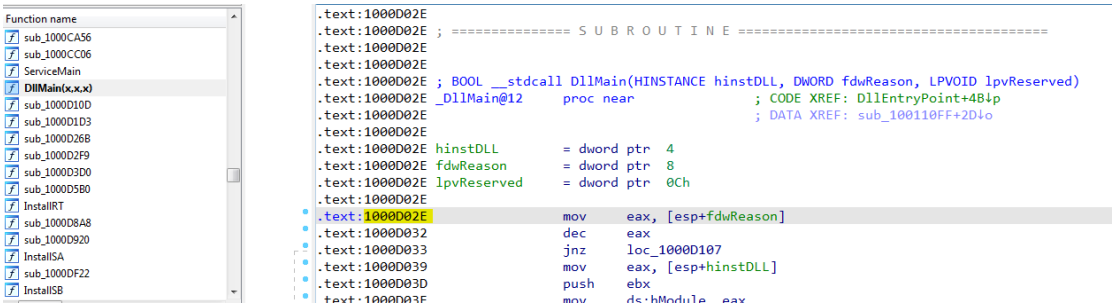
Practical Malware Analysis Chapter 5

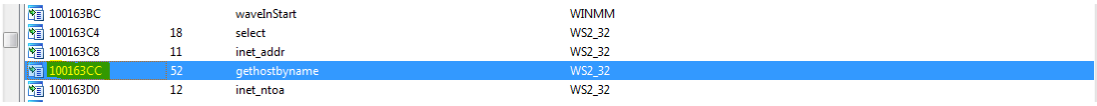
-> What is the address of DllMain?

- address of DllMain is .text:1000D02E.



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

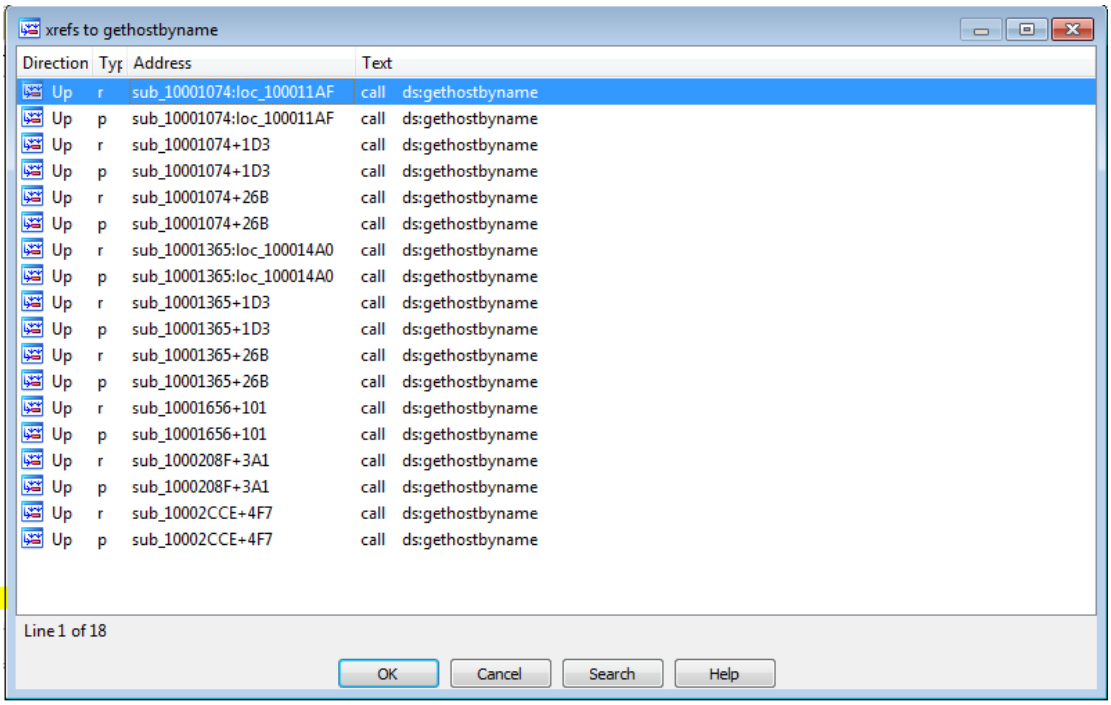
- the address of gethostbyname is .idata:100163CC.



