



网 络 空 间 安 全 学 院

# 恶意代码分析与防治技术 实验报告

Lab 10：使用 WinDbg 调试内核



姓名：辛浩然

学号：2112514

年级：2021 级

专业：信息安全、法学

班级：信息安全、法学

实验目的

实验原理

驱动与内核代码

WinDbg

Rootkit

实验环境

安装内核调试

设置虚拟操作系统并开启内核调试

创建虚拟连接

连接虚拟机

Lab 10-01

基本静态分析

静态分析 Lab10-01.exe

静态分析 Lab10-01.sys

问题一：procmon 查看恶意代码运行时行为

IDA Pro 静态分析 Lab10-01.exe

IDA Pro 静态分析 Lab10-01.sys

问题二：使用 WinDbg 调试

问题三：程序的行为

Lab 10-02

基本静态分析

基本动态分析

问题一：恶意代码创建的文件

问题二：内核组件

问题三：恶意代码的行为

挂钩函数

被替换的原始函数

挂钩函数首先调用原始函数

传入参数具体分析

参数满足条件时挂钩函数的行为

恢复隐藏文件

Lab 10-03

基本静态分析

静态分析 Lab10-03.exe

静态分析 Lab10-03.sys

基本动态分析

IDA Pro 静态分析 Lab10-03.exe

IDA Pro 静态分析 Lab10-03.sys

使用 WinDbg 查找内存中的驱动

分析主函数表中的函数

f8cfb666处的函数

问题一：程序的行为

问题二：如何停止程序

问题三：内核组件的操作

Yara 规则编写

IDA Python 自动化分析

编写脚本查找特定函数，分析其控制流，并识别其中的基本块和交叉引用  
编写脚本查看导入的动态链接库  
实验结论及心得体会

## 实验目的

1. 熟悉 WinDBG 的功能使用，练习使用WinDbg分析内核驱动，熟练内核调试；
2. 加深对内核驱动工作原理的理解和认识；
3. 掌握Rootkit的分析方法；
4. 提高分析恶意代码的综合能力，能够使用多种工具多方面理解恶意代码的行为及原理。

## 实验原理

### 驱动与内核代码

为了系统能够正常工作，驱动程序**必须加载到内核空间**。驱动首次被加载时，首先调用**DriverEntry函数**。

驱动通过注册**回调函数**来提供功能。当用户态的应用程序请求一个服务时，这些回调函数将会被调用。

#### ※ **DriverEntry注册回调函数**。

※ Windows会为每个驱动创建一个**驱动对象**，并以参数形式将其传给**DriverEntry函数**，DriverEntry函数用**回调函数**填充这个**驱动对象**。然后DriverEntry会**创建一个可以被用户态应用程序访问的设备对象**(设备对象由驱动程序创建和销毁，不一定是真实的物理硬件)，应用程序与驱动的交互请求都将通过这个**设备对象**进行。(不直接与驱动程序通信)

比如：考虑来自用户态应用程序的一个读数据请求。最终这个请求被发送到负责管理硬件并存储读入数据的**驱动程序**。首先，用户态应用程序应该获得该硬件设备的一个**文件句柄**，然后在**该句柄上调用函数ReadFile**。接着**内核将会处理ReadFile函数的请求**，最终**由驱动程序的回调函数来响应**对I/O设备的读请求。

**请求内核态恶意组件**的最常见函数是**DeviceIoControl**,它是从用户态模块到内核设备的一种通用请求方法。使用该函数时，用户态应用程序传递一个任意长度的缓冲区数据作为**输入**，并且接收一个任意长度的缓冲区数据作为**输出**。

**用户态应用程序到内核态驱动的调用由操作系统完成**，这种调用难以被跟踪。

※ 恶意驱动做了什么：恶意驱动通常不控制硬件设备，而是与Windows操作系统主要的**内核组件ntoskrnl.exe、hal.dll进行交互**。ntoskrnl.exe组件包含**操作系统核心功能的代码**，hal.dll包含**与主要硬件设备交互的代码**。恶意代码常通过从一个或者多个这样的**内核组件中导入函数，来操纵内核**。

## WinDbg

WinDbg是一个出色的调试器，它支持用户调试和内核调试。

## Rootkit

Rootkit通过修改操作系统内部函数，来隐藏自己存在的痕迹。通过这种修改，Rootkit可以隐藏一个正在运行程序的文件、进程、网络连接以及其他资源。这使得其恶意活动难以被反病毒产品、管理员以及安全分析员发现。

现在大部分Rootkit都是通过采用某种方式修改操作系统内核来工作的。尽管Rootkit可以使用多种隐藏技术，但在实际应用中，系统服务描述表(SSDT)挂钩技术的使用程度远远超过其他技术。这种技术已经有几年的历史，与其他rootkits技术相比，它更容易被探测。然而，由于它容易理解、实现灵活且容易，因此到现在它依然被恶意代码所使用。

## 实验环境

虚拟机：开启内核调试的关闭病毒防护的Windows XP SP3；每次病毒分析前拍摄快照，并在分析后恢复快照。

宿主机：Windows 11。

分析工具：主要使用WinDbg、IDA Pro，还包括Strings、procmon、Process Explorer等分析工具。

## 安装内核调试

### ▲ 设置虚拟操作系统并开启内核调试

编辑 `C:\boot.ini`：

```
boot.ini - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional" /noexecute=optin /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="Microsoft Windows XP Professional with Kernel Debugging" /noexecute=optin /fastdetect /debug /debugport=COM1 /baudrate=115200
```

当下次开机运行虚拟操作系统时，系统会提供开启内核调试的选项。

## ▲ 创建虚拟连接

设置VMware，在虚拟操作系统和宿主操作系统之间创建一个虚拟连接。为此，在VMware上添加一个新的设备来使用宿主系统中的一个命名管道上的串口。

The screenshot shows the 'Device State' (设备状态) section with the 'Connect on startup' (启动时连接(O)) checkbox checked. In the 'Connection' (连接) section, the 'Use named pipe' (使用命名的管道(N)) radio button is selected. The pipe path is set to '\\.\pipe\com\_1'. The 'I/O Mode' (I/O 模式) section shows the 'Round-robin' (轮询) mode selected.

设备状态

☐ 已连接(C)  
☒ 启动时连接(O)

连接

☐ 使用物理串行端口(U):  
自动检测

☐ 使用输出文件(S):  
浏览(B)...

☒ 使用命名的管道(N):  
\\.\pipe\com\_1  
该端是服务器。  
另一端是虚拟机。

I/O 模式

☐ 轮询时主动放弃 CPU(Y)  
允许客户机操作系统在轮询模式下 (而不是中断模式)使用此串行端口。

## ▲ 连接虚拟机

完成虚拟机的配置后启动虚拟机。在宿主操作系统中，进行相应设置，使WinDbg连接虚拟机并开始调试内核。

The screenshot shows the 'Serial Connection' dialog box in WinDbg. The 'Pipe' checkbox is checked. The 'Port' field is set to '\\.\pipe\com\_1'. The 'Baud Rate' is set to 115200. The 'Resets' field is set to 0. The 'Break on connection' checkbox is unchecked. A note at the bottom states: 'Note: Connecting to a virtual COM named pipe may require elevation. Kernel debugging using a serial connection is not recommended. Using network kernel debugging is faster and more reliable.'

EXDI 1394 Paste connection string  
Net COM Local USB

☒ Pipe  
☐ Reconnect

Resets  
0

Baud Rate  
115200

Port  
\\.\pipe\com\_1

☐ Break on connection

Note: Connecting to a virtual COM named pipe may require elevation.

Kernel debugging using a serial connection is not recommended. Using [network kernel debugging](#) is faster and more reliable.

# Lab 10-01

- 本实验包括一个驱动程序和一个可执行文件。驱动程序Lab10-01.sys需要放在C:\Windows\System32目录下。

## 基本静态分析

### 静态分析 Lab10-01.exe

首先查看恶意代码文件的导入表。发现它导入了 `StartServiceA`、`OpenServiceA`、`CreateServiceA`、`ControlService` 等函数。这说明，恶意代码创建了一个服务，并对这个服务进行了一些操作。

Address	Ordinal	Name	Library
000000000004...		StartServiceA	ADVAPI32
000000000004...		OpenServiceA	ADVAPI32
000000000004...		CreateServiceA	ADVAPI32
000000000004...		OpenSCManagerA	ADVAPI32
000000000004...		ControlService	ADVAPI32
000000000004...		GetModuleHandleA	KERNEL32
000000000004...		GetStartupInfoA	KERNEL32
000000000004...		GetCommandLineA	KERNEL32
000000000004...		GetVersion	KERNEL32
000000000004...		ExitProcess	KERNEL32

接下来查看字符串信息。

.data:00405030	00000009	C	Lab10-01
.data:0040503C	00000021	C	C:\\Windows\\System32\\Lab10-01.sys
.data:004052BC	00000040	C	(((

查看到一个字符串 `C:\\Windows\\System32\\Lab10-01.sys`，这说明Lab10-01.sys可能包含整个服务的代码，可能是通过驱动的方式加载恶意代码。

### 静态分析 Lab10-01.sys

查看驱动文件的导入表。`KeTickCount` 函数几乎所有的驱动都会包含它。`RtlCreateRegistryKey` 和 `RtlWriteRegistryValue` 函数，表明驱动可能访问了注册表。

- `RtlCreateRegistryKey`：通过一个给定的注册表相对路径和值创建指定的键。
- `RtlWriteRegistryValue`：将提供的数据以指定的值名称写入指定的相对路径。

Address	Ordinal	Name	Library
000000000000...		RtlCreateRegistryKey	ntoskrnl
000000000000...		KeTickCount	ntoskrnl
000000000000...		RtlWriteRegistryValue	ntoskrnl



查看字符串，看到一些类似注册表键值的字符串。开头的 `\Registry\Machine` 不像是诸如 `HKLM` 此类的常见注册表根键。这是因为，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`。

此外，还看到字符串 `EnableFirewall`，`EnableFirewall` 如果为0，那么Windows XP的防火墙将被关闭。因此，猜测可能会关闭防火墙。

```
.text:000104F0      0000001C      C (16 bits) - UTF-16LE      nableFirewall
.text:0001050C      0000009C      C (16 bits) - UTF-16LE      \\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewa...
.text:000105A8      00000098      C (16 bits) - UTF-16LE      \\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewa...
.text:00010640      0000007C      C (16 bits) - UTF-16LE      \\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall
.text:000106BC      0000005C      C (16 bits) - UTF-16LE      \\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft
.rdata:000107C4      00000054      C                          c:\\winddk\\7600.16385.1\\src\\general\\regwriter\\wdm\\svs\\obif...
```

