

**《恶意代码分析与防治技术》课程实验报告**

**实验五**

****

学 院 网络空间安全学院

专 业 信息安全

学 号 2112060

姓 名 孙蕗

班 级 信息安全1班

《恶意代码分析与防治技术》课程Lab5实验报告

[一、 实验目的 3](#_Toc23876)

[二、 实验原理 3](#_Toc15167)

[三、 实验过程 3](#_Toc11171)

[（一） Lab5-1 3](#_Toc18344)

[（二） Yara 19](#_Toc18299)

（三） IDA Python脚本................................ 20

[四、 实验结论及心得体会 22](#_Toc14867)

1. **实验目的**

分析一个DLL文件中的恶意代码，使用逆向工程技术来了解该恶意代码的行为和功能。

通过IDA Pro工具来分析DLL文件，识别并理解其中的恶意代码，掌握IDA Pro的基本功能，如查找函数、变量、字符串等，以及进行逆向分析，探索DLL文件的导出函数和内部函数的调用关系。

1. **实验原理**

IDA Pro工具：IDA Pro是一种专业的逆向工程工具，用于分析二进制文件，特别是可执行文件和动态链接库。它提供了强大的反汇编和分析功能，允许分析人员查看和理解二进制代码。

导出函数和内部函数：分析DLL文件的导出函数可以帮助了解该库提供的功能。内部函数的调用关系和交叉引用可以揭示代码的逻辑和行为。

字符串和常量分析：分析恶意代码中的字符串和常量可以揭示其功能、特征或目的。这些信息有助于理解代码的作用。

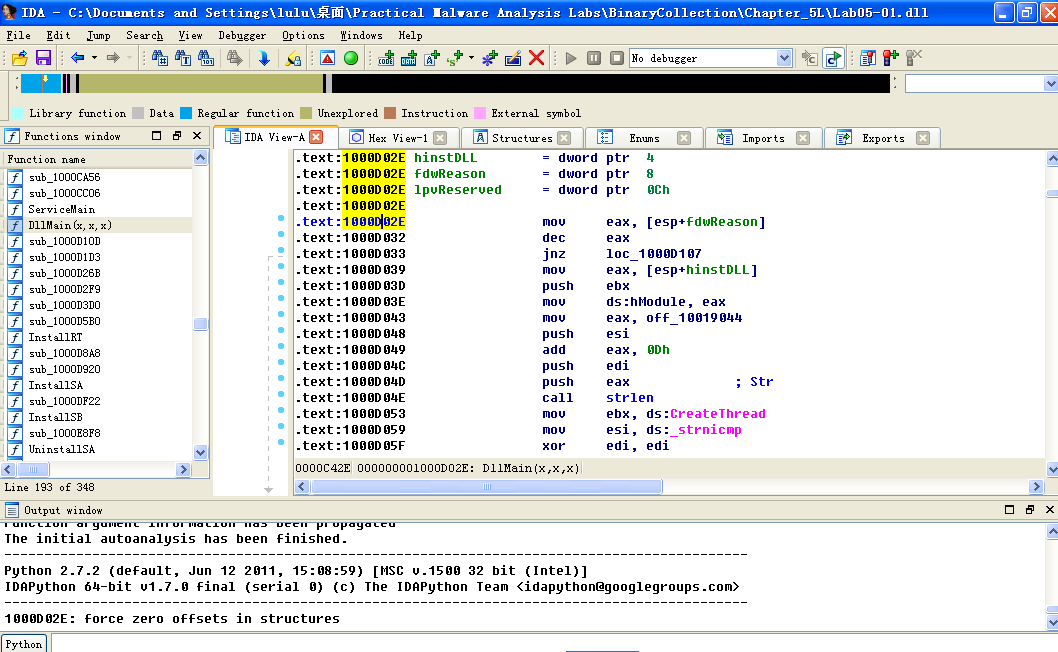
使用Yara规则：Yara是一种用于创建规则以检测恶意代码特征的工具。创建和应用Yara规则可以帮助自动检测和识别恶意代码。

1. **实验过程**
2. **Lab5-1**

**只用IDA Pro分析在文件Lab05-01.dll中发现的恶意代码。这个实验的目标是给你一个用IDA Pro动手的经验。如果你已经用IDA Pro工作过，你可以选择忽略这些问题，而将精力集中在逆向工程恶意代码上。**

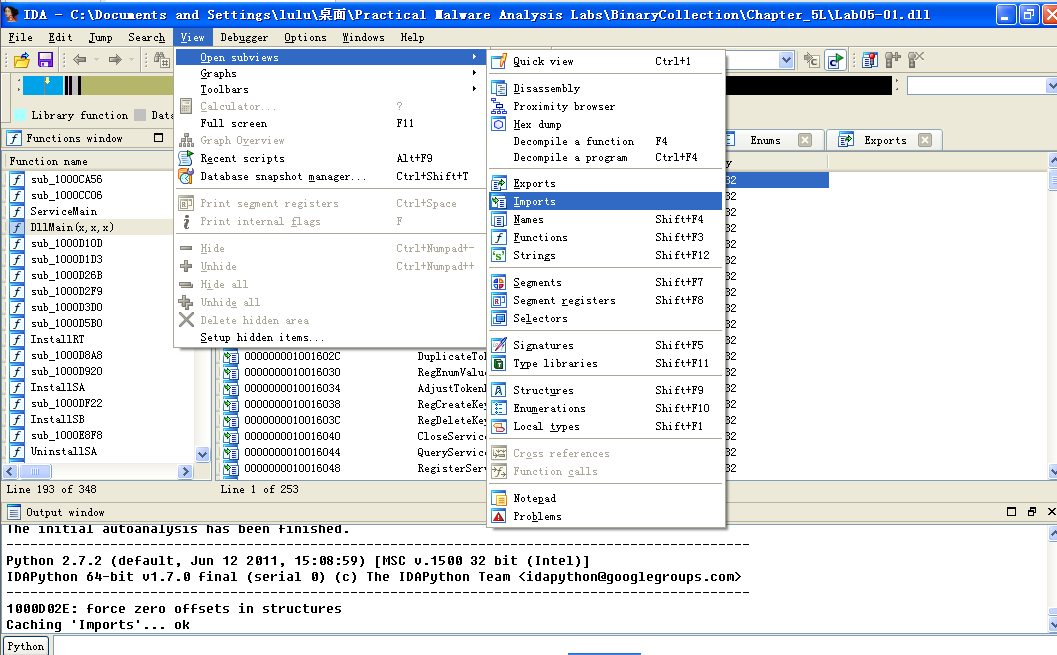
1. **DllMain的地址是什么?**

在IDA Pro中打开Lab05-01.dll文件，使用空格键从图形模式切换到汇编模式，在function name中选择DllMain(x,x,x)，发现DllMain地址在.text节的0x1000D02E处。

