**恶意代码分析与防治技术实验报告**

**Lab7**

**一、实验环境**

Windows7(VMWARE虚拟机)，Windows11本机

**二、实验工具**

STRINGS, IDAPro，PEVIEW，EXEInfo，YARA，IDA Python，Netcat，Wireshark，Procmon

**三、实验内容**

**LAB10-1**

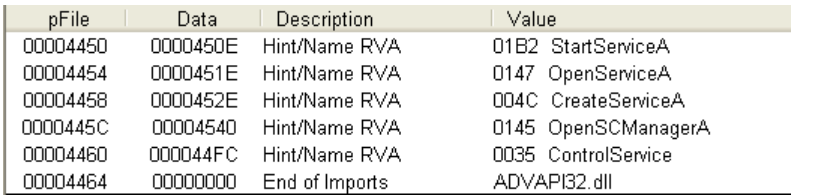
本实验包括一个驱动程序和一个可执行文件。你可以从任意位置运行可执行文件，但为了使程序能够正常运 行，必须将驱动程序放到C:\Windows\System32目录下，这个目录在受害者计算机中已经存在。可执行文件是 Lab10-01.exe，驱动程序是Lab10-01.sys。

**问题1**

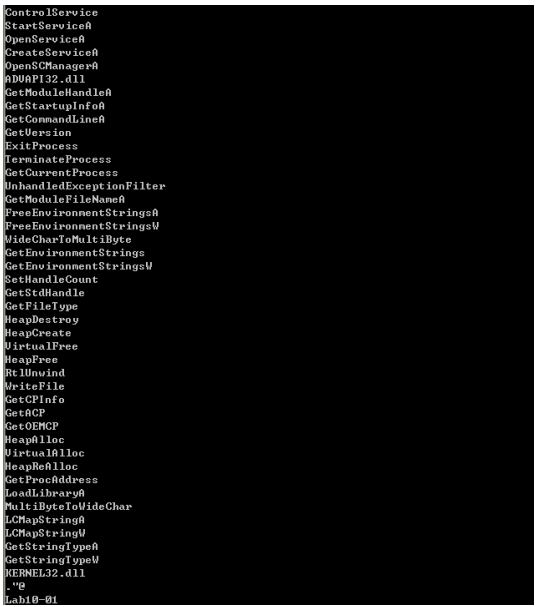
|  |
| --- |
| 这个程序是否直接修改了注册表（使用 procmon 来检查）？ |

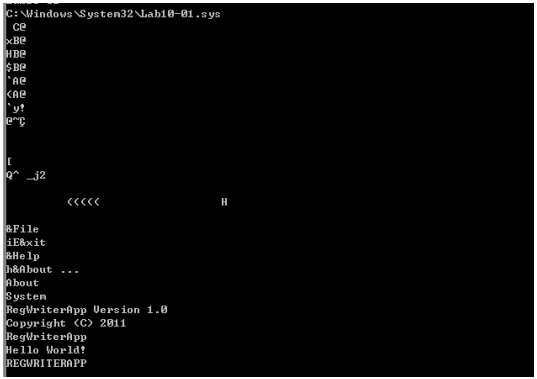
首先还是先进行静态分析

对Lab10-01.exe进行基础静态分析。发现如下导入函数，都与服务有关，推测该程序会创建一个服务，并 且可能启动或者操作了这个服务。



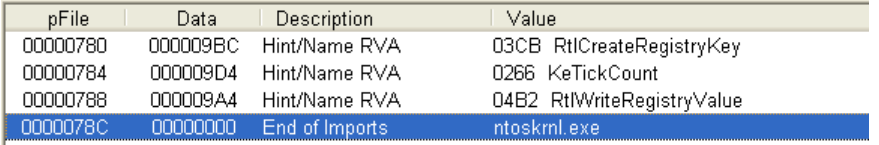
使用Strings工具进行一个简单的静态分析，查看一下有没有哪些比较值得注意的字符串。有意义的字符串如下，包含了C:Windows\System32\Lab10-01.sys这个驱动程序，和一些与操作内容相关的 字符串。



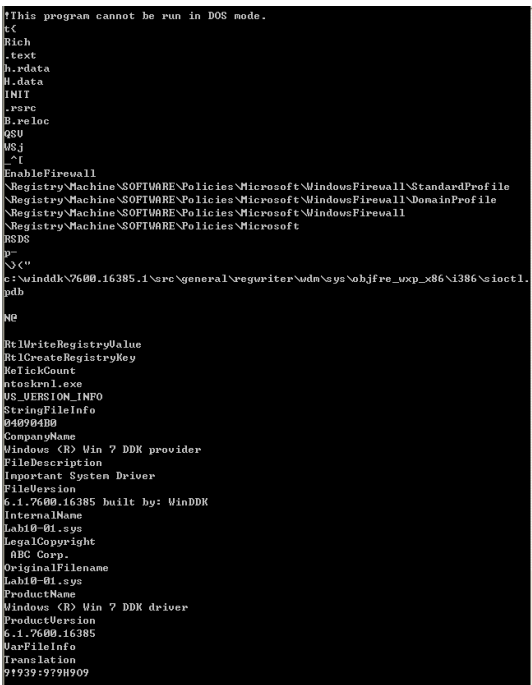


从字符串里可以看到有一个指定文件 C:\Windows\System32\Lab10-01.sys ，从上面的函数名称猜测会 有服务相关的行为，结合之前的文件，猜测可能服务会执行这个文件中的内容。

下面对驱动程序Lab10-01.sys进行基础静态分析，发现它只有三个导入函数，与修改注册表有关。



有意义的字符串如下。其中，EnableFilewall如果被置0，则表示禁用Windowsxp自带的防火墙。此外，害包 含一些注册表键值，内核态访问注册表时的前缀\Registry\Machine相当于用户态下的HKLM，推测恶意代码可 能写了注册表。

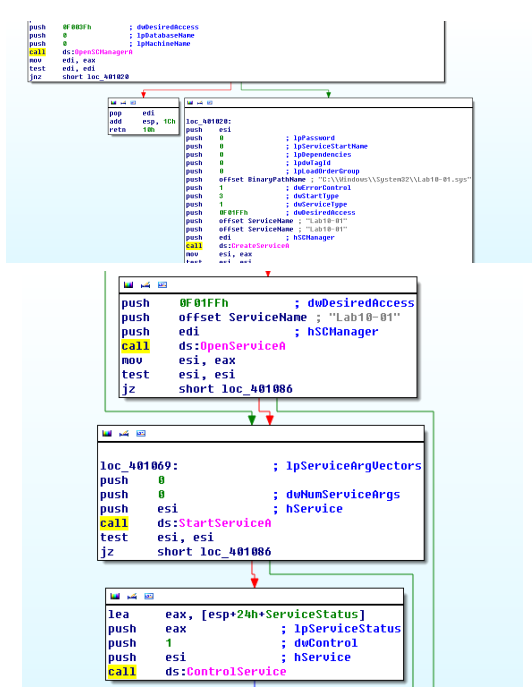


可以看到有类似注册表的操作，以及notskrnl.exe这个可执行文件。

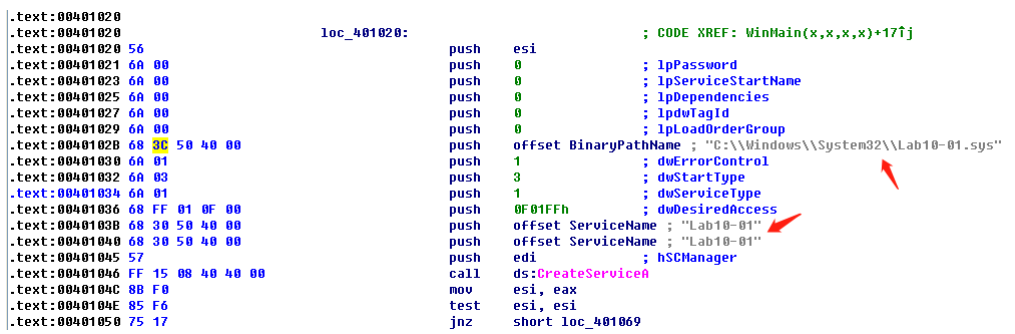
接下来，双击运行Lab10-01.exe，并用procmon监测这个程序都做了些什么。发现确实进行了很多关于注册表 的操作，但写注册表键值的操作只有一个，写入的是HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed。此外，我们还需要查看代码来确定驱动有没有修改注册表。



