

# 南开大学

## 恶意代码分析与防治技术课程实验报告

### Lab7



学 院 网络空间安全学院  
专 业 信息安全  
学 号 221041  
姓 名 李雅帆  
班 级 信安班

## 一、实验目的

1. 完成课本 Lab7 的实验内容，编写 Yara 规则，并尝试 IDA Python 的自动化分析。
2. 深入理解恶意代码的行为特征及其对系统的影响。通过应用逆向工程技术，我们将对恶意代码进行全面分析，以识别其潜在的攻击方式和持久化手段。
3. 特别关注恶意代码如何与操作系统和网络环境进行交互，评估其可能造成的安全威胁。
4. 实验中将利用静态分析与动态分析相结合的方法，以便全面捕捉恶意代码的运行机制和通信模式，为未来的防御措施提供参考。

## 二、实验原理

### 1. 静态分析

- (1) 使用逆向工程工具（如 Ghidra 或 Radare2）对恶意代码进行初步分析，识别其基本结构和组成部分。
- (2) 检查导入表，分析其使用的系统 API，了解恶意代码可能调用的关键功能。
- (3) 识别和解读重要的函数和数据结构，探讨其内部逻辑及潜在的攻击手段。

### 2. 持久化机制分析

- (1) 分析恶意代码是否尝试注册为系统服务或添加到启动项，以实现自我持久化。
- (2) 检查互斥量的使用情况，理解其如何确保同一时间只运行一个实例，从而避免资源争夺和异常。

### 3. 动态分析

- (1) 在隔离的虚拟环境中执行恶意代码，监控其运行时行为。
- (2) 记录系统调用、文件操作和网络活动，以捕捉恶意代码的实际行为。
- (3) 使用工具（如 Wireshark 或 Fiddler）分析网络流量，识别恶意代码与外部服务器的通信模式。

### 4. COM 组件分析

- (1) 调查恶意代码如何利用组件对象模型（COM）与系统进行交互，特别是通过 Internet Explorer 进行网络活动。
- (2) 评估其利用 COM 接口的能力，识别与广告或恶意网站的访问行为。

## 三、实验过程

### • Lab 7-1

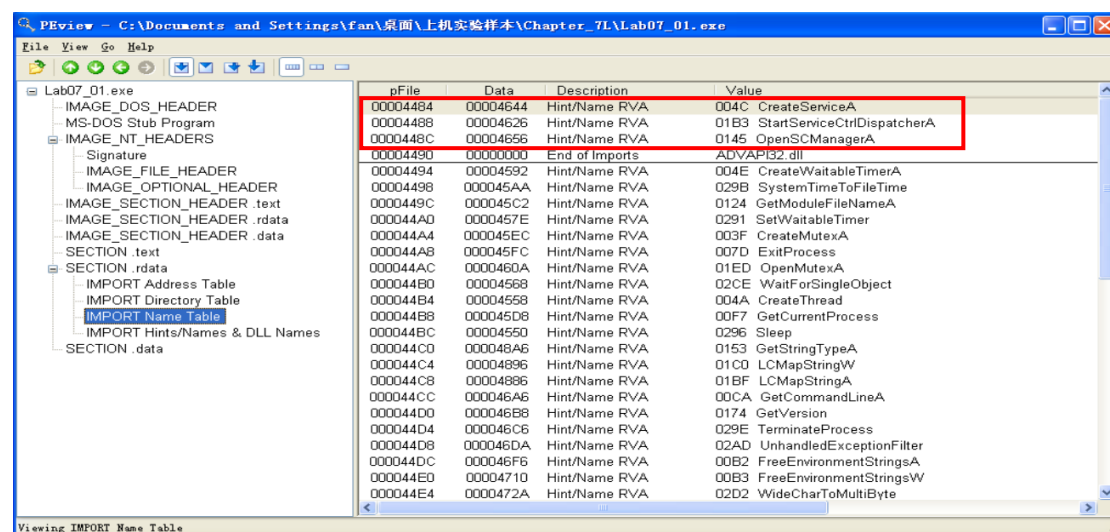
分析在文件 Lab07-01.exe 中发现的恶意代码。

#### 问题

#### 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)?

先使用 PView 查看导入函数表, 从 Advapi32.dll 中导入的函数来看, 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。

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。



进一步分析代码, 我们发现 main 函数首先调用了 StartServiceCtrlDispatcherA 函数, 以实现服务功能。此函数的参数表明该恶意软件注册的服务名称为 “MalService”, 并指定了服务控制函数为 sub\_401040。当调用 StartServiceCtrlDispatcherA 后, sub\_401040 函数会被执行。

```

401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
401000 _main      proc near      ; CODE XREF: start+AF1p
401000
401000 ServiceStartTable= SERVICE_TABLE_ENTRYA ptr -10h
401000 var_8      = dword ptr -8
401000 var_4      = dword ptr -4
401000 argc      = dword ptr 4
401000 argv      = dword ptr 8
401000 envp      = dword ptr 0Ch
401000
401000      sub     esp, 10h
401003      lea     eax, [esp+10h+ServiceStartTable]
401007      mov     [esp+10h+ServiceStartTable.lpServiceName], offset aMalService ; "MalService"
40100F      push    eax ; lpServiceStartTable
401010      mov     [esp+14h+ServiceStartTable.lpServiceProc], offset sub_401040
401018      mov     [esp+14h+var_8], 0
401020      mov     [esp+14h+var_4], 0
401028      call   ds:StartServiceCtrlDispatcherA
40102E      push    0
401030      push    0
401032      call   sub_401040
401037      add     esp, 18h
40103A      retn
40103A _main      endp

```

在 sub\_401040 函数内，代码首先处理与互斥量相关的操作，接着使用 OpenSCManager 打开服务控制管理器的句柄。之后，它调用 GetCurrentProcess 获取当前进程的伪句柄，并使用这个伪句柄调用 GetModuleFileName 函数，从而获取该恶意代码的完整路径名。最终，这个路径名被传递给 CreateServiceA 函数，将恶意代码注册为名为“MalService”的服务。值得注意的是，CreateServiceA 函数的参数 dwStartType 被设置为 2 (SERVICE\_AUTO\_START)，确保服务在启动时自动运行。这样，即使系统重启，该恶意服务也会自动运行，实现持久化驻留。

```

.text:00401064 loc_401064: ; CODE XREF: sub_401040+1A7j
.text:00401064      push    esi
.text:00401065      push    offset Name ; "HGL345"
.text:00401066      push    0 ; bInitialOwner
.text:0040106C      push    0 ; lpMutexAttributes
.text:0040106E      call   ds:CreateMutexA
.text:00401074      push    3 ; dwDesiredAccess
.text:00401076      push    0 ; lpDatabaseName
.text:00401078      push    0 ; lpMachineName
.text:0040107A      call   ds:OpenSCManagerA
.text:00401080      mov     esi, eax
.text:00401082      call   ds:GetCurrentProcess
.text:00401088      lea     eax, [esp+404h+Filename]
.text:0040108C      push    3E8h ; nSize
.text:00401091      push    eax ; lpFilename
.text:00401092      push    0 ; hModule
.text:00401094      call   ds:GetModuleFileNameA
.text:0040109A      push    0 ; lpPassword
.text:0040109C      push    0 ; lpServiceStartName
.text:0040109E      push    0 ; lpDependencies
.text:004010A0      push    0 ; lpdwTagid
.text:004010A2      lea     ecx, [esp+414h+Filename]
.text:004010A6      push    0 ; lpLoadOrderGroup
.text:004010A8      push    ecx ; lpBinaryPathName
.text:004010A9      push    0 ; dwErrorControl
.text:004010AB      push    2 ; dwStartType
.text:004010AD      push    10h ; dwServiceType
.text:004010AF      push    2 ; dwDesiredAccess
.text:004010B1      push    offset DisplayName ; "MalService"
.text:004010B6      push    offset DisplayName ; "MalService"
.text:004010BB      push    esi ; hSCManager
.text:004010BC      call   ds:CreateServiceA
.text:004010C2      xor     edx, edx
.text:004010C4      lea     eax, [esp+404h+FileTime]
.text:004010C8      mov     dword ptr [esp+404h+SystemTime.wYear], edx
.text:004010CC      lea     ecx, [esp+404h+SystemTime]
.text:004010D0      mov     dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
.text:004010D4      push    eax ; lpFileTime
.text:004010D5      mov     dword ptr [esp+408h+SystemTime.wHour], edx
.text:004010D9      push    ecx ; lpSystemTime
.text:004010DA      mov     dword ptr [esp+40Ch+SystemTime.wSecond], edx
.text:004010DE      mov     [esp+40Ch+SystemTime.wYear], 834h
.text:004010E5      call   ds:SystemTimeToFileTime

```

## 2. 为什么这个程序会使用一个互斥量？

互斥量被设计来保证这个可执行程序任意给定时刻只有一个实例在系统上运行。

该恶意代码使用的互斥量的硬编码名为“HGL345”，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。

```
.text:00401064 loc_401064: ; CODE XREF: sub_401040+1A7j
.text:00401064      push     esi
.text:00401065      push     offset Name ; "HGL345"
.text:0040106A      push     0           ; bInitialOwner
.text:0040106C      push     0           ; lpMutexAttributes
.text:0040106E      call     ds:CreateMutexA
.text:00401074      push     3           ; dwDesiredAccess
.text:00401076      push     0           ; lpDatabaseName
.text:00401078      push     0           ; lpMachineName
```