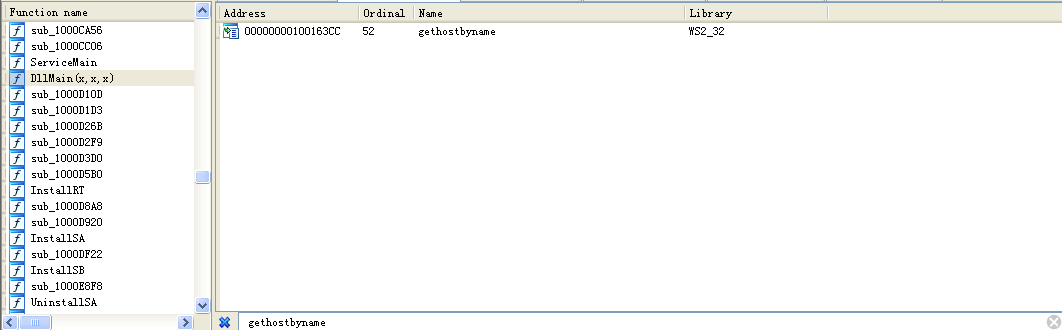


1. **使用Imports窗口并测览到gethostbyname,导入函数定位到什么地址?**

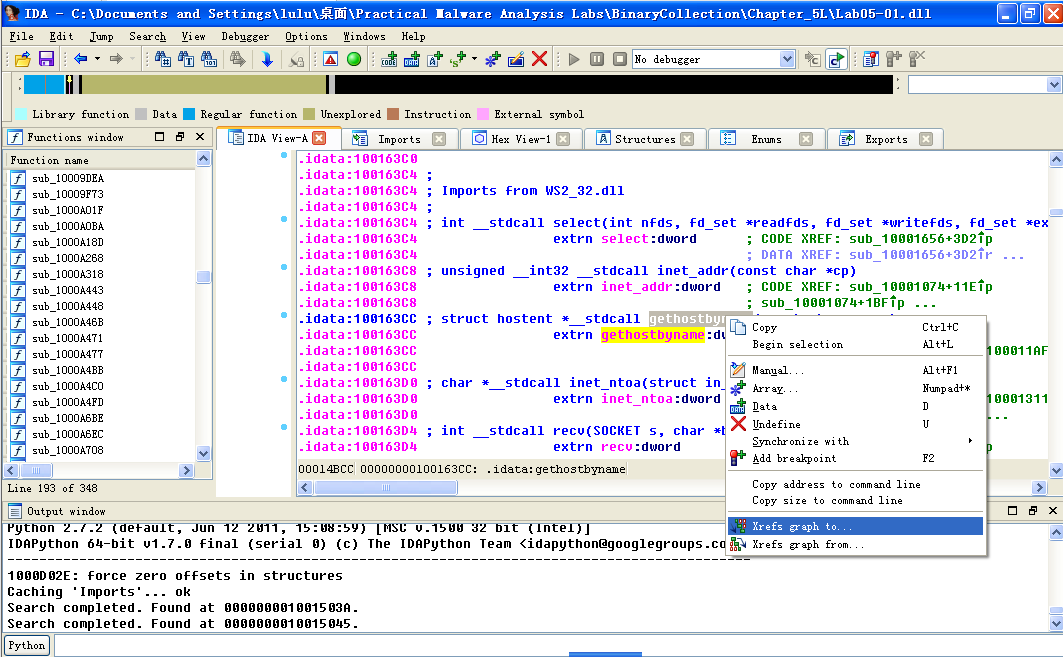
通过View->Open Subviews->Imports查看该程序的导入表



搜索gethostbuname，地址定位到.idata节的0x100163CC处。

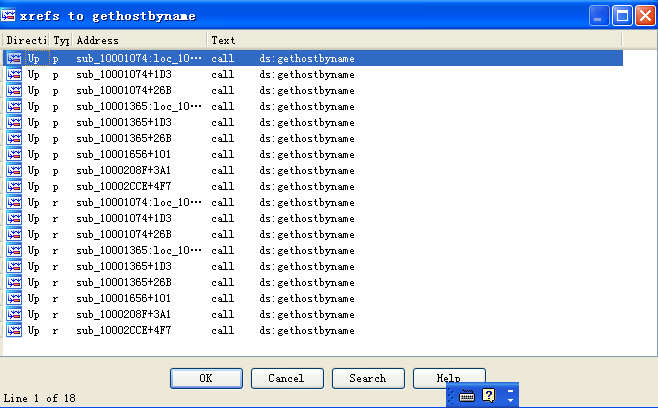


1. **有多少函数调用了gethostbyname?**



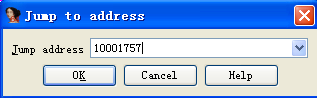
在汇编窗口中找到有gethostbyname函数名的语句，Ctrl+X键来检查其交叉引用，其中p是引用，r是读取，要先读取，再引用。

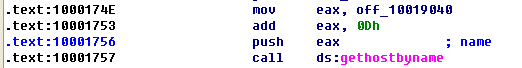
结合地址和偏移信息，发现gethostbyname在被5个不同的函数调用了9次。



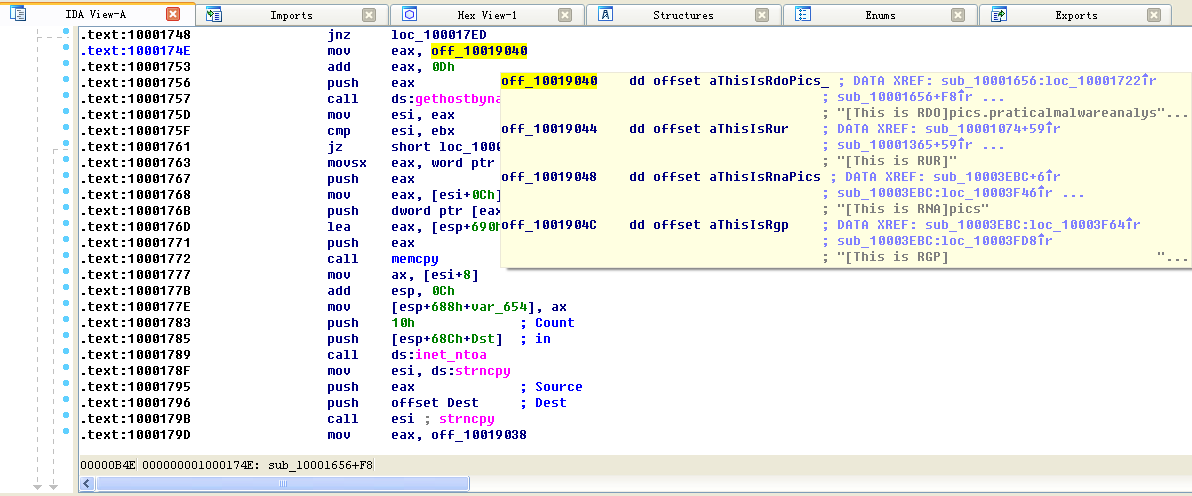
1. **将精力集中在位于0x10001757处的对gethostbyname的调用，你能找出哪个DNS请求将被触发吗?**

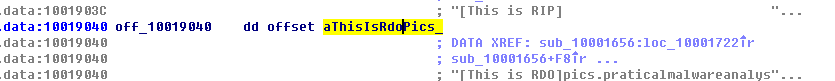
G快速定位到0x10001757，



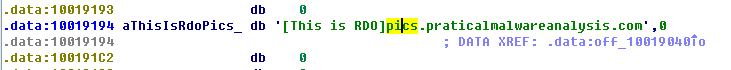


gethostbyname方法用了一个参数——一个包含了域名的字符串，回看确定gethostbyname被调用时off\_10019040被赋值给了eax，在此处看到字符串pics.practicalmalwareanalysis.com。

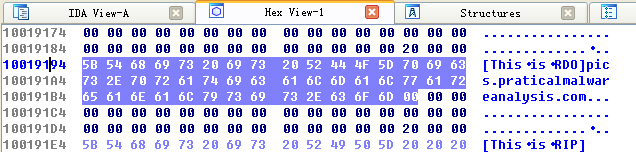




双击aThisIsRdoPics\_，发现了praticalmalwareanalysis.com的域名信息。



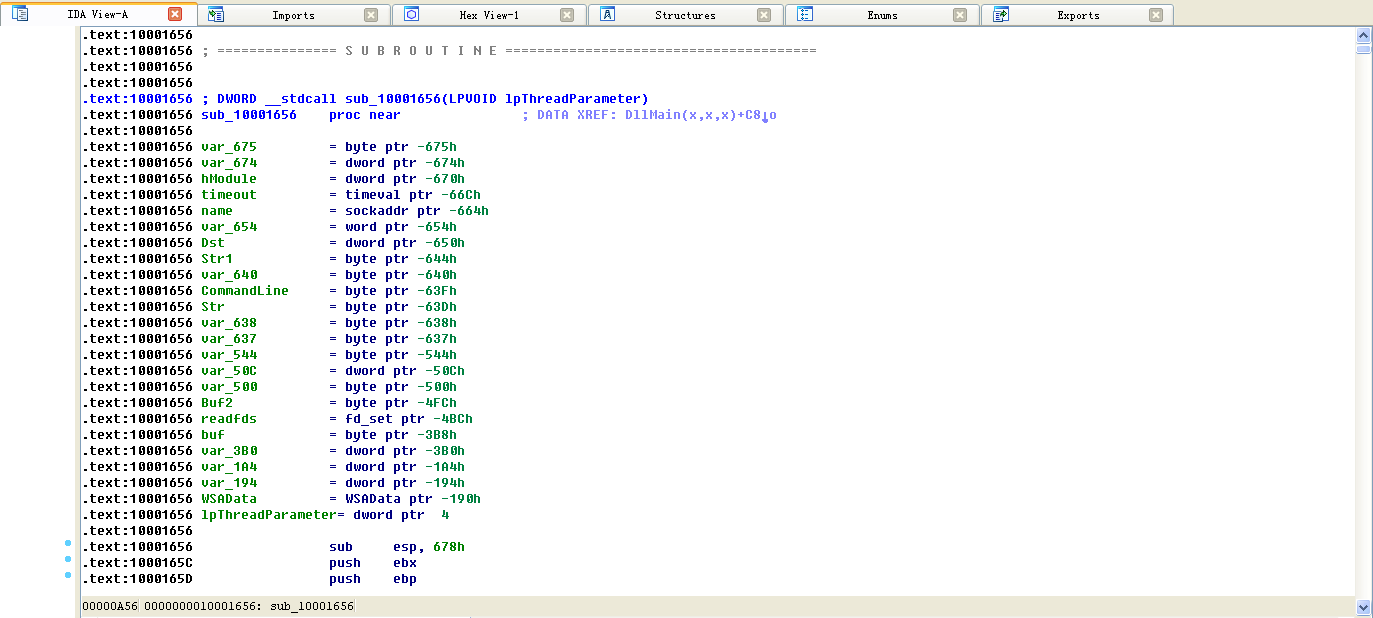
在HEX-View1窗口对0x10019194进行分析，前13字节就是[This is RDO]。0x10001753处的代码令eax加上0Dh，就是将eax指向了praticalmalwareanalysis.com，然后再将eax压入栈作为gethostbyname的参数。



综上所述，恶意代码会发起对pics.practicalmalwareanalysis.com的DNS请求，以获得其IP地址。

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

G快速定位到0x10001656，被标记的局部变量与负偏移值相关，一共识别出23个局部变量。

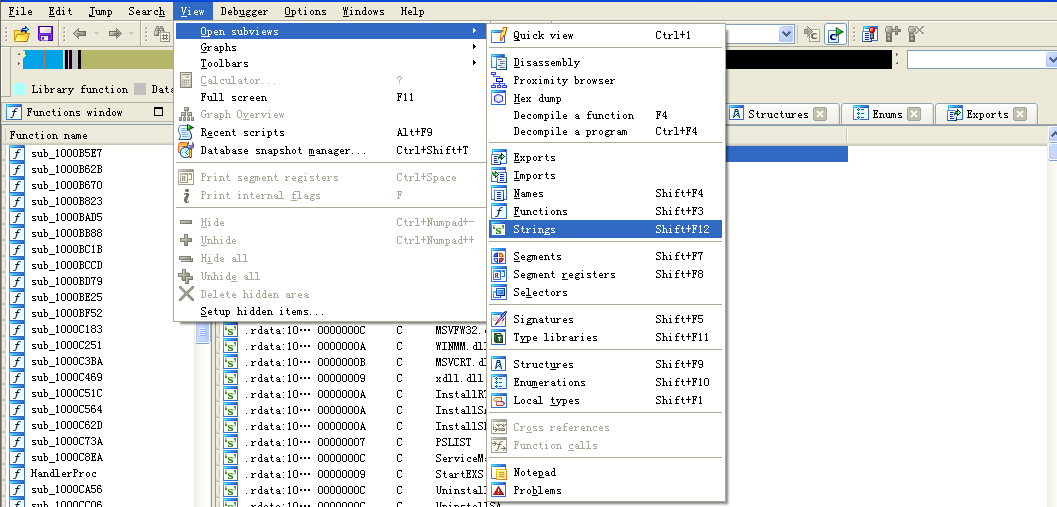


1. **IDA Pro识别了在0x10001656处的子过程中的多少个参数?**

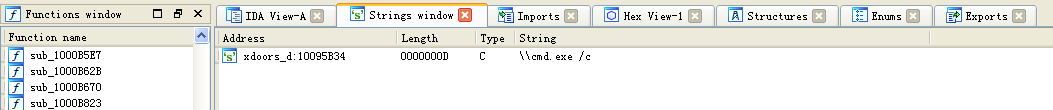
参数被标记和引用为正的偏移值，可以看到IDA Pro识别出函数的一个参数，标记为lpThreadParameter，一共识别出1个参数。