100163BC		waveInStart	WINMM
100163C4	18	select	WS2_32
100163C8	11	inet_addr	WS2_32
100163CC	52	gethostbyname	WS2_32
100163D0	12	inet_ntoa	WS2_32

-> How many functions call gethostbyname?

- 18



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

-pics.practicalmalwareanalysis.com

```
.text:1000173A                 test    eax, eax
.text:1000173C                 jz      loc_100017ED
.text:10001742                 cmp     dword_1008E5CC, ebx
.text:10001748                 jnz     loc_100017ED
.text:1000174E                 mov     eax, off_10019040 ; "[This is RDO]pics.praticalmalwareanalys"...
.text:10001753                 add     eax, 0Dh
.text:10001756                 push    eax             ; name
.text:10001757                 call    ds:gethostbyname
```

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

- 24 variables.

```
.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 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
```

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

- 1 : lpThreadParameter

```
.text:10001656 var_194          = dword ptr -194h
.text:10001656 WSAData          = WSAData ptr -19
.text:10001656 lpThreadParameter= dword ptr  4
.text:10001656
.text:10001656                  sub     esp, 678h
```

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

-address of the string = xdoors_d:10095b34

```
xdoors_d:10095B34 aCmdExeC        db '\cmd.exe /c ',0      ; DATA XREF: sub_1000FF58+278↑o
xdoors_d:10095B41                 align 4
```

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

- It executes these commands but in reverse order:

quit-exit-cd

-> 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.)

- In the beginning I went to the first instruction to use this variable and it was :

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

- I noticed that there was a call instruction before moving the eax value of the variable and (any call the return value is stored in the eax) and I detected this function to see the value turns out to take information about the os version

```
.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                 push    ebp
.text:10003696                 mov     ebp, esp
.text:10003698                 sub     esp, 94h
.text:1000369E                 lea     eax, [ebp+VersionInformation]
.text:100036A4                 mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h ; '”'
.text:100036AE                 push    eax             ; lpVersionInformation
.text:100036AF                 call    ds:GetVersionExA
.text:100036B5                 xor     eax, eax
.text:100036B7                 cmp     [ebp+VersionInformation.dwPlatformId], 2
.text:100036BE                 setz    al
.text:100036C1                 leave
```

- It compares the value of the variable (that contains information about the OS version) and the EBX value.

-> 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)?

- will call the function sub_100052A2

```
.text:10010444 ; ---------------------------------------------------------------------------
.text:10010444
.text:10010444 loc_10010444:                           ; CODE XREF: sub_1000FF58+4E0↑j
.text:10010444                 push    9               ; Size
.text:10010446                 lea     eax, [ebp+Buf1]
.text:1001044C                 push    offset aRobotwork ; "robotwork"
.text:10010451                 push    eax             ; Buf1
.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
.text:10010468 ; ---------------------------------------------------------------------------
```

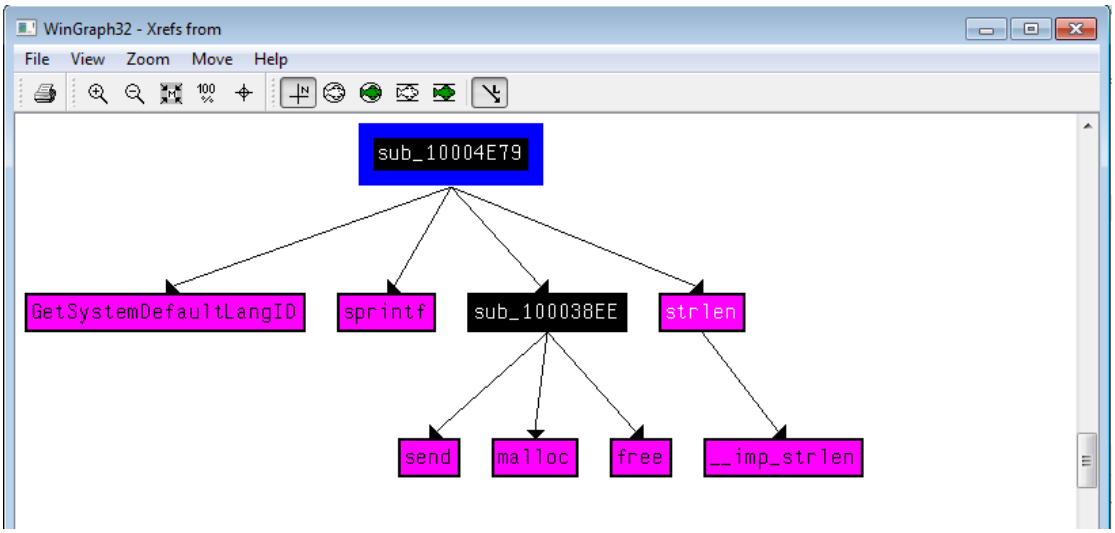
-> What does the export PSLIST do?

