ZeroAccess恶意软件分析

目录

[ZeroAccess恶意软件分析 1](#_Toc29460182)

[1. 概述 3](#_Toc29460183)

[2. 详细分析 3](#_Toc29460184)

[2.1 ZeroAccess.exe 分析 3](#_Toc29460185)

[2.1.1 反调试 3](#_Toc29460186)

[2.1.2 构建编码导出表 4](#_Toc29460187)

[2.1.3 释放可执行文件 6](#_Toc29460188)

[2.1.4 劫持系统调用 7](#_Toc29460189)

[2.2 受感染的DLL分析 9](#_Toc29460190)

[2.2.1 线程注入 9](#_Toc29460191)

[2.2.2 随机选择驱动模块 11](#_Toc29460192)

[2.2.3 注册表写入 14](#_Toc29460193)

[2.2.4 禁用文件保护服务 14](#_Toc29460194)

[2.2.5 感染驱动文件 15](#_Toc29460195)

[2.3 受感染的驱动分析 18](#_Toc29460196)

[2.3.1 劫持磁盘驱动程序 18](#_Toc29460197)

[2.3.2 再次感染驱动文件 21](#_Toc29460198)

[2.3.3 TDI网络行为 22](#_Toc29460199)

[2.4 配置Rookit 25](#_Toc29460200)

[2.4.1 恢复线程 25](#_Toc29460201)

[2.4.2 创建虚拟驱动器 25](#_Toc29460202)

[2.4.3 隐藏驱动器文件存储恶意文件 25](#_Toc29460203)

[2.4.4 TDI配置文件 26](#_Toc29460204)

[2.4.5 注册表存储解密密钥 26](#_Toc29460205)

[2.4.6 创建HTTP请求相关文件 27](#_Toc29460206)

[2.4.7 创建恶意文件max++.00.x86 28](#_Toc29460207)

[2.4.8 向服务器发消息 29](#_Toc29460208)

[2.5 max++.x86.dll 分析 30](#_Toc29460209)

[2.5.1 隐藏DLL加载 30](#_Toc29460210)

[2.5.2 IAT Hook 30](#_Toc29460211)

[2.5.3 远程加载DLL 31](#_Toc29460212)

[2.5.4 网络行为 32](#_Toc29460213)

[3. 执行流程 35](#_Toc29460214)

[4. IOC 35](#_Toc29460215)

# 概述

ZeroAccess是一种特洛伊木马，它使用Rootkit隐藏自身，禁用安全服务，还可以创建隐藏的文件系统，在受感染的计算机上打开后门，通过僵尸网络中下载恶意组件。一旦系统被感染，木马将会进行比特币挖矿和“点击欺诈”的操作。

ZeroAccess通过多种方式传播：将自身伪装成合法文件或隐藏为额外的有效载荷，欺骗用户执行；将受攻击的网站流量重定向到恶意网站进行分发；通过自身的P2P僵尸网络更新自身。

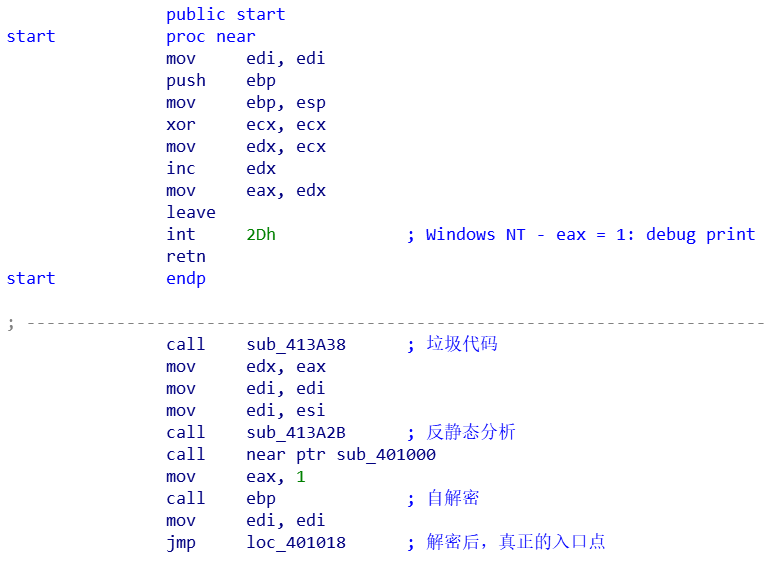
此次分析的样本来源于hybrid-analysis网站，分析时相关域名和服务器已失效，因此部分功能难以分析。

# 详细分析

## 2.1 ZeroAccess.exe 分析

### 2.1.1 反调试

首先，检查木马的入口点代码。这里主要使用了“int 2D”的反调试方法，“int 2D”可以检查调试器（包含ring0和ring3调试器）的存在。“int 2D”指令会触发异常中断，在附加调试器的情况下，异常由调试器处理；否则由程序的结构化异常处理机制处理。这里可以简单地将“int 2D”和其下一条指令“retn”替换成“nop”指令，即可按照正常的程序执行流程运行：



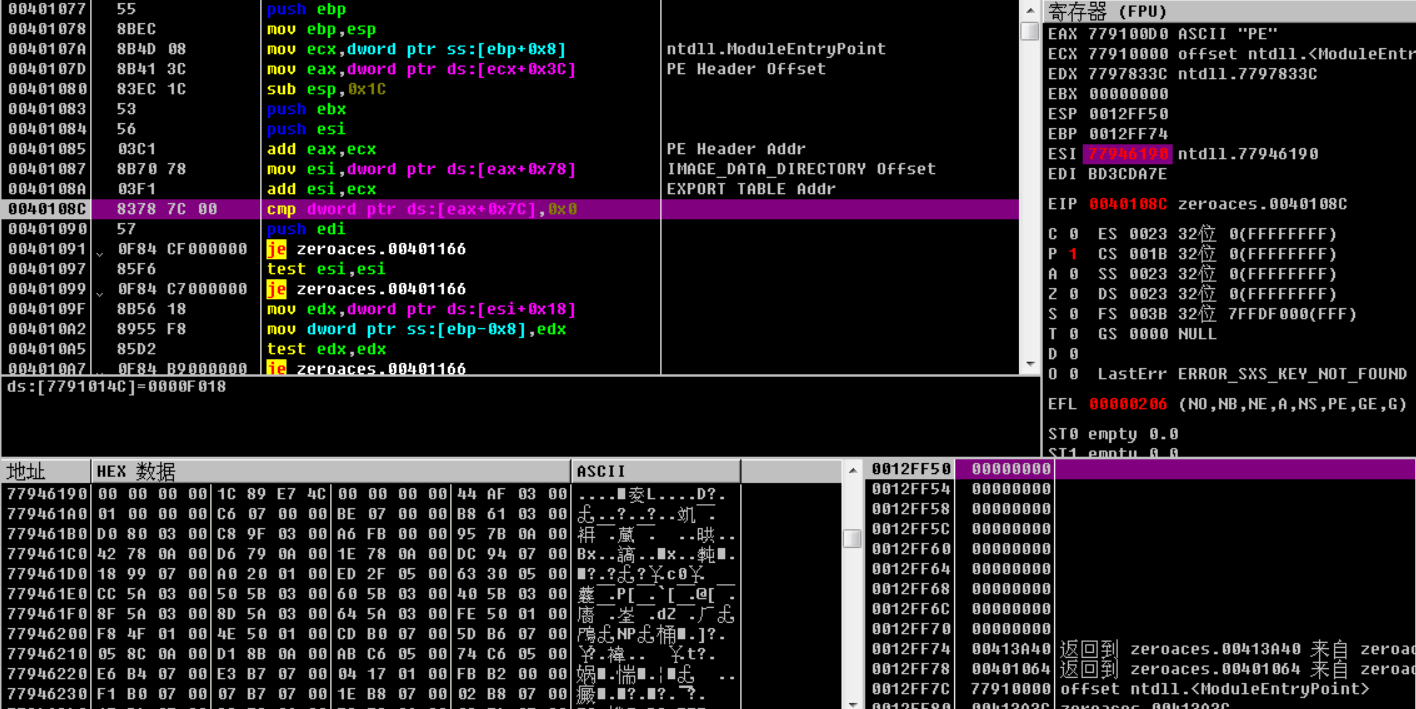
跳过多条“int 2D”指令，执行一段自解密代码，即可进入真正的入口点401018处。

### 2.1.2 构建编码导出表

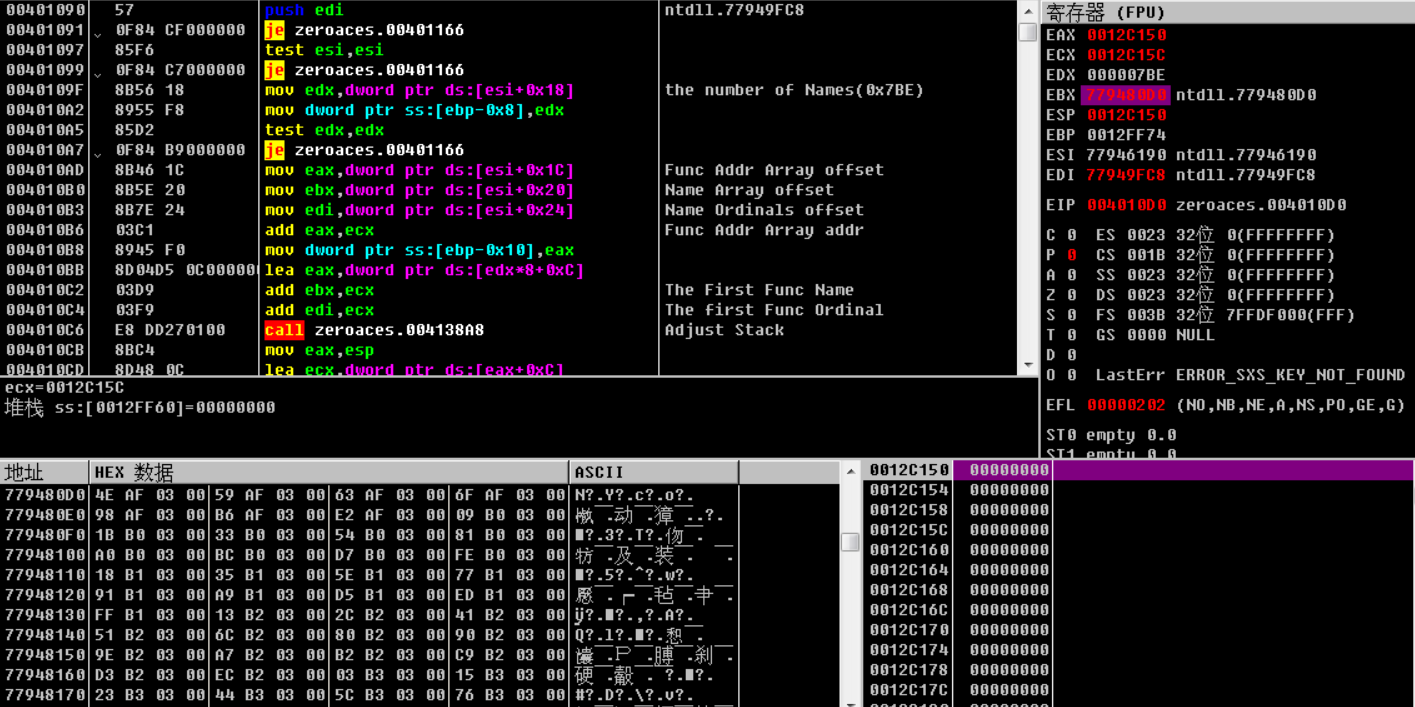
如下图OEP处，这里通过“fs:[0x18]”得到存储的TIB的入口地址TEB，“TEB+0x30”对应于PEB，而“PEB+0xC”对应于\_PEB\_LDR\_DATA结构，“\_PEB\_LDR\_DATA+1Ch”对应于InInitializationOrderModuleList结构，最终指向BaseDllName的缓冲区。所以这部分代码的含义是获取当前进程的DLL名称，这里获取的DLL是ntdll.dll：



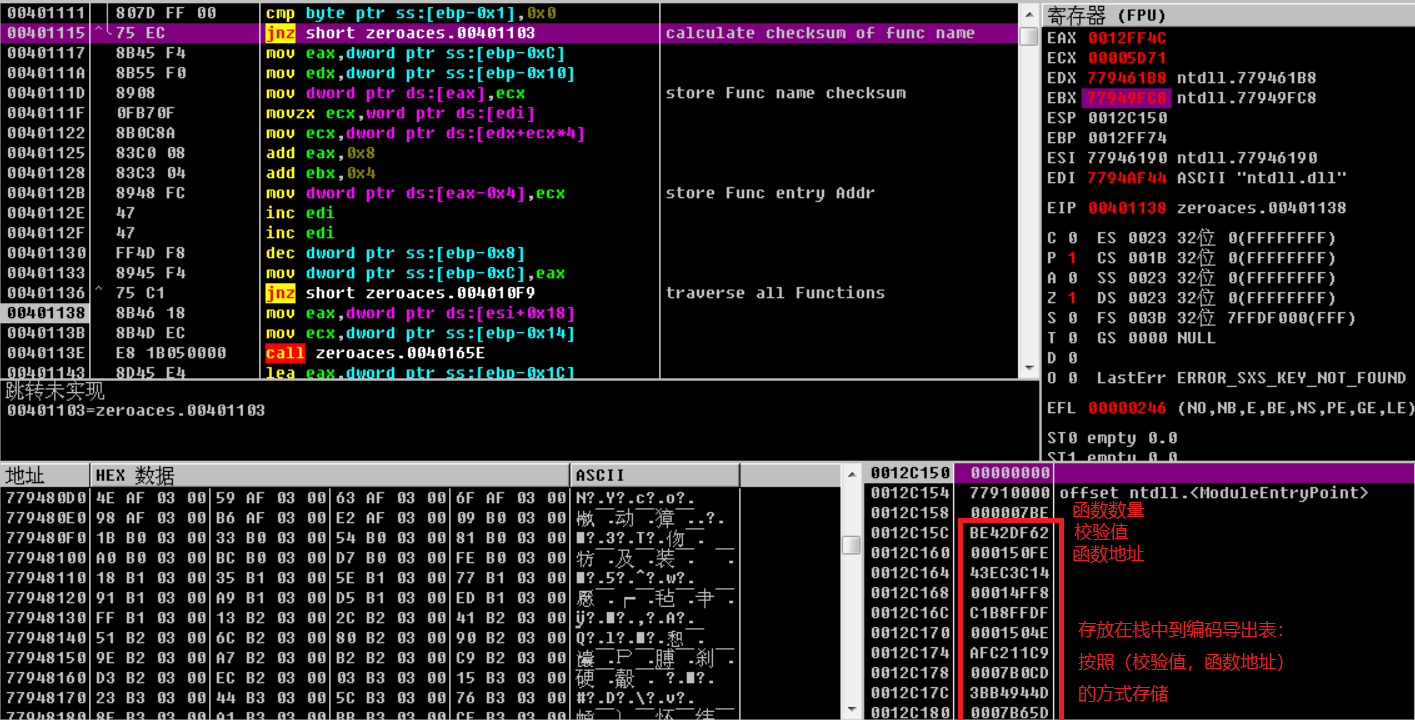
在读取到ntdll.dll后，首先获取该模块的基址，然后获取EXPORT TABLE的起始地址:



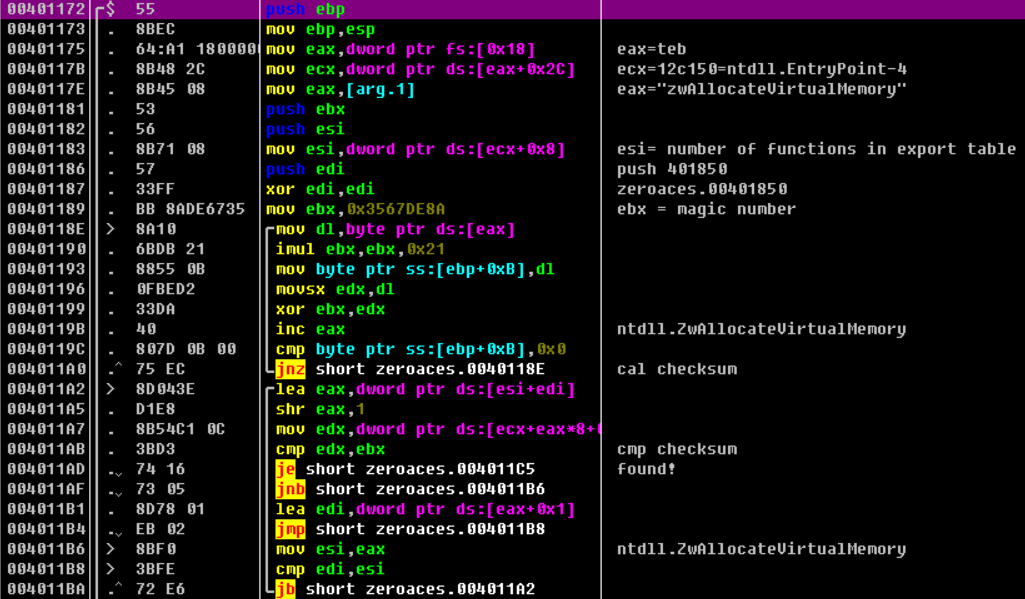
通过导出表，获取到ntdll.dll模块的导出函数数量，存储函数地址数组的地址，名称数组地址，名称序号地址：



接下来，将ntdll.dll模块所有的导出函数计算校验值，并将校验值和函数地址存储在栈中，为自身构建一个编码的导出函数表：

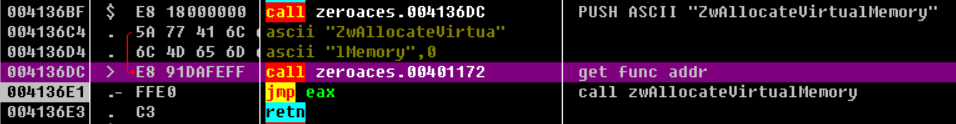


因此，对于ntdll.dll模块中函数的调用，会直接计算其校验值与存储在栈中的函数校验值进行比较，若值相同则取出其函数地址：

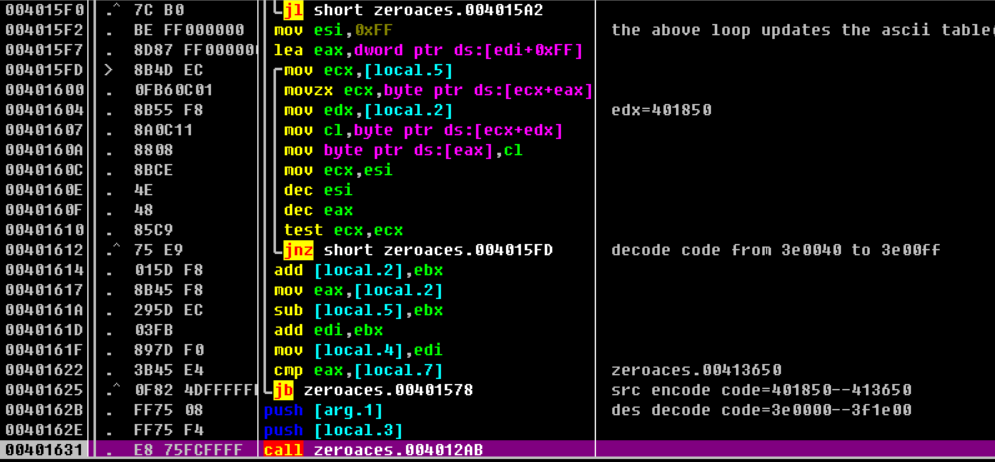


### 2.1.3 释放可执行文件

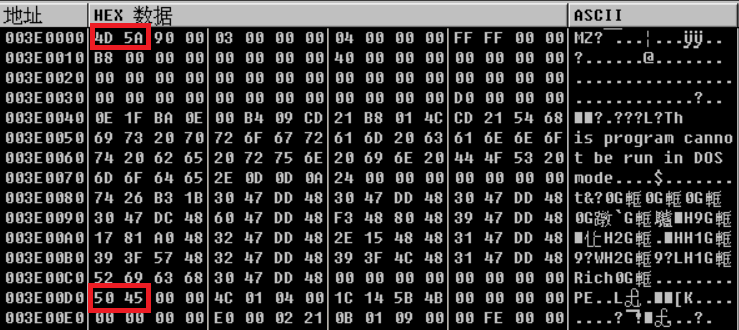
首先是调用ZwAllocateVirtualMemory函数分配内存，通过编码的导出表进行调用：



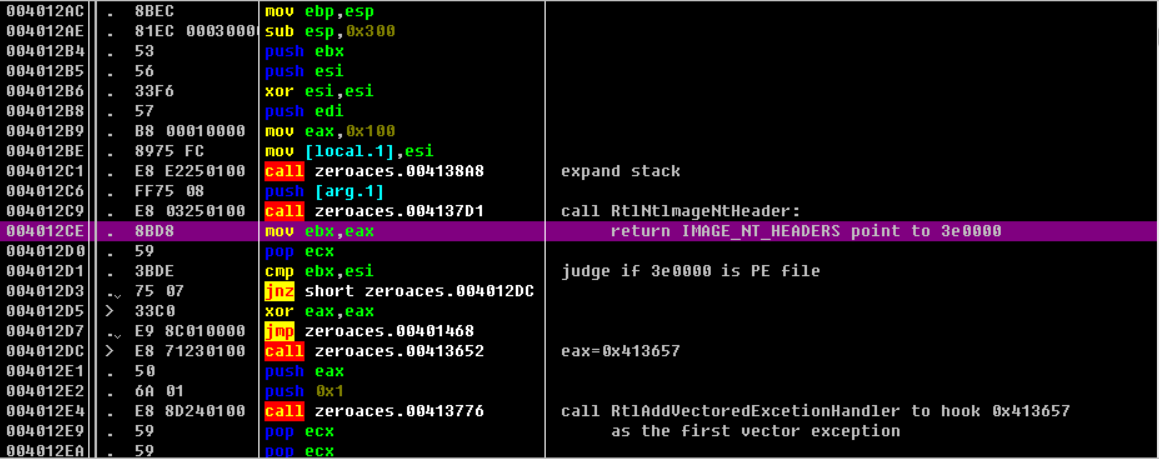
下一步，构造一个解密表，将代码段的数据解密存放到新申请的内存空间中：



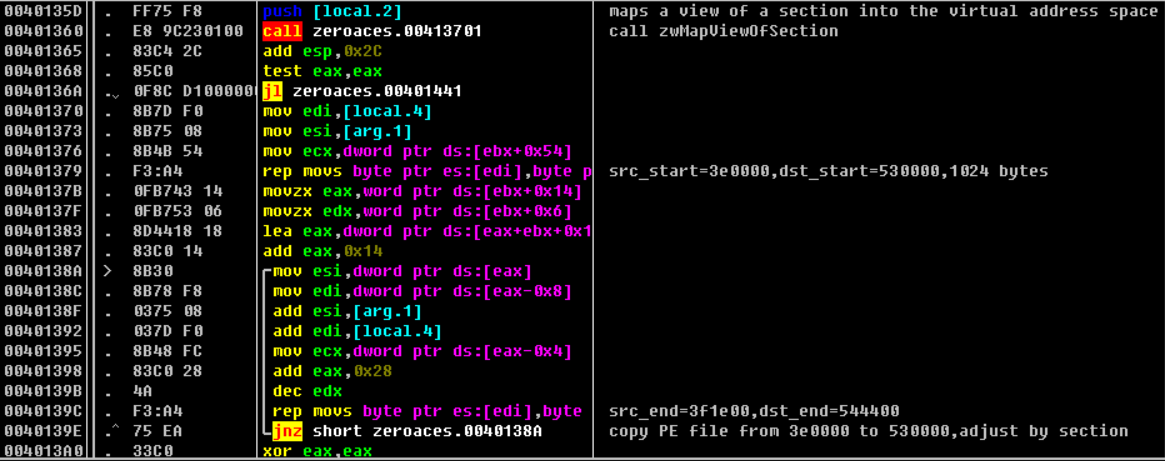
查看内存空间中的数据，发现这是一个可执行文件：



随后，调用RtlImageHeader获得指向新分配内存的IMAGE\_NT\_HEADER指针；调用RtlAddVectoredExceptionHandler函数添加矢量化异常处理程序，异常处理程序位于0x00413657处函数：

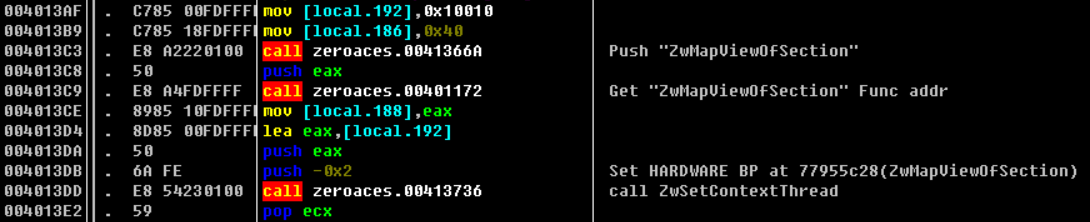


再次申请一块内存空间，将解密后的PE文件拷贝到这块内存中，并根据代码段、数据段等进行调整，使得PE文件头中的偏移量指向准确的段基址：

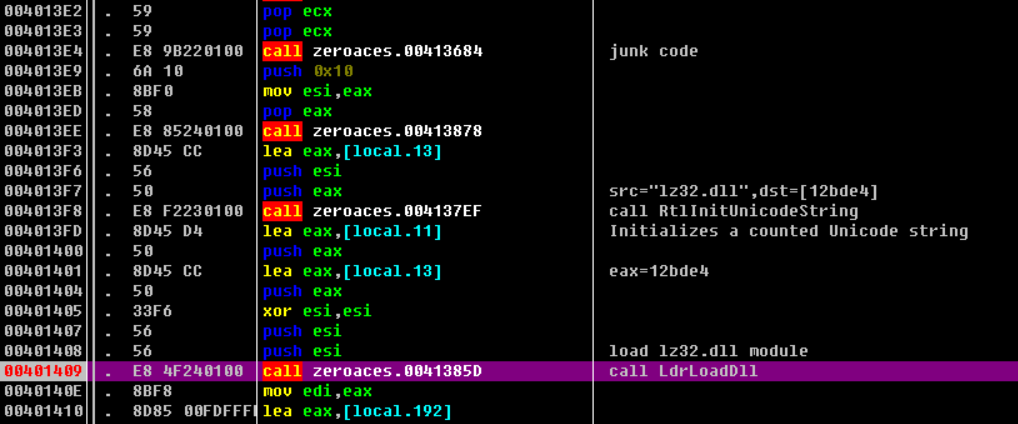


### 2.1.4 劫持系统调用

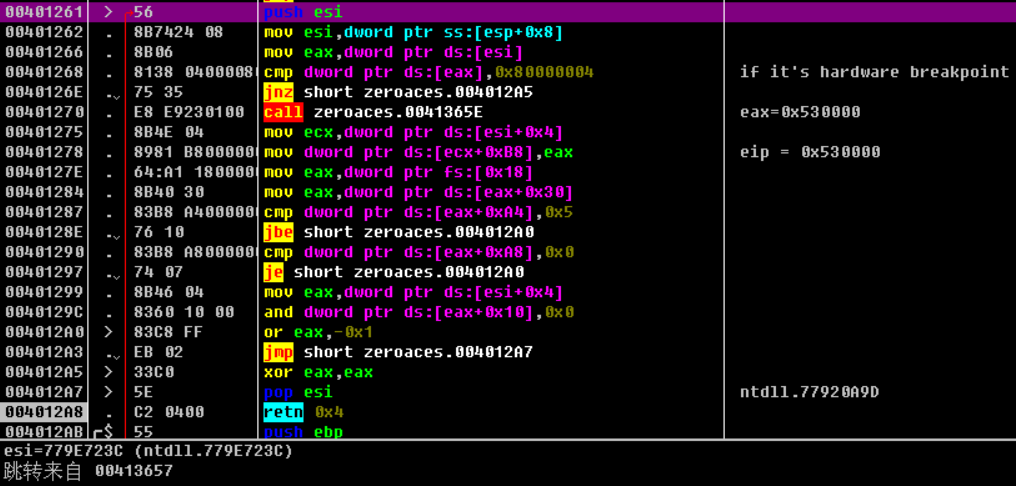
接着调用关键的ZwSetContextThread函数，在0x77955c28处（ZwMapViewOfSection函数）设置一个硬件断点：