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

View->Open Subviews->Strings，查看字符串

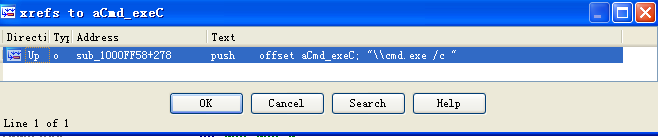


搜索字符串\cmd.exe /c，看到字符串\cmd.exe /c出现在xdoors\_d节的0x10095B34处。



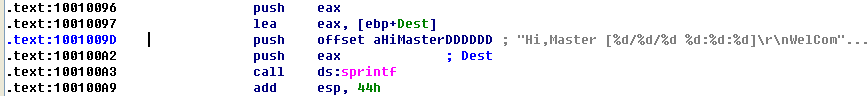
1. **在引用\cmd.exe /c的代码所在的区域发生了什么?**

检查该字符串的交叉引用，只看到一处在0x100101D0，字符串被压到栈上。

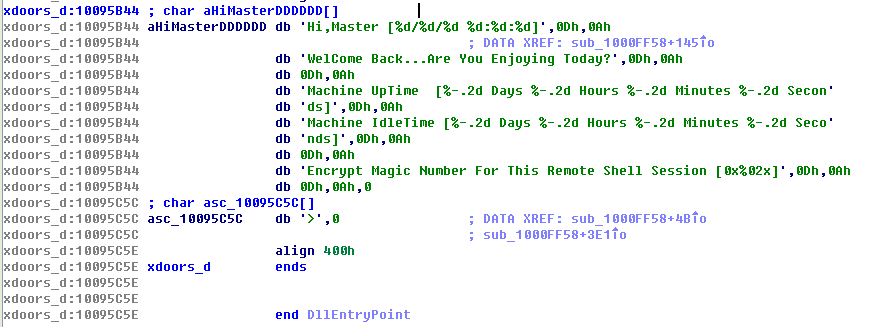




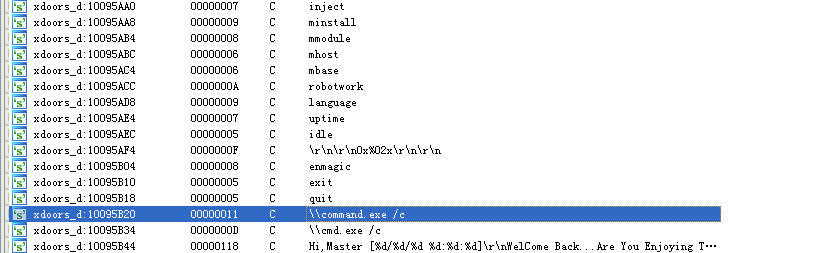
该字符串被引用之前的0x1001009D处，存在aHiMasterDDDDDD，有一个字符串This Remote Shell Session。



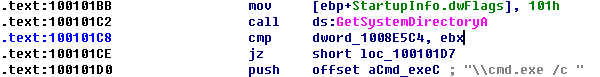
双击aHiMasterDDDDDD查看，



根据上图被选中区域的字符串，分析得出远程shell会话解密一段数字,我们正在查看的是一个远程shell会话函数,攻击者开启一个远程shell会话。

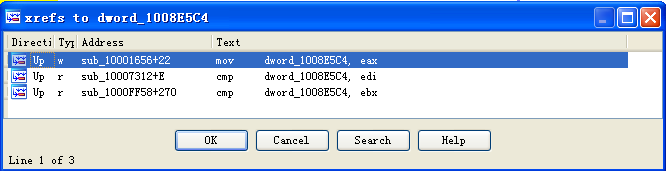


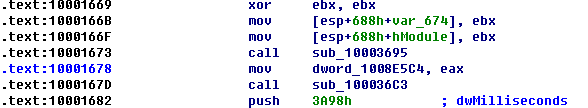
1. **在同样的区域，在0x100101C8处，看起来好像dword\_1008E5C4是一个全局变量，它帮助决定走哪条路径。那恶意代码是如何设置dword\_1008E5C4的呢?(提示：使用dword\_1008E5C4的交叉引用。)**



dword\_1008E5C4函数是一个全局变量，在局部变量和参数以在0x100101C8处双击它，会跳转到内存中的0x1008E5C4处，位于DLL文件的.data节。Ctrl+X键来检查其交叉引用，可以看到被引用了三次，但只有第一处修改了dword\_1008E5C4。

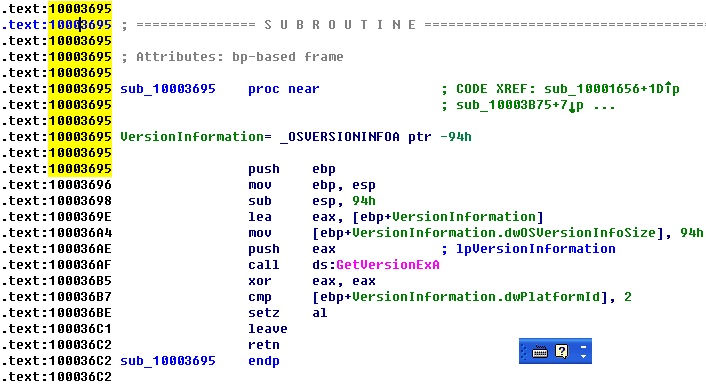






EAX被赋给dword\_1008E5C4,而EAX是前一条指令函数调用的返回值。

双击sub\_10003695查看eax的返回值，该函数包括了一个GetVersionEx的调用，可用于获取当前操作系统版本的信息。



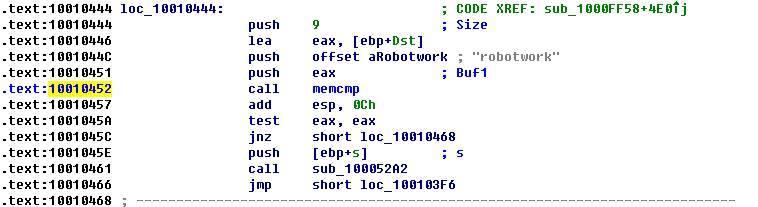
cmp [ebp+VerisonInformation.dwPlatformId],2语句，将VersionInformation.dwPlatformId与数字2进行比较，来确定如何设置AL寄存器，2代表WIN32\_NT系统。如果PlatformId为VER\_PLATFORM\_WIN32\_NT,AL会被置位。

这里的函数会检查一下当前操作系统是不是win32系统或更高版本来决定eax的值，之后再将得到的系统版本号存放入dword\_1008E5C4中，该全局变量通常会被置为1。

