

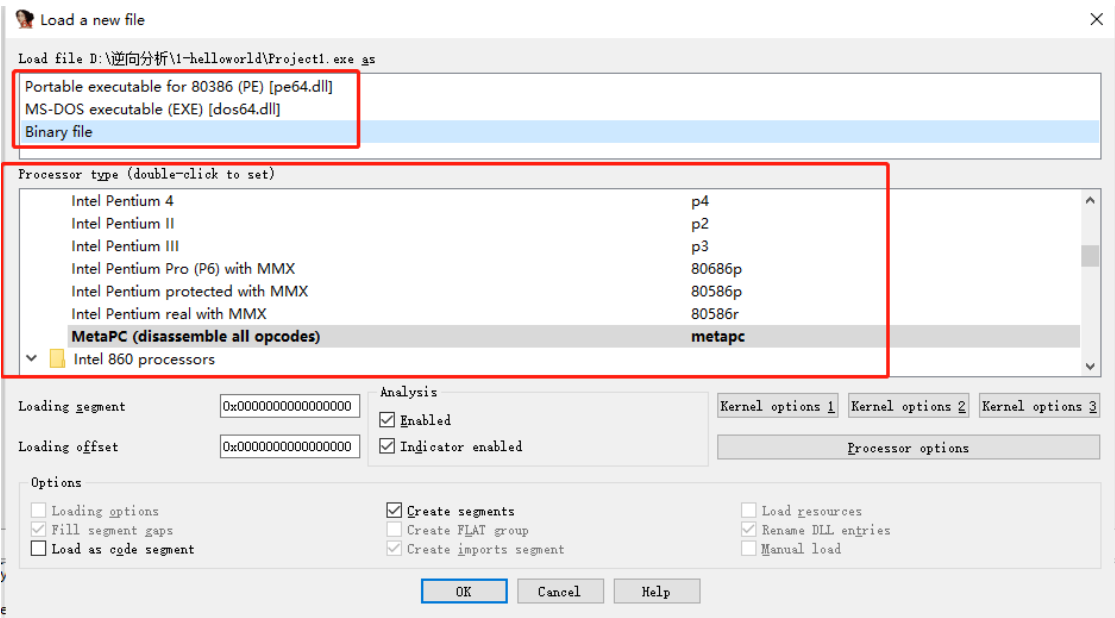
ida使用笔记

笔记本： 逆向笔记  
创建时间： 2022/10/9 10:53

- [一、常见使用方法](#)
- [二、案例分析](#)

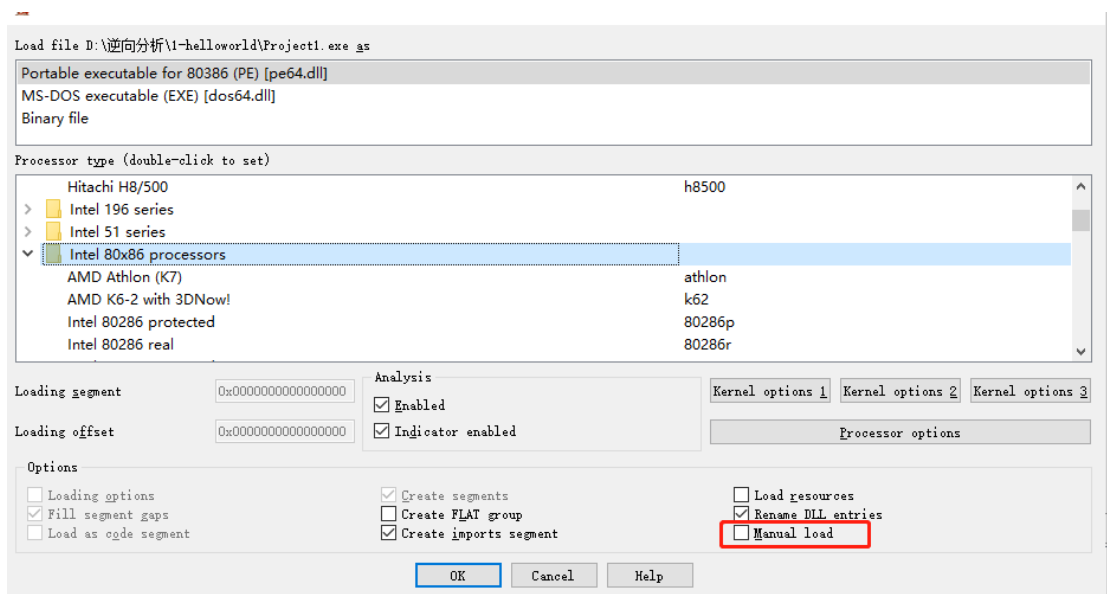
一、常见使用方法

虽然经常用ida但是从来没有花时间去系统的学习下该工具使用，最近正好有机会接触，就简单记录下一些常见的使用方法，当加载可执行文件时，ida会识别该文件的格式以及处理架构，文件通常为x86架构上运行的pe格式文件，当加载文件时，ida像操作系统加载一样，将文件映射到内存中，利用ida加载可执行文件，弹出如下对话框

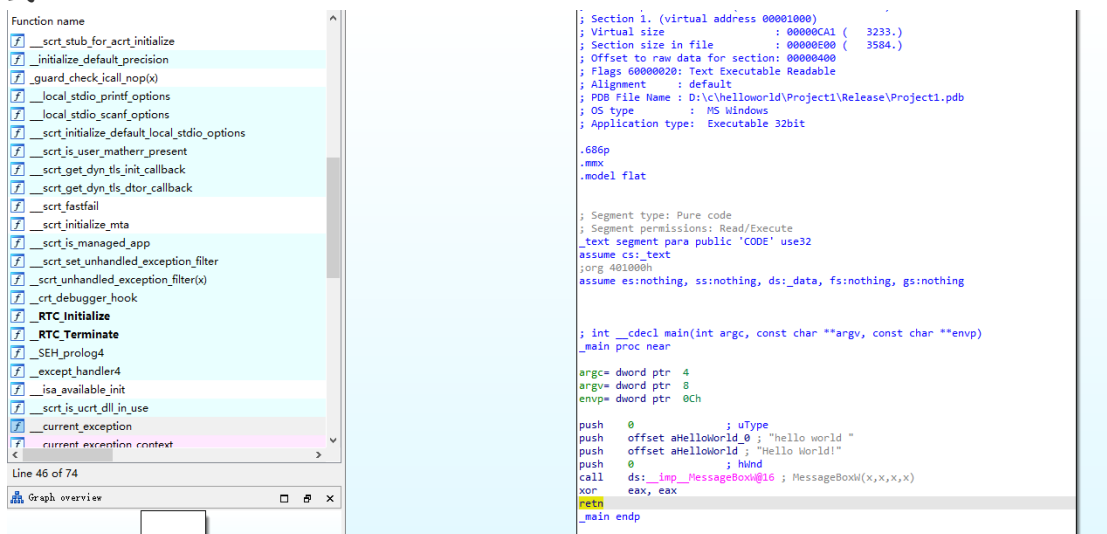


第一红框内表示加载文件的类型，第一个选项为可执行的pe文件，第二个选项为dos下pe文件，在进行恶意文件加载时建议选择binary file,因为恶意代码有时会带有shellcode、其他数据、加密参数等，甚至一些合法的可执行文件都可能会含有恶意代码，当运行此类可执行文件时，恶意代码并不会加载到内存中，选择该项可将一些含有shellcode的原始二进制文件加载并进行反汇编。

第二个红框为系统类型，本次案例是运行在Intel x86架构上，所以此处选择intel 80 x86 在进行分析时可能会出现加载的ida加载的地址和动态加载的不通，这可能是文件被基地址重定向原因，可通过manual load进行手动指定基地址



默认情况下ida的反汇编不包含pe头或资源节，加载完成后进入反汇编窗口，默认为图形模式



空格切换为文本模式，00401007为内存地址，.text为节名，左侧部分为箭头窗口，虚线标记了条件条状，实线标记了无条件跳转，

```
IDA View-A  Hex View-1  Structures  Enums
.text:00401000 ; +-----+
.text:00401000 ;
.text:00401000 ; Input SHA256 : 91970312B738F79322C9828FF86CE934089F97D63BA873E0AFC44E00377754A3
.text:00401000 ; Input MD5 : 52A2E54ECE3FE1C7F9D0284759FF0B0
.text:00401000 ; Input CRC32 : C26DD262
.text:00401000 ;
.text:00401000 ; File Name : D:\逆向分析\1-helloworld\Project1.exe
.text:00401000 ; Format : Portable executable for 80386 (PE)
.text:00401000 ; Imagebase : 400000
.text:00401000 ; Timestamp : 6340CE2B (Sat Oct 08 01:11:07 2022)
.text:00401000 ; Section 1. (virtual address 00001000)
.text:00401000 ; Virtual size : 00000CA1 ( 3233.)
.text:00401000 ; Section size in file : 00000E00 ( 3584.)
.text:00401000 ; Offset to raw data for section: 00000400
.text:00401000 ; Flags 60000020: Text Executable Readable
.text:00401000 ; Alignment : default
.text:00401000 ; PDB File Name : D:\c\helloworld\Project1\Release\Project1.pdb
.text:00401000 ; OS type : MS Windows
.text:00401000 ; Application type: Executable 32bit
.text:00401000 ;
.text:00401000 ; .686p
.text:00401000 ; .mmx
.text:00401000 ; .model flat
.text:00401000 ;
.text:00401000 ; =====
.text:00401000 ; Segment type: Pure code
.text:00401000 ; Segment permissions: Read/Execute
.text:00401000 ; _text segment para public 'CODE' use32
.text:00401000 ; assume cs:_text
.text:00401000 ; org 401000h
.text:00401000 ; assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
.text:00401000 ; ===== SUBROUTINE =====
.text:00401000 ;
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 ; _main proc near ; CODE XREF: __scrt_common_main_seh+F54p
.text:00401000 ;
.text:00401000 ; argc = dword ptr 4
.text:00401000 ; argv = dword ptr 8
.text:00401000 ; envp = dword ptr 0Ch
.text:00401000 ;
```

默认打开字体较小, 可通过options--font进行字体修改

```
000
000 argo
000 argv
000 envp
000
000
000
002
007
00C
00E
014
016
016 _mai
016
017
017 ; ===== SUBROUTINE =====
```