## 问题一：procmon 查看恶意代码运行时行为

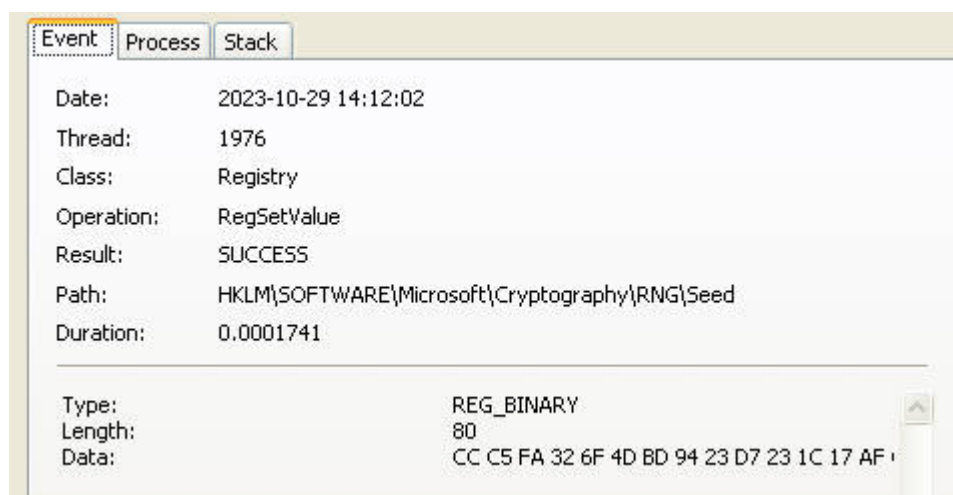
- 问题一：这个程序是否直接修改了注册表(使用procmon来检查)?

动态运行恶意代码并使用procmon监控恶意代码的行为。设置进程名为Lab10-01.exe的过滤器。

Column	Relation	Value	Action
Process Name	is	Lab10-01.exe	Include
Operation	is	RegSetValue	Include

过滤到唯一一条写注册表的行为：

`RegSetValue` 写了 `HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed` 键值。这个注册表键值一直改变，对于恶意代码分析没有什么意义。



调用 `CreateServiceA` 也可以对注册表的一些间接修改，也可以从内核对注册表进行直接修改，这都是procmon所探测不到的。

## IDA Pro 静态分析 Lab10-01.exe

在IDA Pro中加载Lab10-01.exe，分析其代码。

下面是main函数的主要代码。可以看到，恶意代码有如下行为：

1. 调用 `OpenSCManagerA` 获取服务管理器的句柄；
2. 调用 `CreateServiceA` 创建名为Lab10-01的服务，并将它添加到指定的服务控制管理程序的数据库中。其中：
  - `dwStartType` 为3，说明服务会自启动；
  - `dwErrorControl` 为1，表示如果这项服务无法启动，启动程序在事件日志记录，但继续启动操作；
  - `dwServiceType` 为1，即 `SERVICE_KERNEL_DRIVER`，这表明会被加载到内核；
  - `BinaryPathName` 表示服务二进制文件的完全限定路径，即 `C:\\Windows\\System32\\Lab10-01.sys`。

```
.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      ; -----
.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:0040102D      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
```

如果同名服务存在，导致服务创建失败，那么调用 `OpenServiceA` 函数打开同名服务。

```
.text:00401052      push    0F01FFh      ; dwDesiredAccess
.text:00401057      push    offset ServiceName ; "Lab10-01"
.text:0040105C      push    edi          ; hSCManager
.text:0040105D      call    ds:OpenServiceA
```

如果打开成功，则调用 `StartServiceA` 函数启动服务。最后，调用 `ControlService`，第二个参数是发送控制消息的类型，值为0x010，即 `SERVICE_CONTROL_STOP`。这将会卸载驱动，并调用驱动的卸载函数。



```

.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      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

```

## IDA Pro 静态分析 Lab10-01.sys

查看驱动文件的 `DriverEntry` 函数( `sub_00010906` ), 它将一个偏移量移入一个内存位置, 然后没有任何函数调用。

```

INIT:00010906 ; ===== SUBROUTINE =====
INIT:00010906
INIT:00010906 ; Attributes: bp-based frame
INIT:00010906 ; NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject, PUNICODE_STRING RegistryPath)
INIT:00010906 _DriverEntry@8 proc near ; CODE XREF: DriverEntry+B↓j
INIT:00010906
INIT:00010906 DriverObject = dword ptr 8
INIT:00010906 RegistryPath = dword ptr 0Ch
INIT:00010906
INIT:00010906      mov     edi, edi
INIT:00010908      push    ebp
INIT:00010909      mov     ebp, esp
INIT:0001090B      mov     eax, [ebp+DriverObject]
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 _DriverEntry@8 endp
INIT:00010918

```

查看 `sub_10486` 能够看到大量对注册表的操作。分析可知, 通过修改注册表键值, 关闭了防火墙。

```

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
push    ebx
push    esi
mov     esi, ds:RtlCreateRegistryKey
push    edi
xor     edi, edi
push    offset Path      ; "\\Registry\\Machine\\SOFTWARE\\Policies"...
push    edi              ; RelativeTo
mov     [ebp+ValueData], edi
call    esi ; RtlCreateRegistryKey
push    offset aRegistryMachin_0 ; "\\Registry\\Machine\\SOFTWARE\\Policies"...
push    edi              ; RelativeTo
call    esi ; RtlCreateRegistryKey
push    offset aRegistryMachin_1 ; "\\Registry\\Machine\\SOFTWARE\\Policies"...
push    edi              ; RelativeTo
call    esi ; RtlCreateRegistryKey
mov     ebx, offset aRegistryMachin_2 ; "\\Registry\\Machine\\SOFTWARE\\Policies"...
push    ebx              ; Path
push    edi              ; RelativeTo
call    esi ; RtlCreateRegistryKey
mov     esi, ds:RtlWriteRegistryValue
push    4                ; ValueLength
lea     eax, [ebp+ValueData]
push    eax              ; ValueData
push    4                ; ValueType
mov     edi, offset ValueName
push    edi              ; ValueName
push    offset aRegistryMachin_1 ; "\\Registry\\Machine\\SOFTWARE\\Policies"...
push    0                ; RelativeTo
call    esi ; RtlWriteRegistryValue
push    4                ; ValueLength
lea     eax, [ebp+ValueData]
push    eax              ; ValueData

```

## 问题二：使用 WinDbg 调试

问题二：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？

整体来说，必须使用一个运行在虚拟机中的WinDbg实例打开这个可执行文件，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。接下来是具体过程：

在虚拟机中，将可执行程序载入到WinDbg中。

使用命令 `bp 00401080`，在驱动加载和卸载之间设置一个断点，即在 `ControlService` 调用上。

```
.text:00401080      call     ds:ControlService
```

之后，启动程序直到断点命中。当断点命中时，在WinDbg展示了如下的信息：

```

C:\Documents and Settings\Administrator\桌面\Practical Malwar
eax=00241eb4 ebx=7ffde000 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
ntdll!DbgBreakPoint:
7c92120e cs:001b int 3
0:000> !m
start      end      module name
00400000 00407000  image00400000 (deferred)
77da0000 77e49000  ADVAPI32 (deferred)
77e50000 77ee2000  RPCRT4 (deferred)
77fc0000 77fd1000  Secur32 (deferred)
7c800000 7c91e000  kernel32 (deferred)
7c920000 7c9b3000  ntdll (pdb symbols) c:\windows\symbols\dll\ntdll.p
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=7ffde000 ecx=77dbfb6d edx=00000000 esi=00144070 edi=00144f80
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:
00401080 ff1510404000 call dword ptr [image00400000+0x4010 (00404010)] ds:0023

```

接下来，在宿主机对虚拟机进行调试。

使用命令 `!drvobj` 获取驱动对象：

```

kd> !drvobj lab10-01
Driver object (8206cca8) is for:
  \Driver\Lab10-01

Driver Extension List: (id , addr)

Device Object list:

```

输出的信息提供了驱动对象的地址 `8206cca8`。在设备对象列表中没有设备列出，这个驱动没有供用户空间中应用程序访问的设备。

查看该驱动对象，使用命令 `dt 8206cca8`：

```

kd> dt _DRIVER_OBJECT 8206cca8
ntdll!_DRIVER_OBJECT
+0x000 Type           : 0n4
+0x002 Size           : 0n168
+0x004 DeviceObject   : (null)
+0x008 Flags          : 0x12
+0x00c DriverStart    : 0xf8dab000 Void
+0x010 DriverSize     : 0xe80
+0x014 DriverSection  : 0x820cb138 Void
+0x018 DriverExtension : 0x8206cd50 _DRIVER_EXTENSION
+0x01c DriverName     : _UNICODE_STRING "\Driver\Lab10-01"
+0x024 HardwareDatabase : 0x80671ae0 _UNICODE_STRING "\\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch : (null)
+0x02c DriverInit     : 0xf8dab959 long +0
+0x030 DriverStartIo  : (null)
+0x034 DriverUnload   : 0xf8dab486 void +0
+0x038 MajorFunction  : [28] 0x804f454a long nt!IopInvalidDeviceRequest+0

```

重点关注函数 `DriverUnload`，这是驱动卸载时调用的函数。这个函数的地址为 `f8dab486`，在该处加断点，并使用 `g` 指令恢复内核的执行。

```

kd> bp 0xf8dab486
kd> g

```

由于内核调试器命中了断点，虚拟机会卡死。此时，回到内核调试器，单步调试代码。

```
kd> g
Break instruction exception - code 80000003 (first chance)
Lab10_01+0x486:
f8dab486 8bff          mov     edi,edi
kd> t
Lab10_01+0x488:
f8dab488 55           push    ebp
kd> t
Lab10_01+0x489:
f8dab489 8bec          mov     ebp,esp
kd> t
Lab10_01+0x48b:
f8dab48b 51           push    ecx
kd> t
Lab10_01+0x48c:
f8dab48c 53           push    ebx
```

可以看到程序调用了三次 `RtlCreateRegistryKey` 函数，创建了一些注册表键，然后调用了两次 `RtlWriteRegistryValue` 函数，在两个地方设置 `EnableFirewall` 值为0，从而从内核禁用Windows XP防火墙。

