Xiaoyi Zhou

Purdue University CIT581

Malware Forensics

Lab 5: IDA Pro

Due September 26, 2014

Instructor: Samuel Liles

Abstract

This lab is designed for letting us be familiar with IDA Pro. Since it is a rewritten version, I found that I become more familiar with how does IDA Pro analyze malware. Lab5.dll cannot be run at OllyDbg and part of codes cannot be displayed at IDA Pro free version. But there is always be a way to figure out the answer.

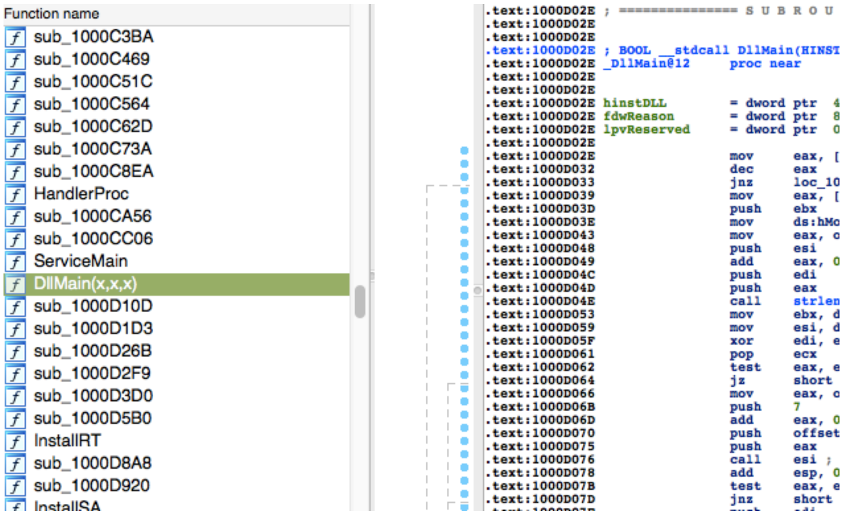
From what I analyze, the malware will intercept with network and TCP protocol. It might execute DNS poison attack. Moreover, it will start a remote shell and start conversation with hacker.

Steps of Process with Reviewed Questions

1. What is the address of DllMain?

Using search text tab to search DllMain. There are a lot of entries containing DllMain, but the real DllMain( )function is located at .text:1000D02E

Or we don't have to use search tab. See the left side function name bar. Scroll censor down or up until we see DllMain(x, x, x), which is easier.

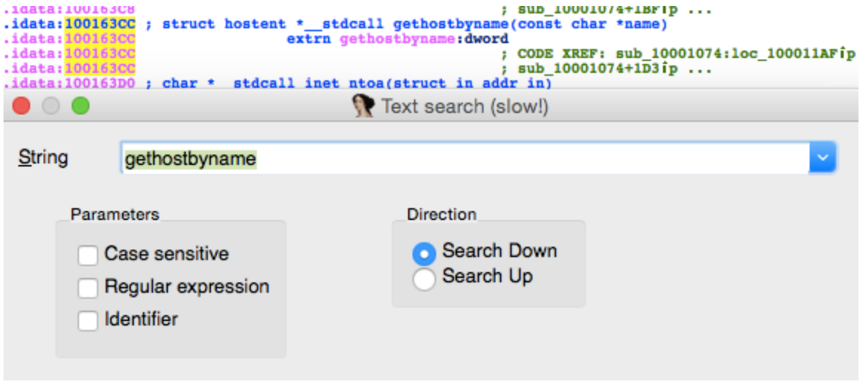


Function name
- sub_1000C3BA
- sub_1000C469
- sub_1000C51C
- sub_1000C564
- sub_1000C62D
- sub_1000C73A
- sub_1000C8EA
- HandlerProc
- sub_1000CA56
- sub_1000CC06
- ServiceMain
- DllMain(x,x,x)
- sub_1000D10D
- sub_1000D1D3
- sub_1000D26B
- sub_1000D2F9
- sub_1000D3D0
- sub_1000D5B0
- InstallRT
- sub_1000D8A8
- sub_1000D920
- InstallSA

```
.text:1000D02E ; =============== S U B R O U
.text:1000D02E
.text:1000D02E
.text:1000D02E ; BOOL __stdcall DllMain(HINST
.text:1000D02E _DllMain@12     proc near
.text:1000D02E
.text:1000D02E
.text:1000D02E hinstDLL        = dword ptr  4
.text:1000D02E fdwReason       = dword ptr  8
.text:1000D02E lpvReserved     = dword ptr  0
.text:1000D02E
.text:1000D02E                 mov     eax, [
.text:1000D032                 dec     eax
.text:1000D033                 jnz     loc_10
.text:1000D039                 mov     eax, [
.text:1000D03D                 push    ebx
.text:1000D03E                 mov     ds:hMo
.text:1000D043                 mov     eax, o
.text:1000D048                 push    esi
.text:1000D049                 add     eax, 0
.text:1000D04C                 push    edi
.text:1000D04D                 push    eax
.text:1000D04E                 call    strlen
.text:1000D053                 mov     ebx, d
.text:1000D059                 mov     esi, d
.text:1000D05F                 xor     edi, e
.text:1000D061                 pop     ecx
.text:1000D062                 test    eax, e
.text:1000D064                 jz      short
.text:1000D066                 mov     eax, o
.text:1000D06B                 push    7
.text:1000D06D                 add     eax, 0
.text:1000D070                 push    offset
.text:1000D075                 push    eax
.text:1000D076                 call    esi ;
.text:1000D078                 add     esp, 0
.text:1000D07B                 test    eax, e
.text:1000D07D                 jnz     short
```

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

I try to use same method that finds DllMain at question one. However, gethostbyname seemingly doesn't appear at left side function bar. Therefore I use traditional method that opened searching box and type gethostbyname( ).



```
.idata:100163C8                                          ; sub_10001074+1BF↑p ...
.idata:100163CC ; struct hostent *__stdcall gethostbyname(const char *name)
.idata:100163CC                 extrn gethostbyname:dword
.idata:100163CC                                          ; CODE XREF: sub_10001074:loc_100011AF↑p
.idata:100163CC                                          ; sub_10001074+1D3↑p ...
.idata:100163D0 ; char * stdcall inet ntoa(struct in addr in)
```