Select font for: Listings

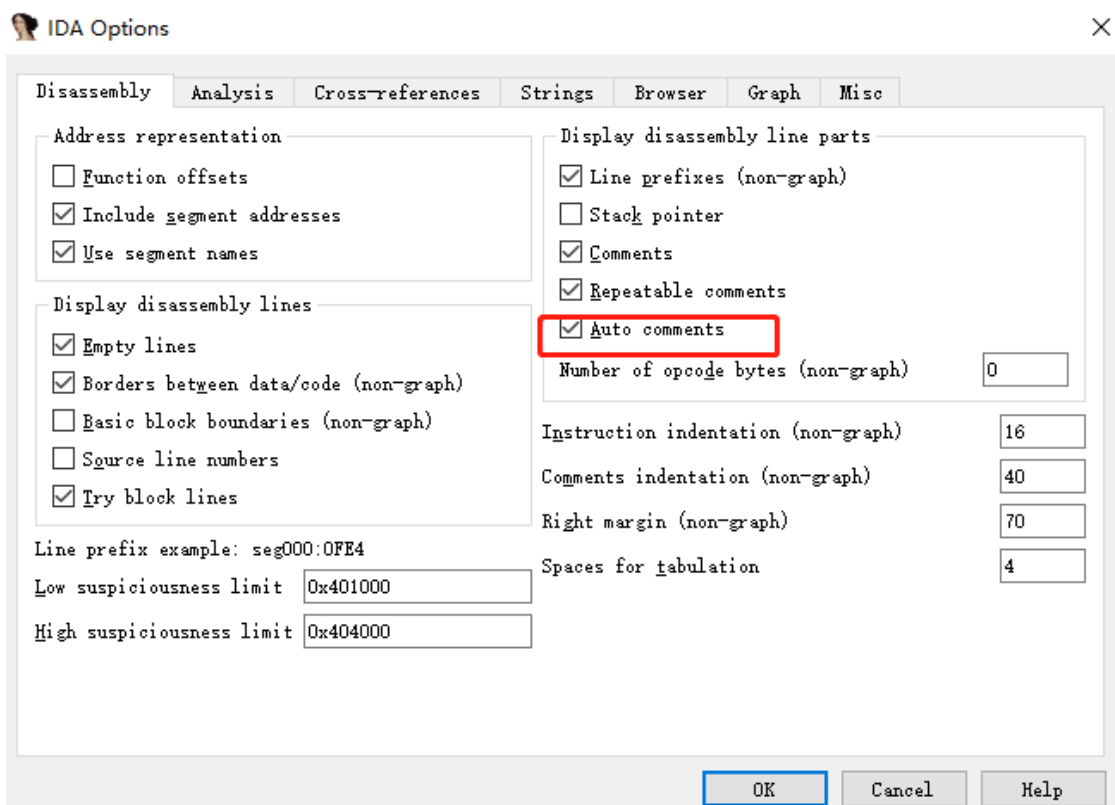
Font	Font style	Size
Consolas	Normal	18
Consolas	Normal	6
Courier	Bold	7
Courier New	Italic	8
Fixedsys	Bold Italic	9
HGOCR_CNKI		10
Lucida Console		11
Lucida Sans Typewriter		12
MS Gothic		14
SimSun-ExtB		16
Terminal		18

Sample: AaBbYyZz

Reset OK Cancel Apply

uType  
"hello w  
"Hello W  
hWnd  
oxW@16 ;

开启注释功能

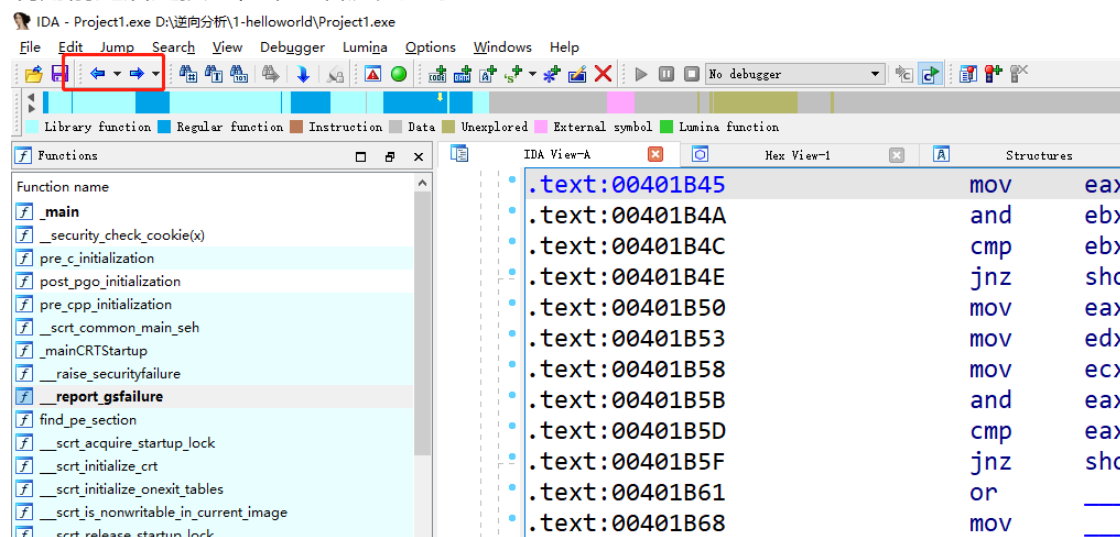


返回默认视图:

windows ---reset desktop

浏览历史:

利用前进后退按钮, 来查看历史记录



函数窗口:



```
1  #include "windows.h"
2  int main()
3  {
4      MessageBox(NULL, L"Hello World!", L"hello world ", MB_OK);
5      return 0;
6  }
```

导航栏:

浅蓝色: FLIRT识别的代码库 (IDA提供的一种函数识别技术,即库文件快速识别与鉴定技术)

红色: 编译器生成的代码

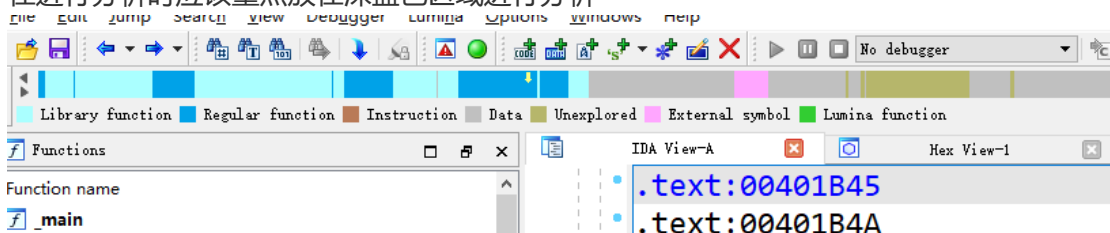
深蓝色: 用户编写的代码

粉红色: 为导入的数据

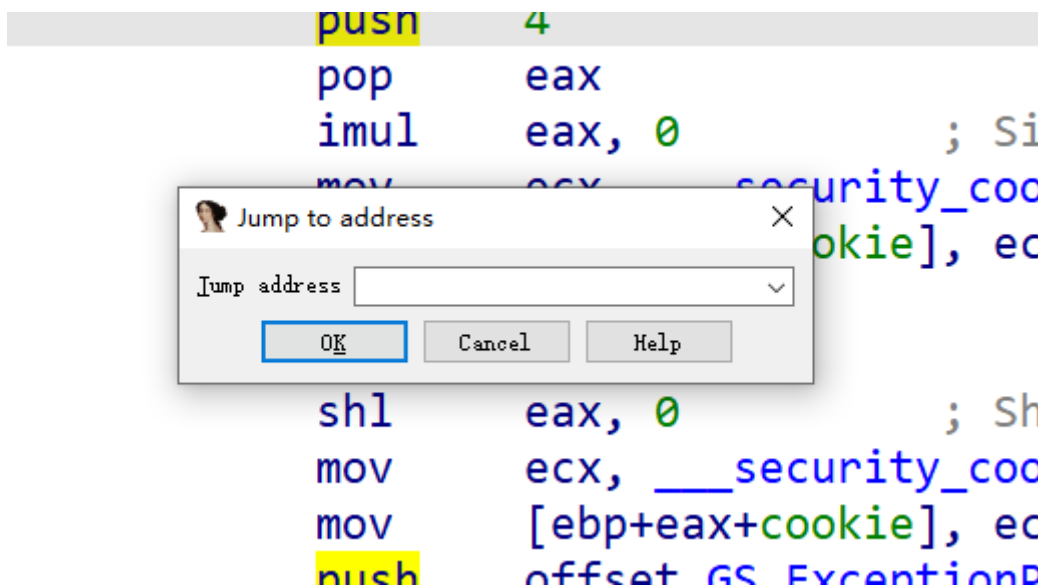
灰色: 已经定义的数据

棕色: 未定义的数据

在进行分析时应该重点放在深蓝色区域进行分析



跳转到任意虚拟内存:按G键, 输入需要跳转的地址



搜索:

search ---next code 移动光标到包含你指定指令的下一个位置

search --text 在整个反汇编串口搜索一个指定的字符串

search -- sequence of bytes 在十六进制试图窗口中对一个特定字节序执行二进制搜索

使用交叉引用:

交叉引用, 在ida中称为xref, 可告诉你一个函数在何处被调用, 或是一个字符串在何处被

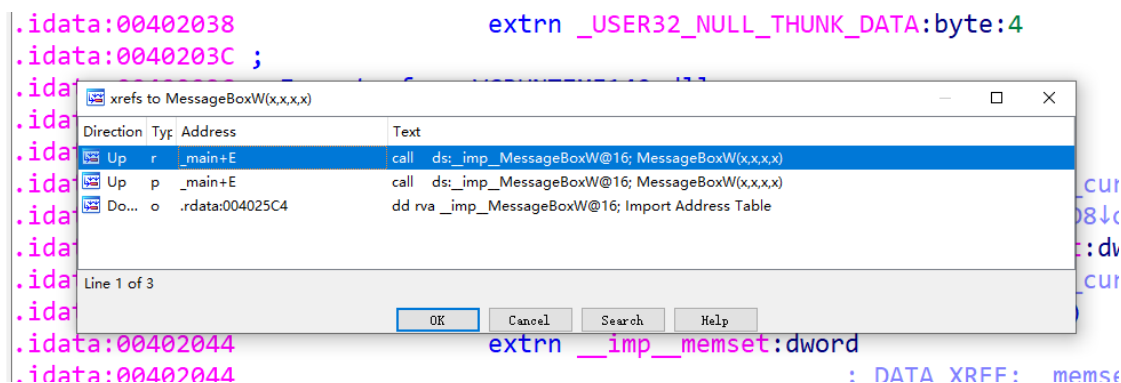
使用，如果想知道一个函数在被调用时用了哪些参数，可以快速浏览这些此参数被放在栈上的什么位置

1处代码的交叉引用，说明函数401000在main函数内部偏移0x3处被调用

2处代码说明哪个跳转带我们到这里，该点是相对401000偏移量为0x19处的jmp指令

```
00401000      sub_401000      proc near      ; ①CODE XREF: _main+3p
00401000      push     ebp
00401001      mov      ebp, esp
00401003      loc_401003:      ; ②CODE XREF: sub_401000+19j
00401003      mov      eax, 1
00401008      test     eax, eax
0040100A      jz       short loc_40101B
0040100C      push     offset aLoop      ; "Loop\n"
00401011      call     printf
00401016      add      esp, 4
00401019      jmp      short loc_401003 ③
```

