课本第十章练习实验报告

专业：信息安全 姓名：魏伯繁 学号：2011395

### **一、实验工具：**

IDAPro7.0 IDAPro7.7 win10虚拟机 winxp虚拟机

### 二、实验环境：

静态分析主要在win10虚拟机完成。

内存4GB

主频2.9GHZ

### 三、实验内容

第一部分：实验分析

WinDbgX.exe -k com:port=\.\pipe\com\_2,baud=115200,pipe

Lab10-01：

1、这个程序是否直接修改了注册表？（使用procmon来检查）

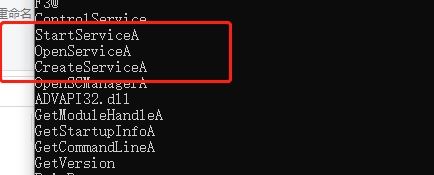
2、用户态的程序调用了ControlService函数，你是否能使用WinDbg设置一个断点，以此来观察由于ControlService的调用导致内核执行了怎样的操作。

3、这个程序做了什么？

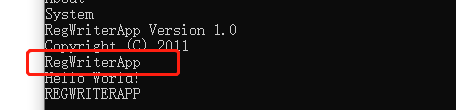
分析过程

第一步：使用基本静态分析检查代码：

利用strings工具查看其基本组成：在exe中可以看到这个程序会启动一个服务，并且注册一个服务



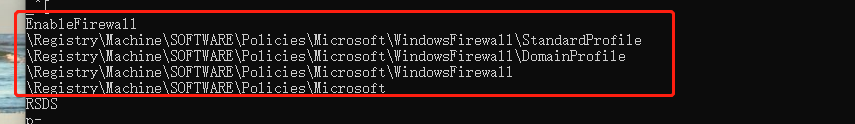
并且，一个很有用的信息就是在最后出现了一个RegWriteApp的字符串，代表了这个exe文件是一个可以修改注册表的可执行程序



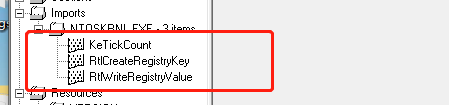
对sys文件使用strings工具验证了我们的猜想，这个程序会对注册表进行修改并注册服务



当然，更重要的是：其包含了一些注册表键值，前缀\register\machine响相当于用户态程序访问HKEY\_LOCAL\_MACHINE，上面的EnableFirewall就是关闭了windows的防火墙



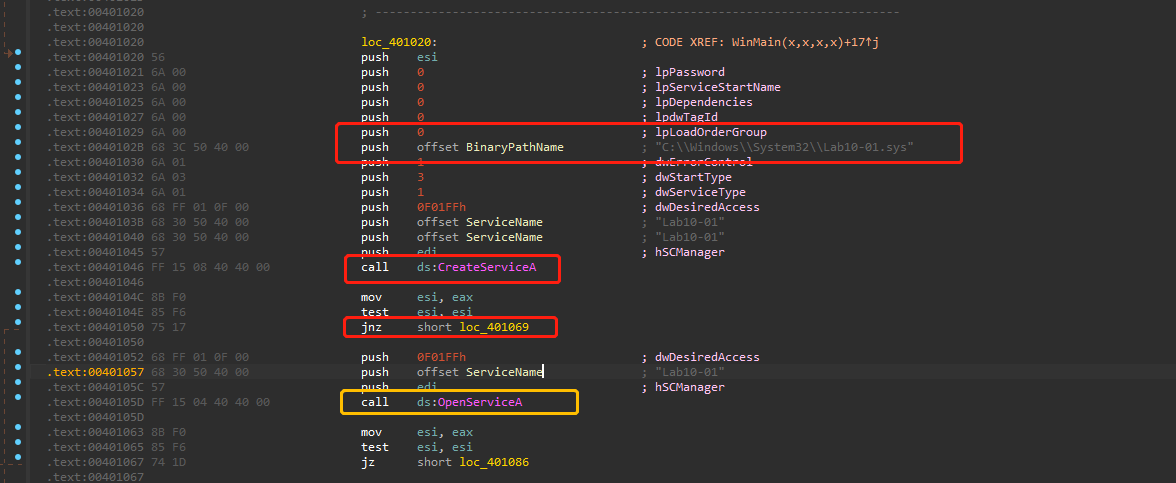
再使用PEBrowser查看sys文件，可以看到他具有三个导入函数：后两个函数都是对注册表的修改所需要的API函数



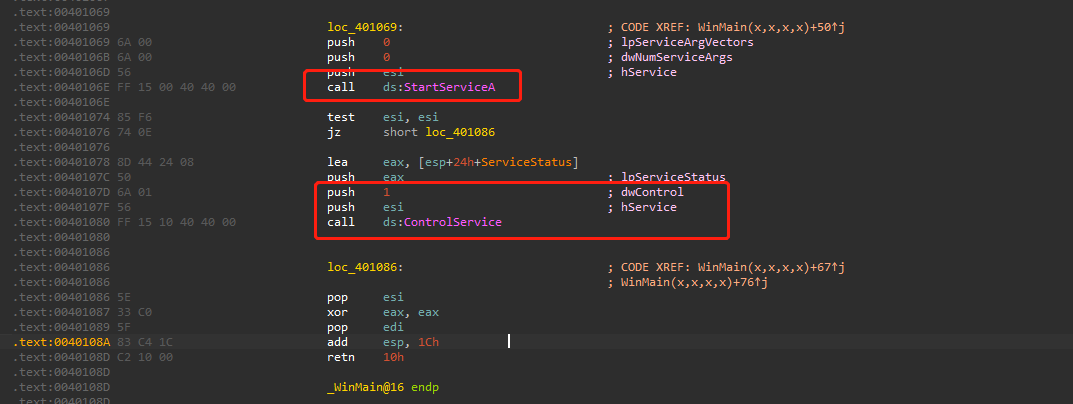
第二步：使用IDAPro分析文件

首先我们通过IDAPro打开exe文件，发现他主要做的事情就是在注册服务，首先通过OpenSCMangerA获取服务管理器的句柄，然后创建一个名字叫做Lab10-01的服务，并且调用了C:\Windows\System32\Lab10-01.sys中的代码，而根据下面的StartType=3判断这个文件会被加载进入内核。

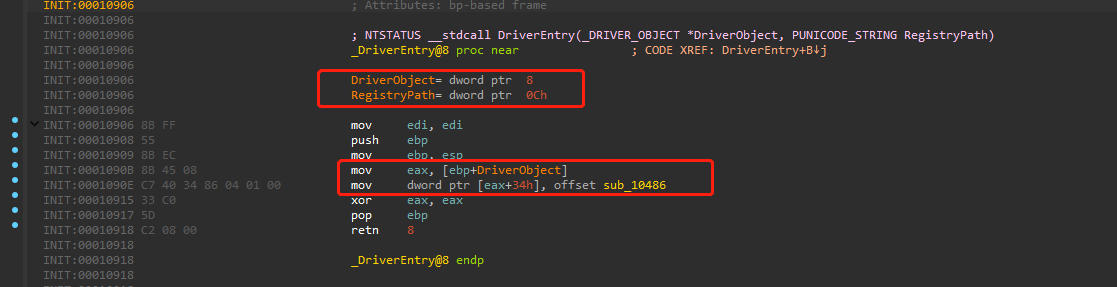
如果成功打开，那么就跳转到401069，如果失败则调用OpenServiceA，代表这个服务在宿主机中已经存在而，不用反复感染，只需要直接返回句柄操作即可。



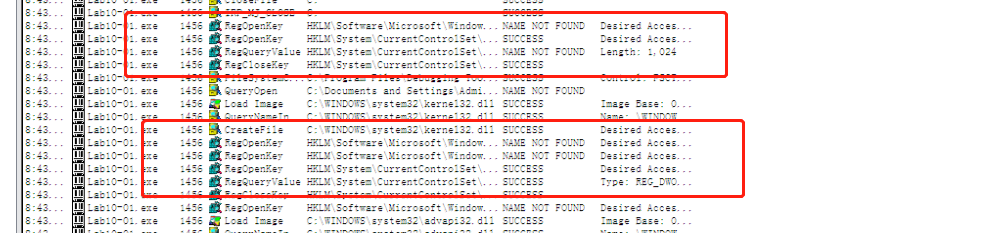
而在最后有一个ControlService函数，根据参数的Control=1判断这个函数的调用将会卸载驱动，并调用驱动的卸载函数。



随后，我们通过IdaPro打开sys文件：一个驱动对象作为参数被传递，整个DriverEntry的结构非常短小，他只是将一个由参数所提供的位置的固定偏移量加载如内存，除此之外再没有其他动作。



