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：2113997
姓名：齐明杰
班级：信安2班

1 实验目的

- **完成课本Lab10的实验内容，编写Yara规则，并尝试IDA Python的自动化分析，在此提交实验报告。**
- **补充R77的验证实验**

2 实验原理

本次实验旨在提供对Windows操作系统内核调试的深入理解，并增强识别和分析恶意软件（特别是Rootkit）的能力。

2.1 WinDbg概述

WinDbg是一个由微软开发的动态调试工具，它支持对用户模式应用程序、设备驱动程序和Windows操作系统内核进行调试。作为Windows调试工具箱的一部分，WinDbg使用图形用户界面和命令行界面，为开发者和系统管理员提供了一种有效方式来分析复杂的软件和硬件问题。

2.2 微软符号表

在进行调试时，符号文件是连接编译后的代码和源代码的重要桥梁。微软的符号服务器为公共Windows组件提供了广泛的符号文件，允许调试器解析函数调用、变量名和其他关键信息。正确配置符号路径是调试过程中的一个重要步骤，它确保了WinDbg能够检索到正确的符号信息，从而使调试更加准确和高效。

2.3 用户模式与内核模式调试

用户模式调试通常关注应用程序的行为，而内核模式调试则关注操作系统核心、硬件抽象层（HAL）和设备驱动程序。用户模式调试通常更直接、简单，因为它不涉及到整个系统的状态。内核模式调试则复杂得多，通常需要两台计算机（一台主机和一台目标机）和一个调试连接（如串行、USB或网络）。

2.4 Rootkit分析

Rootkit是一种特别隐蔽的恶意软件，它设计用来隐藏自己的存在以及它控制的其他软件或进程。由于Rootkit常常在内核模式下运行，它们能够躲避常规的安全检测。通过使用WinDbg这样的内核级调试工具，安全研究人员可以深入系统的底层，发现并分析Rootkit的行为。这包括检查隐藏的进程、分析不正常的系统调用、以及检测钩子等技术。

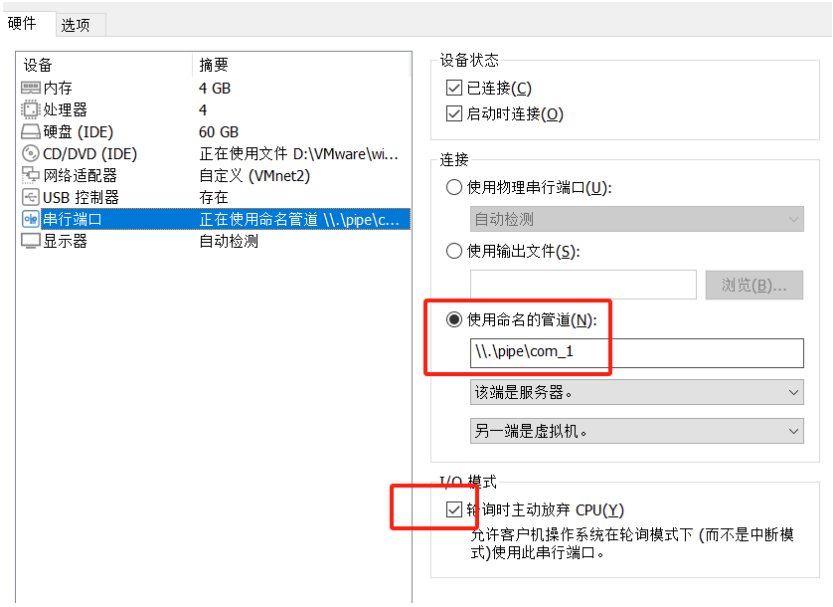
3 实验过程

对于每个实验样本，将采用**先分析，后答题**的流程。

3.1 配置内核调试 Win10 + WinXP + Windbg

我们先配置XP虚拟机，使其可以进行与主机共同的“**双机调试**”。

在虚拟机配置**串行端口**：



并且修改`C:\boot.ini`的内容为：

```
[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=com_1 /baudrate=115200
```

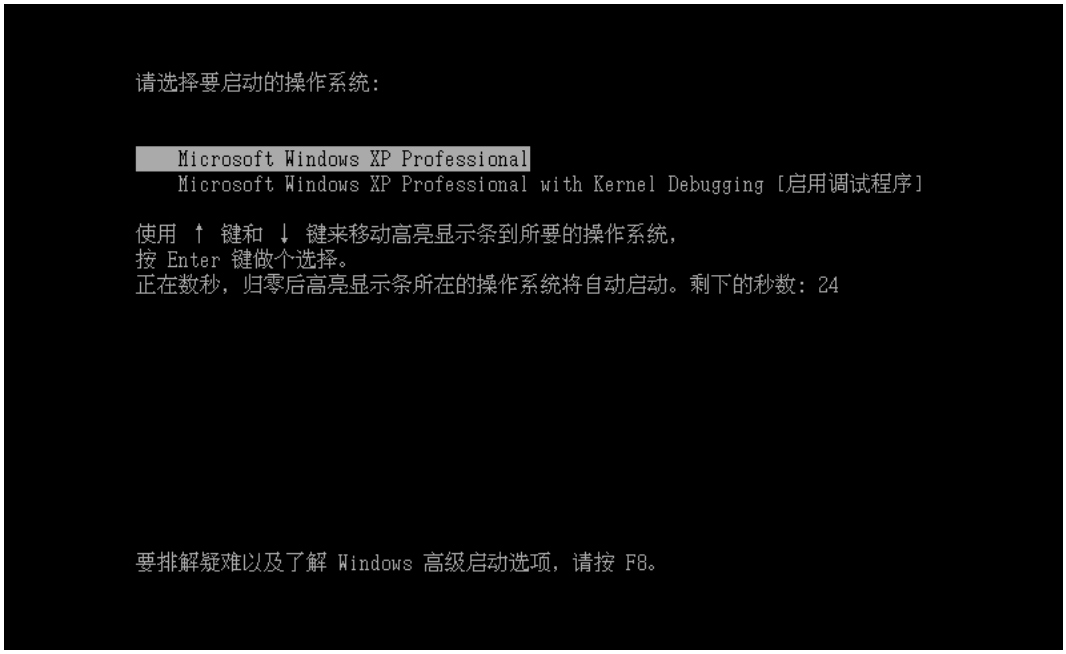
其中，最后一项为**内核调试启动模式**，设置串行端口为`com_1`。

然后，为了方便**物理机对虚拟机进行调试**，我们创建一个**快捷方式**，并加上如下命令：

```
-b -k com:port=\\.\pipe\com_1,baud=115200,pipe
```

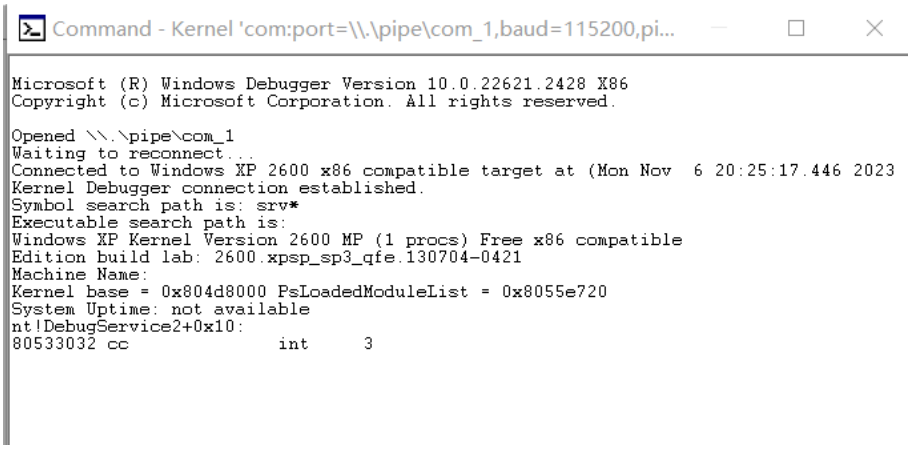
这样，我们使用这个快捷方式启动Windbg即按上述配置自动打开内核调试，十分方便。

然后启动XP虚拟机：

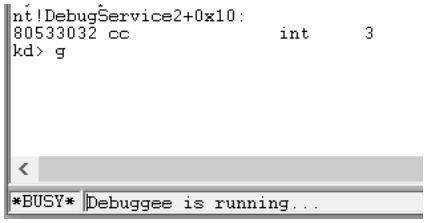


同时启动刚刚物理机Windbg配置好的快捷方式：

可以看到虚拟机出现了内核调试的启动选项，并且物理机的Windbg已经做好调试准备，正在尝试连接。接下来**虚拟机选择Debug模式启动**，随即Windbg可以显示内容：