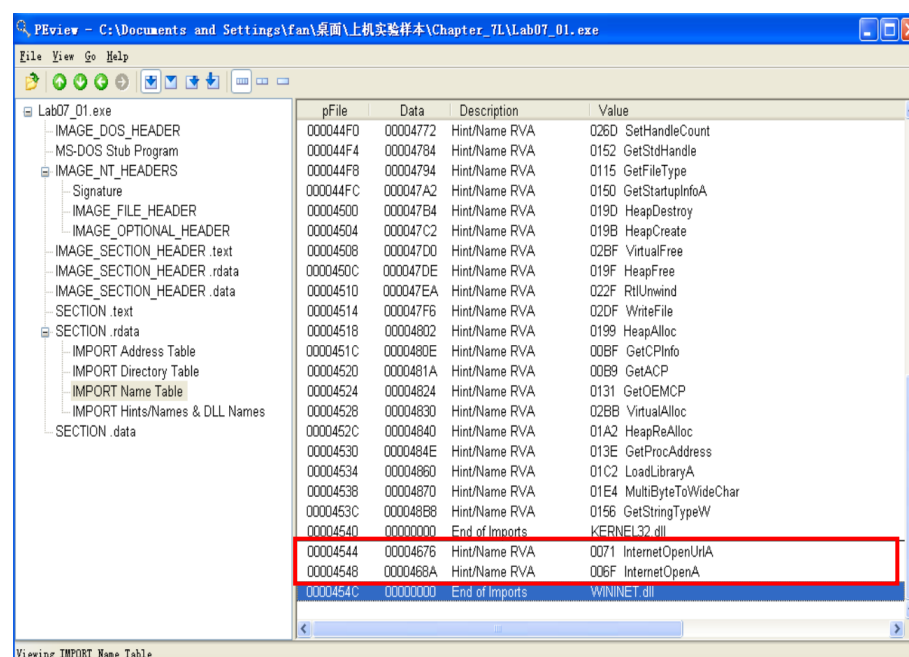
## 3. 可以用来检测这个程序的基于主机特征是什么？

名为“Malservice”的服务和名为“HGL345”的互斥量。

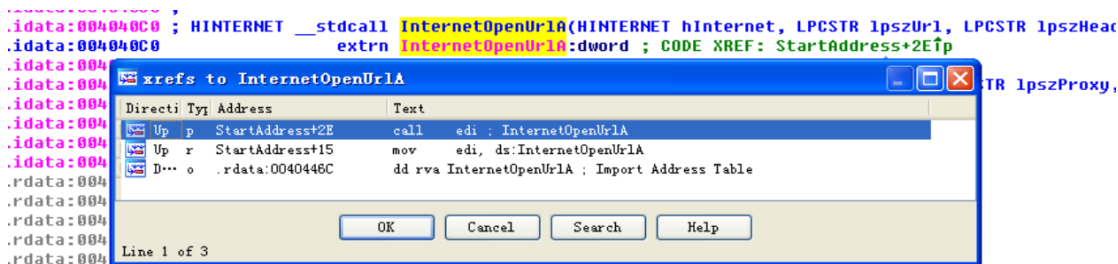
## 4. 检测这个恶意代码的基于网络特征是什么？

通过 PEvent 查看导入函数，该恶意代码从 `WinINet.dll` 中导入 `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。



为了进一步理解 InternetOpenA 函数的实现，使用 Ctrl+X 快捷键来查看它的交叉引用信息。



这两个函数的交叉引用实际上指向了同一处代码。通过双击该引用，我们可以跳转到该引用的位置，并发现 InternetOpenA 和 InternetOpenUrlA 的调用都位于 StartAddress 的子函数中。

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。

```
.text:00401150 StartAddress proc near ; DATA XREF: sub_401040+EC70
.text:00401150 |
.text:00401150 lpThreadParameter= dword ptr 4
.text:00401150
.text:00401150 push esi
.text:00401151 push edi
.text:00401152 push 0 ; dwFlags
.text:00401154 push 0 ; lpzProxyBypass
.text:00401156 push 0 ; lpzProxy
.text:00401158 push 1 ; dwAccessType
.text:0040115A push offset szAgent ; "Internet Explorer 8.0"
.text:0040115F call ds:InternetOpenA
.text:00401165 mov edi, ds:InternetOpenUrlA
.text:00401168 mov esi, eax
.text:0040116D loc_40116D: ; CODE XREF: StartAddress+304j
.text:0040116D push 0 ; dwContext
.text:0040116F push 80000000h ; dwFlags
.text:00401174 push 0 ; dwHeadersLength
.text:00401176 push 0 ; lpzHeaders
.text:00401178 push offset szUrl ; "http://www.malwareanalysisbook.com"
.text:0040117D push esi ; hInternet
.text:0040117E call edi ; InternetOpenUrlA
.text:00401180 jmp short loc_40116D
.text:00401180 StartAddress endp
```

## 5. 这个程序的目的是什么？

该恶意程序初始调用了 StartServiceCtrlDispatcherA 函数，随后调用了 sub\_401040 子函数。在这个子函数中，通过互斥量确保仅有一个该恶意软件实例正在执行，并将其配置为一个自启动的服务。

首先调用了 SystemTimeToFileTime 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，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。

```

.text:004010BC      call     ds:CreateServiceA
.text:004010C2      xor     edx, edx
.text:004010C4      lea     eax, [esp+404h+FileTime]
.text:004010C8      mov     dword ptr [esp+404h+SystemTime.wYear], edx
.text:004010CC      lea     ecx, [esp+404h+SystemTime]
.text:004010D0      mov     dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
.text:004010D4      push    eax                ; lpFileTime
.text:004010D5      mov     dword ptr [esp+408h+SystemTime.wHour], edx
.text:004010D9      push    ecx                ; lpSystemTime
.text:004010DA      mov     dword ptr [esp+40Ch+SystemTime.wSecond], edx
.text:004010DE      mov     [esp+40Ch+SystemTime.wYear], 834h
.text:004010E5      call    ds:SystemTimeToFileTime

```

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，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。

如果 `WaitForSingleObject` 函数是由于计时器对象转换为有信号状态而返回，则返回值是 0，若是出错、拥有 `mutex` 的线程结束而未释放计时器对象、等待时间达到指定毫秒，则返回值非 0。也就是说若执行出现意外，则检测 `eax` 值后 `jnz` 跳转到 `loc_40113B` 出，开始睡眠 `FFFFFFFFh` 毫秒；若等待到计时器出现信号，即等到 2100 年 1 月 1 日 0 点，则正常往下执行。

恶意代码将开始一段循环（典型的 `for` 循环结构），循环次数为 14h 即 20 次，每次循环都创建一个执行 `StartAddress` 子函数的线程，而由 1.4 中的分析，`StartAddress` 函数会以 Internet Explorer 8.0 位代理服务器访问 `http://www.malwarenanlysisbook.com`，且网址访问代码是在一个无限循环中，也就是说每一个执行 `StartAddress` 函数的线程都会无限次持续访问目标网址。`for` 循环结束后，按执行顺序同样也进入 `loc_40113B` 处，休眠 `FFFFFFFFh` 毫秒。

```

.text:004010DE      mov     [esp+40Ch+SystemTime.wYear], 834h
.text:004010E5      call    ds:SystemTimeToFileTime
.text:004010EB      push    0                  ; lpTimerName
.text:004010ED      push    0                  ; bManualReset
.text:004010EF      push    0                  ; lpTimerAttributes
.text:004010F1      call    ds:CreateWaitableTimerA
.text:004010F7      push    0                  ; fResume
.text:004010F9      push    0                  ; lpArgToCompletionRoutine
.text:004010FB      push    0                  ; pfnCompletionRoutine
.text:004010FD      lea     edx, [esp+410h+FileTime]
.text:00401101      mov     esi, eax
.text:00401103      push    0                  ; lPeriod
.text:00401105      push    edx                ; lpDueTime
.text:00401106      push    esi                ; hTimer
.text:00401107      call    ds:SetWaitableTimer
.text:0040110D      push    0FFFFFFFFh         ; dwMilliseconds
.text:0040110F      push    esi                ; hHandle
.text:00401110      call    ds:WaitForSingleObject
.text:00401116      test    eax, eax
.text:00401118      jnz     short loc_40113B
.text:0040111A      push    edi
.text:0040111B      mov     edi, ds:CreateThread
.text:00401121      mov     esi, 14h