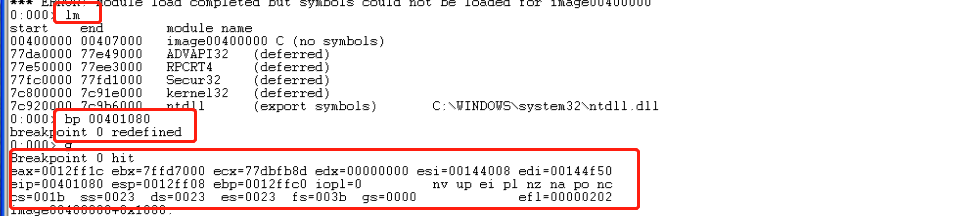
第三步：使用基本动态分析工具procmon分析，发现他确实对注册表有打开、查询等行为，但暂时没有找到具体的更改注册表的操作，这些需要通过进一步的内核调试来完成



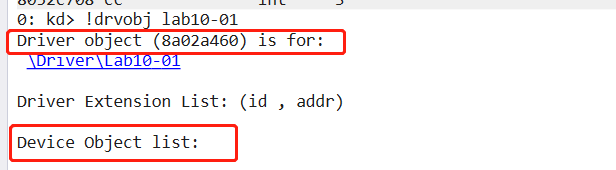
第四步：内核调试

紧接着，我们就可以使用内核调试来获得详细的程序对内核的操作了：

为了在Lab10-01.sys载入内存后使用windbg分析他，在虚拟机中，我们将可执行程序载入到windbg中，使用如下命令可以在驱动加载以及卸载之间打上一个断点，在打断点之前使用lm命令查看目前的模块内容，然后启动程序直到断点命中，当断点命中之后可以观察到如下信息：

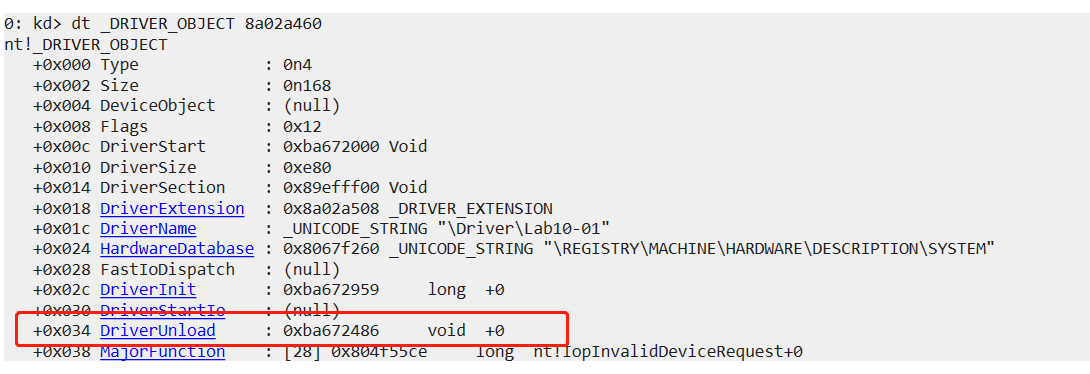


此时，因为sys文件已经成功载入内存，所以我们返回win10宿主机对xp虚拟机进行内核调试。具体的环境配置方法可以参考网上很多的教程，本人踩的坑就是需要把端口号注册为com\_2，我是没有打印机也要注册成com\_2，否则会出现一直连不上虚拟机的情况出现。然后我们就可以通过！drvobj命令来获取驱动对象

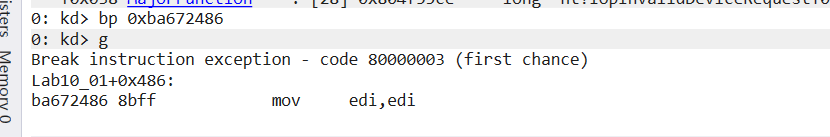


于是我们获得了驱动对象的地址，这个驱动对象是我们在进程中可以获得的，同样可以观察到列表的设备为空，所以可以分析得到这个驱动没有供用户空间中应用程序访问的设备。

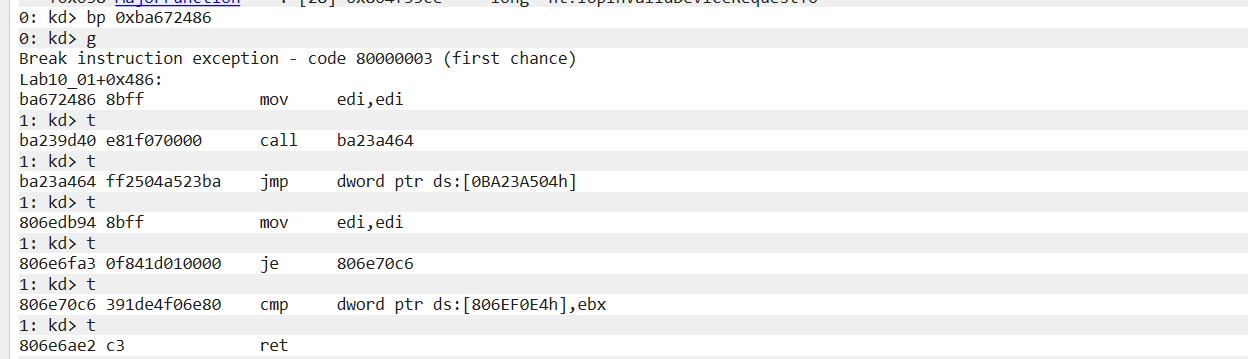
于是，我们获取了驱动对象以及其地址8a02a460，使用dt命令查看驱动对象：可以找到卸载时调用的函数，函数的地址为0xba672486

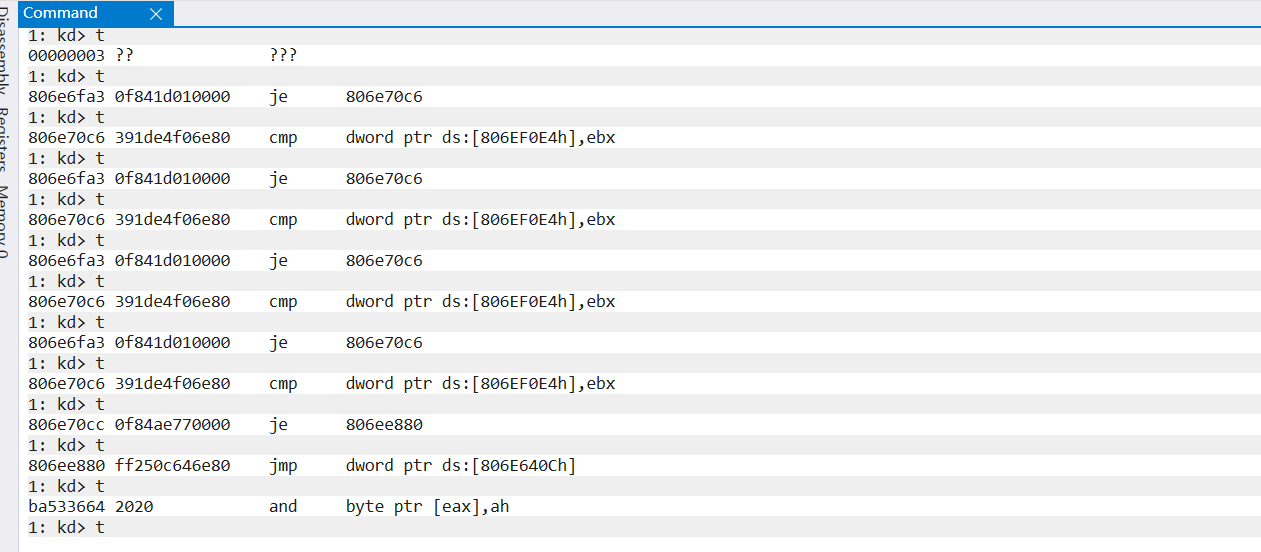


接着，在宿主机设置断点，并且恢复执行，随后在xp中同样使用g命令恢复执行，非常快程序就执行到了断点，此时xp卡死，我们退回到宿主机查看到了一条汇编代码，我们使用单步调试分析后续的指令

