

**《恶意代码分析与防治技术》课程实验报告**

**实验十一**

****

学 院 网络空间安全学院

专 业 信息安全

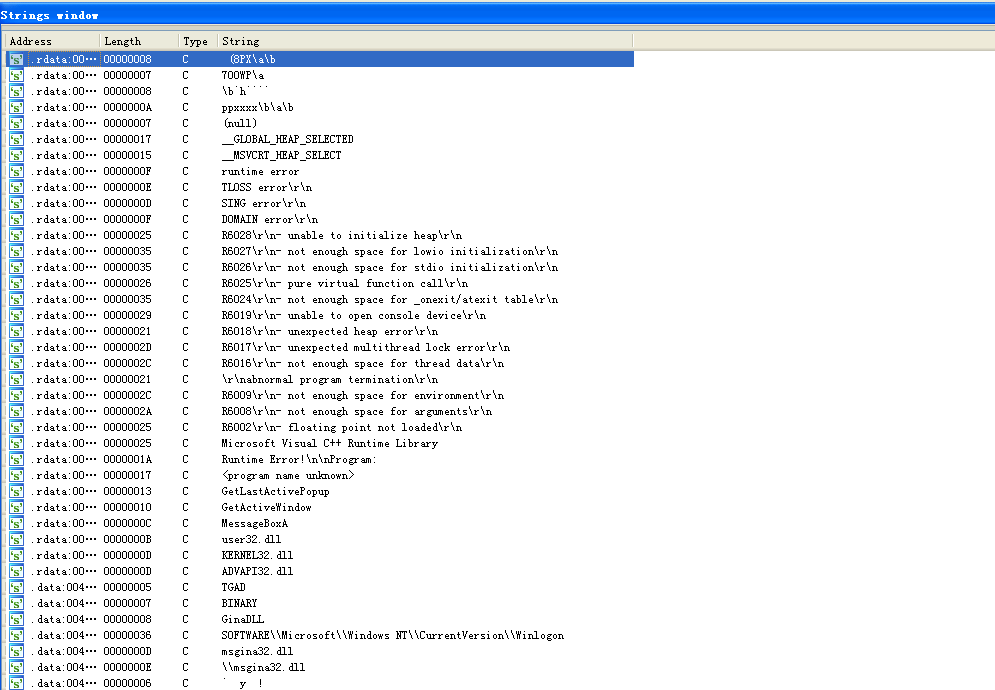
学 号 2112060

姓 名 孙蕗

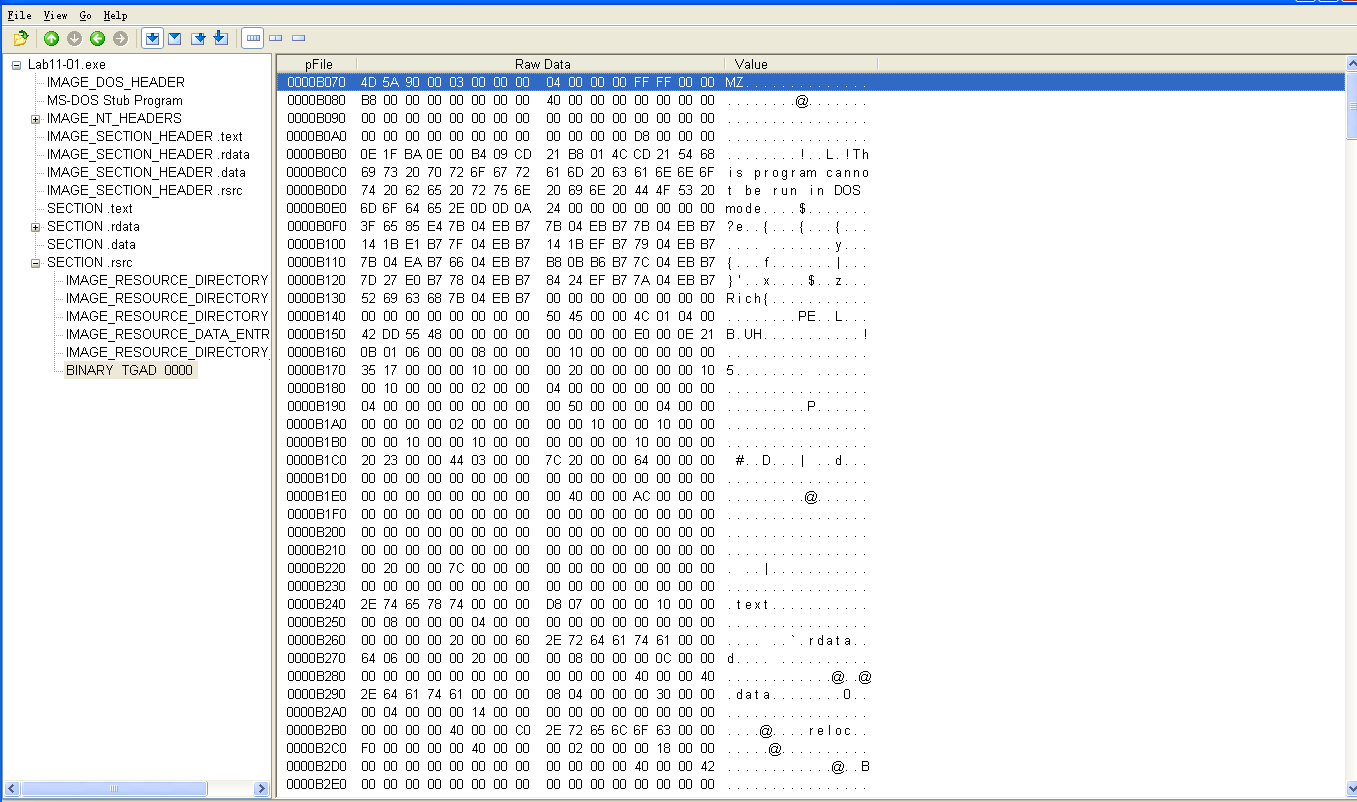
1. **实验目的**
2. 理解恶意代码分析的基本流程和工具的使用。
3. 掌握使用工具如IDA Pro、PEView、Procmon、Strings等进行恶意代码分析的方法。
4. 了解恶意代码对系统的影响，尤其是对用户登录凭证的窃取。
5. 分析挂钩技术的实现，了解用户态Rootkit的原理。
6. 分析恶意代码的网络活动
7. 了解恶意代码的潜在威胁和对系统的影响。
8. **实验原理**
9. 使用IDA Pro对恶意代码进行静态分析，查看字符串列表、资源节、导出函数表等，以了解代码的结构和功能。
10. 使用PEView查看目标可执行文件的文件结构，了解其资源节和导出函数表。
11. 使用Procmon监视目标程序的行为，识别文件创建、注册表修改、网络通信等关键操作。
12. 使用Strings工具查看二进制文件中的字符串，寻找可能的关键信息和指示。
13. 使用 Wireshark 监听网络流量，观察是否有异常的通信行为。
14. 通过逆向工程，查看具体的函数实现，了解恶意代码的行为和对系统的影响。
15. 检查注册表的修改，了解恶意代码如何实现在系统启动时的驻留。
16. 通过模拟用户登录和注销的过程，观察恶意代码对用户登录凭证的窃取和记录。
17. 分析挂钩技术的实现，了解如何在send函数上安装inline挂钩，实现用户态Rootkit的功能。
18. **实验过程**
19. **Lab11-1**

Ida查看字符串列表，存在一系列报错信息等可疑字符串如 Software\Microsoft\Windows NT\CurrentUersion\Winlogon 和 GinaDLL 。

这两个字符串连起来猜测这个样本会对图形化界面和windows的登录界面有一些行为，有可能就是拦截Gina，窃取用户的凭证。

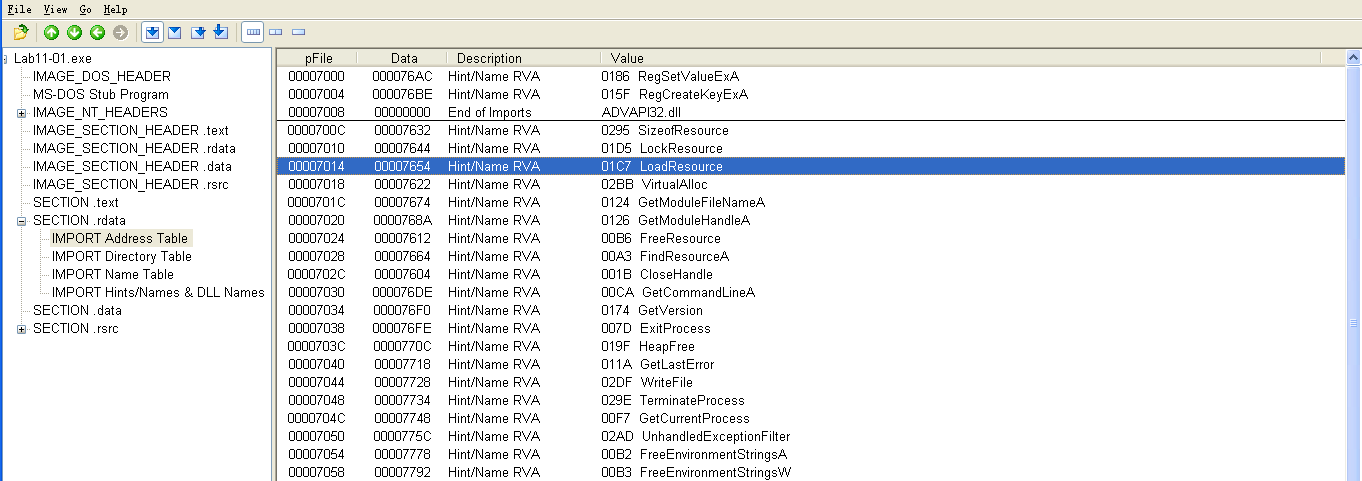


PEView查看Lab11-1.exe的文件结构。



看到一个名为TGAD的资源节。在PEview中单击该节时，看到TGAD包含一个内嵌的PE文件。

查看资源节的内容，看到在这个.rsrc节中，有一个叫BINARY TAGD 0000的二进制文件，点开后可以看到字符串This program cannot be run in DOS mode，这是一个可执行程序。



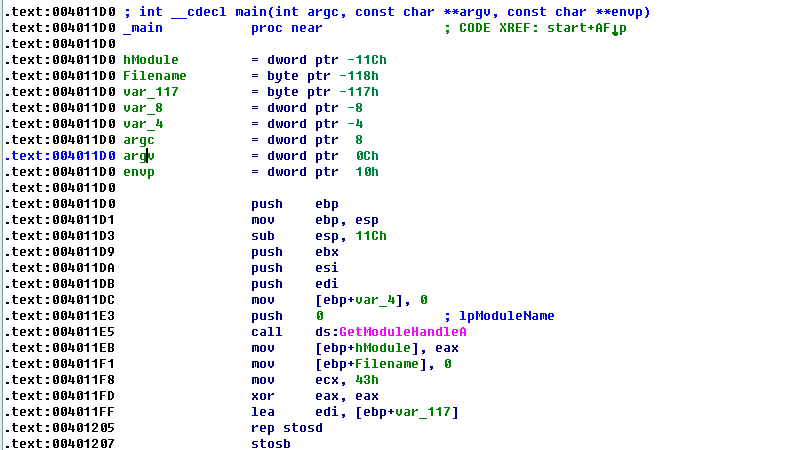
使用Procmon查看样本行为，设置Lab11-02.exe的过滤器。启动恶意代码，发现他创建了一个名为msgina21.dll文件。恶意代码插入msgina32.dll的路径到注册表HKLM\SOFTWARE\Microsoft\tVersion\

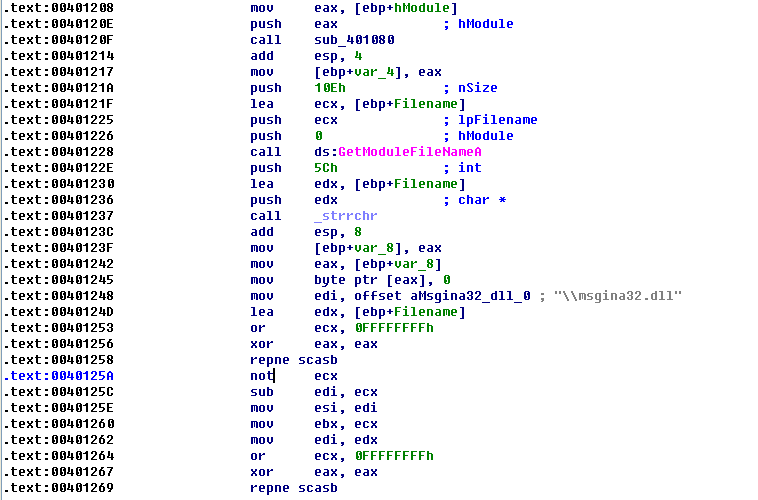
Winlogon\GinaDLL中，所以当系统重启时，WinLogon会加载这个DLL。

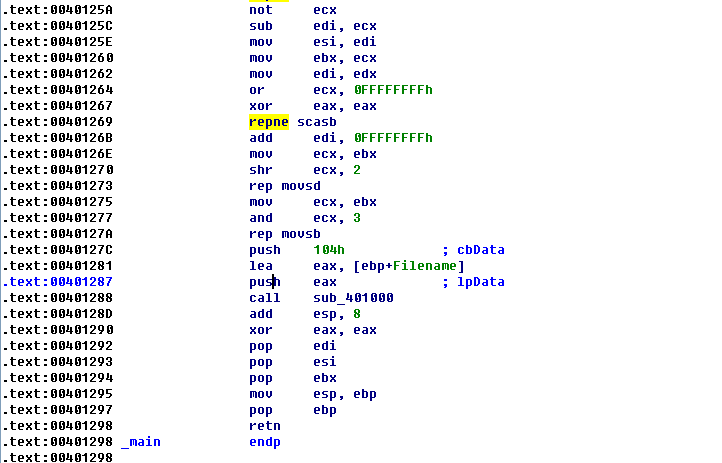
查看main函数，到main函数调用了两个函数：sub\_401080(提取TGAD资源节到msgina32.dll)和sub\_401000(设置GINA注册表的值)。

