

**恶意代码分析与防治课程实验报告**

**实验六：windows恶意代码分析**

****

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**

本章覆盖了对恶意代码分析来说重要的Windows概念。诸如进程、线程，以及网络功能的概念，这些概念在你分析恶意代码时会逐渐遇到。

本章讨论过的许多恶意代码例子都很常见，并且你对它们的熟悉程度会允许你快速在恶意代码中识别它们，以便更好地理解程序的总体目的。这些概念对静态恶意代码分析来说是重要的，并且它们会在本书各章的实验中出现，同时也会在现实世界的恶意代码中普遍出现。

1. **实验原理**
   1. 实验环境

Windows xp，VMWARE，Windows11 ,win8.1

* 1. 实验工具

Ida pro 6.6 ida python,yara

* 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。

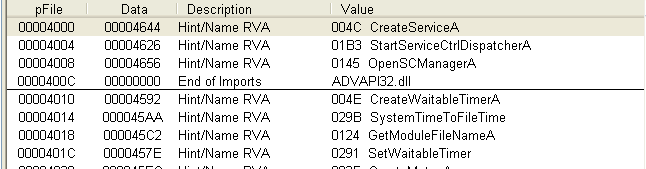
事件驱动：IDA Python还支持事件处理机制，允许用户注册和处理特定的IDA事件。当IDA Pro发生某些事件时，例如载入二进制文件、分析完成、用户交互等，用户可以编写回调函数来响应这些事件，执行相应的操作。

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**

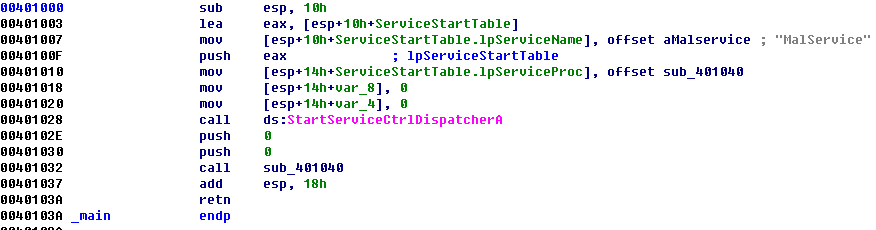
## **Lab 7-1：分析在文件Lab07-01.exe中发现的恶意代码**

## **1.1 当计算机重启后，这个程序如何确保它继续运行（达到持久化驻留）？**



从Advapi32.dll中导入的函数十分可疑，三个函数都与服务相关，从OpenSCManagerA和CreateServiceA函数可以推测出该恶意代码可能会利用服务控制管理器创建一个新服务；StartServiceCtrlDispatcherA函数用于将服务进程的主线程连接到服务控制管理器，这说明该恶意代码确实是个服务（期望自己作为服务运行）。

因此，该恶意代码实现持久化驻留的方法可能是：将自己安装成一个服务，且在调用CreateService函数创建服务时，将参数设置为可自启动。

接下来通过IDA Pro分析具体实现方法。

可以看到该恶意代码的main函数中一上来就调用了StartServiceCtrlDispatcherA函数，该函数被程序用来实现一个服务，指定了服务控制管理器会调用的服务控制函数。从参数可以看出恶意代码安装成的服务名应为“MalService”，指定的服务控制函数为sub\_401040，该子函数会在执行StartServiceCtrlDispatcherA后被调用，双击跳转。

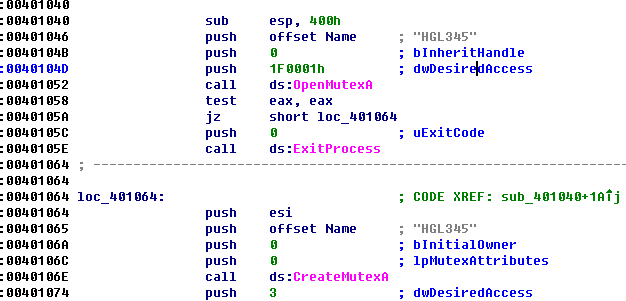


sub\_401040子函数中首先是互斥量相关代码，这部分会在下一问详细讨论，这里先不管。这段代码首先调用OpenSCManager打开一个服务控制管理器的句柄，然后调用GetCurrentProcess获取当前进程的伪句柄，紧接着调用GetModuleFileName函数，并传入刚获取的恶意代码进程伪句柄，从而获取恶意代码的全路径名，这个全路径名被传入CreateServiceA函数，从而将该恶意代码安装成一个名为“Malservice”的服务。此外，CreateServiceA函数的参数中，dwStartType=2即SERVICE\_AUTO\_START使服务为自启动，这样即实现了持久化驻留，即使计算机重启，也能维持运行。

**1.2 为什么这个程序会使用一个互斥量？**

  通常，恶意代码使用互斥量都是为了确保该计算机上只有一个恶意代码实例在运行。

  直接看代码，互斥量相关代码在上面提到的sub\_401040子函数的一开始。



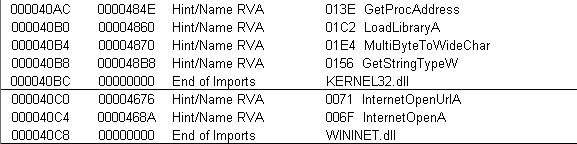
互斥量是全局对象，该恶意代码使用的互斥量的硬编码名为“HGL345”，首先调用OpenMutexA函数尝试访问互斥量，如果访问失败则会返回0，于是jz指令使跳转到loc\_401064处，调用CreateMutexA函数创建名为“HGL345”的互斥量；若打开互斥量成功，则说明已经有一个恶意代码实例运行并创建了互斥量，于是调用ExitProcess函数退出当前进程。

1.3 可以用来检测这个程序的基于主机特征是什么？

  由前两问分析可知有两个基于主机特征：名为“Malservice”的服务、名为“HGL345”的互斥量。

1.4 检测这个恶意代码的基于网络特征是什么？

  还是先通过PEview查看导入函数，看该恶意代码使用了哪些网络相关函数。

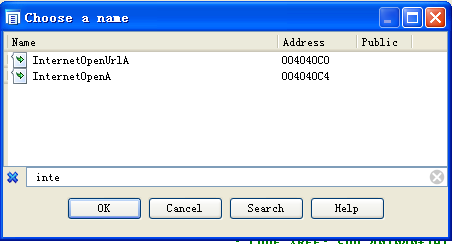


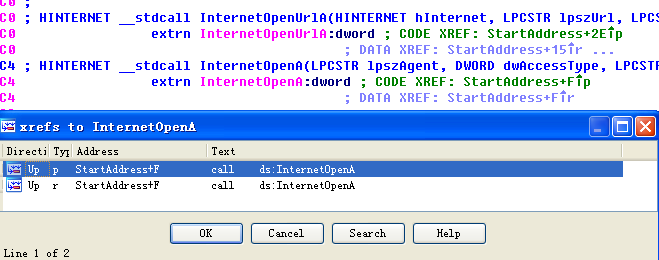
该恶意代码从WinINet.dll中导入了InternetOpenUrlA和InternetOpenA函数。InterOpen函数用于初始化一个到互联网的连接；InternetOpenUrl函数能访问一个URL（可以是一个HTTP页面或一个FTP资源）。