但是此时**虚拟机仍为黑屏**，因为虚拟机被断下，我们需要输入`g`让虚拟机启动：

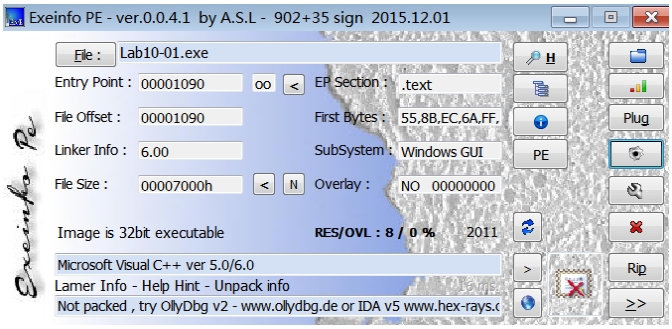


这样内核调试就*配置完成*了。

3.2 Lab10-01.exe & Lab10-01.sys

- **静态分析**

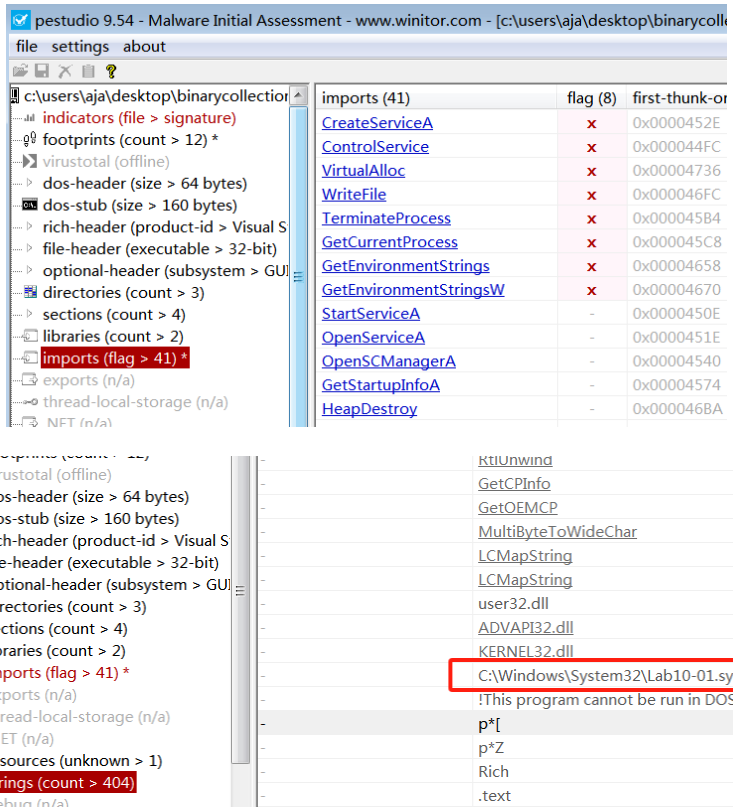
使用exeinfoPE查看加壳：



该恶意代码**无加壳**。

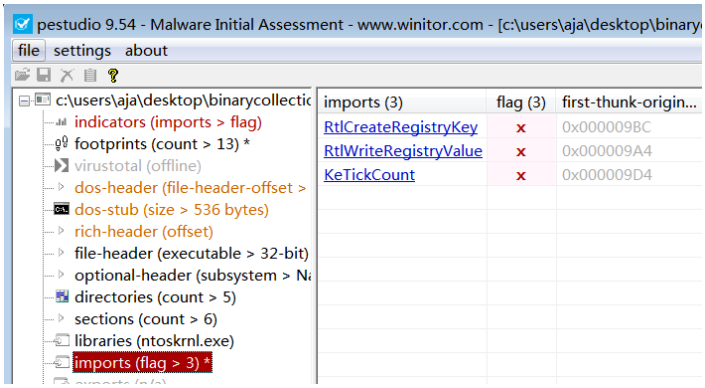
我们打开`Pestudio`进行**基本静态分析**，查看其**导入表**和**字符串**：

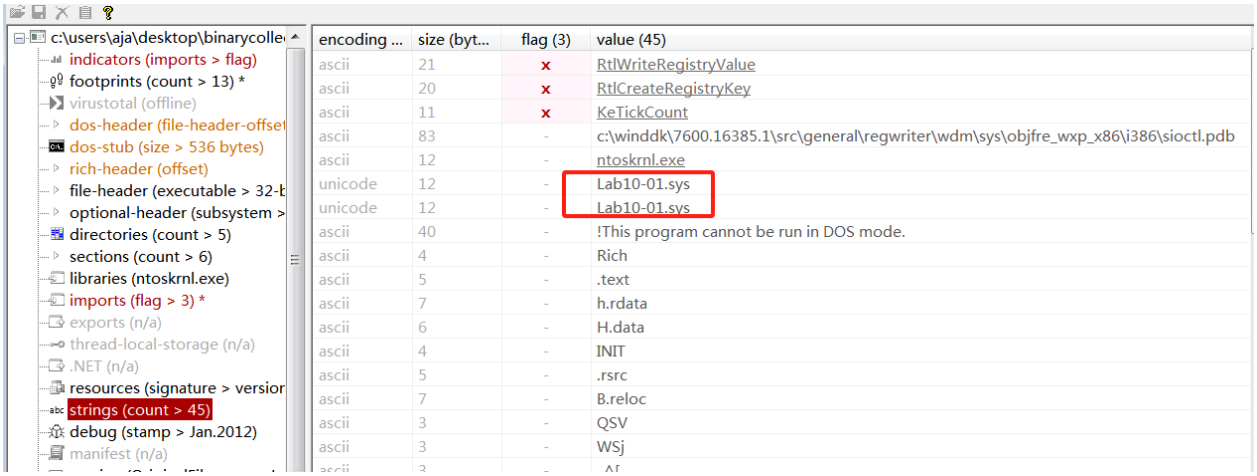
- **exe:**

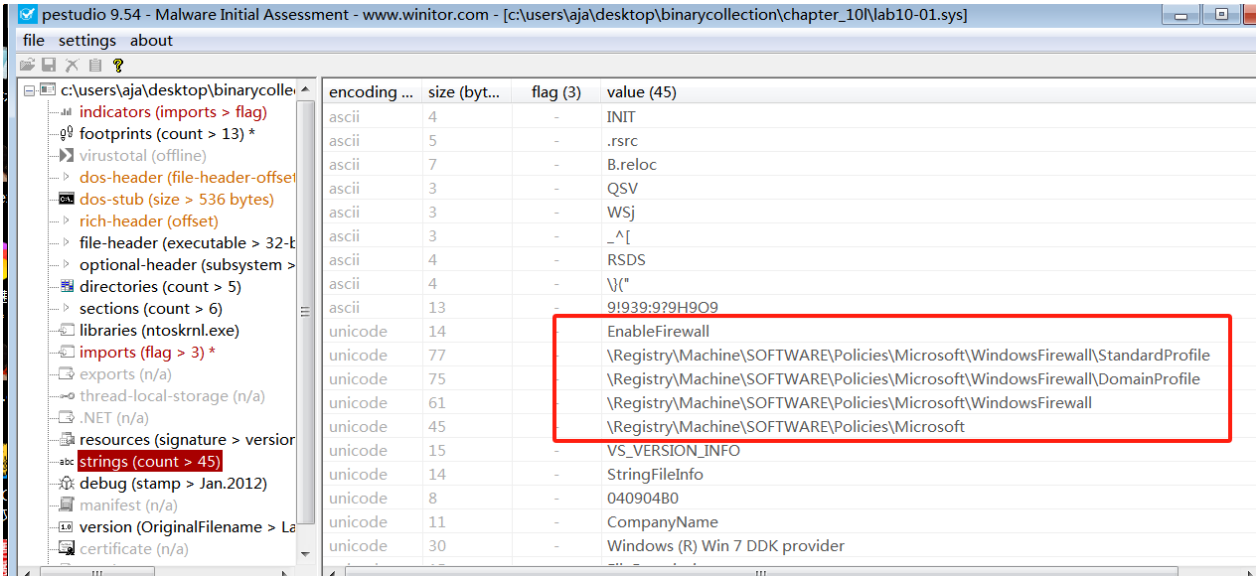


这个字符串似乎说明该exe使用了Lab10-01.sys。

- sys:





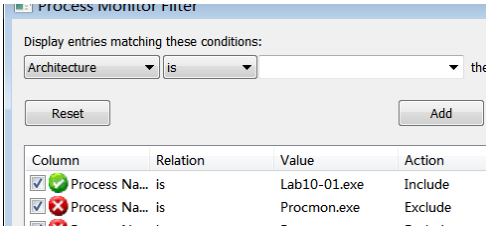


这是一个**驱动文件**，仅仅有三**个导入函数**，`RtlCreateRegistryKey`，`RtlWriteRegistryValue`和`KeTickCount`，前两个函数告诉我们它可能**访问和修改了注册表**。

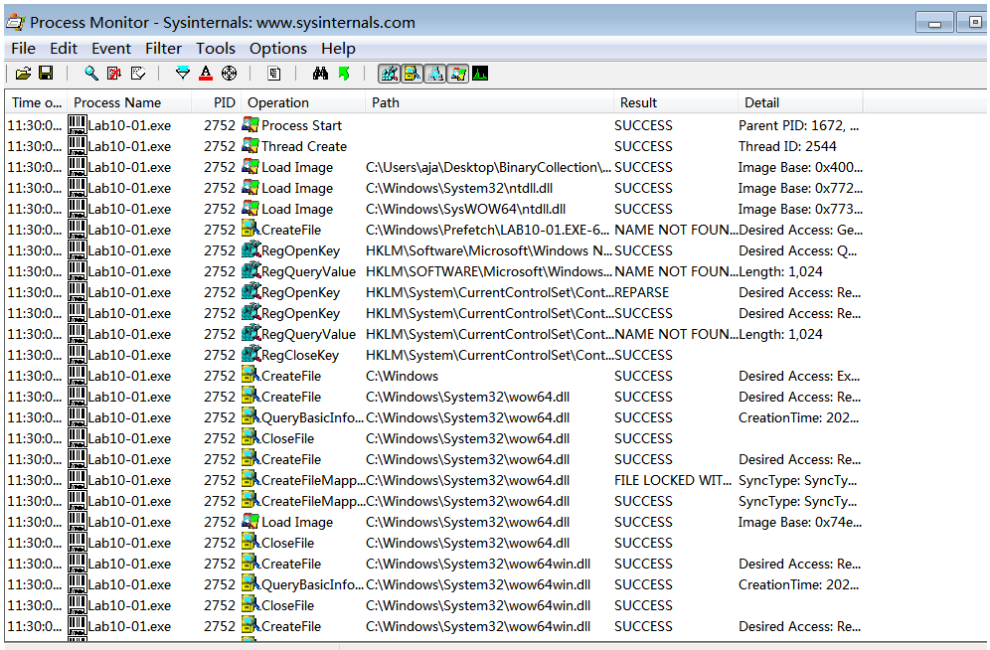
另外，字符串中似乎发现了若干注册表的键值。但这些键值**不同寻常**的`"HKLM"`等**根键**，其开头是`\Registry\Machine`，查阅资料发现，这个开头等价于**用户态**程序访问的`HKEY_LOCAL_MACHINE`，