推测Lab11-01.exe是针对msgina32.dll的安装器，它在系统启动期间由Winlogon加载。运行Lab11-01.exe后，文件夹内多了一项msgina32.dll。

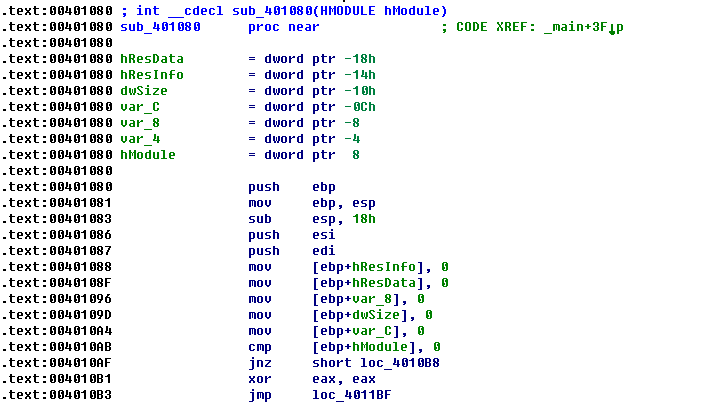




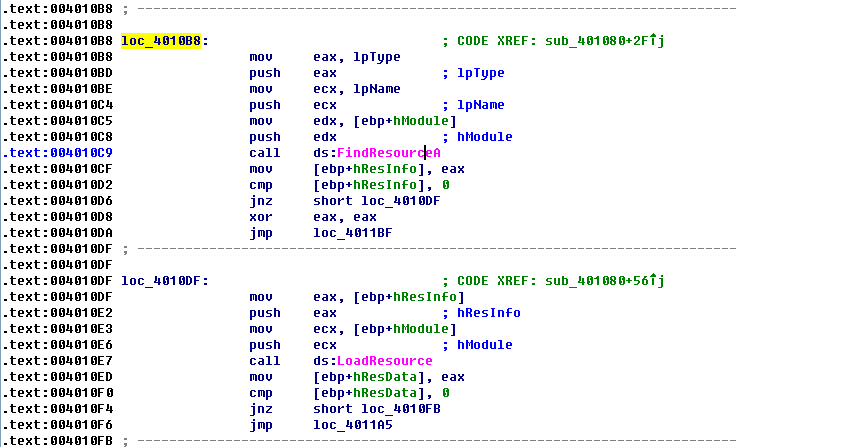


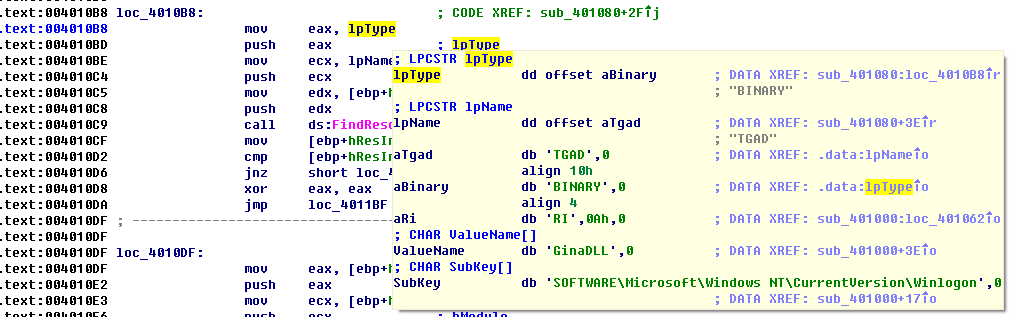


Sub\_401080:



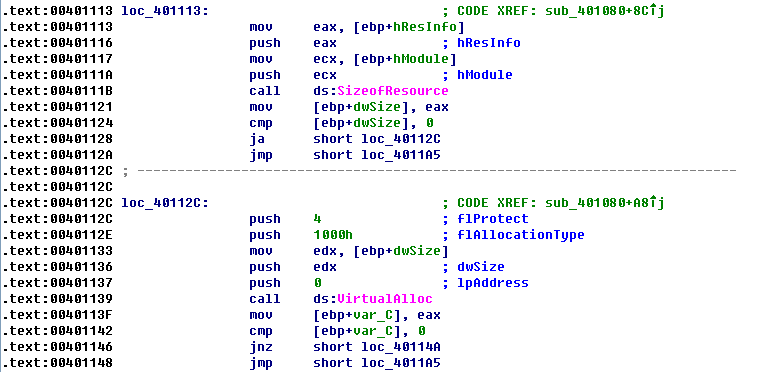
首先判断hModle的值是否为0，即检验上一个函数调用是否成功。





如果成功会调用FindResourceA函数，查看参数的值，说明这个函数是要在资源节中找到类型为BINARY、名字为TGAD的节，并返回对应的句柄。

之后调用函数LoadResource加载这个资源节，调用LockResource把指针指向资源的第一个字节，开始操作其中的内容。



调用SizeofResource获取资源节的大小，调用VirtualAlloc为要导出的DLL分配一个空间。

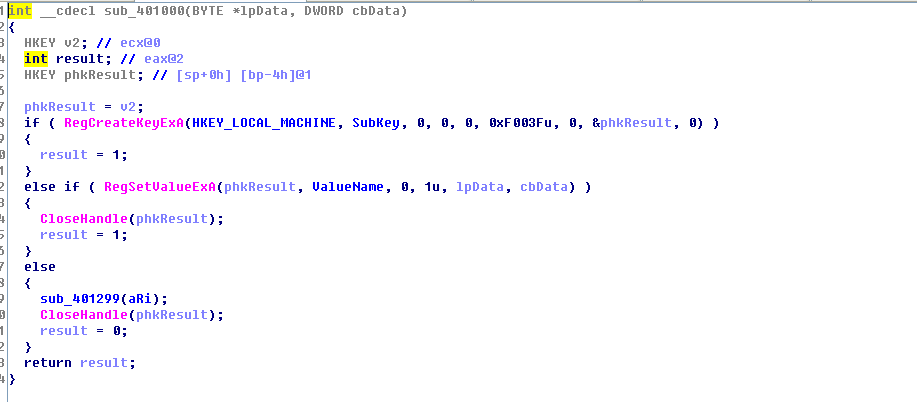


调用了\_fopen以二进制写文件msgina32.dll，\_fwrite函数执行写操作，将资源节的数据写入，最后调 用\_fclose关闭句柄。

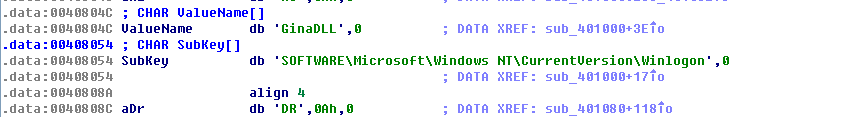
F5查看sub\_401080对应的C代码：



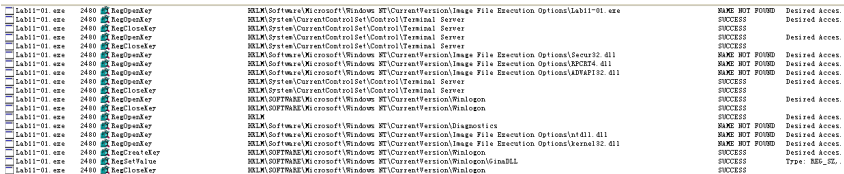
F5查看sub\_401000对应的C代码，发现sub\_401000在修改注册表：



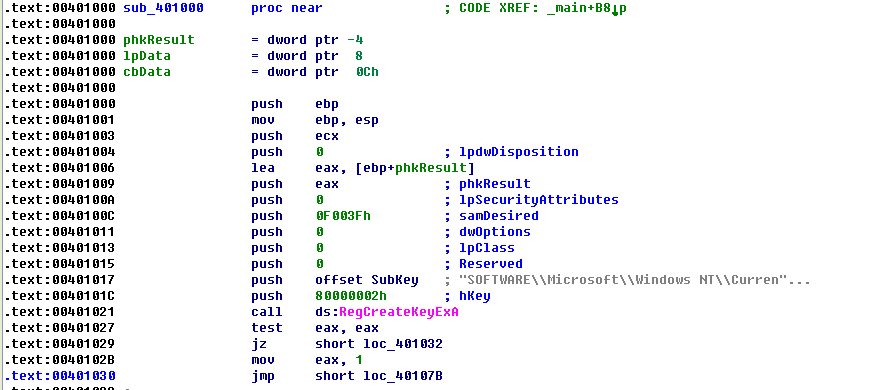
查看subkey，看到winlogon在做持久化，登录的时候就运行。

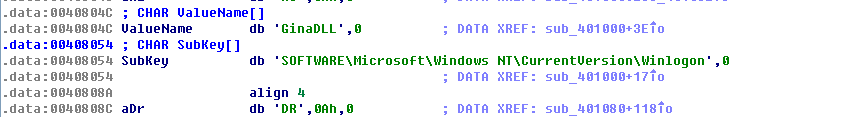


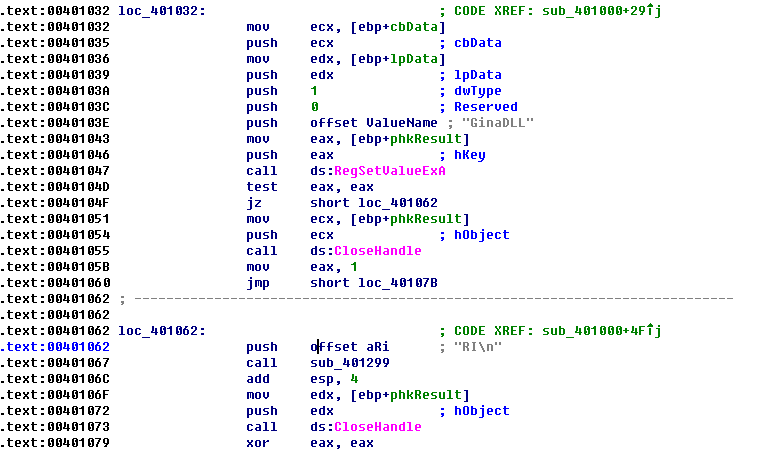
使用Procmon检测，看到多了一项GinaDLL。



分析sub\_401000,发现创建并设置了一个注册表项，设置了GINA注册表的值。





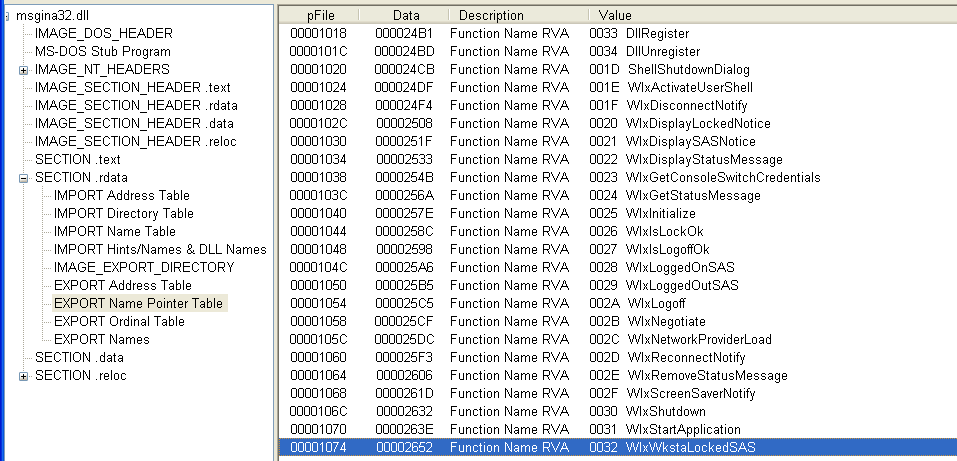


使用Strings分析msgina32.dll的字符串输出，



如果这是一个拦截GINA的恶意代码，则这个列表中的字符串似乎是用来记录用户登录凭证的日志信息。

检查msgina32.dll的导出函数表，看到很多以前缀W1x开始的函数。GINA需要Wlx开头的函数，所以GINA拦截的恶意代码必须包含这些函数。



用IDA Pro分析。



D11Main首先检查参数fdwReason,这个传入的参数表示着DLL入口函数被调用的原因。调用DisableThreadLibraryCalls禁用DLL\_THREAD\_ATTACHE和DLL\_THREAD\_DETACH通知，减少工作集大小。然后调用GetSystemDirectoryW检查系统目录的路径，主要是为了兼容。调用lstracatW拼接字符串，其参数是ecx和\\MSGina，ecx中是GetSystemDirectoryW的返回值，生成了一个路径。

恶意代码检查传入参数是否为DLL\_PROCESS\_ATTACH,在进程启动或者使用LoadLibrary函数加载DLL时，带有这一参数的入口函数将会被调用。如果这个特殊的D11Main在DLL\_PROCESS\_ATTACH期间被调用，那么1000107E开始的代码将被执行。这段代码通过调用0x10001099 LoadLibraryW函数，获取Windows系统目录中msgina.dll的句柄。

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msgina32.dll名字的目的显然是为了欺骗。

恶意代码将这个句柄保存到一个全局变量中，这个变量在IDA Pro中名字为hModule。使用这个变量可以让这个DLL的导入函数恰当地调用Windows DLL msgina.dll中的函数。既然msgina32.dll拦截Winlogon与msgina.dll之间的通信，则它必须恰当地调用msgina.dll中的函数，从而使系统操作继续正常运行。