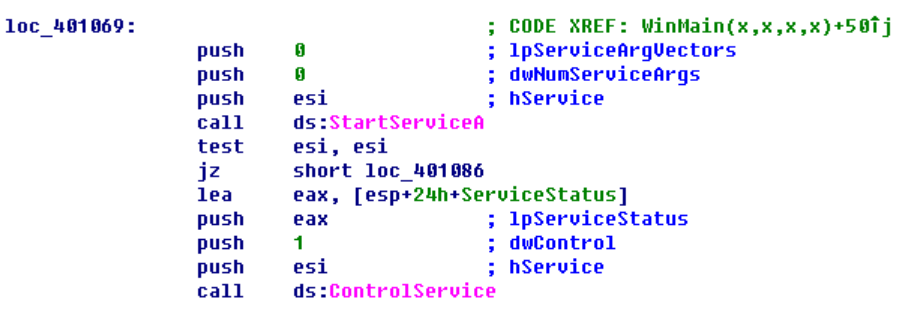
接下来使用IDA简单查看一下exe程序的所有功能



首先调用了OpenSCManagerA函数来获取服务管理器的句柄，然后调用CreateServiceA创建了一个名为 Lab10-01的服务，这个服务使用了C:\\Windows\\System32\\Lab10-01.sys中的代码，服务类型为3，意味 着这个文件将被加载到内核。调用结束后，判断返回值是否为0。如果返回值为0，说明调用失败，会调用 OpenServiceA打开这个已经存在的名为Lab10-01的服务。



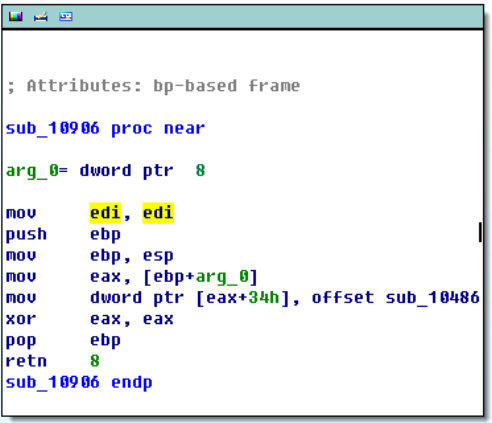
我们发现这个程序的功能比较简单，就是使用了OpenSCManagerA获取当前服务管理器的句柄，然后 CreateServiceA创建一个名为Lab10-01的服务，之后会调用StartServiceA启动服务，最后调用 ContorlService，在经过查阅资料以后可以发现ContorlService传入的第二个参数是1，这个参数的意思就是会卸载驱动。 同时我们可以注意到，在创建服务的时候使用了之前分析的驱动程序的代码，并且dwStartType设置为 3，也就是说这个服务会以内核级运行。



用IDA打开Lab10-01.sys，从DriverEntry开始分析。



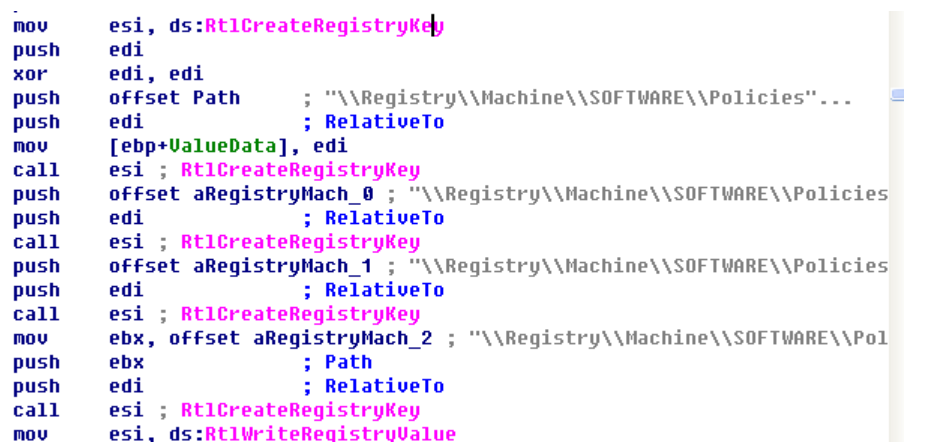
在00010964位置可以看到一个无条件跳转指令，跳转到了sub\_10906，也就是说这个驱动程序真正的入 口点应该是在这个函数的位置。IDA自动进入的是驱动的入口点，但并不是DriverEntry函 数。编译器会在DriverEntry的周围插入封装代码。可以看到，入口点通过jmp指令跳转到了sub\_10906处，那里才是真正的DriverEntry。



但是仅从这里并不能看出来有什么，只看见把一个偏移地址放到了内存中去。



双击进入，发现DriverEntry很短，只是做了一些内存上的操作，虽然没有进行任何函数调用，但是出现了一个 关于函数sub\_10486的赋值操作，进去就会发现这个函数中有一系列关于注册表的操作。

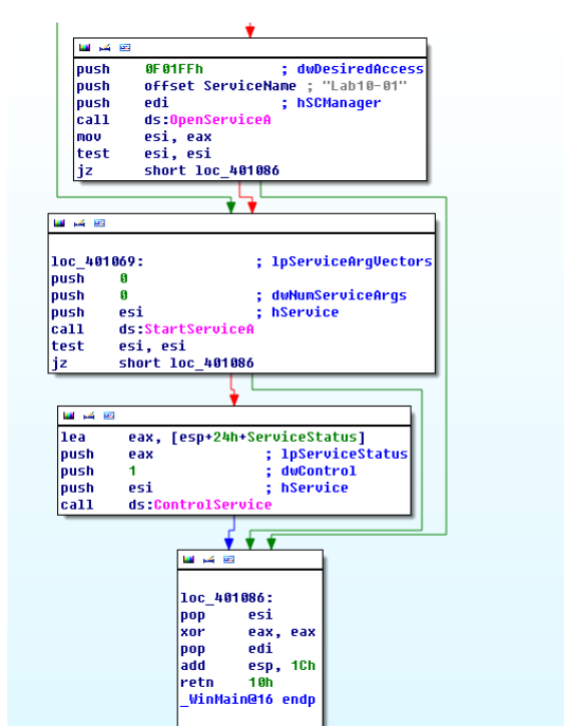


**问题2**

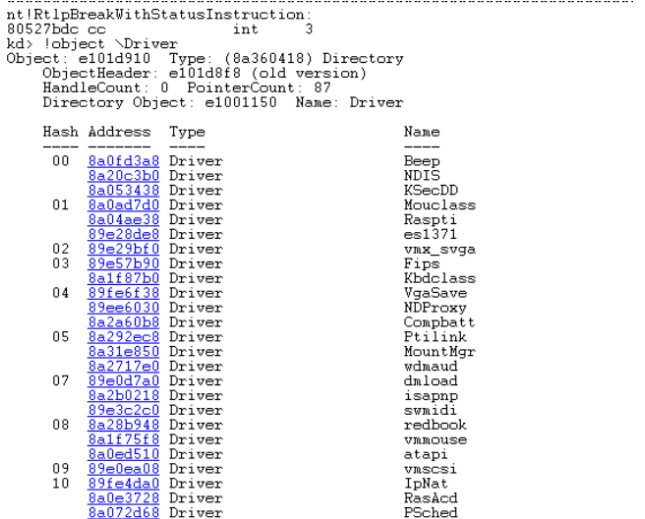
|  |
| --- |
| 用户态的程序调用了ControlService 函数，你是否能够使用WinDbg 设置一个断点，以此来观察由于 ControlService 的调用导致内核执行了怎样的操作？ |

**首先找到要设置断点的位置，在虚拟机中用OllyDbg打开这个程序设置断点，运行到断点处就跳出虚拟机，使用运行在宿主机中的 WinDbg 调试内核。可以使用!drvobj命令获得驱动设备的句柄，进而获得卸载函数所在的地址。接下来，在驱动的卸载函数上设置一个断点，返回虚拟机中继续运行致断点处，触发断点后又回到宿 主机中用WinDbg单步调试，查看具体的代码。**

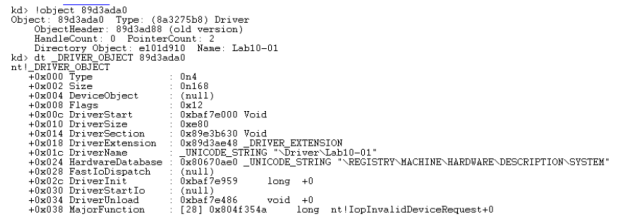
通过前述分析，我们知道了恶意代码打开 Lab10-01 服务，启动服务并最终通过 ControlService 关闭 它，如下图所示：



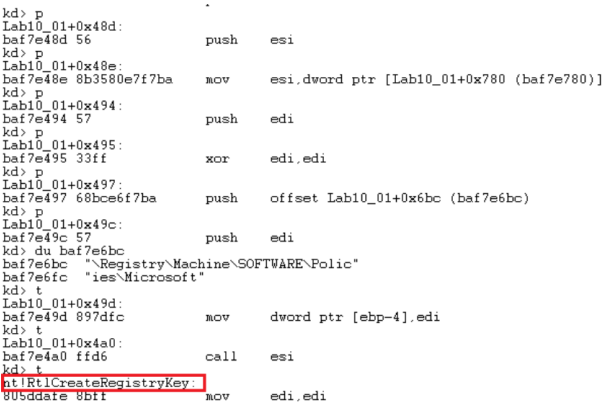
中断内核调试器并使用 !object \Driver 命令来显示已加载的驱动程序