```



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。

6. 这个程序什么时候完成执行？

该程序每个线程都会持续不断地访问目标网址。因此，此程序将无法正常终止执行。

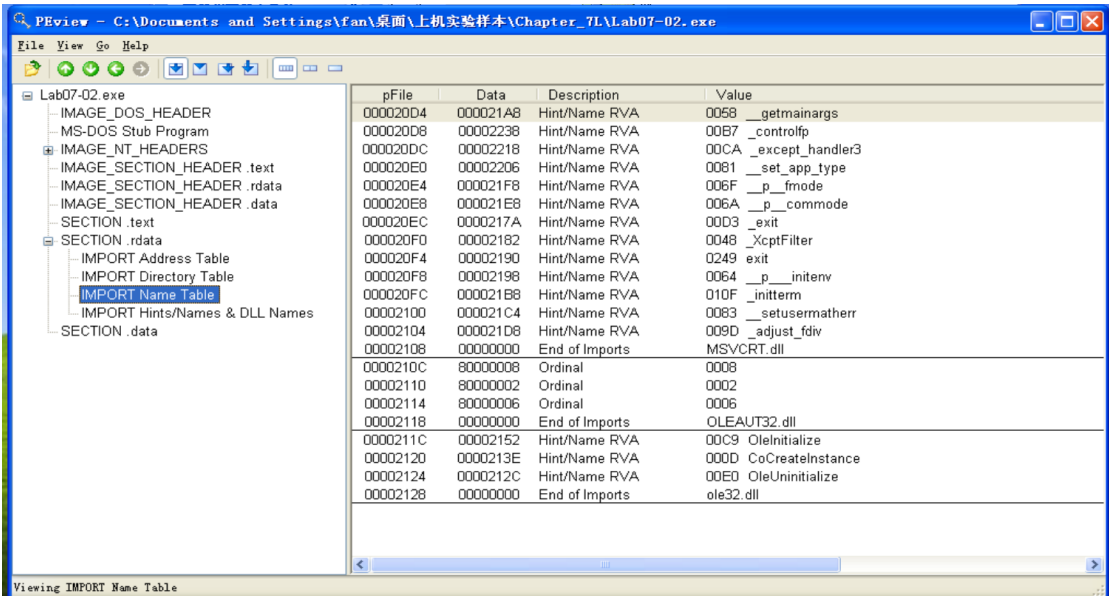
• Lab 7-2

分析在文件 Lab07-02.exe 中发现的恶意代码。

问题

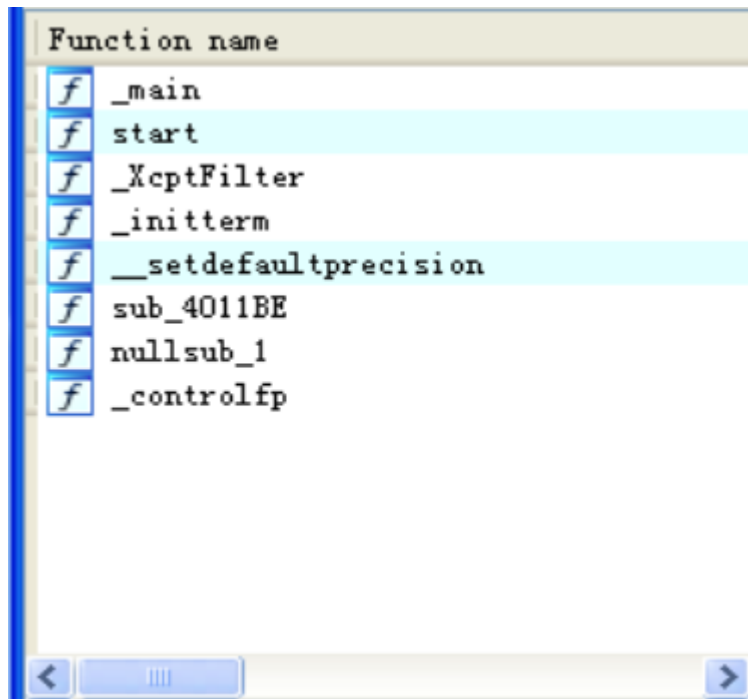
1. 这个程序如何完成持久化驻留？

在使用 PView 观察程序的导入表，未能识别与注册表、服务或其他可能用于实现持久化驻留的功能相关的函数。



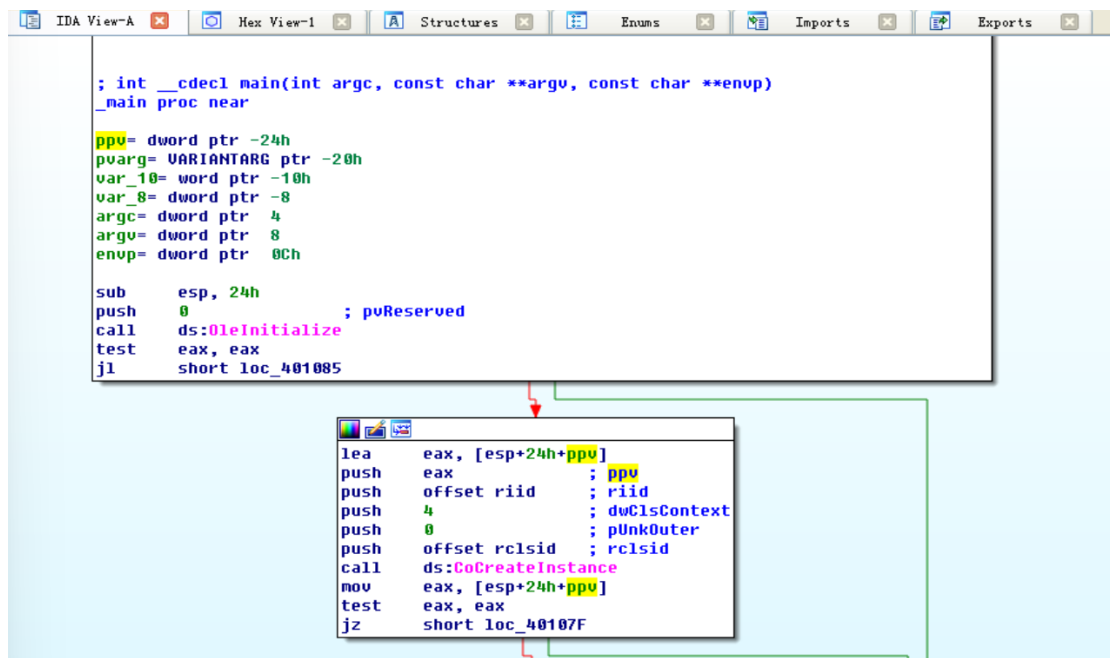
利用 IDA Pro 进行逆向分析进一步检查函数列表和相关代码，也没有找到持久化驻留的代码段。我推断该程序并未实现持久化驻留功能，而是执行一次后即终止。





## 2. 这个程序的目的是什么？

在程序的 main 函数中，首先执行了 OleInitialize 函数，这表示该代码意图使用组件对象模型（COM）的功能。随后，通过 CoCreateInstance 函数，程序尝试获得对 COM 功能的访问权限，并进一步检查了 rclsid 和 riid。



查阅资料可知，类型标识符 CLSID 的值为 0002DF01-0000-0000-C000-00000000046，代表 Internet Explorer 的标识；接口标识符 IID 的值为 D30C1661-CDAF-11D0-8A3E-00C04FC9E26E，代表 IWebBrowser2 接口。

```

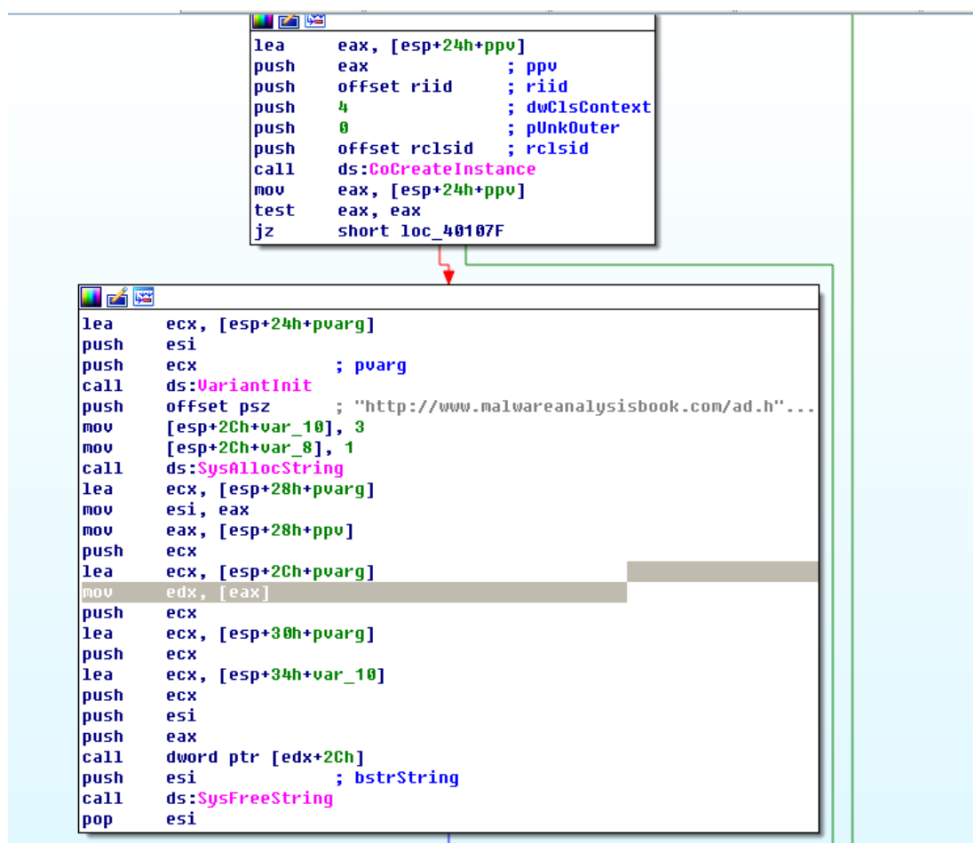
.rdata:00402058 ; Segment permissions: Read
.rdata:00402058 _rdata      segment para public 'DATA' use32
.rdata:00402058          assume cs:_rdata
.rdata:00402058          ;org 402058h
.rdata:00402058          ; IID rclsid
.rdata:00402058 rclsid      dd 2DF01h          ; Data1 ; DATA XREF: _main+1070
.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+1470
.rdata:00402068          dw 0CDAFh          ; Data2
.rdata:00402068          dw 11D0h          ; Data3
.rdata:00402068          db 8Ah, 3Eh, 0, 0C0h, 4Fh, 0C9h, 0E2h, 6Eh ; Data4

