南開大學

恶意代码分析与防治技术实验报告

Lab07：分析恶意 Windows 程序

学　　院　网络空间安全学院

专　　业　信息安全、法学

学　　号　2112514

姓　　名　辛浩然

班　　级　信息安全、法学

一、实验目的

1. 识别恶意代码的主机感染迹象，分析恶意代码目的。

2. 熟悉一些恶意代码使用 windows 功能的独特方式。

二、实验原理

多数恶意代码以 Windows 平台为目标，并且与操作系统进行紧密交互。对基本 Windows 编程概念的深刻理解会帮助识别出恶意代码在主机上的感染迹象，跟踪恶意代码的执行，并最终分析出恶意代码的目的。

恶意 Windows 程序通常有以下恶意行为：

文件系统修改：恶意代码可以修改文件系统，包括创建、删除、修改或隐藏文件和目录。这可以用于隐藏自身，植入恶意文件或删除重要文件。

注册表修改：Windows 系统中的注册表是存储系统配置信息的数据库。恶意代码可以修改注册表中的键和值，以改变系统行为或自动启动恶意程序。

进程注入：恶意代码可以将自己注入到正在运行的合法进程中，以隐藏在系统中，绕过安全检测和监控。

系统服务修改：恶意代码可以修改系统服务或创建虚假的系统服务，以在系统启动时运行，或者用于后门访问系统。

网络通信：恶意代码可以与远程服务器通信，下载进一步的恶意软件、命令和控制服务器。

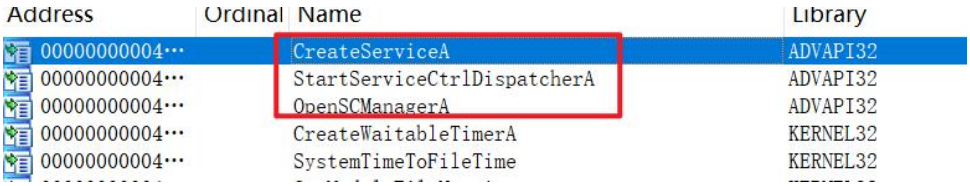
驱动程序安装：某些恶意代码可能安装恶意驱动程序，以在内核级别运行，更深度地修改系统行为和隐藏。

三、实验过程

Lab 07-01：

实验过程：

查看导入表信息，其中的 CreateServiceA 和 OpenSCManagerA 函数说明恶意代码可能创建一个服务，来保证它会在系统被重启时运行。StartServiceCtrlDispatcherA 导入函数提示了这个文件确实是一个服务。



其中的 InternetOpenA 和 InternetOpenUrlA 函数说明这个程序可能连接 Url 并下载内容。

接下来，查看反汇编代码的主函数。

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main           proc near               ; CODE XREF: start+AF↓p
.text:00401000
.text:00401000 ServiceStartTable= SERVICE_TABLE_ENTRYA ptr -10h
.text:00401000 var_8           = dword ptr -8
.text:00401000 var_4           = dword ptr -4
.text:00401000 argc            = dword ptr  4
.text:00401000 argv            = dword ptr  8
.text:00401000 envp            = dword ptr  0Ch
.text:00401000
.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
.text:0040103A _main           endp
.text:0040103A
.text:0040103A ; ---------------------------------------------------------------------------
```

主函数首先调用 StartServiceCtrlDispatcherA，该函数被程序用来实现一个服务，并且它通常立即被调用。这个函数指定了服务控制管理器会调用的服务控制函数。

这个函数的调用说明告恶意代码期望作为一个服务被运行。

随后主函数调用 sub_401040 函数，查看该函数代码。

```
.text:00401040 sub_401040      proc near               ; CODE XREF: _main+32↑p
.text:00401040                                         ; DATA XREF: _main+10↑o
.text:00401040
.text:00401040 SystemTime      = SYSTEMTIME ptr -400h
.text:00401040 FileTime        = _FILETIME ptr -3F0h
.text:00401040 Filename        = byte ptr -3E8h
.text:00401040
.text:00401040                 sub     esp, 400h
.text:00401046                 push    offset Name     ; "HGL345"
.text:0040104B                 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
```

首先试图获取一个名为 HGL345 的互斥量句柄，然后调用了 OpenMutexA 函数，如果这个调用成功，程序就会退出。

如果不成功，则调用 loc_401064 函数。查看该函数代码：

```
.text:00401064 loc_401064:                             ; CODE XREF: sub_401040+1A↑j
.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
.text:0040107A                 call    ds:OpenSCManagerA
.text:00401080                 mov     esi, eax
.text:00401082                 call    ds:GetCurrentProcess
```

loc_401064 首先创建一个名为 HGL345 的互斥量。这两处对互斥量组合调用，被设计来保证这个可执行程序任意给定时刻只有一份实例在系统上运行。如果有一个实例已经在运行了，则 sub_401040 函数对 OpenMutexA 的第一次调用成功，并且这个程序就会退出。

接下来，代码调用 OpenSCManager，它打开一个服务控制器的句柄，以便于这个程序增加或修改服务。

```
.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
```

然后调用 GetCurrentProcess 获取当前进程的伪句柄，紧接着调用 GetModuleFileName 函数，并传入刚获取的恶意代码进程伪句柄，从而获取恶意代码的全路径名，这个全路径名被传入 CreateServiceA 函数，从而将该恶意代码安装成一个名为“Malservice”的服务。

CreateServiceA 函数的参数中，dwStartType=2，即 SERVICE_AUTO_START，使服务为自启动，这样即实现了持久化驻留，即使计算机重启，也能维持运行。

接下来，恶意代码处理一些时间操作。可以看到一个 SystemTime 结构体，它用年、天、时、分、秒等不同域来表示时间。本例中，所有值首先被设置为 0，然后标识年的值被设置为 0x0834。这个事件表示 2100 年 1 月 1 日 0 时 0 分 0 秒。

然后程序调用 SystemTimeToFileTime 将系统时间转换为文件时间格式。

