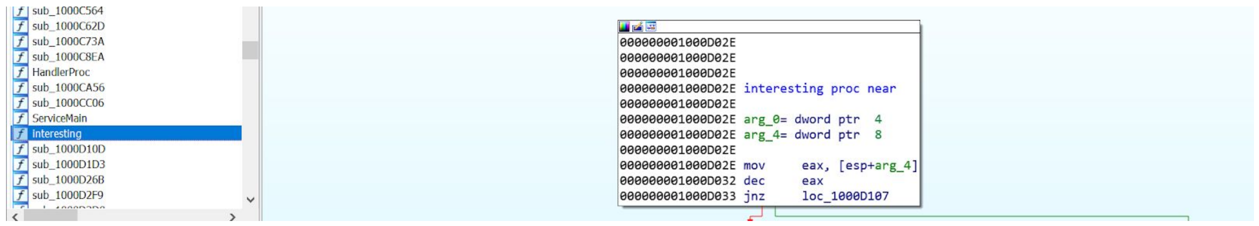
Lab 5-1 Analyze the malware found in the file Lab05-01.dll using only IDA Pro. The goal of this lab is to give you hands-on experience with IDA Pro. If you've already worked with IDA Pro, you may choose to ignore these questions and focus on reverse-engineering the malware.

1. What is the address of DllMain?
G → 1000d02e
Ctrl x ⇒ DllEntryPoint + 4B

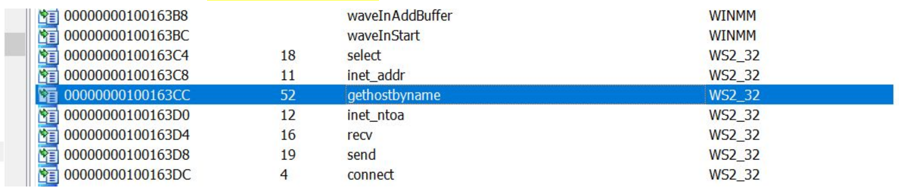
Windows → functions window → Service Main / Main
View → open subviews → functions



```
sub_1000C564
sub_1000C62D
sub_1000C73A
sub_1000C8EA
HandlerProc
sub_1000CA56
sub_1000CC06
ServiceMain
interesting
sub_1000D10D
sub_1000D1D3
sub_1000D26B
sub_1000D2F9
```

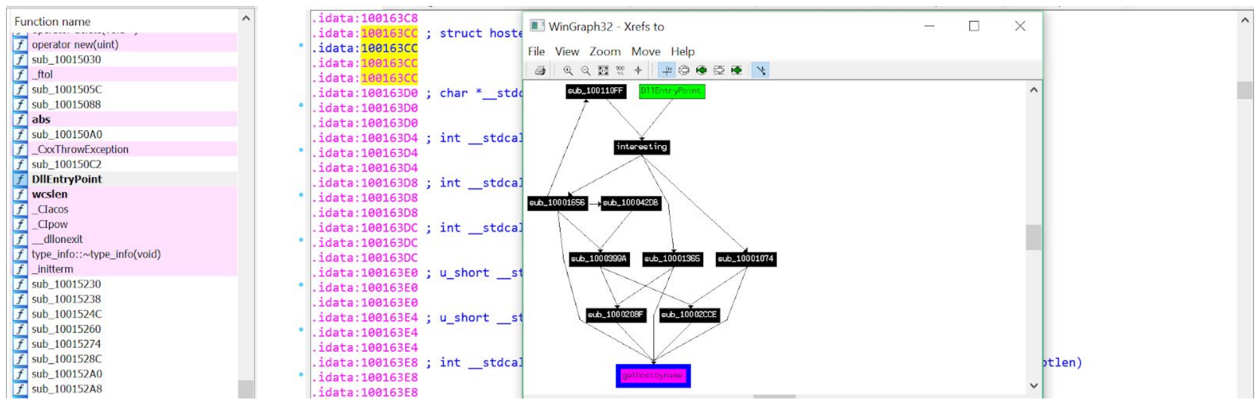
```
000000001000D02E
000000001000D02E
000000001000D02E
000000001000D02E interesting proc near
000000001000D02E
000000001000D02E arg_0= dword ptr  4
000000001000D02E arg_4= dword ptr  8
000000001000D02E
000000001000D02E mov     eax, [esp+arg_4]
000000001000D032 dec     eax
000000001000D033 jnz     loc_1000D107
```

