恶意代码分析与防治技术实验报告

Lab5

学号：2011937　　姓名：姜志凯　　专业：信息安全

一、 实验环境

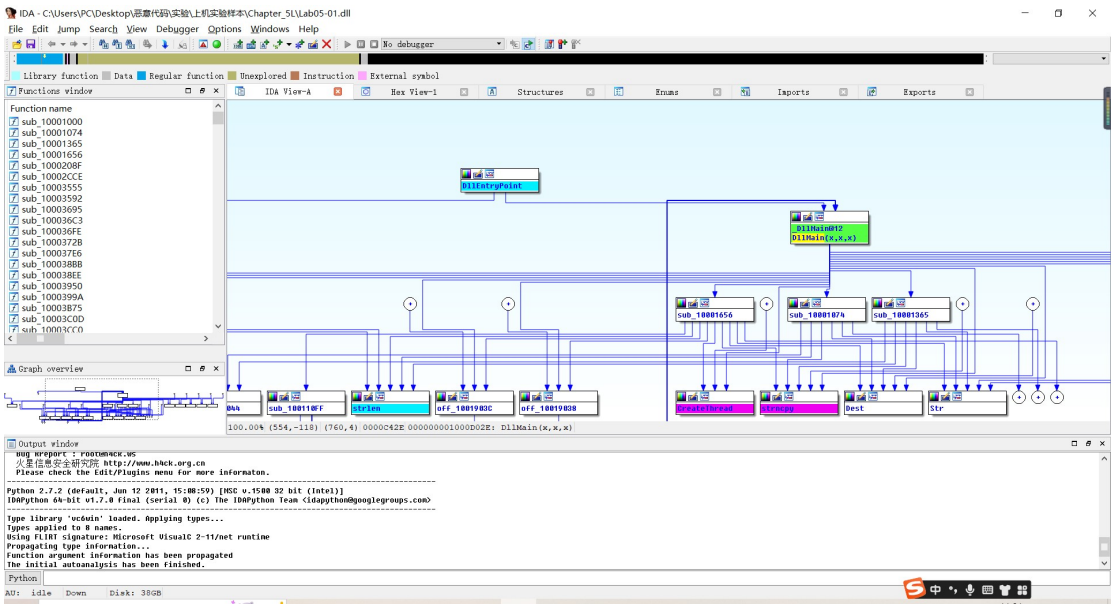
- Windows10

二、 实验工具

- IDA Pro

三、 实验内容

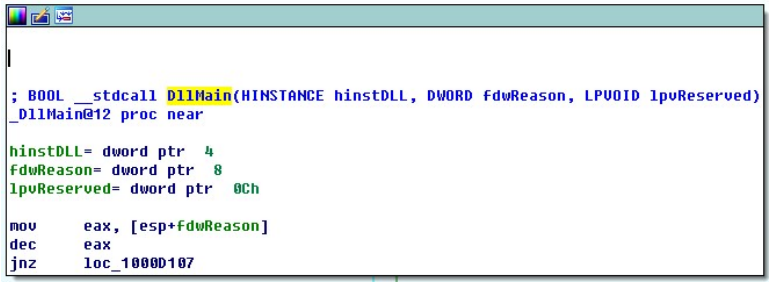
只用 IDA Pro 分析在文件 Lab05-01.dll 中发现的恶意代码。
首先用 IDA Pro 打开 Lab05-01.dll 文件：



然后分析以下问题：

1. DllMain 的地址是什么？

空格键查看 CFG，第一块即 DLLMain 函数



```
; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
_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
```

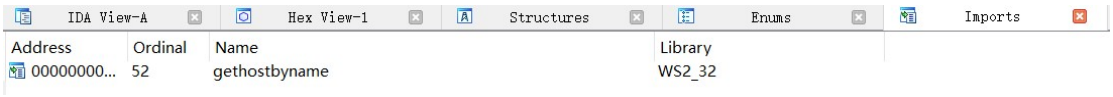
右键查看 text

```
.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
.text:1000D02E
.text:1000D02E                 mov     eax, [esp+fdwReason]
.text:1000D032                 dec     eax
.text:1000D033                 jnz     loc_1000D107
```

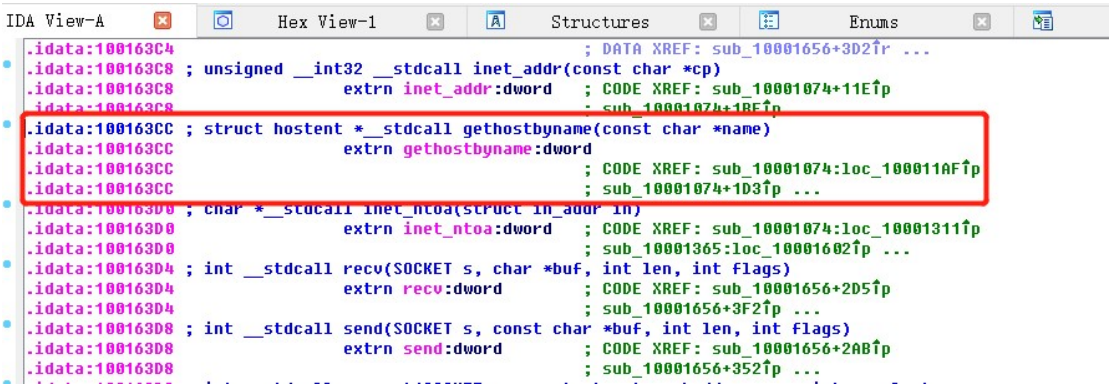
得知，DLLMain 地址为.text 节的 1000D02E。

2. 使用 Imports 窗口并浏览到 gethostbyname，导入函数定位到什么地址?

