南開大學

恶意代码分析与防治技术实验报告

Lab7

学院：网络空间安全学院

专业：信息安全、法学

学号：2113203

姓名：付政烨

班级：信安法班

摘要

在本次实验中, 我们深入研究了几个恶意代码的行为并尝试分析它们的目的和特征。首先, 我们使用 IDA Pro 逆向分析了一个程序的导入表。从导入表中, 我们观察到从 Advapi32.dll 中导入的可疑函数, 这些与服务相关的函数暗示了恶意代码可能会利用服务控制管理器创建一个新服务。通过进一步的代码分析, 我们发现恶意代码确实尝试将自己安装为一个名为“MalService”的自启动服务，从而实现持久化驻留。此外，该恶意代码还使用了一个互斥量来确保同一时间只有一个其实例在运行。此外, 该程序还被设计为在 2100 年 1 月 1 日 0 点启动 20 个线程, 持续访问一个特定网站, 可能用于对目标网站进行 DDoS 攻击。接下来, 我们分析了第二个程序。这个程序在导入表中没有显著的持久化驻留函数。但是，它确实使用了组件对象模型 (COM) 功能来启动 Internet Explorer 并访问一个特定的 Web 地址。这暗示了该恶意代码可能是为了强制用户访问某个广告页面。最后，我们注意到了第三个实验中的恶意可执行程序和 DLL。这两个文件在受害者机器上的同一个目录中被发现，意味着它们可能是相互关联的。实验还警告说，这个实验可能会对计算机造成损害，并且可能安装后难以清除。实验的这一部分尤为具有挑战性，需要结合静态和动态分析技术，并保持对整体的关注，而不是仅仅陷入细节中。

关键字：持久化驻留；　逆向工程；　DDoS 攻击

目录

一、 实验目的

本实验的核心目的是深入探索恶意代码的行为及其内部机制。首先，通过逆向工程工具 IDA Pro，我们对代码的导入表、关键函数和调用关系进行了详细分析，揭示了恶意代码的潜在行为。其中，一个重要的焦点是恶意代码如何实现持久化：分析了它是否试图将自己注册为系统服务以在启动时自动运行，以及如何利用互斥量确保其唯一性运行。这种技巧可以避免资源争夺或触发异常。网络方面，我们研究了恶意代码与外部服务器或网站的通信特征，探索了其如何使用组件对象模型（COM）功能，特别是如何利用 Internet Explorer 进行网络活动。这种分析对于识别和阻止恶意流量至关重要。此外，考虑到恶意代码可能带来的潜在风险，实验特别强调了在受保护的环境中进行分析的重要性，推荐使用带有快照功能的虚拟机。最后，由于这是一个相对复杂的实验，它要求我们综合应用静态和动态分析方法，以获得恶意代码的全面和深入理解。

二、 实验原理

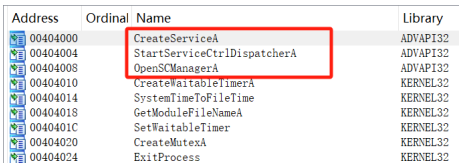
本次实验主要关注于逆向工程的技术，旨在深入分析并理解恶意代码的行为和功能。在第一个实验（Lab07-01）中，我们分析了一个恶意代码，它具备与服务管理器交互的能力，使其可以将自身设置为一个可自启动的服务，从而实现持久化驻留。此外，该代码使用互斥量确保同一时间只有一个其实例在运行，同时预计在 2100 年对特定 URL 进行 DDoS 攻击。第二个实验（Lab07-02）涉及另一个恶意代码，它通过 COM 技术和 IWebBrowser2 接口利用 Internet Explorer 访问特定的广告页面，但它没有实现任何持久化机制，仅在访问网页后即完成执行。第三个实验（Lab07-03）更为复杂，我们在执行前获得了一个恶意的可执行程序及其相关的 DLL。需要特别注意的是，这些恶意代码可能会连接到远程机器，但为了保护实验者，它被更改为连接至本地主机。整体而言，这些实验通过组合动态和静态分析方法，提供了对复杂恶意代码深入理解的机会，但也带来了潜在的风险，需要在受控的环境中进行。

三、 实验过程

(一) Lab06-01

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)

在使用 IDA 进行逆向工程分析时，我们注意到程序的导入表中有几个显眼的函数。这些函数均从 Advapi32.dll 中导入，具有潜在的恶意行为特征。其中三个函数（OpenSCManagerA、CreateServiceA 和 StartServiceCtrlDispatcherA）与 Windows 服务管理有关。基于 OpenSCManagerA 和 CreateServiceA 的存在，可以推断该代码可能使用服务控制管理器创建一个新服务。而 StartServiceCtrlDispatcherA 的作用是连接服务进程的主线程到服务控制管理器，这进一步证明了该代码意图作为一个服务运行。综合以上分析，该恶意代码可能采用的持久化策略是：注册自己为一个服务，并在创建服务时设置为自启动。



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

图 1

进一步分析代码，我们发现 main 函数首先调用了 StartServiceCtrlDispatcherA 函数，以实现服务功能。此函数的参数表明该恶意软件注册的服务名称为“MalService”，并指定了服务控制函数为 sub_401040。当调用 StartServiceCtrlDispatcherA 后，sub_401040 函数会被执行。

```
.text:00401000 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401000 _main           proc near               ; CODE XREF: start+AF↓p
.text:00401000
.text:00401000 ServiceStartTable= SERVICE_TABLE_ENTRYA ptr -10h
.text:00401000 var_8           = dword ptr -8
.text:00401000 var_4           = dword ptr -4
.text:00401000 argc            = dword ptr  4
.text:00401000 argv            = dword ptr  8
.text:00401000 envp            = dword ptr  0Ch
.text:00401000
.text:00401000                 sub     esp, 10h
.text:00401003                 lea     eax, [esp+10h+ServiceStartTable]
.text:00401007                 mov     [esp+10h+ServiceStartTable.lpServiceName], offset aMalservice ; "MalService"
.text:0040100F                 push    eax             ; lpServiceStartTable
.text:00401010                 mov     [esp+14h+ServiceStartTable.lpServiceProc], offset sub_401040
.text:00401018                 mov     [esp+14h+var_8], 0
.text:00401020                 mov     [esp+14h+var_4], 0
.text:00401028                 call    ds:StartServiceCtrlDispatcherA
.text:0040102E                 push    0
.text:00401030                 push    0
.text:00401032                 call    sub_401040
.text:00401037                 add     esp, 18h
.text:0040103A                 retn
.text:0040103A _main           endp
```