接下来，可以使用IDA Pro进行分析。从前面得知 `DriverStart` 的地址和 `DriverUpload` 的地址，从而得到偏移量 `0x486`。在IDA Pro中加载基址为 `0x00100000`，所以函数卸载代码对应的地址为 `0x00100468`。这就是前面分析过的 `sub_10486` 函数。

```

mov     edi, edi
push    ebp
mov     ebp, esp
push    ecx
push    ebx
push    esi
mov     esi, ds:RtlCreateRegistryKey
push    edi
xor     edi, edi
push    offset Path      ; "\\Registry\\Machine\\SOFTWARE\\Policies"...
push    edi              ; RelativeTo
mov     [ebp+ValueData], edi
call    esi ; RtlCreateRegistryKey
push    offset aRegistryMachin_0 ; "\\Registry\\Machine\\SOFTWARE\\Policies"...
push    edi              ; RelativeTo
call    esi ; RtlCreateRegistryKey
push    offset aRegistryMachin_1 ; "\\Registry\\Machine\\SOFTWARE\\Policies"...
push    edi              ; RelativeTo
call    esi ; RtlCreateRegistryKey
mov     ebx, offset aRegistryMachin_2 ; "\\Registry\\Machine\\SOFTWARE\\Policies"...
push    ebx              ; Path
push    edi              ; RelativeTo
call    esi ; RtlCreateRegistryKey
mov     esi, ds:RtlWriteRegistryValue
push    4                ; ValueLength
lea     eax, [ebp+ValueData]
push    eax              ; ValueData
push    4                ; ValueType
mov     edi, offset ValueName
push    edi              ; ValueName
push    offset aRegistryMachin_1 ; "\\Registry\\Machine\\SOFTWARE\\Policies"...
push    0                ; RelativeTo
call    esi ; RtlWriteRegistryValue
push    4                ; ValueLength
lea     eax, [ebp+ValueData]
push    eax              ; ValueData

```

它创建了注册表键

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

和

`\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile` ,

设置这些键值会禁用防火墙。

### 问题三：程序的行为

- 问题三：这个程序做了什么？

这个程序首先创建服务，来加载驱动；然后恶意程序会在内核禁用XP防火墙，实现的方法是：创建了注册表键

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

和

`\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile` ,

设置这些键值会禁用防火墙。从内核禁用Windows XP防火墙的这种方法是难以被安全程序探测到的。



## Lab 10-02

### 基本静态分析

首先进行基本静态分析。查看可执行文件的导入表：

- `CreateServiceA`、`StartServiceA` 等函数表明：程序创建并启动文件；

000000000004...	CreateServiceA	ADVAPI32
000000000004...	StartServiceA	ADVAPI32
000000000004...	CloseServiceHandle	ADVAPI32
000000000004...	OpenSCManagerA	ADVAPI32

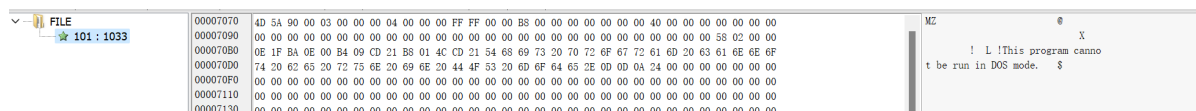
- `CreateFileA`、`WriteFile` 等函数表明：程序会创建、写文件；

000000000004...	CreateFileA	KERNEL32
000000000004...	SizeofResource	KERNEL32
000000000004...	WriteFile	KERNEL32

- `LoadResource` 等函数表明：程序会访问资源节并做出处理。

000000000004...	FindResourceA	KERNEL32
000000000004...	LoadResource	KERNEL32

使用 `Resource Hacker` 检查资源节，看到了资源节包含了另外一个PE头部，这可能是 Lab10-02将要使用的另外一个恶意文件。



### 基本动态分析

运行程序，使用 Regshot 工具比较运行前后的快照。可以发现，增加了一个名为 `486 WS Driver` 的服务，并对其细节进行配置。

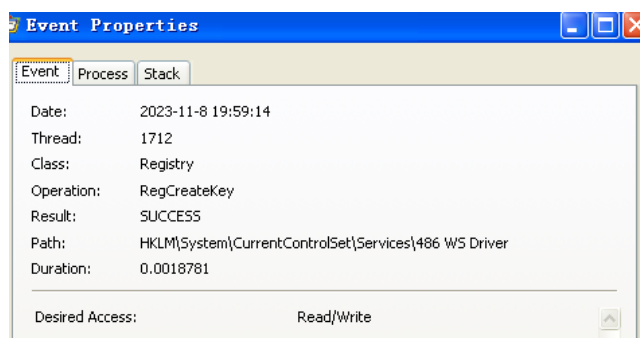
#### Keys added: 14

```
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_PROCMON20\0000\Control
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_486_WS_DRIVER
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_486_WS_DRIVER\0000
HKLM\SYSTEM\ControlSet001\Enum\Root\LEGACY_486_WS_DRIVER\0000\Control
HKLM\SYSTEM\ControlSet001\Services\486 WS Driver
HKLM\SYSTEM\ControlSet001\Services\486 WS Driver\Security
HKLM\SYSTEM\ControlSet001\Services\486 WS Driver\Enum
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_PROCMON20\0000\Control
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_486_WS_DRIVER
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_486_WS_DRIVER\0000
HKLM\SYSTEM\CurrentControlSet\Enum\Root\LEGACY_486_WS_DRIVER\0000\Control
HKLM\SYSTEM\CurrentControlSet\Services\486 WS Driver
HKLM\SYSTEM\CurrentControlSet\Services\486 WS Driver\Security
HKLM\SYSTEM\CurrentControlSet\Services\486 WS Driver\Enum
```



在procmon中，发现 `services.exe` 执行了 `RegCreateKey`，路径与 `Regshot` 中的路径相同。

19:5...	services.exe	676	RegOpenKey	HKLM\System\CurrentControlSet\Services	SUCCESS	Desired Acces...
19:5...	services.exe	676	RegCreateKey	HKLM\System\CurrentControlSet\Services\486 WS Driver	SUCCESS	Desired Acces...
19:5...	services.exe	676	SetPathName	C:\WINDOWS\system32\config\system LOG	SUCCESS	PathName: 53



搜索这个名为 `services.exe`、PID为676的程序的其它行为：

设置筛选条件为 `WriteFile` 之后，就会发现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`。

Time	Process Name	PID	Operation	Path	Result	Detail
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 0,
9:5...	services.exe	676	WriteFile	C:\\$Directory	SUCCESS	Offset: 0,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 512,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 1,5
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 5,6
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 13,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 17,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 22,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 26,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 34,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 38,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 46,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 50,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 54,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 58,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 62,
9:5...	services.exe	676	WriteFile	C:\\$Directory	SUCCESS	Offset: 67,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system.LOG	SUCCESS	Offset: 0,
9:5...	services.exe	676	WriteFile	C:\\$Directory	SUCCESS	Offset: 0,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 0,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 851,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 0,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 2,4
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 2,8
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 1,9
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 2,7
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 3,0
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 2,2
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 802,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 2,9
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 3,0
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 2,8
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 2,8
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\system	SUCCESS	Offset: 0,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\SysEvent.Evt	SUCCESS	Offset: 96,
9:5...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\SysEvent.Evt	SUCCESS	Offset: 96,
0:0...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\SysEvent.Evt	SUCCESS	Offset: 96,
0:0...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\SysEvent.Evt	SUCCESS	Offset: 96,
0:0...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\SysEvent.Evt	SUCCESS	Offset: 97,
0:0...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\AppEvent.Evt	SUCCESS	Offset: 113,
0:0...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\AppEvent.Evt	SUCCESS	Offset: 113,
0:0...	services.exe	676	WriteFile	C:\WINDOWS\system32\config\AppEvent.Evt	SUCCESS	Offset: 113,

## 问题一：恶意代码创建的文件

- 这个程序创建文件了吗？它创建了什么文件？

接下来，在procmon中重新设置过滤器：进程名称为 `Lab10-02.exe`

Ti...	Process Name	PID	Operation	Path	Result	Detail
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\Prefetch\Labio-02.EXE-14D1DD0D.pf	NAME NOT FOUND	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\Documents and Settings\Administrator\桌面\Practical Malware Analysis Labs\BinaryC...	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32\conime.exe	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32\apphelp.dll	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32\apphelp.dll	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\AppPatch\sysmain.sdb	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\AppPatch\sysrest.sdb	NAME NOT FOUND	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32\conime.exe	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32\Mimuz486.sys	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32\conime.exe	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32\conime.exe	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32	SUCCESS	Desired Acces...
19:5...	Labio-02.exe	1728	CreateFile	C:\WINDOWS\system32\conime.exe.Manifest	NAME NOT FOUND	Desired Acces...

在procmon中查看创建的文件 `conime.exe` 的行为:

Ti...	Process Name	PID	Operation	Path	Result	Detail
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\Prefetch\Lab10-02.EXE-14D1DD10D.pf	NAME NOT FOUND	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\Documents and Settings\Administrator\桌面\Practical Malware Analysis Labs\BinaryC...	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32\conime.exe	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32\apphelp.dll	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32\apphelp.dll	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\AppPatch\Sysmain.sdb	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\AppPatch\Systest.sdb	NAME NOT FOUND	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32\conime.exe	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32\Mimuz486.sys	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32\conime.exe	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32\conime.exe	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32	SUCCESS	Desired Acces...
19:5...	Lab10-02.exe	1728	CreateFile	C:\WINDOWS\system32\conime.exe.Manifest	NAME NOT FOUND	Desired Acces...