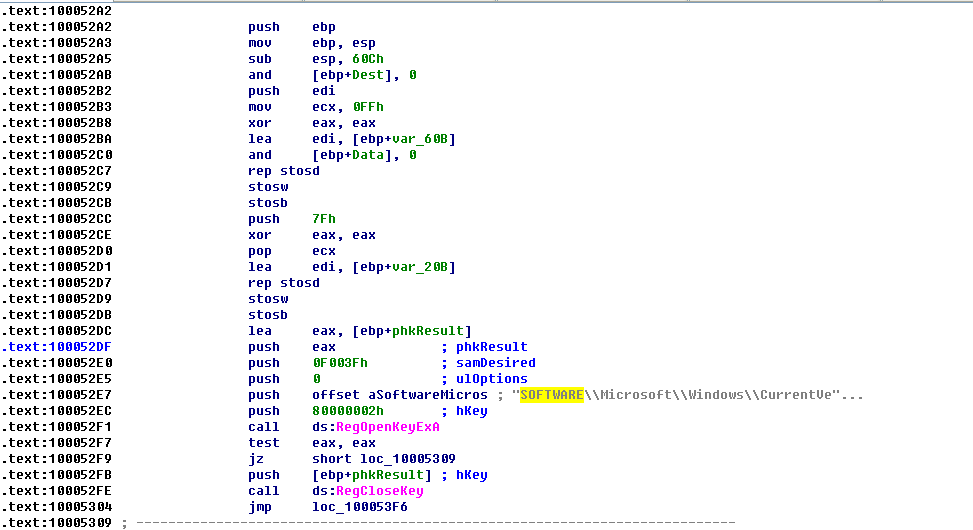
综上所述，操作系统版本号被保存在了dword\_1008E5C4中。

1. **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?**

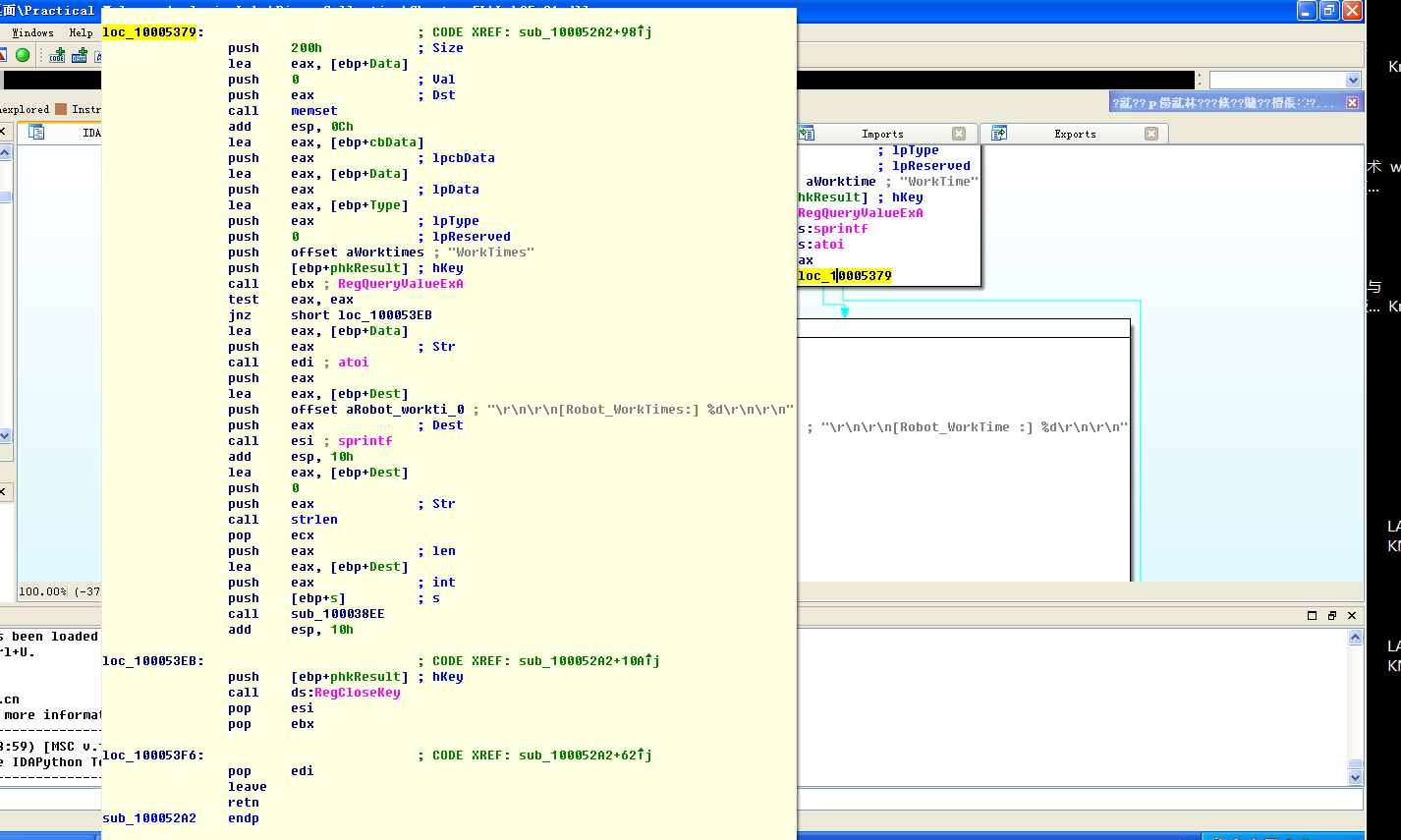
位于0x1000FF58处的远程shell函数从0x1000FF58开始包含了一系列的memcmp函数。其中，在0x10010452,可以看到与robotwork的memcmp。



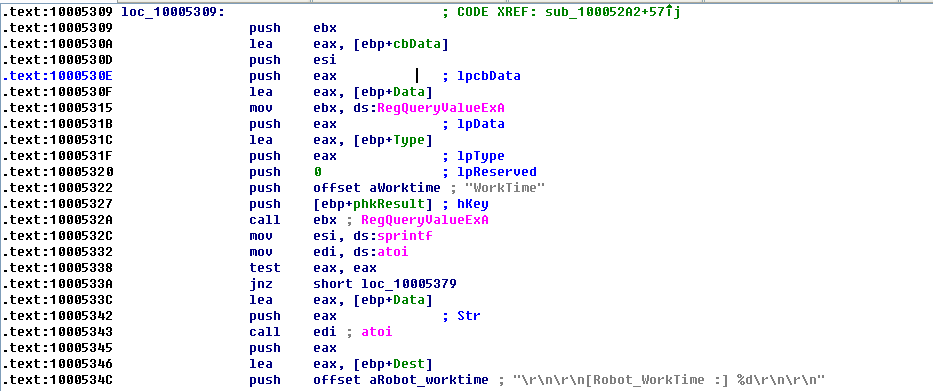
如果该字符串为robotwork,call sub\_100052A2处的代码会被调用。查看sub\_100052A2的代码，可以看到它查询了注册表中HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime和WorkTimes键的值。