图 2

在 sub_401040 函数内，代码首先处理与互斥量相关的操作，接着使用 OpenSCManager 打开服务控制管理器的句柄。之后，它调用 GetCurrentProcess 获取当前进程的伪句柄，并使用这个伪句柄调用 GetModuleFileName 函数，从而获取该恶意代码的完整路径名。最终，这个路径名被传递给 CreateServiceA 函数，将恶意代码注册为名为“Malservice”的服务。值得注意的是，CreateServiceA 函数的参数 dwStartType 被设置为 2（SERVICE_AUTO_START），确保服务在启动时自动运行。这样，即使系统重启，该恶意服务也会自动运行，实现持久化驻留。

```
.text:00401064 loc_401064:                             ; CODE XREF: sub_401040+1A↑j
.text:00401064                 push    esi
.text:00401065                 push    offset Name     ; "HGL345"
.text:0040106A                 push    0               ; bInitialOwner
.text:0040106C                 push    0               ; lpMutexAttributes
.text:0040106E                 call    ds:CreateMutexA
.text:00401074                 push    3               ; dwDesiredAccess
.text:00401076                 push    0               ; lpDatabaseName
.text:00401078                 push    0               ; lpMachineName
.text:0040107A                 call    ds:OpenSCManagerA
.text:00401080                 mov     esi, eax
.text:00401082                 call    ds:GetCurrentProcess
.text:00401088                 lea     eax, [esp+404h+Filename]
.text:0040108C                 push    3E8h            ; nSize
.text:00401091                 push    eax             ; lpFilename
.text:00401092                 push    0               ; hModule
.text:00401094                 call    ds:GetModuleFileNameA
.text:0040109A                 push    0               ; lpPassword
.text:0040109C                 push    0               ; lpServiceStartName
.text:0040109E                 push    0               ; lpDependencies
.text:004010A0                 push    0               ; lpdwTagId
.text:004010A2                 lea     ecx, [esp+414h+Filename]
.text:004010A6                 push    0               ; lpLoadOrderGroup
.text:004010A8                 push    ecx             ; lpBinaryPathName
.text:004010A9                 push    0               ; dwErrorControl
.text:004010AB                 push    2               ; dwStartType
.text:004010AD                 push    10h             ; dwServiceType
.text:004010AF                 push    2               ; dwDesiredAccess
.text:004010B1                 push    offset DisplayName ; "Malservice"
.text:004010B6                 push    offset DisplayName ; "Malservice"
.text:004010BB                 push    esi             ; hSCManager
.text:004010BC                 call    ds:CreateServiceA
.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
```

图 3

2. 为什么这个程序会使用一个互斥量?

在先前的分析中，我们观察到与互斥量相关的代码位于 sub_401040 子函数的起始位置。这个互斥量是一个全局对象，且其标识符为“HGL345”。恶意代码首先调用 OpenMutexA 函数，试图访问该互斥量。如果访问未成功，函数会返回值 0，这时，jz 指令会导致代码跳转至 loc_401064，在那里会调用 CreateMutexA 函数以创建名为“HGL345”的互斥量。若互斥量的打开成功，这

意味着一个恶意代码实例已在运行且已创建了该互斥量，因此会调用 ExitProcess 函数来终止当前进程。这样的设计确保了在同一时刻，只有一个恶意代码实例在该计算机上运行。

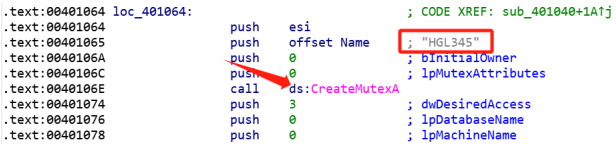


图 4

3 可以用来检测这个程序的基于主机特征是什么?

在前两个问题的研究分析基础上，我们发现名为“Malservice”的服务和名为“HGL345”的互斥量均可作为有效的特征标识。

4. 检测这个恶意代码的基于网络特征是什么?

在导入表中，我们观察到与网络相关的函数。此恶意代码从 WinINet.dll 库中导入了 InternetOpenUrlA 和 InternetOpenA 两个函数。InternetOpenA 函数的主要功能是初始化一个到互联网的连接，而 InternetOpenUrlA 函数的功能是访问特定的 URL。

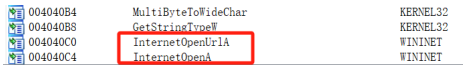


图 5

为了进一步理解 InternetOpenA 函数的实现，可以使用 Ctrl+X 快捷键来查看它的交叉引用信息。

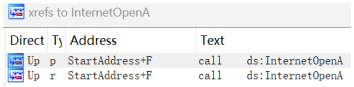


图 6

值得注意的是，这两个函数的交叉引用实际上指向了同一处代码。通过双击该引用，我们可以跳转到该引用的位置，并发现 InternetOpenA 和 InternetOpenUrlA 的调用都位于 StartAddress 的子函数中。

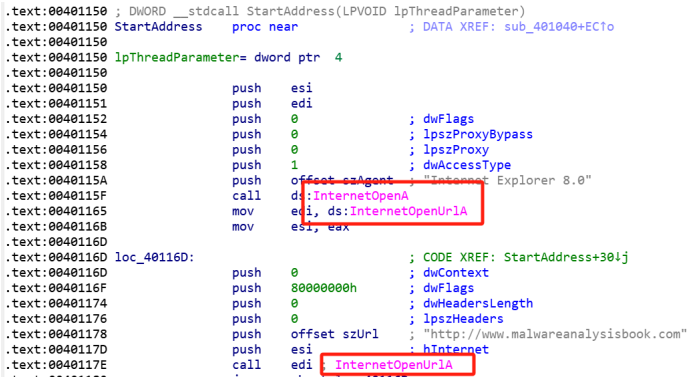


图 7

进一步的代码分析显示，InternetOpenA 函数的 szAgent 参数表明使用的代理服务器是 Internet Explorer 8.0。而 InternetOpenUrlA 函数尝试访问的地址是 http://www.malwarenanlysisbook.com。基于这两个特点，我们可以确定恶意代码的网络行为特征。

5. 这个程序的目的是什么?

根据先前的代码分析，该恶意软件初始调用了 StartServiceCtrlDispatcherA 函数，随后调用了 sub_401040 子函数。在这个子函数中，通过互斥量确保仅有一个该恶意软件实例正在执行，并将其配置为一个自启动的服务。我们接下来将探讨程序的后续部分。

```
.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
```

图 8

该恶意软件首先调用了 SystemTimeToFileTime 函数，用于将时间从系统时间格式转换为文件时间格式。此函数的参数是待转换的时间，同时，我们可以通过 IDA Pro 看到一个 SystemTime 结构体。

```
.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
.text:0040110F                push    esi             ; hHandle
.text:00401110                call    ds:WaitForSingleObject
.text:00401116                test    eax, eax
.text:00401118                jnz     short loc_40113B
.text:0040111A                push    edi
.text:0040111B                mov     edi, ds:CreateThread
.text:00401121                mov     esi, 14h
```

图 9

该代码将已经被清零的 edx 值分别赋给 wYear、wDayOfWeek、wHour 和 wSecond，代表年、日、时、秒。随后，它将 wYear 设置为 834h，代表 2100 年。

```
.text:0040113B loc_40113B:                            ; CODE XREF: sub_401040+D81j
.text:0040113B                push    0FFFFFFFFh      ; dwMilliseconds
.text:0040113D                call    ds:Sleep
.text:00401143                xor     eax, eax
.text:00401145                pop     esi
.text:00401146                add     esp, 400h
.text:0040114C                retn
.text:0040114C sub_401040     endp
```

图 10

代码继续执行，将上述的时间点转换为文件时间格式后，首先调用了 CreateWaitableTimerA 函数以创建一个定时器对象。接着，它调用 SetWaitableTimer 函数来设置该定时器，其中参数 lpDueTime 是前面转换得到的文件时间结构体。最后，它调用 WaitForSingleObject 函数，等待定时器对象变为有信号状态，或等待时间达到 FFFFFFFFh 毫秒（这是一个显然不可能达到的时间长度），意味着它会等待到 2100 年 1 月 1 日 0 点。若 WaitForSingleObject 函数因定时器