```

接下来我们看到，程序首先使用 VariantInit 函数来初始化变量。接着，通过 SysAllocString 函数，为字符串” http://www.malwareanalysisbook.com/ad.html” 分配了内存空间。

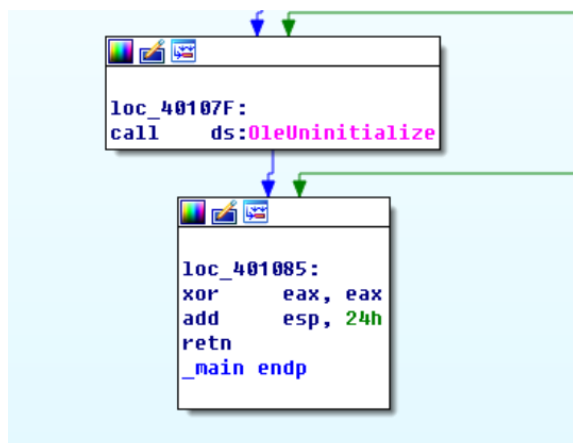
ppv 指向 COM 对象的位置。而指令 mov edx, [eax] 确保 edx 指向 COM 对象的基地址。接下来，指令 call dword ptr [edx+2Ch] 调用了位于 IWebBrowser2 接口偏移地址 0x2Ch（即 44）的函数。考虑到每个函数地址占 4 字节，这实际上是调用了序号 11，也就是第 12 个函数。这个函数被识别为 Navigate 函数，其功能是允许程序启动 Internet Explorer 并导航到一个指定的 Web 地址。

这个特定的地址是” http://www.malwareanalysisbook.com/ad.html”，这一地址已经被存储在之前通过 SysAllocString 分配的内存空间中。



在导航到上述网址后，程序正常终止。

该程序的主要目的是运行后，打开一个广告页面，然后如果联网的话就显示对应的广告内容，之后就退出程序的运行。



### 3. 这个程序什么时候完成执行？

根据之前的分析可知，这个程序在显示这个广告完成后执行。

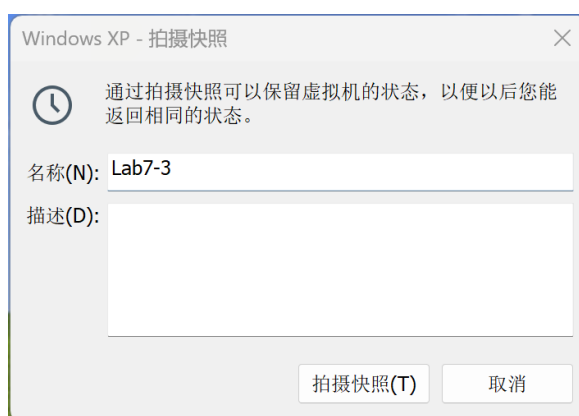
## • Lab 7-3

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。

### 问题

#### 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？

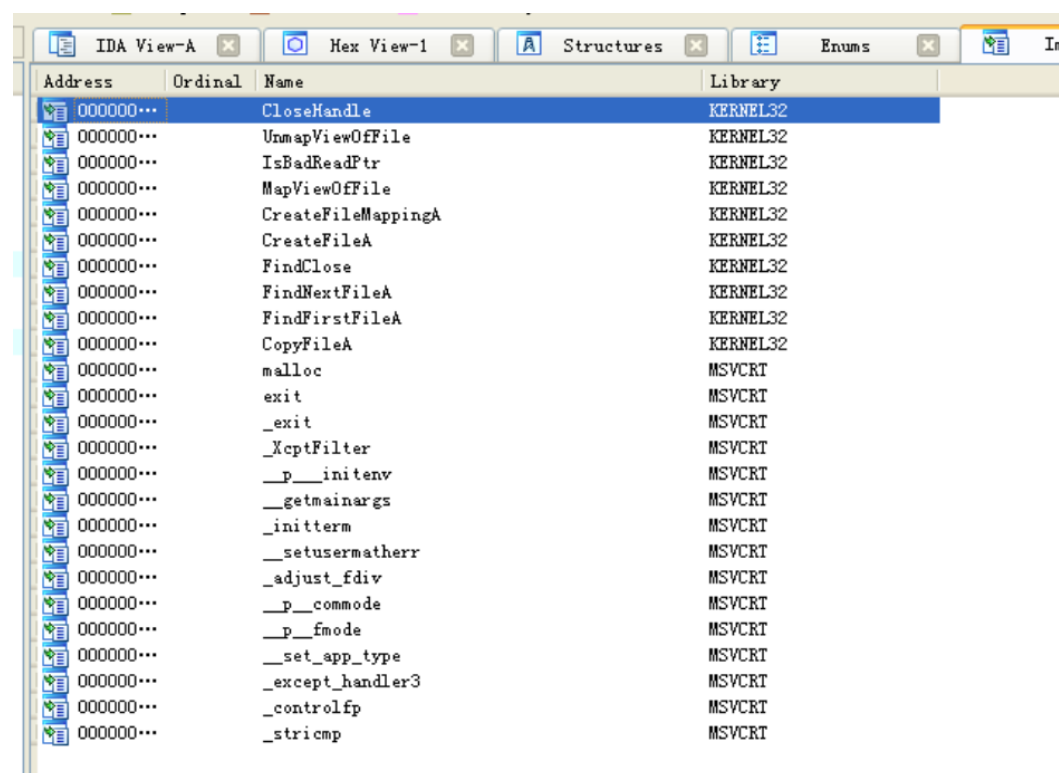
首先拍摄虚拟机快照。



用 IDA Pro 工具对 Lab07-03.exe 进行静态分析，观察其导入表。在此过程中，我们没有发现与注册表或服务相关的函数。然而，我们注意到了几个与文件操作相关的函数，例如 FindFirstFileA 和 FindNextFileA。

这些函数的存在暗示该代码可能被设计为遍历指定目录以查找文件。此外 CopyFileA 函数的存在表明该代码可能会复制找到的文件，可能会更改其位置或名称。

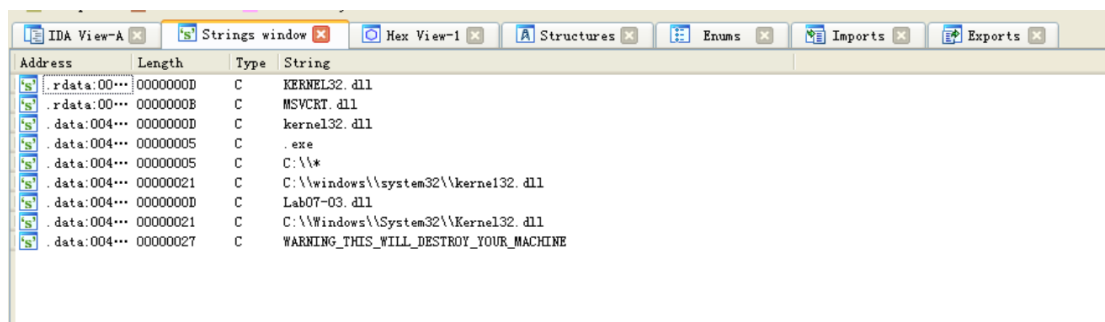
通过 CreateFileA、CreateFileMappingA 和 MapViewOfFile 函数，我们推断恶意代码可能会打开一个文件并将其映射到内存中。然而，值得注意的是，该代码并未导入 Lab07-03.dll 及其相关函数，这意味着这个 dll 文件并不是为 exe 文件提供专门的功能。



Address	Ordinal	Name	Library
00000000...		CloseHandle	KERNEL32
00000000...		UnmapViewOfFile	KERNEL32
00000000...		IsBadReadPtr	KERNEL32
00000000...		MapViewOfFile	KERNEL32
00000000...		CreateFileMappingA	KERNEL32
00000000...		CreateFileA	KERNEL32
00000000...		FindClose	KERNEL32
00000000...		FindNextFileA	KERNEL32
00000000...		FindFirstFileA	KERNEL32
00000000...		CopyFileA	KERNEL32
00000000...		malloc	MSVCRT
00000000...		exit	MSVCRT
00000000...		_exit	MSVCRT
00000000...		_XcptFilter	MSVCRT
00000000...		_p__initenv	MSVCRT
00000000...		__getmainargs	MSVCRT
00000000...		_initterm	MSVCRT
00000000...		__setusermatherr	MSVCRT
00000000...		_adjust_fdiv	MSVCRT
00000000...		_p__commode	MSVCRT
00000000...		_p__fmode	MSVCRT
00000000...		__set_app_type	MSVCRT
00000000...		__except_handler3	MSVCRT
00000000...		__controlfp	MSVCRT
00000000...		_strcmp	MSVCRT