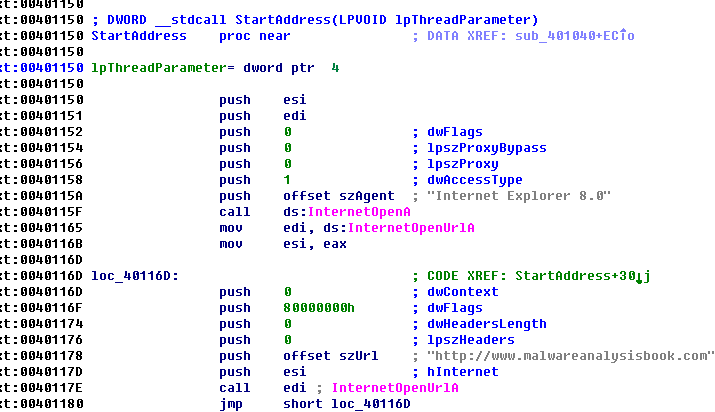
回到IDA Pro，按Ctrl+L打开Name表，按Ctrl+F在表中呼出搜索框，输入InternetOpenA。

双击跳转，按Ctrl+X查看交叉引用。

这两条实际上指向同一处引用，双击跳转，可以看到对InternetOpenA和InternetOpenUrlA的调用都在这个StartAddress子函数中。



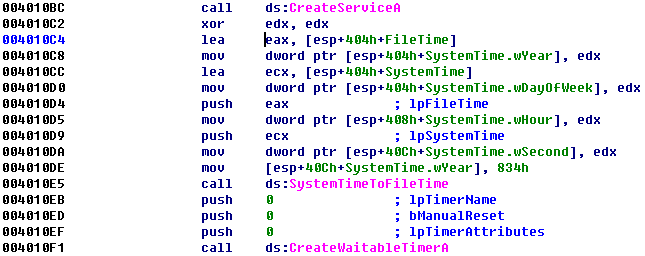




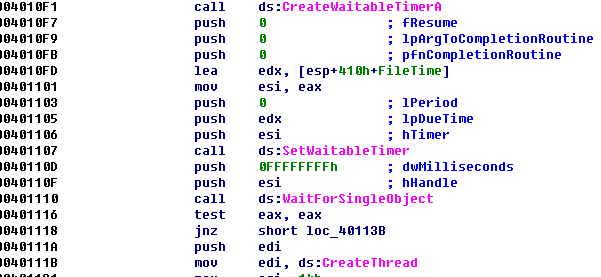
可以看到InternetOpenA函数的szAgent参数，即使用的代理服务器为Internet Explorer 8.0，而InternetOpenUrlA函数要访问的地址是http://www.malwarenanlysisbook.com。这两个就是该恶意代码基于网络的特征。

1.5 这个程序的目的是什么？

  根据上面的分析，该恶意代码首先调用了StartServiceCtrlDispatcherA函数，然后会调用子函数sub\_401040，在该子函数中先利用互斥量确定了只有一个恶意代码实例在运行，然后将自己安装成了一个可自启动的服务，再看看接下来做了什么。

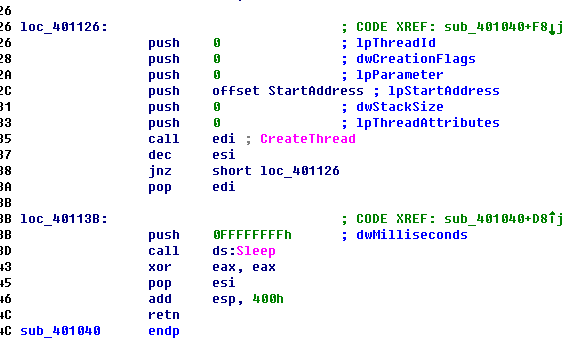


首先调用了SytemTimeToFileTime函数，该函数用来将时间格式从系统时间格式转换为文件时间格式，它的参数即为要转换的时间，可以看到IDA Pro已经识别出了一个SystemTime结构体。先将edx值，即0，赋给wYear、wDayOfWeek、wHour、wSecond即年、日、时、秒，然后将wYear值设置为834h即2100，这个时间代表2100年1月1日0点。

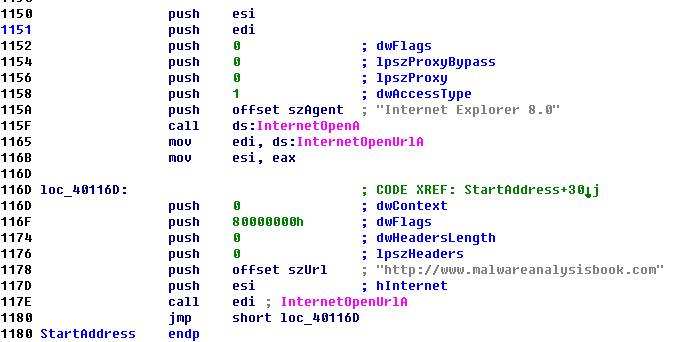


将上述时间点转换为文件时间类型后，先调用了CreateWaitableTimerA函数创建定时器对象，然后调用SetWaitableTimer函数设置定时器，其中参数lpDueTime为上面转换的文件时间结构体，最后调用WaitForSingleObject函数等待计时器对象变为有信号状态或等待时间达到FFFFFFFFh毫秒（这当然是达不到的时间），也就是说会等到2100年1月1日0点然后函数返回继续往下执行。

恶意代码将开始一段循环（典型的for循环结构），循环次数为14h即20次，每次循环都创建一个执行StartAddress子函数的线程。



而由1.4中的分析，StartAddress函数会以Internet Explorer 8.0位代理服务器访问http://www.malwarenanlysisbook.com，且网址访问代码是在一个无限循环中，也就是说每一个执行StartAddress函数的线程都会无限次持续访问目标网址。for循环结束后，按执行顺序同样也进入loc\_40113B处，休眠FFFFFFFFh毫秒。



综上所述，这个恶意代码的目的就是：将自己安装成一个自启动服务，保证只要计算机开启，恶意代码就在运行，然后等到2100年1月1日0点，开启20个线程无限次持续访问http://www.malwarenanlysisbook.com，该恶意代码可能是用来对目标网址进行DDoS攻击。

