****

**恶意代码课程实验报告**

**实验十一：恶意行为**

****

**学 院 网络安全学院**

**专 业 信息安全**

**学 号 2110688**

**姓 名 史文天**

**班 级 1063**

1. **实验目的**

**完成课本Lab11的实验内容，编写Yara规则，并尝试IDA Python的自动化分析。**

1. **实验原理**

**配置XP虚拟机环境，安装动态、静态分析工具，使用静态和动态分析工具。**

1. **实验过程**

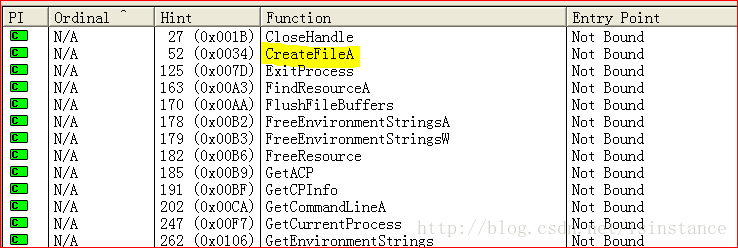
**Lab11-01**

**1.这个恶意代码向磁盘释放了什么？**

**解答： 我们刚刚完成了Windows内核病毒的分析，包括了一些过时的RootKit技术的分析，现在我们回归[二进制](https://so.csdn.net/so/search?q=%E4%BA%8C%E8%BF%9B%E5%88%B6&spm=1001.2101.3001.7020)分析**

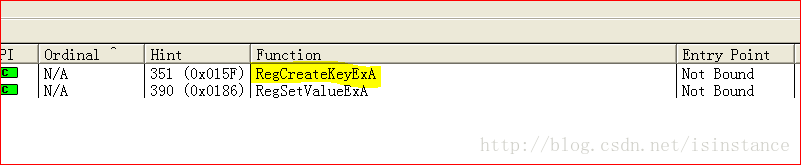
**这里要分析的文件是Lab11-1这个文件，我们先做一些基本的静态分析**

**我们先看KERNEL32.DLL这个导出DLL**

****

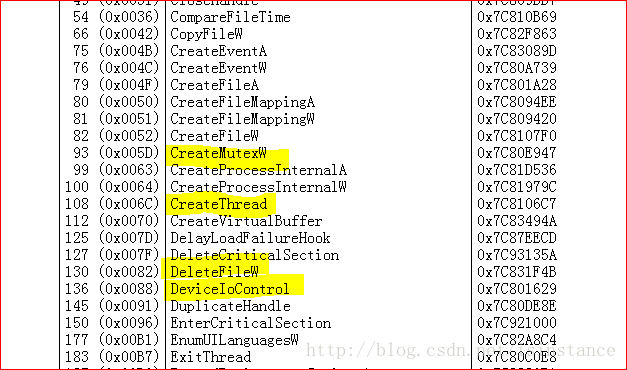
**我们可以看到这里有个我们需要注意的CreateFileA导出函数，还有下面那个ExitProcess函数，我们找找看有没有CreateProcess这个函数，找了一圈没有**

**然后我们再看ADVAPI32.DLL**

****

**这里有个RegCreateKeyExA这个东西，还有下面的RegSetValueExA这个函数，都会改变注册表**

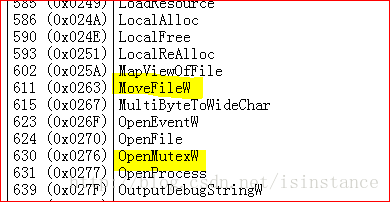
**然后我们看下一个DLL，是ADVAPI32.DLL下面的KERNEL32.DLL里面的导出函数**

****

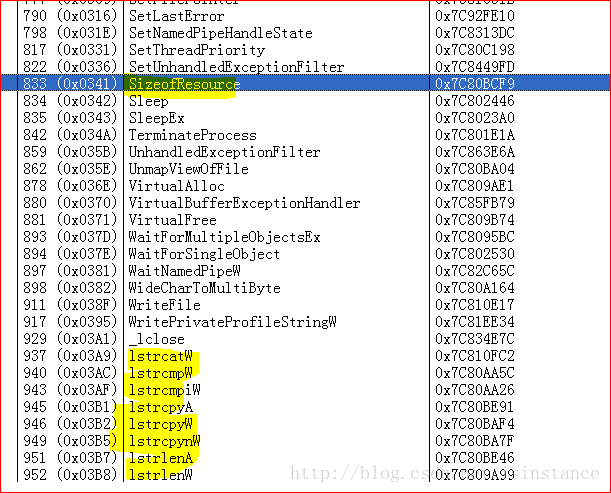
**这里我们注意到这么几个函数比较有意思**

**第一是CreateMutexW，这个函数会创建了一个互斥量，然后就是CreateThread，这个会创建一个线程，然后就是DeleteFileW，这个删除一个文件，然后就是我们内核分析时候见过的DeviceIoControl这个，用于给内核中的驱动发送一个信号的函数**

**下面的导出函数也证明了这个函数可能会操纵Mutex和Thread**

****

**这里我们发现了MoveFileW和OpenMutexW这个函数**

****

**这里还有一个SizeofResource这个函数，说明这个函数对资源节会有一些操作，然后我们还看见了ls\*的一堆对字符串进行操作的函数**

**我们还可以在ADVAPI32.DLL下面发现这么两个DLL**

****

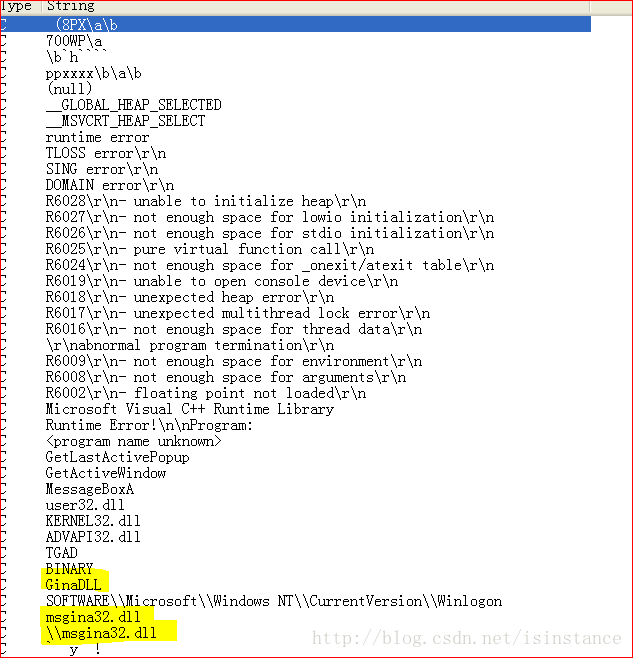
**其中一个是WINTRUST.DLL，另一个是SECUR32.DLL，其中第一个DLL的作用是**

**wintrust.dll是 DLL文件信息，用于验证第三方应用程序的文件，目录，内存使用，数据签名等**

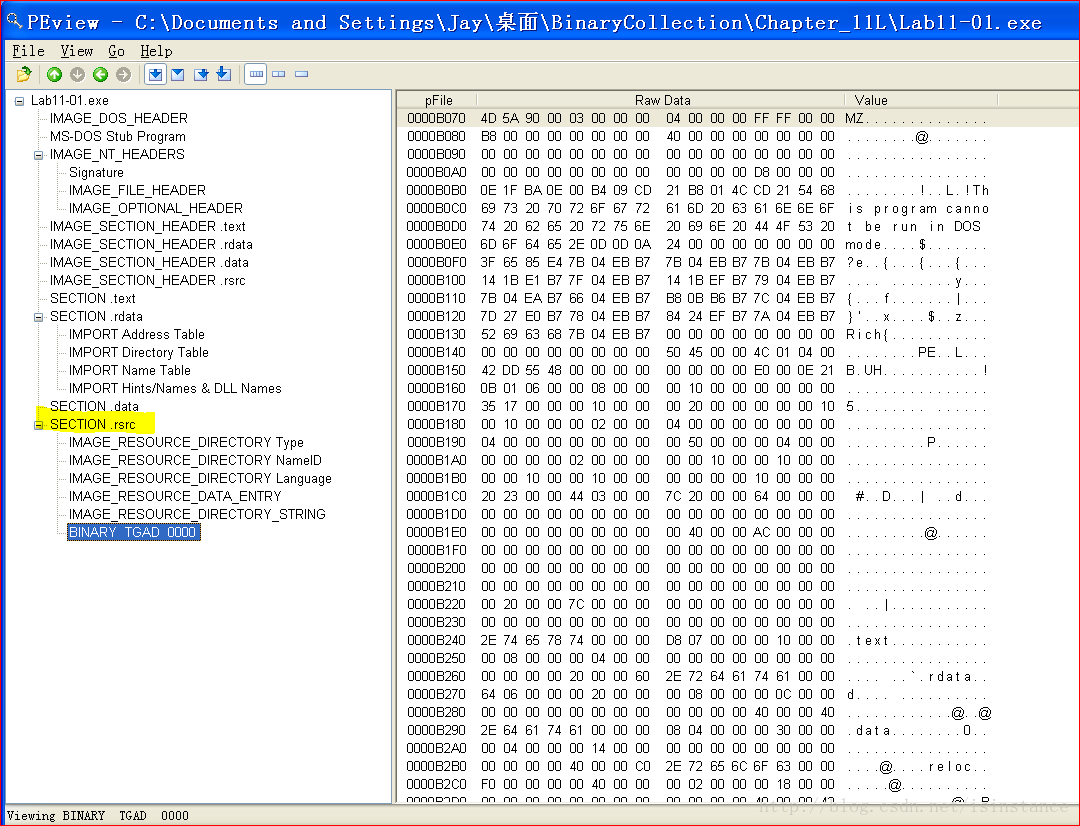
**全名是Microsoft Trust Verification APIs**

**而下面的SECUR32.DLL全名是Security Support Provider Interface，这是一个提供了各种安全支持提供程序的接口调用库**

**然后我们可以看看这个程序的字符串有什么**

****

**我们可以发现这个GinaDLL和msgina32.dll字符串中的显示，然后我们开始看资源节的内容**

****

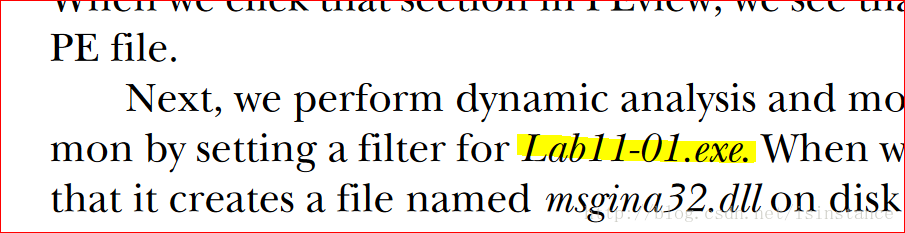
**我们可以看到这个.rsrc节中，有个叫BINARY TGAD 0000的二进制文件，我们点开就可以发现那个This program cannot be run in DOS mode这个字符串，这表明这个二进制是个可执行程序**

**进行完这些基本的静态分析，我们开始使用动态分析，还是先设置DNSfake和Webfake，之后使用procmon设置过滤器来看程序的操作**

**这里书上有个错误，中文版的书上写的是**

**设置Lab11-02.exe**

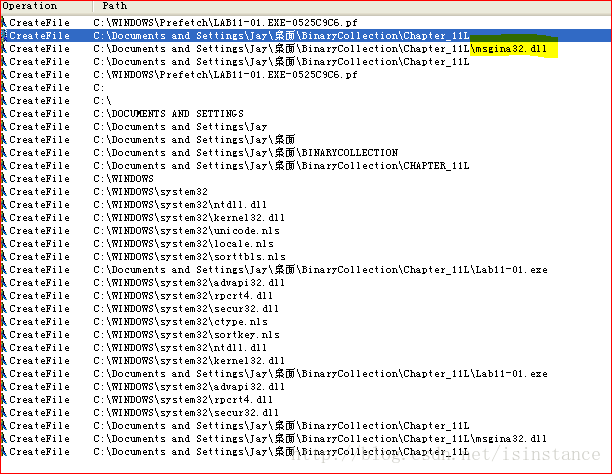
**中文版我没有PDF，所以这里截个英文版的**

****

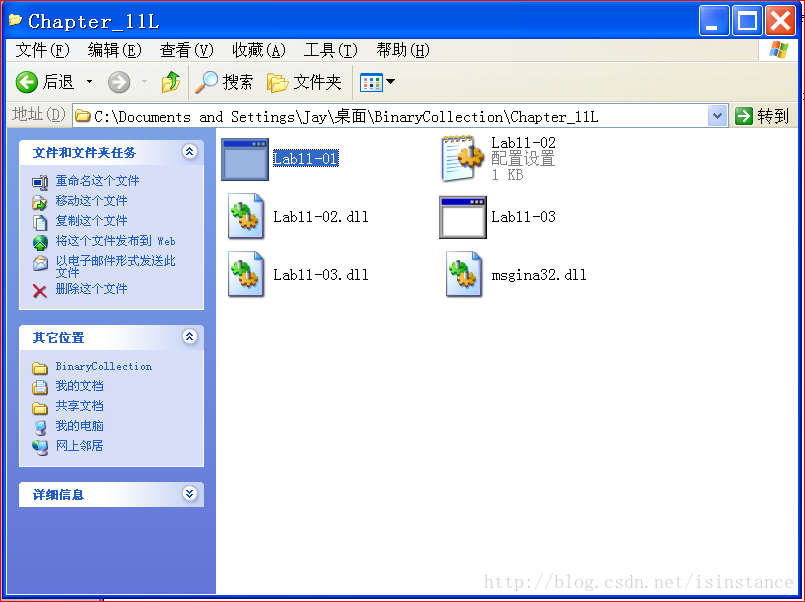
**这里对照了一下英文版的，是Lab11-01.exe，这里应该是译者的翻译错误**

**这里在DNS中病未看见程序查找了什么域名，但是注意这不代表这个程序不会进行网络连接，如果有奇葩的病毒直接用ip来访问的话，是不会进行任何的DNS查询的**

**然后我们再设置一个过滤条件为CreateFile这个东西，之后我们就可以看见**

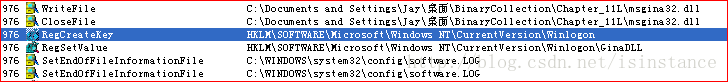
****

**这里我们看见在本地的目录下创建了一个msgina32.dll这个动态链接库**

****

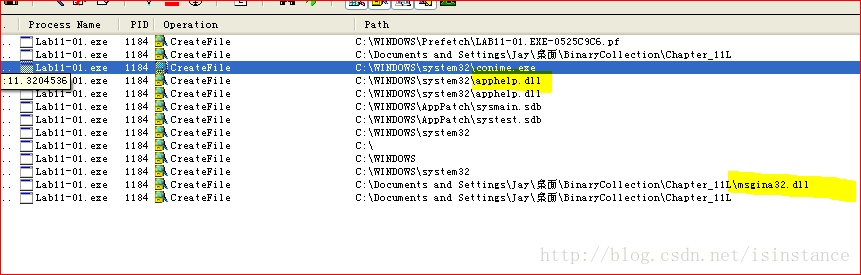
**这里我们看见创建了一个msgina32.dll文件**

**然后我们回到时间顺序，然后看**

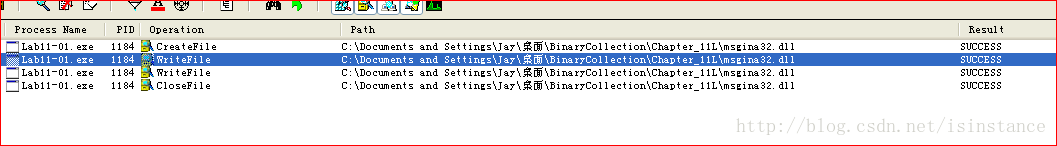
****

**这里我们可以看到这里调用了一个RegCreateKey然后创建了一个HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon**

**然后通过函数RegSetValue来设置了一个值为HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL**

****

**而对这个msgina32.dll的操作只有如下的几个操作**

****

**只有一个CreateFile之后就是WriteFile，然后就CloseFile了**

**我们可以还可以发现这么两个调用CreateFile的参数的比较有意思，其中一个是conime.exe，还有一个是apphelp.dll，其中conime.exe是控制输入法的程序，apphelp.dll是维护程序运行时候的一些错误的修正补丁**

