

个人信息

学号: 1911410

姓名: 付文轩

专业: 信息安全

lab 5-1

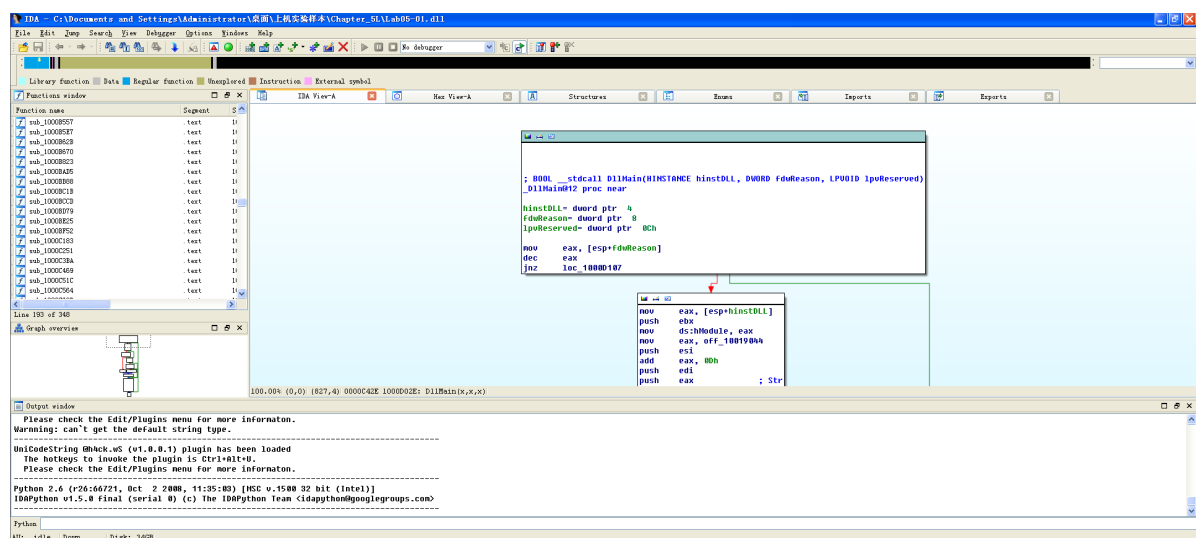
要求

- 1. What is the address of DllMain?
- 2. Use the Imports window to browse to gethostbyname. Where is the import located?
- 3. How many functions call gethostbyname?
- 4. Focusing on the call to gethostbyname located at 0x10001757, can you figure out which DNS request will be made?
- 5. How many local variables has IDA Pro recognized for the subroutine at 0x10001656?
- 6. How many parameters has IDA Pro recognized for the subroutine at 0x10001656?
- 7. Use the Strings window to locate the string `\cmd.exe /c` in the disassembly. Where is it located?
- 8. What is happening in the area of code that references `\cmd.exe /c`?
- 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.)
- 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)?
- 11. What does the export `PSLIST` do?
- 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?
- 13. How many Windows API functions does `DllMain` call directly? How many at a depth of 2?
- 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?

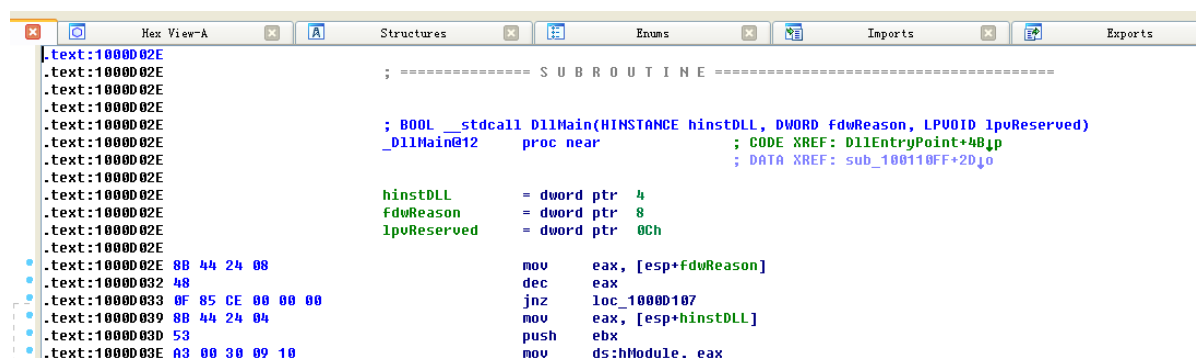
- 15. At 0x10001701 is a call to socket. What are the three parameters?
- 16. Using the MSDN page for socket and the named symbolic constants functionality in IDA Pro, can you make the parameters more meaningful? What are the parameters after you apply changes?
- 17. Search for usage of the in instruction (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?
- 18. Jump your cursor to 0x1001D988. What do you find?
- 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?
- 20. With the cursor in the same location, how do you turn this data into a single ASCII string?
- 21. Open the script with a text editor. How does it work?

实验过程

使用IDA-pro打开lab05-01.dll，得到初始的进入界面如下：

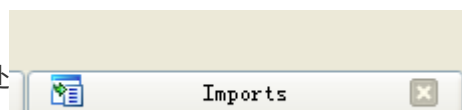


可以看到一进来的时候是图形化界面，并且界面中间就是DLLMain，也就是我们需要找的入口点。按下空格使其进入到反汇编的模式，得到结果：



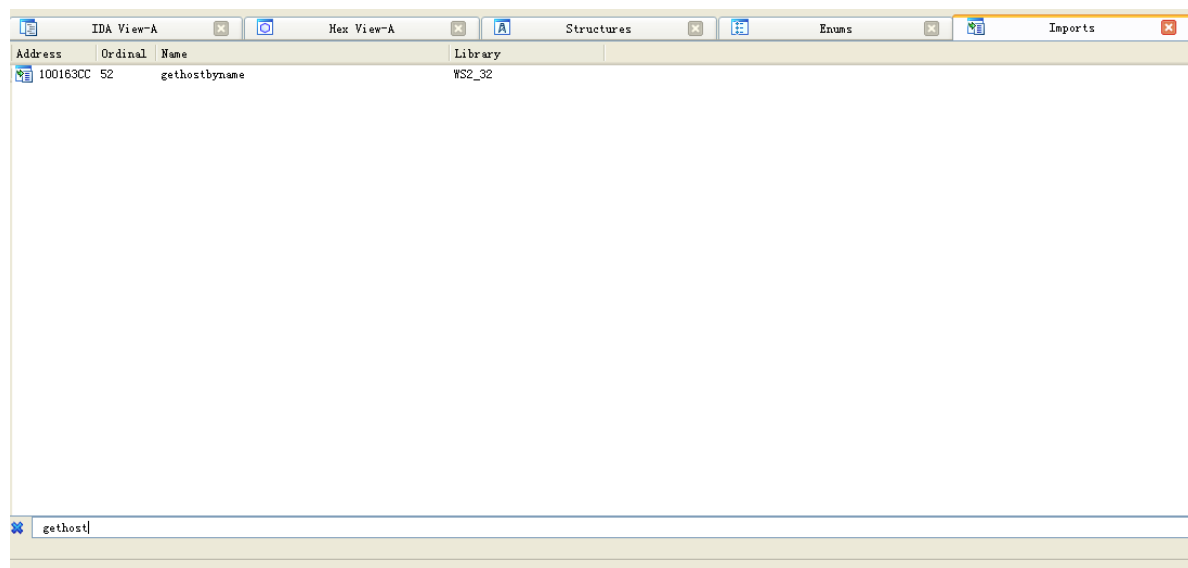
可以很清晰的看出DLLMain的位置

点击此处



切换到Imports窗口，寻找gethostbyname的位置

通过Ctrl+F的组合键，输入gethostbyname可以快速找到对应位置

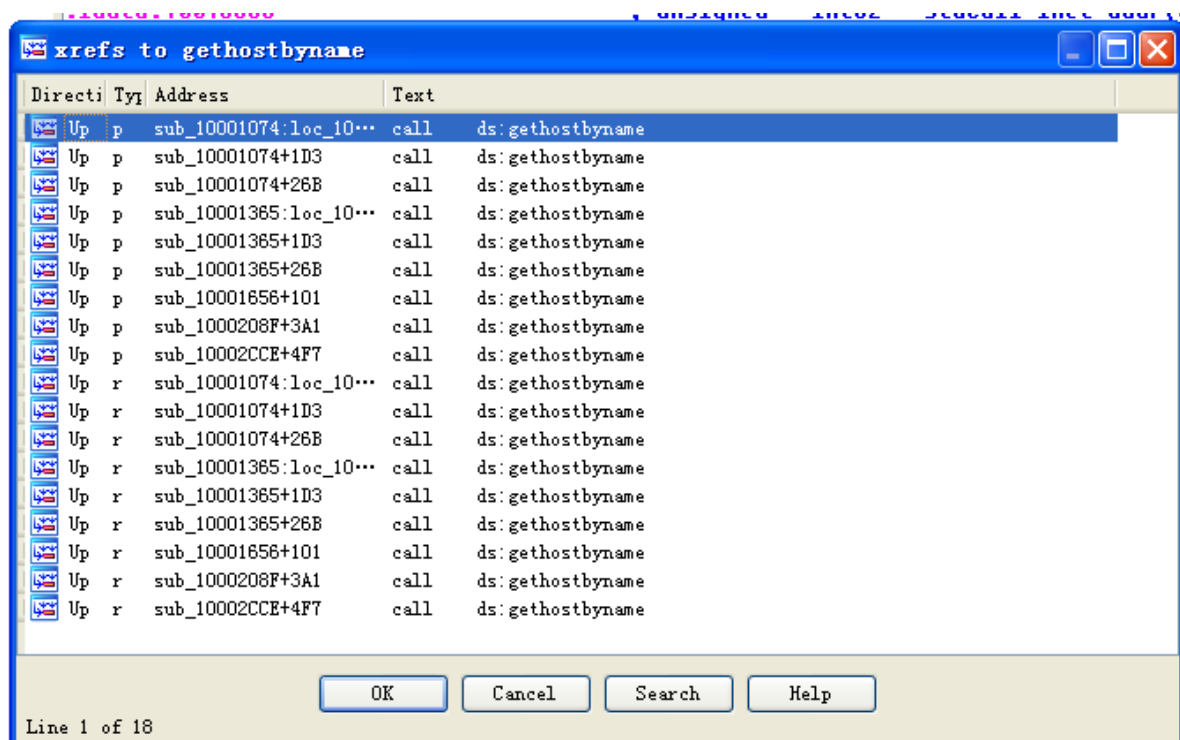


从上图中可以很明显的看出gethostbyname在0x100163CC处，双击查找到的结果可以得到下图：

```
.idata:100163C8 ?? ?? ?? ?? extrn inet_addr:dword ; CODE XREF: sub_10001074+11E1p
.idata:100163C8 ; struct hostent *_stdcall gethostbyname(const char *name) ; sub_10001074+1BF1p ...
.idata:100163CC ; struct hostent *_stdcall gethostbyname(const char *name) extrn gethostbyname:dword
.idata:100163CC ; CODE XREF: sub_10001074:loc_100011AF1p
.idata:100163CC ; sub_10001074+1D31p ...
.idata:100163CC ; char * _stdcall inet_ntoa(struct in_addr in_addr)
```