它使用了`EnableFirewall`这个键值，这个值设置为0的含义是**禁用Windows XP自带的防火墙**。

为了查看它对注册表的操作，我们打开**Procmon**，设置**过滤器**：

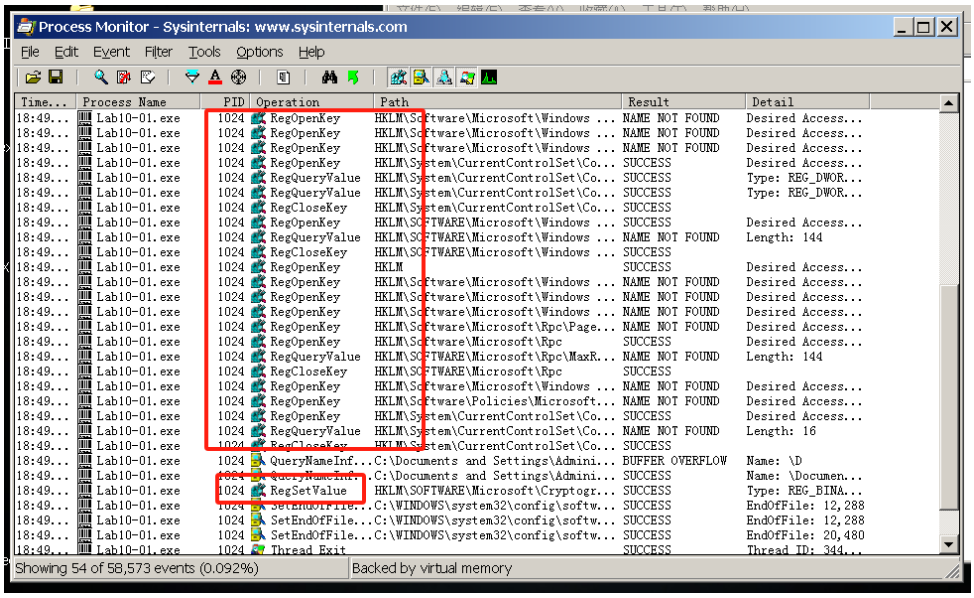


然后运行Lab10-01.exe，抓取到了它许多**对注册表的操作**：



参考答案发现会有一个RegSetValue写操作，但我正在使用虚拟机Win7 x64分析，并没有捕获到这个操作。因此我将更换为Win XP虚拟机来继续查看注册表操作，但静态分析(如使用IDA)将继续在Win7 虚拟机进行。

打开XP虚拟机的Procmon，同样进行过滤：



可以看到有许多对注册表的读操作，**只有一个写操作**，并且它写入的注册表键为`HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed`

接下来打开**IDA**对其进行分析：

查看**Main函数**的代码：

```
1  int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
   lpCmdLine, int nShowCmd)
2  {
3    int result; // eax
```

```
  SC_HANDLE v5; // edi
  SC_HANDLE ServiceA; // esi
  struct _SERVICE_STATUS ServiceStatus; // [esp+4h] [ebp-1Ch] BYREF

  result = (int)OpenSCManagerA(0, 0, 0xF003Fu);
  v5 = (SC_HANDLE)result;
  if ( result )
  {
    ServiceA = CreateServiceA(
                 (SC_HANDLE)result,
                 "Lab10-01",
                 "Lab10-01",
                 0xF01FFu,
                 1u,
                 3u,
                 1u,
                 "C:\\Windows\\System32\\Lab10-01.sys",
                 0,
                 0,
                 0,
                 0,
                 0);
    if ( ServiceA || (ServiceA = OpenServiceA(v5, "Lab10-01", 0xF01FFu))
!= 0 )
    {
      StartServiceA(ServiceA, 0, 0);
      if ( ServiceA )
        ControlService(ServiceA, 1u, &ServiceStatus);
    }
    return 0;
  }
  return result;
}
```

这段代码是一个 **Windows 应用程序的入口点**，通常是一个 GUI 应用程序的主函数。下面是对该代码的逐行解析：

1. `WinMain` 函数的定义：这是 `Windows GUI 应用程序`的入口点函数，它接收**四个参数**：两个`HINSTANCE`参数，一个`LPSTR`命令行参数，和一个表示窗口显示状态的`int`。

2. 本地变量的声明：函数内定义了几个本地变量用于存储服务控制管理器（SCM）句柄和服务句柄，以及一个用于存储服务状态的结构体。

3. 调用`OpenSCManagerA`函数：这个调用试图打开一个到服务控制管理器的连接，以便创建一个服务。权限标志`0xF003Fu`指示程序请求访问SCM的广泛权限。

4. 检查返回值：如果`OpenSCManagerA`成功，它会返回一个句柄；如果失败，返回`NULL`。该句柄被赋值给`v5`。

5. 创建服务：如果成功获取了SCM的句柄，程序会尝试通过调用`CreateServiceA`来创建一个新服务。服务的名称和显示名称都是"Lab10-01"，服务的可执行文件位于`"C:\\Windows\\System32\\Lab10-01.sys"`。

    - `0xF01FFu`为服务的访问权限，表示请求了广泛的权限。
    - `1u`指定服务类型为设备驱动程序。
    - `3u`指定服务的启动类型为“手动启动”。
    - `1u`指定服务的错误控制为“正常”。

6. 检查服务是否创建成功：如果`CreateServiceA`调用失败，代码将尝试通过调用`OpenServiceA`打开现有服务。

7. 启动服务：如果服务已成功创建或打开，代码将尝试使用`StartServiceA`启动服务。

8. 控制服务：如果服务启动，`ControlService`被调用来发送一个**控制信号**（在这里是`1u`，通常是`SERVICE_CONTROL_STOP`，即停止服务，**卸载驱动**）给服务，并将服务的状态更新到`ServiceStatus`结构体中，也就是说，代码**立即卸载驱动**。

9. 清理：如果打开了服务控制管理器但未能创建或打开服务，函数将返回`0`。如果未能打开服务控制管理器，函数将返回`OpenSCManagerA`的结果。

这段代码的目的是在系统上安装并启动一个名为"Lab10-01"的服务，服务是一个驱动程序，并且立即卸载它。

接下来使用IDA分析**sys程序**，即驱动程序，查看入口代码：

```
NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject,
PUNICODE_STRING RegistryPath)
{
  DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_10486;
  return 0;
}
```

这段代码显示了一个Windows驱动程序的入口点函数`DriverEntry`和它关联的卸载函数`sub_10486`。

1. `DriverEntry`函数是**驱动程序的主入口点**。它接收两个参数：一个**指向驱动程序对象**的指针`_DRIVER_OBJECT *DriverObject`和一个**指向注册表路径**的指针`PUNICODE_STRING RegistryPath`。

    - 驱动程序对象包含了指向各种**驱动程序回调函数**的指针。
    - 注册表路径指的是驱动程序的参数存储在**注册表中的位置**。

2. 在`DriverEntry`函数中，`DriverObject->DriverUnload`被设置为函数`sub_10486`的地址。这意味着当**驱动程序被卸载时，`sub_10486`函数将被调用。**

3. `DriverEntry`函数返回状态`NTSTATUS`值为0，这在Win32 API中通常表示成功。

我们查看`sub_10486`函数：

```
NTSTATUS __stdcall sub_10486(int a1)
{
  int ValueData; // [esp+Ch] [ebp-4h] BYREF

  ValueData = 0;
  RtlCreateRegistryKey(0, (PWSTR)L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft");
  RtlCreateRegistryKey(0, (PWSTR)L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall");
  RtlCreateRegistryKey(0, (PWSTR)L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\DomainProfile");
  RtlCreateRegistryKey(
    0,
    (PWSTR)L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\StandardProfile");
  RtlWriteRegistryValue(
    0,
    L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\DomainProfile",
    &ValueName,
    4u,
    &ValueData,
    4u);
  return RtlWriteRegistryValue(
           0,
           L"\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\StandardProfile",
           &ValueName,
           4u,
           &ValueData,
           4u);
}
```

以下是对这个函数的分析：

1. `sub_10486`接收一个整数参数`a1`，但在函数内部并没有使用这个参数。
2. 函数首先将局部变量`ValueData`初始化为0。这个变量将用作注册表值写入的数据。
3. 使用`RtlCreateRegistryKey`函数来**创建若干个注册表键**：
   - `\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft`
   - `\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall`
   - `\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\DomainProfile`
   - `\\Registry\\Machine\\SOFTWARE\\Policies\\Microsoft\\WindowsFirewall\\StandardProfile`

   这些调用尝试在注册表中创建多个键，它们**关联到Windows防火墙的策略设置**。这通常需要管理员权限。
4. 接下来，使用`RtlWriteRegistryValue`写入两个值到`DomainProfile`和`StandardProfile`键：
   - 首先写入到`DomainProfile`键。
   - 然后写入到`StandardProfile`键。

   函数写入的是整数值`0`，这可能是用来修改或**禁用防火墙设置**的。
5. `sub_10486`返回`RtlWriteRegistryValue`的结果，这个调用试图写入值到`StandardProfile`键。

这个驱动程序**在卸载时禁用注册表的Windows防火墙**，这可能是为了**防止防火墙拦截恶意软件的通信，或者保持某种后门打开**。*由于驱动程序运行在内核模式，它拥有对系统的完全控制，包括修改任何注册表项。*

- **内核调试**

那么我们使用`Windbg`来调试它，看看到底会发生什么。*但是如果我们在恶意代码运行之前使用内核调试器，驱动还未加载，我们无法调试它。如果等代码运行完，它又被卸载了*。