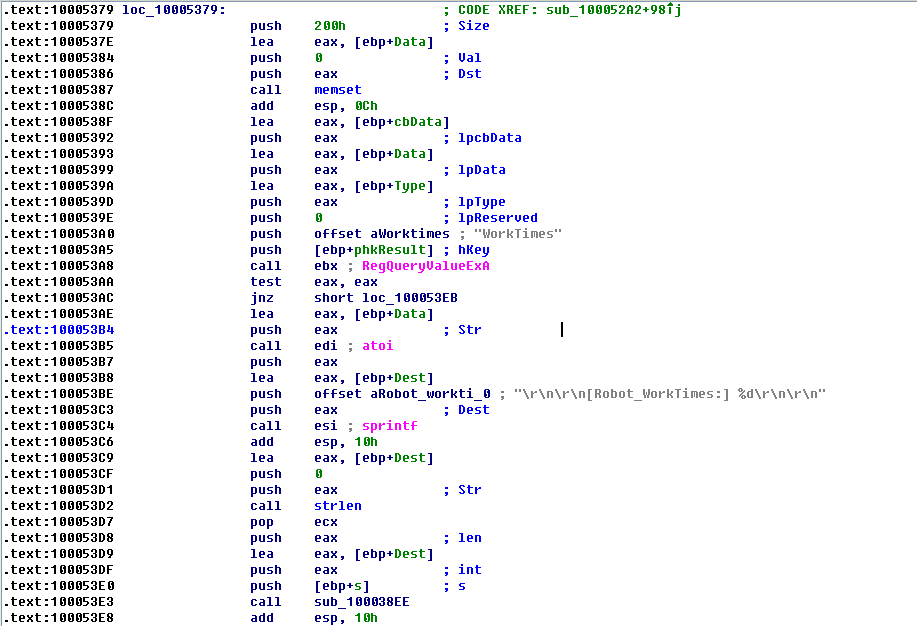
在sub\_100052A2区域中发现调用了sub\_100038EE函数。



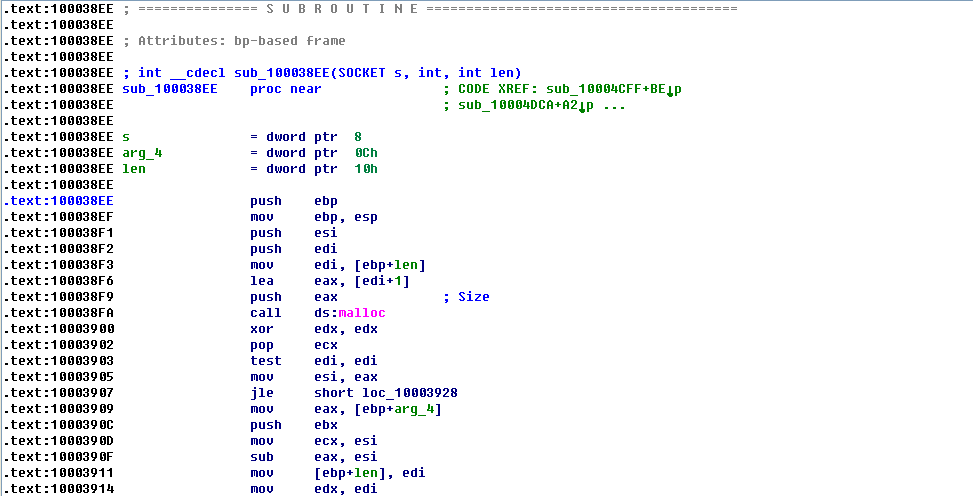
查看loc\_10005309，



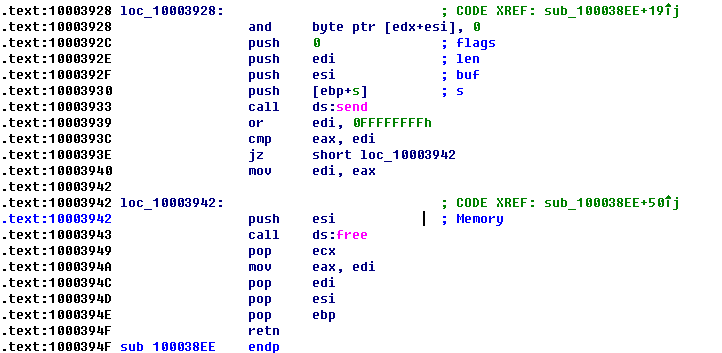
查看loc\_10005379，



查看sub\_100038EE函数，



查看loc\_10003928和loc\_10003942，

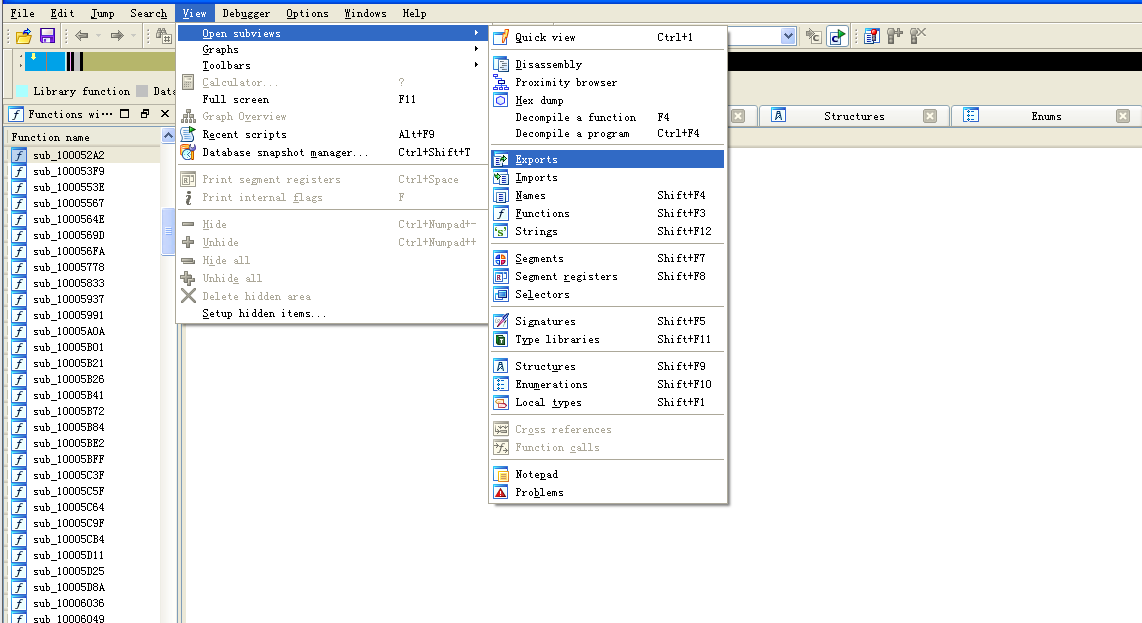


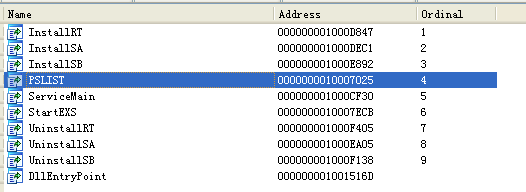
发现其调用了malloc函数创建了内存空间，然后又调用了send函数，最后调用了free函数释放内存空间。

综上所述，对注册表SOFTWARE\Microsoft\Windows\CurrentVersion\WorkTime和Worktimes键的值进行了查询操作，然后将这一信息返回给0x1001045E处的push [ebp+s]传给该函数的网络socket，通过远程shell连接发送出去。

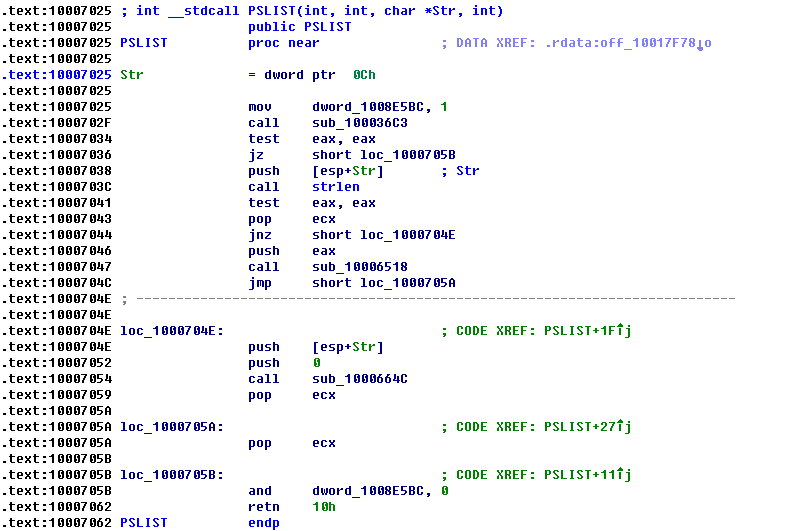
1. **PSLIST导出函数做了什么?**

View→Open Subviews→Exports,查看该DLL的导出表。



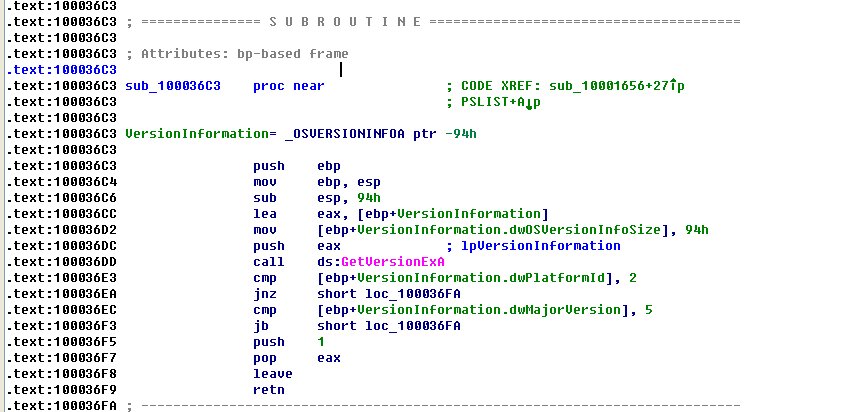


双击PSLIST，跳转到0x10007025，

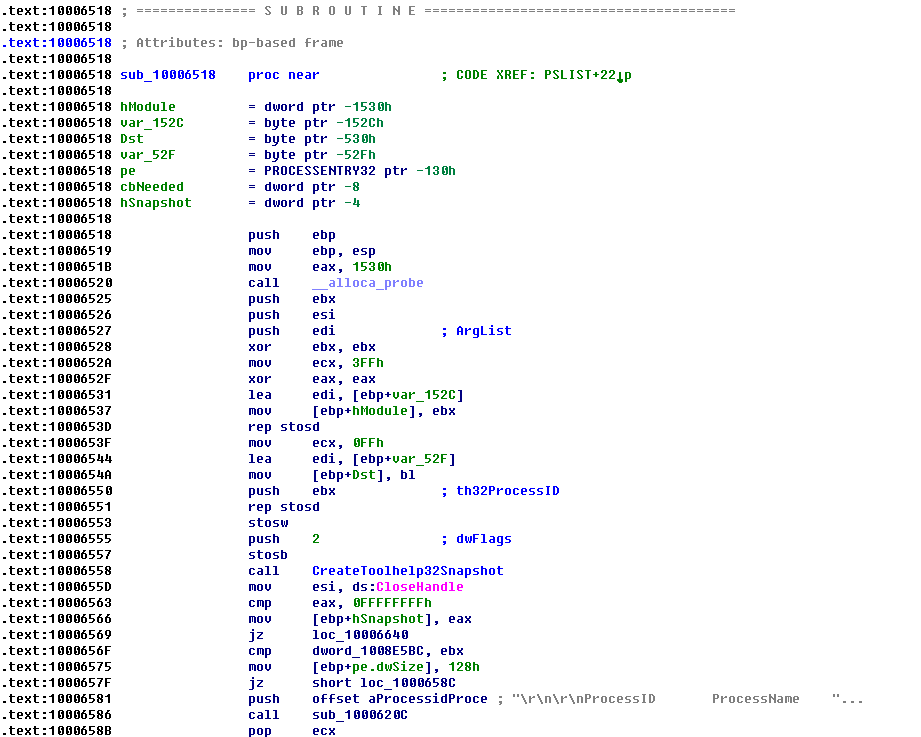


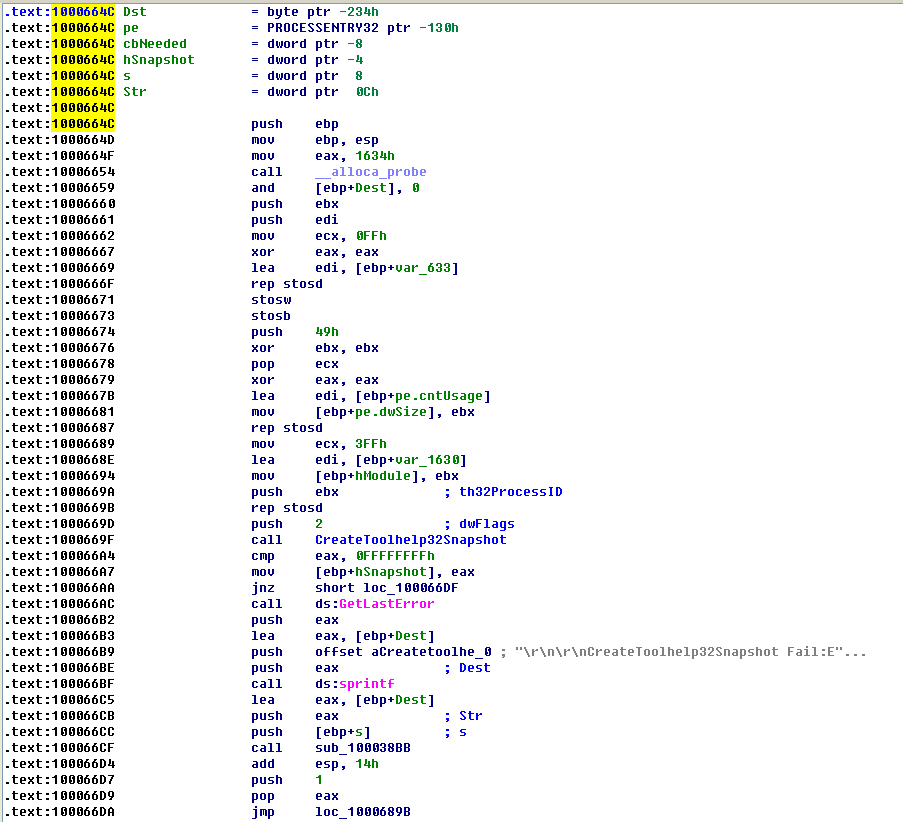
这个函数选择两条路径之一执行，取决于sub\_100036C3的结果。如果返回值为1，则往下执行函数sub\_10006518和sub\_1000664C。