通过观察我们发现SERVICE\_CONTROL\_STOP 将调用 DriverUnload 函数，要弄清楚 DriverUnload 函数 的地址是什么，首先要在 Lab10-01 驱动程序条目上设置一个断点，使用的命令为 bu Lab10\_01!DriverEntry 接下来我们需要单步执行，直到 Lab10\_01.sys 被加载。使用 step out 直至看到 nt！ IopLoadUnloadDriver + 0x45。 使用 !object \Driver 来列出已经加载的驱动程序，然后我们使用 dt 命令 （display out）来查看lab10-01 驱动程序：



在上图中我们可以看见，DriverUnload 函数的地址是0xbaf7e486（图片中的倒数第二行），使用指令 bp 0xbaf7e486 在这个位置上打个断点。重新启动lab10-01.exe，在这一处的断点被触发，之后已使 用 g 进行单步执行



在红框标记的位置可以看见有对注册表的操作，这里是调用了RtlCreateRegistryKey函数。



根据微软官方文档可以知道这个函数的作用就是在注册表中添加一个key。 同时在后面可以看见还有对RtlWriteRegistryValue 的调用，这样一个完整的键值对就被创建成功了。 通过观察调试时的操作，我们可以得到如下的参数列表内容：

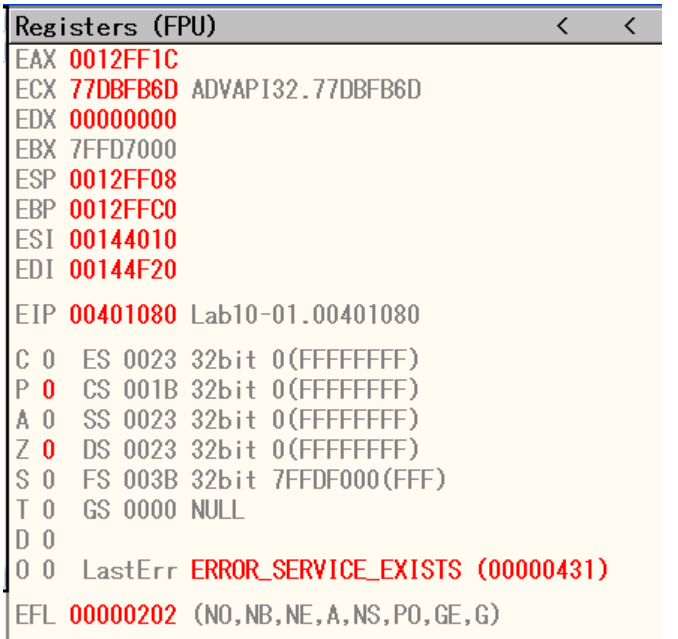
|  |
| --- |
| Java  NTSTATUS RtlWriteRegistryValue(   \_In\_ ULONG RelativeTo = 0,  \_In\_ PCWSTR Path = offset Lab10\_01+0x5a8(f7a545a8),   \_In\_ PCWSTR ValueName = offset Lab10\_01+0x4ee(f7a544ee),   \_In\_ ULONG ValueType = 4,   \_In\_opt\_ PVOID ValueData = eax,   \_In\_ ULONG ValueLength = 4   ); |

在这个参数列表中比价有价值、需要关注的就是Path参数，经过分析以后可以知道这个参数的值应该是 \Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile ，然后 ValueName 的值是 EnableFirewall 我们注意到这个ValueData设置的是eax中的值，根据上面汇编代码可以知道eax中存放的是[ebp-4]。很 幸运的是，windbg是有能够直接查看寄存器中值的指令的，通过 r ebp 可以得到当前ebp中的值是 f78ded58，那么ebp-4也就是f78ded54。使用指令 dc f78ded54 可以看到当前这个位置上的值是0，也 就是说这里参数ValueData的值是被设置为0的。这里的意思就是将 EnableFirewall 这个的值设置为了 0，意义就是从内核禁止了Windows的防火墙功能 继续向下分析，可以发现下面调用的函数是RtlWriteRegistryValue，这个和上一个是配套使用的，同样 是观察一下他的参数结构，经过分析以后可以得到

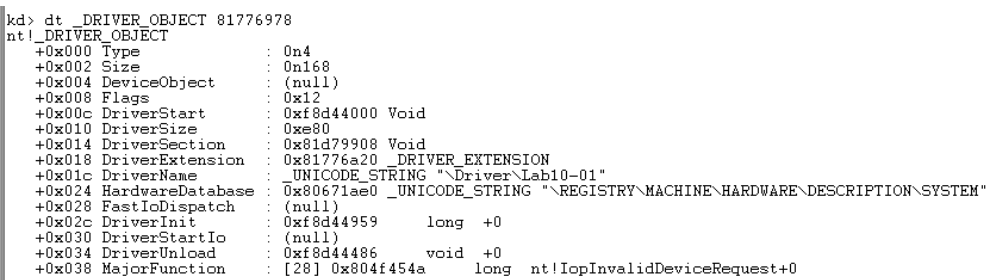
|  |
| --- |
| Java  NTSTATUS RtlWriteRegistryValue(   \_In\_ ULONG RelativeTo = 0,   \_In\_ PCWSTR Path = ebx,   \_In\_ PCWSTR ValueName = edi,   \_In\_ ULONG ValueType = 4,   \_In\_opt\_ PVOID ValueData = eax,   \_In\_ ULONG ValueLength = 4   ); |

这里需要关注的几个参数的具体值为： Path 的值应该是 \Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile ValueName 的值是 EnableFirewall ValueData 的值是 0 从效果上来说和之前的函数是一样的，也是从内核关闭防火墙 之后剩下的内容就是对栈的处理，程序执行结束。

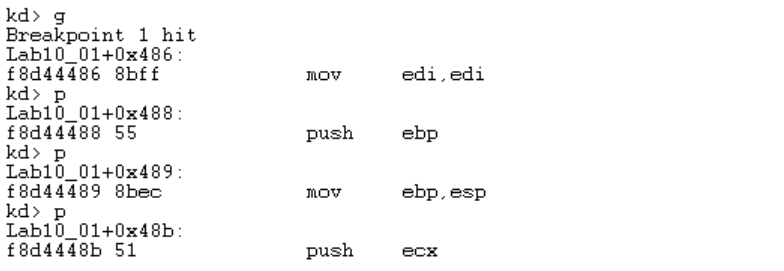
接下来，我们使用WinDbg来调试，以此查看调用ControlService卸载Lab10-01.sys时内核中都执行了哪些 操作。 由于用户态的程序会在加载Lab10-01.sys后立即卸载它，运行程序之前该驱动并不在内存中，而程序运行后 驱动又已经从内存中被卸载，因此我们需要设置一个合适的断点，来保证该驱动在内存中，这样才能进行分 析。 我们已经在IDA里面找到这个ControlService的地址，为0x401080，然后用OD打开Lab10-01.exe，在这个 地址打断点并执行到这里，此时寄存器的状态如下：



现在，我们就可以跳出虚拟机了，连接内核调试器，获取关于Lab10-01.sys的信息。首先使用命令!drvobj lab10-01，来获取驱动对象，结果如下。于是我们知道了驱动对象的地址81776978且这个驱动没有供用户空 间中应用程序访问的设备。



根据这个地址，使用命令bp 0xf8d44486设置一个断点，然后使用命令g恢复虚拟机的运行，回到虚拟机中， 在OD中运行F9调试，停在断点处，之后在WinDbg中使用p单步调试。



根据经验，这里显示的前三行汇编代码应该是一个函数入口的地方，说明刚刚那个断点断在了一个函数的入 口。之后看到的代码与前面用IDA看到的一致，多次调用了RtlCreateRegistryKey函数来创建注册表键，并 调用了两次RtlwriteRegistryalue函数将EnableFirewall的值设置为0，禁用Windows XP防火墙。

**问题3**

|  |
| --- |
| 这个程序做了些什么？ |

**这个程序运行之后会启动一个服务，然后通过运行一个驱动创建修改注册表的值为键 \Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\StandardProfile 和 \Registry\Machine\SOFTWARE\Policies\Microsoft\WindowsFirewall\DomainProfile，来关闭防火 墙。**

