南開大學

《恶意代码分析与防治技术》课程实验报告

实验十

学　　院　网络空间安全学院

专　　业　信息安全

学　　号　2112060

姓　　名　孙蕗

班　　级　信息安全 1 班

一、实验目的

理解恶意软件的用户态行为，包括文件创建、注册表操作等，以及对系统的影响。

学习如何使用工具（如 IDA Pro、Procmon、Resource Hacker 等）来分析和反汇编可执行文件，查看导入函数、资源等信息。

掌握检查内核态组件的方法，了解驱动程序的加载和执行过程。

学习使用调试工具（如 Windbg）分析内核驱动的行为，包括对系统结构的修改和对系统进程的隐藏。

二、 实验原理

恶意软件通过修改系统的注册表、创建文件、创建服务、修改内核、修改代码、加载内核驱动、隐藏进程等方式对系统进行破坏。

使用 Regedit 等工具查看注册表的变化，使用 Procmon 监视系统文件和服务的创建，以及对文件的写入等操作。

恶意软件通过等手段影响系统行为。

使用 Resource Hacker 等工具查看可执行文件的导入函数表、资源节等信息。

使用 Windbg 和 IDA Pro 等工具查看内核驱动的代码，了解其加载过程。

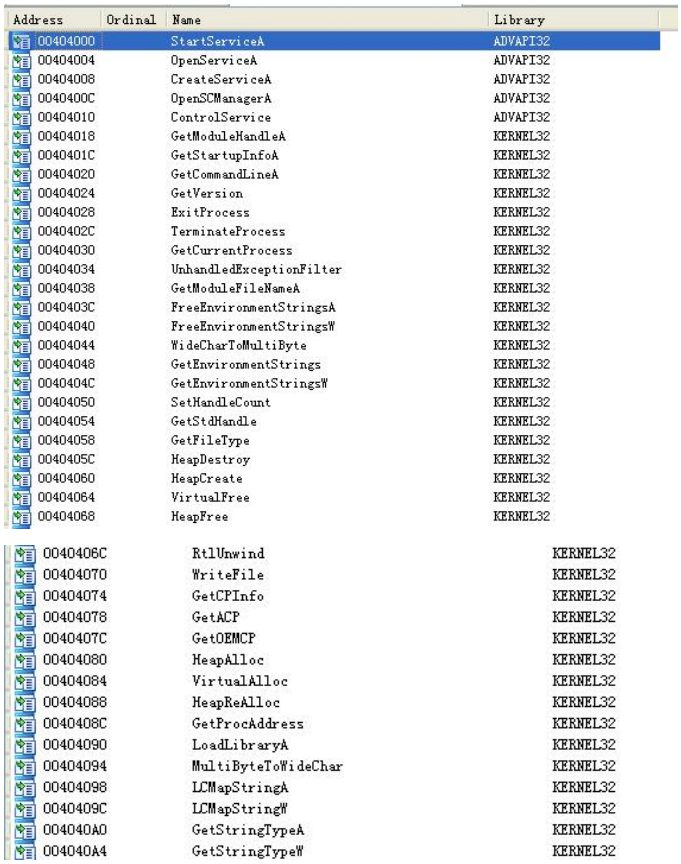
通过分析 DeviceIoControl 请求，学习内核态驱动对系统进程的隐藏原理。

三、实验过程

（一） Lab10-1

必须将驱动程序放到 C:\Windows\System32 目录下,可执行文件是 Lab10-1.exe,驱动程序是 Lab10-01.sys。

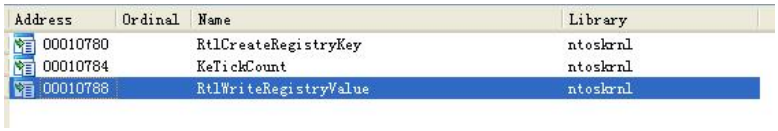
Ida pro 查看 Lab10-1.exe 的导入表



Address	Ordinal	Name	Library
00404000		StartServiceA	ADVAPI32
00404004		OpenServiceA	ADVAPI32
00404008		CreateServiceA	ADVAPI32
0040400C		OpenSCManagerA	ADVAPI32
00404010		ControlService	ADVAPI32
00404018		GetModuleHandleA	KERNEL32
0040401C		GetStartupInfoA	KERNEL32
00404020		GetCommandLineA	KERNEL32
00404024		GetVersion	KERNEL32
00404028		ExitProcess	KERNEL32
0040402C		TerminateProcess	KERNEL32
00404030		GetCurrentProcess	KERNEL32
00404034		UnhandledExceptionFilter	KERNEL32
00404038		GetModuleFileNameA	KERNEL32
0040403C		FreeEnvironmentStringsA	KERNEL32
00404040		FreeEnvironmentStringsW	KERNEL32
00404044		WideCharToMultiByte	KERNEL32
00404048		GetEnvironmentStrings	KERNEL32
0040404C		GetEnvironmentStringsW	KERNEL32
00404050		SetHandleCount	KERNEL32
00404054		GetStdHandle	KERNEL32
00404058		GetFileType	KERNEL32
0040405C		HeapDestroy	KERNEL32
00404060		HeapCreate	KERNEL32
00404064		VirtualFree	KERNEL32
00404068		HeapFree	KERNEL32
0040406C		RtlUnwind	KERNEL32
00404070		WriteFile	KERNEL32
00404074		GetCPInfo	KERNEL32
00404078		GetACP	KERNEL32
0040407C		GetOEMCP	KERNEL32
00404080		HeapAlloc	KERNEL32
00404084		VirtualAlloc	KERNEL32
00404088		HeapReAlloc	KERNEL32
0040408C		GetProcAddress	KERNEL32
00404090		LoadLibraryA	KERNEL32
00404094		MultiByteToWideChar	KERNEL32
00404098		LCMapStringA	KERNEL32
0040409C		LCMapStringW	KERNEL32
004040A0		GetStringTypeA	KERNEL32
004040A4		GetStringTypeW	KERNEL32

有意义的导入函数有 OpenSCManagerA、OpenServiceA、ControlService、StartServiceA 以及 CreateServiceA。这表明这个程序创建了一个服务，并且可能启动或者操作了这个服务，除此之外，它与系统进行了很少的交互。

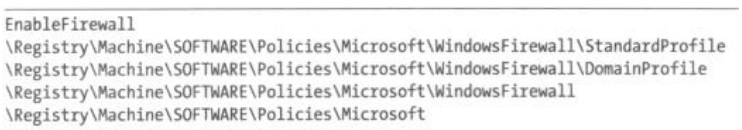
Ida pro 查看 Lab10-1.sys 的导入表



Address	Ordinal	Name	Library
00010780		RtlCreateRegistryKey	ntoskrnl
00010784		KeTickCount	ntoskrnl
00010788		RtlWriteRegistryValue	ntoskrnl

它仅仅导入了 3 个函数。第一个函数是 KeTickCount,几乎所有驱动都包含它，可以忽略。剩下的两个函数是 RtlCreateRegistryKey 和 RtlWriteRegistryvalue,这告诉我们驱动可能访问了注册表。

查看 Lab10-1.sys 的字符串



```
EnableFirewall
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile
\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall
\Registry\Machine\SOFTWARE\Policies\Microsoft
```

这些字符串看起来很像是注册表键值，然而开头\Registry\Machine 却比较怪异，并不像是诸如 HKLM 此类的常见注册表根键。当从内核态访

问注册表时，前缀\Registry\Machine 等同于用户态程序访问的 HKEY_LOCAL_MACHINE。网上搜索一下就可以知道，将 EnableFirewall 置为 0,表示禁用 Windows XP 自带的防火墙。

双击运行 Lab10-01.exe，procmon 中显示有一些读注册表的调用，但是仅有一个写注册表的调用：RegSetValue 写了 HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed 键值。这个注册表键值一直改变，对于恶意代码分析毫无意义，但是因为涉及到内核代码，需要确保驱动没有秘密地修改注册表。

Ida pro 查看 Lab10-1.exe 的 main

```
.text:00401000 ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
.text:00401000 _WinMain@16     proc near               ; CODE XREF: start+C9↓p
.text:00401000
.text:00401000 ServiceStatus   = _SERVICE_STATUS ptr -1Ch
.text:00401000 hInstance       = dword ptr  4
.text:00401000 hPrevInstance   = dword ptr  8
.text:00401000 lpCmdLine       = dword ptr  0Ch
.text:00401000 nShowCmd        = dword ptr  10h
.text:00401000
.text:00401000                 sub     esp, 1Ch
.text:00401003                 push    edi
.text:00401004                 push    0F003Fh         ; dwDesiredAccess
.text:00401009                 push    0               ; lpDatabaseName
.text:0040100B                 push    0               ; lpMachineName
.text:0040100D                 call    ds:OpenSCManagerA
.text:00401013                 mov     edi, eax
.text:00401015                 test    edi, edi
.text:00401017                 jnz     short loc_401020
.text:00401019                 pop     edi
.text:0040101A                 add     esp, 1Ch
.text:0040101D                 retn    10h

.text:00401020 loc_401020:                             ; CODE XREF: WinMain(x,x,x,x)+17↑j
.text:00401020                 push    esi
.text:00401021                 push    0               ; lpPassword
.text:00401023                 push    0               ; lpServiceStartName
.text:00401025                 push    0               ; lpDependencies
.text:00401027                 push    0               ; lpdwTagId
.text:00401029                 push    0               ; lpLoadOrderGroup
.text:0040102B                 push    offset BinaryPathName ; "C:\\Windows\\System32\\Lab10-01.sys"
.text:00401030                 push    1               ; dwErrorControl
.text:00401032                 push    3               ; dwStartType
.text:00401034                 push    1               ; dwServiceType
.text:00401036                 push    0F01FFh         ; dwDesiredAccess
.text:0040103B                 push    offset ServiceName ; "Lab10-01"
.text:00401040                 push    offset ServiceName ; "Lab10-01"
.text:00401045                 push    edi             ; hSCManager
.text:00401046                 call    ds:CreateServiceA
.text:0040104C                 mov     esi, eax
.text:0040104E                 test    esi, esi
.text:00401050                 jnz     short loc_401069
.text:00401052                 push    0F01FFh         ; dwDesiredAccess
.text:00401057                 push    offset ServiceName ; "Lab10-01"
.text:0040105C                 push    edi             ; hSCManager
.text:0040105D                 call    ds:OpenServiceA
.text:00401063                 mov     esi, eax
.text:00401065                 test    esi, esi
.text:00401067                 jz      short loc_401086
```