String: gethostbyname

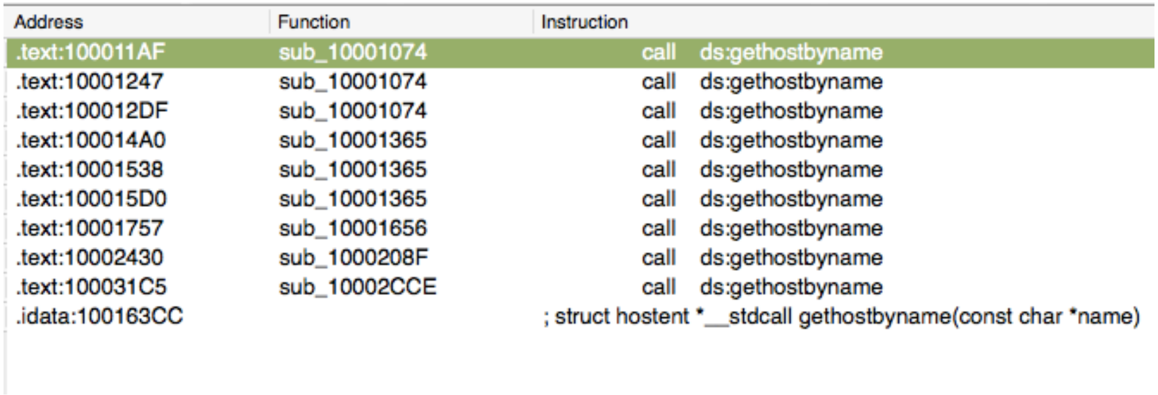
Parameters
- ☐ Case sensitive
- ☐ Regular expression
- ☐ Identifier

Direction
- ● Search Down
- ○ Search Up

gethostbyname( )function is located at .idada:100163CC. Just directly search the name of the function at impart window and we will get the answer.

3. How many functions call gethostbyname?

Open searching box and type gethostbyname( ); Count the locations that contain gethostname( ) except the one at data section because it is import of gethostbyname( ).



Address	Function	Instruction
.text:100011AF	sub_10001074	call ds:gethostbyname
.text:10001247	sub_10001074	call ds:gethostbyname
.text:100012DF	sub_10001074	call ds:gethostbyname
.text:100014A0	sub_10001365	call ds:gethostbyname
.text:10001538	sub_10001365	call ds:gethostbyname
.text:100015D0	sub_10001365	call ds:gethostbyname
.text:10001757	sub_10001656	call ds:gethostbyname
.text:10002430	sub_1000208F	call ds:gethostbyname
.text:100031C5	sub_10002CCE	call ds:gethostbyname
.idata:100163CC		; struct hostent *__stdcall gethostbyname(const char *name)

In this case the last entry idata:100163CC should be excluded.

First time call .text:100011AF

Second time call .text: 10001247

Third time call .text :100012DF

Fourth time call .text:1000014A0

Fifth time call .text: 10001538

Sixth time call .text:100015D0

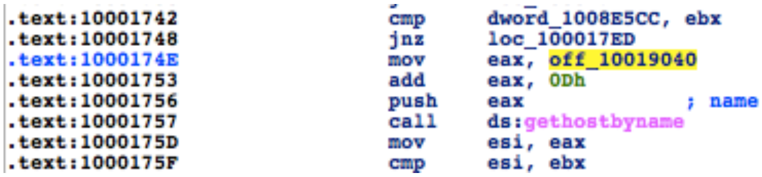
Seventh time call .text:10001757

Eighth time call .text:20002430

Ninth time call .text:10031C5

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

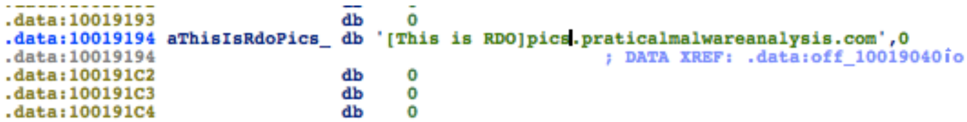
The function call gethostbyname only takes on parameter. The register after push should be the parameter location. In this case, eax is a string pointer at 0x1000174E. The string is located at offset 0x100019404.



```
.text:10001742                 cmp     dword_1008E5CC, ebx
.text:10001748                 jnz     loc_100017ED
.text:1000174E                 mov     eax, off_10019040
.text:10001753                 add     eax, 0Dh
.text:10001756                 push    eax             ; name
.text:10001757                 call    ds:gethostbyname
.text:1000175D                 mov     esi, eax
.text:1000175F                 cmp     esi, ebx
```

Double click on off_10019404, I can see the string value stored here. "[This is

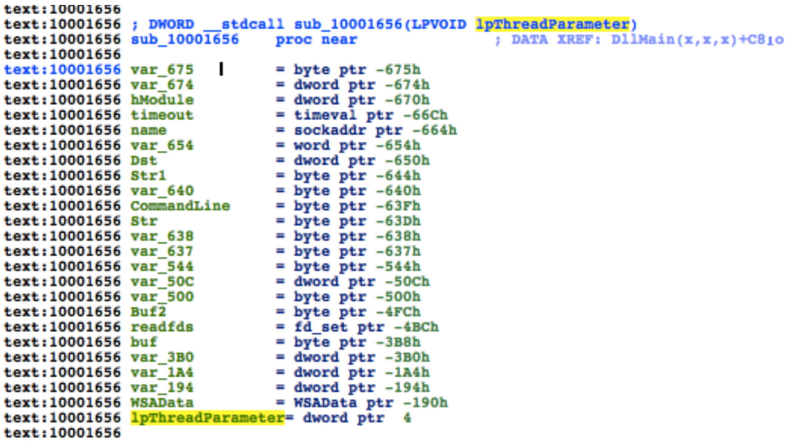
RDO]pics.practicalmalwareanalysis.com" is stored at off_1009040 and then moved to eax. Actually it should drop the double quotation when we analyze it. eax is added by 0xD which is 13 in decimal. Therefore, the pointer pointing to eax should be move right 13bytes. So I wipe 13 characters from [This is RDO]pics.practicalmalwareanalysis.com. The result will be pics.practicalmalwareanalysis.com.



```
.data:10019193                  db    0
.data:10019194 aThisIsRdoPics_  db '[This is RDO]pics.praticalmalwareanalysis.com',0
.data:10019194                                         ; DATA XREF: .data:off_10019040↑o
.data:100191C2                  db    0
.data:100191C3                  db    0
.data:100191C4                  db    0
```

