

**恶意代码分析与防治课程实验报告**

**实验四：利用ida pro分析**

****

学 院 网络空间安全学院

专 业 信息安全

学 号 2111033

姓 名 艾明旭

班 级 信息安全一班

1. **实验目的**

**学会使用ida在winxp系统当中的基本使用方法和常见的使用ida分析程序的各个函数端点的原则，常见的定位功能，函数的各个地址和端口的位置会存在在ida的各个地址当中，ida的地址有存在函数的端口，也可以进行方便的跳转。**

**在观察函数的交叉引用时，我们可以通过函数的交叉引用位置，找到某个函数进行了怎样的调用，查看其是否具有一些产生连接网络，创造病毒等等的功能。**

**学会在ida当中进行程序分析和程序流程的建立，并且观察其是否具有连接到系统当中相应的库的功能。**

**尝试利用ida反汇编，idac和ida python等等一些的工具对一些复杂的汇编语言片段进行烦汇编之后，更加方便我们去理解程序当中各个流程的复杂关系。**

1. **实验原理**
   1. **实验环境**

Windows xp，VMWARE，Windows11

* 1. **实验工具**

**Ida pro 6.6 ida python**

* 1. **原理**

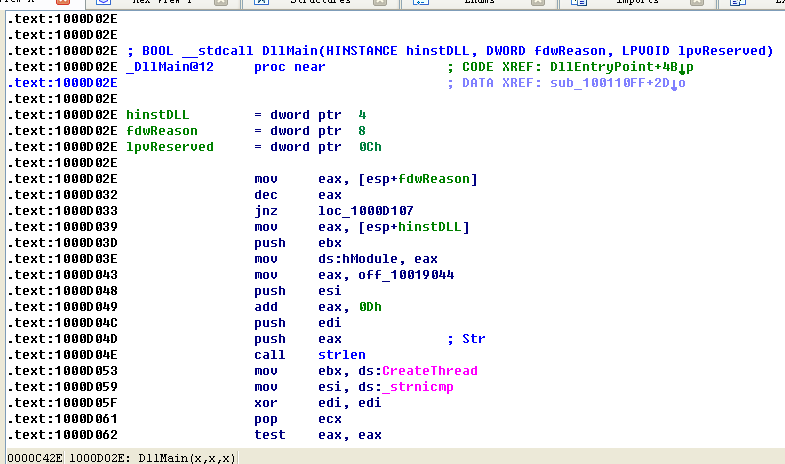
**IDA Python是基于IDA Pro的Python扩展，它允许用户通过编写Python脚本与IDA Pro进行交互和自动化操作。**

**扩展性：IDA Pro是一款反汇编和静态分析工具，IDA Python充分利用了Python的灵活性和强大的标准库，提供了一组API和对象来与IDA Pro进行交互。用户可以通过编写Python脚本来扩展IDA Pro的功能，实现自定义的反汇编、分析、导出等操作。**

**提供API：IDA Python提供了一组完整的API，用于操作和访问IDA Pro的各种特性和数据结构。这些API包括函数、变量、指令、图形界面、数据库查询、导出和导入等功能。用户可以通过调用这些API来实现各种自动化任务，例如自动识别函数、修改指令、导出数据等。**

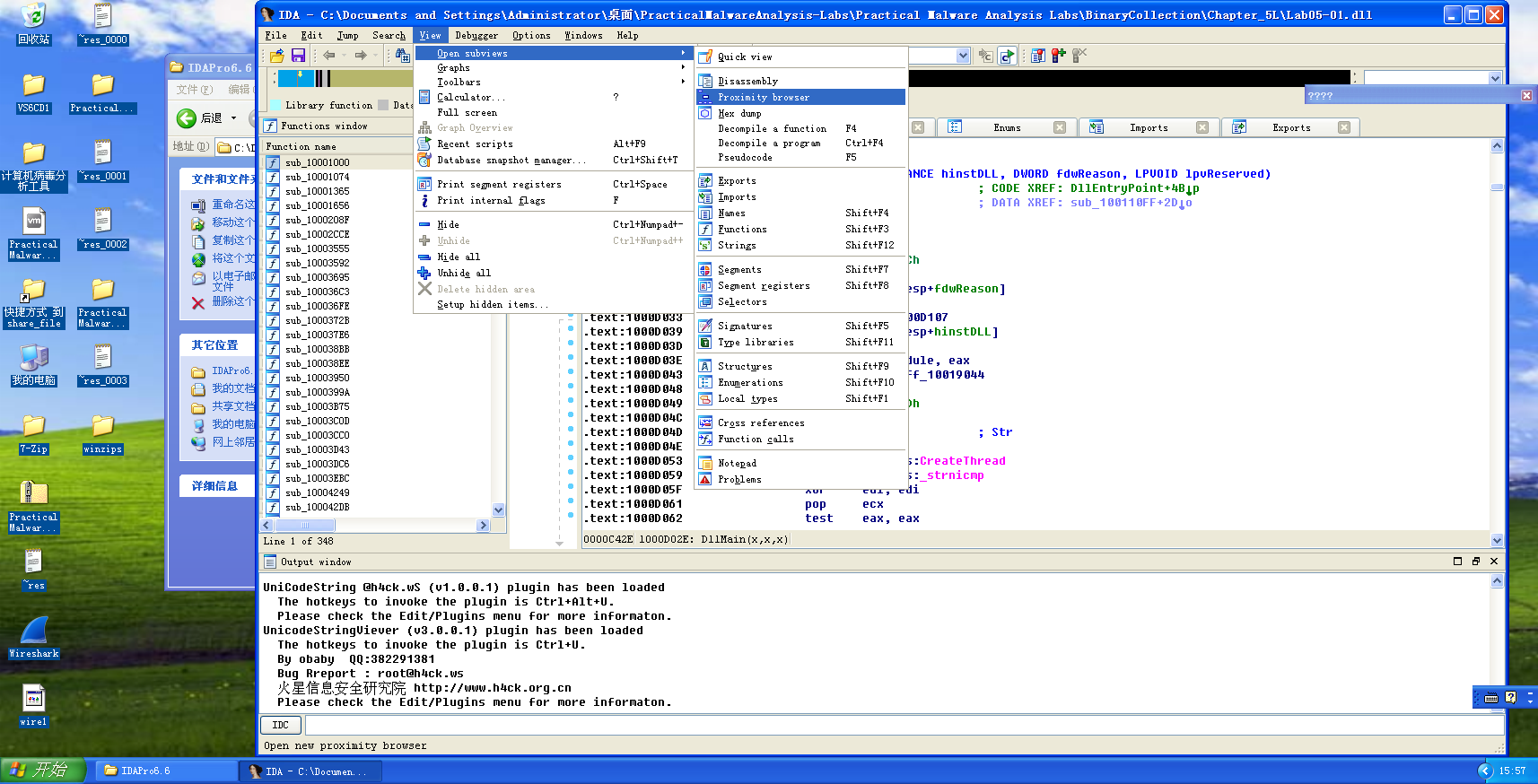
**事件驱动：IDA Python还支持事件处理机制，允许用户注册和处理特定的IDA事件。当IDA Pro发生某些事件时，例如载入二进制文件、分析完成、用户交互等，用户可以编写回调函数来响应这些事件，执行相应的操作。**

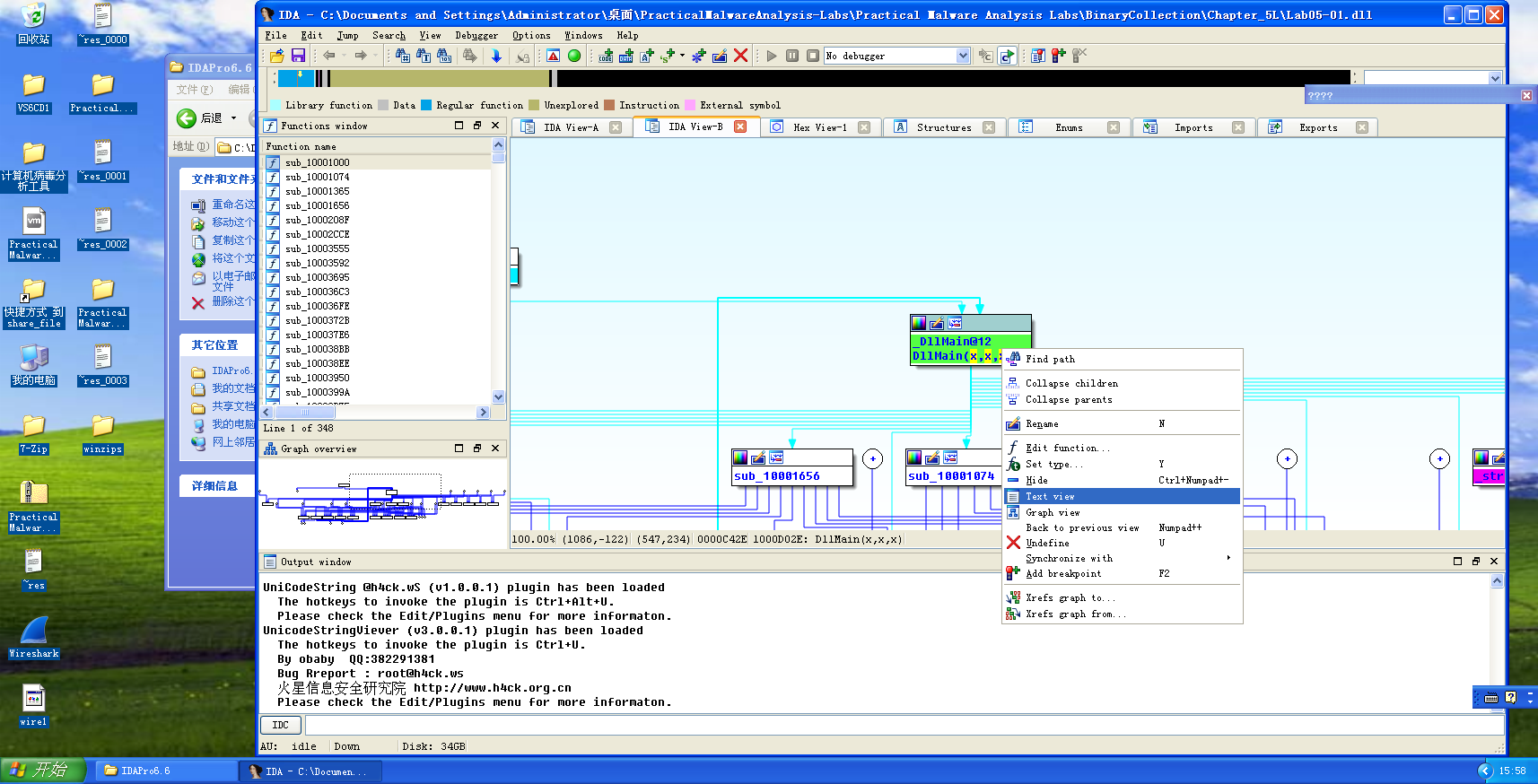
**脚本界面：IDA Python提供了一个交互式的Python解释器，可以在IDA Pro中直接编写和执行Python代码。用户可以通过这个Python解释器与IDA Pro进行实时的交互，进行数据查询、运行脚本等操作。**