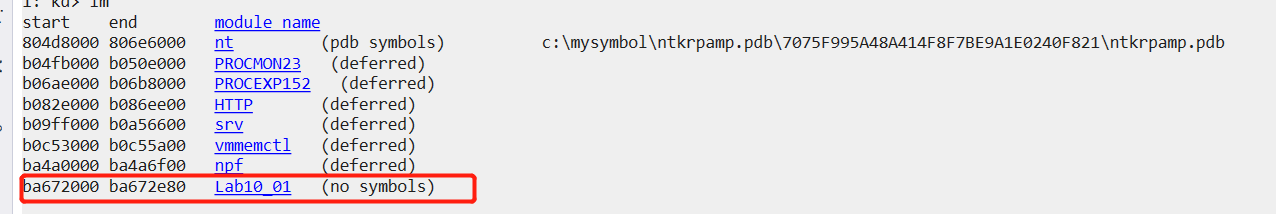


可以观察到程序调用了三次CreateRegistryKey函数，创建了注册表键然后两次调用WriteRegistryValue在两个地方将防火墙的值设置为0

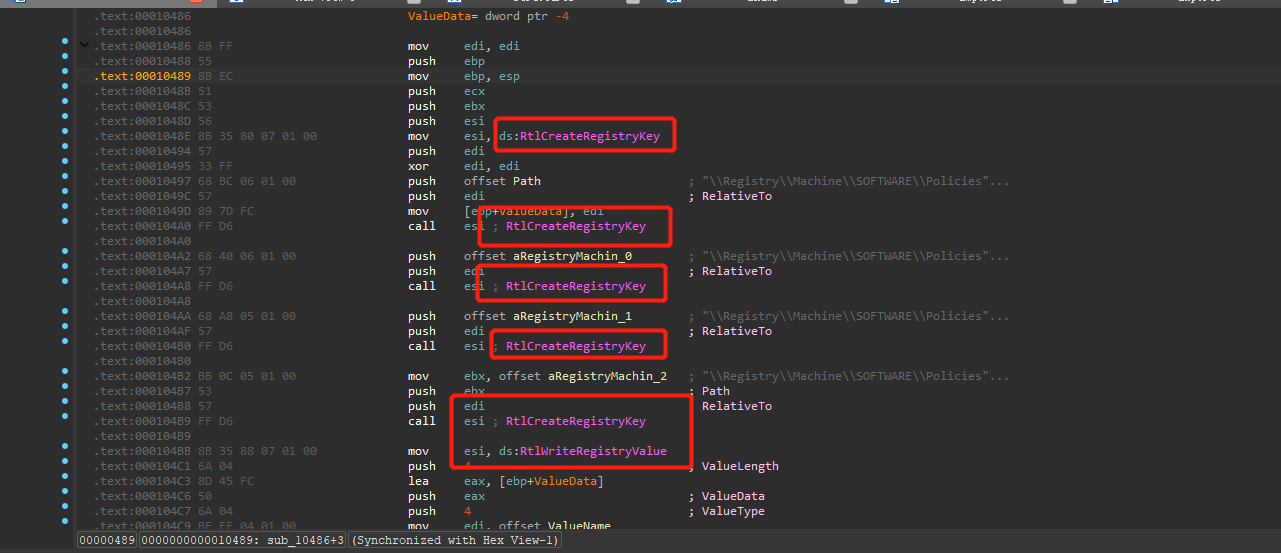




我们可以不在windbg中调试冗长的卸载函数，而是在IDA pro中完成分析，使用lm指令可以发现文件被加载到ba672000，卸载函数被加载到ba672486，然后可以通过在IDAPro中加上基址的方式找到卸载函数



也就是在这里，我们再次返回到用IDAPro加载sys文件：可以看到在基址（0x00100000）+0x486位置果然出现了刚才分析的函数



Lab10-02：

1、这个程序创建文件了吗？它创建了什么文件

2、这个程序有什么内核组件吗？

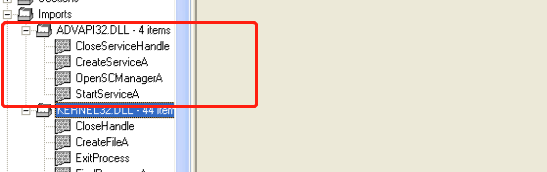
3、这个程序做了什么？

分析过程：

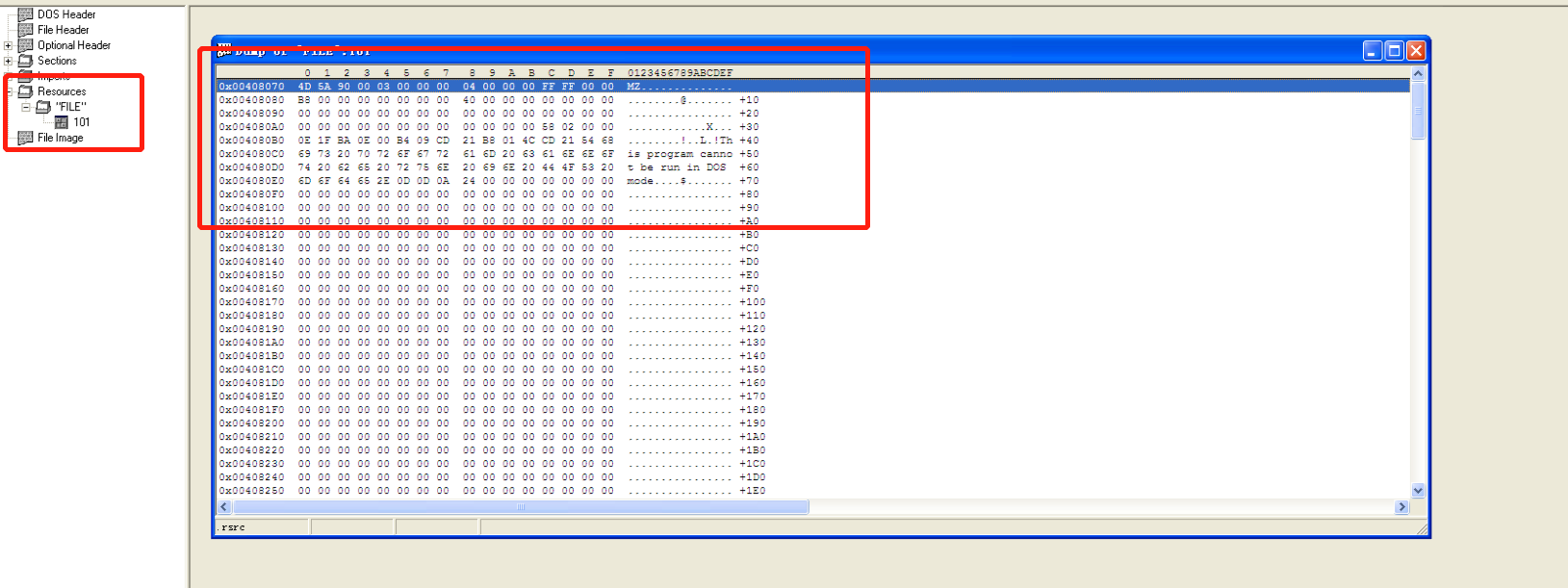
先说明一下，这个lab我的xp虚拟机不知道在抽什么疯，一直告诉我他在把那个Mlwx486写入一个只读内存，我也是前前后后研究了一个多小时都没有弄明白哪里出了问题，甚至回到了10月的虚拟机再试也不行，所以说这篇关于内核调试的方法以及思路包括图片来源于互联网，我会在这个lab的末尾明确标注出来，但是所有静态调试都是自己的截图以及思路。

第一部分：基本静态分析：

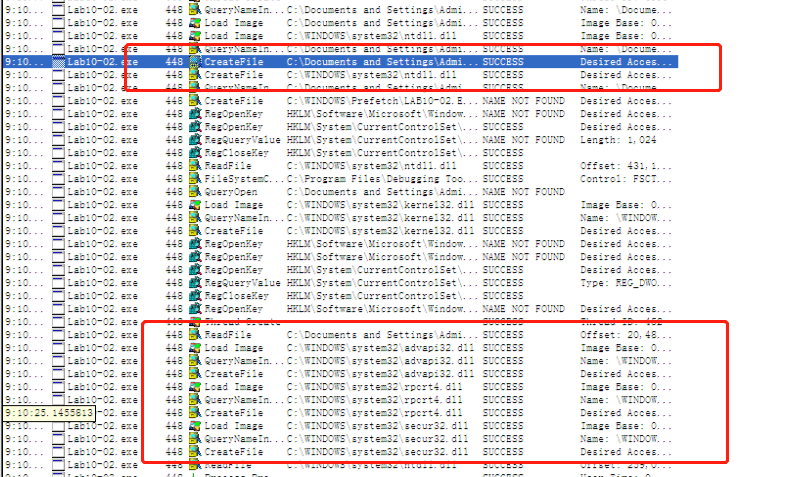
首先，通过分析导入表我们得知，该exe文件导入了StartServiceA和CreateServiceA，这个exe文件想要注册一个服务并且开启这个服务



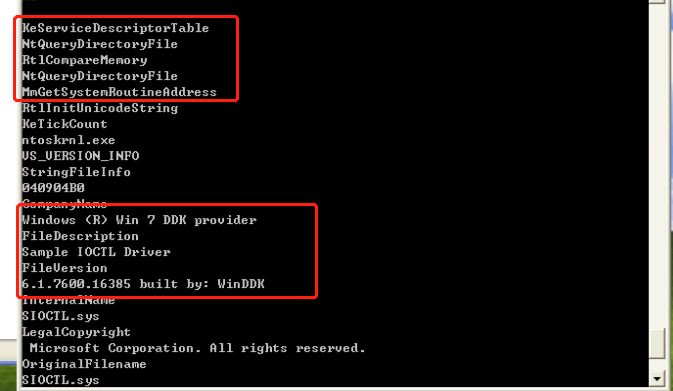
接下来我们发现，这个文件的资源节很特殊，他可以直接被PEBrowser识别为一个二进制文件，于是我们点击资源节可以看到：我们可以精准的识别出一个PE文件呢的头部，那么相当于这个exe携带了另一个文件的代码，那么我们猜测他很有可能是要将这个代码放置在某个地方并为这个文件注册为服务。