为了调试到sys代码，我们**在虚拟机中启动windbg**，在驱动加载和卸载之间下一个断点：

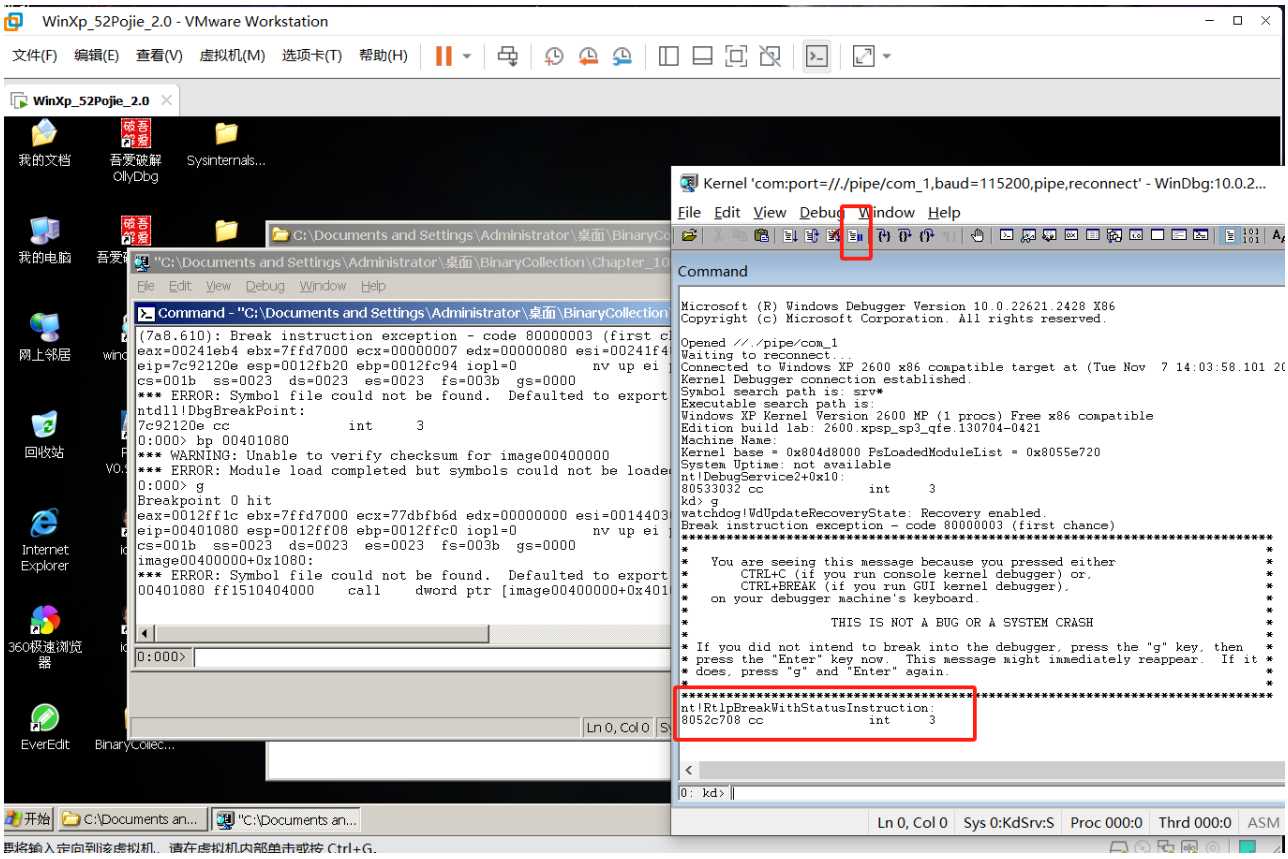
```
bp 00401080
```

这个地址是`ControlService`调用的地址，我们在此下断点恰为时宜。

然后我们**启动程序直到断点命中**，Windbg的信息如下：

```
Breakpoint 0 hit
eax=0012ff1c ebx=7ffd7000 ecx=77dbfb6d edx=00000000 esi=00144030 edi=00144f78
eip=00401080 esp=0012ff08 ebp=0012ffc0 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
image00400000+0x1080:
```

一旦断点暂停，我们就跳出虚拟机，按下主机中Windbg的暂停键，成功断下虚拟机：



然后，在**主机Windbg**输入：

```
!drvobj lab10-01
```

```
0: kd> !drvobj lab10-01
Driver object (8a0fbf38) is for:
 \Driver\Lab10-01

Driver Extension List: (id , addr)

Device Object list:
```

可以看到驱动已经被加载，还未被卸载。驱动对象的地址是 8a0fbf38.

那么我们可以使用dt指令来查看驱动对象的信息，输入：

```
dt _DRIVER_OBJECT 8a0fbf38
```

结果如下：

```
0: kd> dt _DRIVER_OBJECT 8a0fbf38
ntdll!_DRIVER_OBJECT
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : (null)
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xba739000 Void
   +0x010 DriverSize       : 0xe80
   +0x014 DriverSection    : 0x89d3b298 Void
   +0x018 DriverExtension  : 0x8a0fbfe0 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\Lab10-01"
   +0x024 HardwareDatabase : 0x8067f260 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xba739959      long  +0
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0xba739486      void  +0
   +0x038 MajorFunction    : [28] 0x804f55ce      long  nt!IopInvalidDeviceRequest+0
```

可以看到偏移为0x34的信息是驱动被卸载的函数DriverUnload，其地址为0xba739486。

那么我们下一个断点：

```
bp 0xba739486
```

下完断点，我们依次让物理机，虚拟机的windbg恢复运行，**断点立即命中**：

```
0: kd> bp 0xba739486
0: kd> g
Break instruction exception - code 80000003 (first chance)
Lab10_01+0x486:
ba739486 8bff            mov     edi,edi
```

接下来单步调试发现，恶意代码连续调用了三次`RtlCreateRegistryKey`函数，调用了两次`RtlWriteRegistryValue`函数，修改了两次`EnableFirewall`为0，来**禁用防火墙设置**。

那么我们可以使用IDA来分析这个卸载函数，但是需要先计算其相对地址，首先查看装载地址：

```
ba5d2000 ba5d3080   mnmdd       (deferred)
ba5d6000 ba5d7080   RDPCDD      (deferred)
ba5d8000 ba5d9400   Ponerusbmouse    (deferred)
ba6f6000 ba6f6c00   audstub     (deferred)
ba739000 ba739e80   Lab10_01    (no symbols)
ba73e000 ba73eb80   Null        (deferred)
ba7f3000 ba7f3d00   dxgthk      (deferred)
bf000000 bf011600   dxg         (deferred)
bf012000 bf1dda00   Poner_fb    (deferred)
```

那么可以计算

$$0xba739486 - 0xba739000 = 0x486$$

然后在IDA加载sys，由于加载在0x00100000，加上偏移后函数为0x00100486:

```
.text:00010486 ValueData       = dword ptr -4
.text:00010486
.text:00010486                 mov     edi, edi
.text:00010488                 push    ebp
.text:00010489                 mov     ebp, esp
.text:0001048B                 push    ecx
.text:0001048C                 push    ebx
.text:0001048D                 push    esi
.text:0001048E                 mov     esi, ds:RtlCreateRegistryKey
.text:00010494                 push    edi
.text:00010495                 xor     edi, edi
.text:00010497                 push    offset Path     ; "\\Registry\\Machine\\SOFTW
.text:0001049C                 push    edi             ; RelativeTo
.text:0001049D                 mov     [ebp+ValueData], edi
.text:000104A0                 call    esi ; RtlCreateRegistryKey
.text:000104A2                 push    offset aRegistryMachin_0 ; "\\Registry\\Machi
.text:000104A7                 push    edi             ; RelativeTo
.text:000104A8                 call    esi ; RtlCreateRegistryKey
.text:000104AA                 push    offset aRegistryMachin_1 ; "\\Registry\\Machi
.text:000104AF                 push    edi             ; RelativeTo
.text:000104B0                 call    esi ; RtlCreateRegistryKey
.text:000104B2                 mov     ebx, offset aRegistryMachin_2 ; "\\Registry\\
.text:000104B7                 push    ebx             ; Path
.text:000104B8                 push    edi             ; RelativeTo
```

正好是sub_10486的地址，与前面的IDA分析相呼应。

- **Q1：这个程序是否直接修改了注册表(使用procmon 来检查)?**

由上述分析可知，恶意代码唯一**直接写注册表**的操作是调用`RegSetValue`写键值`HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed`。

另外还有使用CreateServiceA函数来对注册表做修改，即**从内核对注册表进行修改，这些修改是无法被procmon探测到的**。

- **Q2：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?**

我们需要进行“**双机调试**”，即在运的行的虚拟机上使用Windbg调试恶意代码，然后在物理机上运行另一个Windbg来调试虚拟机。在虚拟机上恶意代码**驱动卸载函数处设置一个断点**，然后运行到断点处，断点触发，返回物理机的Windbg查看即可。