## 问题二：内核组件

- 这个程序有内核组件吗？

恶意代码同样创建了文件 `Mlwx486.sys`，但试图在文件路径 `C:\Windows\System32` 下查找 `Mlwx486.sys`，没有找到。可能恶意代码对其进行了隐藏。

源程序是怎么创建该驱动文件的呢？在IDA Pro中分析：

```

.text:00401018      push    ds             ; hModule
.text:00401018      call   ds:LoadResource
.text:0040101E      test   edi, edi
.text:00401020      mov    ebx, eax
.text:00401022      jz     loc_4010FF
.text:00401028      push    0              ; hTemplateFile
.text:0040102A      push    80h ; '€'      ; dwFlagsAndAttributes
.text:0040102F      push    2              ; dwCreationDisposition
.text:00401031      push    0              ; lpSecurityAttributes
.text:00401033      push    0              ; dwShareMode
.text:00401035      push    0C000000h      ; dwDesiredAccess
.text:0040103A      push    offset BinaryPathName ; "C:\\Windows\\System32\\Mlwx486.sys"
.text:0040103F      call   ds:CreateFileA
.text:00401045      mov    esi, eax
.text:00401047      cmp    esi, 0FFFFFFFh
.text:0040104A      jz     loc_4010FF

```

可以发现，先是从资源中去除，然后将其写入到路径为

`C:\\Windows\\System32\\Mlwx486.sys` 文件，然后创建服务、打开服务。

这说明，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，最后它作为一个服务加载到内核。

### 问题三：恶意代码的行为

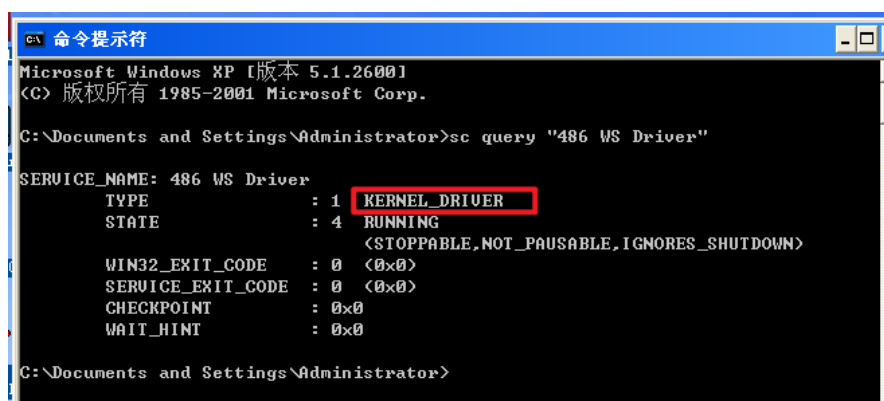
这个程序做了些什么？

总的来说，这是用来隐藏文件的 `RootKit`，它使用 `SSDT` 来挂钩覆盖

`NtQueryDirectoryFile` 函数，会隐藏任何以 `Mlwx` 开头的文件。下面是分析过程。

前面说，恶意代码创建了名为 `486 WS Driver` 的服务，这应该是运行内核驱动程序的服务，使用以下命令对其进行检查：

```
sc query "486 WS Driver"
```



```

C:\Documents and Settings\Administrator>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:\Documents and Settings\Administrator>

```

很明显这个是内核的驱动 (`KERNEL_DRIVER`)。这个服务仍然运行，这说明内核代码在内存中。可是驱动仍然运行，但是它没有在硬盘上。

看到一个与 `Lab10-02.exe` 创建文件名匹配的条目。

```

f8cc5000 f8cc5d80  Mlwx486 (deferred)

```

可以确定文件名为 `Mhwx486.sys` 的驱动被载入到内存，但是文件没有在硬盘上显示，这暗示了这个程序可能是一个Rootkit。

## 挂钩函数

接下来，使用如下命令，检查SSDT的所有修改项。

```
dd dwo(KeServiceDescriptorTable) L100
```

其中，发现异常地址 `f8cc5486`，很明显这个地址位于 `ntoskrnl` 模块的范围之外，位于加载的 `Mhwx486.sys` 驱动内。

```
80502dbc 8060db50 8060db50 8053d02e 80607e68
80502dcc 80608ac8 f8cc5486 805b4de0 805703ca
80502ddc 806063a4 8056d222 8060d2dc 80570c46
```

## 被替换的原始函数

为了确定替换了哪个函数，恢复虚拟机到运行恶意代码之前的状态，并再次检查调试检查。

可以发现，在恶意代码运行之前，被异常地址 `f8cc5486` 替换的是地址 `80570074`。

```
80502dbc 8060db50 8060db50 8053d02e 80607e68
80502dcc 80608ac8 80570074 805b4de0 805703ca
```

而查看该地址处的函数为 `NtQueryDirectoryFile`，这就是被覆盖的函数。

```
kd> u 0xfffffffff80570074
nt!NtQueryDirectoryFile:
80570074 8bff mov edi,edi
80570076 55 push ebp
80570077 8bec mov ebp,esp
80570079 8d452c lea eax,[ebp+2Ch]
8057007c 50 push eax
8057007d 8d4528 lea eax,[ebp+28h]
80570080 50 push eax
80570081 8d4524 lea eax,[ebp+24h]
```

这个函数是一个提取文件和目录信息的通用函数，`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，这也解释了之前不能发现 `Mhwx486.sys` 的原因。

## ▲ 挂钩函数首先调用原始函数

下面详细分析替换后的挂钩函数。在挂钩函数的起始处，它先入栈一些参数。然后调用一个函数。

```
Mlwx486+0x495:
f8cc5493 ff752c          push    dword ptr [ebp+2Ch]
kd> t
Mlwx486+0x496:
f8cc5496 ff7528          push    dword ptr [ebp+28h]
kd> t
Mlwx486+0x499:
f8cc5499 ff7524          push    dword ptr [ebp+24h]
kd> t
Mlwx486+0x49c:
f8cc549c ff7520          push    dword ptr [ebp+20h]
kd> t
Mlwx486+0x49f:
f8cc549f 56             push    esi
kd> t
Mlwx486+0x4a0:
f8cc54a0 ff7518          push    dword ptr [ebp+18h]
kd> t
Mlwx486+0x4a3:
f8cc54a3 ff7514          push    dword ptr [ebp+14h]
kd> t
Mlwx486+0x4a6:
f8cc54a6 ff7510          push    dword ptr [ebp+10h]
kd> t
Mlwx486+0x4a9:
f8cc54a9 ff750c          push    dword ptr [ebp+0Ch]
kd> t
Mlwx486+0x4ac:
f8cc54ac ff7508          push    dword ptr [ebp+8]
kd> t
Mlwx486+0x4af:
f8cc54af e860000000      call    Mlwx486+0x514 (f8cc5514)
kd> t
Mlwx486+0x514:
f8cc5514 ff258055ccf8    jmp     dword ptr [Mlwx486+0x580 (f8cc5580)]
```

函数进入后，首先 `jmp` 跳转至一个地址，而单步调试，接下来所执行的就是被替换的函数 `NtQueryDirectoryFile`。

```
jmp dword ptr[Mlwx486+0x580 (f7ab7580)]
```

```
kd> t
nt!NtQueryDirectoryFile:
80570074 8bff          mov     edi,edi
```

这说明，挂钩函数首先使用原始的参数值调用原始的 `NtQueryDirectoryFile` 函数。

## ▲ 传入参数具体分析

下面分析传入的参数值：

根据 MSDN 的文档，结合入栈的参数值，列出 `NtQueryDirectoryFile` 的定义：

```
NTSTATUS ZwQueryDirectoryFile(  
    _In_      HANDLE      FileHandle = 464,  
    _In_opt_ HANDLE      Event = 0,  
    _In_opt_ PIO_APC_ROUTINE ApcRoutine = 0,  
    _In_opt_ PVOID       ApcContext = 0,  
    _Out_     PIO_STATUS_BLOCK IoStatusBlock = 68 e1 70,  
    _Out_     PVOID       FileInformation = 0070e198,  
    _In_      ULONG       Length = 268,  
    _In_      FILE_INFORMATION_CLASS FileInformationClass = 3,  
    _In_      BOOLEAN     ReturnSingleEntry = 1,  
    _In_opt_ PUNICODE_STRING FileName = 80 e1 70,  
    _In_      BOOLEAN     RestartScan = 0  
);
```

得到隐藏文件后，使用IDA Pro分析挂钩函数

```
.text:000104B4      xor     edi, edi  
.text:000104B6      cmp     [ebp+FileInformationClass], 3  
.text:000104BA      mov     dword ptr [ebp+RestartScan], eax  
--.text:000104BD      jnz     short loc_10505  
.text:000104BF      test    eax, eax  
--.text:000104C1      jl      short loc_10505  
.text:000104C3      cmp     [ebp+ReturnSingleEntry], 0  
--.text:000104C7      jnz     short loc_10505  
.text:000104C9      push    ebx
```

它会检查第8个参数 `FileInformationClass`，如果它是3之外的值，便会返回 `NtQueryDirectoryFile` 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`。

挂钩函数会查找某个参数，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，则挂钩函数将会修改返回值。

## ▲ 参数满足条件时挂钩函数的行为

设置一个断点，分析一下在参数满足条件时，挂钩函数的行为。因此进行如下设置，只有当 `ReturnSingleEntry` 为0时，断点才会被触发。

```
bp f8cc5486 ".if dwo(esp+0x24)==0 {} .else {gc}"
```

继续执行，直至触发断点。



```
Break instruction exception - code 80000003 (first chance)
Mlwx486+0x486:
f8cc5486 8bff          mov     edi,edi
```

单步执行，现在注意到以下这些代码，调用了 `RtlCompareMemory` 函数。

```
Mlwx486+0x4ca:
f8cc54ca 6a08          push    8
<d> t
Mlwx486+0x4cc:
f8cc54cc 681a55ccf8    push    offset Mlwx486+0x51a (f8cc551a)
<d> t
Mlwx486+0x4d1:
f8cc54d1 8d465e        lea     eax,[esi+5Eh]
<d> t
Mlwx486+0x4d4:
f8cc54d4 50            push    eax
<d> t
Mlwx486+0x4d5:
f8cc54d5 32db          xor     bl,bl
<d> t
Mlwx486+0x4d7:
f8cc54d7 ff159055ccf8  call   dword ptr [Mlwx486+0x590 (f8cc5590)]
<d> t
nt!RtlCompareMemory
50542ea0 50            push    esi
```

调用前，首先压入字符串：

```
push offset Mlwa486+0x51a(f8cc551a)
```

而这个值，查到是 `Mlwx`。

```
4d 00 6c 00 77 00 78 00-00 00 00 00 00 00 00 00 M.l.w.x.....
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

还压入了 `eax` 作为参数，而 `eax` 存储了偏移量 `esi+5eh`，这是文件名的偏移量。它包含 `NtQueryDirectoryFile` 填充的信息。查阅相关文档，了解到这是一个 `FILE_BOTH_DIR_INFORMATION` 结构，则 `0x5E` 偏移量是存储宽字符串文件名的地址。所以得到的就是各个文件名。

再通过WinDbg查看参数的意义是什么。

```
64 00 65 00 73 00 6b 00-74 00 6f 00 70 00 2e 00 d.e.s.k.t.o.p..
69 00 6e 00 69 00 00 00-00 18 00 00 00 00 00 00 i.n.i.....
00 00 4a 65 37 f7 04 f1-d9 01 1e 5f d3 54 a3 12 ..Je7....._T..
da 01 c4 cb f2 73 05 f1-d9 01 c4 cb f2 73 05 f1 .....s.....s..
```

可以看出，参数是各个文件的名字，这里抓到的是 `desktop.ini`，结果可能会不同，但是不影响继续分析。

关于调用的函数，引用 [MSDN](#) 的定义：

```
SIZE_T RtlCompareMemory(  
    _In_ const VOID *Source1,  
    _In_ const VOID *Source2,  
    _In_      SIZE_T Length  
);
```

因此，这个函数的目的是，比较 `Mlwx` 和 `dir` 列出来的各个 `filename`。具体来说，首先将第一个串中的第一个字节与第二个串中的第一个字节进行比较，并在字节匹配时继续比较两个串中的连续字节。当串遇到不相等的第一对字节时，或者当匹配的字节数等于 `Length` 参数值(以先出现者为准)时，停止比较字节。

比较之后，会执行下面的代码：

