R·I·T	Rochester Institute of Technology Golisano College of Computing and Information Sciences Department of Computing Security

Name: Smayan Daruka
Date: 11/28/18

PMA 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?

```
.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
```

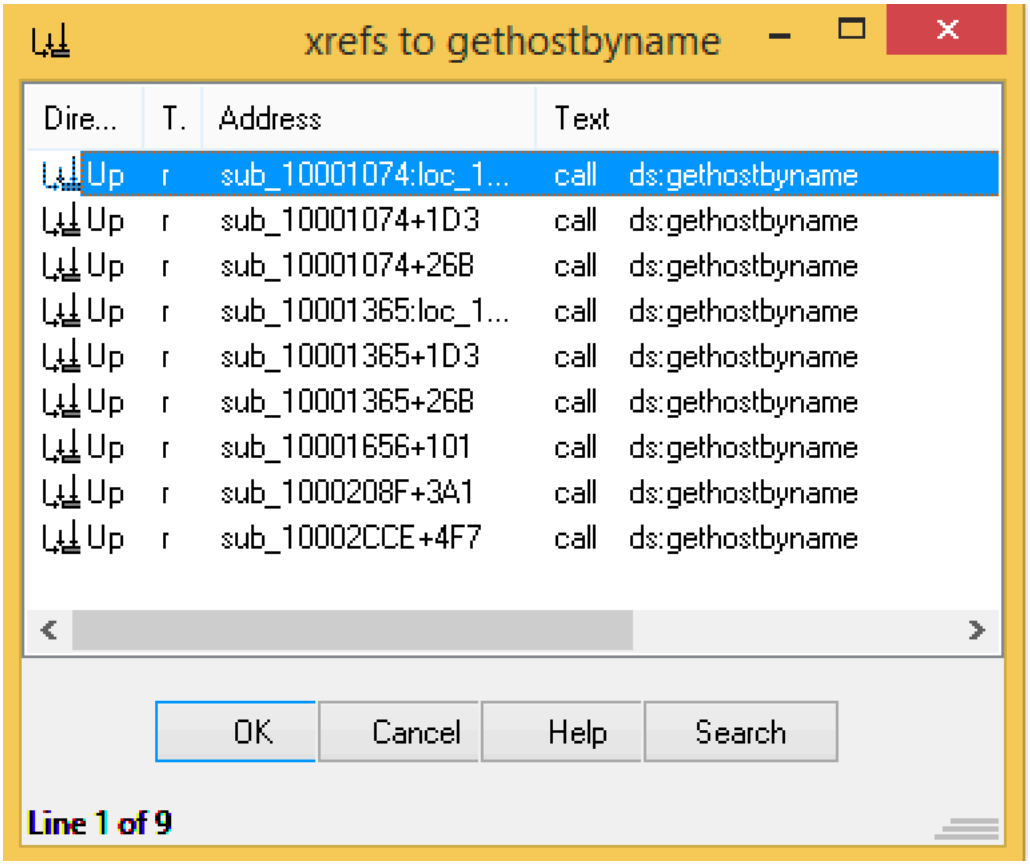
As can be seen in the screenshot above, DllMain is located at 0x1000D02E memory address in the .text section.

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


```
.idata:100163CC ; struct hostent *__stdcall gethostbyname(const char *name)
.idata:100163CC                 extrn gethostbyname:dword
.idata:100163CC                                         ; DATA XREF: sub_10001074:loc_100011AF↑r
.idata:100163CC                                         ; sub_10001074+1D3↑r ...
```

As can be seen in the screenshots above, the import "gethostbyname" is located at 0x100163CC memory address in the .idata section.

3. How many functions call gethostbyname?



As can be seen above, 5 unique functions call gethostbyname 9 times altogether.

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

```
.text:1000174E                  mov     eax, off_10019040
.text:10001753                  add     eax, 0Dh
.text:10001756                  push    eax             ; name
.text:10001757                  call    ds:gethostbyname
```

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

As can be seen in the above two images, a DNS request is made to the hostname “pics.practicalmalwareanalysis.com”.