显示一个函数所有的交叉引用，单击函数名并按x



数据交叉引用：

数据交叉引用，主要是被用来跟踪一个二进制文件中的数据访问，数据访问可以通过内存引用关联代码中引用数据的任意一个字节

1处可见dword 0x7F00001h,该数据在0x401020处的函数中被使用

```
0040C000 dword_40C000      dd 7F000001h      ; ①DATA XREF: sub_401020+14r
0040C004 aHostnamePort      db '<Hostname> <Port>',0Ah,0 ; DATA XREF: sub_401000+30
```

ida是允许你进行反汇编代码修改的，但是没有撤销功能，所以在自行修改时要谨慎。

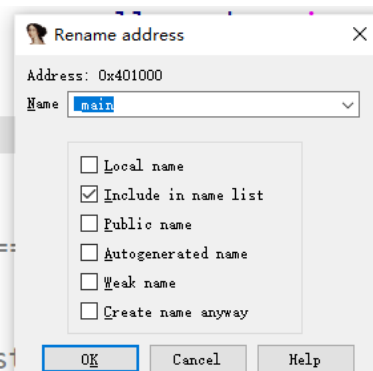
函数重命名：

多数情况下，ida会自动命名虚拟地址，如sub\_40100，为了更方便的记忆，可对这些函数进行重命名。

```

.text:0040100E
.text:00401014
.text:00401016
.text:00401016 _main
.text:00401016
.text:00401017
.text:00401017 ; =====
.text:00401017
.text:00401017 ; void __fast
.text:00401017 @__security_check_cookie@4 proc near ; DATA
.text:00401017 cookie = ecx

```



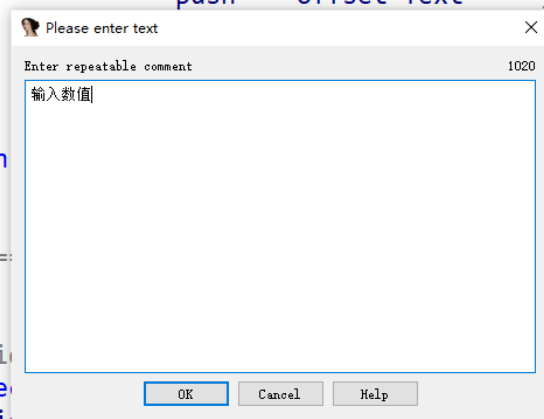
注释:

冒号; 可添加注释

```

00401000 envp          = dword ptr 0Ch
00401000
00401000          push    0          ; uType
00401002          push    offset Caption ; 111
00401007          push    offset Text   ; "Hello World!"
0040100C          hWnd
0040100E          ; MessageBoxW@16 ; MessageBoxW(x,x
00401014
00401016
00401016 _main
00401016
00401017
00401017 ; =====
00401017
00401017 ; void
00401017 @__se
00401017 cookie = ecx

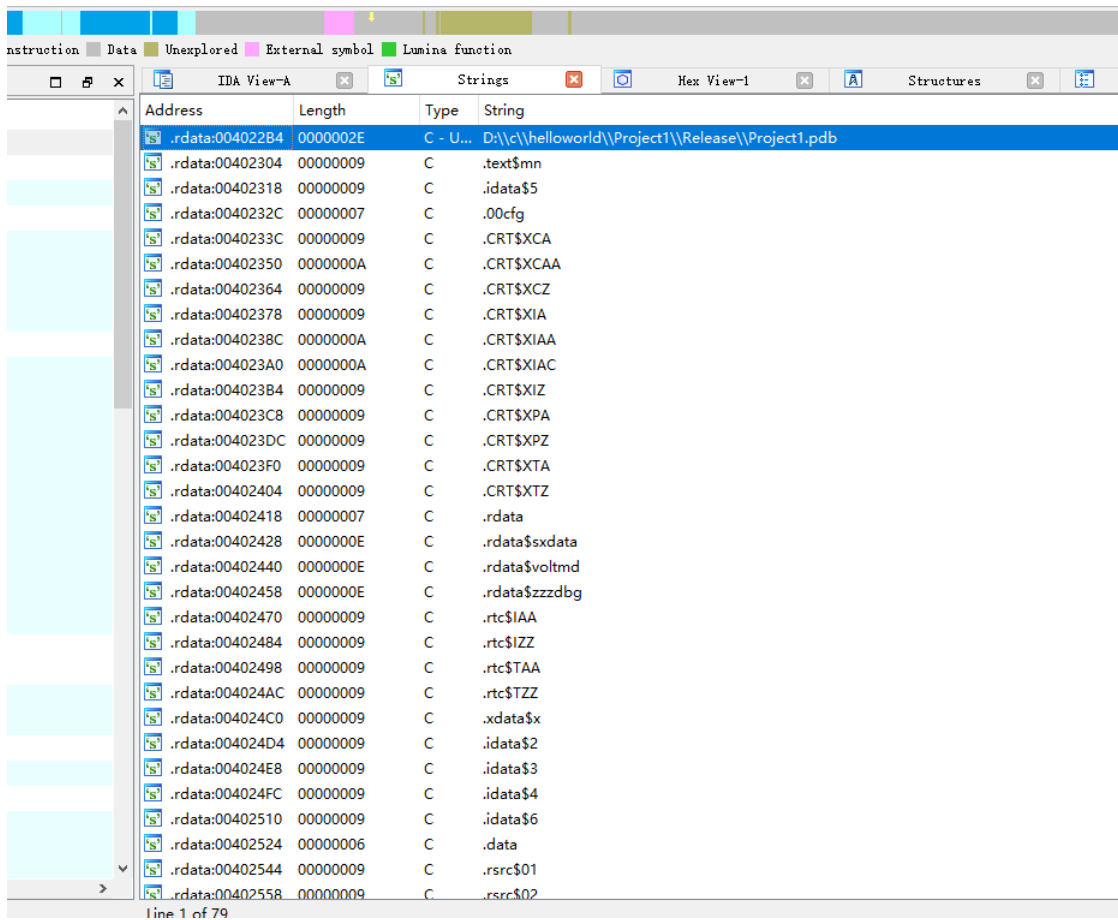
```



字符串窗口:

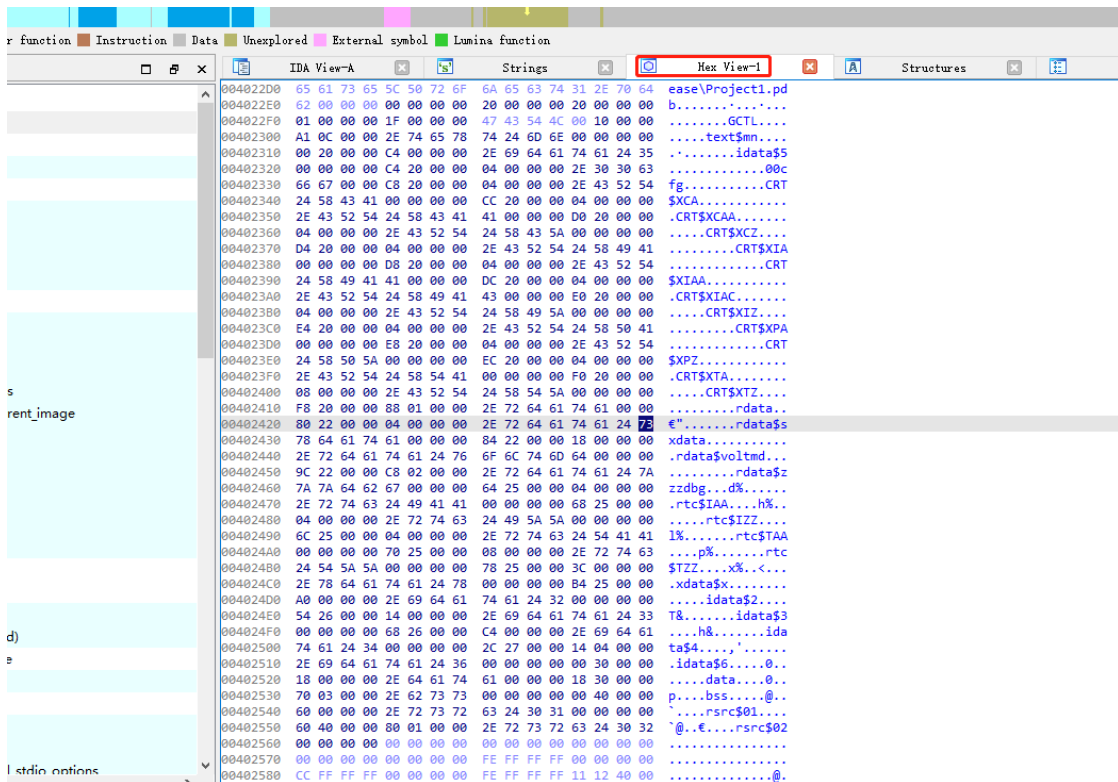
shift+f12





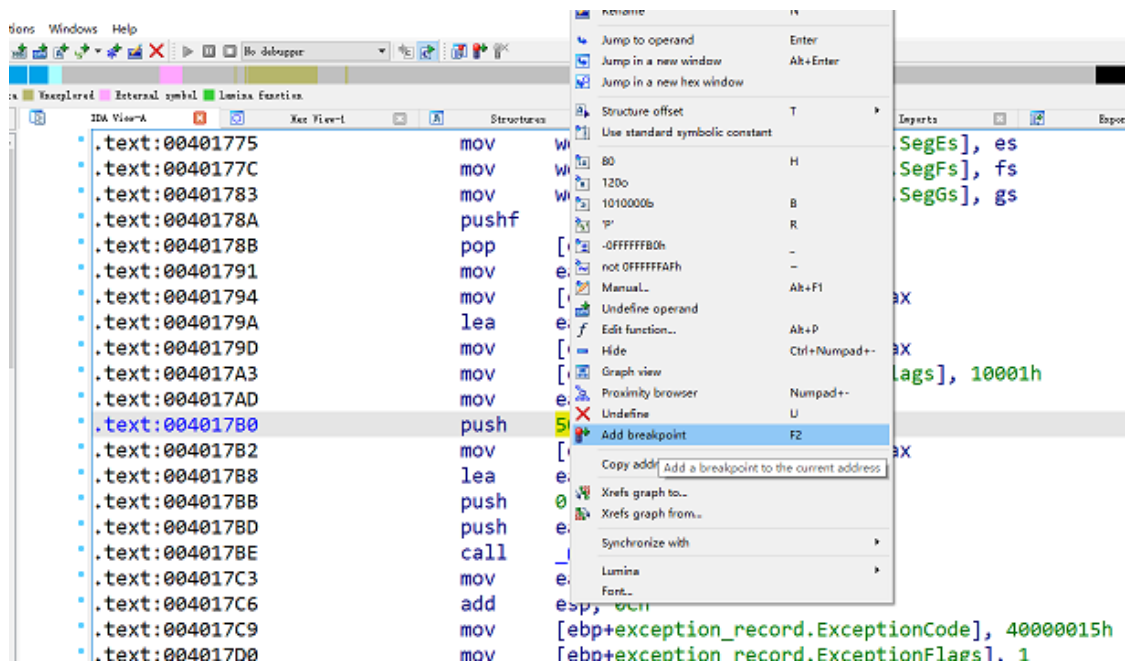
会显示所有的字符串，在进行给te'zhe定位时，是一种很好用的方法。

十六进制窗口：



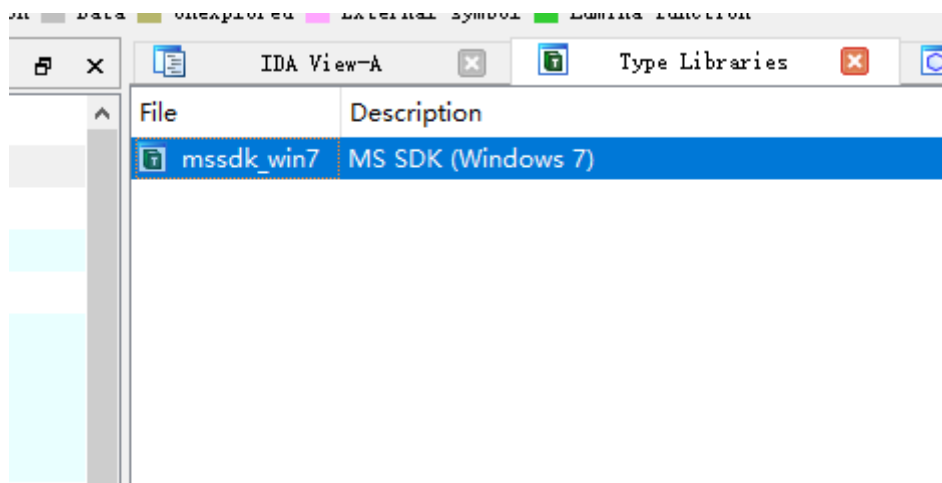
格式化操作数

ida会对反汇编的每一条指令的操作数进行格式化，被显示的数据被格式化为16进制的值，如push 50h，分别为修改为10进制、8进制、二进制、ASCII字符




























手动加载类型库

当发现查看的标准符号常量不会显示时，可进行手动加载，选择view--->open subviews-->type libraries，来查看当前加载库



可进行手动加载

File	Loaded	Description
 bc31		Borland C++ v3.1
 bc5dos		Borland C++ v5.x 16bit DOS
 bc5w16		Borland C++ v5.x 16bit Windows
 bcb4win		Borland CBuilder v4 <windows.h>
 bcb5win		Borland CBuilder v5 <windows.h>
 geos		GEOS types
 gnucmn		GNU C++ common
 gnu1nx_x64		GNU C++ x64 Linux
 gnu1nx_x86		GNU C++ x86 Linux
 gnuunx		GNU C++ unix
 gnuunx64		GNU C++ 64bit unix
 gnuwin		GNU C++ cygwin
 macos11_s...		MacOSX11.0.sdk 64-bit headers
 macosx		Mac OS 32-bit headers (deprecated, use MacOSX.sdk tils instead)
 macosx64		Mac OS 64-bit headers (deprecated, use MacOSX.sdk tils instead)
 macosx64_...		MacOSX10.14.sdk 64-bit headers
 macosx64_...		MacOSX10.15.sdk 64-bit headers
 macosx_sd...		MacOSX10.14.sdk 32-bit headers
 ms16dos		Microsoft C 16bit DOS
 ms16win		Microsoft C 16bit Windows
 mscor		Microsoft Visual Studio.Net
 mssdk		MS SDK (Windows XP)
 mssdk64_vi...		MS SDK (Windows Vista x64)
 mssdk64_w...		MS SDK (Windows 10 x64)
 mssdk64_w...		MS SDK (Windows 7 x64)

Line 14 of 94

重定义代码和数据：

利用ida对一个程序进行初始反汇编时，字节会出现错误分类，如代码被定义为数据，数据被定义为代码，按U键来取消函数、代码或数据的定义。

C键定义原始字节为代码；

D或A键分别定义原始字节为数据或ASCII字符串

## 二、案例分析

因为主要是参考恶意代码分析实战一书，本次主要以lab5-1为案例进行分析：

1、dllmain地址

地址为1000D02E

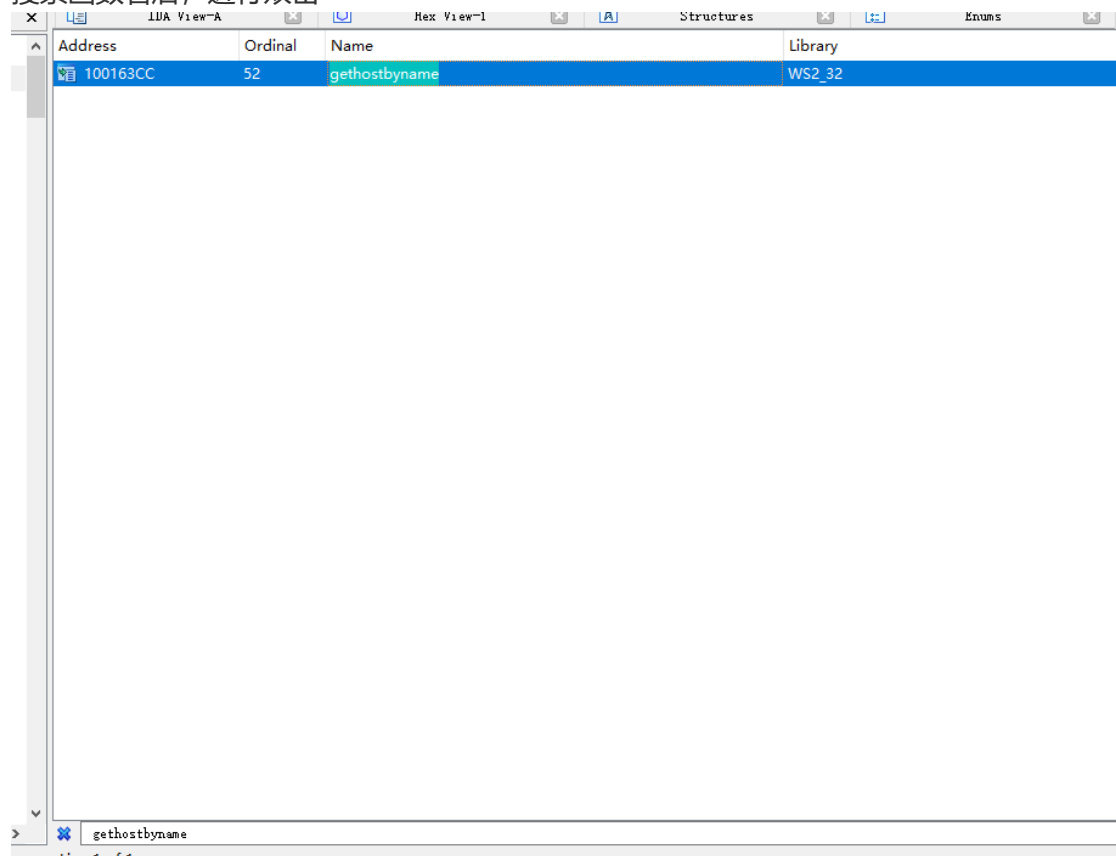
```

.text:1000D02E
.text:1000D02E
.text:1000D02E ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
.text:1000D02E _DllMain@12      proc near      ; CODE XREF: DllEntryPoint+4
.text:1000D02E                                     ; DATA XREF: sub_100110FF+2C
.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
.text:1000D039      mov     eax, [esp+hinstDLL]
.text:1000D03D      push    ebx
.text:1000D03E      mov     ds:hModule, eax
.text:1000D043      mov     eax, off_10019044 ; "[This is RUR]"
.text:1000D048      push    esi
.text:1000D049      add     eax, 0Dh
.text:1000D04C      push    edi
.text:1000D04D      push    eax      ; Str

```

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

搜索函数名后, 进行双击



双击跳转到函数地址, 100163CC

```

.idata:100163C8 ; sub_100163C8
.idata:100163C8 ; Import
.idata:100163CC ; struct hostent *(__stdcall *gethostbyname)(const char *, struct hostent *)
.idata:100163CC extrn gethostbyname:dword
.idata:100163CC ; CODE
.idata:100163CC ; sub_100163CC
.idata:100163CC ; Import
.idata:100163D0 ; char *(__stdcall *inet_ntoa)(struct in_addr)
.idata:100163D0 extrn inet_ntoa:dword ; CODE
.idata:100163D0 ; sub_100163D0
.idata:100163D0 ; Import
.idata:100163D4 ; int (__stdcall *recv)(SOCKET s, char *buf, int len, int flags)
.idata:100163D4 extrn recv:dword ; CODE
.idata:100163D4 ; sub_100163D4

```

3、有多少函数调用了gethostbyname

选中函数ctrl+x查看交叉引用，共有18行，5个函数引用sub\_10001074、sub\_10001365、sub\_10001656、sub\_1000208F、sub\_10002CCE

xrefs to gethostbyname

Direction	Type	Address	Text
Up	r	sub_10001074+1D3	call ds:gethostbyname
Up	p	sub_10001074+1D3	call ds:gethostbyname
Up	r	sub_10001074+26B	call ds:gethostbyname
Up	p	sub_10001074+26B	call ds:gethostbyname
Up	r	sub_10001074:loc_100011AF	call ds:gethostbyname
Up	p	sub_10001074:loc_100011AF	call ds:gethostbyname
Up	r	sub_10001365+1D3	call ds:gethostbyname
Up	p	sub_10001365+1D3	call ds:gethostbyname
Up	r	sub_10001365+26B	call ds:gethostbyname
Up	p	sub_10001365+26B	call ds:gethostbyname
Up	r	sub_10001365:loc_100014A0	call ds:gethostbyname
Up	p	sub_10001365:loc_100014A0	call ds:gethostbyname
Up	r	sub_10001656+101	call ds:gethostbyname
Up	p	sub_10001656+101	call ds:gethostbyname
Up	r	sub_1000208F+3A1	call ds:gethostbyname
Up	p	sub_1000208F+3A1	call ds:gethostbyname
Up	r	sub_10002CCE+4F7	call ds:gethostbyname
Up	p	sub_10002CCE+4F7	call ds:gethostbyname