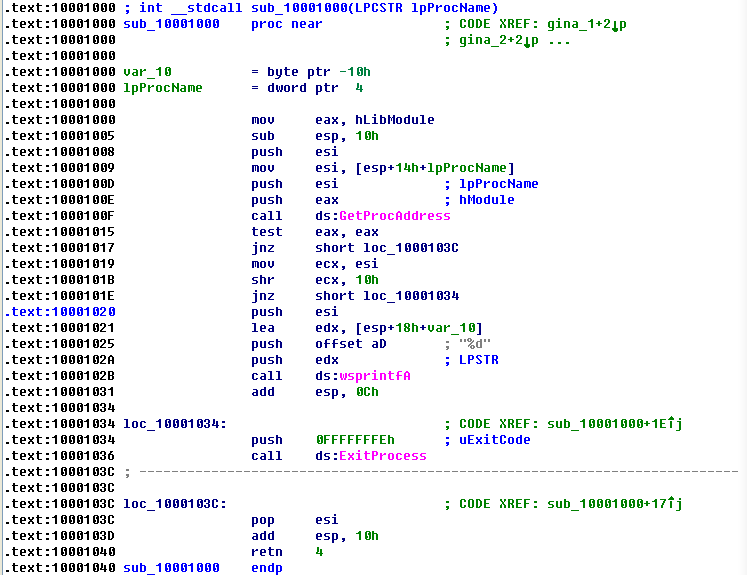
分析 WlxLoggedOnSAS。

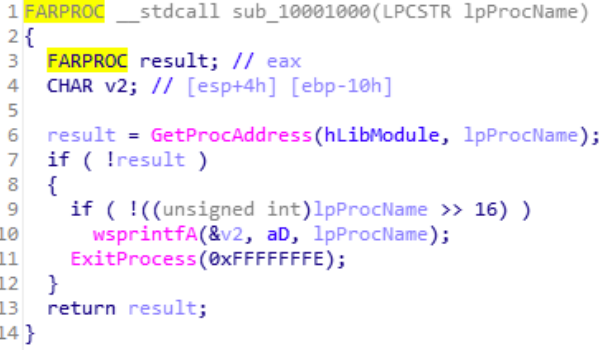
传递字符串WlxLoggedonSAS到sub\_10001000,然后跳转到程序入口地址。函数sub\_10001000传入hModule句柄(msgina.dll的句柄)和字符串(WlxLoggedOnSAS),来使用GetProcAddress函数解析msgina.dll中的函数。恶意代码解析了msgina.dll中sub\_10001000的地址，跳转到该函数。

通过跳转并调用WlxLoggedOnSAS函数，这段代码将不会创建一个栈帧，或者将返回地址压入栈。当msgina.dll中WlxLoggedonSAS被调用时，因为它在栈中的返回地址与调用jmp eax在栈上的返回地址相同，因此它将直接返回到Winlogon上运行。



查看sub\_10001000

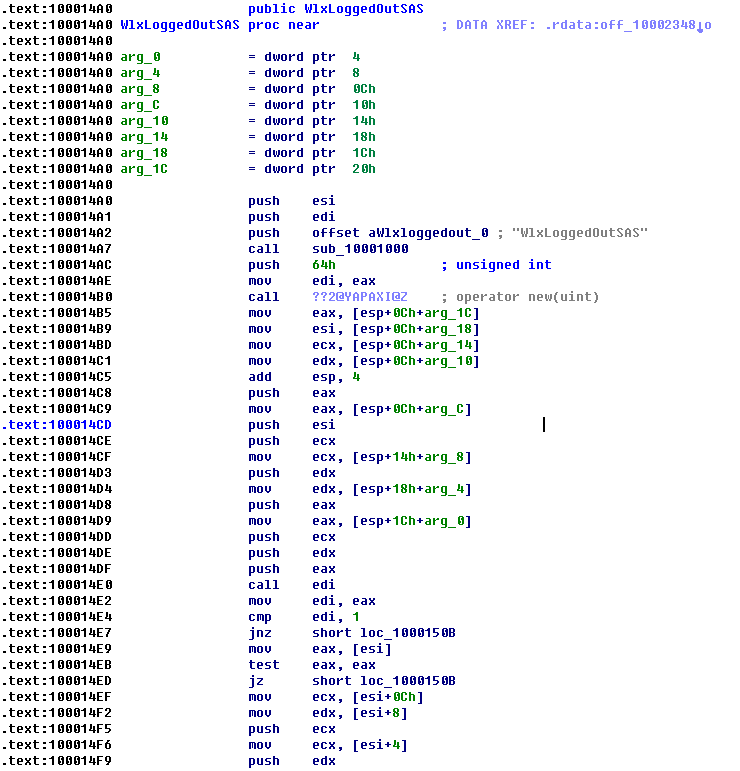


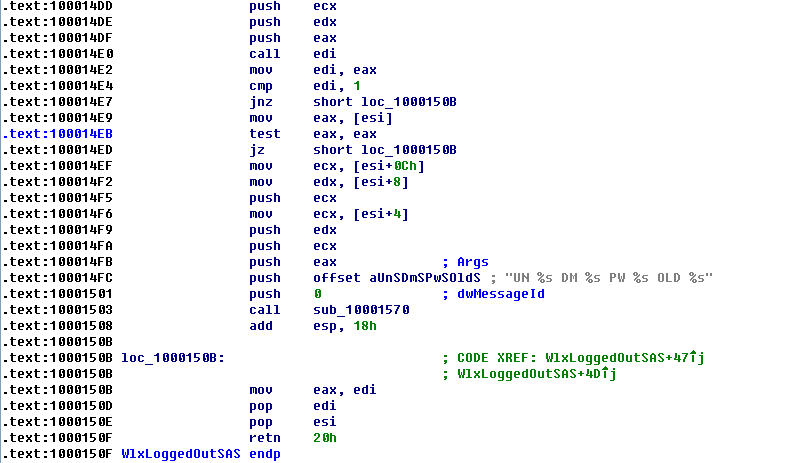


调用完sub\_10001000之后就会跳转到eax，而sub\_10001000在调用完GetProcAddress后并没有再修改eax的值，因此不再分析后面的代码，eax中存储的就是这个导出函数的真实地址，也就是这里只是劫持了该函数，在执行完恶意程序中的代码sub\_10001000之后就又会跳转到真实的WlxLoggedOnSAS函数中去执行。

继续分析其他导出函数，将看到大部分与WlxLoggedOnSAS(它们是中转函数)中的类似操作，然而WlxLoggedOutSAS例外，它包含一些额外的代码(当系统注销时调用WlxLoggedOutSAS)

这个导出首先通过使用GetProcAddress函数，解析出msgina.dll中的WlxLoggedoutSAS函数，然后调用它。





代码传入一堆参数和一个格式字符串aUnSDmSPwSOldS,这个字符串被传入到sub\_10001570

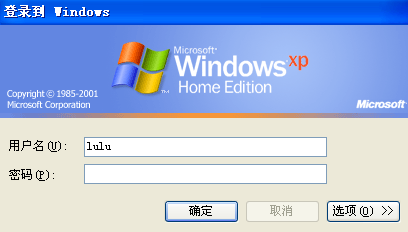
查看sub\_10001570



vsnwprintf调用填充了传入WlxLoggedoutSAS导出函数的格式化字符串。接下来，恶意代码打开msutil32.sys文件，msutil32.sys文件创建在C:\Windows\System32\目录下，call \_wstrtime和call\_wstrtime记录日期和时间。Call fwprintf记录登录凭证信息。msutil32.sys不是一个驱动，它是被用来存储记录的日志文件，虽然它的名字暗示着它是一个驱动。

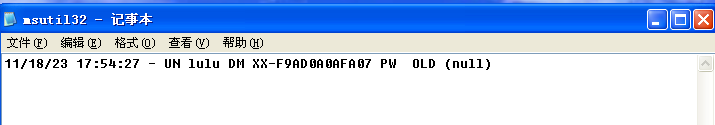
运行Lab11-01.exe,重启机器，然后登录系统并注销系统，可以强制恶意代码记录用户登录凭证。

重启后多了一个登录窗口



在C:\WINDOWS\system32路径下找到一个名为msutil32文件，可以看到时间戳、用户名、密码等信息。





1. **这个恶意代码向磁盘释放了什么?**

恶意代码从名为TGAD资源节中提取出文件msgina32.dll,然后将其释放到硬盘上。

1. **这个恶意代码如何进行驻留?**

为了让msgina32.dll作为GINA DLL安装，恶意代码将自己添加到注册表：HKLM\SOFTWARE\Microsoft\Window NT\CurrentVersion\Winlogon\GinaDLL中。这使得系统重启后，msgina32.dll将被加载。

1. **这个恶意代码如何窃取用户登录凭证?**

恶意代码用GINA拦截窃取用户登录凭证。msgina32.dll能够拦截所有提交到系统认证的用户登录凭证。

1. **这个恶意代码对窃取的证书做了什么处理?**

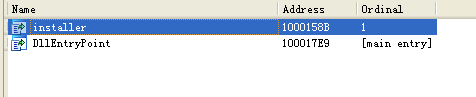
恶意代码将被盗窃的登录凭证记到%SystemRoot%\System32\msutil32.sys中。用户名、域名称、密码、时间戳都将被记录到该文件。

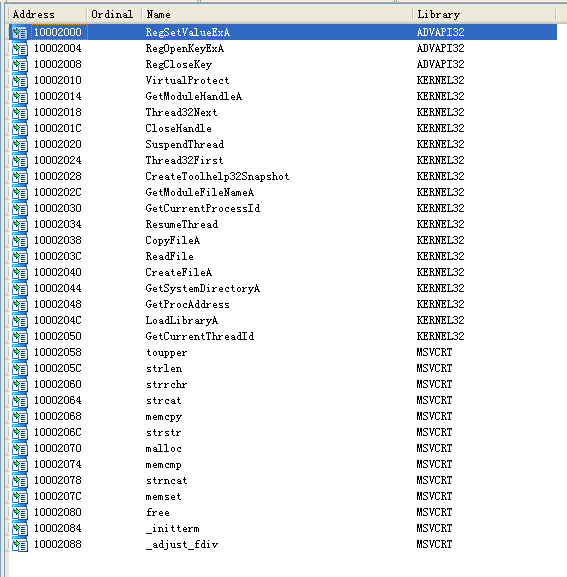
1. **如何在你的测试环境让这个恶意代码获得用户登录凭证?**

释放并且安装恶意代码后，必须重启系统才能启动GINA拦截。仅当用户注销时，恶意代码才记录登录凭证，所以注销然后再登录系统，就能看到记录到日志文件的登录凭证。

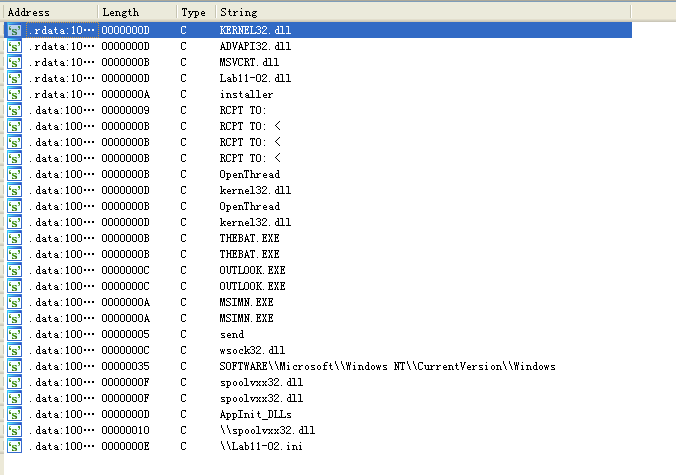
1. **Lab11-2**

Ida打开Lab11-02.dll，查看导入函数和导出函数。这个dll只有一个名为installer的导出函数。这个恶意代码包含操作注册表的导入函数(RegSetValueExA)、改变文件系统的导入函数(CopyFileA),以及搜索一个进程或者线程列表的导入函数(CreateToolhelp32Snapshot)等。



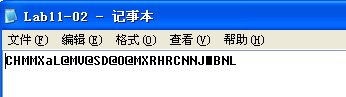


查看字符串列表



字符串AppInit\_DLLs和SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows表明恶意代码使用AppInit\_DLLs来永久安装自身。字符串\Lab11-02.ini表明恶意代码使用本实验提供的INI文件。

检查Lab11-02.ini，发现是无意义的乱码，它似乎包含编码或者加密的数据。

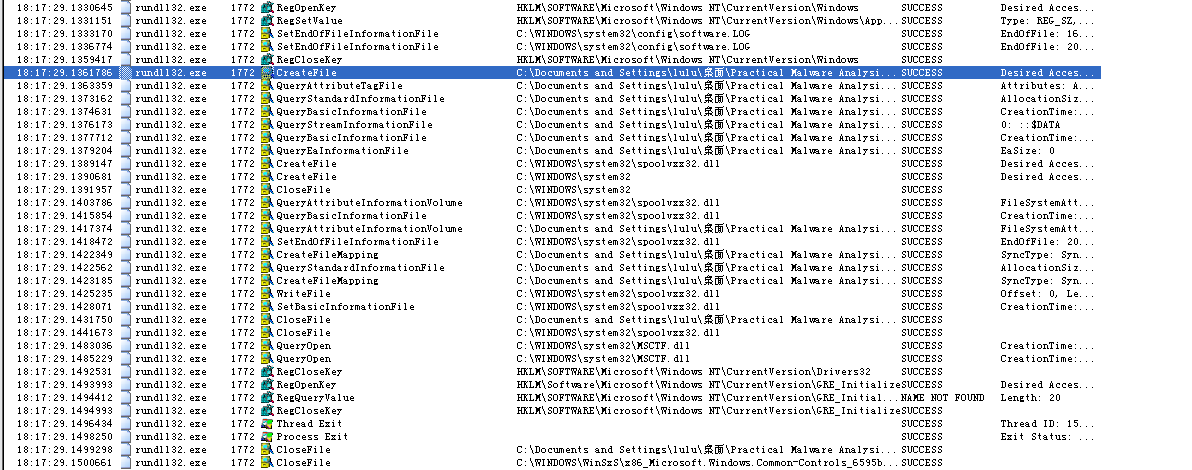


字符串send和wsock32.dll表明恶意代码可能使用了网络功能。进程名(OUTLOOK.EXE、MSIMN,EXE和THEBAT,EXE)是邮件客户端，结合“RCPT TO:”与这些字符串，怀疑这个恶意代码可能对邮件做了某些处理。

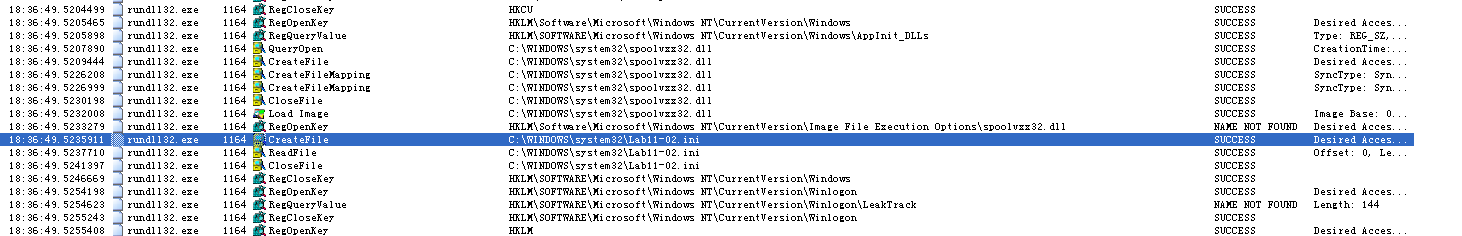
使用process monitor监控恶意程序的运行，并设置过滤器。使用命令rundll32.exe Lab11-02.dll,installer安装恶意代码。