```
.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
```

接下来，程序调用 CreateWaitableTimer、SetWaitableTimer 以及 WaitForSingleObject。查看传给 SetWaitableTimer 的 lpDueTime 参数，也就是 SystemTimeToFileTime 返回的 FileTime,。这段代码随后使用 WaitForSingleObject 进入等待，直到 2100 年 1 月 1 日。

```
.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
```

等到 2100 年 1 月 1 日 0 时 0 分后，代码接着进入一个 20 次的循环，在每次循环中调用 CreateThread 创建线程，然后 lpStartAddress 参数是该线程的起始地址，在这里是 StartAddress。

双击查看 StartAddress。函数调用 InternetOpen 初始化一个到网络的连接。

然后进入一个死循环，调用 InternetopenUrlA，并且一直下载 www.malwareanalysisbook.com 的主页。而且由于 CreateThread 被调用了 20 次，所以会有 20 个线程一直调用 InternetopenUrlA。

通过前面分析，可以发现恶意代码使用互斥量来保证同一时刻只有一份实例在运行，它创建一个服务确保系统重启后它再次运行。通过将自己在多个机器上安装成一个服务，进而启动一个 DDoS 攻击。这个程序等到 2100 年 1 月 1 日的半夜，那时无限期地下载 www.malwareanalysisbook.com。如果所有被感染的机器在在同一时间(2100 年 1 月 1 日)连接到服务器，它们可能使服务器过载并无法访问该站点。

结合上面的分析，回答以下问题：

7-1-1 当计算机重启后，这个程序如何确保它继续运行(达到持久化驻留)?

这个程序创建服务 MalService，来确保它每次在系统启动后运行。

7-1-2 为什么这个程序会使用一个互斥量?

程序使用互斥量，保证同一时间这个程序只有一份实例在运行。

7-1-3 可以用来检测这个程序的基于主机特征是什么?

名为 HGL345 的互斥量和 MalService 服务。

7-1-4 检测这个恶意代码的基于网络特征是什么?

用户代理 Internet Explorer 8.0 ， 并和 www.malwareanalysisbook.com 通信。

7-1-5 这个程序的目的是什么?

这个程序等到 2100 年 1 月 1 日的半夜，那时无限期地下载 www.malwareanalysisbook.com。

如果所有被感染的机器在在同一时间(2100 年 1 月 1 日)连接到服务器，它们可能使服务器过载并无法访问该站点。

7-1-6 这个程序什么时候完成执行?

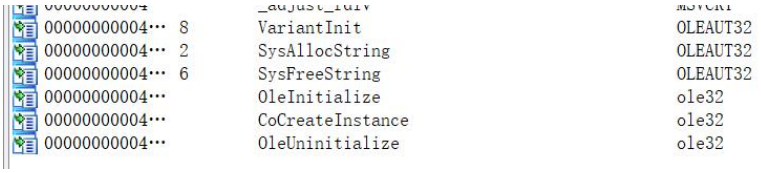
这个程序永远不会完成。它在一个定时器上等待直到 2100 年，到时候创建 20 个线程，每一个运行一个无限循环。

Lab 07-02

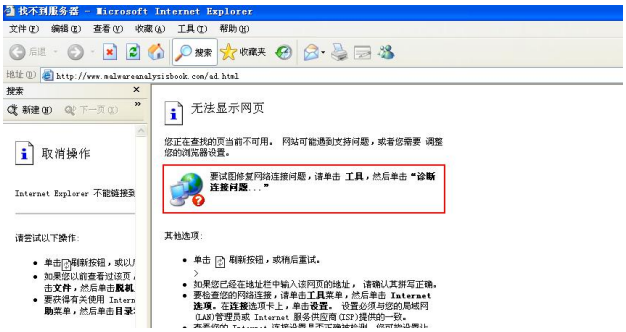
首先进行静态分析，可以看到一个 Unicode 字符串，是一个网址。

查看恶意代码的导入表：

看到与 COM 相关的 SysFreeString、VariantInit 等函数。CoCreateInstance 和 OleInitialize 是使用 COM 功能尤为重要的。



动态运行恶意代码，打开一个 Internet Explorer，试图访问刚看到的网站。



接下来，对恶意代码进行深入分析。查看 main 函数

```
01000 ; int __cdecl main(int argc, const char **argv, const char **envp)
01000 _main           proc near               ; CODE XREF: start+DE↓p
01000
01000 ppv             = dword ptr -24h
01000 pvarg           = VARIANTARG ptr -20h
01000 var_10          = word ptr -10h
01000 var_8           = dword ptr -8
01000 argc            = dword ptr  4
01000 argv            = dword ptr  8
01000 envp            = dword ptr  0Ch
01000
01000                 sub     esp, 24h
01003                 push    0               ; pvReserved
01005                 call    ds:OleInitialize
0100B                 test    eax, eax
0100D                 jl      short loc_401085
0100F                 lea     eax, [esp+24h+ppv]
01013                 push    eax             ; ppv
01014                 push    offset riid     ; riid
01019                 push    4               ; dwClsContext
0101B                 push    0               ; pUnkOuter
0101D                 push    offset rclsid   ; rclsid
01022                 call    ds:CoCreateInstance
01028                 mov     eax, [esp+24h+ppv]
0102C                 test    eax, eax
0102E                 jz      short loc_40107F
01030                 lea     ecx, [esp+24h+pvarg]
01034                 push    esi
01035                 push    ecx             ; pvarg
01036                 call    ds:VariantInit
```

函数首先调用了 OleInitialize 函数，初始化 Ole 的运行环境。然后调用 CoCreateInstance 用来创建组件，并返回这个组件的接口，即获得了一个 COM 对象。返回的 COM 对象保存在栈上，IDA Pro 标记为 ppv。

为了弄清楚程序在使用什么 COM 功能，查看调用 CoCreateInstance 的参数接口标识符 IID 和类标识符 CLSID。