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

IDA Pro recognized 20 local variables for the subroutine at 0x10001656 as can be seen in the image below.

```
.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
```

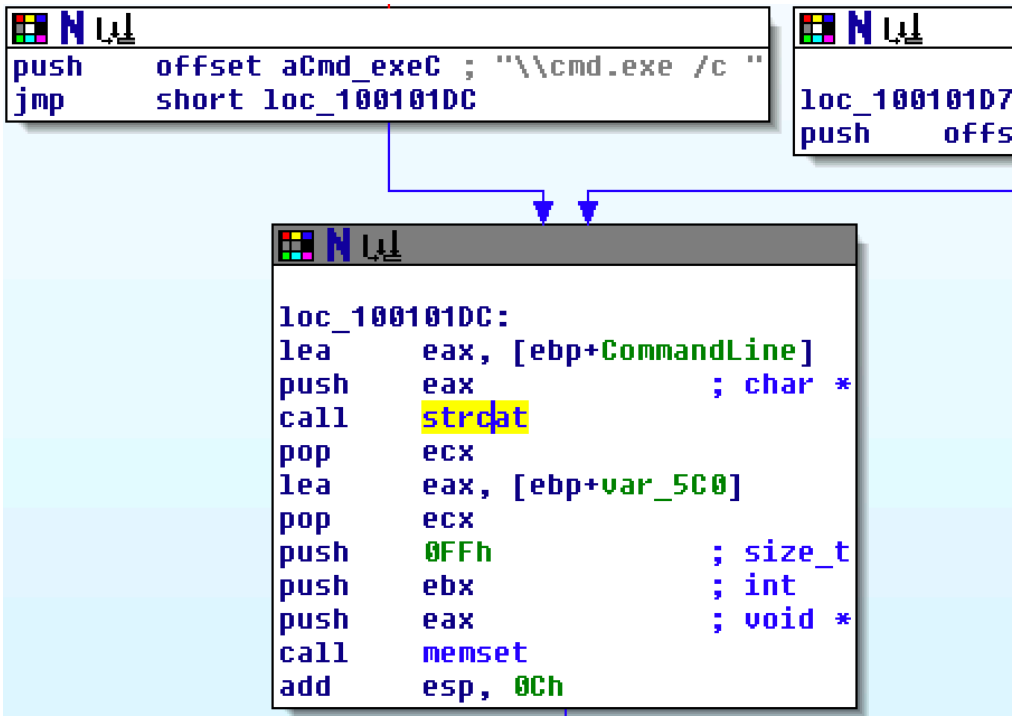
6. How many parameters has IDA Pro recognized for the subroutine at 0x10001656?

IDA Pro recognized one parameter which can be seen in the image in #5 above. Parameters are referenced with positive offsets.

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

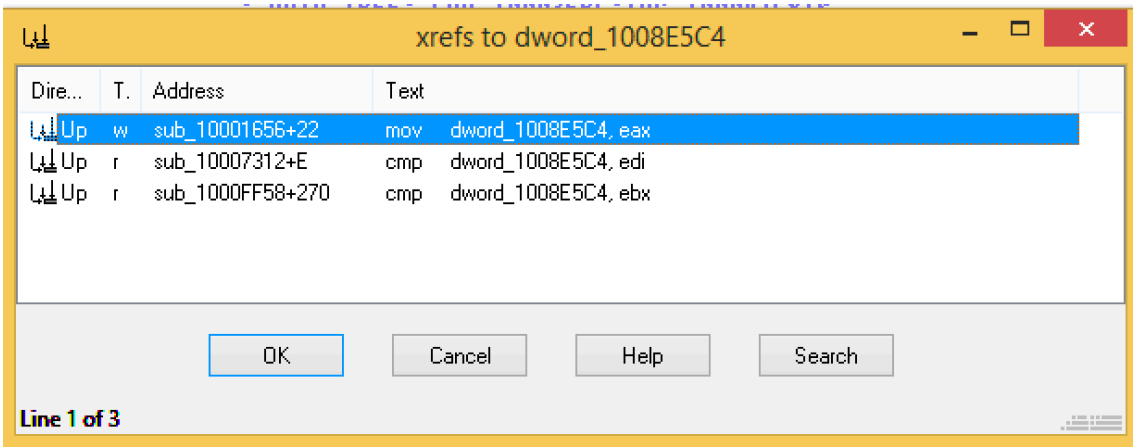
As can be seen above, the string “\cmd.exe /c” is located at 0x10095B34 memory location.