首先，它在 0x0040100D 调用了 OpenSCManagerA 获取服务管理器的句柄，然后在 0x00401046 调用 CreateServiceA,创建一个名为 Lab10-01 的服务。CreateServiceA 调用说明服务使用 0x00402B 的 C:\Windows\System32\Lab10-01.sys 中的代码，服务类型为 3(SERVICE_KERNEL_DRIVER),意味着这个文件将被加载到内核。

如果 CreateServiceA 调用失败，代码会使用相同的服务名调用 OpenServiceA,如 0x0040100D 所示。如果因为服务已经存在而导致调用 CreateServiceA 失败，这将打开 Lab10-01 服务的句柄。

```
.text:00401069 loc_401069:                             ; CODE XREF: WinMain(x,x,x,x)+50↑j
.text:00401069                 push    0               ; lpServiceArgVectors
.text:0040106B                 push    0               ; dwNumServiceArgs
.text:0040106D                 push    esi             ; hService
.text:0040106E                 call    ds:StartServiceA
.text:00401074                 test    esi, esi
.text:00401076                 jz      short loc_401086
.text:00401078                 lea     eax, [esp+24h+ServiceStatus]
.text:0040107C                 push    eax             ; lpServiceStatus
.text:0040107D                 push    1               ; dwControl
.text:0040107F                 push    esi             ; hService
.text:00401080                 call    ds:ControlService
.text:00401086
.text:00401086 loc_401086:                             ; CODE XREF: WinMain(x,x,x,x)+67↑j
.text:00401086                                         ; WinMain(x,x,x,x)+76↑j
.text:00401086                 pop     esi
.text:00401087                 xor     eax, eax
.text:00401089                 pop     edi
.text:0040108A                 add     esp, 1Ch
.text:0040108D                 retn    10h
.text:0040108D _WinMain@16     endp
```

接下来，程序调用 StartServiceA 来启动服务，如 loc_401069 所示。最后，0x00401080 调用 ControlService.ControlService 的第二个参数是发送控制消息的类型，它的值是 0x010,SERVICE_CONTROL_STOP。这将会卸载驱动，并调用驱动的卸载函数。

Ida pro 查看 Lab10-1.sys 的 DriverEntry 函数

```
INIT:00010959 ; NTSTATUS __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)
INIT:00010959                 public DriverEntry
INIT:00010959 DriverEntry     proc near
INIT:00010959
INIT:00010959 DriverObject    = dword ptr  8
INIT:00010959 RegistryPath    = dword ptr  0Ch
INIT:00010959
INIT:00010959                 mov     edi, edi
INIT:0001095B                 push    ebp
INIT:0001095C                 mov     ebp, esp
INIT:0001095E                 call    sub_10920
INIT:00010963                 pop     ebp
INIT:00010964                 jmp     sub_10906
INIT:00010964 DriverEntry     endp
```

这个函数是驱动的入口点，但是它不是 DriverEntry 函数。编译器在 DriverEntry 的周围插入封装代码。真正的 DriverEntry 函数位于 sub_10906。

```
INIT:00010906 sub_10906       proc near               ; CODE XREF: DriverEntry+B↓j
INIT:00010906
INIT:00010906 arg_0           = dword ptr  8
INIT:00010906
INIT:00010906                 mov     edi, edi
INIT:00010908                 push    ebp
INIT:00010909                 mov     ebp, esp
INIT:0001090B                 mov     eax, [ebp+arg_0]
INIT:0001090E                 mov     dword ptr [eax+34h], offset sub_10486
INIT:00010915                 xor     eax, eax
INIT:00010917                 pop     ebp
INIT:00010918                 retn    8
INIT:00010918 sub_10906       endp
```

DriverEntry 函数的主体部分似乎将一个偏移量移入到一个内存位置，然后除此之外，它没有进行任何函数调用，也没有与系统进行交互。

使用 WinDbg 分析 Lab10-1.sys

```
Microsoft (R) Windows Debugger Version 6.12.0002.633 X86
Copyright (c) Microsoft Corporation. All rights reserved.

CommandLine: "C:\Documents and Settings\lulu\桌面\Practical Malware Analysis Labs\BinaryCollection\Chapter_10L\Lab10-01.exe"
Symbol search path is: *** Invalid ***
****************************************************************************
* Symbol loading may be unreliable without a symbol search path.           *
* Use .symfix to have the debugger choose a symbol path.                   *
* After setting your symbol path, use .reload to refresh symbol locations. *
****************************************************************************
Executable search path is:
ModLoad: 00400000 00407000   image00400000
ModLoad: 7c920000 7c9b3000   ntdll.dll
ModLoad: 7c800000 7c91e000   C:\WINDOWS\system32\kernel32.dll
ModLoad: 77da0000 77e49000   C:\WINDOWS\system32\ADVAPI32.dll
ModLoad: 77e50000 77ee2000   C:\WINDOWS\system32\RPCRT4.dll
ModLoad: 77fc0000 77fd1000   C:\WINDOWS\system32\Secur32.dll
(4f4.438): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffdf000 ecx=00000007 edx=00000080 esi=00241f48 edi=00241eb4
eip=7c92120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll -
ntdll!DbgBreakPoint:
7c92120e cc              int     3
```

将可执行程序加载到 windbg 中，结合之前使用 ida 得到的 controlservice 地址进行断点。使用 bp 00401080，在驱动和卸载之间设置一个断点（在 ControlService 调用上。启动程序 debug->go 直到断点命中。

```
0:000> bp 00401080
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
0:000> g
Breakpoint 0 hit
eax=0012ff1c ebx=7ffdf000 ecx=77dbfb6d edx=00000000 esi=00144058 edi=00144f68
eip=00401080 esp=0012ff08 ebp=0012ffc0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
image00400000+0x1080:
```

一旦程序在断点处暂停，跳出虚拟机，以便连接内核调试器，并且获取关于 Lab10-01.sys 的信息。打开 Windbg 的另一个实例，选择 File→Kernel Debug，设置管道为\\.\pipe\com_1，波特率(baud rate)为 115200，将宿主机上运行的 WinDbg 实例与虚拟机中的内核连接上。通过使用命令!drvobj，来获取驱动对象。

```
nt!RtlpBreakWithStatusInstruction:
80527bdc cc              int     3
kd> !object \Driver
Object: e101d910  Type: (8a360418) Directory
    ObjectHeader: e101d8f8 (old version)
    HandleCount: 0  PointerCount: 87
    Directory Object: e1001150  Name: Driver

    Hash Address  Type                      Name
    ---- -------  ----                      ----
     00  8a0fd3a8 Driver                    Beep
         8a20c3b0 Driver                    NDIS
         8a053438 Driver                    KSecDD
     01  8a0ad7d0 Driver                    Mouclass
         8a04ae38 Driver                    Raspti
         89e28de8 Driver                    es1371
     02  89e29bf0 Driver                    vmx_svga
     03  89e57b90 Driver                    Fips
         8a1f87b0 Driver                    Kbdclass
     04  89fe6f38 Driver                    VgaSave
         89ee6030 Driver                    NDProxy
         8a2a60b8 Driver                    Compbatt
     05  8a292ec8 Driver                    Ptilink
         8a31e850 Driver                    MountMgr
         8a2717e0 Driver                    wdmaud
     07  89e0d7a0 Driver                    dmload
         8a2b0218 Driver                    isapnp
         89e3c2c0 Driver                    swmidi
     08  8a28b948 Driver                    redbook
         8a1f75f8 Driver                    vmmouse
         8a0ed510 Driver                    atapi
     09  89e0ea08 Driver                    vmscsi
     10  89fe4da0 Driver                    IpNat
         8a0e3728 Driver                    RasAcd
         8a072d68 Driver                    PSched
```

使用 dt _DRIVER_OBJECT 地址 解析地址的数据结构。

```
kd> !object 89d3ada0
Object: 89d3ada0  Type: (8a3275b8) Driver
    ObjectHeader: 89d3ad88 (old version)
    HandleCount: 0  PointerCount: 2
    Directory Object: e101d910  Name: Lab10-01
kd> dt _DRIVER_OBJECT 89d3ada0
nt!_DRIVER_OBJECT
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : (null)
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xbaf7e000 Void
   +0x010 DriverSize       : 0xe80
   +0x014 DriverSection    : 0x89e3b630 Void
   +0x018 DriverExtension  : 0x89d3ae48 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\Lab10-01"
   +0x024 HardwareDatabase : 0x80670ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xbaf7e959     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0xbaf7e486     void  +0
   +0x038 MajorFunction    : [28] 0x804f354a     long  nt!IopInvalidDeviceRequest+0
```

确定驱动卸载时调用的函数——偏移量 0x034 的信息 DriverUnload。

使用命令 bp 0xbaf7e486 设置一个断点,恢复内核运行。回到虚拟机中运行可执行程序的 WinDbg,也恢复它的运行。由于内核调试器命中了断点，整个 Guest OS 会卡死。