查看 imports 窗口，找到 gethostbyname 函数



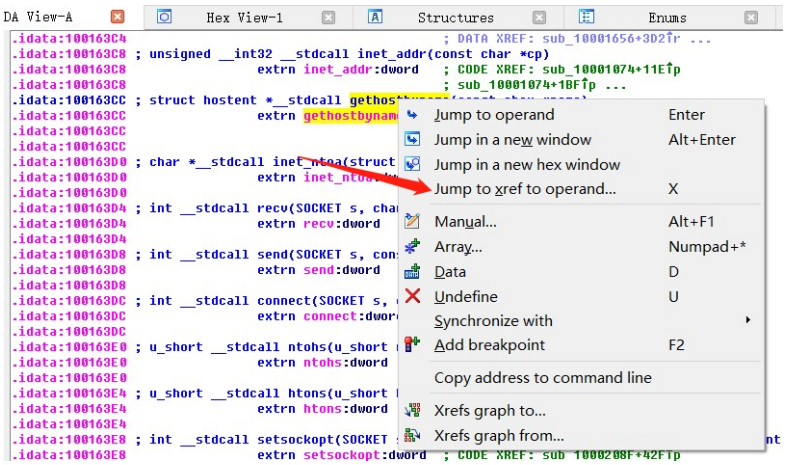
双击进入查看详细信息

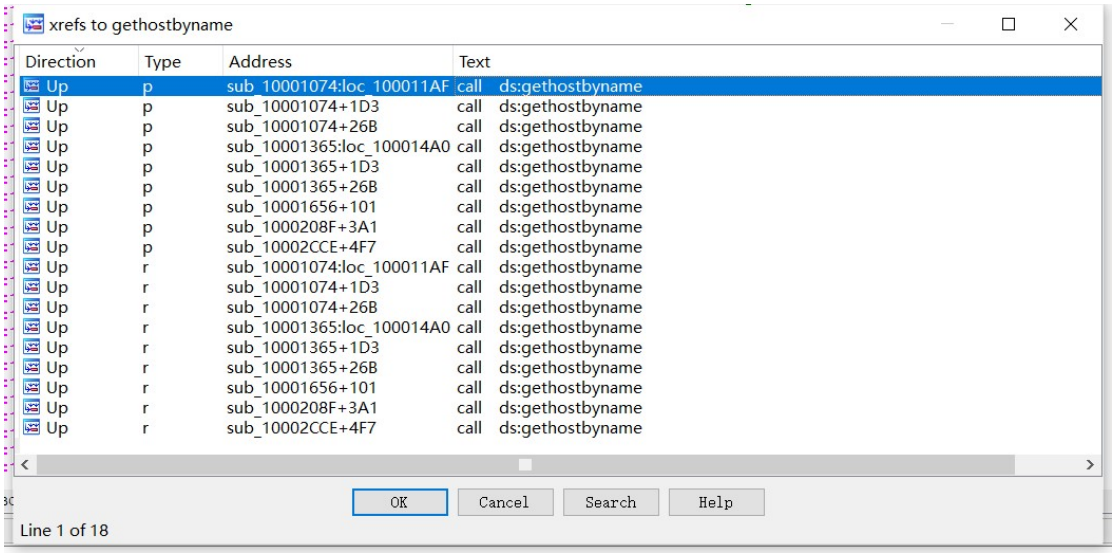


得知，gethostbyname 函数地址为.data 节的 100163CC。

3. 有多少函数调用了 gethostbyname?

查看交叉引用 Ctrl+X



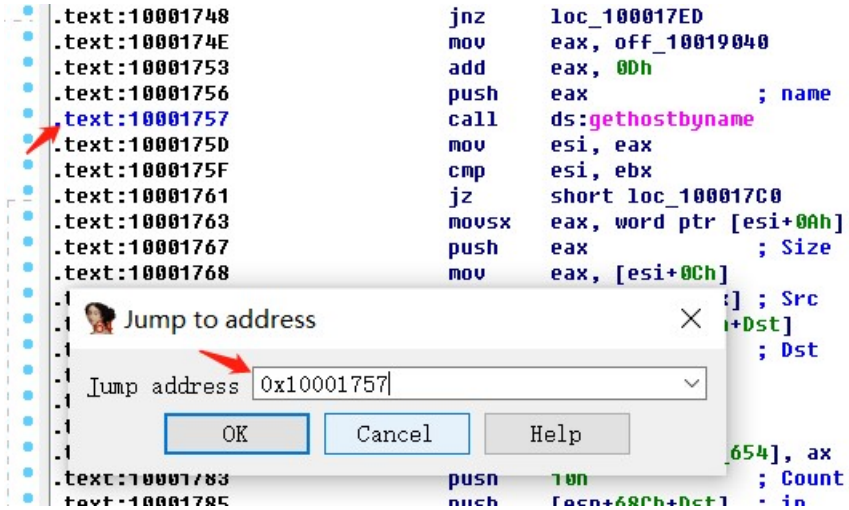


这里的 type 值，r 是被”读取”的引用，CPU 必须先读取这个导入项，再调用它，类型 p 是被调用的引用。

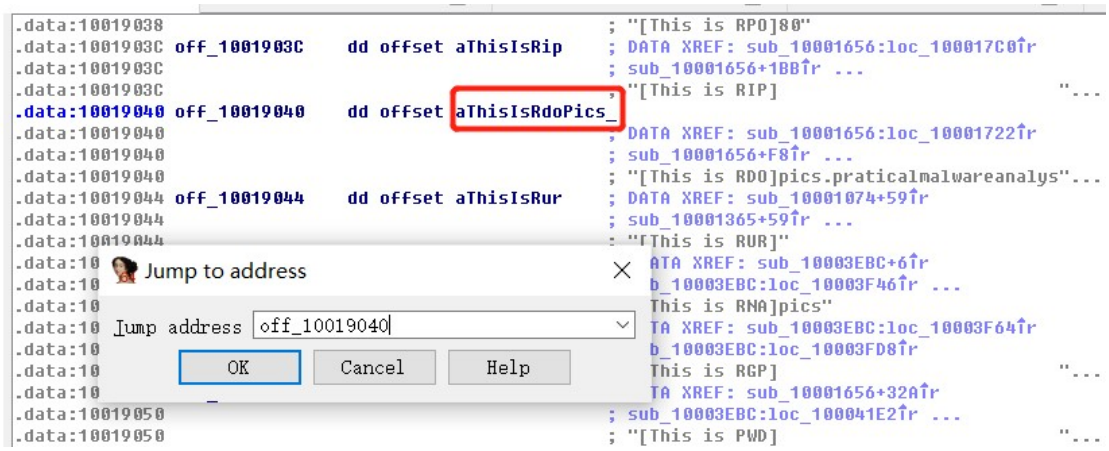
这里我们可以看见共有 5 个函数引用了 gethostname，而且可以看出共引用了 9 次。

4. 将精力集中在位于 0x10001757 处的对 gethostbyname 的调用，你能找出哪个 DNS 请求将被触发吗?

按 g 跳转，输入 0x10001757 找到对应调用



call 函数默认将栈顶第一个元素作为参数传递给调用函数，向上找 push，发现了 push eax，再找 eax 的值，发现 off_10019040，查看 off_10019040



发现 off_10019040 指向名为 aThisRdoPics 的字符串，查看 aThisRdoPics

发现字符串值为[This is RDO]pics.praticalmalwareanalysis.com，off_10019040 指向字符串的首字符，然后赋值给 eax，eax 又加了 0Dh，即 13，正好指向了网站的首字母“p”。

所以，最后 push 进栈的值为 pics.praticalmalwareanalysis.com，即为 gethostbyname 函数的参数，触发 pics.praticalmalwareanalysis.com DNS 请求。

5. IDA Pro 识别了在 0x10001656 处的子过程中的多少个局部变量?

g 跳转查看

```
.text:10001656
.text:10001656 ; ============== S U B R O U T I N E =======================================
.text:10001656
.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
.text:10001656                 sub     esp, 678h
```

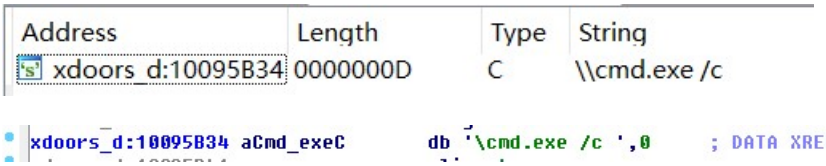
如图可知，共识别了 24 个局部变量。

6. IDA Pro 识别了在 0x10001656 处的子过程中的多少个参数?

由上图可知，共识别了一个参数，为 LPVOID 类型的 lpThreadParameter

7. 使用 Strings 窗口，在反汇编中定位字符串\cmd.exe /c。它位于哪?