**现在我们将资源节里面的二进制导出来看看和这个msgina32.dll是不是一样的**

****

**这个Out.dll是我们从资源节导出来的文件**

****

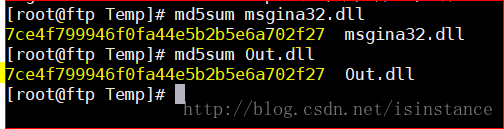
**我们可以看到这两个dll大小是一样的，当然，这样比较是比较肤浅的，比较严谨的比较方式是计算其MD5和SHA1的值，我们先在也计算一下**

**这里我们介绍的是在Linux下面计算的方法，Windows的有专门的软件，或者用在线的工具**

**我这里有台CentOS 7的机器用做ftp，直接上传然后计算就好了，因为在Linux里面自带了计算MD5的工具**

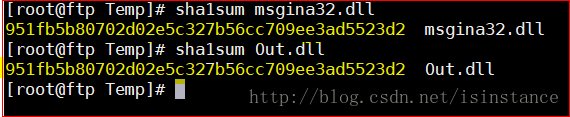
****

**然后就计算MD5**

****

**可以看到我们的两个文件的MD5值是一样的**

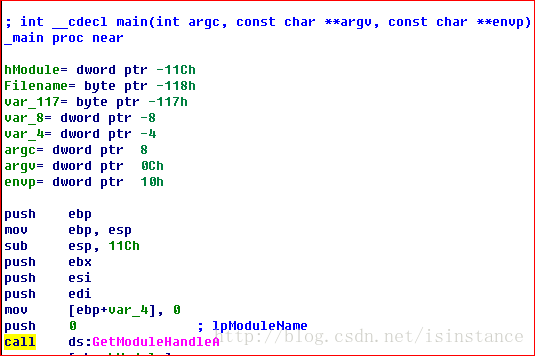
**然后我们计算SHA1的值**

****

**都是一样的**

**所以我们就可以确定这个在资源节的二进制文件就是msgina32.dll这个文件**

**接下来还是一样的，我们打开IDA来分析这个Lab11-01.exe，一样的，我们每次遇到call就做一段分析**

****

**然后我们遇到的第一个call是GetModuleHandleA，这个函数在MSDN中定义是**

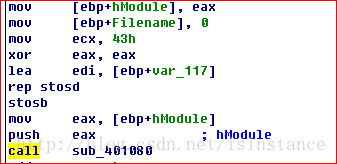
**检索指定模块的模块句柄。 该模块必须已由调用进程加载**

**然后我们可以看到这个函数的入参是0也就是NULL**

**如果此参数为NULL，则GetModuleHandle将返回用于创建调用进程（.exe文件）的文件的句柄**

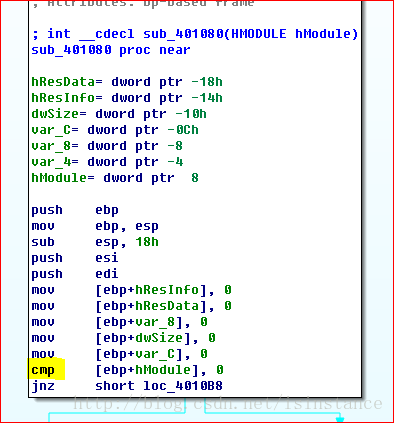
**这句话的意思就是，如果这个入参是0，那么这个函数返回的是这个函数的本身的句柄**

**所以我们这里的返回值是指向这个exe文件的句柄，然后我们开始看下一个call**

****

**我们可以看到这里压入了一个参数eax，而这个eax虽然经过上面那么多的变换，还是GetModuleHandleA的返回值**

**进来这个函数之后，我们可以看到如下的初始化操作**

****

**注意这里的cmp指令，这里的指令让[ebp+hModule]与0进行了比较，我们注意这里的hModule的偏移值是dword ptr 8，也就是最后这个的值是[ebp+8]，根据我们以前的栈分析，ebp+8的值代表了调用函数的最后一个入参，如果只有一个入参的话，就代表了那个入参的值**

**当然你这里也可以不用像我们这样分析栈的组成，因为IDA已经给你标出来了这个值是hModule**

**OK，言归正传之后，我们开始分析这段代码**

**汇编代码到push edi之前的代码都是在初始化栈**

**之后的代码**

**mov [ebp+hResInfo], 0**

**mov [ebp+hResData], 0**

**mov [ebp+var\_8], 0**

**mov [ebp+dwSize], 0**

**mov [ebp+var\_C], 0**

**cmp [ebp+hModule], 0**

**jnz shrt loc\_4010B8**

**1**

**2**

**3**

**4**

**5**

**6**

**7**

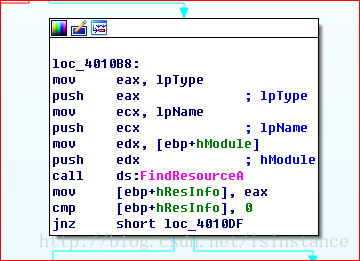
**前面的mov指令都是在为参数赋值，这里将hResInfo等等的参数都赋值为了0**

**然后下面就是比较hModule的值是否为0，如果函数调用成功的话，返回的是一个指针肯定不等于0，如果函数调用失败，就返回NULL也就是0了**

**如果这个返回的指针为0，则cmp指令之后，ZF=1，jnz不跳转，执行红色的线**

**如果返回指针不为0，cmp之后，ZF=0，jnz跳转，走绿色的线**

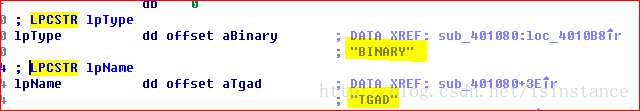
**绿色的线就是**

****

**下面就会执行这个FindResourceA函数，而这个函数在MSDN中的定义是**

**确定指定模块中具有指定类型和名称的资源的位置**

**这里我们注意这个函数的入参，通过查看就可以得出如下的入参表**

****

**HRSRC WINAPI FindResource(**

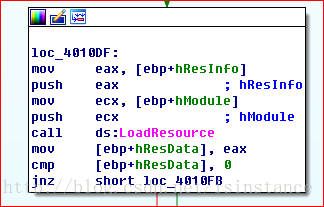
**\_In\_opt\_ HMODULE hModule = [ebp+hModule],**

**\_In\_ LPCTSTR lpName = TGAD,**

**\_In\_ LPCTSTR lpType = BINARY**

**);**

**这个函数的意思就是将在资源节中名字叫TGAD，类型是BINARY的节找出来，函数执行成功之后，将返回一个指定资源的信息块的句柄**

****

**下一个函数是LoadResource这个函数，在MSDN中的定义是**

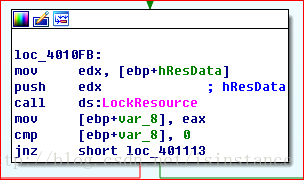
**检索可用于获取指向内存中指定资源的第一个字节的指针的句柄**

**这个函数调用成功之后，会返回**

**如果函数成功，则返回值是与资源关联的数据的句柄**

**这个返回值最后会存在hResData里面**

**之后下一个调用是LockResource**

****

**这个函数在MSDN里面的意思就是**

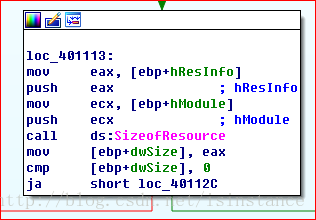
**检索指向内存中指定资源的指针**

**这个函数的返回值如下**

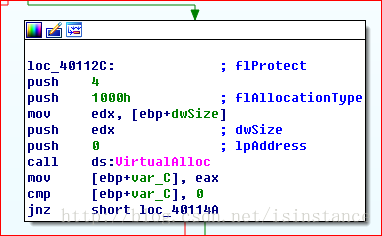
**如果加载的资源可用，则返回值是指向资源第一个字节的指针;否则，它是NULL**

**这就会把指针开始指向了资源的第一个字节，从这个时候开始才开始操作资源节的内容**

**然后下一个代码块**

****

**这个函数会返回资源节的大小，还是一样的做了一个结果比较跳转**

****

**之后调用VirtualAlloc分配一个空间给要导出的DLL**

**其中，这个函数在MSDN中的定义如下**

**LPVOID WINAPI VirtualAlloc(**

**\_In\_opt\_ LPVOID lpAddress = 0,**

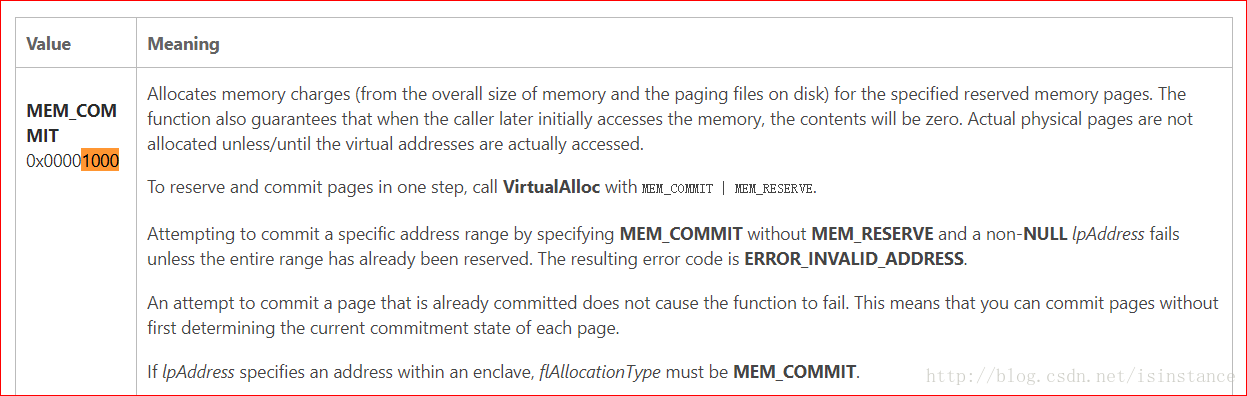
**\_In\_ SIZE\_T dwSize = dwSize,**

**\_In\_ DWORD flAllocationType = 1000h,**

**\_In\_ DWORD flProtect = 4**

**);**

**其中需要我们注意的是flAlloctaionType这个参数，这个参数的值为1000h在MSDN里面代表了MEM\_COMMIT**

****

**意思是**

**为指定的保留内存页分配内存费用（从内存的总大小和磁盘上的分页文件）。 该函数还保证当调用者最初访问内存时，内容将为零。 除非实际访问虚拟地址，否则不会分配实际的物理页面。**

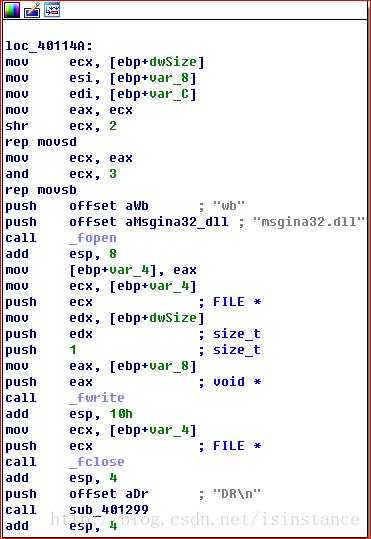
**要一步保留并提交页面，请使用MEM\_COMMIT |调用VirtualAllocMEM\_RESERVE。**

**尝试通过指定MEM\_COMMIT而不使用MEM\_RESERVE来提交特定地址范围，除非整个范围已被保留，否则非空lpAddress将失败。 由此产生的错误代码是ERROR\_INVALID\_ADDRESS。**

**尝试提交已提交的页面不会导致函数失败。 这意味着您可以在不首先确定每个页面的当前承诺状态的情况下提交页面。**

**如果lpAddress指定飞地内的地址，则flAllocationType必须是MEM\_COMMIT**

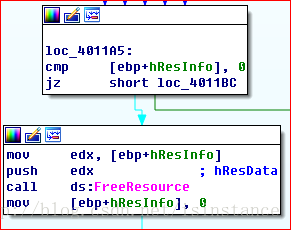
**这一步就是分配了一个空间给将来要释放出来的dll的**

****

**之后的代码，我们可以看到这里调用了\_fopen函数，并且这个函数的mode是wb，也就是二进制写入，名字是msgina32.dll**

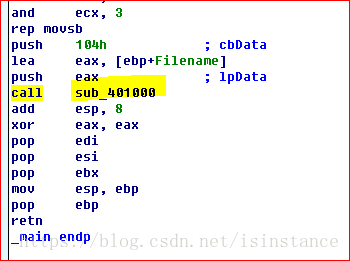
**之后用\_fwrite函数将指向资源节的数据写入了这个\_fopen打开的文件里面，之后就调用了\_fclose关闭句柄，之后还有一个函数叫sub\_401299，起始这个是printf函数，以前我们分析过为什么是printf函数**

**这里就是调用printf函数输出了一个DR\n**

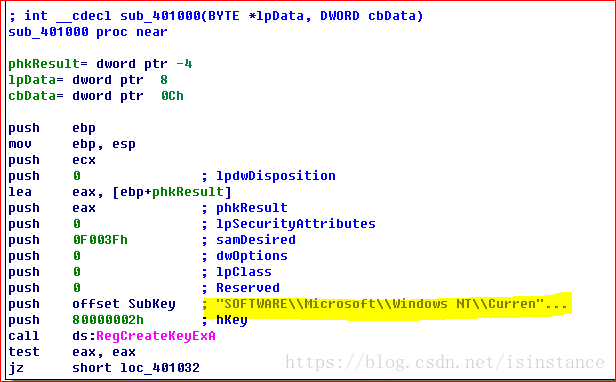
****

**之后用FreeResource来释放这个操作Resource的指针**

**之后我们看这个sub\_401000函数的操作**

****

**这是一个在整个函数底部的函数**

****

**进入这个函数之后我们可以看见，第一个函数调用是RegCreateKeyExA这个函数，我们可以得到这个函数在MSDN中的定义**

**LONG WINAPI RegCreateKeyEx(**

**\_In\_ HKEY hKey = 80000002h,**

**\_In\_ LPCTSTR lpSubKey = SubKey,**

**\_Reserved\_ DWORD Reserved = 0,**

**\_In\_opt\_ LPTSTR lpClass = 0,**

**\_In\_ DWORD dwOptions = 0,**

**\_In\_ REGSAM samDesired = 0F003Fh,**

**\_In\_opt\_ LPSECURITY\_ATTRIBUTES lpSecurityAttributes = 0,**

**\_Out\_ PHKEY phkResult phkResult,**

**\_Out\_opt\_ LPDWORD lpdwDisposition = 0**

**);**

**其中hKey的意思是A handle to an open registry key，也就是一个一个开放注册表项的句柄**

**而lpSubKey的意思此函数打开或创建的子项的名称**

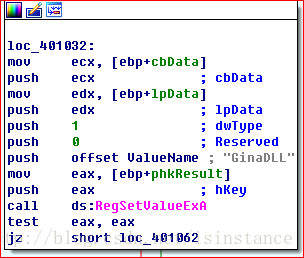
**这里我们大概知道这个函数会创建一个叫**

**SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon**

**1**

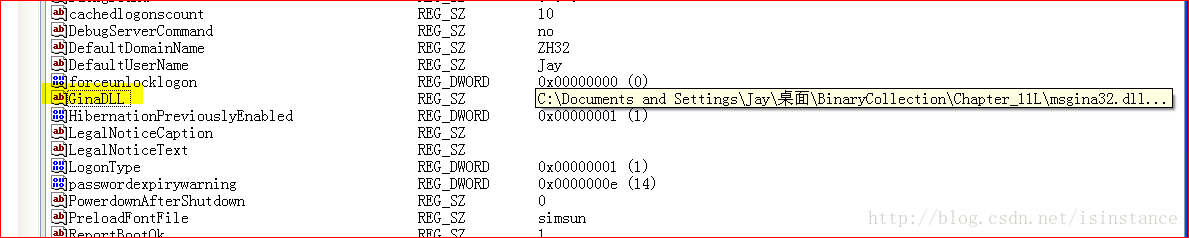
**的键，这个键按照字面意思winlogon理解就是登录，这个在MSDN中的解释是**

