandle, socket, send, ntohs, GetTickCount, WSAStartup, LoadLibraryA, memcmp, GetProcAddress, and ExitThread. That makes 32 depth 2 Windows API calls.

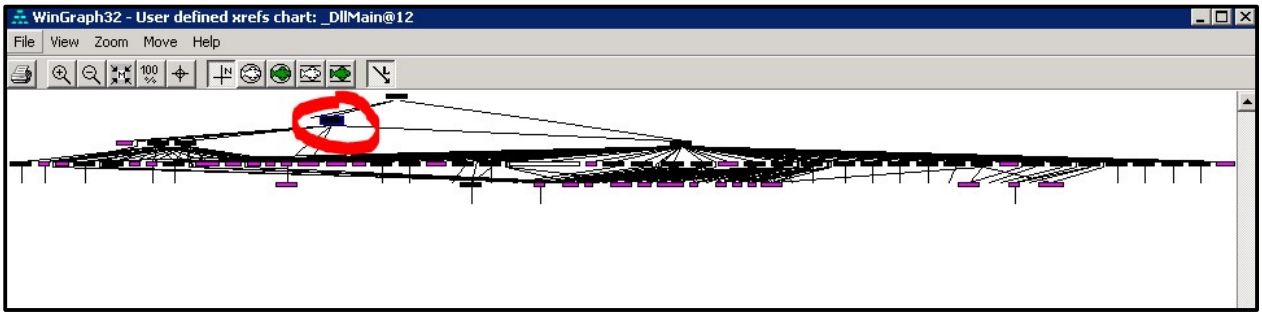


Figure 57: Depth of 1 Windows API calls in pink.

Exploring the depth of Windows API calls and other function calls at different depths is extremely interesting because it allows a high-level overview of a program's functionality. Simply by seeing which functions call which Windows API calls (and other functions), an analyst can glean information as to how the program runs and what its potential functionality may be. It can also see how each function is related and identify areas of recursion (as pointed out in Figure 57). Although pressing spacebar to access graph view within IDA View is helpful, this window allows a user to gain a visual representation of the entire structure of a program. As the saying goes, "you can't see the forest from the trees," this window provides a view from the top of a single tree or from a helicopter; the analyst just chooses the altitude (the "depth" in this analogy).

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?

BLUF: 30 Seconds.

In Figure 58, we notice the call to Sleep is preceded by a push EAX call, meaning that whatever value was in EAX prior to the call is the number of milliseconds the program is to sleep for.

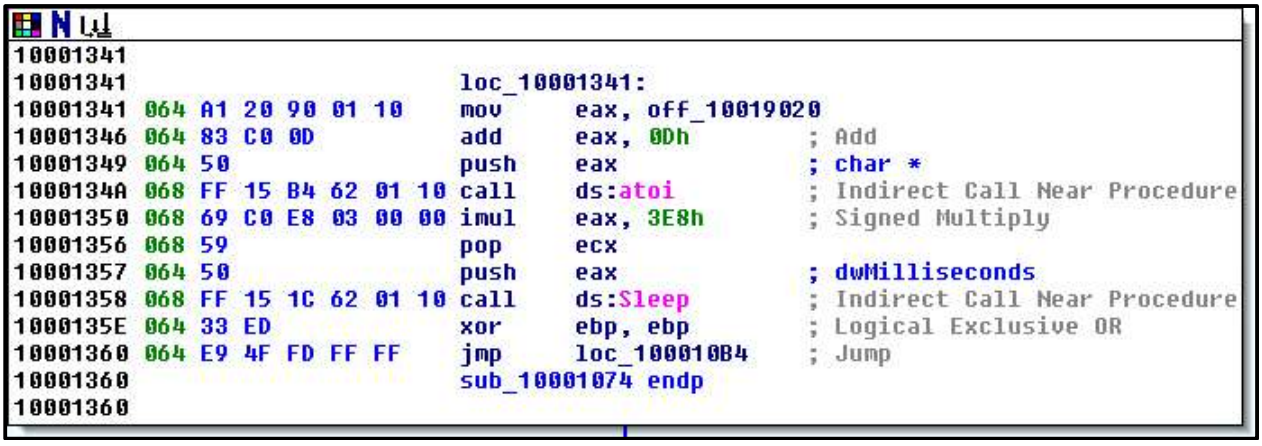


Figure 58: Function at 0x10001358.

