

**恶意代码课程实验报告**

**实验十：WinDBG内核调试**

****

学 院 网络安全学院

专 业 信息安全

学 号 2110688

姓 名 史文天

班 级 1063

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。**

1. **实验原理**

**配置虚拟机环境，安装动态、静态分析工具，使用静态和动态分析工具。**

1. **实验过程**
2. **Lab 10-01**

**本实验包括一个驱动程序和一个可执行文件。你可以从任意位置运行可执行文件，但为了使程序能够正常运行，必须将驱动程序放到C:\Windows\System32目录下，这个目录在受害者计算机中已经存在。可执行文件是Lab10-01.exe,驱动程序是Lab 10-01.sys.**

**问题**

1. **这个程序是否直接修改了注册表？**

**修改了。**

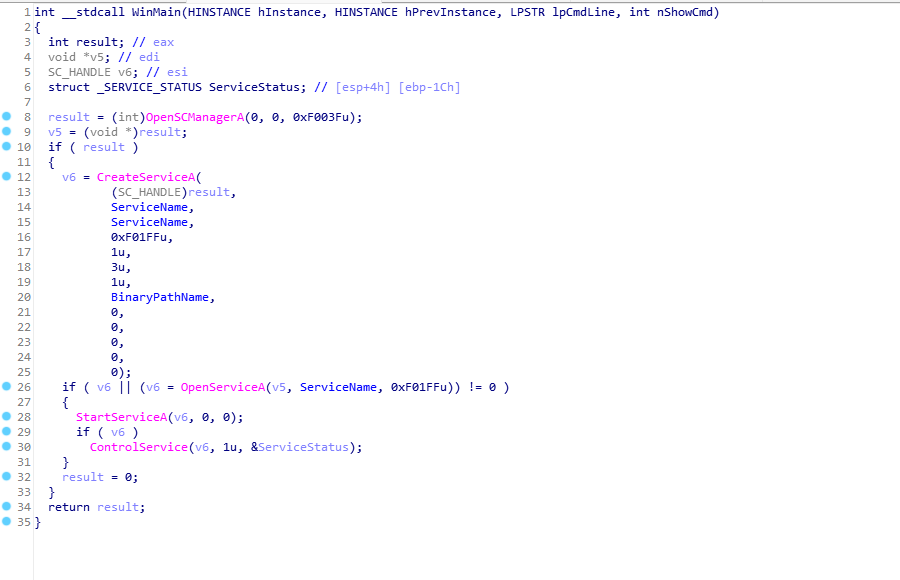
1. **用户态的程序调用了ControlService函数，你是否能够使用WinDbg设置一个断点，以此来观察由于ControlService的调用导致内核执行了怎样的操作？**

**修改注册表，关闭防火墙。**

1. **这个程序做了些什么？**

**通过创建服务来加载驱动，驱动代码会创建并修改注册表键值来关闭防火墙。**

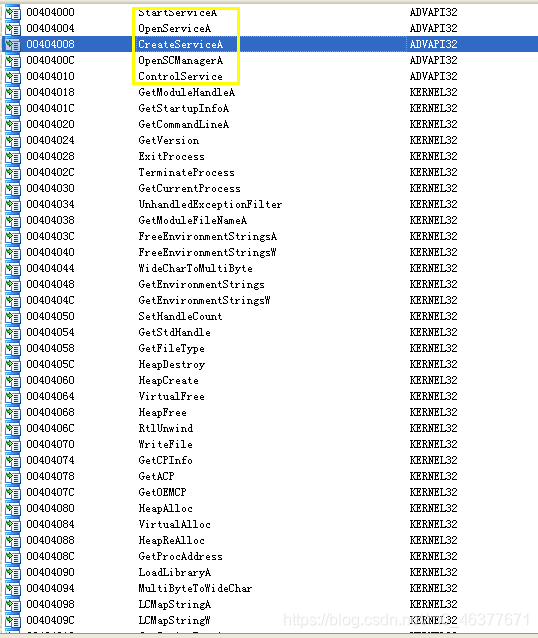
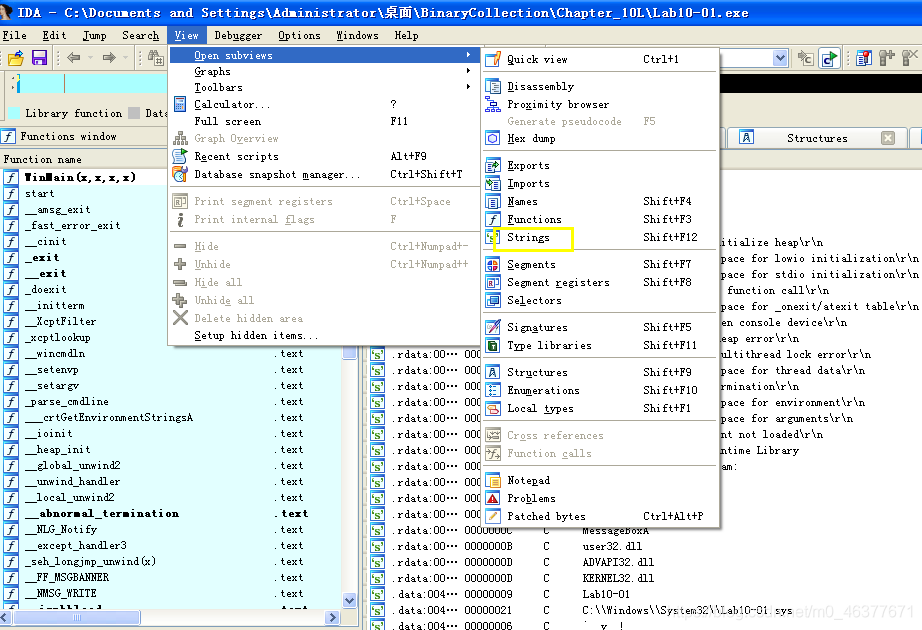
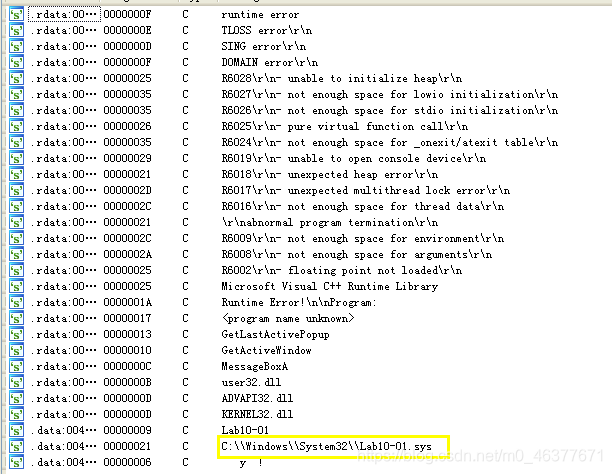
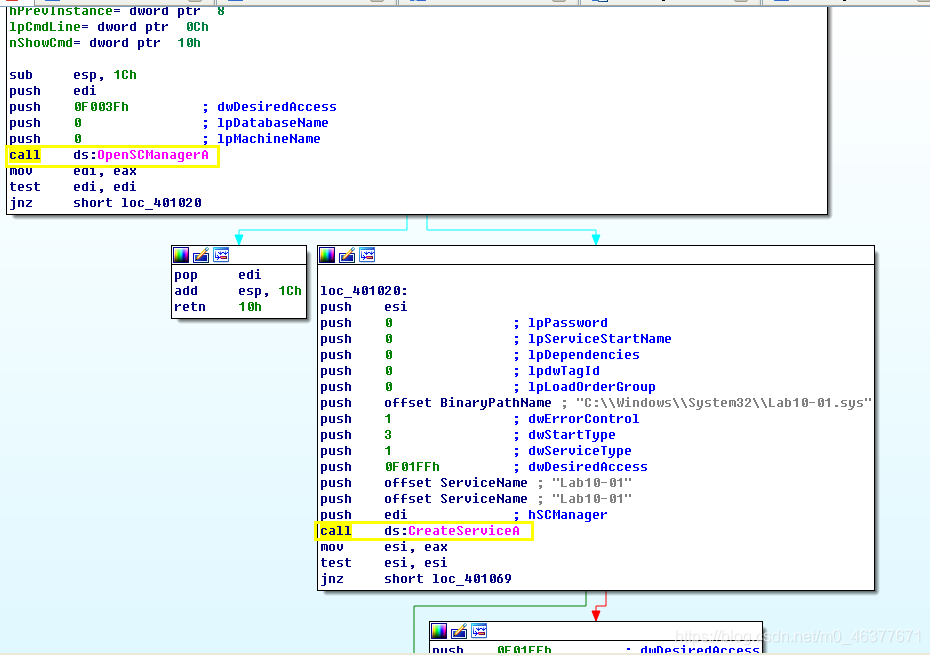
**使用ida对程序lab 10-01.exe进行分析。**

****

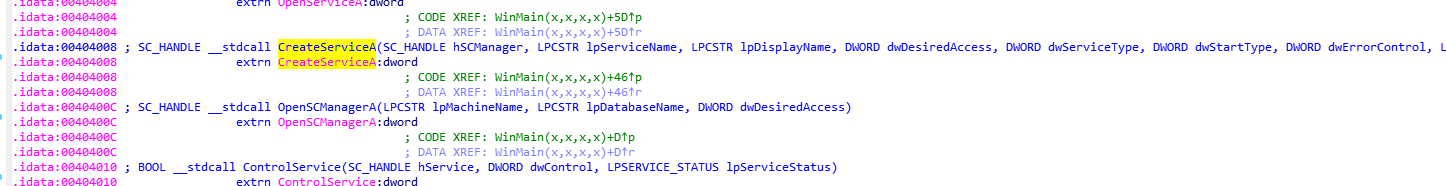
**运行看到服务的交互，是由services.exe完成的！services.exe是微软Windows操作系统的一部分。用于管理启动和停止服务。该进程也会处理在计算机启动和关机时运行的服务。这个程序对你系统的正常运行是非常重要的。终止进程后会重启。**

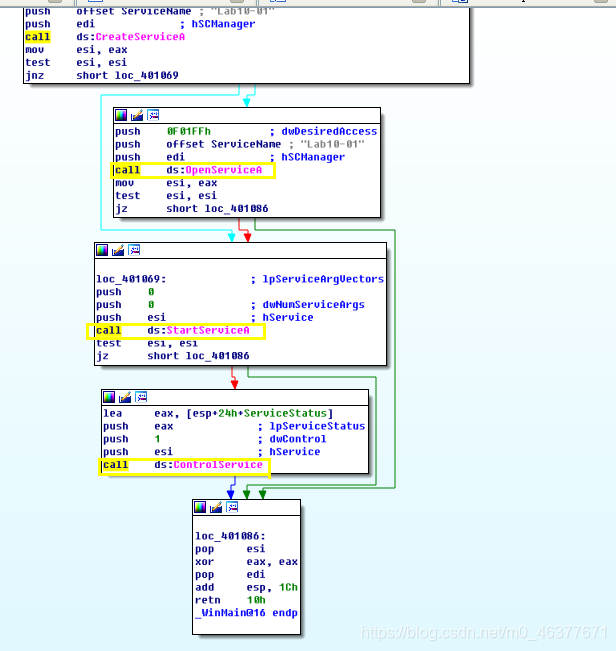
**这尼玛EDR后面咋关联检测啊。、。。。。todo。。。**

**先反汇编看下，可以知道是在创建服务并启动！然后再看strings，基本判断是通过驱动方式来加载恶意代码。**

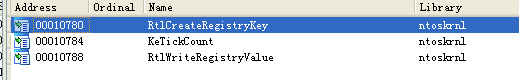
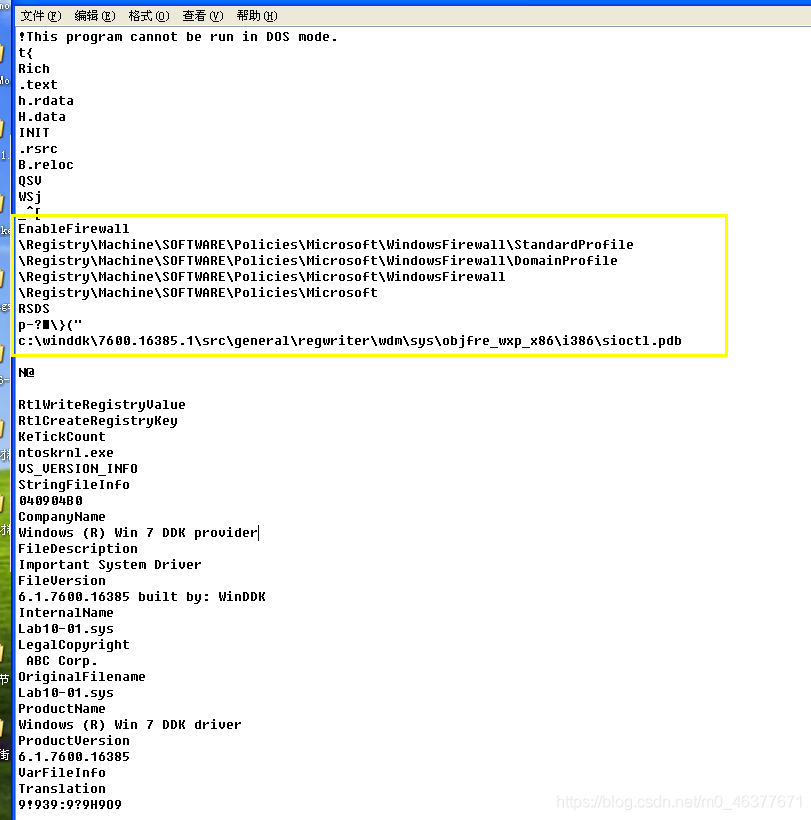
**  
可以看到有与服务相关的敏感操作。通过ida的strings进行分析。  
  