下一步是构造Unicode字符串“ lz32.dll”，动态加载这个lz32.dll模块：



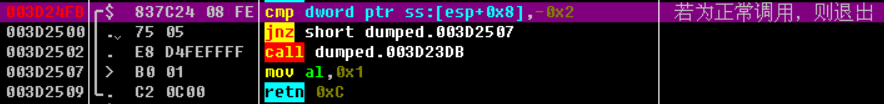
但实际上，由于在ZwMapViewOfSection函数入口点设置了硬件断点，加载过程会被中断，转而执行0x00413657处的异常处理程序。异常处理程序会判断是否为硬件断点异常，若是，则对内存进行一些写入，设置线程上下文中的EIP指向0x550000（解密后调整过的PE文件）。所以，异常处理程序返回后，不会执行lz32.dll的实际代码，而是加载了木马自身的可执行文件：



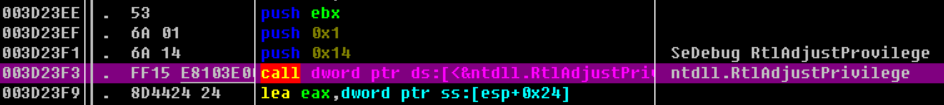
## 2.2 受感染的DLL分析

### 2.2.1 线程注入

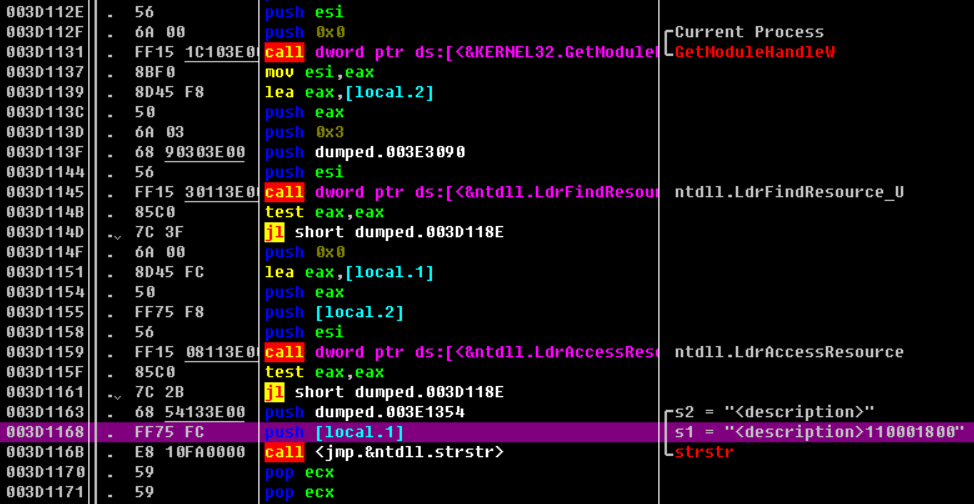
将可执行文件dump下来并修复IAT分析，发现这是个DLL文件。使用LoadDll插件进行动态调试，Dll入口点简单判断若为合法调用，则不执行恶意功能，退出：



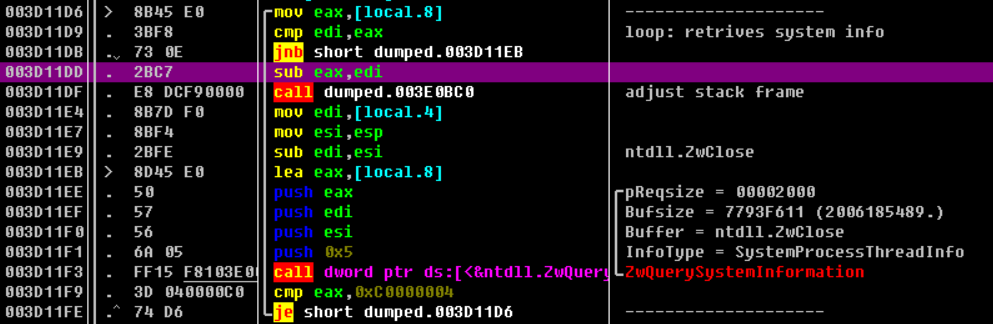
进入模块的功能区域，首先，木马获取了SeDebug特权（该特权可以打开系统中的任何其他进程）：



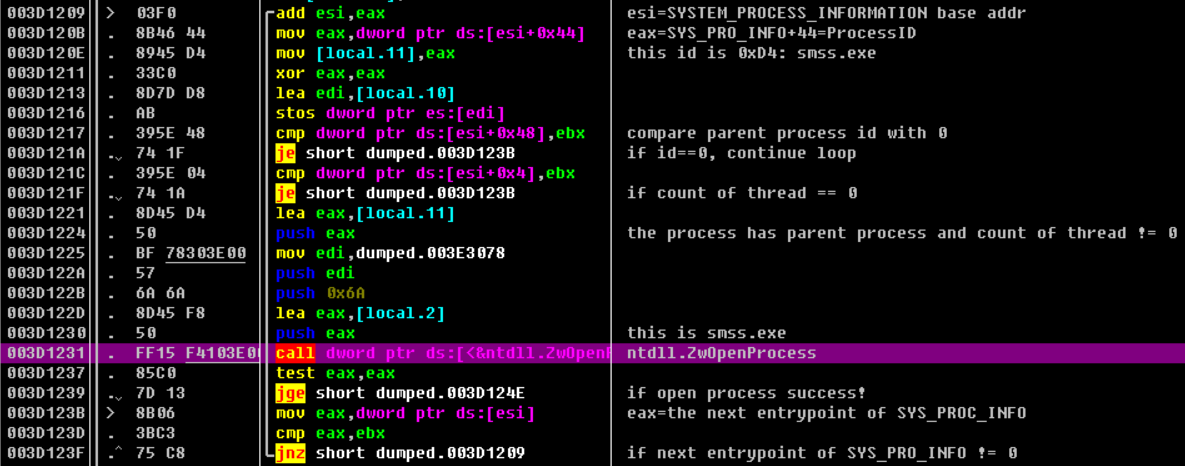
通过获取当前进程的句柄，查找与当前进程相关的资源，进而提取出一个解密密钥“110001800”存储到堆栈中：



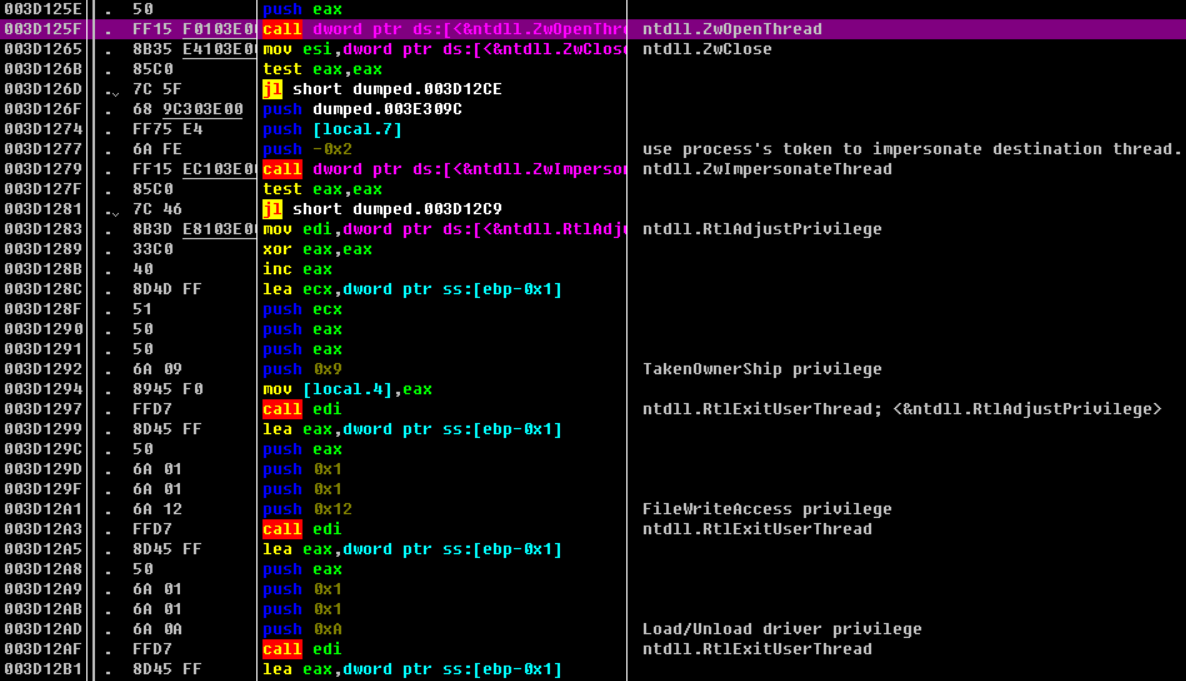
在一个循环中遍历检查所有的进程信息：



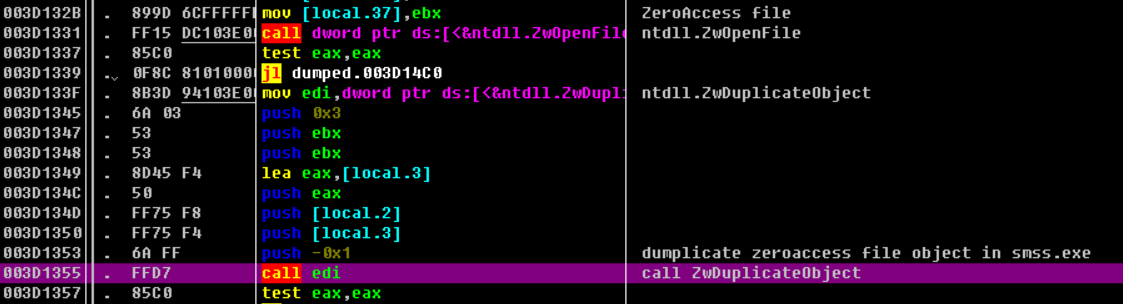
然后根据父进程和子线程的条件，挑选出一个满足条件的进程（这里是smss.exe）：



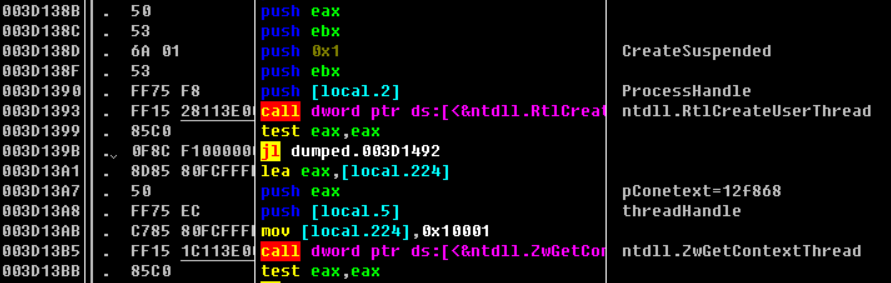
打开这个进程的主线程，为该线程分配多种权限：



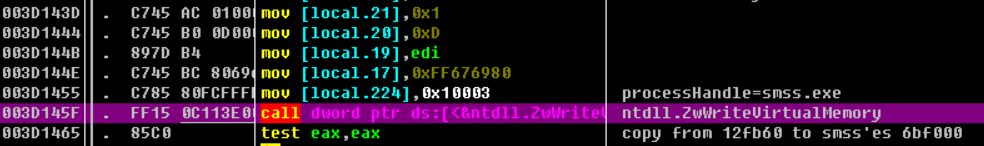
然后打开自身文件，在smss.exe进程中复制文件对象：



下一步，在smss.exe进程中创建一个挂起的线程，ZwGetContextThread函数获取这个线程的上下文：



将0x4c字节的数据写入smss.exe进程的内存空间中：

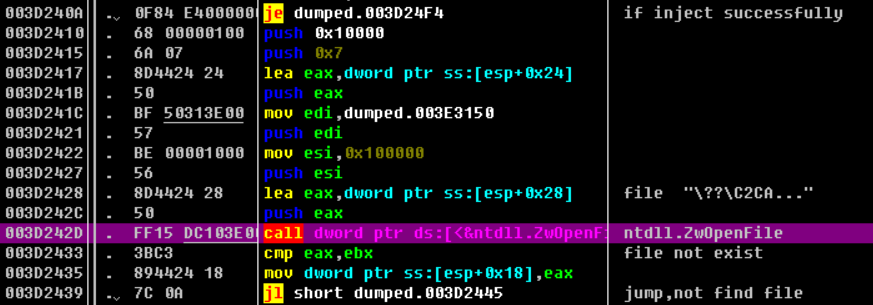


接着设置新创建线程的上下文，esp寄存器设置为内存地址0x6bf000，eip寄存器设置为ZwWaitForSingleObject函数入口地址。设置完毕后，调用ZwResumeThread函数等待时机执行：