We notice at 0x10001350 that EAX is multiplied by hex value 3E8, which is 1000 in decimal (Figure 59). We also see in Figure 58 at 0x10001341 that the variable 'off_10019020' moved into EAX and then a hex value of 'D' (decimal 13) is added to it. This is most likely an offset like we determined in question 4 to find the domain name of pics.practicalmalwareanalysis.com. Navigating to the address of the off_10019020 variable, we see it defined in the .data section as 'offset unk_100192AC' (Figure 60).

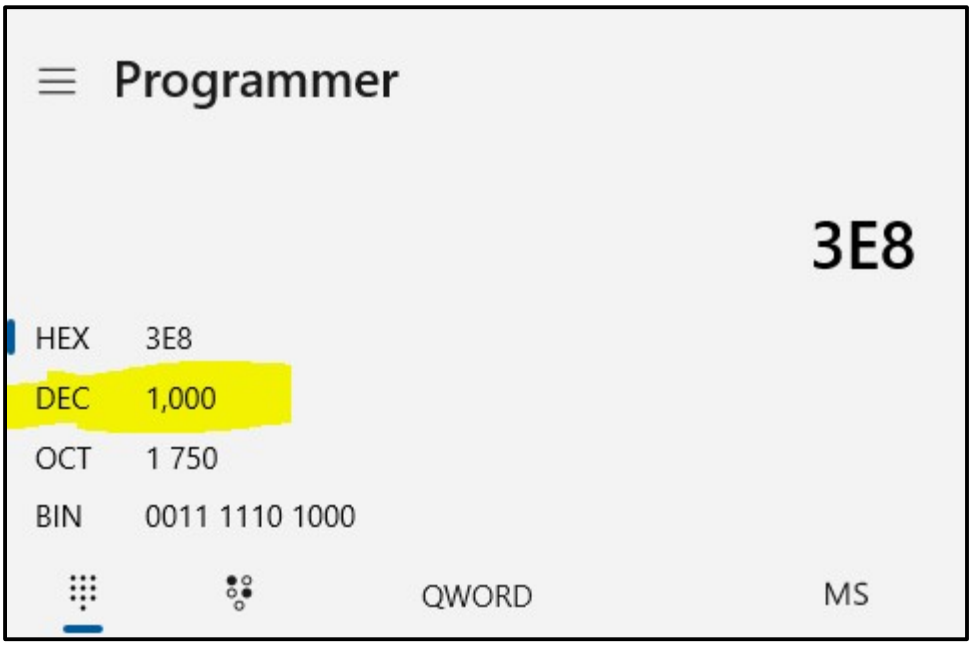


Figure 59: Hex 3E8 = 1000 decimal.

```
.data:10019020 AC 92 01 10          off_10019020    dd offset unk_100192AC  ; DATA XREF: sub_10001074:loc_10001341↑r
.data:10019020                                                              ; sub_10001365:loc_10001632↑r ...
.data:10019024 98 92 01 10          off_10019024    dd offset aThisIsNti30  ; DATA XREF: sub_10001656+357↑r
.data:10019024                                                              ; sub_10004EE0+227↑r
```

Figure 60: off_10019020 defined.

Navigating to unk_100192AC, the string did not appear in a usable form and did not include the proper value (Figure 61).

```
data:100192AB 00                    db    0
data:100192AC 5B    unk_100192AC    db  5Bh ; [            ; DATA XREF: .data:off_10019020↑o
data:100192AD 54                    db  54h ; T
data:100192AE 68                    db  68h ; h
data:100192AF 69                    db  69h ; i
data:100192B0 73                    db  73h ; s
data:100192B1 20                    db  20h
data:100192B2 69                    db  69h ; i
data:100192B3 73                    db  73h ; s
data:100192B4 20                    db  20h
data:100192B5 43                    db  43h ; C
data:100192B6 54                    db  54h ; T
data:100192B7 49                    db  49h ; I
data:100192B8 5D                    db  5Dh ; ]
```

Figure 61: Windows XP IDA Freeware not able to define the variable.

However, when it was loaded into the newer IDA Freeware on a Windows 10 machine, we can see that the string of this variable is '[This is CTI]30' (Figure 62). This means that the first 13 characters are ignored due to the offset and the string 30 is moved into EAX. The 'atoi' function

call converts the string into an integer and then is multiplied by 1000. 30,000 milliseconds equals 30 seconds, therefore the program will sleep for 30 seconds.