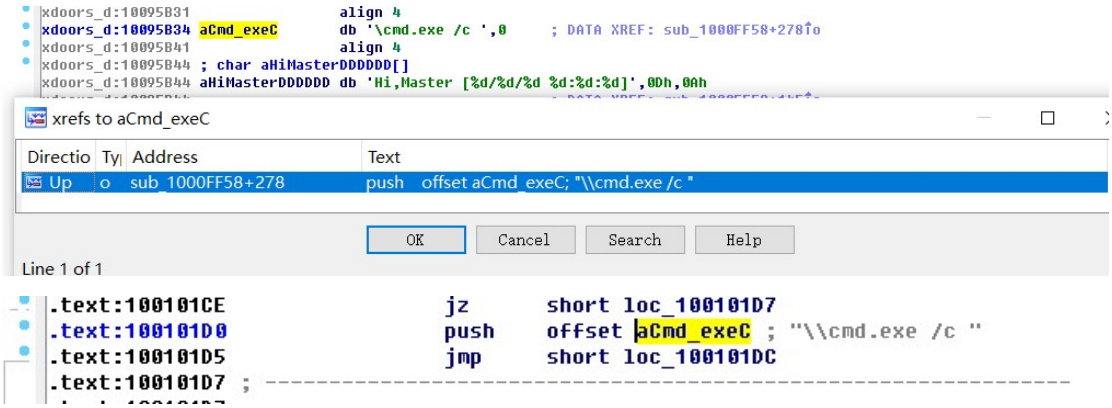
Shift+F12 查看 Strings 窗口，找到\cmd.exe /c，双击



位于 xdoors_d:10095B34。

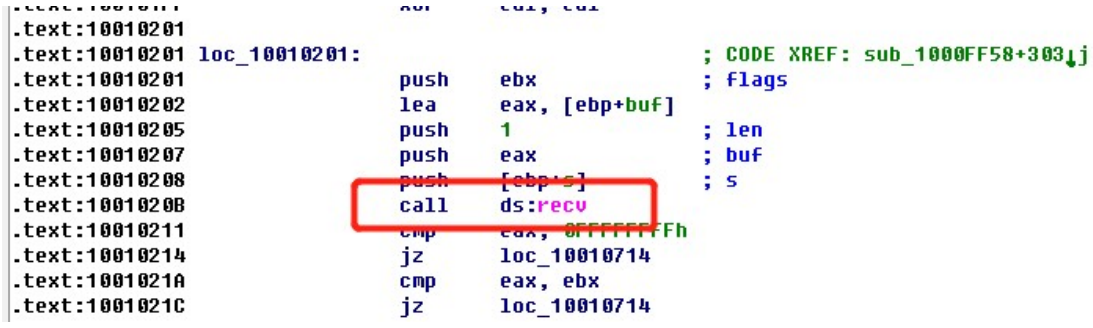
8. 在引用\cmd.exe /c 的代码所在区域发生了什么?

Ctrl+X 查看交叉引用：



可以发现，该字符串被压入栈中，可能作为函数参数。

继续看跳转的位置，发现了 recv 和 send 的调用，基于计算机网络的学习，可能与远程通信有关。



回到\cmd.exe /c 代码区域，在它下方发现了 Encrypt Magic Number For This Remote Shell

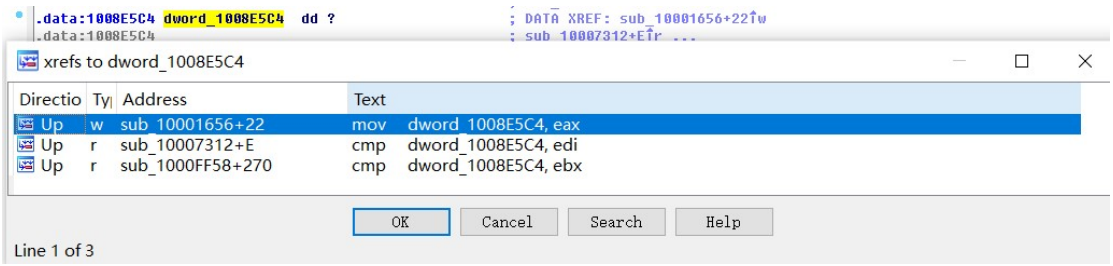
Session [0x%02x]、UpTime 等字符串信息。

```
xdoors_d:10095B31                       align 4
xdoors_d:10095B34 aCmd_exeC             db '\cmd.exe /c ',0     ; DATA XREF: sub_1000FF58+278↑o
xdoors_d:10095B41                       align 4
xdoors_d:10095B44 ; char aHiMasterDDDDDD[]
xdoors_d:10095B44 aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%d:%d]',0Dh,0Ah
xdoors_d:10095B44                                       ; DATA XREF: sub_1000FF58+145↑o
xdoors_d:10095B44                       db 'WelCome Back...Are You Enjoying Today?',0Dh,0Ah
xdoors_d:10095B44                       db 0Dh,0Ah
xdoors_d:10095B44                       db 'Machine UpTime  [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Secon'
xdoors_d:10095B44                       db 'ds]',0Dh,0Ah
xdoors_d:10095B44                       db 'Machine IdleTime [%-.2d Days %-.2d Hours %-.2d Minutes %-.2d Seco'
xdoors_d:10095B44                       db 'nds]',0Dh,0Ah
xdoors_d:10095B44                       db 0Dh,0Ah
xdoors_d:10095B44                       db 'Encrypt Magic Number For This Remote Shell Session [0x%02x]',0Dh,0Ah
xdoors_d:10095B44                       db 0Dh,0Ah,0
xdoors_d:10095C5C ; char asc_10095C5C[]
xdoors_d:10095C5C asc_10095C5C          db '>',0                ; DATA XREF: sub_1000FF58+4B↑o
xdoors_d:10095C5C                                               ; sub_1000FF58+3E1↑o
xdoors_d:10095C5E                       align 400h
xdoors_d:10095C5E xdoors_d              ends
xdoors_d:10095C5E
xdoors_d:10095C5E
xdoors_d:10095C5E                       end DllEntryPoint
```

所以这可能是一个远程 shell 会话函数，开启远程 shell 会话。

9. 在同样的区域，在 0x100101C8 处，看起来好像 dword_1008E5C4 是一个全局变量，它帮助决定走哪条路径。那恶意代码是如何设置 dword_1008E5C4 的呢?（提示：使用 dword_1008E5C4 的交叉引用）