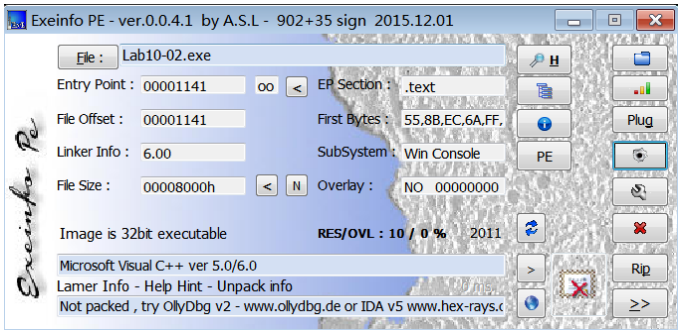
- **Q3：这个程序做了些什么?**

这个程序将**创建一个服务来加载驱动**。然后创建几个键值(具体见分析)，这几个键值将**禁用系统防火墙**。

3.3 Lab10-02.exe

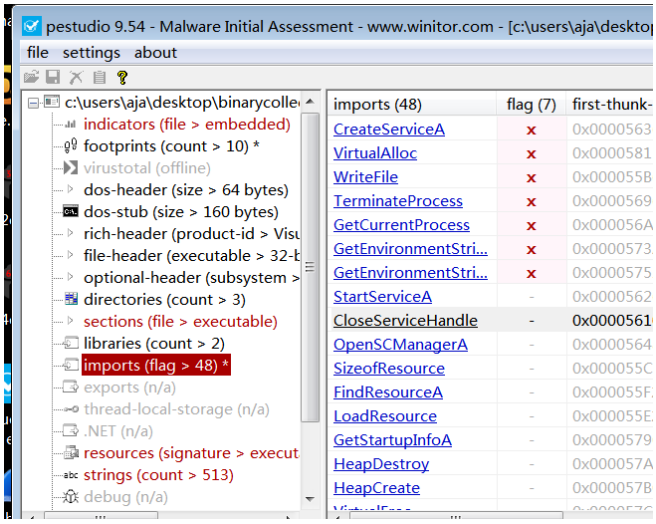
- **静态分析**

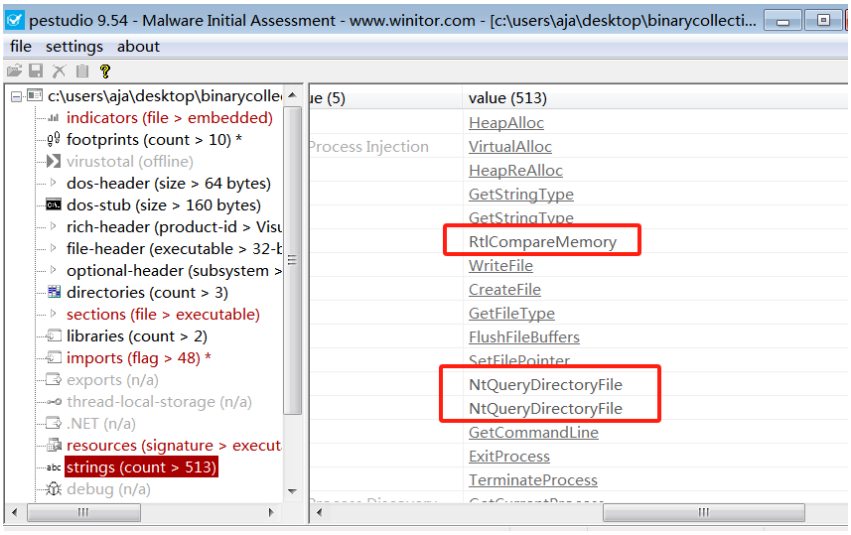
使用exeinfoPE查看加壳：

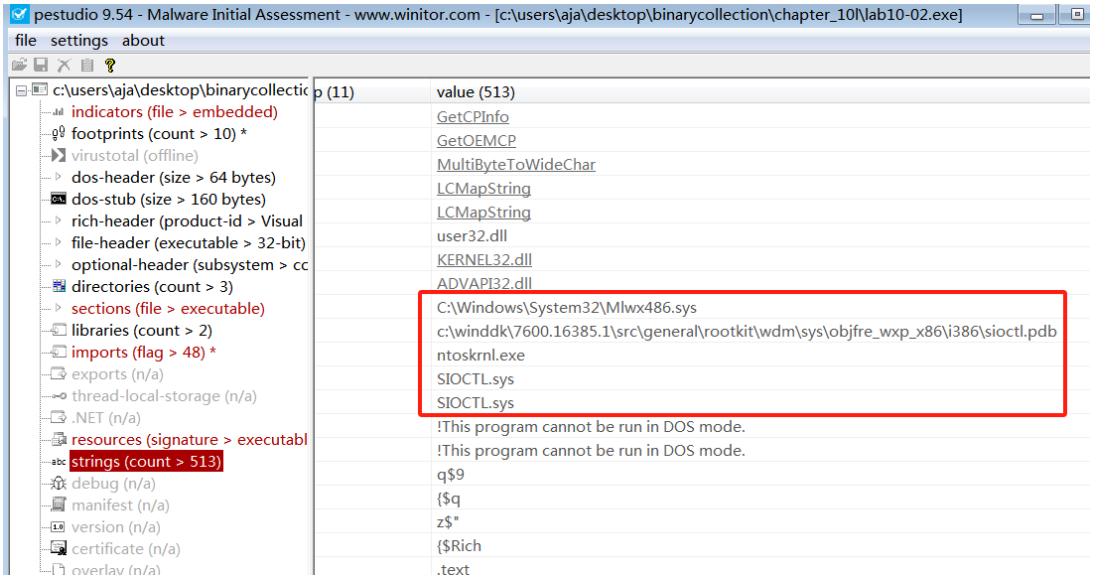


该恶意代码**无加壳**。

我们打开`Pestudio`进行**基本静态分析**，查看其**导入表**和**字符串**：



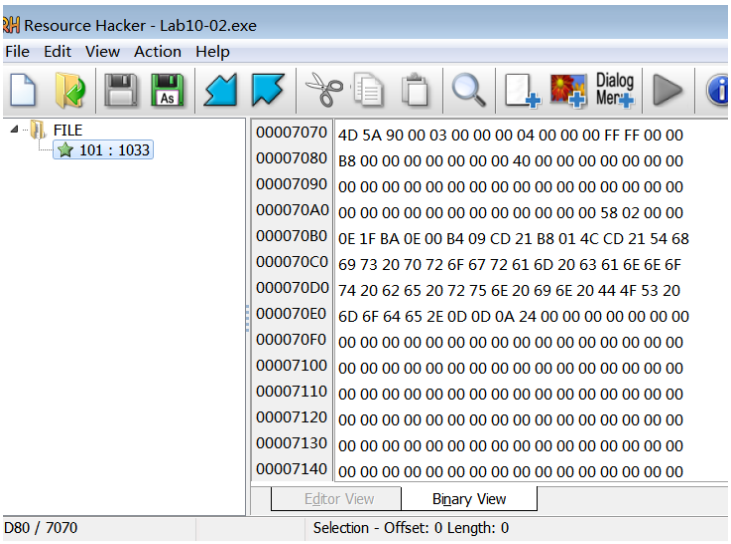




在导入表，我们发现了`CreateServiceA`，`StartServiceA`，`WriteFile`等函数，**猜测代码可能存在创建一个服务，并启动一个服务，以及写入文件等操作。**

在字符串我们看到了`Mlwx486.sys`，`SIOCTL.sys`等字符串，**猜测代码创建一个驱动程序，然后利用驱动程序来创建服务。**

在导入表我们还看到了`LoadResource`，程序访问了资源节，因此我们使用Resource Hacker查看：



发现其资源节存在一个PE文件的内容，可能是另一个恶意代码。

接下来打开IDA对其进行分析，查看main函数的代码：

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    HRSRC ResourceA; // edi
4    HGLOBAL Resource; // ebx
5    HANDLE FileA; // esi
6    DWORD v6; // eax
7    SC_HANDLE v7; // eax
8    SC_HANDLE ServiceA; // eax
9    SC_HANDLE v10; // esi
10   DWORD NumberOfBytesWritten; // [esp+Ch] [ebp-4h] BYREF
11
12   ResourceA = FindResourceA(0, (LPCSTR)0x65, "FILE");
13   Resource = LoadResource(0, ResourceA);
14   if ( ResourceA )
15   {
16     FileA = CreateFileA("C:\\Windows\\System32\\Mlwx486.sys", 0xC0000000,
   0, 0, 2u, 0x80u, 0);
17     if ( FileA != (HANDLE)-1 )
18     {
19       v6 = SizeofResource(0, ResourceA);
20       WriteFile(FileA, Resource, v6, &NumberOfBytesWritten, 0);
21       CloseHandle(FileA);
22       v7 = OpenSCManagerA(0, 0, 0xF003Fu);
23       if ( !v7 )
```

```
      {
        printf("Failed to open service manager.\n");
        return 0;
      }
      ServiceA = CreateServiceA(
                   v7,
                   "486 WS Driver",
                   "486 WS Driver",
                   0xF01FFu,
                   1u,
                   3u,
                   1u,
                   "C:\\Windows\\System32\\Mlwx486.sys",
                   0,
                   0,
                   0,
                   0,
                   0);
      v10 = ServiceA;
      if ( !ServiceA )
      {
        printf("Failed to create service.\n");
        return 0;
      }
      if ( !StartServiceA(ServiceA, 0, 0) )
        printf("Failed to start service.\n");
      CloseServiceHandle(v10);
    }
  }
  return 0;
}
```

恶意代码行为解析:

该程序执行以下步骤:

1. 定位并加载一个资源，通过调用 `FindResourceA` 和 `LoadResource` 函数。
2. 在系统目录 `C:\\Windows\\System32\\` 下创建一个名为 `Mlwx486.sys` 的文件。这个文件位置和命名方式引起了恶意活动的怀疑，因为它似乎是**为了隐藏在正常的系统文件中**。
3. 将先前定位的**资源内容**写入到新创建的 `.sys` 文件中。这通常是一个安装驱动程序的步骤，但在这里，它可能是在**写入一个恶意驱动程序**。
4. 尝试通过 `OpenSCManagerA` 连接到**服务控制管理器**，以获取服务控制的权限。
5. 创建一个名为 "486 WS Driver" 的新服务，并将 `Mlwx486.sys` **设置为服务的可执行文件**。
6. 尝试**启动该服务**。如果服务创建或启动失败，程序会输出错误信息。
7. 关闭所有打开的句柄，包括文件和服务句柄，以进行清理。

那么，该**恶意代码的行为**也就确定了：通过提取和写入资源到系统目录中的文件，并创建及尝试启动一个服务，来安装一个可能的恶意驱动程序，并确保它在系统启动时自动运行，以便在受感染的计算机上保持活跃。

我们没必要进行繁琐的动态分析，直接使用`Pestudio`工具dump出来资源节的文件，命名为Lab10-02.sys，拖入IDA分析：

```
NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject,
PUNICODE_STRING RegistryPath)
{
  PVOID SystemRoutineAddress; // edi
  _DWORD *v3; // eax
  int i; // ecx
  struct _UNICODE_STRING SystemRoutineName; // [esp+8h] [ebp-10h] BYREF
  struct _UNICODE_STRING DestinationString; // [esp+10h] [ebp-8h] BYREF

  RtlInitUnicodeString(&DestinationString, "N");
  RtlInitUnicodeString(&SystemRoutineName, L"KeServiceDescriptorTable");
  SystemRoutineAddress = MmGetSystemRoutineAddress(&DestinationString);
  v3 = *(_DWORD **)MmGetSystemRoutineAddress(&SystemRoutineName);
  for ( i = 0; i < 284; ++i )
  {
    if ( (PVOID)*++v3 == SystemRoutineAddress )
      break;
  }
  dword_1068C = (int)SystemRoutineAddress;
  dword_10690 = (int)v3;
  *v3 = sub_10486;
  return 0;
}
```

这是**驱动程序的主入口点**，它执行一系列初始化操作，包括设置Unicode字符串并利用`MmGetSystemRoutineAddress`函数搜索内核服务表`KeServiceDescriptorTable`，寻找一个特定的系统服务。一旦找到，它将系统服务表中的这个服务的**指针替换为**指向`sub_10486`的指针，这样，当系统尝试调用原始服务时，会转而调用`sub_10486`函数。这是一个**典型的内核挂钩技术，意在拦截和修改系统的正常行为**。

那么接下来查看`sub_10486`函数：

```
NTSTATUS __stdcall sub_10486(
        HANDLE FileHandle,
        HANDLE Event,
        PIO_APC_ROUTINE ApcRoutine,
        PVOID ApcContext,
        PIO_STATUS_BLOCK IoStatusBlock,
        PVOID FileInformation,
```

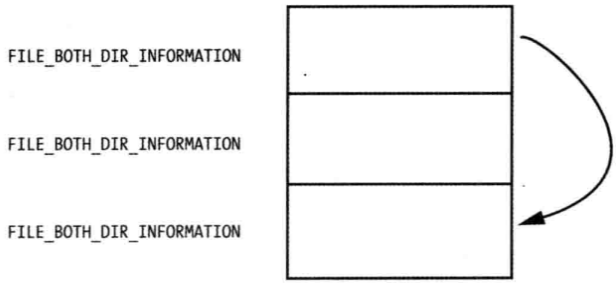
```
        ULONG Length,
        FILE_INFORMATION_CLASS FileInformationClass,
        BOOLEAN ReturnSingleEntry,
        PUNICODE_STRING FileName,
        BOOLEAN RestartScan)
{
  _DWORD *v11; // esi
  NTSTATUS DirectoryFile; // eax
  _DWORD *v13; // edi
  char v14; // bl
  NTSTATUS RestartScana; // [esp+38h] [ebp+30h]

  v11 = FileInformation;
  DirectoryFile = NtQueryDirectoryFile(
                    FileHandle,
                    Event,
                    ApcRoutine,
                    ApcContext,
                    IoStatusBlock,
                    FileInformation,
                    Length,
                    FileInformationClass,
                    ReturnSingleEntry,
                    FileName,
                    RestartScan);
  v13 = 0;
  RestartScana = DirectoryFile;
  if ( FileInformationClass == FileBothDirectoryInformation &&
DirectoryFile >= 0 && !ReturnSingleEntry )
  {
    while ( 1 )
    {
      v14 = 0;
      if ( RtlCompareMemory((char *)v11 + 94, &word_1051A, 8u) == 8 )
      {
        v14 = 1;
        if ( v13 )
        {
          if ( *v11 )
            *v13 += *v11;
          else
            *v13 = 0;
        }
      }
      if ( !*v11 )
        break;
```