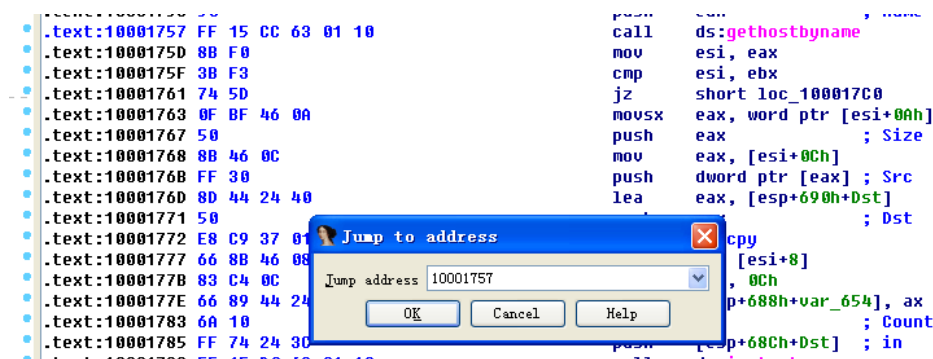
由此我们更加清楚得到了他所在的节是.idata

在gethostbyname处通过快捷键Ctrl+X使用交叉引用功能，可以看到窗口



在窗口中我们可以看到总共是有18个条目。其中在Type窗口中有9个p和9个r，r表示read读取，p表示函数的调用，经过观察可以发现，p中有不少是重复的，总共应该是只有5个函数对gethostbyname进行了调用。

使用快捷键G，跳转到我们需要关注的地址0x10001757

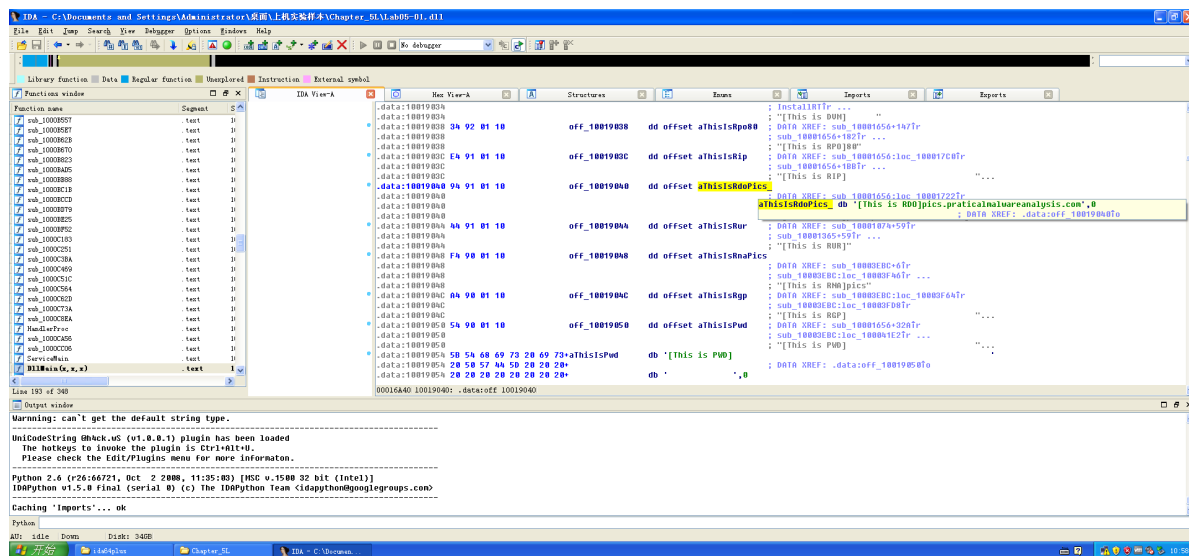


可以看到这里发生的一个函数调用，对于函数调用我们需要关心的应当是他的参数和局部变量，而通常情况下，对于调用某个函数来说，如果需要让其实现对不同需求给出相应的类似的效果的话，是需要将某个“不同”作为参数传递进函数的。所以这里我们先观察本次调用的参数。

一个函数的调用顺序是：参数入栈-返回地址入栈-栈帧切换，由此我们需要关心的应当是在

gethostbyname发生调用之前的位置，也就是 `mov eax, off_10019040`, `add eax, 0Dh`, `push eax`, `call ds:gethostbyname` 这三个操作。

可以发现最先对eax寄存器造成影响的是off_10019040这个地方的量，之后在这个的基础上增加了0Dh。根据这个地址，找到相应的量为：



从右边的注释可以看到这个访问的域名应该是：[This is RDO]pics.practicalmalwareanalysis.com。注意之前所说，在将这个量放入eax中以后，又增加了0Dh，经过计数可以发现，在域名前面的[This is RDO]刚好是13个字符的长度，也就是说在增加了0Dh以后，这个地址就正好指向字符“p”，也就是说在增加了0Dh以后这个eax中存放的就是真正需要访问的域名的字符串了。

同样是通过使用快捷键G，定位到0x10001656这个位置，得到结果如下图所示

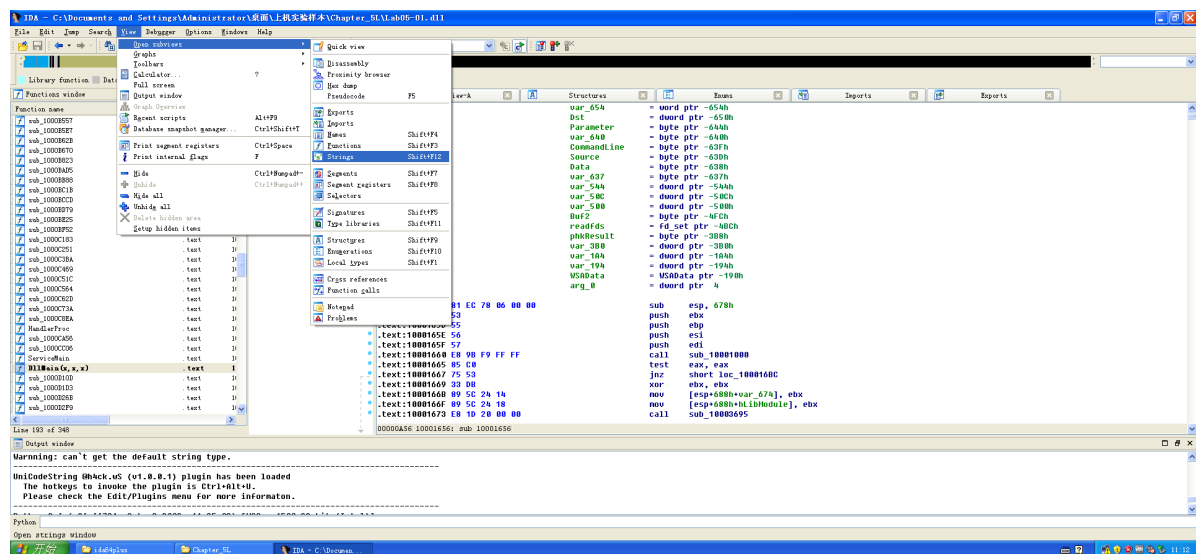
```

.text:10001656
.text:10001656 ; DWORD __stdcall sub_10001656(LPVOID)
.text:10001656 sub_10001656 proc near ; DATA XREF: DllMain(x,x,x)+C810
.text:10001656
.text:10001656 var_675 = byte ptr -675h
.text:10001656 var_674 = dword ptr -674h
.text:10001656 hLibModule = 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 Parameter = byte ptr -644h
.text:10001656 var_640 = byte ptr -640h
.text:10001656 CommandLine = byte ptr -63Fh
.text:10001656 Source = byte ptr -63Dh
.text:10001656 Data = byte ptr -638h
.text:10001656 var_637 = byte ptr -637h
.text:10001656 var_544 = dword ptr -544h
.text:10001656 var_50C = dword ptr -50Ch
.text:10001656 var_500 = dword ptr -500h
.text:10001656 Buf2 = byte ptr -4FCh
.text:10001656 readfds = fd_set ptr -4BCCh
.text:10001656 phkResult = byte ptr -388h
.text:10001656 var_380 = dword ptr -380h
.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 81 EC 78 06 00 00 sub esp, 678h
.text:1000165C 53 push ebx
.text:1000165D 55 push ebp

```

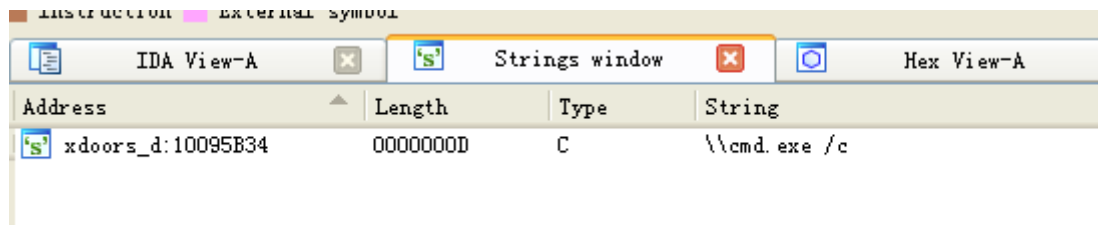
对于一个函数调用来说，参数的位置应当是在调用之前，局部变量的位置应当是在调用之后，从地址上来说就是参数的地址应当高于调用处地址，局部变量地址应当低于调用处地址。体现在相对位置上就是参数地址应当是正数，局部变量地址应当是负数。经过对上图中最右边的地址观察和计数可以发现，总共应该是有23个局部变量、1个参数。

之后寻找strings窗口



Address	Length	Type	String
.rdata:100185A4	00000005	C	vids
.rdata:10018F80	0000000C	C	SHELL32.dll
.rdata:100170A2	0000000A	C	GDI32.dll
.rdata:100170DA	0000000A	C	PSAPI.dll
.rdata:100170E4	0000000B	C	WS2_32.dll
.rdata:10017102	0000000D	C	iphlpapi.dll
.rdata:1001773E	0000000D	C	KERNEL32.dll
.rdata:10017920	0000000B	C	USER32.dll
.rdata:10017BA6	0000000D	C	ADVAPI32.dll
.rdata:10017C0C	0000000A	C	ole32.dll
.rdata:10017C16	0000000D	C	OLEAUT32.dll
.rdata:10017C68	0000000C	C	MSVFP32.dll
.rdata:10017CEC	0000000A	C	WINMM.dll
.rdata:10017EC8	0000000B	C	MSVCRT.dll
.rdata:10017FD2	00000009	C	xdl.dll
.rdata:10017FDB	0000000A	C	InstallRT
.rdata:10017FE5	0000000A	C	InstallSA
.rdata:10017FEF	0000000A	C	InstallSB
.rdata:10017FF9	00000007	C	FSLIST
.rdata:10018000	0000000C	C	ServiceMain
.rdata:1001800C	00000009	C	StartEIS
.rdata:10018015	0000000C	C	UninstallRT
.rdata:10018021	0000000C	C	UninstallSA
.rdata:1001802C	0000000C	C	UninstallSB
.data:10019054	0000004F	C	[This is PWD]
.data:100190A4	0000004F	C	[This is BGP]
.data:100190F4	00000012	C	[This is RNA]pics
.data:10019144	0000000E	C	[This is RIR]
.data:10019194	0000002E	C	[This is EIO]pics.practicalmalwareanalysis.com
.data:100191E4	0000004F	C	[This is RIP]