对象变为有信号状态而返回，其返回值为 0。如果出现错误、拥有互斥量的线程结束但未释放定时器对象、或等待时间达到指定毫秒，返回值则为非 0。在异常情况下，代码会根据 eax 的值决定是否执行到 loc_40113B，并进入一个长达 FFFFFFFFh 毫秒的休眠状态。若定时器成功触发，代码将继续执行。

```
.text:00401118                jnz     short loc_40113B
.text:0040111A                push    edi
.text:0040111B                mov     edi, ds:CreateThread
.text:00401121                mov     esi, 14h
.text:00401126
.text:00401126 loc_401126:                            ; CODE XREF: sub_401040+F8↓j
.text:00401126                push    0               ; lpThreadId
.text:00401128                push    0               ; dwCreationFlags
.text:0040112A                push    0               ; lpParameter
.text:0040112C                push    offset StartAddress ; lpStartAddress
```

图 11

```
.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
```

图 12

接下来的部分是一个循环，循环次数为 14h（即 20 次）。在每次循环中，它都会创建一个线程来执行 StartAddress 子函数。根据先前的分析，StartAddress 函数将作为 Internet Explorer 8.0 的代理服务器无限次地访问 http://www.malwarenanlysisbook.com。当这个循环结束后，代码将按照执行顺序进入 loc_40113B，并休眠 FFFFFFFFh 毫秒。

总结来说，这段恶意代码的目的是配置自身为一个自启动服务，确保只要计算机启动，它就会运行。然后，它会等到 2100 年 1 月 1 日 0 点，随后启动 20 个线程，每个线程都会无限次地访问 http://www.malwarenanlysisbook.com。这可能是一种针对指定网站的 DDoS 攻击方法。

6. 这个程序什么时候完成执行?

在先前的分析中，我们已经确定每个线程都会持续不断地访问目标网址。因此，此程序将无法正常终止执行。

(二) Lab06-02

1. 这个程序如何完成持久化驻留?

在使用 IDA Pro 进行逆向分析时，通过观察程序的导入表，我们未能识别与注册表、服务或其他可能用于实现持久化驻留的功能相关的函数。

Address	Ordinal	Name	Library
00402000		__getmainargs	MSVCRT
00402004		_controlfp	MSVCRT
00402008		_except_handler3	MSVCRT
0040200C		__set_app_type	MSVCRT
00402010		__p__fmode	MSVCRT
00402014		__p__commode	MSVCRT
00402018		_exit	MSVCRT
0040201C		_XcptFilter	MSVCRT
00402020		exit	MSVCRT
00402024		__p___initenv	MSVCRT
00402028		_initterm	MSVCRT
0040202C		__setusermatherr	MSVCRT
00402030		_adjust_fdiv	MSVCRT
00402038	8	VariantInit	OLEAUT32
0040203C	2	SysAllocString	OLEAUT32
00402040	6	SysFreeString	OLEAUT32
00402048		OleInitialize	ole32
0040204C		CoCreateInstance	ole32
00402050		OleUninitialize	ole32

图 13

进一步检查函数列表和相关代码后，也没有找到持久化驻留的代码段。因此，可以推断该程序并未实现持久化驻留功能，而是执行一次后即终止。

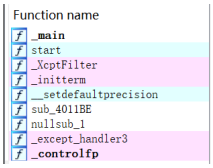


图 14

2 这个程序的目的是什么?

在程序的 main 函数中，首先执行了 OleInitialize 函数，这表示该代码意图使用组件对象模型 (COM) 的功能。随后，通过 CoCreateInstance 函数，程序尝试获得对 COM 功能的访问权限，并进一步检查了 rclsid 和 riid。

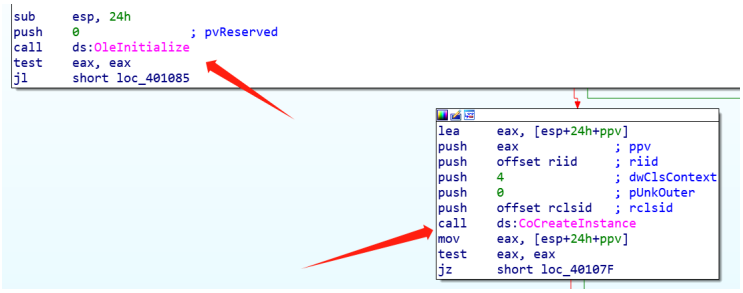


图 15

我们可以识别出，类型标识符 CLSID 的值为 0002DF01-0000-0000-C000-000000000046，这代表 Internet Explorer 的标识。

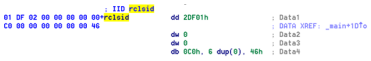


图 16

同时，接口标识符 IID 的值为 D30C1661-CDAF-11D0-8A3E-00C04FC9E26E，这对应于 IWebBrowser2 接口。

图 17

继续分析代码，它首先使用 VariantInit 函数来初始化变量。接着，通过 SysAllocString 函数，为字符串”http://www.malwareanalysisbook.com/ad.html” 分配了内存空间。ppv 指向 COM 对象的位置。而指令 mov edx,[eax] 确保 edx 指向 COM 对象的基地址。接下来，指令 call dword ptr [edx+2Ch] 调用了位于 IWebBrowser2 接口偏移地址 0x2Ch (即 44) 的函数。考虑到每个函数地址占 4 字节，这实际上是调用了序号 11，也就是第 12 个函数。这个函数被识别为 Navigate 函数，其功能是允许程序启动 Internet Explorer 并导航到一个指定的 Web 地址。在此情境下，这个特定的地址是”http://www.malwareanalysisbook.com/ad.html”，这一地址已经被存储在之前通过 SysAllocString 分配的内存空间中。

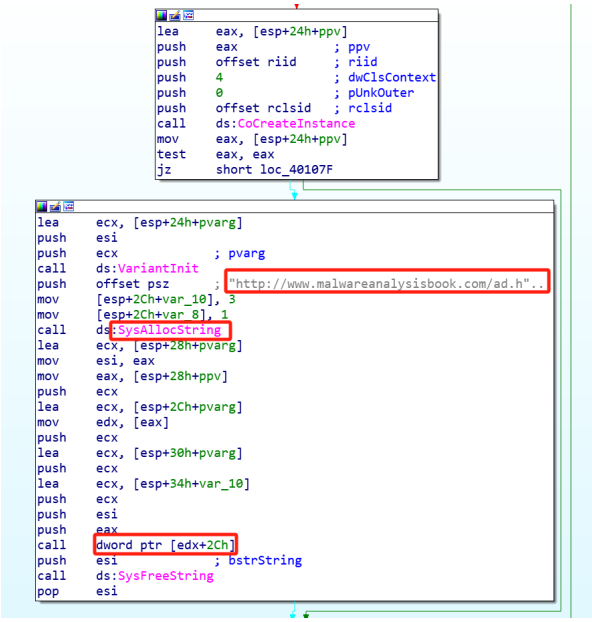


图 18

在导航到上述网址后，程序正常终止。从此行为，我们可以推断出，该代码的主要目的是导航到一个特定的 Web 地址。根据 URL 中的”ad” 子串，这可能是一个广告页面。但是，当前尝试访问这个网址会收到一个 404 错误，意味着该页面已经不存在。