找到 dword_1008E5C4 的位置，查看引用



发现两个比较，一个修改，双击查看修改处的操作

```
.text:1000166F                  mov     [esp+688h+hModule], ebx
.text:10001673                  call    sub_10003695
.text:10001678                  mov     dword_1008E5C4, eax
.text:1000167D                  call    sub_100036C3
```

将 eax 的值传给 dword_1008E5C4，而 eax 值为上一个函数 sub_10003695 的返回值，双击查看函数

```
.text:10003695 ; =============== S U B R O U T I N E =======================================
.text:10003695
.text:10003695 ; Attributes: bp-based frame
.text:10003695
.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
.text:100036C2                 retn
.text:100036C2 sub_10003695    endp
```

下面对函数进行简单分析：

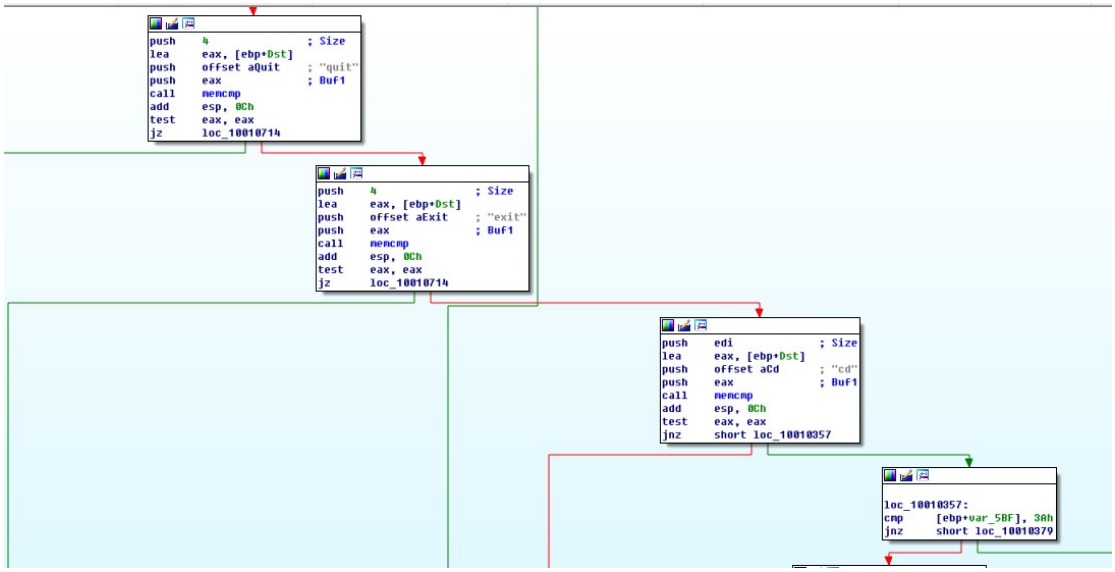
```
sub_10003695      proc near                ; CODE XREF:
VersionInformation= _OSVERSIONINFOA ptr -94h
                  push     ebp
                  mov       ebp, esp
                  sub       esp, 94h          ; 将 esp 增加 94h 也就是 148d，37 个字节
                  lea       eax, [ebp+VersionInformation] ; 将 ebp+VersionInformation 的地址
赋值给 eax
                  mov       [ebp+VersionInformation.dwOSVersionInfoSize], 94h ; 将这个地
址的值赋值成 94h
                  push      eax                ; lpVersionInformation
                  call      ds:GetVersionExA ; 在一个 OSVERSIONINFO 结构中载入与平台
和操作系统有关的版本信息
                                               ; 上面有定义这个 OSVERSIONINFO 结构
                  xor       eax, eax          ; 将 eax 置 0（因为 GetVersionExA 的返回值在
eax 中）
                  cmp       [ebp+VersionInformation.dwPlatformId], 2 ; 这里将 dwPlatformId
和 2 进行比较，2 代表 VER_PLATFORM_WIN32_NT，说明系统是 win8，即如果系统是 win8
的话就相等，相等即置位，al 为 1，即 eax 为 1
                  setz      al                ; Set Byte if Zero (ZF=1) ;
                  leave                       ; High Level Procedure Exit
                  retn                        ; Return Near from Procedure
sub_10003695      end
```

所以 eax 被置为 1，函数的返回值为 1，dword_1008E5C4 被设置为 1，所以这个全局变量一直是 1。

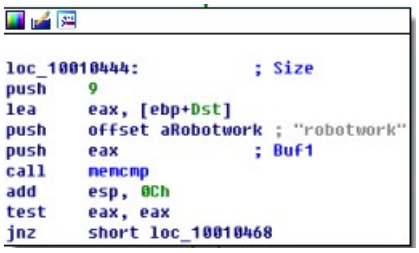
10. 在位于 0x1000FF58 处的子过程中的几百行指令中，有一系列使用 memcmp 来比较字符串的指令。如果对 rotbotwork 的字符串比较是成功的

（memcmp 返回 0），会发生什么？

g 跳转查看，空格查看 CFG



从这往下，可以看到一系列使用 memcmp 来比较字符串的指令，再往下找，找到 rotbotwork 的字符串比较：



分析：

首先将 robotwork 字符串指针入栈，紧接着压入 eax，调用 memcmp，如果两个数相同，返回 0；

然后 add esp, 0Ch，0Ch 是 12，即 12 个字节，因为 push 后面跟的是立即数，一个数占 4 字节，所以，一开始的 push 9，和后面的两次 push，加起来一共是 3 次，正好 12 个字节，所以这里回收了 12 字节的空间；

test eax, eax，如果 eax 为 0，则 ZF 置为 1，jnz 不跳转；

所以当字符串比较成功（当 memcmp 返回 0）时，JNZ 不跳转，程序继续按从上到下的顺序执行：

```
.text:10010444 ; ---------------------------------------------------------------------------
.text:10010444
.text:10010444 loc_10010444:                           ; CODE XREF: sub_1000FF58+4E0↑j
.text:10010444                 push    9               ; Size
.text:10010446                 lea     eax, [ebp+Dst]
.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 ; ---------------------------------------------------------------------------
```

push [ebp+s]；将 ebp(esp 是栈顶指针，ebp 是栈基址)地址增加 s
；(栈中，esp 地址减小，栈空间增大，ebp 增加，ebp 将向栈底偏移)
；将 ebp 向下 s 的指针地址压栈