sub\_100036C3函数可用于检查操作系统的版本是Windows Vista/7,或是Windows XP/2003/2000，如果是，则返回1。

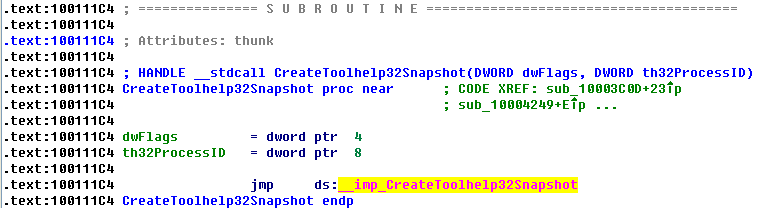


查看sub\_10006518和sub\_1000664C，

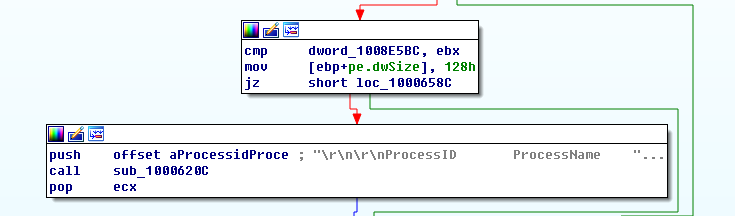


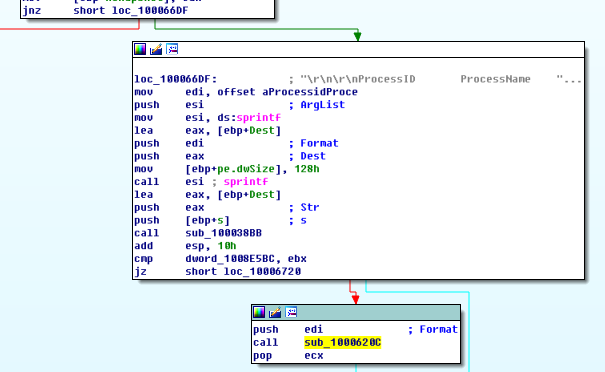


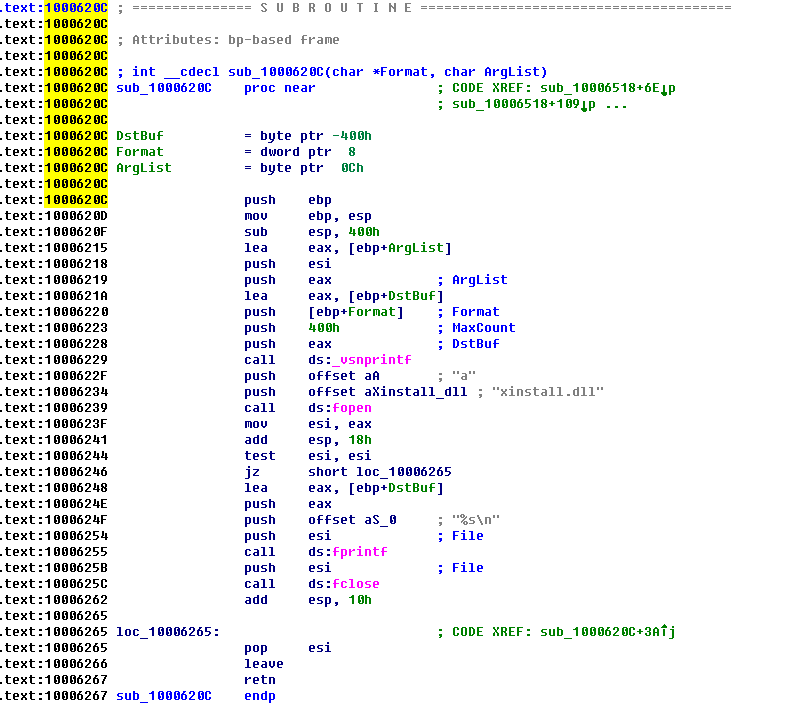
sub\_10006518和sub\_1000664C都使用CreateToolhelp32Snapshot函数，可用于获得主机的进程列表。



sub\_10006518和sub\_1000664C还使用了sub\_1000620C函数，应该是将查询到的进程信息写到一个文件中。



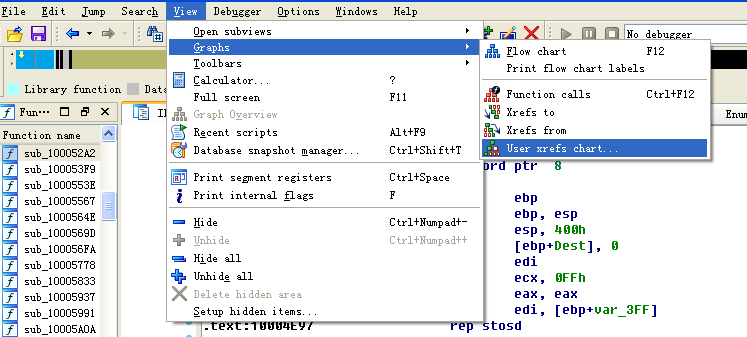


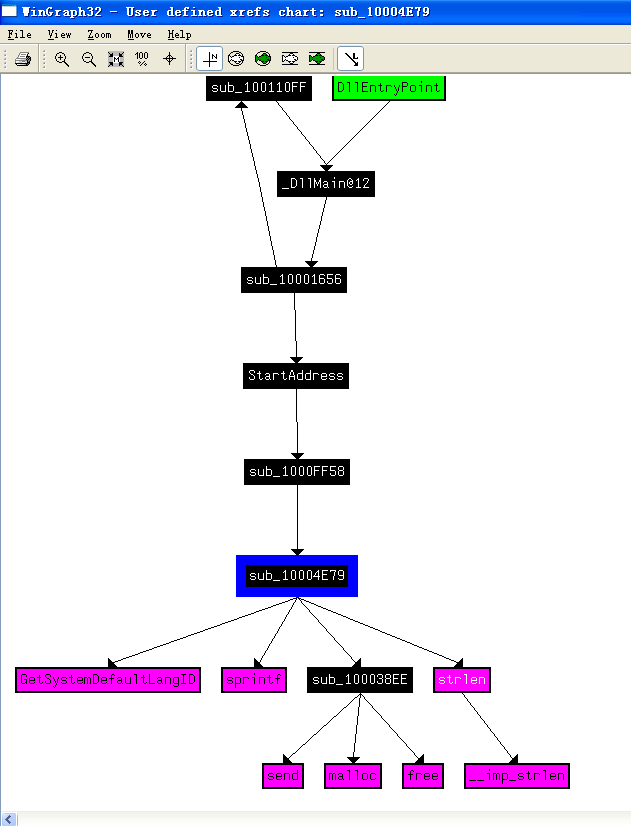


综上所述，PSLIST导出项可以寻找该列表中某个指定的进程名并获取其信息保存到文件中，并通过send将进程列表通过socket发送。

1. **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?**

G快速跳转到0x10004E79，View→Graphs→User Xrefs Chart，得到交叉引用图。

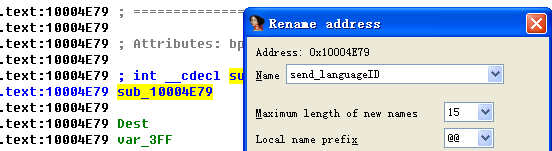




在sub\_10004E79处的函数中调用了GetSystemDefaultLangID、send和sprintf这三个API。该函数又调用了sub\_100038EE函数，sub\_100038EE函数又调用了send，malloc，free，\_\_imp\_strlen函数。上面的信息告诉我们，该函数可能通过socket发送语言标志。

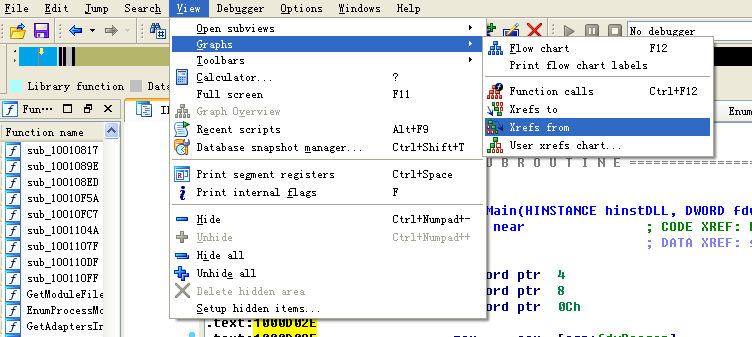
该函数调用了GetSystemDefaultLangID函数，又间接调用了send函数，猜测该函数会获取系统默认语言ID后，通过send函数发送，可将该函数取名GetSystemLanguage。

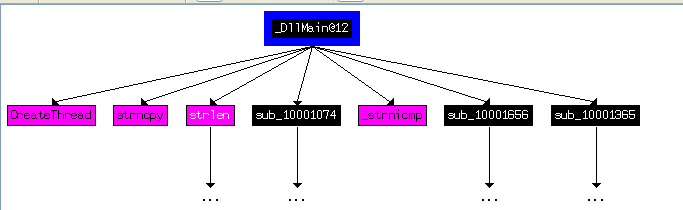
右击函数名，即可重命名为GetSystemLanguage。

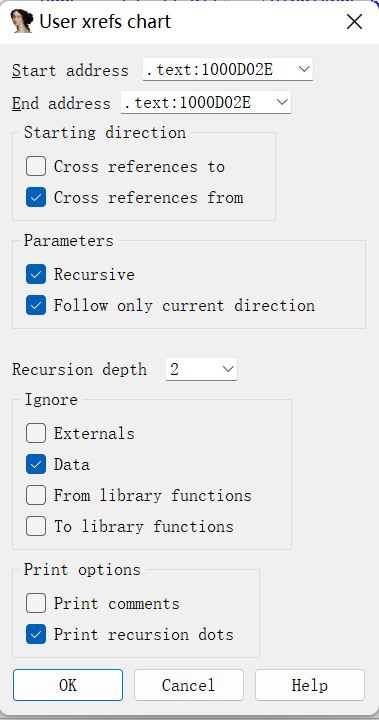


1. **DllMain直接调用了多少个Windows API?多少个在深度为2时被调用?**

找到DllMain的起始地址0x1000D02E，View→Graphs→Xrefs From，选取Recursion depth递归深度为1，以显示其直接调用函数。