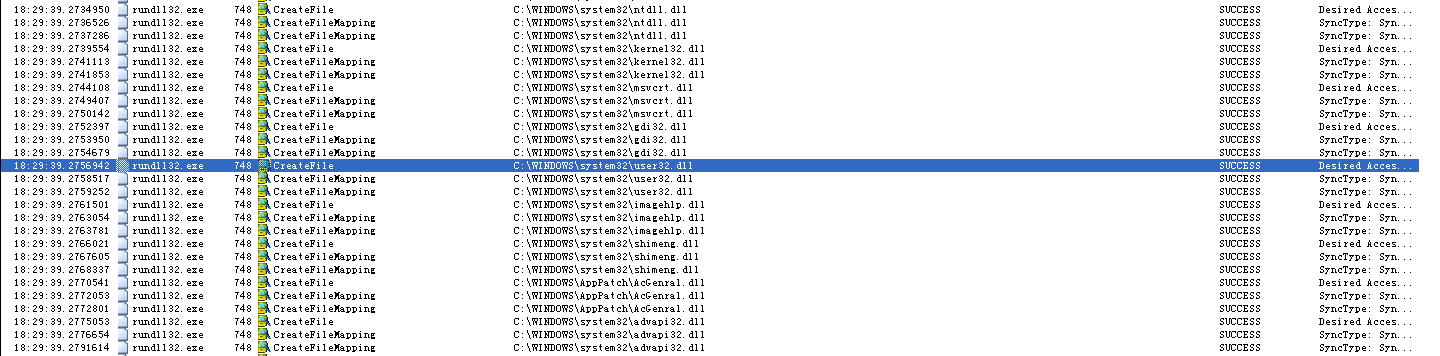
看到恶意代码在Windows系统目录下创建了一个名为spoolvxx32.dll的文件,这个文件与Lab11-02.dll相同。在procmon后面的列表中，看到恶意代码将spoolvcx32.dll添加到AppInit\_DLLs列表中(导致恶意代码被加载到所有装载User32.dll的进程中)。最后，看到恶意代码从Windows系统目录中试图打开Lab11-02.ini文件。

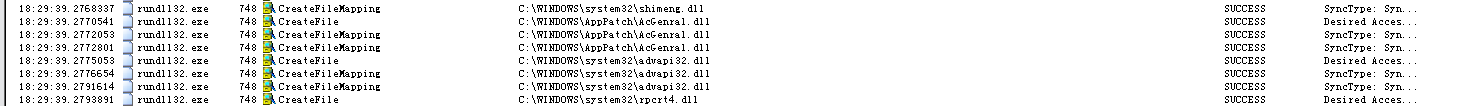
为了恶意代码能够访问这个文件，应该将这个文件复制到Windows系统目录中。



Lab11-02.ini放在Windows系统目录重新运行后，发现这个恶意代码会在最后将自己加载到user32.dll中，使得所有加载了user32.dll的进程也会加载它。这个恶意代码创建了AcGenral.dll，同时这个恶意代码还在系统目录下创建了名为 spoolvxx32.dll 的文件。



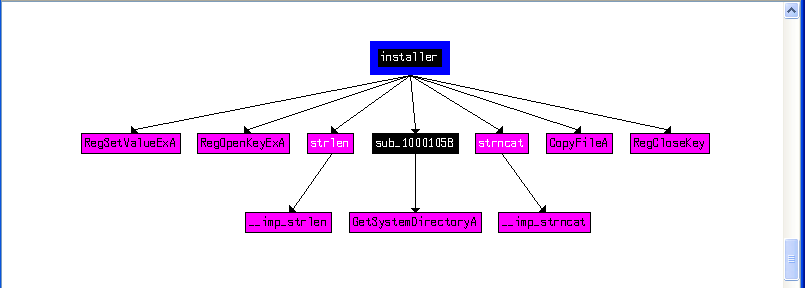




查看 installer

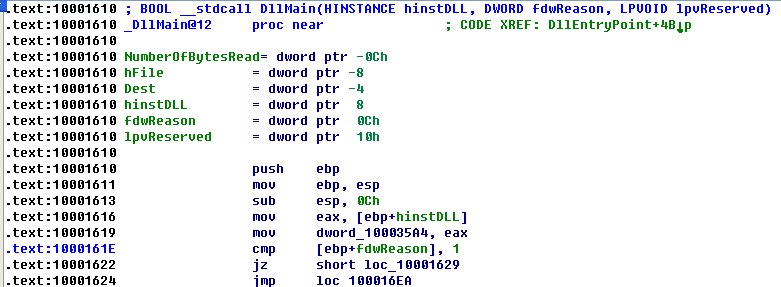


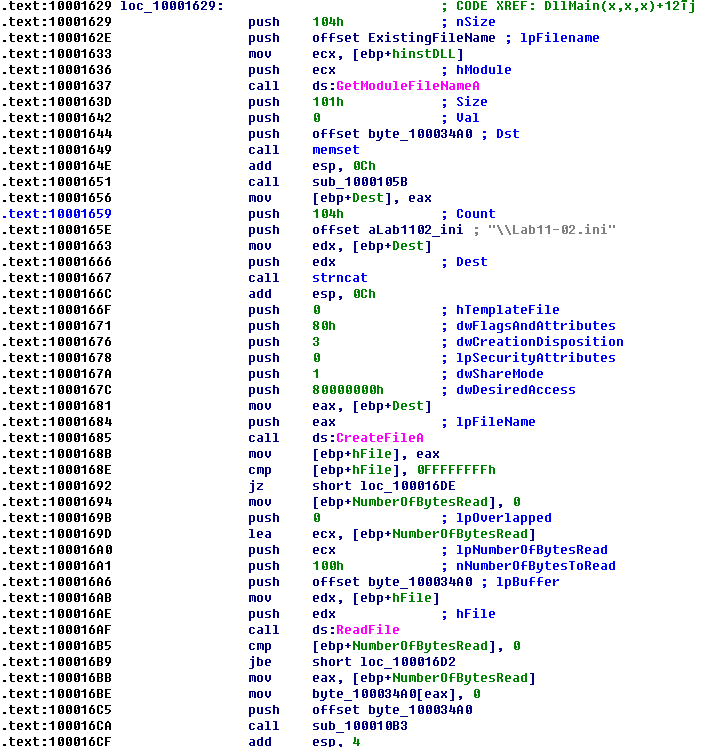
查看installer的交叉引用图



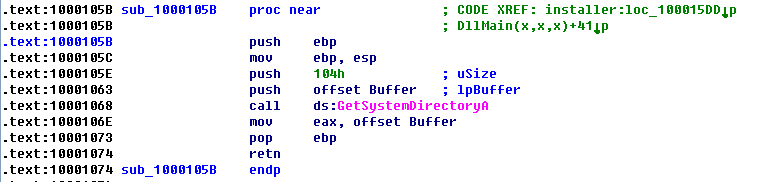
installer在注册表中设置了一个值，并且将一个文件复制到Windows的系统目录。installer函数的唯一目的是复制恶意代码到spoohxx32.dll,并且设置它作为一个APPInit\_DLLs值。

查看DLLMain函数



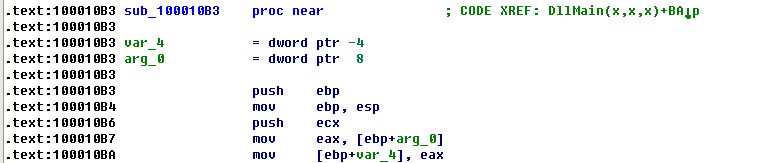


查看sub\_1000105B

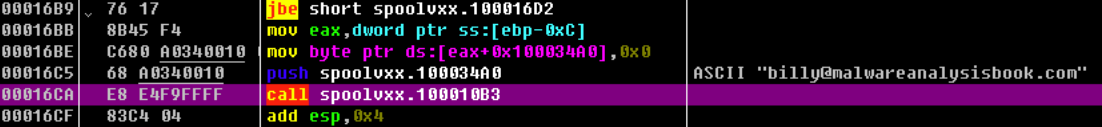


call sub\_1000105B处程序提取Windows系统目录，以及字符串Lab11-02.ini。call strncat处将他们组成一个路径。call ds:CreateFileA打开一个用来读的INI文件。如果这个文件不能被打开，则DLLMain返回。如果恶意代码成功打开了INI文件，它将这个文件读入到一个全局变量缓冲区中。ReadFile调用后，100016B5处的恶意代码检查以确保文件的大小大于0。接下来，100016C5处将包含文件内容的缓冲区传递给sub\_100010B3。

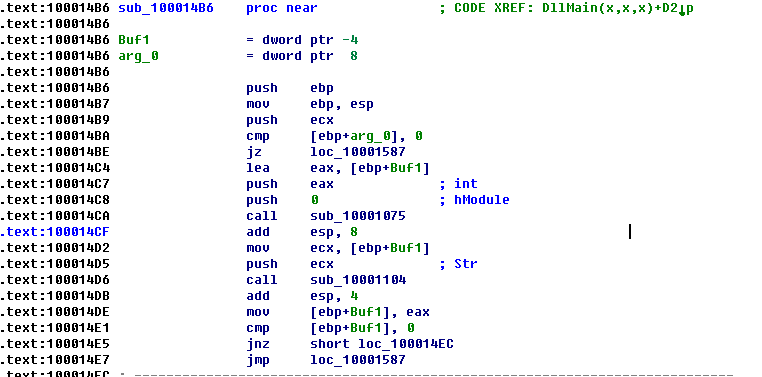
查看sub\_100010B3，应该是一个解密的函数。

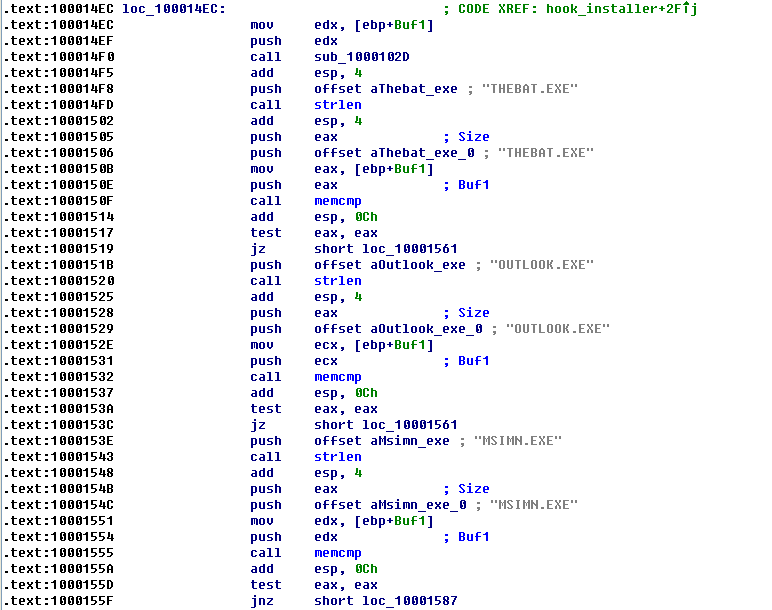


OllyDBG运行后看到解密出的内容是一个邮箱地址billy@malwareanalysisbook.com。寄存器EAX指向它。这个邮箱地址存储在全局变量byte\_100034A0中，在IDA Pro中将它重命名为email\_address,来帮助后续分析。



在D11Main中，还有最后一个需要分析的函数：sub\_100014B6。因为这个函数安装了一个inline挂钩，所以重命名它为hook\_installer。







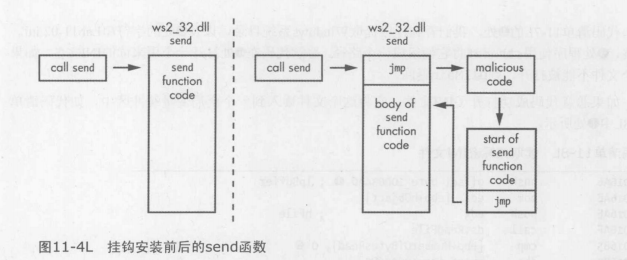
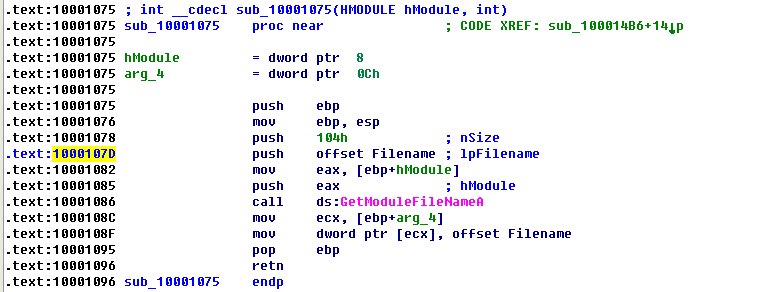
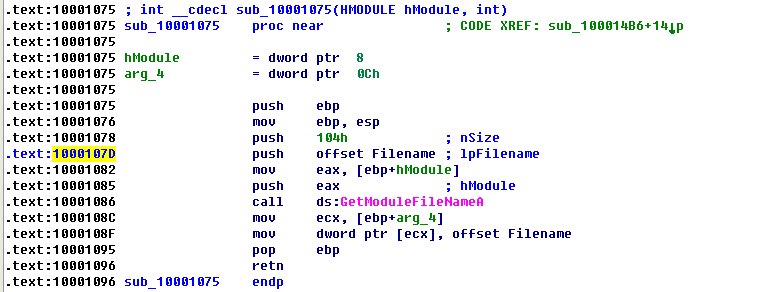