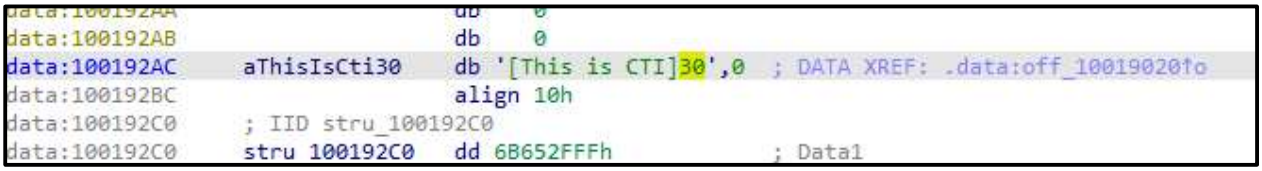


Figure 62: The variable in question.

LAB 5-1 Question 15

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

BLUF: The address family, type specification, and protocol.

Prior to the external function 'socket' calls, there are three integer values that are pushed onto the stack (Figure 63). According to the Microsoft documentation of this function, this function is part of the winsock2.h library.

```
.text:100016FB 688 6A 06                          push    6               ; protocol
.text:100016FD 68C 6A 01                          push    1               ; type
.text:100016FF 690 6A 02                          push    2               ; af
.text:10001701 694 FF 15 F8 63 01 10              call    ds:socket       ; Indirect Call Near Procedure
.text:10001707 688 8B F8                          mov     edi, eax
.text:10001709 688 83 FF FF                       cmp     edi, 0FFFFFFFFh ; Compare Two Operands
.text:1000170C 688 75 14                          jnz     short loc_10001722 ; Jump if Not Zero (ZF=0)
.text:1000170E 688 FF 15 FC 63 01 10              call    ds:WSAGetLastError ; Indirect Call Near Procedure
.text:10001714 688 50                             push    eax
.text:10001715 68C 68 54 35 09 10                 push    offset aSocketGetlaste ; "socket() GetLastError reports %d\n"
.text:1000171A 690 FF 15 A8 62 01 10              call    ds:__imp_printf ; Indirect Call Near Procedure
.text:10001720 690 59                             pop     ecx
.text:10001721 68C 59                             pop     ecx
```

Figure 63: Code surrounding address 0x10001701.

The documentation shows the syntax for this function as taking three parameters: int af, int type, and int protocol (Figure 64). When values are pushed onto the stack for a function, the first value that is pushed onto the stack gets placed in the last variable. Therefore, the values for the variables are as follows: int af = 2; int type = 1; int protocol = 6. This is confirmed by the annotations provided by IDA Pro in Figure 63.

Syntax

C++ Copy

```
SOCKET WSAAPI socket(
  [in] int af,
  [in] int type,
  [in] int protocol
);
```

Figure 64: Syntax for 'socket' function.

Referencing the documentation again, we can see that based on these parameters, the socket will be configured as part of the IPv4 address family (Figure 65), have out of band (OOB) transmission (Figure 66), and use the TCP protocol for the communication (Figure 67). TCP over IPv4 is commonly used for HTTP connections.

AF_INET 2	The Internet Protocol version 4 (IPv4) address family.

Figure 65: Syntax for 'socket' function.

Type	**Meaning**
SOCK_STREAM 1	A socket type that provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism. This socket type uses the Transmission Control Protocol (TCP) for the Internet address family (AF_INET or AF_INET6).

Figure 66: Syntax for 'socket' function.

IPPROTO_TCP 6	The Transmission Control Protocol (TCP). This is a possible value when the *af* parameter is **AF_INET** or **AF_INET6** and the *type* parameter is **SOCK_STREAM**.

Figure 67: Syntax for 'socket' function.

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?

BLUF: AF_INET, SOCK_STREAM, and IPPROTO_TCP.

Like we identified in question 15, we can make these parameters more meaningful by renaming them in accordance with the documentation as AF_INET, SOCK_STREAM, and IPPROTO_TCP. We do this by right-clicking on the value and selecting "Use Symbolic Constant". A window pops up titled, "Please choose a symbol." We simply scroll down until we find the proper label or search for the string. An example of finding the IPPROTO_TCP symbol is given in Figure 68. The completed changes are shown in Figure 69 for the parameters and can be compared with the "before" Figure 63.