1. **实验过程**
2. **使用IDA打开后，鼠标所在位置即为目标地址**

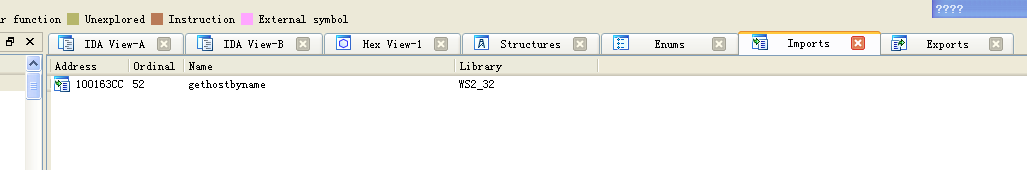
或者点击view->open subviews->proximity browser进入到下图的程序流程图界面，之后右键点击之后选择text view即可到达指定位置，回到了dllmain的开头地址。

或者选中函数列表，Ctrl+F，搜索DLLMain，双击该函数，IDA自动定位到该函数的内存地址处。地址是1000D02E

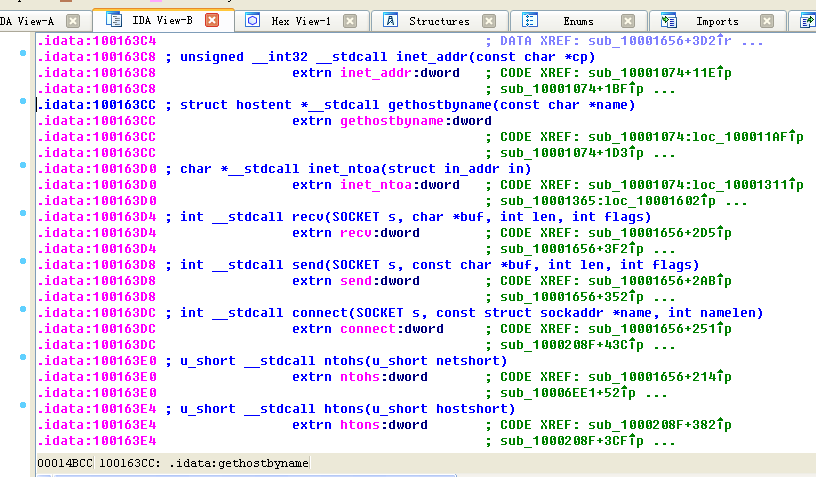




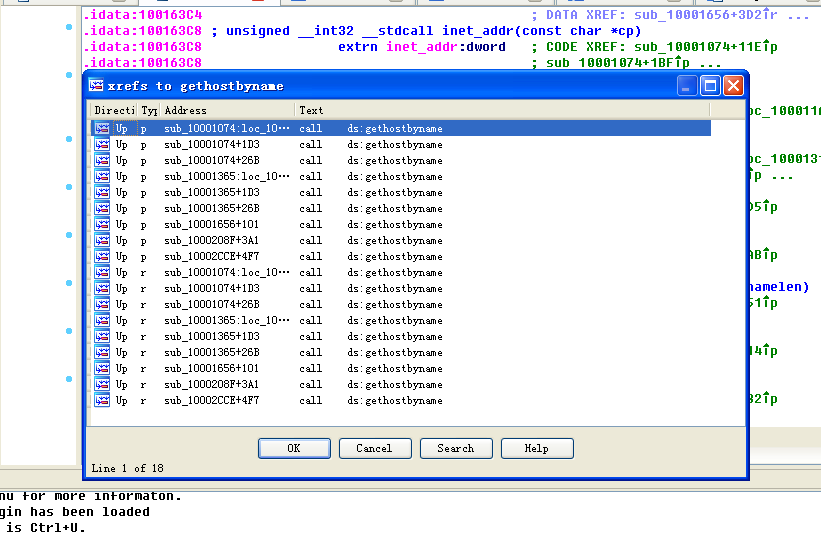
**2.在imports窗口直接搜索函数名，双击之后进入到相应的函数当中。**



在.idata节的0x100163CC处。



**3.在第二步，双击gethostbyname，链接到该函数处，按“ctrl+X”键，获得交叉引用次数。除去重复的，由此可知有5个函数调用了它总共9次。**

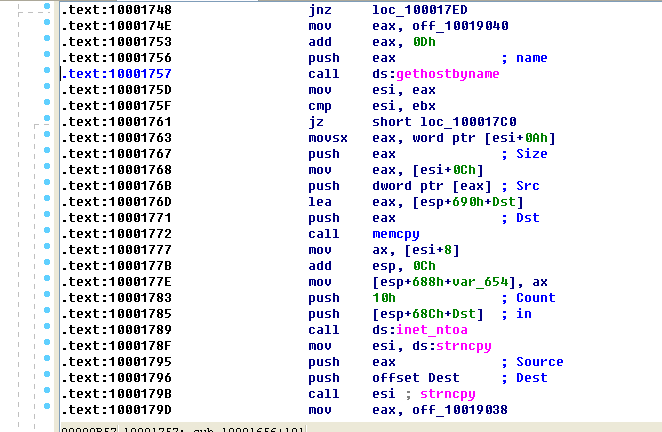


**4.将外部引用窗口关闭，然后按G，直接定位0x10001757跳转过去**

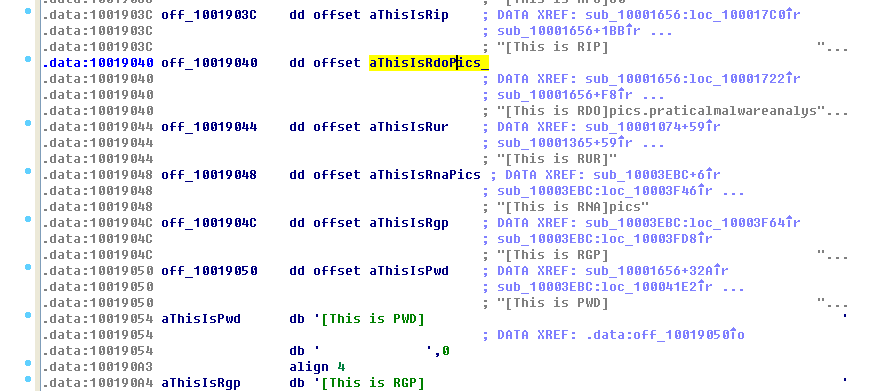
先跳转到10001757处，在这个汇编代码先将10019040中的内容存放到中寄存器eax中，’

看调用，使用了call函数调用，百度了一下，call在操作栈，把栈顶的数据作为参数传给函数，往上看发现一个push命令，将eax值压入栈中，在找eax值，网上是个add指令，加法指令，加上0D，十进制为13，为什么要加十三呢？在往上看，发现一个mov指令，将off\_10019040赋值给eax，直接双击该参数，就跳转到了这个位置。

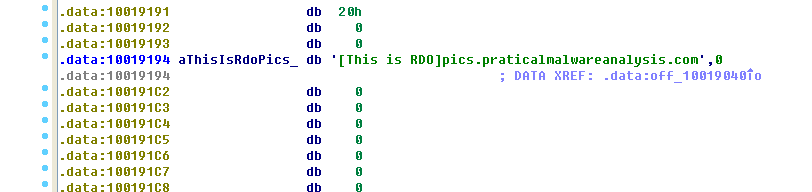
发现存在pics.praticalmalwareanalys的字样，然后双击aThisIsRdoPics\_，发现了praticalmalwareanalysis.com的域名信息。



发现一个aThisIsRdoPicsP的东西。再双击该该变量，直接获取字符串。

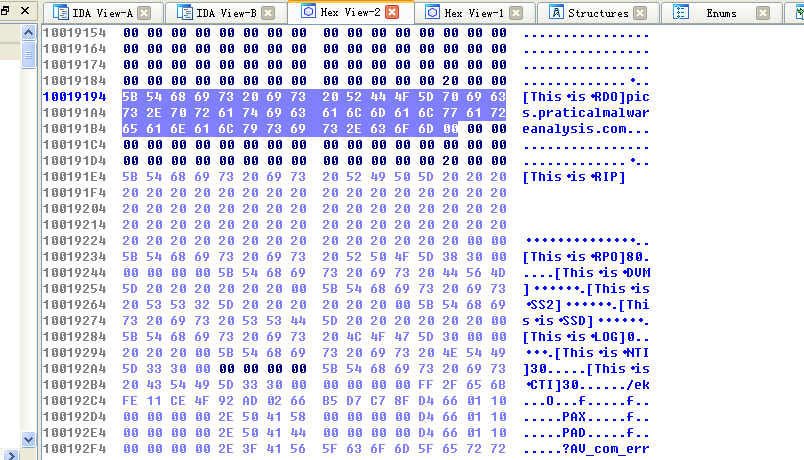


发现存在pics.praticalmalwareanalys的字样，然后双击aThisIsRdoPics\_，发现了praticalmalwareanalysis.com的域名信息。