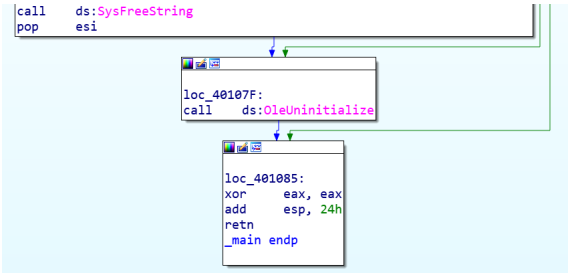


图 19

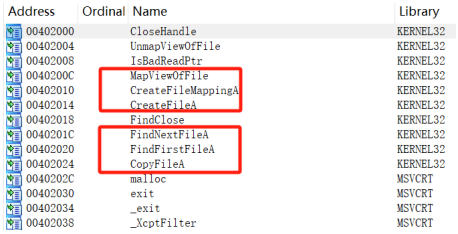
3. 这个程序什么时候完成执行?

在先前的分析中，我们观察到该恶意代码在访问了一个广告页面后便正常终止其执行，且未观察到其他异常行为。

(三) Lab06-03

1. 这个程序如何完成持久化驻留，来确保在计算机被重启后它能继续运行?

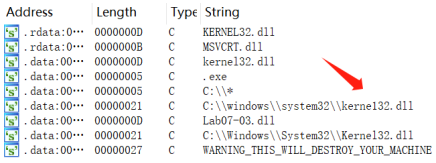
首先，我们对目标 exe 文件采用 IDA Pro 工具进行了静态分析，并详细观察了其导入表。在此过程中，我们没有发现与注册表或服务相关的函数。然而，我们注意到了几个与文件操作相关的函数，例如 FindFirstFileA 和 FindNextFileA。这些函数的存在暗示该代码可能被设计为遍历指定目录以查找文件。此外，CopyFileA 函数的存在表明该代码可能会复制找到的文件，可能会更改其位置或名称。通过 CreateFileA、CreateFileMappingA 和 MapViewOfFile 函数，我们推断恶意代码可能会打开一个文件并将其映射到内存中。然而，值得注意的是，该代码并未导入 Lab07-03.dll 及其相关函数，这意味着这个 dll 文件并不是为 exe 文件提供专门的功能。



Address	Ordinal	Name	Library
00402000		CloseHandle	KERNEL32
00402004		UnmapViewOfFile	KERNEL32
00402008		IsBadReadPtr	KERNEL32
0040200C		MapViewOfFile	KERNEL32
00402010		CreateFileMappingA	KERNEL32
00402014		CreateFileA	KERNEL32
00402018		FindClose	KERNEL32
0040201C		FindNextFileA	KERNEL32
00402020		FindFirstFileA	KERNEL32
00402024		CopyFileA	KERNEL32
0040202C		malloc	MSVCRT
00402030		exit	MSVCRT
00402034		_exit	MSVCRT
00402038		_XcptFilter	MSVCRT

图 20

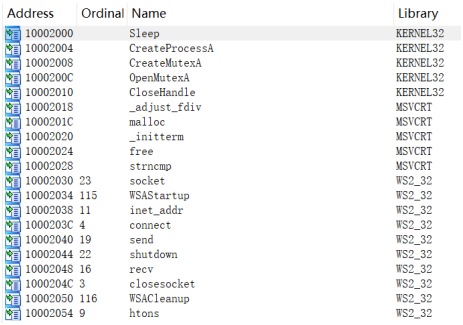
在进一步观察字符串信息时，我们发现了一个尤其引人注目的字符串：“kerne132.dll”。该字符串对“kernel32.dll”中的字母“l”进行了替换，将其变为了数字“1”。这明显是一种伪装手段，旨在掩盖恶意代码的文件名，以避免被检测。在其附近，我们还发现了字符串“Lab07-03.dll”。结合上述导入函数的分析和路径字符串 `"C:\windows\system32\kerne132.dll"`，我们推测该代码可能会遍历当前目录，找到 `Lab07-03.dll` 文件，然后将其复制到 `C:\windows\system32\`下，并将其重命名为 `kerne132.dll`。这种策略与我们在 `lab1` 中观察到的相似，暗示该代码可能利用名字相似性来进行混淆。



Address	Length	Type	String
.rdata:0…	0000000D	C	KERNEL32.dll
.rdata:0…	0000000B	C	MSVCRT.dll
.data:00…	0000000D	C	kernel32.dll
.data:00…	00000005	C	.exe
.data:00…	00000005	C	C:*
.data:00…	00000021	C	C:\\windows\\system32\\kernel32.dll
.data:00…	0000000D	C	Lab07-03.dll
.data:00…	00000021	C	C:\\Windows\\System32\\Kernel32.dll
.data:00…	00000027	C	WARNING_THIS_WILL_DESTROY_YOUR_MACHINE

图 21

进一步分析 Lab07-03.dll 的导入函数表时，我们注意到从 kernel32.dll 中导入了 CreateProcessA 以及与互斥量相关的函数。此外，它还按序号导入了 ws2_32.dll 中的函数，这是 Winsock 库，其内部的函数与网络通信相关。因此，我们可以推测该 dll 文件涉及网络访问操作。



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

图 22

在进一步查看该 dll 的字符串信息时，我们发现了”exec”、”sleep” 以及”127.26.152.13” 等关键字符串。其中，”exec” 和”sleep” 很可能是两个功能函数，分别用于执行命令和休眠。而”127.26.152.13” 显然是一个 IP 地址。结合之前的分析，我们推测该 dll 可能会与此 IP 地址进行通信。

Address	Length	Type	String
.rdata:1000214E	0000000D	C	KERNEL32.dll
.rdata:1000215C	0000000B	C	WS2_32.dll
.rdata:10002172	0000000B	C	MSVCRT.dll
.data:10026010	00000005	C	exec
.data:10026018	00000006	C	sleep
.data:10026020	00000006	C	hello
.data:10026028	0000000E	C	127.26.152.13
.data:10026038	00000009	C	SADFHUHF

图 23

然而，要准确分析该恶意代码如何实现持久化驻留，我们还需进一步研究其具体的代码实现。

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

var_44= dword ptr -44h
var_40= dword ptr -40h
var_3C= dword ptr -3Ch
var_38= dword ptr -38h
var_34= dword ptr -34h
var_30= dword ptr -30h
var_2C= dword ptr -2Ch
var_28= dword ptr -28h
var_24= dword ptr -24h
var_20= dword ptr -20h
var_1C= dword ptr -1Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_10= dword ptr -10h
var_C= dword ptr -0Ch
hObject= dword ptr -8
var_4= dword ptr -4
argc= dword ptr  4
argv= dword ptr  8
envp= dword ptr  0Ch

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

图 24

在主函数的开始，程序首先对 argc 的值与 2 进行比对，以验证执行该可执行文件时的命令行参数是否恰为两个。若不满足此条件，则程序会跳转至 loc_401813 位置并立即终止执行。

```
loc_401813:
pop     edi
pop     esi
pop     ebp
xor     eax, eax
pop     ebx
add     esp, 44h
retn
_main endp
```