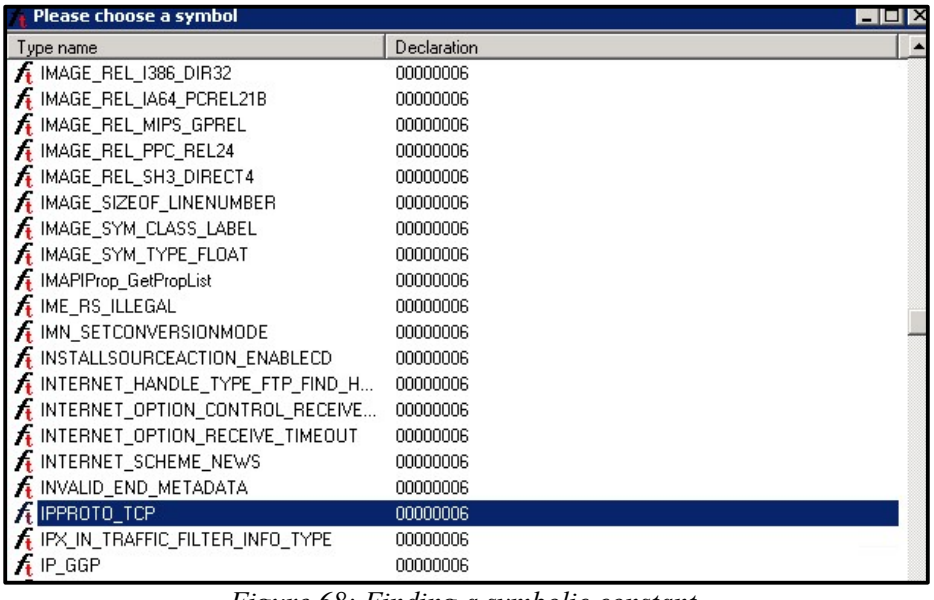


Figure 68: Finding a symbolic constant.

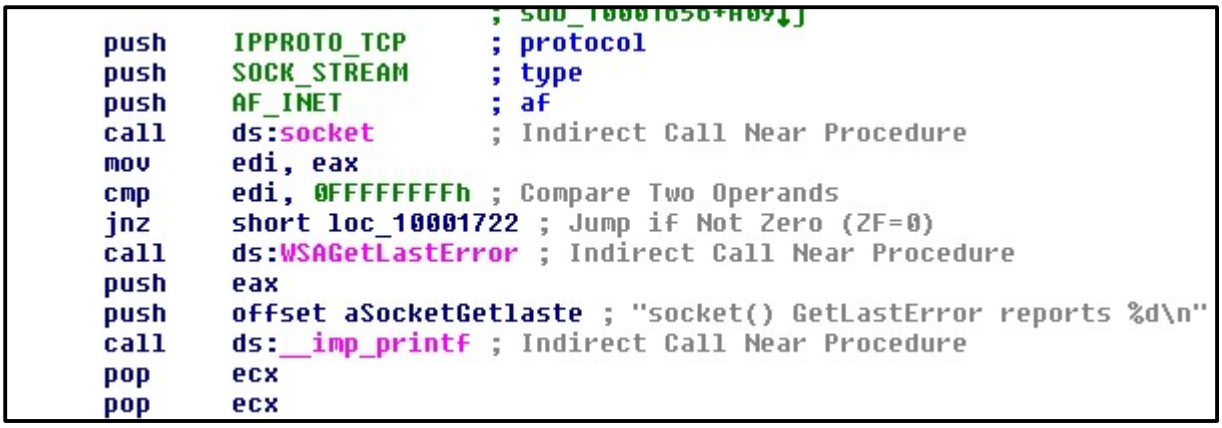


Figure 69: Symbolic constants applied.

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?

BLUF: Yes. Three installer functions leverage the VM detection and cancels installation if one is detected.

When searching for "ED" with Search > Sequence of Bytes, an 'in' instruction was found at 0x100061DB (Figure 70).