```
402058 ; const IID rclsid
402058 rclsid          dd 2DF01h               ; Data1
402058                                         ; DATA XREF: _main+1D↑o
402058                 dw 0                    ; Data2
402058                 dw 0                    ; Data3
402058                 db 0C0h, 6 dup(0), 46h  ; Data4
402068 ; const IID riid
402068 riid            dd 0D30C1661h           ; Data1
402068                                         ; DATA XREF: _main+14↑o
402068                 dw 0CDAFh               ; Data2
402068                 dw 11D0h                ; Data3
402068                 db 8Ah, 3Eh, 0, 0C0h, 4Fh, 0C9h, 0E2h, 6Eh; Data4
```

通过检索，这个 IID 是 IWebBrowser2，CLSID 对应 Internet Explorer。

```
ext:00401035                push    ecx             ; pvarg
ext:00401036                call    ds:VariantInit
ext:0040103C                push    offset psz      ; "http://www.malwareanalysisbook.com
ext:00401041                mov     [esp+2Ch+var_10], 3
ext:00401048                mov     [esp+2Ch+var_8], 1
ext:00401050                call    ds:SysAllocString
ext:00401056                lea     ecx, [esp+28h+pvarg]
ext:0040105A                mov     esi, eax
ext:0040105C                mov     eax, [esp+28h+ppv]
ext:00401060                push    ecx
ext:00401061                lea     ecx, [esp+2Ch+pvarg]
ext:00401065                mov     edx, [eax]
ext:00401067                push    ecx
ext:00401068                lea     ecx, [esp+30h+pvarg]
ext:0040106C                push    ecx
ext:0040106D                lea     ecx, [esp+34h+var_10]
ext:00401071                push    ecx
ext:00401072                push    esi
ext:00401073                push    eax
ext:00401074                call    dword ptr [edx+2Ch]
ext:00401077                push    esi             ; bstrString
ext:00401078                call    ds:SysFreeString
ext:0040107E                pop     esi
```

首先调用 VariantInit 释放空间，初始化变量；然后调用 SysAllocString 为域名字符串分配空间。接下来，使 EDX 指向之前创建的 COM 对象的基址。随后调用这个对象中偏移 0x2C 处的一个函数。查阅资料知道，IWebBrowser2 接口的偏移 0x2C 是 Navigate 函数。因此，Navigate 函数被调用，Internet Explorer 将导航网址 http://www.malwareanalysisbook.com/ad.html。

调用 Navigate 函数之后，会执行一些清理函数，然后程序终止。这个程序不会持久化地安装它自己，并且也不修改系统。它简单地显示一个一次性的广告。

结合上面分析，回答以下问题：

7-2-1 这个程序如何完成持久化驻留？

这个程序没有完成持久化驻留，它运行一次然后退出。

7-2-2 这个程序的目的是什么？

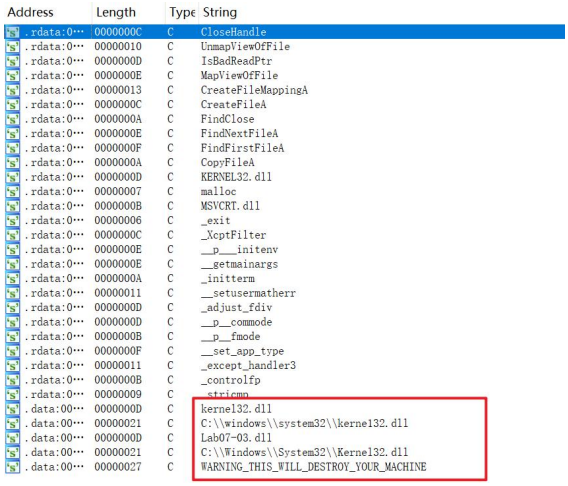
这个程序显示一个广告网页。

7-2-3 这个程序什么时候完成执行？

显示网页后完成执行。

Lab07-03

静态分析 exe 文件。首先查看字符串信息：



Address	Length	Type	String
.rdata:0…	0000000C	C	CloseHandle
.rdata:0…	00000010	C	UnmapViewOfFile
.rdata:0…	0000000D	C	IsBadReadPtr
.rdata:0…	0000000E	C	MapViewOfFile
.rdata:0…	00000013	C	CreateFileMappingA
.rdata:0…	0000000C	C	CreateFileA
.rdata:0…	0000000A	C	FindClose
.rdata:0…	0000000E	C	FindNextFileA
.rdata:0…	0000000F	C	FindFirstFileA
.rdata:0…	0000000A	C	CopyFileA
.rdata:0…	0000000D	C	KERNEL32.dll
.rdata:0…	00000007	C	malloc
.rdata:0…	0000000B	C	MSVCRT.dll
.rdata:0…	00000006	C	_exit
.rdata:0…	0000000C	C	_XcptFilter
.rdata:0…	0000000E	C	__p___initenv
.rdata:0…	0000000E	C	__getmainargs
.rdata:0…	0000000A	C	_initterm
.rdata:0…	00000011	C	__setusermatherr
.rdata:0…	0000000D	C	_adjust_fdiv
.rdata:0…	0000000D	C	__p__commode
.rdata:0…	0000000B	C	__p__fmode
.rdata:0…	0000000F	C	__set_app_type
.rdata:0…	00000011	C	_except_handler3
.rdata:0…	0000000B	C	_controlfp
.rdata:0…	00000009	C	_stricmp
.data:00…	0000000D	C	kernel32.dll
.data:00…	00000021	C	C:\\windows\\system32\\kernel32.dll
.data:00…	0000000D	C	Lab07-03.dll
.data:00…	00000021	C	C:\\Windows\\System32\\Kernel32.dll
.data:00…	00000027	C	WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

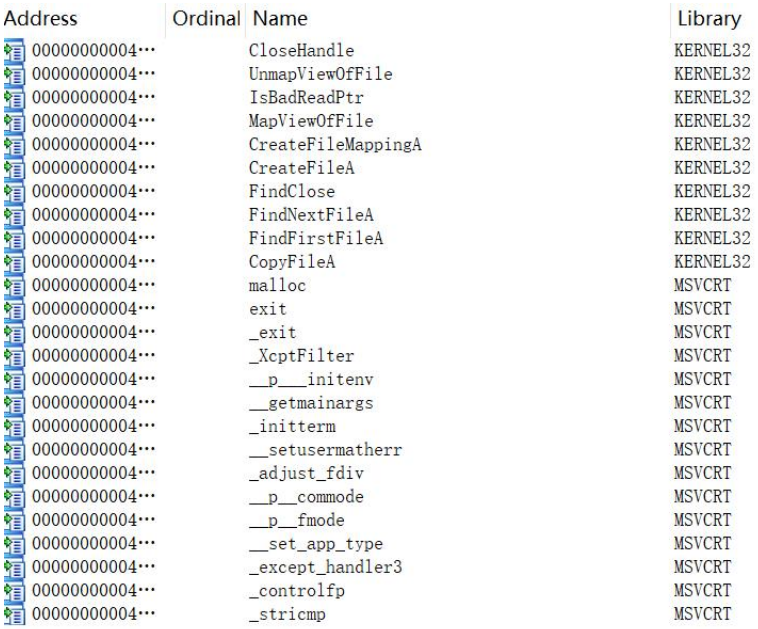
可以看到字符串 kerne132.dll，这是 kernel 将 l 替换为 1。