发现调用了 sub_100052A2，查看

```
.text:100052A2                 push    ebp
.text:100052A3                 mov     ebp, esp
.text:100052A5                 sub     esp, 60Ch
.text:100052AB                 and     [ebp+Dest], 0
.text:100052B2                 push    edi
.text:100052B3                 mov     ecx, 0FFh
.text:100052B8                 xor     eax, eax
.text:100052BA                 lea     edi, [ebp+var_60B]
.text:100052C0                 and     [ebp+Data], 0
.text:100052C7                 rep stosd
.text:100052C9                 stosw
.text:100052CB                 stosb
.text:100052CC                 push    7Fh
.text:100052CE                 xor     eax, eax
.text:100052D0                 pop     ecx
.text:100052D1                 lea     edi, [ebp+var_20B]
.text:100052D7                 rep stosd
.text:100052D9                 stosw
.text:100052DB                 stosb
.text:100052DC                 lea     eax, [ebp+phkResult]
.text:100052DF                 push    eax             ; phkResult
.text:100052E0                 push    0F003Fh         ; samDesired
.text:100052E5                 push    0               ; ulOptions
.text:100052E7                 push    offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVe"...
.text:100052EC                 push    80000002h       ; hKey
.text:100052F1                 call    ds:RegOpenKeyExA
.text:100052F7                 test    eax, eax
.text:100052F9                 jz      short loc_10005309
.text:100052FB                 push    [ebp+phkResult] ; hKey
.text:100052FE                 call    ds:RegCloseKey
.text:10005304                 jmp     loc_100053F6
.text:10005309 ; ---------------------------------------------------------------------------
```

可以发现，这是一个 socket 函数，进行网络通信使用的。

函数还获取了 SOFTWARE\Microsoft\Windows\CurrentVersion，再往下看，发现

```
.text:1000531C                 lea     eax, [ebp+Type]
.text:1000531F                 push    eax             ; lpType
.text:10005320                 push    0               ; lpReserved
.text:10005322                 push    offset aWorktime ; "WorkTime"
.text:10005327                 push    [ebp+phkResult] ; hKey
.text:1000532A                 call    ebx ; RegQueryValueExA
.text:1000532C                 mov     esi, ds:sprintf
```

```
.text:1000539D                 push    eax             ; lpType
.text:1000539E                 push    0               ; lpReserved
.text:100053A0                 push    offset aWorktimes ; "WorkTimes"
.text:100053A5                 push    [ebp+phkResult] ; hKey
.text:100053A8                 call    ebx ; RegQueryValueExA
.text:100053AA                 test    eax, eax
.text:100053AC                 jnz     short loc_100053EB
```

经过分析，得知程序获取了注册表的一些信息，即

“SOFTWARE\Microsoft\Windows\CurrentVersion”的 WorkTime 键值和 WorkTimes 键值。

这些信息通过（push [ebp+s]）传给 socket，且向 socket 发送两个注册表键值，分别为

HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime

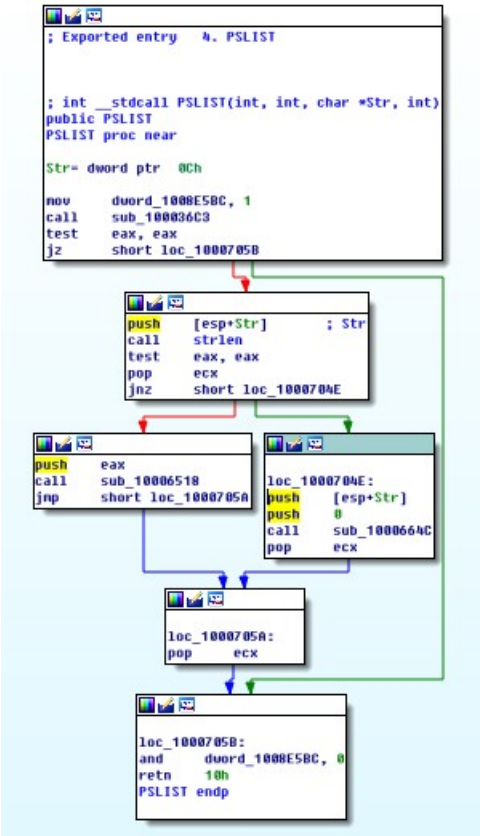
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTimes

11. PSLIST 导出函数做了什么？

打开导出函数表

Name	Address	Ordinal
InstallRT	000000001000D847	1
InstallSA	000000001000DEC1	2
InstallSB	000000001000E892	3
PSLIST	0000000010007025	4
ServiceMain	000000001000CF30	5
StartEXS	0000000010007ECB	6
UninstallRT	000000001000F405	7
UninstallSA	000000001000EA05	8
UninstallSB	000000001000F138	9
DllEntryPoint	000000001001516D	

双击 PSLIST 进入，并查看 CFG



发现该函数有两条路径，由函数 sub_100036C3 决定，查看它的 CFG

call ds:GetVersionExA ;　//调用函数查看系统版本

cmp [ebp+VersionInformation.dwPlatformId], 2 ;　//如果不相等则跳转，直接跳转结束；如果相等，则继续执行；

cmp [ebp+VersionInformation.dwMajorVersion], 5; 5 代表特殊版本的 windows

jb short loc_100036FA ; 无符号比较，如果[ebp+VersionInformation.dwMajorVersion]小于 5 跳转；否则跳转结束；