同样使用Ctrl+F搜索 \cmd.exe /c，找到结果如下：



观察寻找到的结果过可以发现， \cmd.exe /c的位置是在xdoors_d:10095B34

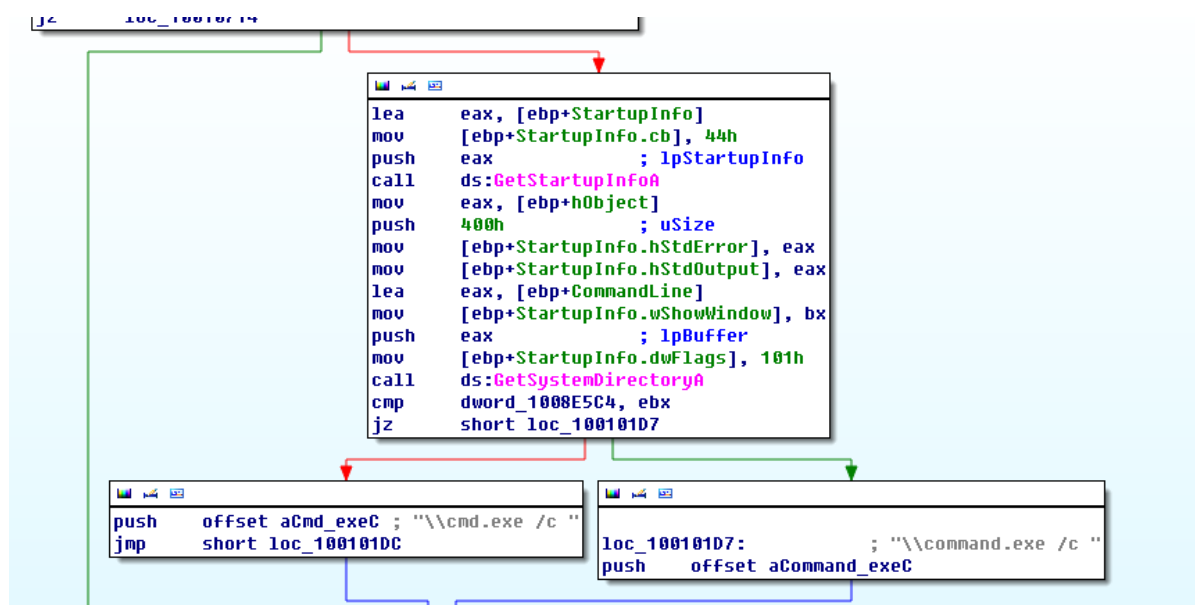
双击搜索到的条目，进入到反汇编窗口，可以看到

```
xdoors_d:10095B31 00 00 00 align 4
xdoors_d:10095B34 5C 63 6D 64 2E 65 78 65+aCmd_exeC db '\\cmd.exe /c ',0 ; DATA XREF: sub_1000FF58+2787d
xdoors_d:10095B41 00 00 00 align 4
xdoors_d:10095B44 00 00 00 align 4
```

发现在最右边，有个上箭头，也就是说这里发生了对当前字符串的调用。双击箭头处进入到调用的位置

```
.text:100101B8 C7 45 B8 01 01 00 00 push     eax
.text:100101C2 FF 15 D0 61 01 10 mov     [ebp+StartupInfo.dwFlags], 101h
.text:100101C8 39 1D C4 E5 08 10 call    ds:GetSystemDirectoryA
.text:100101CE 74 07 cmp     dword_1008E5C4, ebx
.text:100101D0 68 34 58 09 10 jz      short loc_100101D7
.text:100101D5 E8 05 push    offset aCmd_exeC ; "\\cmd.exe /c "
.text:100101D7 jmp     short loc_100101DC
;-----
.text:100101D7 loc_100101D7: push    offset aCommand_exeC ; "\\command.exe /c "
.text:100101D7 68 20 58 09 10 push    offset aCommand_exeC ; "\\command.exe /c "
.text:100101DC loc_100101DC: ; CODE XREF: sub_1000FF58+27D7j
```

可以看到在这个区域中出现了一个call，也就是函数调用，通过图像形式先大致观察一下



同图形化界面可以清晰的看出这里是一个类似于if else的语句，当dword_1008E5C4上的值和ebx中不一样的时候，会将\\cmd.exe /c字符串压入栈中。

往上翻发现另一个地方push了字符串

```
.text:10010092 0F B7 45 DC movzx   eax, [ebp+SystemTime.wYear]
.text:10010096 50 push    eax
.text:10010097 8D 85 40 F1 FF FF lea     eax, [ebp+Str]
.text:1001009D 68 44 58 09 10 push    offset aHiMasterDDDDDD ; "Hi,Master [%d/%d/%d %d:%d:%d]\r\nWelCom"...
.text:100100A2 50 push    eax ; Dest
.text:100100A3 FF 15 F4 62 01 10 call    ds:sprintf
.text:100100A9 83 C4 44 add     esp, 44h
```

双击点进去查看具体的字符串

```

; char aHiMasterDDDDDD[]
xdoors_d:10095844 48 69 2C 4D 61 73 74 65+aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah
xdoors_d:10095844 72 20 5B 25 64 2F 25 64+ ; DATA XREF: sub_1000FF58+14510
xdoors_d:10095844 2F 25 64 20 25 64 3A 25+ db 'WelCome Back...Are You Enjoying Today?',0Dh,0Ah
xdoors_d:10095844 64 3A 25 64 5D 00 0A 57+ db 0Dh,0Ah
xdoors_d:10095844 65 6C 43 6F 6D 65 20 42+ db 'Machine UpTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Secon'
xdoors_d:10095844 61 63 68 2E 2E 2E 41 72+ db 'ds]',0Dh,0Ah
xdoors_d:10095844 65 20 59 6F 75 20 45 6E+ db 'Machine IdleTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Seco'
xdoors_d:10095844 6A 6F 79 69 6E 67 20 54+ db 'nds]',0Dh,0Ah
xdoors_d:10095844 6F 64 61 79 3F 00 0A 0D+ db 0Dh,0Ah
xdoors_d:10095844 0A 4D 61 63 68 69 6E 65+ db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',0Dh,0Ah
xdoors_d:10095844 20 55 70 54 69 6D 65 20+ db 0Dh,0Ah,0
xdoors_d:10095C5C ; char asc_10095C5C[]

```

这里的字符串中发现有一句话提到了Remote Shell Session，推测这一段程序应该是发起一段远程的会话。

再往下看，发现里面有不少对字符串的操作

```

; CODE XREF: sub_1000FF58+2701j
lea     eax, [ebp+CommandLine]
push    eax ; Dest
call    strcat
pop     ecx
lea     eax, [ebp+Dst]
pop     ecx
push    0FFh ; Size
push    ebx ; Val
push    eax ; Dst
call    memset
add     esp, 0Ch

; CODE XREF: sub_1000FF58+2FA0j
xor     edi, edi

; CODE XREF: sub_1000FF58+3030j
push    ebx ; flags
lea     eax, [ebp+buf]
push    1 ; len
push    eax ; buf
push    [ebp+s] ; s
call    ds:recv
cmp     eax, 0FFFFFFFFh
jz      loc_10010714
cmp     eax, ebx
jz      loc_10010714
mov     al, [ebp+buf]
sub     al, byte ptr dword_1008E5D0

loc_10010714: ; CODE XREF: sub_1000FF58+3030j
; sub_1000FF58+110j
push    [ebp+s] ; s
call    ds:closesocket
cmp     [ebp+hFile], ebx
jz      short loc_1001072B
push    [ebp+hFile] ; hObject
call    ds:CloseHandle

loc_1001072B: ; CODE XREF: sub_1000FF58+3030j
cmp     [ebp+hObject], ebx
pop     edi
pop     esi
pop     ebx
jz      short loc_1001073C
push    [ebp+hObject] ; hObject
call    ds:CloseHandle

```

还有一些函数调用的名称值得关注，如socket等

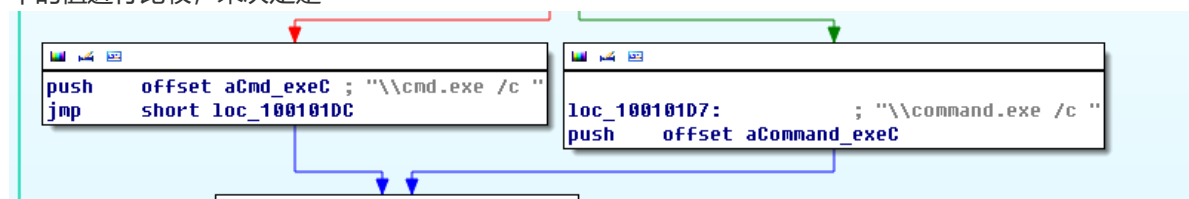

```

push    offset Dest      ; cp
call    ds:socket
mov     esi, eax
cmp     esi, 0FFFFFFFh
jz      loc_100107E1
push    offset Dest      ; cp
mov     [ebp+name.sa_family], 2
call    ds:inet_addr
push    offset byte_1008E5E8 ; S
mov     dword ptr [ebp+name.sa_d
call    ds:atoi
pop     ecx
push    eax              ; netsho
call    ds:ntohs
mov     word ptr [ebp+name.sa_da
lea     eax, [ebp+name]
push    10h             ; namele
push    eax              ; name
push    esi              ; s
call    ds:connect

```

综上，我认为这个样本的作用应当是开启一个远程的shell会话。

当我们回到100101C8这个位置，看见决定程序接下来走向的是dword_1008E5C4这个变量，他将和ebx中的值进行比较，来决定走



这两条路线中的哪一条。

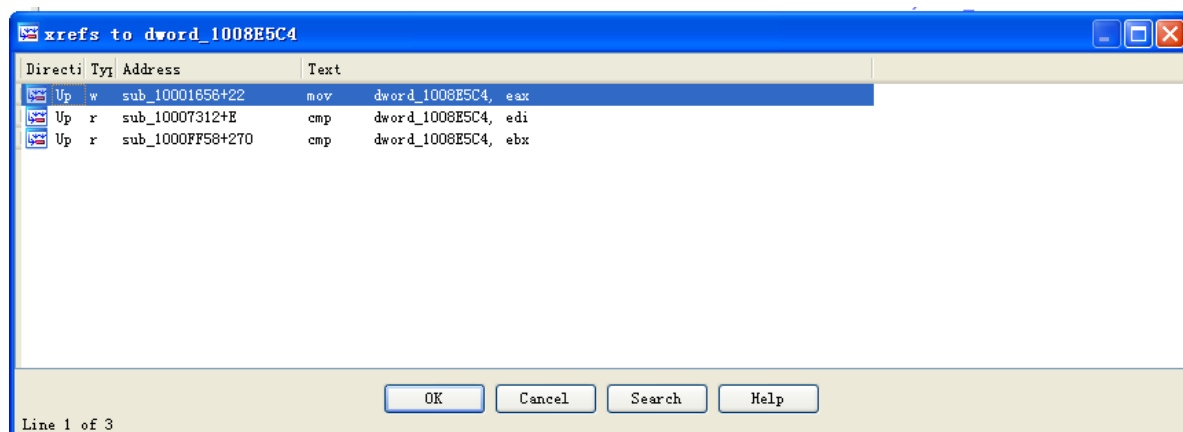
双击这个变量，发现这个变量是在.data段，但是这个变量的初值是没有给定的

```

.data:1008E5C4 ?? ?? ?? ??      dword_1008E5C4 dd ?          |          ; DATA XREF: sub_10001656+221w
.data:1008E5C4                  ; sub_10007312+E1r ...
.data:1008E5C4 00 00 00 00      ; DATA XREF: sub_10004477+0477

```

但是我们发现对这个变量有多次引用，想来在后面的引用过程中应该会对这个变量进行赋值。由此使用快捷键Ctrl+X查看对这个变量的交叉引用：



可以发现对这个变量总共有三次的引用，其中两次读一次是写，那么自然我们需要关注的就是这一次的写操作，双击定位到写操作位置。

```

.text:10001673 E8 1D 20 00 00      call sub_10003695
.text:10001678 A3 C4 E5 00 10      mov dword_1008E5C4, eax

```


观察调用的sub_100052A2函数

```

mov     ecx, [ebp+cbData] ; cbData
mov     eax, 0 ; ulOptions
mov     offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVe"...
mov     eax, 80000002h ; hKey
call    ds:RegOpenKeyExA
mov     eax, eax
mov     short loc_10005309
mov     [ebp+hKey] ; hKey
call    ds:RegCloseKey
jmp     loc_100053F6
-----
```

```

; CODE XREF: sub_100052A2+57↑j
mov     ebx
lea     eax, [ebp+cbData]
mov     esi
mov     eax ; lpcbData
lea     eax, [ebp+Str]
mov     ebx, ds:RegQueryValueExA
mov     eax ; lpData
```

往下看可以发现一些关键的函数，都是对注册表的操作。同时观察到调用了自己写的函数

```

push    eax
push    [ebp+s]
call    sub_100038EE
add     esp, 10h
```