  
可以看到出现了敏感文件路径。  
  
OpenSCManger:在指定及其上创建与服务控制管理程序的联系，并打开指定的数据库，返回的是一个服务管理器的句柄。  
CreateService:创建一个服务对象，并将它添加到指定的服务控制管理程序的数据库中。Service为创建的服务名称，此处为lab10-01。**

**dwServiceType为服务类型，1表示此服务为驱动服务（此文件会加载到内核中去）.dwStartType为服务启动类型，3表示此服务会自动启动 。**

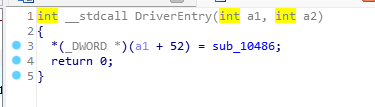
****

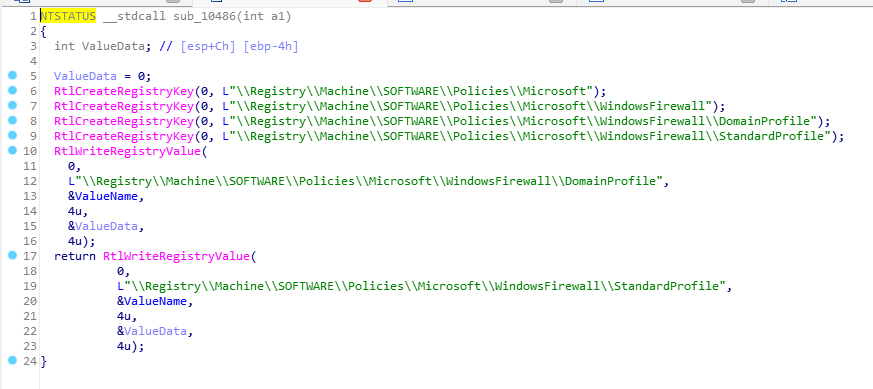
**dwErrorControl表示严重性错误，以及采取的行动，如果这项服务无法启动，1表示启动程序在事件日志中记录，但继续启动操作。BinaryPathName表示服务二进制文件的完全限定路径，dwDesiredAccess为访问权限，0xF01FF表示除此表中的所有访问权限外，还包括STANDARD\_RIGHTS\_REQUIRED 。  
  
如果服务存在导致服务创建失败，则使用OpenService打开同名服务。如果打开成功，使用StartService开启服务。  
ControlService: hservice，OpenService或CreateService 返回的服务句柄。  
dwControl，要发送的控制码，此处为1，表示CONTROL \_SERVICE\_STOP,将会卸载驱动并调用驱动卸载的函数。==》此次实验就是要在运行过程中，打断点，看看这个程序究竟安装了什么驱动！因为它会自删除驱动。。。**

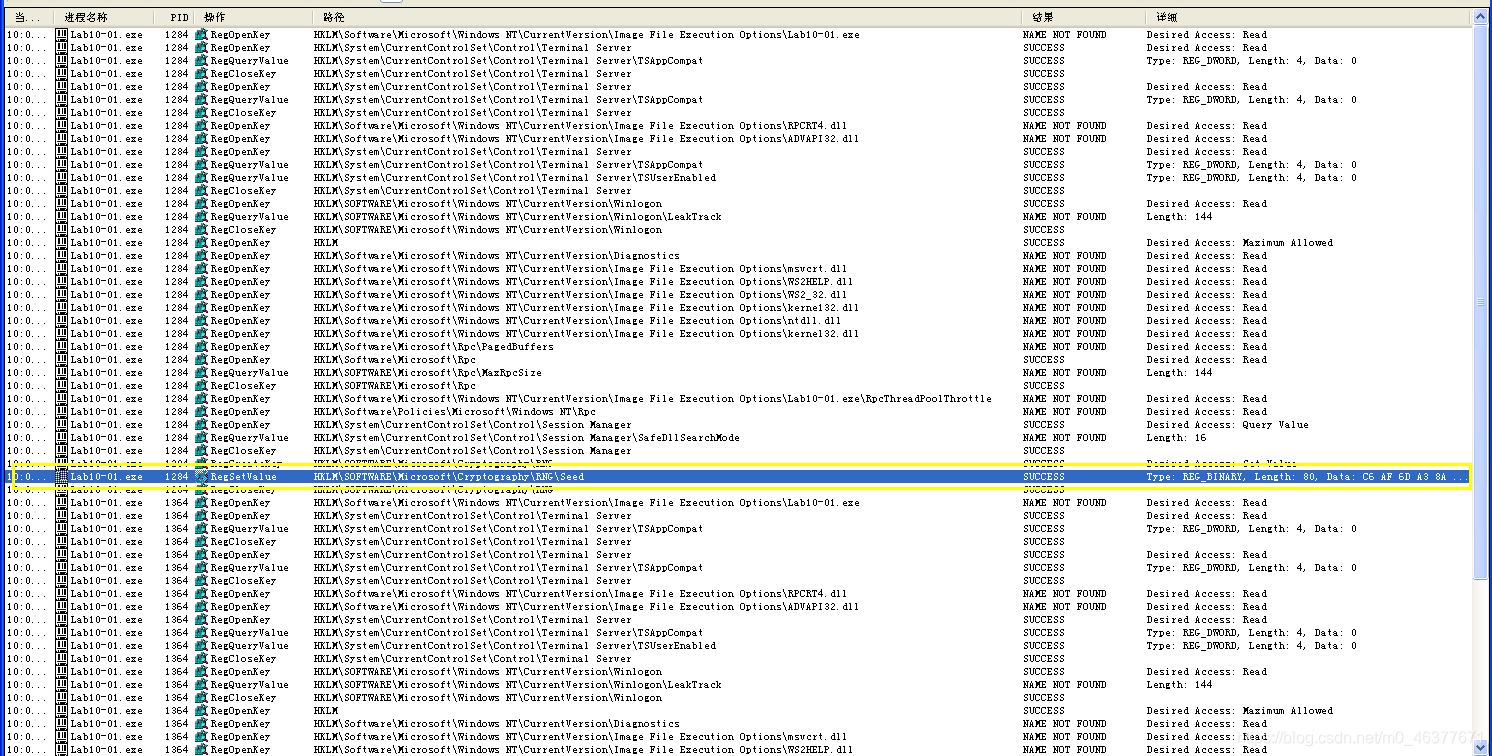
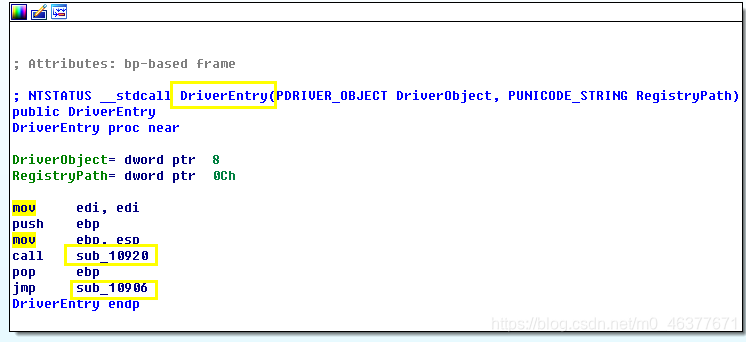
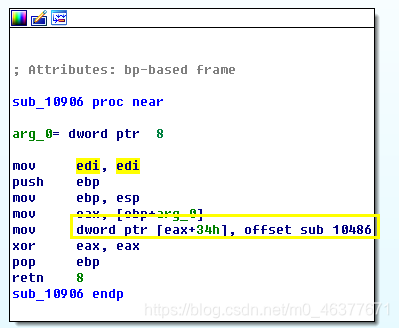
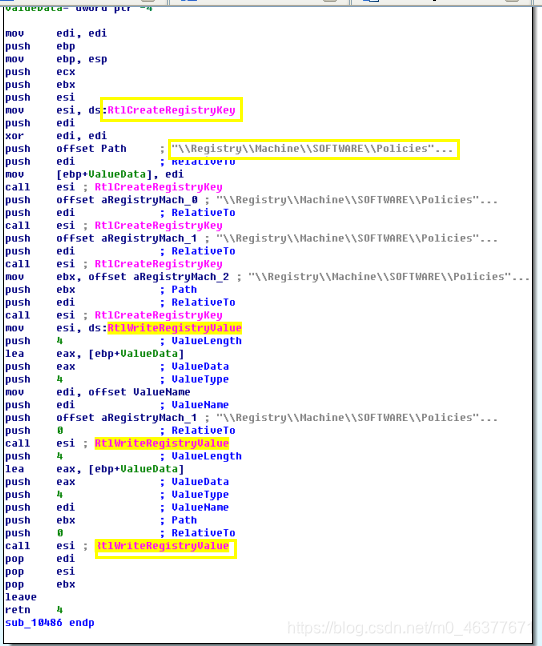
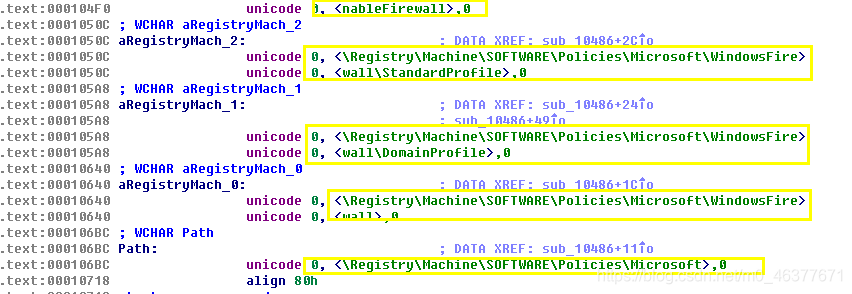
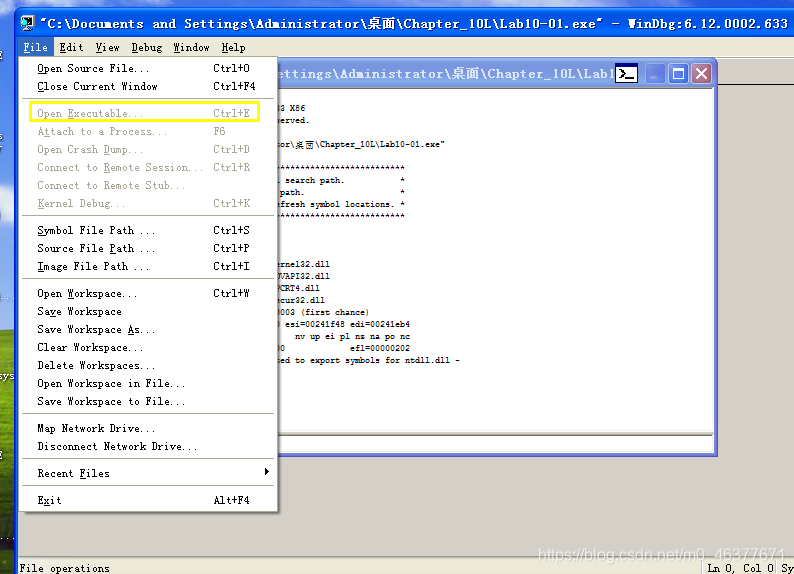
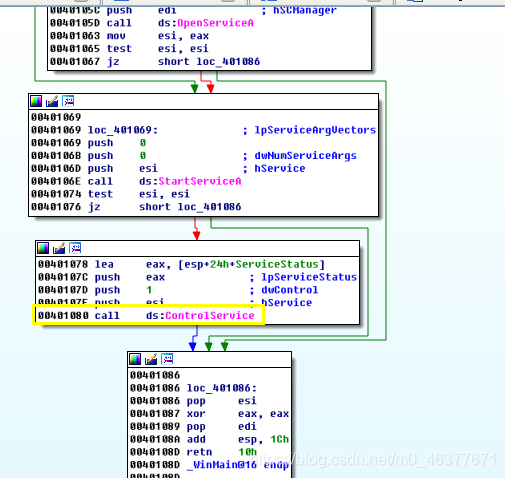
**IpServiceStatus，返回值，指向存储服务最新状态的结构体Service，返回信息来自SCM中最近的服务状态报告。**