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

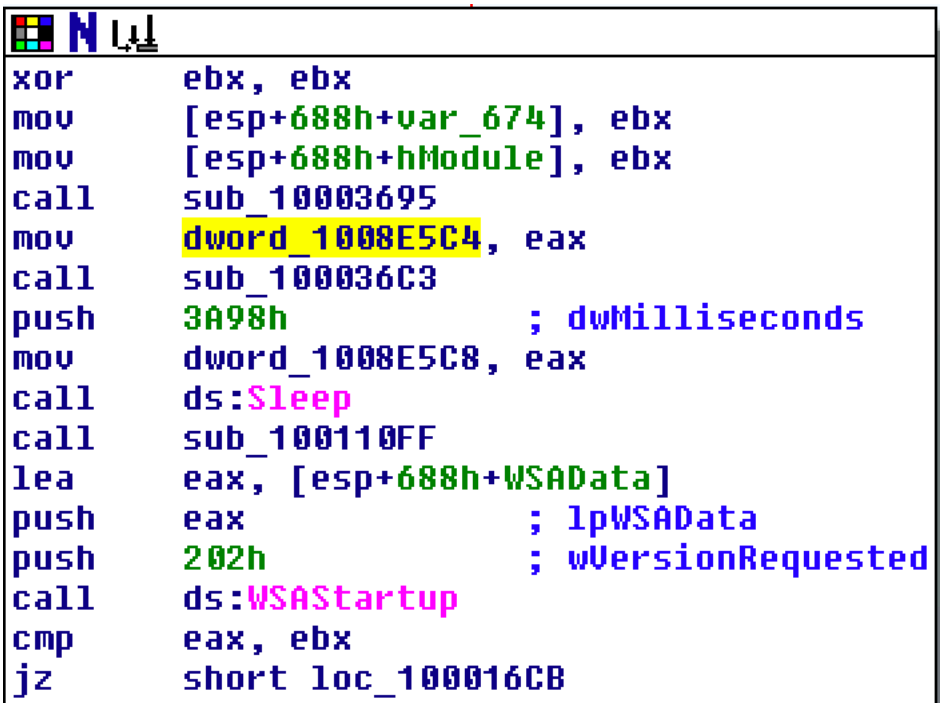


As can be seen above, this area of code looks like it creates a remote shell. There are calls to multiple memory compare functions.

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.)