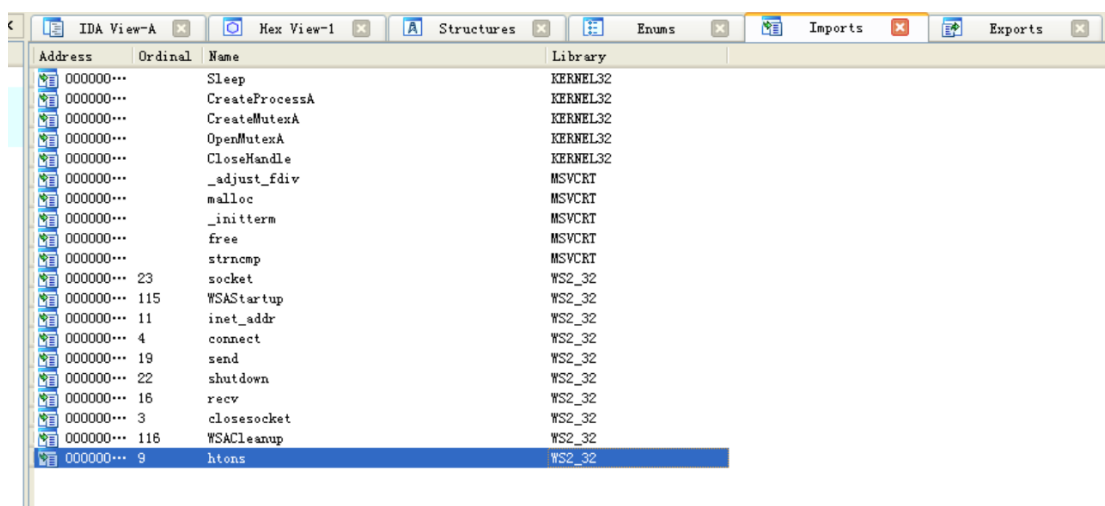
观察字符串信息时，“kernel32.dll”字符串将“kernel32.dll”中的字母“l”修改为了数字“1”，显然这是为了伪装恶意代码文件名，使其不容易被发现，在临近位置还出现了“Lab07-03.dll”。

综合上述导入函数的分析和路径字符串“C:\windows\system32\kernel32.dll”，我们推测该代码可能会遍历当前目录，找到 Lab07-03.dll 文件，然后将其复制到 C:\windows\system32\下，并将其重命名为 kernel32.dll。

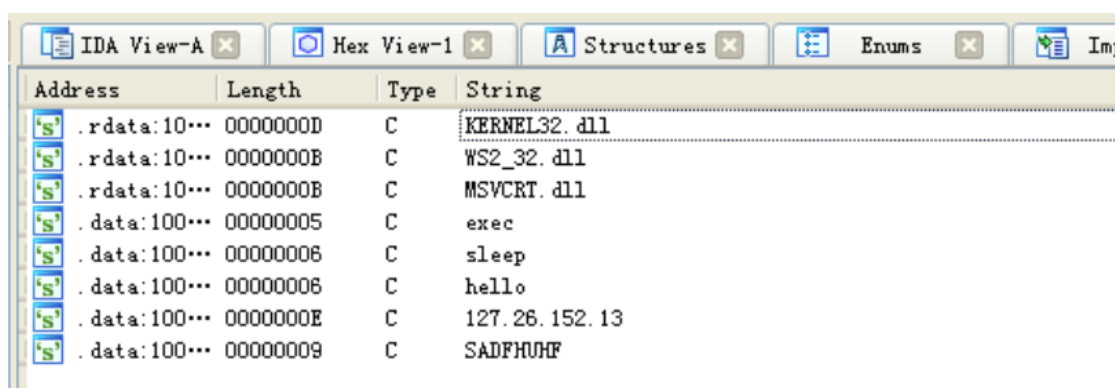


用 IDA Pro 工具对 Lab07-03.dll 进行静态分析，从 kernel32.dll 中导入了 CreateProcessA 以及与互斥量相关的函数。此外，它还按序号导入了 ws2\_32.dll 中的函数，这是 Winsock 库，其内部的函数与网络通信相关。

因此，我们可以推测该 dll 文件涉及网络访问操作。



查看 Lab07-03.dll 的字符串信息，我们发现“exec”、“sleep”以及“127.26.152.13”等关键字字符串。“exec”和“sleep”很可能是两个功能函数，分别用于执行命令和休眠。而“127.26.152.13”是一个 IP 地址。结合之前的分析，我们推测该 dll 可能会与此 IP 地址进行通信。



进一步研究代码的具体实现。

在 Lab07-03.exe 主函数的开始, 程序首先对 argc 的值与 2 进行比对, 以验证执行该可执行文件时的命令行参数是否恰为两个。若不满足此条件, 则程序会跳转至 loc\_401813 位置并立即终止执行。

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

var_44= dword ptr -44h
var_40= dword ptr -40h
var_3C= dword ptr -3Ch
var_38= dword ptr -38h
var_34= dword ptr -34h
var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_0C= dword ptr -0Ch
hObject= dword ptr -8
var_4= dword ptr -4
argc= dword ptr 4
argv= dword ptr 8
envp= dword ptr 0Ch

mov     eax, [esp+argc]
sub     esp, 44h
cmp     eax, 2
push    ebx
push    ebp
push    esi
push    edi
jnz     loc_401813

loc_401813:
pop     edi
pop     esi
pop     ebp
xor     eax, eax
pop     ebx
add     esp, 44h
retn
_main endp
```

之后程序获取命令行参数的地址列表, 记为 argv[]。通过地址偏移 [eax+4] 将 argv[1] 的值存入 eax 寄存器。

同时, 程序将字符串 "WARNING\_THIS\_WILL\_DESTROY\_YOUR\_MACHINE" 存入 esi 寄存器, 并可能后续对这两个寄存器中的值进行操作。

```
mov     eax, [esp+54h+argv]
mov     esi, offset aWarning_this_w ; "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
mov     eax, [eax+4]
```

在 loc\_401460 位置, 程序中有一个循环用于比较 eax 和 esi 寄存器中的值。具体地说, 它验证传入的第二个命令行参数是否与字符串 "WARNING\_THIS\_WILL\_DESTROY\_YOUR\_MACHINE" 匹配。





```

mov     edi, ds:CreateFileA
push    eax                ; hTemplateFile
push    eax                ; dwFlagsAndAttributes
push    3                  ; dwCreationDisposition
push    eax                ; lpSecurityAttributes
push    1                  ; dwShareMode
push    80000000h          ; dwDesiredAccess
push    offset FileName ; "C:\\Windows\\System32\\Kernel32.dll"
call    edi ; CreateFileA
mov     ebx, ds:CreateFileMappingA
push    0                  ; lpName
push    0                  ; dwMaximumSizeLow
push    0                  ; dwMaximumSizeHigh
push    2                  ; flProtect
push    0                  ; lpFileMappingAttributes
push    eax                ; hFile
mov     [esp+6Ch+hObject], eax
call    ebx ; CreateFileMappingA
mov     ebp, ds:MapViewOfFile
push    0                  ; dwNumberOfBytesToMap
push    0                  ; dwFileOffsetLow
push    0                  ; dwFileOffsetHigh
push    4                  ; dwDesiredAccess
push    eax                ; hFileMappingObject
call    ebp ; MapViewOfFile
push    0                  ; hTemplateFile
push    0                  ; dwFlagsAndAttributes
push    3                  ; dwCreationDisposition
push    0                  ; lpSecurityAttributes
push    1                  ; dwShareMode
mov     esi, eax
push    10000000h          ; dwDesiredAccess
push    offset ExistingFileName ; "Lab07-03.dll"
mov     [esp+70h+argc], esi
call    edi ; CreateFileA
cmp     eax, 0FFFFFFFFh
mov     [esp+54h+var_4], eax
push    0                  ; lpName
jnz     short loc_401503

```

大概观察一下这一段不难发现这里进行了创建文件、将文件映射到内存中的操作。我们注意到他在 C 盘目录下打开了 Kernel32.dll 文件，同时还有一个地方创建并打开了 Lab07-03.dll。

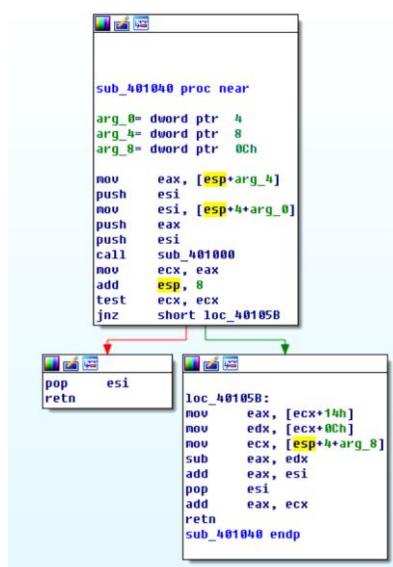
从地址 loc\_401538 开始，我们观察到一系列的 mov 和 push 指令。尽管这些指令的具体目的不易于直接解读，但值得注意的是，这些 mov 和 push 指令之间，存在几次 call 指令的执行。在每一组 mov 和 push 指令后，都伴随着对 sub\_401040 的调用。此外，在连续的调用之后，还有对 sub\_401070 的调用。