Line 5 of 18

OK Cancel Search Help

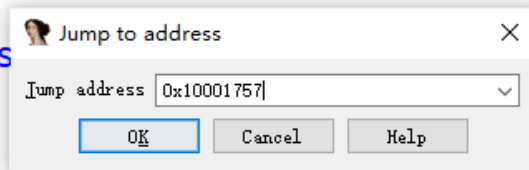
4、将精力集中在位于0x10001757处的对gethostbyname的调用，你能找出那个DNS请求将被触发吗？

G直接跳转到该地址：

```

0163CC ; col
0163CC ; sub
0163CC ; Imp
0163D0 ; char *(__stdcall *inet_ntoa)(struct in_addr
0163D0 extrn inet_ntoa:dword ; COE
0163D0 ; sub
0163D0 ; Imp
0163D4 ; int (__s char *buf,
0163D4 ; COE
0163D4 ; sub
0163D4 ; Imp
0163D8 ; int (__stdcall *send)(SOCKET s, const char
0163D8 extrn send:dword ; COE

```



跳转后代码如下，eax最开始的值为[This is RDO]pics.practicalmalwareanalysis.com，在add 0dh后，也就是进行13位偏移后，变为pics.practicalmalwareanalysis.com，所以1757函数带入的参数位上述网址

```

.text:1000173A test     eax, eax
.text:1000173C jz       loc_100017ED
.text:10001742 cmp      dword_1008E5CC, ebx
.text:10001748 jnz      loc_100017ED
.text:1000174E mov      eax, off_10019040 ; "[This is RDO]pics.practicalmalwareanalys..."
.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
.text:10001761 jz       short loc_100017C0
.text:10001763 movsx    eax, word ptr [esi+0Ah]
.text:10001767 push     eax ; Size
.text:10001768 mov      eax, [esi+0Ch]
.text:1000176B push     dword ptr [eax] ; Src
.text:1000176D lea      eax, [esp+690h+in]
.text:10001771 push     eax ; void *
.text:10001772 call     memcpy
.text:10001777 mov      av, [esi+8]

```

查看1757函数的伪代码也可发现，调用gethostname函数访问该地址：

```

79 {
80     Error = WSAGetLastError();
81     printf("socket() GetLastError reports %d\n", Error);
82 }
83 if ( memcmp(off_10019040[0] + 13, asc_10093540, 0x10u) && !dword_1008E5C
84 {
85     v8 = gethostbyname(off_10019040[0] + 13);
86     v9 = v8;
87     if ( v8 )
88     {
89         memcpy(&in, *(const void **)v8->h_addr_list, v8->h_length);
90         h_addrtype = v9->h_addrtype;
91         v10 = inet_ntoa(in);
92         strncpy(cp, v10, 0x10u);
93         strncpy(String, off_10019038[0] + 13, 5u);
94         WinExec(CmdLine, 0);
95     }

```

5、ida pro识别了在0x10001656处的子过程中的多少个局部变量  
直接G跳转到该地址,如下图

```

.text:10001656 sub_10001656      proc near                                ; DATA XREF:
.text:10001656
.text:10001656 var_675          = byte ptr -675h
.text:10001656 var_674          = dword ptr -674h
.text:10001656 hModule          = dword ptr -670h
.text:10001656 timeout          = timeval ptr -66Ch
.text:10001656 name             = sockaddr ptr -664h
.text:10001656 var_654          = word ptr -654h
.text:10001656 in               = in_addr ptr -650h
.text:10001656 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

```

局部参数通常以var开头，偏移值为负值，参数的偏移值为正，所以此处的局部变量数为23

6、ida pro识别了在0x10001656处的子过程中的多少个参数

参数通常正cun'z偏移，且一般使用arg为前缀，所以此处识别出一个全局参数

```

.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                ; Inte
• .text:1000165C                push    ebx
• .text:1000165D                push    ebp
• .text:1000165E                push    esi

```

7、使用strings窗口，来在反汇编中定位字符串cmd.exe /c，它位于哪

f12定位到string窗口，然后ctrl+f进行搜索，发现cmd.exe



Address	Length	Type	String
xdoors_d:1009... 0000000A		C	startxcmd
xdoors_d:1009... 0000000D		C	\\cmd.exe /c

双击进行找到具体位置，其地址为10095B34

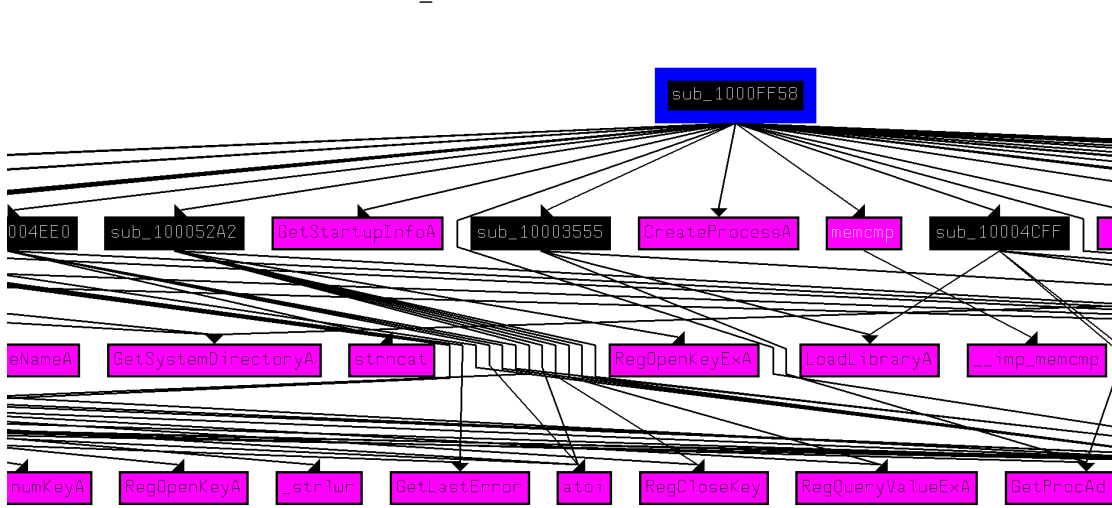
```

xdoors_d:10095B1D align 10h
xdoors_d:10095B20 ; char aCommandExeC[]
xdoors_d:10095B20 aCommandExeC db '\command.exe /c ',0 ; DAT
xdoors_d:10095B31 align 4
xdoors_d:10095B34 aCmdExeC db '\cmd.exe /c ',0 ; DAT
xdoors_d:10095B41 align 4
xdoors_d:10095B44 ; char aHiMasterDDDDDD[]
xdoors_d:10095B44 aHiMasterDDDDDD db 'Hi,Master [%d/%d/%d %d:%c
xdoors_d:10095B44 ; DAT
xdoors_d:10095B44 db 'WelCome Back...Are You Er
xdoors_d:10095B44 db 00h 00h

```

8、在引用cmd.exe /c代码所在的区域发生了什么

双击定位到该部分代码，发现sub\_1000FF58函数引用了该部分内容



查看该函数伪代码，通过对函数上下进行判断，“Encrypt Magic Number For This Remote Shell Session [0x%02x]\r\n”，可判断该程序是调用cmd执行一个远程shell



```

PipeAttributes.lpSecurityDescriptor = 0;
PipeAttributes.bInheritHandle = 1;
if ( CreatePipe(&hReadPipe, &hWritePipe, &PipeAttributes, 0) )
{
    StartupInfo.cb = 68;
    GetStartupInfo(&StartupInfo);
    StartupInfo.hStdError = hWritePipe;
    StartupInfo.hStdOutput = hWritePipe;
    StartupInfo.wShowWindow = 0;
    StartupInfo.dwFlags = 257;
    GetSystemDirectoryA(Destination, 0x400u);
    if ( dword_1008E5C4 )
        strcat(Destination, aCmdExeC);
    else
        strcat(Destination, aCommandExeC);
    memset(Buf1, 0, 0xFFu);
}
BEL_8:
    v3 = 0;
    while ( 1 )
    {

```

只能进行初步判断，不能确定

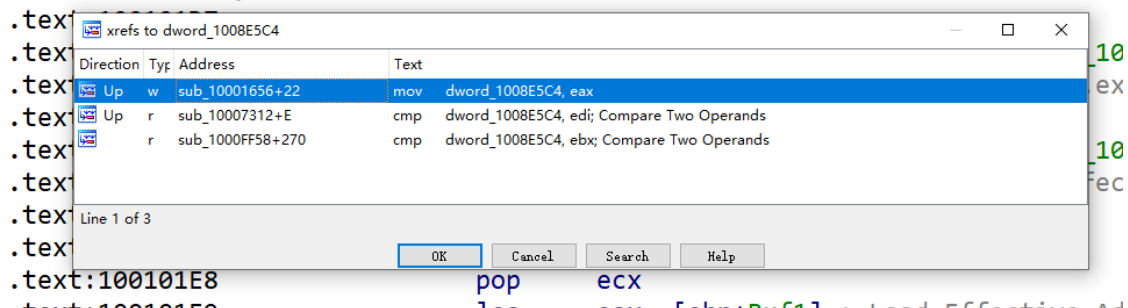
9、在同样的区域里，在0x100101C8处，看起来好像dword\_1008E5C4是一个全局变量，它帮助决定走那一条路径？恶意代码是如何设置dword\_1008E5C4的？（提示：使用dword\_1008E5C4的交叉引用）

G跳转到该行代码，发现是cmp是个比较，x查看哪些地方引用了1008e5c4

```

.text:100101C2      call     ds:GetSystemDirectoryA ; Indirect C
.text:100101C8      cmp     dword_1008E5C4, ebx ; Compare Two O
.text:100101CE      jz      short loc_100101D7 ; Jump if Zero (
.text:100101D0      push    offset aCmdExeC ; "\\cmd.exe /c "
.text:100101D5      jmp     short loc_100101DC ; Jump
.text:100101D7 ; -----

```



发现存在3处交叉引用，选择第一处查看

```

.text:1000166B      mov     [esp+688h+var_674], ebx
.text:1000166F      mov     [esp+688h+hModule], ebx
.text:10001673      call    sub_10003695 ; Call Procedure
.text:10001678      mov     dword_1008E5C4, eax
.text:1000167D      call    sub_100036C3 ; Call Procedure
.text:10001682      push    3A98h ; dwMilliseconds
.text:10001687      mov     dword_1008E5C8, eax
.text:1000168C      call    ds:Sleep ; Indirect Call Near Proc
.text:10001692      call    sub_100110FF ; Call Procedure
.text:10001697      lea     eax, [esp+688h+WSAData] ; Load Effective
.text:1000169E      push    eax ; lpWSAData