双击进去查看

```

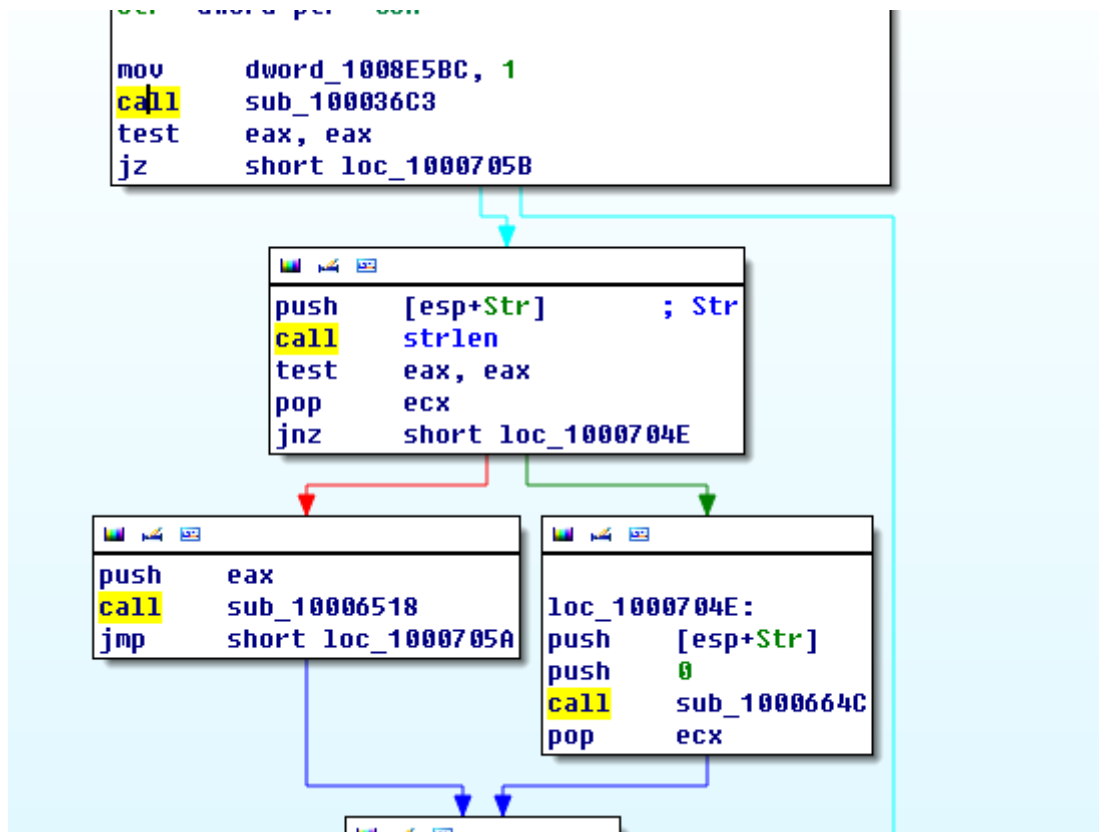
mov     edi, [ebp+len]
lea     eax, [edi+1]
push    eax ; !
call    ds:malloc
xor     edx, edx
pop     ecx

push    u
push    edi
push    esi
push    [ebp+s]
call    ds:send
or      edi, 0FFFFFFFFh
cmp     eax, edi
jz      short loc_10005309
mov     edi, eax

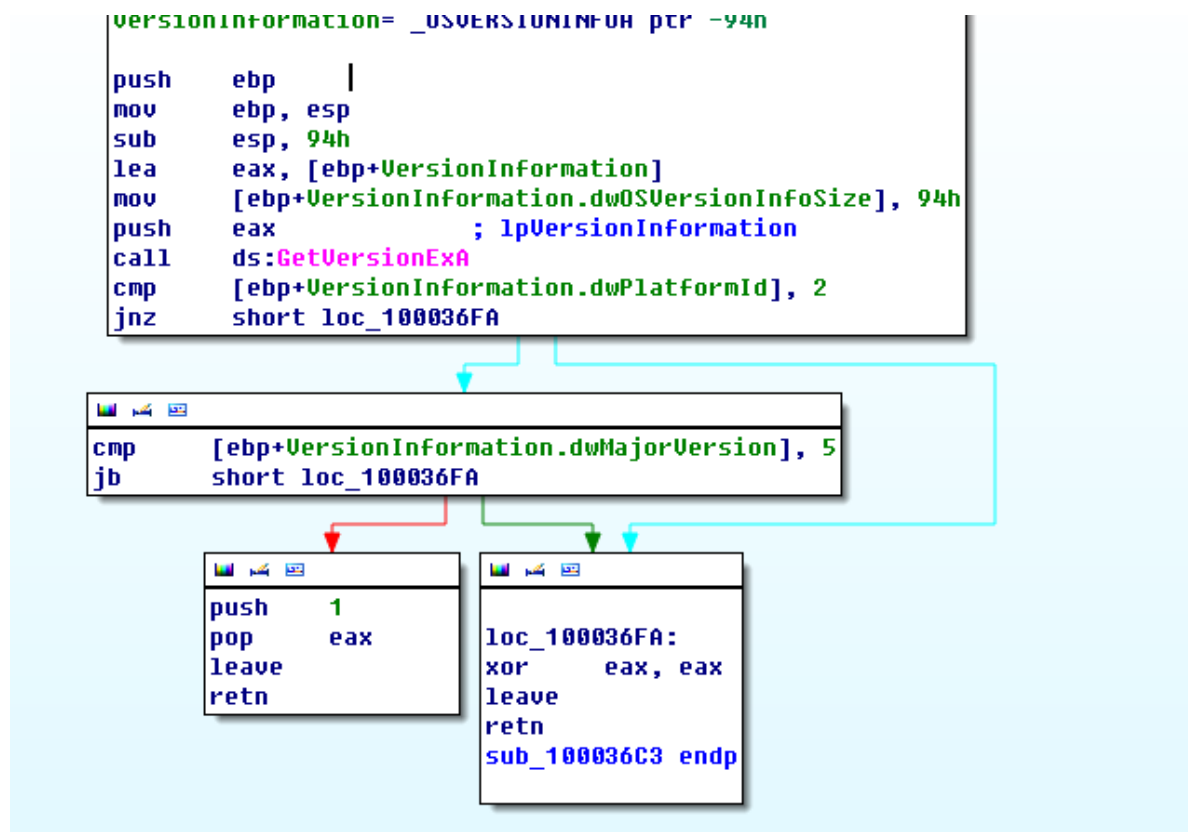
loc_10005309:
push    esi
call    ds:free
pop     ecx
mov     eax, edi
pop     edi
-----
```

发现里面有分配并释放空间、send发送信息。那么由此就可以推测出，当robotwork匹配成功以后，该程序会对注册表的信息进行访问，并向远端发送信息。

之后在View-open subviews-Exports中查看导出表并找到需要查看的PSLIST，双击定位以后我们发现



这个函数首先是调用了sub_100036C3这个函数，点进去查看



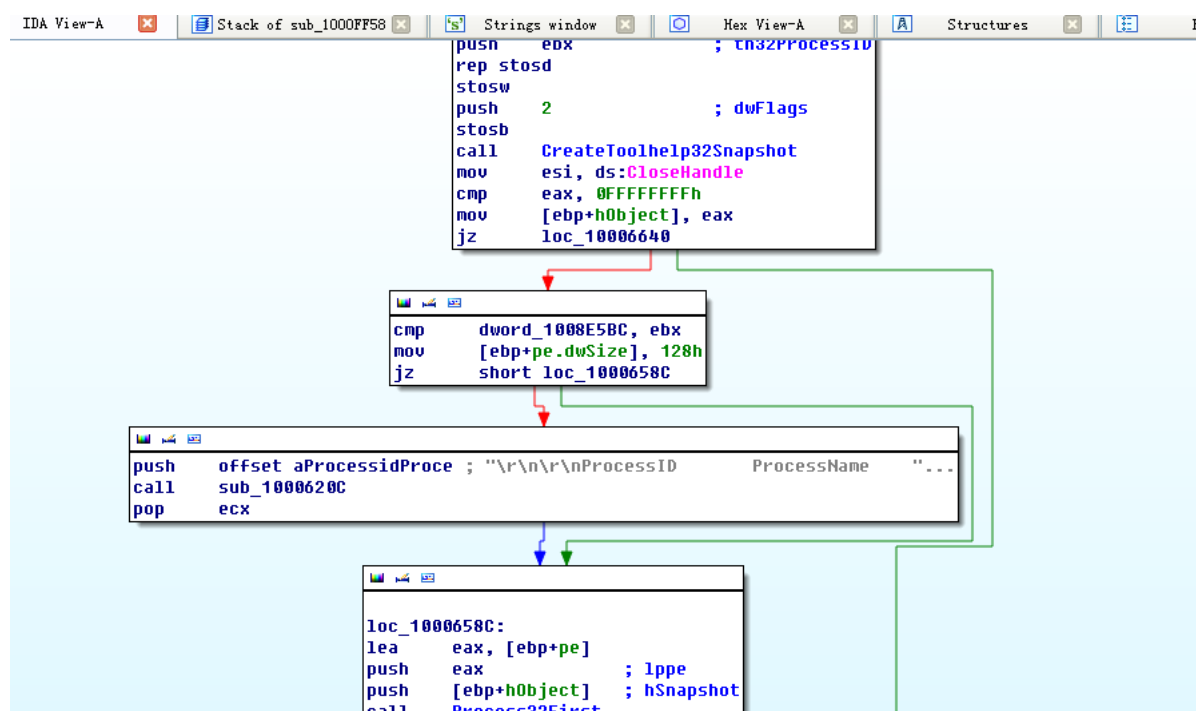
发现嗲用的这个函数中一样是需要验证当前操作系统的平台是否是Win32-NT平台，同时查看当前平台的版本是哪一种操作系统

查询了一下这个相关的信息

Windows 7	6.1	6	1	OSVERSIONINFOEX.wProductType == VE
Windows Server 2008 R2	6.1	6	1	OSVERSIONINFOEX.wProductType != VEF
Windows Server 2008	6.0	6	0	OSVERSIONINFOEX.wProductType != VEF
Windows Vista	6.0	6	0	OSVERSIONINFOEX.wProductType == VE
Windows Server 2003 R2	5.2	5	2	GetSystemMetrics(SM_SERVERR2) != 0
Windows Home Server	5.2	5	2	OSVERSIONINFOEX.wSuiteMask & VER_SI
Windows Server 2003	5.2	5	2	GetSystemMetrics(SM_SERVERR2) == 0
Windows XP Professional	5.2	5	2	(OSVERSIONINFOEX.wProductType == VE (SYSTEM_INFO.wProcessorArchitecture ==

可以发现如果是大于5的，其版本是在Vista以上。

回到之前的定位，我们发现当其往下执行时有两条路径，两条路径中都会有call函数。首先查看一下左边的调用



发现这个调用里call了另一个函数 `CreateToolhelp32Snapshot`，这个函数的功能是获取进程列表。同时还有ProcessID和ProcessName字样，并且作为参数传递给sub_1000620C这个函数，再查看这个函数

```

push    [ebp+Format]      ; Format
push    400h              ; MaxCount
push    eax               ; DstBuf
call    ds:_vsprintf
push    offset aA         ; "a"
push    offset aXinstall_dll ; "xinstall.dll"
call    ds:fopen
mov     esi, eax
add     esp, 18h
test    esi, esi
jz      short loc_10006265

```

```

lea     eax, [ebp+DstBuf]
push    eax
push    offset aS_0       ; "%s\n"
push    esi               ; File
call    ds:fprintf
push    esi               ; File
call    ds:fclose
add     esp, 10h