Put the censor at the address 0x00174E three lines above the gethostbyname, the 0x10019040 located the DNS address of pics.praticalmalwareanalysis.com

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



```
text:10001656
text:10001656 ; DWORD __stdcall sub_10001656(LPVOID lpThreadParameter)
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 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
```

Use censor or searching box to find ox10001656. Here the subroutine just takes one parameter. If we don't which one is considered as parameter at the left side list, we can check the function and see how many parameters are in brackets. Moreover, local variable is mapped with offset while parameter is pointed by a pointer. Except the parameters, the rest listed on left hand are local variable. We can count them as 23 local variables. I think using IDA Pro demo version is better than free version because demo version can give me more information.

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

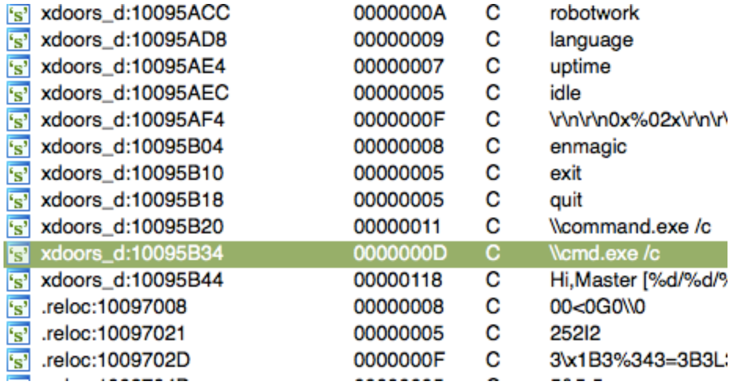
There is only one parameter like the following displayed.

lpThreadAddress =dword ptr 4.

In IDA Pro free version, the parameter is denoted as arg_0. Although arg_0 is more like a parameter, lpThreadAddress can indicate what is stored at this parameter.

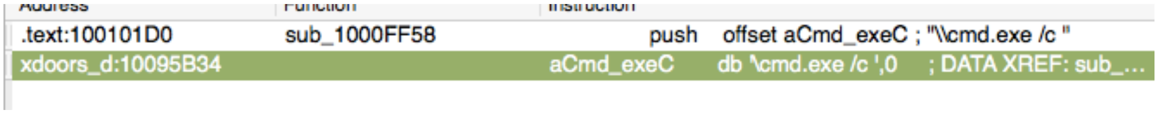
7. Use the Strings window to locate the string \cmd.exe /c in the disassembly. Where is it located?

According to this question, we should open string view at IDA pro. But when I open string view, I found there are too many strings. It is a little bit time consuming to find a specific string at string view.



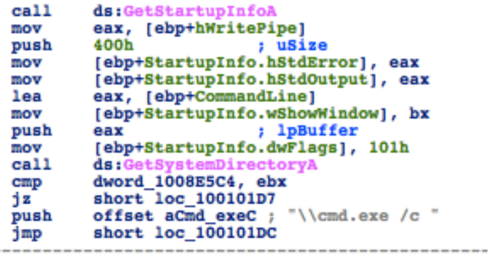
xdoors_d:10095ACC	0000000A	C	robotwork
xdoors_d:10095AD8	00000009	C	language
xdoors_d:10095AE4	00000007	C	uptime
xdoors_d:10095AEC	00000005	C	idle
xdoors_d:10095AF4	0000000F	C	\r\n\r\n0x%02x\r\n\r\
xdoors_d:10095B04	00000008	C	enmagic
xdoors_d:10095B10	00000005	C	exit
xdoors_d:10095B18	00000005	C	quit
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/%
.reloc:10097008	00000008	C	00<0G0\\0
.reloc:10097021	00000005	C	252l2
.reloc:1009702D	0000000F	C	3\x1B3%343=3B3L:

So I think a more convenient way is to type the key words at searching box and we can find that the string is located at 0x10095B34 and took as parameter at 0x100101D0.



Address	Function	Instruction	
.text:100101D0	sub_1000FF58	push	offset aCmd_exeC ; "\\cmd.exe /c "
xdoors_d:10095B34		aCmd_exeC	db '\cmd.exe /c ',0 ; DATA XREF: sub_…

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



```
call    ds:GetStartupInfoA
mov     eax, [ebp+hWritePipe]
push    400h            ; uSize
mov     [ebp+StartupInfo.hStdError], eax
mov     [ebp+StartupInfo.hStdOutput], eax
lea     eax, [ebp+CommandLine]
mov     [ebp+StartupInfo.wShowWindow], bx
push    eax             ; lpBuffer
mov     [ebp+StartupInfo.dwFlags], 101h
call    ds:GetSystemDirectoryA
cmp     dword_1008E5C4, ebx
jz      short loc_100101D7
push    offset aCmd_exeC ; "\\cmd.exe /c "
jmp     short loc_100101DC
-------------------------------------------------
```

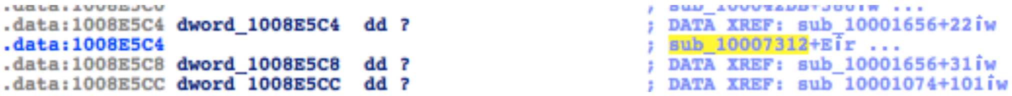
In the area of the code, there is getsystemdirectoryA()function, and reference involving starting information and showing window function.

For example "Encrypt Magic number For This Remote Shell Session".