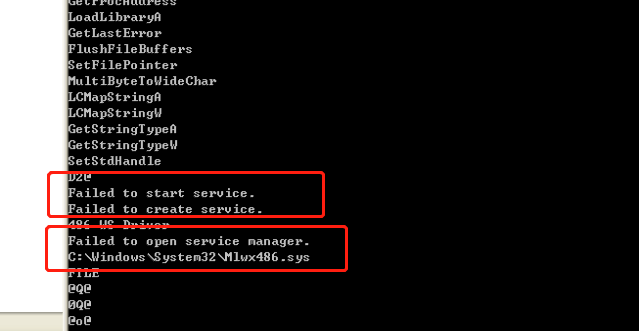
紧接着，我们使用procmon监视进程，发现这个可执行文件使用CreateFile更改了系统文件，并且查找、查看了注册表



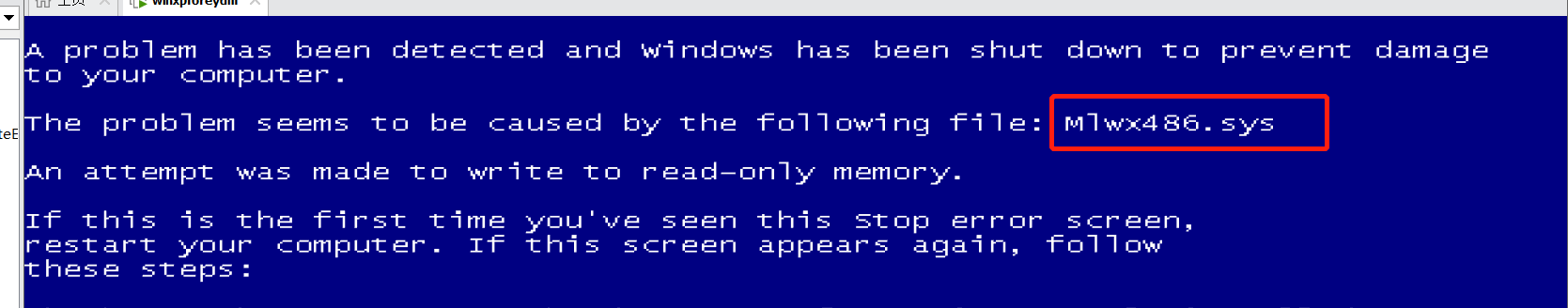
然后，我们使用strings工具对可执行文件进行扫描，发现其是一个内核文件，他的函数以及DDK都证明他是一个影响内核的文件（然后这里明确写了需要Win7的DDK，我在想是不是因为我是xp所以这个sys文件会拷贝的内存的只读位置）



接下来这张图片进一步验证了我们的猜想，资源节的文件会被生成为sys文件，并被命名为Mlwx486放置在System32文件夹下



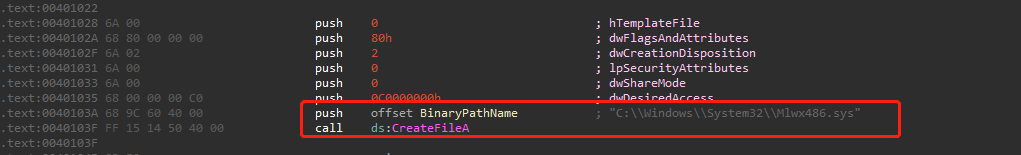
然后就出现了经典的一幕....加载到内存出错，那也没办法，只能硬着头皮往下做了



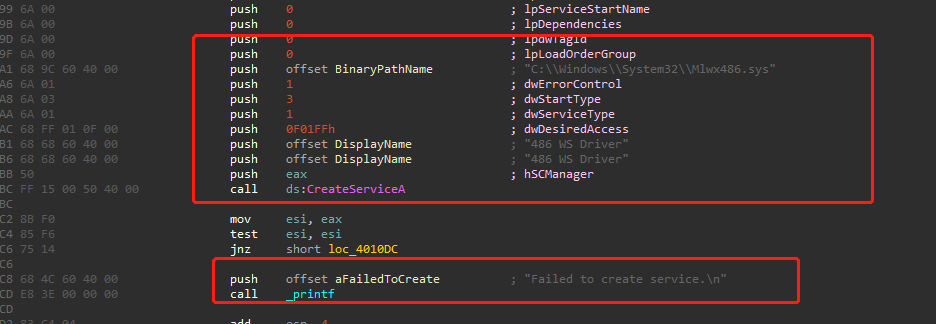
既然这样，不如我是用ResourceHacker把sys文件导出然后在idapro下看一看，于是接下来进入高级静态分析的部分：

第二部分：高级静态分析

首先是对exe文件的IDAPro分析，exe文件的结构相对简单，在前面我们已经分析的差不多了，简单的看一下就是：创建一个Mlwx486文件，也就是资源文件中的二进制内容



接下来就是注册服务的过程，果不其然，exe文件将资源节的文件注册为了一个服务，服务的名称就是486 WS Driver，位置就是Mlwx486.sys文件的位置

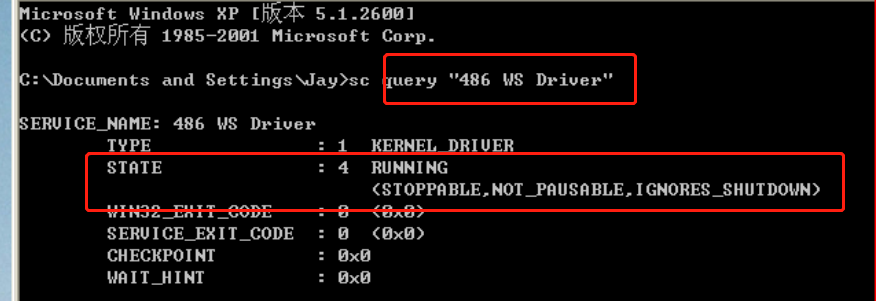


第三部分：内核调试

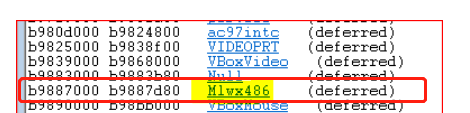
由于我的虚拟机抽风，这一部分不得不假手于网上的部分图片，关于Mlwx486.sys的IDa分析会融合如这部分的内核调试。

图片参考网址：[(116条消息) 恶意代码分析实战 Lab 10-2 习题笔记\_isinstance的博客-CSDN博客](https://blog.csdn.net/isinstance/article/details/79363150)

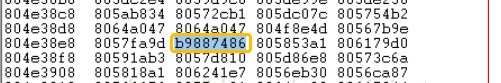
那么，我们知道他要在system32下创建了一个sys文件，但是当我们去找的时候却并没有看见这个文件，于是我们大概可以猜到这是一个Rootkit，在虚拟机中使用sc命令查找服务得到如下结果：



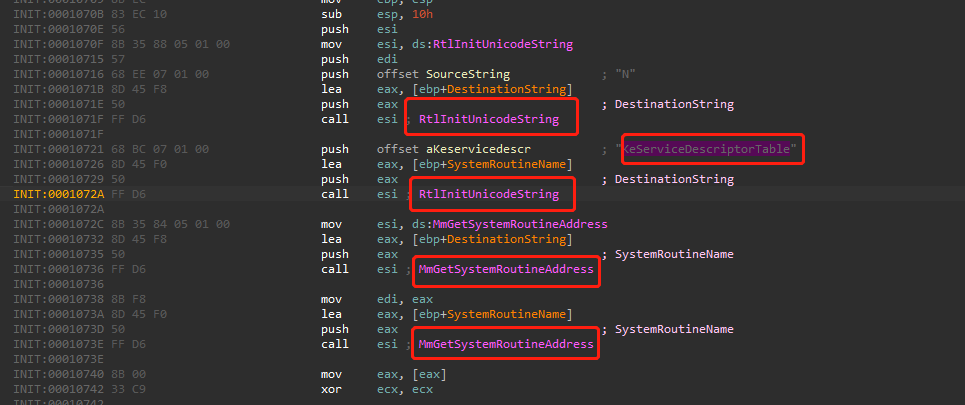
在宿主机上对虚拟机进行内核调试，可以观察到确实有一个与静态分析时一致的内核模块被正确加载了：