2. Use the Imports window to browse to gethostbyname. Where is the import located?
gethostbyname is found at 0x100163CC



00000000100163B8		waveInAddBuffer	WINMM
00000000100163BC		waveInStart	WINMM
00000000100163C4	18	select	WS2_32
00000000100163C8	11	inet_addr	WS2_32
00000000100163CC	52	gethostbyname	WS2_32
00000000100163D0	12	inet_ntoa	WS2_32
00000000100163D4	16	recv	WS2_32
00000000100163D8	19	send	WS2_32
00000000100163DC	4	connect	WS2_32

3. How many functions call gethostbyname?
The gethostbyname import is called nine times by five different functions throughout the malware

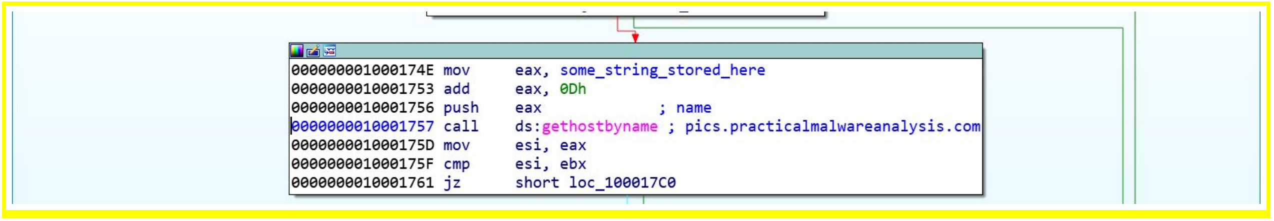


```
Function name
operator new(uint)
sub_10015030
_ftol
sub_1001505C
sub_10015088
abs
sub_100150A0
_CxxThrowException
sub_100150C2
DllEntryPoint
wcslen
_CIacos
_CIpow
__dllonexit
type_info::~type_info(void)
_initterm
sub_10015230
sub_10015238
sub_1001524C
sub_10015260
sub_10015274
sub_1001528C
sub_100152A0
sub_100152A8
```

```
.idata:100163C8
.idata:100163CC ; struct hoste
.idata:100163CC
.idata:100163CC
.idata:100163CC
.idata:100163D0 ; char *__stdc
.idata:100163D0
.idata:100163D0
.idata:100163D4 ; int __stdcal
.idata:100163D4
.idata:100163D4
.idata:100163D8 ; int __stdcal
.idata:100163D8
.idata:100163D8
.idata:100163DC ; int __stdcal
.idata:100163DC
.idata:100163DC
.idata:100163E0 ; u_short __st
.idata:100163E0
.idata:100163E0
.idata:100163E4 ; u_short __st
.idata:100163E4
.idata:100163E4
.idata:100163E8 ; int __stdcal
.idata:100163E8
.idata:100163E8
```

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

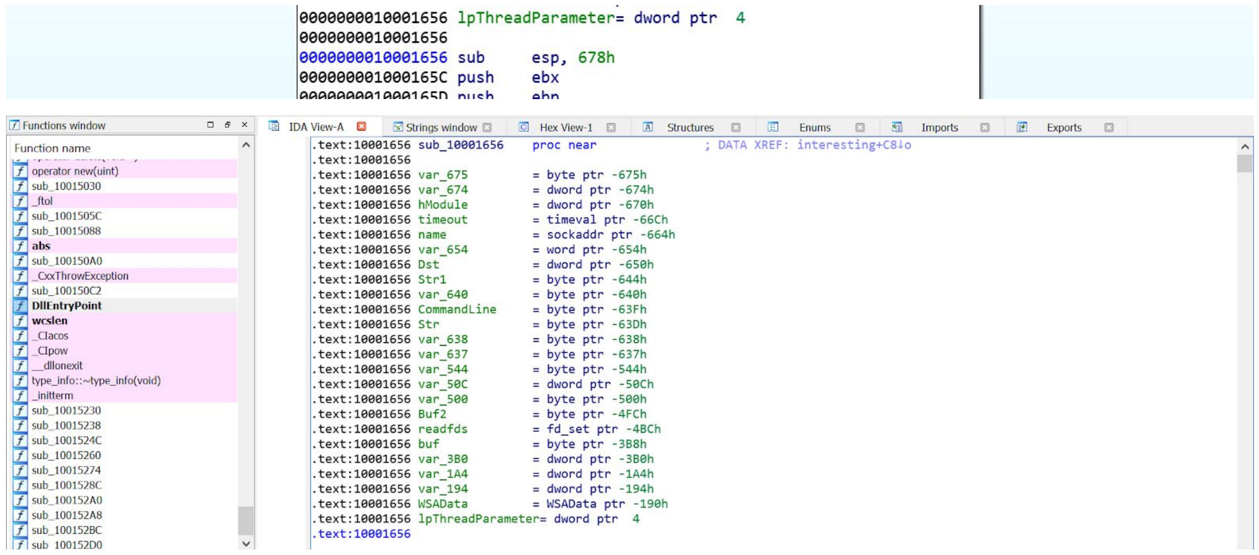
Pics.practicalmalwareanalysis.com



```
000000001000174E mov     eax, some_string_stored_here
0000000010001753 add     eax, 0Dh
0000000010001756 push    eax             ; name
0000000010001757 call    ds:gethostbyname ; pics.practicalmalwareanalysis.com
000000001000175D mov     esi, eax
000000001000175F cmp     esi, ebx
0000000010001761 jz      short loc_100017C0
```