```

将eax的值给1008E5E4,而该值为函数10003695的返回值, 查看10003695的代码信息

```
.text:10003695 VersionInformation= _OSVERSIONINFOA ptr -94h
.text:10003695
.text:10003695      push    ebp
.text:10003696      mov     ebp, esp
.text:10003698      sub     esp, 94h          ; Integer Subtraction
.text:1000369E      lea     eax, [ebp+VersionInformation] ; Load Effective Address
.text:100036A4      mov     [ebp+VersionInformation.dwOSVersionInfoSize], 94h
.text:100036AE      push    eax                ; lpVersionInformation
.text:100036AF      call    ds:GetVersionExA ; Indirect Call Near Procedure
.text:100036B5      xor     eax, eax            ; Logical Exclusive OR
.text:100036B7      cmp     [ebp+VersionInformation.dwPlatformId], 2 ; Compare T
.text:100036BE      setz    al                 ; Set Byte if Zero (ZF=1)
.text:100036C1      leave   ; High Level Procedure Exit
.text:100036C2      retn    ; Return Near from Procedure
.text:100036C2 sub_10003695      endp
.text:100036C3
```

首先eax为VersionInformation的地址, 然后94h空间赋值给

VersionInformation.dwOSVersionInfoSize, eax入栈, 调用GetVersionExA函数, 该函数为获取系统版本, eax异或为空, VersionInformation.dwPlatformId与2进行比较判断系统类型

为0表示为win3x系统; 为1表示为win9x系统; 为2表示为winNT; 为3表示为win2000系统; 若判断为2的话, zf为1, 执行setz指令后, al寄存器为1, 否则为0, 函数的返回值为al, 也即是eax的低8位。所以该函数影响了al的值, 主要用于确定系统版本。

10、在位于0x1000FF58处的子过程中的几百行指令中, 一系列使用memcmp来比较字符串的比较。如果对robotwork的字符串比较是成功的(当memcmp返回0), 会发生什么?

首先G跟进到该函数位置, 找到robotwork的位置

```
.text:10010444 ; -----
.text:10010444
.text:10010444 loc_10010444: ; CODE XREF: sub_1000FF58+
.text:10010444      push    9                ; Size
.text:10010446      lea     eax, [ebp+Buf1]
.text:1001044C      push    offset aRobotwork ; "robotwork"
.text:10010451      push    eax                ; Buf1
.text:10010452      call    memcmp
.text:10010457      add     esp, 0Ch
.text:1001045A      test    eax, eax
.text:1001045C      jnz     short loc_10010468
.text:1001045E      push    [ebp+s]           ; s
.text:10010461      call    sub_100052A2
.text:10010466      jmp     short loc_100103F6
.text:10010468 ; -----
.text:10010468
```

当memcmp为0时, zf被置位, 所以不进行跳转, 则调用sub\_100052A2函数, 然后跳转到loc\_100103F6

查看函数sub\_100052A2内容

```

    return RegCloseKey(phkResult);
if ( !RegQueryValueExA(phkResult, aWorktime, 0, &Type, Data, &cbData) )
{
    v2 = atoi((const char *)Data);
    sprintf(Buffer, "\r\n\r\n[Robot_WorkTime :] %d\r\n\r\n", v2);
    v3 = strlen(Buffer);
    sub_100038EE(s, (int)Buffer, v3);
}
memset(Data, 0, sizeof(Data));
if ( !RegQueryValueExA(phkResult, aWorktimes, 0, &Type, Data, &cbData) )
{
    v4 = atoi((const char *)Data);
    sprintf(Buffer, "\r\n\r\n[Robot_WorkTimes:] %d\r\n\r\n", v4);
    v5 = strlen(Buffer);
    sub_100038EE(s, (int)Buffer, v5);
}
return RegCloseKey(phkResult);
}
xt:100052D7      rep stosa
xt:100052D9      stosw
xt:100052DB      stosb
xt:100052DC      lea     eax, [ebp+phkResult]
xt:100052DF      push    eax                ; phkResult
xt:100052E0      push    0F003Fh           ; samDesired
xt:100052E5      push    0                 ; ulOptions
xt:100052E7      push    offset aSoftwareMicros ; "SOFTWARE\\Microsoft\\Windows\\CurrentVe"
xt:100052EC      push    80000002h         ; hKey
xt:100052F1      call    ds:RegOpenKeyExA
xt:100052F7      test    eax, eax
xt:100052F9      jz      short loc_10005309
xt:100052FB      push    [ebp+phkResult] ; hKey
xt:100052FE      call    ds:RegCloseKey

```

获取WorkTime和WorkTimes的键值。然后将格式化的字符串调用sub\_100038EE发送。

11、PSLIST导出函数做了什么？

进入exports查看该函数

Name	Address	Ordinal
InstallRT	1000D847	1
InstallSA	1000DEC1	2
InstallSB	1000E892	3
PSLIST	10007025	4
ServiceMain	1000CF30	5
StartEXS	10007ECB	6
UninstallRT	1000F405	7
UninstallSA	1000EA05	8
UninstallSB	1000F138	9
<b>DllEntryPoint</b>	<b>1001516D</b>	<b>[main entry]</b>

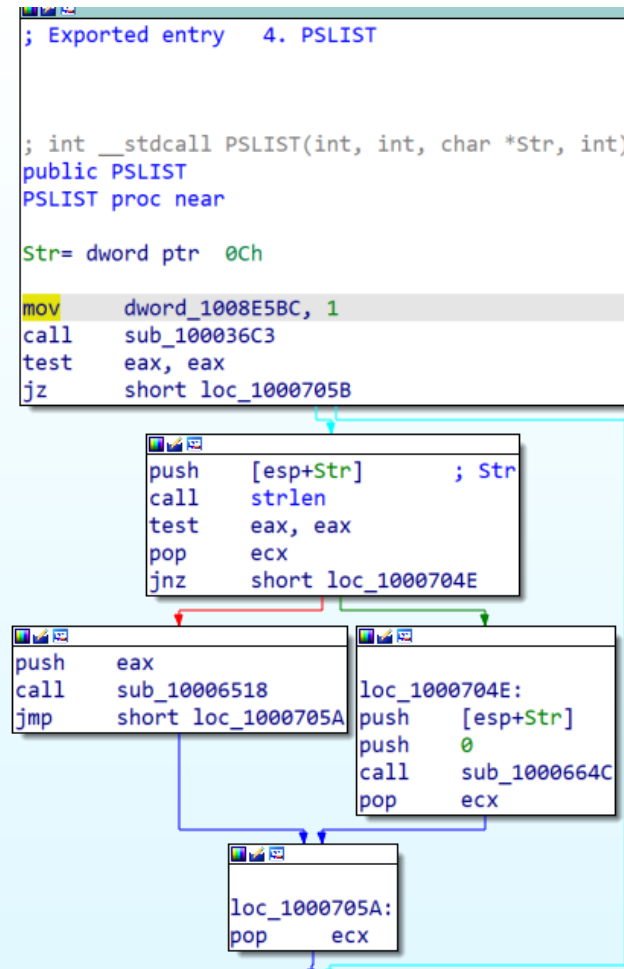
函数具体内容如下：

```

IDA View-A x Pseudocode-D x Pseudocode-C x Pseudocode-B x Pseudocode-A x Hex View-1 x Structures x
1 int __stdcall PSLIST(int a1, int a2, char *Str, int a4)
2 {
3     int result; // eax
4
5     dword_1008E5BC = 1;
6     result = sub_100036C3();
7     if ( result )
8     {
9         if ( strlen(Str) )
10            result = sub_1000664C(0, Str);
11        else
12            result = sub_10006518(0);
13    }
14    dword_1008E5BC = 0;
15    return result;
16 }

```

查看该函数流程图如下：



该函数将`dword_1008E5BC`赋值为1，然后调用`sub_100036C3`，跟进该函数，发现该函数主要时获取平台版本信息，获取的信息与2进行对比，判断是否为win系统

```

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

```

执行完该函数后，出现两个判断分支，如果是的话就直接进行退出，如果否的话继续执行，调用 sub\_10006518、sub\_1000664C这两个函数，通过查看这两个函数内容，发现函数调用了CreateToolhelp32Snapshot函数，该函数主要是通过获取进程信息为指定的进程、进程使用的堆[HEAP]、模块[MODULE]、线程建立一个快照。说到底，可以获取系统中正在运行的进程信息，线程信息，等。

```

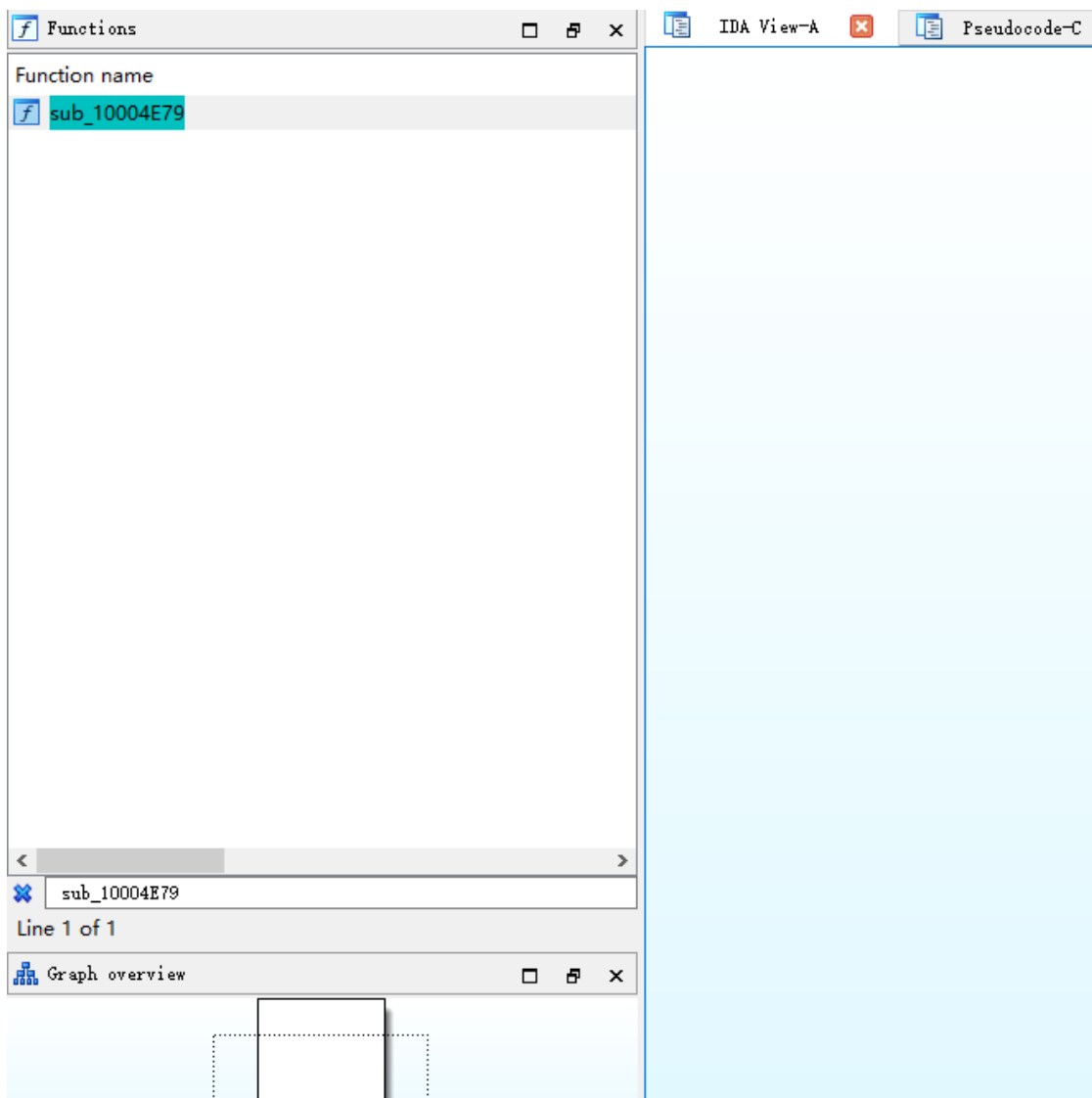
mov     ecx, 3FFh
lea     edi, [ebp+var_1630]
mov     [ebp+var_1634], ebx
push    ebx                ; th32ProcessID
rep stosd
push    2                  ; dwFlags
call    CreateToolhelp32Snapshot
cmp     eax, 0FFFFFFFFh
mov     [ebp+hSnapshot], eax
jnz     short loc_100066DF