此时，回到内核调试器，g 单步调试代码。

```
kd> p
Lab10_01+0x48d:
baf7e48d 56              push    esi
kd> p
Lab10_01+0x48e:
baf7e48e 8b3580e7f7ba    mov     esi,dword ptr [Lab10_01+0x780 (baf7e780)]
kd> p
Lab10_01+0x494:
baf7e494 57              push    edi
kd> p
Lab10_01+0x495:
baf7e495 33ff            xor     edi,edi
kd> p
Lab10_01+0x497:
baf7e497 68bce6f7ba      push    offset Lab10_01+0x6bc (baf7e6bc)
kd> p
Lab10_01+0x49c:
baf7e49c 57              push    edi
kd> du baf7e6bc
baf7e6bc  "\Registry\Machine\SOFTWARE\Polic"
baf7e6fc  "ies\Microsoft"
kd> t
Lab10_01+0x49d:
baf7e49d 897dfc          mov     dword ptr [ebp-4],edi
kd> t
Lab10_01+0x4a0:
baf7e4a0 ffd6            call    esi
kd> t
nt!RtlCreateRegistryKey:
805ddafe 8bff            mov     edi,edi
```

看到程序调用了多次 RtlCreateRegistryKey 函数，创建了一些注册表键，然后调用了多次 RtlWriteRegistryValue 函数，在两个地方设置 EnableFirewall 值为 0。从内核禁用 Windows XP 防火墙的这种方法难以被安全程序探测到。

IDA 中 ControlService 的地址为 0x401080,使用命令!drvobj lab10-01，来获取驱动对象。使用命令 bp 0xf8d44486 设置一个断点，在 Windbg 进行单步调试。

```
kd> dt _DRIVER_OBJECT 81776978
nt!_DRIVER_OBJECT
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : (null)
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xf8d44000 Void
   +0x010 DriverSize       : 0xe80
   +0x014 DriverSection    : 0x81d79908 Void
   +0x018 DriverExtension  : 0x81776a20 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\Lab10-01"
   +0x024 HardwareDatabase : 0x80671ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xf8d44959     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0xf8d44486     void  +0
   +0x038 MajorFunction    : [28] 0x804f454a     long  nt!IopInvalidDeviceRequest+0
```

```
kd> g
Breakpoint 1 hit
Lab10_01+0x486:
f8d44486 8bff            mov     edi,edi
kd> p
Lab10_01+0x488:
f8d44488 55              push    ebp
kd> p
Lab10_01+0x489:
f8d44489 8bec            mov     ebp,esp
kd> p
Lab10_01+0x48b:
f8d4448b 51              push    ecx
```

前三行汇编代码是一个函数入口的地方，说明刚刚那个断点断在了一个函数的入口。之后多次调用了 RtlCreateRegistryKey 函数来创建注册表键，并调用了两次 RtlwriteRegistryalue 函数将 EnableFirewall 的值设置为 0，禁用 Windows XP 防火墙。

1. **这个程序是否直接修改了注册表(使用 procmon 来检查)？**

如果使用 procmon 监视这个程序，你会看到唯一写注册表的调用是写键值 HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed 的 RegSetValue 调用。对注册表的一些间接修改通过调用 CreateServiceA 来完成，但这个程序也从内核对注册表做了直接修改，这些修改却不能被 procmon 探测到。

2. **用户态的程序调用了 ControlService 函数，你是否能够使用 WinDbg 设置一个断点，以此来观察由于 ControlService 的调用导致内核执行了怎样的操作？**

要设置一个断点来查看内核发生了什么，你必须使用一个运行在虚拟机中的 WinDbg 实例，来打开这个可执行文件，而调试内核使用运行在宿主操作系统中的 WinDbg 另外一个实例。当 Lab10-01.exe 在虚拟机中被暂停后，使用!drvobj 命令获得驱动设备的句柄，它包含一个卸载函数的指针。接下来，在驱动的卸载函数上设置一个断点。重启 Lab10-01.exe 之后，断点将会被触发。

3. **这个程序做了些什么？**

这个程序创建一个服务来加载驱动。然后，驱动代码会创建注册表键\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile 和\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile。在 Windows XP 系统中，设置这些键值将禁用防火墙。

（二） Lab10-2

查看这个可执行文件的导入函数表，看到 CloseServiceHandle、CreateServiceA、OpenSCManagerA 以及 StartServiceA,说明这个程序创建并启动一个服务。程序也调用了 CreateFile 和 WriteFile,说明程序将在某个时间点写文件。LoadResource 和 SizeOfResource 调用，说明这个程序对 Lab10-02.exe 的资源节做了某些处理。

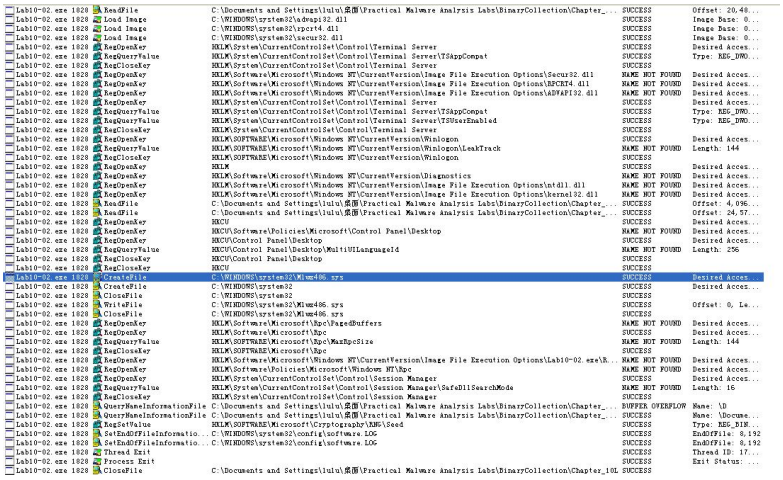
Address	Ordinal	Name	Library
00405000		CreateServiceA	ADVAPI32
00405004		StartServiceA	ADVAPI32
00405008		CloseServiceHandle	ADVAPI32
0040500C		OpenSCManagerA	ADVAPI32
00405014		CreateFileA	KERNEL32
00405018		SizeofResource	KERNEL32
0040501C		WriteFile	KERNEL32
00405020		FindResourceA	KERNEL32
00405024		LoadResource	KERNEL32
00405028		CloseHandle	KERNEL32
0040502C		GetCommandLineA	KERNEL32
00405030		GetVersion	KERNEL32
00405034		ExitProcess	KERNEL32
00405038		TerminateProcess	KERNEL32
0040503C		GetCurrentProcess	KERNEL32
00405040		UnhandledExceptionFilter	KERNEL32
00405044		GetModuleFileNameA	KERNEL32
00405048		FreeEnvironmentStringsA	KERNEL32
0040504C		FreeEnvironmentStringsW	KERNEL32
00405050		WideCharToMultiByte	KERNEL32
00405054		GetEnvironmentStrings	KERNEL32
00405058		GetEnvironmentStringsW	KERNEL32
0040505C		SetHandleCount	KERNEL32
00405060		GetStdHandle	KERNEL32
00405064		GetFileType	KERNEL32
00405068		GetStartupInfoA	KERNEL32
0040506C		HeapDestroy	KERNEL32
00405070		HeapCreate	KERNEL32
00405074		VirtualFree	KERNEL32
00405078		HeapFree	KERNEL32
0040507C		RtlUnwind	KERNEL32
00405080		HeapAlloc	KERNEL32
00405084		GetCPInfo	KERNEL32
00405088		GetACP	KERNEL32
0040508C		GetOEMCP	KERNEL32
00405090		VirtualAlloc	KERNEL32
00405094		HeapReAlloc	KERNEL32
00405098		GetProcAddress	KERNEL32
0040509C		LoadLibraryA	KERNEL32
004050A0		GetLastError	KERNEL32
004050A4		FlushFileBuffers	KERNEL32
004050A8		SetFilePointer	KERNEL32
004050AC		MultiByteToWideChar	KERNEL32
004050B0		LCMapStringA	KERNEL32
004050B4		LCMapStringW	KERNEL32
004050B8		GetStringTypeA	KERNEL32
004050BC		GetStringTypeW	KERNEL32
004050C0		SetStdHandle	KERNEL32

使用 Resource Hacker 来检查资源节，发现这里有一个 FILE，里面包含了另一个 PE 头。这可能是 Lab10-02 将要使用的另外一个恶意文件。

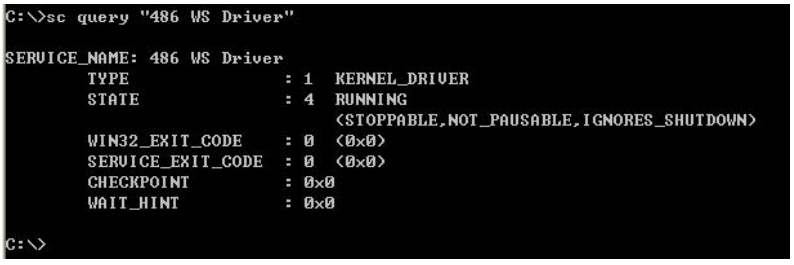
```
00007070  4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00
00007080  B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00
00007090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000070A0  00 00 00 00 00 00 00 00 00 00 00 00 58 02 00 00
000070B0  0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68
000070C0  69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F
000070D0  74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20
000070E0  6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00
000070F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007100  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00007110  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

运行程序，发现它创建了一个文件和服务。使用 procmon,看到程序在 C:\Windows\System32 目录下创建了一个文件，并且以这个文件作为一个可执行程序创建了一个名为 486 WS Driver 服务。这个文件包含了由操作系统加载的内核代码。这个文件一共写了三个文件，一个是 system.LOG，一个是 system，还有一个是 SysEvent.Evt。文件创建了 conime.exe、apphelp.dll，sysmain.sdb，systest.sdb，以及 Mlwx486.sys。

然而到上述目录下却没有找到程序创建的文件，且在 procmon 中并没有关于这个文件被删除的操作，推测这个文件被隐藏了，是一个 Rootkit。



使用 sc 命令，检查运行内核驱动程序的服务状态