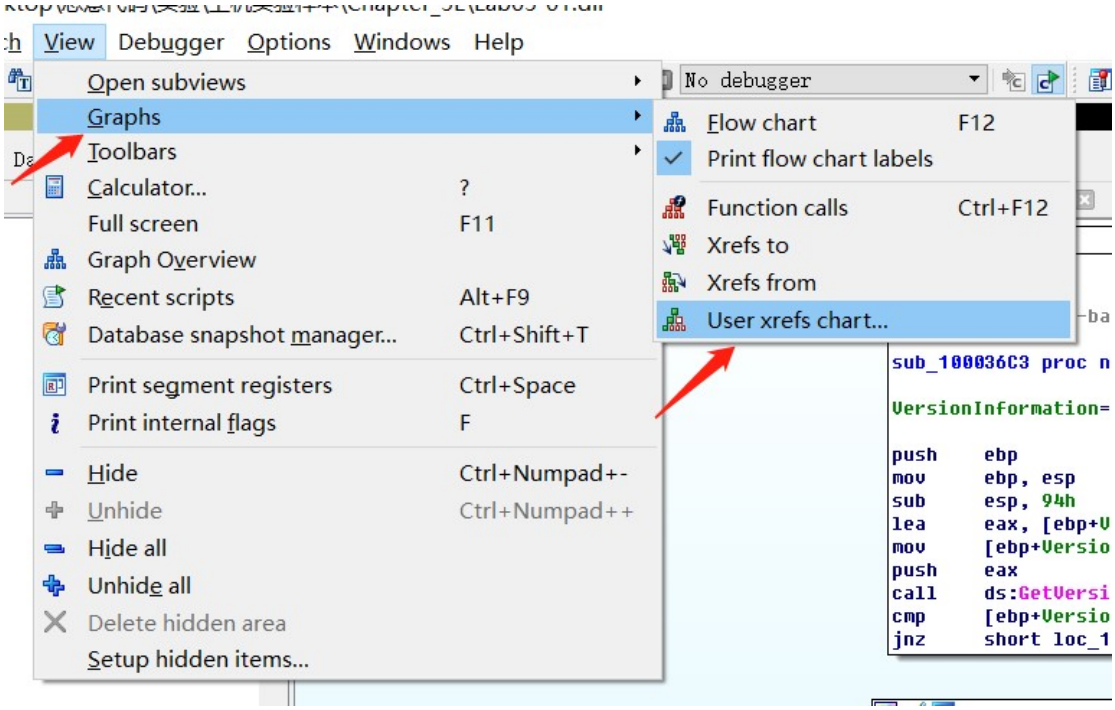
push 1

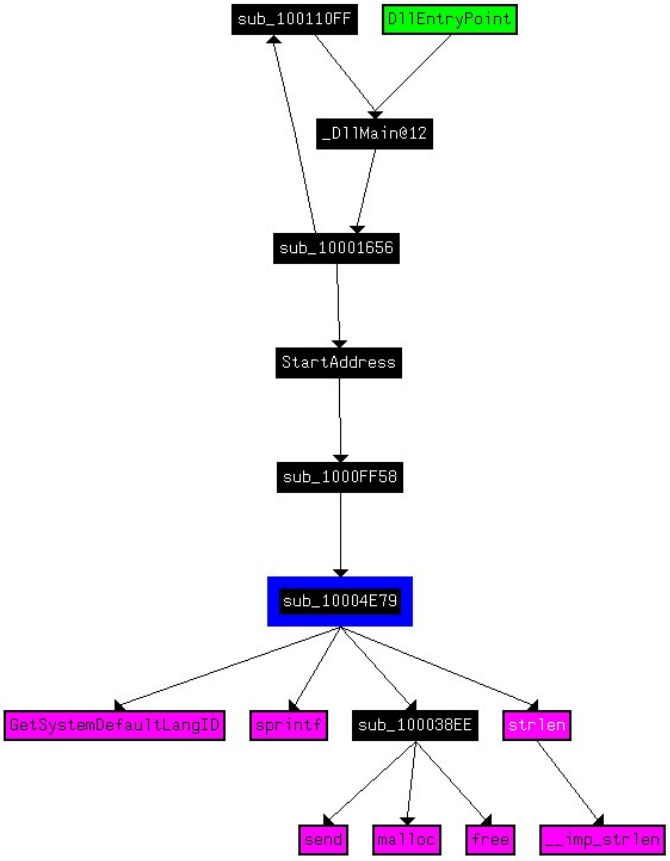
pop eax leave ;　//程序返回 1

所以这个函数的作用就是判断目标主机的系统版本，如果是版本过低，就直接跳转结束；符合要求就返回 1。

12.　使用图模式来绘制出对 sub_10004E79 的交叉引用图。当进入这个函数时，哪个 API 函数可能被调用？仅仅基于这些 API 函数，你会如何重命名这个函数？

按如下方式打开交叉引用图





主要调用了：

- GetSystemDefaultLangID：获取系统的默认语言的函数
- Sprintf：打印
- sub_100038EE：主要调用：send：socket 发送消息函数；malloc：分配堆栈的函数；

free 释放堆栈的函数

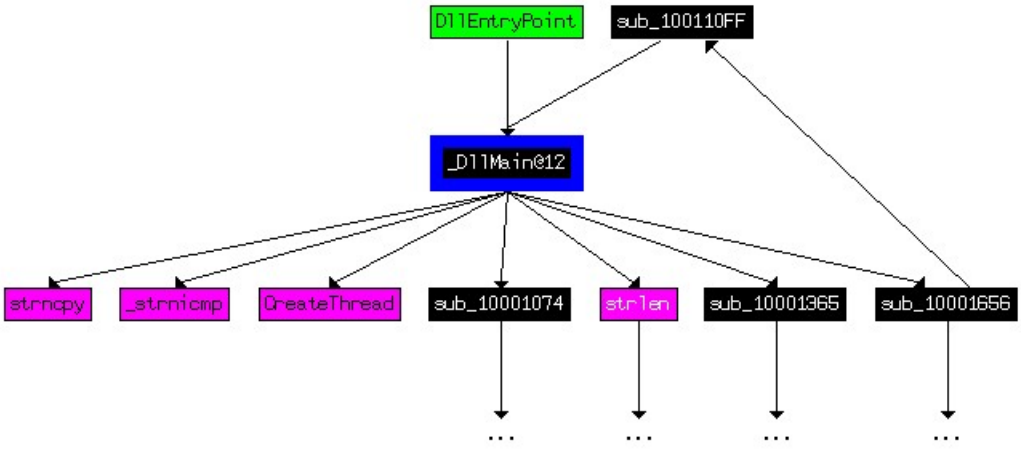
- Strlen：调用__imp_strlen，获取字符串长度

推测其功能是发送系统默认语言，可以重命名为 send_languageID。

13. DllMain 直接调用了多少个 Windows API？多少个在深度为 2 时被调用？

打开 DLLMain 的交叉引用图

极其多，设置深度为 1



DllMain 在深度为 1 直接调用的 API 有 strncpy、_strnicmp、CreateThread、strlen

设置深度为 2

非常多，数不过来。

14. 在 0x10001358 处，有一个对 Sleep（一个使用包含要睡眠的毫秒数的参数的 API 函数）的调用。顺着代码向后看，如果这段代码执行，这个程序会睡眠多久？

g 跳转查看

```
.text:10001341 loc_10001341:                           ; CODE XREF: sub_10001074+10F↑j
.text:10001341                                         ; sub_10001074+1B0↑j ...
.text:10001341                 mov     eax, off_10019020
.text:10001346                 add     eax, 0Dh
.text:10001349                 push    eax             ; Str
.text:1000134A                 call    ds:atoi
.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
```

Sleep 函数的参数为上一步的 eax，而 eax 的值为 eax*3E8h，eax 的值是 atoi 函数的返回值，atoi 函数的参数为它上边的 eax，而这个 eax 指向 off_10019020+0Dh 的位置，所以查看 off_10019020，再反推回去即可。

```
.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"
.data:10019024 off_10019024    dd offset aThisIsNti30  ; DATA XREF: sub_10001656+357↑r
.data:10019024                                         ; sub_10004EE0+227↑r
.data:10019024                                         ; "[This is NTI]30"
.data:10019028 off_10019028    dd offset aThisIsLog0   ; DATA XREF: sub_10003592+9↑r
.data:10019028                                         ; sub_10004EE0+292↑r
```

最后 eax 指向 3，入栈，最后 eax 等于 30，然后传给 atoi 函数，乘上 3E8h，等于 30000，**sleep 的单位为 ms**，所以会休眠 30s。

15. 在 0x10001701 处是一个对 socket 的调用。它的 3 个参数是什么？

g 跳转查看

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

3 个参数为 6、1、2。

16. 使用 MSDN 页面的 socket 和 IDA Pro 中的命名符号常量，你能使参数更加有意义吗？在你应用了修改以后，参数是什么？

由上图可知，6、1、2 分别对应 protocol 协议类型、type 套接字类型、af 地址类型。

到 socket 官方文档中找 6、1、2 分别对应的实际参数

```
#defineIPPROTO_TCP6/*tcp*/
#defineSOCK_STREAM1/*streamsocket*/
#defineAF_INET2/*internetwork:UDP,TCP,etc.*/
```