```
53          if ( !v14 )
54            v13 = v11;
55          v11 = (_DWORD *)((char *)v11 + *v11);
56        }
57      }
58      return RestartScana;
59    }
```

这个函数作为挂钩的一部分，会在系统试图查询目录文件时被调用。它执行标准的`NtQueryDirectoryFile`查询，但在返回结果之后，会检查文件信息以确定是否需要对其进行修改。如果查询结果中的**文件名称**与驱动程序要查找的模式匹配（此处代码中为`word_1051A`的值，即`"Mlwx"`），该函数会**修改文件信息链表，以隐藏或更改某些文件条目的显示**，用来避免检测到恶意文件。

修改链表示意图如下：



至此完成了分析。

- **Q1：这个程序创建文件了吗?它创建了什么文件?**

它创建了文件`C:\Windows\System32\Mlwx486.sys`，但这个文件**被隐藏了**，无法在硬盘上看到它。

- **Q2：这个程序有内核组件吗?**

这个恶意代码将一个内核模块保存在其**资源节**，将其写入到磁盘并使它作为一个**服务**加载到内核中。

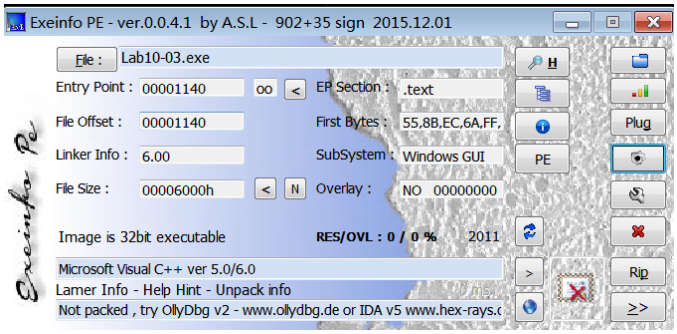
- **Q3：这个程序做了些什么?**

它是一个**用于隐藏文件的Rootkit**，它使用了**SSTD钩子**来替代`NtQueryDirectoryFile`的入口，实现隐藏文件的功能，它将**隐藏任何以Mlwx为开头的文件**。

3.4 Lab10-03.exe & Lab10-03.sys

- **静态分析**

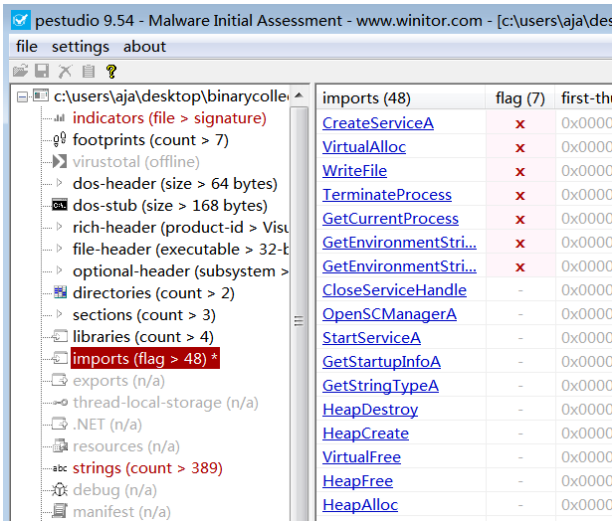
使用exeinfoPE查看加壳：

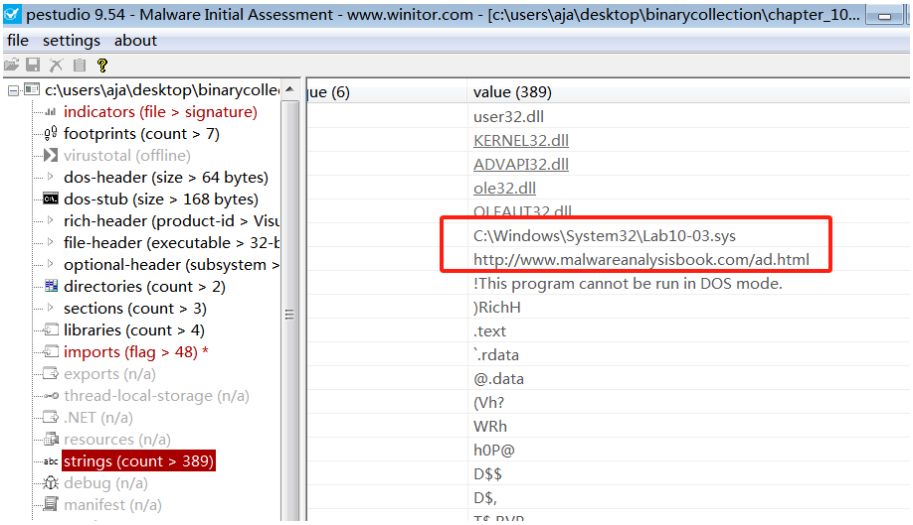


该恶意代码**无加壳**。

我们打开`Pestudio`进行**基本静态分析**，查看其**导入表**和**字符串**：

- **exe:**

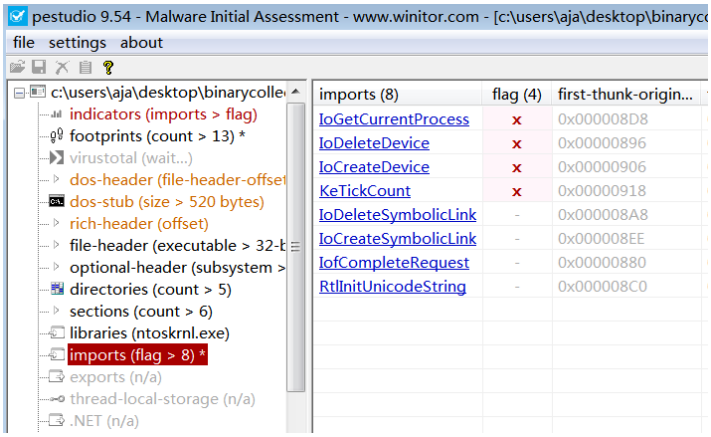


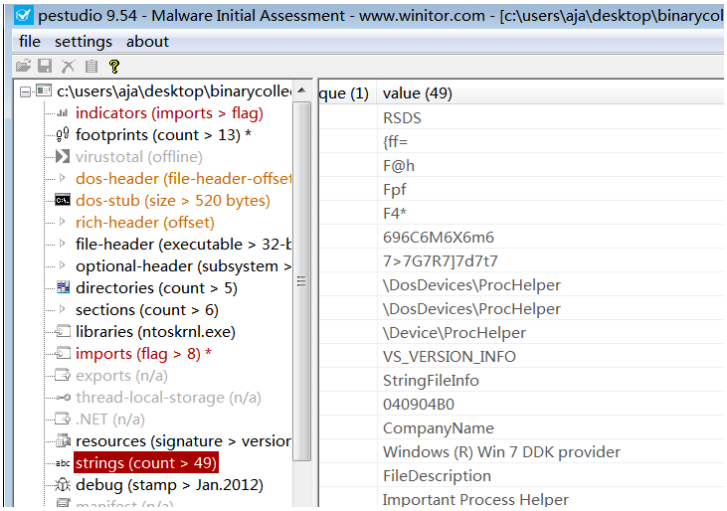


可以看到其加载了`CreateServiceA`，`StartServiceA`等与服务相关的函数，猜测其创建服务并启动服务，并且字符串出现了**驱动的路径**，**猜测其创建该驱动为一个服务。**

另外还出现了一个网址，**该代码的功能似乎与广告有关。**

- **sys:**





该驱动程序使用了`IoGetCurrentProcess`函数，这个函数表面它会**修改正在运行的进程，或获取正在运行的进程信息。**

接下来打开**IDA**对其进行分析：

打开IDA载入EXE，进入main函数查看代码：

```
1  int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR
   lpCmdLine, int nShowCmd)
2  {
3    SC_HANDLE v4; // eax
4    SC_HANDLE ServiceA; // eax
5    SC_HANDLE v6; // esi
6    HANDLE FileA; // eax
7    BSTR v9; // esi
8    LPVOID ppv; // [esp+4h] [ebp-28h] BYREF
9    DWORD BytesReturned; // [esp+8h] [ebp-24h] BYREF
10   VARIANTARG pvarg; // [esp+Ch] [ebp-20h] BYREF
11   __int16 v13[4]; // [esp+1Ch] [ebp-10h] BYREF
12   int v14; // [esp+24h] [ebp-8h]
13
14   v4 = OpenSCManagerA(0, 0, 0xF003Fu);
15   if ( v4 )
```

```
16  {
17    ServiceA = CreateServiceA(
18                 v4,
19                 "Process Helper",
20                 "Process Helper",
21                 0xF01FFu,
22                 1u,
23                 3u,
24                 1u,
25                 "C:\\Windows\\System32\\Lab10-03.sys",
26                 0,
27                 0,
28                 0,
29                 0,
30                 0);
31    v6 = ServiceA;
32    if ( ServiceA )
33      StartServiceA(ServiceA, 0, 0);
34    CloseServiceHandle(v6);
35    FileA = CreateFileA("\\\\.\\ProcHelper", 0xC0000000, 0, 0, 2u, 0x80u,
    0);
36    if ( FileA == (HANDLE)-1 )
37      return 1;
38    DeviceIoControl(FileA, 0xABCDEF01, 0, 0, 0, 0, &BytesReturned, 0);
39    if ( OleInitialize(0) >= 0 )
40    {
41      CoCreateInstance(&rclsid, 0, 4u, &riid, &ppv);
42      if ( ppv )
43      {
44        VariantInit(&pvarg);
45        v13[0] = 3;
46        v14 = 1;
47        v9 =
    SysAllocString(L"http://www.malwareanalysisbook.com/ad.html");
48        while ( 1 )
49        {
50          (*(void (__stdcall **)(LPVOID, BSTR, __int16 *, VARIANTARG *,
    VARIANTARG *, VARIANTARG *))(*(_DWORD *)ppv + 44))(
51            ppv,
52            v9,
53            v13,
54            &pvarg,
55            &pvarg,
56            &pvarg);
57          Sleep(0x7530u);
58        }
```