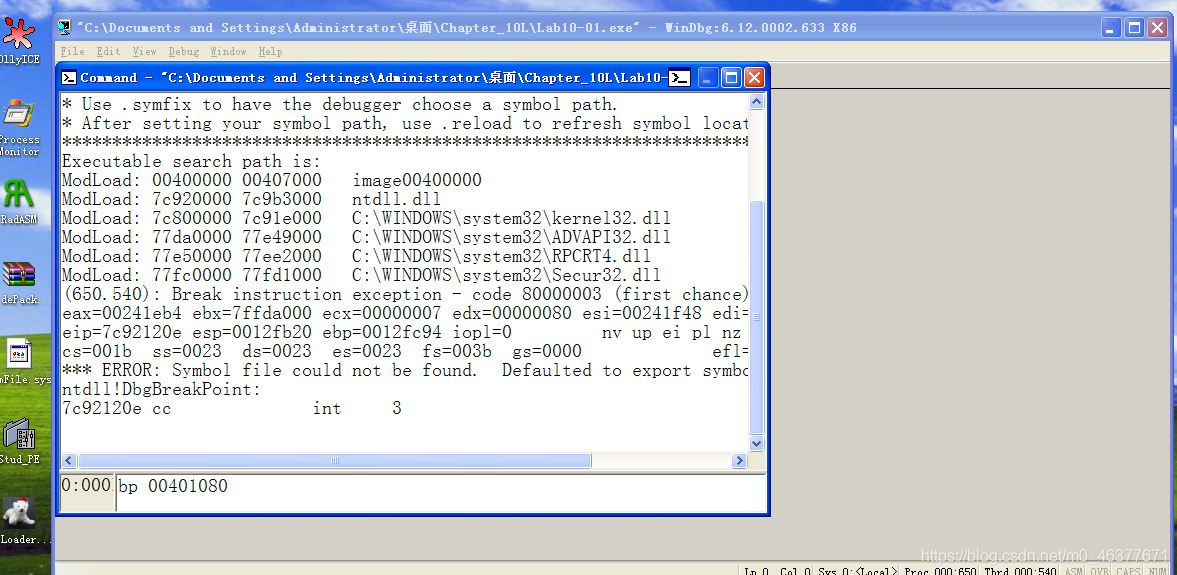
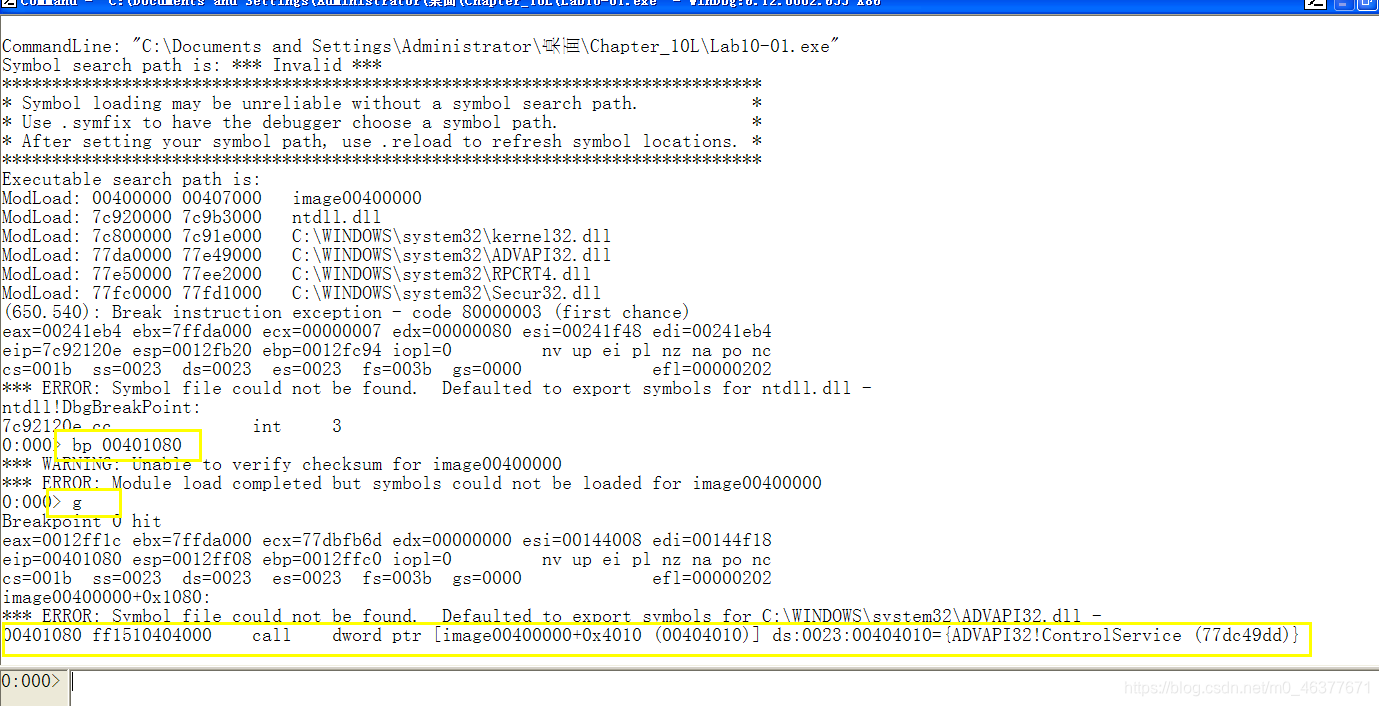
**使用ida对Lab 10-01.sys进行分析，首先查看导入函数。  
  
能够看到有与**[**注册表**](https://so.csdn.net/so/search?q=%E6%B3%A8%E5%86%8C%E8%A1%A8&spm=1001.2101.3001.7020)**相关的敏感操作。其中KeTickCount几乎所有驱动程序都会包括这个函数，可以忽略。  
通过strings工具对其进行分析。  
  
有与防火墙有关注册表相关。**

**反汇编看下：**

****

****

**的确是在修改注册表。  
使用process monitor进行分析。  
  
可以看到有关注册表的更改子键，设置了seed（随机的更改）。  
  
此处的DriverEntry为sub\_10906,进入查看。  
  
查看sub\_10486  
  
能够看到大量对注册表的操作。  
RtlCreaterRegistrykey:通过一个给定的注册表相对路径和值创建指定的键。**[**RtlWriteRegistryValue**](https://docs.microsoft.com/en-us/windows-hardware/drivers/ddi/wdm/nf-wdm-rtlwriteregistryvalue)**:将提供的数据以指定的值名称写入指定的相对路径。参数实际的意义，还需要学习了解。  
  
可以得知通修改注册表键值关闭了防火墙。  
使用**[**windbg**](https://so.csdn.net/so/search?q=windbg&spm=1001.2101.3001.7020)**调试内核。  
我使用的是win7+win xp sp3，符号表使用了win xp3的下载器，因为之前使用的是win xp sp2加上微软的符号表链接，symbols文件只有十几兆，一直报错，于是使用下载器下载，共六百多兆，能够运行了。  
在虚拟机中使用windbg加载lob 10-01.exe  
  
在之前使用ida得到的controlservice地址进行断点，bp 00401080  
**

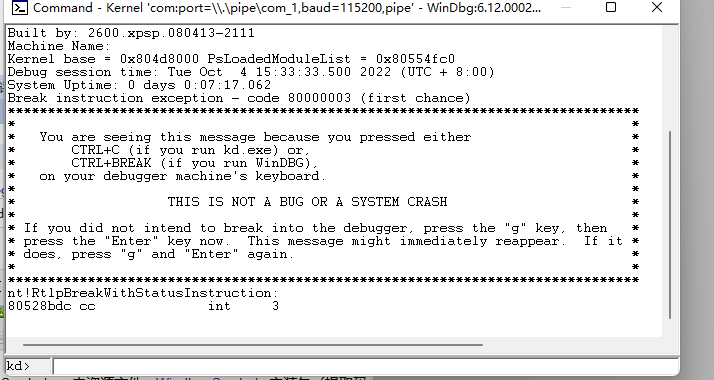
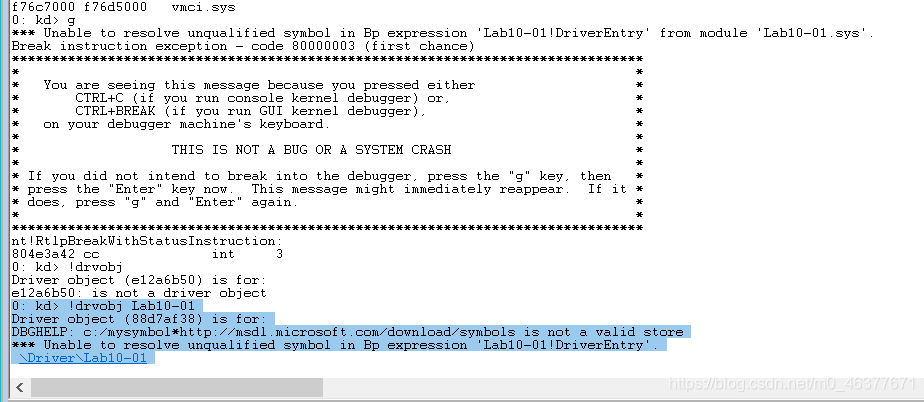
**  
**

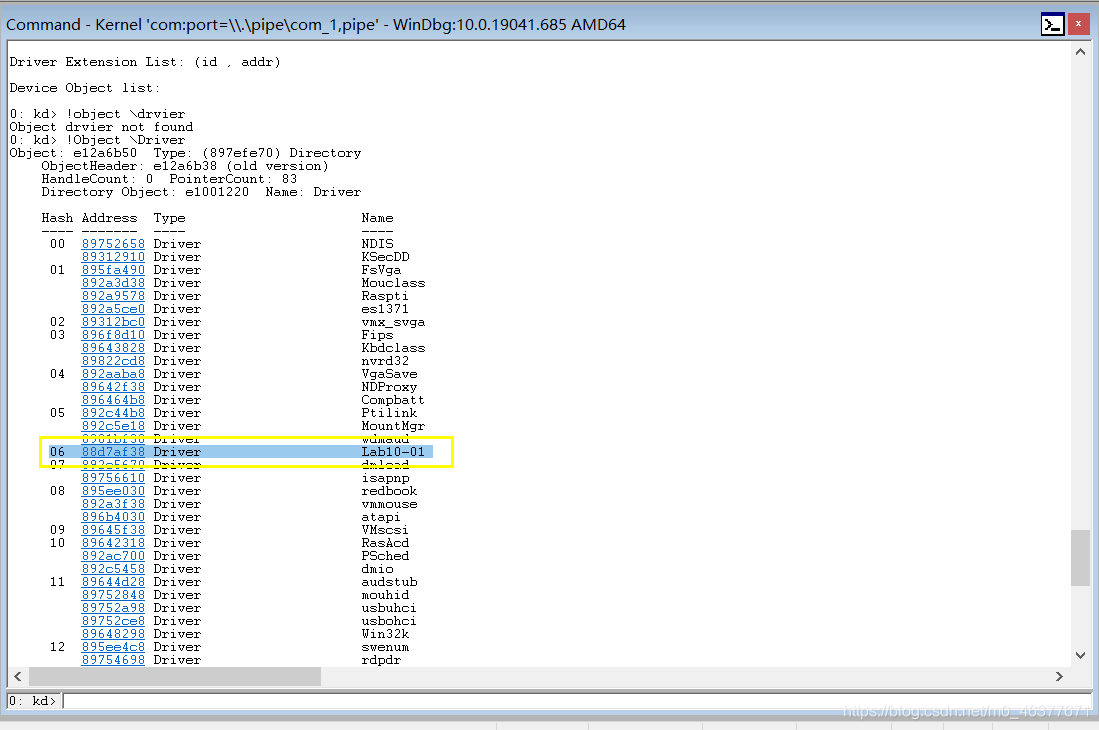
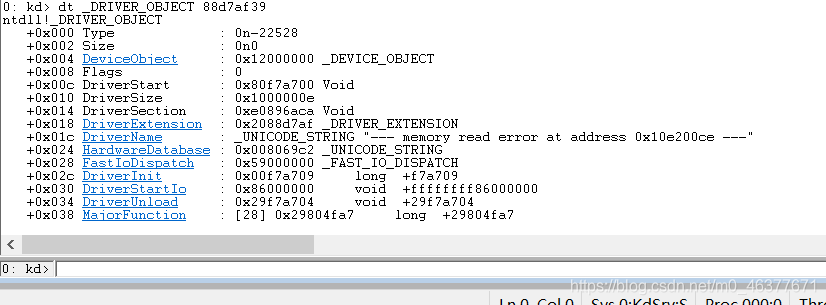
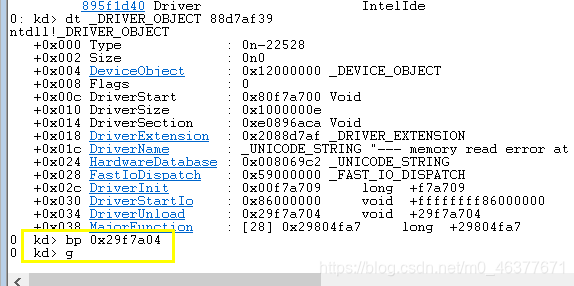
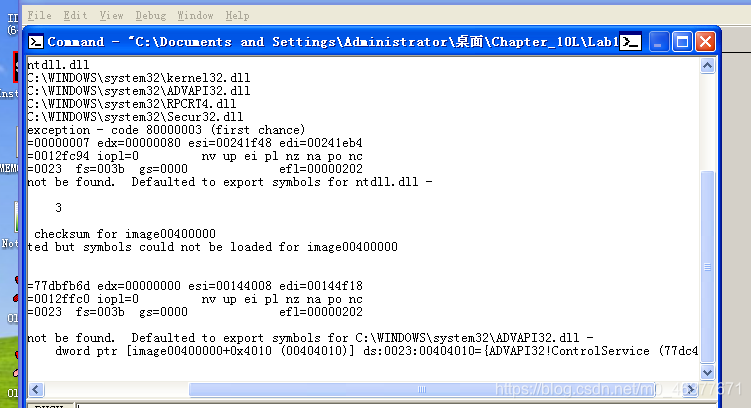
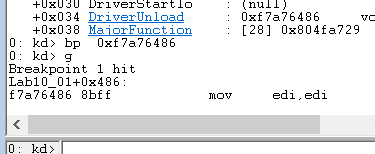
**使用win7宿主机的winbg进行调试。**