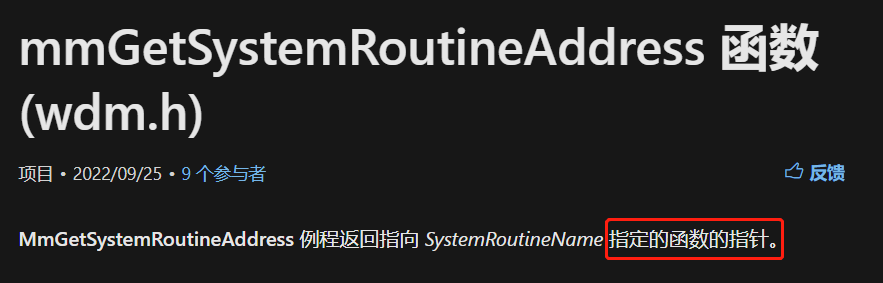
紧接着，使用命令dd dwo(keServiceDescriptorTable) L100查看SSDT的所有修改项,可以得到如下结果：我们看到有一个内存位置明显偏移于ntoskrnl的模块范围之外，所以也就是说这个sys文件替换覆盖了某一个函数



为了确定到底替换了哪一个函数，我们可以直接使用ResourceHacker导出Vlwx486文件并放置在IDAPro中进行检查：可以看到这里sys文件调用了我们在strings分析时看到的函数



前面的函数还能猜出来，就是初始化一个unicode编码的字符串，然后后面的函数没见过，于是到微软官网查了一下，发现是用来返回一个函数指针的

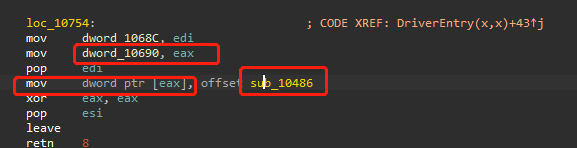


那么他读入的是哪个函数呢，就是前面初始化化的KeServiceDescriptorTable函数，这个函数是干嘛用的呢，我也不太懂，于是我查了一下:

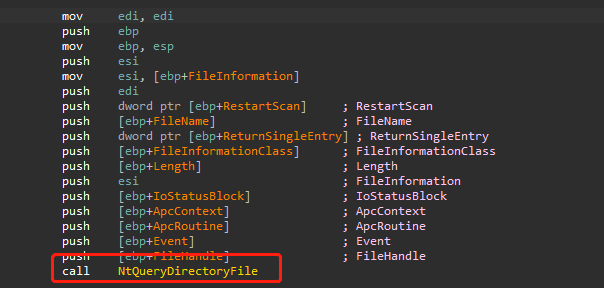
KeServiceDescriptorTable 结构   
KeServiceDescriptorTable：是由内核（Ntoskrnl.exe）导出的一个表，这个表是访问SSDT的关键，具体结构是  
 typedef struct ServiceDescriptorTable {  
 PVOID ServiceTableBase;  
 PVOID ServiceCounterTable(0);  
 unsigned int NumberOfServices;  
 PVOID ParamTableBase;  
 }

也就是在为之后替换覆盖函数，进行hook攻击做准备

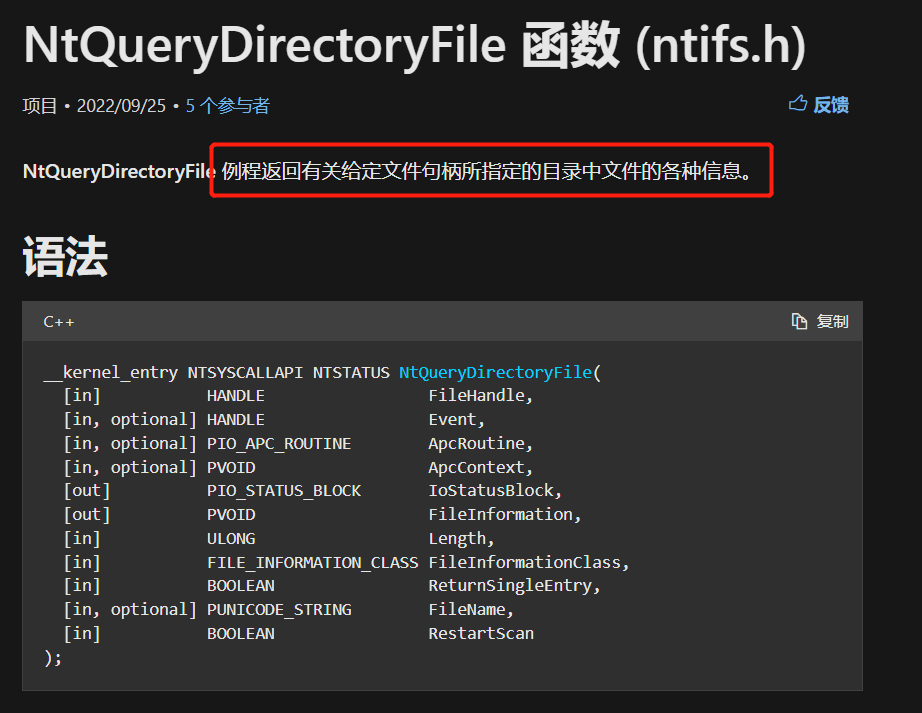
随后我们可以看到一个地址替换：那么在sub\_10486位置的是什么东西呢？我们点击进去看一下



是一个函数，这个函数是NtQueryDirectoryFile



这个函数是干嘛用的我也不清楚，于是也是到微软官网查了一下：该函数用于返回给定文件句柄锁指定的目录文件中的各种信息



微软对这个函数的具体描述如下：

在对 **NtQueryDirectoryFile** 的第一次调用中，如果为找到的第一个条目创建的结构太大而不适合输出缓冲区，则例程会将结构的固定部分写入输出缓冲区。 然后，该例程将写入输出缓冲区，使其与 **FileName** 字符串一样多。 (结构固定部分包含除最终 **FileName** 字符串以外的所有字段。在第一次调用（但不在后续调用中）时，I/O 系统可确保缓冲区足够大，足以容纳相应 **FILE\_\*XXX\* 的** 固定部分\_INFORMATION structure.) 发生这种情况时， **NtQueryDirectoryFile** 将返回适当的状态值，例如STATUS\_BUFFER\_OVERFLOW。

每次调用时， **NtQueryDirectoryFile** 都会返回尽可能多的 **FILE*\*XXX\**INFORMATION** 结构 (每个目录条目) ，完全可以包含在 **FileInformation** 指向的缓冲区中。 在第一次调用中，仅当输出缓冲区包含至少一个完整结构时， **NtQueryDirectoryFile** 才会返回STATUS\_SUCCESS。 在后续调用中，如果输出缓冲区不包含任何结构，**NtQueryDirectoryFile** 将返回STATUS\_SUCCESS，但设置 **IoStatusblock-Information**> = 0 以通知调用方此条件。 在这种情况下，调用方应分配更大的缓冲区并再次调用 **NtQueryDirectoryFile** 。 未报告任何剩余条目的相关信息。 因此，除了上述只返回一个条目的情况下，必须至少调用两次 **NtQueryDirectoryFile** 来枚举整个目录的内容。

那么他是如何做到隐藏文件的呢？FileInformation的结构由一系列FILE\_BOTH\_DIR\_INFORMATION结构组成，这个结构的第一个域指向下一个FILE\_BOTH\_DIR\_INFORMATION结构的偏移量，只要我们能够修改第一个结构的域的指针指向的位置，使其指向第三个结构那么我们就不能看到第二个结构了，也正是运用了这种方式让每个以Mlwx开头的文件都可以被跳过。

Lab10-03：

1、这个程序做了什么？

2、一旦程序运行，该怎么停止他？

3、他的内核组件做了什么？

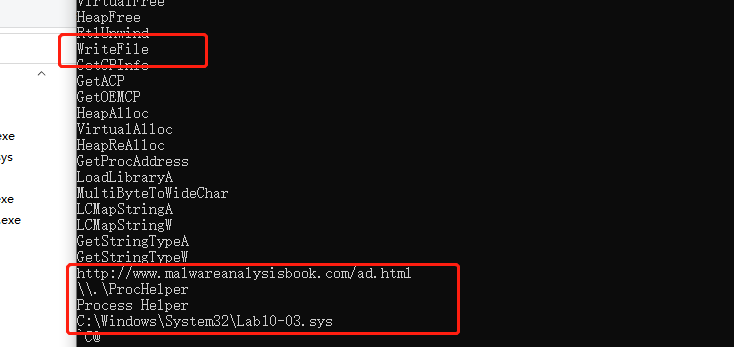
分析过程：

第一步：使用简单的静态分析

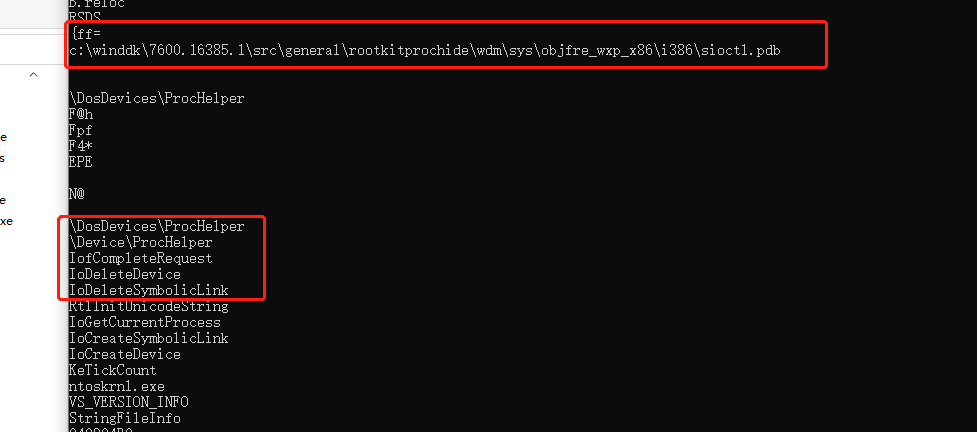
首先对exe文件进行分析，发现他需要创建一个文件并且会创建并启动一个服务，和其他恶意代码的功能差不多，exe函数看起来是在为恶意程序创造一个良好的运行环境