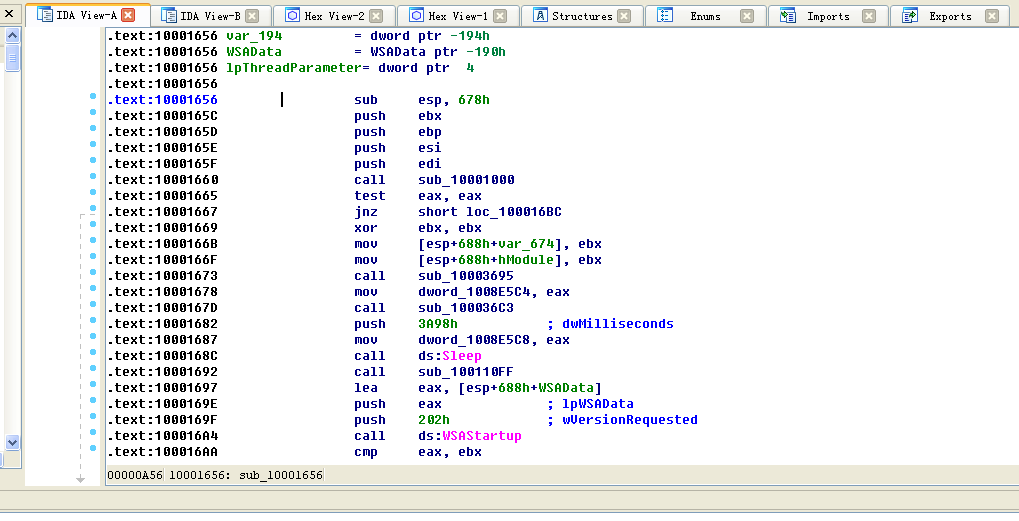


用hex窗口打开aThisIsRdoPics\_

发现前13个字节就是 [This is RDO],也就是说，后来令eax加上0Dh，就是将eax指向了praticalmalwareanalysis.com，然后再将eax压入栈作为gethostbyname的参数。所以位于0x10001757处的对gethostbyname的调用访问了praticalmalwareanalysis.com



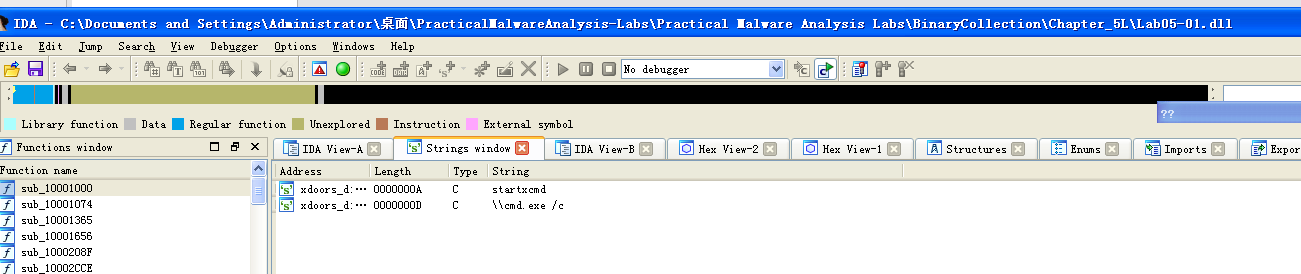
**5.直接跳转到此处，**跳转方法此处略过，直接看IDA给你是被出来的东西，根据书上所说，IDA会把局部变量标记为var前缀，存在负偏移，因此数一数有多少个var前缀的就行和有负偏移的个数，所以有23个。



**6.借用5题图，结合书本所说，**参数一般使用arg前缀，存在正偏移，所以只有1个参数（lpThreadParameter）。

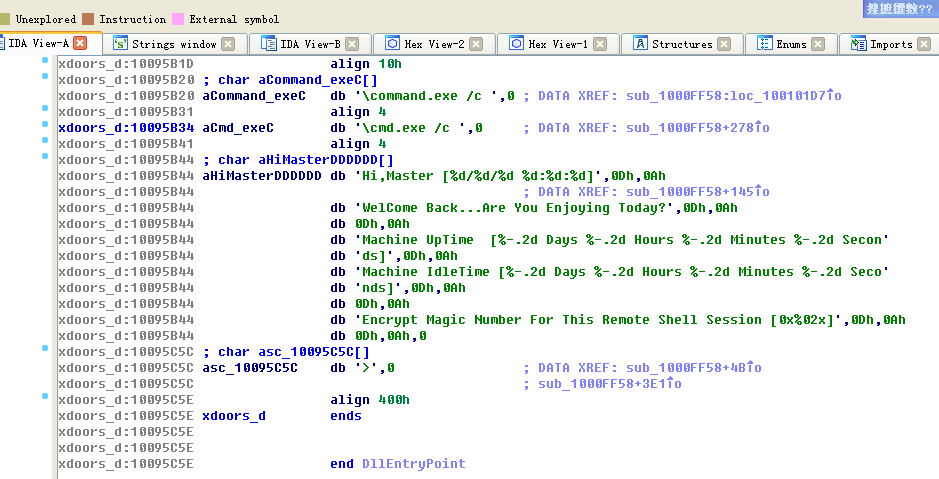
**7. 使用String窗口，来在反汇编中定位字符串\cmd.exe /c。它位于哪？**

view -> opensubview -> strings，打开strings窗口，然后Ctrl+F，搜索\cmd.exe /c。其地址为10095B34。

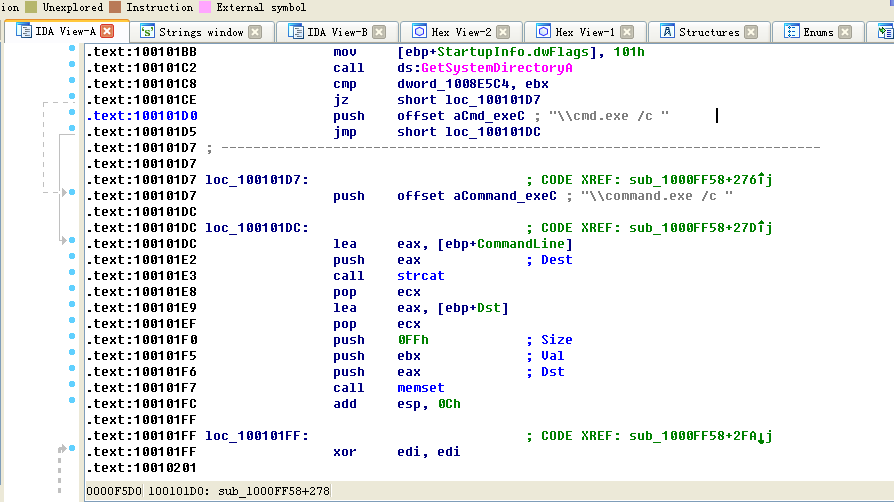


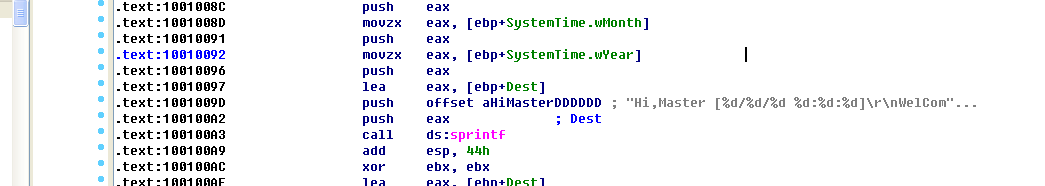
**8. 在引用\cmd.exe /c的代码所在区域发生了什么。**

双击该选项，跳转到代码块。



但是这只是个变量，没有任何代码，它是被其他地方引用了，再双击后面的自动注释，跳转到这个字符串被引用的地方。当然可能好多地方都引用了，所以选中变量名，按“X”，发现只有一个地方被引用了，就直接跳转吧。



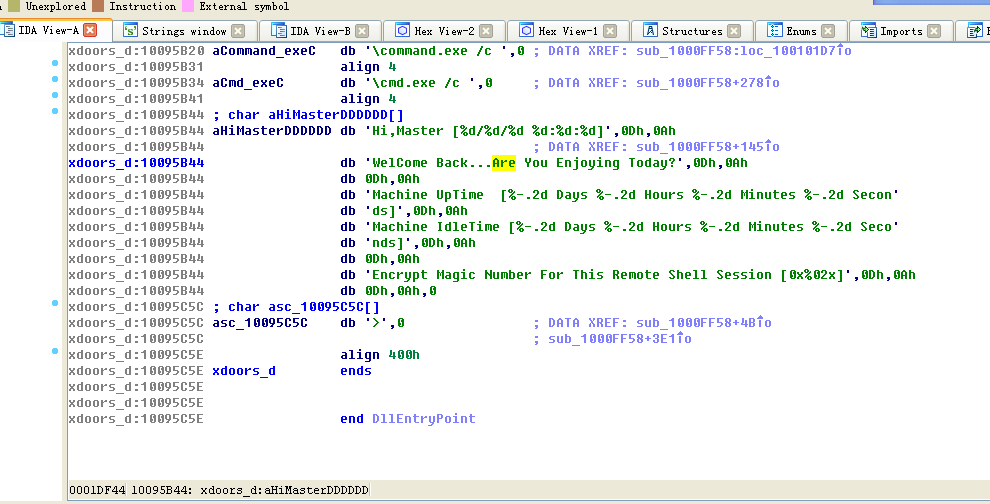


看到再100101D0处该字符串被压入栈

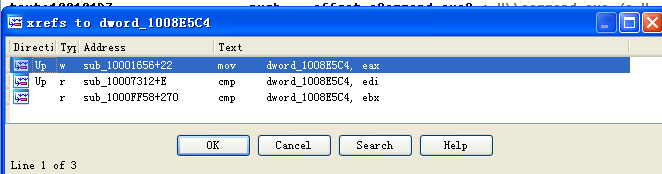
往上翻发现在该字符串被引用之前还有一个字符串

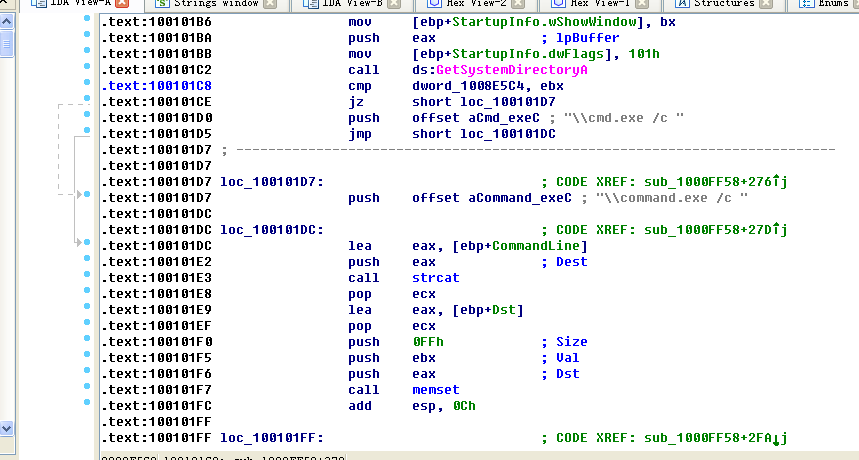
点开对应的字符串

图中字符串表示为远程shell会话解密一段数字，即开启一段远程session会话。

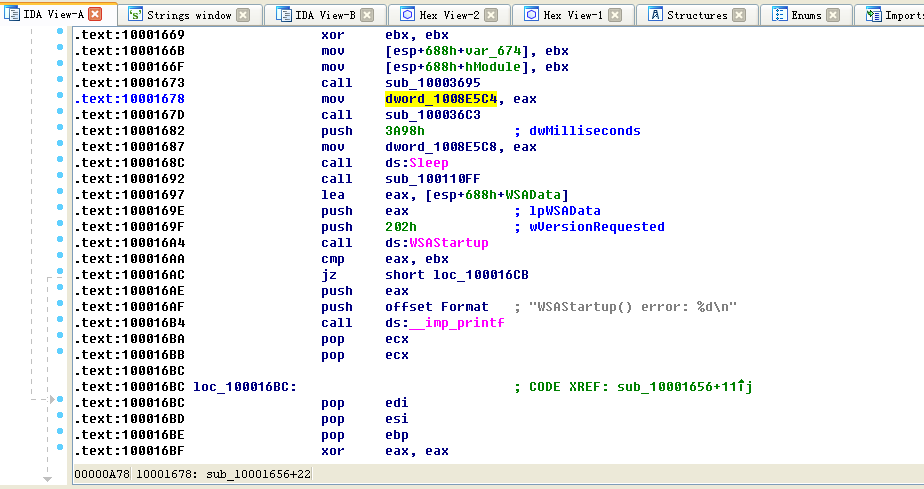


**9. 在同样的区域里，在0x100101C8处，看起来好像dword\_1008E5C4是一个全局变量，它帮助决定走那一条路径？恶意代码是如何设置dword\_1008E5C4的？（提示：使用dword\_1008E5C4的交叉引用）**