- You call the function (sub_100036C3 (This function takes information about the OS version)) and then execute test eax, eax and the eax value is a retrieval from the function before it, and certainly the zero flag status will be 1, and therefore it will go to the location (loc_1000705B)

```
.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+Str]       ; Str
.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
```

```
.text:1000705B
.text:1000705B loc_1000705B:                           ; CODE XREF: PSLIST+11↑j
.text:1000705B                 and     dword_1008E5BC, 0
.text:10007062                 retn    10h
.text:10007062 PSLIST          endp
```

-> 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?



- GetSystemDefualtLangID – sprintf – sub_100038EE - strlen

```
.text:10004E7C                 sub     esp, 400h
.text:10004E82                 and     [ebp+Buffer], 0
.text:10004E89                 push    edi
.text:10004E8A                 mov     ecx, 0FFh
.text:10004E8F                 xor     eax, eax
.text:10004E91                 lea     edi, [ebp+var_3FF]
.text:10004E97                 rep stosd
.text:10004E99                 stosw
.text:10004E9B                 stosb
.text:10004E9C                 call    ds:GetSystemDefaultLangID
.text:10004EA2                 movzx   eax, ax
.text:10004EA5                 push    eax
.text:10004EA6                 lea     eax, [ebp+Buffer]
.text:10004EAC                 push    offset aLanguageId0xX ; "\r\n\r\n[Language:] id:0x%x\r\n\r\n"
.text:10004EB1                 push    eax             ; Buffer
.text:10004EB2                 call    ds:sprintf
```

Syntax

C++ Copy

```
LANGID GetSystemDefaultUILanguage();
```

Return value

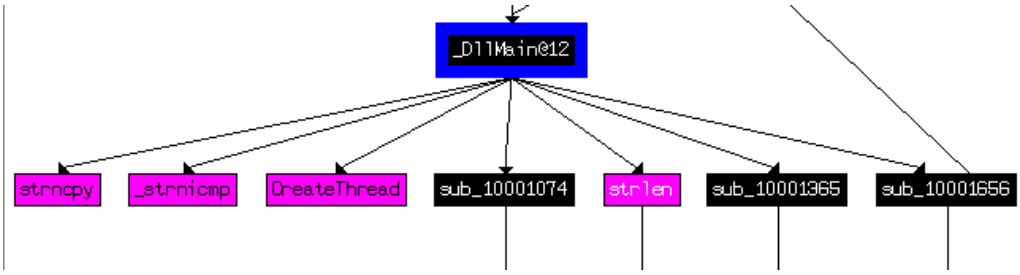
Returns the language identifier for the system default UI language of the operating system. For more information, see the Remarks section.

- From the MSDN description that you identify the default language for the operating system

```
.text:10004E79
.text:10004E79 ; int __cdecl identify_lang(SOCKET s)
.text:10004E79 identify_lang   proc near               ; CODE XREF: sub_1000FF58+4E5↓p
.text:10004E79
.text:10004E79 Buffer          = byte ptr -400h
.text:10004E79 var_3FF         = byte ptr -3FFh
.text:10004E79 s               = dword ptr  8
```

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

- strncpy-_strnicmp-CreateThread-sub_10001074-strlen-sub_10001365-sub_10001656 → windows API function



- oopps , The number of windows API function is very large

-> 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?

```
.text:10001341
.text:10001341 loc_10001341:                          ; CODE XREF: sub_10001074+10F↑j
.text:10001341                                        ; sub_10001074+1B0↑j ...
.text:10001341                 mov     eax, off_10019020 ; "[This is CTI]30"
.text:10001346                 add     eax, 13
.text:10001349                 push    eax             ; String
.text:1000134A                 call    ds:atoi
.text:10001350                 imul    eax, 1000
.text:10001356                 pop     ecx
.text:10001357                 push    eax             ; dwMilliseconds
.text:10001358                 call    ds:Sleep
.text:1000135E                 xor     ebp, ebp
.text:10001360                 jmp     loc_100010B4
.text:10001360 sub_10001074    endp
```