```

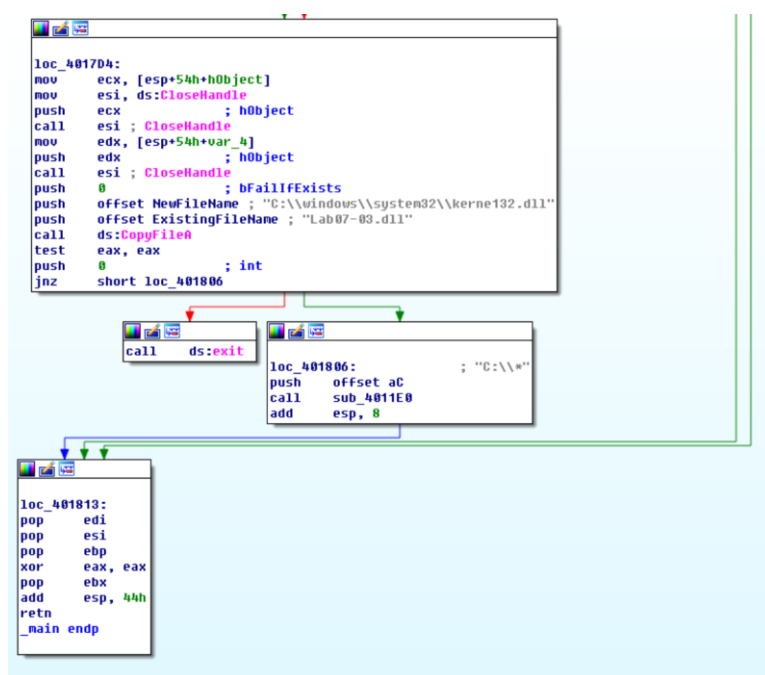
loc_401538:
mov     edi, [esi+3Ch]
push    esi
add     edi, esi
push    edi
mov     [esp+5Ch+var_1C], edi
mov     eax, [edi+78h]
push    eax
call    sub_401040
mov     esi, [ebp+3Ch]
push    ebp
add     esi, ebp
mov     ebx, eax
push    esi
mov     [esp+68h+var_30], ebx
mov     ecx, [esi+78h]
push    ecx
call    sub_401040
mov     edx, [esp+6Ch+argc]
mov     ebp, eax
mov     eax, [ebx+1Ch]
push    edx
push    edi
push    eax
call    sub_401040
mov     ecx, [esp+78h+argc]
mov     edx, [ebx+24h]
push    ecx
push    edi
push    edx
mov     [esp+84h+var_38], eax
call    sub_401040
mov     ecx, [ebx+20h]
mov     [esp+84h+var_20], eax
mov     eax, [esp+84h+argc]
push    eax
push    edi
push    ecx
call    sub_401040
mov     edx, [esp+90h+argv]
mov     edi, [esi+7Ch]
mov     [esp+90h+var_24], eax
mov     eax, [esi+78h]
push    edx
push    esi
push    eax
call    sub_401070
mov     ecx, edi
mov     esi, ebx
mov     edx, ecx
mov     edi, ebp
shr     ecx, 2
rep movsd
mov     ecx, edx

```

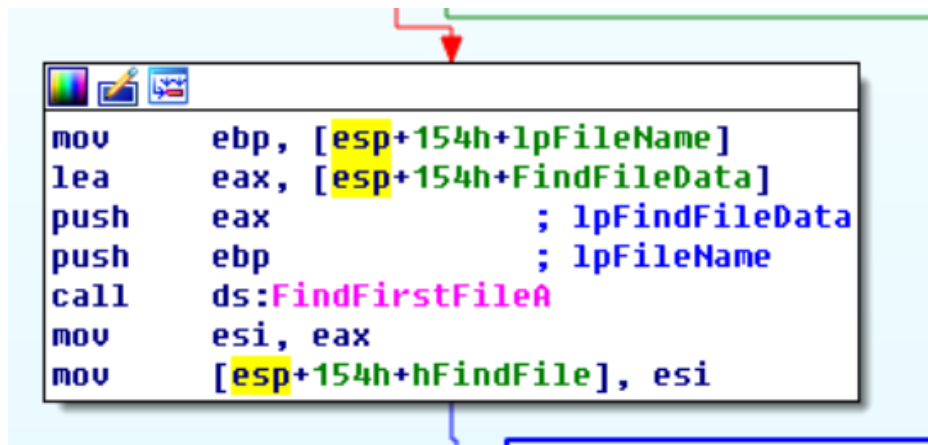
接下来，我们将具体研究这两个函数。观察发现，这两个函数主要执行了一系列内存操作。结合之前的恶意代码，它打开并将两个 d11 文件映射到内存中，我们推测这些操作可能是对这两个 d11 文件的处理。



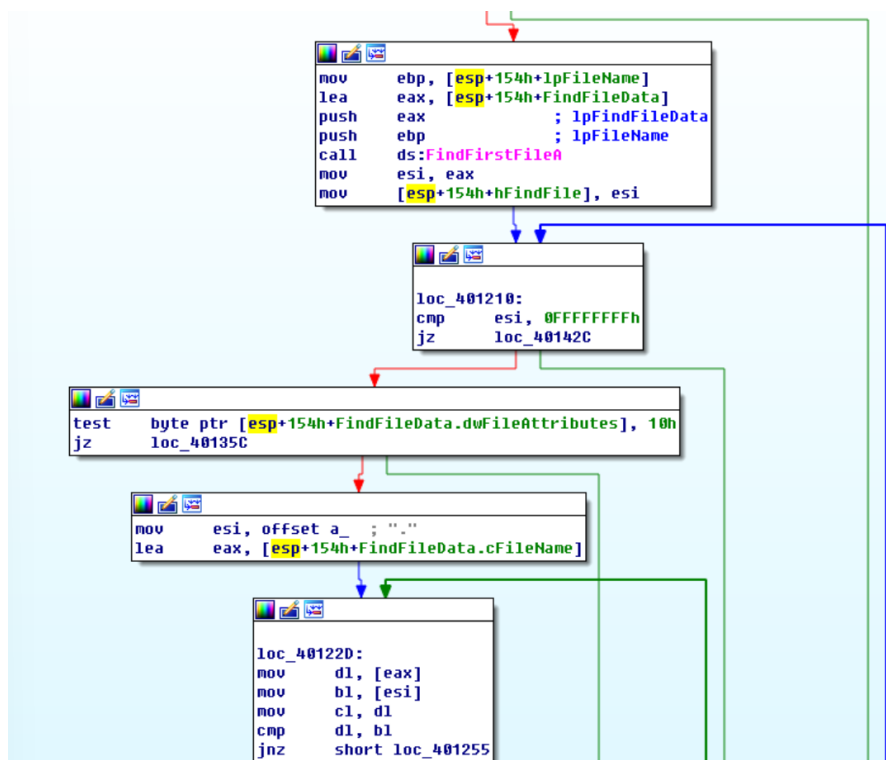
继续分析，我们发现还有大量的内存操作，直至达到地址 loc\_4017D4。从此处开始，出现了 Windows API 的调用以及一些关键的字符串信息。从先前的操作中，我们知道 hObject 和 var\_4 保存了两个文件的句柄，由此推断，此处的两次 CloseHandle 调用的目的是关闭这两个文件句柄。这可能意味着恶意代码已经完成了文件内存映射的修改，并将其保存回文件。接着是 CopyFileA 函数的调用，从其参数中我们可以推断，它是将 Lab07-03.dll 复制到 C:\windows\system32\并重新命名为 kernel32.dll。



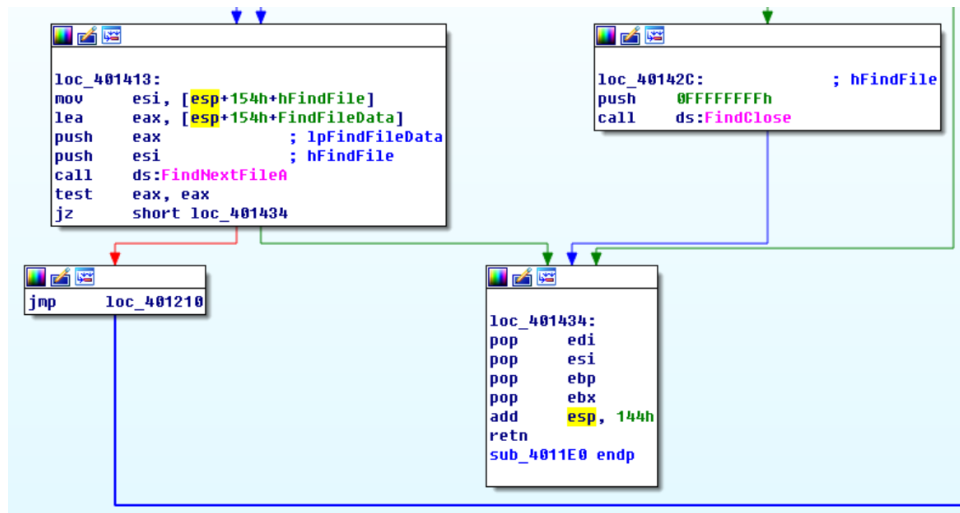
在地址 loc\_401806，我们注意到了另一个关键的函数调用，即对 sub\_4011E0 子函数的调用，其参数为字符串“C:\*”。对该子函数的深入分析显示，它首先调用了 FindFirstFile 函数，在 C:\下搜索第一个文件或目录。接着是一系列比较和算术运算指令，其目的不易解读。但在这些代码中，我们发现了可能的两次 malloc 函数调用和一次可能的 sub\_4011E0 调用，这意味着这个函数可能有递归调用的情况。最后，值得关注的是一次 strcmp 函数的调用，其参数为字符串“.exe”和 FindFirstFile 函数返回的 FindFileData 结构中的 dwReserved1 字段。接下来的逻辑是，如果这两个字符串相同，则调用 sub\_4010A0 函数。



FindFirstFile 和 FindNextFile 函数结合使用可以遍历目录。在调用了 FindNextFileA 函数后，只要其返回值不为 0（即目录遍历未完成），程序就会跳转回到刚调用完 FindFirstFileA 的位置，loc\_401210。



综上所述，sub\_4011E0 函数的主要目的是遍历 C:\ 目录，搜索 .exe 文件。每当发现一个 .exe 文件时，它都会调用 sub\_4010A0 函数。之前提到的可能的递归调用是因为存在子目录，当遍历到一个子目录时，会递归调用 sub\_4011E0。



我们需要进一步探索 `sub_4010A0` 函数的功能和目的。首先，观察到了一系列的 Windows API 函数调用，包括 `CreateFileA`、`CreateFileMappingA` 和 `MapViewOfFile`。这些函数被用于根据传入的文件路径参数将文件打开并映射到内存。基于此，存在一定的可能性，恶意代码会直接操作这块映射的内存以修改文件内容，而非通过其他的 Windows API。