图11-4L的左侧展示了ws2\_32.dll中正常的send函数调用的形式。图的右侧展示了hook\_installer如何安装一个send函数的inline挂钩。send函数的开头指令被一个跳转到恶意代码的指令替换，恶意代码调用了trampoline(如图中右下角的方块所示)。trampline仅仅运行send函数开始的一些代码后(第一个跳转指令覆盖的指令),就跳转到原始的send函数，从而使send函数与没有安装挂钩之前的操作一样。



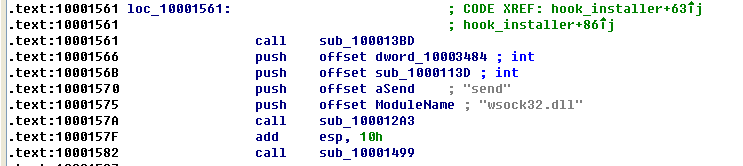
hook\_installer安装挂钩之前，它会检查恶意代码正在哪个进程中运行。为了做到这一点，它调用3个函数来获取当前的进程名。



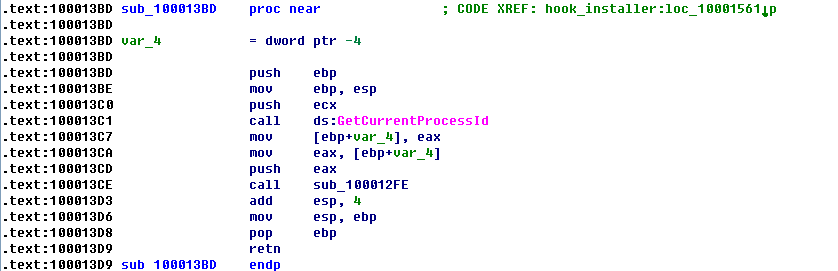
可以看到，恶意代码调用了GetModuleFileNameA函数，因为在调用这个函数之前，它的参数hModule被置为0,所以函数会返回加载这个DLL进程的绝对路径。接下来，恶意代码在arg\_4中返回路径名(字符串指针被传递给函数)。这个字符串被传入至少两个函数，这些函数解析文件名并将文件名中的所有字符转变为大写。

以AppInit\_DLLs作为驻留机制的恶意代码经常使用GetModuleFileNameA。这个恶意DLL几乎被加载到系统中所有启动的进程。因为恶意代码的编写者可能仅针对某些进程，所以他们必须确定运行恶意代码进程的名称。

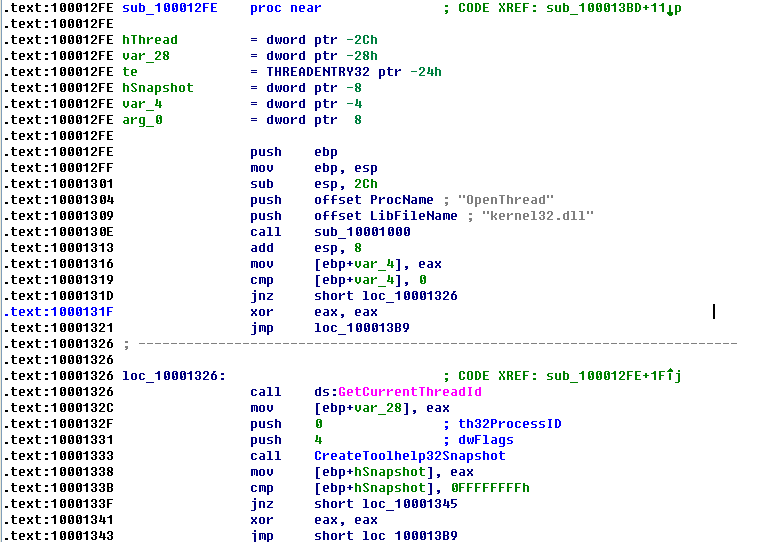
接下来，将当前进程大写字母的进程名与THEBAT.EXE、OUTLOOK.EXE和MSIMN.EXE相比较。如果进程名不等于其中的任何一个，恶意代码就会退出。然而，如果恶意代码已经加载入到这三个进程中的任意一个，下面的恶意代码段就会运行。



查看sub\_100013BD，首先调用了用GetCurrentProcessId，然后是sub\_100012FE。



查看sub\_100012FE，并将其重命名为suspend\_threads

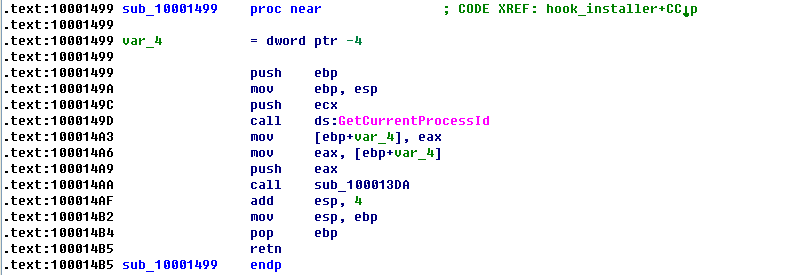




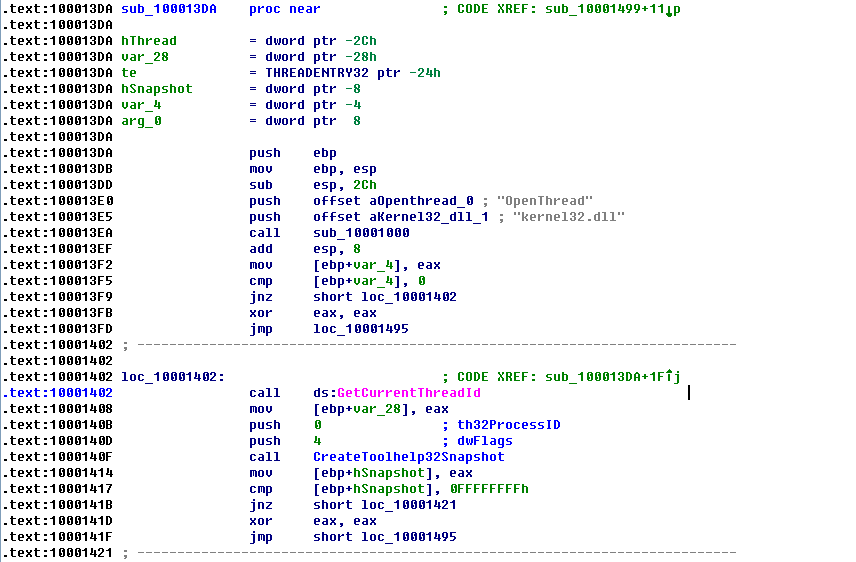


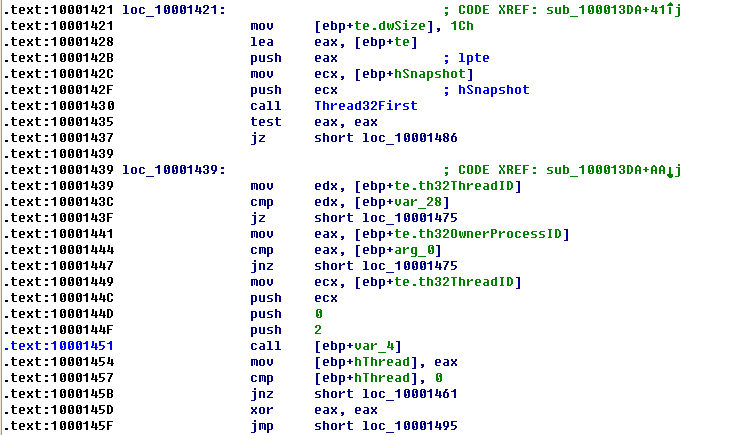
suspend\_threads函数调用了GetCurrentThreadId,它返回当前运行线程的线程标识符(TID)。接下来，suspend\_threads调用CreateToolhelp32Snapshot,并且使用结果循环遍历当前进程的所有TID。如果TID不是当前的线程，则用TID调用SuspendThread。

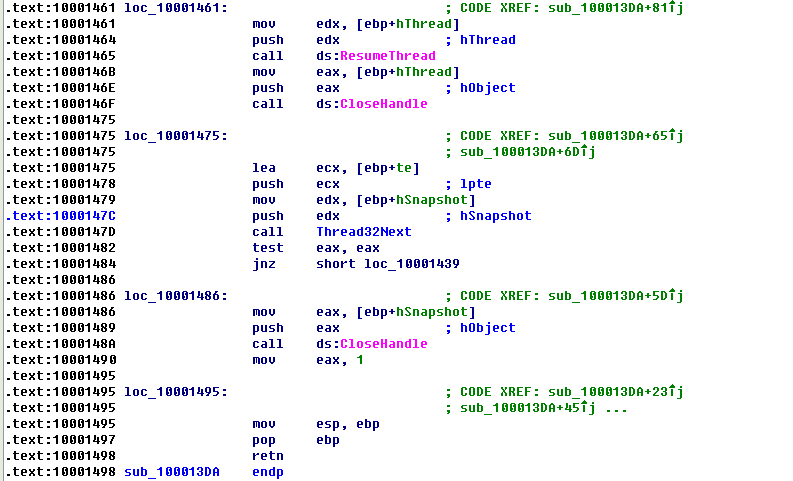
查看sub\_10001499



查看sub\_100013DA，使用ResumeThread恢复所有的线程

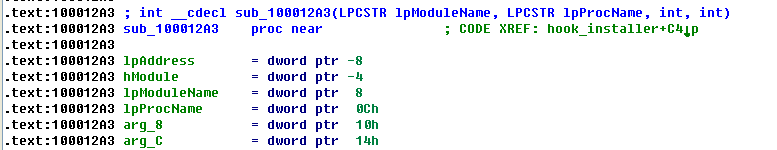




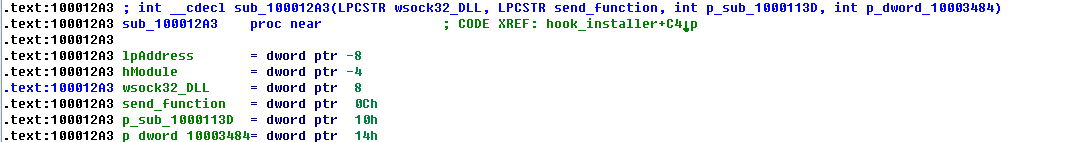


所以分析loc\_10001561处的两个函数的意图，它先被挂起，然后恢复执行。恶意代码执行一些影响改变当前进程运行行为的动作比较常见，例如修改内核或是安装一个inline挂钩。

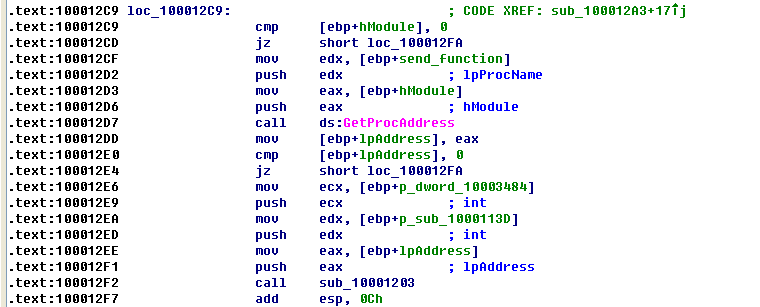
分析sub\_100012A3,函数sub\_100012A3带有4个参数。



重命名它的所有参数，让这些参数与传入函数的参数相匹配，如下图所示。

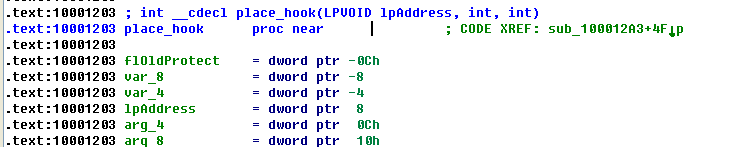


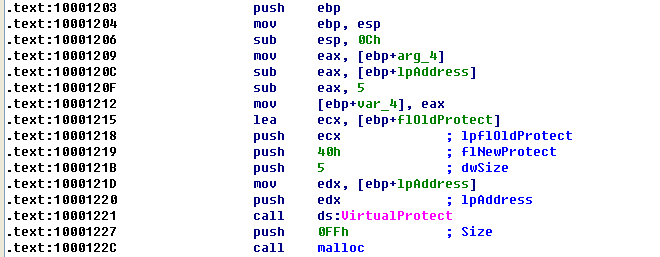




使用GetModuleHandleA函数获得wsock32.dll的句柄，恶意代码最终传递send函数地址和另外两个参数(sub\_1000113D和dword\_10003484)到sub\_10001203函数，将它重命名为place\_hook。

检查place\_hook并且重命名相应的参数





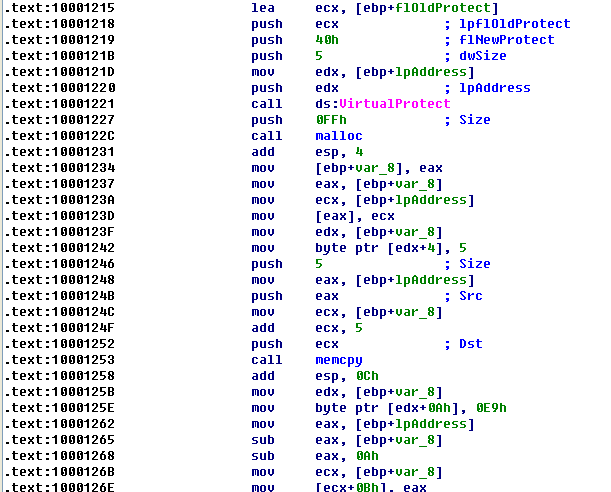
计算send函数的内存地址与sub\_1000113D开始的内存地址之间的差。在将差值移到var\_4变量之前，需要从中减去额外的5个字节。随后的代码使用var\_4变量时，在它的前面加上0xE9(jmp的操作码),使这个5字节的指令能够跳转到sub\_1000113D。

查看恶意代码是如何在后面的place\_hook中安装作为挂钩的这段代码的。send函数开始部分的指令被下面所示的指令修改。



代码将机器码0xE9复制到send函数的开头，之后代码复制var\_4到0xe9之后的内存。var\_4变量包含跳转的目的地址，sub\_1000113D。在send函数的开始部分放置了一个jmp指令，从而让代码跳转到我们DLL中sub\_1000113D处的函数，现在将这个函数重命名为hook\_function。