At the area of address 10095B44 (about 10 lines), the IDA pro listed the malicious conversation that is obviously revoking the host. Therefore, I think the malware starts a shell remotely and try to talk with the host.

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.)

Navigate to the address and double click on dword_1008E5C4 because the current location cannot give us further useful information related to dword_1008E5C4.



```
.data:1008E5C4 dword_1008E5C4  dd ?                          ; DATA XREF: sub_10001656+22↑w
.data:1008E5C4                                               ; sub_10007312+E↑r ...
.data:1008E5C8 dword_1008E5C8  dd ?                          ; DATA XREF: sub_10001656+31↑w
.data:1008E5CC dword_1008E5CC  dd ?                          ; DATA XREF: sub_10001074+101↑w
```

There are data cross-reference and a function reference. We need to choose the data reference and see how it is set by the malware. We navigate the cross-reference of the key words, at the address of 0x1008E5C4 where we found the important lines:

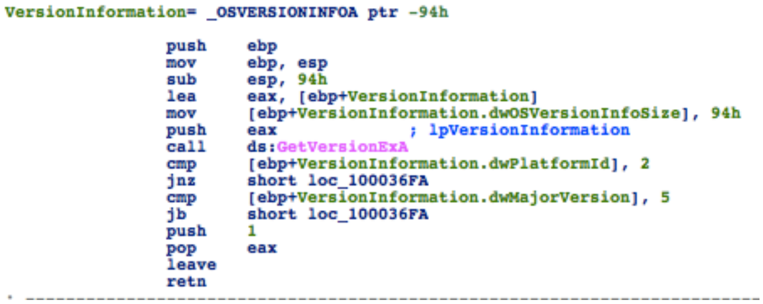
call sub_10003695

mov dword_1008E5C4, eax

The data in eax is stored in dword_1008E5C4, so we should figure out what is in eax and what does the function call do. Then we navigate to the address 0x10001678, here we find the two important lines:

call sub_10003695

mov dword_1008E5C4, eax



```
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
                cmp     [ebp+VersionInformation.dwPlatformId], 2
                jnz     short loc_100036FA
                cmp     [ebp+VersionInformation.dwMajorVersion], 5
                jb      short loc_100036FA
                push    1
                pop     eax
                leave
                retn
```

Put censor at sub_10003695, the function shows the OSVERSIONINFO as result.

see the code: VersionInformation=_OSVERSIONINFOA ptr -94h

Therefore, the global variable dword_10008E5C4 is used to store the OS information.

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)?

When I first navigate to the address 0x1000FF58, I didn't have any clue so that we directly research key words "robotwork". Then we go to the address containing robotwork which is 0x10001044C. Around that code area, we can find that:

```
call memcmp   ;the result is 0 indicating success.
add esp, 0ch
test eax, eax    ;the condition for the comparison. The value of ZF.
jnz short loc_100010468        ;since the memcmp returns value 0, ZF value is 0. This
                               code will not be executed. We can ignore it so far.
push [ebp+s]   ;this is variable used by sub_100052A2 ().
call sub_100052A2
```

Now, we should navigate to the function sub_100052A2(). We noticed that it is a subroutine of sub_1000FF58. In the function sub_100052A2(), the malware will get the information of registry value.

SOFTWARE\Microsoft\Windows\\CurrentVersion
WorkTime
Robot_WorkTime.

The function will return the three information as return value to sub_1000FF58(). Now we have to go back to 0x1000FF58 again.

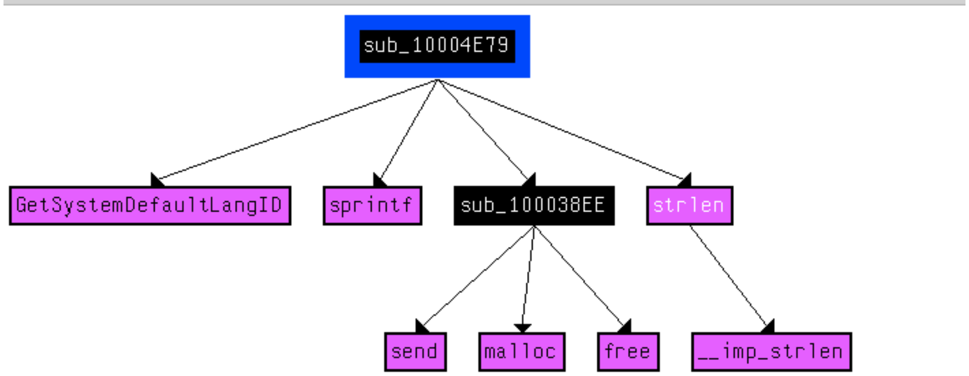
There are a lot of functions related to socket and system and malicious conversation at question 8. Such as GetTickCount() and GetCurrentDirectoryA(). Under GetTickCount(), we also can find closesocket(). Thus, I assume the function sub_1000FF68 will also start a shell. And it will display the value of work time, robot work time and windows current version at the shell window, which aims to create or fake the system.

11. What does the export PSLIST do?

It seems like the function sub_100036C3 at PSLIST wants to get the current OS version information and return the current version.

Then, comparing the current version with the version needed by the malware. If the two versions are same, the function sub_10006518() will be executed. In this function, we see the function CreateToolhelp32SnapShot() with the processID. When the function succeeds, it returns an open handle to the specified snapshot. In other words, this function takes a snapshot of the processes, heaps, modules, and threads used by the process.
Now the malware gets all the information of the process.
After the function CreateToolhelp32SnapShot(), the malware called the function sub_1000620C, I assume the malware DLL file took the process as the host exe to run.

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?



sub_100004E79 has the function calls (in called order):
GetSystemDefaultLangID()
sprintf()
strlen()
I assume this function is collecting the system language information.
Inside of the sub_100004E79, there is a function called sub_100038EE that has the API calls as following:
malloc()
send()
free()
Since the parameter in sub_100038EE is (socket s, int, int), I assume this function will

send the system language information gained by the former function call to a remote shell, or just send it to the network.

By right click sub_10004E79, we can rename it like SystemLanguageInfo.