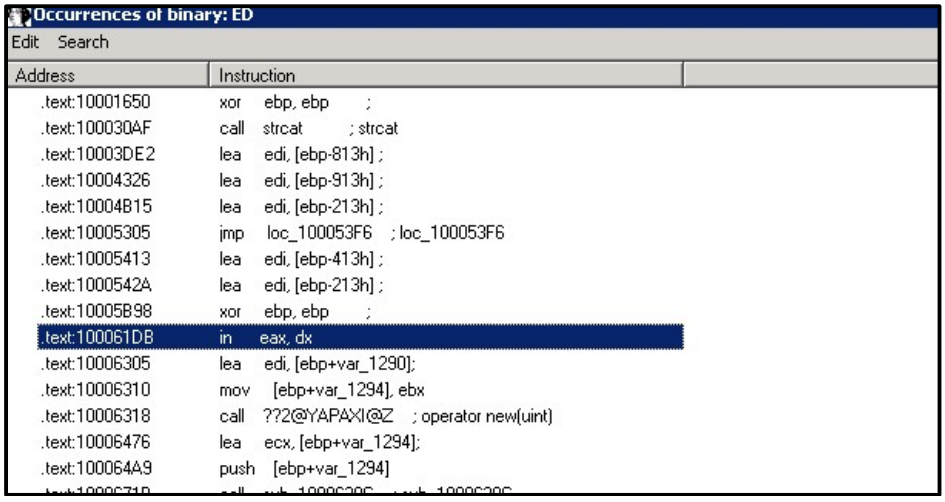


Figure 70: Occurrences of binary "ED".

When navigating to the location, the IDA Pro comment did not populate like it showed in the book at 0x100061C7 (Figure 71). But the hex value matched and when the ASCII string "VMXh" was placed into an ASCII-to-hex converter, the same value of 564D5868 was outputted (Figure 72). Not being able to replicate the output of certain portions of the book demonstrations is certainly annoying, but noting the hex value for VMXh is important to keep in mind for future

reference. But there is a functionality within IDA where you can click on the hex value, press "R", and the hex value will change to ASCII-representation (Figure 73).

```
.text:100061C6 034 53                                  push    ebx
.text:100061C7 038 B8 68 58 4D 56                      mov     eax, 564D5868h
.text:100061CC 038 BB 00 00 00 00                      mov     ebx, 0
.text:100061D1 038 B9 0A 00 00 00                      mov     ecx, 0Ah
.text:100061D6 038 BA 58 56 00 00                      mov     edx, 5658h
.text:100061DB 038 ED                                  in      eax, dx
.text:100061DC 038 81 FB 68 58 4D 56                   cmp     ebx, 564D5868h  ; Compare
```

Figure 71: "VMXh" not appearing in IDA-generated comments.

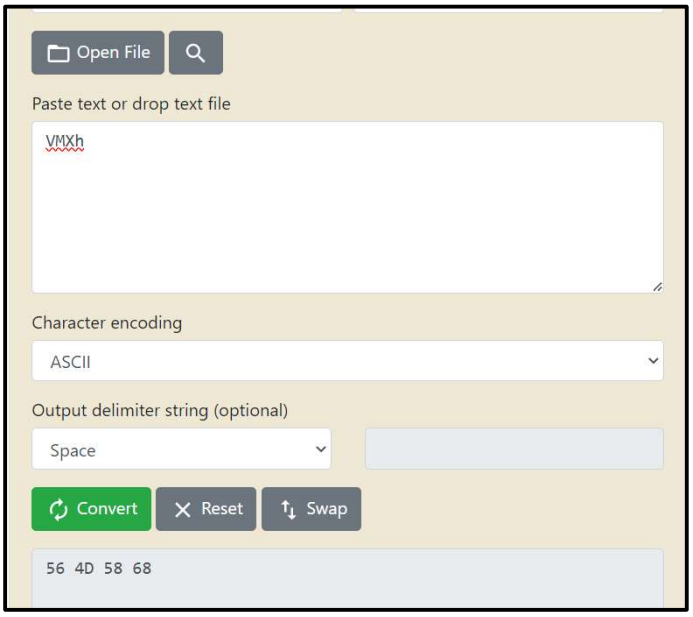


Figure 72: VMXh converted to hex.

```
        push    edi
        mov     [ebp+var_18], esp
        mov     [ebp+var_1C], 1
        and     [ebp+var_4], 0  ; Logical AND
        push    edx
        push    ecx
        push    ebx
        mov     eax, 'VMXh'
        mov     ebx, 0
        mov     ecx, 0Ah
        mov     edx, 'VX'
        in      eax, dx
        cmp     ebx, 'VMXh'     ; Compare Two Operands
        setz    [ebp+var_1C]    ; Set Byte if Zero (ZF=
        pop     ebx
        pop     ecx
        pop     edx
```

Figure 73: Press "R" to change a hex value to ASCII.

By checking cross-references to this function, we can check to see if this “magic string” which is used for VM detection is actually used within the code for anti-VM checks. We see that there are three cross-references for this function within the program (Figure 74).