```
cmp  eax, 8  
jne  Mlwx486+0x4f4
```

因此，这段代码比较每个文件的文件名，以查看它们开头的四个字节是否是 `Mlwx`。

如果返回值不等于 `8` 之后，程序就会跳到 `Mlwx486+0x4f4` 这个地方，这个地方的代码如下：

```
mov  eax, dword ptr [esi]  
test eax, eax  
je   Mlwx486+0x500 # if eax == 0 -> jump and return 2Ch  
test bl, bl      # bl always equal 0  
jne  Mlwx486+0x500 # if bl != 0 -> jump back to push '8'  
mov  edi, esi  
add  esi, eax  
jmp  Mlwx486+0x4ca # jump back to 'push 8'  
pop  ebx  
pop  esi  
pop  ebp  
ret  2Ch
```

上面的代码一般正常情况下，就会返回 `2Ch` 之后就退出函数了，逻辑上来说就是，如果每次传入的 `FileName` 前四个字节和 `Mlwx` 不相等，函数直接就退出。

如果 `RtlCompareMemory` 返回值是 `8` 的话，就会执行以下这些代码：

```
inc  bl          # bl now is equal 0 by [xor bl, bl]  
test edi, edi  
je   Mlwx486+0x4f4 # if edi == 0, jump here -----
```

```

mov  eax, dword ptr [esi] # esi -> FileInformation structure
test eax, eax
jne  Mlwx486+0x4f2      # if eax !=0, jump here -----
and  dword ptr [edi], eax #
jmp  Mlwx486+0x4f4      # jump here -----|->
add  dword ptr [edi], eax # <-----
mov  eax, dword ptr [esi] # <-----
test eax, eax
je   Mlwx486+0x504      # return 2Ch
test bl, bl
jne  Mlwx486+0x500      # if bl != 0, jump here -----
mov  edi, esi
add  esi, eax           # <-----
                           # esi now point to the next
                           # FILE_BOTH_DIR_INFORMATION structure
jmp  Mlwx486+0x4ca      # jump back to push'8'
pop  ebx
mov  eax, dword ptr [ebp+30h]
pop  edi
pop  esi
pop  ebp
ret  2Ch

```

现在分析它是如何修改 `NtQueryDirectoryFile` 的返回值然后隐藏 `Mlwx486.sys` 文件的，可以查一下 `NtQueryDirectoryFile` 的文档，可以知道，`NtQueryDirectoryFile` 的返回值 `FILE_BOTH_DIR_INFORMATION` 结构是由一系列 `FILE_BOTH_DIR_INFORMATION` 结构串联而成的：

```

-----
| FILE_BOTH_DIR_INFORMATION | ---
-----
--- | FILE_BOTH_DIR_INFORMATION | <---
|
--> | FILE_BOTH_DIR_INFORMATION |
-----

```

通常来说，第一个结构体是指向第二个结构体的，然后第二个结构体指向第三个结构体。

而这个函数的大致操作是如果当前项拥有一个以 `Mhwx` 开头的文件，则移动前面项的偏移量，使其指向当前项的下一个结构，因此隐藏了中间的结构体，相当于抹除了 `Mlwx486.sys` 文件的 `FILE_BOTH_DIR_INFORMATION` 结构，就达到了隐藏文件的目的。

## 恢复隐藏文件

使用DOS命令复制文件，复制后的文件没有被隐藏：

```
copy Mlwx486.sys mmmmmm.sys
```

```
C:\WINDOWS\system32>copy Mlwx486.sys mmmmmm.sys
已复制      1 个文件。
```

在IDA Pro中打开复制后的文件。

可以看到 `DriverEntry` 以参数 `KeServiceDescriptorTable` 和 `NtQueryDirectoryFile` 调用了 `RtlInitUnicodeString` 函数，然后调用 `MmGetSystemRoutineAddress` 函数来查找这两个地址的偏移量。

```
INIT:0001070F      mov     esi, ds:RtlInitUnicodeString
INIT:00010715      push    edi
INIT:00010716      push    offset SourceString ; "N"
INIT:0001071B      lea     eax, [ebp+DestinationString]
INIT:0001071E      push    eax                ; DestinationString
INIT:0001071F      call   esi ; RtlInitUnicodeString
INIT:00010721      push    offset aKeservicedescr ; "KeServiceDescriptorTable"
INIT:00010726      lea     eax, [ebp+SystemRoutineName]
INIT:00010729      push    eax                ; DestinationString
INIT:0001072A      call   esi ; RtlInitUnicodeString
INIT:0001072C      mov     esi, ds:MmGetSystemRoutineAddress
INIT:00010732      lea     eax, [ebp+DestinationString]
INIT:00010735      push    eax                ; SystemRoutineName
INIT:00010736      call   esi ; MmGetSystemRoutineAddress
INIT:00010738      mov     edi, eax
INIT:0001073A      lea     eax, [ebp+SystemRoutineName]
INIT:0001073D      push    eax                ; SystemRoutineName
INIT:0001073E      call   esi ; MmGetSystemRoutineAddress
INIT:00010740      mov     eax, [eax]
INIT:00010742      xor     ecx, ecx
TNTT:00010744
```

接下来，它在SSDT中查找 `NtQueryDirectoryFile` 函数项，并用挂钩函数的地址覆盖这个项。

```
INIT:00010744 loc_10744:      ; CODE XREF: DriverEntry(x,x)+4C↓j
INIT:00010744      add     eax, 4
INIT:00010747      cmp     [eax], edi
INIT:00010749      jz      short loc_10754
INIT:0001074B      inc     ecx
INIT:0001074C      cmp     ecx, 11Ch
INIT:00010752      jl      short loc_10744
INIT:00010754      ; CODE XREF: DriverEntry(x,x)+43↑j
INIT:00010754 loc_10754:      mov     dword_1068C, edi
INIT:00010754      mov     dword_10690, eax
INIT:0001075A      pop     edi
INIT:0001075F      mov     dword ptr [eax], offset sub_10486  挂钩函数
INIT:00010760      xor     eax, eax
INIT:00010766      pop     esi
INIT:00010768      leave
INIT:00010769      retn    8
INIT:0001076A
```

# Lab 10-03

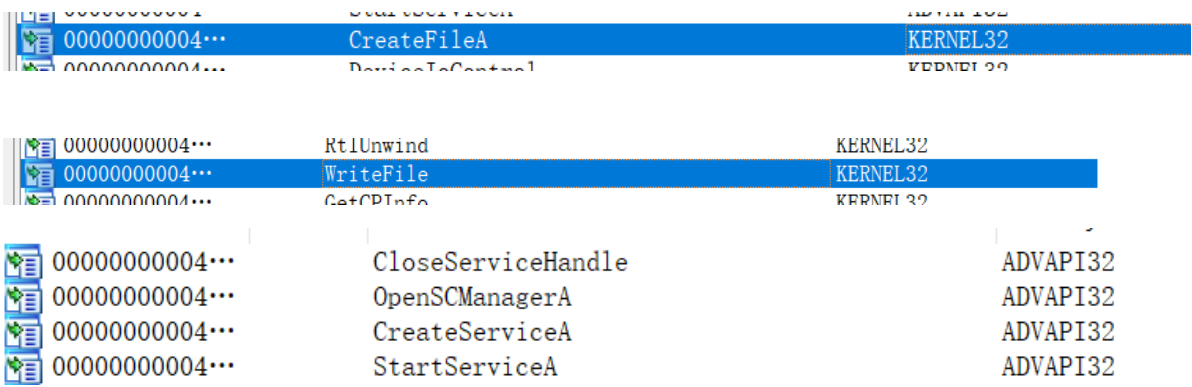
- 本实验包括一个驱动程序和一个可执行文件。驱动程序Lab10-03.sys需要放在C:\Windows\System32目录下。

## 基本静态分析

### 静态分析 Lab10-03.exe

先分析可执行文件Lab10-03.exe。查看导入表：

- 发现导入 `CreateFile` 和 `WriteFile` 函数，说明恶意代码会创建、写文件；
- 还导入了 `OpenSCManagerA`、`StartServiceA`、`CreateServiceA` 等函数，说明会在系统中创建一个服务。



000000000004...	CreateFileA	KERNEL32
000000000004...	DeviceIoControl	KERNEL32
000000000004...	RtlUnwind	KERNEL32
000000000004...	WriteFile	KERNEL32
000000000004...	CloseServiceHandle	ADVAPI32
000000000004...	OpenSCManagerA	ADVAPI32
000000000004...	CreateServiceA	ADVAPI32
000000000004...	StartServiceA	ADVAPI32

### 静态分析 Lab10-03.sys

再静态分析驱动文件。查看导入表，可以发现导入了以下函数：

Address	Ordinal	Name	Library
000000000000...		IoCompleteRequest	ntoskrnl
000000000000...		IoDeleteDevice	ntoskrnl
000000000000...		IoDeleteSymbolicLink	ntoskrnl
000000000000...		RtlInitUnicodeString	ntoskrnl
000000000000...		IoGetCurrentProcess	ntoskrnl
000000000000...		IoCreateSymbolicLink	ntoskrnl
000000000000...		IoCreateDevice	ntoskrnl
000000000000...		KeTickCount	ntoskrnl

其中：

- `IoCreateDevice`：创建供驱动程序使用的设备对象；
- `IoCreateSymbolicLink`：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`：返回一个指向当前进程的指针；

- `IofCompleteRequest`：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；长为49.7天；
- `RtlInitUnicodeString`：初始化一个统计的Unicode字符串。

从 `IoGetCurrentProcess` 这个例程可以看出，这个驱动或者在修改正在运行的进程，或者需要关于进程的信息。

## 基本动态分析

拍摄快照。将驱动文件复制到 `C:\Windows\System32`，双击可执行程序运行。可以看到弹出浏览器窗口，而且每隔一段时间(约30秒)程序还会弹出窗口。



当打开任务管理器试图终止进程时，程序并未列出，而且Process Explorer也没有列出。



Process	CPU	Private D...	Working Set	PID	Description	Company Name
System Idle Process	76.29		K	28 K	0	
System	1.03		K	296 K	4	
Interrupts	< 0.01		K		n/a Hardware Interrupts a...	
smss.exe		168 K		404 K	372 Windows NT Session Ma...	Microsoft Corporation
csrss.exe	1.03	1,840 K		5,708 K	604 Client Server Runtime...	Microsoft Corporation
winlogon.exe		6,728 K		4,416 K	628 Windows NT Logon Appl...	Microsoft Corporation
services.exe		1,780 K		3,628 K	672 Services and Controll...	Microsoft Corporation
vmacthlp.exe		688 K		2,620 K	852 VMware Activation Helper	VMware, Inc.
svchost.exe		3,124 K		5,004 K	880 Generic Host Process ...	Microsoft Corporation
vmtoolsd.exe		3,596 K		8,296 K	1040 WMI	Microsoft Corporation
vmtoolsd.exe		2,048 K		4,984 K	1392 WMI	Microsoft Corporation
IEEXPLORE.EXE		12,756 K		22,832 K	2428 Internet Explorer	Microsoft Corporation
svchost.exe		1,908 K		4,504 K	992 Generic Host Process ...	Microsoft Corporation
svchost.exe		14,504 K		24,100 K	1136 Generic Host Process ...	Microsoft Corporation
wsntctfy.exe		664 K		2,484 K	1308 Windows Security Cent...	Microsoft Corporation
vmtoolsd.exe		5,396 K		5,396 K	1648 Automatic Updates	Microsoft Corporation
svchost.exe		1,408 K		3,732 K	1180 Generic Host Process ...	Microsoft Corporation
svchost.exe		1,780 K		4,536 K	1228 Generic Host Process ...	Microsoft Corporation
spoolsv.exe		4,328 K		6,956 K	1548 Spooler SubSystem App	Microsoft Corporation
svchost.exe		2,244 K		3,380 K	1772 Generic Host Process ...	Microsoft Corporation
VGAuthService...		6,344 K		9,120 K	1980 VMware Guest Authent...	VMware, Inc.
vmtoolsd.exe		11,548 K		14,660 K	312 VMware Tools Core Ser...	VMware, Inc.
alg.exe		1,224 K		3,676 K	1824 Application Layer Gat...	Microsoft Corporation
lsass.exe		3,872 K		6,124 K	684 LSA Shell (Export Ver...	Microsoft Corporation
explorer.exe		19,044 K		24,944 K	500 Windows Explorer	Microsoft Corporation

为了停止弹出浏览器窗口的弹出，恢复快照至病毒运行前。

## IDA Pro 静态分析 Lab10-03.exe

首先找到 `WinMain` 函数，看到第一个调用的函数是 `OpenSCManagerA` 函数，这是一个用于打开服务控制的函数，说明程序打算在这里操作服务。