图 25

随后，程序获取命令行参数的地址列表，记为 argv[]。通过地址偏移 [eax+4]（每个地址长度为 4 字节）将 argv[1]（即第二个命令行参数）的值存入 eax 寄存器。同时，程序将字符串”WARNING_THIS_WILL_DESTROY_YOUR_MACHINE” 存入 esi 寄存器，并可能后续对这两个寄存器中的值进行操作。

```
push    esi
push    edi
jnz     loc_401813

mov     eax, [esp+54h+argv]
mov     esi, offset aWarningThisWil ; "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE"
mov     eax, [eax+4]

loc_401460:
mov     dl, [eax]
```

图 26

在 loc_401460 位置，程序中有一个循环用于比较 eax 和 esi 寄存器中的值。具体地说，它验证传入的第二个命令行参数是否与字符串”WARNING_THIS_WILL_DESTROY_YOUR_MACHINE” 匹配。如果两者不匹配，程序将跳转至 loc_401488 并执行一系列操作使 eax 的值非零。若两者匹配，程序将跳转至 loc_401484 并将 eax 的值设为零。

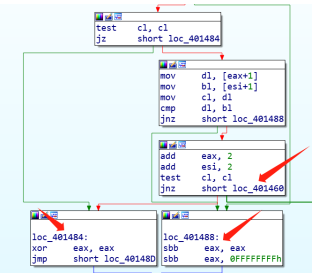


图 27

不论两者是否匹配，程序最终都将进入 loc_40148D。在此位置，程序首先检查 eax 的值是否为零。若其值非零，表示传入的参数与指定字符串不匹配，程序会跳转至 loc_401813 并终止执行。因此，为了使该恶意程序正常执行，用户需要在命令行中输入如下命令：[Lab07-03.exe 路径] WARNING_THIS_WILL_DESTROY_YOUR_MACHINE，并在一个虚拟机环境中进行测试。后续的分析需考察在满足上述条件下，恶意代码的进一步行为。

图 28

该恶意代码使用 CreateFileA、CreateFileMappingA 和 MapViewOfFile 函数来打开”Kernel32.dll” 文件，并将其映射到内存中。使用相同的方法，程序还打开并映射了”Lab07-03.dll” 到内存中。

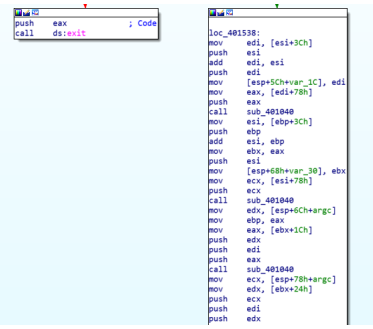


图 29

从地址 loc_401538 开始，我们观察到一系列的 mov 和 push 指令。尽管这些指令的具体目的不易于直接解读，但值得注意的是，这些 mov 和 push 指令之间，存在几次 call 指令的执行。在每一组 mov 和 push 指令后，都伴随着对 sub_401040 的调用。此外，在连续的调用之后，还有对 sub_401070 的调用。接下来，我们将对这两个函数进行深入的探讨。

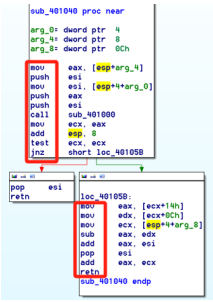


图 30

这两个函数主要执行了一系列内存操作。结合之前的恶意代码，它打开并将两个 dll 文件映射到内存中，我们推测这些操作可能是对这两个 dll 文件的处理。

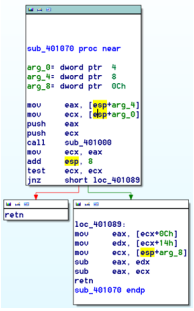


图 31

继续分析后，我们发现还有大量的内存操作，直至达到地址 `loc_4017D4`。从此处开始，出现了 Windows API 的调用以及一些关键的字符串信息。从先前的操作中，我们知道 `hObject` 和 `var_4` 保存了两个文件的句柄，由此推断，此处的两次 `CloseHandle` 调用的目的是关闭这两个文件句柄。这可能意味着恶意代码已经完成了文件内存映射的修改，并将其保存回文件。接着是 `CopyFileA` 函数的调用，从其参数中我们可以推断，它是将 `Lab07-03.dll` 复制到 `C:\windows\system32\` 并重新命名为 `kerne132.dll`。

图 32

在地址 loc_401806，我们注意到了另一个关键的函数调用，即对 sub_4011E0 子函数的调用，其参数为字符串“C:*”。对该子函数的深入分析显示，它首先调用了 FindFirstFile 函数，在 C:\下搜索第一个文件或目录。接着是一系列比较和算术运算指令，其目的不易解读。但在这些代码中，我们发现了可能的两次 malloc 函数调用和一次可能的 sub_4011E0 调用，这意味着这个函数可能有递归调用的情况。最后，值得关注的是一次 stricmp 函数的调用，其参数为字符串“.exe”和 FindFirstFile 函数返回的 FindFileDta 结构中的 dwReserved1 字段。接下来的逻辑是，如果这两个字符串相同，则调用 sub_4010A0 函数。

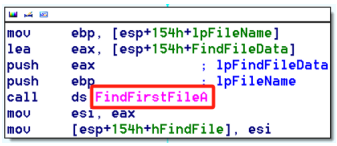


图 33

FindFirstFile 和 FindNextFile 函数结合使用可以遍历目录。在调用了 FindNextFileA 函数后，只要其返回值不为 0（即目录遍历未完成），程序就会跳转回到刚调用完 FindFirstFileA 的位置，loc_401210。

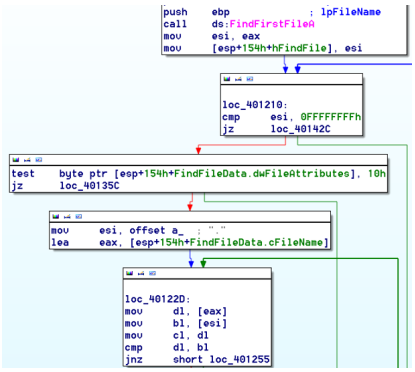


图 34

综上所述，sub_4011E0 函数的主要目的是遍历 C:\目录，搜索.exe 文件。每当发现一个.exe 文件时，它都会调用 sub_4010A0 函数。之前提到的可能的递归调用是因为存在子目录，当遍历到一个子目录时，会递归调用 sub_4011E0。

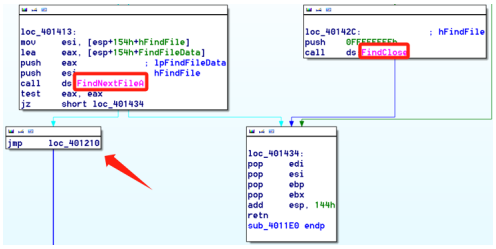


图 35

最后，我们需要进一步探索 sub_4010A0 函数的功能和目的。首先，观察到了一系列的 Windows API 函数调用，包括 CreateFileA、CreateFileMappingA 和 MapViewOfFile。这些函数被用于根据传入的文件路径参数将文件打开并映射到内存。基于此，存在一定的可能性，恶意代码会直接操作这块映射的内存以修改文件内容，而非通过其他的 Windows API。这种操作策略可能会加大分析者对于恶意代码具体修改行为的解析难度。