先跳转到这个位置，然后选中这个变量，然后按“Ｘ”查看交叉引用，发现有3处交叉引用。



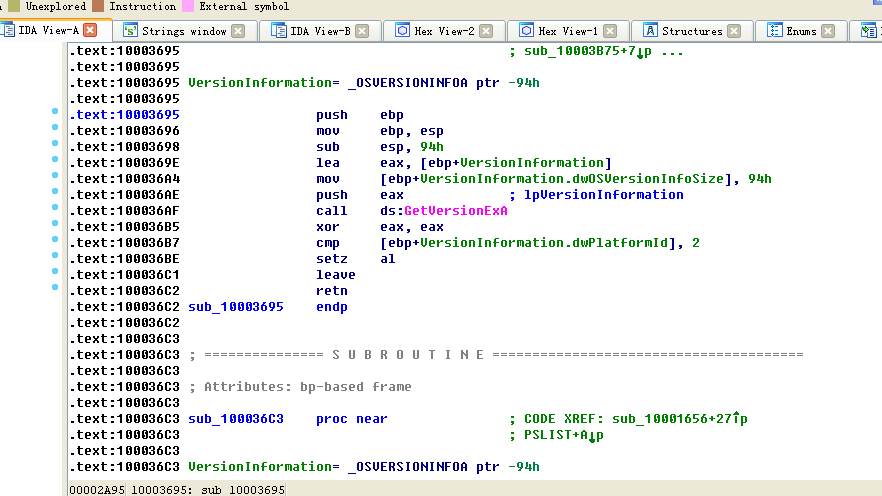
根据direction，双击第一个，进入第一次被引用的地方（从书本后面的答案来看，跳到这里的目的也是因为只有在这里被修改过）。然后向上看代码，第一次被引用的地方向上不多就到了函数的起始部分。



从其指令来看，是把eax的值给了dword\_1008E5C4，所以是把sub\_10003695这个函数返回值赋给了dword\_1008E5C4，先不看它如何被赋值，先看这个值帮助决定走哪条路。直接看代码还是挺累的，在查看其余两处引用后，第三个dword\_1008E5C4决定是否走command.exe

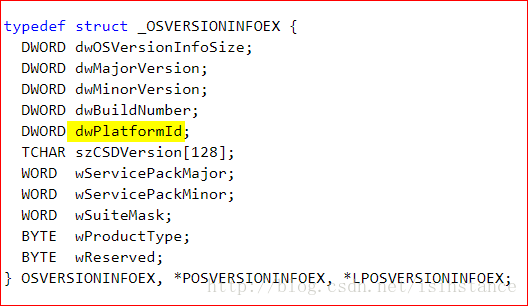
发现进行写操作的只有第一条

双击10001656+22，发现它将eax的值赋给了dword\_1008E5C4，而EAX是前一条指令函数调用的返回值。因此双击前面的函数调用sub\_10003695查看eax的返回值



可以看到该函数包括了一个GetversionExA的调用,用于获取当前操作系统版本的信息,之后再将VersionInformation.dwPlatformId与2比较来决定寄存器al的值即eax的值。

其中2代表WIN32\_NT系统，也就是说，该处的函数会检查一下是不是win32系统来决定eax的值，之后再存放入dword\_1008E5C4中，而dword\_1008E5C4的值也就是0或1。



我们可以来查看一个OSVERSIONINFOA的结构

setz意思是当ZF标志被设定时，AL寄存器设1

因为刚刚我们cmp了两个数，所以如果两个数相同，ZF=1，然后setz，AL被设置为1，反之不相同的话，AL被设置为0(AL是EAX的低8位，对应的AH是EAX的高8位)

一般来说，会运行这个机器的都是上面那几种windows机器，所以这里比较一般都是会相同的，所以，AL被设置成了1，然后就是用retn返回了eax中的值

但是为什么书上说的是返回的是1，因为我们前面执行过

eax现在是被异或都成了0（eax有16位）

我们现在分析一下，01234567代表的是高8位(ah)，剩下的代表的是低八位(al)

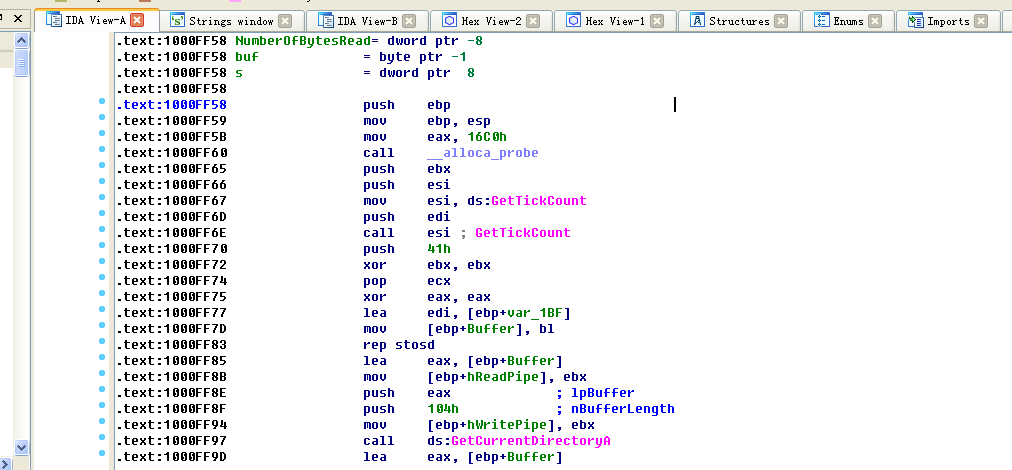
先是xor eax, eax 这个运行完之后的eax 然后我们接着运行

cmp [ebp+VersionInformation.dwPlatformId], 2 setz al 1 2

之后的eax，89ABCDEF代表的al被置为了1，其余的ah保存不变

然后这个值其实就是十进制的1，所以sub\_10003694的返回值是1，于是mov dword\_1008E5C4, eax 1。最后的dword\_1008E5C4的值就被赋成了1。

所以这个全局变量在程序运行的时候一直保持的是1。

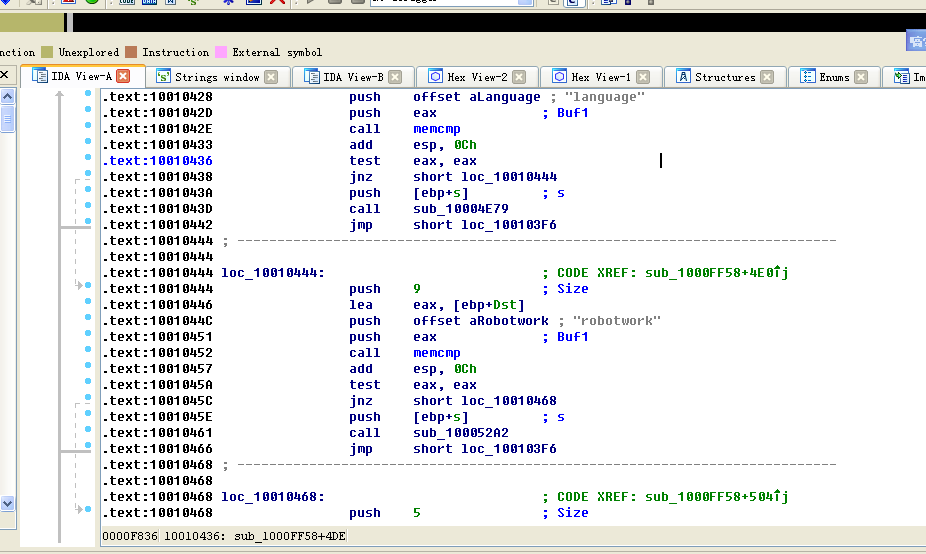


**(10)在位于0x1000FF58处的子过程中的几百行指令中，一系列使用memcmp来比较字符串的比较。如果对robotwork的字符串比较是成功的（当memcmp返回0），会发生什么？**

跳转到0x1000FF58处向下翻，一开始是比较了quit和eax的值，因为这两个值被压入了栈中

然后我们找到robotwork，慢慢往右下角拖就是了。

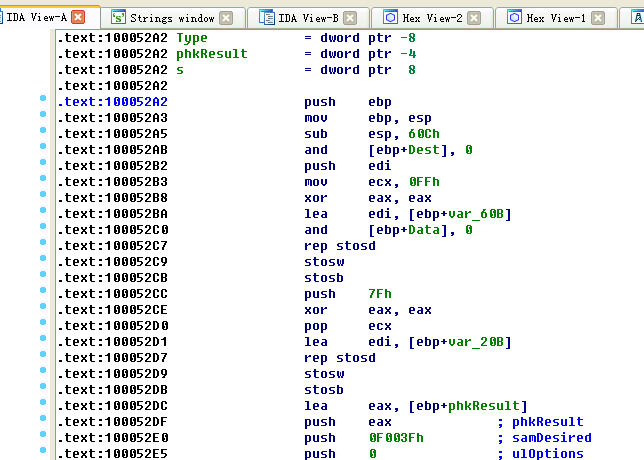
它首先压入了一个robotwork字符串指针，然后压入了eax，然后call memcmp，如果两个数相同，返回0，然后add eap, 0Ch 这里0Ch是12d，也是4(字节)\*3(个)，因为push后面跟的是立即数，所以一个数占4字节，然后offset也是4个字节，所以，一开始的push 9，和后面的两次push，加起来一共是3次，所以这里回收了这3个一共12字节的空间 test eax, eax