字符串 Lab07-03.dll 说明：exe 可能以某种方式在这个实验中访问这个 DLL。

查看导入表。

导入函数如 CreateFileA、CreateFileMappingA 以及 MapViewOfFile 说明这个程序可能打开一个文件，并将它映射到内存中。

FindFirstFileA 和 FindNextFileA 函数说明这个程序可能搜索目录，并使用 CopyFileA 来复制它找到的文件。



Address	Ordinal	Name	Library
00000000004…		CloseHandle	KERNEL32
00000000004…		UnmapViewOfFile	KERNEL32
00000000004…		IsBadReadPtr	KERNEL32
00000000004…		MapViewOfFile	KERNEL32
00000000004…		CreateFileMappingA	KERNEL32
00000000004…		CreateFileA	KERNEL32
00000000004…		FindClose	KERNEL32
00000000004…		FindNextFileA	KERNEL32
00000000004…		FindFirstFileA	KERNEL32
00000000004…		CopyFileA	KERNEL32
00000000004…		malloc	MSVCRT
00000000004…		exit	MSVCRT
00000000004…		_exit	MSVCRT
00000000004…		_XcptFilter	MSVCRT
00000000004…		__p___initenv	MSVCRT
00000000004…		__getmainargs	MSVCRT
00000000004…		_initterm	MSVCRT
00000000004…		__setusermatherr	MSVCRT
00000000004…		_adjust_fdiv	MSVCRT
00000000004…		__p__commode	MSVCRT
00000000004…		__p__fmode	MSVCRT
00000000004…		__set_app_type	MSVCRT
00000000004…		_except_handler3	MSVCRT
00000000004…		_controlfp	MSVCRT
00000000004…		_stricmp	MSVCRT

静态分析 dll 文件。首先查看字符串，可以看到一个 IP 地址：127.26.152.13，这个恶意代码可能会连接这个 IP 地址。

Address	Length	Type	String
.rdata:1000210A	0000000C	C	CloseHandle
.rdata:10002118	00000006	C	Sleep
.rdata:10002120	0000000F	C	CreateProcessA
.rdata:10002132	0000000D	C	CreateMutexA
.rdata:10002142	0000000B	C	OpenMutexA
.rdata:1000214E	0000000D	C	KERNEL32.dll
.rdata:1000215C	0000000B	C	WS2_32.dll
.rdata:1000216A	00000008	C	strncmp
.rdata:10002172	0000000B	C	MSVCRT.dll
.rdata:10002188	0000000A	C	_initterm
.rdata:10002194	00000007	C	malloc
.rdata:1000219E	0000000D	C	_adjust_fdiv
.data:10026018	00000006	C	sleep
.data:10026020	00000006	C	hello
.data:10026028	0000000E	C	127.26.152.13
.data:10026038	00000009	C	SADFHUHF

查看导入表，CreateProcessA 函数说明恶意代码可能创建一个进程；connect、send 等网络连接函数说明恶意代码会通过网络接受和发送数据。

Address	Ordinal	Name	Library
00000000100…		Sleep	KERNEL32
00000000100…		CreateProcessA	KERNEL32
00000000100…		CreateMutexA	KERNEL32
00000000100…		OpenMutexA	KERNEL32
00000000100…		CloseHandle	KERNEL32
00000000100…		_adjust_fdiv	MSVCRT
00000000100…		malloc	MSVCRT
00000000100…		_initterm	MSVCRT
00000000100…		free	MSVCRT
00000000100…		strncmp	MSVCRT
00000000100…	23	socket	WS2_32
00000000100…	115	WSAStartup	WS2_32
00000000100…	11	inet_addr	WS2_32
00000000100…	4	connect	WS2_32
00000000100…	19	send	WS2_32
00000000100…	22	shutdown	WS2_32
00000000100…	16	recv	WS2_32
00000000100…	3	closesocket	WS2_32
00000000100…	116	WSACleanup	WS2_32
00000000100…	9	htons	WS2_32

查看导出表，它没有任何导出函数。

IDA View-A | Hex View-1 | Structures | Enums | Imports | Exports

Name	Address	Ordinal
DllEntryPoint	00000000100012FA	[main entry]

进一步深入分析 dll 文件。查看代码执行 call 指令所调用的函数。

Address	Called function
.text:10001015	call __alloca_probe
.text:10001059	call ds:OpenMutexA
.text:1000106E	call ds:CreateMutexA
.text:1000107E	call ds:WSAStartup
.text:10001092	call ds:socket
.text:100010AF	call ds:inet_addr
.text:100010BB	call ds:htons
.text:100010CE	call ds:connect
.text:10001101	call ds:send
.text:10001113	call ds:shutdown
.text:10001132	call ds:recv
.text:1000114B	call ebp ; strncmp
.text:10001159	call ds:Sleep
.text:10001170	call ebp ; strncmp
.text:100011AF	call ebx ; CreateProcessA
.text:100011C5	call ds:Sleep
.text:100011D5	call ds:CloseHandle
.text:100011DC	call ds:closesocket
.text:100011E2	call ds:WSACleanup

首先调用库函数_alloca_probe，来在空间中分配栈。可以说明这个函数使用一个巨大的栈空间。随后调用 OpenMutexA 和 CreateMutexA 函数，这和 Lab07-91 中的恶意代码的行为一样，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。

结合以上分析，恶意代码发送和接收数据、创建进程，这说明它可能被设计来从一个远程机器接收命令。

接下来，需要查看恶意代码传输了什么信息。

首先，检查这个连接的目标地址。在 connect 调用前几行，可以看到一个对 inet_addr 的调用使用了固定的 IP 地址 127.26.152.13。也看到端口参数是 0x50，也就是端口 80，这个端口通常被 Web 流量所使用。