5. How many local variables has IDA Pro recognized for the subroutine at 0x10001656? When you have this address selected, hit spacebar to see local variables.

Note: Count the variables with a negative offset ⇒ 23



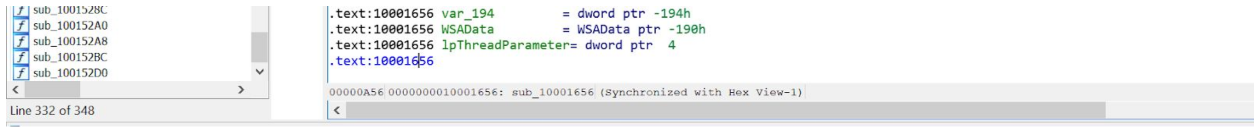
```
0000000010001656 lpThreadParameter= dword ptr  4
0000000010001656
0000000010001656 sub     esp, 678h
000000001000165C push    ebx
000000001000165D push    ebp
```

```
Function name
operator new(uint)
sub_10015030
_ftol
sub_1001505C
sub_10015088
abs
sub_100150A0
_CxxThrowException
sub_100150C2
DllEntryPoint
wcslen
_CIacos
_CIpow
__dllonexit
type_info::~type_info(void)
_initterm
sub_10015230
sub_10015238
sub_1001524C
sub_10015260
sub_10015274
sub_1001528C
sub_100152A0
sub_100152A8
sub_100152BC
sub_100152D0
```

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

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

The parameter count is 1. Because there is only one positive offset in the list

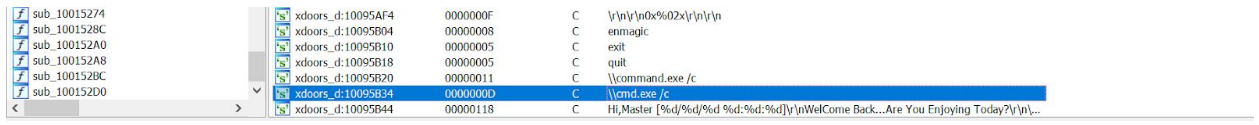


```
.text:10001656 var_194         = dword ptr -194h
.text:10001656 WSAData         = WSAData ptr -190h
.text:10001656 lpThreadParameter= dword ptr  4
.text:10001656

00000A56 0000000010001656: sub_10001656 (Synchronized with Hex View-1)
```

Line 332 of 348

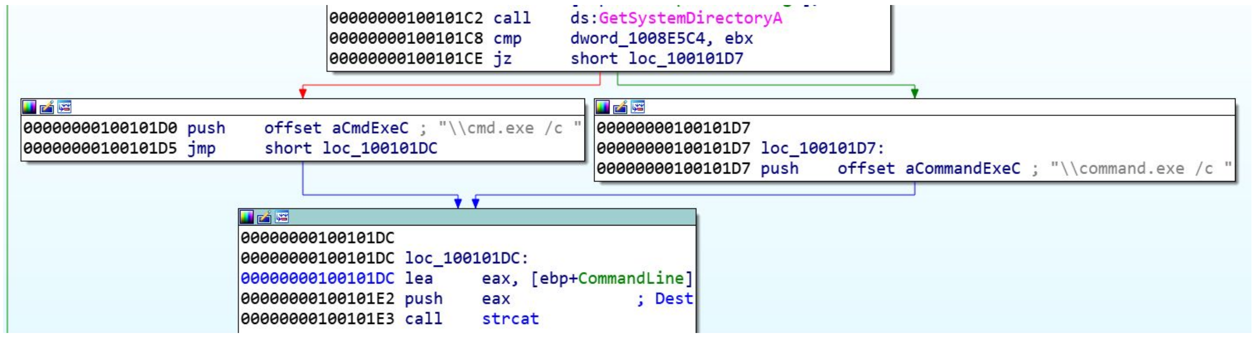
7. Use the Strings window to locate the string \cmd.exe /c in the disassembly. Where is it located?