```
C:\>sc query "486 WS Driver"

SERVICE_NAME: 486 WS Driver
        TYPE               : 1  KERNEL_DRIVER
        STATE              : 4  RUNNING
                                (STOPPABLE,NOT_PAUSABLE,IGNORES_SHUTDOWN)
        WIN32_EXIT_CODE    : 0  (0x0)
        SERVICE_EXIT_CODE  : 0  (0x0)
        CHECKPOINT         : 0x0
        WAIT_HINT          : 0x0

C:\>
```

查询名为 486 WS Driver 的服务,它在 CreateServiceA 中被指定。看到这个服务仍然运行,这告诉我们内核代码在内存中。奇怪的是驱动仍然运行，但是它没有在硬盘上。

连接虚拟机与内核调试器，然后使用 lm 命令，查看驱动是否被真正加载。文件名为 Mlwx486.sys 的驱动被载入到内存，但文件没有在硬盘上显示，推测这个文件可能是一个 Rootkit。使用命令 dd dwo(KeServiceDescriptorTable) L100 检查 SSDT 的修改情况，发现异常地址 b9887486。

```
804e38a8  8058b1bd 8056e992 8056e903 8064a075
804e38b8  805dc2e4 8059d9c6 805de99e 805de238
804e38c8  805ab834 80572cb1 805dc07c 805754b2
804e38d8  8064a047 8064a047 804f8e4d 80567b9e
804e38e8  8057fa9d b9887486 805853a1 806179d0
804e38f8  80591ab3 8057d810 805d86e8 80573c6a
804e3908  805818a1 806241e7 8056eb30 8056ca87
```

使用命令 u b9887486 查看此处的内容，发现是一个 Mlwx486 里面自带的函数。

```
kd> u b9887486
*** ERROR: Module load completed but symb
Mlwx486+0x486:
b9887486 8bff            mov     edi,edi
b9887488 55              push    ebp
b9887489 8bec            mov     ebp,esp
```

b9887486 处条目所在的内存位置很明显位于 ntoskrnl 模块的范围之外，位于加载的 Mhwx486.sys 驱动内。恢复虚拟机到 Rootkit 安装之前的状态，使用命令 dd dwo(KeServiceDescriptorTable) L100 以便查看存储在 SSDT 中的哪个函数被覆盖了。

```
804e3888  805790e5 805963a9 805727c7 8056eef5
804e3898  8056f0ee 80571fd7 8059ffc5 80591902
804e38a8  8058b1bd 8056e992 8056e903 8064a075
804e38b8  805dc2e4 8059d9c6 805de99e 805de238
804e38c8  805ab834 80572cb1 805dc07c 805754b2
804e38d8  8064a047 8064a047 804f8e4d 80567b9e
804e38e8  8057fa9d 80573111 805853a1 806179d0
804e38f8  80591ab3 8057d810 805d86e8 80573c6a
804e3908  805818a1 806241e7 8056eb30 8056ca87
804e3918  8056f65f 8057ee21 8064ac33 80617844
804e3928  80571a6d 8064f320 8064a5b8 805804a8
804e3938  8064f529 80568348 80618297 8057e4cc
```

被覆盖的函数是 NtQueryDirectoryFile,它是一个提取文件和目录信息的通用函数，而 FindFirstFile 和 FindNextFile 都是调用它来遍历目录结构的。Windows 资源管理器也会利用它来显示文件和目录。如果 Rootkit 挂钩了这个函数，它可以隐藏文件，这也解释了不能发现 Mlwx486.sys 的原因。

在代替 NtQueryDirectoryFile 调用的挂钩函数 PatchFunction 上设置一个断点，发现它做的第一件事是用原始参数调用原始的 NtQueryDirectoryFile 函数。

使用命令 bp f7a75486 设置断点，进行单步调试。

```
d> bp f7a75486
d> g
reakpoint 0 hit
lwx486+0x486:
7a75486 8bff            mov     edi,edi
```

```
mov    edi, edi
push ebp
mov    ebp, esp
push esi
mov    esi, dword ptr [ebp+1Ch]
push edi
push dword ptr [ebp+30h]
push dword ptr [ebp+2Ch]
push dword ptr [ebp+28h]
push dword ptr [ebp+24h]
push dword ptr [ebp+20h]
push esi
push dword ptr [ebp+18h]
push dword ptr [ebp+14h]
push dword ptr [ebp+10h]
push dword ptr [ebp+0Ch]
push dword ptr [ebp+8]
call Miwx486+0x514(f7a75514)
```

到这里，函数开始调用函数 Miwx486+0x514。先计算地址的值，然后查看内存地址上的值是多少。

```
kd> dc b8197d60
b8197d60  00000000 0348f0e8 7c92e4f4 badb0d00
b8197d70  0348ede4 b7f13d98 b7f13dcc 00000000
b8197d80  00000000 00000000 00000000 00000000
b8197d90  00000000 00000000 00000023 00000023
b8197da0  73ce559a 00149318 00000000 00000001
b8197db0  ffffffff 0000003b 00100001 7c92d580
```

```
mov    esi, dword ptr [ebp+1Ch]
push edi                        = b95a5d64
/* Note: ebp = b95a5d30 */
push dword ptr [ebp+30h] = [b95a5d60] = 0
```

```
push dword ptr [ebp+2Ch] = [b95a5d5c] = 80 e1 70
push dword ptr [ebp+28h] = [b95a5d58] = 1
push dword ptr [ebp+24h] = [b95a5d54] = 3
push dword ptr [ebp+20h] = [b95a5d50] = 268
push esi = 0070e198
push dword ptr [ebp+18h] = [b95a5d48] = 68 e1 70
push dword ptr [ebp+14h] = [b95a5d44] = 0
push dword ptr [ebp+10h] = [b95a5d40] = 0
push dword ptr [ebp+0Ch] = [b95a5d3c] = 0
push dword ptr [ebp+8]     = [b95a5d38] = 464
call Miwx486+0x514(f7ab7514)
```

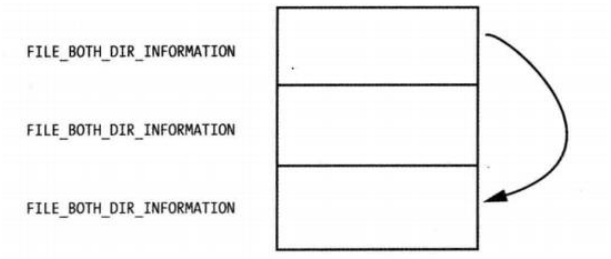
PatchFunction 函数检查第 8 个参数 FileInformationClass,如果它是 3 之外的值，便会返回 NtoueryDirectoryFile 的原始返回值。另外，它也检查 NtQueryDirectoryFile 的返回值与第 9 个参数值 ReturnSingleEntry。PatchFunction 查找某个参数，如果参数不符合条件，那么它的功能与原始函数一样。如果参数满足了条件，则 PatchFunction 将会修改返回值。

```
kd> p
Mlwx486+0x4af:
f7ab74af e860000000      call    Mlwx486+0x514 (f7ab7514)
kd> t
Mlwx486+0x514:
f7ab7514 ff258075abf7    jmp     dword ptr [Mlwx486+0x580 (f7ab7580)]
```

在 PatchFunction 上设置一个断点，但是当且仅当 ReturnSingleEntry 为 0 时，断点才被触发，断点此处是 RtlCompareMemory 函数调用。

RtlCompareMemory 的第一个参数是 eax,它存储了偏移量 esi+5eh,这是文件名的偏移量。在先前的反汇编代码中，esi 是 FileInformation,包含 NtQueryDirectoryFile 填充的信息。这是一个 FILE_BOTH_DIR_INFORMATION 结构，则 0x5E 偏移量是存储宽字符串文件名的地址，这段代码用于比较每个文件的文件名开头的四个字节是否是"Mlwx" 。

使用命令 db esi+5eh 查看存储在 esi+5eh 的内容,如果传入的 FileName 和 Mlwx 不相等，函数直接退出。



假设第一个结构体的偏移量为 00000060，结构体长度为 0x60，再假设恶意驱动在 NtQueryDirectoryFile 返回的第三个结构体里面发现了文件名字和 Mlwx 匹配，之后通过赋值语句 mov eax, dword ptr[esi]获得了 Mlwx486.sys 这个文件的 FILE_BOTH_DIR_INFORMATION 结构里面的 NextEntryOffset 的值，然后将这个这个偏移值乘以 2。

程序在第三个结构体发现 Mlwx 之后，通过 add dword ptr [edi], eax 偏移值 NextEntryOffset 的值变成了 00000180（c0*2）然后计算机通过上面那个公式计算真实地址本来第三个结构体的地址是 000000c0，但是经过这么一个通过改变 offset 之后，计算机计算之后，得出的地址就变成 00000180

计算机通过计算之后，认为第三个结构体存在 00000180 这个地址上，就去 00000180 上取数据，从而跳过了第三个结构体，所以这个通过改变 offset 在不改变数据结构的前提之下，达到了隐藏文件的目的。

使用 DOS 命令 copy Mlwx486.sys NewFilename.sys 来复制文件，文件 NewFilename.sys 应该不会被隐藏。

使用 IDA 打开 NewFilename.sys 查看 DriverEntry 例程