**Windows操作系统的一部分，提供交互式登录支持。  
Winlogon是围绕交互式登录模型设计的，该模型由三部分组成：Winlogon可执行文件，图形标识和认证动态链接库(DLL)(称为GINA)以及任意数量的网络提供程序。**

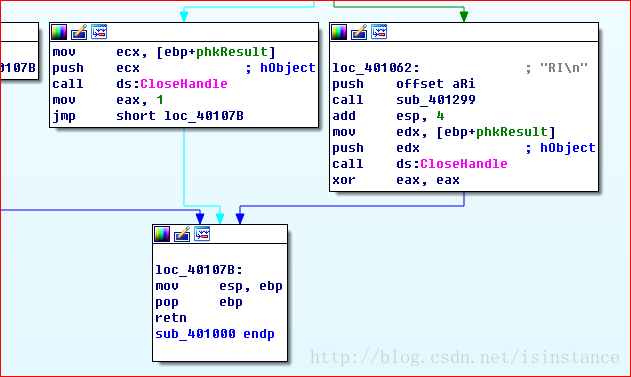
****

**下一个调用的函数是RegSetValueExA这个函数，这个函数我们注意这个ValueName和lpData的值，一个是键的名字，一个是键的值**

**我们已经运行过了这个代码，所以我们现在可以打开注册表来查看这个位置的值**

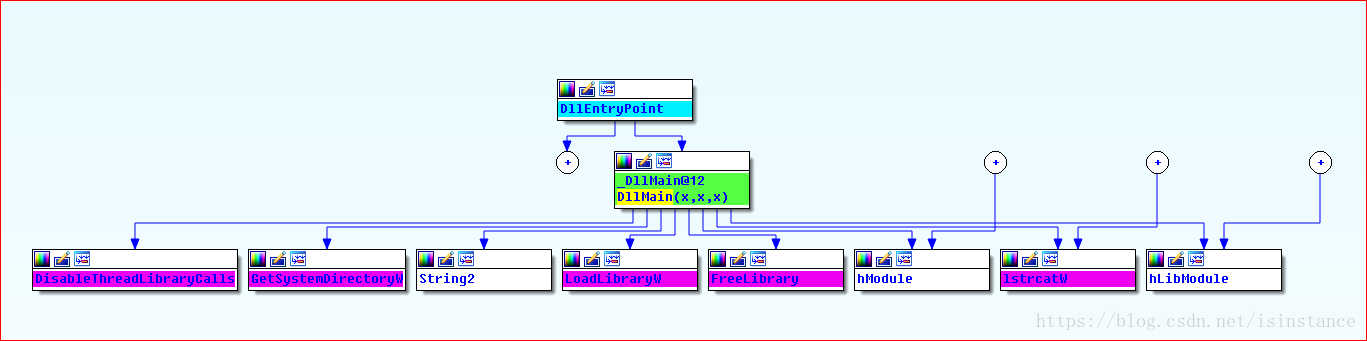
****

**可以看见，我们这里的的GinaDLL的值被设置成了msgina32.dll的绝对路径，不管你在吧这个Lab11-01.exe放哪里，都可以将对应的msgina32.dll的绝对路径写入键值中**

****

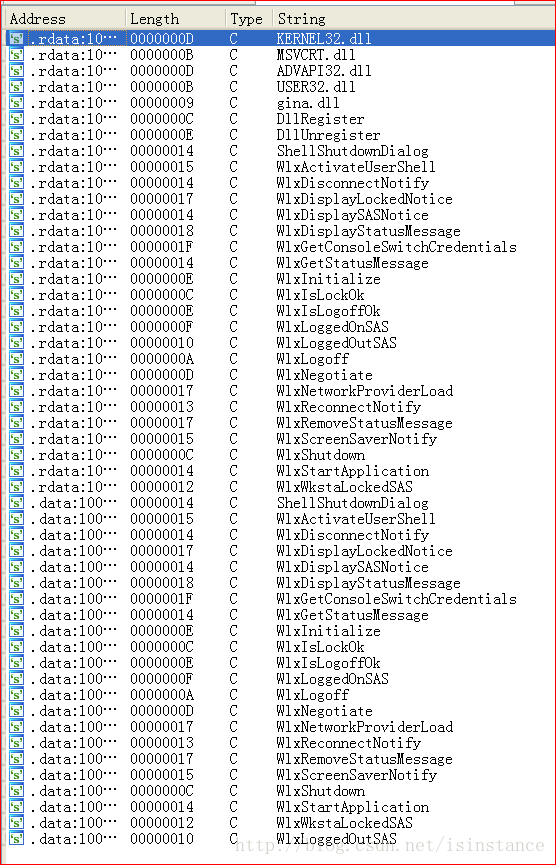
**之后，函数就关闭句柄，然后就退出了**

**下面我们进行的msgina32.dll的分析**

****

**我们打开这个msgina32.dll文件，可以看到这样的结构**

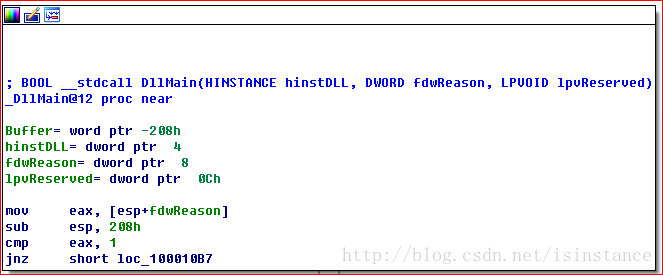
**我们查看字符串**

****

**这里我们可以看到许多的函数名，还有一些DLL的名称**

**在这里我们并没有看见书中说的那些字符串的位置，然后我们进入高级静态分析的过程**

**先找到DLL的开始的地方DllMain这个地方，可以看到如下的样子**

****

**这里从外部传入了一个fdwReason参数，这个fdwReason参数的意义是**

**reason code说明了为什么调用DLL入口函数**

**而这个是正DllMain的结构函数**

**BOOL WINAPI DllMain(**

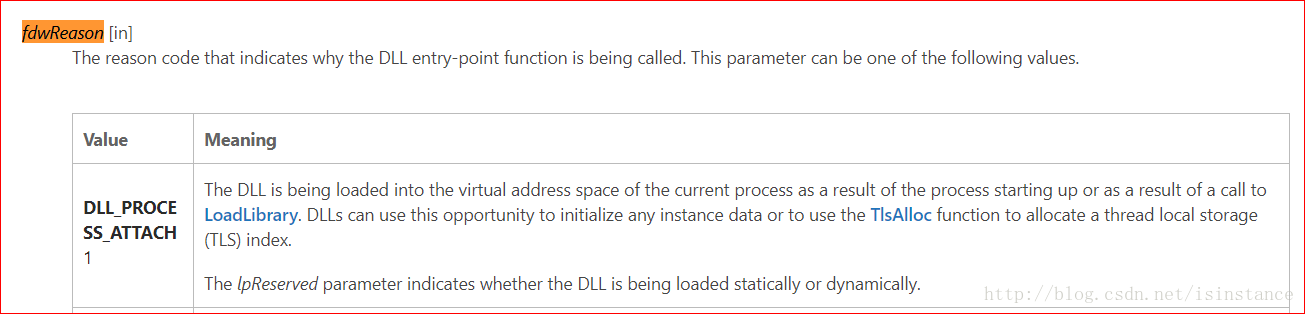
**\_In\_ HINSTANCE hinstDLL,**

**\_In\_ DWORD fdwReason,**

**\_In\_ LPVOID lpvReserved**

**);**

**这里将fdwReason和1进行了比较，之后跳转，这里我们要明确一下这个1代表什么意义在DllMain中**

****

**这里我们可以看到这个1代表的意思是DLL\_PROCESS\_ATTACH，而它的解释翻译过来是这样的**

**由于进程启动或由于调用LoadLibrary，DLL正被加载到当前进程的虚拟地址空间中。 DLL可以使用此机会初始化任何实例数据或使用TlsAlloc函数分配线程本地存储（TLS）索引。**

**看不懂，大概意思是这个DLL会被调用程序加载进调用程序的虚拟地址空间之中，这也就是ATTACH的意思，想起OD的ATTACH功能了不？**

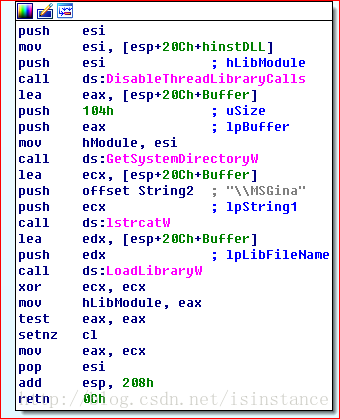
**然后就是一个jnz跳转，我们考虑一下这里的jnz会在什么情况之下跳转**

**cmp eax, 1**

**1**

**之后，如果eax也就是fdwReason的值是1的话，cmp之后，ZF=1，之后jnz不会跳转，而如果eax的值不是1的话，ZF=0，之后jnz跳转，跳转之后执行的是函数清理和退出**

**所以这里的代码期望的值是1，这样就可以继续执行如下的代码**

****

**这里执行的第一个函数是DisableThreadLibraryCalls，这个函数的解释是**

**为指定的动态链接库(DLL)禁用DLL\_THREAD\_ATTACH和DLL\_THREAD\_DETACH通知，这可以减少某些应用程序的工作集大小**

**之后下一个调用是GetSystemDirectoryW，这个函数的解释是**

**检索系统目录的路径，系统目录包含系统文件，如动态链接库和驱动程序**

**此功能主要是为了兼容性而提供的。应用程序应将代码存储在Program Files文件夹中，并将持久数据存储在用户配置文件的Application Data文件夹中**

**下一个调用函数lstrcatW，这个函数熟悉的C语言的同学应该非常熟悉了**

**追加一个字符串到另一个**

**这个函数会将两个字符串拼接在一起**

**LPTSTR WINAPI lstrcat(**

**\_Inout\_ LPTSTR lpString1 = ecx,**

**\_In\_ LPTSTR lpString2 = "\\MSGina"**

**);**

**这里我们可以看到，第一个字符串的值是ecx，而要拼接在ecx之后的字符串是\\MSGina，而我们也可以知道，这个ecx的值是GetSystemDirectoryW的返回值**

**在MSDN中，这样解释lpBuffer**

**指向接收路径的缓冲区的指针，除非系统目录是根目录，否则此路径不会以反斜杠结尾  
例如，如果系统目录在驱动器C上名为Windows\System32，则此函数检索到的系统目录的路径为C:\Windows\System32**

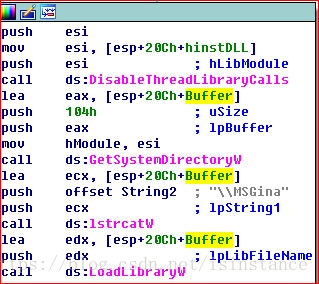
**所以我们可以预测这个ecx的值是C:\Windows\System32，之后，和字符串\\MSGina拼接之后，就成了**

**C:\Windows\System32\MSGina**

**1**

**下一个要调用的函数是LoadLibraryW，这个函数在MSDN中的解释是**

**将指定的模块加载到调用进程的地址空间中，指定的模块可能会导致其他模块被加载**

****

**这里我们可以分析得出，这个[esp+20Ch+Buffer]其实就是**

**C:\Windows\System32\MSGina**

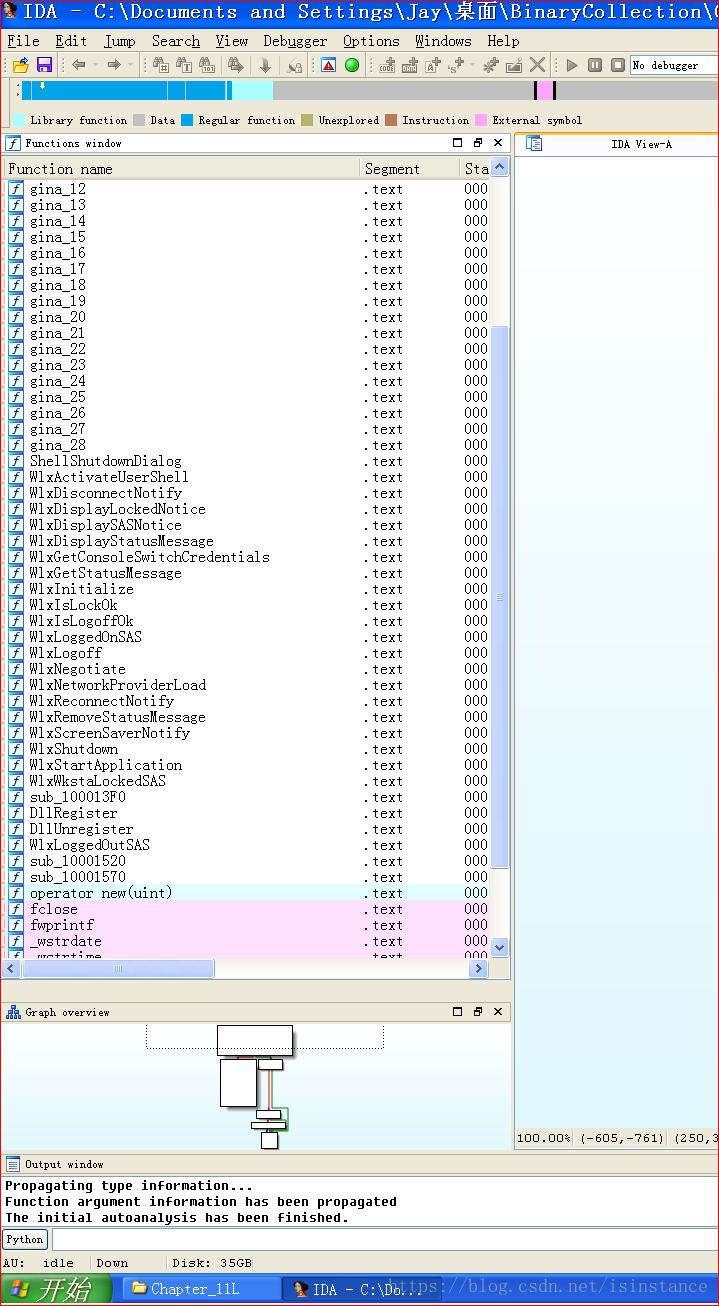
**所以这里的的LoadLibraryW其实要加载的DLL就是上面那个路径上的MSGina，这里需要我们注意一下，这个路径用LoadLibraryW调用之后，会加载的DLL其实是C:\Windows\System32\msgina.dll，之后函数就是做了清理工作就退出了。**

**msgina.dll是实现GINA的Windows DLL，然而msgina32.dll是拦截GINA的恶意DLL程序。设计成msgina.dll名字的目的显然是为了欺骗分析人员**

**之后这个代码会将打开的msgina.dll 的句柄保存在hLibModule中，这个变量是个全局变量，使用这个变量msgina32.dll会在合适的时候调用msgina.dll中的函数，但是这个函数明显是被msgina32.dll劫持了，这样可以保持系统的正常运行**

**到这里为止，我们已经完成了DllMain的分析，之后我们分析这个dll的导出函数**

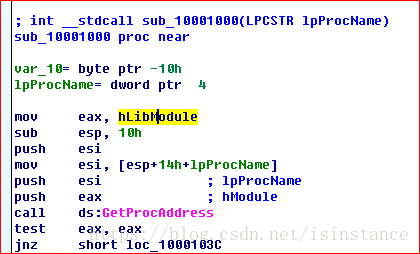
**这些导出函数我们可以在这里找到**

****

**这里我们可以随便点一个函数，比如第一个函数**

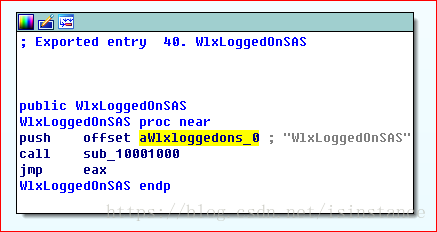
****

**之后我们可以看到这个hLibModule，佐证了这个变量是全局变量**

****

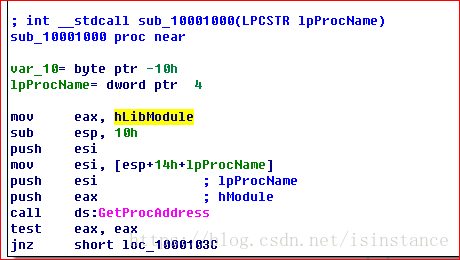
**我们还是按照书上的步骤来走，先来分析一下这个WlxLoggedOnSAS函数的代码**