所以参数为：

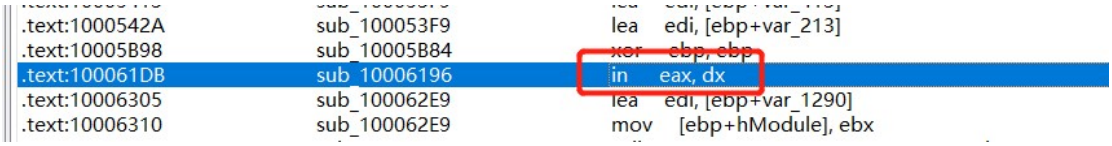
```

#defineAF_INET2/*internetwork:UDP,TCP,etc.*/

#defineSOCK_STREAM1/*streamsocket*/

#defineIPPROTO_TCP6/*tcp*/

```

可以对其重命名是的参数更有意义：

IPPROTO_TCP、SOCK_STREAM、AF_INET

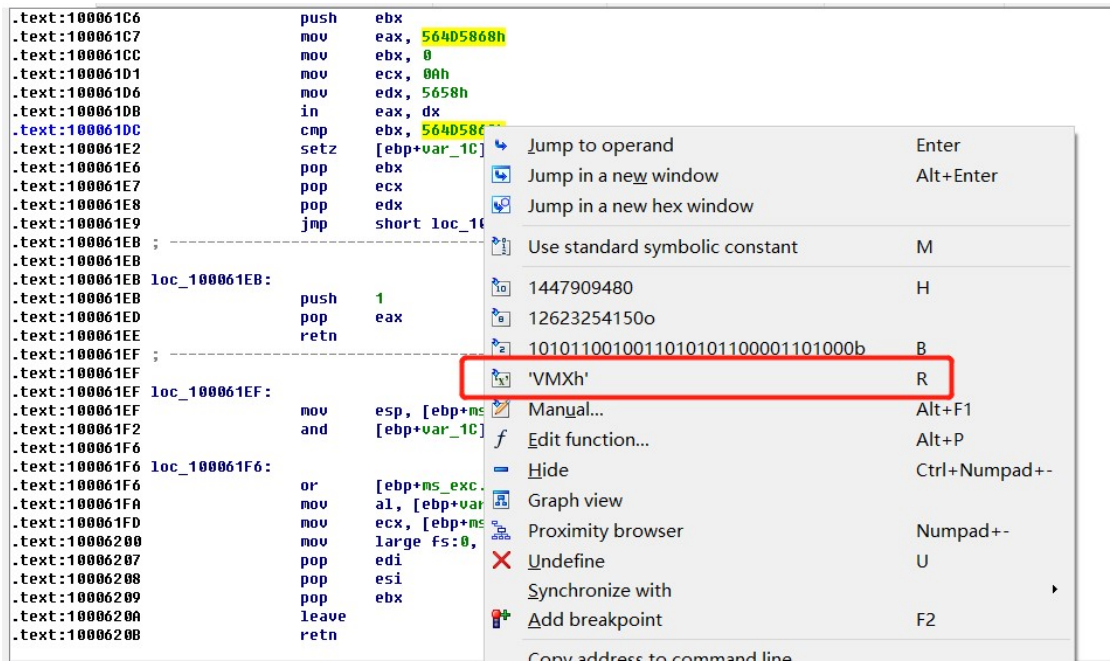
17. 搜索 in 指令（opcode 0xED）的使用。这个指令和一个魔术字符串 VMXh 用来进行 VMware 检测。这在这个恶意代码中被使用了吗？使用对执行 in 指令函数的交叉引用，能发现进一步检测 VMware 的证据吗?

用 Search -> Sequence of Bytes，搜索 `0xED` 并选择 Find All Occurrences，可以发现



所以 in 在这里被使用了。

双击进入



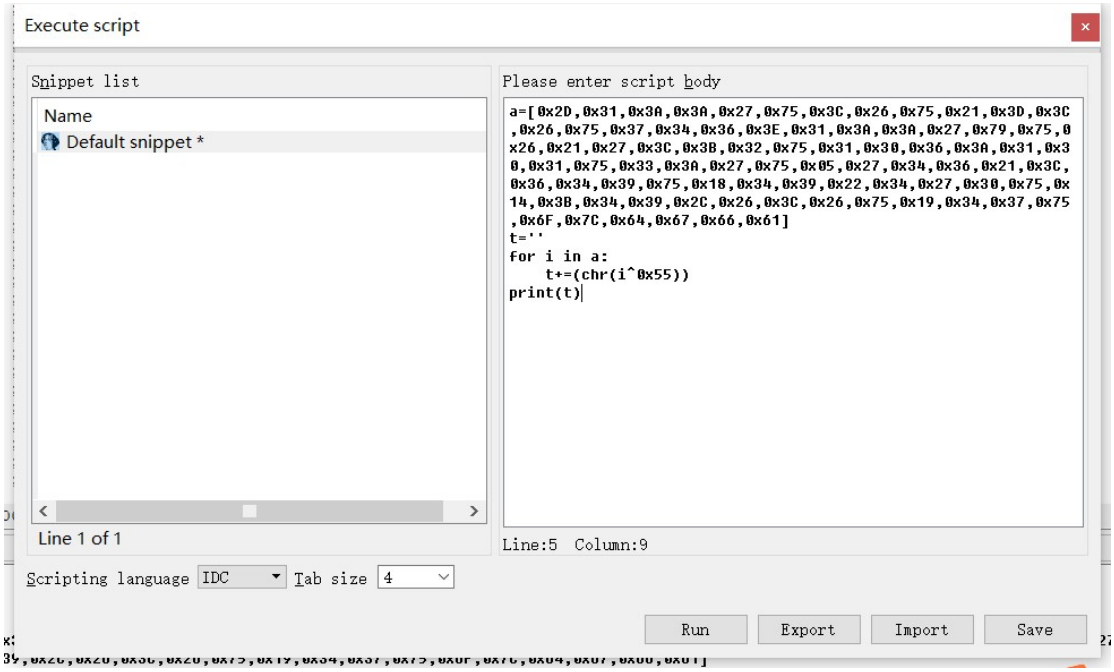
发现魔术字符串 VMXh。

18. 将你的光标跳转到 0x1001D988 处，你发现了什么？

跳转到 0x1001D988，发现是一堆乱码，猜测是异或加密

```
.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
```

每个字节与 0x55 进行 XOR 操作然后连接，得到：



结果为：xdoor is this backdoor, string decoded for Pratical Malware Analysis Lab:)1234

19. 如果你安装了 IDA Python 插件（包括 IDA Pro 的商业版本的插件），运行 Lab05-01.py，一个本书中随恶意代码提供的 IDA Pro Python 脚本，（确定光标是在 0x1001D988 处）在你运行这个脚本后发生了什么？