**LAB10-2**

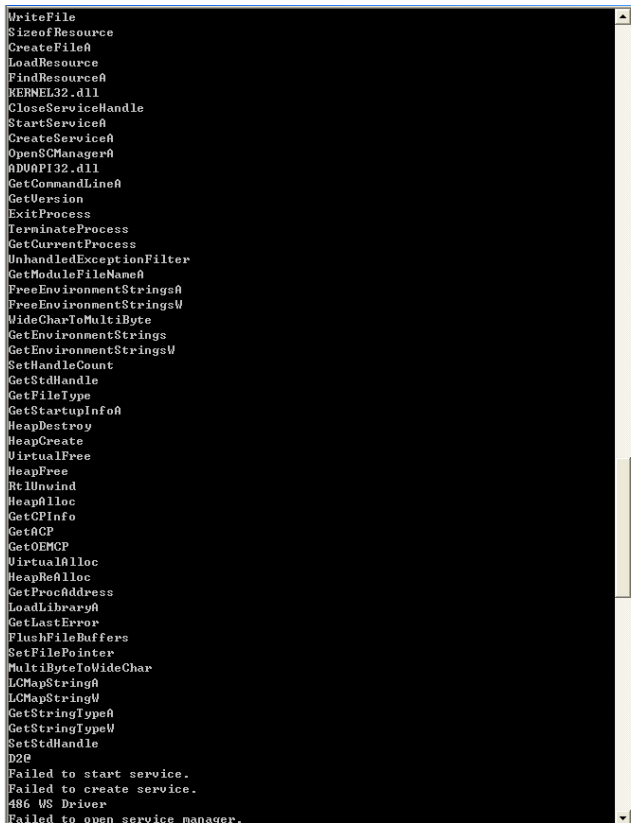
**问题一**

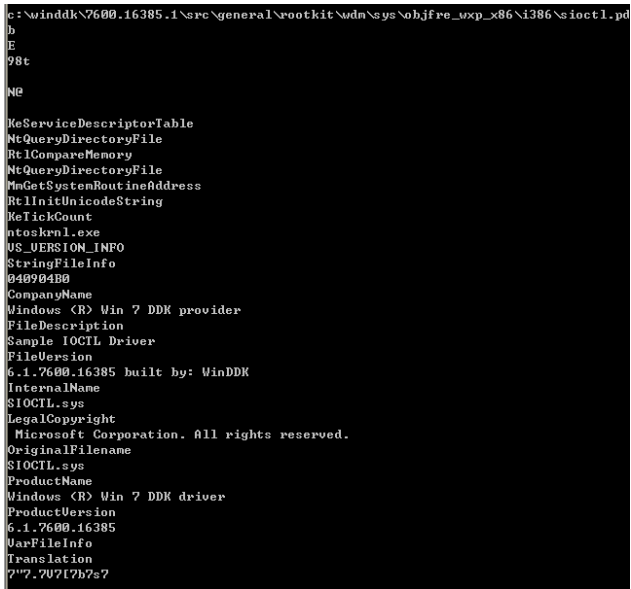
|  |
| --- |
| 这个程序创建文件了吗？它创建了什么文件？ |

**该程序创建了文件文件C:\Windows\System32\Mlwx486.sys。**

首先对Lab10-02.exe进行基础静态分析。

首先使用strings工具简单查看一下



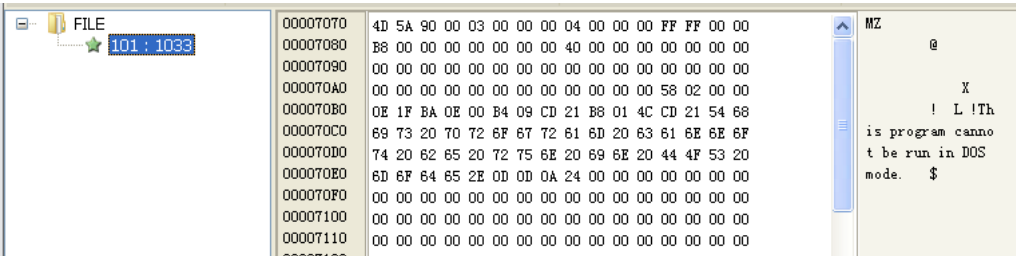


可以看见这个程序里有对服务的创建、打开等一套操作，还有对文件的写操作等。

然后用PEView看一下，发现如下导入函数，与服务、文件和资源有关，推测该程序会创建、 启动或者操作服务，创建和写文件，并且还会对资源节做一些操作。

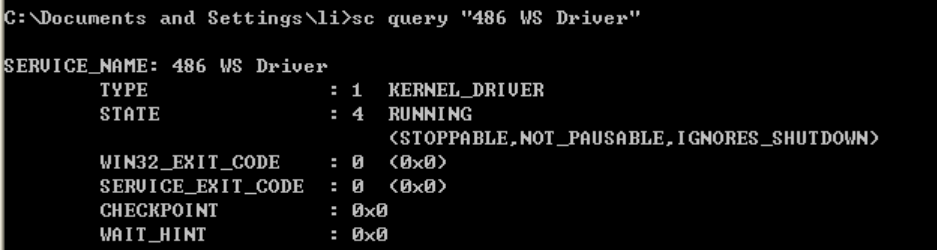


于是我们使用Resource Hacker来检查资源节，发现这里有一个FILE，里面包含了一个PE头。



接下来进行基础动态分析。双击运行程序，发现它打开一个命令行窗口后立即退出。使用procmon监测到它修 改了注册表，在C:\Windows\System32目录下创建了一个文件，增加了一个名为486 WS Driver的服务。然 而我们到上述目录下却没有找到程序创建的文件，且在procmon中并没有关于这个文件被删除的操作，我们推 测这个文件被隐藏了，是一个Rootkit。

接下来，使用命令sc query "486 WS Driver"查询这个服务的状态，可以看到，这个服务仍然在运行，但是 它没有在硬盘上。现在，开始使用WinDbg在主机上调试。



**问题二**

|  |
| --- |
| 这个程序有内核组件吗? |

**有，这个程序拥有一个内核模块，存储在这个文件的资源节中，它会生成一个sys文件并把这个sys文件加载到内核中执行。**

使用resource hacker可以把资源节里的内容提取出来并保存 由于前2个字节是4D 5A，我们将其保存为.exe文件，并用IDA进行打开



打开以后我们可以看见这个的入口是DriverEntry，也就是说这个就是驱动程序了。 那么综上所述，Lab10-02.exe有一个内核模块，这个模块被存放在程序的资源节中，然后这个驱动程序 会被写入到硬盘作为一个服务加载到内核。

**问题三**

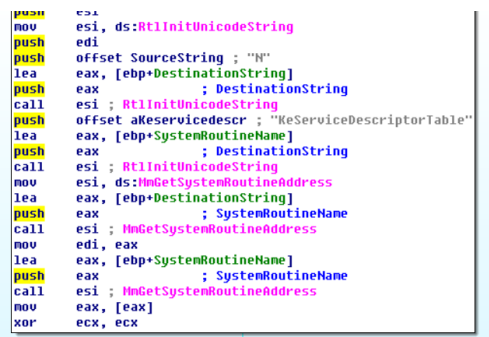
|  |
| --- |
| 这个程序做了些什么? |

**通过上面的分析，我们知道，这个程序是用来隐藏文件的RootKit，它使用SSDT来挂钩覆盖 NtQueryDirectoryFile函数，通过一些比较和移动指针的操作，来隐藏以"Mlwx"开头的文件。**

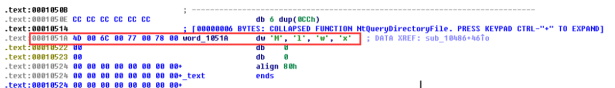
Lab10-02.exe这个程序的目的其实就是为了将驱动程序装载到内核、启动服务，也就是说这个程序的主要功能模块是在驱动程序中，所以这里具体分析一下驱动程序都干了什么



可以看到这个驱动程序主要功能位置是sub\_10706函数，进入到这个函数体内部



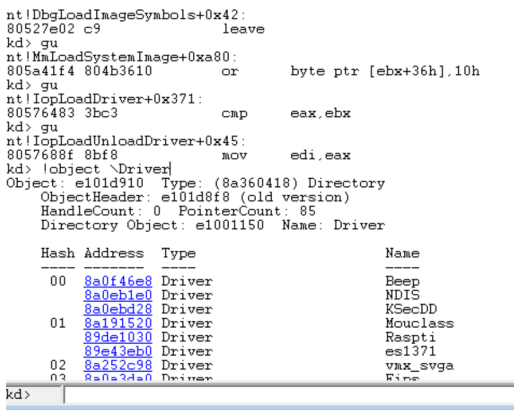
恶意软件调用MmGetSystemRoutineAddress 以获取指向 NtQueryDirectoryFile 和 KeServiceDescriptorTable 子例程的指针。然后它遍历服务描述符表，寻找NtQueryDirectoryFile 的地 址。一旦找到，它将用恶意钩子（自定义子程序）来覆盖地址。 在驱动程序中，NTQueryDirectoryFile 函数被调用 根据 MSDN，此函数返回有关给定文件句柄指定的目录中的文件的各种信息。 再往下，我们可以看到对 RtlCompareMemory 的调用。文件名和&word\_1051A 处的字符串之间进行 了比较。如果匹配，则隐藏文件。&word\_1051A 指向的地址内容如下：