```
sub_10706 proc near

SystemRoutineName= UNICODE_STRING ptr -10h
DestinationString= UNICODE_STRING ptr -8

mov     edi, edi
push    ebp
mov     ebp, esp
sub     esp, 10h
push    esi
mov     esi, ds:RtlInitUnicodeString
push    edi
push    offset SourceString ; "N"
lea     eax, [ebp+DestinationString]
push    eax             ; DestinationString
call    esi ; RtlInitUnicodeString
push    offset aKeservicedescr ; "KeServiceDescriptorTable"
lea     eax, [ebp+SystemRoutineName]
push    eax             ; DestinationString
call    esi ; RtlInitUnicodeString
mov     esi, ds:MmGetSystemRoutineAddress
lea     eax, [ebp+DestinationString]
push    eax             ; SystemRoutineName
call    esi ; MmGetSystemRoutineAddress
mov     edi, eax
lea     eax, [ebp+SystemRoutineName]
push    eax             ; SystemRoutineName
call    esi ; MmGetSystemRoutineAddress
```

RtlInitUnicodeString 以参数 KeServiceDescirptorTable 和 NtQueryDircetoryFile 做入参，然后用 MmGetSystemRoutineAddress 这个函数来查找这个两个地址的偏移量，接下来，它在 SSDT 中查找 NtQueryDirectoryFile 函数项，并用 PatchFunction 的地址覆盖这个项。它没有创建设备，也没有向驱动对象添加任何处理函数。

1. 这个程序创建文件了吗?它创建了什么文件?

这个程序创建了文件 C:\WindowslSystem32\Mlhwx486.sys。你可以使用 procmon 或者其他动态监视工具来查看文件创建，但是，因为文件被隐藏，所以在硬盘上看不到它。

2. 这个程序有内核组件吗?

这个程序拥有一个内核模块。这个内核模块被存储在这个文件的资源节中，然后写入硬盘并作为一个服务加载到内核。

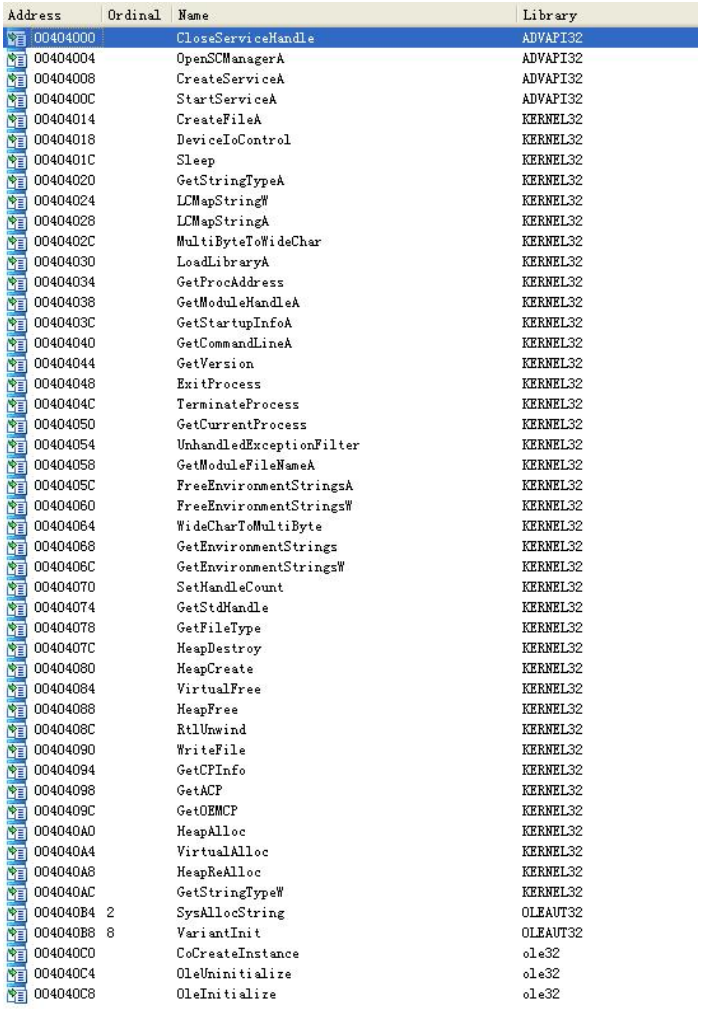
3. 这个程序做了些什么?

这个程序是一个被设计来隐藏文件的 Rootkit.它使用 SSDT 挂钩来覆盖 NtQueryDirectoryFile 的入口，它会隐藏目录列表中任何以 M/wx(区分大小写)开头的文件。

（三） Lab10-3

必须将驱动程序放到 C:\Windows\System32 目录下,可执行文件是 Lab10-3.exe,驱动程序是 Lab10-03.sys

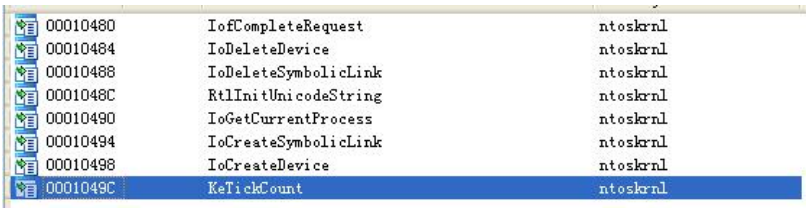
查看 Lab10-03.exe 的导入表



Address	Ordinal	Name	Library
00404000		CloseServiceHandle	ADVAPI32
00404004		OpenSCManagerA	ADVAPI32
00404008		CreateServiceA	ADVAPI32
0040400C		StartServiceA	ADVAPI32
00404014		CreateFileA	KERNEL32
00404018		DeviceIoControl	KERNEL32
0040401C		Sleep	KERNEL32
00404020		GetStringTypeA	KERNEL32
00404024		LCMapStringW	KERNEL32
00404028		LCMapStringA	KERNEL32
0040402C		MultiByteToWideChar	KERNEL32
00404030		LoadLibraryA	KERNEL32
00404034		GetProcAddress	KERNEL32
00404038		GetModuleHandleA	KERNEL32
0040403C		GetStartupInfoA	KERNEL32
00404040		GetCommandLineA	KERNEL32
00404044		GetVersion	KERNEL32
00404048		ExitProcess	KERNEL32
0040404C		TerminateProcess	KERNEL32
00404050		GetCurrentProcess	KERNEL32
00404054		UnhandledExceptionFilter	KERNEL32
00404058		GetModuleFileNameA	KERNEL32
0040405C		FreeEnvironmentStringsA	KERNEL32
00404060		FreeEnvironmentStringsW	KERNEL32
00404064		WideCharToMultiByte	KERNEL32
00404068		GetEnvironmentStrings	KERNEL32
0040406C		GetEnvironmentStringsW	KERNEL32
00404070		SetHandleCount	KERNEL32
00404074		GetStdHandle	KERNEL32
00404078		GetFileType	KERNEL32
0040407C		HeapDestroy	KERNEL32
00404080		HeapCreate	KERNEL32
00404084		VirtualFree	KERNEL32
00404088		HeapFree	KERNEL32
0040408C		RtlUnwind	KERNEL32
00404090		WriteFile	KERNEL32
00404094		GetCPInfo	KERNEL32
00404098		GetACP	KERNEL32
0040409C		GetOEMCP	KERNEL32
004040A0		HeapAlloc	KERNEL32
004040A4		VirtualAlloc	KERNEL32
004040A8		HeapReAlloc	KERNEL32
004040AC		GetStringTypeW	KERNEL32
004040B4	2	SysAllocString	OLEAUT32
004040B8	8	VariantInit	OLEAUT32
004040C0		CoCreateInstance	ole32
004040C4		OleUninitialize	ole32
004040C8		OleInitialize	ole32

存在 OpenSCManagerA，还有 StartServiceA 以及 CreateServiceA 这三个导出函数，说明这个代码会创建一个服务在系统中。

查看 Lab10-03.sys 的导入表



00010480	IofCompleteRequest	ntoskrnl
00010484	IoDeleteDevice	ntoskrnl
00010488	IoDeleteSymbolicLink	ntoskrnl
0001048C	RtlInitUnicodeString	ntoskrnl
00010490	IoGetCurrentProcess	ntoskrnl
00010494	IoCreateSymbolicLink	ntoskrnl
00010498	IoCreateDevice	ntoskrnl
0001049C	KeTickCount	ntoskrnl

IoCreateDevice 例程创建供驱动程序使用的设备对象

IoCreateSymbolicLink 例程在设备对象名称和设备的用户可见名称之间建立符号链接

IoGetCurrentProcess 例程返回一个指向当前进程的指针

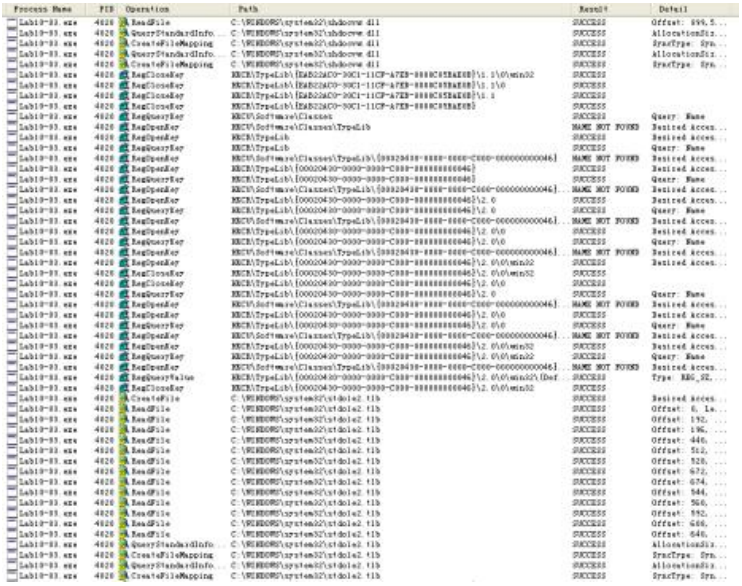
IoCompleteRequest 例程指示调用者已完成给定 I/O 请求的所有处理，并将给定的 IRP 返回给 I/O 管理器

KeTickCount 检索自系统启动以来经过的毫秒数，最长为 49.7 天

RtlInitUnicodeString 初始化一个统计的 Unicode 字符串

其中仅有 IoGetCurrentProcess 导入函数提供了足够信息(其他的导入仅仅被驱动用来创建用户态可访问的设备)。IoGetCurrentProcess 调用告诉我们这个驱动或者修改正在运行进程，或者需要关于进程的信息。将驱动文件复制到 C:\Windows\System32,双击可执行程序运行它，看到一个弹出广告。

查看 Procmon 的结果，看到很多对注册表、文件的操作。



首先，查看服务是否被正确安装，并且验证恶意.sys 是否作为服务的一部分。同时，注意到大约 30 秒后，程序再次弹出广告，每隔 30 秒

执行一次。当打开任务管理器试图终止进程时，看到程序没有被列出，另外在 Process Explorer 中也没有列出。程序继续打开广告，没有一种简单办法可以停止它。因为没有在进程列表中，所以不能通过杀掉进程的方式，来结束它。

因为程序没有在 WinDbg 或者 OllyDbg 进程列表中显示，因此也不能附加一个调试器到进程。此时，唯一的选择是还原到最近快照或是重启系统，希望程序不能持续，如果它不能持续，重启会停止它。

在 IDA pro 中分析.exe 可执行文件，找到 WinMain 函数。