```
.text:1000109A  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 ; 'P'       ; hostshort
.text:100010B7  mov     dword ptr [esp+120Ch+name.sa_data+2], eax
.text:100010BB  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
```

查看 send 函数的调用：

```
.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
```

buf 参数保存了通过网络发送的数据，是一个字符串 hello。

再查看 recv 函数的调用：

```
.text:10001124  lea     eax, [esp+120Ch+buf]
.text:1000112B  push    1000h           ; len
.text:10001130  push    eax             ; buf
.text:10001131  push    esi             ; s
.text:10001132  call    ds:recv
```

首先获得一个指向栈中缓存区的指针，然后对 recv 的调用将连入的网络流量保存到栈上。而保存的值在后面会被检查。

```
.text:1000113C  lea     ecx, [esp+1208h+buf]
.text:10001143  push    5               ; MaxCount
.text:10001145  push    ecx             ; Str2
.text:10001146  push    offset Str1     ; "sleep"
.text:1000114B  call    ebp ; strncmp
.text:1000114D  add     esp, 0Ch
.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
```

这段代码调用 strncmp，并且它检查前 5 个字符是不是字符串 sleep。如果是，它调用 Sleep 函数来睡眠 60 秒。这说明，如果远程服务器发送 sleep 命令，这个程序将调用 Sleep 函数。

在后面的指令中缓冲区再次被检查：

```
.text:10001161 loc_10001161:                           ; CODE XREF: DllMain(x,x,x)+142↑j
.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+1230h+StartupInfo.cb], 44h ; 'D'
.text:100011AF                 call    ebx ; CreateProcessA
.text:100011B1                 jmp     loc_100010E9
```

这段代码检查这个缓冲区是否是以 exec 开始的。如果是，将调用 CreateProcessA 函数。

查看调用 CreateProcessA 函数时的 CommandLine 参数，它告诉我们要被创建的进程。

可以查看到 CommanLine 的值，值为 0xFFB：

```
.text:10001010
.text:10001010 ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
.text:10001010 _DllMain@12     proc near               ; CODE XREF: DllEntryPoint+4B↓p
.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
```

接收缓冲区在 0x1000 开始，CommanLine 的值为 0xFFB，可以知道这里是接收缓冲区的 5 个字节。这说明这个要被执行的命令是接收缓冲区中保存的任意 5 字节的东西。

这意味着从远程服务器接收到的数据将会是 exec Ful1PathOfProgramToRun。当这个恶意代码从远程服务器接收到这个 exec FullPathOfProgramToRun 命令行字符串时，它会用 FullPathOfProgramToRun 来调用 CreateProcessA。

总之，这个 DLL 实现了后门功能，这允许攻击者通过发送回复给 80 端口上的一个数据包，来启动一个系统上的可执行文件。

接下来，分析 exe 文件。

```
.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 aWarningThisWil ; "WARNING_THIS_WILL_DESTROY_YOUR_MAC
.text:0040145D                 mov     eax, [eax+4]
.text:00401460
.text:00401460 loc_401460:                             ; CODE XREF: _main+42↓j
.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
```

首先判断参数的个数是否为 2，如果参数不是 2，代码会提前退出。

接着将 argv[1]赋给 eax，将 WARNING_THIS_WILL_DESTROY_YOUR_MACHINE 字

符串存至 esi 寄存器。然后比较两个寄存器的值，即比较 argv[1]与指定字符串。如果二者不同，则返回退出。

接下来，可以看到对 CreateFile、CreateFileMapping 以及 MapViewofFile 的调用，打开了的 Kernel32.dll 文件和 Lab07-03.dll，并将其映射到了内存。

```
.text:0040148D                test    eax, eax
.text:0040148F                jnz     loc_401813
.text:00401495                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               ; lpName
.text:004014B6                push    0               ; dwMaximumSizeLow
.text:004014B8                push    0               ; dwMaximumSizeHigh
.text:004014BA                push    2               ; flProtect
.text:004014BC                push    0               ; lpFileMappingAttributes
.text:004014BE                push    eax             ; hFile
.text:004014BF                mov     [esp+6Ch+hObject], eax
.text:004014C3                call    ebx ; CreateFileMappingA
.text:004014C5                mov     ebp, ds:MapViewOfFile
.text:004014CB                push    0               ; dwNumberOfBytesToMap
.text:004014CD                push    0               ; dwFileOffsetLow
.text:004014CF                push    0               ; dwFileOffsetHigh
.text:004014D1                push    4               ; dwDesiredAccess
.text:004014D3                push    eax             ; hFileMappingObject
.text:004014D4                call    ebp ; MapViewOfFile
.text:004014D6                push    0               ; hTemplateFile
.text:004014D8                push    0               ; dwFlagsAndAttributes
.text:004014DA                push    3               ; dwCreationDisposition
.text:004014DC                push    0               ; lpSecurityAttributes
.text:004014DE                push    1               ; dwShareMode
.text:004014E0                mov     esi, eax
.text:004014E2                push    10000000h       ; dwDesiredAccess
.text:004014E7                push    offset ExistingFileName ; "Lab07-03.dll"
.text:004014EC                mov     [esp+70h+argc], esi
.text:004014F0                call    edi ; CreateFileA
.text:004014F2                cmp     eax, 0FFFFFFFFh
```

继续向后分析，可以看到它在两个打开的文件上调用 CloseHandle。先前调用了两次 CreateFileA 函数后对返回值的处理可以知道：hObject 和 var_4 分别保存了打开两个文件的句柄，因此这里先调用的两次 CloseHandle 是将两个被打开文件的句柄关闭，这意味着恶意代码完成了对文件内存映射的编辑操作并保存回文件。

然后它调用 CopyFile，复制 Lab07-03.dll 并把它放在 C:\Windows\System32\kerne132.dll，这很明显是想使其看起来像 kernel32.dll。