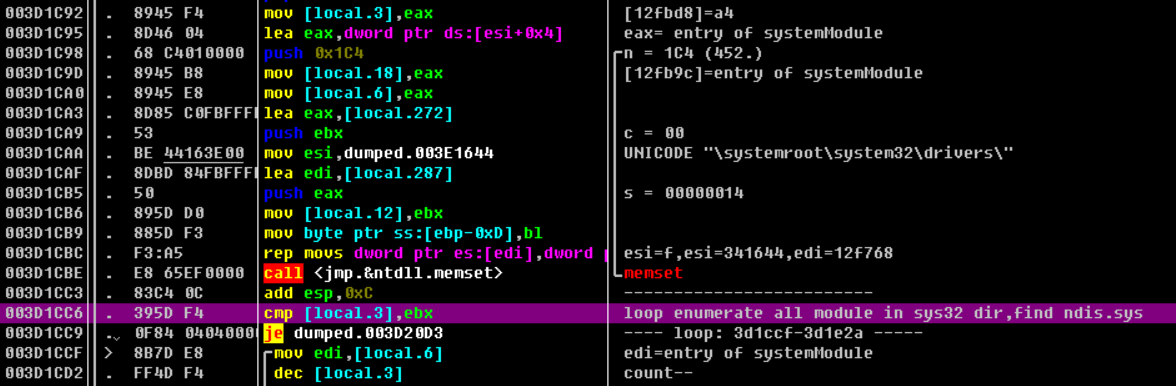


### 2.2.2 随机选择驱动模块

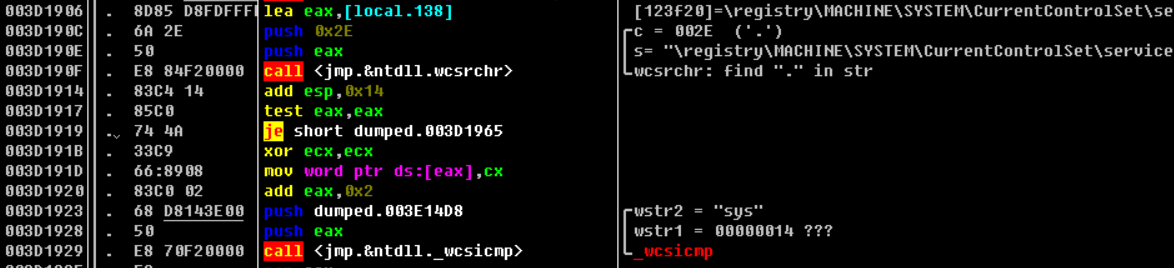
成功注入线程后，会尝试打开“\ ?? \ C2CAD972＃4079＃4fd3＃A68D＃AD34CC121074”的虚拟卷（已被感染后才存在这个虚拟卷），因此这里打开失败：



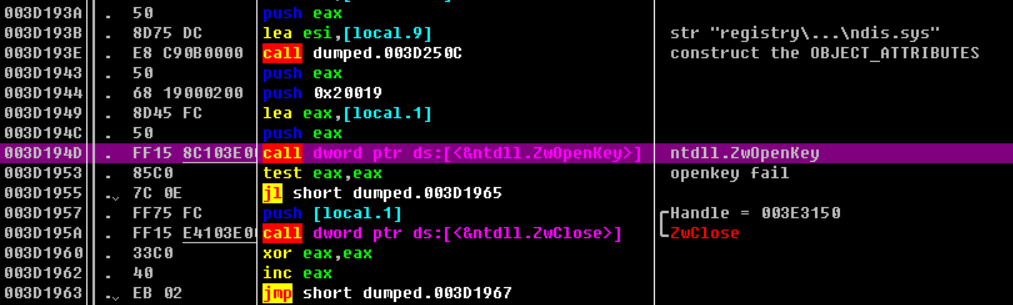
之后，代码循环调用ZwQueryInformation函数来获取系统模块列表，并找出名为“ndis.sys”的系统模块。ndis.sys是控制网络适配器的驱动程序文件。



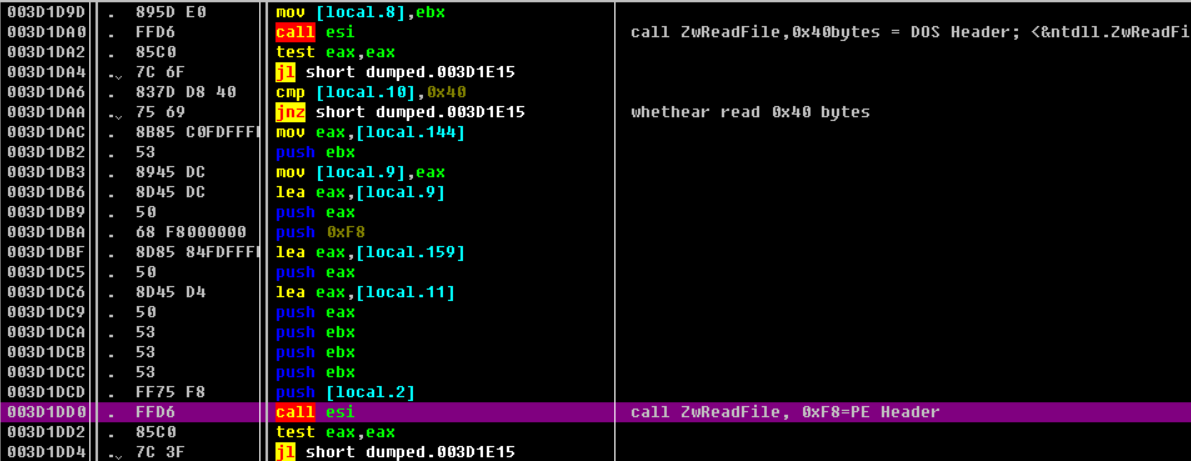
找到nids.sys系统模块后，检查驱动文件名和后缀名，确保是正常的系统驱动文件:



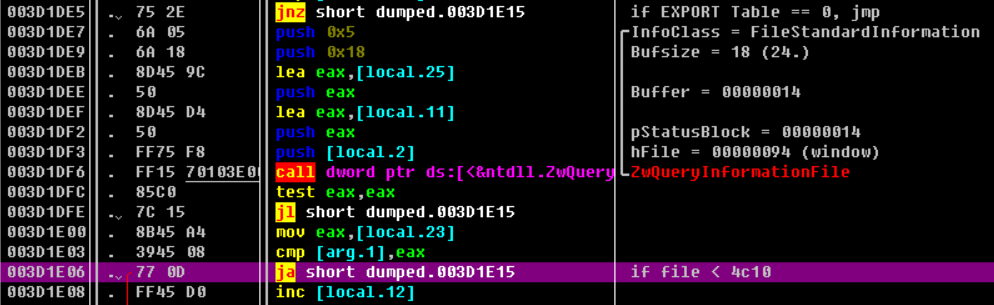
构建\_OBJECT\_ATTRIBUTES结构体，检查注册表文件是否存在：



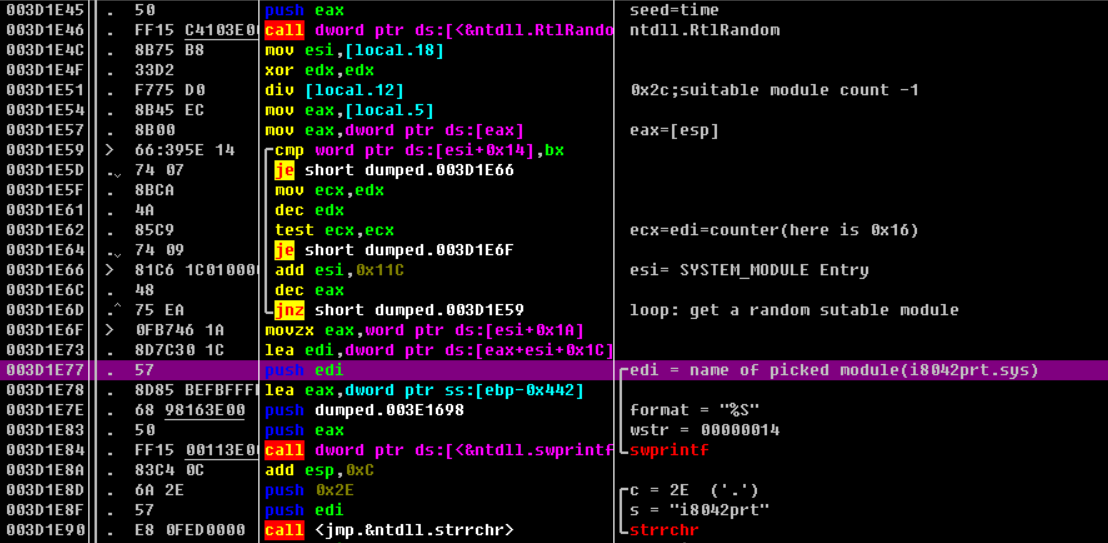
之后，打开\systemroot\system32\drivers\目录下的ndis.sys文件，两次调用ZwReadFile函数：第一次读取了0x40字节（DOS Header），第二次读取了0xF8字节（PE Header）。



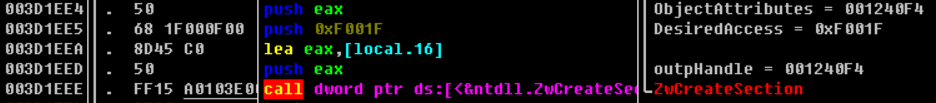
然后，检查ndis.sys文件的EXPORT TABLE的大小是否为零，若是，则继续搜索并检查下一个系统模块（ndis.sys导出表为零）；否则，通过导出表检查，读取模块的FILE\_STANDARD\_INFORMATION信息，获取模块文件大小，若小于0x4C10，也将继续搜索：



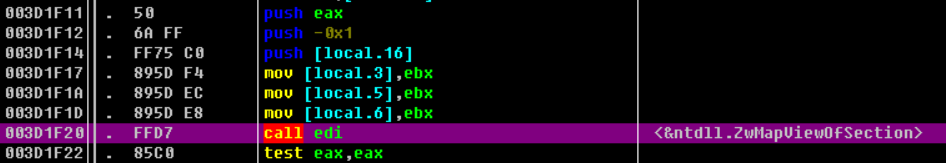
获取了所有符合标准的系统模块后，从中随机选择一个模块（这里是i8042prt.sys）：



检查模块系统功能一：调用ZwCreateSection函数创建一个节（进程之间的共享内存对象）：



检查模块系统功能二：调用ZwMapViewOfSection函数将节对象映射到进程的地址空间中：



检查模块系统功能三：调用ZwReadFile函数将文件内容读入内存区域：



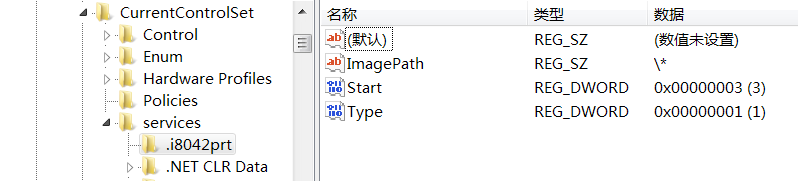
### 2.2.3 注册表写入

若通过了上述检查，则进行真正的恶意感染操作。这里要感染的驱动文件是i8042prt.sys，

则创建一个"\registry\MACHINE\SYSTEM\CurrentControlSet\Services\.i8042prt"的注册表项。



在注册表项下，会创建键值“start”、“Type”和“ImagePath”，如下图所示：

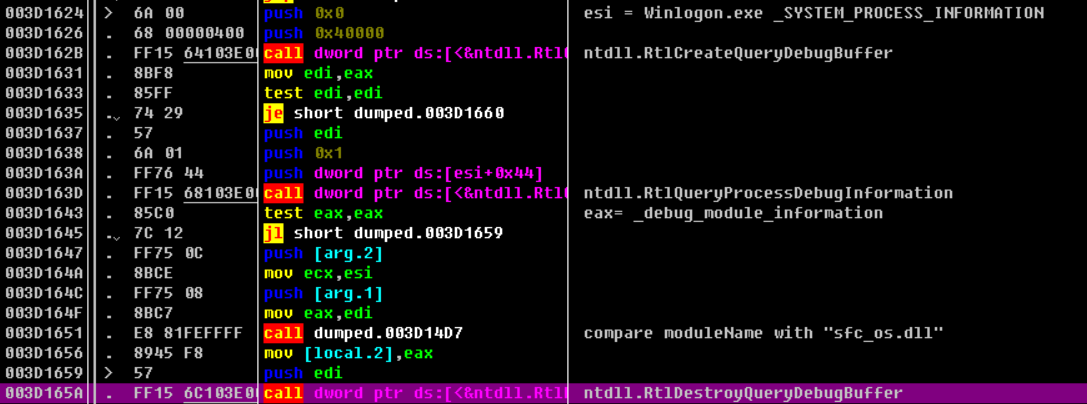


### 2.2.4 禁用文件保护服务

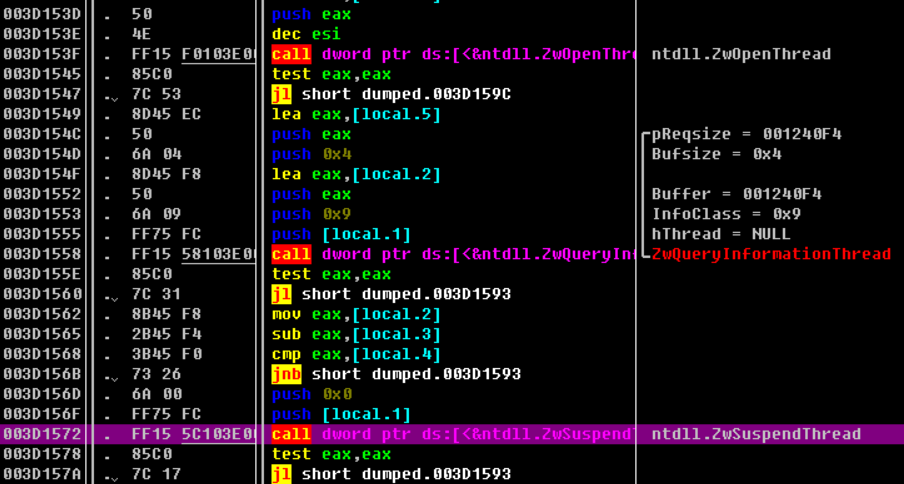
完成注册表相关操作后，下一步为禁用“sfc\_os.dll”的文件保护服务。首先，调用ZwQuerySystemInformation检索系统中的当前进程信息，将每个进程名和“Winlogon.exe”进行比较，直到找到名为“Winlogin.exe”的进程。



创建DEBUG\_BUFFER, 读取Winlogon.exe进程的DEBUG\_MODULE\_INFORMATION结构，将名称和“sfc\_os.dll”进行比较，直到找到“ sfc\_os.dll”模块的DEBUG\_MODULE\_INFORMATION结构。