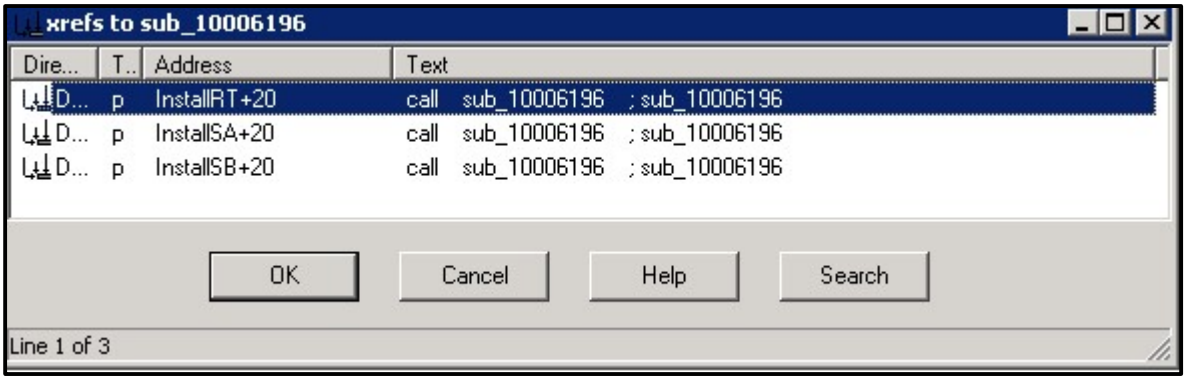


Figure 74: XRefs for anti-VM function.

The first call in Figure 74 occurs at 0x1000D867 and the string “Found Virtual Machine, Install Cancel” is displayed (Figure 75). We the same thing for the other two XRefs (Figures 76 and 77).

```
:1000D863 004 84 C0                           test    al, al          ; Logical Compare
:1000D865 004 75 09                           jnz     short loc_1000D870 ; Jump if Not Zero (ZF=0)
:1000D867 004 E8 2A 89 FF FF                  call    sub_10006196    ; Call Procedure
:1000D86C 004 84 C0                           test    al, al          ; Logical Compare
:1000D86E 004 74 1E                           jz      short loc_1000D88E ; Jump if Zero (ZF=1)
:1000D870
:1000D870                         loc_1000D870:                       ; CODE XREF: InstallRT+1E↑j
:1000D870 004 68 F0 E5 08 10                  push    offset unk_1008E5F0 ; char *
:1000D875 008 E8 18 5D FF FF                  call    sub_10003592    ; Call Procedure
:1000D87A 008 C7 04 24 88 4F 09+              mov     [esp+8+var_8], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
:1000D881 008 E8 0C 5D FF FF                  call    sub_10003592    ; Call Procedure
:1000D886 008 59                              pop     ecx
:1000D887 004 E8 DB 7C FF FF                  call    sub_10005567    ; Call Procedure
:1000D88C 004 EB 16                           jmp     short loc_1000D8A4 ; Jump
```

Figure 75: Install cancel for first XRef.

```
text:1000DEDF 004 75 09                       jnz     short loc_1000DEEA ; Jump if Not Zero (ZF=0)
text:1000DEE1 004 E8 B0 82 FF FF              call    sub_10006196    ; Call Procedure
text:1000DEE6 004 84 C0                       test    al, al          ; Logical Compare
text:1000DEE8 004 74 1E                       jz      short loc_1000DF08 ; Jump if Zero (ZF=1)
text:1000DEEA
text:1000DEEA                     loc_1000DEEA:                       ; CODE XREF: InstallSA+1E↑j
text:1000DEEA 004 68 F0 E5 08 10              push    offset unk_1008E5F0 ; char *
text:1000DEEF 008 E8 9E 56 FF FF              call    sub_10003592    ; Call Procedure
text:1000DEF4 008 C7 04 24 88 4F 09+          mov     [esp+8+var_8], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
text:1000DEFB 008 E8 92 56 FF FF              call    sub_10003592    ; Call Procedure
text:1000DF00 008 59                          pop     ecx
text:1000DF01 004 E8 61 76 FF FF              call    sub_10005567    ; Call Procedure
text:1000DF06 004 EB 16                       jmp     short loc_1000DF1E ; Jump
```