```

sub     esp, 28h
push    esi
push    0F003Fh          ; dwDesiredAccess
push    0                ; lpDatabaseName
push    0                ; lpMachineName
call    ds:OpenSCManagerA
test    eax, eax
jz      loc_401131

```

如果 `OpenSCManagerA` 调用成功，就会调用 `CreateServiceA` 函数创建一个服务。重点分析一下传入的参数。

```
push 0 ; lpPassword
push 0 ; lpServiceStartName
push 0 ; lpDependencies
push 0 ; lpdwTagId
push 0 ; lpLoadOrderGroup
push offset BinaryPathName ; "C:\\Windows\\System32\\Lab10-03.sys"
push 1 ; dwErrorControl
push 3 ; dwStartType
push 1 ; dwServiceType
push 0F01FFh ; dwDesiredAccess
push offset DisplayName ; "Process Helper"
push offset DisplayName ; "Process Helper"
push eax ; hSCManager
call ds:CreateServiceA
mov esi, eax
test esi, esi
jz short loc_401057
```

- `BinaryPathName` : 传入驱动文件的路径 `C:\\Windows\\System32\\Lab10-03.sys` , 这是用于指明服务的二进制文件的位置的参数;
- `dwStartType` : 值为 `3` , 说明此服务会自动启动;
- `dwServiceType` : 值为`1`, 指明这是一个驱动服务;
- `lpServiceName` 和 `lpDisplayName` 说明的这个服务的名字是 `Process Helper` 。

<b>SERVICE_DEMAND_START</b> 0x00000003	A service started by the service control manager when a process calls the <code>StartService</code> function. For more information, see <a href="http://blog.csdn.net/isinstance">Starting Services on Demand</a> .
---	---

如果调用 `CreateServiceA` 成功, 会执行 `StartService` , 启动创建的 `Process Helper` 服务, 加载恶意驱动 `Lab10-03.sys` 至内核。

```
push 0 ; lpServiceArgVectors
push 0 ; dwNumServiceArgs
push esi ; hService
call ds:StartServiceA
```

接下来, 创建了一个文件在 `\\.\ProcHelper` 这个地方并作为一个句柄打开。

```

push    0                ; hTemplateFile
push    80h ; '€'        ; dwFlagsAndAttributes
push    2                ; dwCreationDisposition
push    0                ; lpSecurityAttributes
push    0                ; dwShareMode
push    0C0000000h       ; dwDesiredAccess
push    offset FileName ; "\\.\ProcHelper"
call    ds:CreateFileA
cmp     eax, 0FFFFFFFFh
jnz     short loc_40108C

```

成功后，执行下面的代码，调用 `DeviceIoControl` 函数。

```

loc_40108C:
lea     ecx, [esp+2Ch+BytesReturned]
push    0                ; lpOverlapped
push    ecx              ; lpBytesReturned
push    0                ; nOutBufferSize
push    0                ; lpOutBuffer
push    0                ; nInBufferSize
push    0                ; lpInBuffer
push    0ABCDEF01h       ; dwIoControlCode
push    eax              ; hDevice
call    ds:DeviceIoControl
push    0                ; pvReserved
call    ds:OleInitialize
test    eax, eax
jl      short loc_401131

```

`DeviceIoControl` 函数的用途为：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。因此，这里调用的目的是发送一个请求到Lab10-03.sys驱动。

根据 MSDN 中的定义，可以得出传入的参数列表：

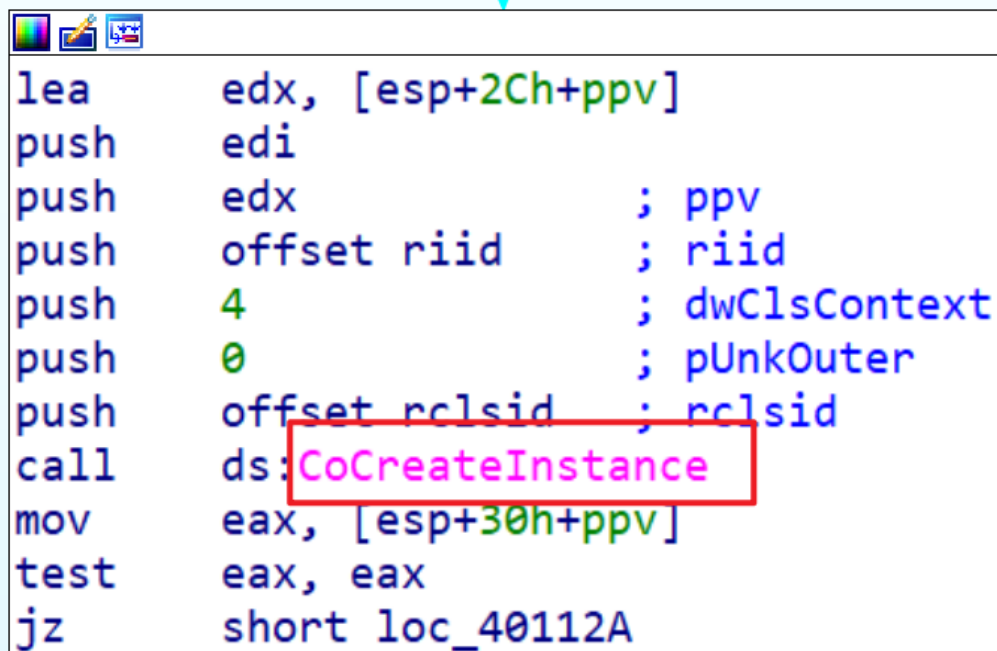
```

BOOL WINAPI DeviceIoControl(
    _In_      HANDLE      hDevice = eax,
    _In_      DWORD       dwIoControlCode = 0abcdef01h,
    _In_opt_  LPVOID      lpInBuffer = 0,
    _In_      DWORD       nInBufferSize = 0,
    _Out_opt_ LPVOID      lpOutBuffer = 0,
    _In_      DWORD       nOutBufferSize = 0,
    _Out_opt_ LPDWORD      lpBytesReturned = eax,
    _Inout_opt_ LPOVERLAPPED lpOverlapped = 0
);

```

其中，`lpInBuffer` 和 `lpOutBuffer` 被设置为了 `Null`，这意味着这个请求没有发送任何的信息到内核驱动中( `lpInBuffer = 0` )，并且内核驱动也没有反馈( `lpOutBuffer = 0` )；此外，`dwIoControlCode` 的值是 `abcdedf01`，这个值很可疑。

接下来，调用 `CoCreateInstance`，用于一个用于创建 COM 对象。