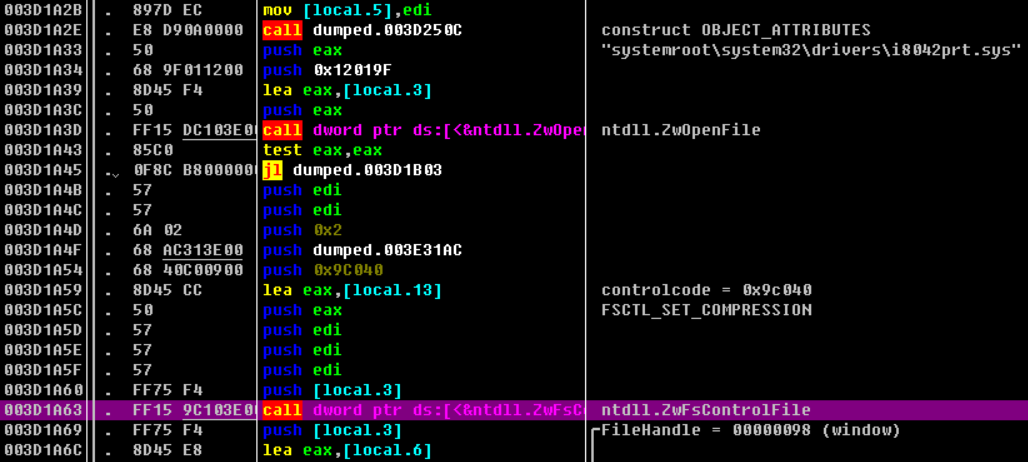


得到sfc\_os.dll模块的基址，循环搜索Winlogon.exe中的所有线程，杀死线程基地址在sfc\_os映像范围中的线程。



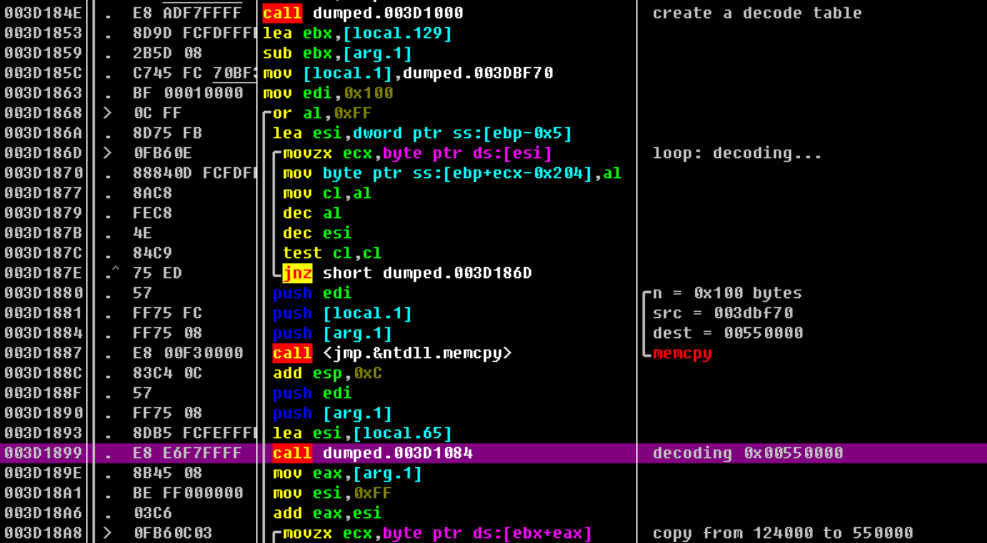
### 2.2.5 感染驱动文件

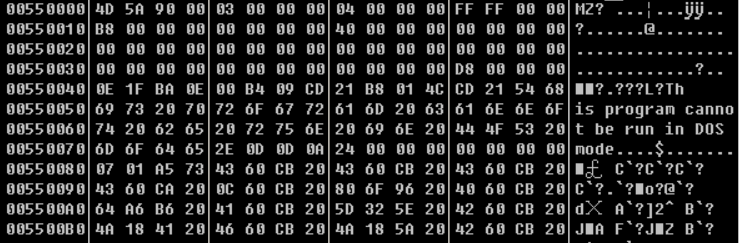
完成上述操作后，开始真正的感染操作。首先，调用ZwFsControlFile函数确保随机选择的驱动程序未压缩。



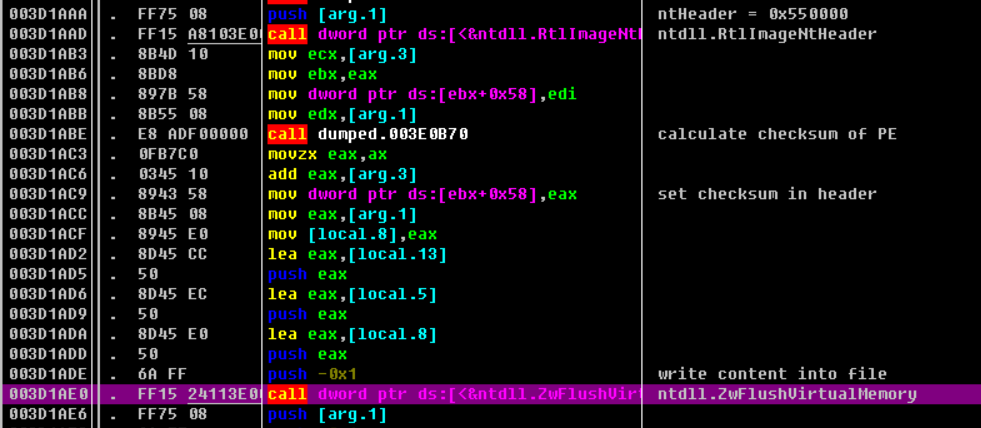
随后创建一个节，该节对象将要感染的驱动程序文件的内容映射到内存中，然后创建该节对象的映射（Windows使用节对象在进程之间共享内存）。



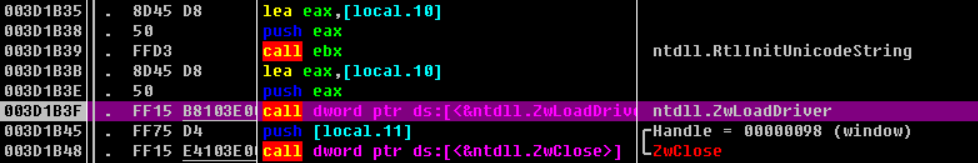
然后，创建一个解码表，将部分内容从堆栈拷贝到文件映射区域（0x550000）。经过几轮的解码和复制，最终在0x550000开始的映射区域中解码出一个PE文件。



设置NT头为映射区域（0x550000），计算PE文件的校验和，通过调用ZwFlushVirtualMemory函数将内存区域写入被感染的驱动文件（i8042prt.sys）。这种感染方法直接覆盖要感染驱动文件，被感染的驱动文件会失去原始功能。



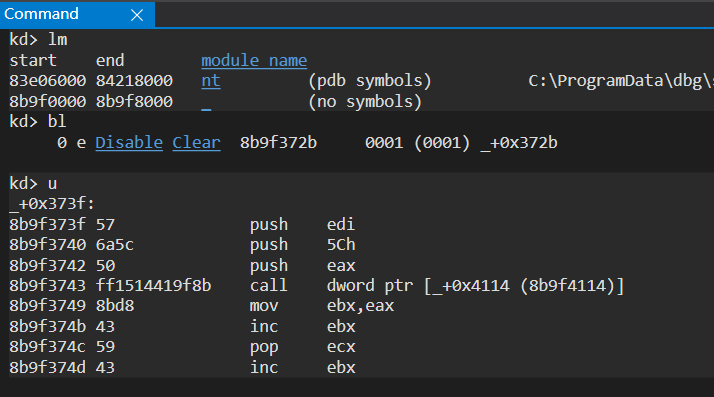
最后，立即调用ZwLoadDriver函数来加载受感染的驱动文件（i8042prt.sys）.



## 2.3 受感染的驱动分析

在执行ZwLoadDriver函数前，使用Windbg连接这个系统映像，并在“Settings-Event&exceptions-Load module on all modules”选项上设置为“break”，即加载新模块时中断。同时，静态分析被感染的驱动文件，发现其入口点为 “\_+372b”，所以可以在该处下断点。

然后在系统映像中执行ZwLoadDriver函数，第一次中断在加载模块时，第二次中断在模块入口点。



### 2.3.1 劫持磁盘驱动程序

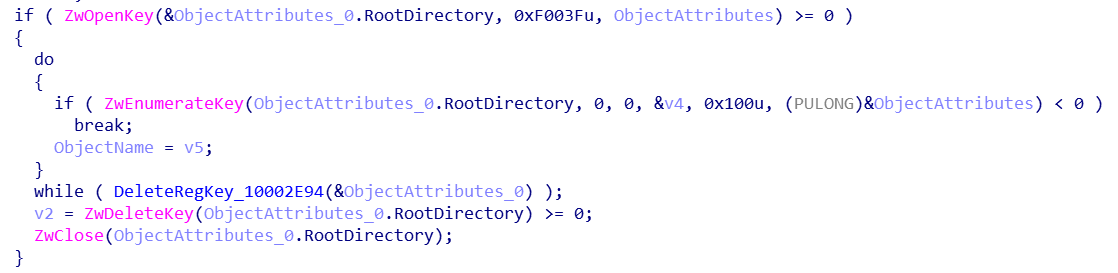
首先获取与驱动程序关联的服务名称，验证当前运行的服务程序是被感染的驱动程序：

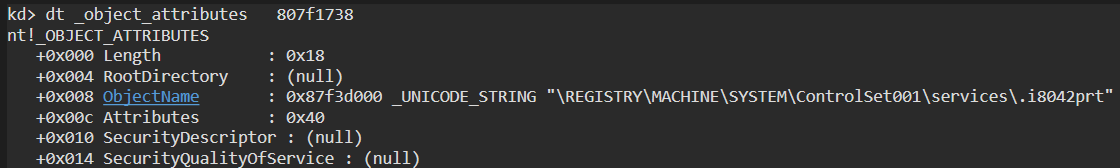


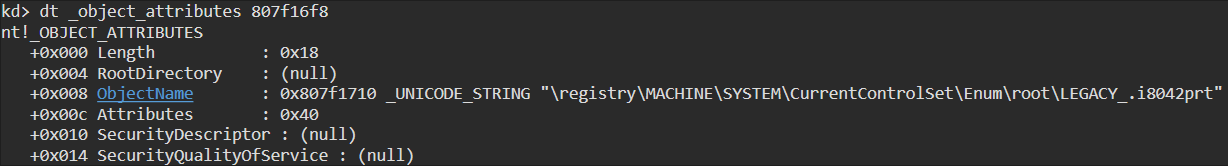
如果是被感染的驱动程序，则递归删除注册表项“\REGISTRY\MACHINE\SYSTEM\

ControlSet001\services\.i8042prt”和“\registry\MACHINE\SYSTEM\CurrentControlSet\Enum\

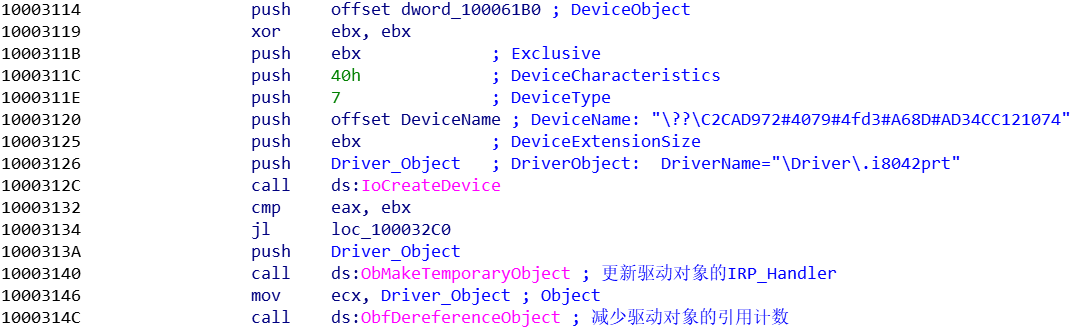
root\LEGACY\_.i8042prt”中的键值：



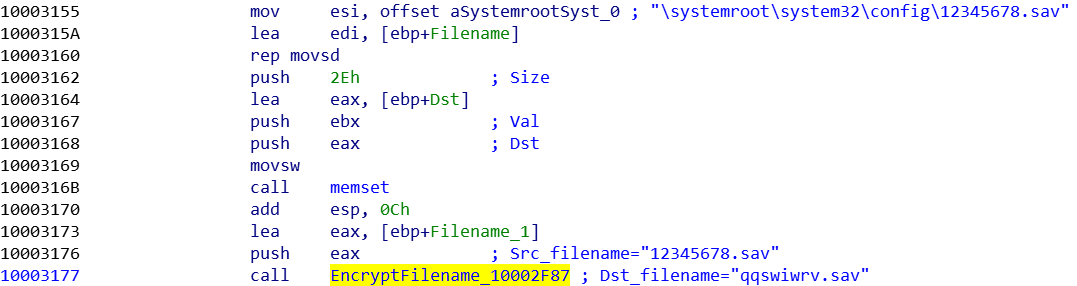




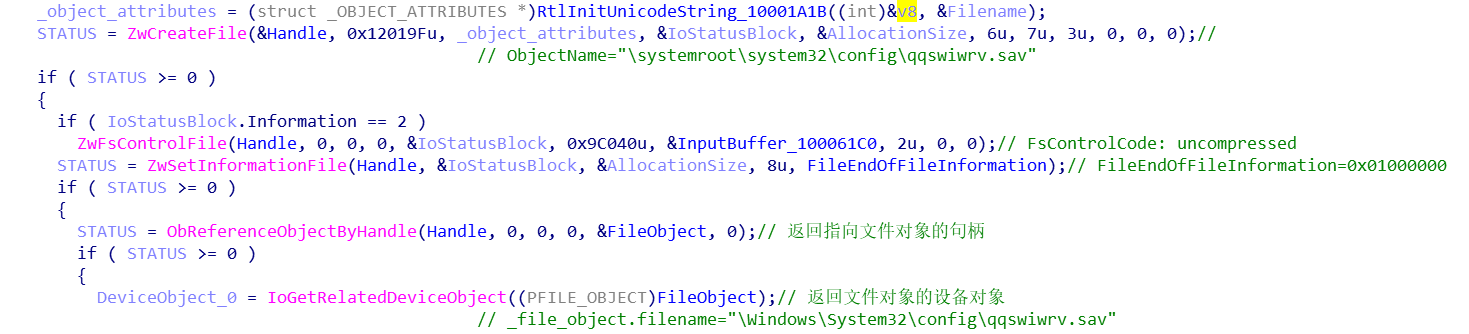
接下来，创建一个名为“\??\C2CA…”的设备对象，驱动程序为“\Driver\.i8042prt”。