```
loc_4017D4:                             ; CODE XREF: _main+20D↑j
                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
```

接下来，可以发现调用 sub_4011E0，参数为 C:*。

```
.text:00401806 loc_401806:                             ; CODE XREF: _main+3BE↑j
.text:00401806                push    offset aC        ; "C:\\*"
.text:0040180B                call    sub_4011E0
.text:00401810                add     esp, 8
```

查看这个函数。它将传入的第一个参数标记为 lpFilename，然后调用 FindFirstFile，在 C:\下查找第一个文件或目录并返回其句柄。

```
.text:004011E0 ; int __cdecl sub_4011E0(LPCSTR lpFileName, int)
.text:004011E0 sub_4011E0      proc near               ; CODE XREF: sub_4011E0+16F↓p
.text:004011E0                                         ; _main+3CB↓p
.text:004011E0
.text:004011E0 hFindFile       = dword ptr -144h
.text:004011E0 FindFileData    = _WIN32_FIND_DATAA ptr -140h
.text:004011E0 lpFileName      = dword ptr  4
.text:004011E0 arg_4           = dword ptr  8
.text:004011E0
.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
```

接下来，可以看到许多算数运算与比较。接下来的分析过程中，重点关注一些函数调用：在后面可以看到调用自身 sub_4011e0，这说明这是一个调用自己的递归函数。

```
.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
```

后面还调用了 stricmp 函数，两个参数分别是字符串“.exe”和调用 FindFirstFile 函数返回的 FindFileDta 结构中的 dwReserved1 字段。比较二者是否相同，如果相同，则跳转至 sub_4010A0。

```
.text:0040136C                 lea     ebx, [esp+ecx+154h+FindFileData.dwReserved1]
.text:00401370                 or      ecx, 0FFFFFFFFh
.text:00401373                 repne scasb
.text:00401375                 not     ecx
.text:00401377                 dec     ecx
.text:00401378                 lea     edi, [esp+154h+FindFileData.cFileName]
.text:0040137C                 mov     edx, ecx
.text:0040137E                 or      ecx, 0FFFFFFFFh
.text:00401381                 repne scasb
.text:00401383                 not     ecx
.text:00401385                 dec     ecx
.text:00401386                 lea     eax, [edx+ecx+1]
.text:0040138A                 push    eax             ; Size
.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, 0FFFFFFFFh
.text:0040139F                 xor     eax, eax
.text:004013A1                 push    offset aExe     ; ".exe"
.text:004013A6                 repne scasb
.text:004013A8                 not     ecx
.text:004013AA                 sub     edi, ecx
.text:004013AC                 push    ebx             ; String1
.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
.text:0040140C loc_40140C:                             ; CODE XREF: sub_4011E0+221↑j
.text:0040140C                 mov     ebp, [esp+154h+lpFileName]
.text:00401413
.text:00401413 loc_401413:                             ; CODE XREF: sub_4011E0+177↑j
.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_401210
```

如果不同，则调用 FindNextFileA 函数，而 FindFirstFile 和 FindNextFile 函数结合使用可以遍历目录。在调用了 FindNextFileA 函数后只要返回值不为 0（没有遍历完）就会跳转

回 loc_401210 处，即刚调用完 FindFirstFileA 函数的地方。也就是说 sub_4011E0 函数用来遍历 C:\，查找.exe 文件，只要发现了一个.exe 文件就调用一次 sub_4010A0，而之前说的可能出现递归，则应该是由于子目录的存在，需要遍历整个文件系统。

接下来，详细分析函数 sub_4010A0：

可以看到它首先调用 CreateFile、CreateFileMapping 以及 MapViewOfFile 来映射整个文件到内存中。这说明整个文件被映射到内存空间。随后可以看到对 IsBadPtr 的算术调用，这主要验证指针是有效的。然后看到一个对 stricmp 的调用。