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

Depth of 2 means a function has one more function call inside of it. Navigate to the location by pressing G. Using graphic version is easier to answer this question.

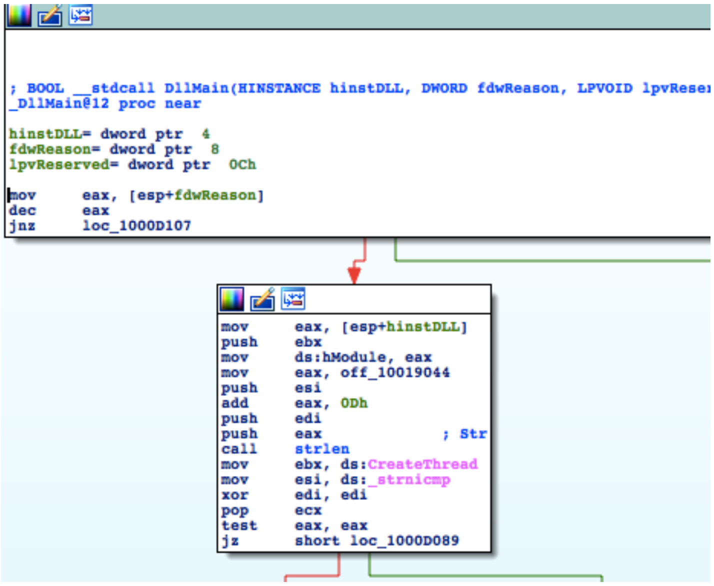
Directly called function as following:

strlen()

strnicmp()

Createthread()

strncpy()



```
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReser
_DllMain@12 proc near

hinstDLL= dword ptr  4
fdwReason= dword ptr  8
lpvReserved= dword ptr  0Ch

mov     eax, [esp+fdwReason]
dec     eax
jnz     loc_1000D107
```

```
mov     eax, [esp+hinstDLL]
push    ebx
mov     ds:hModule, eax
mov     eax, off_10019044
push    esi
add     eax, 0Dh
push    edi
push    eax             ; Str
call    strlen
mov     ebx, ds:CreateThread
mov     esi, ds:_strnicmp
xor     edi, edi
pop     ecx
test    eax, eax
jz      short loc_1000D089
```

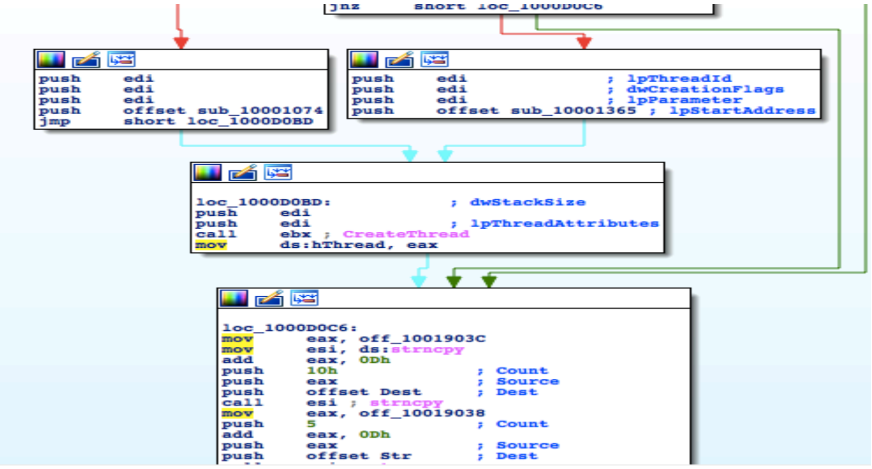
Inside the DllMain(), the functions such as strlen(), sub_10001074, sub_10001365, sub_10001656 has more than 1 depth.

strlen() contains imp_strlen().

sub_10001074 contains strchr(), inet_addr(), gethostbyname(), memcpy(), strcpy(), atoi(), and sleep().

sub_10001365 contains almost the same API function calls.

sub_10001656 contains sleep(), WSAStartup(), GetProcAddress(), closesocket(), socket(), imp_printf(), WSAGetLastError(), memcmp(), ntohs(), connect(), memset(), send(), recv(), select(), createthread(), closehandle(), and exitthread().



```
jnz     short loc_1000D0C6
```

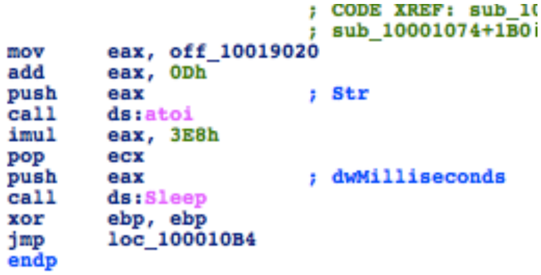
```
push    edi
push    edi
push    edi
push    offset sub_10001074
jmp     short loc_1000D0BD
```

```
push    edi                     ; lpThreadId
push    edi                     ; dwCreationFlags
push    edi                     ; lpParameter
push    offset sub_10001365 ; lpStartAddress
```

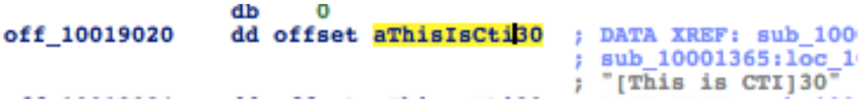
```
loc_1000D0BD:                   ; dwStackSize
push    edi
push    edi                     ; lpThreadAttributes
call    ebx ; CreateThread
mov     ds:hThread, eax
```

```
loc_1000D0C6:
mov     eax, off_1001903C
mov     esi, ds:strncpy
add     eax, 0Dh
push    10h                     ; Count
push    eax                     ; Source
push    offset Dest             ; Dest
call    esi ; strncpy
mov     eax, off_10019038
push    5                       ; Count
add     eax, 0Dh
push    eax                     ; Source
push    offset Str              ; Dest
```

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?



```
                        ; CODE XREF: sub_1(
                        ; sub_10001074+1B0↓
mov     eax, off_10019020
add     eax, 0Dh
push    eax             ; Str
call    ds:atoi
imul    eax, 3E8h
pop     ecx
push    eax             ; dwMilliseconds
call    ds:Sleep
xor     ebp, ebp
jmp     loc_100010B4
endp
```



```
                db    0
off_10019020    dd offset aThisIsCti30   ; DATA XREF: sub_100
                                         ; sub_10001365:loc_1
                                         ; "[This is CTI]30"
```

Look backward through the code.

mov eax, off_10019020 ; eax here is a pointer. eax = '[This is CIT]30', we can double click the offset, and double click on unk_100192AC.

add eax, 0Dh ; eax=30 (string here)

push eax

call ds:atoi ; eax=30 (integer here)

imul eax, 3E8h ; eax=30000 (milliseconds)

Therefore the process will sleep for 30 seconds.