```

lea     edx, [esp+2Ch+ppv]
push    edi
push    edx          ; ppv
push    offset riid  ; riid
push    4            ; dwClsContext
push    0            ; pUnkOuter
push    offset rclsid ; rclsid
call    ds:CoCreateInstance
mov     eax, [esp+30h+ppv]
test    eax, eax
jz      short loc_40112A

```

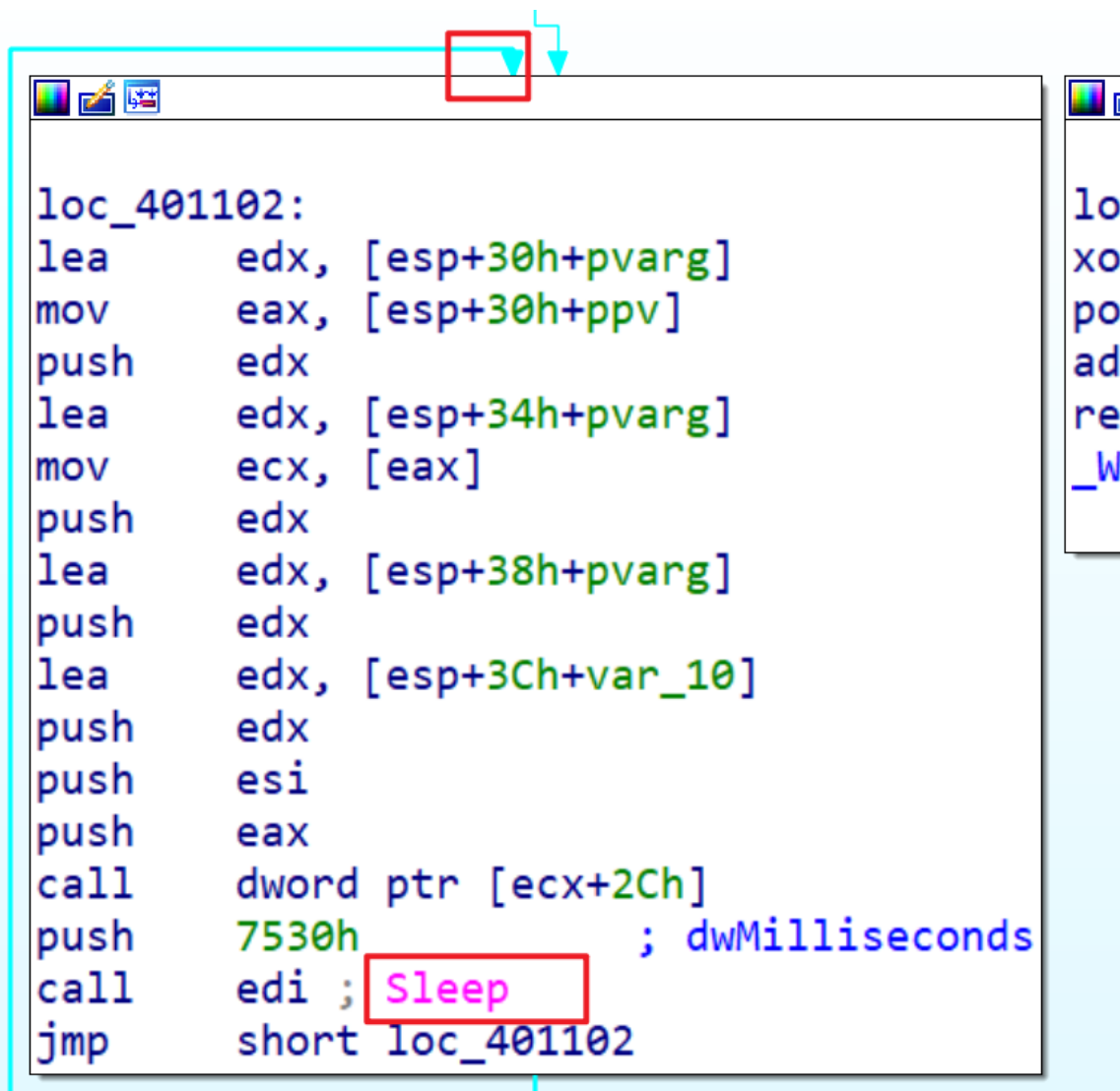
接下来调用 `SysAllocString` 函数，调用之前压入了字符串 `http://www.malwareanalysisbook.com/ad.html`。这就是会打开的那个广告页面的 URL。

```

lea     eax, [esp+30h+pvarg]
push    eax                ; pvarg
call    ds:VariantInit
push    offset psz         ; "http://www.malwareanalysisbook.com/ad.h"...
mov     [esp+34h+var_10], 3
mov     [esp+34h+var_8], 1
call    ds:SysAllocString
mov     edi, ds:Sleep
mov     esi, eax

```

最后调用了 `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。



```

loc_401102:
lea     edx, [esp+30h+pvarg]
mov     eax, [esp+30h+ppv]
push    edx
lea     edx, [esp+34h+pvarg]
mov     ecx, [eax]
push    edx
lea     edx, [esp+38h+pvarg]
push    edx
lea     edx, [esp+3Ch+var_10]
push    edx
push    esi
push    eax
call    dword ptr [ecx+2Ch]
push    7530h              ; dwMilliseconds
call    edi ; Sleep
jmp     short loc_401102

```

## IDA Pro 静态分析 Lab10-03.sys

在IDA Pro中加载Lab10-03.sys。首先调用 `RtlInitUnicodeString` 函数，初始化一个统计的Unicode字符串。而传入的参数是 `\\Device\\ProcHelper`。

```

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

```

接下来调用 `IoCreateDevice` 函数，这个函数的定义如下：

```

NTSTATUS IoCreateDevice(
    _In_      PDRIVER_OBJECT DriverObject = esi,
    _In_      ULONG          DeviceExtensionSize = 0,
    _In_opt_  PUNICODE_STRING DeviceName = eax,
    _In_      DEVICE_TYPE    DeviceType = 22h,
    _In_      ULONG          DeviceCharacteristics = 100h,
    _In_      BOOLEAN        Exclusive = 0,
    _Out_     PDEVICE_OBJECT *DeviceObject = eax
);

```

因此，创建了一个名为 `\Device\ProcHelper` 的设备。

随后，调用 `RtlInitUnicodeString` 函数，传入的参数一个是 `eax`，一个是 `word_1070E`。进一步查看可知第二个参数为 `DosDevices\ProcHelper`。

```

INIT:000107DE word_107DE      dw 5Ch ; DATA XREF: DriverEntry(x,x)+4Bto
INIT:000107E0 aDosdevicesProc_0:
INIT:000107E0      text "UTF-16LE", 'DosDevices\ProcHelper',0

```

下面调用 `IoCreateSymbolicLink` 函数，该函数用于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。

```

INIT:00010742      test     eax, eax
INIT:00010744      jnl     short loc_10789
INIT:00010746      mov     eax, offset sub_10606
INIT:00010748      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      ....

```

因此，这个 `IoCreateSymbolicLink` 创建了一个符号链接供用户态的应用程序访问这个设备。

最后一个调用是 `IoDeleteDevice` 这个函数，这个函数删除驱动之后就退出了。

## 使用WinDbg 查找内存中的驱动

宿主机打开WinDbg调试虚拟机。使用命令 `!devobj ProcHelper` 查找设备对象。



```
kd> !devobj ProcHelper
Device object (81db22b0) is for:
  ProcHelper \Driver\Process Helper DriverObject 81f9a9f8
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
SecurityDescriptor e1326f70 DevExt 00000000 DevObjExt 81db2368
ExtensionFlags (0000000000)
Characteristics (0x00000100) FILE_DEVICE_SECURE_OPEN
Device queue is not busy.
```

这里可以看到 `ProcHelper` 这个 `DriverObject` 存储的位置。`DriverObject` 包含了所有函数的指针，当用户空间的程序访问设备对象调用这些函数时，`DriverObject` 存储在一个叫做 `DRIVER_OBJECT` 的结构里面。

使用 `dt` 命令具体查看这个驱动对象。

```
<d> dt nt!_DRIVER_OBJECT 81f9a9f8
+0x000 Type           : 0n4
+0x002 Size           : 0n168
+0x004 DeviceObject    : 0x81db22b0 _DEVICE_OBJECT
+0x008 Flags           : 0x12
+0x00c DriverStart     : 0xf8cfb000 Void
+0x010 DriverSize      : 0xe00
+0x014 DriverSection   : 0x820fc370 Void
+0x018 DriverExtension : 0x81f9aaa0 _DRIVER_EXTENSION
+0x01c DriverName      : _UNICODE_STRING "\Driver\Process Helper"
+0x024 HardwareDatabase : 0x80671ae0 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
+0x028 FastIoDispatch  : (null)
+0x02c DriverInit      : 0xf8cfb7cd long +0
+0x030 DriverStartIo   : (null)
+0x034 DriverUnload    : 0xf8cfb62a void +0
+0x038 MajorFunction   : [28] 0xf8cfb606 long +0
```

这里包含了几个应该注意的函数指针，包括前面在IDA Pro中分析的 `DriverInit` 和 `DriverEntry`。此外，还有 `DriverUnload`，是驱动卸载时候的操作地址。

## 分析主函数表中的函数

接下来查看主函数表：

```
kd> dd 81f9a9f8+0x38 L1c
81f9aa30 f8cfb606 804f454a f8cfb606 804f454a
81f9aa40 804f454a 804f454a 804f454a 804f454a
81f9aa50 804f454a 804f454a 804f454a 804f454a
81f9aa60 804f454a 804f454a f8cfb666 804f454a
81f9aa70 804f454a 804f454a 804f454a 804f454a
81f9aa80 804f454a 804f454a 804f454a 804f454a
81f9aa90 804f454a 804f454a 804f454a 804f454a
```

表中大多数函数都是 `804f454a` 的项，查看这个函数做了什么。

```
kd> bp 804f454a
breakpoint 0 redefined
kd> g
Break instruction exception - code 80000003 (first chance)
nt!IopInvalidDeviceRequest:
804f454a 8bff          mov     edi,edi
```

该处的函数被命名为了 `IopInvalidDeviceRequest`，从字面意思理解就是这个函数是处理驱动无法处理的非法请求的。

此外，在主函数表中，可以发现三个不同于 `804f454a` 的地址。

```
kd> dd 81f9a9f8+0x38 L1c
81f9aa30 f8cfb606 804f454a f8cfb606 804f454a
81f9aa40 804f454a 804f454a 804f454a 804f454a
81f9aa50 804f454a 804f454a 804f454a 804f454a
81f9aa60 804f454a 804f454a f8cfb666 804f454a
81f9aa70 804f454a 804f454a 804f454a 804f454a
81f9aa80 804f454a 804f454a 804f454a 804f454a
81f9aa90 804f454a 804f454a 804f454a 804f454a
```

首先查看 `f8cfb606` 处的函数。

```
kd> u f8cfb606
ReadVirtual: f8cfb606 not properly sign extended
f8cfb606 8bff          mov     edi,edi
f8cfb608 55             push    ebp
f8cfb609 8bec          mov     ebp,esp
f8cfb60b 8b4d0c        mov     ecx,dword ptr [ebp+0Ch]
f8cfb60e 83611800      and     dword ptr [ecx+18h],0
f8cfb612 83611c00      and     dword ptr [ecx+1Ch],0
f8cfb616 32d2          xor     dl,dl
f8cfb618 ff1580b4cff8  call   dword ptr [Lab10_03+0x480 (f8cfb480)]
```

它只调用了 `f8cfb480` 处的函数，然后返回。进一步查看，这个地址被标记为 `IofCompleteRequest` 函数。这个函数的主要功能是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。这告诉操作系统请求成功，但是什么也没有做。

## ▲ `f8cfb666`处的函数

重点关注 `f8cfb666` 处的函数。