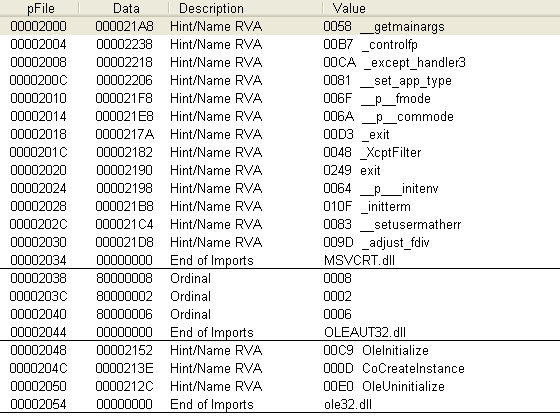
1.6 这个程序什么时候完成执行？

  正如1.5中的分析，每个访问网址的线程都会无限循环访问目标网址，这个程序不会完成执行。

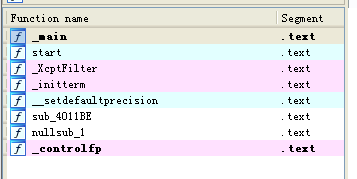
Lab 7-2：分析在文件Lab07-02.exe中发现的恶意代码

2.1 这个程序如何完成持久化驻留？

  依然先使用PEview查看导入函数表，通过导入函数推测下可能使用的持久化驻留方法。



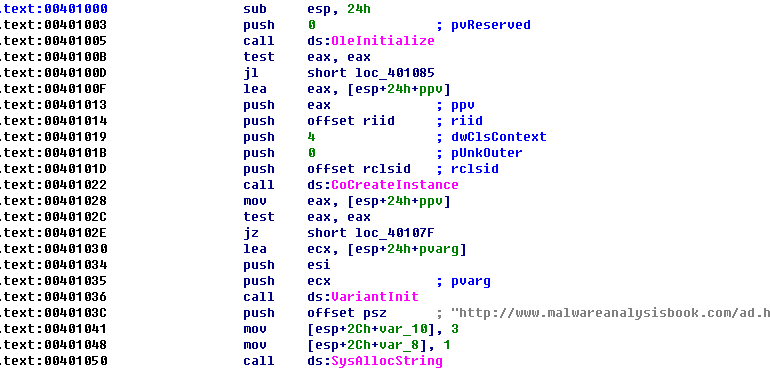
没有[注册表](https://so.csdn.net/so/search?q=%E6%B3%A8%E5%86%8C%E8%A1%A8&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/qq_43443410/article/details/_blank)相关函数、没有服务相关函数、没有任何可能可用来实现持久化驻留的函数，无法推测了，只能直接打开IDA Pro看代码。



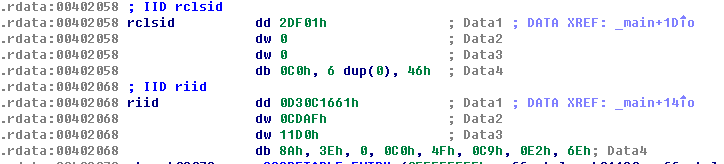
该恶意代码的子函数很少，并没有哪里有持久化驻留相关代码，因此这个恶意代码没有实现持久化驻留。

**2.2 这个程序的目的是什么？**

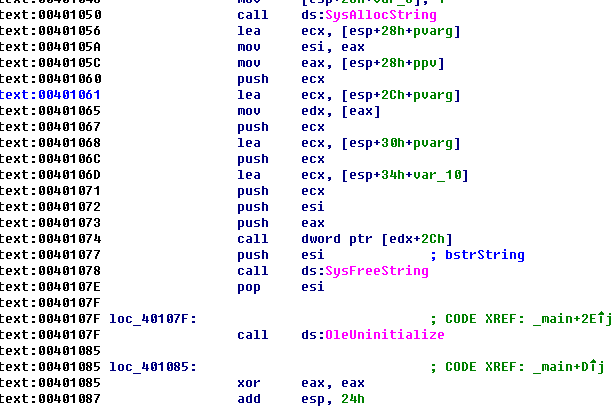
  对\_main中代码进行分析。



首先调用了OleInitialize函数，这意味着该恶意代码要使用组件对象模型COM功能；然后调用了CoCreateInstance函数来获取对COM功能的访问，查看rclsid和riid



可得到类型标识符CLSID值为0002DF01-0000-0000-C000-000000000046，这代表Inter Explorer；接口标识符IID值为D30C1661-CDAF-11D0-8A3E-00C04FC9E26E，这代表IWebBrowser2接口。



然后调用VariantInit函数初始化变量，再调用SysAllocString函数为字符串“http://www.malwareanalysisbook.com/ad.html”分配内存。

ppv指向了COM对象的位置，mov edx,[eax]指令使edx指向COM对象基地址，call dword ptr [edx+2Ch]调用了IWebBrowser2接口偏移0x2Ch即44处的函数，每个函数地址占4字节，也就是调用了序号11即第12个函数，这个函数是Navigate函数，它允许一个程序启动Internet Explorer并访问一个Web地址，而压入的参数esi中保存着调用SysAllocString分配了内存空间的字符串“http://www.malwareanalysisbook.com/ad.html”。



访问完该网址后，程序正常退出。

  也就是说该恶意代码的目的就是访问一个网址，从网址来看是一个广告页面。

2.3 这个程序什么时候完成执行？

从2.2中分析来看，这个恶意代码在访问了一个广告页面后就正常退出完成执行了，并没有其他操作。

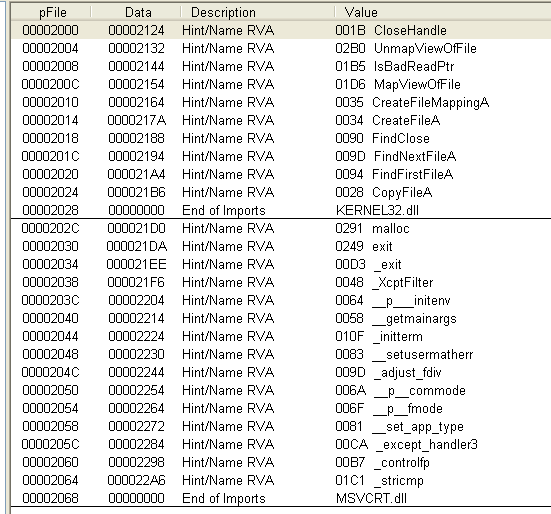
Lab 7-3

对于这个实验，我们在执行前获取到恶意的可执行程序，Lab07-03.exe，以及DLL，Lab07-03.dll。声明这一点很重要，这是因为恶意代码一旦运行可能发生改变。两个文件在受害者机器上的同一个目录下被发现。如果你运行这个程序，你应该确保两个文件在分析机器上的同一个目录中。一个以127开始的IP字符串（回环地址）连接到了本地机器。（在这个恶意代码的实际版本中，这个地址会连接到一台远程机器，但是我们已经将它设置成连接本地主机来保护你。）

这个实验可能比前面那些有更大的挑战。。你将需要使用静态和动态方法的组合，并聚焦在全局视图上，避免陷入细节。

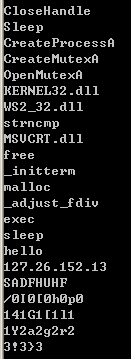
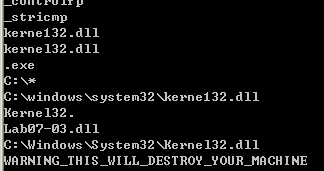
3.1 这个程序如何完成持久化驻留，来确保在计算机被重启后它能继续运行？