可以看到这个exe文件还具有写文件的能力，并且这个exe文件的目的在这张图片里已经很清楚了，他是为了显示一个广告页面而产生的，那么为什么简单的产生一个网站页面需要驱动来辅助呢，带着这个问题我们继续往下分析。



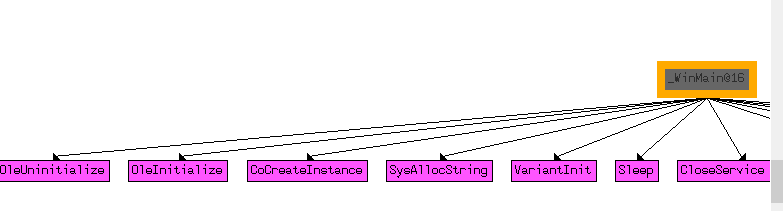
在对sys文件进行基本静态分析时我们发现了这样一个函数IoGetCurrentProcess导入函数，这个函数为进程提供了足够的信息，这个函数的调用告诉我们这个驱动或者修改正在运行的进程或者需要进程的信息。

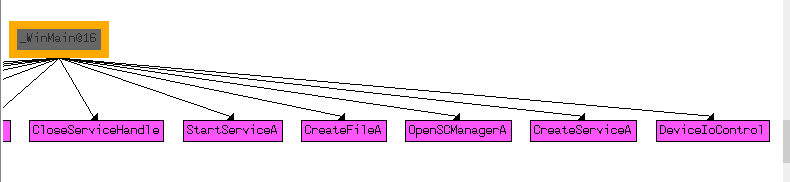


这个时候，我们将sys文件拷贝到system32目录下然后运行exe文件，可以发现没过30秒网页就会跳转一次但是我们却没办法终止他，任务管理器没有将程序列出，在Process Exploer中也没发现Lab10-03的存在，这个时候我们才明白，这个程序不仅仅是弹出广告页，而且还让我们无法终止这个程序的运行。

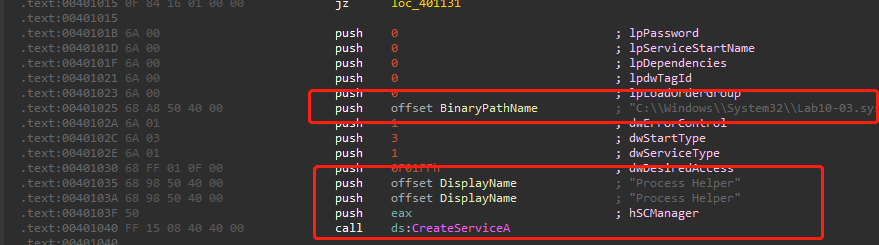
第二步：高级静态分析

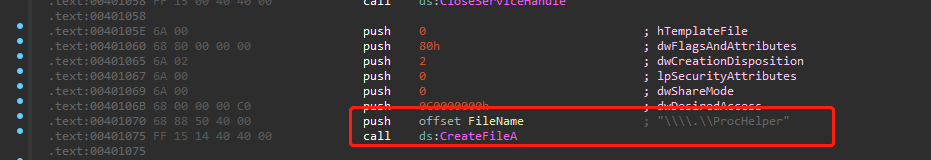
首先，我们使用IdaPro加载exe文件，在主函数中分析其调用的函数，WinMain函数的功能大体上可以分为两部分，第一部分是OpenSCManager、CreateService、StartService、CloseServiceHandle、CreateFile、DeviceIoControl函数，包括创建启动服务，发送请求到内核，第二部分由其余部分组成，由OleInitialize函数可知这是一个COM对象的调用



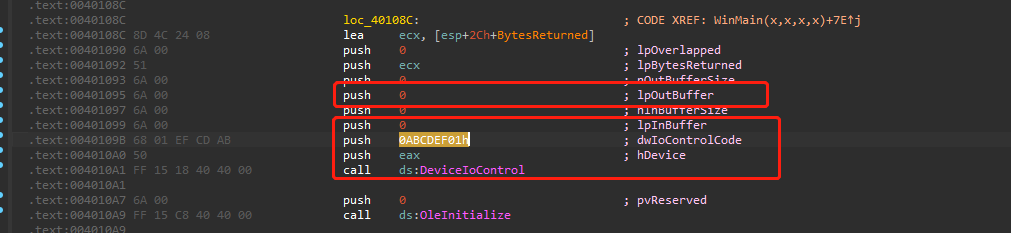


在CreateService函数中，我们可以看到程序创建了一个Process Helper的服务，他负责加载驱动Lab10-03.sys然后启动该服务加载sys文件到内核并打开句柄.\\ProcHelper

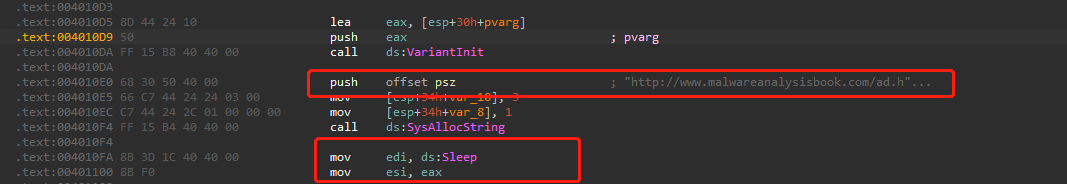




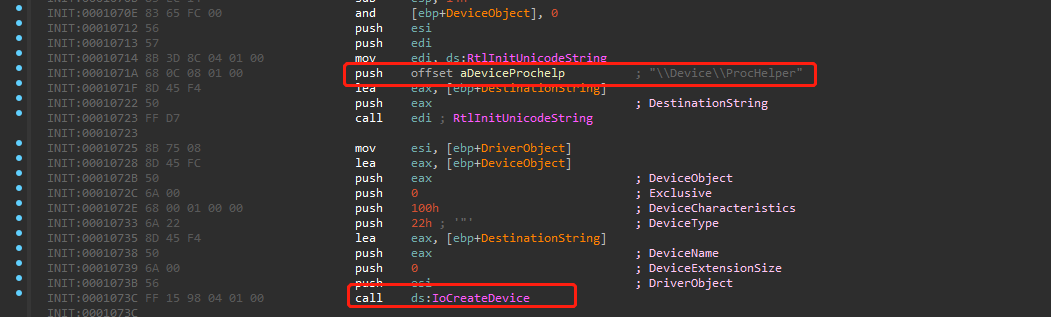
然后我们就来到了DeviceIoControl函数，可以观察到这个函数的lpOutBuffer和lpInBuffer都被设置为0，这意味着他并没有传输任何信息到内核也没有接受内核传递的任何信息，但是我们传递了0ABCDEF01作为dwIoControICode被传递给内核驱动



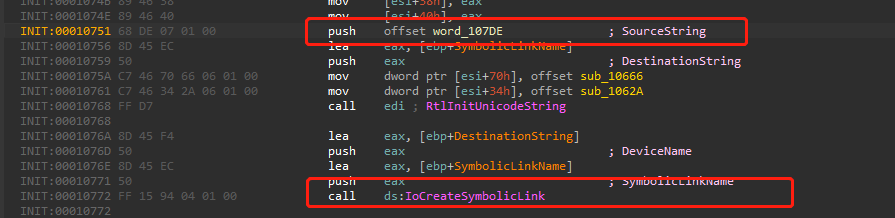
随后的内容体现了exe文件的基本内容：也就是访问一个网站然后睡眠30秒之后再次循环运行