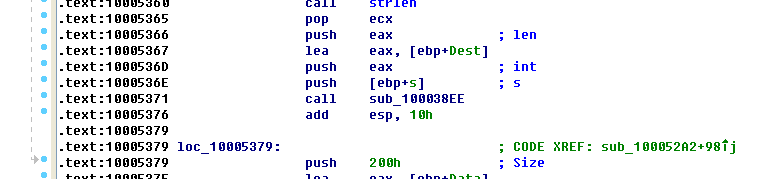
如果eax为0，则ZF置为1，JZ跳转，eax为0说明前面的memcmp比较的结果是相同，也就是如果前面两个数相同，则JZ跳转，JNZ不跳转

然后问题是当字符串比较成功，memcmp返回0

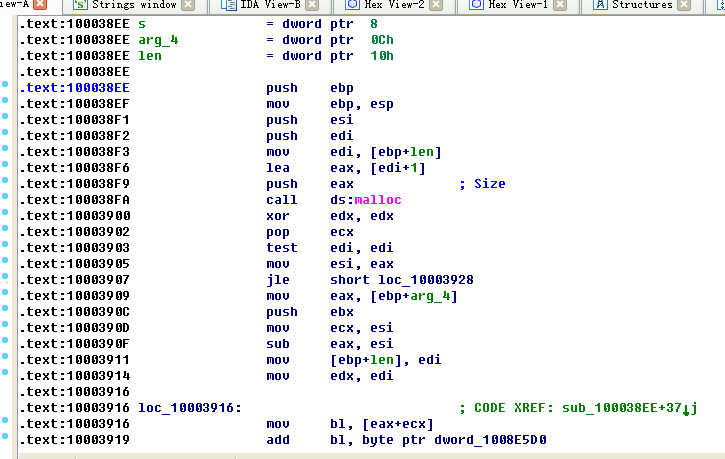
JNZ不跳转，程序继续按从上到下的顺序执行，下面要执行的就是



往下翻发现调用了sub\_100038EE函数



双击进入sub\_100038EE函数查看，发现其调用了了malloc函数创建了内存空间，然后又调用了send函数，最后调用了free函数释放内存空间，所以猜测可能是对注册表SOFTWARE \Microsoft\Windows\CurrentVersion进行了修改或查询操作，然后再将结构发送出去。



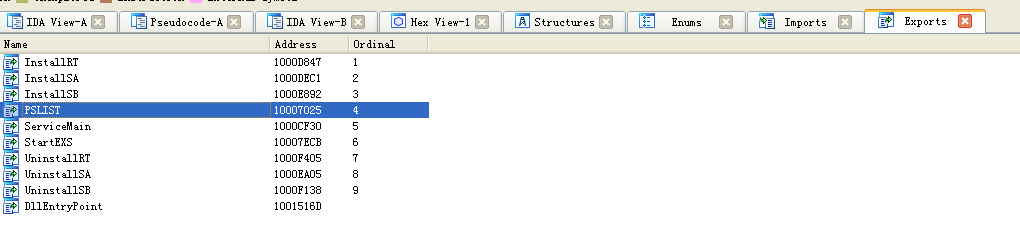
为了更直观详细的查看sub\_100052A2函数到底做了什么，我们返回sub\_100052A2函数内，按下F5查看伪c代码

发现首先用RegOpenKeyExA函数打开了注册表HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion，然后查询了aWorktime和aWorktimes键也就是worktime和worktimes键的值，然后调用了sub\_100038EE函数，将这些信息发了出去。

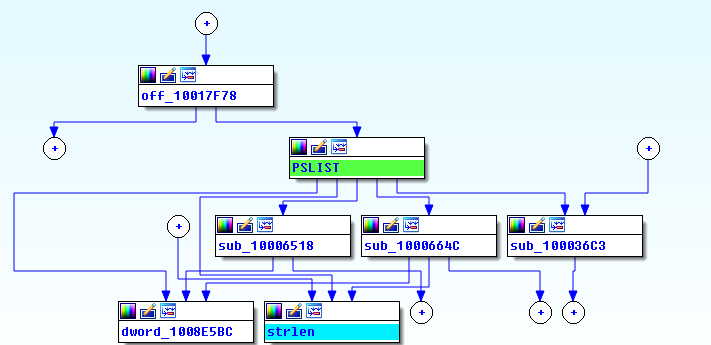


**(11)PSLIST导出函数做了什么？**

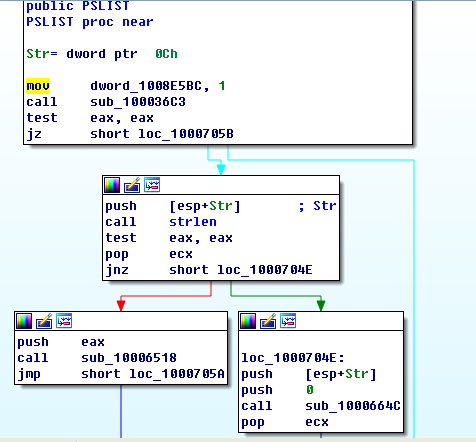
进入导出列表窗口

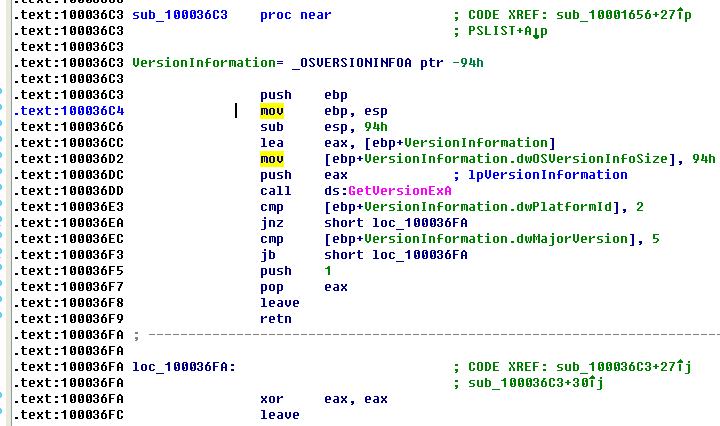


双击PSLIST函数，打开他的proximity browser看到它的汇编代码，首先调用sub\_100036C3函数，如果返回值为1则往下执行函数sub\_10006518和sub\_1000654C。

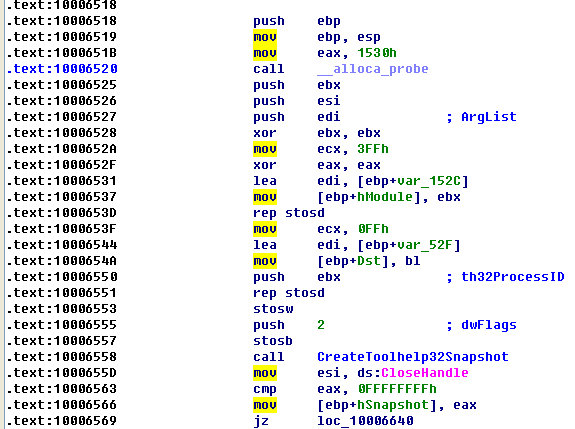


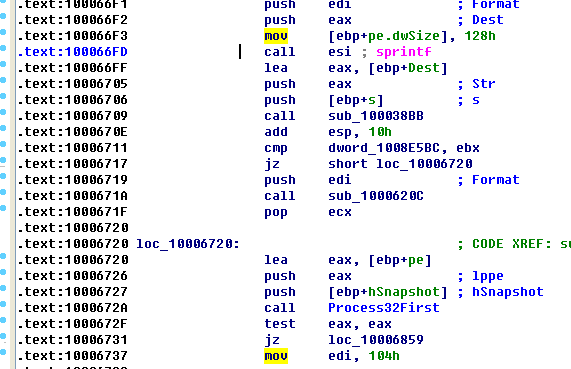
首先进入到sub\_100036C3函数中查看，并按F5查看伪c代码，可以看到sub\_100036C3函数做出了一个判断，判断系统是否是win32且版本大于win2000，如果是则返回1



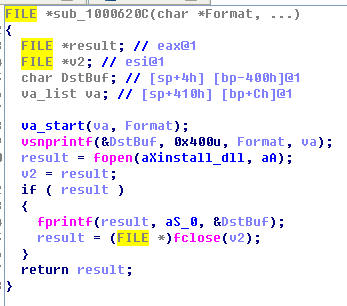


我们进入到sub\_10006518和sub\_1000654两个函数中查看





发现它们都调用了CreateToolhelp32Snapshot函数和sub\_1000620C函数，CreateToolhelp32Snapshot函数就是获取主机的进程信息，而sub\_1000620C，我们点进去观察伪C代码发现它就是将查询到的进程信息写入到一个文件中。



所以PSLIST导出函数的作用就是搜寻主机内的进程并把它写入到一个文件中形成一个进程列表。

**(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？**

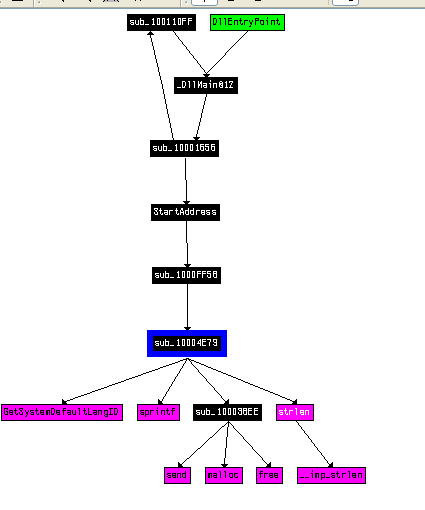
跳转到该函数处，选中sub\_10004E79，然后点击view->open subview->User xrefs chart生成交叉引用图