At IDA Pro free version, the string is '[This is CIT]', without 30. But IDA Pro demo version will show full string. Here eax is a pointer register and it is added to 0xD (13 in decimal) so it moves left to right by 13bytes. 'This is CIT' got dropped. Function call atoi( ) convert string 30 to decimal 30. That is how I get the number 30.

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

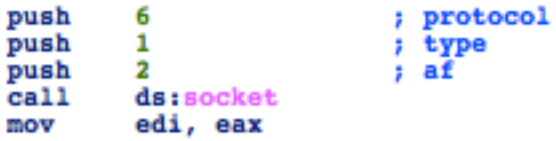
Use censor or press G to navigate to 0x10001701. A good way to check parameter is to find the register after operand push but not always. In this case, before the function call of socket(), there are three parameters like the following:

push 6

push 1

push 2

So the answer should be 6, 1, and 2.



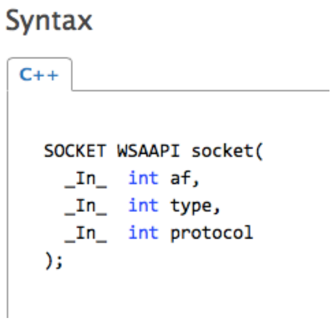
```
push    6               ; protocol
push    1               ; type
push    2               ; af
call    ds:socket
mov     edi, eax
```

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?

A part of following answers is retrieved from MSDN page of socket function.

http://msdn.microsoft.com/en-us/library/windows/desktop/ms740506(v=vs.85).aspx



Syntax

C++

```
SOCKET WSAAPI socket(
  _In_  int af,
  _In_  int type,
  _In_  int protocol
);
```

The three parameters are (6, 1, 2). According to the MSDN pages for socket, the three parameters indicates the address family as af, address type as type, protocol as protocol We click each number and choose use standard symbolic constant. However, each number can represent a lot of type names. We have to go through the type names and find the one we need.

af: The address family specification. Possible values for the address family are defined in the Winsock2.h header file.

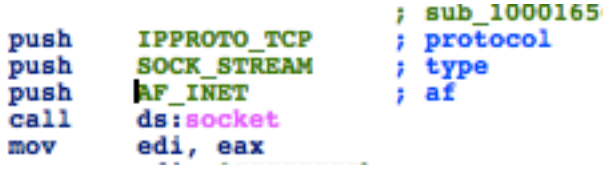
when af=2, af=AF_INET=The Internet Protocol version 4 (IPv4) address family.

type: The type specification for the new socket. Possible values for the socket type are defined in the Winsock2.h header file.

when type=1, type=SOCKET_STREAM=A socket type that provides sequenced, reliable, two-way, connection-based byte streams with an OOB data transmission mechanism.

when protocol is 6, protocol=IPPROTO_TCP=The Transmission Control Protocol (TCP).

On most x86 machines, function arguments are pushed on the stack from right side to left side. Therefore, 6 is protocol, 1 is address type, and 2 is address family. We can note 6 as IPROTO_TCP, 1 as SOCKET_STREAM, and 2 as AF_INET.



```
                                ; sub_1000165
push    IPPROTO_TCP             ; protocol
push    SOCK_STREAM             ; type
push    AF_INET                 ; af
call    ds:socket
mov     edi, eax
```

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?

Yes, we the string VMXh is used at this malware. We use search sequence function to find 0xED, and double click the function "in" at the address 0x100061DB. To get the value stored in eax, we moved the censor to upper code.

```
mov     eax, 564D5868h  ;VMXh
mov     ebx, 0
mov     exc, 0Ah
mov     edx, 5658h   ; VM
in      eax, dx
cmp     ebx, 564D5868h  ;VMXh. If two values are equal, Z is set to 1 otherwise 0.
```

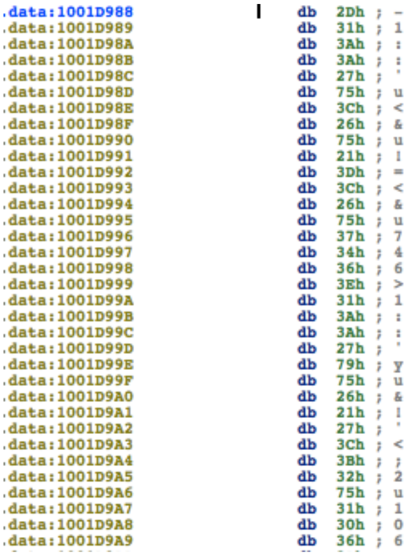
……

……

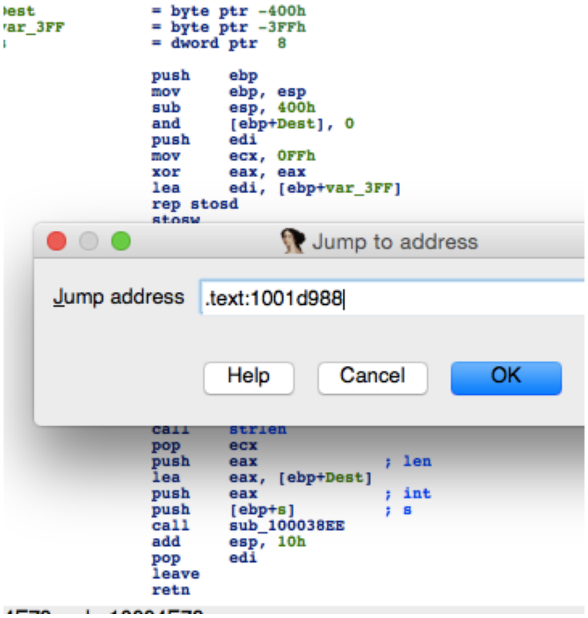
jmp short loc_100061F6; jump to the address 0x100061F6

In this code area, the value stored at eax is 564D5868h whose ASCII code is VMXh by right click on it. For the further evidence, double click on the code cross reference, and we found the operand like that:
mov [esp+8+var_8], offset aFoundVirtualMa: "Found Virtual Machine, Install Cancel."
which could be result of detecting virtual machine.