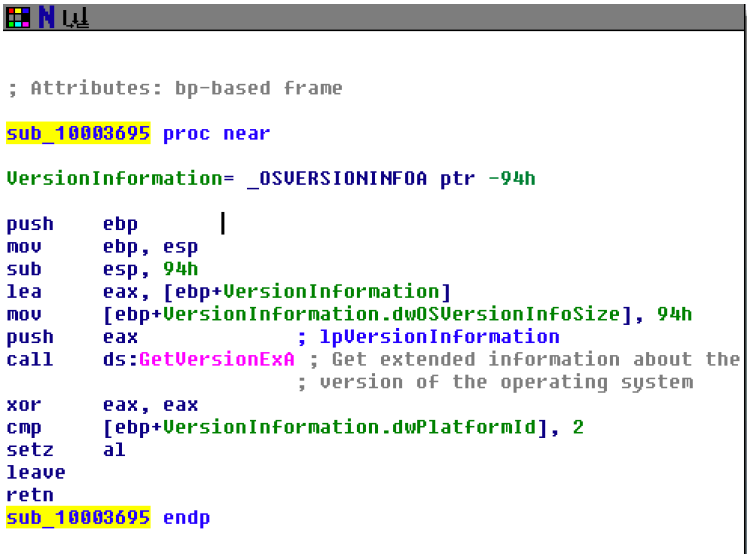


As can be seen above, there are 3 cross references to the global variable. Only one of them actually modifies the variable which is the first cross reference.



```
xor     ebx, ebx
mov     [esp+688h+var_674], ebx
mov     [esp+688h+hModule], ebx
call    sub_10003695
mov     dword_1008E5C4, eax
call    sub_100036C3
push    3A98h           ; dwMilliseconds
mov     dword_1008E5C8, eax
call    ds:Sleep
call    sub_100110FF
lea     eax, [esp+688h+WSAData]
push    eax             ; lpWSAData
push    202h            ; wVersionRequested
call    ds:WSAStartup
cmp     eax, ebx
jz      short loc_100016CB
```

The above screenshot shows the code at the first cross reference's location. As can be seen, there is a call to a function on line 4 and line 5 modifies the global variable we are looking for.



```
; Attributes: bp-based frame

sub_10003695 proc near

VersionInformation= _OSVERSIONINFOA ptr -94h

push    ebp
mov     ebp, esp
sub     esp, 94h
lea     eax, [ebp+VersionInformation]
mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
push    eax             ; lpVersionInformation
call    ds:GetVersionExA ; Get extended information about the
                         ; version of the operating system
xor     eax, eax
cmp     [ebp+VersionInformation.dwPlatformId], 2
setz    al
leave
retn
sub_10003695 endp
```

As can be seen in the screenshot above, the global variable "dword_1008E5C4" is actually the version of the operating system.

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)?

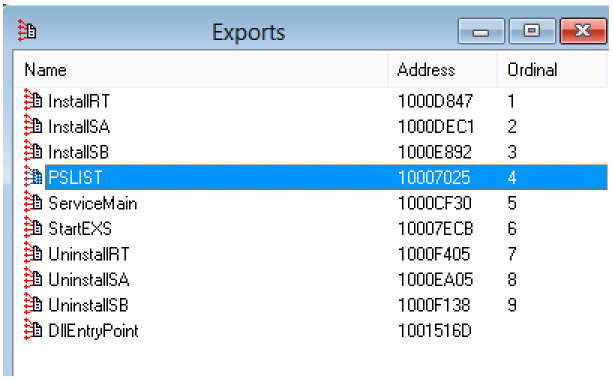
```
.text:10010444 ; ---------------------------------------------------------------------------
.text:10010444
.text:10010444 loc_10010444:                           ; CODE XREF: sub_1000FF58+4E0↑j
.text:10010444                 push    9               ; size_t
.text:10010446                 lea     eax, [ebp+var_5C0]
.text:1001044C                 push    offset aRobotwork ; "robotwork"
.text:10010451                 push    eax             ; void *
.text:10010452                 call    memcmp
.text:10010457                 add     esp, 0Ch
.text:1001045A                 test    eax, eax
.text:1001045C                 jnz     short loc_10010468
.text:1001045E                 push    [ebp+s]         ; s
.text:10010461                 call    sub_100052A2
.text:10010466                 jmp     short loc_100103F6
```

As can be seen above, if the string comparison to robotwork is successful, we do not jump, but instead we continue down to the call instruction. The call instruction is as follows:

```
.text:100052DC                 lea     eax, [ebp+hKey]
.text:100052DF                 push    eax             ; phkResult
.text:100052E0                 push    0F003Fh         ; samDesired
.text:100052E5                 push    0               ; ulOptions
.text:100052E7                 push    offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVersi"...
.text:100052EC                 push    80000002h       ; hKey
.text:100052F1                 call    ds:RegOpenKeyExA
```

As we can see, registry keys, more specifically “SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime” and “SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTimes” are queried by the function. These are then passed back over the network through the socket used previously.