```

```

loc_10006265:
pop     esi
leave
retn

```

发现在后续的调用中用到了fprintf和fclose，这两个就是典型的对流操作的函数，也就是要将之前或得到的ID和Name写入到某个文件中

返回到之前的定位，查看一下右边函数的调用

```

rep stosd
push    2                 ; dwFlags
call    CreateToolhelp32Snapshot
cmp     eax, 0FFFFFFFFh
mov     [ebp+hObject], eax
jnz     short loc_100066DF

```

```

loc_100066DF:
mov     edi, offset aProcessIDProc     ProcessName "...
push    esi                       ; ArgList
mov     esi, ds:sprintf
lea     eax, [ebp+buf]
push    edi                       ; Format
push    eax                       ; Dest
mov     [ebp+pe.dwSize], 128h
call    esi                       ; sprintf
lea     eax, [ebp+buf]
push    eax                       ; buf
push    [ebp+s]                   ; s
call    sub_100038BB
add     esp, 10h
cmp     dword_1000E5BC, ebx
jz      short loc_10006720

```

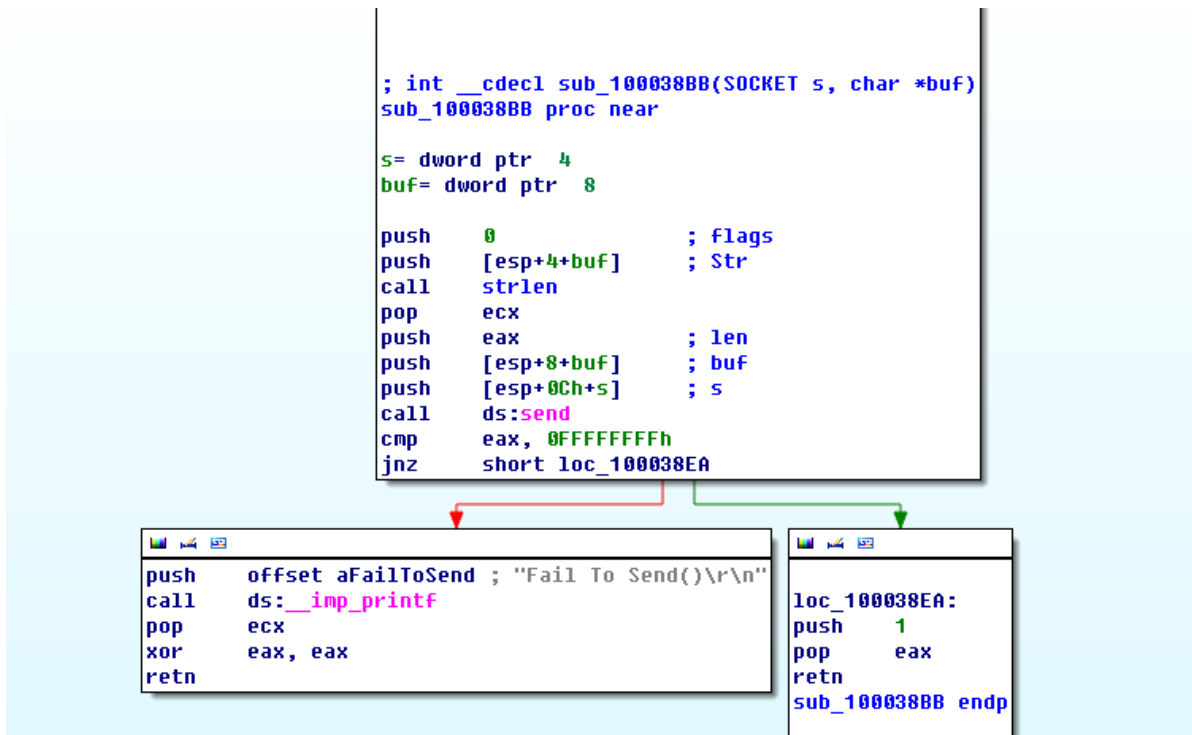
```

push    edi                       ; Format
call    sub_10006200

```

1000664C+8

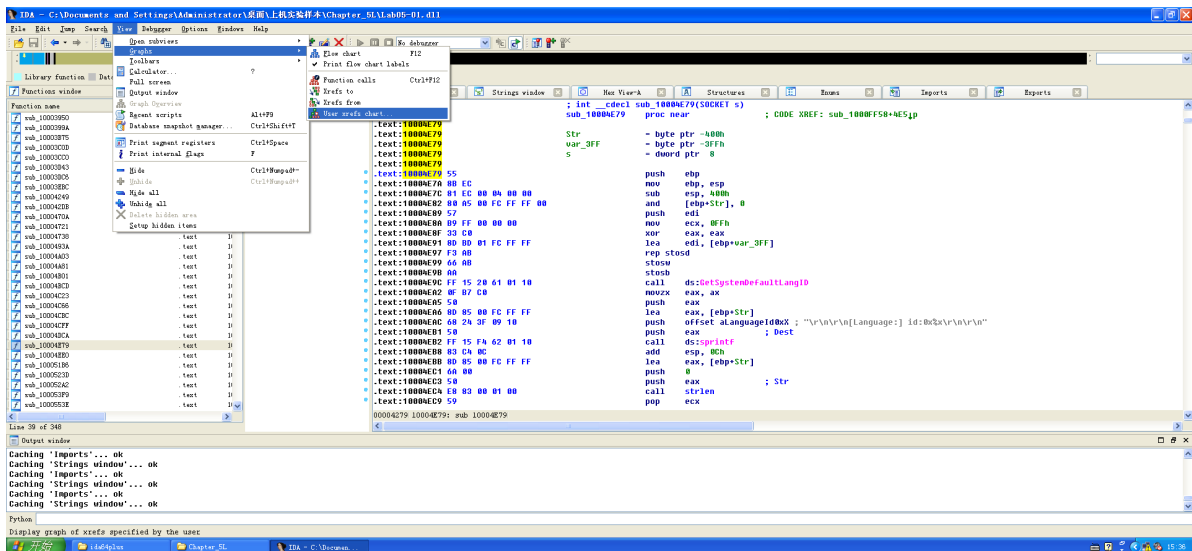
发现在右边的函数也有和刚刚类似的操作，进入到sub_10038BB这个函数中

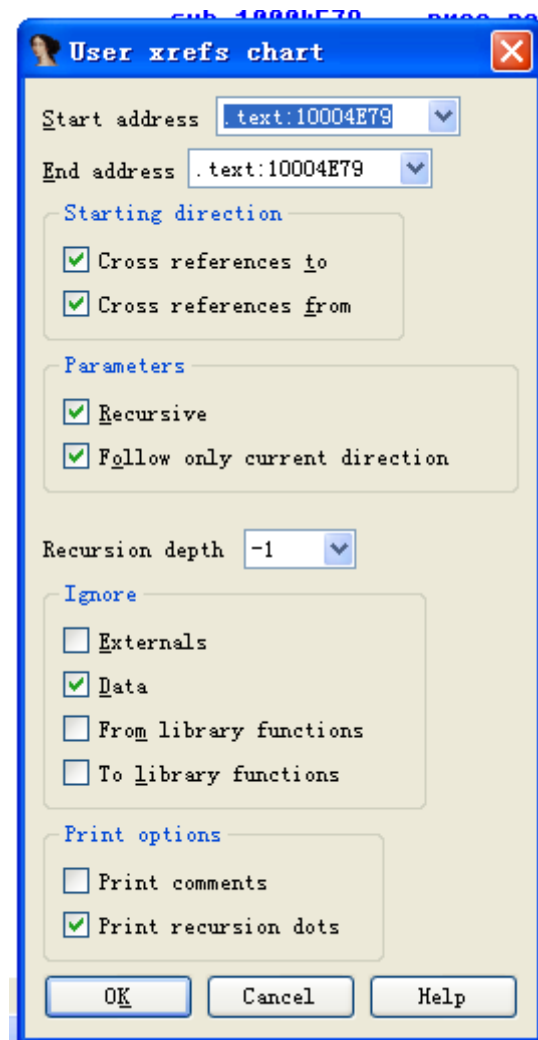


可以看见这个函数则是进行了send，也就是将之前获得的那些信息都通过远程会话发送信息。同时在上
面还可以注意到有关字符串比较的内容，由此猜测应该是查看进程列表，寻找是否有攻击者需要找到的
进程信息，并将这个发送给攻击者。