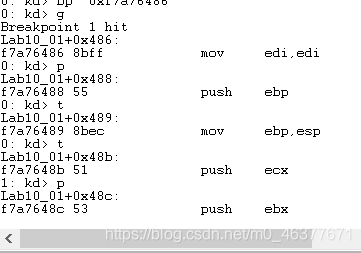
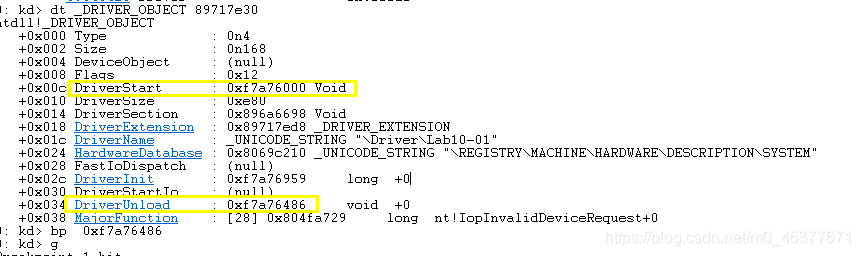
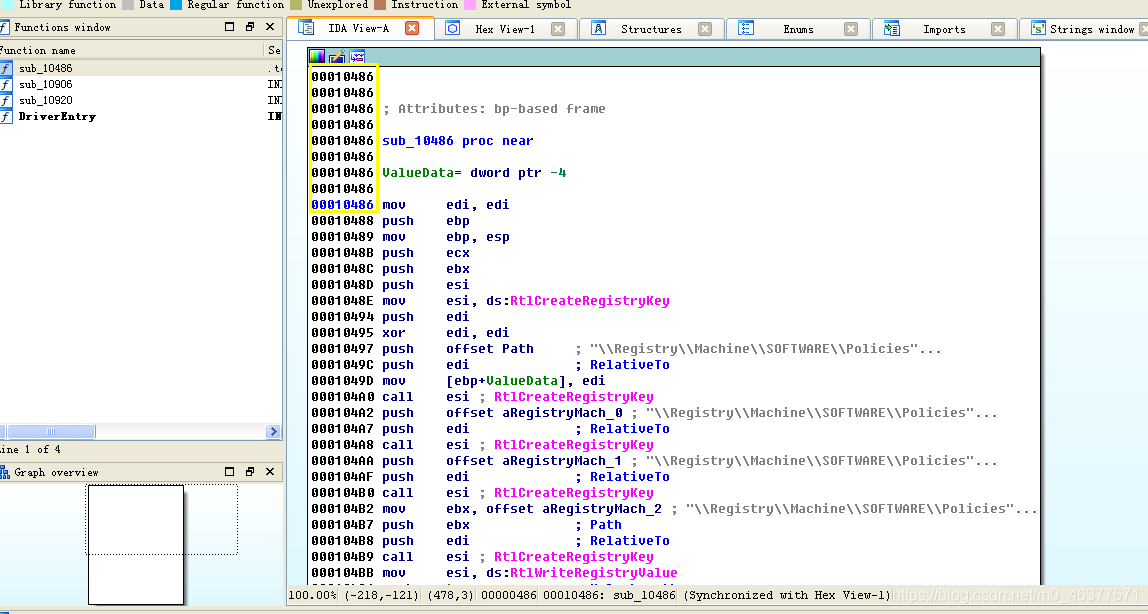
**找到Windbg的文件夹，然后把这个程序创建一个桌面快捷方式，目标修改下：**

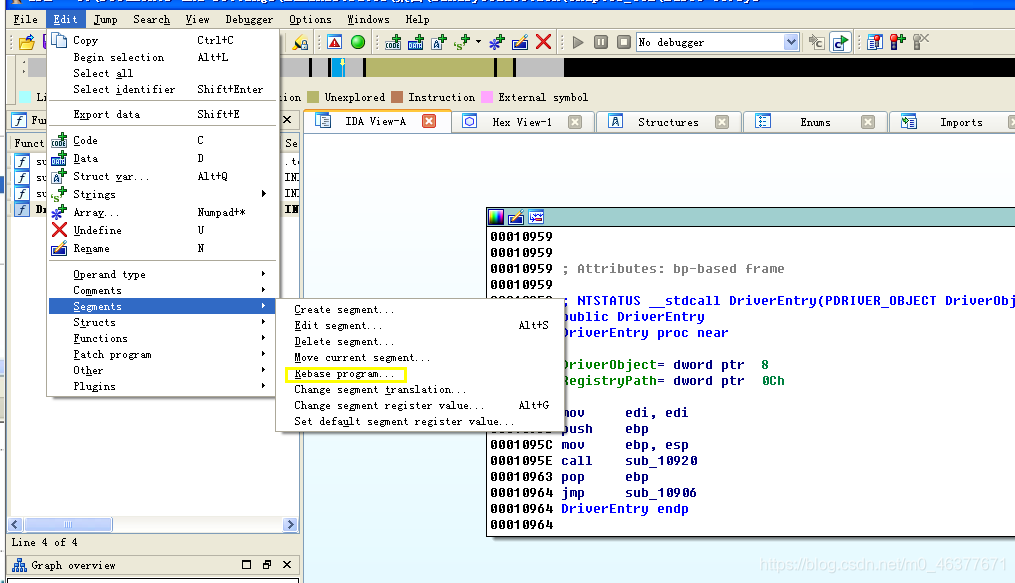
****

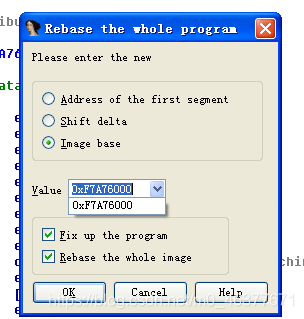
**成功后，然后我们双击windbg快捷方式，从下图看，就说明已经处于调试状态了！**

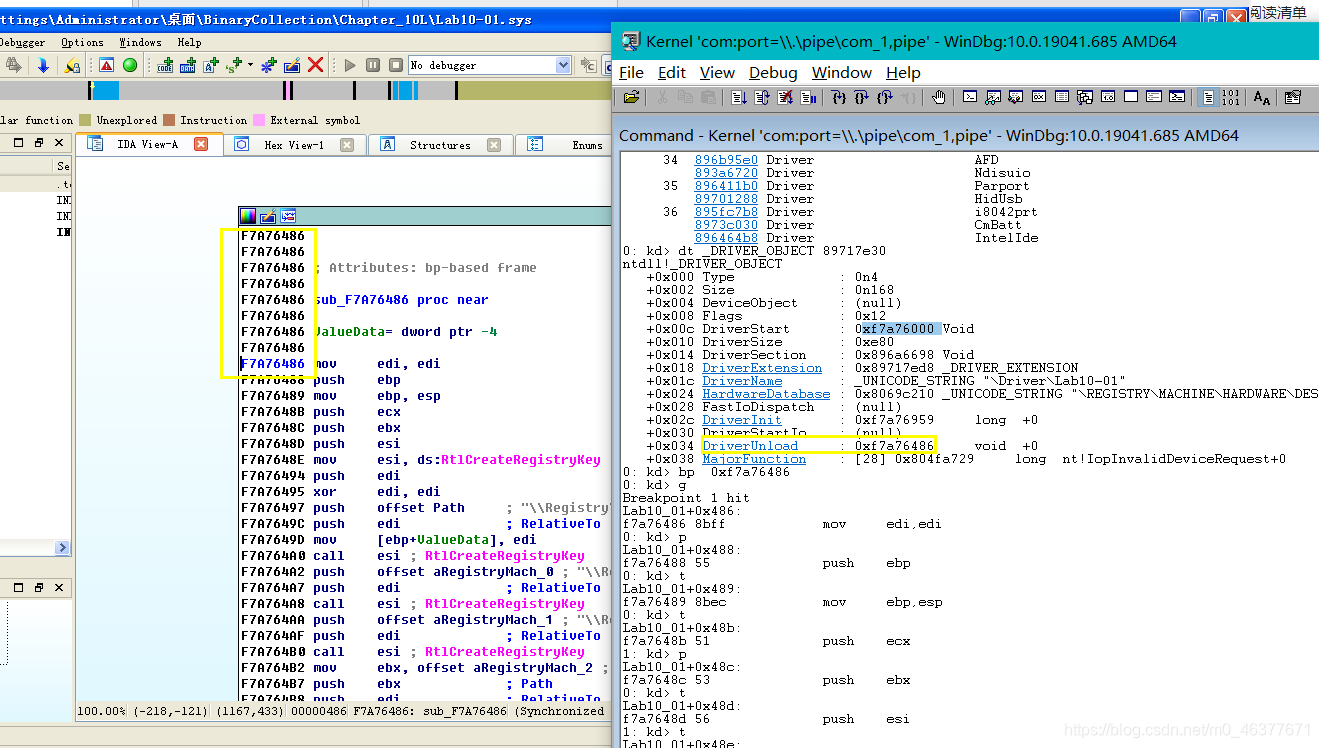
**  
使用**[**!drvobj**](https://docs.microsoft.com/zh-cn/windows-hardware/drivers/debugger/-drvobj)**查看Lab 10-01.exe创造的服务lab 10-01  
  
在这里设备列表为空，这个驱动没有供应用程序访问的设备，也可以使用命令!Object \Driver获得所有的驱动列表。**

**  
使用dt \_DRIVER\_OBJECT 地址 来解析地址的数据结构。  
  
重点观察DriverUnload函数，地址为0x29f7a704，使用bp指令在此加断点,并使用g指令恢复内核的执行。  
  
在虚拟机中继续运行  
  
在win7并运行到断点处  
  
通过按t单步执行下一条指令**

**  
可以使用ida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中driver的默认地址的sys文件是从0x00010000开始的，所以函数卸载代码对应的地址为0x00010468。另外一个方法则是重新设置ida默认的基地址  
**

**  
将基地址修改为driverstart的地址**

****

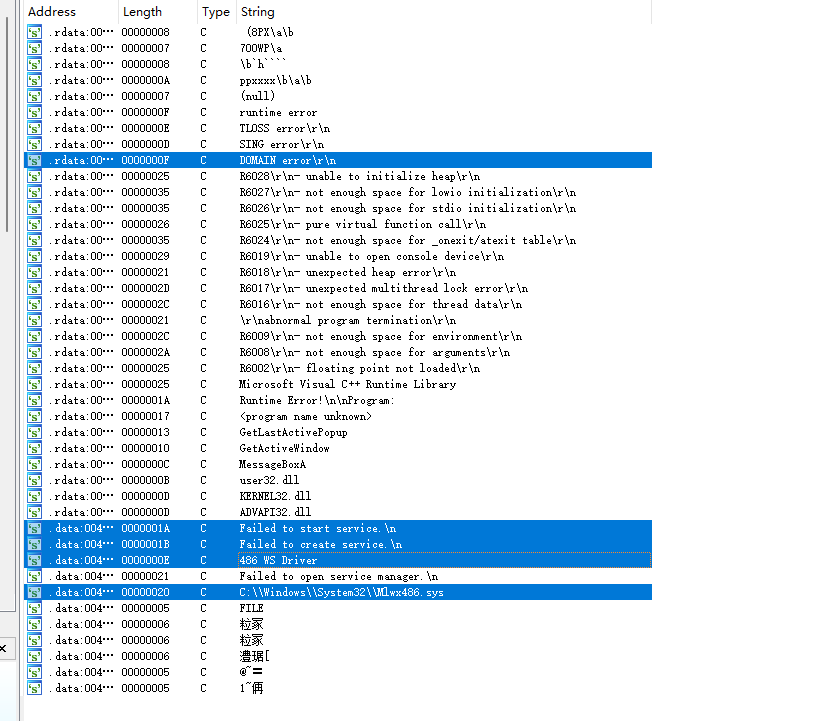
****

1. **Lab 10-2**

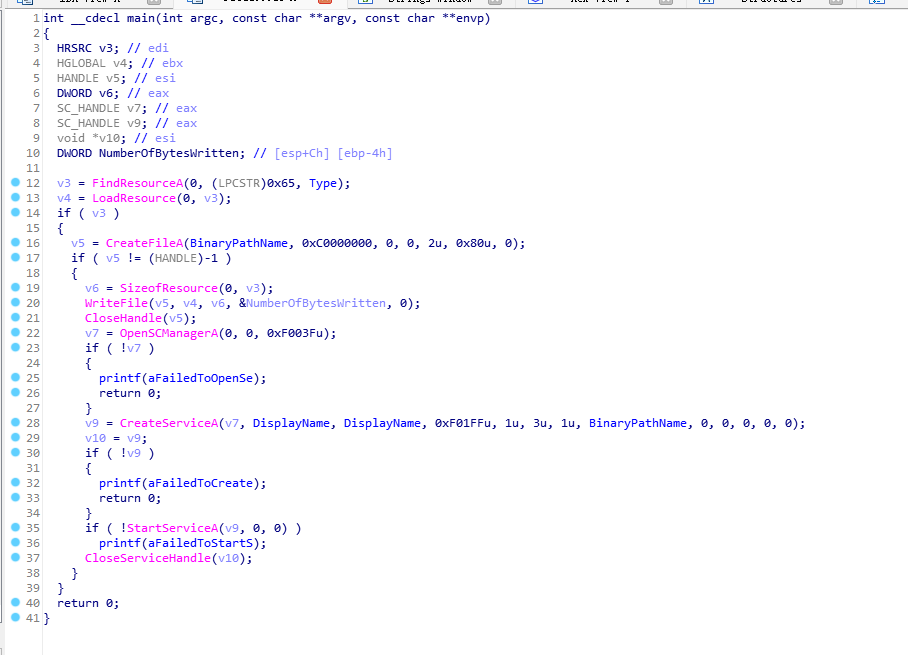
**问题**

**1.这个程序创建文件了吗？它创建了什么文件？**

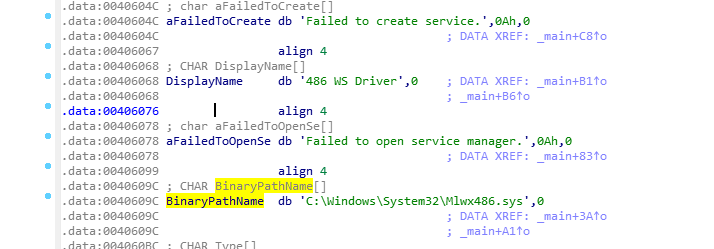
**解答： 我们依旧先从静态分析开始，这里我们在第一个导入DLL里面注意到的有趣的函数是这个WriteFile，说明这个代码会改变这个文件。**