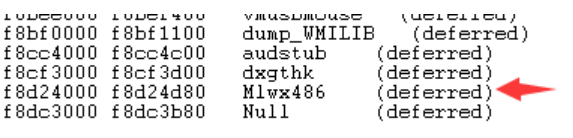
所以说&word\_1051A 字符串就是“Mlwx“。要查看所有这些 win 操作，需要启动 Windbg 并将其附加到 内核。 使用命令 dps nt!KiServiceTable l 100 列出服务描述符表。此表尚未被篡改：



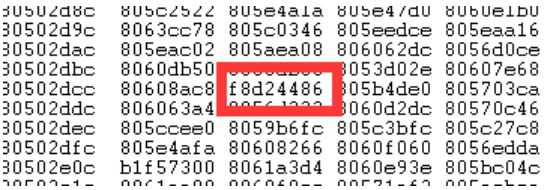
使用 bu Mlwx486!DriverEntry 命令设置断点。运行 Lab10-02.exe，windbg会中断。在 nt!IopLoadDriver+0x66a 处设置断点并让程序再次运行。一旦内核中断，就可以运行 !object \Driver 列出加载的驱动程序。恶意代码的 DriverInit 在此阶段尚未被执行，因此可以在此设置断点



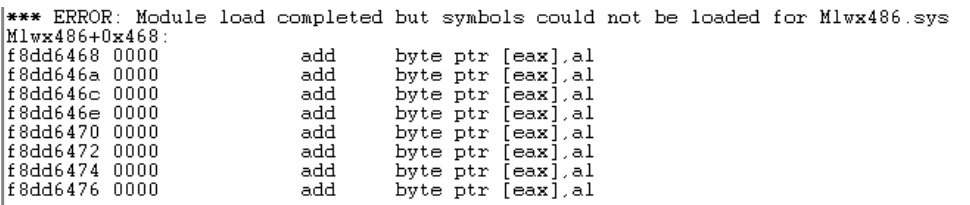
使用lm命令列举加载的所有模块，可以看到Mlwx486已经被加载到内存中了，但是并没有在硬盘上显 示，说明这可能是一个Rootkit。



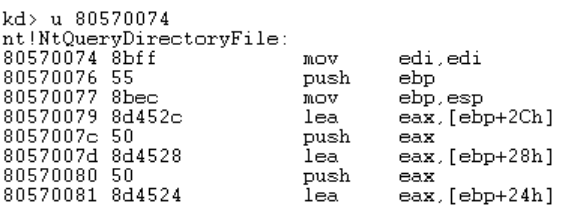
接下来，使用命令dd dwo(KeServiceDescriptorTable) L100检查SSDT的修改情况，可以看到其中有一条 记录在Mlwx486.sys的范围内。



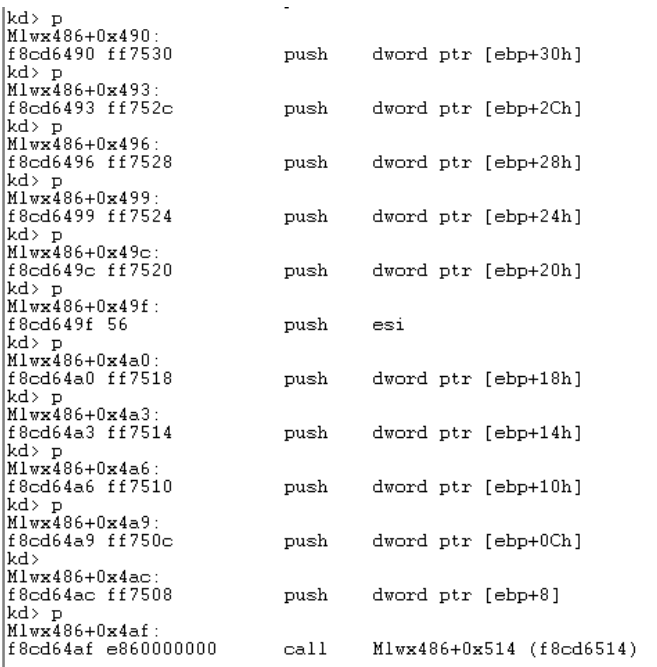
使用命令u f8d24486查看该地址，这是一个Mlwx486里面自带的函数，并不能提供什么有效信息。



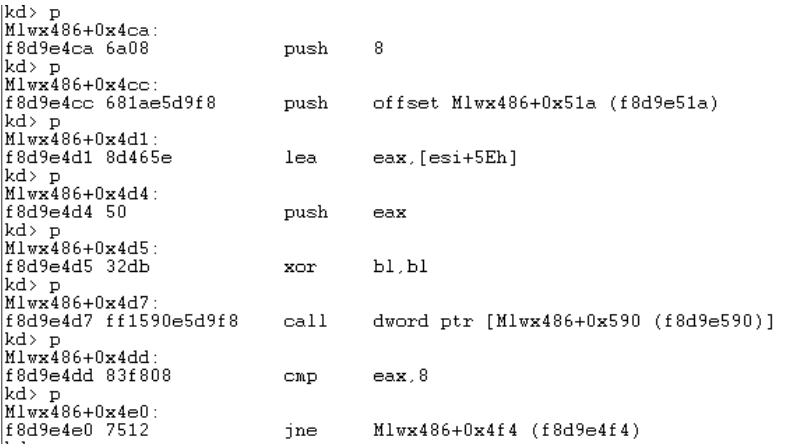
于是通过快照将虚拟机恢复到安装Rootkit之前的状态，再次使用命令dd dwo(KeServiceDescriptorTable) L100，找到修改前的地址为 80570074，于是使用命令u 80570074查看被覆盖的是哪个函数，是 NtOueryDirectoryFile，这是一个提取文件和目录信息的通用函数。Rootkit可以通过挂钩这个函数来隐藏文 件，下面我们开始分析这个函数做了什么。



我们现在这里设置一个断点bp f8d24486，然后回到虚拟机中运行程序，命中断点后又回到WinDbg开始单步调试。（下面由于重启过虚拟机，改变了地址）



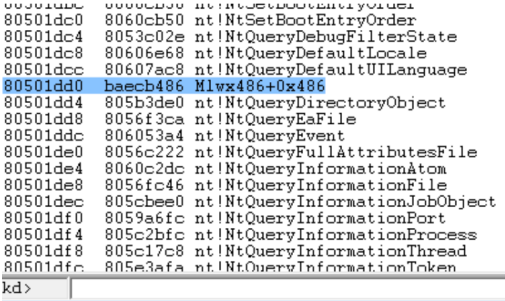
到这里，调用了函数Miwx486+0x514，它的第8个参数FileInformationClass的值是3，如果不是3，便会返 回NtQueryDirectoryFile的原始返回值。此外，挂钩函数也会检查NtQueryDirectoryFile的返回值与第9个 参数值ReturnSingleEntry。当参数不符合要求时，它的功能会与原始函数一样，只有当参数满足要求时才会 修改返回值。 接下来我们使用命令bp f8cd6486 ".if dwo(esp+0x24)==0 {} .else {gc}"在挂钩函数上设置一个条件断 点，仅当参数满足ReturnSingleEntry为0时这个断点才被触发。然后使用命令dir C:\WINDOWS\system32来 列出文件夹C:\WINDOWS\system32这个文件夹下面的所有文件和文件夹，发现这个条件断点被命中了。 之后，它调用了0xf8d9e590处的函数，IDA将其标注为RtlCompareMemory，查看它的参数，它要将eax要和字 符串"Mlwx"进行比较，比较的最大长度为8。而eax中存储的是[esi+5Eh]，在前面我们已经看到esi是 FileInformation，而且这个值是为3，具体意义是FileBothDirectoryInformation，偏移量Ox5E是结构 FileName的起始地址，这段代码用于比较每个文件的文件名开头的四个字节是否是"Mlwx"



使用命令db esi+5eh查看存储在esi+5eh的内容，这里是xs.l。如果传入的FileName和Mlwx不相等，函数直 接就退出了。

现在我们要分析它是如何修改NtQueryDirectoryFile的返回值然后隐藏Mlwx486.sys文件的。 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，如果RtlCompareMemory返回值是8的话，即找到了要隐藏 的"Mlwx"开头的文件，就会通过两次指令操作把指针往后移动，抹除了Mlwx486.sys文件的 FILE\_BOTH\_DIR\_INFORMATION结构，达到隐藏文件的目的。