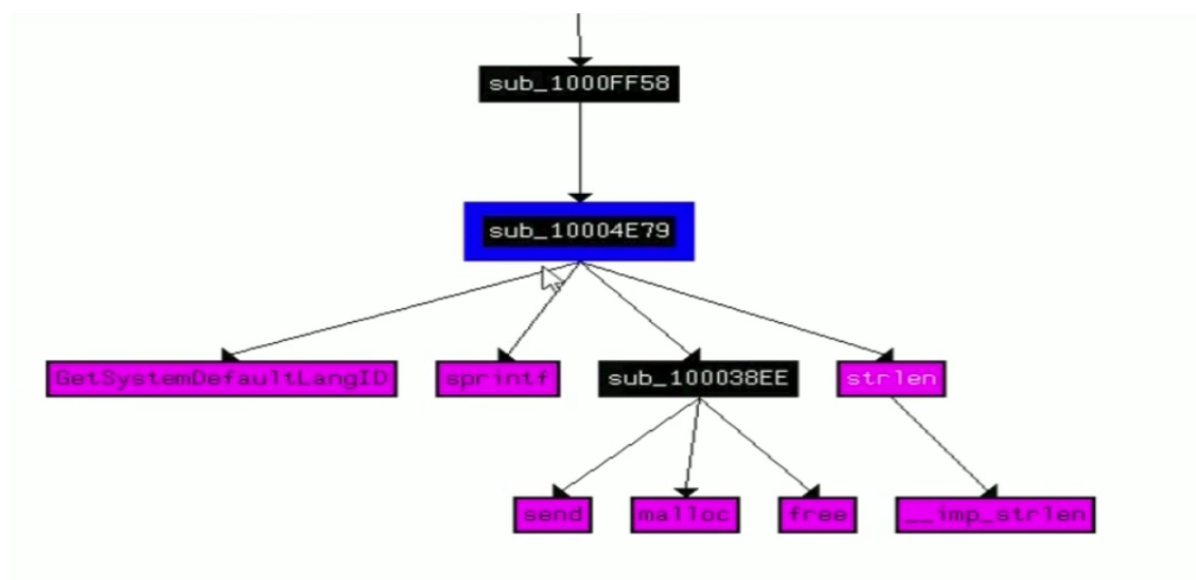
以上这些就是PSLIST的两个功能。

定位到sub_10004E79位置，通过view中的功能可以获取图模式



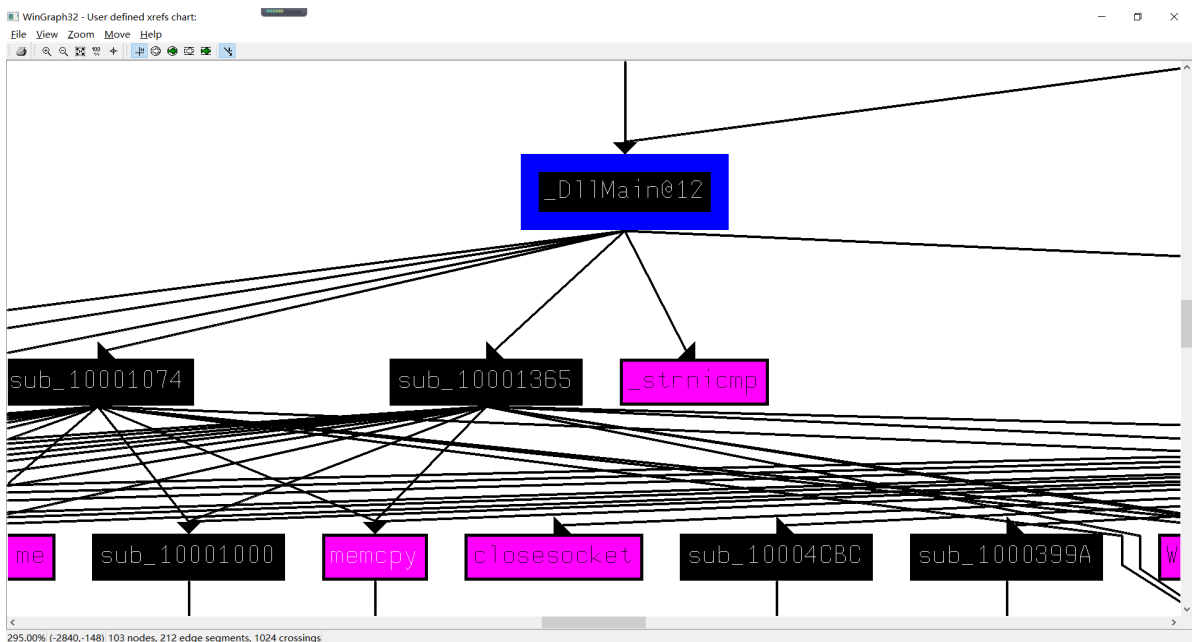
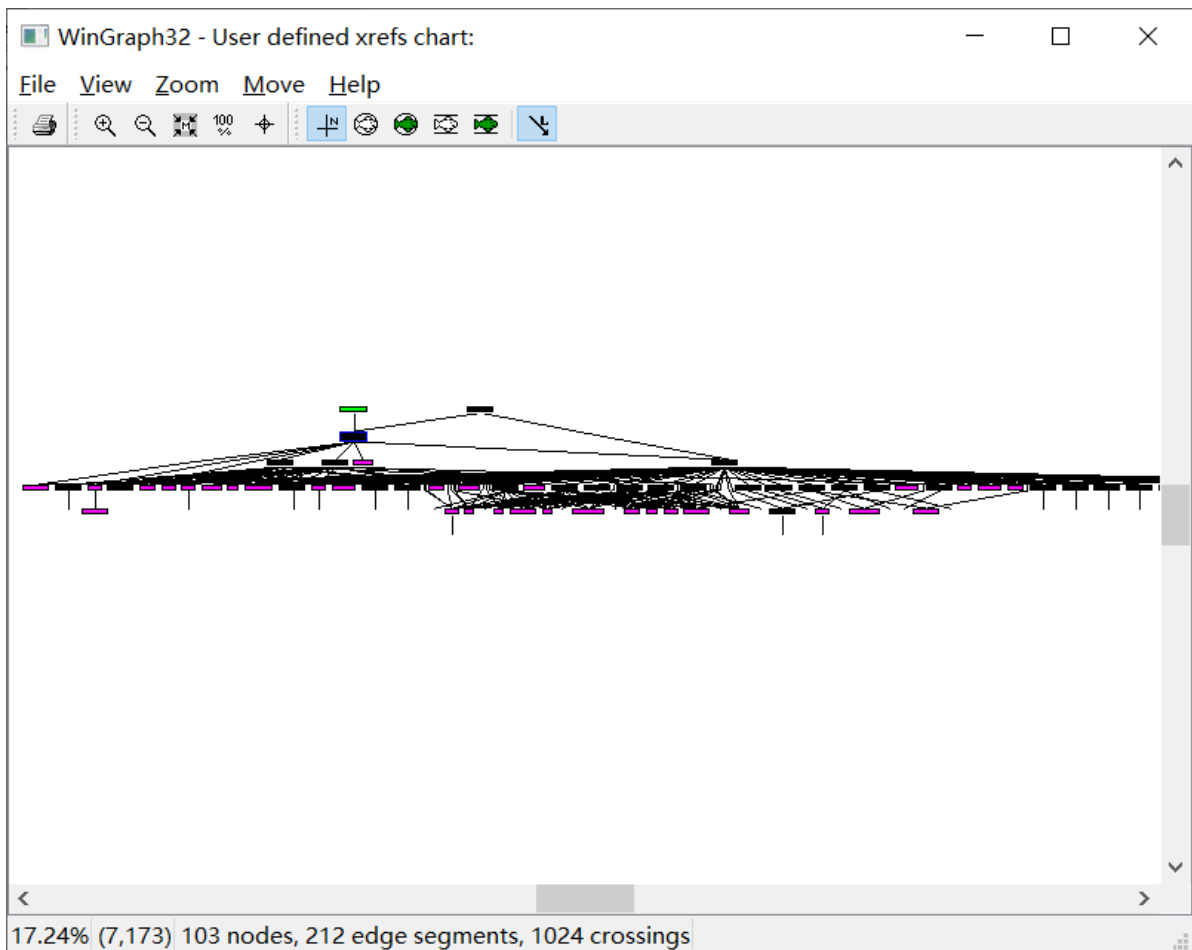


点击OK以后得到图像



在这个图像中可以看见此函数直接调用了4个函数，其中比较关键一些的就是 `GetSystemDefaultLangID` 和 `sub_100038EE` 调用的 `send` 函数，其中 `GetSystemDefaultLangID` 主要功能就是获取当前系统的语言

进一步，对 `DllMain` 重复刚刚的操作，将深度设置为2，得到CFG为：



可以看见这个图是非常的复杂的，由此也能知道DllMain调用了非常非常多的函数。根据箭头可以看到一些第一层调用的函数，有sub_10001365、_strnicmp、strncpy、strlen等函数；之后这些函数又进一步的调用了很多API，如__imp_strlen、memcpy、gethostbyname等，其中有一些是和网络有关的

使用快捷键G，定位到10001358这个位置，找到sleep

```

.text:10001350      imul     eax, 3E8h
.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
.text:10001360
.text:10001365
.text:10001365      ; ===== S U B R O U T I N E =====

```

在这一段我们发现，他在前面压入了一个eax，并且在之前有对eax进行了有符号的乘法，乘数是3E8h；暂时先不看eax中具体的值是多少，先进入到Sleep的函数体内

```

.data:100016210      ; sub_10001365+7D1p ...
.data:1001621C      ; void __stdcall Sleep(DWORD dwMilliseconds)
.data:1001621C      extrn Sleep:dwword      ; CODE XREF: sub_10001074+2E4↑p
.data:1001621C      ; sub_10001365+2E4↑p ...

```

发现这个Sleep就是只有和eax中保存的数据有关，那么现在我们再返回去看eax中到底是多少

```

mov     eax, off_10019020 ; "[This is CTI]30"
add     eax, 0Dh
push     eax                ; String
call     ds:atoi
imul     eax, 3E8h
pop      ecx
push     eax                ; dwMilliseconds
call     ds:Sleep

```

我们发现对eax的操作是从mov开始，之后增加0Dh，调用了函数之后再乘3E8h。

先查看off_10019020中存放的数据。

```

.data:10019020      off_10019020      dd offset aThisIsCti30 ; DATA XREF: sub_10001074:loc_10001341↑r
.data:10019020      ; sub_10001365:loc_10001632↑r ...
.data:10019020      ; "[This is CTI]30"

```

发现这里存放了一个字符串，字符串最后写了个30。同时要注意的是，这里放进来的其实是off_10019020这个地址，那么也就是说，这个地址在加上了0D之后，指针是指向了字符串中的“3”。

之后进入到atoi这个函数体内部查看一下

```

34 ; int __cdecl atoi(const char *String)
34      extrn atoi:dwword      ; CODE XREF: sub_10001074+2D6↑p
34      ; sub_10001365+2D6↑p ...

```

通过查阅发现，这个其实就是把字符串转换成数值的意思，也就是说从“30”转换成了30，那么此时eax中保存的数据就是30。

最后eax再乘上3E8h，同时要注意的是，上面的30应该是十进制的30，而3E8h是16进制，将其转换成10进制就是1000，再乘上30就是30000毫秒，也就是30秒。也就得到了后面的sleep会睡眠30秒。

首先定位到10001701位置

```

IDA VIEW-1      hex view-1      Structures      Enums
.text:10001701      call     ds:socket
.text:10001707      mov      edi, eax
.text:10001709      cmp      edi, 0FFFFFFFh
.text:1000170C      jnz      short loc_10001722
.text:1000170E      call     ds:WSAGetLastError
.text:10001714      push     eax

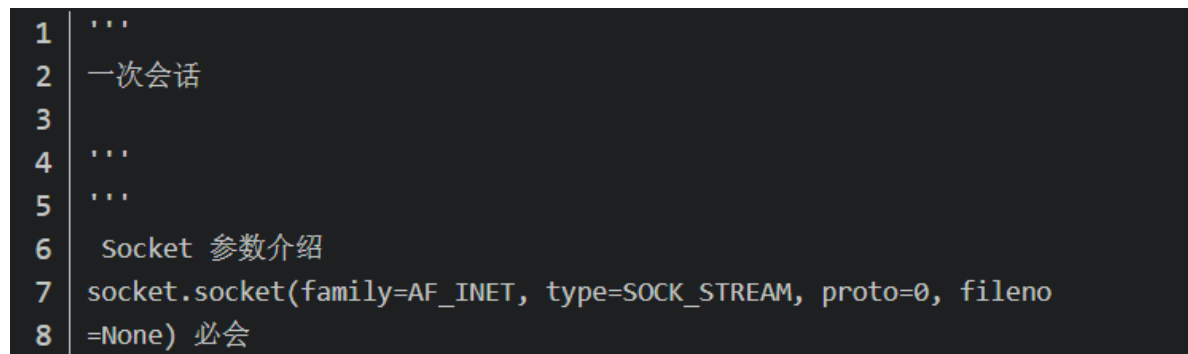
```

发现在上面有三个push

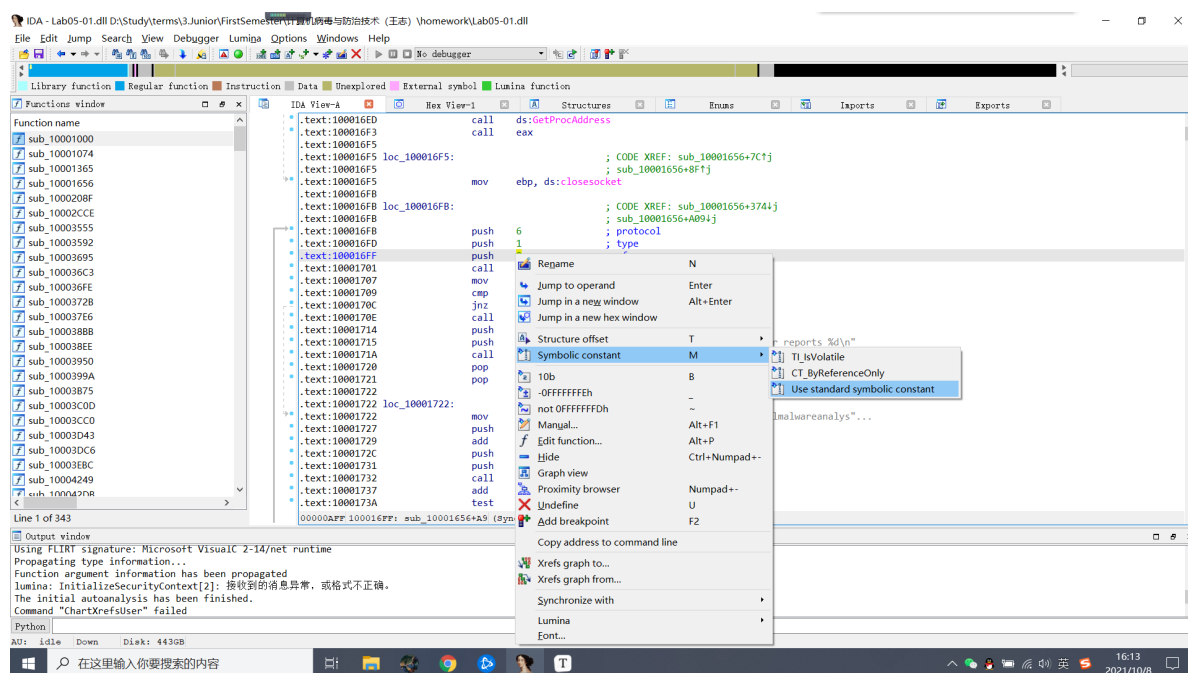
```
push    6           ; protocol
push    1           ; type
push    2           ; af
call    ds:socket
```