```

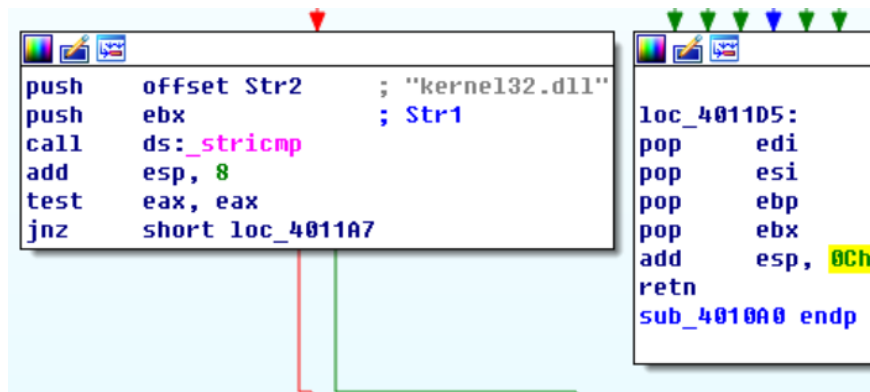
; int __cdecl sub_4010A0(LPCSTR lpFileName)
sub_4010A0 proc near

var_C= dword ptr -0Ch
hObject= dword ptr -8
var_4= dword ptr -4
lpFileName= dword ptr 4

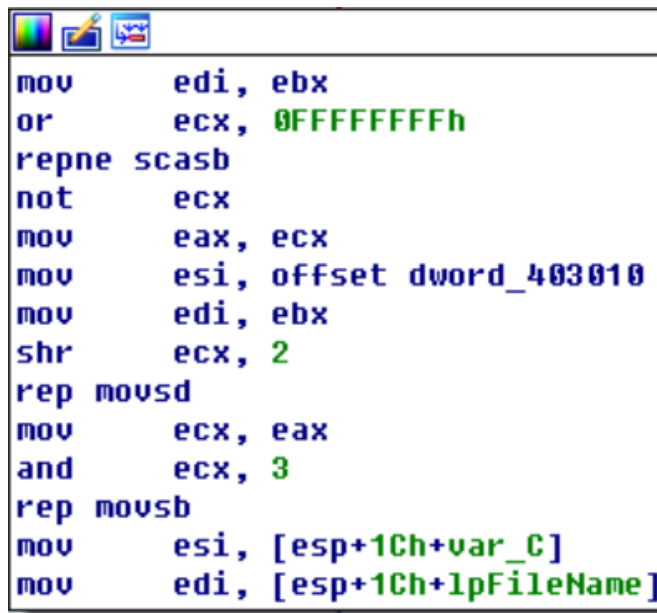
sub     esp, 0Ch
push    ebx
mov     eax, [esp+10h+lpFileName]
push    ebp
push    esi
push    edi
push    0           ; hTemplateFile
push    0           ; dwFlagsAndAttributes
push    3           ; dwCreationDisposition
push    0           ; lpSecurityAttributes
push    1           ; dwShareMode
push    10000000h    ; dwDesiredAccess
push    eax         ; lpFileName
call    ds:CreateFileA
push    0           ; lpName
push    0           ; dwMaximumSizeLow
push    0           ; dwMaximumSizeHigh
push    4           ; flProtect
push    0           ; lpFileMappingAttributes
push    eax         ; hFile
mov     [esp+34h+var_4], eax
call    ds:CreateFileMappingA
push    0           ; dwNumberOfBytesToMap
push    0           ; dwFileOffsetLow
push    0           ; dwFileOffsetHigh
push    0F001Fh     ; dwDesiredAccess
push    eax         ; hFileMappingObject
mov     [esp+30h+hObject], eax
call    ds:MapViewOfFile
mov     esi, eax
test    esi, esi
mov     [esp+1Ch+var_C], esi
jz      loc_4011D5

```

接下来观察到了 `IsBadReadPtr` 函数被调用了四次。该函数的主要目的是检查进程是否具有访问指定内存块的权限，即验证指针的合法性。



存在一个调用 `stricmp` 的实例，该函数用于比较一个通过内存地址和偏移获取的字符串与“kernel32.dll”是否相同。这种从内存中提取字符串的方法可能使得具体的字符串位置难以确定。



在 API 调用之外，识别到了三个特殊的指令：`repne scasb`、`rep movsd` 和 `rep movsb`。`Repne scasb` 指令与 00401183 至 0040118A 的代码段结合，用于计算字符串的长度。此处，`or` 指令被用于设置循环次数为-1，而 `repne scasb` 则持续搜索直至 `edi` 地址指向的字符串结束符。最终，`eax` 保存了字符串的长度，而 `ebx` 保存了名为 `Str1` 的字符串的地址。



`rep movsd` 指令与 0040118C 至 00401196 的代码段结合，用于将 `dword_403010` 位置的 `ecx` 个 `dword` 复制到 `ebx` 保存的地址。对于该复制的内容，经

过十六进制到字符串的转换，我们可以看到它是”kernel32.dll”。基于此，与之前的分析相结合，可以推测该代码段的目的是将一个.exe 文件中的”kernel32.dll” 字符串替换为”kernel32.dll”。