当进入这个函数时GetSystemDefaultLangID、sprintf、strlen这三个API被调用。

该函数又调用了sub\_100038EE函数，而sub\_100038EE函数又调用了send，malloc，free，\_\_imp\_strlen函数。

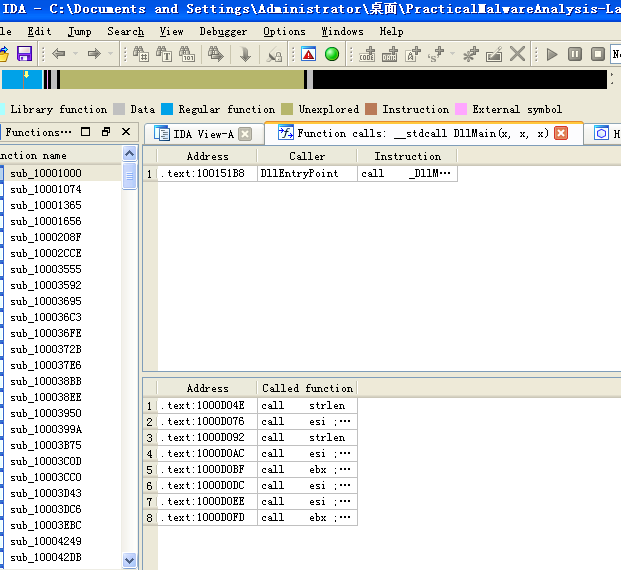
由于该函数调用了GetSystemDefaultLangID函数，又间接调用了send函数，所以猜测该函数会获取系统默认语言ID后，通过send函数发送远程连接的对方，所以该函数可以取名GetSystemLanguageID。



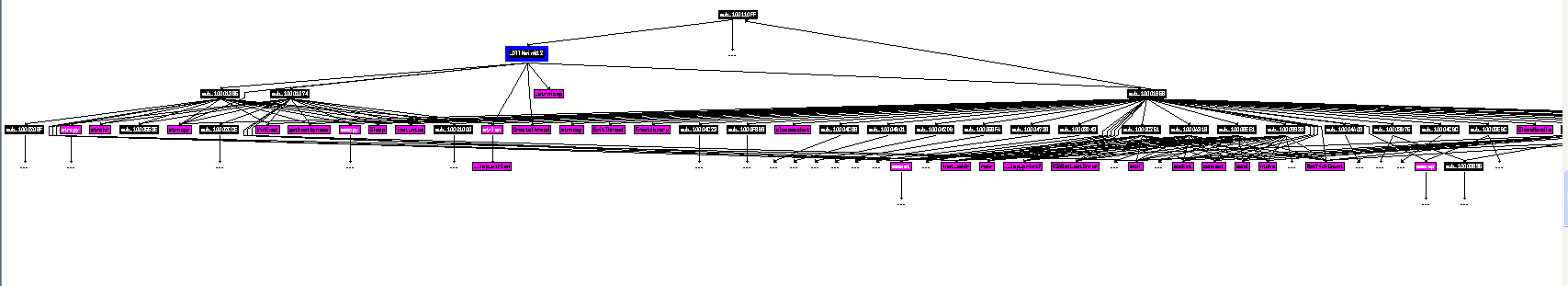
**(13)DllMain直接调用了多少个Windows API？多少个在深度为2时被调用？**

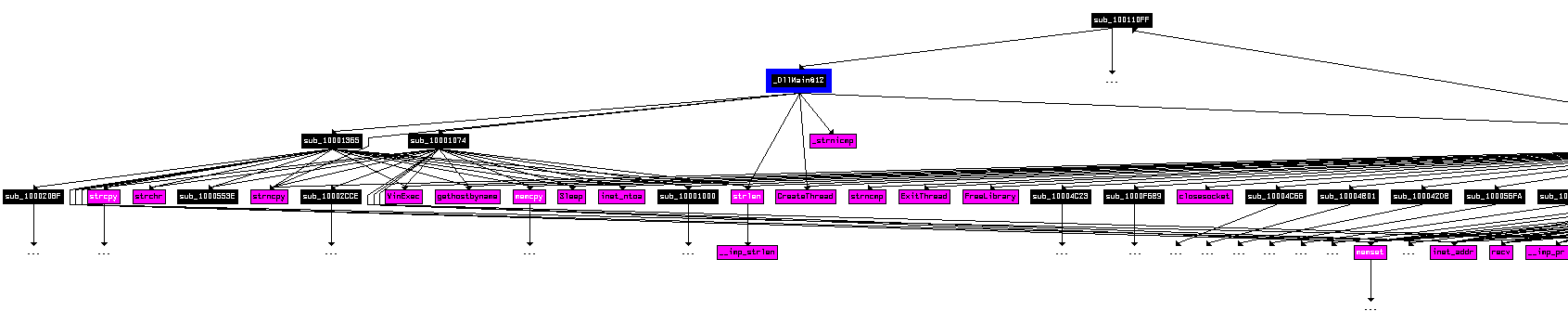
跳转到DLLMain函数处，选中DLLMain函数，点击view→Subviews→Function Calls查看函数调用窗口可以看到直接调用了strlen、strnicmp、CreateThread、strncpy这4个Windows API。

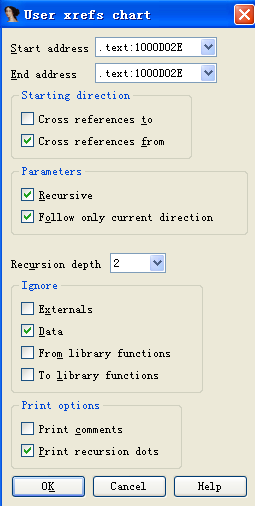
打开User xrefs chart,勾选掉Cross，references to，并且将深度设为2



放大后查看，大约有31个API在深度为2时被调用



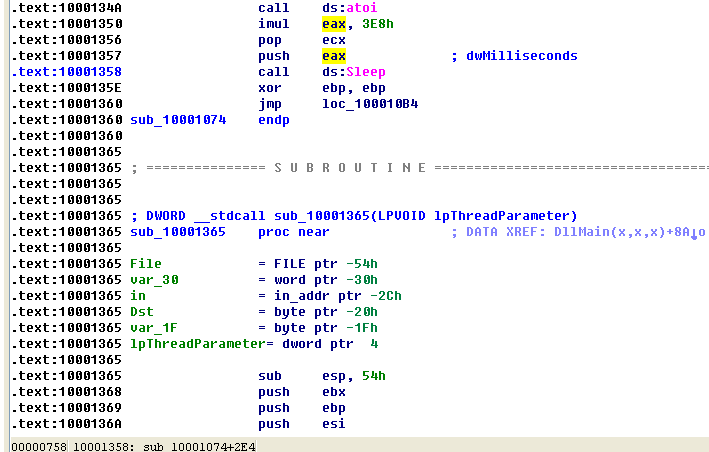


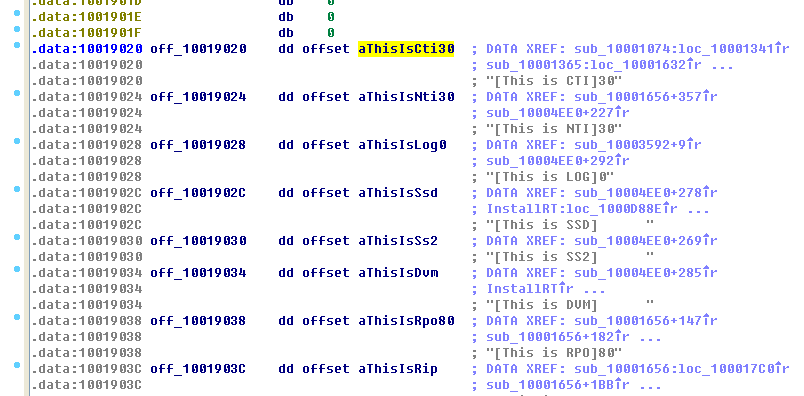


**(14)在0x10001358处，有一个对Sleep（一个使用一个包含要睡眠的毫秒数的参数的API函数）的调用。顺着代码向后看，如果这段代码执行，这个程序会睡眠多久？**

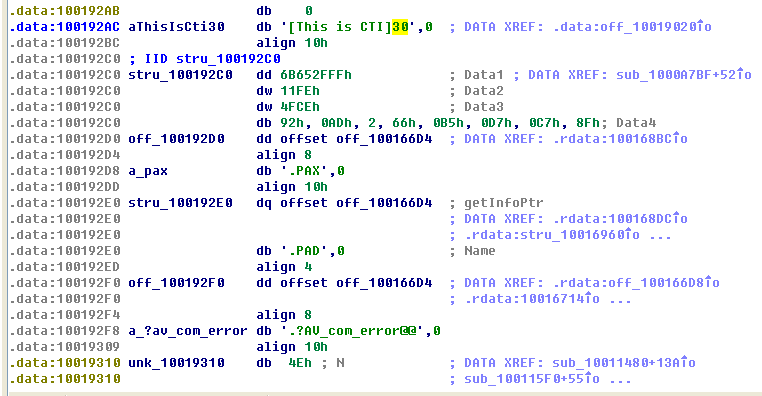
跳转到0x10001358处，放下在调用Sleep前先将eax压入了栈作为Sleep函数的参数，也就是睡眠时间，所以往前看找到eax的值，即为睡眠时间。

从中可以看出，在这段汇编代码中，先将off\_10019020存放到了eax中，我们双击进入off\_10019020，查看其内容





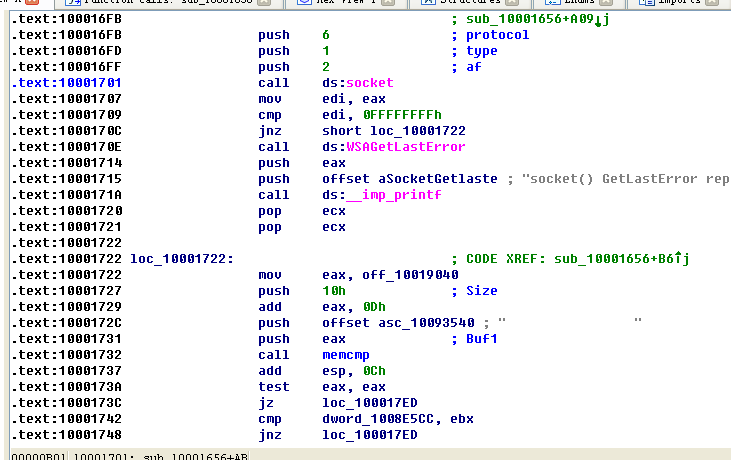
发现了[This is CTI]30的字样，双击进入aThisIsCti30，发现它定义的值为[This is CTI]30



在Sleep那段汇编代码中又将eax加了0DH，也就是13字节，即对应信息为[This is CTI]，也就是eax指向了30紧接着将eax压入栈，调用了atoi函数将eax的值化成了整数，也就是说eax的值为30，然后imul指令又将eax乘以3E8h，也就是30\*1000=30000ms=30s，所以sleep函数会使程序睡眠30s。