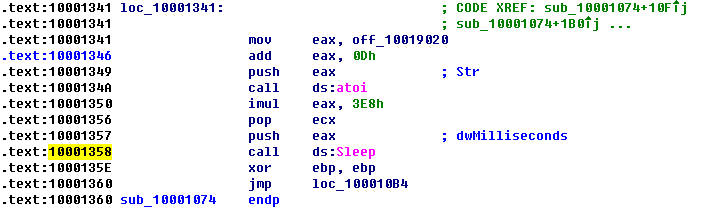




可以看到，DllMain直接调用了strncpy、strnicmp、CreateThread和strlen这些API。继续查看递归深度2所有被调用函数,可以看到非常多的API（30多个）,包括Sleep、WinExec、gethostbyname、CreateThread、ExitThread,以及许多其他网络函数调用。

1. **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?**

G跳转到0x10001358,该处有一个对Sleep的调用。Sleep只使用休眠的毫秒数作为参数，eax被压入栈上作为Sleep函数的参数。



eax被乘了0x3E8(十进制是1000),也就是说，对atoi调用的结果被乘以1000,得到要休眠的毫秒数。

off\_10019020被赋给eax,双击off\_10019020，指向了一个字符串[This is CTI]30。



0xD被加到eax上作为偏移，也就是13字节，对应信息就是[This is CTI]，因此eax指向了30。调用atoi,把字符串30转成了数字30。30乘上1000,得到30000毫秒(30秒),也就是程序将休眠30秒。

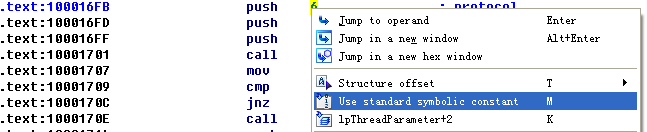
1. **在0x10001701处是一个对socket的调用。它的3个参数是什么?**

G快速跳转到0x10001701，看到6、1、2被压到栈上。因此三个参数为6、1、2。

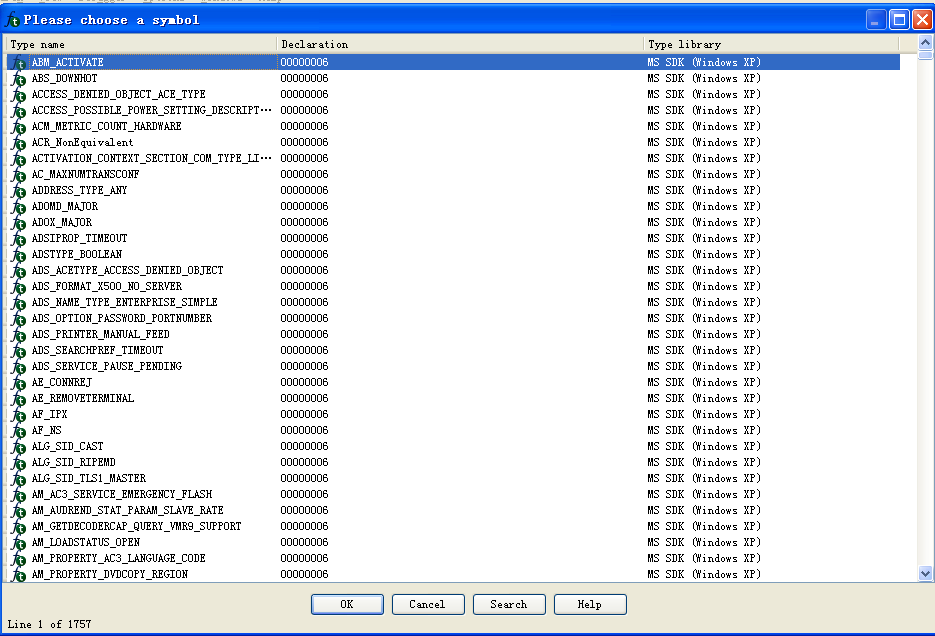


1. **使用MSDN页面的socket和IDA Pro中的命名符号常量，你能使参数更加有意义吗?在你应用了修改以后，参数是什么?**

右键每个数字，选择Use standard Symbolic constant。

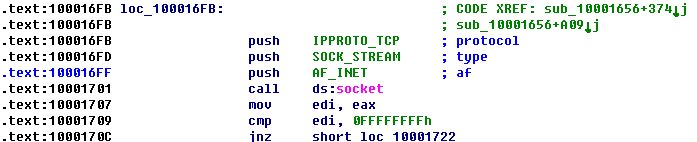


弹出一个列举IDA Pro为这个特定值找到所有的对应常量的对话框。



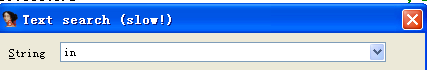
3个参数分别为domain（协议域，或协议族family）、type（socket类型）、protocol（协议）。domain=2，指的是AF\_INET,用于设置一个IPv4 socket，决定了要用IPv4地址（32位）与端口号（16位）的组合;type=1，指的是SOCK\_STREAM，即流式套接字。protocol=6，指的是IPPROTO\_TCP，即TCP协议。这个socket会被配置为基于IPv4的TCP连接(常被用于HTTP)。

应用了修改后的参数如下：

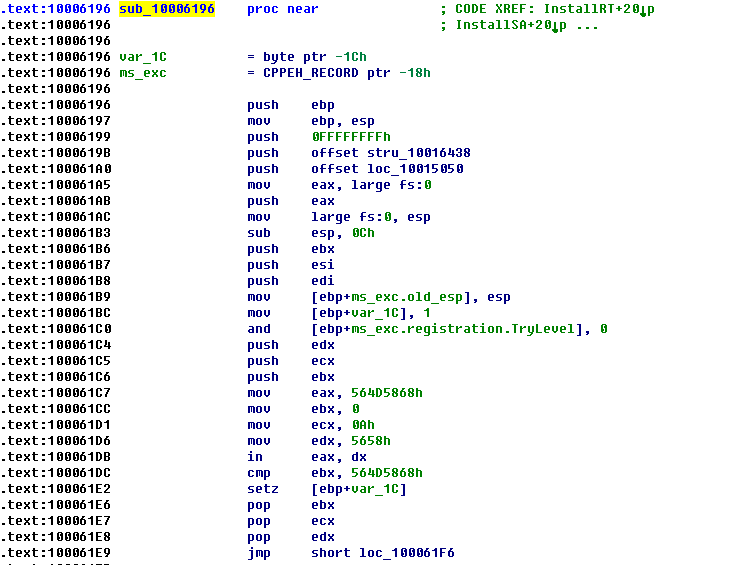


1. **搜索in指令(opcode 0xED)的使用。这个指令和一个魔术字符串VMXh用来进行VMware检测。这在这个恶意代码中被使用了吗?使用对执行in指令函数的交叉引用，能发现进一步检测VMware的证据吗?**

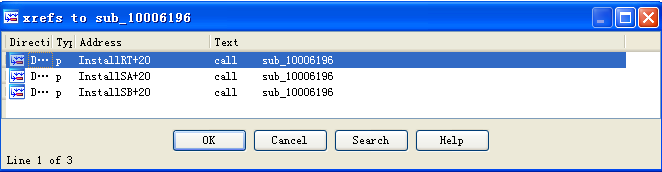
Search→Text，输入in.

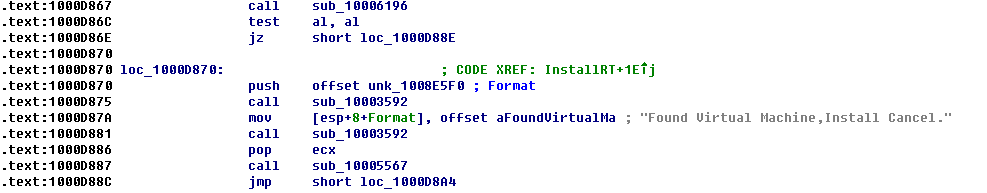


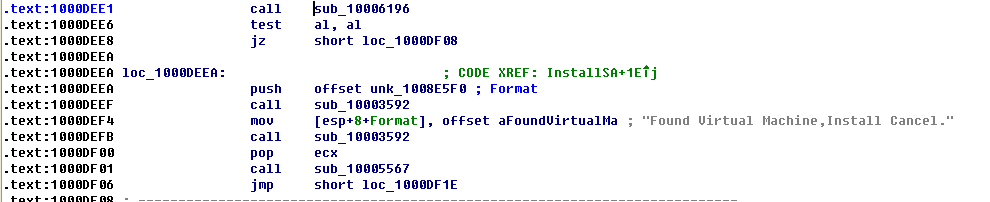
看到在0x100061DB处调用了in指令，在0x100061C7处的mov指令将0x564D5868赋给EAX。这个16进制值相当于ASCII字符串VMXh，该恶意代码采用了反虚拟机技巧。

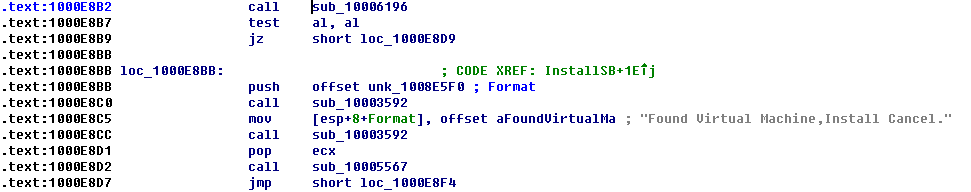


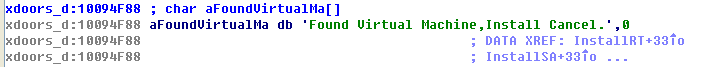
检查sub\_10006196的交叉引用，在调用函数中均发现Found Virtual Machine，Install Cancel字符串。