而后再次运行 kd> dps nt!KiServiceTable l 100 ，能够看出服务描述符表已被修改



**LAB10-3**

**问题1**

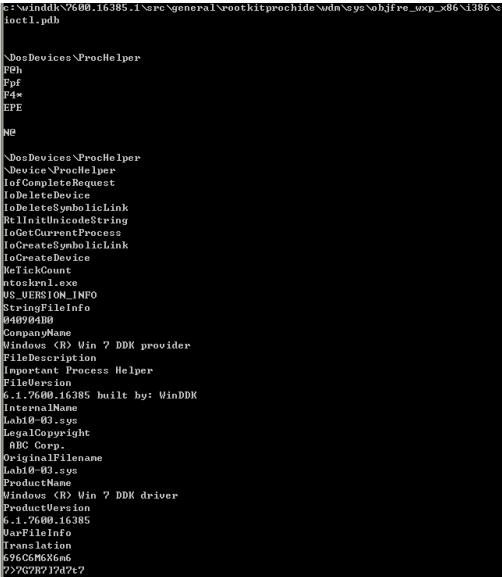
|  |
| --- |
| 这个程序做了些什么? |

**双击运行这个程序，每隔 30 秒就弹出一个广告页面。进行分析后我们知道，这个程序会在用户态程序加载驱 动，这个驱动摘除PEB中自己的\_LIST\_ENTRY来隐藏进程。**

首先使用strings工具简单查看一下exe文件

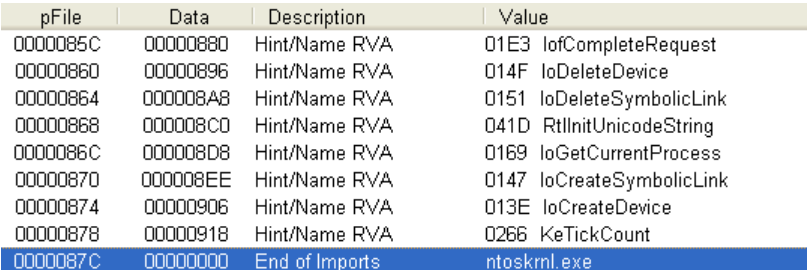


可以看见这个程序里有对服务的创建、打开等一套操作，同时还有一些以前没有见过的函数，比如获取 环境中的字符串；然后可以看到有一个http的网址，结合之前的创建服务，猜测服务中可能会包含有对这个URL的访问。 再查看一下sys文件

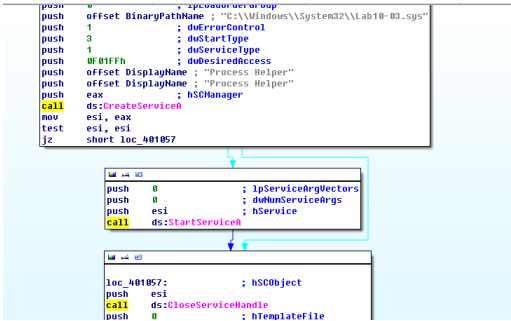


可以看见在sys文件中出现了InternetName字符串，结合之前exe中的猜测，想来可能是会进行访问。 其他的从这里也看不出来什么了。

继续对这两个文件进行基础静态分析。Lab10-03.exe的导入函数与前面的基本一致，有关于创建、启动或者 操作服务，创建和写文件的函数。看到Lab10-03.sys有如下导入函数，从IoGetCurrentProcess可以看出这 个驱动在修改正在运行的进程，或者需要关于进程的信息。

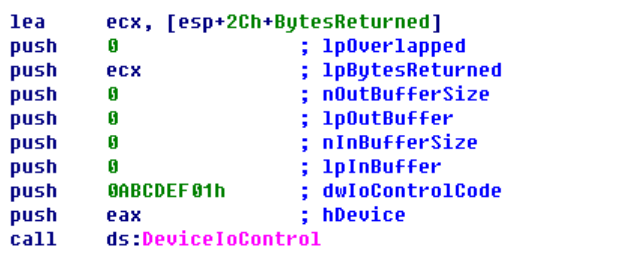


接下来，双击运行可执行程序，进行基础静态分析。我们看到它会不断地弹出一个广告网页，与在Lab7-2中看 到的相同，但是在任务管理器和Process Explorer中根本没有这个进程。于是我们只能通过恢复快照的方式 “关闭”这个进程。 之后用IDA和OD打开文件，准备进行高级静态分析。

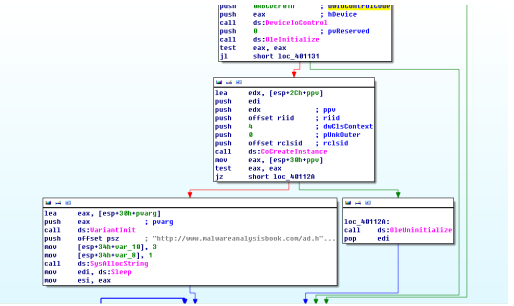


我们从WinMain函数开始分析。

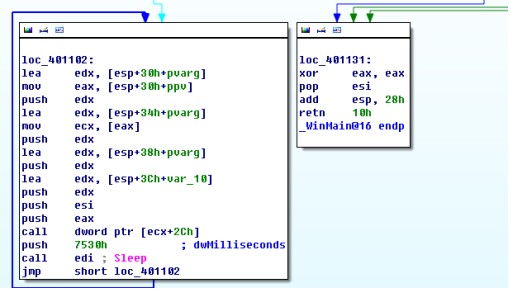
首先调用了函数OpenSCManagerA来打开服务控制，说明程序打算在这里操作服务。之后调用函数 CreateServiceA创建一个服务，查看其参数，BinaryPathName的值是驱动文件的路径 C:\\Windows\\System32\\Lab10-03.sys，用于指明服务的二进制文件的位置；其他参数分别表明了这是个 驱动服务，且服务的名字是Process Helper。调用函数StartService开启这个服务，驱动Lab10-03.sys会 被加载到内核中。 然后调用函数CreateFileA创建了一个路径为\\.\ProcHelper的文件，并作为一个句柄打开。调用函数 DeviceIoControl，分析它的参数，lpInBuffer和lpOutBuffer为0，意味着这个请求没有发送任何消息到内 核驱动中，并且内核驱动也没有给出任何的反馈；参数dwIoControlCode的值是0ABCDEF01h，一看就是人工 设置的。



可以看到前面和之前一样是创建一个服务，服务的名字是Process Helper，并且设置的驱动文件路径和 之前lab 10-01是同一个目录下。之后是熟悉的打开服务。 接下来的内容和之前就有些不一样了



可以看见这里压入了之前分析的url，然后会打开一个网页，之后调用sleep



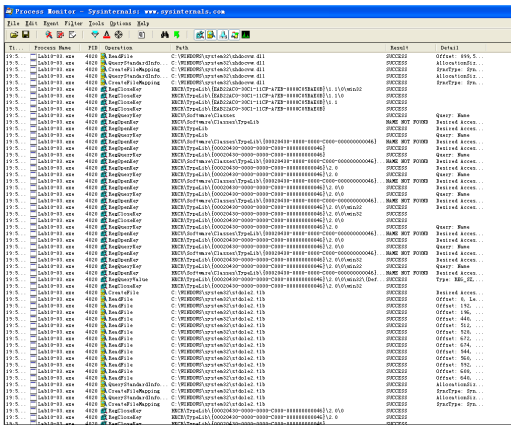
可以看见这里是一个循环，每次sleep时间是0x7530h ms 经过资料查询，可以看到有个DeviceIoControl函数的调用，这个函数有一个比较特殊的地方就是作为参 数传递给她的输入和输出参数，将会被发送到内核中去，所以需要特别关注一下这个函数的参数。



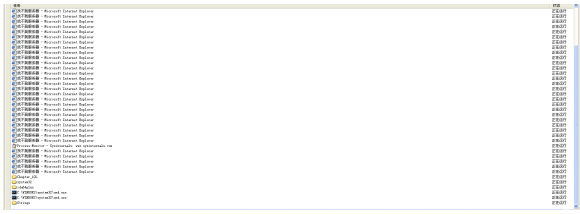
可以看见参数中LpOutBuffer和LpInputBuffer都被设置为空，然后在dwIoControlCode这里的参数也是 有点特殊，传递进去的是0ABCDEF01h，具体作用在内核位置再进行分析。 尝试使用Procmon查看一下这个程序具体的影响 首先在双击运行的时候，可以看见弹出了一个浏览器窗口



观察可以发现打开的就是之前我们在strings中看见的url，同时可以注意到每过一段时间都会打开一个这个页面，也就是上面分析的30s。 查看Procmon中的结果，可以看到有很多对注册表、文件的操作



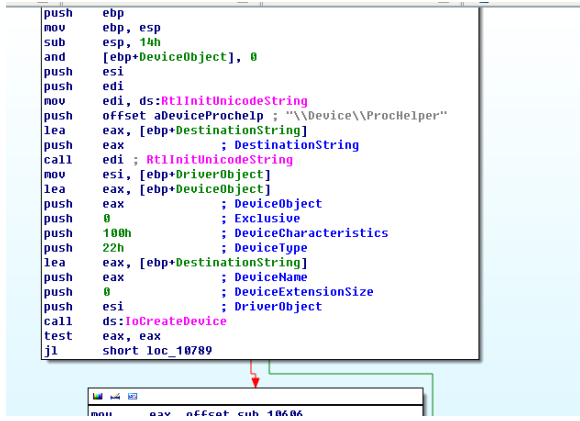
由于这个程序是会sleep，然后每隔一段时间就打开一次浏览器，所以在进程管理器里查看一下他的常驻进程名是什么