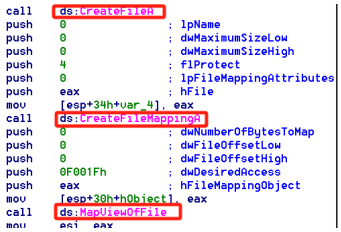


图 36

接下来，观察到了 IsBadReadPtr 函数被调用了四次。该函数的主要目的是检查进程是否具有访问指定内存块的权限，即验证指针的合法性。

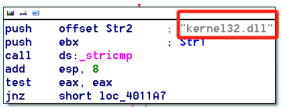


图 37

之后，存在一个调用 stricmp 的实例，该函数用于比较一个通过内存地址和偏移获取的字符串与”kernel32.dll” 是否相同。这种从内存中提取字符串的方法可能使得具体的字符串位置难以确定。

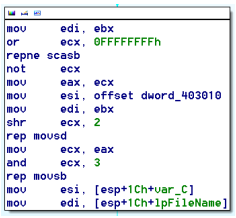


图 38

在 API 调用之外，识别到了三个特殊的指令：repne scasb、rep movsd 和 rep movsb。repne scasb 指令与 00401183 至 0040118A 的代码段结合，用于计算字符串的长度。此处，or 指令被用

于设置循环次数为-1，而 repne scasb 则持续搜索直至 edi 地址指向的字符串结束符。最终，eax 保存了字符串的长度，而 ebx 保存了名为 Str1 的字符串的地址。

图 39

对于 rep movsd 指令，它与 0040118C 至 00401196 的代码段结合，用于将 dword_403010 位置的 ecx 个 dword 复制到 ebx 保存的地址。对于该复制的内容，经过十六进制到字符串的转换，我们可以看到它是”kerne132.dll”。基于此，与之前的分析相结合，可以推测该代码段的目的是将一个.exe 文件中的”kernel32.dll” 字符串替换为”kerne132.dll”。

图 40

然而，rep movsb 指令在此上下文中似乎没有明确的功能。最后，存在一个重要的向上跳转指令。这可能意味着存在一个循环结构。进一步分析显示，跳转的目标点位于最后一个 IsBadReadPtr 函数调用之前。如果 IsBadReadPtr 检测到非法指针，则会跳出循环；如果指针合法，则会执行 stricmp 来比较字符串。这些行为暗示该子函数可能遍历映射到内存中的文件内容，寻找”kernel32.dll” 字符串并将其替换为”kerne132.dll”。函数的最后部分则涉及关闭映射和句柄，以及进行函数返回前的清理工作。

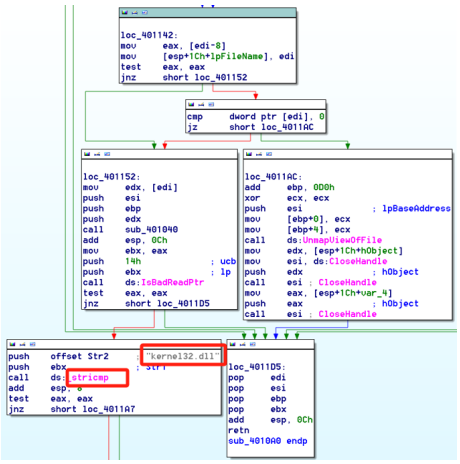


图 41

综合考虑，对于该软件的功能和相关操作可以概述如下：该程序首先复制“Lab07-03.dll”到“C:\Windows\System32\”目录下，并对其进行重命名，命名为“kerne132.dll”。随后，程序对 C 盘下的所有文件进行扫描，识别出“.exe”后缀的文件，并将文件内容中的“kernel32.dll”字符串替换为“kerne132.dll”。一般情况下，当“.exe”文件中存在“kernel32.dll”字符串时，意味着此文件试图导入“kernel32.dll”中的函数。因此，通过上述操作，当 C 盘下的“.exe”文件试图导入“kernel32.dll”的函数时，实际上会加载“kerne132.dll”。这种操作策略即为该恶意代码实现在系统中持续存在的手段。

2. 这个恶意代码的两个明显的基于主机特征是什么?

根据上一问的分析结果,我们观察到一个显著特征,即存在一个硬编码的文件名”kerne132.dll”。在 Lab07-03.exe 中，除此之外，未发现其他明确的基于主机的特征。然而，通过审查 dll 文件的导入表，我们发现导入了与互斥量相关的函数。互斥量的命名通常采用硬编码方式，这可能为我们提供一个有意义的特征点，进一步研究相关代码。

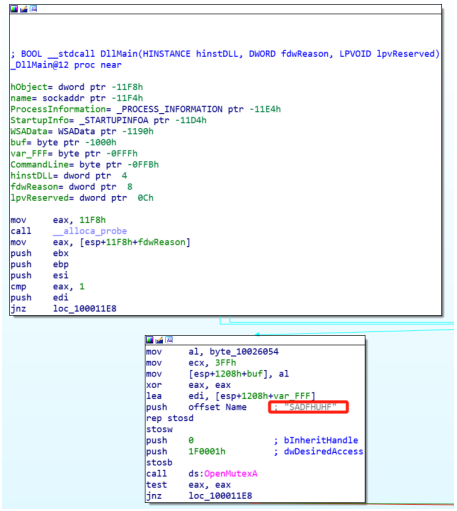


图 42

在 Lab07-03.dll 的 DllMain 函数中，我们首先注意到了互斥量相关函数的使用。此函数试图打开（或创建）一个名为“SADFHUHF”的硬编码命名的互斥量，这是基于主机的另一个显著特征。

3. 这个程序的目的是什么?

基于之前的分析，Lab07-03.exe 主要是为了持久化驻留而设计，并起到一个辅助的功能。核心功能应该是在 Lab07-03.dll 中实现。接下来，我们将对相关的互斥量代码进行进一步的分析。

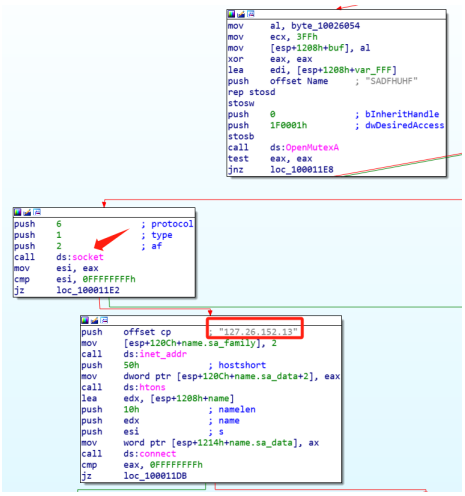


图 43

在确认只有一个 Lab07-03.dll 恶意代码实例运行之后，该程序立即调用 WSAStartup 函数。

为了使用 ws2_32.dll（即 Winsock 库）中的函数，必须先调用 WSAStartup 函数，以初始化 Win32 sockets 系统。这提示我们该程序可能接下来要使用 Winsock API。进一步分析显示，该程序首先调用 socket 函数以创建套接字。然后，它识别了一个 IP 地址字符串“127.26.152.13”并使用 inet_addr 函数将其转换为一个无符号长整型数。在调用 htons 函数之前，它指定了端口号为 0x50（十进制 80），并使用该函数将整型端口号从主机字节顺序转换为网络字节顺序。最终，它使用 connect 函数与 127.26.152.13 上的远程套接字建立连接。