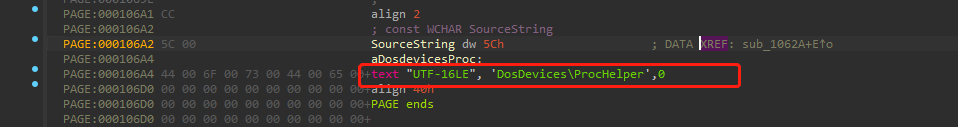


然后，我们用IdaPro打开内核文件，可以看到他调用了IoCreateDevice函数，创建了一个\Device\ProHelper的设备



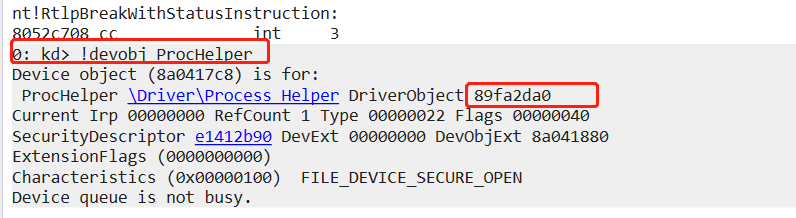
随后函数调用IoCreateSymbolicLink函数，创建了一个名为\DosDevices\ProcHelper的符号链接来提供用户态的程序访问



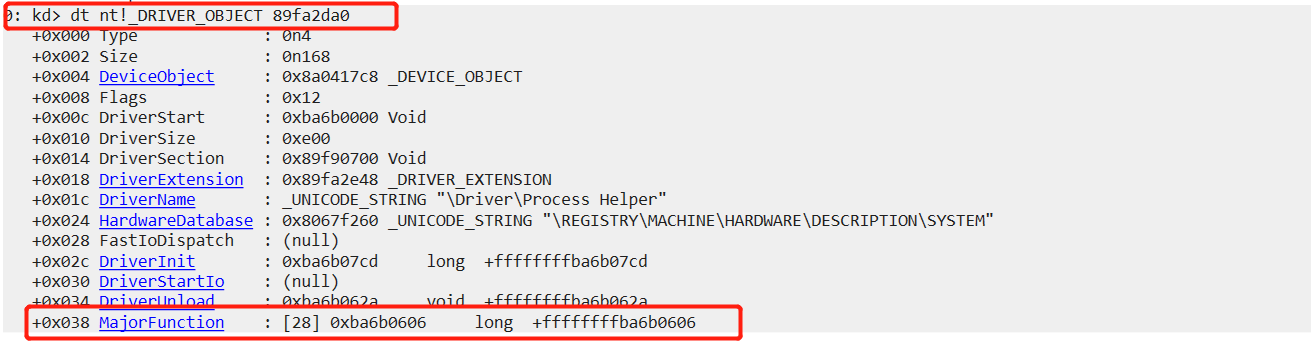


第三部分：使用Windbg进行内核调试

在虚拟机运行恶意代码将内核驱动加载到内存，通过前面的分析我们能知道设备对象在\Device\ProcHelper，所以我们来启动它，为了找到在设备对象中被执行的函数，我们必须找到他的设备对象，还是使用！devobj来完成

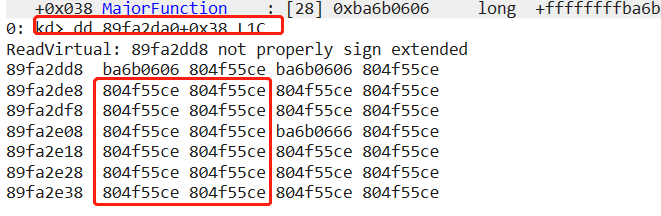


DriverObject包含所有函数的指针，当用户空间的程序访问设备对象时调用这些函数，DriverObject存储在一个叫做DRIVER\_OBJECT的数据结构中，我们可以使用dt命令查看标注的驱动对象：

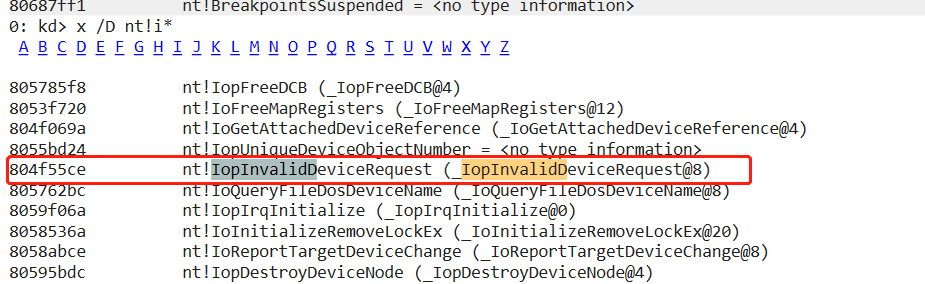


这段代码包含了几个需要注意的函数指针，可以看到DriverInit、DriverEntry、DriverUnload在实验一中被重点分析，当驱动卸载时被调用，可以在IdaPro中看到它删除了符号链表和DriverEntry创建的设备

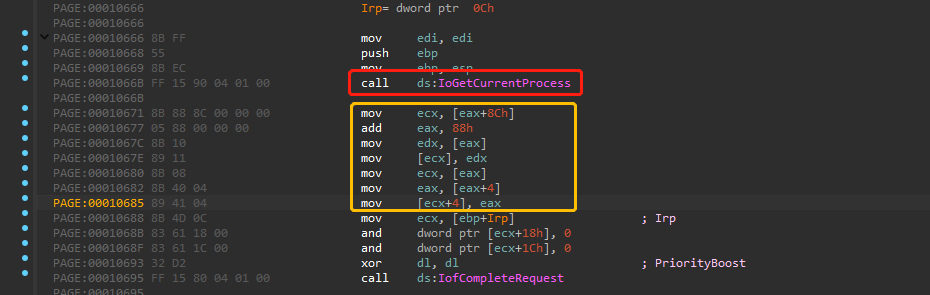
接下来可以分析主函数表，最具有意义的驱动通常在这完成，我们可以通过dd命令查看主函数中表项，表中的每一项表示不同类型的请求，其中大部分都是804f55ce，这个项表示了驱动不能处理的一个请求类型。



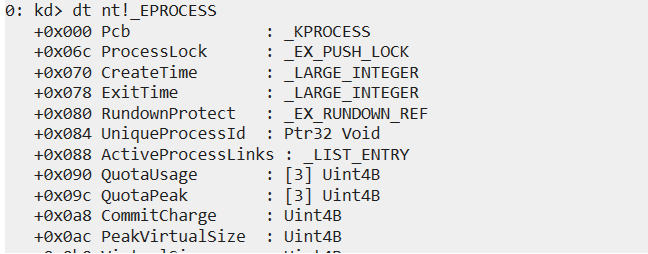
我们可以根据他的表项查找到对应的函数是什么，使用x /D命令，并选择以I开头的内容可以看到对应地址的内容为IopInvalidDeviceRequest，这意味他处理驱动无法处理的非法请求



返回IDAPro查看这个函数：可以看到DeviceIoControl操作了当前进程的PEB，这个函数做的第一件事就是调用函数IoGetCurrentProcess，这个函数会返回DeviceIoControl的EPROCESS结构，然后这个函数在接下来访问偏移量为0x88处的数据，再然后访问偏移量为0x8c处的下一个DWORD



使用dt命令发现存储在PEB结构偏移量0x88和0x8c的LIST\_ENTRY



LIST\_ENTRY和操作系统课程中使用的双向链表很像，他就是一个包含prev和next的双向链表，sys呈现首先获取列表中指向下一个结构的指针，然后获取列表中指向前一个的指针，再覆盖下一项的prev指针使其指向前一项，在此之前，下一项的prev指针指向覆盖项，也就是说，整个链表结构跳过了中间的LIST\_ENTRY结构

在操作系统征程运行时，每个进程都有一个指针指向它前一个进程和后一个进程的指针，然后在本恶意代码中，他通过修改LIST\_ENTRY使其跳过Lab10-03进程让我们不能通过控制面板或者process explorer查看到该进程。

第二部分：YARA 规则编写

LAB10-01.exe的yara规则

import "pe"  
rule IsPE{  
 meta:  
 description = "检查文件是否为PE文件"  
 author="wbf"  
 date="2022-11-19"  
 condition:  
 uint16(0) == 0x5A4D and //“MZ”头  
 uint32(uint32(0x3C)) == 0x00004550 // “PE”头  
  
}  
  
rule smallFileSize{  
 meta:  
 description = "检查文件小是否大概率为exe文件"  
 author="wbf"  
 date="2022-11-19"  
 condition:  
 filesize<500KB  
}  
  
rule mayBeMaleware{  
 meta:  
 description = "检查文件小是否大概率为maleware文件"  
 author="wbf"  
 date="2022-11-19"  
 strings:  
 $a = { E8 00 00 00 00 }  
 $b = "malware" nocase  
 condition:  
 $a at pe.entry\_point or $b  
}  
  
rule Lab1001exe{  
 meta:  
 description = "检查改文件是否大概率与Lab010-01的文件属于同一类型的恶意代码"  
 author="wbf"  
 date="2022-11-19"  
 strings:  
 $a = "REGWRITERAPP" nocase  
 $b = "C:\Windows\System32\Lab10-01.sys"  
 $c = "GetLastActivePopup"  
 $d = "OpenServiceA"  
 $e = "StartServiceA"  
 $f = "OpenSCManagerA"  
 $g = "GetCurrentProcess"  
 $h = "GetOEMCP"  
 $i = "RegWriterApp Version 1.0"  
 condition:  
 (6 of them) and (IsPE and smallFileSize) or (mayBeMaleware)  
}