```
.text:0040116E                push    offset String2  ; "kernel32.dll"
.text:00401173                push    ebx             ; String1
.text:00401174                call    ds:_stricmp
.text:0040117A                add     esp, 8
.text:0040117D                test    eax, eax
.text:0040117F                jnz     short loc_4011A7
.text:00401181                mov     edi, ebx
.text:00401183                or      ecx, 0FFFFFFFFh
.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
```

主要就是调用了一次 stricmp 函数检查一个字符串是否为“kernel32.dll”。在该指令后面，指令后面，看到这个程序调用 repne scasb 及 rep movsd，这在功能上和 strlen 以及 memcpy 函数是等价的。

edi 中所保存的是传入 stricmp 的参数，也是通过调用试图写入的值。进一步查看这个值到底是什么，也就是偏移 dword_403010 处存放的内容。

跳转至该处，并转为字符串显示，内容为 kerne132.dll：

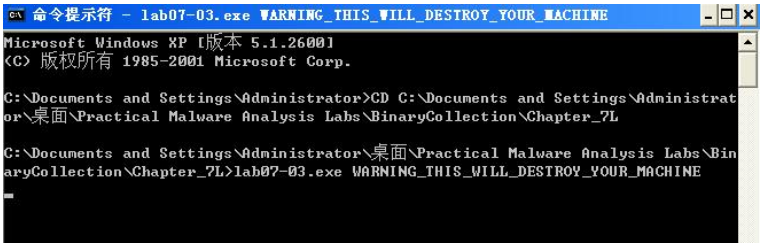
```
.data:00403010 aKerne132Dll   db 'kerne132.dll',0     ; DATA XREF: sub_4010A0+EC↑o
.data:00403010                                        ; _main+1A8↑r ...
.data:0040301D                db    0
.data:0040301E                db    0
.data:0040301F                db    0
```

结合上述分析，可以知道恶意代码遍历整个文件系统来查找以.exe 结尾的文件，在.exe 文件中找到字符串 kernel32.dll 的位置，并使用 kerne132.dll 替换它。而 Lab07-03.dll 被复制到 C:\Windows\System32 目录中并被命名为 kerne132.dll。

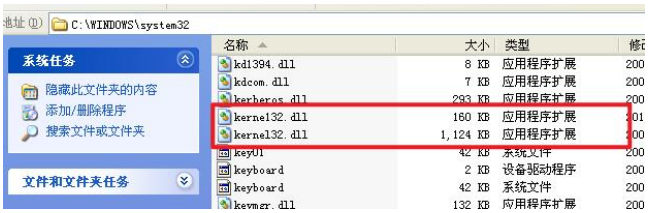
因此，恶意代码修改可执行文件让它们访问 kerne132.dll,而不是 kernel32.dll。这意味着可执行文件会加载写入的 kerne132.dll 而不是原本的 kernel32.dll。

基于以上分析，来进行基础动态分析。

动态运行该恶意代码：Lab07-03.exe WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

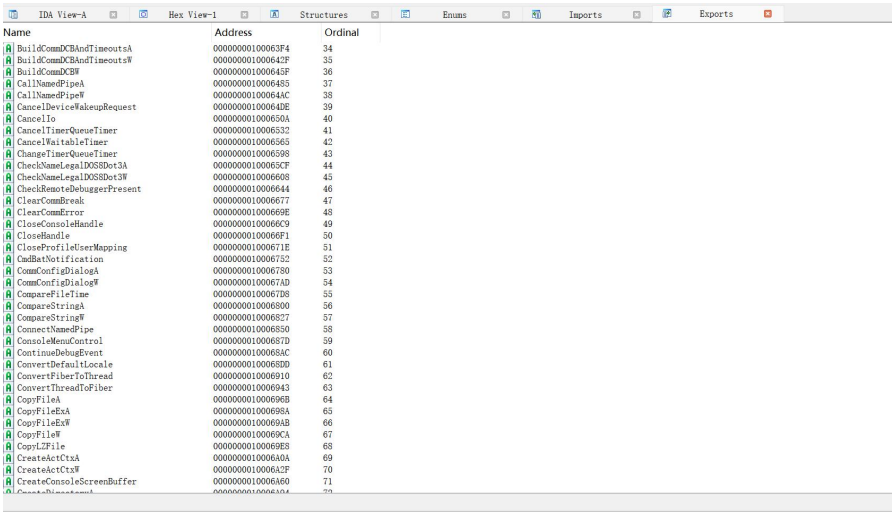


能够在 c:\windows\system32 目录下发现 kernel132.dll 的存在。



选择一个已经被打开并检查过导入函数的.exe 文件，可以确认来自 kernel32.dll 的导入函数已经被替换成来自 kerne132.dll 的导入函数。这意味着系统上的每一个可执行文件将会试图加载我们的恶意 DLL。

而查看修改过的 Lab07-03.dll（即 kerne132.dll），可以看到它导出了所有 kernel32.dll 的导出函数。这个修改的总体效果是任何时候当一个 exe 文件在这台计算机上运行时，它将加载恶意的 kerne132.dll 并运行在 DllMain 中的代码。除此之外，所有功能将不会改变，代码将还是像调用原始的 kernel32.dll 那样正常执行。



根据以上分析，回答下面问题：

7-3-1 这个程序如何完成持久化驻留，来确保在计算机被重启后它能继续运行？

它首先将 Lab07-03.dll 复制到 C:\Windows\System32\下并重命名为 kerne132.dll，然后将扫描 C:\下所有文件，找出.exe 文件并将其中的“kernel32.dll”字符串全部修改为“kerne132.dll”来达到持久化驻留。

7-3-2 这个恶意代码的两个明显的基于主机特征是什么？

这个程序通过硬编码来使用文件名 kerne132.dll；

Lab07-03.dll 的 DllMain 中一开始就使用了互斥量相关函数，尝试打开（创建）的互斥量采用硬编码命名，名字为“SADFHUHF”。

7-3-3 这个程序的目的是什么？

结合前面对 dll 文件的分析，这个程序的目的是创建一个很难删除的后门，来连接到一个远程主机。这个后门有两个命令：一个用来执行命令，一个用来睡眠。

7-3-4 一旦这个恶意代码被安装，你如何移除它？

这个程序它感染系统上的每一个.exe 文件，很难被移除。最好方法是从一个备份恢复系统。也可以在恶意的 kerne132.dll 中删除恶意的内容。或者可以复制 kernel32.dll，并将它命名为 kerne132.dll；或者写 一个程序来取消所有对 PE 文件的修改。

Lab07-04 编写 Yara 规则

根据前面的分析，编写 Yara 规则如下：

```
rule Lab07_01 {
    meta:
        description = "Lab07_01.exe"
    strings:
        $s1 = "Malservice" fullword ascii
        $s2 = "http://www.malwareanalysisbook.com" fullword ascii
        $s3 = "Internet Explorer 8.0" fullword ascii
        $s4 = "HGL345" fullword ascii
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and filesize < 70KB and
        all of them
}
rule Lab07_02 {
    meta:
        description = "Lab07-02.exe"
    strings:
        $s1 = "http://www.malwareanalysisbook.com/ad.html" fullword wide
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and filesize < 50KB and
        all of them
}
rule Lab07_03dll {
```