****

**然后反汇编看下：**

****

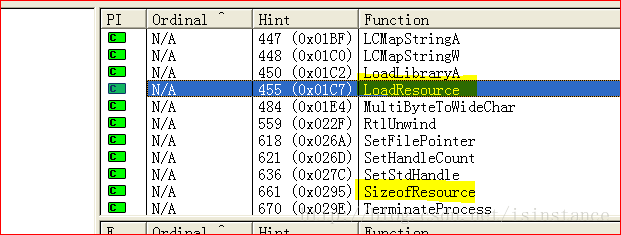
**其中pathname就是sys这个。**

****

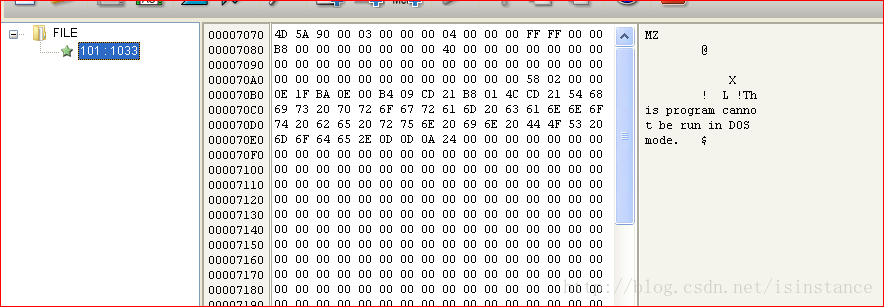
**基本上可以确定是在利用资源文件创建服务，服务是一个sys驱动。**

**查看导入函数， OpenSCManagerA是用来打开服务管理器的函数，StartServiceA是用来启动一个服务的函数，CreateServiceA是创建一个服务的，说明这个代码会在宿主计算机上创建一个服务来运行代码。**

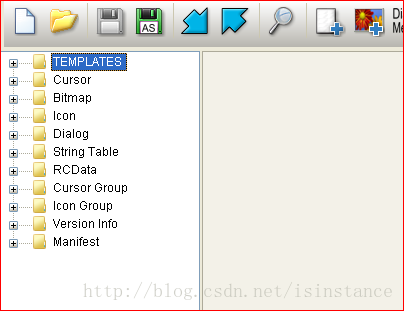
**书中还说了这两个函数LoadResource和SizeOfResource，说明这个代码对Lab10-02.exe的资源节做了一些操作，我们找找这两个函数。**

****

**这两个函数是KERNEL32.DLL的导入函数，不注意看还是难发现的，然后我们知道了这个代码会操作自己的资源节，那我们就去检查一下这个程序的资源节。**

****

**这里发现了一个FILE，里面包含了一个PE头，正常程序的资源节长什么样，如下**

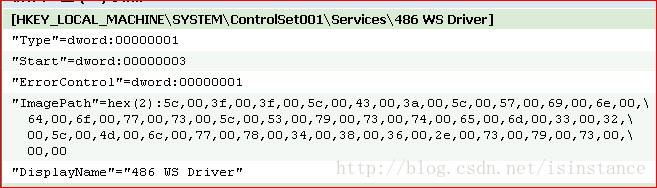
****

**这是ResourceHacker的资源节的样子，对比一下就知道区别在哪里了**

**接下来我们进行基础动态分析，对注册表做快照之后的结果 。**

****

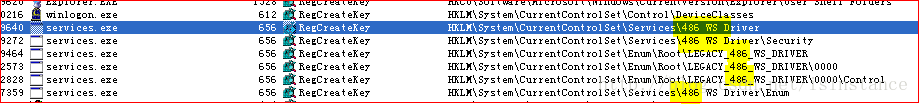
**运行代码之后，增加了6个键，18个值，改变了1个值，增加的有以下**

****

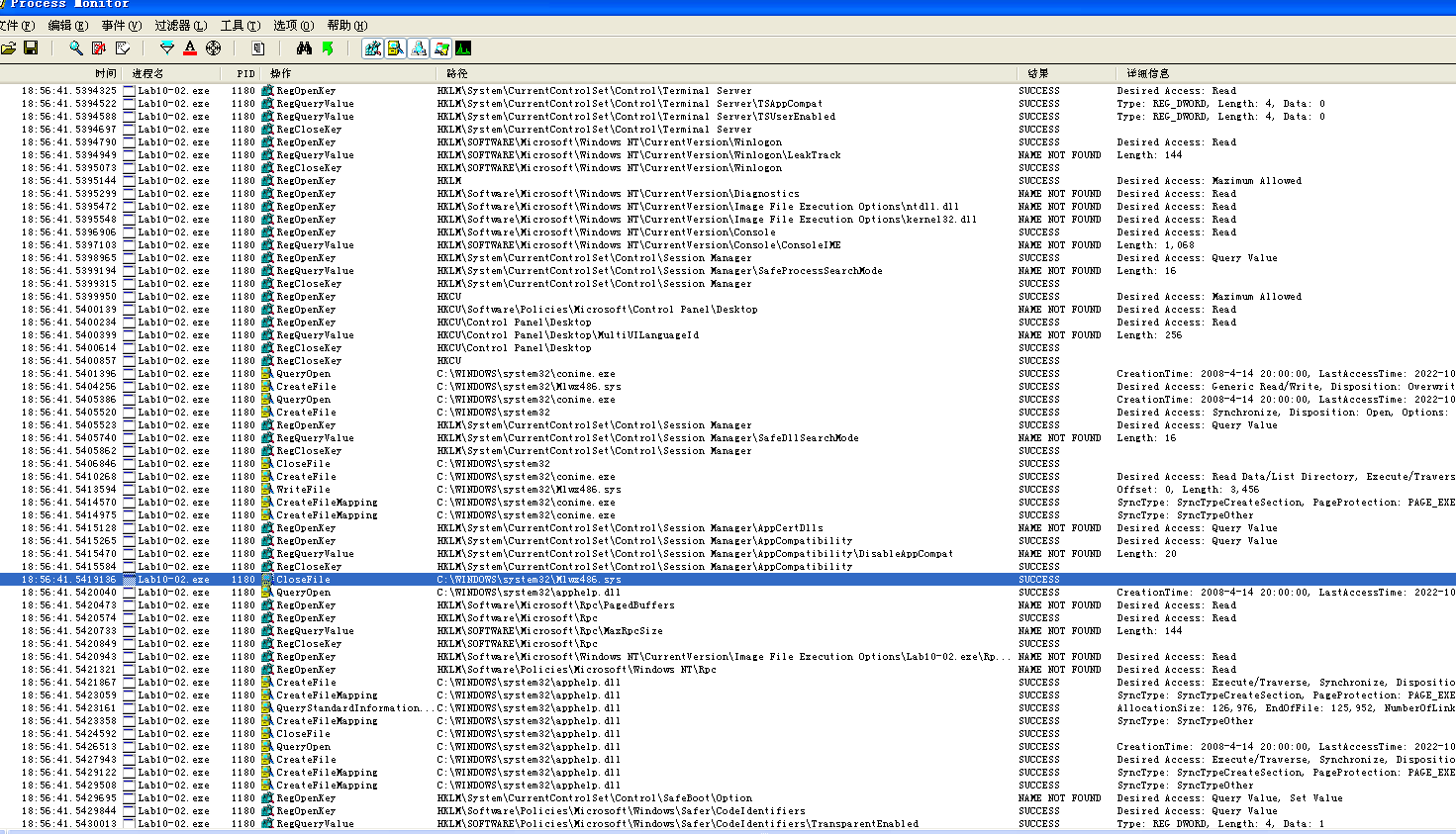
**在服务这里增加了一个叫486 WS Driver的服务，然后下面就是对这个服务的集体细节进行配置**

****

**既然知道了这个代码已经改变了注册表，我们现在追着这个线索来搜索一下。**

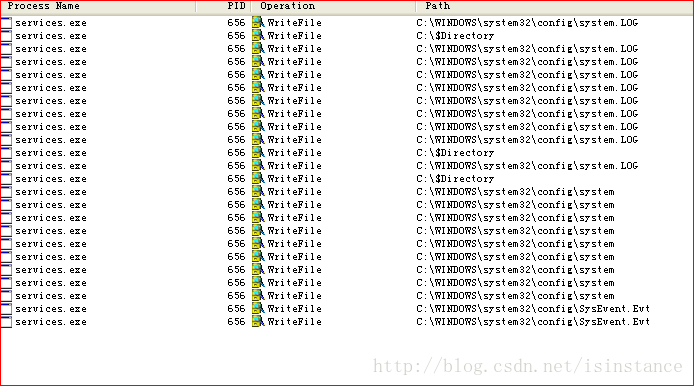
****

**其他事件：**

****

**这里我们发现了一个叫services.exe的代码，执行了RegCreateKey，而且路径也和我们Regshot的结果相同。**

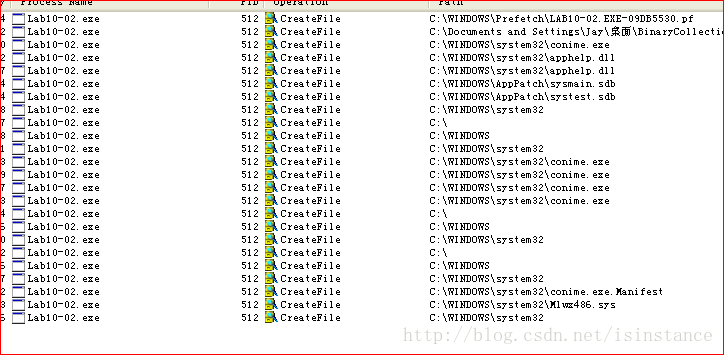
**然后我们缩小搜索范围，搜索这个名叫services.exe的程序做了哪些其他事，记住此时这个程序的PID为656。**