11. What does the export PSLIST do?

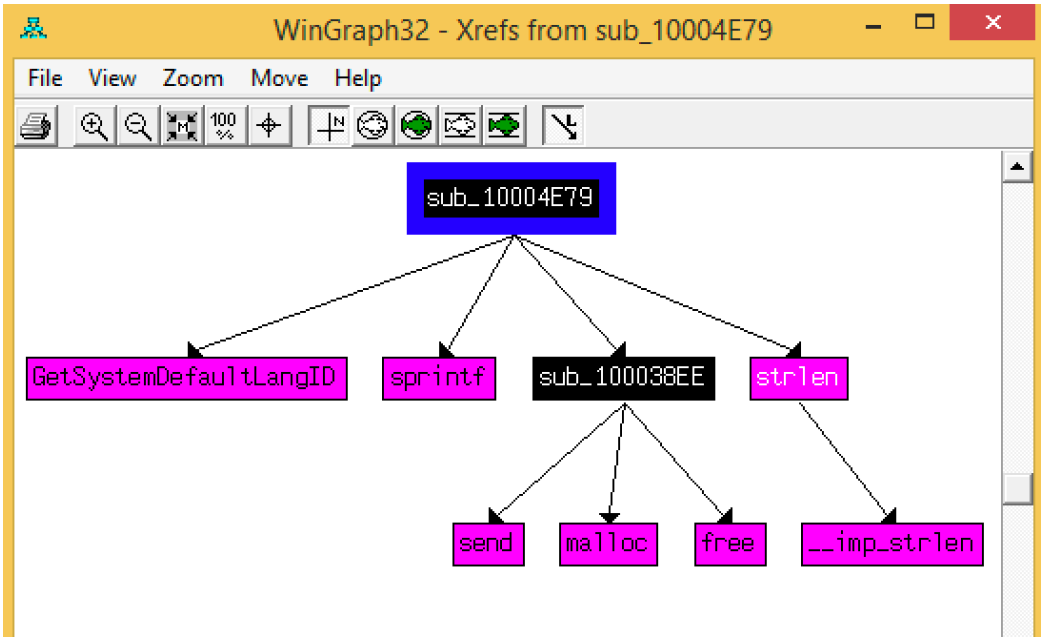


As can be seen above, there is an export called “PSLIST” which does the following:

```
.text:10007025 ; int __stdcall PSLIST(int,int,char *,int)
.text:10007025                 public PSLIST
.text:10007025 PSLIST          proc near
.text:10007025
.text:10007025 arg_8           = dword ptr  0Ch
.text:10007025
.text:10007025                 mov     dword_1008E5BC, 1
.text:1000702F                 call    sub_100036C3
.text:10007034                 test    eax, eax
.text:10007036                 jz      short loc_1000705B
.text:10007038                 push    [esp+arg_8]     ; char *
.text:1000703C                 call    strlen
.text:10007041                 test    eax, eax
.text:10007043                 pop     ecx
.text:10007044                 jnz     short loc_1000704E
.text:10007046                 push    eax
.text:10007047                 call    sub_10006518
.text:1000704C                 jmp     short loc_1000705A
```

We see that this export calls another function and can go one of two ways depending on the result. Both ways return the process listing which is done using an API call and this result is sent back over the socket.