查看sub\_10001203

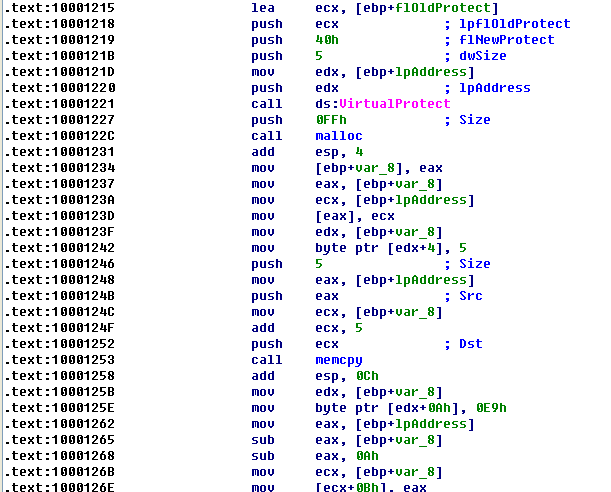


place\_hook在send函数代码的开始调用VirtualProtect函数。这个调用修改了内存的运行、读以及写等保护权限，因此可以使恶意代码修改send函数的执行。然后，在调用VirtualProtect之后，恶意代码立即使用malloc分配0xFF字节的内存，并且将结果保存到var\_8中。因为这个动态分配的内存作为一个trampoline,它在安装我们的挂钩过程中扮演一个重要的角色，所以将var\_8重命名为trampoline。



在函数尾部的VirtualProtect调用，恢复原始的内存保护设置。

查看trampoline代码的创建过程

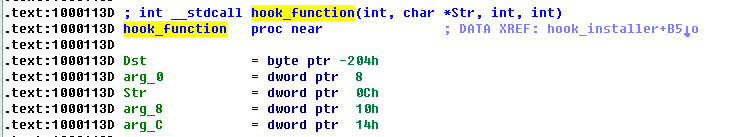


memcpy复制send函数的前5个字节到trampoline,需要确保保存原始指令。恶意代码假设send函数前几条指令严格按照5个字节对齐，但是情况可能并不总是如此。

接下来，恶意代码在1000125E和1000126E处添加一个jmp指令到trampoline代码。在1000125E处，添加机器码0xE9,在1000126E处，添加跳转的地址。通过从send函数的地址中减去trampoline的地址(意味着它将跳回到send函数),来计算跳转地址。

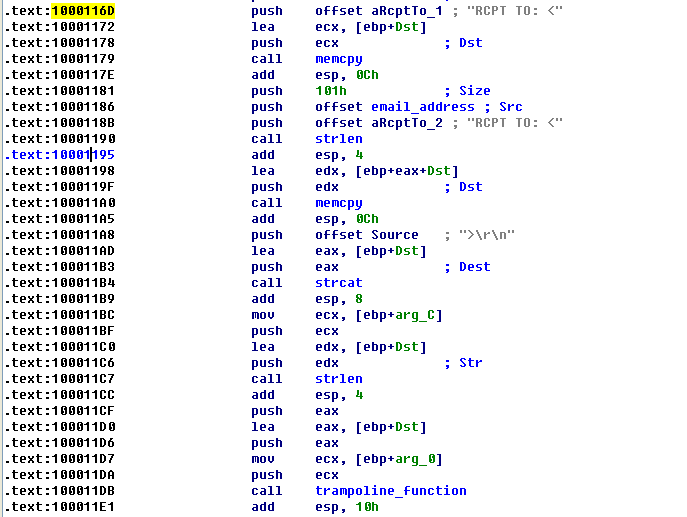
最后，place\_hook在末尾，将全局变量dword\_10003484变量设置为trampoline的地址。重命名dword\_10003484为trampoline\_function,来帮助分析。

分析hook\_function(sub\_1000113D),因为它是安装的挂钩函数，所以它与send函数拥有相同的参数。



选择Set Function Type并且输入int \_stdcall hook\_function(SOCXET s,char \* buf, int len, int flags)。

挂钩函数在buf中查找字符串RCPT TO:。如果没有发现这个字符串，恶意代码则只调用trampoline\_function,这使得send函数操作与没有安装挂钩前一样。否则，下面的代码将运行。



创建一个添加到向外传输的缓冲区中的字符串。这个字符串以RCPT TO:<开头,随后是email\_address,最后以>\r\n结束。本例中email\_address的值为billy@mahwareanalysisbookcom(从Lab11-02.ini提取，与先前看到的这个文件内容一致)。这段代码向所有的发出邮件中添加了一个收件人。

挂钩函数操作如下：

①程序调用send函数。

②send函数的第一个指令转换到sub\_1000113D运行。

③当且仅当外传缓冲区包含一个RCPT TO时，sub\_1000113D操纵外传的缓冲区。

④sub\_1000113D调用堆中由dword\_10003484指向的trampoline代码。

⑤trampoline代码运行send函数的前3条原始指令(它被覆盖来安装挂钩)。

⑥trampoline代码跳回send函数的第5个字节，从而使send函数可以正常运行。

在安装恶意代码并启动Outlook Express后(Outlook与微软Windows XP绑定并且以msimn.exe运行),使用OllyDbg检查inline挂钩。

使用File→Attach,并从进程列表中选择msimn.exe,来附加到这个进程。附加进程后会立即中止它的所有线程。如果检查内存映射，会看到spoolvxx32.dll被加载到进程中，因为它是一个AppInit\_DLLs值。

接下来，通过按Ctrl+G组合键，在文本框中输入send,来检查send函数。

抓取网络流量

为了抓住这个活动的恶意代码和查看它如何操纵网络流量，按如下设置一个安全环境：

① 开启虚拟机中host-only网络。

② 用命令rund1132.exe Lab11-02.exe,installer在你的虚拟机中安装恶意代码。

③ 复制Lab11-02到目录C:\Windows\System32\中。

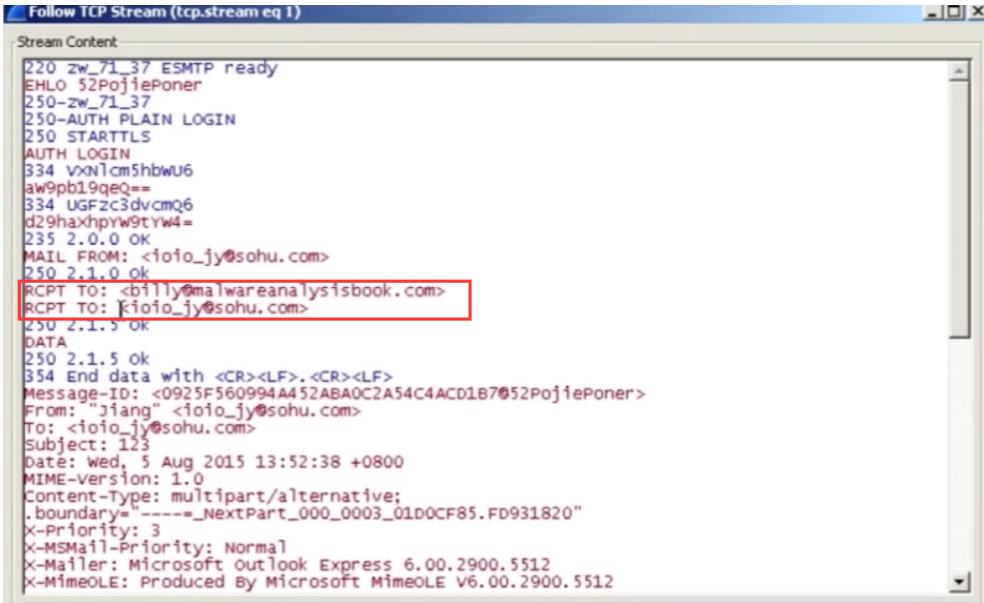
④ 启动Wireshark,并且在虚拟机网络接口上启动抓包。

⑤ 设置Outlook Express发送邮件到主机系统中。

⑥ 使用命令python -m smtpd -n -c DebuggingServer IP:25在你的宿主系统中运行一个假冒的邮件服务器，其中IP是宿主系统的IP地址。

⑦ 从Outlook Express中发送一封邮件。

⑧ 检查在Wireshark中抓获的包并且在邮件信息上选择Follow TCP Stream。



捕获到了一个SMTP的数据包，在打开以后发现，这里不仅发送了我们输入的邮箱地址，还发送到了之前解析出来的恶意邮箱。

Lab11-2是一个导出installer函数的恶意DLL,它使用AppInit\_DLLs键值来永久安装恶意代码，这导致大多数进程都会加载这个恶意代码。恶意代码通过一个预设的目标进程列表，来查看它是否运行在一个邮件客户端程序中。如果恶意代码确定它是运行在这些预设进程的其中之一时，它通过在send函数安装一个inline挂钩，来扮演用户态Rootkit。这个挂钩的实现方式，是在send函数的开始放置一个jmp指令。挂钩运行一个函数，它扫描由send函数发送的每个数据缓冲区，来搜索字符串RCPT TO。如果恶意代码发现了RCPT TO字符串，它会插入一个额外的RCPT TO,它包含从解密的Lab11-02.ini中提取的邮件地址，其目的是从目标邮件程序中复制所有邮件发送给恶意代码编写者。

1. **这个恶意DLL导出了什么?**

Lab11-02.dll包含一个名为installer的导出函数。

1. **使用rundll32.exe安装这个恶意代码后，发生了什么?**

如果使用命令rundll32.exe Lab11-02.dll,installer从命令行启动恶意代码，恶意代码会作为spoolvxx32.dll将自身复制到系统目录中，并且在AppInit\_DLLs键值下永久安装。另外，恶意代码尝试着从系统目录中打开Lab11-02.ini,但是它在那里并没有发现这个文件。

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

为了恶意代码的正常运行，Lab11-02.ini必须位于%SystemRoot%\System32\目录下。

1. **这个安装的恶意代码如何驻留?**

恶意代码将自身安装到AppInit\_DLLs的注册表键值中，这可以使恶意代码加载到所有装载User32.dll的进程中。

1. **这个恶意代码采用的用户态Rootkit技术是什么?**

这个恶意代码针对send函数安装了一个inline挂钩(hook)。

1. **挂钩代码做了什么?**

这个挂钩检查向外发出的包，看外传的包是否是包含RCPT TO:的电子邮件信息，如果发现了这个字符串，则它会添加一个额外的RCPT TO行，来增加一个恶意的电子邮件账户。

1. **哪个或者哪些进程执行这个恶意攻击，为什么?**

恶意代码的攻击目标仅针对MSIMN.exe、THEBAT.exe和OUTLOOK.exe,之所以这样，是因为它们都是电子邮件客户端软件。除非恶意代码运行在这些进程空间中，否则它不会安装挂钩。

1. **.ini文件的意义是什么?**

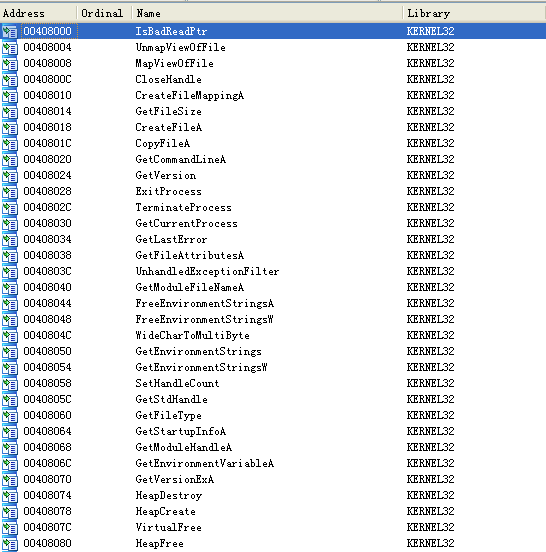
INI文件中包含一个加密的邮件地址，解密Lab11-02.ini之后，看到它包含billy@malwareanalysisbook.com。

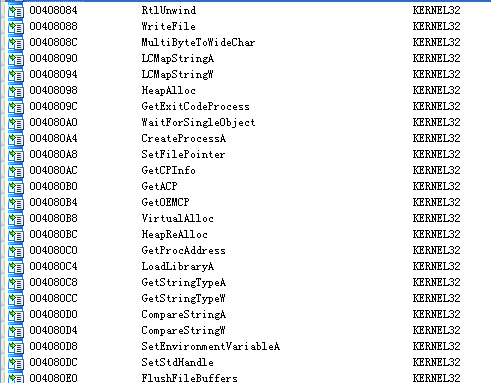
1. **你怎样用Wireshark动态抓获这个恶意代码的行为?**

通过“抓取网络流量(Capturing the Network Traffic)”的方法，通过Wireshark抓取的网络数据，可以看到一个假冒的邮件服务器以及Outlook Express客户端。