还是先看一下exe文件的导入函数表，没有发现注册表、服务相关函数，但有一些文件相关函数。



FindFirstFileA和FindNextFileA函数意味着该恶意代码可能有遍历某一目录查找文件的行为；又调用了CopyFileA函数，说明可能会复制找到的目标文件（到另一希望的目录下甚至修改为其他名字等）；CreateFileA用于创建或打开文件，又调用了CreateFileMappingA和MapViewOfFile函数说明恶意代码可能会打开一个文件并将其映射到内存中。但有意思的是，该恶意代码没有导入Lab07-03.dll及其中的函数，这说明dll文件的作用并不是为exe文件提供特制函数。

  再使用String.exe查看字符串信息，将Lab07-03.exe复制到strings.exe同一目录下，使用命令string Lab07-03.exe，除了无意义字符串、导入表相关信息外，比较可疑、有意思的字符串如下：

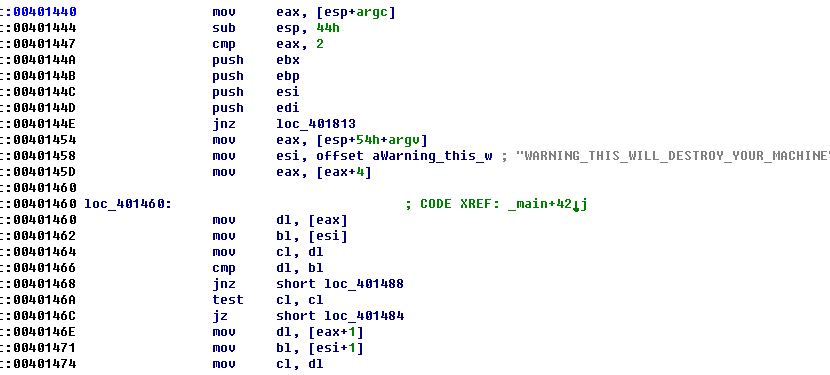


最引人注意的是“kerne132.dll”，该字符串将“kernel32.dll”中的字母“l”修改为了数字“1”，显然这是为了伪装恶意代码文件名，使其不容易被发现，在临近位置还出现了“Lab07-03.dll”，再结合上面对导入函数的分析以及路径字符串“C:\windows\system32\kerne132.dll”，可推测：该恶意代码可能会遍历当前目录，找到Lab07-03.dll文件，将其复制到C:\windows\system32\下并将名字改为kerne132.dll。

  再查看Lab07-03.dll的导入函数表：

从Kernel32.dll中导入了进程创建函数CreateProcessA和互斥量相关函数；另外还按序号导入了ws2\_32dll中的函数，这是Winsock库，里面的函数与网络相关，因此该dll应该有网络访问相关操作；此外，该DLL文件并没有什么导出函数。

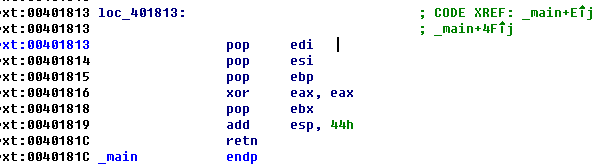
  再用string.exe查看字符串信息，除导入函数等信息外主要是以下字符串



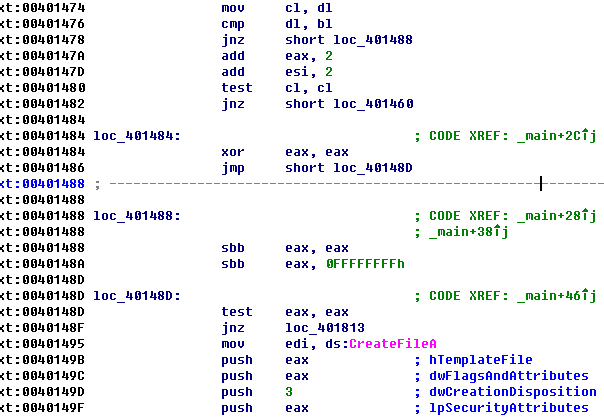
exec、sleep应该是两个函数，可能分别是执行命令、休眠功能，最重要的是“127.26.152.13”，这显然是一个IP地址，再结合前面说到该dll文件导入了ws2\_32.dll会有网络相关行为，该dll很可能会访问这个IP地址。

但分析到这，还是看不出该恶意代码实现持久化驻留的方法是什么（其他可能的行为倒是推测了不少…），还是得用IDA Pro看代码。

main函数一开始先将argc值与2进行比较，检查了运行这个exe文件的命令行参数是否为2，如果不是，则会跳转至loc\_401813处：



这使程序直接结束，也就是说如果我们尝试进行动态调试时采用双击运行的方式，程序会直接退出。然后该程序获取命令行参数地址列表argv[]，并通过偏移（[eax+4]，每个地址4字节）将argv[1]（第二个参数）保存在eax中，此外，还将一字符串“WARNING\_THIS\_WILL\_DESTROY\_YOUR\_MACHINE”保存在esi中，接下来可能会对两者进行操作。



紧接着在loc\_401460处出现了一个循环，该循环用于比较eax和esi中值是否相同，也就是比较运行该恶意代码的第二个命令行参数是否是字符串“WARNING\_THIS\_WILL\_DESTROY\_YOUR\_MACHINE”。若不相同，则会跳转至loc\_401388处，在这里对eax进行操作使eax不为0。

若遍历完参数到末尾，即两者相同，会跳转到loc\_401484处，使eax值为0。

无论是否相同，最后都会执行到loc\_40148D处（正常按序执行到此处或通过跳转），唯一不同的是：若不相同，则eax值不为0；若相同，则eax值为0。于是loc\_40148D处首先就检查eax值是否为零：

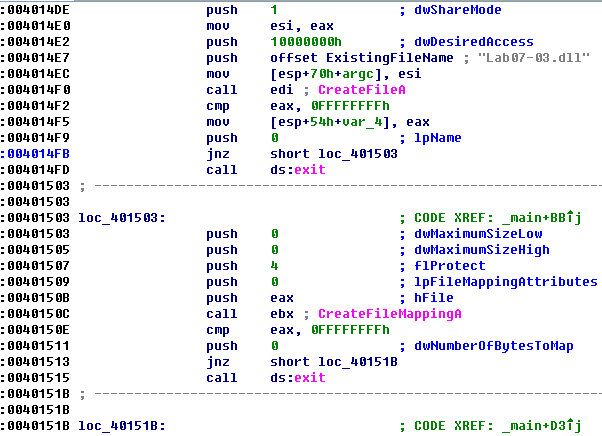
若不为零，即参数与字符串不相等，则跳转至loc\_401813处，会使程序直接退出。

也就是说这个恶意代码首先会检查命令行参数是否有两个，然后检查第二个参数是否为指定字符串，若不满足，则恶意代码都会直接退出。因此使恶意代码正常运行的方式应该是：在命令行中输入命令[Lab07-03.exe路径] WARNING\_THIS\_WILL\_DESTROY\_YOUR\_MACHINE。