****

****

**这个函数打开之后就可以看到这些代码**

**这个函数压栈了一个参数WlxLoggedOnSAS，然后调用了sub\_10001000，很巧的是，这个函数就是我们刚刚随便点的那个函数**

****

**这里函数会用全局变量hLibModule这个打开的msgina.dll的句柄来赋值给eax**

**然后下一个函数调用是GetProcAddress，这个函数的定义就是**

**从指定的动态链接库(DLL)中检索导出的函数或变量的地址**

**而这个要检索的DLL就是msgina.dll，这个是由eax来决定的，而eax刚刚才被赋值为hLibModule**

**而这个lpProcName其实就是刚刚上面压栈的那个字符串WlxLoggedOnSAS，下面我们来分析一下这个**

**执行完如下代码**

**push offset aWlxLoggedons\_0**

**1**

**之后，栈空间如下**

**----------------**

**| WlxLoggedOnSAS | <--- esp**

**----------------**

**| ... |**

**----------------**

**.**

**.**

**.**

**----------------**

**| ... | <--- ebp**

**----------------**

**进入调用函数之后，会第一时间push一个返回地址入栈，这个操作在代码上是不会显示的，但是却是实实在在操作了的**

**----------------**

**| retuen address | <--- esp**

**----------------**

**| WlxLoggedOnSAS |**

**----------------**

**| ... |**

**----------------**

**.**

**.**

**.**

**----------------**

**| ... | <--- ebp**

**----------------**

**sub esp, 10h**

**----------------**

**| ... |-- <--- esp**

**---------------- |**

**| ... | |**

**---------------- |---> 0x10h = 16d**

**| ... | |**

**---------------- |**

**| ... |--**

**----------------**

**| retuen address |**

**----------------**

**| WlxLoggedOnSAS |**

**----------------**

**| ... |**

**----------------**

**.**

**.**

**.**

**----------------**

**| ... | <--- ebp**

**----------------**

**之后就是下面这句操作**

**push esi**

**----------------**

**| esi | <--- esp**

**----------------**

**| ... |--**

**---------------- |**

**| ... | |**

**---------------- |---> 0x10h = 16d**

**| ... | |**

**---------------- |**

**| ... |--**

**----------------**

**| retuen address |**

**----------------**

**| WlxLoggedOnSAS |**

**----------------**

**| ... |**

**----------------**

**.**

**.**

**.**

**----------------**

**| ... | <--- ebp**

**----------------**

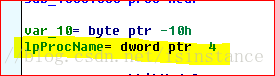
**之后的指令操作，注意这个指令比较关键**

**mov esi, [esp+14h+lpProcName]**

**而lpProcName在函数的开头就标注了值是4，但是注意一下这里标注的数据类型是dword ptr类型**

**这里有两个类型，一个是byte ptr -10h，就是上面那个var\_10，一个是dword ptr的类型，这两个类型是完全不同的两个数据类型**

**其中byte是一个字节的数据，而dword是两个字节的数据**

****

**我们推导一下esp+18h是哪里**

**从上图简略的栈图中，我们可以看出，存储esi的空间占了四个地址，加上刚刚申请的0x10h个字节，就是0x14h个字节的空间**

**于是我们得到esp+18h的值**

**----------------**

**| esi | <--- esp**

**----------------**

**| ... | <--- esp+4h**

**----------------**

**| ... | <--- esp+8h**

**----------------**

**| ... | <--- esp+ch**

**----------------**

**| ... | <--- esp+10h**

**----------------**

**| retuen address | <--- esp+14h**

**----------------**

**| WlxLoggedOnSAS | <--- esp+18h**

**----------------**

**| ... |**

**----------------**

**.**

**.**

**.**

**----------------**

**| ... | <--- ebp**

**----------------**

**mov esi, [esp+18h]**

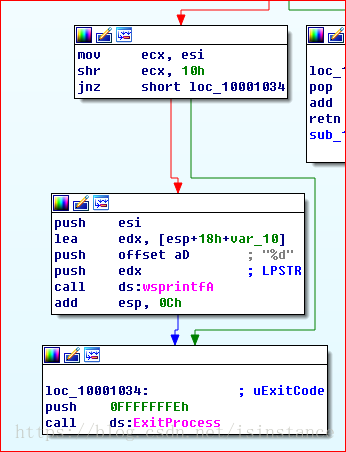
**查明是因为函数调用后第一时间会push一个返回地址进去，但是这个操作不会体现在汇编代码中**

**这样，而不是想移动，再赋值，所以esp指向的地方永远是空值**

**所以这里的函数会在用GetProcAddress来在真正的msgina.dll中找到函数WlxLoggedOnSAS的地址，之后**

**之后如果函数调用成功，会做一个判断跳转，这里依旧是假设我们的函数调用成功**

**就会执行下面这些函数**

****

**这里的第一个指令是**

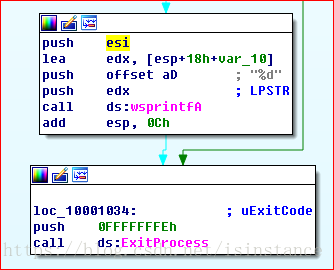
**mov ecx, esi**

**之后做的操作是**

**shr ecx, 10h**

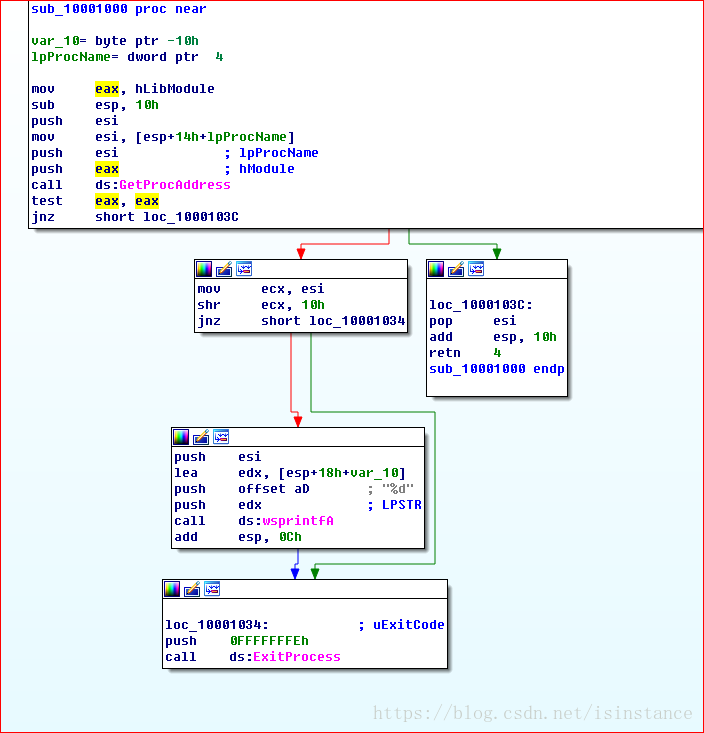
**这句指令的意思是将ecx指向的那个字符串的指针的值置为0，esi的值并未改变。**

**因为shr是逻辑右，左边用0补齐，所以移动0x10h之后，整个指针的16bit都为了0。因为最后的结果为0，所以ZF=1，则JNZ不会跳转，最后会按照红线的路线执行。**

****

**这里会打印一个字符串，这里的var\_10=-10，最后其实寻找的地址是**

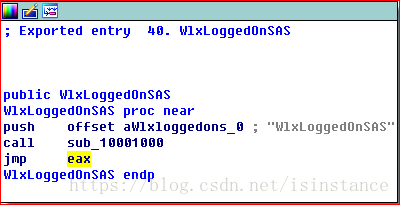
**[esp+8h]。然后我们注意这里的eax，自从调用了GetProcAddress之后，就一直未改变过。**

****

**这里可以看到，最后一次出现eax是**

**test eax, eax**

**这句代码，之后就再也没改变过eax的值，这里的eax的值代表了WlxLoggedOnSAS在msgina.dll中的地址，真实的那个WlxLoggedOnSAS的地址，之后**

****

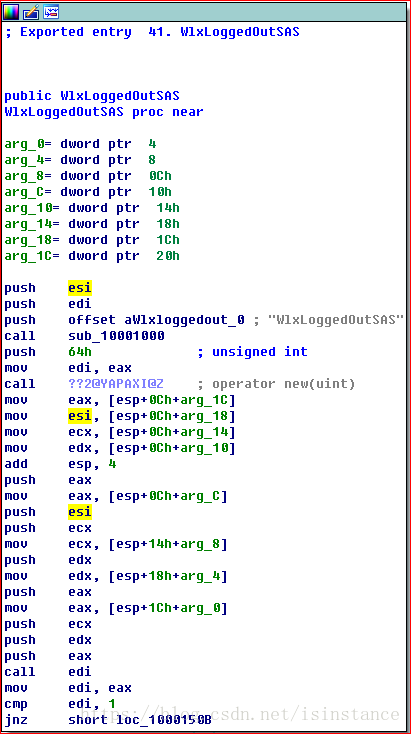
**函数调用完sub\_10001000之后就会跳转到eax的位置，这个位置就是WlxLoggedOnSAS的真实地址，也就是这里只是劫持了WlxLoggedOnSAS函数，然后执行完恶意程序作者的自己的代码sub\_10001000之后就又会跳转到真实的WlxLoggedOnSAS函数去执行**

**这样保证了系统的正常运行**

**下面书上的说法是**

**如果我们继续分析其他导出函数，将看到大部分于WlxLoggedOnSAS（它们是中转函数）中的类似操作，然而WlxLoggedOutSAS例外，它包含了一些额外的代码（当系统注销时调用WlxLoggedOutSAS）**

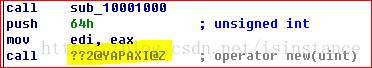
**下面我就缩短分析的步骤，直接去分析这个WlxLoggedOutSAS函数**

****

**上图是这个函数第一部分**

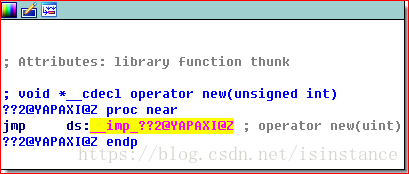
**我们可以看到这个函数的第一个调用是sub\_10001000，但是这次传入的参数是WlxLoggedOutSAS这个字符串**

**我们可以知道，这个函数传入一个字符串之后，就会调用GetProcAddress来查找这个函数在真正的msgina.dll中的地址**

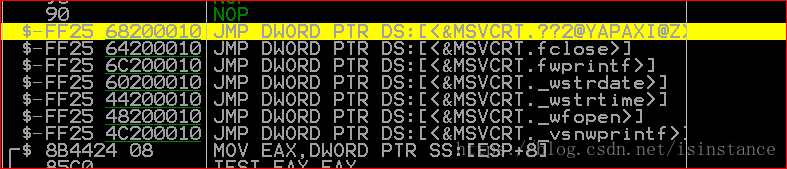
****

**这里有个IDA中无法注释的函数**

**我们打开看看会发现**

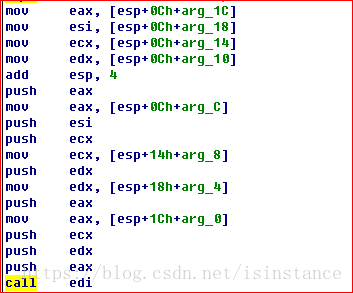
****

**这个调用的函数是在MSVCPRT中的函数，用IDA可能还不好看，可以用OD打开看看**

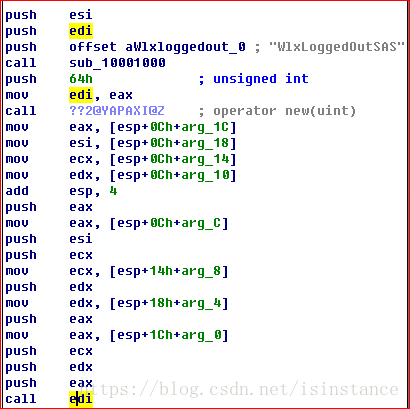
****

**可以看到，这个有点像乱码的函数和fclose和fwprintf都是出自MSVCRT的**

**下一个调用是**

****

**这里的edi我们可以看看是从哪里来的参数**

****

**这里的edi我们可以从代码就看出来，就是eax，而eax是调用sub\_10001000之后的的返回值，也就是函数WlxLoggedOutSAS的真实地址**

**所以这个代码会在这里去调用真实的WlxLoggedOutSAS函数，但是这个函数的入参我们需要注意一下，分析图中的各种mov和push操作我们可以得到下面的参数表**

**int WlxLoggedOutSAS(**

**\_In\_ PVOID pWlxContext = [esp+1Ch+arg\_0] = [esp+20h],**

**\_In\_ DWORD dwSasType = [esp+18h+arg\_4] = [esp+20h],**

**\_Out\_ PLUID pAuthenticationId = [esp+14h+arg\_8] = [esp+20h],**

**\_Inout\_ PSID pLogonSid = [esp+0Ch+arg\_C] = [esp+1ch],**

**\_Out\_ PDWORD pdwOptions = [esp+0Ch+arg\_10] = [esp+20h],**

**\_Out\_ PHANDLE phToken = [esp+0Ch+arg\_14] = [esp+24h],**

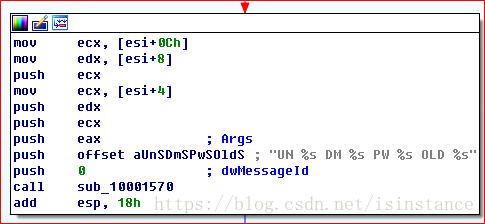
**\_Out\_ PWLX\_MPR\_NOTIFY\_INFO pNprNotifyInfo = [esp+0Ch+arg\_18] = [esp+28h],**

**\_Out\_ PVOID \*pProfile = [esp+0Ch+arg\_1C] = [esp+2ch]**

**);**

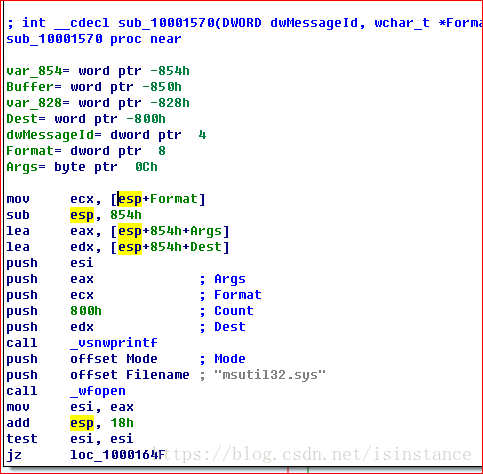
**这里就直接会调用在真正msgina.dll中的函数**

**调用之后就会执行下面的函数**

****

**这里调用了一个函数sub\_10001570，这个函数我们可以看看是什么**

**这里入栈的参数有一个eax和aUnSDmSpwSOlds这个东西**

****

**函数的开始，第一个call是调用了\_vsnwprintf这个函数，这个函数使用是这样的**

**int \_vsnprintf(char \*buffer, size\_t max\_count, const char \*format, va\_list vArgList);**

**这里的各个参数之间是作用是这样的**

**1. char \*buffer [out],把生成的格式化的字符串存放在这里.**

**2. size\_t max\_count [in], buffer可接受的最大字节数,防止产生数组越界.**

**3. const char \*format [in], 格式化字符串**

**4. va\_list vArgList [in], va\_list变量. va:variable-argument:可变参数**

**对照我们的汇编代码，我们可以得出下面结论**

**buffer = edx = [esp+54h]**

**max\_count = 800h**

**format = "UN %s DM %s PW %s OLD %s"**

**vArgList = eax [esp+860h]**

**这里需要注意的是[esp+54h]和[esp+860h]均是指向sub\_10001570函数调用之前栈空间，这里我们就没法溯源这些数据到底是什么了**

**之后函数会调用\_wfopen，这个函数会打开一个指向文件的文件句柄**

**FILE \*\_wfopen(**

**const wchar\_t \*filename,**

**const wchar\_t \*mode**

**);**

**这里的两个入参是这样的**

**filename = msutil32.sys**

**mode = 61h**

**我们查一下这个sys是什么，查不到，查出来的都是这个书的解题答案的网站。**

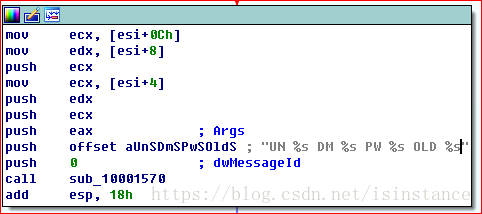
**\_vsnwprintf 调用填充了传入WlxLoggedOutSAS导出函数的格式化字符串**