图 44

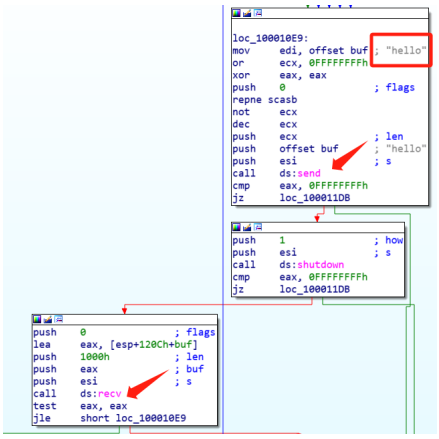


图 45

程序接下来检查了函数的返回值，并将 strncmp 和 CreateProcessA 函数的地址分别分配给了 ebp 和 ebx 寄存器。但在此之前，并未立即调用这两个函数。我们可以看到，buf 变量存储了字符串“hello”，随后通过 send 函数将该字符串发送到远程服务器。在发送完毕后，它使用 recv 函数从远程套接字接收数据，并将其存储在 buf 中。

图 46

程序进一步检查接收到的数据：如果数据的前五个字符是“sleep”，则程序会调用 Sleep 函数，使其休眠 0x60000 毫秒，然后跳转到 loc_100010E9。如果不是，程序则跳转到 loc_10001161。

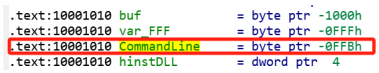


图 47

在另一部分逻辑中，如果数据的开头不是“sleep”，程序会检查数据的前四个字符是否为“exec”。如果是，则会调用 CreateProcessA 函数创建一个新的进程。在调用此函数时，最关键的参数是 lpCommandLine，它指定了要创建的进程的文件路径。但在分析中，我们并没有找到 CommandLine 在哪里被赋值。然而，CommandLine 在内存中的位置位于 buf 的起始地址之后的 5 个字节处。如果远程服务器发送的指令格式为“exec [FilePath]”，则 CommandLine 将存储 [FilePath]，从而使整个逻辑连贯。

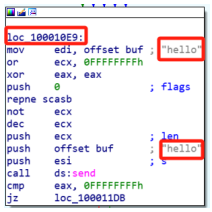


图 48

只要接收到的指令以“sleep”或“exec”开始，恶意代码都会执行相应的操作然后跳转到 loc_100010E9，然后再次发送“hello”并等待远程指令。如果都不符合上述条件，它会跳转到 loc_100011B6，检查接收到的指令是否为字符“q”。如果是，则跳转到 loc_100011D0，关闭所有资源并结束进程。否则，它会休眠 0x60000 毫秒，然后跳转到 loc_100010E9，开始新一轮的操作。

图 49

综上所述，Lab07-03.exe 的主要任务是安装 Lab07-03.dll 后门 dll 文件，并将所有 C 盘下的 exe 文件链接到这个 dll 文件。此后门会连接远程服务器并接收指令，它可以被指示进行休眠或创建新的进程。

4. 一旦这个恶意代码被安装，你如何移除它？

首先，需要移除 kerne132.dll 文件。但是，C 盘下的所有.exe 文件都与 kerne132.dll 建立了链接。为了解决这个问题，可以编写一个脚本来遍历 C 盘中的所有.exe 文件并进行相应的修改。然而，为了确保这些.exe 文件在当前计算机上正常运行，一个更为简便的方法是复制 kernel32.dll 文件并将其重命名为 kerne132.dll。

四、 Yara 规则编写

(一) Lab06-01.exe

```
rule Lab07_01 {
   meta:
      description = "Detects Lab07_01.exe malware"

   strings:
      $malware_url = "http://www.malwareanalysisbook.com" fullword ascii
      $malservice_lowercase = "Malservice" fullword ascii
      $malservice_propercase = "MalService" fullword ascii
      $internet_explorer_version = "Internet Explorer 8.0" fullword ascii
      $obfuscated_string1 = "YYh P@" fullword ascii
      $obfuscated_string2 = "HGL345" fullword ascii

   condition:
      uint16(0) == 0x5a4d and
      uint32(uint32(0x3c)) == 0x00004550 and
      filesize < 70KB and
      all of them
}
```

元数据:

- 这部分提供了关于规则的一些描述性信息。在这里，我们只有一个描述 description，告诉我们这个规则是为了检测 Lab07_01.exe 恶意软件。

字符串:

- 这部分定义了一系列的字符串模式。如果这些模式在文件中被匹配到，那么这个文件很可能是我们要找的恶意软件。
- 例如，$malware_url 检测是否包含特定的恶意 URL，而 $internet_explorer_version 则检测文件中是否有“Internet Explorer 8.0”这样的特定字符串。

条件:

- 文件的开头两个字节为 0x5a4d（这通常表示一个 Windows 可执行文件）。
- 在文件的某个位置匹配到 0x00004550（这可能是另一个标志表示 Windows 可执行文件）。
- 文件大小小于 70KB。
- 上述定义的所有字符串都在文件中出现。

(二) Lab06-02.exe

```
rule Lab07_02 {
   meta:
      description = "Detection rule for Lab07-02.exe malware sample"

   strings:
      $malicious_url = "http://www.malwareanalysisbook.com/ad.html" fullword wide
      $app_type_string = "set app type" fullword ascii
```

```
8
9     condition:
10       is_PE() and
11       filesize < 50KB and
12       ( $malicious_url or $app_type_string )
13 }
```

元数据：

- 这部分提供了关于规则的额外信息。
- 例如，description 描述了这个规则是为了检测哪个特定的恶意软件样本。

字符串：

- $malicious_url:这个字符串匹配了一个恶意 URL,即 http://www.malwareanalysisbook.com/ad.html。
- $app_type_string：这个字符串匹配了一个特定的 ASCII 文本，即 set app type。

条件：

- is_PE()：检查文件是否为 PE（Portable Executable）格式，这是 Windows 上常见的执行文件格式。
- filesize < 50KB：检查文件大小是否小于 50KB。
- ($malicious_url or $app_type_string)：检查文件中是否至少包含上述定义的其中一个字符串。

(三) Lab06-03.exe

```
1  rule Lab07_03_exe {
2     meta:
3        description = "Detects Lab07-03.exe malware"
4
5     strings:
6        $path_fake_kernel = "C:\\windows\\system32\\kerne132.dll" fullword ascii
7        $path_real_kernel = "C:\\Windows\\System32\\Kernel32.dll" fullword ascii
8        $filename_fake_kernel = "kerne132.dll" fullword ascii
9        $filename_malware_dll = "Lab07-03.dll" fullword ascii
10       $real_kernel_prefix = "Kernel32." fullword ascii
11       $destructive_string = "WARNING_THIS_WILL_DESTROY_YOUR_MACHINE" fullword ascii
12
13    condition:
14       uint16(0) == 0x5a4d and
15       uint32(uint32(0x3c)) == 0x00004550 and
16       filesize < 50KB and
17       1 of ($path*) and all of them
18 }
```