It is located at 10095b34



xdoors_d:10095AF4	0000000F	C	\r\n\r\n0x%02x\r\n\r\n
xdoors_d:10095B04	00000008	C	enmagic
xdoors_d:10095B10	00000005	C	exit
xdoors_d:10095B18	00000005	C	quit
xdoors_d:10095B20	00000011	C	\\command.exe /c
xdoors_d:10095B34	0000000D	C	\\cmd.exe /c
xdoors_d:10095B44	00000118	C	Hi,Master [%d/%d/%d %d:%d:%d]\r\nWelCome Back...Are You Enjoying Today?\r\n\...

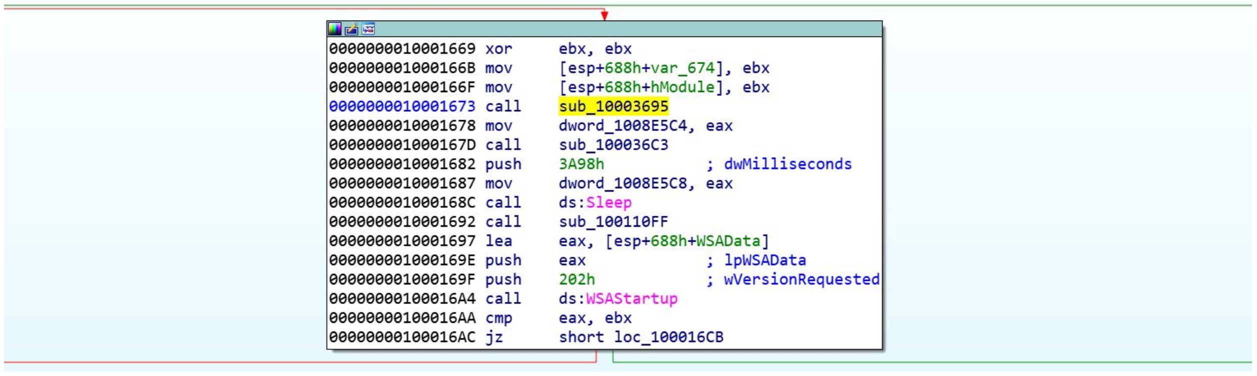
8. What is happening in the area of code that references \cmd.exe /c?
Remote shell session
Note: recv function is being called. This can mean the program is waiting for commands
Recv = receive a message from a connected socket



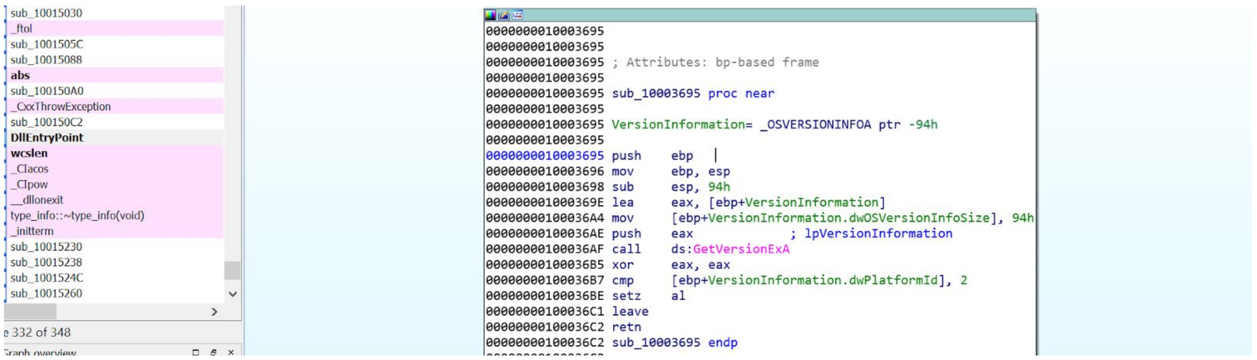
```
00000000100101C2 call    ds:GetSystemDirectoryA
00000000100101C8 cmp     dword_1008E5C4, ebx
00000000100101CE jz      short loc_100101D7

00000000100101D0 push    offset aCmdExeC ; "\\cmd.exe /c "
00000000100101D5 jmp     short loc_100101DC

00000000100101D7
00000000100101D7 loc_100101D7:
00000000100101D7 push    offset aCommandExeC ; "\\command.exe /c "

00000000100101DC
00000000100101DC loc_100101DC:
00000000100101DC lea     eax, [ebp+CommandLine]
00000000100101E2 push    eax             ; Dest
00000000100101E3 call    strcat
```

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.)