****

**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，然后我们试试查找Lab10-02.exe这个进程名字，恶意行为分析本身就很繁琐**

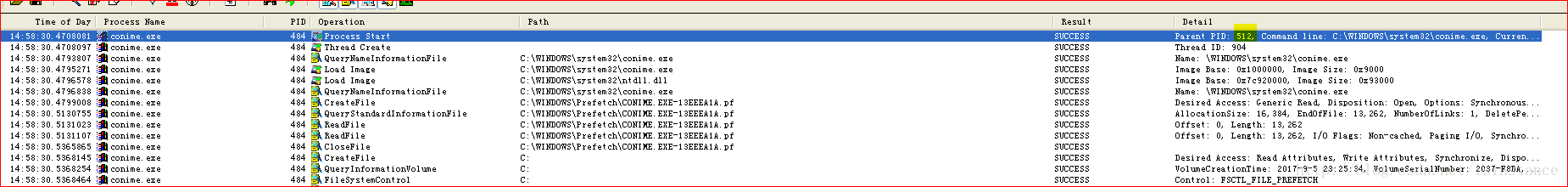
****

**这里我们可以看到Lab10-02.exe这个文件创建了一个文件在C:\WINDOWS\system32\conime.exe，我们继续缩小搜索的范围**

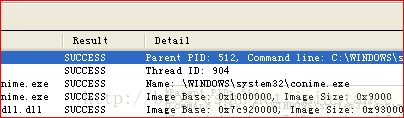
****

**这里文件不仅创建了conime.exe，还有apphelp.dll，sysmain.sdb，systest.sdb，最后当然还有那个sys驱动Mlwx486.sys**

**然后我们搜搜这个conime.exe有没有做过其他操作**

****

**这里我们可以看到这个conime.exe的所有操作，包括这个进程的启动，这里我们注意到这个parents pid，放大一点**

****

**这个512就是Lab10-02.exe的PID**

****

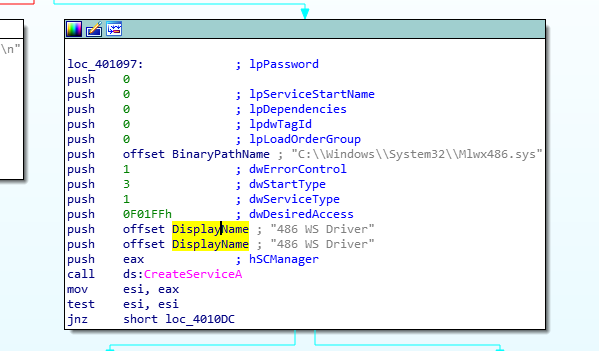
**我们获取一下内核驱动的状态**

**sc query "486 WS Driver"**

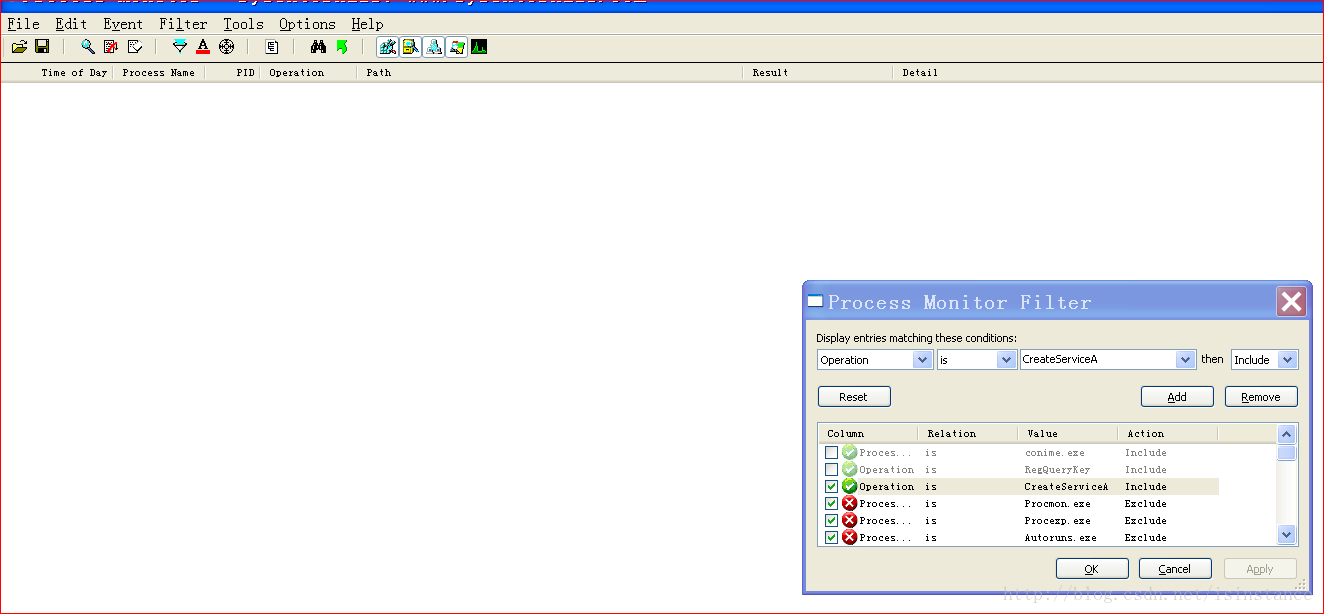
****

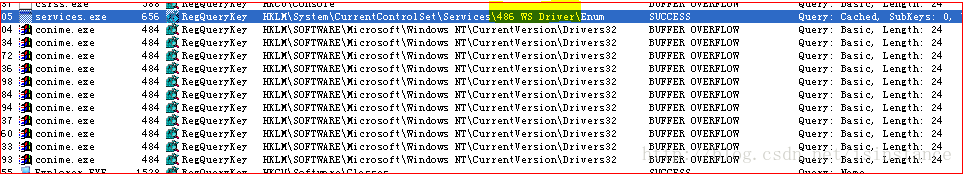
**可以很明显的看出来这个是内核的驱动(KERNEL\_DRIVER)**

**作者是如何知道这个内核驱动的名字是486 WS Driver呢？**

****

**我们可以从注册表中可以找到这个字符串的位置，然后也可以从从CreateServiceA中看出。**

****

****

**然后还发现这个conime.exe有很多注册表修改失败(BUFFER OVERFLOW)的操作，估计这就是为啥这个conime.exe没有从内核删除自己的原因吧**

**言归正传，这里我们就会发现这个486 WS Driver是个内核驱动，然后状态是还在运行(RUNNING)**

**所以这个问题的答案就是创建了conime.exe，还有apphelp.dll，sysmain.sdb，systest.sdb，最后当然还有那个sys驱动Mlwx486.sys**

**2.这个程序有内核组件吗？**

**解答： 现在我们就要连接内核调试器来操作了**

**WinDbg里面运行命令**

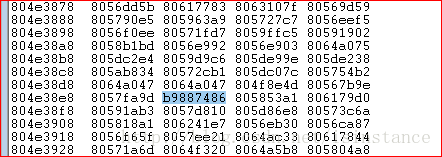
**lm**

**然后仔细找就可以找到这个驱动，这里如果不事先告诉你这个驱动的名字叫Mlwx486还真是难找，不过如果你回想刚刚我们查看创建的文件里面，就有一个Mlwx486.sys**

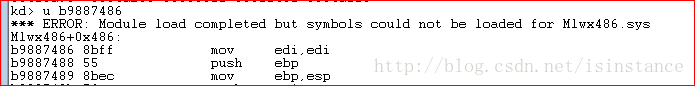
**补充windbg分析,依照课本查看ssdt,发现蹊跷：**

**dd dwo(KeServiceDescriptorTable) L100**

**在这里我们就可以看到这个跳转很多的内存地址b9887486，明显是和其他地址不是一路的！！！**

****

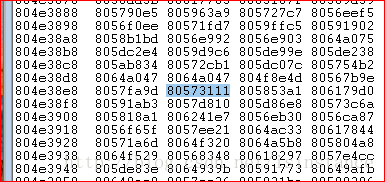
**然后书中下一步是要把虚拟机恢复成Rootkit安装之前的状态来查找这个位置上原来的函数是什么，我们在恢复之前可以看看这个地址b9887486上的是什么函数**

****

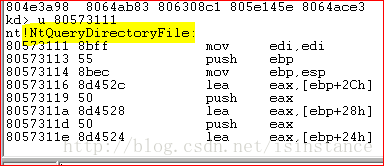
**这个是一个Mlwx486里面自带的函数，然后我们重启，并且恢复虚拟机成未运行病毒状态**

**然后我们运行**

**dd dwo(KeServiceDescriptorTable) L100**

****

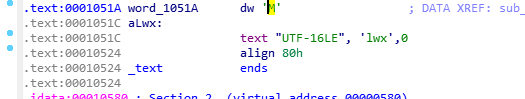
**我们找到这个未被改变之前的值，为80573111这个值，我们下一步查查这个函数是什么**

****

**这个函数原来的位置是nt!NtQueryDirectoryFile，然后接下来我们运行这个病毒，开始继续分析这个病毒，现在我们已经运行了病毒，找到那个函数的位置！为了搞清楚这个函数在做啥，我们导出资源文件里面的PE，IDA反编译：**

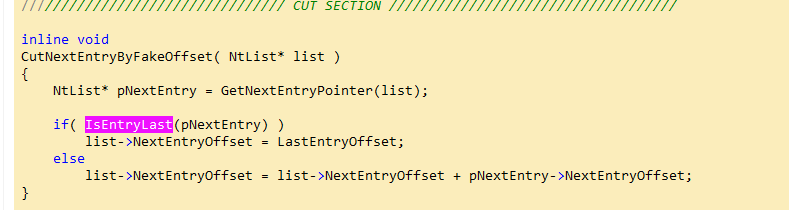
**隐藏文件的部分，应该就是那个if部分的代码了。**

**word\_1051A看看是啥？**

****

**就是Mlwx，看来是匹配文件是Mlws前缀！不就是这个**

**本质上就是链表修改节点而已。**

****

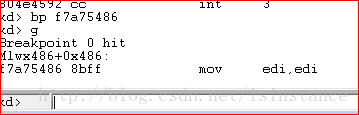
**我们先设置一个断点在f7a75486这里**

**bp f7a75486**

**然后**

**g**

**这时候断点不会马上名字，因为没涉及到查询文件夹的操作，我们回到虚拟机里面，点开我的电脑，断点马上就命中了**

****

**然后开始单步调试，来查明这个函数到底会做什么操作，函数开头的四句都栈初始化的**

**mov edi, edi**

**push ebp**

**mov ebp, esp**

**push esi**

**下面继续**

**mov esi, dword ptr [ebp+1Ch]**

**push edi**

**push dword ptr [ebp+30h]**

**push dword ptr [ebp+2Ch]**

**push dword ptr [ebp+28h]**

**push dword ptr [ebp+24h]**

**push dword ptr [ebp+20h]**

**push esi**

**push dword ptr [ebp+18h]**

**push dword ptr [ebp+14h]**

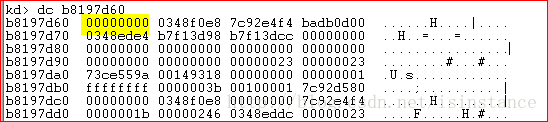
**push dword ptr [ebp+10h]**

**push dword ptr [ebp+0Ch]**

**push dword ptr [ebp+8]**

**call Miwx486+0x514(f7a75514)**

**到这里，函数开始调用函数Miwx486+0x514，为了搞清楚这个入参都是什么东西，我们一个一个的分析和调查这个入参，先计算地址的值，然后查看内存地址上的值是多少，比如[ebp=30h]我们可以查到它的值是0，然后可以得出下面的关系**

****

**这里要注意的是数据结构的存储和识别，比如上面这图的第二段数据为0348f0e8，地址是b8197d64即(b8197d60 + 4)，其实真实是数据在计算机上是这样的(倒着存放的)**

**b8197d64 e8**

**b8197d65 f0**

**b8197d66 48**

**b8197d67 03**

**知道这点就好办了**

**mov esi, dword ptr [ebp+1Ch]**

**push edi = b95a5d64**

**/\* Note: ebp = b95a5d30 \*/**

**push dword ptr [ebp+30h] = [b95a5d60] = 0**

**push dword ptr [ebp+2Ch] = [b95a5d5c] = 80 e1 70**

**push dword ptr [ebp+28h] = [b95a5d58] = 1**

**push dword ptr [ebp+24h] = [b95a5d54] = 3**

**push dword ptr [ebp+20h] = [b95a5d50] = 268**

**push esi = 0070e198**

**push dword ptr [ebp+18h] = [b95a5d48] = 68 e1 70**

**push dword ptr [ebp+14h] = [b95a5d44] = 0**

**push dword ptr [ebp+10h] = [b95a5d40] = 0**

**push dword ptr [ebp+0Ch] = [b95a5d3c] = 0**

**push dword ptr [ebp+8] = [b95a5d38] = 464**

**call Miwx486+0x514(f7ab7514)**

**然后我们根据这个MSDN的文档，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,**

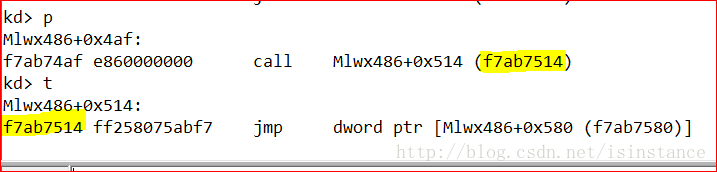
**\_In\_opt\_ PUNICODE\_STRING FileName = 80 e1 70,**