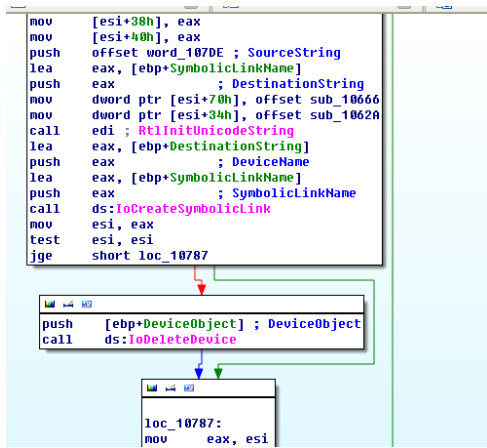
可是意外的是，在进程管理器里并没有看见关于这个进程的信息，也就是说想要从进程管理器里关闭这个进程从而达到关闭浏览器弹出是不可行的。具体是如何隐藏进程信息的在之后会进行分析。

IDA分析驱动

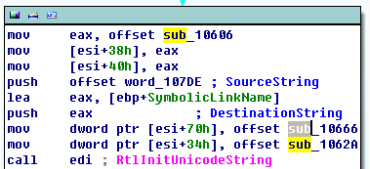
和之前一样，可以看见这个驱动真正的入口点不是在一开始的Entry处，而是在jmp后面的地址，直接跳转到那个地址进行分析



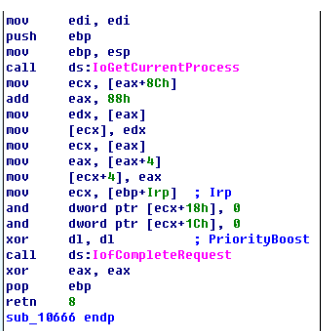
可以看见驱动调用了一个函数IOCreateDevice，创建了一个名为\Device\ProcHelper的设备



之后在下面调用了IoCreateSymbolicLink来创建了一个名为\DosDevices\ProHelper的符号链接，来提 供用户态的应用程序访问.。 同时我们还可以看见调用了一个RtlInitUnicodeString函数，这个函数有两个参数，其中第一个参数是这 个驱动的名字： DosDevices\ProcHelper ，第二个函数是eax中的值，从边上的注释可以看出这个参数 是DestinationString 最后一个函数是IoDeleteDevice，这个函数就是用来删除驱动程序。 注意到以下代码：



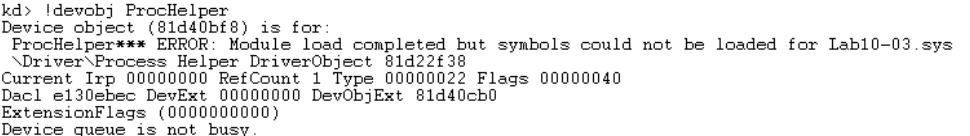
这里将3个函数的偏移地址放到了内存中，其中第一个位置将 MajorFunction [IRP\_MJ\_DEVICE\_CONTROL]设置为 0x10666 根据 MSDN，IRP\_MJ\_DEVICE\_CONTROL 请求，是由 I / O 管理器、其他操作系统组件或其他内核模式 驱动程序发送的。通常，此 IRP 代表已调用 Microsoft Win32 DeviceIoControl 函数的用户模式应用程 序或代表已调用 ZwDeviceIoControlFile 的内核模式组件 注意到10666位置处的函数获取了当前进程，猜测这里和隐藏进程有关，主要分析一下这一段的代码



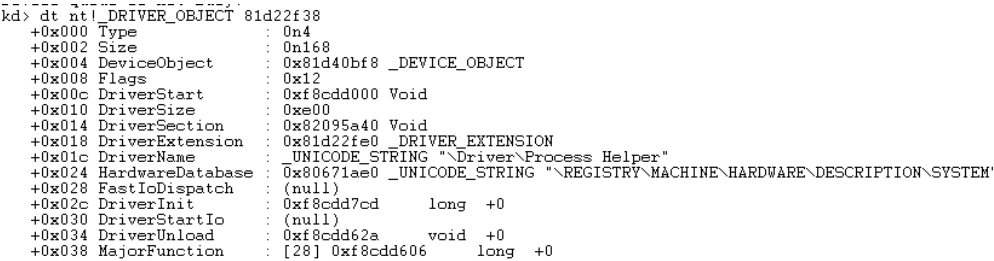
mov ecx, [eax+8Ch]指令获取列表中指向下一项的指针。 mov edx, [eax]指令获取列表中指向前一项的指针。 mov [ecx], edx 指令覆盖下一项的 BLINK 指针，使其指向前一项，也就是 mov [ecx], edx 指令。在此之前，下一项的 BLINK 指针指向当前项 mov ecx, [eax]。mov [ecx], edx 会 覆盖 BLINK 指针，从而使它跳过当前指针。 mov ecx, [eax]; mov eax, [eax+4]; mov [ecx+4], eax;指令执行相同的步骤，除了覆盖列表中前一项的 FIINK 指针来跳过当前项。 除了修改当前进程的 EPROCESS 结构之外，上述代码还会修改进程链中的前一个或者后一个进程的 EPROCESS 结构。这个六条指令通过从加载进程的列表中解除链接，来隐藏当前进程。其实也就是在获 取进程的时候，将链表的结构断开，然后将左右两边的节点连接起来，从而达到隐藏的效果。

之后我们连接上WinDbg进行内核调试。

首先使用命令!devobj ProcHelper，查看到这个设备存储的位置是81d22f38。



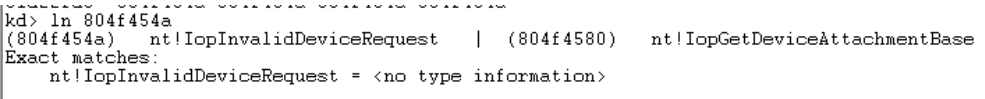
然后使用命令dt nt!\_DRIVER\_OBJECT 81d22f38查看标注的驱动对象，其中，DriverInit是驱动初始化的操 作地址，DriverUnload是驱动卸载时候的操作地址，我们刚刚在IDA分析的时候看到过这个函数。



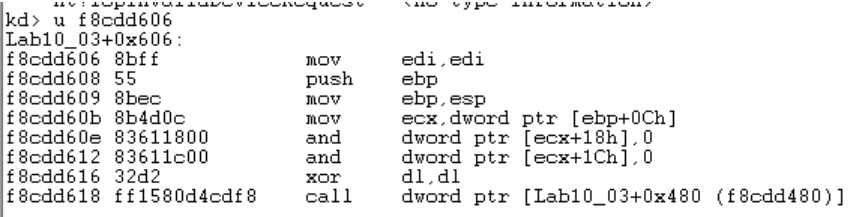
接下来，我们使用命令dd 81d22f38+0x38 L1c命令查看主函数表中表项：



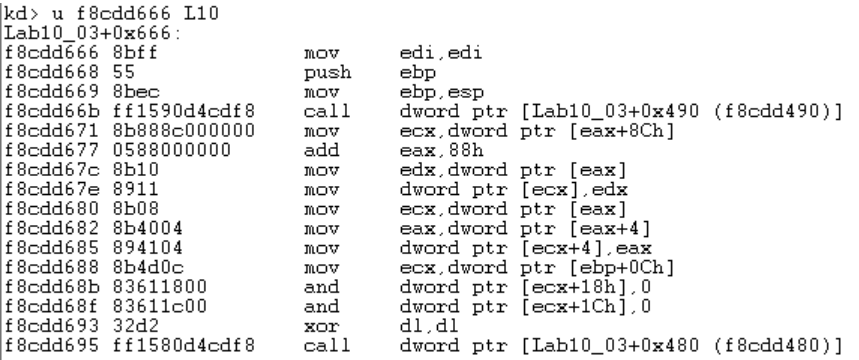
可以看到表中大多数函数都是804f454a，我们使用命令ln 804f454a看看这个函数做了什么，可以看到，这个 函数被命名为IopInvalidDeviceRequest，从字面意思看是处理驱动无法处理的非法请求的。