接下来我们再看恶意代码正常运行后会做哪些事。



通过调用CreateFileA、CreateFileMappingA、MapViewOfFile来打开Kernel32.dll并将其映射到了内存。

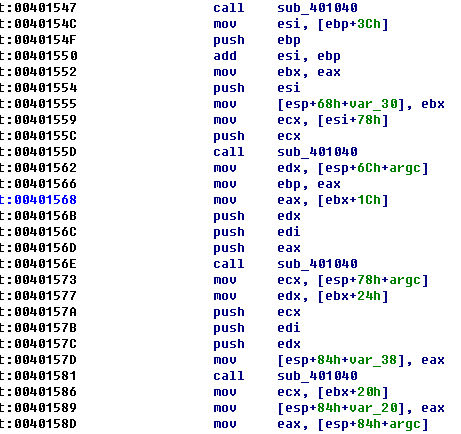


然后同样的方法打开了Lab07-03.dll并映射到内存。



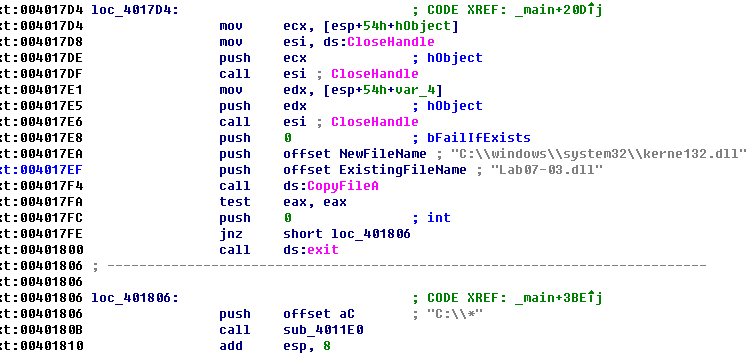
从loc\_401538开始是大量的mov、push操作，这些操作是在干嘛并不能轻易看懂，但在这些mov、push操作中夹杂了几次call指令，每一组mov、push操作后都会调用一次sub\_401040，几次调用之后还调用了sub\_401070,这两个函数也都是在进行各种内存操作，并不能清楚地分析出操作地目的是什么，但结合前面恶意代码将两个dll文件打开并映射到内存中，这些操作应该是在对它们进行操作。

 跳出这两个调用继续往下看，仍旧是非常多的内存操作，直到loc\_4017D4处，这里开始出现了Windows API调用以及重要字符串信息。



从先前调用了两次CreateFileA函数后对返回值的处理可以知道：hObject和var\_4分别保存了打开两个文件的句柄，因此这里先调用的两次CloseHandle是将两个被打开文件的句柄关闭，这意味着恶意代码完成了对文件内存映射的编辑操作并保存回文件。

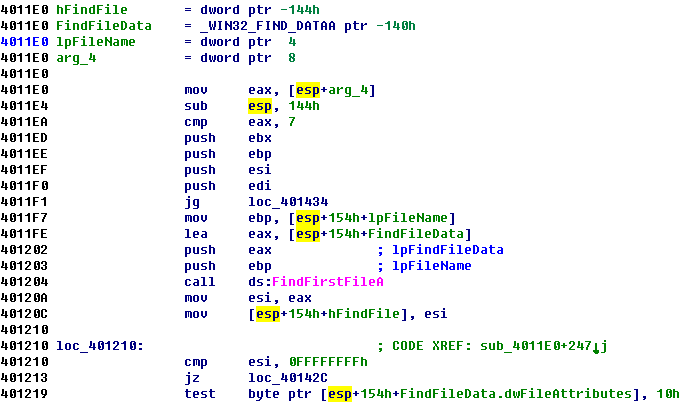
然后调用了CopyFileA，从参数来看，是将Lab07-03.dll复制到C:\windows\system32\下，并命名为kerne132.dll。



在接下来的loc\_401806处又出现了一次重要的函数调用：调用了sub\_4011E0子函数，传入的参数为字符串“C:\\*”，接下来双击跳转至该子函数查看。



首先要注意，传入的参数被标记为lpFileName。

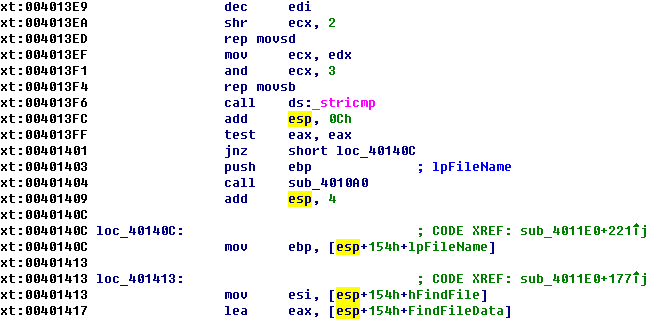


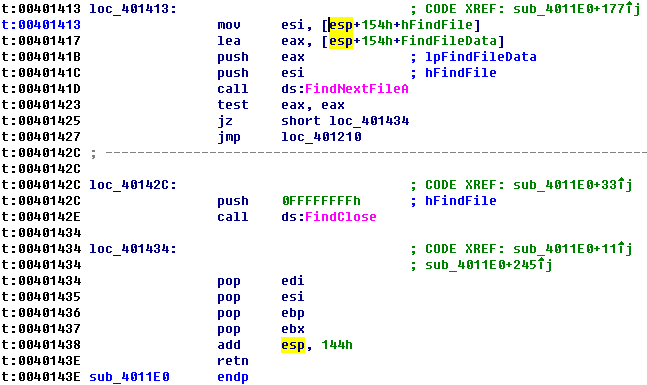
先调用了FindFirstFile，在C:\下查找第一个文件或目录并返回其句柄。

然后是大量的比较、算术运算等指令，难以轻易看出其目的，暂不分析。这些代码中夹杂着可能的两个malloc函数、可能的一次sub\_4011E0调用，也就是我们正在分析的这个子函数，说明该子函数可能出现递归调用。比较重要的是后面出现了一次stricmp函数调用。

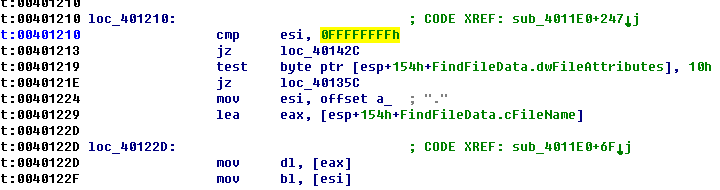


stricmp函数的参数压栈指令在上面一段距离处，两个参数分别是字符串“.exe”和调用FindFirstFile函数返回的FindFileDta结构中的dwReserved1字段，该字段是系统保留字段，然后stricmp函数比较两者是否相同，相同则返回值为0，于是两字符串相同则不跳转值loc\_40140C，而是压栈lpFileName然后调用sub\_4010A0，这里先不直接分析sub\_4010A0函数，而是看看调用sub\_4010A0后还做了什么。