```
.text:00401000 ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
.text:00401000 _WinMain@16     proc near               ; CODE XREF: start+C9↓p
.text:00401000
.text:00401000 ppv             = dword ptr -28h
.text:00401000 BytesReturned   = dword ptr -24h
.text:00401000 pvarg           = VARIANTARG ptr -20h
.text:00401000 var_10          = word ptr -10h
.text:00401000 var_8           = dword ptr -8
.text:00401000 hInstance       = dword ptr  4
.text:00401000 hPrevInstance   = dword ptr  8
.text:00401000 lpCmdLine       = dword ptr  0Ch
.text:00401000 nShowCmd        = dword ptr  10h
.text:00401000
.text:00401000                 sub     esp, 28h
.text:00401003                 push    esi
.text:00401004                 push    0F003Fh         ; dwDesiredAccess
.text:00401009                 push    0               ; lpDatabaseName
.text:0040100B                 push    0               ; lpMachineName
.text:0040100D                 call    ds:OpenSCManagerA
.text:00401013                 test    eax, eax
.text:00401015                 jz      loc_401131
.text:0040101B                 push    0               ; lpPassword
.text:0040101D                 push    0               ; lpServiceStartName
.text:0040101F                 push    0               ; lpDependencies
.text:00401021                 push    0               ; lpdwTagId
.text:00401023                 push    0               ; lpLoadOrderGroup
.text:00401025                 push    offset BinaryPathName ; "C:\\Windows\\System32\\Lab10-03.sys"
.text:0040102A                 push    1               ; dwErrorControl
.text:0040102C                 push    3               ; dwStartType
.text:0040102E                 push    1               ; dwServiceType
.text:00401030                 push    0F01FFh         ; dwDesiredAccess
.text:00401035                 push    offset DisplayName ; "Process Helper"
.text:0040103A                 push    offset DisplayName ; "Process Helper"
.text:0040103F                 push    eax             ; hSCManager
.text:00401040                 call    ds:CreateServiceA
.text:00401046                 mov     esi, eax
.text:00401048                 test    esi, esi
.text:0040104A                 jz      short loc_401057
.text:0040104C                 push    0               ; lpServiceArgVectors
.text:0040104E                 push    0               ; dwNumServiceArgs
.text:00401050                 push    esi             ; hService
.text:00401051                 call    ds:StartServiceA
```

第一个调用的函数是 OpenSCManagerA，如果调用成功，执行 0x00401018。字符串变量 C:\\Windows\\System32\\Lab10-03.sys 被压入栈中。BinaryPathName 的值是那个驱动文件的路径，用于指明服务的二进制文件的位置的参数。dwStartType 参数的值为 3，意思是用户可以使用“服务”控制面板实用程序启动服务。 用户可以在“开始参数”字段中为服务指定参数。 服务控制程序可以启动服务并使用 StartService 函数指定其参数。下一个参数是 dwServiceType 的值为 1，意义是表明这是个驱动服务，最后的 lpServiceName 和 lpDisplayName 说明的是这个服务的名字是 Process Helper 如果调用 CreateServiceA 成功的话，执行 StartService，恶意驱动 Lab10-03.sys 就会被加载到内核中。

```
.text:00401057 loc_401057:                             ; CODE XREF: WinMain(x,x,x,x)+4A↑j
.text:00401057                 push    esi             ; hSCObject
.text:00401058                 call    ds:CloseServiceHandle
.text:0040105E                 push    0               ; hTemplateFile
.text:00401060                 push    80h             ; dwFlagsAndAttributes
.text:00401065                 push    2               ; dwCreationDisposition
.text:00401067                 push    0               ; lpSecurityAttributes
.text:00401069                 push    0               ; dwShareMode
.text:0040106B                 push    0C0000000h      ; dwDesiredAccess
.text:00401070                 push    offset FileName ; "\\\\.\\ProcHelper"
.text:00401075                 call    ds:CreateFileA
.text:0040107B                 cmp     eax, 0FFFFFFFFh
.text:0040107E                 jnz     short loc_40108C
.text:00401080                 mov     eax, 1
.text:00401085                 pop     esi
.text:00401086                 add     esp, 28h
.text:00401089                 retn    10h
```

创建了一个文件在\\.\ProcHelper 这个地方并作为一个句柄打开。

```
.text:0040108C loc_40108C:                             ; CODE XREF: WinMain(x,x,x,x)+7E↑j
.text:0040108C                 lea     ecx, [esp+2Ch+BytesReturned]
.text:00401090                 push    0               ; lpOverlapped
.text:00401092                 push    ecx             ; lpBytesReturned
.text:00401093                 push    0               ; nOutBufferSize
.text:00401095                 push    0               ; lpOutBuffer
.text:00401097                 push    0               ; nInBufferSize
.text:00401099                 push    0               ; lpInBuffer
.text:0040109B                 push    0ABCDEF01h      ; dwIoControlCode
.text:004010A0                 push    eax             ; hDevice
.text:004010A1                 call    ds:DeviceIoControl
.text:004010A7                 push    0               ; pvReserved
.text:004010A9                 call    ds:OleInitialize
.text:004010AF                 test    eax, eax
.text:004010B1                 jl      short loc_401131
.text:004010B3                 lea     edx, [esp+2Ch+ppv]
.text:004010B7                 push    edi
.text:004010B8                 push    edx             ; ppv
.text:004010B9                 push    offset riid     ; riid
.text:004010BE                 push    4               ; dwClsContext
.text:004010C0                 push    0               ; pUnkOuter
.text:004010C2                 push    offset rclsid   ; rclsid
.text:004010C7                 call    ds:CoCreateInstance
.text:004010CD                 mov     eax, [esp+30h+ppv]
.text:004010D1                 test    eax, eax
.text:004010D3                 jz      short loc_40112A
.text:004010D5                 lea     eax, [esp+30h+pvarg]
.text:004010D9                 push    eax             ; pvarg
.text:004010DA                 call    ds:VariantInit
.text:004010E0                 push    offset psz      ; "http://www.malwareanalysisbook.com/ad.h"...
.text:004010E5                 mov     [esp+34h+var_10], 3
.text:004010EC                 mov     [esp+34h+var_8], 1
.text:004010F4                 call    ds:SysAllocString
.text:004010FA                 mov     edi, ds:Sleep
.text:00401100                 mov     esi, eax
```

DeviceIoControl 函数是将控制代码直接发送到指定的设备驱动程序，导致相应的设备执行相应的操作。这里 DeviceIoControl 的参数 lpInBuffer 和 lpOutBuffer 被设置为了 Null 很不寻常，意味着这个请求没有发送任何的信息到内核驱动中(lpInBuffer=0)，并且内核驱动的反馈也是没有的(lpOutBuffer=0)。

还有个异常的地方就是 dwIoControlCode 的值是 abcdedf01。

CoCreateInstance 创建与指定的 CLSID 关联的类的单个未初始化对象。

在调用 SysAllocString 之前，字符串”http://www.malwareanalysisbook.com/ad.html”被压入了栈中，这个字符串就是开始运行时不断打开的链接。

```
.text:00401102 loc_401102:                             ; CODE XREF: WinMain(x,x,x,x)+128↓j
.text:00401102                 lea     edx, [esp+30h+pvarg]
.text:00401106                 mov     eax, [esp+30h+ppv]
.text:0040110A                 push    edx
.text:0040110B                 lea     edx, [esp+34h+pvarg]
.text:0040110F                 mov     ecx, [eax]
.text:00401111                 push    edx
.text:00401112                 lea     edx, [esp+38h+pvarg]
.text:00401116                 push    edx
.text:00401117                 lea     edx, [esp+3Ch+var_10]
.text:0040111B                 push    edx
.text:0040111C                 push    esi
.text:0040111D                 push    eax
.text:0040111E                 call    dword ptr [ecx+2Ch]
.text:00401121                 push    7530h           ; dwMilliseconds
.text:00401126                 call    edi ; Sleep
.text:00401128                 jmp     short loc_401102
.text:0040112A ; ---------------------------------------------------------------------------
.text:0040112A
.text:0040112A loc_40112A:                             ; CODE XREF: WinMain(x,x,x,x)+D3↑j
.text:0040112A                 call    ds:OleUninitialize
.text:00401130                 pop     edi
.text:00401131
.text:00401131 loc_401131:                             ; CODE XREF: WinMain(x,x,x,x)+15↑j
.text:00401131                                         ; WinMain(x,x,x,x)+B1↑j
.text:00401131                 xor     eax, eax
.text:00401133                 pop     esi
.text:00401134                 add     esp, 28h
.text:00401137                 retn    10h
.text:00401137 _WinMain@16     endp
```

最后调用 Sleep 函数休眠了 0x7530h 毫秒，然后就是一直循环这个代码块，直到关机为止。

WinMain 在逻辑上可以分为两部分：第一部分由 OpenSCManager 到 DeviceIoControl 之间的函数调用组成，包含加载和发送请求到内核驱动的函数。第二部分由其余的函数调用组成，这表明是一个 COM 对象的使用。

程序创建了一个叫做 Process Helper 的服务，它负责加载内核驱动：C:\Windows\System32\Lab10-03.sys。然后启动 Process Helper 服务，加载 Lab10-03.sys 到内核，并且打开句柄\\.\ProcHelper,这打开一个由 ProcHelper 驱动创建的内核设备句柄。

在 IDA pro 中打开.sys 内核文件，查看汇编代码