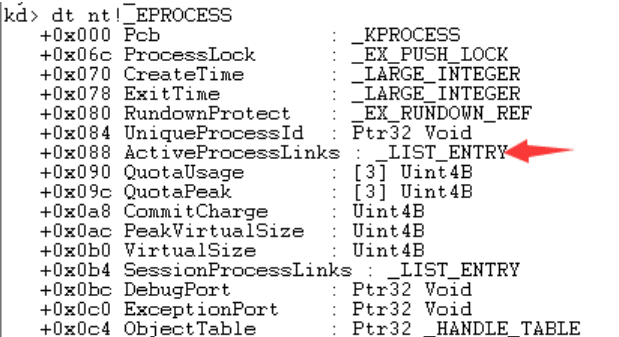
此外，主函数表中还有三个不同于804f454a的函数，分别是f8cdd606、f8cddd606和f8cdd666，它们在 896587e0+0x38处的偏移量分别是0，2和0xe，对应着Create、Close和DeviceIoControl。进入这两个（前 两个相同）位置查看具体的代码。 在f8cdd606处，看到它只调用了一个函数，指向的是804ef1b0这个地址，使用指令ln 804ef1b0，看到这个 函数是IofCompleteRequest，用于指示调用者已完成给定I/O请求的所有处理，并将给定的IRP返回给I/O管理 器，意味着告诉操作系统请求这个驱动成功了，但是没做什么其他的操作。



在f8cdd666处看到该函数在初始化栈操作之后，首先调用了一个在dword ptr [Lab10\_03+0x490]的指针函数， 指向的是804ef608这个地址，使用指令ln 804ef1b0，看到这个函数是nt!IoGetCurrentProcess，用于返回 一个指向当前进程的指针。也就是这个代码会获得一个当前进程的指针，操纵当前进程的PEB。



之后，这个函数访问偏移量0x88处的数据和偏移量0x8C处数据。我们使用dt命令发现这两处数据是PEB结构中 的\_LIST\_ENTRY。这个结构用于描述双向链接列表中的条目或充当此列表的头部，该程序不仅读取LIST\_ENTRY 结构，还会修改它，通过一系列对指针的操作修改PEB，使得没有任何的指针将指向他的LIST\_ENTRY结构，把 自己的进程隐藏起来。



**问题2**

|  |
| --- |
| 一旦程序运行，你怎样停止它? |

只能通过重启来停止。

**问题3**

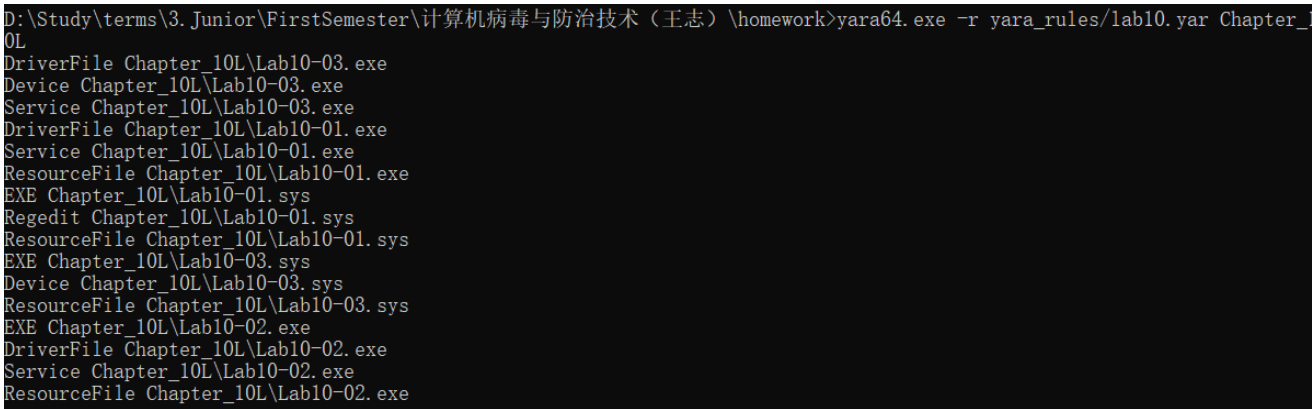
|  |
| --- |
| 它的内核组件做了什么操作? |

内核组件通过调用函数DeviceIoControl修改进程链接表的结构，隐藏自己的LIST\_ENTRY，进而把进程隐藏起来。

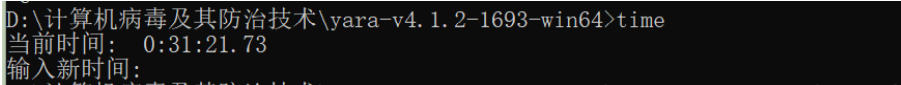
**YARA规则撰写**

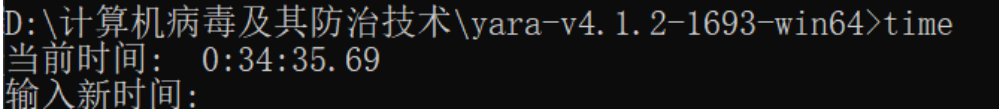
|  |
| --- |
| import "pe"  rule UrlRequest {  strings:  $http = "http"  $com = /[a-zA-Z0-9\_]\*.com/  condition:  $http or $com }  rule EXE {  strings:  $exe = /[a-zA-Z0-9\_]\*.exe/  condition:  $exe }  rule Regedit {  strings:  $system = "Registry"  $software = "SOFTWARE"  condition:  $system or $software  }  rule DriverFile {  strings:  $name = ".sys"  condition:  $name }  rule Device {  strings:  $name = "Device"  condition:  $name }  rule Service {  strings:  $create = "CreateService"  $start = "StartService"  condition:  $create or $start  }  rule ResourceFile {  strings:  $name = ".rsrc"  condition:  $name } |

结果如下所示：



然后我们得到如下的统计结果：





可以得到大致是三分钟时间左右。

我们同时编写一个python脚本用来统计时间：

|  |
| --- |
| import os, time # output= subprocess.check\_output([r'C:\Users\Despacito\Desktop\yara64.exe',r'C:\Users\Despacito\Desktop\demo\_regedit.yara',r'C:\Users\Despacito\Desktop\virus']) #得到bytes类型的，需要转换成为str类型，来使用str的统计函数count begin\_time = time.time() os.system(r"yara64.exe -r rules.yar C:\\") end\_time = time.time() print(end\_time - begin\_time) |

得到如下输出结果：

|  |
| --- |
| 307.204055070877075 |

可以看出我们的yara规则还是很快的

**IDAPython**

首先对某字符串进行搜索，找到后返回字符串地址：

|  |
| --- |
| print hex(FindBinary(MinEA(),SEARCH\_DOWN,'HGL345')) print hex(FindBinary(MinEA(),SEARCH\_DOWN,'http://www.malwareanalysisbook.com')) |

从当前地址查找第一个指令并返回指令地址，从当前地址查找第一个数据项并返回数据地址。

|  |
| --- |
| print hex(FindCode(MinEA(),SEARCH\_DOWN))  print hex(FindData(MinEA(),SEARCH\_DOWN)) |

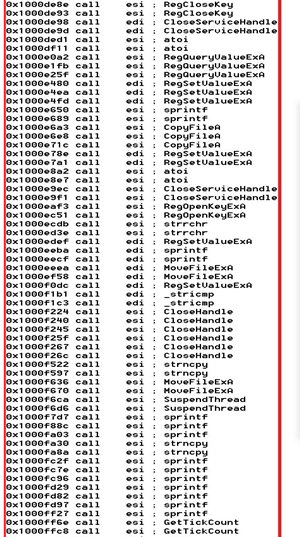
获取代码段中的所有函数、函数中的参数、函数名及函数中调用了哪些函数。

|  |
| --- |
| **for** **seg** **in** Segments():   #如果为代码段  **if** SegName(**seg**) == '.text':  **for** function\_ea **in** Functions(**seg**,SegEnd(**seg**)):  FunctionName=GetFunctionName(function\_ea)  print FunctionName  nextFunc=NextFunction(function\_ea)  print nextFunc |

遍历所有函数，并查找所有对每个函数执行的调用，引用将存储在两个字典中。

|  |
| --- |
| from sets import Set ea=ScreenEA() Par=dict() son=dict() for fun in Functions(SegStart(ea),SegEnd(ea)): #遍历函数  f\_name=GetFunctionName(fun)  Par[f\_name]=Set(map(GetFunctionName,CodeRefsTo(fun,0))) #创建一个集合，其中包含调用（引用）的所有函数的名称  for fun\_son in CodeRefsTo(fun,0): #遍历所有的引用  fname\_son=GetFunctionName(fun\_son) #获取引用函数的名称  son[fname\_son]=son.get(fname\_son,Set())  son[fname\_son].add(f\_name); #将当前函数添加到函数列表中  functions=Set(Par.keys()+son.keys()) #获取所有函数的列表 for per in functions:  print "%d %s %d" % (len(Par.get(per,[])),per,len(son.get(per,[]))) |

得到如下的部分结果



**四、实验心得**

**本次实验，我熟悉了对恶意代码分析工具有了更深入的理解。同时得到了充分的锻炼，尤其是本次实验对WINDbg的使用，也加深了我对内核调试部分的理解和相关知识的掌握。在实验过程中，通过亲手分析恶意代码，我收获了很多，过程是非常快乐的。最后，我对本门课程的实验开始得心应手，做实验的速度越来越快了。**