```

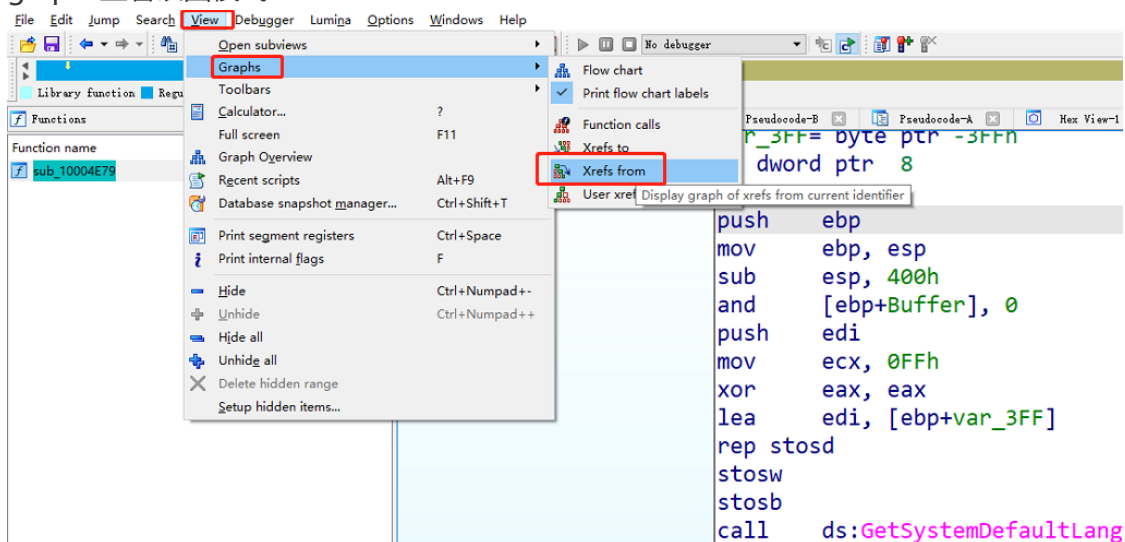
所以PSLIST该函数主要是为了获取进程列表信息。

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将会被调用？仅仅基于这些API函数，你会如何重命名这个函数？

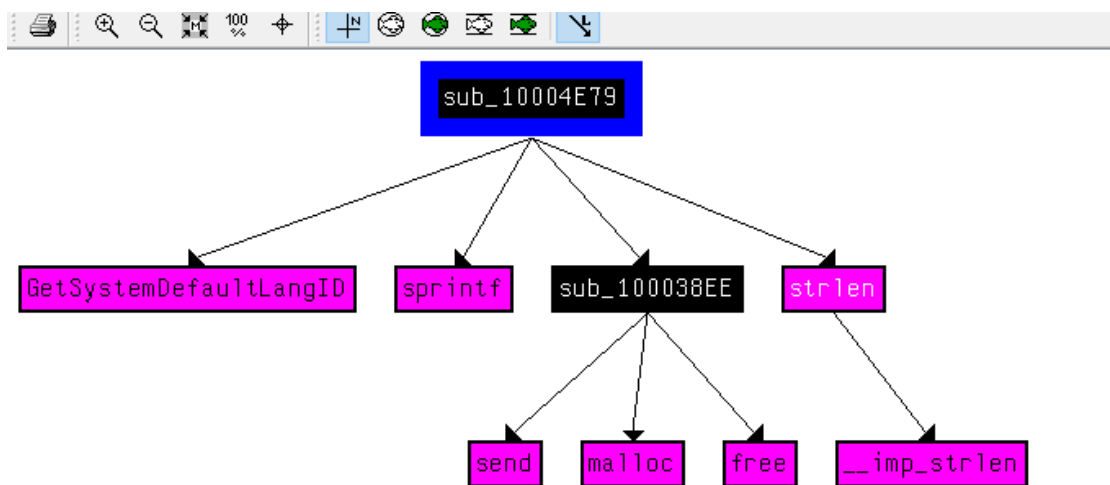
ctrl+f搜索该函数



graphs查看该图模式

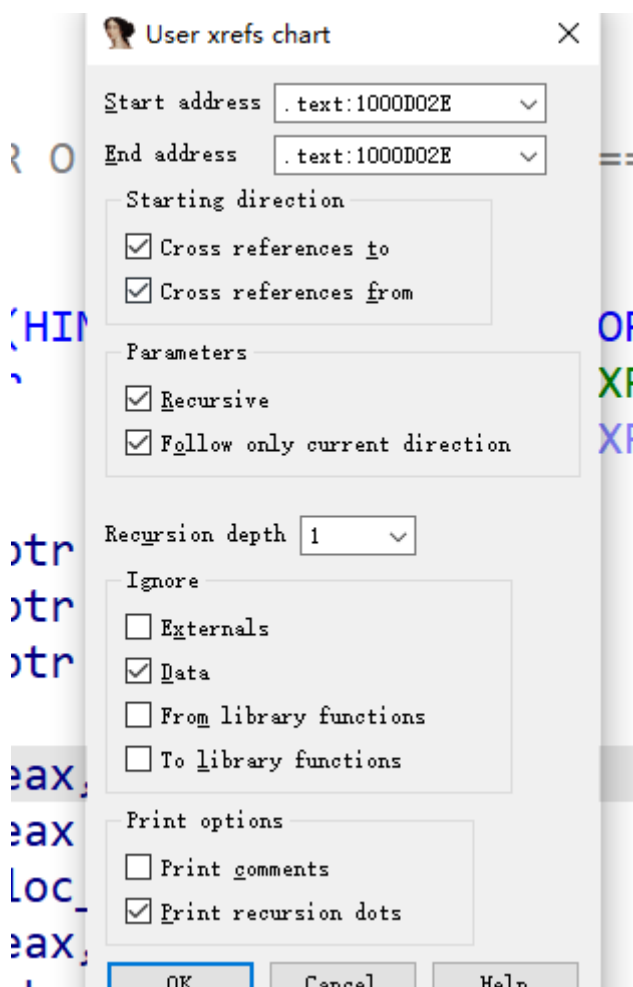


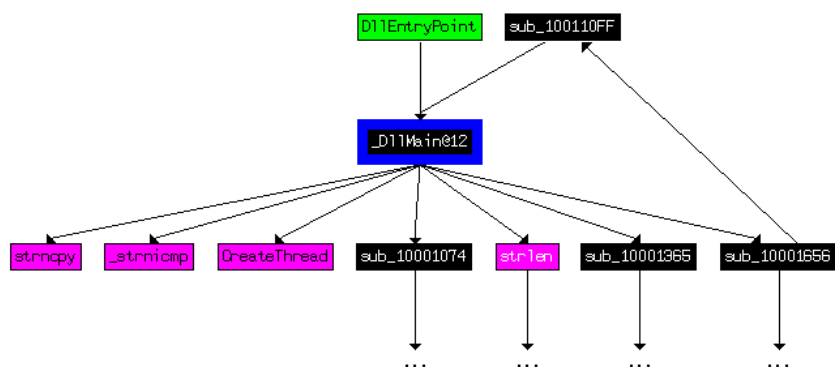
调用GetSystemDefaultLangID，获取该系统的默认语言，然后利用sent进行发送，所以该函数可命名为获取系统语言，get\_Language



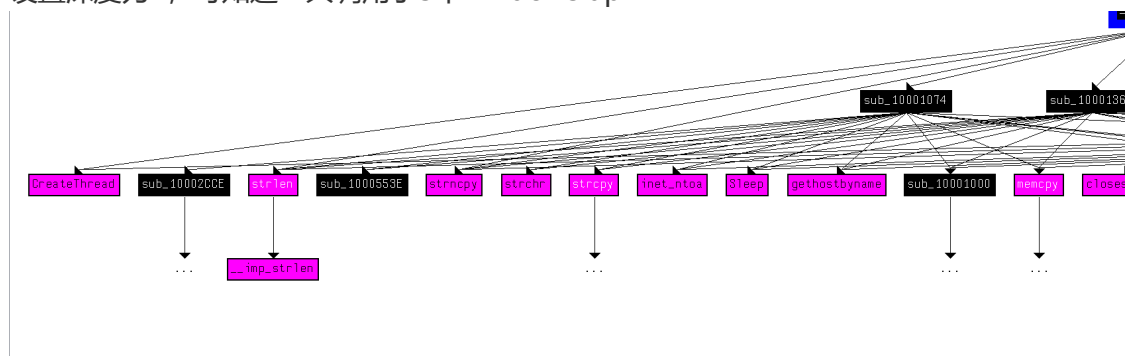
13、DllMain直接调用了多少个Windows API? 多少个在深度2时使用?