**\_In\_ BOOLEAN RestartScan = 0**

**);**

**这里我们注意这个第八个入参FileInformationClass的值为3，然后我们按t来进入这个函数中**

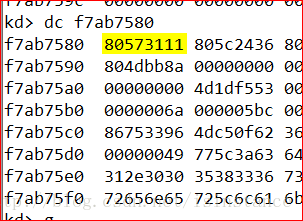
**t**

****

**一进来这个函数的第一个代码就是一个jmp跳转来跳转到其他地方**

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

**注意这里不是跳转到f7ab7580，而是跳转到保存在地址f7ab7580上那个地址。**

****

**所以下一个代码就会跳转来到这里**

****

**然后这里WinDbg已经将这个函数标注为nt!NtQueryDirectoryFile，就是那个被替换的函数的本身。**

**然后运行到这里内核就退出恶意驱动函数的调用了，因为比较ReturnSIngleEntry这里时候，我们实际值是1，代码的期待值是0**

**总结一下这个函数，函数的全部代码如下**

**/\* 栈初始化开始 \*/**

**mov edi, edi**

**push ebp**

**mov ebp, esp**

**push esi**

**/\* 栈初始化结束 \*/**

**mov esi, dword ptr [ebp+1Ch]**

**push edi**

**push dword ptr [ebp+30h] // RestartScan**

**push dword ptr [ebp+2Ch] // FileName**

**push dword ptr [ebp+28h] // ReturnSingleEntry**

**push dword ptr [ebp+24h] // FileInformationClass**

**push dword ptr [ebp+20h] // Length**

**push esi // FileInformation**

**push dword ptr [ebp+18h] // IoStatusBlock**

**push dword ptr [ebp+14h] // ApcContext**

**push dword ptr [ebp+10h] // PacRoutine**

**push dword ptr [ebp+0Ch] // Event**

**push dword ptr [ebp+8] // FileHandle**

**call Mlwx486+0x514 //-> jmp dword ptr [Mlwx486+0x580 (f7ab2580)] -> nt!NtQueryDirectoryFile**

**xor edi, edi**

**cmp dword ptr [ebp+24h], 3 // FileInformationClass = 3**

**mov dword ptr [ebp+30h], eax // RestartScan, 0**

**jne Mlwx486+0x505 // if [ebp+24h] != 3 -> jmp and ret 2Ch**

**test eax, eax // eax is NtQueryDirectoryFile return value(success return 0)**

**jl Mlwx486+0x505 // if eax < 0 -> jmp and ret 2Ch**

**cmp byte ptr [ebp+28h], 0 // ReturnSingleEntry = 1**

**jne Mlwx486+0x505 // if [ebp+28h] != 0 -> jmp and ret 2Ch**

**push ebx //-> p(f7ab2486)**

**push 8 // function Mlwx486+0x4ca here**

**push offset Mlwx486+0x51a //-> 'Mlwx'**

**lea eax, [esi+5Eh]**

**push eax**

**xor bl, bl**

**call dword ptr [Mlwx486+0x590] // standard windows nt function RtlCompareMemory**

**cmp eax, 8 // eax = 0**

**jne Mlwx486+0x4f4**

**|\_ mov eax , dword ptr [esi] // [esi] = 0**

**test eax, eax**

**je Mlwx486+0x504 // if eax == 0 -> jmp and ret 2Ch**

**test bl, bl // bl always equal 0**

**jne Mlwx486+0x500 // if bl != 0 -> jmp back to 'push 8'**

**mov edi, esi**

**add esi, eax**

**jmp Mlwx486+0x4ca // jmp back to 'push 8'**

**pop ebx**

**mov eax, dword ptr [ebp+30h]**

**pop edi**

**pop esi**

**pop ebp**

**ret 2Ch**

**inc bl**

**test edi, edi**

**je Mlwx486+0x4f4**

**mov eax, dword ptr [esi]**

**test eax, eax**

**jne Mlwx486+0x4f2**

**and dword ptr [edi], eax**

**jmp Mlwx486+0x4f4**

**add dword ptr [edi], eax**

**mov eax, dword ptr [esi]**

**test eax, eax**

**je Mlwx486+0x504**

**test bl, bl**

**jne Mlwx486+0x500**

**mov edi, esi**

**add esi, eax**

**jmp Mlwx486+0x4ca**

**pop ebx**

**mov eax dword ptr [ebp+30h]**

**pop edi**

**pop esi**

**pop ebp**

**ret 2Ch**

**我们刚刚跳转结束那里吗可以设置一个条件断点，当returnSingleEntry为0时候，才会中断，然后我们看看这个断点怎么设置：**

**bp f7ab2486 ".if dwo(esp+0x24)==0 {} .else {gc}"**

**这里的f7ab2486为RootKit替换的那个SSDT地址，每次运行都不会相同**

**然后这里我们用dir命令去查看C:\WINDOWS\system32\。**

**这里为了在ReturnSingleEntry=0时候中断，而ReturnSingleEntry的值应该是[ebp+0x28]，所以我们这里一般会觉得这个条件中断的语句应该这样写**

**bp f7ab2486 ".if dwo(ebp+0x28)==0 {} .else {gc}"**

**但是，书上什么写的[esp+24h]，为什么是esp+24h，这里我们着重分析一下**

**首先我们必须要明白，函数在被调用之后，第一步要做的操作就是保存调用着的堆栈信息，就是所谓的函数初始化堆栈，初始化的过程如下(代码截取于上面恶意驱动)**

**push ebp**

**mov ebp, esp**

**push esi**

**由于我们的断点是设置在外面的大循环，在中断的时候，并未执行栈初始化的过程，现在我们设准备调用函数的旧函数里面的ebp的值为ebp1，esp的值为esp1，执行初始化之后的调用函数l里的ebp值为ebp2，esp值为esp2**

**由此我们可得如下关系**

**ebp2 = esp1 - 0x4**

**esp2 = esp1 - 0x8**

**已知我们在被调用函数里面的ReturnSingleEntry的值为[ebp+28h]**

**也就是ebp2+28h = esp1-4h+28h=esp1+24h**

**因为我们断点是在函数调用之前会被命中的，所以我们这里的断点要设置为esp+24h=0**

**讲了这么多，为什么是esp不是ebp就解释到这里**

**然后我们输入上面这个语句来设置条件中断，然后我们用dir命令来列出C:\WINDOWS\system32这个文件夹下面的所有文件和文件夹**

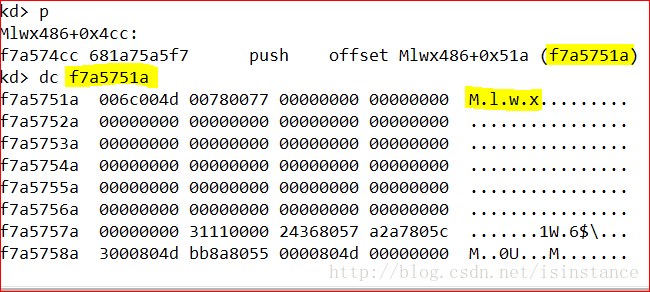
**dir C:\WINDOWS\system32**

**然后就会发现，我们的条件断点被命中了，因为这里我们已经把这个恶意驱动的所有代码都列在了上面，所以这里我们就只列出必要的代码来进行分析**

**现在我们注意到以下这些代码，存在一个字符串**

**push offset Mlwa486+0x51a**

**这里压栈的这个值，我们可以查到这个值是Mlwx**

****

**然后这段函数是这样的**

**push ebx // ebx is Mlwx486 function start address**

**push 8**

**push offset Mlwx486+0x51a // Mlwx**

**lea eax, [esi+5Eh]**

**push eax**

**xor bl, bl**

**call dword ptr [Mlwa486+0x590] // RtlCompareMemory**

**然后引用MSDN的定义**

**SIZE\_T RtlCompareMemory(**

**\_In\_ const VOID \*Source1,**

**\_In\_ const VOID \*Source2,**

**\_In\_ SIZE\_T Length**

**);**

**由图中可知道，eax要和Mlwx这个字符串进行比较，然后这个比较的最大长度为8，这里我在刚刚的调用Mlwx486+0x514时，就分别标注过各个参数在MSDN中的意义和名称，其中esi的值被标注为FileInformation，而且这个值是为3**

**这里我们就可以确定这个FileInformation的具体意义就是FileBothDirectoryInformation，然后这个值返回的是一个FILE\_BOTH\_DIR\_INFORMATION的结构**

**我们在header里面找寻这个变量，但是不知道Windows这样的闭源操作系统会不会开放他的header出来，所以这里是根据书上和各种道听途说的搜索知道了这个对应的是FileBothDirectoryInformation，但是如果下次变成了2的话就不知道怎么对应。。。**

**我们可以观察一下这个结构的定义**

**typedef struct \_FILE\_BOTH\_DIR\_INFORMATION {**

**ULONG NextEntryOffset;**

**ULONG FileIndex;**

**LARGE\_INTEGER CreationTime;**

**LARGE\_INTEGER LastAccessTime;**

**LARGE\_INTEGER LastWriteTime;**

**LARGE\_INTEGER ChangeTime;**

**LARGE\_INTEGER EndOfFile;**

**LARGE\_INTEGER AllocationSize;**

**ULONG FileAttributes;**

**ULONG FileNameLength;**

**ULONG EaSize;**

**CCHAR ShortNameLength;**

**WCHAR ShortName[12];**

**WCHAR FileName[1];**

**} FILE\_BOTH\_DIR\_INFORMATION, \*PFILE\_BOTH\_DIR\_INFORMATION;**

**现在我们确定了这个结构体是FILE\_BOTH\_DIR\_INFORMATION，然后这个结构体的定义就是上面这个定义的**

**所以根据我们上面分析的，esi是FileInformation这个东西，然后这个东西是函数返回的一个结构体，这个结构体现在确定是FILE\_BOTH\_DIR\_INFORMATION**

**所以在代码**

**lea eax,[esi+5Eh]**

**此处，我们可以找到这个esi+5Eh这个地方为WCHAR FileName[1]这个地方，分析过程如下**

**首先我们确定各个数据类型所占的字节数，因为我们这个运行的虚拟机是32位的，所以可以得出如下结论**

**ULONG = 8 byte**

**LARGE\_INTEGER = 8 byte**

**CCHAR = 1 byte**

**WCHAR = 2 byte**

**这里我们要记住这个定理**

**原则1：数据成员对齐规则：结构（struct或联合union）的数据成员，第一个数据成员放在offset为0的地方，以后每个数据成员存储的起始位置要从该成员大小的整数倍开始（比如int在32位机为４字节，则要从4的整数倍地址开始存储）。**

**原则2：结构体作为成员：如果一个结构里有某些结构体成员，则结构体成员要从其内部最大元素大小的整数倍地址开始存储。（struct a里存有struct b，b里有char，int，double等元素，那b应该从8的整数倍开始存储。）**

**原则3：收尾工作：结构体的总大小，也就是sizeof的结果，必须是其内部最大成员的整数倍，不足的要补齐。**

**重点是第一点和第三点，结构会存在对齐的特性。**

**然后我们看，根据结构体定义，可以得出下面这样地址递增列表：**

**typedef struct \_FILE\_BOTH\_DIR\_INFORMATION { START - END**

**ULONG NextEntryOffset; 8 byte(32 bit) addr = esi+0x00 - esi+0x07**

**ULONG FileIndex; 8 byte(32 bit) esi+0x08 - esi+0x0f**

**LARGE\_INTEGER CreationTime; 8 byte(64 bit) esi+0x10 - esi+0x17**

**LARGE\_INTEGER LastAccessTime; 8 byte(64 bit) 0x18 - 0x1f**

**LARGE\_INTEGER LastWriteTime; 8 byte(64 bit) 0x20 - 0x27**

**LARGE\_INTEGER ChangeTime; 8 byte(64 bit) 0x28 - 0x2f**

**LARGE\_INTEGER EndOfFile; 8 byte(64 bit) 0x30 - 0x37**

**LARGE\_INTEGER AllocationSize; 8 byte(64 bit) 0x38 - 0x3f**

**ULONG FileAttributes; 8 byte(64 bit) 0x40 - 0x47**

**ULONG FileNameLength; 8 byte(64 bit) 0x48 - 0x4f**

**ULONG EaSize; 8 byte(64 bit) 0x50 - 0x57**

**CCHAR ShortNameLength; 1 byte(8 bit) (4 byte) 0x58 - 0x5b**

**WCHAR ShortName[12]; 2 byte(16 bit) 2\*12=0x18 0x5c - 0x5d**

**WCHAR FileName[1]; 2 byte(16 bit) 0x5e - 0x5f**

**} FILE\_BOTH\_DIR\_INFORMATION, \*PFILE\_BOTH\_DIR\_INFORMATION;**

**然后我们再把5Eh比较一下，就可以得出这个esi+5eh就是FileName这个结构元素的起始地址，然后我们继续往下**

**push 8 // length**

**push offset Mlwx486+0x51a // Mlwx**

**lea eax, [esi+5Eh]**

**push eax // filename**

**xor bl, bl**

**call dword ptr [Mlwx486+0x590] // RtlCompareMemory**

**到这时，就可以确定，这个函数的入参的具体值是什么**

**SIZE\_T RtlCompareMemory(**

**\_In\_ const VOID \*Source1,**

**\_In\_ const VOID \*Source2,**

**\_In\_ SIZE\_T Length**

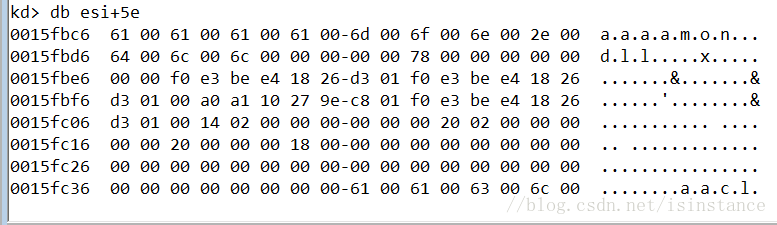
**);**

**再根据MSDN关于RtlCompareMemory的定义，我们可以知道，这个函数在比较Mlwx和dir列出来的各个filename，这个函数的意义在于，比较的是第一个入参和第二个入参，如果想的就返回了Length的值，如果不同就返回相同的字节数。**

**RtlCompareMemory returns the number of bytes in the two blocks that match. If all bytes match up to the specified Length value, the Length value is returned.**

**我们现在就去看看这个地址上的FileName具体是什么值了，使用db来查看**

**db esi+5eh**

****

**这里可以看出，FileName参数是在C:\WINDOWS\system32下的各个文件的名字，这里我们抓到的是aaaamon.dll，结果可能会不同，但是不影响我们的继续分析**

**这里的操作是将C:\WINDOWS\system32下的各个文件名和Mlwx比较，比较完之后会执行下面的操作**

**cmp eax, 8**

**jne Mlwx486+0x4f4**

**这里比较返回值和8的大小，一般来说，这个返回值是不会等于8的，除非你遇到Mlwx486.sys**

**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和Mlwx不相等，函数直接就退出**

**然后我们继续往下**

**现在我们分析它是如何修改NtQueryDirectoryFile的返回值然后隐藏Mlwx486.sys文件的，我们可以查一下NtQueryDirectoryFile的文档，然后就可以知道，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，如下图**