也就得到了socket的三个参数，分别是protocol=6,type=1,af=2

经过资料查阅，可以发现socket中的参数大概如下：



其中可以看见参数从左到右的顺序是family, type, proto, fileno, 那么其实根据这个上面的内容就可以进行重命名了。同时根据MSDN上可以找到，2代表按AF_INET。



在弹出的列表中找到AF_INET，选中，点击OK

Please choose a symbol		
Symbol name	Value	Type library
AE_AUTODIS	00000002	MS SDK (Windows 7)
AE_BUSY_ACD	00000002	MS SDK (Windows 7)
AE_LIM_EXPIRED	00000002	MS SDK (Windows 7)
AE_RINGING	00000002	MS SDK (Windows 7)
AE_SESSLOGOFF	00000002	MS SDK (Windows 7)
AE_SRVCONT	00000002	MS SDK (Windows 7)
AE_UAS_MODAL	00000002	MS SDK (Windows 7)
AE_UNSHARE	00000002	MS SDK (Windows 7)
AFSR_BACKNEW	00000002	MS SDK (Windows 7)
AF_INET	00000002	MS SDK (Windows 7)
AF_OP_COMM	00000002	MS SDK (Windows 7)
AGP_FLAG_NO_2X_RATE	00000002	MS SDK (Windows 7)
AIF_NOSKIP	00000002	MS SDK (Windows 7)
AIM_VERSION	00000002	MS SDK (Windows 7)
AI_CANONNAME	00000002	MS SDK (Windows 7)
AJ_IFC_SECURITY_OFF	00000002	MS SDK (Windows 7)
ALERT_SYSTEM_WARNING	00000002	MS SDK (Windows 7)
ALGO_MODE_CFB	00000002	MS SDK (Windows 7)
ALG_SID_DH_EPHEM	00000002	MS SDK (Windows 7)
ALG_SID_DSS_DMS	00000002	MS SDK (Windows 7)
ALG_SID_MD4	00000002	MS SDK (Windows 7)

Line 117 of 10117

OK Cancel Search Help

可以发现在反汇编窗口这里的参数已经变成了AF_INET


```

                                ; sub_1000165
push    6                      ; protocol
push    1                      ; type
push    AF_INET                ; af
call    ds:socket
mov     edi, eax

```

之后同理，照参数的顺序可以依次重命名为：AF_INET,SOCK_STREAM,IPPROTO_TCP。

在search中寻找他的opcode，也就是ED


Binary search
✕

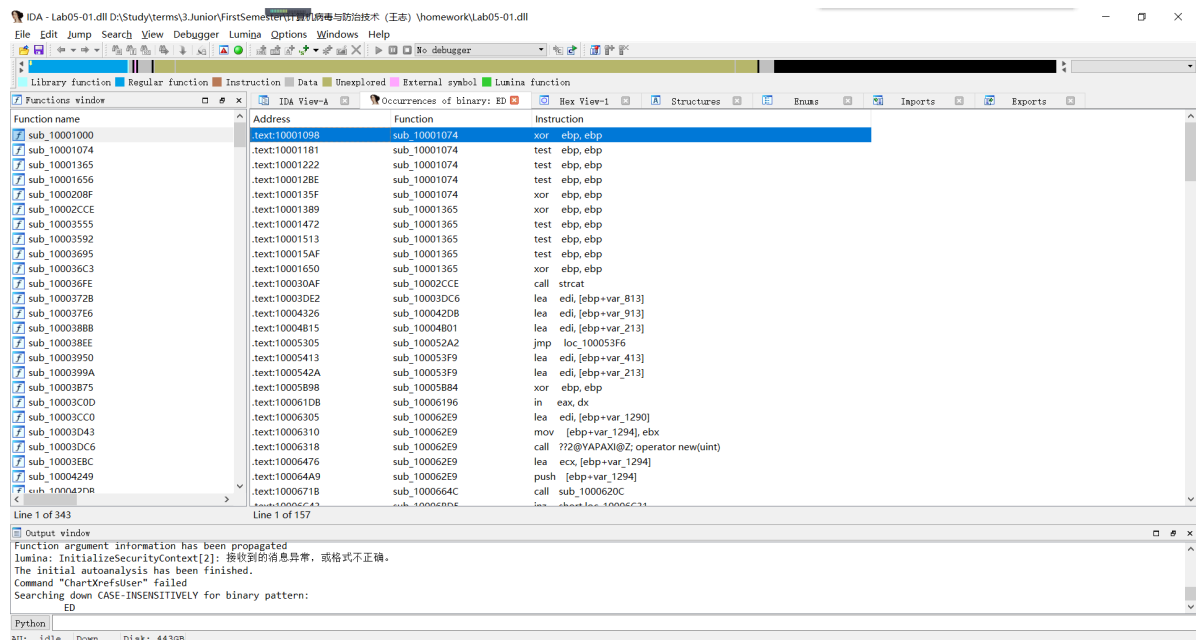
Enter binary search string:

String

☐ Match case
 ☐ Search Up
 ☒ Find all occurrences

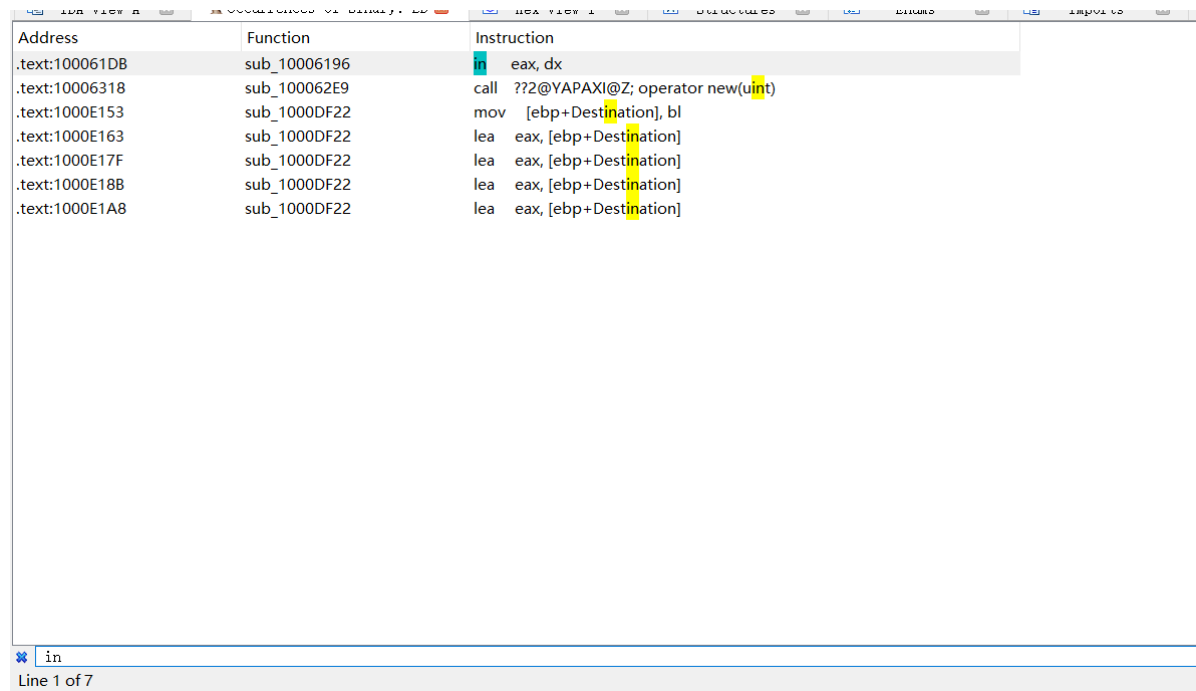
☒ Hex
 ☐ Decimal
 ☐ Octal

String encoding

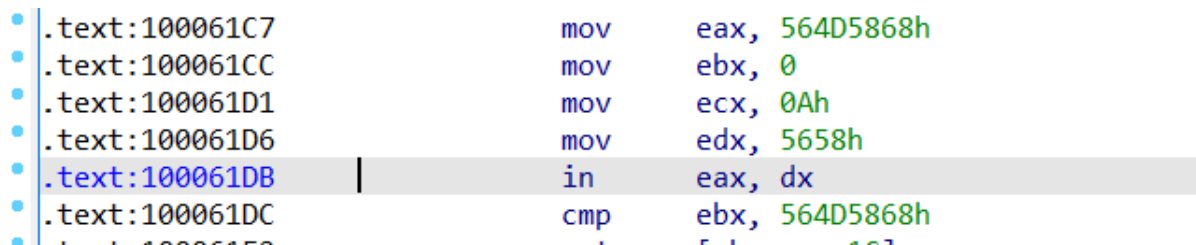


可以看到找到了非常多的这个操作

然后在这里根据字符串搜索，可以筛选出含有in的



双击查看第一条



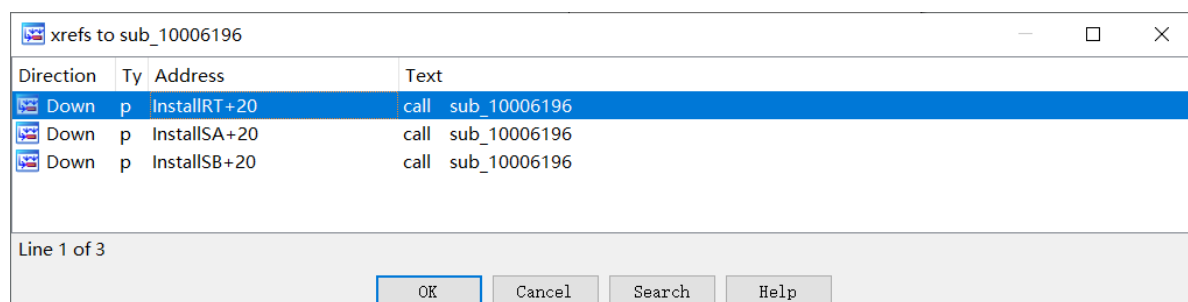
可以发现在100061DB这里出现了in，然后在他的上下有两个一样的564D5868h，猜测这个应该是一个ASCII组成的字符串，将其转换成字符串以后可以得到：

```
push    eax
mov     eax, 'VMXh'
mov     ebx, 0
mov     ecx, 0Ah
mov     edx, 5658h
in      eax, dx
cmp     ebx, 'VMXh'
setz    [ebp+var_10]
```

也就是题目中提到的VMXh魔术字符串

根据上下文环境，可以发现之类是把VMXh这个字符串给了eax，然后调用了in和cmp，猜测这里应该就是一处调用，去查看是或是VMware环境，想来这个应该就是专门用来检测环境的函数，后面其他的函数应该会调用这一段。