查找相关dllmain函数, , view->Graphs->Xrefs chart, 起始地址和结束地址都写DllMain的起始地址, 然后选取深度 (Recursion deptch) 为1, 然后绘图, 可发现系统总共调用了4个api接口





设置深度为2，可知道一共调用了3个windows api



14、在0x10001358处，有一个对Sleep(一个使用一个包含睡眠毫秒的参数的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？

首先eax赋值为字符串[This is CTI]30;

add eax, 0Dh eax偏移13个字符串为30

push eax eax入栈，也就是字符串30入栈

call ds:atoi 将字符串转化为整型30

imul eax, 3E8h imul函数为计算函数，3E8h为1000，计算为30乘以1000，也就是30000毫秒，最终休眠事件为30秒

```

Pseudocode-B  Pseudocode-A  Hex View-1  Structures  Enums
mov     eax, off_10019020 ; "[This is CTI
add     eax, 0Dh
push    eax              ; String
call    ds:atoi
imul    eax, 3E8h
pop     ecx
push    eax              ; dwMilliseconds
call    ds:Sleep
xor     ebp, ebp
jmp     loc_100010B4
sub_10001074 endp
  
```

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

G跳转到该函数，调用的3个函数依次如下：



6 protocol 常用的有 IPPROTO\_TCP 和 IPPROTO\_UDP, 分别表示 TCP 传输协议和 UDP 传输协议, 6为tcp

1 type 为数据传输方式/套接字类型, 常用的有 SOCK\_STREAM (流格式套接字/面向连接的套接字) 和 SOCK\_DGRAM (数据报套接字/无连接的套接字), 1为internet地址, 利用ip进行通信

2 af af 为地址族 (Address Family), 也就是 IP 地址类型, 常用的有 AF\_INET 和 AF\_INET6, 2为ipv4进行通信

```
.text:100016F5 mov     ebp, ds:closesocket
.text:100016FB
.text:100016FB loc_100016FB: ; CODE XREF: sub_10001656
.text:100016FB ; sub_10001656+A09↓j
.text:100016FB push    6 ; protocol
.text:100016FD push    1 ; type
.text:100016FF push    2 ; af
.text:10001701 call     ds:socket
.text:10001707 mov     edi, eax
.text:10001709 cmp     edi, 0FFFFFFFh
.text:1000170C jnz     short loc_10001722
.text:1000170E call     ds:WSAGetLastError
.text:10001714 push    eax
.text:10001715 push    offset aSocketGetlaste ; "socket() GetLas
.text:1000171A call     ds:__imp_printf
.text:10001720 pop     ecx
```

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?

结合上一题目, 利用rename进行重命名

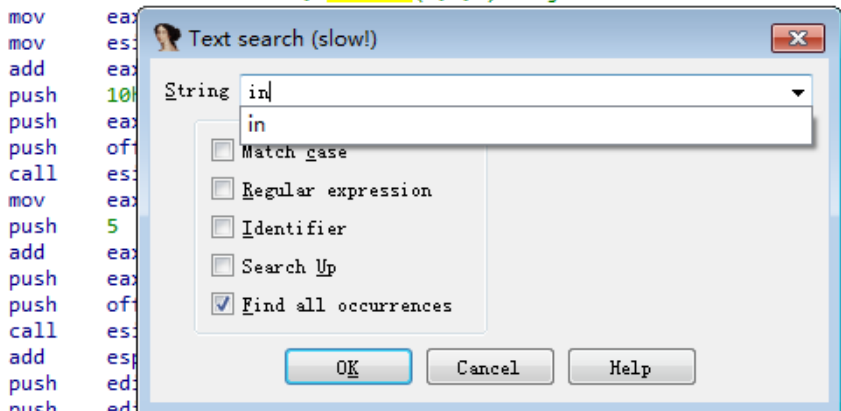
```
.text:100016F5 mov     ebp, ds:closesocket
.text:100016FB
.text:100016FB tcp协议: ; CODE XR
.text:100016FB ; sub_10001
→.text:100016FB push    6 ; protocol
.text:100016FD
.text:100016FD ip通信: ; type
.text:100016FD push    1
.text:100016FF
.text:100016FF ipv4: ; af
.text:100016FF push    2
.text:10001701 call     ds:socket
.text:10001707 mov     edi, eax
.text:10001709 cmp     edi, 0FFFFFFFh
.text:1000170C jnz     short loc_10001722
.text:1000170E
.text:1000170F
```

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?

搜索in关键字，就能把所有带in的关键字搜索出来

:\_1000D0C6:

```
; CODE XREF: DllMain(x,x,x)+6C↑j  
; DllMain(x,x,x)+85↑j
```



```
.text:10006045 sub_10006045 inc eax  
.text:10006049 sub_10006049 ; int __stdcall sub_10006049(int, size_t Size)  
.text:10006069 sub_10006069 push [ebp+arg_0] ; int  
.text:1000609F sub_1000609F ; _unwind { // loc_10015256  
.text:100060E3 sub_100060E3 ; _unwind { // loc_1001526A  
.text:10006119 sub_10006119 sub_10006119 proc near ; CODE XREF: InstallRT+171p  
.text:10006196 sub_10006196 sub_10006196 proc near ; CODE XREF: InstallRT+201p  
.text:100061DB sub_10006196 in eax, dx  
.text:1000620C sub_1000620C ; int __cdecl sub_1000620C(char *Format, char ArgList)  
.text:10006229 sub_1000620C call ds:vsprintf  
.text:10006234 sub_1000620C push offset aXinstallDll ; "xinstall.dll"  
.text:10006255 sub_1000620C call ds:fprintf  
.text:10006268 sub_10006268 ; int __cdecl sub_10006268(HANDLE TokenHandle, LPCSTR lpName, int)  
.text:100062D7 sub_10006268 call ds:_imp_printf  
.text:100062E9 sub_100062E9 ; int __cdecl sub_100062E9(SOCKET s)  
.text:10006313 sub_100062E9 push esi ; unsigned int  
.text:10006316 sub_100062E9 call ??2@YAPAXI@Z ; operator new(uint)  
.text:1000631B sub_100062E9 mov [esp+12A4h+lpProcName], offset aNtquerysystemi ; "Nt  
.text:10006440 sub_100062E9 inc eax
```

跟进这一行，发现存在一个cmp指令，通过按r进行转换可发现vmxh字样

```
.text:100061C7 mov eax, 564D5868h  
.text:100061CC mov ebx, 0  
.text:100061D1 mov ecx, 0Ah  
.text:100061D6 mov edx, 5658h  
.text:100061DB in eax, dx  
.text:100061DC cmp ebx, 564D5868h  
.text:100061E2 setz [ebp+var_1C]  
.text:100061E6 pop ebx  
.text:100061E7 pop ecx  
.text:100061E8 pop edx  
.text:100061E9 jmp short loc_100061F6  
.text:100061EB ; -----  
.text:100061EB loc_100061EB: ; DATA XREF: .rdata:s  
.text:100061EB ; except filter // owned by 100061C6  
.text:100061C6 push ebx  
.text:100061C7 mov eax, 'VMXh'  
.text:100061CC mov ebx, 0  
.text:100061D1 mov ecx, 0Ah  
.text:100061D6 mov edx, 5658h  
.text:100061DB in eax, dx  
.text:100061DC cmp ebx, 'VMXh'  
.text:100061E2 setz [ebp+var_1C]  
.text:100061E6 pop ebx  
.text:100061E7 pop ecx  
.text:100061E8 pop edx  
.text:100061E9 jmp short loc_100061F6  
.text:100061EB ; -----
```

查看该函数交叉引用，在每个引用的函数下面都会发现如下字符，Found Virtual

## Machine, Install Cancel, 说明确实存在虚拟机检测

```
ext:10006196 sub_10006196 proc near ; CODE XREF: InstallRT+204p
ext:10006196 ; InstallSA+204p ...
ext:10006196
ext:10006196 var_1C = byte ptr -1Ch
ext:10006196 ms_exc = CPPEH_RECORD ptr -18h
ext:10006196
ext:10006196 ; __unwind { __except_handler3
ext:10006196
ext:10006196 xrefs to sub_10006196
ext:10006196
ext:10006196 Direction Typ Address Text
ext:10006196 Do... p InstallRT+20 call sub_10006196
ext:10006196 Do... p InstallSA+20 call sub_10006196
ext:10006196 Do... p InstallSB+20 call sub_10006196
ext:10006196
ext:10006196 Line 1 of 3
ext:10006196 OK Cancel Search Help
ext:10006196
ext:10006196 and [ebp+ms_exc.registration.nlevel], 0
ext:10006196 push edx
ext:10006196
ext:10006196
ext:10006196 .text:1000D865 jnz short loc_1000D870
ext:10006196 .text:1000D867 call sub_10006196
ext:10006196 .text:1000D86C test al, al
ext:10006196 .text:1000D86E jz short loc_1000D88E
ext:10006196 .text:1000D870
ext:10006196 .text:1000D870 loc_1000D870: ; CODE XREF: InstallRT+1E1j
ext:10006196 .text:1000D870 push offset byte_1008E5F0 ; Format
ext:10006196 .text:1000D875 call sub_10003592
ext:10006196 .text:1000D87A mov [esp+8+Format], offset aFoundVirtualMa ; "Found Virtual Machine,Install Cancel."
ext:10006196 .text:1000D881 call sub_10003592
ext:10006196 .text:1000D886 pop ecx
ext:10006196 .text:1000D887 call sub_10005567
ext:10006196 .text:1000D88C jmp short loc_1000D8A4
ext:10006196 .text:1000D88E ;
ext:10006196 .text:1000D88E loc_1000D88E: ; CODE XREF: InstallRT+151j
ext:10006196 .text:1000D88E ; InstallRT+271i
```

18、将你的光标跳转到0x1001D988处，你发现了什么？

发现了一堆字符串

```
.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 ; <
.data:1001D9A4 db 3Bh ; ;
```

19、如果你安装了IDA python插件（包括IDA pro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脚本，（确定光标实在0x1001D988处。）在你运行

了这个脚本后发生了什么？

通过看脚本发现是与0x55进行异或处理，书中给出的脚本在进行加载时发现出错，提取所有的字符串

```
2D 31 3A 3A 27 75 3C 26 75 21 3D 3C 26 75 37 34 36 3E 31 3A 3A 27 79 75 26 21
27 3C 3B 32 75 31 30 36 3A 31 30 31 75 33 3A 27 75 05 27 34 36 21 3C 36 34 39
75 18 34 39 22 34 27 30 75 14 3B 34 39 2C 26 3C 26 75 19 34 37 75 6F 7C 64 67 66
61
```

通过异或再将ascii转换为字符，

```
ecoded_byte = 0x31 ^ 0x55
```

```
zifu=chr(ecoded_byte)print(zifu)
```

最终可获取解密字符：

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

由于对ida的脚本编写不是很熟悉，没有进行批量化脚本完成。

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

按A即可转化为ascii字符

21、使用一个文本编辑器打开这个脚本。它是如何工作的？

脚本内容如下：

```
sea = ScreenEA()

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

获取光标所在位置，也就是初始位置，然后在0x00,0x50之间逐一进行循环相加，正好循环80次，获取所有的字符，然后在分别与0x55进行异或，最终获取结果。

小结：主要是参考书上的课上问题进行一步步分析，也参考了很多网上的资料，通过分析确实对ida有了更进一步的认识，对木马的分析也有了一些新的认识，但是并没有对该恶意文件进进行全面分析，还是存在很多问题的，还需要进一步学习。