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
.data:1001D996                 db  37h ; 7
.data:1001D997                 db  34h ; 4
.data:1001D998                 db  36h ; 6
.data:1001D999                 db  3Eh ; >
.data:1001D99A                 db  31h ; 1
.data:1001D99B                 db  3Ah ; :
.data:1001D99C                 db  3Ah ; :
.data:1001D99D                 db  27h ; '
.data:1001D99E                 db  79h ; y
.data:1001D99F                 db  75h ; u
.data:1001D9A0                 db  26h ; &
.data:1001D9A1                 db  21h ; !
.data:1001D9A2                 db  27h ; '
.data:1001D9A3                 db  3Ch ; <
.data:1001D9A4                 db  3Bh ; ;
.data:1001D9A5                 db  32h ; 2
.data:1001D9A6                 db  75h ; u
.data:1001D9A7                 db  31h ; 1
.data:1001D9A8                 db  30h ; 0
.data:1001D9A9                 db  36h ; 6
```



```
est                 = byte ptr -400h
ar_3FF              = byte ptr -3FFh
                    = dword ptr  8

                    push    ebp
                    mov     ebp, esp
                    sub     esp, 400h
                    and     [ebp+Dest], 0
                    push    edi
                    mov     ecx, 0FFh
                    xor     eax, eax
                    lea     edi, [ebp+var_3FF]
                    rep stosd
                    stosw
```

Jump address .text:1001d988

```
                    call    strlen
                    pop     ecx
                    push    eax             ; len
                    lea     eax, [ebp+Dest]
                    push    eax             ; int
                    push    [ebp+s]         ; s
                    call    sub_100038EE
                    add     esp, 10h
                    pop     edi
                    leave
                    retn
```

If we want to jump to a specific location, we can press G and put the address on the message box. I found there isn't any operation but the data 0x2D.

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?

We saw the python script like the following:

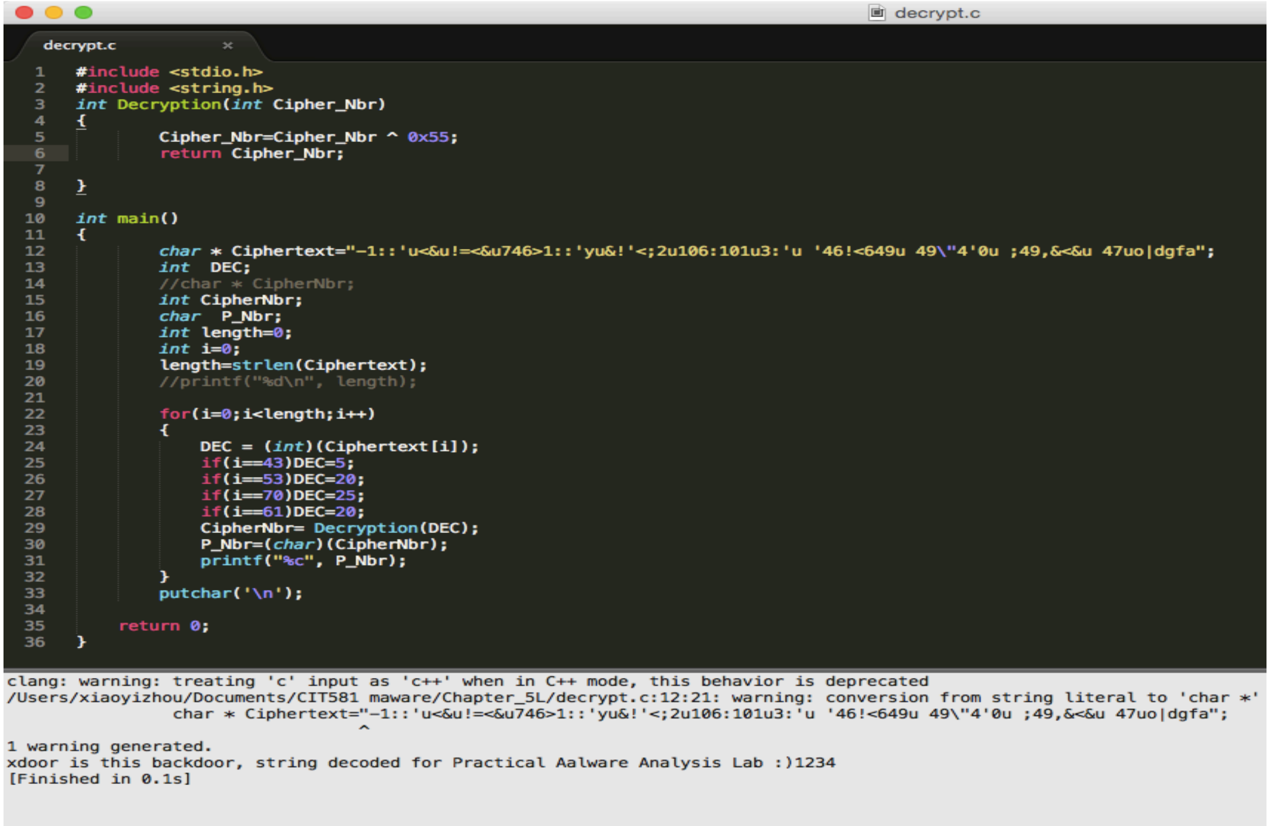
sea= ScreenEA() /* current location of the censor at 0x2D */