**这个字符串的格式是这样的**

**UN %s DM %s PW %s OLD %s**

**这里我们可以猜测这个UN可能是UserName的简写，而PW可能是PassWord的简写。**

**根据这个对应关系，我们回到调用这个函数之前的地方，可以得出一下结论**

****

**这里的eax对应的应该就是用户的UserName，而[esi+4]则对应了DM，[esi+8]对应了PassWord，最后的[esi+0ch]对应了我们的OLD**

**我们溯源回去可以发现，eax其实就是调用真正的WlxLoggedOutSAS的一个参数，在[esp+0Ch+arg\_18]上**

**我们画出函数调用sub\_10001570时候的栈图**

**--------------**

**| 0 | <--- esp**

**--------------**

**|aUnSDmSPwSOlds|**

**--------------**

**| UN |**

**--------------**

**| DM |**

**--------------**

**| PW |**

**--------------**

**| OLD |**

**--------------**

**| --- |**

**--------------**

**之后我们进入了函数之后，第一时间会压栈返回地址**

**--------------**

**|return address| <--- esp**

**--------------**

**| 0 |**

**--------------**

**|aUnSDmSPwSOlds|**

**--------------**

**| UN |**

**--------------**

**| DM |**

**--------------**

**| PW |**

**--------------**

**| OLD |**

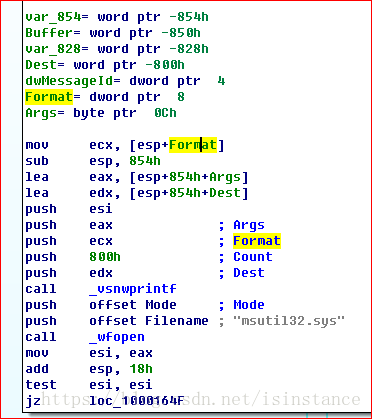
**--------------**

**| --- |**

**--------------**

**所以这里我们可以看出，赋值给ecx的[esp+8]其实就是那个格式化字符串的地址aUnSDmSPwSOLDs**

**然后，会有一个sub操作来扩大栈**

****

**--------------**

**| | --- <--- esp**

**-------------- |**

**. |**

**. | ---> 0x854h**

**. |**

**-------------- |**

**|return address| ---**

**--------------**

**| 0 |**

**--------------**

**|aUnSDmSPwSOlds|**

**--------------**

**| UN |**

**--------------**

**| DM |**

**--------------**

**| PW |**

**--------------**

**| OLD |**

**--------------**

**| --- |**

**--------------**

**之后的eax其实就是UN的地址**

**但是我们这里注意这个格式化字符串，需要4个入参，于是函数会自动往后依次去四个数据，也就是把那四个缩写都取了进去**

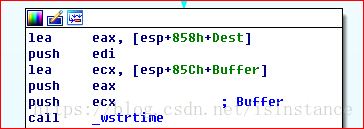
**由于我们这里不知道具体变量UN代表了什么值，所以这里我们用小写来代替它的值**

**于是我们可以这样写这个调用函数**

**\_vsnwprintf(esp+54, 800h, "UN %s DM %s PW %s OLD %s", un, dm, pw, old)**

**我们可以看出，这里是将变量都格式化到了字符中，然后保存在esp+54的地址上**

**我们理清这个栈关系之后，我们就可以继续下面的分析**

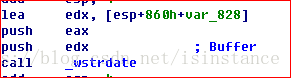
****

**这个函数\_wstrtime的作用是**

**将时间复制到缓冲区**

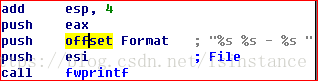
**这里的Dest其实就是我们刚刚格式化字符串的那个缓冲区的地址，但是这里没用到**

**\_wstrtime的入参只需要一个，这里我们记住Buffer这里存储的是时间字符串**

****

**下一个调用函数是\_wstrdate其实就是获得日期的函数**

**之后**

****

**之后函数就会调用fwprintf函数来格式化一个字符串**

**这里是定义**

**int fwprintf(**

**FILE \*stream,**

**const wchar\_t \*format [,**

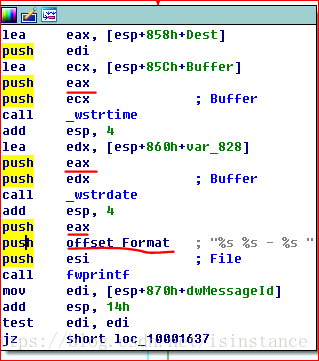
**argument ]...**

**);**

**这里的esi我们回头去看其实就是打开msutil32.sys的那个句柄**

**这里的eax则是上面的上面那个格式化字符串操作之后保存结果的地址**

**我们这里再截一个图来看整体的代码**

****

**这里分析的时候要注意的是这里的三个参数不是一次性push入栈的，是分了三次**

**最后一个push的eax代表了\_wstrdate的返回值**

**而倒数第二个的eax代表了\_wstrtime的返回值**

**倒数第三个的eax代表了我们前面格式化字符串后的那个字符串**

**我们可以写出伪代码的样子就是这样**

**fwprintf(esi, "%s %s - %s ", date, time, "UN un DM dm PW pw OLD old")**

**于是这个程序会把这些信息写入了msutil32.sys中**

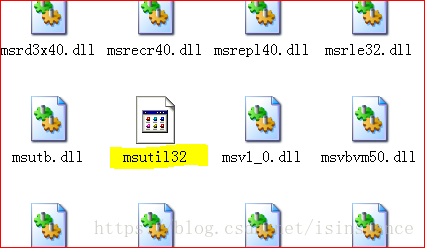
**如果我们重启机器，然后就可以在msutil32.sys中发现我们的用户和密码**

**我们现在验证一下~**

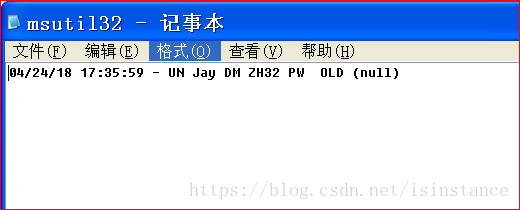
**我们在**

**C:\WINDOWS\system32\**

**中可以找到这个文件**

****

**然后我们打开看看**

****

**我们可以发现，我们的信息已经被记录下来了，我这里用的是空密码，所以PW后面是个空值**

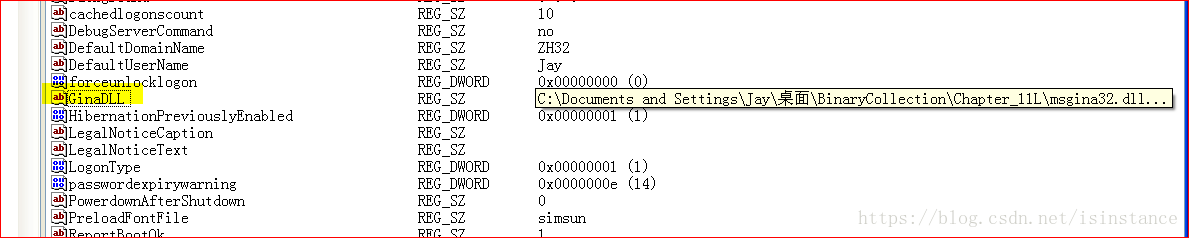
**病毒分析到这里基本我们就清除他的运作方式了**

**于是我们可以来回答这个问题也顺便回顾一下这个病毒**

**解答：这个恶意代码会从名叫TGAD的资源节提取一个文件并命名为msgina32.dll**

**2. 这个恶意代码如何进行驻留？**

**解答：这个恶意代码会在注册表中添加一个键值来安装这个DLL**

****

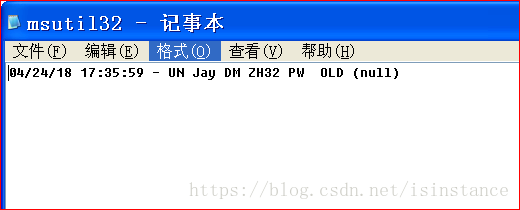
**系统重启之后，依旧会加载这个DLL**

**3. 这个恶意代码如何窃取用户登录凭证**

**解答：恶意代码用GINA机制来拦截用户的登录凭证，msgina32.dll会拦截所有提交到系统认证的用户登录凭证**

**4.这个恶意代码对窃取的凭证做了什么处理？**

**解答：这个恶意代码会将凭证保存在C:\WINDOWS\system32\msutil32.dll中**

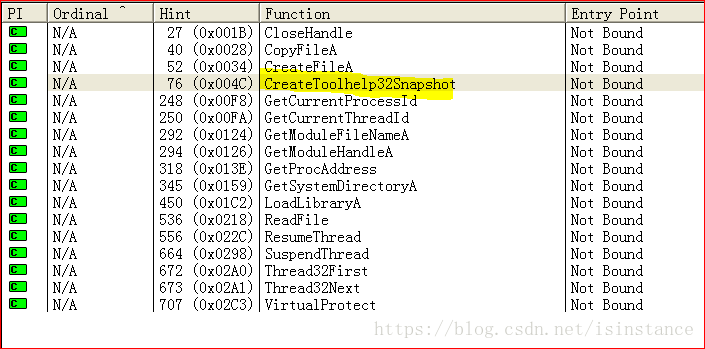
****

**5.如何在你的测试环境让这个恶意代码获得用户登录凭证？**

**解答：重启系统。**

**Lab11-02**

**先做一些静态的分析**

****

**这里有一个我们以前没见过的函数叫CreateToolhelp32Snapshot**

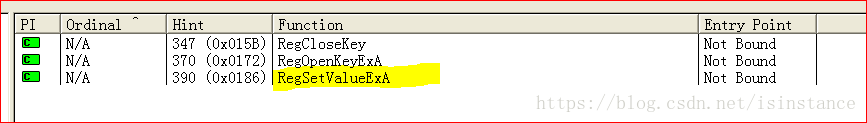
**这个函数在MSDN里面的定义是这样的**

**获取指定进程的快照, 以及这些进程使用的堆、模块和线程**

**书中对这个导入函数的解释是**

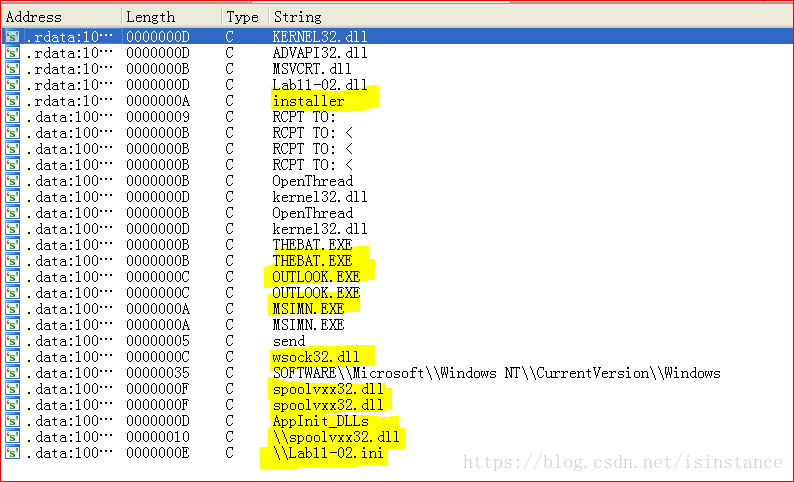
**搜索一个进程或者线程列表的导入函数**

**当然，我们还可以看见一些包括就像CopyFile和CreateFile函数，说明这个代码会操纵一些文件**

****

**还有三个操作注册表的函数**

**之后我们查看一下字符串有哪些**

****

**这里我们看到有趣的字符串有**

**installer**

**THEBAT.EXE**

**OUTLOOK.EXE**

**MSIMN.EXE**

**wsock32.dll**

**SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Windows**

**spoolvxx32.dll**

**AppInit\_DLLs**

**\\spoolvxx32.dll**

**\\Lab11-02.ini**

**这些，尤其是AppInit\_DLLs需要我们注意**

**对于这个键值的解释有**

**AppInit\_Dlls**

**键值位于注册表**

**HKLM\Microsoft\Windows NT\CurrentVersion\Windows**

**下面，相对于其他的注册表启动项来说**

**这个键值的特殊之处在于任何使用到User32.dll 的EXE、DLL、OCX等类型的PE文件都会读取这个地方**

**并且根据约定的规范将这个键值下指向的DLL文件进行加载，加载的方式是调用LoadLibrary。**

**众所周知，Windows服务程序的启动时机是可以非常早的，往往在用户登录之前就完成启动了**

**而这个时候最常见的Run键值还不一定被处理完，而且Windows服务程序拥有相当高的权限**

**（默认是Local System，可以对系统里面所有的资源进行操作）**

**因此如果一个恶意软件被加载到Windows服务里面，那么是会非常危险的**

**前文提到，任何进程使用了User32.DLL，都会对AppInit\_Dlls键值指向的DLL进行加载，如果是一个Windows服务程序，也不例外。**

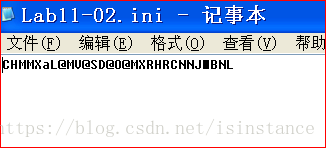
**上面这些话的意思就是，这个Appinit\_DLLs驻留方式是在系统启动之前就会把恶意DLL加载在Windows服务中。**

**而\Lab11-02.ini表明了这个程序有可能使用我们看到的那个Lab11-02.ini**

**其中还有一些像OUTLOOK.EXE和THEBAT.EXE还有MSIMN.EXE的字符串，书上说**

**这些都是邮件客户端，说明这个程序有可能操作这些客户端来发送邮件什么的**

**然后我们打开Lab11-02.ini来看看是什么：**

****

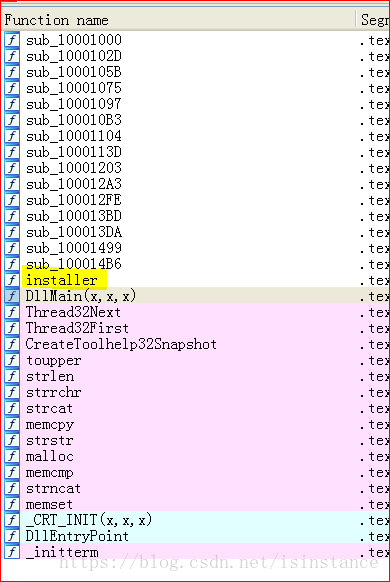
**这个没有意义的乱码说明这个程序很可能有加解密的功能**

**CHMMXaL@MV@SD@O@MXRHRCNNJBNL**

**字符串中还有wsock32.dll，这个说明这个程序可能会使用网络的功能**

**还有那么几个RCPT的字符串， RCPT是RECIPIENT 的缩写，为SMTP协议中的一个命令。说明这个代码很有可能用了邮件客户端来发送了一个邮件之类的。**

**然后我们开始准备运行这个DLL，这里我们运行的时候要加一个DLL导出函数名字，这个名字就DLL的导出函数。**

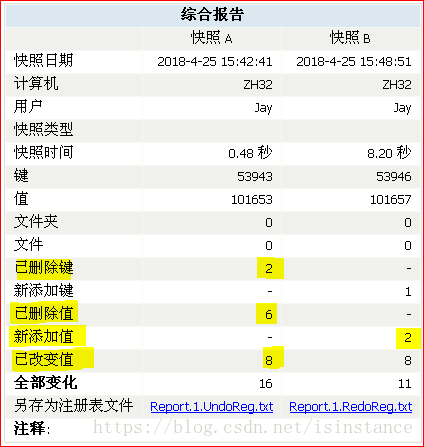
****

**我们可以在这里来找**

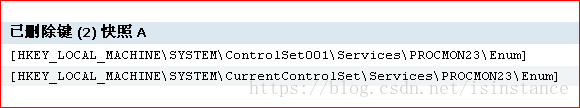
**现在我们运行，运行的同时记得做好系统的监控**

**rundll32.exe Lab11-02.dll, installer**

**执行之后会发现，什么也没出现，就结束了，然后我们看看我们的监控的结果**

****

**这里删除了两个，分别是**

****

**新增加的键**

****

**这里是在浏览器的位置增加的键**

**以改变的值里面有这么一个需要注意的**

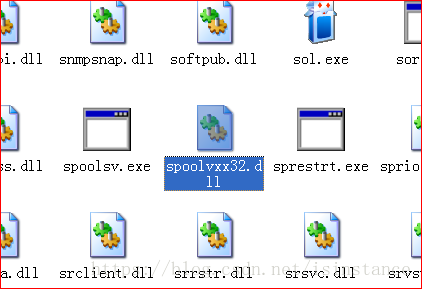
****

**这里在这个位置上修改AppInit\_DLLs的值，修改为spoolvxx32.dll，说明很有可能这里有个spoolvxx32.dll被释放出来了。**

**然后我们看看Procmon和Procexp的监控情况，对于不会常驻内存的进程来说，Procexp没多大作用。**

**我们看Procmon的记录：我们设置过滤为Process name为rundll32.exe，因为注册表的值我们刚刚已经比对过了，现在我们再设置一个过滤是操作CreateFile。**

**这里要注意的就是虽然DLL调用了CreateFile，但是不一定都是创建文件，有可能是打开文件的操作，我们最后找会发现在C:\WINDOWS\system32\下面创建了一个文件叫spoolvxx32.dll。**

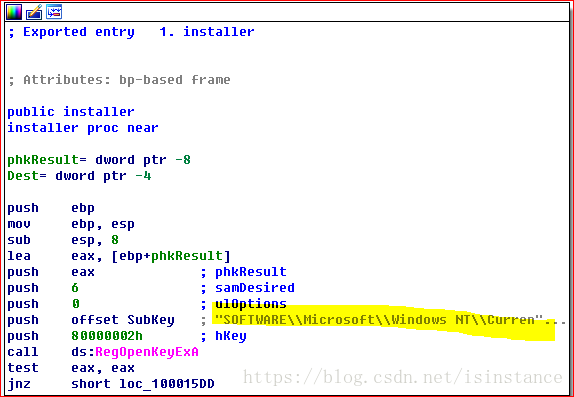
****

**其实这个文件和我们的Lab11-02.dll是一样的，最后恶意代码会尝试在C:\WINDOWS\system32\中打开Lab11-02.ini。**

**为了让这个恶意代码能访问这个文件，我们把这个Lab11-02.ini放在C:\WINDOWS\system32\下。**

**这个恶意代码会在最后将自己加载到user32.dll中，然后所有加载了user32.dll的进程也会加载了它。**

**然后我们开始用IDA来分析：**

****

**导出函数installer的内部第一个函数调用是RegOpenKeyExA**

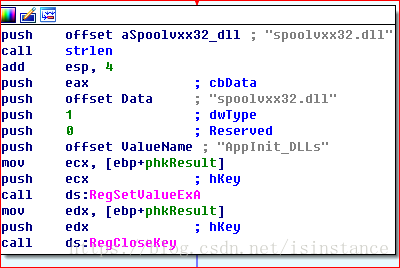
**这是打开键的函数，我们主要看他的键的位置**

**SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows**

**1**

**如果RegOpenKeyExA这个函数执行成功，函数返回ERROR\_SUCCESS也就是0**

**在test之后，因为eax为0，那么ZF=1，jnz不会跳转**

****

**之后函数会来到这里执行**

**这里的第一个函数调用是strlen，计算字符串长度的**

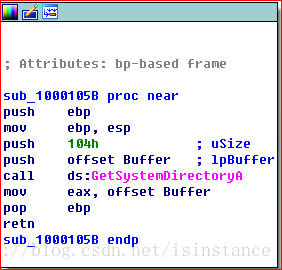
**这个函数的返回值就是eax，成为了下一个函数调用RegSetValueExA的最后一个参数cbData**