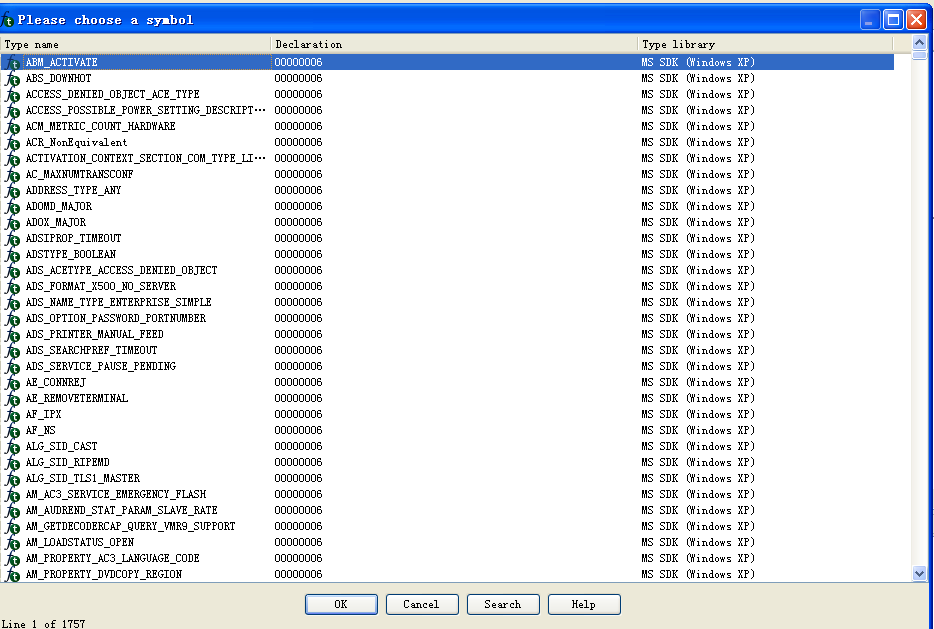
**(15)在0x10001701处是一个对socket的调用，它的3个参数是什么？**

跳转到0x10001701处，发现在调用socket函数前先将6，1，2这三个数压入了栈，故它的三个参数为6，1，2

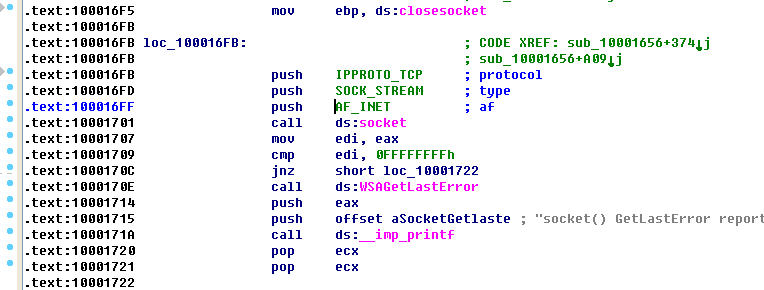


**(16)使用MSDN页面的socket和IDA Pro中的命名符号常量，你能使参数更有意义吗？修改后，参数是什么？**

右键单机这三个参数，选择 use standard symbolic constant，会列举出IDA Pro 为这个特定值找到的所有对应量。



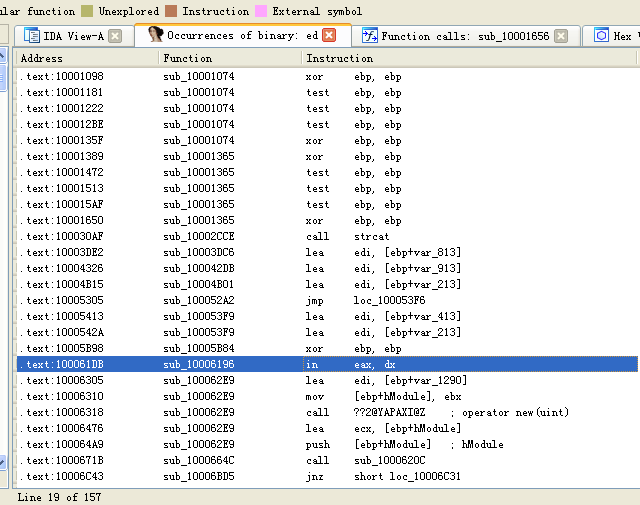
而对于socket函数，三个参数分别为domain（协议域，或协议族family）、type（socket类型）、protocol（协议）。在这个socket函数中，domain=2对应AF\_INET，即决定了要用IPv4地址（32位）与端口号（16位）的组合；type=1对应的是SOCK\_STREAM，即流式套接字；protocol=6对应的是IPPROTO\_TCP，即TCP协议。应用了修改后参数如下：



**(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。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？**

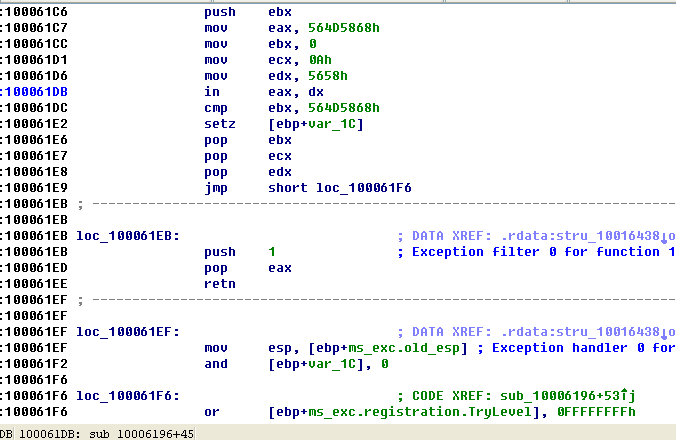
搜索in指令

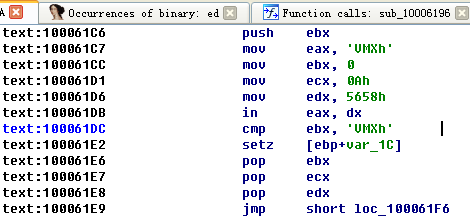


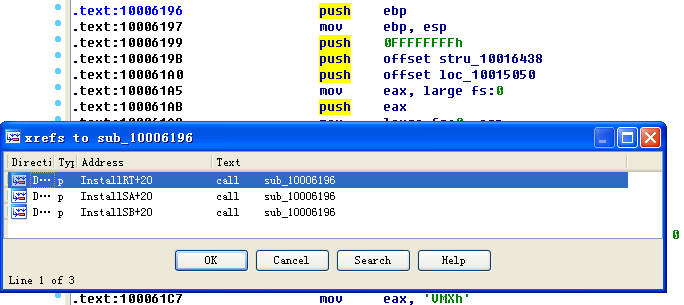


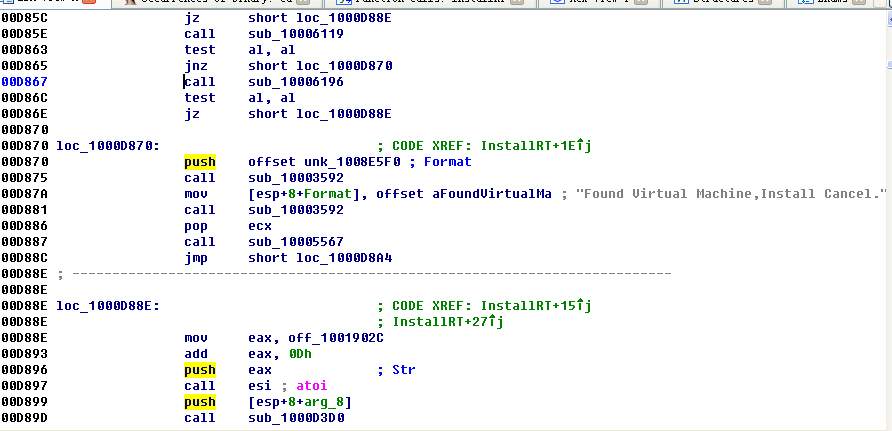
双击sub\_10006196跳转到in指令地址处，发现了一些类似于字符串的十六进制数

选中这些十六进制数，按R键将其转化为字符串，发现它们正是VMXh，所以这个恶意代码使用了Vmware检测技术。



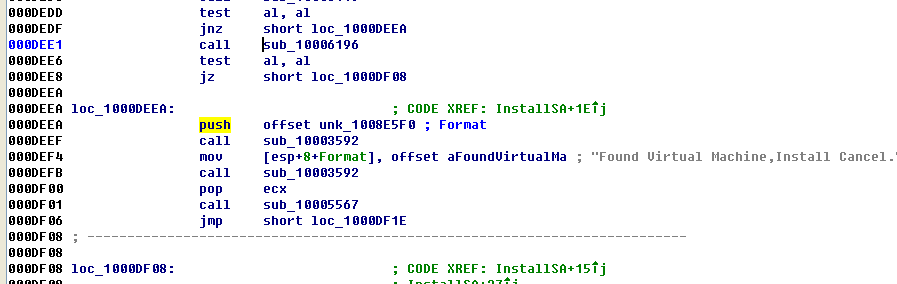


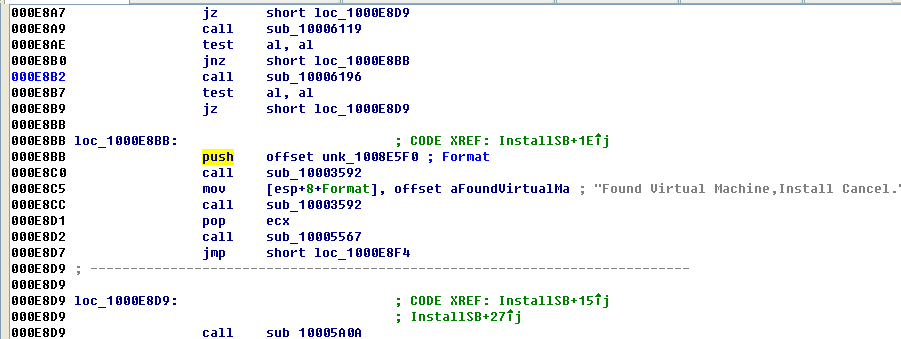




单机该in指令所在的函数名，按ctrl+x查看交叉引用，发现一共有三个函数调用了该函数

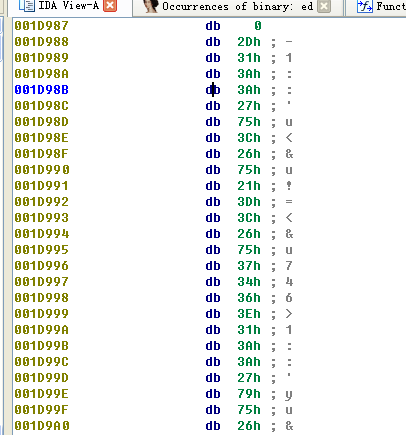
分别查看这三个函数，发现均存在Found Virtual Machine,Install Cancel的字样，这些便是对VMware进一步检测的证据。