- After searching MSDN, I understood that the value measured in milliseconds is the time value to execution the function sleep API, and therefore it will be in sleep mode after 1000 milliseconds (1 second) of the function execution, and then it will be in sleep mode for 30 seconds

Syntax

C++ Copy

```cpp
void Sleep(
  [in] DWORD dwMilliseconds
);
```

Parameters

`[in] dwMilliseconds`

The time interval for which execution is to be suspended, in milliseconds.

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

- protocol – type – af

```
.text:100016FB          push    6               ; protocol
.text:100016FD          push    1               ; type
.text:100016FF          push    2               ; af
.text:10001701          call    ds:socket
```

-> 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?

- IPROTO_TCP : 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.

- SOCK_STREAM : 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).

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

```
.text:100016FB                 push    IPPROTO_TCP     ; protocol
.text:100016FD                 push    SOCK_STREAM     ; type
.text:100016FF                 push    AF_INET         ; af
.text:10001701                 call    ds:socket
```

-> 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:10001650    sub_10001365    xor   ebp, ebp
.text:100030AF    sub_10002CCE    call  strcat
.text:10003DE2    sub_10003DC6    lea   edi, [ebp+var_813]
.text:10004326    sub_100042DB    lea   edi, [ebp+var_913]
.text:10004B15    sub_10004B01    lea   edi, [ebp+var_213]
.text:10005305    sub_100052A2    jmp   loc_100053F6
.text:10005413    sub_100053F9    lea   edi, [ebp+var_413]
.text:1000542A    sub_100053F9    lea   edi, [ebp+var_213]
.text:10005B98    sub_10005B84    xor   ebp, ebp
.text:100061DB    sub_10006196    in    eax, dx
.text:10006305    sub_100062E9    lea   edi, [ebp+var_1290]
.text:10006310    sub_100062E9    mov   [ebp+var_1294], ebx
.text:10006318    sub_100062E9    call  ??2@YAPAXI@Z; operator new(uint)
.text:10006476    sub_100062E9    lea   ecx, [ebp+var_1294]
.text:100064A9    sub_100062E9    push  [ebp+var_1294]
.text:1000671B    sub_1000664C    call  sub_1000620C
.text:10006C43    sub_10006BD5    jnz   short loc_10006C31
```

- in: This is the mnemonic of the instruction, which indicates that it is an input operation.

```
;    __try { // __except at loc_100061EF
and     [ebp+ms_exc.registration.TryLevel], 0
push    edx
push    ecx
push    ebx
mov     eax, 'VMXh'
mov     ebx, 0
mov     ecx, 0Ah
mov     edx, 'VX'
in      eax, dx
cmp     ebx, 'VMXh'
setz    [ebp+var_1C]
pop     ebx
pop     ecx
pop     edx
jmp     short loc_100061F6
```

- yes, used.

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

- I don't know if this ASCII or not.

- I don't understand that.

```
.data:1001D984                 db    0
.data:1001D985                 db    0
.data:1001D986                 db    0
.data:1001D987                 db    0
.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
```

-> 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?

- Sorry, the Python version is not compatible with the IDA version.

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

- press of button A.

```
.data:1001D985                 db    0
.data:1001D986                 db    0
.data:1001D987                 db    0
.data:1001D988 a1UUU7461Yu2u10 db '-1::',27h,'u<&u!=<&u746>1::',27h,'yu&!',27h,'<;2u106:101u3:',27h,'u'
.data:1001D9B3                 db    5
.data:1001D9B4 a46649u         db 27h,'46!<649u'
.data:1001D9BD                 db  18h
.data:1001D9BE a4940u          db '49"4',27h,'0u'
.data:1001D9C5                 db  14h
.data:1001D9C6 a49U            db ';49,&<&u'
.data:1001D9CE                 db  19h
.data:1001D9CF                 db  34h ; 4
.data:1001D9D0                 db  37h ; 7
```

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

- I think it's encrypting something.

Lab05-01.py

```
sea = ScreenEA()

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