**这里调用RegSetValueExA来将键AppInit\_DLLs的值改为了spoolvxx32.dll**

**之后代码关闭了设置键的句柄**

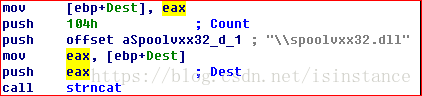
****

**之后函数就会来到这里，这里调用的第一个函数是sub\_1000105B，我们看看这个函数是干什么的**

****

**这个函数是来查询系统的目录的，函数的返回值是目录的长度，目录的路径是保存到了Buffer这里，调用完函数之后，代码将路径赋值给了eax来返回**

**然后下面的函数调用是**

****

**这里的eax其实就是上面那个sub\_1000105B函数调用之后查出来的系统路径，一般就是C:\WINDOWS\system32\**

**写成伪代码的形势就是**

**strncat("C:\WINDOWS\system32\", "\\spoolvxx32.dll", 104h)**

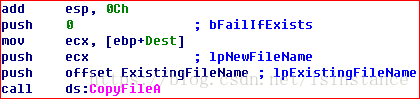
**1**

**这个就是最后会得到一个这样的字符串**

**C:\WINDOWS\system32\spoolvxx32.dll**

**1**

**之后**

****

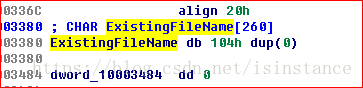
**开始调用函数CopyFileA**

**这个从字面意思就是赋值文件的意思**

**这里的入参lpNewFileName其实就是上面组装成的字符串C:\WINDOWS\system32\spoolvxx32.dll，因为strncat的第一个参数是指向[ebp+Dest]的**

**而lpExistingFileName的意思是我们要复制的文件名字**

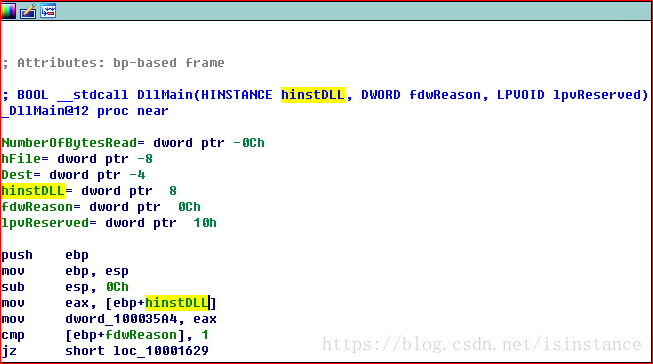
**而这个lpExistingFileName是个260大小的字符数组**

****

**因为这个函数里面并没有调用ExistingFileName这个参数，所以我们不知道这个变量的值是从哪里来的，不过我猜想就是我们现在Lab11-02.dll这个文件的绝对路径**

**之后就是一个CopyFileA之后导出函数就退出了**

**下面我们看看DLL的主函数**

****

**其中这里的hinstDLL是**

**DLL模块的句柄，该值是DLL的基地址。 DLL的HINSTANCE与DLL的HMODULE相同**

**而fdwReason是**

**原因码说明了为什么调用DLL入口函数**

**这里将DLL文件的基地址赋值到了dword\_100035a4中去**

**还比较了fdwReason和1的大小**

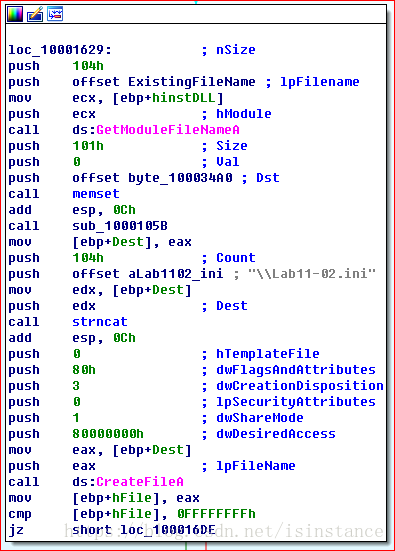
**其中1代表的意思就是**

**由于进程启动或由于调用LoadLibrary，DLL正被加载到当前进程的虚拟地址空间中， DLL可以使用此机会初始化任何实例数据或使用TlsAlloc函数分配线程本地存储（TLS）索引**

**如果这个值等于1的话，cmp之后ZF=1**

**则jz会跳转，如果不跳转，就会退出了DLLMain**

**之后**

****

**我们上面说的hinstDLL指向的是这个DLL的基地址**

**所以这里的GetModuleFileNameA其实返回的是当前这个DLL的绝对路径**

**之后调用memset分配了几个空间**

**所以我们这里知道了，ExistingFileName的值其实就是这个DLL的绝对路径**

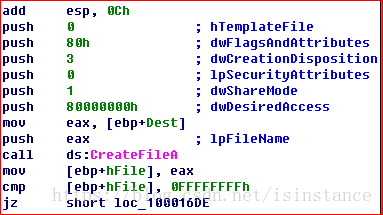
**之后会调用sub\_1000105B这个函数我们上面分析过，会返回系统目录的路径**

**之后调用了strncat，不过这次最后拼接出来的字符串是这样的**

**C:\WINDOWS\system32\Lab11-02.ini**

**1**

**这就印证我们在Procmon中看到的调用C:\WINDOWS\system32\Lab11-02.ini的操作**

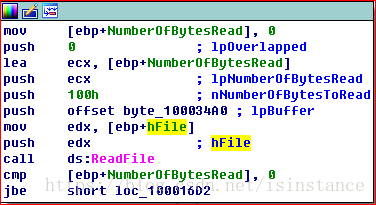
****

**之后代码会调用CreateFileA，这里的lpFileName其实就是我们上面strncat返回的那个最终的字符串**

**也就是这里会创建一个文件在**

**C:\WINDOWS\system32\Lab11-02.ini**

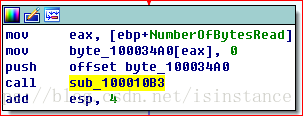
**1**

****

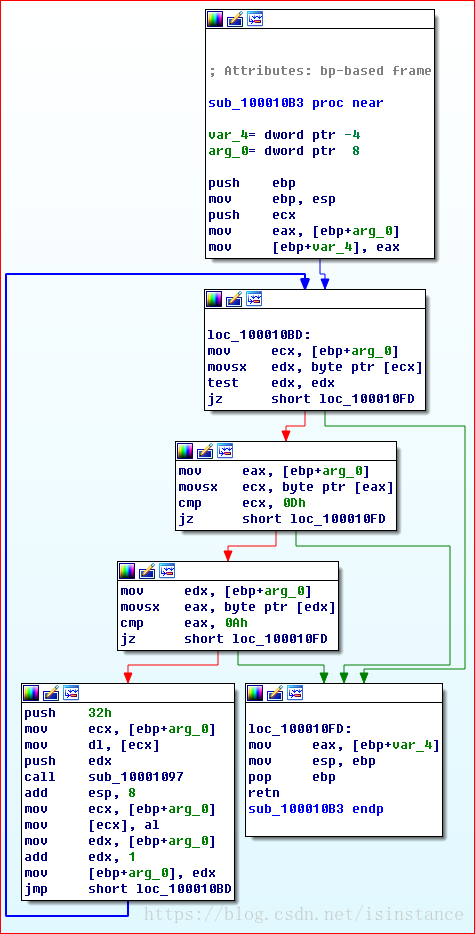
**然后这里开始调用ReadFile来读这个文件**

**这里的[ebp+hFile]其实在上面被eax赋值了**

**之后的函数调用**

****

**这里把我们上面读进内存的数据传了进去sub\_100010B3中**

****

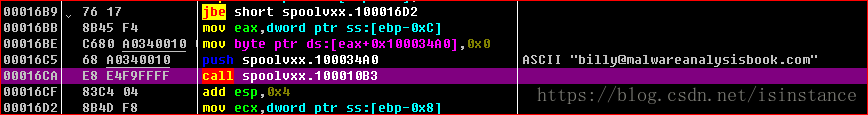
**这个函数我们看着像是一个解密的函数，这里我们可以硬分析这个编码，但是我们也可以通过OD运行之后直接看结果就行了~**

****

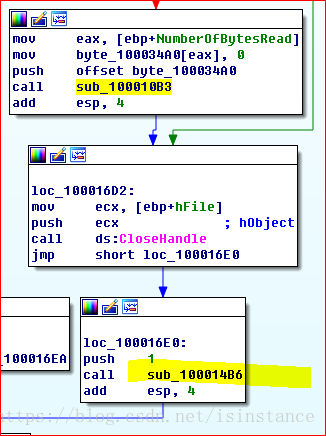
**这里我们找到函数的地址**

**100016CA**

**跳到这个位置的时候，OD中已经自动标注出了这个字符串的解密：**[**billy@malwareanalysisbook.com**](mailto:billy@malwareanalysisbook.com)**。**

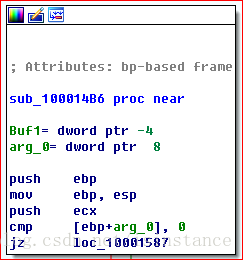
****

**然后下面还有一个函数调用**

****

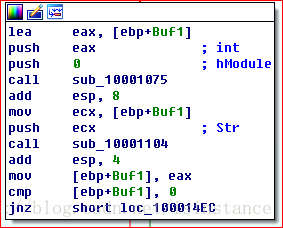
**在这里的地方，叫sub\_100014B6，这个函数就是书上说的hook\_install函数，这个函数会在这里安装恶意代码的hook**

**这个函数入参了一个1**

****

**之后这里会比较入参的值是否和0相当呢个，之后如果这个值是0的话，cmp之后ZF=1，则jz跳转，跳转之后就直接退出了**

**如果不等于0的话，像我们现在的值是1，就不会跳转继续往下执行**

****

**之后函数会来到这里**

**这里出现一个[ebp+Buf1]的地址**

**这里的Buf1等于-4**

**我们画一下栈图就知道这个值是多少了**

**-------**

**| ecx | <--- ebp-4**

**-------**

**| oebp | <--- ebp**

**-------**

**| 1 | <--- ebp+4**

**-------**

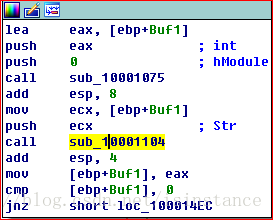
**.**

**.**

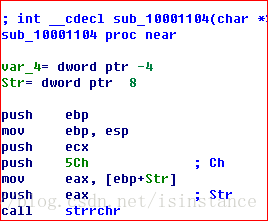
**.**

**所以[ebp-4]其实就是ecx的地址**

**然后这步第一个调用的函数是sub\_10001075，这个函数会返回系统的系统路径**

****

**之后函数又会调用sub\_10001104这个函数**

****

**之后函数进入这个函数之后的第一调用是strrchr**

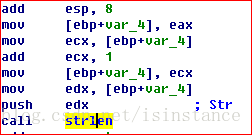
**这个函数的意思就是找到最后在字符串中最后出现某个字符的位置**

**这里我们可以推算得出，这个eax或者说strrchr的第一个入参其实就是我们的系统路径的返回值，因为外面的一个函数将ebp+Buf1的地址赋值给了eax，于是sub\_10001104的返回值就会直接写到ebp+Buf1上**

**所以这里的Str就是C:\WINDOWS\system32**

**然后要查找的值是5Ch换算成ASCII的话就是“\”。这个会查找最后一个字符为\出现的位置，这里注意我们的系统路径最后是没有\，也就是我们的系统路径只是C:\WINDOWS\system32，而不是C:\WINDOWS\system32\。因为我们前面拼接字符串的时候，后面拼接的字符串是这样的：\\Lab11-02.ini**

**所以前面的字符串是不会提供\的。于是我们这里的strrchr返回的就是“\system32”。**

****

**这里先将返回值eax赋值给了[ebp+var\_4]，然后将又将[ebp+var\_4]赋值给了ecx，然后ecx又加1，**

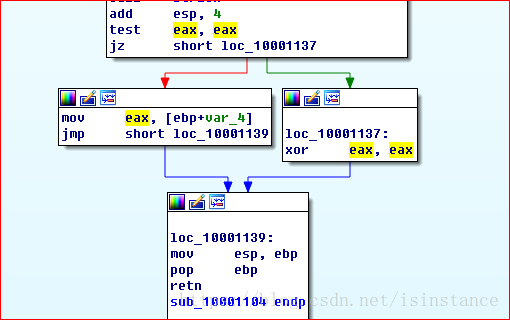
**这里的ecx代表了字符串的地址，于是+1就是往后偏移了一个位置**

**最后的字符串就变成了**

**system32**

**1**

**然后计算字符串的长度strlen**

****

**返回值是在eax中，这里计算eax的值是否为0**

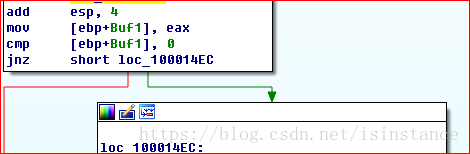
**如果是0的话，test之后ZF=1**

**之后jz就会跳转，跳转之后就会看到将eax异或置0了**

**如果不跳转，也就是eax不为0**

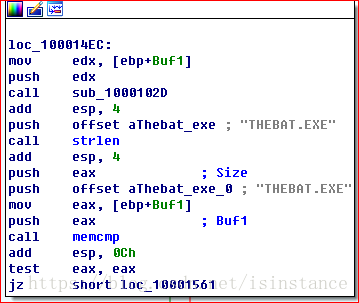
**之后就会将[ebp+var\_4]的值赋值给了eax也就是system32这个字符串**

**之后函数就退出了**

****

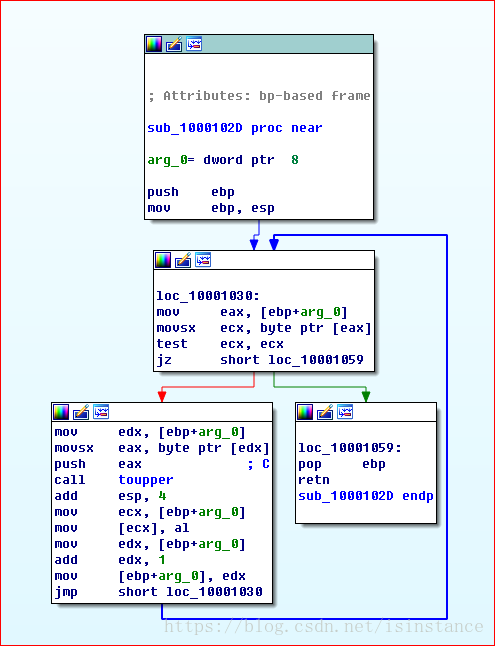
**从函数中出来之后，将返回值的eax赋值给了[ebp+Buf1]之后，和0比较大小，如果为0就jnz跳转之后就结束了**