**---------------------------**

**| 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，这样依次下去**

**知道这些我们下面就可以来分析接下来的代码了，如果我们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**

**这个函数的大致操作就是如上所示，注意那出现了两次的那个指令你就明白它把指针往后移动了以为，抹除了Mlwx486.sys文件的FILE\_BOTH\_DIR\_INFORMATION结构，之后就达到了隐藏文件的目的。**

**mov eax, dword ptr[esi]**

**第二问的答案就是这个程序拥有一个内核模块，存储在程序的资源节上，执行的时候释放sys文件，然后这个sys文件就会加载到内核中执行**

**3.这个程序做了些什么？**

**解答：通过上面的分析，可以得出，这是用来隐藏文件的RootKit，它使用SSDT来挂钩覆盖NtQueryDirectoryFile函数，通过自定义一些操作，来隐藏文件。**

**我们可以把被隐藏的sys文件导出来看看，书中给我们提供了三种方法来导出这个被隐藏的文件：**

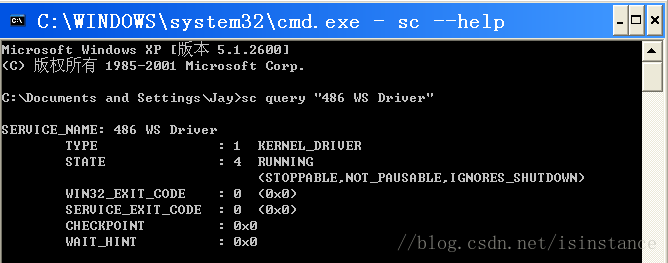
**1. 禁用驱动的服务。**

**2. 从安装的资源节提取出这个文件。**

**3. 访问文件的目录，用cp命令将文件重命名后显示。**

**我们这里先试试第一种，也是推荐的方法，这里需要重启。**

**我们先用cmd来查询这个服务在运行了没有。**

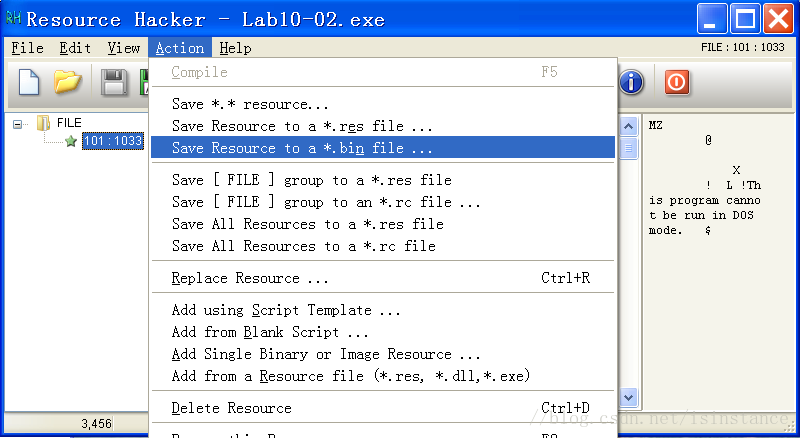
****

**然后我们输入命令。**

**sc stop "486 WS Driver"**

****

**服务无法被控制，那么没办法，再试试第二个。**

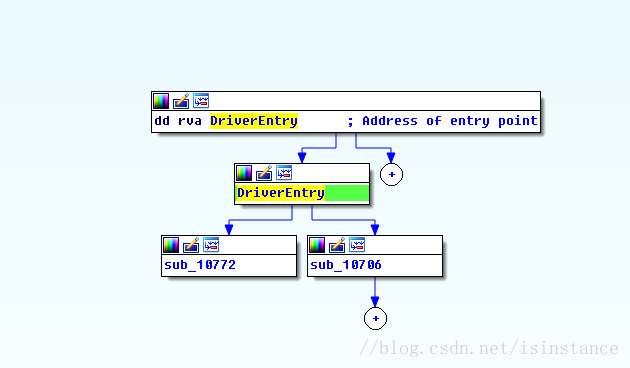
****

**点这个然后保存到桌面上，用IDA来打开就行了。**

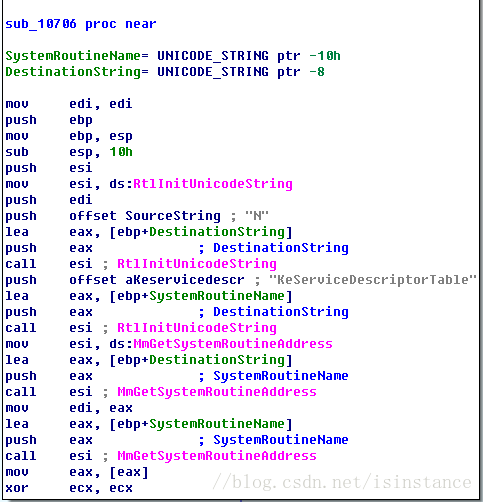
**我们试试第三种方法。**

****

**成功了，然后我们打开看看，这就是这个文件打开的样子。**

****

**我们进入DriverEntry这个例程。**

****

**这里就不详细分析这个代码了，书上说是RtlInitUnicodeString以参数KeServiceDescirptorTable和NtQueryDircetoryFile做入参，然后用MmGetSystemRoutineAddress这个函数来查找这个两个地址的偏移量，接下来他把地址做了一个替换。**

1. **Yara规则**

**根据以上分析结果，得到Yara规则如下。**

rule lab0701

{

strings:

$string1 = "Malservice"

$string2 = "[**http://www.malwareanalysisbook.com**](http://www.malwareanalysisbook.com)"

$string3 = " Internet Explorer 8.0"

condition:

filesize < 100KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and all of them

}

rule lab0702

{

strings:

$string1 = "http://www.practicalmalwareanalysis.com/cc.html"

$string2 = "CoCreateInstance"

condition:

filesize < 100KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and all of them

}

rule lab0703

{

strings:

$string1 = "127.26.15.13"

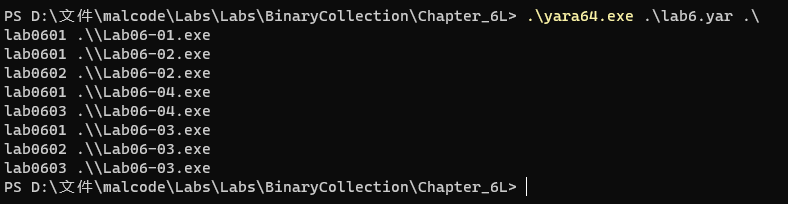
$string2 = "WARNING\_THIS\_WILL\_DESTORY\_YOUR\_MACHINE"

$string3 = "Lab07-03.dll"

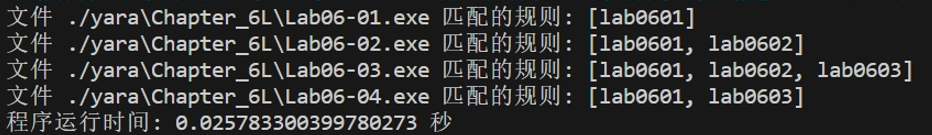
condition:

filesize < 100KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and all of them

**运行结果如下，可看到成功匹配所有文件。**



**在存放所有样例的文件夹下测试其检测效率，结果如下。**



1. **IDA Python脚本编写**

**遍历所有函数，排除库函数或简单跳转函数，当反汇编的助记符为call或者jmp且操作数为寄存器类型时，输出该行反汇编指令。**

import idautils

for func in idautils.Functions():

flags = idc.GetFunctionFlags(func)

if flags & FUNC\_LIB or flags & FUNC\_THUNK:

continue

dism\_addr = list(idautils.FuncItems(func))

for line in dism\_addr:

m = idc.GetMnem(line)

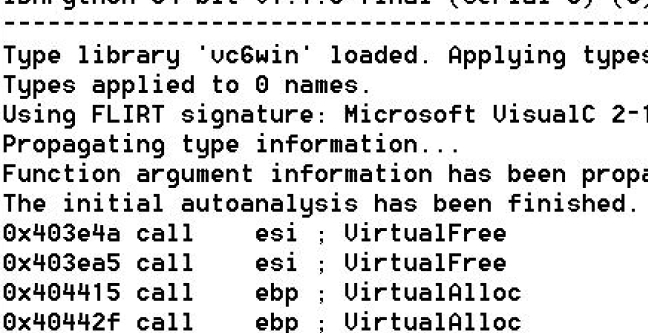
if m == 'call' or m == 'jmp':

op = idc.GetOpType(line,0)

if op == o\_reg:

print '0x%x %s' % (line,idc.GetDisasm(line))

**结果如下**

****

1. **R77验证实验**

**r77-Rootkit是一款功能强大的无文件Ring 3 Rootkit，并且带有完整的安全工具和持久化机制，可以实现进程、文件和网络连接等操作及任务的隐藏。**

**r77能够在所有进程中隐藏下列实体：**

**文件、目录、连接、命名管道、计划任务；**

**进程；**

**CPU用量；**

**注册表键&值；**

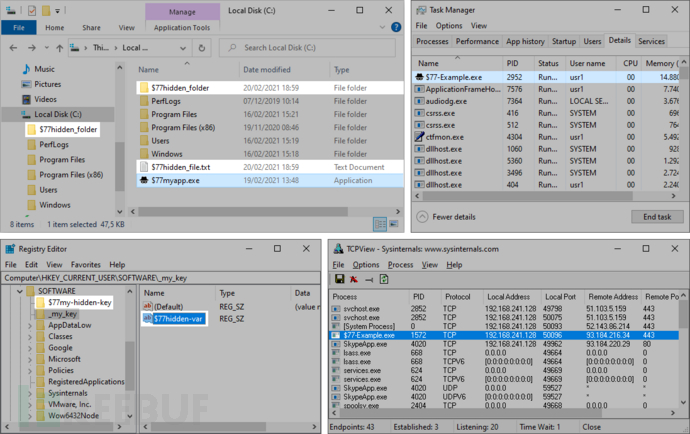
**服务；**

**TCP&UDP连接；**

**该工具兼容32位和64位版本的Windows 7以及Windows 10。**

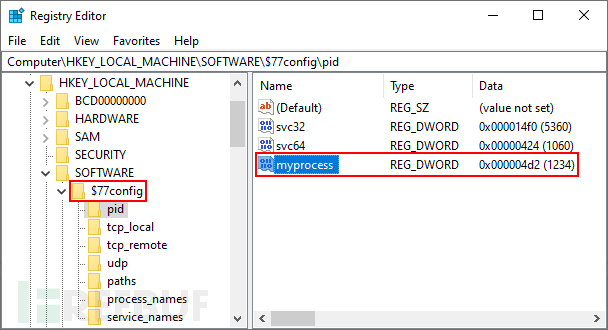
**通过前缀隐藏**

**所有以“$77”为前缀命名的实体都将被隐藏：**

****

**配置系统**

**动态配置系统允许广大研究人员通过PID或抿成来隐藏进程，通过完整路径来隐藏文件系统，或通过指定端口隐藏TCP&UDP连接：**

****

**配置信息存储在“HKEY\_LOCAL\_MACHINE\SOFTWARE\$77config”中，并且可以在未提权状态下由任何进程写入。这个键的DACL被设置为可以给任意用户授予完整访问权。**

**“$77config”键在注册表编辑器被注入了Rootkit之后会自动隐藏。**

**安装工具**

**r77可以直接使用单独的“Install.exe”进行安装，安装工具会将r77服务在用户登录之前开启，后台进程会向所有当前正在运行以及后续生成的进程中注入命令。这里需要使用两个进程来分别注入32位和64位进程，这两个进程都可以使用配置系统和PID来进行隐藏。**

**“Uninstall.exe”程序负责将r77从系统中卸载掉，并解除Rootkit跟所有进程的绑定关系。**

**无文件持久化**

**Rootkit将驻留在系统内存中，不会将任何文件写入磁，这种机制是分多个阶段实现的。**

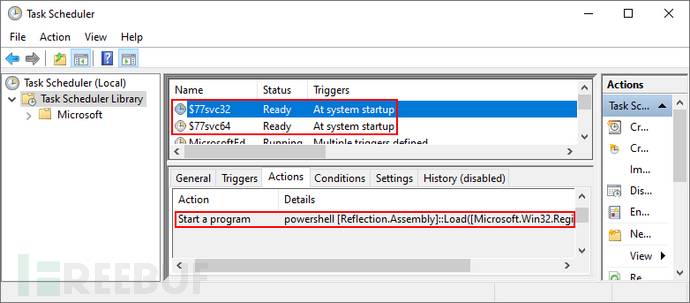
**阶段一**

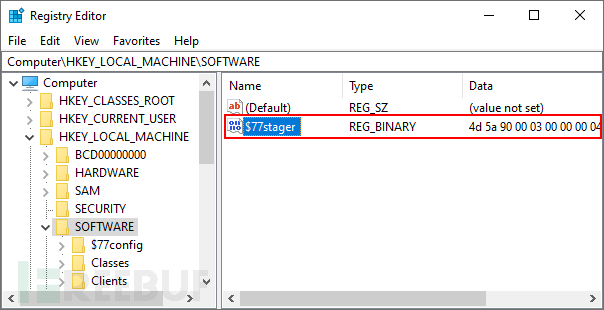
**安装程序为32位和64位r77服务创建两个计划任务。计划任务确实需要存储名为$77svc32.job和$77svc64.job的文件，这是无文件概念的唯一例外。但是，一旦Rootkit运行，计划任务也会通过前缀隐藏。**

**计划任务将使用下列命令开启“powershell.exe”：**

**[Reflection.Assembly]::Load([Microsoft.Win32.Registry]::LocalMachine.OpenSubkey('SOFTWARE').GetValue('$77stager')).EntryPoint.Invoke($Null,$Null)**

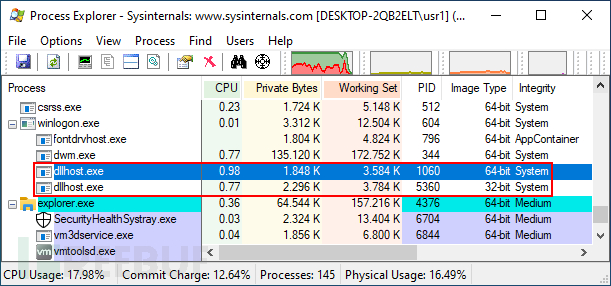
**该命令是内联的，不需要.ps1脚本。这里，使用PowerShell的.NET Framework功能从注册表加载C#可执行文件并在内存中执行。由于命令行的最大长度为260（MAX\_PATH），因此只有足够的空间执行简单的Assembly.Load().EntryPoint.Invoke()。**

****

****

**阶段二**

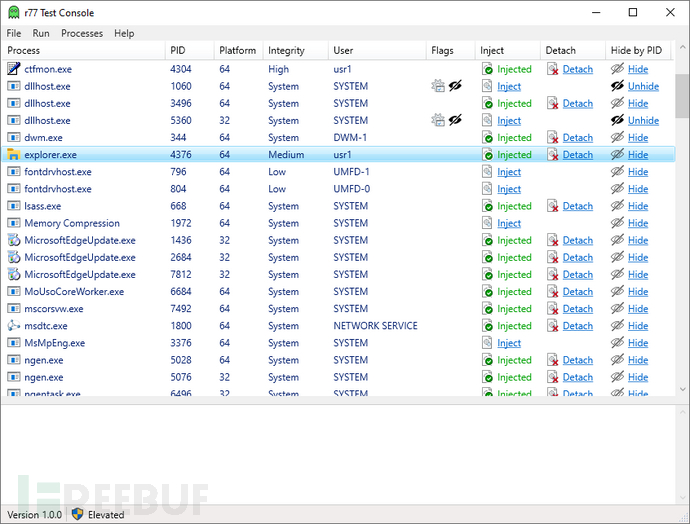
**执行的C#代码为stager，它将会使用Process Hollowing技术创建r77服务进程。r77服务是一个本地可执行文件，分别以32位和64位架构继续宁编译。父进程被设置为了winlogon.exe以增加欺骗性（模糊性）。另外，这两个进程被ID隐藏，在任务管理器中不可见。**

****

**磁盘上从未存储可执行文件或DLL。stager存储在注册表中，并从其资源加载r77服务可执行文件。**

**测试环境**

**测试控制台可以用来向单独进程注入r77，或接触进程跟Rootkit的绑定关系：**

****

1. **实验结论及心得体会**
2. **理解了WinDBG内核调试的过程。**
3. **了解了R77rootkit工具的原理和使用方法。**