```
INIT:00010706 sub_10706        proc near               ; CODE XREF: DriverEntry+B↓j
INIT:00010706
INIT:00010706 SymbolicLinkName= UNICODE_STRING ptr -14h
INIT:00010706 DestinationString= UNICODE_STRING ptr -0Ch
INIT:00010706 DeviceObject     = dword ptr -4
INIT:00010706 DriverObject     = dword ptr  8
INIT:00010706
INIT:00010706                  mov     edi, edi
INIT:00010708                  push    ebp
INIT:00010709                  mov     ebp, esp
INIT:0001070B                  sub     esp, 14h
INIT:0001070E                  and     [ebp+DeviceObject], 0
INIT:00010712                  push    esi
INIT:00010713                  push    edi
INIT:00010714                  mov     edi, ds:RtlInitUnicodeString
INIT:0001071A                  push    offset aDeviceProchelp ; "\\Device\\ProcHelper"
INIT:0001071F                  lea     eax, [ebp+DestinationString]
INIT:00010722                  push    eax             ; DestinationString
INIT:00010723                  call    edi ; RtlInitUnicodeString
INIT:00010725                  mov     esi, [ebp+DriverObject]
INIT:00010728                  lea     eax, [ebp+DeviceObject]
INIT:0001072B                  push    eax             ; DeviceObject
INIT:0001072C                  push    0               ; Exclusive
INIT:0001072E                  push    100h            ; DeviceCharacteristics
INIT:00010733                  push    22h             ; DeviceType
INIT:00010735                  lea     eax, [ebp+DestinationString]
INIT:00010738                  push    eax             ; DeviceName
INIT:00010739                  push    0               ; DeviceExtensionSize
INIT:0001073B                  push    esi             ; DriverObject
INIT:0001073C                  call    ds:IoCreateDevice
```

调用了 IoCreateDevice，创建一个名为\Device\ProHelper 的设备。

```
INIT:00010742                test    eax, eax
INIT:00010744                jl      short loc_10789
INIT:00010746                mov     eax, offset sub_10606
INIT:0001074B                mov     [esi+38h], eax
INIT:0001074E                mov     [esi+40h], eax
INIT:00010751                push    offset word_107DE ; SourceString
INIT:00010756                lea     eax, [ebp+SymbolicLinkName]
INIT:00010759                push    eax             ; DestinationString
INIT:0001075A                mov     dword ptr [esi+70h], offset sub_10666
INIT:00010761                mov     dword ptr [esi+34h], offset sub_1062A
INIT:00010768                call    edi ; RtlInitUnicodeString
INIT:0001076A                lea     eax, [ebp+DestinationString]
INIT:0001076D                push    eax             ; DeviceName
INIT:0001076E                lea     eax, [ebp+SymbolicLinkName]
INIT:00010771                push    eax             ; SymbolicLinkName
INIT:00010772                call    ds:IoCreateSymbolicLink
INIT:00010778                mov     esi, eax
INIT:0001077A                test    esi, esi
INIT:0001077C                jge     short loc_10787
INIT:0001077E                push    [ebp+DeviceObject] ; DeviceObject
INIT:00010781                call    ds:IoDeleteDevice

PAGE:000106A2 ; const WCHAR SourceString
PAGE:000106A2 SourceString    dw 5Ch                ; DATA XREF: sub_1062A+E1o
PAGE:000106A4 aDosdevicesProc:
PAGE:000106A4                 unicode 0, <DosDevices\ProcHelper>,0
PAGE:000106D0                 align 40h
PAGE:000106D0 PAGE            ends
```

调用 IoCreateSymbolicLink 时，创建一个名为 \DosDevices\ProHelper 的符号链接，供用户态应用程序访问。

使用 Windbg 查找内存中的驱动

使用命令!Devobj ProcHelper 查找 DriverObject 存储的位置。

```
kd> !devobj ProcHelper
Device object (81d40bf8) is for:
 ProcHelper*** ERROR: Module load completed but symbols could not be loaded for Lab10-03.sys
 \Driver\Process Helper DriverObject 81d22f38
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e130ebec DevExt 00000000 DevObjExt 81d40cb0
ExtensionFlags (0000000000)
Device queue is not busy.
```

使用命令 dt nt!_DRIVER_OBJECT 81d22f38 查看标注的驱动对象，其中，DriverInit 是驱动初始化的操 作地址，DriverUnload 是驱动卸载时候的操作地址。在 IDA Pro 中查看 DriverUnload 时，会看到它删除了符号链表和 DriverEntry 创建的设备。

```
kd> dt nt!_DRIVER_OBJECT 81d22f38
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : 0x81d40bf8 _DEVICE_OBJECT
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xf8cdd000 Void
   +0x010 DriverSize       : 0xe00
   +0x014 DriverSection    : 0x82095a40 Void
   +0x018 DriverExtension  : 0x81d22fe0 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\Process Helper"
   +0x024 HardwareDatabase : 0x80671ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xf8cdd7cd     long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0xf8cdd62a     void  +0
   +0x038 MajorFunction    : [28] 0xf8cdd606     long  +0
```

分析主函数表中的函数

使用命令 dd 81d22f38+0x38 L1c 命令查看主函数表中表项

```
kd> dd 81d22f38+0x38 L1c
81d22f70  f8cdd606 804f454a f8cdd606 804f454a
81d22f80  804f454a 804f454a 804f454a 804f454a
81d22f90  804f454a 804f454a 804f454a 804f454a
81d22fa0  804f454a 804f454a f8cdd666 804f454a
81d22fb0  804f454a 804f454a 804f454a 804f454a
81d22fc0  804f454a 804f454a 804f454a 804f454a
81d22fd0  804f454a 804f454a 804f454a 804f454a
```

表中大部分项都是 0x804f454a 表示驱动不能处理的一个请求类型。

使用命令 ln 804f454a，看到这个函数被命名为 IopInvalidDeviceRequest，用于处理驱动无法处理的非法请求。

```
kd> ln 804f454a
(804f454a)   nt!IopInvalidDeviceRequest   |  (804f4580)   nt!IopGetDeviceAttachmentBase
Exact matches:
    nt!IopInvalidDeviceRequest = <no type information>
```

查看 wdm.h,发现偏移量 0、2、0xe 存储 Create、Close 以及 DeviceIoControl 函数。主函数表中偏移量为 0、2 的两项指向同一个函数（0xf8cdd606）。看到这个函数只调用 IofCompleteRequest,然后返回。这告诉操作系统请求成功，但是什么也没有做。主函数表中剩下的唯一函数是处理 DeviceIoControl 的请求，它是最有价值的。

```
kd> u f8cdd606
Lab10_03+0x606:
f8cdd606 8bff            mov     edi,edi
f8cdd608 55              push    ebp
f8cdd609 8bec            mov     ebp,esp
f8cdd60b 8b4d0c          mov     ecx,dword ptr [ebp+0Ch]
f8cdd60e 83611800        and     dword ptr [ecx+18h],0
f8cdd612 83611c00        and     dword ptr [ecx+1Ch],0
f8cdd616 32d2            xor     dl,dl
f8cdd618 ff1580d4cdf8    call    dword ptr [Lab10_03+0x480 (f8cdd480)]
```

查看 DeviceIoControl,看到它操纵了当前进程的 PEB。

```
kd> u f8cdd666 L10
Lab10_03+0x666:
f8cdd666 8bff            mov     edi,edi
f8cdd668 55              push    ebp
f8cdd669 8bec            mov     ebp,esp
f8cdd66b ff1590d4cdf8    call    dword ptr [Lab10_03+0x490 (f8cdd490)]
f8cdd671 8b888c000000    mov     ecx,dword ptr [eax+8Ch]
f8cdd677 0588000000      add     eax,88h
f8cdd67c 8b10            mov     edx,dword ptr [eax]
f8cdd67e 8911            mov     dword ptr [ecx],edx
f8cdd680 8b08            mov     ecx,dword ptr [eax]
f8cdd682 8b4004          mov     eax,dword ptr [eax+4]
f8cdd685 894104          mov     dword ptr [ecx+4],eax
f8cdd688 8b4d0c          mov     ecx,dword ptr [ebp+0Ch]
f8cdd68b 83611800        and     dword ptr [ecx+18h],0
f8cdd68f 83611c00        and     dword ptr [ecx+1Ch],0
f8cdd693 32d2            xor     dl,dl
f8cdd695 ff1580d4cdf8    call    dword ptr [Lab10_03+0x480 (f8cdd480)]
```

DeviceIoControl 函数做的第一件事情就是调用 IoGetCurrentProcess,它返回调用 DeviceIoControl 进程的 EPROCESS 结构。然后，这个函数在处访问偏移量 0x88 处的数据，再然后，访问偏移量 0x8C 处的下一个 DWORD。

使用 dt 命令发现存储在 PEB 结构偏移量 0x88 和 0x8C 的 LIST_ENTRY。