**然后假设我们这里没有跳转结束**

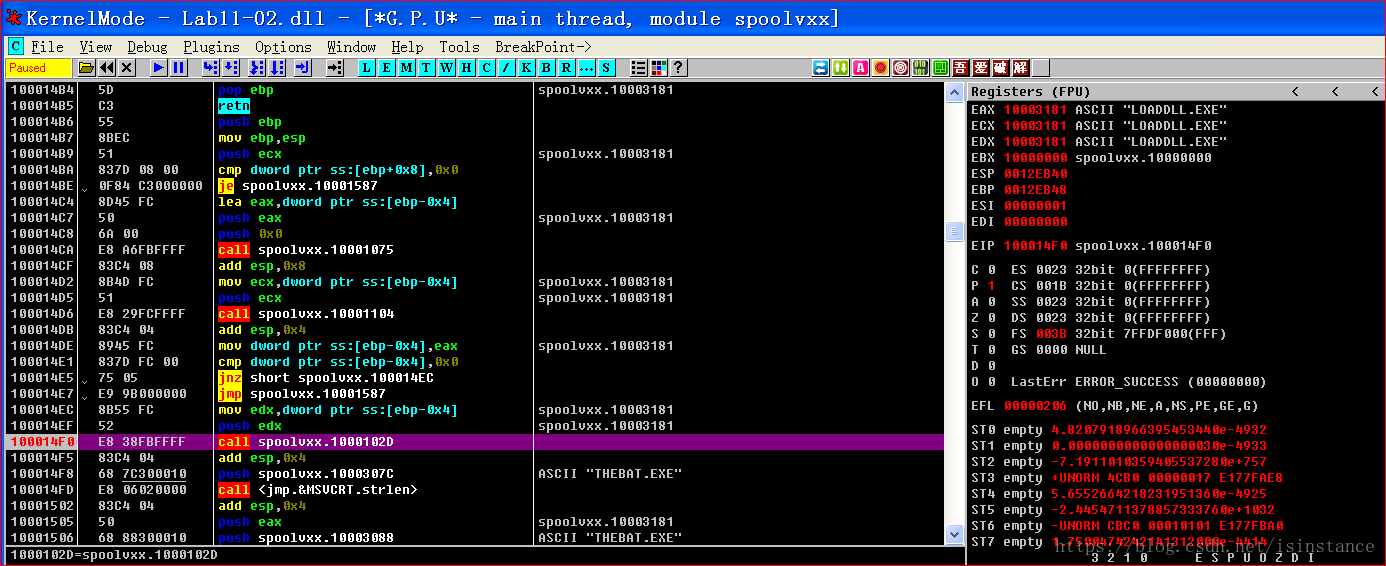
****

**之后函数就会来到这里执行**

**这里的第一个函数调用是sub\_1000102D这个函数**

****

**我们可以用OD来进行动态分析：**

****

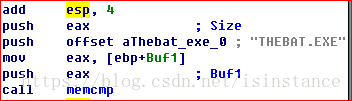
**我们可以看见，这时候我们入栈的参数是LOADDLL.EXE这个字符串**

**通过动态运行我们可以看出，这个函数sub\_1000102D作用其实就是将字符串中的字母由小写变成大写的字母。**

**然后一个栈缩小之后，将esp指向我们变成大写的字符串**

**图片**

**之后计算字符串THEBAT.EXT的长度**

****

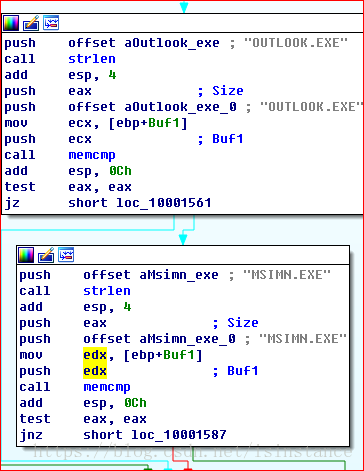
**之后又一个add esp, 4**

**将esp指回我们的变成大写的字符串**

**然后这里通过memcmp函数来比较我们刚刚变成大写的那个字符串和现在这个字符串THEBAT.EXE的相等与否**

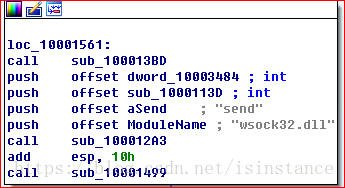
**比较的长度为strlen("THEBAT.EXT")**

**如果不想等，就会继续比较其他字符串**

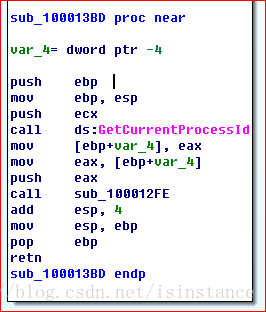
****

**包括OUTLOOK.EXE和MSIMN.EXE**

**如果其中一个相等，马上跳转到这里执行这些代码，如果不相等，马上跳转结束**

****

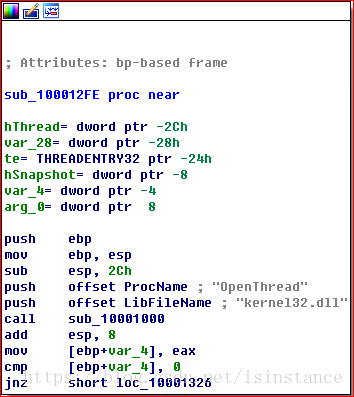
**这里首先调用的是sub\_100013BD这个函数**

****

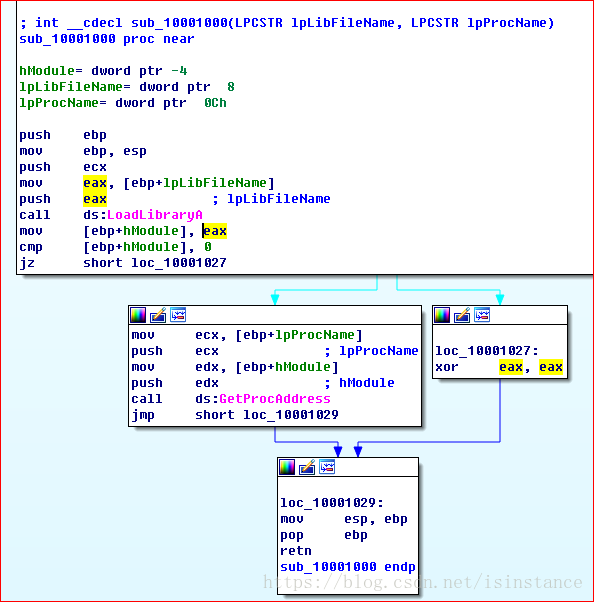
**这个函数首先会调用GetCurrentProcessId，这个函数是返回这个进程的PID的**

**之后准备调用sub\_100012FE**

**在函数sub\_100012FE中我们可以看到一些有意思的调用操作**

****

**第一个函数调用是sub\_10001000，这个函数的目的主要是返回第一个push的DLL的基地址**

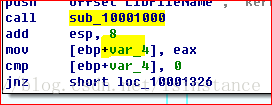
****

**函数在调用LoadLibraryA的时候，push入栈的参数是eax，也就是[ebp+8]，这个地址其实是调用这个函数的第一个入参，也就是我们上面的kernel32.dll**

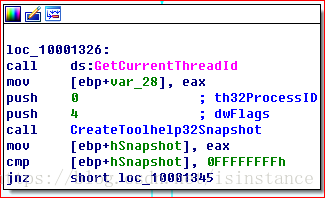
**这里恶意代码用LoadLibraryA将这个kernel32.dll加载到内存中**

**之后把这个句柄保存到hModule中，之后函数会调用GetProcAddress来检索kernel32.dll上函数，我们来看这个函数名字的地址是lpProcName，也就是ebp+0Ch，这个地址其实指向的是我们上面的那个入参OpenThread**

**然后函数找到了这个函数的地址，就返回了**

****

**我们注意这里的之后函数将返回值，也就是OpenThread函数的地址，赋值给了[ebp+var\_4]也就是[ebp-4]**

****

**之后代码会调用GetCurrentThreadId函数，这个函数的作用是获取当前线程的标识符**

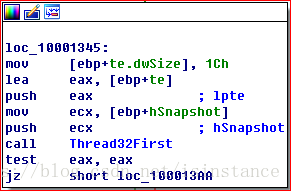
**之后会调用CreateToolHelp32Snapshot来获取当前进程的**

**获取指定进程的快照，以及这些进程使用的堆，模块和线程**

**这里的dwFlags的值是0x4也就是代表了TH32CS\_SNAPTHREAD，意思就是**

**快照包括系统中的所有线程，同时会枚举线程**

**之后代码把返回值保存到了hSnapshot中**

****

**之后代码会来到这里**

**代码会调用Thread32First函数，来**

**检索有关系统快照中遇到的任何进程的第一个线程的信息**

**这里的[ebp+te.dwSize]的te是个THREADENTRY32结构**

**描述当拍摄快照时从系统中执行的线程列表中的条目**

**调用函数Thread32First成功之后，返回的结构就会保存到[ebp+te]中，这个结构是这样的**

**typedef struct tagTHREADENTRY32 {**

**DWORD dwSize;**

**DWORD cntUsage;**

**DWORD th32ThreadID;**

**DWORD th32OwnerProcessID;**

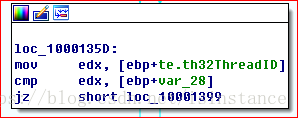
**LONG tpBasePri;**

**LONG tpDeltaPri;**

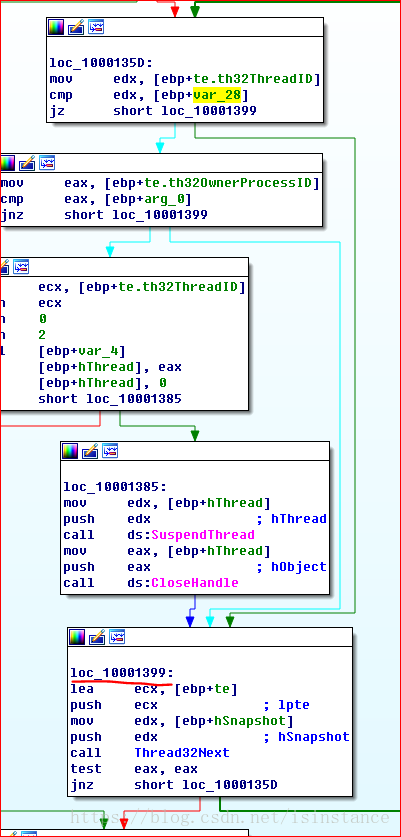
**DWORD dwFlags;**

**} THREADENTRY32, \*PTHREADENTRY32;**

**之后函数会把结构中th32ThreadID和[ebp+var\_28]比较，这个var\_28是函数当前线程的标识符**

****

**如果两个值相等的话，执行cmp之后ZF=1，那么jz就会跳转到最下面的loc\_10001399这个地方**

****

**这里将我们的那个结构和hSnapshot压入了栈，然后就调用了Thread32Next这个函数**

**这个函数在MSDN中的解释如下**

**检索有关系统内存快照中遇到的任何进程的下一个线程的信息**

**然后这个函数的返回值是这样的**

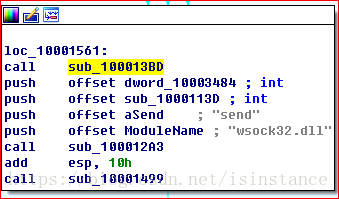
**如果线程列表的下一个条目已被复制到缓冲区，则返回TRUE，否则返回FALSE**

**假设返回的是TRUE，在计算机中TRUE的值一般是非零值，而FALSE的值一般是0**

**所以这里test返回值eax的时候，如果在内存中找不到已经复制的缓冲区，返回了FALSE，也就是0，那么ZF=1，之后就会沿着红线执行**

**就会结束这个小代码，如果找到了缓冲区，那么就会跳回去迭代循环比较**

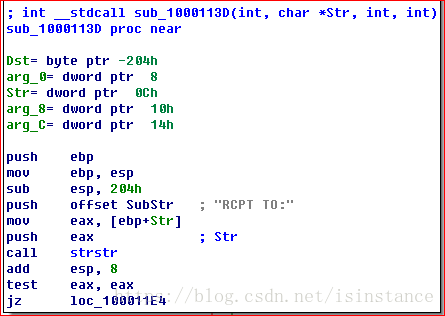
**好了，我们退出这个函数的分析，回到主函数**

****

**我们上面是在sub\_100013BD里面分析这个函数的作用，他是一个在进程中搜索某个线程的函数**

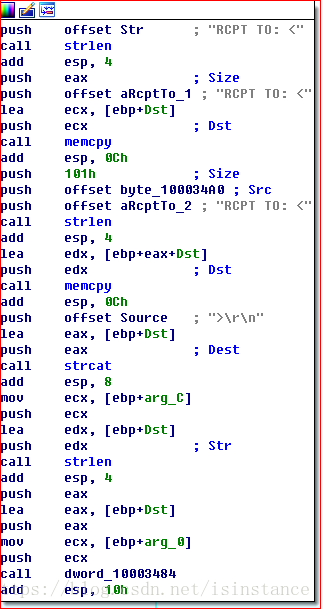
**之后下面的函数会将这个一个int类型的值和sub\_1000113d压入栈中**

**其中我们快速看一下sub\_1000113d这个函数的作用**

****

**这个函数的第一个call是strstr函数**

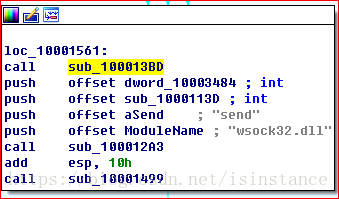
**而两个入参一个是字符串RCPT TO:，另一个是函数sub\_1000113d的第二个入参，也就是上面的第二个char \*Str的这个**

****

**我们可以看一下剩余的函数，基本可断定，这是一个构造邮件的函数**

**且如果在邮件中发现了RCPT TO的字样，这个代码会加上我们在ini文件中发现的邮箱，加上这个邮箱，这台电脑发送的邮件都会发到这个邮箱一份，这应该是一个盗窃邮件中关键信息的病毒**

**我们会到外面的函数**

****

**然后还会压栈几个参数，一个是字符串send，一个是字符串wsock32.dll，这个wsock32.dll函数是用来发送东西用的**

**之后就会调用函数sub\_100012A3这个函数，我们看看这个函数**

****

**这个函数会去查找那个wsock32.dll函数**

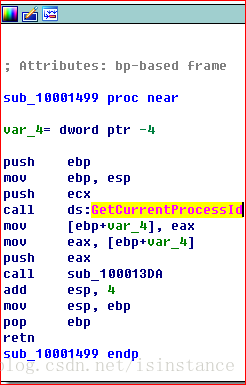
**其中MSDN对GetModuleHandleA的定义是这样的**

**检索指定模块的模块句柄。该模块必须已由调用进程加载**

**这个函数其实就是找wsock32.dll的地址的作用**

**之后会调用sub\_10001499这个函数**

**这个函数我们也是粗略的看一下**

****

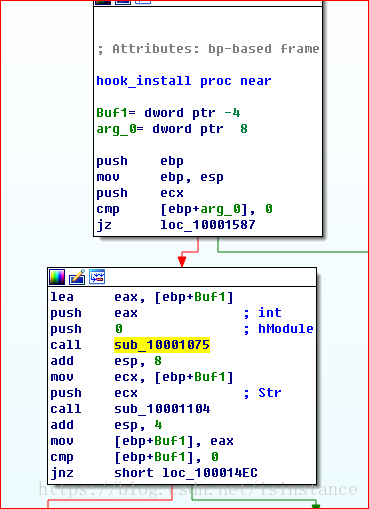
**这个函数的第一个调用的是GetCurrentProcessId，这个函数在MSDN中的解释是**