LAB10-01.sys的YARA规则

import "pe"  
rule IsPE{  
 meta:  
 description = "检查文件是否为PE文件"  
 author="wbf"  
 date="2022-11-19"  
 condition:  
 uint16(0) == 0x5A4D and //“MZ”头  
 uint32(uint32(0x3C)) == 0x00004550 // “PE”头  
  
}  
  
rule smallFileSize{  
 meta:  
 description = "检查文件小是否大概率为exe文件"  
 author="wbf"  
 date="2022-11-19"  
 condition:  
 filesize<500KB  
}  
  
rule mayBeMaleware{  
 meta:  
 description = "检查文件小是否大概率为maleware文件"  
 author="wbf"  
 date="2022-11-19"  
 strings:  
 $a = { E8 00 00 00 00 }  
 $b = "malware" nocase  
 condition:  
 $a at pe.entry\_point or $b  
}  
  
rule Lab1001sys{  
 meta:  
 description = "检查改文件是否大概率与Lab010-01sys的文件属于同一类型的恶意代码"  
 author="wbf"  
 date="2022-11-19"  
 strings:  
 $a = "6.1.7600.16385"   
 $b = "RtlCreateRegistryKey"  
 $c = "RtlCreateRegistryKey"  
 $d = "ntoskrnl.exe"  
 $e = "c:\winddk\7600.16385.1\src\general\regwriter\wdm\sys\objfre\_wxp\_x86\i386\sioctl.pdb"  
 $f = "\Registry\Machine\SOFTWARE\Policies\Microsoft"  
 $g = "\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall"  
 $h = "EnableFirewall"  
 $i = "\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile"  
 $j = "\Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile"  
 condition:  
 (7 of them) and (IsPE and smallFileSize) or (mayBeMaleware)  
}

LAB10-02.exe的YARA规则：

import "pe"  
rule IsPE{  
 meta:  
 description = "检查文件是否为PE文件"  
 author="wbf"  
 date="2022-11-19"  
 condition:  
 uint16(0) == 0x5A4D and //“MZ”头  
 uint32(uint32(0x3C)) == 0x00004550 // “PE”头  
  
}  
  
rule smallFileSize{  
 meta:  
 description = "检查文件小是否大概率为exe文件"  
 author="wbf"  
 date="2022-11-19"  
 condition:  
 filesize<500KB  
}  
  
rule mayBeMaleware{  
 meta:  
 description = "检查文件小是否大概率为maleware文件"  
 author="wbf"  
 date="2022-11-19"  
 strings:  
 $a = { E8 00 00 00 00 }  
 $b = "malware" nocase  
 condition:  
 $a at pe.entry\_point or $b  
}  
  
rule Lab1002exe{  
 meta:  
 description = "检查改文件是否大概率与Lab010-02exe的文件属于同一类型的恶意代码"  
 author="wbf"  
 date="2022-11-19"  
 strings:  
 $a = "6.1.7600.16385"   
 $b = "Windows (R) Win 7 DDK driver"  
 $c = "ntoskrnl.exe"  
 $d = "MmGetSystemRoutineAddress"  
 $e = "NtQueryDirectoryFile"  
 $f = "KeServiceDescriptorTable"  
 $g = "c:\winddk\7600.16385.1\src\general\rootkit\wdm\sys\objfre\_wxp\_x86\i386\sioctl.pdb"  
 $h = "C:\Windows\System32\Mlwx486.sys"  
 $i = "486 WS Driver"  
 $j = "GetOEMCP"  
 $k = "OpenServiceA"  
 $l = "StartServiceA"  
 $m = "OpenSCManagerA"  
 condition:  
 (8 of them) and (IsPE and smallFileSize) or (mayBeMaleware)  
}

LAB10-03.exe

import "pe"  
rule IsPE{  
 meta:  
 description = "检查文件是否为PE文件"  
 author="wbf"  
 date="2022-11-19"  
 condition:  
 uint16(0) == 0x5A4D and //“MZ”头  
 uint32(uint32(0x3C)) == 0x00004550 // “PE”头  
  
}  
  
rule smallFileSize{  
 meta:  
 description = "检查文件小是否大概率为exe文件"  
 author="wbf"  
 date="2022-11-19"  
 condition:  
 filesize<500KB  
}  
  
rule mayBeMaleware{  
 meta:  
 description = "检查文件小是否大概率为maleware文件"  
 author="wbf"  
 date="2022-11-19"  
 strings:  
 $a = { E8 00 00 00 00 }  
 $b = "malware" nocase  
 condition:  
 $a at pe.entry\_point or $b  
}  
  
rule Lab1003exe{  
 meta:  
 description = "检查改文件是否大概率与Lab010-03exe的文件属于同一类型的恶意代码"  
 author="wbf"  
 date="2022-11-19"  
 strings:  
 $a = "C:\Windows\System32\Lab10-03.sys"   
 $b = "Process Helper"  
 $c = "\\.\ProcHelper"  
 $d = "http://www.malwareanalysisbook.com/ad.html"  
 $e = "GetOEMCP"  
 $f = "GetCurrentProcess"  
 $g = "OLEAUT32.dll"  
 $h = "ole32.dll"  
 $i = "OleInitialize"  
 $j = "CoCreateInstance"  
 $k = "Sleep"  
 $l = "GetModuleFileNameA"  
 condition:  
 (8 of them) and (IsPE and smallFileSize) or (mayBeMaleware)  
}

LAB10-02.sys的YARA规则

import "pe"  
rule IsPE{  
 meta:  
 description = "检查文件是否为PE文件"  
 author="wbf"  
 date="2022-11-19"  
 condition:  
 uint16(0) == 0x5A4D and //“MZ”头  
 uint32(uint32(0x3C)) == 0x00004550 // “PE”头  
  
}  
  
rule smallFileSize{  
 meta:  
 description = "检查文件小是否大概率为exe文件"  
 author="wbf"  
 date="2022-11-19"  
 condition:  
 filesize<500KB  
}  
  
rule mayBeMaleware{  
 meta:  
 description = "检查文件小是否大概率为maleware文件"  
 author="wbf"  
 date="2022-11-19"  
 strings:  
 $a = { E8 00 00 00 00 }  
 $b = "malware" nocase  
 condition:  
 $a at pe.entry\_point or $b  
}  
  
rule Lab1003sys{  
 meta:  
 description = "检查改文件是否大概率与Lab010-03sys的文件属于同一类型的恶意代码"  
 author="wbf"  
 date="2022-11-19"  
 strings:  
 $a = "Lab10-03.sys"   
 $b = "OriginalFilename"  
 $c = "VS\_VERSION\_INFO"  
 $d = "ntoskrnl.exe"  
 $e = "IoCreateDevice"  
 $f = "IoCreateSymbolicLink"  
 $g = "IoGetCurrentProcess"  
 $h = "RtlInitUnicodeString"  
 $i = "IoDeleteSymbolicLink"  
 $j = "\DosDevices\ProcHelper"  
 condition:  
 (7 of them) and (IsPE and smallFileSize) or (mayBeMaleware)  
}

第三部分：IDAPYTHON脚本编写：

因为有些程序很长，我们无法精准阅读每一个汇编指令，但我们查看其所有的函数调用：

for func in idautils.Functions(): print hex(func), idc.get\_func\_name(func)

查看函数的开始以及结束地址：

ea = here()  
func = idaapi.get\_func(ea) # 获取地址所在函数的范围  
print "Start: 0x%x, End 0x%x" % (func.startEA,func.endEA)

获得对应函数的汇编代码：

ea = here()  
start = idc.get\_func\_attr(ea, FUNCATTR\_START)  
end = idc.get\_func\_attr(ea, FUNCATTR\_END)  
cur\_addr = start  
while cur\_addr <= end:   
 print hex(cur\_addr), idc.generate\_disasm\_line(cur\_addr, 0)  
 cur\_addr = idc.next\_head(cur\_addr, end)

查看库函数：

for func in idautils.Functions():  
 flags = idc.get\_func\_attr(func, FUNCATTR\_FLAGS)  
 if flags & FUNC\_LIB:  
 print hex(func), "FUNC\_LIB", idc.get\_func\_name(func)

综合获取信息：

for func in idautils.Functions(): # 获取已知函数list  
 flags = idc.get\_func\_attr(func,FUNCATTR\_FLAGS) # 获取函数的标志  
 if flags & FUNC\_LIB or flags & FUNC\_THUNK: # 标志是否是FUNC\_LIB 或者 FUNC\_FLAGS  
 continue  
 dism\_addr = list(idautils.FuncItems(func)) # 函数指令地址  
 for line in dism\_addr:  
 m = idc.print\_insn\_mnem(line)  
 if m == 'call' or m == 'jmp':  
 op = idc.get\_operand\_type(line,0)  
 if op == o\_reg:  
 print "0x%x %s" % (line,idc.generate\_disasm\_line(line,0))