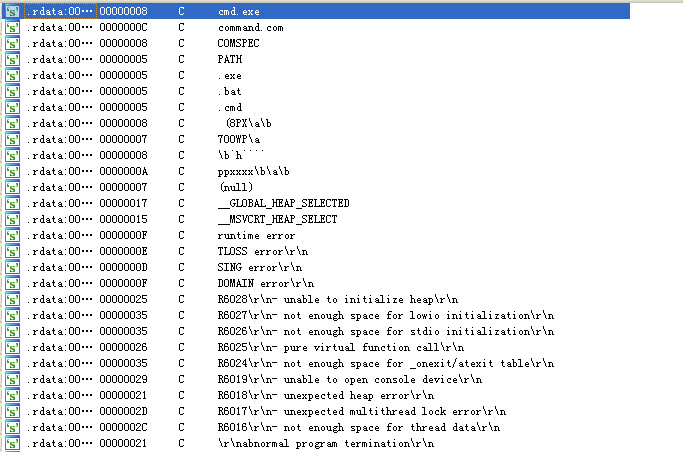
1. **Lab11-3**

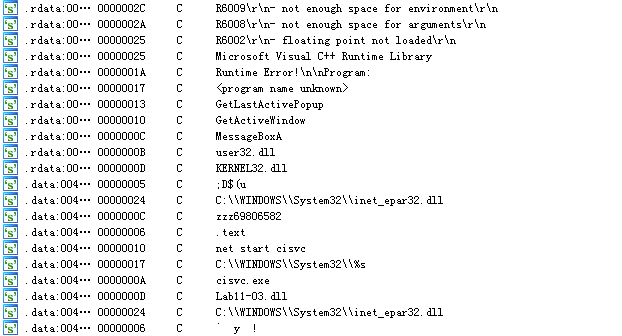
查看Lab11-03.exe的导入函数，





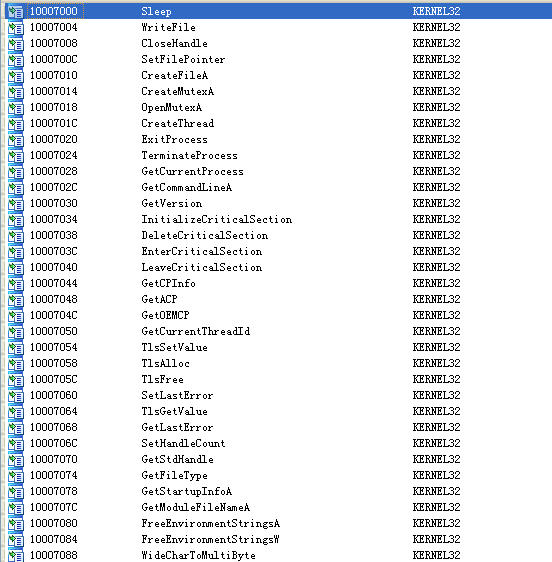
查看Lab11-03.exe的字符串列表

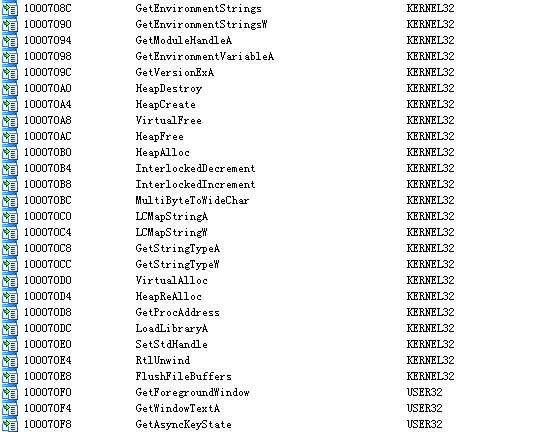




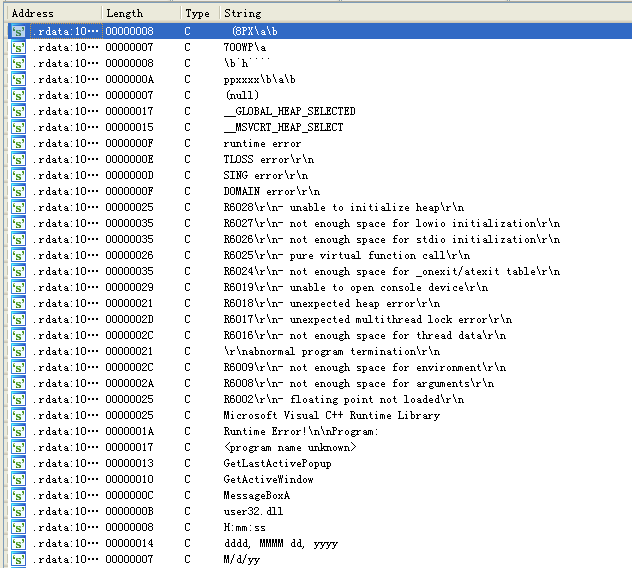
Lab11-03.exe包含字符串inet\_epar32.d11和net start cisvc。net start命令用来在Windows机器中启动一个服务，但是我们还不知道恶意代码为什么在该系统上启动索引服务。

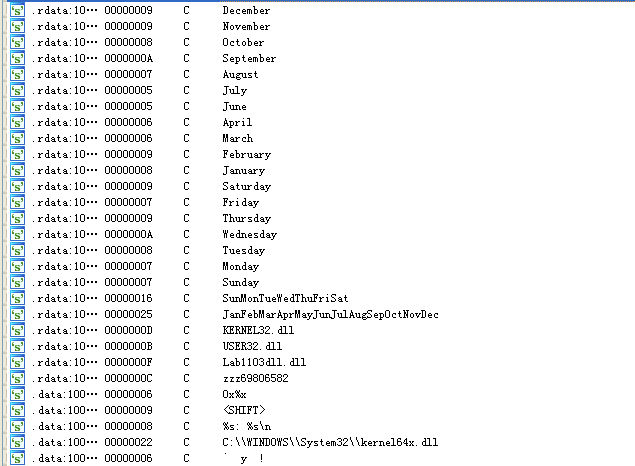
查看Lab11-03.dll的导入表





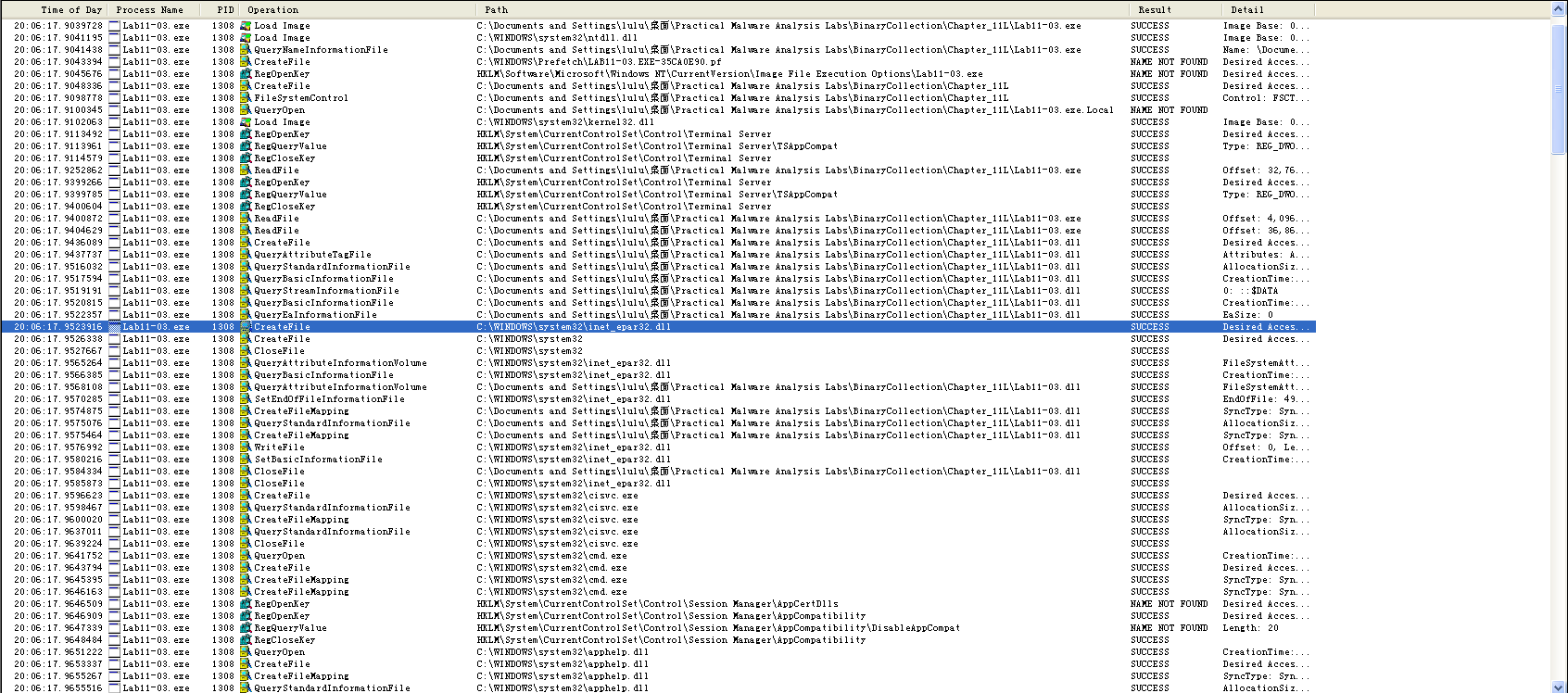
查看Lab11-03.dll的字符串列表



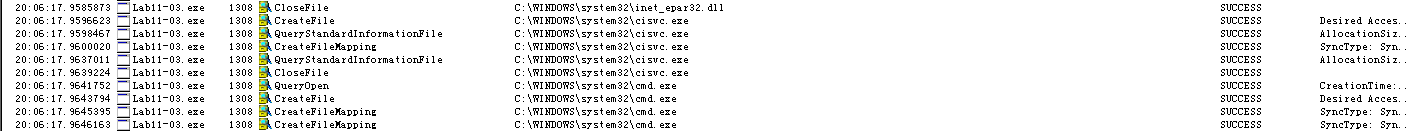


Lab11-03.dll包含字符串C:\WINDOWS\System32\kerne164x.d11,它导入API函数GetAsyncKeyState和GetForegroundWindow，怀疑这是一个击键记录器，它将击键记录到kernel64x.dll。同时，Lab11-03.dll也包含有奇怪命名的导出函数zzz69806582。

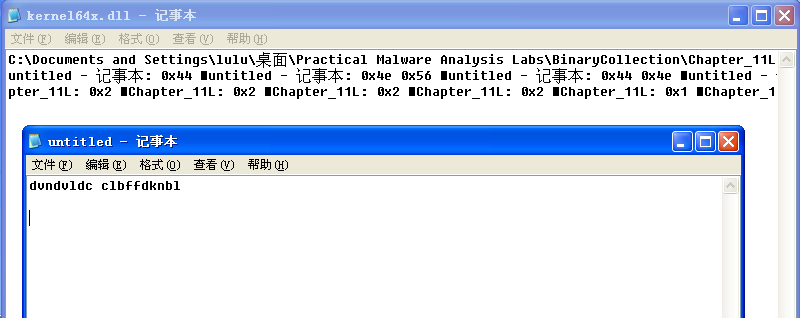
启动procmon,并且过滤Lab11-03.exe,看到恶意代码创建了C:\Windows\System32\inet\_epar32.dll。DLL inet\_epar32.dll与Lab11-03.dll相同，说明恶意代码复制Lab11-03.dll到Windows系统目录。



恶意代码打开了cisvc.exe的句柄，但是没有看到任何WriteFile操作。最后，恶意代码通过发出命令net start cisvc来启动索引服务。通过使用Process Explorer,观察到cisvc.exe正在系统中运行。



怀疑恶意代码可能记录击键事件，打开notepad.exe(记事本)并且输入一串字符。看到kernel64x.dll被创建。怀疑击键事件被记录，在十六进制编辑器中打开kernel64x.dll,看到了如下输出：



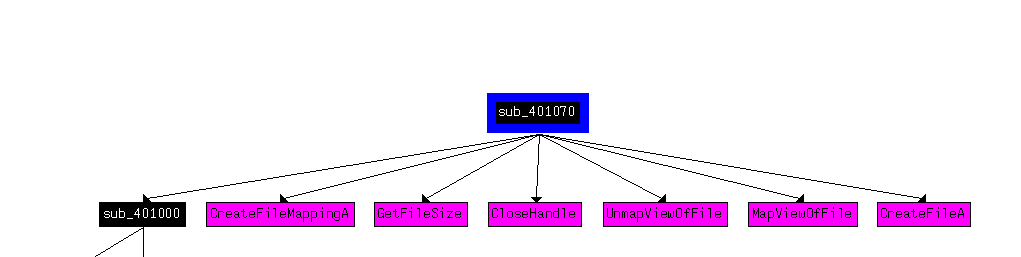
击键已经被记录到kernel64x.dll,可以看到输入击键的程序(Notepad)与十六进制的击键数据一起都被记录。

查看Lab11-03.exe的main函数



main函数刚开始就复制Lab11-03.dll到C:\Windows\System32系统目录中inet\_epar32.dll。接下来，它创建一个字符串C:\WINDOWS\System32\cisvc.exe,并且将这个字符串传递给sub\_401070。最后，恶意代码通过使用系统运行命令net start cisvc,来启动索引服务。

查看sub\_401070的交叉索引图

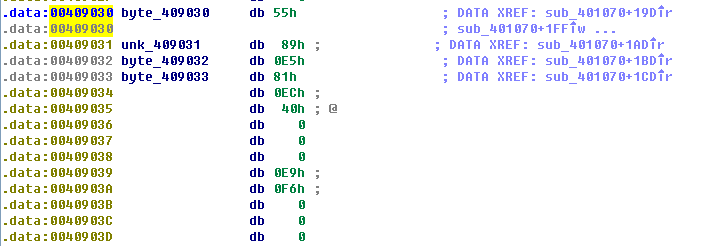


为了操纵cisvc.exe,sub\_401070函数调用CreateFileA、CreateFileMappingA和MapViewOfFile,将cisvc.exe映射到内存中。这些函数都有访问这个文件的读写权限。MapViewOfFile返回的内存映射视图的基地址(在IDA Pro中标注为1pBaseAddress)可以被读写。在UnmapViewOfFile调用之后，对这个文件做的任何修改都会被写入到硬盘，这解释了为什么没有在procmon的输出中看到WirteFile函数。

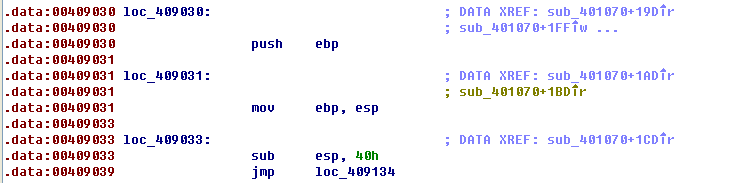


文件的映射位置被移入到EDI,并且用var\_28来调整一些偏移量。接下来，将0x4E载入ECX,写入这个数量的DWORD(movsd)。因此总字节数为0x4E\*4=312字节(十进制)。最后，byte\_409030被移入ESI，并且rep movsd复制byte\_409030的数据到映射文件中。