下一步，读入一个文件名“12345678.sav”，使用最近的agp写入时间加密并生成一个文件名，这里是“qqswiwrv.sav”。



在系统目录\Windows\System32\config\下创建文件qqswiwrv.sav；调用ZwFsControlFile函数确保文件未压缩；调用ZwSetInformationFile函数将文件的末尾设置为0x01000000；返回文件qqswiwrv.sav的设备对象。



接下来尝试打开卷"\??\C2CAD972#4079#4fd3#A68D#AD34CC121074", 但是打开失败了（因为尚未为新的磁盘驱动程序设置IRP处理），所以后面跳转到\_+2c95函数处。



\_+2c95处的函数首先获取了在“Driver\Disk”中注册的设备对象列表；然后，创建一个新的磁盘设备对象，将其插入IRP请求处理程序链。这样，与磁盘相关的所有IRP请求都将首先连接到这个恶意驱动程序。

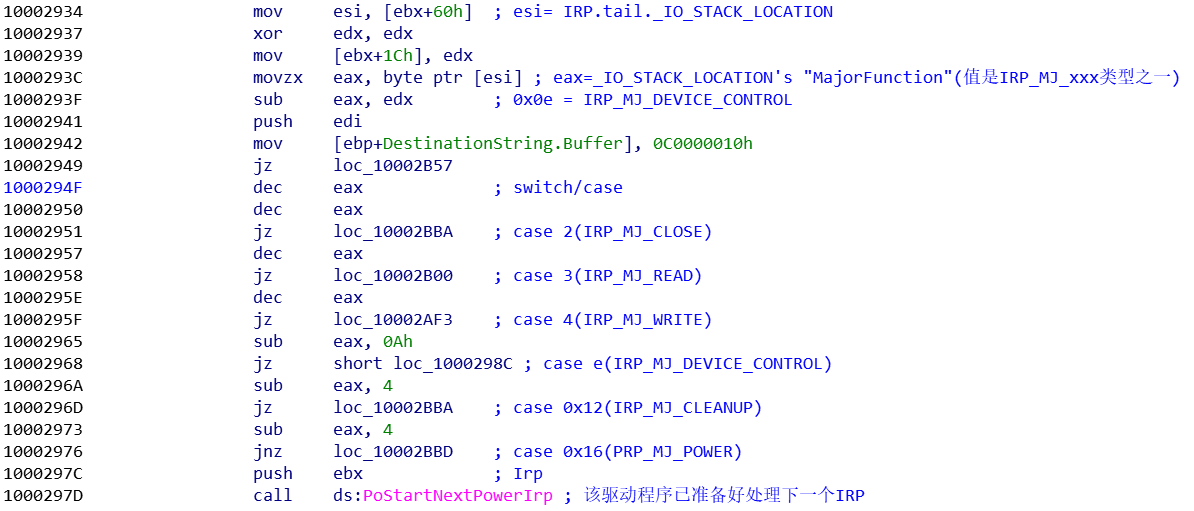


将恶意驱动程序插入IRP请求处理程序链时，还需要实现对IRP的处理。首先是对传入的参数DeviceObject进行比较：若是对“\??\C2CAD972#4079#4fd3#A68D#AD34CC121074”

的请求，则执行自定义的函数执行实际操作以模拟虚拟磁盘；否则，调用PoCallDriver将请求中继到低级（实际）驱动程序以完成工作



执行IRP实际操作的函数，判断了IRP的\_IO\_STACK\_LOCATION的MajorFunction的值，根据这个取值执行Switch/Case语句，重定向控制流以执行不同的IRP请求。



因此，对磁盘卷的读取操作，是通过前面创建的“12345678.sav”文件上构造一个新的\_IO\_STACK\_LOCATION任务来实现的。

### 2.3.2 再次感染驱动文件

接下来，打开"\systemroot\system32\drivers\i8042prt.sys"文件（要感染的驱动文件，且该文件已被感染过了），查询文件的标准文件信息，创建虚拟页面，将要感染的驱动程序文件读入这些页面，然后直接释放内存页。（猜测这是用于测试文件操作和内存分配功能能否使用）。



同样的，将“\.i8042prt”文件内容（用于感染驱动的恶意代码）复制到新申请的内存页中，复制完成后立即关闭新申请的内存页。



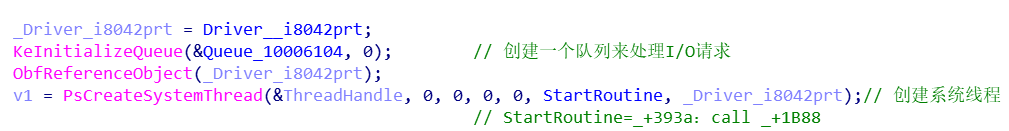
通过前面两次的功能测试后，进行真正的感染驱动程序操作。打开正常的i8042prt.sys文件，创建一个节并将节对象映射到内存空间，将存放恶意代码的内存页复制到文件的内存空间，最后刷新内容回文件。

完成驱动感染后，从模块链表中删除受感染的驱动程序，达到隐藏的功能；复制实际驱动磁盘驱动程序中的属性来配置恶意磁盘驱动程序。

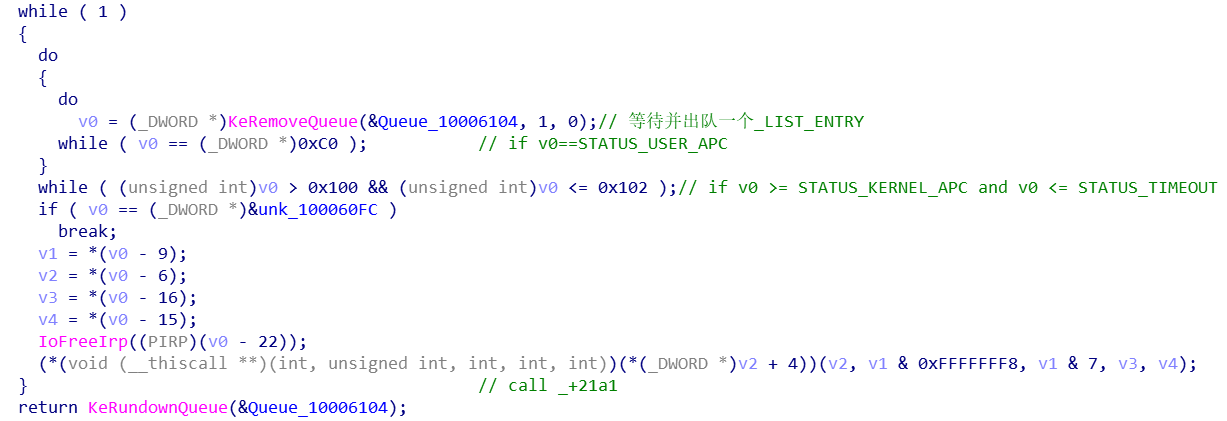


### 2.3.3 TDI网络行为

调用KeInitializeQueue函数创建一个队列来处理I/O请求，接着创建一个系统线程，被感染的磁盘驱动器对象作为参数，而功能实现在启动例程函数中。



跟进启动例程函数，该函数是出队一个\_LIST\_ENTRY实例，接着调用了位于\_+21a1处的函数：



\_+21a1处的函数是驱动程序处理逻辑，包含很多判断和跳转，跳转后，执行\_+1660处的函数。继续跟进，这个函数的功能为生成一个GUID，拼接HTTP请求头，然后调用\_+2090处的函数。

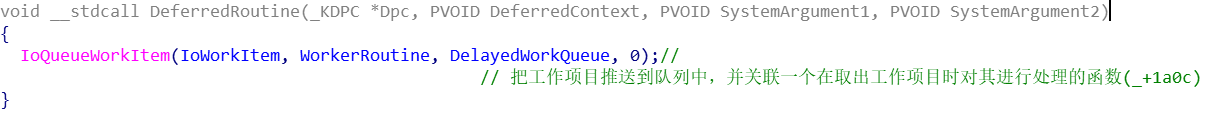


最终，\_+2090处的函数是建立IRP请求并调用网络驱动程序以发出请求：  


下一部分：调用IoAllocateWorkItem函数分配系统线程的工作项；初始化一个计时器对象；调用KeInitializeDpc函数初始化延迟过程调用（因为所需文件还未创建）；设置计时器时间。



其中，KeInitializeDpc函数参数中包含一个注册函数，跟进注册函数，其功能是将工作项目推送到队列中，并关联一个在取出工作项目时对其进行处理的函数。



继续跟进WorkerRoutine函数，该函数首先从隐藏卷文件中读取数据，生成解密表解密数据；获得要连接的IP地址和域名，最后调用函数进行连接：



## 2.4 配置Rookit

前面加载受感染的驱动文件（i8042prt.sys）完成恶意操作后，控制权返回到注入lz32.dll的恶意代码中，即调用ZwLoadDriver(“i8042prt.sys”)的下一条指令处。

### 2.4.1 恢复线程

在一个循环体中，逐个恢复之前挂起的线程（前面挂起的包含sfc\_os.dll模块的线程，即重启Windows文件保护服务）：

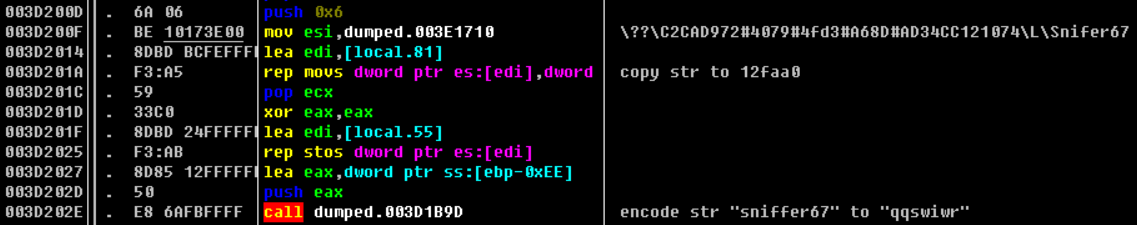


### 2.4.2 创建虚拟驱动器

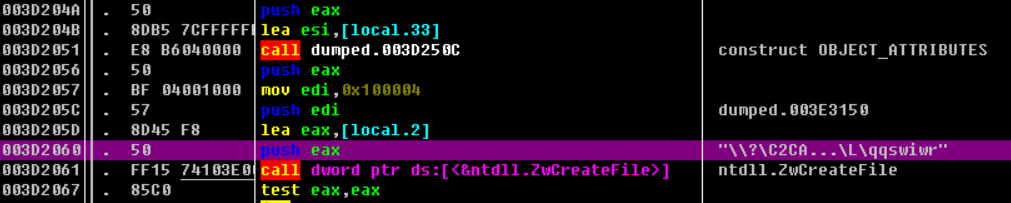
加载fmifs.dll，获取FormatEx函数地址，调用该函数格式化驱动器“\??\C2CAD972#4079#4fd3#A68D#AD34CC121074”，如下图所示：



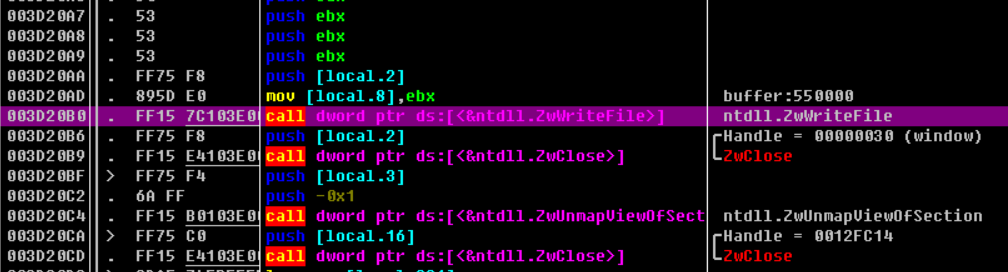
### 2.4.3 隐藏驱动器文件存储恶意文件

首先，加密一个文件名：  


用这个文件名在隐藏驱动器中创建一个文件：



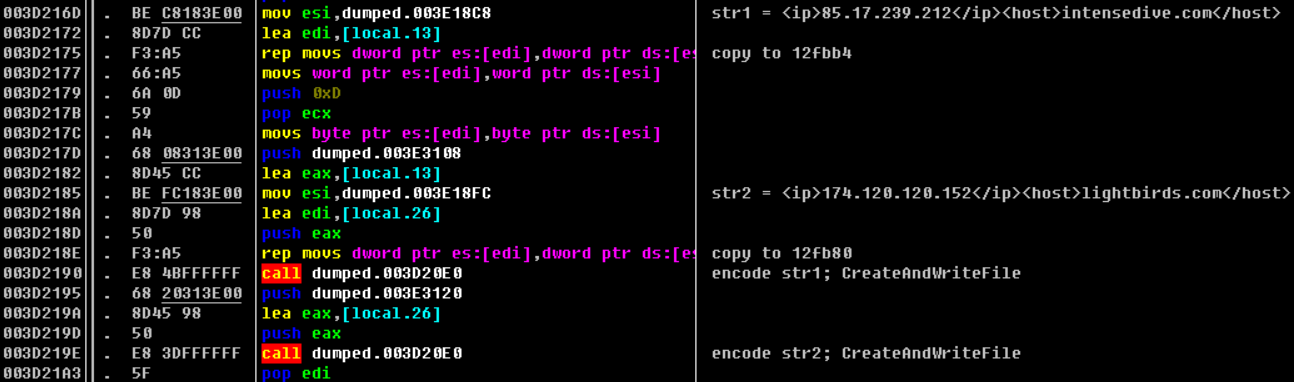
最后将受感染的驱动程序文件（\.i8042prt.sys）内容存储在隐藏文件“\??\C2CAD972#4079#4fd3#A68D#AD34CC121074\L\qqswiwr”中：