Figure 76: Install cancel for second XRef.

```
.text:1000E8B0 004 75 09                      jnz     short loc_1000E8BB ; Jump if Not Zero (ZF=0)
.text:1000E8B2 004 E8 DF 78 FF FF             call    sub_10006196    ; Call Procedure
.text:1000E8B7 004 84 C0                      test    al, al          ; Logical Compare
.text:1000E8B9 004 74 1E                      jz      short loc_1000E8D9 ; Jump if Zero (ZF=1)
.text:1000E8BB
.text:1000E8BB                    loc_1000E8BB:                       ; CODE XREF: InstallSB+1E↑j
.text:1000E8BB 004 68 F0 E5 08 10             push    offset unk_1008E5F0 ; char *
.text:1000E8C0 008 E8 CD 4C FF FF             call    sub_10003592    ; Call Procedure
.text:1000E8C5 008 C7 04 24 88 4F 09+         mov     [esp+8+var_8], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
.text:1000E8CC 008 E8 C1 4C FF FF             call    sub_10003592    ; Call Procedure
.text:1000E8D1 008 59                         pop     ecx
.text:1000E8D2 004 E8 90 6C FF FF             call    sub_10005567    ; Call Procedure
.text:1000E8D7 004 EB 1B                      jmp     short loc_1000E8F4 ; Jump
.text:1000E8D9                    ; ---------------------------------------------------------------------------
```

Figure 77: Install cancel for third XRef.

The functions that called sub_10006196 are all installer subroutines for InstallRT, InstallSA, and InstallSB (Figures 78, 79, and 80 respectively). It is unclear what the last two characters of each "InstallXX" subroutines mean, but it appears to be important that to the designer of this program to not install on a VM. A common tactic that malware designers use is to implement anti-VM measures within their code. This is to prevent sandbox analysis of the malware and to infect real machines.

```
ext:1000D847
ext:1000D847                    ; ||||||||||||||| S U B R O U T I N E |||||||||||||||||||||||||||||||||||||||
ext:1000D847
ext:1000D847
ext:1000D847                    ; int __fastcall InstallRT(int,int,int,int,char *,int)
ext:1000D847                            public InstallRT
ext:1000D847                    InstallRT       proc near
ext:1000D847
ext:1000D847                    var_8           = dword ptr -8
ext:1000D847                    arg_8           = dword ptr  0Ch
ext:1000D847
ext:1000D847 000 A1 34 90 01 10                 mov     eax, off_10019034
ext:1000D84C 000 56                             push    esi             ; char
ext:1000D84D 004 8B 35 B4 62 01 10              mov     esi, ds:atoi
```

Figure 78: InstallRT.

```
.text:1000DEC1
.text:1000DEC1                  ; int __fastcall InstallSA(int,int,int,int,int,int)
.text:1000DEC1                          public InstallSA
.text:1000DEC1                  InstallSA       proc near
.text:1000DEC1
.text:1000DEC1                  var_8           = dword ptr -8
.text:1000DEC1                  arg_8           = dword ptr  0Ch
.text:1000DEC1
.text:1000DEC1 000 A1 34 90 01 10               mov     eax, off_10019034
.text:1000DEC6 000 56                           push    esi             ; char
.text:1000DEC7 004 8B 35 B4 62 01 10            mov     esi, ds:atoi
.text:1000DECD 004 83 C0 0D                     add     eax, 0Dh        ; Add
.text:1000DED0 004 50                           push    eax             ; char *
```

Figure 79: InstallSA.

```
.text:1000E892
.text:1000E892                  ; int __fastcall InstallSB(int,int,int,int,int,int)
.text:1000E892                          public InstallSB
.text:1000E892                  InstallSB       proc near
.text:1000E892
.text:1000E892                  var_8           = dword ptr -8
.text:1000E892                  arg_8           = dword ptr  0Ch
.text:1000E892
.text:1000E892 000 A1 34 90 01 10               mov     eax, off_10019034
.text:1000E897 000 56                           push    esi             ; char
.text:1000E898 004 8B 35 B4 62 01 10            mov     esi, ds:atoi
.text:1000E89E 004 83 C0 0D                     add     eax, 0Dh        ; Add
```

Figure 80: InstallSB.