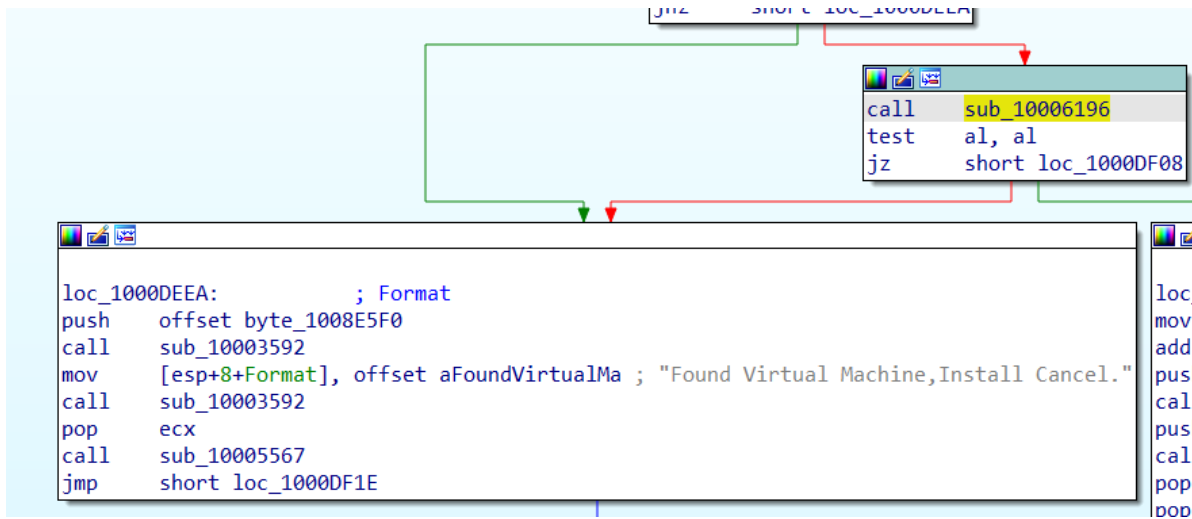
之后找到这个函数的头部位置，查看交叉引用



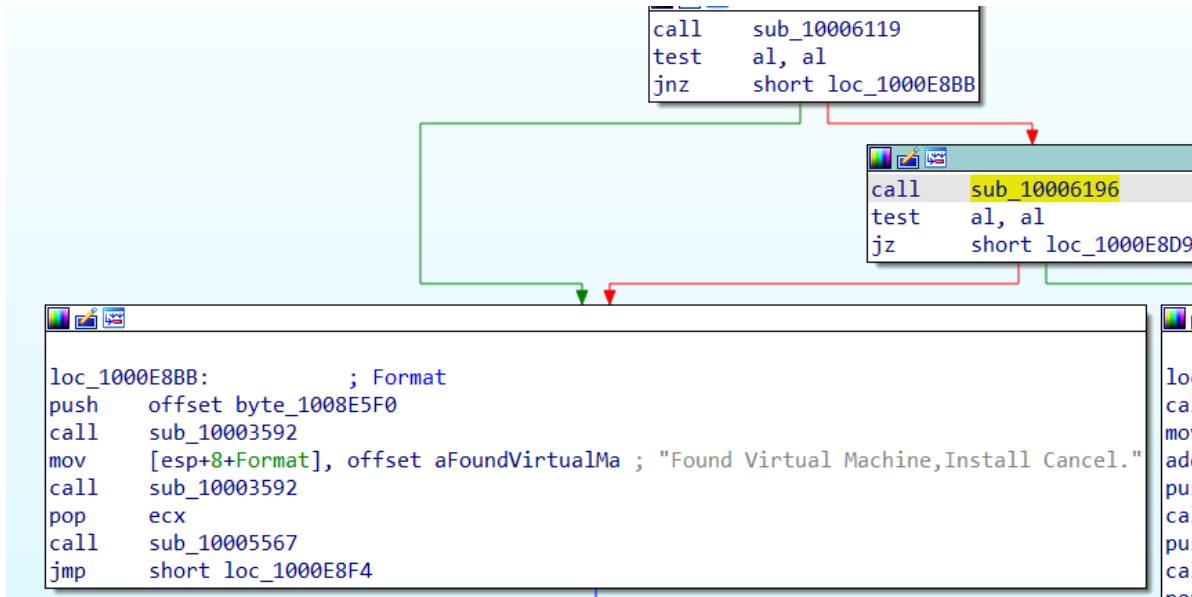
发现有三处交叉引用，依次进入查看函数主体



首先在第一处就看见了有相关的字符串显示："Found Virtual Machine,Install Cancel."



第二处可以发现依旧是如此



第三处也是。

那么综上就能看出来，sub_10006196就是对虚拟机环境进行检测的函数，并且多次被调用进行检测。

接下来跳转到0x1001D988这个地址

.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
.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	; <

0001B388 1001D988: .data:1001D988 (Synchronized with Hex View-1)

可以发现这个地址上有很多不明意义的字符，并且有重复出现，看上去像是随机生成的字符，如果拼在一起应该会形成乱码。

查看了一下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)

```

可以发现这个是按位进行了异或的运算，结合之前得到的乱码，想来这里应该是进行一个解密的操作，应该能将那些乱码解密成正常的字符串。在这里可以使用idc或者是idapython，但是没有购买idapython官方的插件，外加破解版的环境配置有些麻烦，所以就暂时先使用idc。

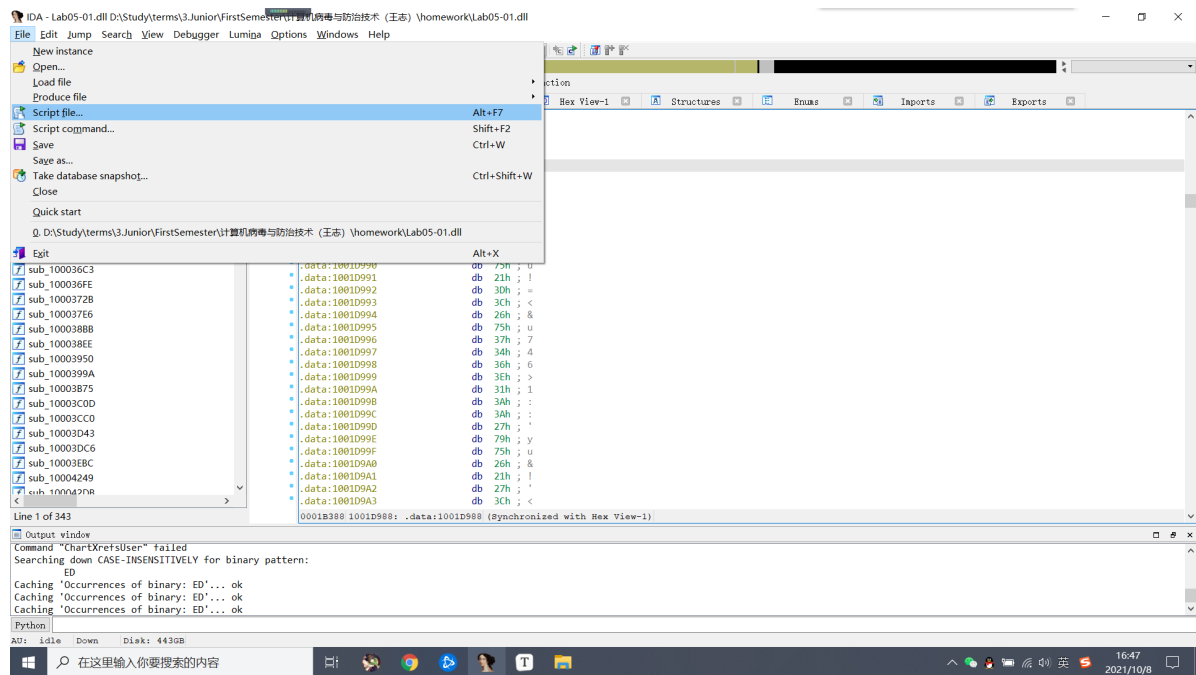
根据python脚本的内容改写一个idc脚本：

```

1  #include<idc.idc>
2
3  static main()
4  {
5      auto ea = ScreenEA(), b, i, decoded_byte;
6
7      for ( i=0x0; i<0x50; i++)
8      {
9          b = Byte(ea+i);
10         decoded_byte = b ^ 0x55;
11         PatchByte(ea+i, decoded_byte);
12     }
13 }

```

在ida中载入脚本



得到解密以后的内容如下

.data:1001D987	db	0	
.data:1001D988	db	78h	; x
.data:1001D989	db	64h	; d
.data:1001D98A	db	6Fh	; o
.data:1001D98B	db	6Fh	; o
.data:1001D98C	db	72h	; r
.data:1001D98D	db	20h	
.data:1001D98E	db	69h	; i
.data:1001D98F	db	73h	; s
.data:1001D990	db	20h	
.data:1001D991	db	74h	; t
.data:1001D992	db	68h	; h
.data:1001D993	db	69h	; i
.data:1001D994	db	73h	; s
.data:1001D995	db	20h	
.data:1001D996	db	62h	; b
.data:1001D997	db	61h	; a
.data:1001D998	db	63h	; c
.data:1001D999	db	6Bh	; k
.data:1001D99A	db	64h	; d
.data:1001D99B	db	6Fh	; o
.data:1001D99C	db	6Fh	; o
.data:1001D99D	db	72h	; r
.data:1001D99E	db	2Ch	; ,
.data:1001D99F	db	20h	
.data:1001D9A0	db	73h	; s
.data:1001D9A1	db	74h	; t
.data:1001D9A2	db	72h	; r
.data:1001D9A3	db	69h	; i
0001B388 1001D988: .data:1001D988 (Synchronized with Hex View-1			

大概的字符串内容就是xdoor is this backdoor,

按下A键以后就能看见一个可读的字符串了

```

.data:1001D986      db      0
.data:1001D987      db      0
.data:1001D988  aXdoorIsThisBac db  'xdoor is this backdoor, string decoded for Practical Malware Anal'
.data:1001D988      db  'ysis Lab :)1234',0
.data:1001D9D9      db      0
.data:1001D9DA      db      0
.data:1001D9DB      db      0

```

问题回答

Q1

从IDA的反汇编结果可以看出DllMain在.text节的0x1000D02E位置

Q2

gethostbyname在.idata节的0x100163CC处

Q3

gethostbyname总共被5个函数调用了9次

Q4

该样本会对pics.practicalmalwareanalysis.com这个域名发送请求

Q5

一共有23个局部变量

Q6

一共有1个参数

Q7

`\cmd.exe c/` 出现在xdoors_d:10095B34

Q8

该样本像是开启一个远程的shell会话，并且会对输入的字符串等进行验证，如果匹配失败则关闭会话

Q9

dword_1008E5C4中应该是存放了操作系统平台的信息

Q10

当robotwork匹配成功以后，该程序会对注册表项：`SOFTWARE\Microsoft\Windows\CurrentVersion` 进行访问，获取其中的WorkTime和WorkTimes，并向远端发送信息。

Q11

获取进程列表并通过网络发送，或者是寻找列表中某个特定的进程信息

Q12

从图像中可以看见此函数可能会调用 `GetSystemDefaultLangID` `send` `sprintf` 等函数，其中比较关键一些的就是 `GetSystemDefaultLangID` 和 `sub_100038EE` 调用的 `send` 函数。其中

`GetSystemDefaultLangID` 这个函数的主要功能就是获取当前系统的语言类型，然后有一个send会将前面获取的信息发送出去，所以根据功能可以重命名为 `GetSystemLanguageAndSend`

Q13

根据得到的cfg中的箭头可以看到一些第一层调用的函数，有`sub_10001365`、`_strnicmp`、`strncpy`、`strlen`等函数；之后这些函数又进一步的调用了很多API，如 `__imp_strlen`、`memcpy`、`gethostbyname`等，其中有一些是和网络有关的

Q14

会休眠30s

Q15

可以看到三个参数是protocol=6,type=1,af=2

Q16

可以依次重命名为：AF_INET,SOCK_STREAM,IPPROTO_TCP

Q17

在0x1000D867, 0x1000DEE1, 0x1000E8B2都进行了调用，用来检测是否是虚拟机环境。同时可以看见，如果检测出来是虚拟机环境，还会提示一个字符串："Found Virtual Machine,Install Cancel."

Q18

可以发现这个地址上有很多不明意义的字符，并且有重复出现，看上去像是随机生成的字符。

Q19

运行了脚本以后可以看见之前的乱码都变成了英文字符和标点，如果竖着看可以得到一个字符串，大概内容为xdoor is this backdoor,

Q20

在页面按下A键以后可以得到：xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234

Q21

通过对代码的分析，可以得到这个脚本的工作原理是：获取当前光标的位置，读取0x50长度的字符，然后按位进行了异或运算