dword_1008E5C4 is set by the function 10003695



```
0000000010001669 xor     ebx, ebx
000000001000166B mov     [esp+688h+var_674], ebx
000000001000166F mov     [esp+688h+hModule], ebx
0000000010001673 call    sub_10003695
0000000010001678 mov     dword_1008E5C4, eax
000000001000167D call    sub_100036C3
0000000010001682 push    3A98h           ; dwMilliseconds
0000000010001687 mov     dword_1008E5C8, eax
000000001000168C call    ds:Sleep
0000000010001692 call    sub_100110FF
0000000010001697 lea     eax, [esp+688h+WSAData]
000000001000169E push    eax             ; lpWSAData
000000001000169F push    202h            ; wVersionRequested
00000000100016A4 call    ds:WSAStartup
00000000100016AA cmp     eax, ebx
00000000100016AC jz      short loc_100016CB
```

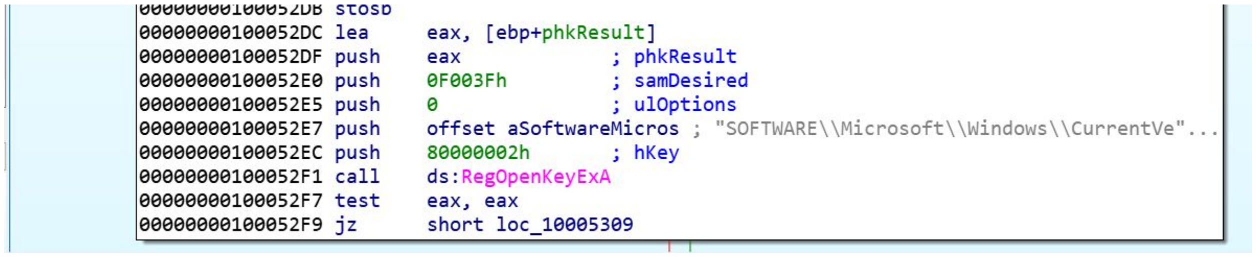


```
sub_10015030
_ftol
sub_1001505C
sub_10015088
abs
sub_100150A0
_CxxThrowException
sub_100150C2
DllEntryPoint
wcslen
_CIacos
_CIpow
__dllonexit
type_info::~type_info(void)
_initterm
sub_10015230
sub_10015238
sub_1001524C
sub_10015260
e 332 of 348
Graph overview
```

```
0000000010003695
0000000010003695
0000000010003695 ; Attributes: bp-based frame
0000000010003695
0000000010003695 sub_10003695 proc near
0000000010003695
0000000010003695 VersionInformation= _OSVERSIONINFOA ptr -94h
0000000010003695
0000000010003695 push    ebp
0000000010003696 mov     ebp, esp
0000000010003698 sub     esp, 94h
000000001000369E lea     eax, [ebp+VersionInformation]
00000000100036A4 mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
00000000100036AE push    eax             ; lpVersionInformation
00000000100036AF call    ds:GetVersionExA
00000000100036B5 xor     eax, eax
00000000100036B7 cmp     [ebp+VersionInformation.dwPlatformId], 2
00000000100036BE setz    al
00000000100036C1 leave
00000000100036C2 retn
00000000100036C2 sub_10003695 endp
```

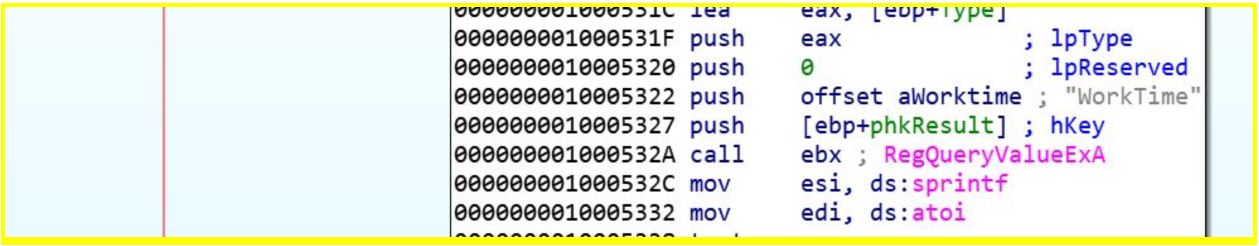
10. A few hundred lines into the subroutine at 0x1000FF58, a series of comparisons use memcmp to compare strings. What happens if the string comparison to robotwork is successful (when memcmp returns 0)?