```
for i range (0x00, 0x50)        /*for loop starts at 0, end at 80 */
        b=byte(sea+i)           /*move the plaintext to the next one */
        decode_byte=b^0x55              /*encrypt the data with XOR 0x55, 80 in decimal */
        Patcgbyte(sea+i, decode_byte)
```

This is an encryption function.

Data from 0x2D to 0x61 has been encrypted. After we run the lab5-01.py, we can see the plaintext has been encrypted into readable character. See the ciphertext by press A. The ciphertext is xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234.

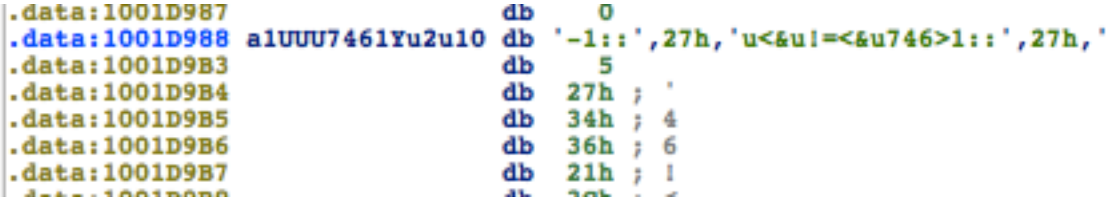
Actually we don't have to run the python script if our IDA version doesn't have the python plug in. In IDA Pro demo version, we don't have it. Therefore we can write a program to decrypt this string. See next page. However, we should manipulate and change some characters because 0x5, 0x14, 0x18 and 0x19 are special characters at ASCII table. So we need to pass the decimal value to cipher number variable.



```c
#include <stdio.h>
#include <string.h>
int Decryption(int Cipher_Nbr)
{
        Cipher_Nbr=Cipher_Nbr ^ 0x55;
        return Cipher_Nbr;

}

int main()
{
        char * Ciphertext="-1::'u<&u!=<&u746>1::'yu&!'<;2u106:101u3:'u '46!<649u 49\"4'0u ;49,&<&u 47uo|dgfa";
        int  DEC;
        //char * CipherNbr;
        int CipherNbr;
        char  P_Nbr;
        int length=0;
        int i=0;
        length=strlen(Ciphertext);
        //printf("%d\n", length);

        for(i=0;i<length;i++)
        {
            DEC = (int)(Ciphertext[i]);
            if(i==43)DEC=5;
            if(i==53)DEC=20;
            if(i==70)DEC=25;
            if(i==61)DEC=20;
            CipherNbr= Decryption(DEC);
            P_Nbr=(char)(CipherNbr);
            printf("%c", P_Nbr);
        }
        putchar('\n');

    return 0;
}
```

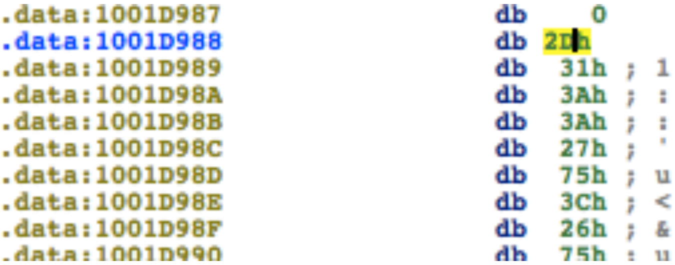
```
clang: warning: treating 'c' input as 'c++' when in C++ mode, this behavior is deprecated
/Users/xiaoyizhou/Documents/CIT581 maware/Chapter_5L/decrypt.c:12:21: warning: conversion from string literal to 'char *'
            char * Ciphertext="-1::'u<&u!=<&u746>1::'yu&!'<;2u106:101u3:'u '46!<649u 49\"4'0u ;49,&<&u 47uo|dgfa";
                        ^

1 warning generated.
xdoor is this backdoor, string decoded for Practical Aalware Analysis Lab :)1234
[Finished in 0.1s]
```

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



```
.data:1001D987                 db    0
.data:1001D988 a1UUU7461Yu2u10 db '-1::',27h,'u<&u!=<&u746>1::',27h,'
.data:1001D9B3                 db    5
.data:1001D9B4                 db  27h ; '
.data:1001D9B5                 db  34h ; 4
.data:1001D9B6                 db  36h ; 6
.data:1001D9B7                 db  21h ; !
```

If we press A on keyboard, the data area at 0x2D will be like this. Press D on keyboard again, there will be a massage box showing that convert between data and ASCII, just click ok, and the data will be converted to hex digit again.

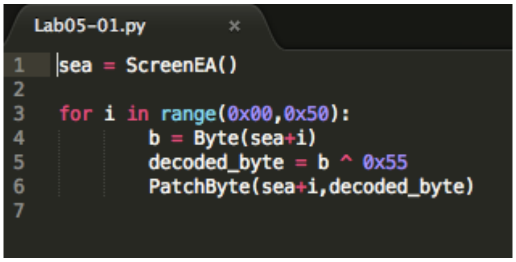


```
.data:1001D987                 db    0
.data:1001D988                 db 2Dh
.data:1001D989                 db  31h ; 1
.data:1001D98A                 db  3Ah ; :
.data:1001D98B                 db  3Ah ; :
.data:1001D98C                 db  27h ; '
.data:1001D98D                 db  75h ; u
.data:1001D98E                 db  3Ch ; <
.data:1001D98F                 db  26h ; &
.data:1001D990                 db  75h ; u
```

We can press A and D respectively to convert between hex data and ASCII code.

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

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



```
sea = ScreenEA()

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

See the decryption program at question 19. This is a python script which could be easily converted to C/C++. The script will perform an encryption function.

The plaintext is the unreadable data from 0x2D to 0x61.

The key is 0x55.

The cipher is XOR

The ciphertext is xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234.

Issues or Problems

I try to install IDA Pro python plugin but it didn't work. And sometimes demo version is better then free version.

Conclusion

I noticed that this malware is able to detect if it is running on virtual machine, which is interesting and seems like sneaky. I would like to see this kind of malware in future.