```
    meta:
        description = "Lab07-03.dll"
    strings:
        $s1 = "SADFHUHF" fullword ascii
        $s2 = "127.26.152.13" fullword ascii
        $s3 = "hello" fullword ascii
        $s4 = "sleep" fullword ascii
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and filesize < 500KB and
        all of them
}
rule Lab07_03exe {
    meta:
        description = "Lab07-03.exe"
    strings:
        $s1 = "C:\\windows\\system32\\kerne132.dll" fullword ascii
        $s2 = "C:\\Windows\\System32\\Kernel32.dll" fullword ascii
        $s3 = "kerne132.dll" fullword ascii
        $s4 = "Lab07-03.dll" fullword ascii
        $s5 = "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE" fullword ascii
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and filesize < 50KB and
        all of them
}
```

扫描结果如下，能够验证 Yara 规则的正确性：

```
PS D:\NKU\23Fall\恶意代码分析与防治技术\yara-4.3.2-2150-win64> ./yara64 Lab07.yar Chapter_7L
Lab07_02 Chapter_7L\Lab07-02.exe
Lab07_03dll Chapter_7L\Lab07-03.dll
Lab07_03exe Chapter_7L\Lab07-03.exe
Lab07_01 Chapter_7L\Lab07_01.exe
```

四、Ida Python

编写脚本获取函数列表及起始地址：

```
import idaapi
import idautils


def list_functions():
```

```
    # 获取当前二进制文件的函数列表
    functions = idautils.Functions()

    for func_ea in functions:
        func_name = idc.get_func_name(func_ea)
        print(f"Function: {func_name} @ 0x{func_ea:08X}")

if __name__ == "__main__":
    list_functions()
```

运行结果如下：

```
Function: sub_401000 @ 0x00401000
Function: sub_401040 @ 0x00401040
Function: sub_401070 @ 0x00401070
Function: sub_4010A0 @ 0x004010A0
Function: sub_4011E0 @ 0x004011E0
Function: _main @ 0x00401440
Function: start @ 0x00401820
Function: _XcptFilter @ 0x00401930
Function: _initterm @ 0x00401936
Function: __setdefaultprecision @ 0x0040193C
Function: UserMathErrorFunction @ 0x0040194E
Function: nullsub_1 @ 0x00401951
Function: _except_handler3 @ 0x00401960
Function: _controlfp @ 0x00401966
```

编写脚本获得函数指令：

```
import idaapi
import idautils

# 指定要查找的函数名
target_function_name = "sub_00401040"

# 获取函数的起始地址
target_function_ea = idc.get_name_ea_simple(target_function_name)

if target_function_ea != idc.BADADDR:
    print(f"函数 {target_function_name} 的地址: {target_function_ea:X}")

    # 遍历函数中的指令并列出
    for ea in idautils.FuncItems(target_function_ea):
        disasm = idc.GetDisasm(ea)
        print(f"{ea:X}: {disasm}")
else:
```

```
    print(f"没有找到函数 {target_function_name}")
```

运行结果如下：

```
函数 sub_00401040 的地址: 401040
401040: mov     eax, [esp+arg_4]
401044: push    esi
401045: mov     esi, [esp+4+arg_0]
401049: push    eax
40104A: push    esi
40104B: call    sub_401000
401050: mov     ecx, eax
401052: add     esp, 8
401055: test    ecx, ecx
401057: jnz     short loc_40105B
401059: pop     esi
40105A: retn
40105B: mov     eax, [ecx+14h]
40105E: mov     edx, [ecx+0Ch]
401061: mov     ecx, [esp+4+arg_8]
401065: sub     eax, edx
401067: add     eax, esi
401069: pop     esi
40106A: add     eax, ecx
40106C: retn
```

编写脚本查找特定函数，分析其控制流，并识别其中的基本块和交叉引用：

```
import idaapi
import idautils
import idc

def analyze_function(function_name):
    # 查找二进制文件中的函数地址
    func_ea = idc.get_name_ea(idc.BADADDR, function_name)

    if func_ea == idc.BADADDR:
        print(f"函数 '{function_name}' 未找到")
        return

    # 分析函数的控制流
    f = idaapi.get_func(func_ea)
    if not f:
        print(f"无法获取函数 '{function_name}'")
        return

    print(f"分析函数 '{function_name}'，入口地址: 0x{func_ea:08X}")

    for block in idaapi.FlowChart(f):
```

```
            print(f"基本块: 0x{block.start_ea:08X}")

            # 使用 idautils.XrefsTo 来获取交叉引用
            for head in idautils.Heads(block.start_ea, block.end_ea):
                disasm = idc.GetDisasm(head)
                print(f"    0x{head:08X}: {disasm}")

                # 识别交叉引用
                for ref in idautils.XrefsTo(head):
                    print(f"        引用自: 0x{ref.frm}")

if __name__ == '__main__':
    target_function = "sub_4011E0"

    analyze_function(target_function)
```

运行结果如下：

```
分析函数 'sub_4011E0'，入口地址: 0x004011E0
基本块: 0x004011E0
    0x004011E0: mov     eax, [esp+arg_4]
        引用自: 0x4199247
        引用自: 0x4200459
    0x004011E4: sub     esp, 144h
        引用自: 0x4198880
    0x004011EA: cmp     eax, 7
        引用自: 0x4198884
    0x004011ED: push    ebx
        引用自: 0x4198890
    0x004011EE: push    ebp
        引用自: 0x4198893
    0x004011EF: push    esi
        引用自: 0x4198894
    0x004011F0: push    edi
        引用自: 0x4198895
    0x004011F1: jg      loc_401434
        引用自: 0x4198896
基本块: 0x004011F7
    0x004011F7: mov     ebp, [esp+154h+lpFileName]
        引用自: 0x4198897
    0x004011FE: lea     eax, [esp+154h+FindFileData]
        引用自: 0x4198903
    0x00401202: push    eax; lpFindFileData
        引用自: 0x4198910
    0x00401203: push    ebp; lpFileName
        引用自: 0x4198914
    0x00401204: call    ds:FindFirstFileA
        引用自: 0x4198915
```

五、实验结论及心得体会

本次实验分析恶意 Windows 程序，对恶意代码利用 windows 功能的方式的理解更为深刻；进一步熟练了恶意代码综合分析方法。