元数据：

- description 字段提供了关于此规则的简短描述。

字符串定义：

- $path_fake_kernel：伪造的 kerne132.dll 文件路径。
- $path_real_kernel：真正的 Kernel32.dll 文件路径。
- $filename_fake_kernel：伪造的 kerne132.dll 文件名。
- $filename_malware_dll：恶意软件关联的 Lab07-03.dll 文件名。
- $real_kernel_prefix：真正的 Kernel32.dll 文件名前缀。
- $destructive_string：包含警告信息的字符串，可能表明该文件具有破坏性。

条件：

- 文件的前两个字节应该是 0x5a4d。
- 在偏移量 0x3c 的位置，应该有一个值为 0x00004550 的 32 位数字。
- 文件大小应小于 50KB。
- 文件中应包含定义的路径之一（如伪造或真正的 Kernel 路径）。
- 文件中应包含所有定义的字符串。

(四) Lab06-03.dll

```
rule Lab07_03_dll {
   meta:
      description = "Detects Lab07-03.dll based on specific strings"

   strings:
      $string1 = "SADFHUHF" fullword ascii
      $ip_address = "127.26.152.13" fullword ascii
      $pattern1 = "141G1[1l1" fullword ascii
      $pattern2 = "1Y2a2g2r2" fullword ascii

   condition:
      uint16(0) == 0x5a4d and
      uint32(uint32(0x3c)) == 0x00004550 and
      filesize < 500KB and
      all of them
}
```

元数据 (meta)：

- description 表明该规则是为了检测名为”Lab07-03.dll” 的文件。

字符串：

- $string1：匹配”SADFHUHF” 这个字符串。
- $ip_address：匹配”127.26.152.13” 这个 IP 地址。
- $pattern1 和 $pattern2：分别匹配两个特定的字符串模式。

条件：

- uint16(0) == 0x5a4d：检查文件的前两个字节是否为"MZ"（这是 PE（Portable Executable）格式文件的典型标识）。
- uint32(uint32(0x3c)) == 0x00004550：检查 PE 文件的另一个标识。
- filesize < 500KB：确保文件大小小于 500KB。
- all of them：确保上述定义的所有字符串都在文件中被匹配到。

（五） 运行结果

图 50

五、 Ida Python 脚本编写

（一） 代码部分

该 IDA Pro Python 脚本为逆向分析提供了一个快速探索二进制文件的工具。通过遍历文件中的所有段，它精确地提取和打印段名称、起始/结束地址、大小、类型、权限和函数数量等关键信息。这种快速概览使分析师能够立即了解文件的结构和组织，为进一步的深入分析打下基础。从安全分析的角度，识别各个段的权限和功能有助于确定可能的攻击点或潜在的恶意行为，例如不应该是可执行的数据段突然变得可执行可能是恶意活动的标志。此外，通过识别段中的函数数量，分析师可以更加聚焦地分析主要的功能代码，从而更加高效地进行恶意软件分析或二进制安全评估。

```python
# 导入必要的 IDA 模块
import idaapi
import idautils

# 定义一个函数来打印段信息
def print_segments_info():
    # 遍历所有段
    for seg_ea in idautils.Segments():
        # 获取当前段的信息
        seg = idaapi.getseg(seg_ea)

        # 如果能成功获取到段信息，则进行下面的操作
        if seg:
            # 获取段的名称
            seg_name = idaapi.get_segm_name(seg)
            # 获取段的起始地址
            seg_start = seg.start_ea
```

```
            # 获取段的结束地址
            seg_end = seg.end_ea
            # 计算段的大小
            seg_size = seg.end_ea - seg.start_ea
            # 获取段的类别（如代码段、数据段等）
            seg_type = idaapi.get_segm_class(seg)

            # 获取段的权限，初始化为空字符串
            seg_perm = ""
            # 判断该段是否有读权限
            seg_perm += "R" if seg.perm & idaapi.SEGPERM_READ else "-"
            # 判断该段是否有写权限
            seg_perm += "W" if seg.perm & idaapi.SEGPERM_WRITE else "-"
            # 判断该段是否有执行权限
            seg_perm += "X" if seg.perm & idaapi.SEGPERM_EXEC else "-"

            # 计算该段内的函数数量
            functions_count = sum(1 for _ in idautils.Functions(seg_start, seg_end)
                )

            # 打印上述收集的所有信息
            print("[*] Segment Name: {}, Start Address: 0x{:X}, End Address: 0x{:X
                }, Size: {} bytes, Type: {}, Permissions: {}, Functions Count: {}".
                format(
                seg_name, seg_start, seg_end, seg_size, seg_type, seg_perm,
                    functions_count
            ))

# 执行上面定义的函数，打印段信息
print_segments_info()
```

代码分析：

- 定义函数 print_segments_info: 此函数的目的是遍历所有段并打印相关信息。
- 遍历所有段: 使用 idautils.Segments() 可以获取当前 IDA 数据库中的所有段。
- 获取段信息: 对于每一个段，我们使用 idaapi.getseg(seg_ea) 来获取段的信息。这里的 seg_ea 是段的起始地址。
- 获取段名称: 使用 idaapi.get_segm_name(seg)。
- 获取段起始和结束地址: 直接从 seg 结构体中取得 start_ea 和 end_ea。
- 计算段大小: 通过结束地址减去起始地址。
- 获取段类别: 使用 idaapi.get_segm_class(seg) 可以得知这是一个代码段、数据段还是其他类型的段。
- 获取段权限: 使用 seg.perm 获取段的权限，并与 idaapi.SEGPERM_* 常数进行按位与操作，以确定其是否具有读、写和执行权限。
- 计算段中的函数数量: 使用 idautils.Functions(seg_start, seg_end) 遍历在给定段范围内的所有函数。我们只需要函数的数量，所以使用了一个简单的生成器表达式来计数。
- 打印收集的信息: 使用 print 函数将收集到的所有信息打印到输出窗口。

- 执行函数: 在脚本的最后，我们调用了 print_segments_info() 函数，这样在加载脚本时就会执行这个函数，从而打印所有段的相关信息。

(二) 运行结果

图 51

六、 实验结论及心得体会

在本次实验中，我深入探讨了恶意代码的行为和特性。通过对三个不同的样本进行详细分析，我明白了持久性在恶意代码中的关键作用。例如，某些代码会通过安装自己为服务来实现系统中的长期存在，从而持续对系统造成威胁。此外，我也对恶意代码的各种功能和目的有了更深入的了解。有些代码可能会在特定时间发起 DDoS 攻击，而有些则可能只是为了打开广告页面。这些发现强调了进行全面分析的重要性，这既包括静态代码分析，也包括动态的行为观察。实验中，我充分利用了 IDA Pro 这一强大的工具，它帮助我查看和理解了复杂的代码逻辑和数据结构，使我能够更准确地确定恶意代码的行为。在处理恶意代码时，实验的安全性也是至关重要的。我被时刻提醒要在安全的环境中运行这些代码，如使用虚拟机，并始终保持警惕。总体来说，这次实验极大地增强了我的逆向分析技巧，并为我未来的研究和分析工作打下了坚实的基础。

参考文献

[1] SM-D.Practical Malware Analysis[J].Network Security, 2012, 2012(12):4-4.DOI:10.1016/S1353-4858(12)70109-5.