```

.data:00403010 aKerne132_dll db 'kerne132.dll',0 ; DATA XREF: sub_4010A0+EC10
.data:00403010 db 0 ; _main+1A81r ...
.data:0040301D db 0
.data:0040301E db 0
.data:0040301F db 0
.data:00403020 ; char Str2[]
.data:00403020 Str2 db 'kernel32.dll',0 ; DATA XREF: sub_4010A0+CE10
.data:0040302D align 10h
.data:00403030 ; char a_exe[]
.data:00403030 a_exe db '.exe',0 ; DATA XREF: sub_4011E0+1C110
.data:00403035 align 4
.data:00403038 asc_403038 db '\*',0 ; DATA XREF: sub_4011E0+13D10
.data:0040303B align 4
.data:0040303C a_ db '...',0 ; DATA XREF: sub_4011E0+8210
.data:0040303F align 10h
.data:00403040 a_ db '...',0 ; DATA XREF: sub_4011E0+4410
.data:00403042 align 4
.data:00403044 ; CHAR ac[]
.data:00403044 ac db 'C:\*',0 ; DATA XREF: _main:loc_40180610
.data:00403049 align 4
.data:0040304C ; CHAR NewFileName[]
.data:0040304C NewFileName db 'C:\windows\system32\kerne132.dll',0

```

rep movsb 指令在此上下文似乎没有明确的功能。最后，存在一个重要的向上跳转指令。这可能意味着存在一个循环结构。进一步分析显示，跳转的目标点位于最后一个 Is-BadReadPtr 函数调用之前。如果 IsBadReadPtr 检测到非法指针，则会跳出循环；如果指针合法，则会执行 stricmp 来比较字符串。这些行为暗示该子函数可能遍历映射到内存中的文件内容，寻找”kernel32.dll” 字符串并将其替换为”kernel32.dll”。函数的最后部分则涉及关闭映射和句柄，以及进行函数返回前的清理工作。

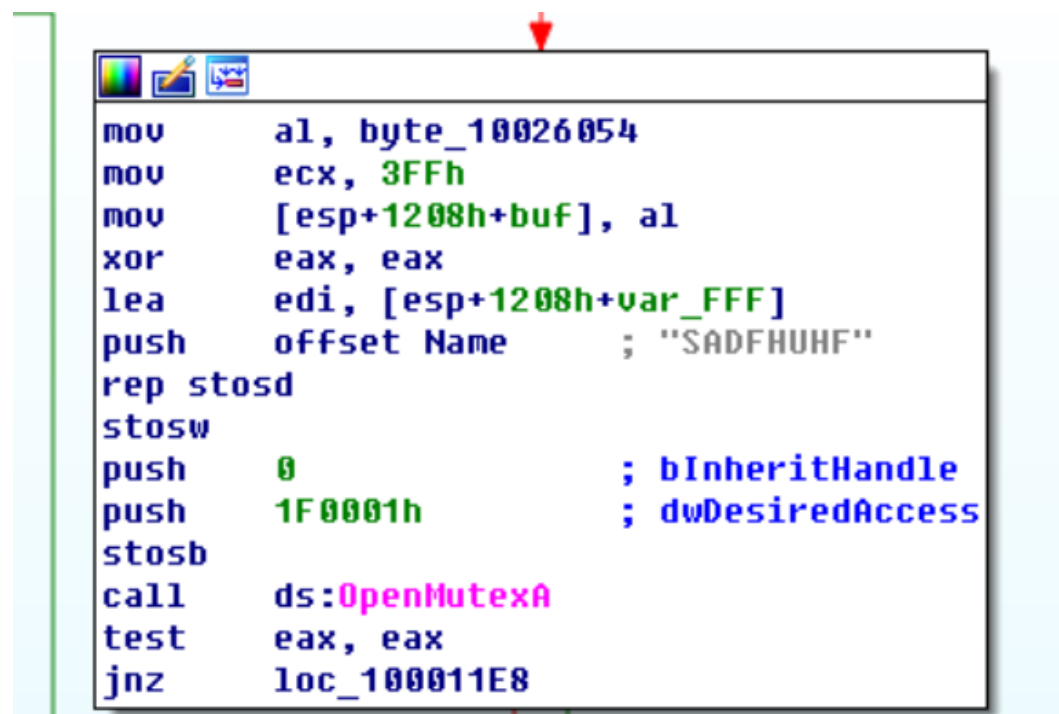
对于该恶意代码的功能和相关操作可以概述如下：该程序首先复制 “Lab07-03.dll” 到 “C:\Windows\System32\” 目录下，并对其进行重命名，命名为 “kernel32.dll”。随后，程序对 C 盘下的所有文件进行扫描，识别出 “.exe” 后缀的文件，并将文件内容中的 “kerne132.dll” 字符串替换为 “kernel32.dll”。一般情况下，当 “.exe” 文件中存在 “kernel32.dll” 字符串时，意味着此文件试图导入 “kernel32.dll” 中的函数。因此，通过上述操作，当 C 盘下的 “.exe” 文件试图导入 “kernel32.dll” 的函数时，实际上会加载 “kernel32.dll”。

## 2. 这个恶意代码的两个明显的基于主机特征是什么？

在 Lab07-03.exe 中，根据之前的分析结果，我们观察到一个显著特征，即存在一个硬编码的文件名”kernel32.dll”。



在 Lab07-03.dll 的 DllMain 函数中，我们注意到了互斥量相关函数的使用。此函数试图打开一个名为“SADFHUHF”的硬编码命名的互斥量，这是基于主机的另一个显著特征。



### 3. 这个程序的目的是什么？

根据之前的分析可知，Lab07-03.exe 的主要任务是安装 Lab07-03.dll 后门 dll 文件，并将所有 C 盘下的 exe 文件链接到这个 dll 文件。此后门会连接远程服务器并接收指令，它可以被指示进行休眠或创建新的进程。

### 4. 一旦这个恶意代码被安装, 你如何移除它？

从微软官方下载官方的 kernel32.dll，然后将它命名成 kernel32.dll 替换恶意文件，之后再留一个 kernel32.dll 的文件备份放在同目录下以供之后的程序进行使用。或者可以人工修改受到感染的 kernel32.dll，删除其中的恶意代码，只保留正常功能。

## • Yara 规则

### 1. 编写依据

通过 Strings 工具观察字符串，结合该.dll 和功能性函数以及文件大小和 PE 文件特征进行综合编写 Yara 检测规则。

### 2. yara 规则

```

rule lab0701exe
{
strings:
    $string1 = "HGL345"
    $string2 = "http://www.malwareanalysisbook.com"
    $string3 = "Internet Explorer 8.0"
condition:
    filesize<50KB and uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550 and all of
them
}

rule lab0702exe
{
strings:
    $string1 = "_controlfp"
    $string2="__setusermatherr"
    $fun1 = "OleUninitialize"
    $fun2 = "CoCreateInstance"
    $fun3= "OleInitialize"
    $dll1="MSVCRT.dll" nocase
    $dll2="OLEAUT32.dll" nocase
    $dll3="ole32.dll" nocase
condition:
    filesize<100KB and uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550 and all of
them
}

rule lab0703exe
{
strings:
    $string1 = "kernel32.dll"
    $string2 = "Lab07-03.dll"
condition:
    filesize<50KB and uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550 and all of
them
}

rule lab0703dll
{
strings:
    $string1 = "127.26.152.13"
    $string2 = "_adjust_fdiv"
condition:
    filesize<200KB and uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550 and all of

```

```
them  
}
```

### 3. 运行结果

```
PS C:\Users\lenovo> cd D:\fan\homework3\恶意代码分析与防治技术\计算机病毒分析工具\yara  
PS D:\fan\homework3\恶意代码分析与防治技术\计算机病毒分析工具\yara> .\yara64.exe -r .\yara7.y .\D:\fan\homework3\恶意代码分析与防治技术\上机实验样本\Chapter_7L\  
error scanning .\D:\fan\homework3\恶意代码分析与防治技术\上机实验样本\Chapter_7L\  
L\ : could not open file  
PS D:\fan\homework3\恶意代码分析与防治技术\计算机病毒分析工具\yara> .\yara64.exe -r .\yara7.y .\Chapter_7L\  
lab0702exe .\Chapter_7L\Lab07-02.exe  
lab0701exe .\Chapter_7L\Lab07-01.exe  
lab0703dll .\Chapter_7L\Lab07-03.dll  
lab0703exe .\Chapter_7L\Lab07-03.exe  
PS D:\fan\homework3\恶意代码分析与防治技术\计算机病毒分析工具\yara>
```

## • IDA Python 自动化分析

### 1. 功能

获取光标所在函数的函数名、开始地址和结束地址，分析函数 FUNC\_FAR、FUNC\_USERFAR、FUNC\_LIB（库代码）、FUNC\_STATIC（静态函数）、FUNC\_FRAME、FUNC\_BOTTOMBP、FUNC\_HIDDEN 和 FUNC\_THUNK 标志，获取当前函数中 jmp 或者 call 指令。

### 2. 代码

```
import idutils  
ea=idc.ScreenEA()  
funcName=idc.GetFunctionName(ea)  
func=idapi.get_func(ea)  
print("FuncName:%s"%funcName) # 获取函数名  
print "Start:0x%x,End:0x%x" % (func.startEA,func.endEA) # 获取函数开始地址和结束地址  
# 分析函数属性  
flags = idc.GetFunctionFlags(ea)  
if flags&FUNC_NORET:  
    print "FUNC_NORET"  
if flags & FUNC_FAR:  
    print "FUNC_FAR"  
if flags & FUNC_STATIC:  
    print "FUNC_STATIC"  
if flags & FUNC_FRAME:  
    print "FUNC_FRAME"  
if flags & FUNC_USERFAR:  
    print "FUNC_USERFAR"  
if flags & FUNC_HIDDEN:
```

```

    print "FUNC_HIDDEN"
if flags & FUNC_THUNK:
    print "FUNC_THUNK"
if not(flags & FUNC_LIB or flags & FUNC_THUNK):# 获取当前函数中 call 或者 jmp 的指令
    dism_addr = list(idautils.FuncItems(ea))
    for line in dism_addr:
        m = idc.GetMnem(line)
        if m == "call" or m == "jmp":
            print "0x%x %s" % (line, idc.GetDisasm(line))

```

### 3. 运行结果

```

FuncName: _main
Start: 0x401440, End: 0x40181d
0x401486 jmp     short loc_40148D
0x4014ac call    edi ; CreateFileA
0x4014c3 call    ebx ; CreateFileMappingA
0x4014d4 call    ebp ; MapViewOfFile
0x4014f0 call    edi ; CreateFileA
0x4014fd call    ds:exit
0x40150c call    ebx ; CreateFileMappingA
0x401515 call    ds:exit
0x401525 call    ebp ; MapViewOfFile
0x401532 call    ds:exit
0x401547 call    sub_401040
0x40155d call    sub_401040
0x40156e call    sub_401040
0x401581 call    sub_401040
0x401597 call    sub_401040
0x4015b0 call    sub_401070
0x4016c1 call    sub_401040
0x4017df call    esi ; CloseHandle
0x4017e6 call    esi ; CloseHandle
0x4017f4 call    ds:CopyFileA
0x401800 call    ds:exit
0x40180b call    sub_4011E0

```

## 四、实验结论及心得体会

在本次实验中，我深入学习了恶意代码的行为特征及其对系统的影响。这次实验让我对恶意代码的持久化机制、动态与静态分析方法有了更深刻的理解，也让我意识到网络安全的重要性。

我更加熟练掌握了如何使用静态分析工具对恶意代码进行初步分析，还学会了检查导入表，以了解其可能调用的系统 API。特别是对持久化机制的分析，让我明白恶意代码如何利用系统服务或注册表项实现自启动，从而在计算机重启后继续运行。

在动态分析环节，我在隔离的虚拟环境中执行恶意代码，监控其运行时行为。这一过程让我认识到，监控系统调用、文件操作和网络活动的重要性。使用工具

如 Wireshark 分析网络流量，使我能够识别恶意代码与外部服务器的通信模式，从而为制定防御措施提供了重要依据。

通过本次实验，我还特别关注了恶意代码与操作系统和网络环境的交互。这让我认识到，恶意代码不仅会对系统造成直接威胁，还可能通过网络传播，影响其他系统的安全。因此，加强对恶意代码的检测和防范显得尤为重要。