紧接着便是调用FindNextFileA函数，我们之前就说过：FindFirstFile和FindNextFile函数结合使用可以遍历目录，而在调用了FindNextFileA函数后只要返回值不为0（没有遍历完）就会跳转回loc\_401210处，

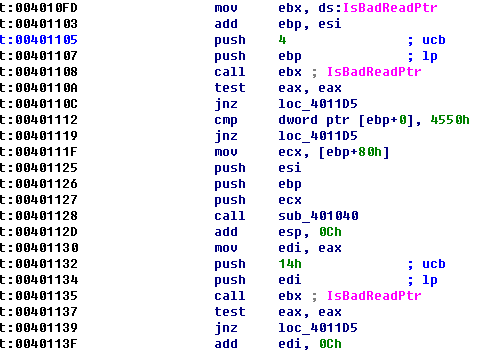


该位置在刚调用完FindFirstFileA函数处。也就是说sub\_4011E0函数用来遍历C:\，查找.exe文件，只要发现了一个.exe文件就调用一次sub\_4010A0，而之前说的可能出现递归，则应该是由于子目录的存在，遍历到一个子目录便会有一次递归调用sub\_4011E0出现。

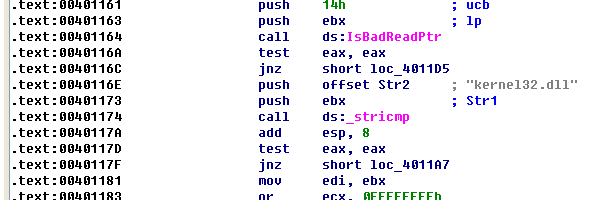
 接下来我们再看一下sub\_4010A0函数的作用。



首先是CreateFileA、CreateFileMappingA、MapViewOfFile函数的组合调用，这意味着该函数先是将传入的参数（文件路径）打开并映射到内存中，恶意代码接下来大概率会直接对内存进行操作实现文件修改，而不调用其他Windows API，这使我们会难以分析恶意代码具体做出了什么修改。

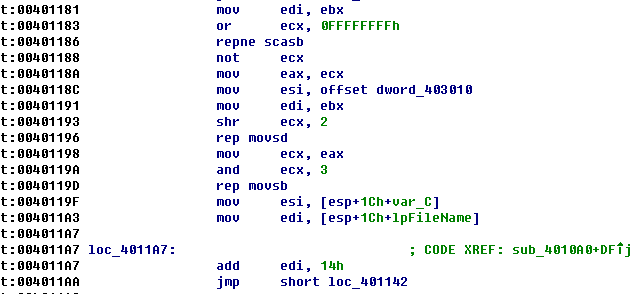


然后是四次IsBadReadPtr函数的调用，该函数用来检查进程是否有权限访问指定的内存块，即检查指针是否有效。

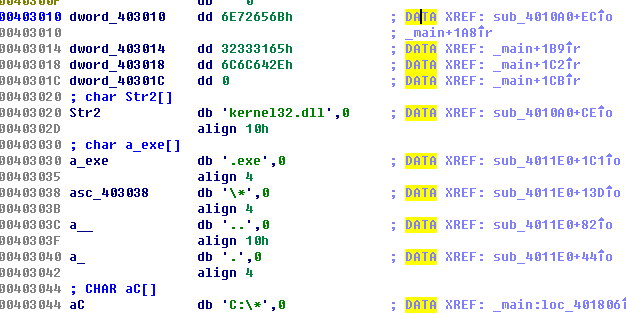


紧接着调用了一次stricmp函数检查一个字符串是否为“kernel32.dll”，这个字符串也是通过内存地址、偏移获取的，并不好分析具体是什么位置的一个字符串。

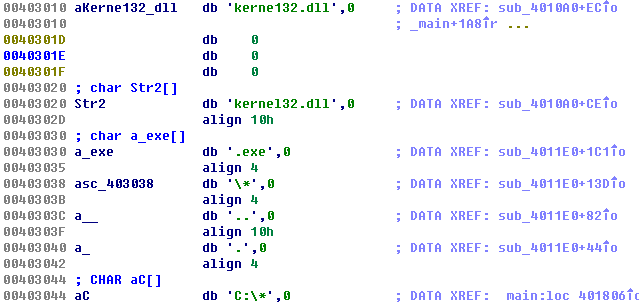
然后虽然没有出现API调用，但是有三个十分特殊的指令：repne scasb、rep movsd、rep movsb。首先是repne scasb，00401183至0040118A的代码配合使用来计算字符串长度，or指令设置循环次数为-1，然后repnz scasb一直重复搜索到edi中地址指向的字符串末尾的0，最后eax中保存字符串长度。而ebx值赋给了edi，ebx中保存的是上面那个Str1字符串的地址。



rep movsb指令，这个指令一般也是用来搬移字符串，但是看上去在这里并没有发挥实质性作用。



然后是rep movsd指令，rep是重复执行，前面shr指令设置重复次数，每次ecx不等于0便执行一次movsd指令，0040118C至00401196的代码配合使用来将dword\_403010处的ecx个dword复制到ebx中保存的地址处。先来看看这个要复制的字符串内容是什么，双击dword\_403010跳转，

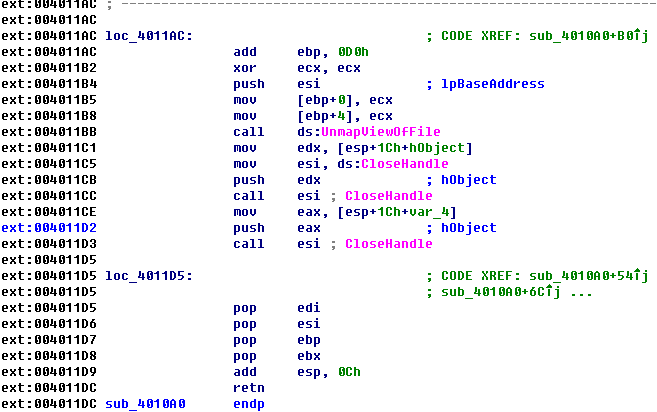


可以看到这个字符串是“kerne132.dll”。而我们先前分析了，文件加载到内存中后某处的Str1先被用来跟“kernel32.dll”作比较，再结合此处，可以推测这部分代码是要将一个.exe文件中的“kernel32.dll”字符串特换成“kerne132.dll”。

再往下比较重要的是一个向上的跳转，这意味着可能有一个循环，看看跳转到哪里，向上跳转的目的地是最后一个IsBadReadPtr函数调用前，且若IsBadReadPtr检测到指针不合法，就会跳出该循环，如果合法则调用stricmp函数比较字符串是否为“kernel32.dll”，然后再正常向下执行，这意味着该子函数应该是将文件打开映射到内存中后，遍历这一段内存空间（或其中某一段），查找“kernel32.dll”字符串，查找到后则将该字符串替换为“kerne132.dll”。