运行脚本后

```
Caching 'Strings window'... ok
Caching 'Strings window'... ok
```

捕捉到字符串，即破解了刚才被异或加密的字符串。

查看 Lab05-01.py

sea = ScreenEA()　　#得到当前光标处的地址

for i in range(0x00,0x50):　　#向后 50 个字节

　　b = Byte(sea+i)　　#得到当前的一个字节

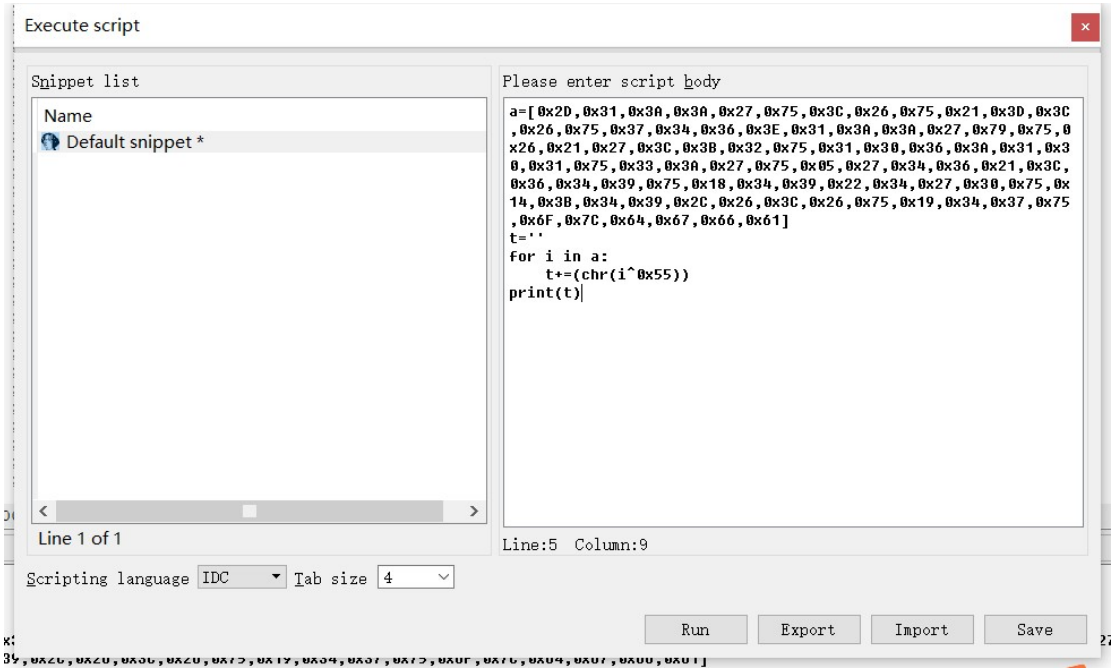
　　decoded_byte = b ^ 0x55　　#与 0x55 进行异或

　　PatchByte(sea+i,decoded_byte)　　#修改 sea+i 处的字节

即解密程序。

20. 将光标放在同一位置，你如何将这个数据转成一个单一的 ASCII 字符串？

运行我自己编写的脚本



21. 使用一个文本编辑器打开这个脚本。它是如何工作的?

查看 Lab05-01.py

sea = ScreenEA()　　#得到当前光标处的地址

for i in range(0x00,0x50):　　#向后 50 个字节

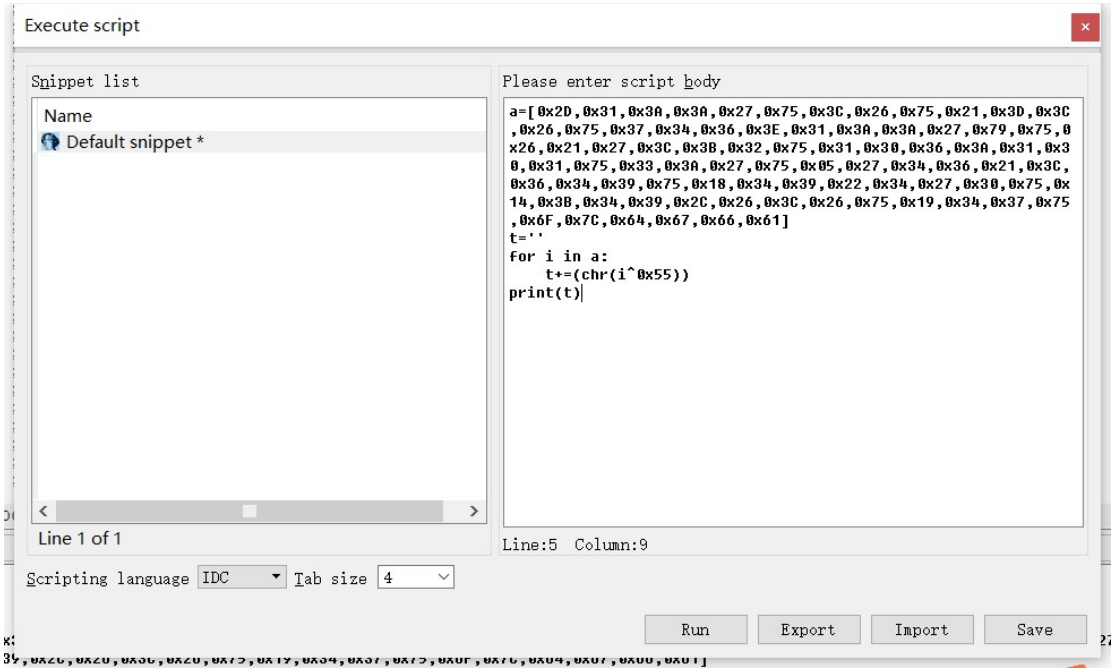
　　b = Byte(sea+i)　　#得到当前的一个字节

　　decoded_byte = b ^ 0x55　　#与 0x55 进行异或

　　PatchByte(sea+i,decoded_byte)　　#修改 sea+i 处的字节

即解密程序。

22. 编写的 Python 脚本



用于异或解密。

23. 编写 Yara 规则

rule Lab5_dll_feature{

meta:

description = "Lab05-01.dll's features"

strings:

$s1 = "\cmd.exe /c" fullword ascii

$s2 = "pics.praticalmalwareanalysis.com"

```
        $s3 = "xacq.dll" fullword ascii
    condition:
        $s1 and $s2 and $s3
}
```

测试：

```
D:\yara-v4.1.2-1693-win64>yara64 Lab3_rules.txt Lab05-01.dll
Lab3_1_exe_feature Lab05-01.dll
```

正确！

四、 实验心得

本次实验应用 IDA Pro 较为全面地对 Lab05-01.dll 进行了分析，掌握了许多 IDA Pro 分析方法：导入表、字符串、导出表、交叉引用、程序 CFG、字节序列、跳转、函数窗口等等一系列分析方法，极大提高了静态分析能力；

掌握 IDA Python 的基本语法和使用方法，利用 IDA Python 进行了简单的解密，掌握了在 IDA 中编写 Python 程序进行简化操作以及运行 Python 脚本的方法，提升了综合分析能力。