```
      }
      OleUninitialize();
    }
  }
  return 0;
}
```

我们可以将程序的行为概括为以下几个**关键步骤**：

1. **服务安装和启动**：程序首先尝试打开服务控制管理器，创建一个名为`"Process Helper"`的服务，并指向`"C:\\Windows\\System32\\Lab10-03.sys"`作为服务的执行文件。然后尝试启动这个服务。
2. **与设备驱动程序通信**：接着，程序尝试打开一个设备（可能是上一步创建的服务对应的驱动程序），并通过`DeviceIoControl`发送一个控制码，这通常用于执行驱动程序定义的操作。
3. **周期性访问网页**：程序初始化 COM 库，创建一个 COM 对象实例，然后进入一个无限循环，在该循环中，它会使用`Navigate`方法周期性地导航到 "http://www.malwareanalysisbook.com/ad.html"。每次导航后，程序暂停大约30秒再次执行，达到一个**广告展示效果**。

综上所述，这段代码表明恶意软件正在尝试安装一个服务（很可能与它的驱动程序通信），并且定期地访问或打开一个指定的网址，可能用于下载更新、接收命令或通过广告点击欺诈来产生收入。

我们使用IDA查看驱动程序入口函数：

```
NTSTATUS __stdcall DriverEntry(_DRIVER_OBJECT *DriverObject,
PUNICODE_STRING RegistryPath)
{
  NTSTATUS result; // eax
  int v3; // esi
  struct _UNICODE_STRING SymbolicLinkName; // [esp+8h] [ebp-14h] BYREF
  struct _UNICODE_STRING DestinationString; // [esp+10h] [ebp-Ch] BYREF
  PDEVICE_OBJECT DeviceObject; // [esp+18h] [ebp-4h] BYREF

  DeviceObject = 0;
  RtlInitUnicodeString(&DestinationString, L"\\Device\\ProcHelper");
  result = IoCreateDevice(DriverObject, 0, &DestinationString, 0x22u,
0x100u, 0, &DeviceObject);
  if ( result >= 0 )
  {
    DriverObject->MajorFunction[0] = (PDRIVER_DISPATCH)sub_10606;
    DriverObject->MajorFunction[2] = (PDRIVER_DISPATCH)sub_10606;
    DriverObject->MajorFunction[14] = (PDRIVER_DISPATCH)sub_10666;
    DriverObject->DriverUnload = (PDRIVER_UNLOAD)sub_1062A;
    RtlInitUnicodeString(&SymbolicLinkName, &word_107DE);
    v3 = IoCreateSymbolicLink(&SymbolicLinkName, &DestinationString);
```

```
    if ( v3 < 0 )
      IoDeleteDevice(DeviceObject);
    return v3;
  }
  return result;
}
```

`DriverEntry` 函数是驱动程序的初始化入口点，当驱动程序被加载时，这个函数被调用。其基本步骤包括：

1. **创建设备对象**：
   - `RtlInitUnicodeString` 初始化**设备名称字符串**（`\\Device\\ProcHelper`）。
   - `IoCreateDevice` 创建一个设备对象，这是驱动程序与外界通信的接口。
2. **设置I/O请求处理函数**：
   - `DriverObject->MajorFunction[IRP_MJ_CREATE]` 和 `DriverObject->MajorFunction[IRP_MJ_CLOSE]` 被设置为 `sub_10606`，用于处理设备的创建和关闭请求。
   - `DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]` 被设置为 `sub_10666`，用于处理设备控制请求。
3. **设置卸载函数**：
   - `DriverObject->DriverUnload` 被设置为 `sub_1062A`，用于在驱动程序卸载时执行清理工作。
4. **创建符号链接**：
   - `IoCreateSymbolicLink` 创建一个**用户模式可以访问的符号链接**。
5. **错误处理**：
   - 如果创建符号链接失败，则调用 `IoDeleteDevice` 删除设备对象并返回错误代码。

那么我们来分别看一下它调用的三个函数：

```
int __stdcall sub_10606(int a1, PIRP Irp)
{
  Irp->IoStatus.Status = 0;
  Irp->IoStatus.Information = 0;
  IofCompleteRequest(Irp, 0);
  return 0;
}

int __stdcall sub_10666(int a1, PIRP Irp)
{
```

```
  PEPROCESS CurrentProcess; // eax
  _DWORD *v3; // ecx

  CurrentProcess = IoGetCurrentProcess();
  v3 = (_DWORD *)*((_DWORD *)CurrentProcess + 35);
  CurrentProcess = (PEPROCESS)((char *)CurrentProcess + 136);
  *v3 = *(_DWORD *)CurrentProcess;
  *(_DWORD *)(*(_DWORD *)CurrentProcess + 4) = *((_DWORD *)CurrentProcess
+ 1);
  Irp->IoStatus.Status = 0;
  Irp->IoStatus.Information = 0;
  IofCompleteRequest(Irp, 0);
  return 0;
}

void __stdcall sub_1062A(int a1)
{
  struct _DEVICE_OBJECT *v1; // esi
  struct _UNICODE_STRING DestinationString; // [esp+4h] [ebp-8h] BYREF

  v1 = *(struct _DEVICE_OBJECT **)(a1 + 4);
  RtlInitUnicodeString(&DestinationString, &SourceString);
  IoDeleteSymbolicLink(&DestinationString);
  if ( v1 )
    IoDeleteDevice(v1);
}
```

咋一看看不出来这些函数到底是在干什么，我们使用内核调试分析。

- **内核调试**

我们知道**设备对象**位于`\Device\ProcHelper`，运行恶意代码，然后找到这个设备对象：

```
0: kd> !devobj ProcHelper
Device object (8a07be00) is for:
 ProcHelper \Driver\Process_Helper DriverObject 8a189788
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
SecurityDescriptor e1351050 DevExt 00000000 DevObjExt 8a07beb8
ExtensionFlags (0000000000)
Characteristics (0x00000100)  FILE_DEVICE_SECURE_OPEN
Device queue is not busy.
```

Driverobject包含所有函数的指针，当用户空间的程序访问设备对象时调用这些函数。Driverobject存储在一个叫做DRIVER OBJECT的数据结构中。我们可以使用dt命令**查看标注的驱动对象**:

```
0: kd> dt nt!_DRIVER_OBJECT 8a189788
   +0x000 Type             : 0n4
   +0x002 Size             : 0n168
   +0x004 DeviceObject     : 0x8a07be00 _DEVICE_OBJECT
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xba736000 Void
   +0x010 DriverSize       : 0xe00
   +0x014 DriverSection    : 0x89e65a58 Void
   +0x018 DriverExtension  : 0x8a189830 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING "\Driver\Process Helper"
   +0x024 HardwareDatabase : 0x8067f260 _UNICODE_STRING "\REGISTRY\MACHINE\HARDWARE\DESCRIPTION\SYSTEM"
   +0x028 FastIoDispatch   : (null)
   +0x02c DriverInit       : 0xba7367cd     long  +ffffffffba7367cd
   +0x030 DriverStartIo    : (null)
   +0x034 DriverUnload     : 0xba73662a     void  +ffffffffba73662a
   +0x038 MajorFunction    : [28] 0xba736606     long  +ffffffffba736606
```

这里面就包括DriverUnload函数，它将删除符号链表和DriverEntry创建的设备。

接下来查看**主函数表项**：

```
0: kd> dd 8a189788+0x38 L1C
ReadVirtual: 8a1897c0 not properly sign extended
8a1897c0  ba736606 804f55ce ba736606 804f55ce
8a1897d0  804f55ce 804f55ce 804f55ce 804f55ce
8a1897e0  804f55ce 804f55ce 804f55ce 804f55ce
8a1897f0  804f55ce 804f55ce ba736666 804f55ce
8a189800  804f55ce 804f55ce 804f55ce 804f55ce
8a189810  804f55ce 804f55ce 804f55ce 804f55ce
8a189820  804f55ce 804f55ce 804f55ce 804f55ce
```

其中，大部分函数为`0x804f55ce`，我们找到这个函数：

```
0: kd> ln ffffffff804f55ce
Browse module
Set bu breakpoint

(804f55ce)   nt!IopInvalidDeviceRequest   |  (804f5604)   nt!IopGetDeviceAttachmentBase
Exact matches:
    nt!IopInvalidDeviceRequest (_IopInvalidDeviceRequest@8)
```

它用于处理驱动无法处理的请求。

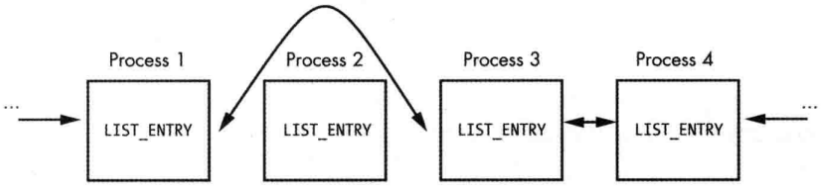
查看主函数表另一个函数`ba736606`，它实际上就是上面IDA的`sub_10606`，它只是调用`IofCompleteRequest`后就返回了。

那么我们查看主函数表的函数`ba736666`，它实际上就是IDA的`sub_10666`，它调用`IoGetCurrentProcess`，返回了调用`DeviceIoControl`进程的`EPROCESS`结构，然后访问其偏移量为**0x88**的数据，接着访问偏移量为**0x8C**的下一个数据。

那么我们可以使用dt命令来查看对应偏移量的数据在运行时到底是什么：

```
0: 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
   +0x0cc WorkingSetLock   : _FAST_MUTEX
   +0x0ec WorkingSetPage   : Uint4B
   +0x0f0 AddressCreationLock : _FAST_MUTEX
   +0x110 HyperSpaceLock   : Uint4B
   +0x114 ForkInProgress   : Ptr32 _ETHREAD
```

发现数据是`_LIST_ENTRY`，这是一个**双向链表**，那么`sub_10666`实际上的操作是实现了**修改进程链表，来隐藏当前进程**的功能，做法如下所示：



至此完成分析。

- **Q1：这个程序做了些什么？**

exe恶意代码将加载驱动，然后**每30秒弹出一次广告**，这个驱动通过摘除进程环境块PEB，来隐藏它的进程。

- **Q2：一旦程序运行，你怎样停止它？**

只能通过**重启**来停止程序。

- **Q3：它的内核组件做了什么操作？**

从**进程链接表**去除`DeviceIoControl`请求。

3.5 yara规则编写

综合以上，可以完成该恶意代码的yara规则编写：