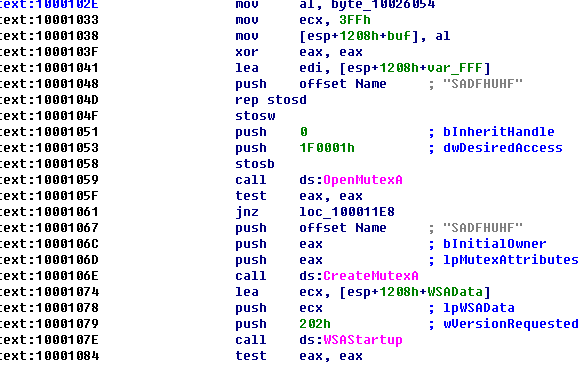
最后是关闭映射和句柄、做函数返回前的清理工作并返回。

  现在我们可以对这个Lab07-03.exe做的事做一个总结：它首先将Lab07-03.dll复制到C:\Windows\System32\下并重命名为kerne132.dll，然后将扫描C:\下所有文件，找出.exe文件并将其中的“kernel32.dll”字符串全部修改为“kerne132.dll”，而.exe文件中出现kernel32.dll字符串的情况一般都是该文件要导入kernel32.dll中的函数，因此这样做使C盘下的.exe文件在试图导入kernel32.dll的函数时都去加载kerne132.dll。这便是该恶意代码实现持久化驻留的方法。



**3.2 这个恶意代码的两个明显的基于主机特征是什么？**

由上述分析，一个明显的特征就是它会使用一个硬编码的文件名“kerne132.dll”，此外Lab07-03.exe中就没有其他明显的基于主机特征了，但回想起最开始的静态分析，通过PEview看到了Lab07-03.dll中导入了互斥量相关函数，而互斥量常采用硬编码命名，这可能是个不错的特征。接下来就去看看相关代码。



Lab07-03.dll的DllMain中一上来就使用了互斥量相关函数，尝试打开（创建）的互斥量采用硬编码命名，名字为“SADFHUHF”，这是另一个明显的基于主机特征。

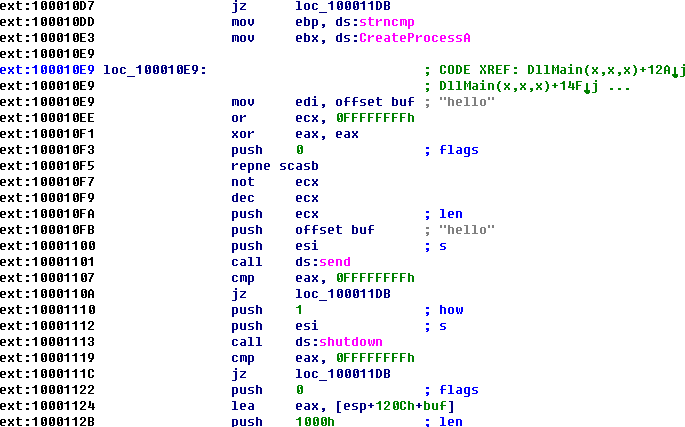
**3.3 这个程序的目的是什么？**

  根据先前的分析，Lab07-03.exe的主要功能是实现持久化驻留，起一个辅助作用，程序的只要目的应还是在Lab07-03.dll中实现，接着上面互斥量相关代码继续往下分析。

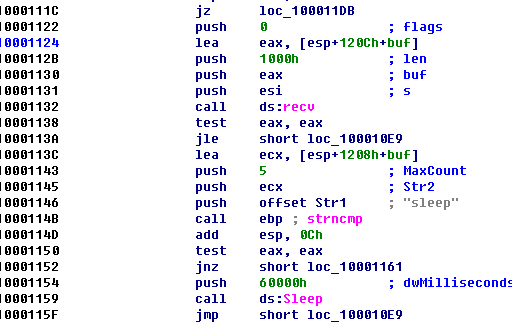
DLL程序确定了只有一个恶意代码实例在运行后，立马调用了WSAStartup函数，在该章的学习中我们知道：如果想调用ws2\_32.dll（Winsock库）中的函数，必须先调用WSAStartup函数初始化Win32 sockets系统——该函数的调用意味着接下来要使用Winsock API了。



先是调用socket函数创建套接字；然后可以看到出现了一个IP地址字符串“127.26.152.13”，inet\_addr函数将其转换成一个无符号长整型数；调用htons函数前指定了端口号50h，也就是80端口，再用该函数将整型端口号变量从主机字节顺序转变成网络字节顺序；最后调用connect函数向127.26.152.13下的远程套接字打开一个连接。

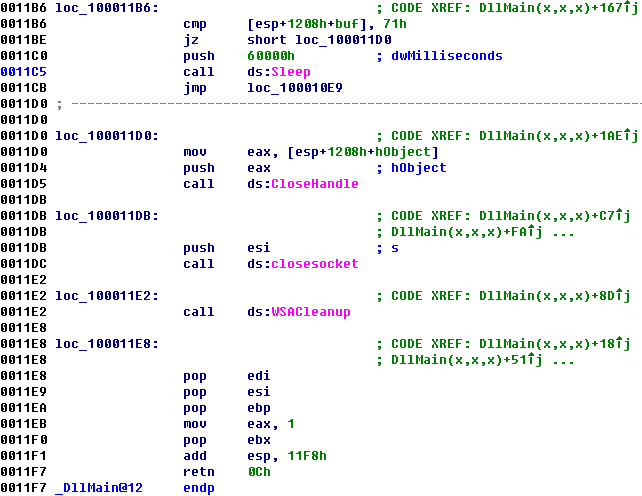


不过并没有马上调用那两个函数。可以看到buf中储存着字符串“hello”，然后调用send函数将该字符串发送到远程服务器端。发送完“hello”后便调用recv函数从远程套接字接收数据并保存在buf中。然后将先比较接收到的数据中前5个字符是否为“sleep”，如果是则会调用Sleep函数休眠60000h毫秒再跳转到loc\_100010E9处；如果不是，则跳转到loc\_10001161处。





如果开头不是“sleep”则会再检查开头4个字符是否为“exec”，如果是的话则会调用CreateProcessA函数创建一个进程，压进栈的CreateProcessA函数参数中最重要的是lpCommandLine，它的值为CommandLine，指明了要打开什么文件来创建进程，即要执行的文件路径，但跟踪CommandLine向上看，并没有发现它在什么地方被赋值，但有一个值得的注意的点是CommandLine在内存空间中的位置为10001010。我们说CommandLine中应该存储可执行文件路径，而其位置在buf起始地址5个字节后，“exec”为4个字节，如果远程服务器发来的指令是“exec [FilePath]”，“exec ”（exec加空格）刚好为5个字节，那么此时CommandLine中就存储了[FilePath]，这样一切就说得通了。只要发来的指令开头是“sleep”或“exec”，恶意代码都会执行相应函数然后跳转到loc\_100010E9处，再发送一遍“hello”然后等待远程指令。如果都不是，则会跳转到loc\_100011B6处执行，它会检查发送来的指令是否为71h对应的字符“q”，