software/microsoft/windows/currentversion



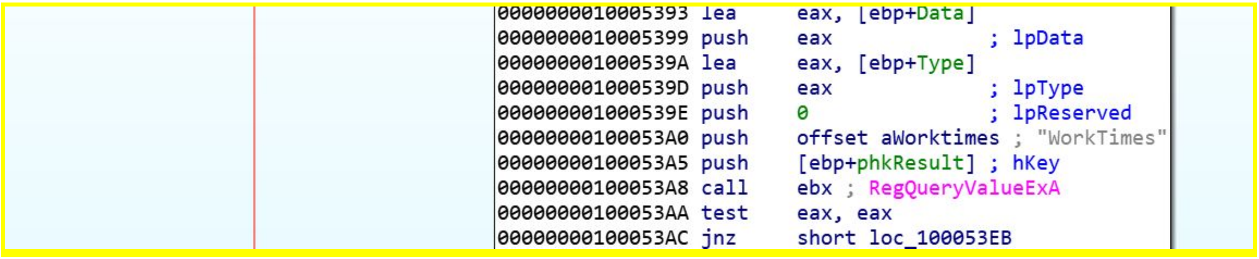
```
00000000100052DB stosb
00000000100052DC lea     eax, [ebp+phkResult]
00000000100052DF push    eax              ; phkResult
00000000100052E0 push    0F003Fh          ; samDesired
00000000100052E5 push    0                ; ulOptions
00000000100052E7 push    offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVe"...
00000000100052EC push    80000002h        ; hKey
00000000100052F1 call    ds:RegOpenKeyExA
00000000100052F7 test    eax, eax
00000000100052F9 jz      short loc_10005309
```

Worktime



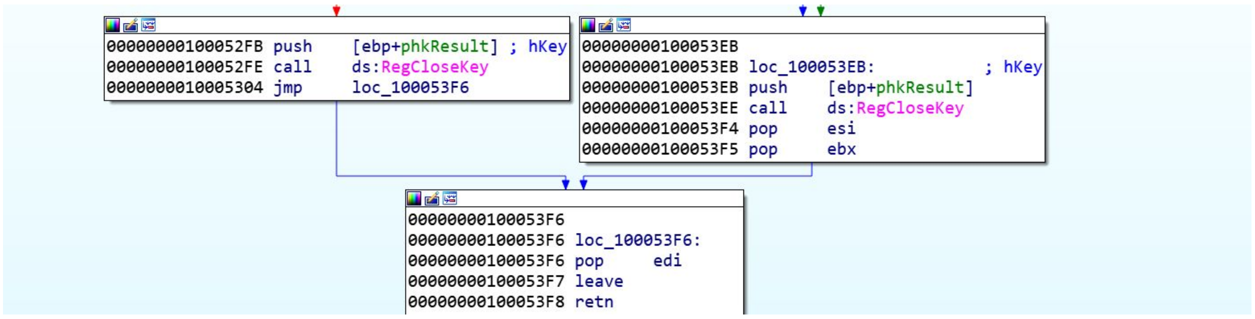
```
000000001000531C lea     eax, [ebp+Type]
000000001000531F push    eax              ; lpType
0000000010005320 push    0                ; lpReserved
0000000010005322 push    offset aWorktime ; "WorkTime"
0000000010005327 push    [ebp+phkResult] ; hKey
000000001000532A call    ebx ; RegQueryValueExA
000000001000532C mov     esi, ds:sprintf
0000000010005332 mov     edi, ds:atoi
```

Worktimes



```
0000000010005393 lea     eax, [ebp+Data]
0000000010005399 push    eax              ; lpData
000000001000539A lea     eax, [ebp+Type]
000000001000539D push    eax              ; lpType
000000001000539E push    0                ; lpReserved
00000000100053A0 push    offset aWorktimes ; "WorkTimes"
00000000100053A5 push    [ebp+phkResult] ; hKey
00000000100053A8 call    ebx ; RegQueryValueExA
00000000100053AA test    eax, eax
00000000100053AC jnz     short loc_100053EB
```

Regclosekey



```
00000000100052FB push    [ebp+phkResult] ; hKey
00000000100052FE call    ds:RegCloseKey
0000000010005304 jmp     loc_100053F6
```

```
00000000100053EB
00000000100053EB loc_100053EB:            ; hKey
00000000100053EB push    [ebp+phkResult]
00000000100053EE call    ds:RegCloseKey
00000000100053F4 pop     esi
00000000100053F5 pop     ebx
```