```
//首先判断是否为PE文件
private rule IsPE
{
condition:
     filesize < 10MB and     //小于10MB
    uint16(0) == 0x5A4D and //"MZ"头
    uint32(uint32(0x3C)) == 0x00004550 // "PE"头
}

//Lab10-01
rule lab10_1
{
strings:
    $s1 = "C:\\Windows\\System32\\Lab10-01.sys"
    $s2 = "CreateService"
condition:
    IsPE and $s1 and $s2
}

//Lab10-01
```

```
rule lab10_1_sys
{
strings:
    $s1 = "EnableFirewall"
    $s2 = "Lab10-01.sys"
condition:
    $s1 and $s2
}

//Lab10-02
rule lab10_2
{
strings:
    $s1 = "NtQueryDirectoryFile"
    $s2 = "C:\\Windows\\System32\\Mlwx486.sys"
condition:
    IsPE and $s1 and $s2
}

//Lab10-03
rule lab10_3
{
strings:
    $s1 = "C:\\Windows\\System32\\Lab10-03.sys"
    $s2 = "Process Helper"
condition:
    IsPE and $s1 and $s2
}

//Lab10-03
rule lab10_3_sys
{
strings:
    $s1 = "IoGetCurrentProcess"
    $s2 = "Lab10-03.sys"
condition:
    $s1 and $s2
}
```

把上述Yara规则保存为`rule_ex10.yar`，然后在Chapter_10L上一个目录输入以下命令:

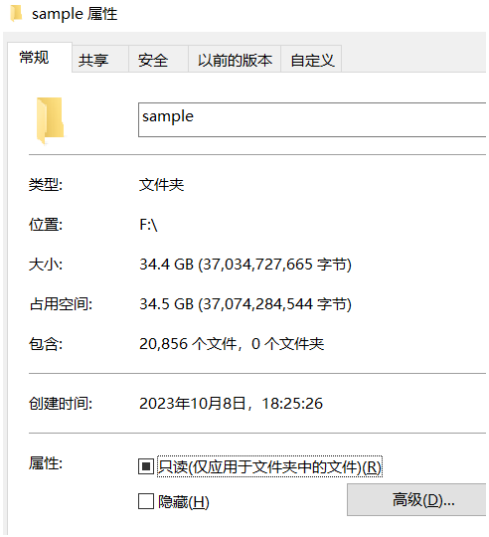
```
yara64 -r rule_ex10.yar Chapter_10L
```

结果如下，**样本检测成功**:

```
D:\study\大三\恶意代码分析与防治技术\Practical Malware Analysis Labs\BinaryCollection>yara64 -r rule_ex10.yar
Chapter_10L
lab10_1 Chapter_10L\Lab10-01.exe
lab10_1_sys Chapter_10L\rule_ex10.yar
lab10_3_sys Chapter_10L\rule_ex10.yar
lab10_3 Chapter_10L\Lab10-03.exe
lab10_2 Chapter_10L\Lab10-02.exe
```

接下来对运行`Scan.py`获得的sample进行扫描。

sample文件夹大小为34.4GB，含有20856个可执行文件：



我们编写一个**yara扫描脚本**`yara_unittest.py`来完成扫描：

```python
import os
import yara
import datetime

# 定义YARA规则文件路径
rule_file = './rule_ex10.yar'

# 定义要扫描的文件夹路径
folder_path = './sample/'

# 加载YARA规则
try:
    rules = yara.compile(rule_file)
except yara.SyntaxError as e:
    print(f"YARA规则语法错误: {e}")
    exit(1)

# 获取当前时间
start_time = datetime.datetime.now()

# 扫描文件夹内的所有文件
scan_results = []
```

```python

for root, dirs, files in os.walk(folder_path):
    for file in files:
        file_path = os.path.join(root, file)
        try:
            matches = rules.match(file_path)
            if matches:
                scan_results.append({'file_path': file_path, 'matches': [str(match) for match in matches]})
        except Exception as e:
            print(f"扫描文件时出现错误: {file_path} - {str(e)}")

# 计算扫描时间
end_time = datetime.datetime.now()
scan_time = (end_time - start_time).seconds

# 将扫描结果写入文件
output_file = './scan_results.txt'

with open(output_file, 'w') as f:
    f.write(f"扫描开始时间: {start_time.strftime('%Y-%m-%d %H:%M:%S')}\n")
    f.write(f'扫描耗时: {scan_time}s\n')
    f.write("扫描结果:\n")
    for result in scan_results:
        f.write(f"文件路径: {result['file_path']}\n")
        f.write(f"匹配规则: {', '.join(result['matches'])}\n")
        f.write('\n')

print(f"扫描完成，耗时{scan_time}秒，结果已保存到 {output_file}")
```

运行得到扫描结果文件如下：

```
扫描开始时间: 2023-11-07 00:15:43
扫描耗时: 92s
扫描结果:
文件路径: ./sample/Lab10-01.exe
匹配规则: lab10_1

文件路径: ./sample/Lab10-02.exe
匹配规则: lab10_2

文件路径: ./sample/Lab10-03.exe
匹配规则: lab10_3
```

将几个实验样本扫描了出来，共耗时 **92秒**。

3.6 IDA Python脚本编写

我们可以编写如下Python脚本来辅助分析：

```
import idaapi
import idautils
import idc

# 获得所有已知API的集合
def get_known_apis():
    known_apis = set()

    def imp_cb(ea, name, ord):
        if name:
            known_apis.add(name)
        return True

    for i in range(ida_nalt.get_import_module_qty()):
        ida_nalt.enum_import_names(i, imp_cb)

    return known_apis

known_apis = get_known_apis()

def get_called_functions(start_ea, end_ea, known_apis):
    """
    给定起始和结束地址，返回该范围内调用的所有函数的集合。
    """
    called_functions = set()  # 使用集合避免重复
    for head in idautils.Heads(start_ea, end_ea):
        if idc.is_code(idc.get_full_flags(head)):
            insn = idautils.DecodeInstruction(head)
            if insn and insn.get_canon_mnem() == "call":
                for op in insn.ops:
                    if op.type in [idaapi.o_mem, idaapi.o_phrase,
idaapi.o_displ]:
                        if op.addr != idaapi.BADADDR:
                            func_name = idc.get_name(op.addr,
ida_name.GN_VISIBLE)
                            if func_name:
                                called_functions.add(func_name)
                        break
    return called_functions

def main(name, known_apis):
    # 获取该函数的地址
```

```
    main_addr = idc.get_name_ea_simple(name)
    if main_addr == idaapi.BADADDR:
        print("找不到 '{}' 函数。".format(name))
        return

    main_end_addr = idc.find_func_end(main_addr)

    # 列出该函数调用的所有函数
    main_called_functions = get_called_functions(main_addr, main_end_addr,
known_apis)

    print("被 '{}' 调用的函数:".format(name))
    for func_name in main_called_functions:
        # 如果函数名在已知API集合中，则不进一步追踪
        if func_name in known_apis:
            print(func_name)
            continue

        print(func_name)

        # 获取每个函数的结束地址
        func_ea = idc.get_name_ea_simple(func_name)
        if func_ea == idaapi.BADADDR:
            continue

        func_end_addr = idc.find_func_end(func_ea)

        # 列出被main调用的函数内部调用的函数或API
        called_by_func = get_called_functions(func_ea, func_end_addr,
known_apis)
        print("\t被 {} 调用的函数/APIs: ".format(func_name))
        for sub_func_name in called_by_func:
            if sub_func_name in known_apis:
                continue
            print("\t\t{}".format(sub_func_name))

if __name__ == "__main__":
    names = ['_main', '_WinMain@16']
    for name in names:
        main(name, known_apis)

```

该 IDAPython 脚本的功能是自动化地遍历特定函数的指令，识别所有直接的函数调用，并排除那些属于已知标准库或系统调用的函数。它输出每个分析的函数所调用的函数列表，并对那些不在已知 API 列表中的函数递归地执行相同的操作，从而构建出一个函数调用图。

对恶意代码的EXE分别运行上述`IDA Python`脚本，结果如下：

- **Lab10-01.exe**

```
找不到 '_main' 函数。
被 '_WinMain@16' 调用的函数:
OpenSCManagerA
OpenServiceA
ControlService
CreateServiceA
StartServiceA
```

- **Lab10-02.exe**

```
被 '_main' 调用的函数:
OpenSCManagerA
CloseServiceHandle
CreateFileA
SizeofResource
CreateServiceA
FindResourceA
CloseHandle
LoadResource
StartServiceA
WriteFile
找不到 '_WinMain@16' 函数。
```

- **Lab10-03.exe**

```
找不到 '_main' 函数。
被 '_WinMain@16' 调用的函数:
CreateFileA
OleInitialize
OleUninitialize
OpenSCManagerA
VariantInit
SysAllocString
CoCreateInstance
DeviceIoControl
StartServiceA
CloseServiceHandle
CreateServiceA
```

据此我们可以直观地看出**函数调用的API**。

4 实验结论及心得体会

在本次实验，我们分析了几个带有sys驱动程序的样本，它们需要在内核模型下运行，因此需要使用Windbg进行调试。在结合IDA分析后，我们不难分析出每个样本的特征，它们的功能和实现方法等，这些可以作为该类样本的经验特征。

心得体会：通过此次实验，我对使用WinDbg进行内核调试有了深入的理解和实践。在实验的过程中，我不仅学习了如何设置和使用WinDbg，还掌握了如何在用户模式和内核模式下进行代码调试的技巧。最初，配置WinDbg的符号路径和建立起内核调试环境对我来说是一个挑战。随着实验的深入，我逐渐理解了符号文件在调试过程中的重要性，它们如何帮助我识别函数调用和变量。此外，我也了解到用户模式和内核模式之间的根本差异，以及每种模式下的调试策略。

通过这次实验，我不仅提高了我的技术能力，也加深了我对操作系统内核工作原理的理解。此外，我也认识到在现实世界中对付恶意软件的复杂性，以及作为一名安全专业人士，持续学习和实践的重要性。