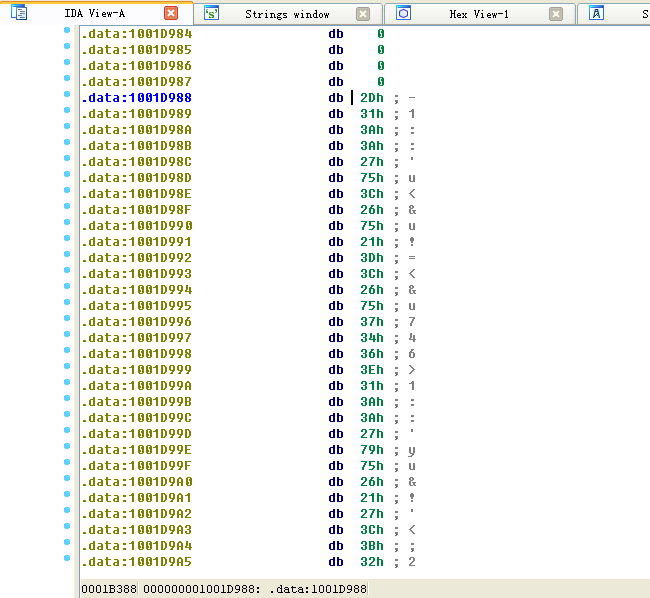






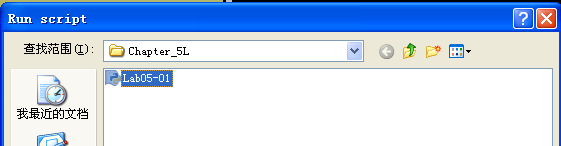
1. **将你的光标跳转到0x1001D988处，你发现了什么?**

G快速跳转到1001D988，看到一些像是随机的数据，没有可读性。

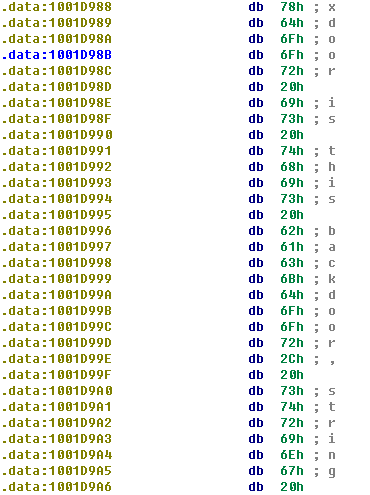


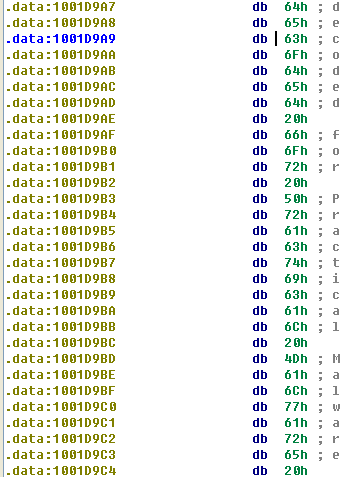
1. **如果你安装了IDA Python插件(包括IDA Pro的商业版本的插件),运行Lab05-01.py,一个本书中随恶意代码提供的IDA Pro Python脚本，(确定光标是在0x1001D988处。)在你运行这个脚本后发生了什么?**

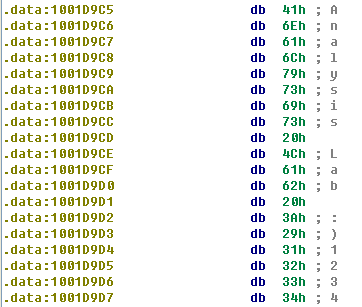
File->Script File，选择Lab05-01 Python脚本运行。



运行后再次查看0x1001D988的内容，变成明文







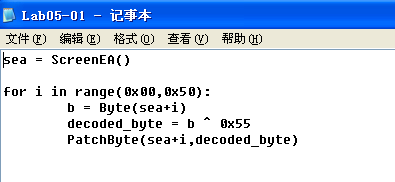
看到这段数据被反混淆得到一个字符串。

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

光标放在0x1001D988处，按下A键，将其变为一个可读的字符串：xdoor is this backdoor, string decoded for Practical Malware Analysis Lab:)1234。



1. **使用一个文本编辑器打开这个脚本。它是如何工作的?**



sea = ScreenEA()用于获得光标当前位置，用作要解码数据的偏移值。

然后从0到0x50循环，调用Byte获得每个字节的值，将这些字节与0x55进行XOR异或操作。

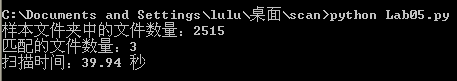
最后将这些字节用PatchByte函数写回IDA Pro的显示中，但不修改原始文件。

1. **Yara**

根据上面的分析，编写如下的yara规则：

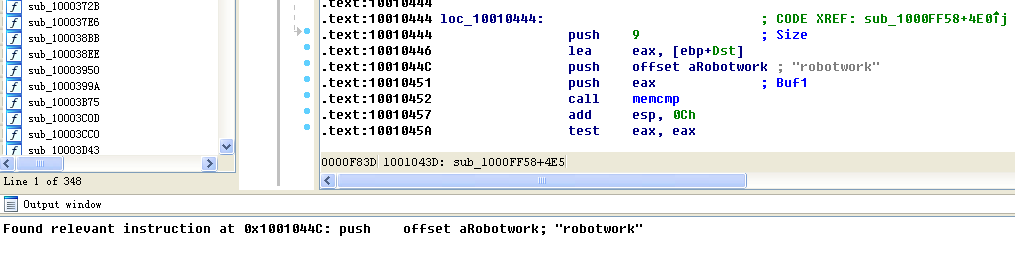
|  |
| --- |
| rule Detect\_Lab05 |
| { |
| meta: |
| description = "YARA rule to detect Lab05 malicious code" |
| strings: |
|  |
| $x1 = "Get DLL FileName Error,Update Failed" fullword ascii |
| $x2 = "Move '%s' To '%s' Failed,Perhaps Other Process Updateing|Updated Same Module" |
| $x3 = "Process '%s' Not Found,Uninject Failed" fullword ascii |
| $x4 = "Uninject '%s' From Process '%s' " fullword ascii |
| $x5 = "error on get process info. " fullword ascii |
| $x6 = "Process '%s' Not Found ,Inject Failed" fullword ascii |
| $x7 = "Inject '%s' To Process '%s' Failed" fullword ascii |
| $x8 = "Old Module Have Been Free,New ModuleName As Old,Will Take Effect Soon." fullword ascii |
| $x9 = "You Specify Service Name Not In Svchost\\netsvcs, must be one of following:" fullword ascii |
| $x10 = "Uninject '%s' From Process '%s' Failed, Module '%s' Not Found" fullword ascii |
| $x11 = "MayBe Inject Mode,You Only Need Close Master Process '%s' To Uninatll This Mode" fullword ascii |
| $x12 = "Resume '%s' To '%s' Failed,Update Failed" fullword ascii |
| $x13 = "Move '%s' To '%s' Failed,Update Failed" fullword ascii |
| $x14 = "Replace Old Service '%s' Failed,Install Cancle" fullword ascii |
| $x15 = "Old Module Have Been Free,New ModuleName As Old,Will Take Effect Soon." fullword ascii |
| $x16 = "Repair Successfully,Need Reboot Machine!" fullword ascii |
| condition: |
| //检查目标文件的前两个字节是否等于 0x5a4d，这是Windows可执行文件（PE文件）的标识符;检查目标文件中偏移为 0x3c 处的四个字节，看是否等于 0x00004550。在PE文件中，这个值是PE头的标识，指示文件的格式;前面的条件中至少有12个条件为真才匹配。 |
| uint16(0) == 0x5a4d and |
| uint32(uint32(0x3c))==0x00004550 and |
| 1 of ($x\*) and 12 of them |
| } |

进行扫描，结果如下

****

1. **IDA Python脚本**

编写脚本解决问题10中查找代码中robotwork字符串的需求。



|  |
| --- |
| import idc |
| import idautils |
|  |
| # 目标注释 |
| target\_string = "robotwork" |
|  |
| # 获取函数的起始和结束地址 |
| start\_address = 0x1000FF58 |
| end\_address = 0x1001073F |
|  |
| # 初始化一个列表来存储包含特定注释的汇编语句 |
| relevant\_instructions = [] |
|  |
| # 遍历指令，查找包含指定字符串的汇编语句 |
| current\_address = start\_address |
|  |
| while current\_address <= end\_address: |
| # 获取当前指令的反汇编文本 |
| disasm = idc.GetDisasm(current\_address) |
|  |
| # 检查是否包含指定字符串 |
| if target\_string in disasm: |
| relevant\_instructions.append((current\_address, disasm)) |
|  |
| # 移动到下一条指令 |
| current\_address = idc.NextHead(current\_address) |
|  |
| # 输出找到的相关汇编语句 |
| for instruction in relevant\_instructions: |
| print("Found relevant instruction at 0x%X: %s" % (instruction[0], instruction[1])) |

1. **实验结论及心得体会**
2. **实验结论**

本次实验通过使用IDA Pro工具，成功分析了Lab05-01.dll文件中的恶意代码。识别了DLL文件的入口点，即DllMain函数，以及重要的导入函数gethostbyname，它在恶意代码中被多次调用。此外，还找到了用于执行命令的关键字符串"\cmd.exe /c"，以及用于检测虚拟机环境的反虚拟机技巧。在分析函数内的代码时，成功识别了局部变量、参数和全局变量，这有助于理解函数的工作原理和目的。此外，发现恶意代码调用了多个Windows API函数，包括网络和文件操作函数。确定了Socket的调用三个参数的含义，分别表示协议域、Socket类型和协议，配置了一个IPv4的TCP连接。

1. **心得体会**

本次实验加深了对逆向工程和恶意代码分析的理解。通过实际操作，学习了如何使用IDA Pro等工具DLL文件。熟悉了IDA Pro工具的基本功能，包括查找函数、查看导入表、分析字符串等。这些技能对于深入研究和理解恶意代码非常有帮助。分析恶意代码，了解了如何追踪函数调用关系、查找字符串、分析代码块等技术，对于理解代码的逻辑和功能非常重要。逆向工程和恶意代码分析是网络安全领域的重要技能，通过实际操作和学习，更深刻地理解了这些技术的应用和重要性。