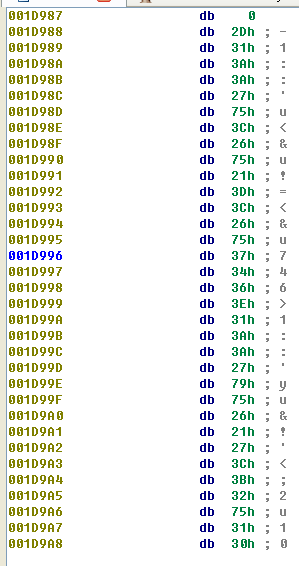
**(18)将你的光标跳转到0x1001D988处，你发现了什么？**

发现了一串没有含义的随机数据



1. **如果你安装了IDA Python插件（包括IDA Pro的商业版本插件），运行Lab05-01.py，一个IDA Pro Python脚本（确定光标是在0x1001D988处），在运行该脚本后发生了什么？**

运行脚本后变成了明文字符串



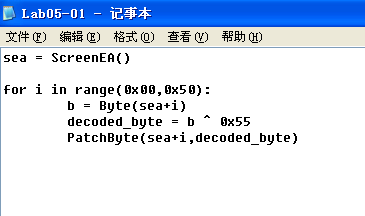
1. **将光标放于同一位置，你如何将这个数据转成一个单一的ASCII字符串？**



按下a键后

该字符串是xdoor is this backdoor, string decoded for Practical Malware Analysis Lab :)1234

**(21)使用一个文本编辑器打开这个脚本。它是如何工作的？**



ScreenEA()函数获取IDA调试窗口中光标所指向代码的地址，for循环范围为连续50个字节，Byte函数读取每个字节的值，然后再将该值于0x55进行异或运算，最后将结果输出返回到IDA对应的地址中。

1. **实验结论及心得体会**

1.DLLMain的地址是什么？

0x1000D02E

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

0x100163CC

3.有多少函数调用了gethostbyname？

被5个不同的函数调用调用了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？

如果在0x10001757处调用gethostbyname，恶意代码会发起对praticalmalwareanalysis.com的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？

识别出了23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？

识别出了一个参数

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

位于0x10095B34处

8.在引用 \cmd.exe /c的代码所在区域发生了什么？

为攻击者开启一段远程shell会话

9.在同样的区域，在0x100101C8处，看起来dword\_1008E5C4好像是一个全局变量，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：使用交叉引用）

sub\_10003695函数先检查主机是否为win32\_NT系统，再将判断结果(0或1)存放入eax寄存器中，再放入dword\_1008E5C4中

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？

注册表HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime和Worktimes的值会被查询，然后通过远程shell连接发送给攻击者。

11.PSLIST导出函数做了什么？

PSLIST导出函数会查询所有进程，并将该进程列表通过网络发送出去，或者寻找某个特定进程并获取其信息

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？

调用了GetSystemDefaultLangID、sprintf、strlen三个API，可以重命名为GetSystemLanguageId

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

直接调用了strncpy、strnicmp、CreateThread和strlen这四个API，大约有31个API再深度为2时被调用

14.在0x10001358处，有一个对Sleep（一个使用一个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？

恶意代码会使程序休眠30秒

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

这三个参数为6，1，2

16.使用MSDN页面的socket和IDA Pro中的命名符号常量，你能使参数更有意义吗？修改后，参数是什么？

domain=2对应AF\_INET；type=1对应的是SOCK\_STREAM；protocol=6对应的是IPPROTO\_TCP

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。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？

在这个代码中存在in指令和VMXh的使用，还存在Found Virtual Machine,Install Cancel的字符串，证明该恶意代码做出了进一步的Vmware检测

18.将你的光标跳转到0x1001D988处，你发现了什么？

发现了一串随机数据

19.如果你安装了IDA Python插件（包括IDA Pro的商业版本插件），运行Lab05-01.py，一个IDA Pro Python脚本（确定光标是在0x1001D988处），在运行该脚本后发生了什么？

那些随机数据变成了一串字符串明文

20.将光标放于同一位置，你如何将这个数据转成一个单一的ASCII字符串？

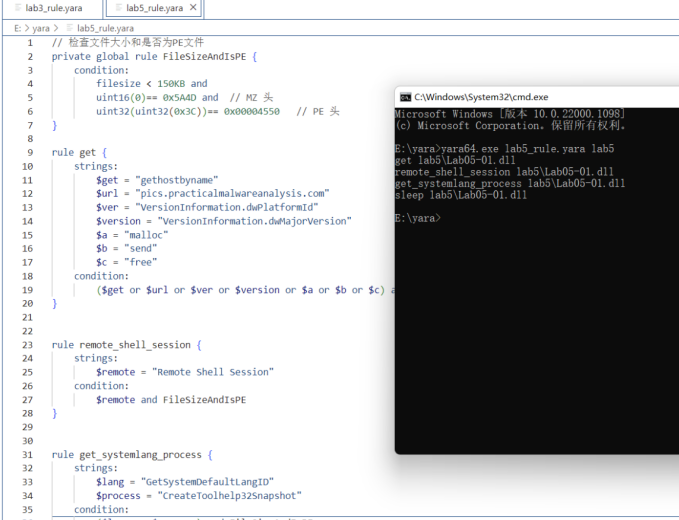
按下a键，这些单一字符转换为一个单一的ASCII字符串

21.使用一个文本编辑器打开这个脚本。它是如何工作的？

ScreenEA()函数获取IDA调试窗口中光标所指向代码的地址，for循环范围为连续50个字节，Byte函数读取每个字节的值，然后再将该值于0x55进行异或运算，最后将结果输出返回到IDA对应的地址中。

**Yara规则**

|  |
| --- |
| // 检查文件大小和是否为PE文件  private global rule FileSizeAndIsPE {      condition:          filesize < 150KB and          uint16(0)== 0x5A4D and  // MZ 头          uint32(uint32(0x3C))== 0x00004550   // PE 头  }  rule get {      strings:          $get = "gethostbyname"          $url = "pics.practicalmalwareanalysis.com"          $ver = "VersionInformation.dwPlatformId"          $version = "VersionInformation.dwMajorVersion"          $a = "malloc"          $b = "send"          $c = "free"      condition:          ($get or $url or $ver or $version or $a or $b or $c) and FileSizeAndIsPE  }  rule remote\_shell\_session {      strings:          $remote = "Remote Shell Session"      condition:          $remote and FileSizeAndIsPE  }  rule get\_systemlang\_process {      strings:          $lang = "GetSystemDefaultLangID"          $process = "CreateToolhelp32Snapshot"      condition:          ($lang or $process) and FileSizeAndIsPE  }  rule sleep {      strings:          $sleep = "sleep" nocase      condition:          $sleep and FileSizeAndIsPE  }  rule Internet {      strings:          $a = "IPPROTO\_TCP"          $b = "SOCK\_STREAM"          $c = "AF\_INET"      condition:          ($a or $b or $c) and FileSizeAndIsPE  }  rule VM {      strings:          $VM = "VMXh"      condition:          $VM and FileSizeAndIsPE  }  rule xdoor {      strings:          $a = "xdoor is this backdoor, string decoded for practical Malware Analysis Lab"      condition:          $a and FileSizeAndIsPE  } |



**结果分析：**

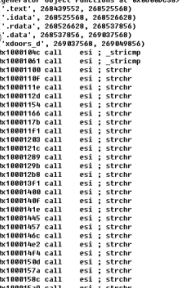
最后三个rule（Internet、VM、xdoor）并没有被扫描出来，**可能是因为本身程序中不含有这三个rule中的字符串**。rule Internet中的字符串是我们在Q16中“Use Standard Symbolic Constant”替换出来的；rule VM中的“VMXh”是我们在Q17中将十六进制串变为字符串得到的；xdoor中的字符串是我们利用IDA Python的script file进行解密得到的。所以上述三条rule中的字符串其实在原文中并没有体现，我们可以使用其原文中的地址/十六进制串/密文等判断。**其余规则扫描Lab05-01.dll成功！**

**IDA Python**

我们可以编写**script file**如下：

|  |
| --- |
| import idautils  sea = ScreenEA()  for i in range(0x00,0x50):  b = Byte(sea+i)  decoded\_byte = b ^ 0x55  PatchByte(sea+i,decoded\_byte)  print(idautils.Functions())  for i in idautils.Segments():  print(idc.SegName(i),idc.SegStart(i),idc.SegEnd(i))  for i in idautils.Functions():  flags = idc.GetFunctionFlags(i)  if flags&FUNC\_LIB or flags &FUNC\_THUNK:  continue  dism\_addr = list(idautils.FuncItems(i))  for line in dism\_addr:  j = idc.GetMnem(line)  if j == 'call':  op=idc.GetOpType(line,0)  if op == o\_reg:  print("0x%x%s"%(line, idc.GetDisasm(line))) |

**结果显示：**



**分析：**

我们编写的python代码返回了IDA识别出的所有函数入口点列表；遍历所有的指令，所有的交叉引用地址,还有搜索所有的代码和数据；输出调用call指令的寄存器。（最前面一段代码为解密代码可以继续使用）

**本次实验我们利用win xp系统下的ida pro6.6进行了相关的实验，在实验过程当中，我们发现了lab05-01.dll程序的各种特殊的功能，其在操作系统当中还进行了许多的检测，是我们在程序分析的一个新的特色角度，其在操作系统当中可能进行的检测有可能会造成程序动态链接到某些位置，从而产生一些错误的情况，需要我们在程序流程当中进行相关的分析，之后得到问题的一个合适解。**

**本次实验当中我们采用各种方法对程序当中的一部分片段进行了反汇编，其中idapro6.6对于idac和ida python在反汇编的运用技巧并不相同，题目给了我们一些研究讨论问题的思路，这些思路给了我们很多观察和研究问题的新角度，能让我们对于ida分析有一定的新理解。**

**研究本次实验，我们不仅需要掌握ida的一些使用技巧和分析策略，还要在查找到相关的特殊链接库之后，尝试性的在网络当中探寻其有哪些作用，以及会造成什么样的影响，这些在我们进行恶意代码分析当中尤为重要，可以将其中的特殊问题作为程序的一种攻击或者隐藏的方法。**