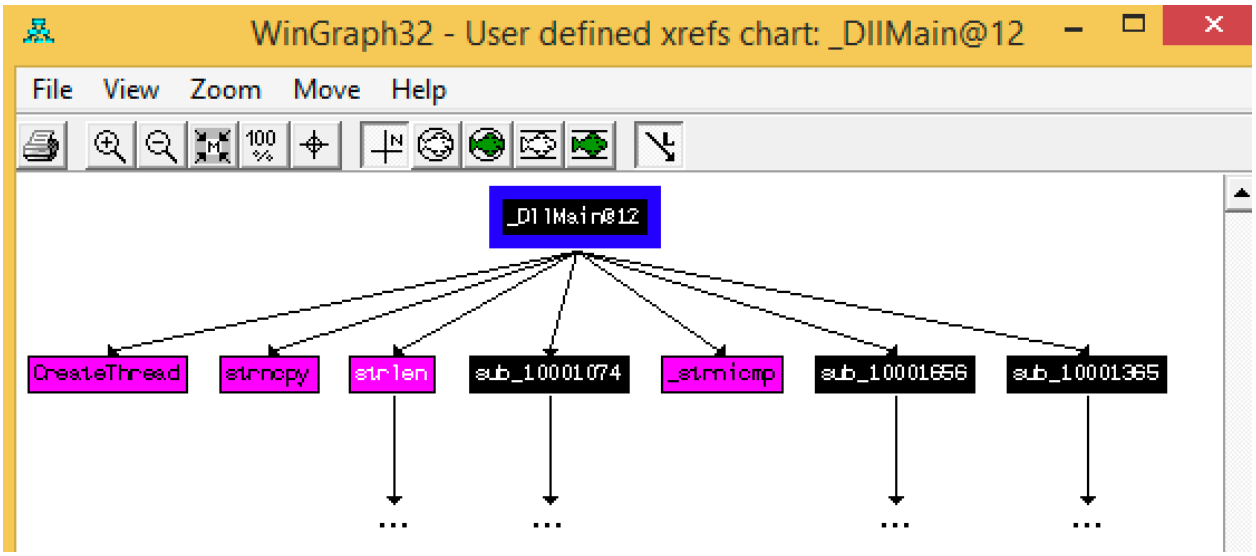
12. Use the graph mode to graph the cross-references from sub_10004E79. Which API functions could be called by entering this function? Based on the API functions alone, what could you rename this function?



As can be seen above, API functions “GetSystemDefaultLangID”, “send”, and “sprintf” could be called. Judging by the names, we could rename this function to “getSystemLanguageID”.

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

As can be seen in the screenshot below, DllMain directly calls “strncpy”, “strlen”, “strnicmp”, and “CreateThread”.



At a depth of 2, there are over 100 nodes, and some API calls include “gethostbyname” and “WinExec”.

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?

The malware will sleep for 30 seconds.

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

```
.text:100016FB                                        ; sub_10001656+A09↓j
.text:100016FB                push    6               ; protocol
.text:100016FD                push    1               ; type
.text:100016FF                push    2               ; af
.text:10001701                call    ds:socket
.text:10001707                mov     edi, eax
.text:10001709                cmp     edi, 0FFFFFFFFh
.text:1000170C                jnz     short loc_10001722
.text:1000170E                call    ds:WSAGetLastError
```

As can be seen above, the three parameters are 6, 1, and 2.

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?

According to the MSDN page for sockets, the parameters should be renamed as following:

- 6 -> IPPROTO_TCP
- 1 -> SOCK_STREAM
- 2 -> 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?

```
.text:100061DB                in      eax, dx
.text:100061DC                cmp     ebx, 564D5868h
.text:100061E2                setz    [ebp+var_1C]
.text:100061E6                pop     ebx
.text:100061E7                pop     ecx
.text:100061E8                pop     edx
.text:100061E9                jmp     short loc_100061F6
```

As can be seen above, this instruction is actually in use by the malware. There is further evidence that this technique is in use when the string “Found Virtual Machine” is found in the cross references.

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

```
.data:1001D988                db  2Dh ; -
.data:1001D989                db  31h ; 1
.data:1001D98A                db  3Ah ; :
.data:1001D98B                db  3Ah ; :
.data:1001D98C                db  27h ; '
.data:1001D98D                db  75h ; u
.data:1001D98E                db  3Ch ; <
.data:1001D98F                db  26h ; &
.data:1001D990                db  75h ; u
.data:1001D991                db  21h ; !
.data:1001D992                db  3Dh ; =
.data:1001D993                db  3Ch ; <
.data:1001D994                db  26h ; &
.data:1001D995                db  75h ; u
```

As can be seen above, there is random data at this memory location.

19. 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?

After running the script, the random data that was previously at the memory location is now transformed into a readable string.

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

After pressing the 'A' key, the string reads "xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234".

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

The script first gets the location of the cursor, which is used as the offset to decode the data. Then, the script XORs each byte with 0x55 and modifies the bytes in IDA Pro using PatchByte.