```
kd>dt nt!_EPROCESS
+0x000 Pcb：KPROCESS
+0x06c ProcessLock：EX_PUSH_LOCK
+0x070 CreateTime：_LARGE_INTEGER
+0x078 ExitTime：_LARGE_INTEGER
+0x080 RundownProtect：_EX_RUNDOWN_REF
```

```
+0x084 UniqueProcessId：Ptr32 Void
+0x088 ActiveProcessLinks：_LIST_ENTRY
+0x090 QuotaUsage：[3]Uint4B
+0x09c QuotaPeak：[3]Uint4B
+0x0a8 CommitCharge：Uint4B
+0x0ac PeakVirtualSize：Uint4B
+0x0b0 VirtualSize：Uint4B
+0x0b4 SessionProcessLinks :_LIST_ENTRY
+0x0bc DebugPort：Ptr32 Void
+0x0c0 ExceptionPort：Ptr32 Void
+0x0c40bjectTable：Ptr32 _HANDLE_TABLE
```

LIST_ENTRY 结构是一个包含两个值的双向链表：第一个是 BLINK,它指向列表中的前一项，第二个是 FLINK,它指向列表中的下一项。程序不仅读取 LIST_ENTRY 结构而且修改这个结构。

```
mov eax,[eax+8Ch]  ①
add eax,88h
mov edx,[eax]  ②
mov [ecx],edx  ③
mov ecx,[eax]  ④
mov eax,[eax+4]  ⑤
mov [ecx+4],eax  ⑥
```

①处指令获取列表中指向下一项的指针。②处指令获取列表中指向前一项的指针。③处的指令覆盖下一项的 BLINK 指针，使其指向前一项⑥。在此之前，下一项的 BLINK 指针指向当前项。③处的指令覆盖 BLINK 指针，从而使它跳过当前指针。④、⑤、⑥处的指令执行相同的步骤，除了覆盖列表中前一项的 FIINK 指针来跳过当前项。

除了修改当前进程的 EPROCESS 结构之外，上面代码还会修改进程链中的前一个或者后一个进程的 EPROCESS 结构。这个六条指令通过从加载进程的列表中解除链接，来隐藏当前进程。

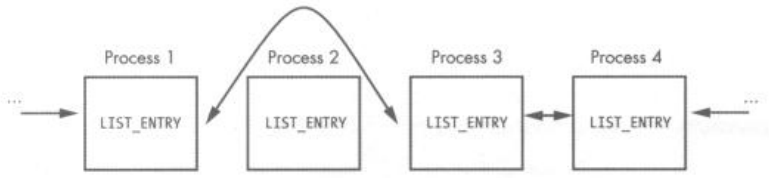


图10-3L 从进程链表中移除一个进程，从而使得它从如任务管理器的工具中隐藏

当操作系统正常运行时，每个进程都有一个指向它前一个进程或者后一个进程的指针。然后，在图 10-3L 中，进程 2 被这个 Rootkit 隐藏，当 OS 遍历进程链表时，隐藏进程总是被跳过。

1. 这个程序做了些什么?

用户态程序加载驱动，然后每隔 30 秒就弹出一个广告。这个驱动通过从系统链表中摘除进程环境块(PEB),来隐藏进程。

2. 一旦程序运行，你怎样停止它?

一旦程序运行，除了重启以外，没有任何一种办法可以轻易停止它。

3. 它的内核组件做了什么操作?

为了对用户隐藏进程，内核组件负责响应，从进程链接表中摘除进程的 DeviceIoControl 请求。

四、Yara 规则

根据特征字符串编写 yara 如下：

```
rule RukeforLab10_01exe {
   meta:
      description = "Lab10-01.exe"
   strings:
      $s1 = "C:\\Windows\\System32\\Lab10-01.sys" fullword ascii
      $s2 = "Hello World!" fullword wide
      $s3 = "RegWriterApp Version 1.0" fullword wide
      $s4 = "REGWRITERAPP" fullword wide
      $s5 = "RegWriterApp" fullword wide
```

```
      $s6 = "System" fullword wide
      $s7 = "Copyright (C) 2011" fullword wide
   condition:
      uint16(0) == 0x5a4d and
      uint32(uint32(0x3c))==0x00004550and filesize < 80KB and
      all of them
}
```

```
rule RukeforLab10_01sys {
   meta:
      description = "Lab10-01.sys"
   strings:
      $s1                                                              =
"c:\\winddk\\7600.16385.1\\src\\general\\regwriter\\wdm\\sys\\objfre_
wxp_x86\\i386\\sioctl.pdb" fullword ascii
      $s2 = "Lab10-01.sys" fullword wide
      $s3 = "Important System Driver" fullword wide
      $s4                                                              =
"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall"
fullword wide
      $s5    =    "\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft"
fullword wide
      $s6 = "6.1.7600.16385 built by: WinDDK" fullword wide
      $s7 = " ABC Corp." fullword wide
   condition:
      uint16(0) == 0x5a4d and
      uint32(uint32(0x3c))==0x00004550and filesize < 10KB and
      all of them
}
```

```
rule RukeforLab10_02 {
```

```
   meta:
      description = "Lab10-02.exe"
   strings:
      $s1 = "C:\\Windows\\System32\\Mlwx486.sys" fullword ascii
      $s2 = "c:\\winddk\\7600.16385.1\\src\\general\\rootkit\\wdm\\sys\\objfre_wxp_x86\\i386\\sioctl.pdb" fullword ascii
      $s3 = "SIOCTL.sys" fullword wide
      $s4 = "Failed to open service manager." fullword ascii
      $s5 = "Failed to start service." fullword ascii
      $s6 = "Sample IOCTL Driver" fullword wide
      $s7 = "\"WWShtT@" fullword ascii
      $s8 = "VWuBhhT@" fullword ascii
      $s9 = "486 WS Driver" fullword ascii
      $s10 = "6.1.7600.16385 built by: WinDDK" fullword wide
      $s11 = "KeServiceDescriptorTable" fullword wide
      $s16 = "Failed to create service." fullword ascii
   condition:
      uint16(0) == 0x5a4d and
      uint32(uint32(0x3c))==0x00004550and filesize < 100KB and
      6 of them
}
```

```
rule RukeforLab10_03exe {
   meta:
      description = "Lab10-03.exe"
   strings:
      $s1 = "C:\\Windows\\System32\\Lab10-03.sys" fullword ascii
      $s2 = "http://www.malwareanalysisbook.com/ad.html" fullword wide
      $s3 = "Process Helper" fullword ascii
```

```
      $s4 = "\\\\.\\ProcHelper" fullword ascii
   condition:
      uint16(0) == 0x5a4d and
      uint32(uint32(0x3c))==0x00004550and filesize < 70KB and
      all of them
}
```

```
rule RukeforLab10_03sys {
   meta:
      description = "Lab10-03.sys"
   strings:
      $s1 = "c:\\winddk\\7600.16385.1\\src\\general\\rootkitprochide\\wdm\\sys\\objfre_wxp_x86\\i386\\sioctl.pdb" fullword ascii
      $s2 = "Lab10-03.sys" fullword wide
      $s3 = "Important Process Helper" fullword wide
      $s4 = "\\DosDevices\\ProcHelper" fullword wide
      $s5 = "\\Device\\ProcHelper" fullword wide
      $s6 = "6.1.7600.16385 built by: WinDDK" fullword wide
      $s7 = " ABC Corp." fullword wide
   condition:
      uint16(0) == 0x5a4d and
      uint32(uint32(0x3c))==0x00004550and filesize < 10KB and
      all of them
}
```

五、IDA Python

遍历所有函数并显示每个函数调用的目标函数：

```
# -*- coding: utf-8 -*-


import idc
import idautils


def analyze_call_graph(start_address):
    # 创建函数调用图的字典
    call_graph = {}

    # 遍历所有函数
    for func_ea in idautils.Functions():
        func_name = idc.GetFunctionName(func_ea)
        call_graph[func_name] = []

        # 获取函数信息
        func_start = idc.GetFunctionAttr(func_ea, idc.FUNCATTR_START)
        func_end = idc.GetFunctionAttr(func_ea, idc.FUNCATTR_END)

        # 遍历函数内的指令
        for head in idautils.Heads(func_start, func_end):
            # 获取指令的助记符
            mnemonic = idc.GetMnem(head)

            # 检查指令是否为"call"
            if mnemonic == "call":
                target_address = idc.GetOperandValue(head, 0)
                target_func_name = idc.GetFunctionName(target_address)
```

```
                # 如果目标函数不为空，则将其添加到调用图中
                if target_func_name:
                    call_graph[func_name].append(target_func_name)

    # 显示调用图
    for func, calls in call_graph.items():
        print("Function: {}".format(func))
        if calls:
            print("Calls the following functions:")
            for called_func in calls:
                print("  {}".format(called_func))
        else:
            print("This function does not call other functions.")
        print("=" * 30)

# 在此处替换为要分析的起始地址
start_address_to_analyze = 0x401000  # 替换为实际地址

# 运行分析程序
analyze_call_graph(start_address_to_analyze)
```

```
  ___   _     _
============================
Function: __NMSG_WRITE
Calls the following functions:
  _strcpy
  _strlen
  _strlen
  _strncpy
  _strcpy
  _strcat
  _strcat
  _strcat
  ___crtMessageBoxA
  _strlen
============================
Function: __heap_init
Calls the following functions:
  ___sbh_heap_init
```

六、实验结论及心得体会

本次实验深入了解了恶意软件的行为分析方法，从用户态和内核态两个层面对其进行了全面的分析。通过使用多种工具和调试器，掌握了分析恶意软件的基本技能，包括查看导入函数表、资源、监视系统调用等。同时，了解了恶意软件的一些隐匿手法，如内核态的进程隐藏。这次实验对安全分析和逆向工程能力提升有很大的帮助。