```

kd> u f8cfb666 L13
f8cfb666 8bff          mov     edi,edi
f8cfb668 55             push   ebp
f8cfb669 8bec          mov     ebp,esp
f8cfb66b ff1590b4cff8      call   dword ptr [Lab10_03+0x490 (f8cfb490)]
f8cfb671 8b888c000000     mov     ecx,dword ptr [eax+8Ch]
f8cfb677 0588000000       add     eax,88h
f8cfb67c 8b10          mov     edx,dword ptr [eax]
f8cfb67e 8911          mov     dword ptr [ecx],edx
f8cfb680 8b08          mov     ecx,dword ptr [eax]
f8cfb682 8b4004          mov     eax,dword ptr [eax+4]
f8cfb685 894104          mov     dword ptr [ecx+4],eax
f8cfb688 8b4d0c          mov     ecx,dword ptr [ebp+0Ch]
f8cfb68b 83611800       and     dword ptr [ecx+18h],0
f8cfb68f 83611c00       and     dword ptr [ecx+1Ch],0
f8cfb693 32d2          xor     dl,dl
f8cfb695 ff1580b4cff8      call   dword ptr [Lab10_03+0x480 (f8cfb480)]
f8cfb69b 33c0          xor     eax,eax
f8cfb69d 5d            pop     ebp
f8cfb69e c20800       ret     8

```

这个函数做的首先调用 `IoGetCurrentProcess`，它返回一个 `EPROCESS` 结构。然后，这个函数访问偏移量 `0x88` 处的数据，再然后，访问偏移量 `0x8C` 处的下一个 `DWORD`。

使用WinDbg检查这个 `EPROCESS` 结构：

```
dt nt!_EPROCESS
```

```

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
+0x0c4 ObjectTable : Ptr32 _HANDLE_TABLE
+0x0c8 Token : _EX_FAST_REF

```

找到偏移 `0x88` 的地方，我们可以看到一个叫 `ActiveProcessLinks` 的 `_LIST_ENTRY`。这个 `LIST_ENTRY` 是个双向链表。成员 `Flink` 负责指向后一个结构体，而成员 `Blink` 负责指向前一个结构体，最后这个链表会形成一个闭环。

详细分析对 `_LIST_ENTRY` 的操作代码：

```
f8cfb671 8b888c000000    mov     ecx,dword ptr [eax+8Ch]
f8cfb677 0588000000    add     eax,88h
f8cfb67c 8b10          mov     edx,dword ptr [eax]
f8cfb67e 8911          mov     dword ptr [ecx],edx
f8cfb680 8b08          mov     ecx,dword ptr [eax]
f8cfb682 8b4004        mov     eax,dword ptr [eax+4]
f8cfb685 894104        mov     dword ptr [ecx+4],eax
```

```
mov     ecx, dword ptr [eax+8Ch]
```

该步操作将 `LIST_ENTRY` 的前一个结构体的指针 `Blink` 指针赋值给了 `ecx`。

```
add     eax, 88h
mov     edx, dword ptr [eax]
```

这里将 `eax` 偏移了 `0x88h`，所以现在 `eax` 不再是指向 `_EPROCESS` 这个结构体的开头，而是指向 `_LIST_ENTRY` 的开始地址，而之后的 `mov` 操作将 `Flink` 指针的值赋值给了 `edx`。

```
mov     dword ptr [ecx], edx
```

`ecx` 指向前一个结构体，`[ecx]` 的值代表了前一个结构体的 `Flink` 的值。`edx` 的值是当前结构体的 `Flink`，也就是代表了当前结构体下一个结构体的地址。这里将 `edx` 的值赋值给了 `[ecx]`。因此，这步操作，将当前结构体的下一个结构体赋值为当前结构体的前一个结构体的下一个结构体 `Flink`，达到了跳过了当前进程的结构体的目的。

```
mov     ecx, dword ptr [eax]
```

`eax` 是指向当前进程的 `LIST_ENTRY`，而 `ecx` 指向的是前一个结构体，这个操作将 `ecx` 的值改变为了当前结构体的 `Flink` 的值，即后一个结构体的起始地址。

```
mov     eax, dword ptr [eax+4]
```

`eax+4` 将指向了这个结构体的 `Blink` 的起始地址，赋值后，`eax` 指向当前结构体的前一个结构体。

```
mov     dword ptr [ecx+4], eax
```

这步操作，将当前结构体的前一个结构体赋值给后一个结构体的 `Blink` 指针。

综上所述，这个函数就是在双向链表中，通过修改前后节点的指针，将自己的结构体从双向链表中卸下。

因此，可以回答问题一了。

## 问题一：程序的行为

- 这个程序做了些什么？

程序加载驱动，运行时每隔30秒就弹出一个浏览器窗口。但是无法在进程管理器中发现运行的程序，是因为驱动通过从系统链表中修改前后节点的指针摘除自身进程环境块(PEB)，实现隐藏进程。

## 问题二：如何停止程序

- 一旦程序运行，你怎样停止它？

只能重启系统，或者在程序运行前拍摄快照，运行后恢复快照至程序运行前。

## 问题三：内核组件的操作

- 它的内核组件做了什么操作？

在前面对主函数表中偏移 `0xeh` 的函数的分析中可以知道，内核组件修改了进程链接表的结构，通过修改前后节点的指针，将自己的 `LIST_ENTRY` 从双向链表PEB中摘下，隐藏了自己的 `LIST_ENTRY`，从而实现对用户隐藏进程。

# Yara 规则编写

根据前面的分析，编写Yara规则如下：

```
rule Lab10_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 < 100KB and
        all of them
    }
    rule Lab10_01sys {
        meta:
            description = "Lab10-01.sys"
        strings:
            $s1 =
"c:\winddk\7600.16385.1\src\general\regwriter\wdm\sys\objfre_wxp_x8
6\i386\siocctl.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
        condition:
            uint16(0) == 0x5a4d and
            uint32(uint32(0x3c))==0x00004550and filesize < 100KB and
            all of them
    }
    rule Lab10_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\siocctl.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 = "486 WS Driver" fullword ascii
            $s8 = "6.1.7600.16385 built by: WinDDK" fullword wide
            $s9 = "KeServiceDescriptorTable" fullword wide
        condition:
            uint16(0) == 0x5a4d and
            uint32(uint32(0x3c))==0x00004550and filesize < 100KB and
            6 of them
    }
    rule Lab10_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 < 100KB and
    all of them
}
rule Lab10_03sys {
    meta:
        description = "Lab10-03.sys"
    strings:
        $s1 =
"c:\\winddk\\7600.16385.1\\src\\general\\rootkitprochide\\wdm\\sys\\objfre_
wpx_x86\\i386\\siocntl.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
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550and filesize < 100KB and
        all of them
}

```

能够扫描到相应的病毒样本，验证了Yara规则的正确性：

```

PS D:\NKU\23Fall\恶意代码分析与防治技术\yara-4.3.2-2150-win64> ./yara64 Lab10.yar Chapter_10L
Lab10_01sys Chapter_10L\Lab10-01.sys
Lab10_01exe Chapter_10L\Lab10-01.exe
Lab10_03exe Chapter_10L\Lab10-03.exe
Lab10_03sys Chapter_10L\Lab10-03.sys
Lab10_02 Chapter_10L\Lab10-02.exe

```

收集电脑PE文件并进行扫描，结果如下：

	sample
类型:	文件夹
位置:	D:\NKU\code\Python
大小:	17.1 GB (18,457,236,441 字节)
占用空间:	17.2 GB (18,484,576,256 字节)
包含:	14,589 个文件, 0 个文件夹
创建时间:	2023年10月4日, 18:39:02

```
Microsoft Visual Studio 调试 × + v
扫描到的文件:
Lab10_01exe D:\NKU\code\Python\sample\Lab10-01.exe
Lab10_01sys D:\NKU\code\Python\sample\Lab10-01.sys
Lab10_02 D:\NKU\code\Python\sample\Lab10-02.exe
Lab10_03exe D:\NKU\code\Python\sample\Lab10-03.exe
Lab10_03sys D:\NKU\code\Python\sample\Lab10-03.sys
运行时间为 10 s
```

## IDA Python 自动化分析

编写脚本查找特定函数，分析其控制流，并识别其中的基本块和交叉引用

```
import idaapi
import idautils
import idc

def analyze_function(function_name):
    # 查找二进制文件中的函数地址
    func_ea = idc.get_name_ea(idc.BADADDR, function_name)

    if func_ea == idc.BADADDR:
        print(f"函数 '{function_name}' 未找到")
        return

    # 分析函数的控制流
```

```

f = idaapi.get_func(func_ea)
if not f:
    print(f"无法获取函数 '{function_name}'")
    return

print(f"分析函数 '{function_name}', 入口地址: 0x{func_ea:08X}")

for block in idaapi.FlowChart(f):
    print(f"基本块: 0x{block.start_ea:08X}")

    # 使用idautils.XrefsTo来获取交叉引用
    for head in idautils.Heads(block.start_ea, block.end_ea):
        disasm = idc.GetDisasm(head)
        print(f"    0x{head:08X}: {disasm}")

        # 识别交叉引用
        for ref in idautils.XrefsTo(head):
            print(f"        引用自: 0x{ref.frm}")

if __name__ == '__main__':
    target_function = "sub_10606"

    analyze_function(target_function)

```

分析 `Lab10-01.sys` 的函数 `sub_10606`，结果如下：

分析函数 'sub\_10606', 入口地址: 0x00010606  
基本块: 0x00010606

```
0x00010606: mov     edi, edi
           引用自: 0x67398
0x00010608: push   ebp
           引用自: 0x67078
0x00010609: mov     ebp, esp
           引用自: 0x67080
0x0001060B: mov     ecx, [ebp+Irp]; Irp
           引用自: 0x67081
0x0001060E: and     dword ptr [ecx+18h], 0
           引用自: 0x67083
0x00010612: and     dword ptr [ecx+1Ch], 0
           引用自: 0x67086
0x00010616: xor     dl, dl; PriorityBoost
           引用自: 0x67090
0x00010618: call    ds:IofCompleteRequest
           引用自: 0x67094
0x0001061E: xor     eax, eax
           引用自: 0x67096
0x00010620: pop     ebp
           引用自: 0x67102
0x00010621: retn    8
           引用自: 0x67104
```

Python

## 编写脚本查看导入的动态链接库

```
import idaapi

def get_imported_libraries():
    # 创建一个用于存储导入库的集合
    imported_libraries = set()

    # 遍历导入表中的所有模块
    for i in range(idaapi.get_import_module_qty()):
        modname = idaapi.get_import_module_name(i)
        if modname:
            imported_libraries.add(modname)

    return sorted(imported_libraries)

def main():
    # 获取导入库列表
```

```
imported_libraries = get_imported_libraries()

if imported_libraries:
    print("导入的动态链接库：")
    for lib in imported_libraries:
        print(lib)
else:
    print("没有找到导入的动态链接库。")

if __name__ == "__main__":
    main()
```

分析 Lab10-01.exe 导入的动态链接库：

```
导入的动态链接库：
ADVAPI32
KERNEL32
OLEAUT32
ole32
Python
```

## 实验结论及心得体会

通过本次实验，从静态、动态多方面、多工具综合分析恶意代码，提高了分析恶意代码的能力。通过分析这几个实验样本的行为，对于内核组件和内核恶意代码的认识更为深刻，也初步掌握了一些分析它们的方法，熟练了WinDbg工具的使用。