**检索调用进程的进程标识符**

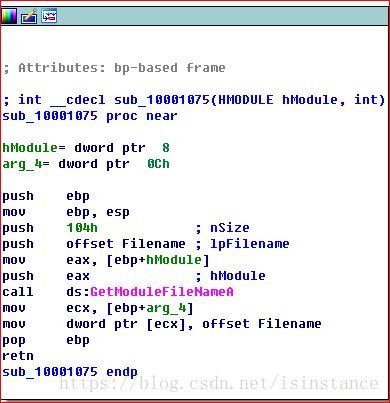
**其后就会退出这个函数了**

**上面分析了这么多偏的函数，我们现在总结归纳一下（因为时间过去久远）**

**函数进入了这个hook\_install之后**

****

**第一个调用会是sub\_10001075这个函数，这个函数的里面就是**

****

**有一个GetModuleFileNameA，因为这个函数的hModule参数的值被设置为了0**

**所以这个函数的会返回加载了这个DLL的进程的绝对路径，之后就会把这个路径返回了**

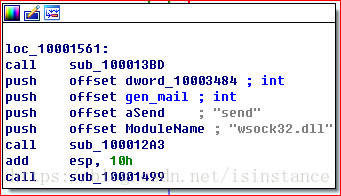
**以AppInit\_DLLs作为驻留机制的恶意代码常常使用GetModuleFileNameA。这个恶意DLL几乎被加载到系统中的所有启动的进程，而恶意代码作者可能只是针对某些进程，所以他们必须确定运行恶意代码进程的名称**

**确定了这个返回值之后，之后会剔除多余的路径名，并且进行大写变换**

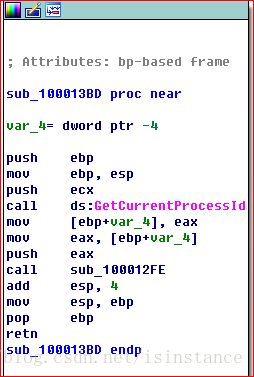
**之后就会于进程名THEBAT.EXE和OUTLOOK.EXE还有MSIMN.EXE比较，确定这个加载了恶意DLL的进程是这三个中的一个**

**如果不属于他们中的一个，就会退出**

**然后代码确定了自己加载到了怎么三个进程中的其中一个，就会执行下面的代码**

****

**在第一个调用sub\_100013BD中，我们可以看到以下内容**

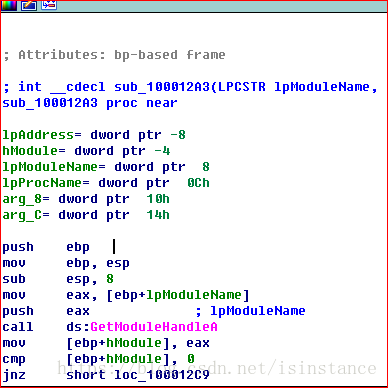
****

**这个函数里面第一个调用是GetCurrentProcessId这个，然后就会调用sub\_100012FE这个函数，这个函数会返回当前运行进程内的所有线程标识符（TID，进程标识符是PID），之后这个sub\_100012FE就会调用CreateToolHelp32Snapshot来遍历所有的线程**

****

**如果这个线程不是当前的线程，就会挂起这个线程，所以这个函数会挂起所有不是当前线程的所有线程**

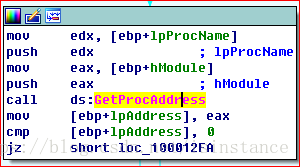
**然后下一个函数sub\_100012A3这里，这个函数就是执行安装挂钩的地方**

****

**这里第一个调用是GetModuleHandleA来获取wsock32.dll的句柄，之后用LoadLibrary加载这个DLL到空间中**

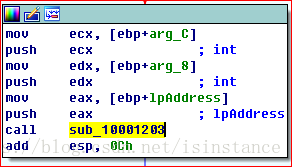
**将指定的模块加载到调用进程的地址空间中。指定的模块可能会导致其他模块被加载。**

**之后函数在wsock32.dll中查找send函数的地址**

****

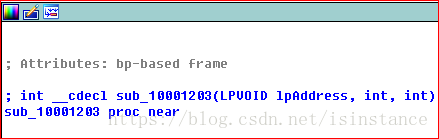
**并且把这个地址赋值到了lpAddress中存起来了**

**之后就会调用sub\_10001203**

****

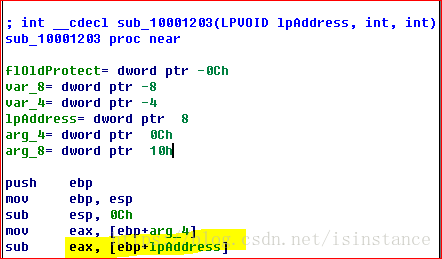
**这里的arg\_C和arg\_8就是我们外面调用这个函数的时候那两个int类型的入参**

**这里的sub\_10001203函数的入参是这样的**

****

**第一个参数是我们的send函数的地址，第二个参数就是arg\_8，第三个是arg\_C**

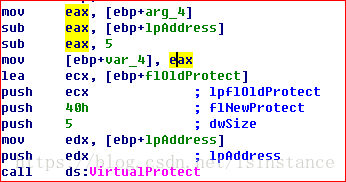
**这里会计算一个地址差**

****

**arg\_4是我们第二个参数，这个int类型的参数其实是sub\_1000113D的地址，这个函数就是构建邮件文件的那个函数**

**之后我们代码开始计算两个地址之间的地址差值**

**之后将这个差值减去5之后，存储在一个变量var\_4中**

****

**这里为什么要减去5我们继续往下看就知道了**

**之后代码会调用函数VirtualProtect**

**这个函数的作用是**

**‎> 更改调用进程的虚拟地址空间中已提交页的区域的保护**

**然后地这个函数的lpAddress参数的值是**

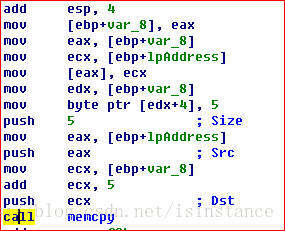
**一个地址, 描述要更改其访问保护属性的页区域的起始页**

**这个地址就是我们构造邮件发送的那个函数的地址，这里用VirtualProtect改变了这个地址的保护属性**

**之后，这个函数会调用malloc**

**突破**

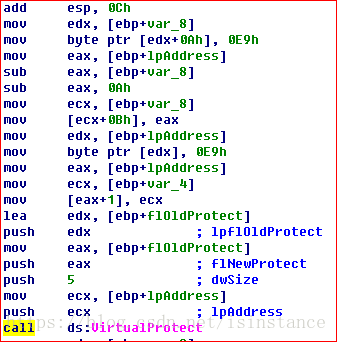
**分配了一个0FFh大小的空间**

****

**下一个调用的函数就是memecpy这个函数**

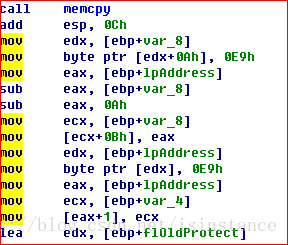
**这个函数的作用是将我们上面刚刚分配好的空间，复制了send函数的前五个字节，为了方便跳转回来之后保证send函数代码的完整性**

**之后**

****

**在段函数我们应该分成两段来看**

**第一段是一个暂时不知道什么操作的代码，从memcpy之后到lea操作**

****

**第二段就是我们调用VirtualProtect这个函数的的函数入栈操作，这个段函数会再次调用VirtualProtect函数来对我们的地址进行保护更改**

**我们先看第一段**

**通过代码**

**mov edx, [ebp+var\_8]**

**来把memcpy函数分配的空间的地址赋值给了edx**

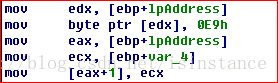
**之后从空间偏移量为0Ah的地方开始，将值赋值成为0E9h，这个十六进制书上已经提醒我们是jmp的操作码，因为这里在我们从正常的send跳到恶意函数之后，要从恶意函数调回来正常的send函数，也会计算send到我们分配的内存空间的地址差，之后就会利用这个地址差来跳转到我们的复制了5字节send函数代码的空间，之后用这个代码来接上send函数之下的代码**

**这样可以保证代码的运行**

**之后会再次计算构造邮件那个函数的地址和memcpy分配空间地址之间的差值**

**之后将这个差值减去0Ah**

**下面**

****

**代码将0E9h机器码复制到send函数的开头，这里的edx其实就是send\_address**

**之后代码将我们的var\_4的值复制给ecx，var\_4中存储了send函数和我们的构造邮件函数地址send\_address之间的地址差值，这个差值还要减去5**

**这里有个问题，书上说，var\_4存储的是我们想要跳转的地址，但是在这段代码中调用了var\_4一共就两次，现在我们遇到的是第二次也是最后一次**

**然后第一次是计算函数send和sub\_1000113d之间的差值，并把这个值减去5，之后保存在var\_4中，为什么这个var\_4现在成了要跳转的函数地址？**

**这个var\_4不是应该是函数之间地址的差值么？**

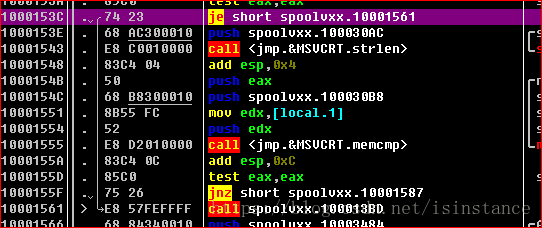
**这里的有一个比较难以理解的就是这个地址的差值为什么要减去5**

**因为这个5，是存储0E9h和我们要跳转地址的所需空间，也就是比如我们的send函数和sub\_1000113d之间差值是10d，且send函数在sub\_1000113d之前，且send函数的开始地址是0，那么推理可知sub\_1000113d的地址就应该是10d，这里用十进制说明了，方便理解**

**但是我们的0E9h和地址查占了5个字节的空间，之后就如果我们要跳转地址10d的地方，我们就需要要写jmp 5d，这个5d的意思是在jmp开始的地方开始数，往后5个地址**

**这里有个概念就是，函数的跳转地址，不是直接写0x1000113d这种，而是往后跳转多少字节这种**

**比如我们这里，有一个代码是这样的，这里用的是je跳转**

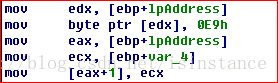
****

**0x74h是je的机器码，然后后面的0x23h就是我们要往后跳转多少字节**

**我们地址的下一个地址开始是0x1000153eh，那么加上0x23h之后，就是0x10001561h，所以就会跳转到这里的call spoolvxx.100013BD这个地方**

**这是汇编的跳转原理，理解这里，上面这个跳转就好理解了**

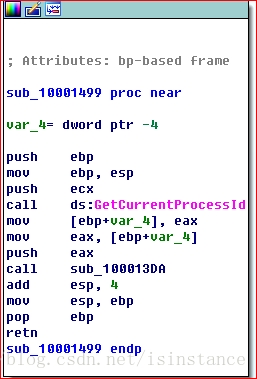
**函数先计算了地址差值，之后减去5，是为我们将来要增加的机器码和跳转地址做准备，之后**

****

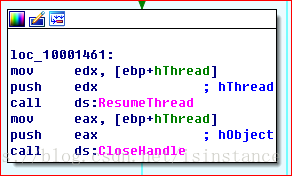
**我们将我们要跳转的字节写入send\_address中，这样在我们调用send函数的时候，代码找到send的地址，一进去第一个代码就是jmp，这样就将我们的send函数替换成为了gen\_mail函数，而这个函数会发送恶意代码定义的东西**

**这个所谓的hook\_function的工作原理就是这个**

**之后函数在最后一个调用sub\_10001499这里**

****

**会执行和sub\_100013BD相反的操作**

****

**之后这个函数会恢复所有的进程**

**1. 这个恶意DLL导出了什么？**

**解答： 这个恶意DLL中包含了一个教installer的导出函数**

**2. 使用run32dll.exe来安装这个恶意代码之后，发生了什么？**

**解答： 恶意代码会将spoolvxx32.dll复制到系统目录中C:\\WINDOWS\system32\**

**之后会从这个目录中打开Lab11-02.ini**

**3. 为了使这个恶意代码正确安装，Lab11-02.ini必须放置在何处？**

**解答： 放在C:\\WINDOWS\system32\下**

**4. 这个安装的恶意代码如何驻留？**

**解答： 恶意代码会将自己安装到AppInit\_DLLS中，然后这个恶意代码就可以load到了所有导入了User32.dll的进程中**

**5. 这个恶意代码采用的用户态Rootkit技术是什么？**

**解答： 这个恶意代码针对wsock32.dll中的send函数安装了一个inline挂钩**

**6. 挂钩代码做了什么？**

**解答： 这个挂钩会检查对外的所有邮件，之后，会增加一个RCPT TO的邮箱**

**7. 哪个或者哪些进程执行了这个恶意攻击，为什么？**

**解答： 这个恶意代码仅针对的是MSIMN.exe或者THEBAT.exe或者OUTLOOK.exe这些程序，除非这些程序运行在进程空间中，不然这个代码不会安装**

**8. .ini文件的意义是什么？**

**解答： 这个ini文件提供了一个恶意的邮箱地址**

**9. 你怎样用WireShark动态抓获这个恶意代码的行为？**

**解答： 可以看到这个恶意代码发送的邮件信息**

**Lab09-03**

1. **Yara规则**

**根据以上分析结果，得到Yara规则如下。**

**rule lab0901**

**{**

**strings:**

**$string1 = "Malservice"**

**$string2 = "http://www.malwareanalysisbook.com"**

**$string3 = " Internet Explorer 8.0"**

**condition:**

**filesize < 100KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and all of them**

**}**

**rule lab0902**

**{**

**strings:**

**$string1 = "http://www.practicalmalwareanalysis.com/cc.html"**

**$string2 = "CoCreateInstance"**

**condition:**

**filesize < 100KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and all of them**

**}**

**rule lab0903**

**{**

**strings:**

**$string1 = "127.26.15.13"**

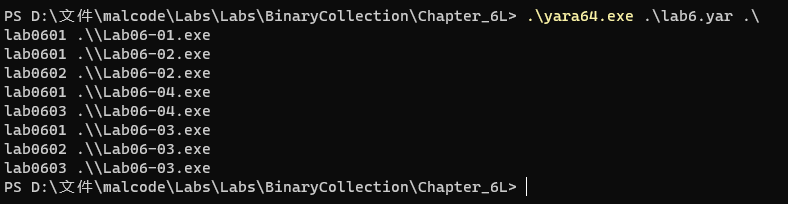
**$string2 = "ping www.malwareanalysisbook. com"**

**$string3 = "Lab09-03.dll"**

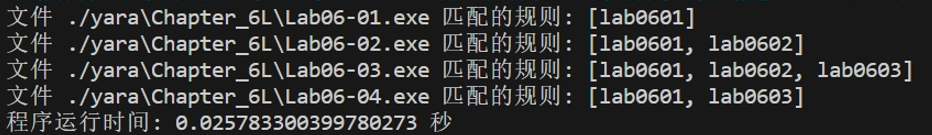
**condition:**

**filesize < 100KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and all of them**

**运行结果如下，可看到成功匹配所有文件。**

****

**在存放所有样例的文件夹下测试其检测效率，结果如下。**

****

1. **IDA Python脚本编写**

**遍历所有函数，排除库函数或简单跳转函数，当反汇编的助记符为call或者jmp且操作数为寄存器类型时，输出该行反汇编指令。**

**import idautils**

**for func in idautils.Functions():**

**flags = idc.GetFunctionFlags(func)**

**if flags & FUNC\_LIB or flags & FUNC\_THUNK:**

**continue**

**dism\_addr = list(idautils.FuncItems(func))**

**for line in dism\_addr:**

**m = idc.GetMnem(line)**

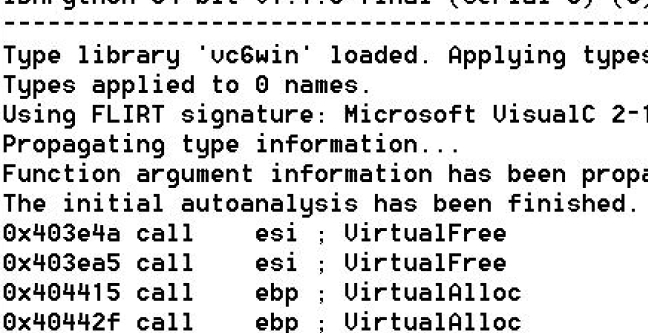
**if m == 'call' or m == 'jmp':**

**op = idc.GetOpType(line,0)**

**if op == o\_reg:**

**print '0x%x %s' % (line,idc.GetDisasm(line))**

**结果如下**

****

1. **实验结论及心得体会**
2. **对程序结合IDA和Ollydbg的分析有了进一步了解。**
3. **了解了更多Windows恶意代码的形式和特征。**