LAB 5-1 Question 18

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

BLUF: xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234

Initially when we jump to 0x1001D988, we see what appears to be random data (Figure 81).

```
.data:1001D988 2D                                      db  2Dh ; -
.data:1001D989 31                                      db  31h ; 1
.data:1001D98A 3A                                      db  3Ah ; :
.data:1001D98B 3A                                      db  3Ah ; :
.data:1001D98C 27                                      db  27h ; '
.data:1001D98D 75                                      db  75h ; u
.data:1001D98E 3C                                      db  3Ch ; <
.data:1001D98F 26                                      db  26h ; &
.data:1001D990 75                                      db  75h ; u
.data:1001D991 21                                      db  21h ; !
.data:1001D992 3D                                      db  3Dh ; =
.data:1001D993 3C                                      db  3Ch ; <
.data:1001D994 26                                      db  26h ; &
.data:1001D995 75                                      db  75h ; u
.data:1001D996 37                                      db  37h ; 7
.data:1001D997 34                                      db  34h ; 4
.data:1001D998 36                                      db  36h ; 6
```

Figure 81: Random data?

To get an idea of what the data might be, the book suggests that we use the python script provided by going to File > Script File. However, the IDA Free does not have this capability, so we examine the script and see if there is any way that we can replicate the commands in another tool (Figure 82).

Lab05-01.py

```
1  sea = ScreenEA()
2
3  for i in range(0x00,0x50):
4          b = Byte(sea+i)
5          decoded_byte = b ^ 0x55
6          PatchByte(sea+i,decoded_byte)
7
```

Figure 82: The provided script.

What we see in the script is loops from 0x00 to 0x50 and XORs each byte value with 0x55. When we add the address of 0x1001D988 by 50, we get 0x1001D9D8 which is the first line in this series of random characters that doesn't contain anything (Figure 83).

```
.data:1001D9D5                  db  67h ; g
.data:1001D9D6                  db  66h ; f
.data:1001D9D7                  db  61h ; a
.data:1001D9D8                  db    0
.data:1001D9D9                  db    0
.data:1001D9DA                  db    0
.data:1001D9DB                  db    0
.data:1001D9DC                  db    0
.data:1001D9DD                  db    0
.data:1001D9DE                  db    0
.data:1001D9DF                  db    0
```

Figure 83: 0x1001D9D8.

To recreate it, I simply copied the hexadecimal values into a python array. Then I made a for() loop where I XOR'd each value which was left with the decimal notation. I then used the built-in library function "chr()" to convert the decimal value to ASCII. Then the character was simply added to an empty string. We get the final output of *"xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234"* (Figure 84).

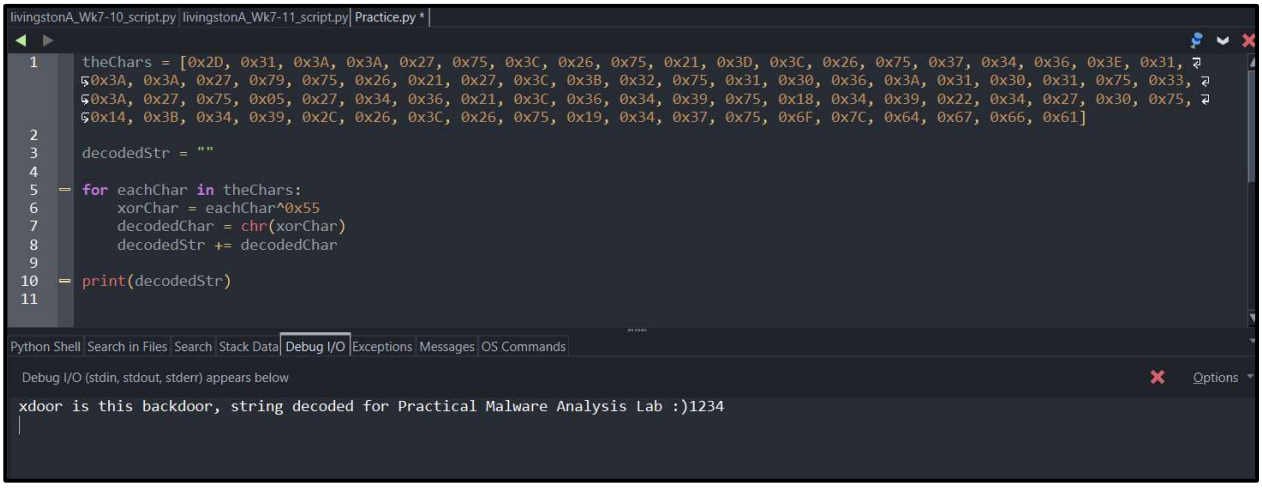


Figure 84: The recreated script and output.