### 2.4.4 TDI配置文件

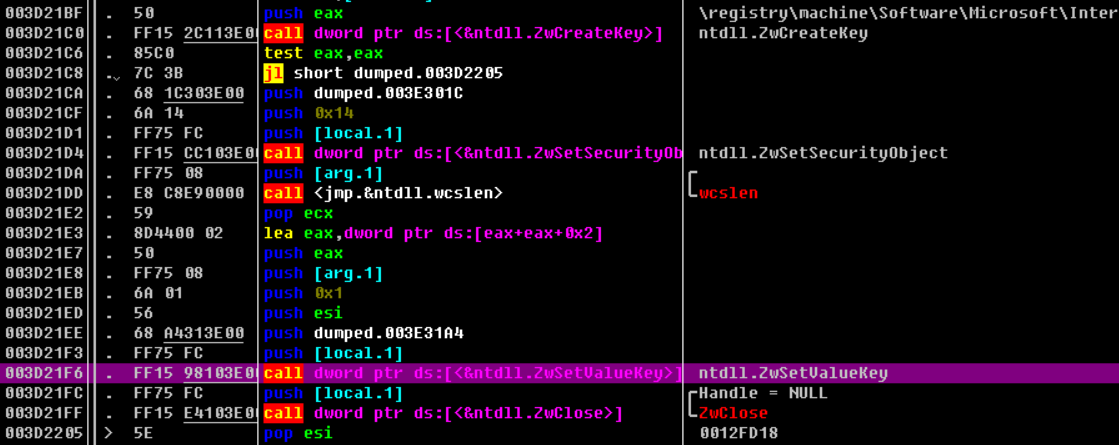
从硬编码的数据中读入标签<ip>和<host>的内容，将其加密后，写入新创建的隐藏文件

“… \{49B474EB-92D0-464f-B1DD-1F37AABF9D95}”和“…\{B2E11362-D348-4411-A4B4-82673555A60B}”中：



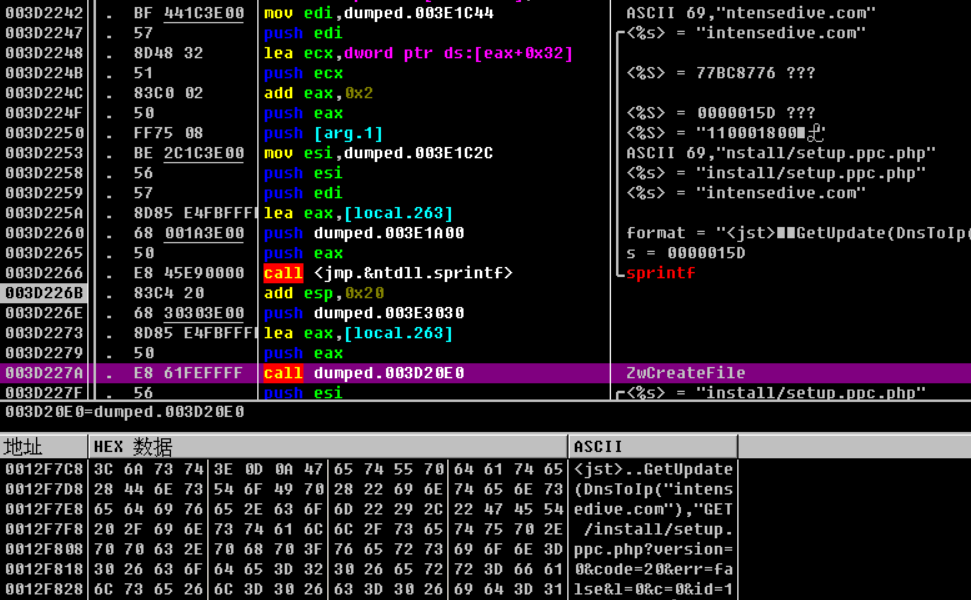
### 2.4.5 注册表存储解密密钥

在注册表“\registry\machine\Software\Microsoft\Internet Explorer\Main”下创建键值，键名为“{F9197A7E-CE10-458e-85F8-5B0CE6DF2BBE}”，键值为“110001800”：

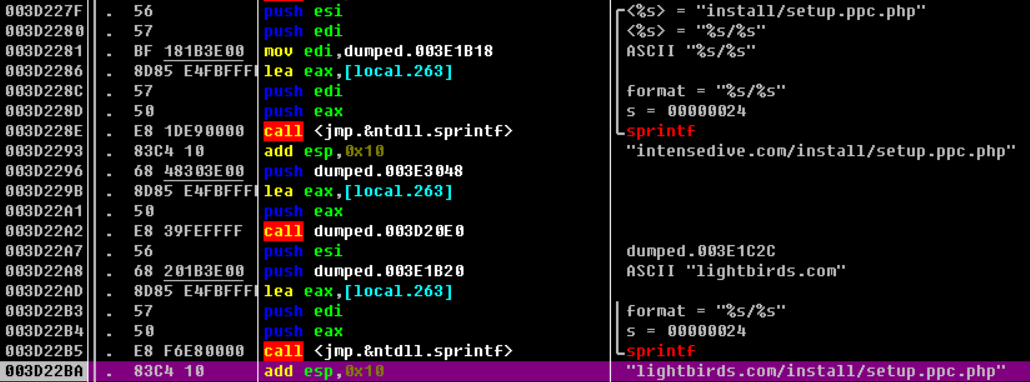


### 2.4.6 创建HTTP请求相关文件

构造HTTP请求头，再创建隐藏文件写入并保存，如下图所示：

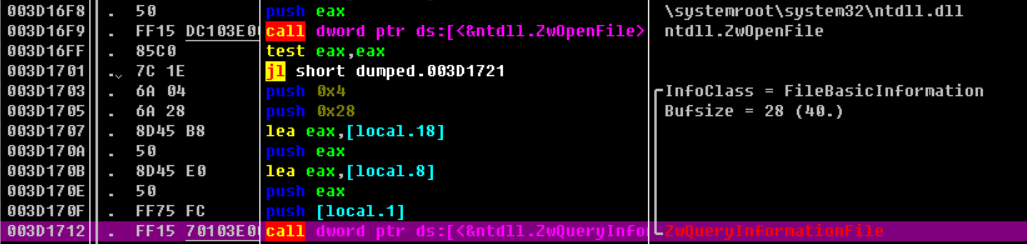


同样的，又分别构造了字符串"intensedive.com/install/setup.ppc.php"和"lightbirds.com/install/setup.ppc.php"，分布创建隐藏文件写入并保存：

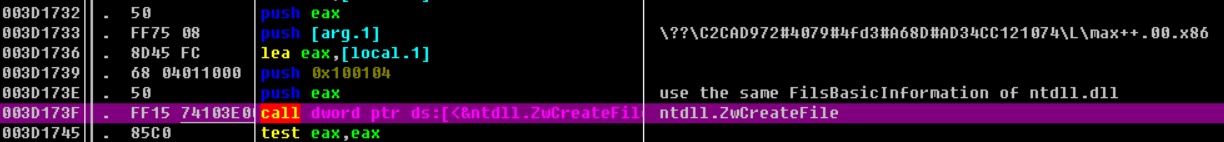


### 2.4.7 创建恶意文件max++.00.x86

首先打开ntdll.dll文件读取基本文件信息：

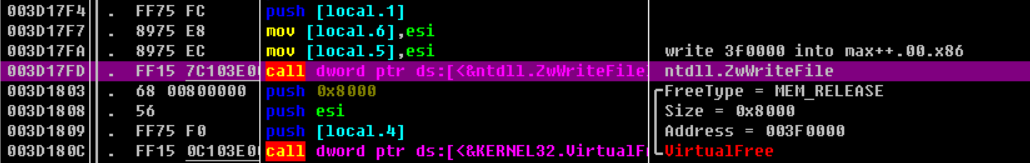


然后在隐藏驱动器创建一个max++.00.x86文件，文件基本信息设置与ntdll.dll相同：



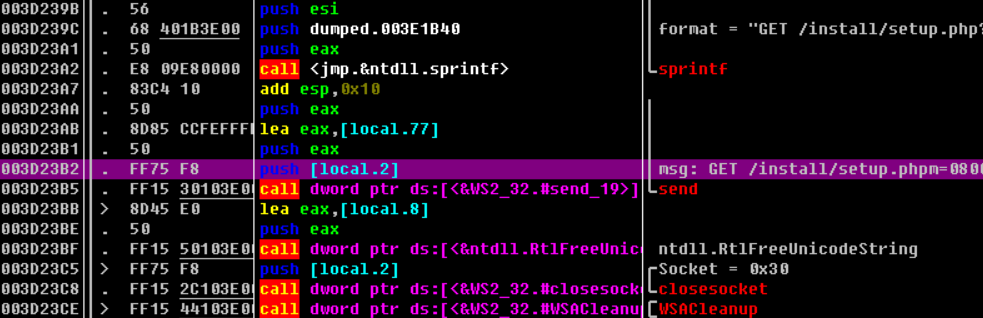
然后，分配一块内存空间，建立一个解码表，多次解密代码到新创建的内存中：



最后，将内存中的数据写入max++.00.x86文件中（后面会调用系统驱动加载这个文件）： 

### 2.4.8 向服务器发消息

调用ws2\_32.dll模块的相关API，和远程服务器建立连接，发送一个消息，表示感染完成。



## 2.5 max++.x86.dll 分析

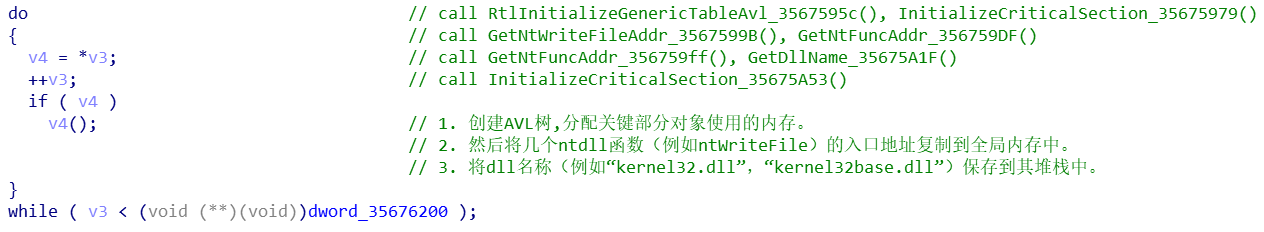
### 2.5.1 隐藏DLL加载

完成前面Rookit的配置后，会将max++.00.x86文件作为DLL进行加载。在入口点，首先调用ZwTestAlert()函数清除APC队列，然后调用DisableThreadLibraryCalls()函数禁用DLL\_THREAD\_ATTACH通知，从而调试器难以捕获DLL的加载信息。



### 2.5.2 IAT Hook

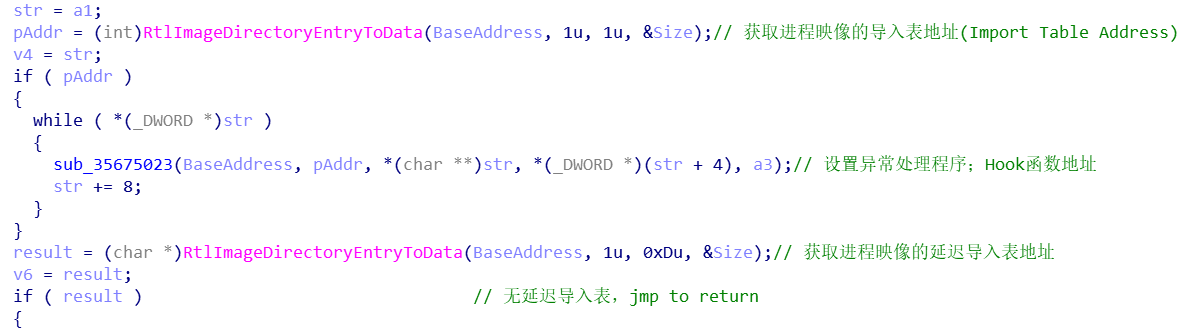
通过一个循环，直接从数组中取出函数地址进行调用，所调用的函数和功能如下图所示：



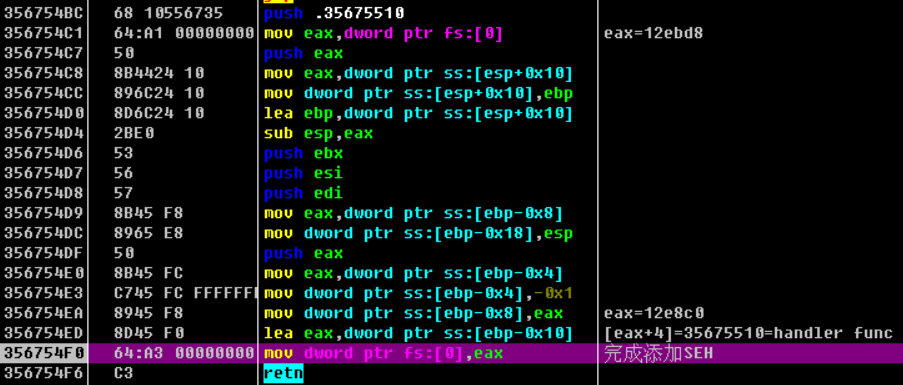
接下来，对两个关键模块KernelBase.dll和kernel32.dll进行处理，执行Hook前的操作：