如果是的话就跳转到loc\_100011D0处，关闭句柄、socket，清理空间，结束进程。如果不是的话，则会休眠60000h毫秒，然后跳转到loc\_100010E9处，开始新一轮的发送“hello”、等待指令。因此，该恶意代码的作用就是：Lab07-03.exe将Lab07-03.dll后门dll文件安装到计算机中，并让C盘下所有exe文件链接该dll文件，该后门会连接远程主机并接收命令，能够睡眠或创建进程。

3.4 一旦这个恶意代码被安装，你如何移除它？

  首先肯定是要把kerne132.dll删除，不过C盘下所有exe文件都链接了kerne132.dll，只能是写个脚本来遍历C盘下的exe文件，再给它修改回去。但最简单的能让这些exe在本机上可以正常运行的方法是：复制一份kernel32.dll并把它重命名为kerne132.dll。

**Yara规则的编写**

通过对三个文件的字符串的提取，我们可以明显的看到其中的字符集，因此编写了下列的yara规则：

import "pe"

rule UrlRequest {

strings:

$http = "http"

condition:

$http

}

rule Explorer {

strings:

$name = "Internet Explorer"

condition:

$name

}

rule kerne132 {

strings:

$dll\_name = "kerne132.dll"

condition:

$dll\_name

}

rule EXE {

strings:

$exe = /[a-zA-Z0-9\_]\*.exe/

condition:

$exe

}

rule scanC {

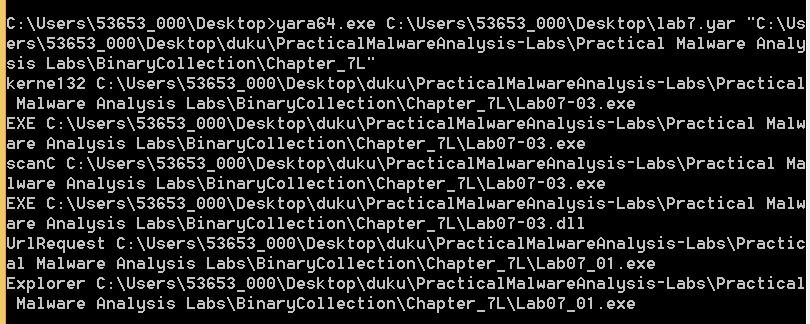
strings:

$c = /C:./

condition:

$c

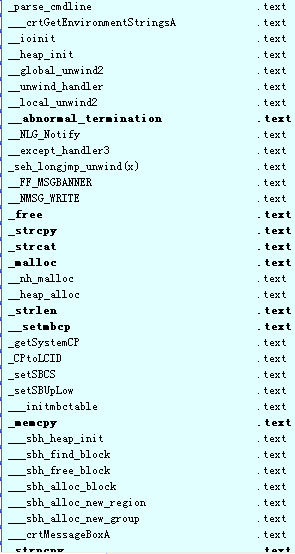
}



**Ida python的编写**

通过观察三个lab里面的函数名，我们可以根据相应的函数编写ida python脚本。

运行脚本即可搜索函数当中的字符串，得到目标结果。

Lab07-01.exe  
#coding:utf-8 from idaapi import \*

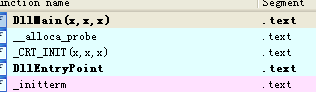
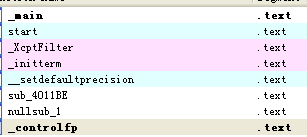
设置颜色

def judgeAduit(addr): ''' not safe function handler ''' MakeComm(addr,"### AUDIT HERE ###") SetColor(addr,CIC\_ITEM,0x0000ff) #set backgroud to red pass

# 函数标识

def flagCalls(danger\_funcs): ''' not safe function finder ''' count = 0 for func in danger\_funcs:  
faddr = LocByName( func )  
if faddr != BADADDR: # Grab the cross-references to this address  
cross\_refs = CodeRefsTo( faddr, 0 )  
for addr in cross\_refs: count += 1 Message("%s[%d] calls 0x%08x\n"%(func,count,addr))  
judgeAduit(addr)

if **name** == '**main**': ''' handle all not safe functions ''' print "-------------------------------" # 列表存储需要识别的函数 danger\_funcs = ["\_\_\_crtGetEnvironmentStringsA","\_\_heap\_init","\_\_abnormal\_termination","\_free","RtlUnwind"] flagCalls(danger\_funcs)



Lab07-02.exe

#coding:utf-8

from idaapi import \*

# 设置颜色

def judgeAduit(addr):

'''

not safe function handler

'''

MakeComm(addr,"### AUDIT HERE ###")

SetColor(addr,CIC\_ITEM,0x0000ff) #set backgroud to red

pass

# 函数标识

def flagCalls(danger\_funcs):

'''

not safe function finder

'''

count = 0

for func in danger\_funcs:

faddr = LocByName( func )

if faddr != BADADDR:

# Grab the cross-references to this address

cross\_refs = CodeRefsTo( faddr, 0 )

for addr in cross\_refs:

count += 1

Message("%s[%d] calls 0x%08x\n"%(func,count,addr))

judgeAduit(addr)

if \_\_name\_\_ == '\_\_main\_\_':

'''

handle all not safe functions

'''

print "-------------------------------"

# 列表存储需要识别的函数

danger\_funcs = ["\_XcptFilter","\_initterm","nullsub\_1","\_except\_handler3","\_controlfp"]

flagCalls(danger\_funcs)

Lab07-03.exe

#coding:utf-8 from idaapi import \*

设置颜色

def judgeAduit(addr): ''' not safe function handler ''' MakeComm(addr,"### AUDIT HERE ###") SetColor(addr,CIC\_ITEM,0x0000ff) #set backgroud to red pass

函数标识

def flagCalls(danger\_funcs): ''' not safe function finder ''' count = 0 for func in danger\_funcs:  
faddr = LocByName( func )  
if faddr != BADADDR: # Grab the cross-references to this address  
cross\_refs = CodeRefsTo( faddr, 0 )  
for addr in cross\_refs: count += 1 Message("%s[%d] calls 0x%08x\n"%(func,count,addr))  
judgeAduit(addr)

if name == 'main': ''' handle all not safe functions ''' print "-------------------------------" # 列表存储需要识别的函数 danger\_funcs = ["start","\_XcptFilter","\_initterm","\_\_setdefaultprecision","\_controlfp"] flagCalls(danger\_funcs)

1. **实验结论及心得体会**

这一次的实验是恶意代码与防治分析的Lab7实验，对理论课上讲的IDA Python编写技术有了一定的了解，也对IDA Pro的使用比如说交叉引用、语句跳转、反汇编分析等更加的熟练。在本次实验中，也对所检测程序编写了相应的yara规则，对于yara规则的编写也更加的熟练。通过本次实验，也知道了一些应用程序的代码结构和其基本功能，认识到自己作为一名信息安全专业学生的责任，需要我们用更加严谨认真的态度学习新的知识和思路。

未来我还将联系操作系统和本课程当中关于进程和线程的相关知识，对windows恶意代码的特殊作用进行进一步的讨论。