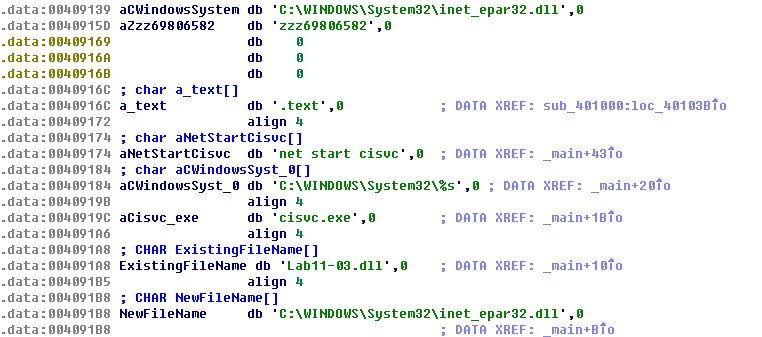
检查byte\_409030处的数据



C显示反汇编结果，shellcode是人工构造的汇编代码，本例中，它被用来进行进程注入。



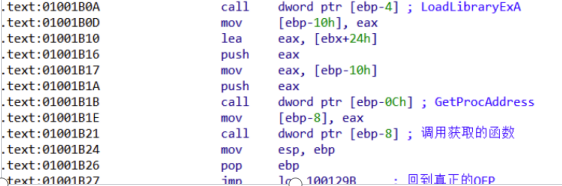
从开头到312个字节的shellcode的末尾，看到两个字符串：



inet\_epar32.dll路径和导出函数的出现，说明这个shellcode加载了这个DLL,并且调用了它的导出函数。

比较恶意代码运行前后cisvc.exe二进制文件的版本差异。看到两个不同：在PE头部有312字节shellcode的插入，另一个则是PE头部两个字节的改变。将这两个二进制文件载入到PEview中，观察PE头部的不同。感染前二进制文件入口点为0000129B，感染后入口点为00001A28。

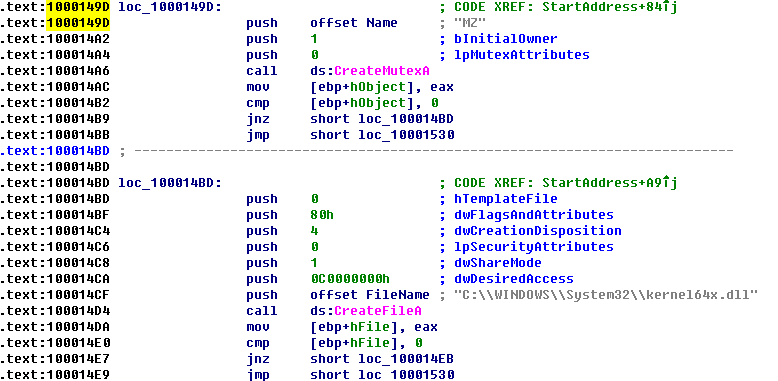
使用ida打开感染后的cisvc.exe。在0x1001B0A处恶意代码调用LoadLibrary函数，将inet\_epar载入内存。之后恶意代码在1001B1B用参数zzz69806582调用GetProcAddress,来获取导出函数的地址。在1001B21处调用zzz69806582函数。最后恶意代码跳转到原始的入口点,从而使服务正常执行。shellcode功能与我们先前怀疑加载inet\_epar32.dll,并且调用它导出的功能相吻合。



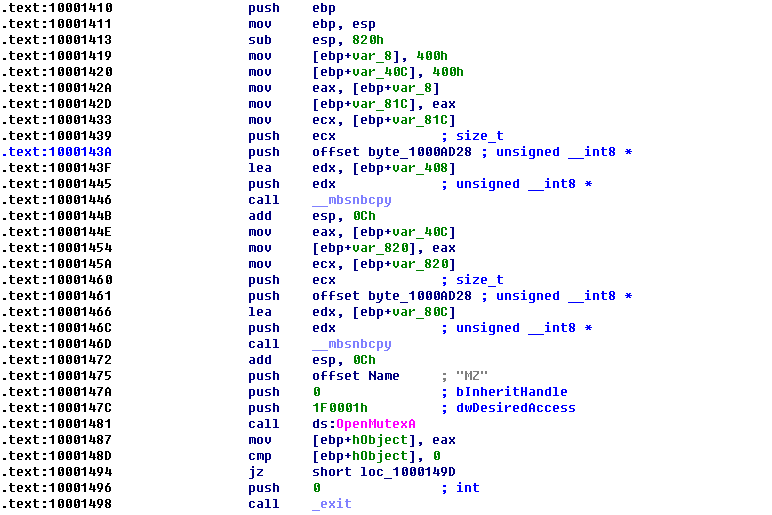
恶意代码通过入口点重定向进行特洛伊木马化索引服务，从而使它永久性地安装Lab11-03.dll。它通过重定向入口点，来运行加载这个DLL的shellcode。

分析inet epar32.dll,它与Lab11-03.dI相同。

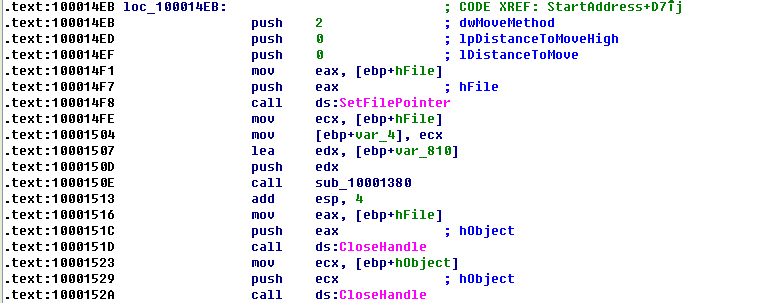
Ida分析Lab11-03.dll,大部分代码来自于导出函数zzz69806582。这个导出函数启动一个线程并且返回，分析这个线程。

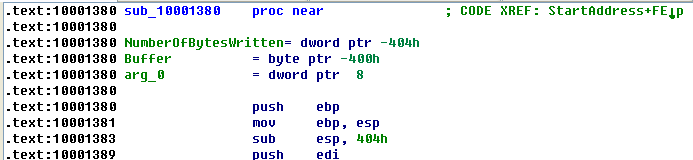


恶意代码创建一个名为MZ的互斥量，这个互斥量阻止恶意代码多个实例的运行，如果互斥量MZ已经存在，前面OpenMutex会终止这个线程。



接下来，恶意代码打开或者创建一个名为kernel64x.dll的文件，来写入日志。获得kernel64x.dll的句柄之后，恶意代码将文件指针指向文件的末尾，并且调用包含一个循环的sub\_10001380函数。这个循环包含对GetAsyncKeyState、GetForegroundWindow和WriteFile的调用。这与“用户态击键记录器(User-Space Keyloggers)”中击键记录方法一致。







Lab11-03.exe以木马方式侵入系统，然后启动Windows索引服务(cisvc.exe)。木马程序的shellcode加载一个DLL,并且调用启动击键记录器的导出函数。这个导出函数创建一个MZ互斥量，并且将所有的击键记录到Windows系统目录中的kernel64x.dll日志文件中。

1. **使用基础的静态分析过程，你可以发现什么有趣的线索?**

Lab11-03.exe包含字符串inet\_epar32.dll和net start cisvc,这意味着它可能启动cisvc索引服务。包含字符串C:\WIND0WS\System32\kernel64x.d11的Lab11-03.dll导入了API函

数GetAsyncKeyState和GetForegroundwindow,怀疑它是一个记录到文件kernel64x.dll的击键记录器。

1. **当运行这个恶意代码时，发生了什么?**

恶意代码首先复制Lab11-03.dll到Windows系统目录的inet\_epar32.dll中，它向ciswc.exe写入数据并且启动索引服务。恶意代码也似乎向C:\Windows\System32\kernel64x.dll写入击键记录。

1. **Lab11-03.exe如何安装Lab11-03.dll使其长期驻留?**

恶意代码通过入口点重定向进行特洛伊木马化索引服务，从而使它永久性地安装Lab11-03.dll。它通过重定向入口点，来运行加载这个DLL的shellcode。

1. **这个恶意代码感染Windows系统的哪个文件?**

加载inet\_epar32.dll,恶意代码感染了cisvc.exe,然后调用了inet\_epar32.dll的导出函数zzz69806582。

1. **Lab11-03.dll做了什么?**

Lab11-03.dll是一个轮询的密钥记录器，这在它的导出函数zzz69806582中得到实现。

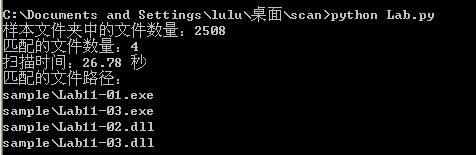
1. **这个恶意代码将收集的数据存放在何处?**

恶意代码存储击键记录和窗体输入记录，其中击键记录被存入到C:\Windows\System32\kernel64x.dll。

1. **Yara 规则**

根据特征字符串，编写yara规则如下

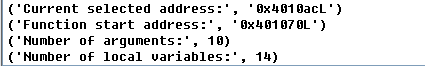
|  |
| --- |
| rule RuleforLab11\_01 {  meta:  description = "Lab11-01.exe"  strings:  $s1 = "gina.dll" fullword ascii  $s2 = "msgina32.dll" fullword ascii  $s3 = "MSGina.dll" fullword wide  $s4 = "msutil32.sys" fullword wide  $s5 = "\\msgina32.dll" fullword ascii  condition:  uint16(0) == 0x5a4d and  uint32(uint32(0x3c))==0x00004550 and filesize < 200KB and  3 of them  } |
| rule RuleforLab11\_02dll {  meta:  description = "Lab11-02.dll"  strings:  $s1 = "spoolvxx32.dll" fullword ascii  $s2 = "THEBAT.EXE" fullword ascii  $s3 = "\\spoolvxx32.dll" fullword ascii  $s4 = "Lab11-02.dll" fullword ascii  $s5 = "\\Lab11-02.ini" fullword ascii  $s6 = "AppInit\_DLLs" fullword ascii  $s7 = "RCPT TO: <" fullword ascii  $s8 = "MSIMN.EXE" fullword ascii  condition:  uint16(0) == 0x5a4d and  uint32(uint32(0x3c))==0x00004550 and filesize < 60KB and  6 of them  } |
| rule RuleforLab11\_02ini {  meta:  description = "Lab11-02.ini"  strings:  $s1 = "CHMMXaL@MV@SD@O@MXRHRCNNJ" fullword ascii  condition:  uint16(0) == 0x5a4d and  uint32(uint32(0x3c))==0x00004550 and filesize < 1KB and  all of them  } |
| rule RuleforLab11\_03dll {  meta:  description = "Lab11-03.dll"  strings:  $x1 = "C:\\WINDOWS\\System32\\kernel64x.dll" fullword ascii  $s2 = "Lab1103dll.dll" fullword ascii  $s3 = "VWuBh@u" fullword ascii  $s4 = "5@8D8H8L8" fullword ascii  $s5 = "2?3E3S3" fullword ascii  $s6 = "2T3[3j3" fullword ascii  $s7 = "<SHIFT> " fullword ascii  $s8 = "20272?2D2H2L2u2" fullword ascii  $s9 = "zzz69806582" fullword ascii  condition:  uint16(0) == 0x5a4d and  uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and  1 of ($x\*) and 4 of them  } |
| rule RuleforLab11\_03exe {  meta:  description = "Lab11-03.exe"  strings:  $x1 = "C:\\WINDOWS\\System32\\inet\_epar32.dll" fullword ascii  $s2 = "cisvc.exe" fullword ascii  $s3 = "C:\\WINDOWS\\System32\\%s" fullword ascii  $s4 = "Lab11-03.dll" fullword ascii  $s5 = "net start cisvc" fullword ascii  $s6 = "COMSPEC" fullword ascii  $s7 = "^}%95D" fullword ascii  $s8 = "zzz69806582" fullword ascii  condition:  uint16(0) == 0x5a4d and  uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and  1 of ($x\*) and all of them  } |



1. **IDA Python**

显示当前选定地址的函数起始地址、参数数量和局部变量数量

|  |
| --- |
| # -\*- coding: utf-8 -\*-  import idaapi  # 获取指定地址的函数的参数数量  def get\_function\_args\_count(ea):  func = idaapi.get\_func(ea)  if func:  cfunc = idaapi.decompile(func.startEA)  if cfunc:  # 获取函数的参数列表  args = cfunc.get\_lvars()  return len(args)  return None  # 获取指定地址的函数的局部变量数量  def get\_function\_locals\_count(ea):  func = idaapi.get\_func(ea)  if func:  return func.frsize // 4 # 假设是32位架构  else:  return None  # 获取指定地址的函数的起始地址  def get\_function\_start\_address(ea):  func = idaapi.get\_func(ea)  if func:  return func.startEA  else:  return None  # 获取当前光标所在地址  current\_address = ScreenEA()  # 获取当前地址的函数信息  args\_count = get\_function\_args\_count(current\_address)  locals\_count = get\_function\_locals\_count(current\_address)  start\_address = get\_function\_start\_address(current\_address)  # 打印函数信息  print("Current selected address:", hex(current\_address))  print("Function start address:", hex(start\_address))  print("Number of arguments:", args\_count)  print("Number of local variables:", locals\_count) |



1. **实验结论及心得体会**
2. 实验结论：

Lab11-1： Lab11-1.exe 可能是一个潜在的恶意代码，通过动态分析观察到其在运行时创建了文件并进行了网络通信，这提示可能存在潜在的恶意行为。

Lab11-2： Lab11-2.exe 在静态分析中显示出一些可疑的字符串和导入函数，通过动态分析观察到其与远程服务器的通信，这表明可能存在与命令和控制服务器的连接。

Lab11-3： Lab11-3.exe 和 Lab11-3.dll 显示出明显的恶意行为，包括文件复制、启动索引服务、创建击键记录器等。动态分析进一步验证了这些行为，以及它是如何通过入口点重定向实现对索引服务的特洛伊木马化。

1. 心得体会

这三个实验通过深入分析不同的恶意代码示例，对恶意代码分析的方法和步骤有了更深的理解。静态分析帮助理解代码的结构、导入的函数和使用的字符串，而动态分析则能够实时监测代码在运行时的行为。

使用工具如 IDA Pro、Process Monitor、Wireshark 等在分析过程中发挥了重要作用。静态分析提供了全貌，而动态分析则能够看到代码的实际执行情况。学会使用这些工具，并理解它们的输出对于深入了解和分析恶意代码至关重要。

通过这些实验，还学到了如何识别可能的恶意行为，比如不寻常的文件操作、网络通信和对系统进程的干扰。这些技能对于在网络安全领域从事威胁狩猎和恶意代码分析工作都是非常有用的。