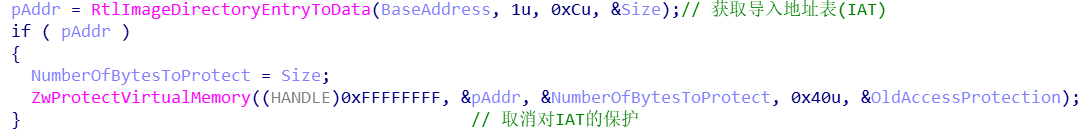
首先获取模块的导入表地址：



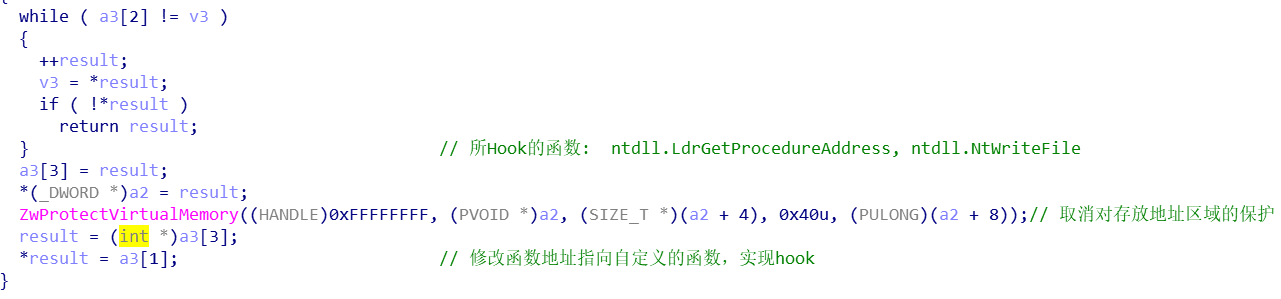
并且在进行函数Hook前，设置异常处理函数：



然后获取模块的IAT，调用ZwProtectVirtualMemory函数取消对IAT的保护：

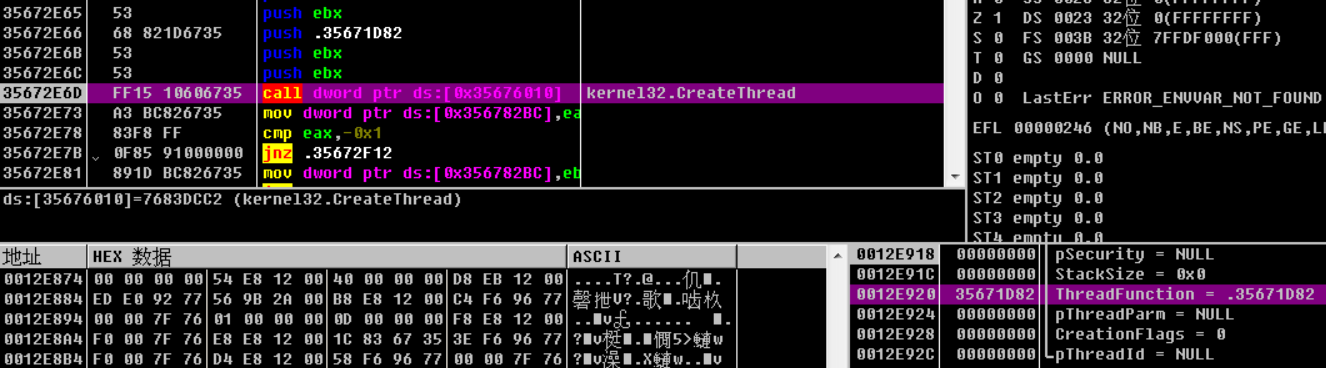


最后，通过函数名找到存放函数地址的地址，修改该地址为自定义的函数，实现Hook（Hook的函数为ntdll.LdrGetProdureAddress和ntdll.NtWriteFile）：

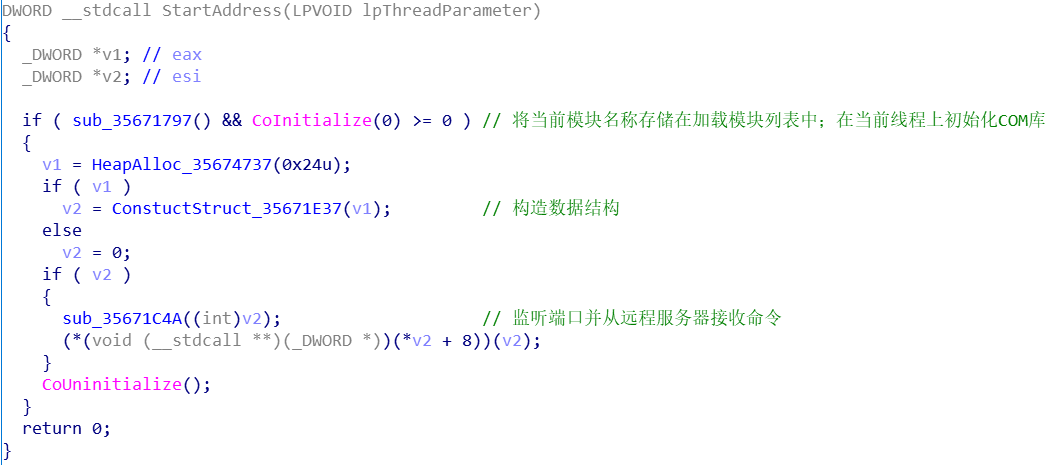


### 2.5.3 远程加载DLL

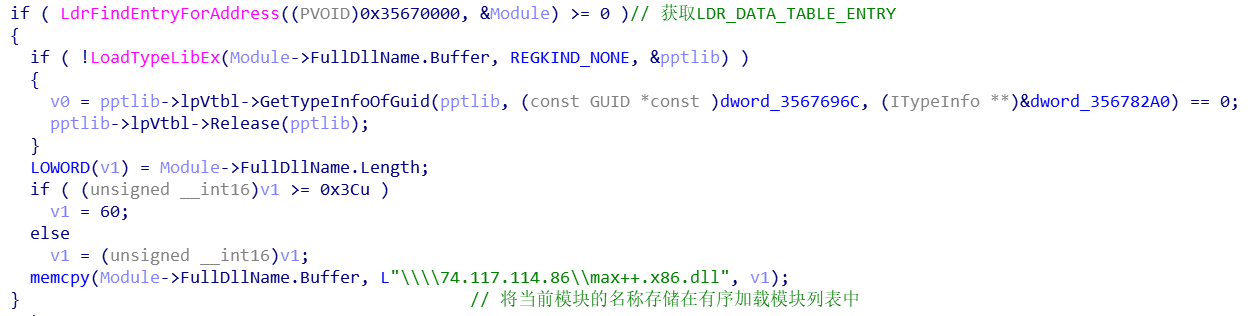
创建新线程执行函数例程：



启动函数实现了这个恶意模块的主要功能，如下图所示：



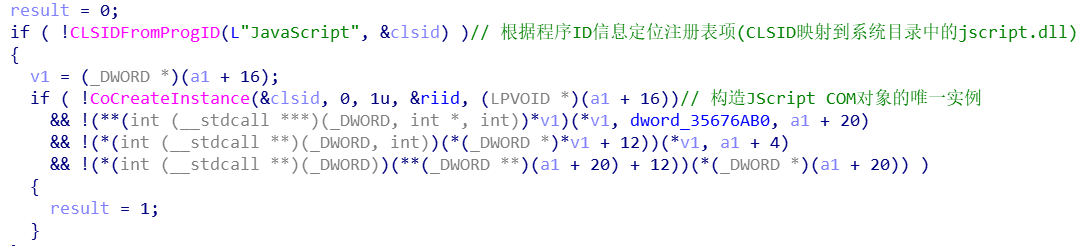
位于35671797的函数将当前模块的名称“\\74.117.114.86\max++.x86.dll”存储在有序加载模块列表中。这样，函数返回时，通过调用Ole32.CoInitialize函数，将尝试再次加载当前模块，从而迫使系统加载远程DLL “\\ 74.117.114.86 \ max ++。x86.dll”。当然，由于文件不可访问，不能进行分析。



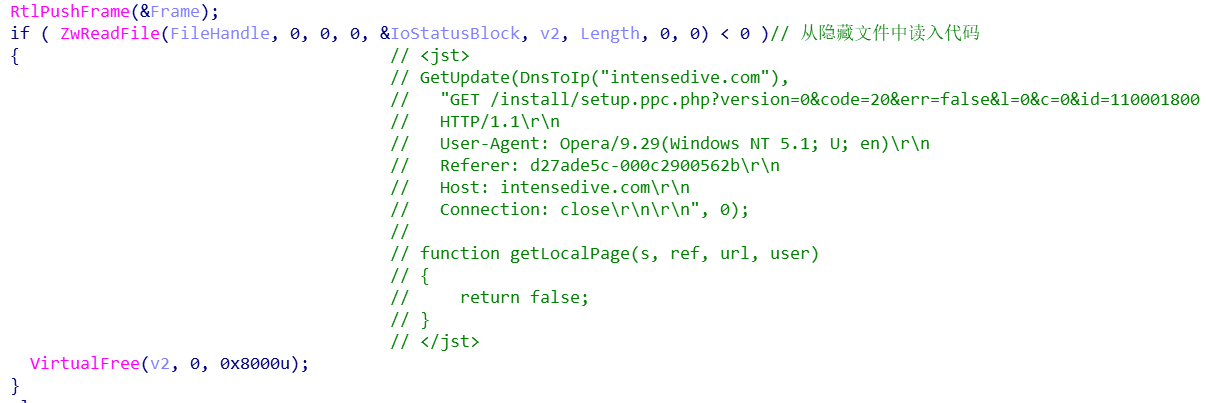
### 2.5.4 网络行为

接下来调用位于35671797的函数，这是与网络操作相关的函数。

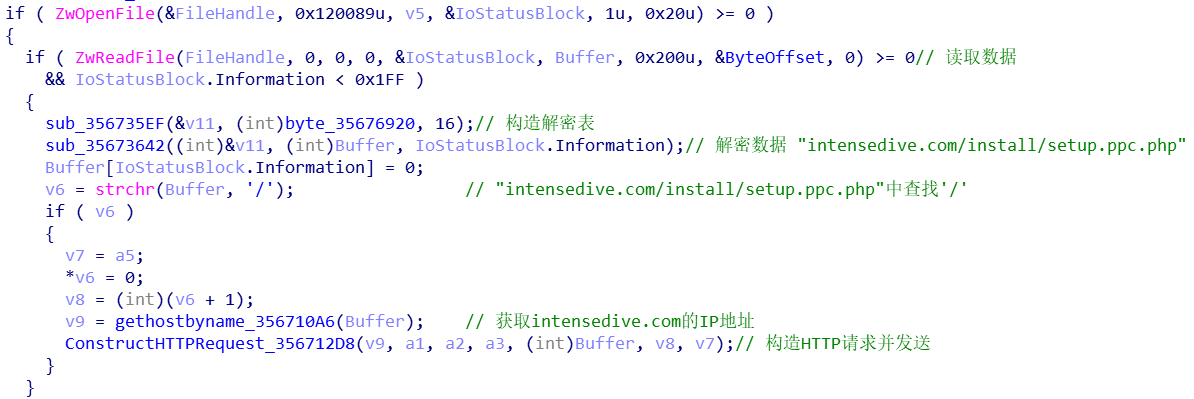
这部分首先构造JScript COM对象实例，并包含与该实例相关的调用。但由于初始化失败，无法继续分析：



从隐藏文件中读取HTTP请求头：



再从另一个隐藏文件中读取数据，获取域名intensedive.com的IP地址：



最后向目标地址发送数据包：



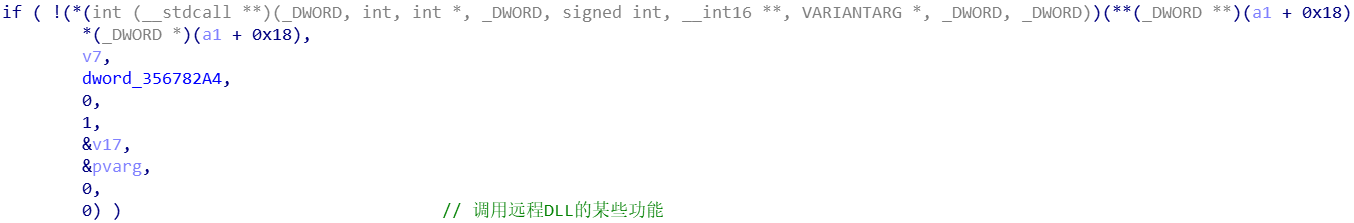
下一步，打开本机的5000端口，保持监听该端口，等待返回的数据包：



如果返回的数据包为特定类型，则选择采取相关操作：

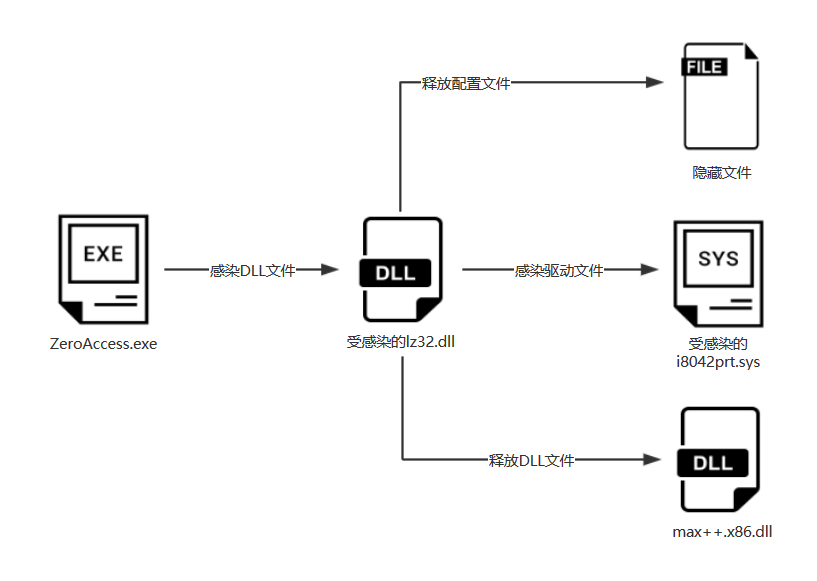


由于无法访问远程DLL，不能确切知道执行的远程DLL功能，猜测可能是命令控制功能：



根据服务器返回的数据包完成相应的功能操作后，继续监听本机5000端口，等待端口接收数据包，再次执行相应的功能操作。

# 执行流程



# IOC

|  |
| --- |
| intensedive.com |
| lightbirds.com |
| 85.17.239.212 |
| 174.120.120.152 |
| 74.117.114.86 |