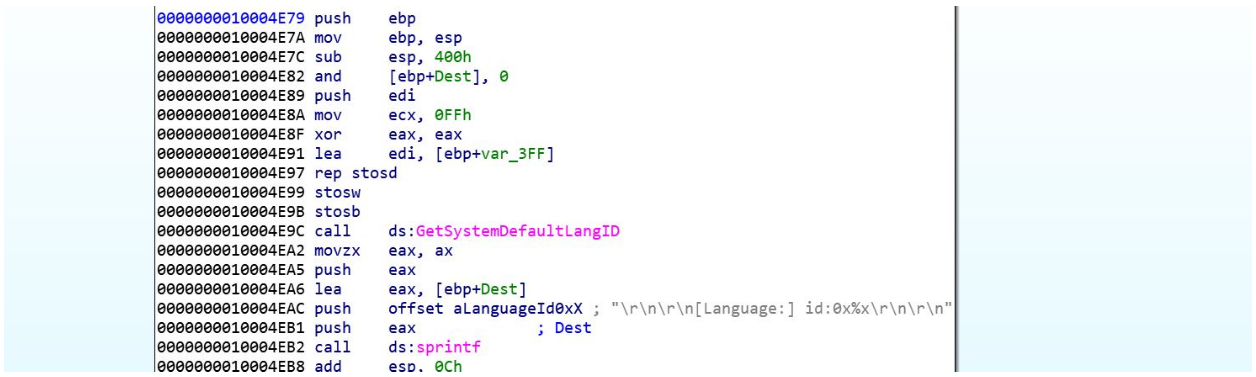
```
00000000100053F6
00000000100053F6 loc_100053F6:
00000000100053F6 pop     edi
00000000100053F7 leave
00000000100053F8 retn
```

11. What does the export PSLIST do?

The PSLIST export sends a process listing across the network or finds a particular process name in the listing and gets information about it

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?

Getsystemdefault and sprintf ⇒ It can be renamed setlanguage

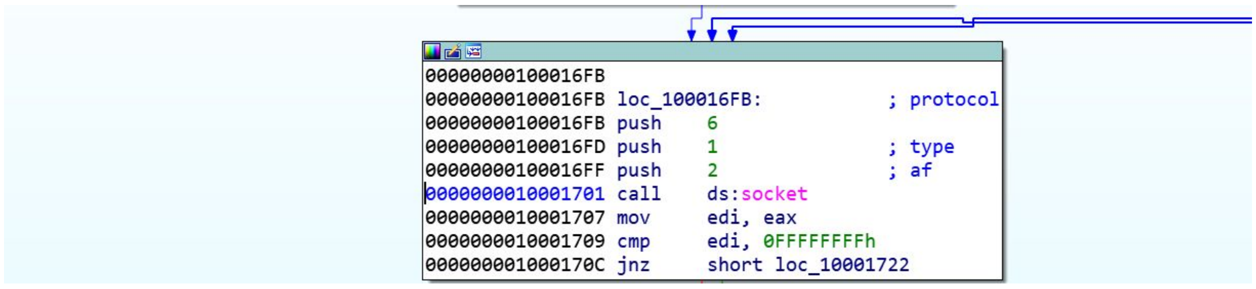

```
0000000010004E79 push    ebp
0000000010004E7A mov     ebp, esp
0000000010004E7C sub     esp, 400h
0000000010004E82 and     [ebp+Dest], 0
0000000010004E89 push    edi
0000000010004E8A mov     ecx, 0FFh
0000000010004E8F xor     eax, eax
0000000010004E91 lea     edi, [ebp+var_3FF]
0000000010004E97 rep stosd
0000000010004E99 stosw
0000000010004E9B stosb
0000000010004E9C call    ds:GetSystemDefaultLangID
0000000010004EA2 movzx   eax, ax
0000000010004EA5 push    eax
0000000010004EA6 lea     eax, [ebp+Dest]
0000000010004EAC push    offset aLanguageId0xX ; "\r\n\r\n[Language:] id:0x%x\r\n\r\n"
0000000010004EB1 push    eax             ; Dest
0000000010004EB2 call    ds:sprintf
0000000010004EB8 add     esp, 0Ch
```

13. How many Windows API functions does DllMain call directly? How many at a depth of 2?
Direct calls ⇒ strncpy, strnicmp, CreateThread, and strlen
Depth of 2 ⇒ of API calls, including Sleep, WinExec, gethostbyname

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?

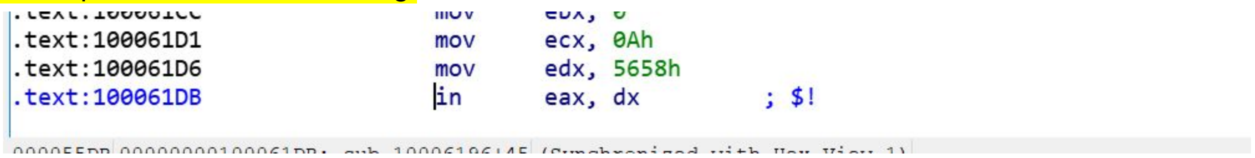
It adds 13
It is converted into integer w/ atoi function.
It is multiplied by 1000.
eax contains 30000(milliseconds)
Sleep for 30 seconds

15. At 0x10001701 is a call to socket. What are the three parameters?
Protocol, type, af [6,1,2]

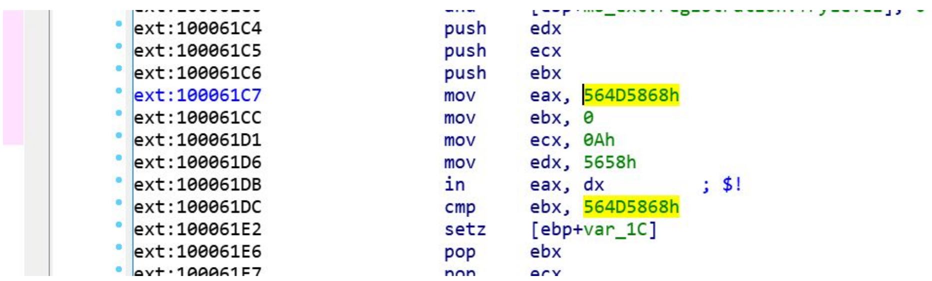

```
00000000100016FB
00000000100016FB loc_100016FB:           ; protocol
00000000100016FB push    6
00000000100016FD push    1               ; type
00000000100016FF push    2               ; af
0000000010001701 call    ds:socket
0000000010001707 mov     edi, eax
0000000010001709 cmp     edi, 0FFFFFFFFh
000000001000170C jnz     short loc_10001722
```

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?
IPPROTO_TCP, SOCK_STREAM, and AF_INET

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?
 The in instruction is used for virtual machine detection at 0x100061DB, and the 0x564D5868h corresponds to the VMXh string.



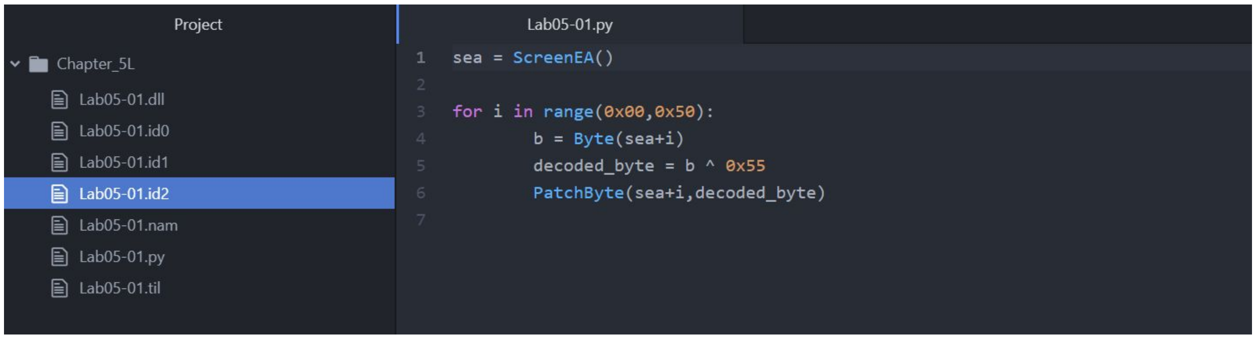
```
.text:100061D1                 mov     ecx, 0Ah
.text:100061D6                 mov     edx, 5658h
.text:100061DB                 in      eax, dx         ; $!
```



```
ext:100061C4                 push    edx
ext:100061C5                 push    ecx
ext:100061C6                 push    ebx
ext:100061C7                 mov     eax, 564D5868h
ext:100061CC                 mov     ebx, 0
ext:100061D1                 mov     ecx, 0Ah
ext:100061D6                 mov     edx, 5658h
ext:100061DB                 in      eax, dx         ; $!
ext:100061DC                 cmp     ebx, 564D5868h
ext:100061E2                 setz    [ebp+var_1C]
ext:100061E6                 pop     ebx
```

18. Jump your cursor to 0x1001D988. What do you find?
A string of characters.

19/20/21. If you have the IDA Python plug-in installed (included with the commercial version of IDA Pro), run Lab05-01.py, an IDA Pro Python script provided with the malware for this book. (Make sure the cursor is at 0x1001D988.) What happens after you run the script?
I did not notice changes to the random characters from question 18
Below is the text editor output after running the python command



```python
sea = ScreenEA()

for i in range(0x00,0x50):
        b = Byte(sea+i)
        decoded_byte = b ^ 0x55
        PatchByte(sea+i,decoded_byte)
```