摘 要

Windows 操作系统是目前主流的操作系统，基于这个平台下的各种程序软件层出不穷，相应的木马病毒也在不断的进步。为了对抗杀毒软件，这些木马病毒企图霸占电脑主权，更进一步的扩展恶意行为。这些恶意程序往往具备对抗当今主流杀毒软件的能力，给用户带来了不可估量的损失。

研发探究这些病毒木马的原理及其行为，制作出相应的安全工具来对抗它们，显得十分必要。然而由于 Windows 操作系统是不开源的，这对研究其内部的原理增加了一定的难度，制作相应的 Anti-Rootkits 工具也并非易事。但正是由于这些原因，使得对抗当前流行的病毒木马变得更加具有价值和挑战性。

本论文着重讨论 Windows 平台下主流的 Rootkit 技术及防御技术，并且最终将实现一个具有相对完善功能的反 Rootkit 软件 SnowShadow。目前 SnowShadow 的最新版本是 1.2，综合有进程、文件、注册表进程等数十项功能。本软件大量使用微软未公开技术，许多技术建立在逆向操作系统内核代码的基础之上，使用本软件有一定风险。使用本软件可以实现实现手工发现并清除 Rootkit 病毒，进程管理可以用于结束顽固进程，枚举隐藏进程，文件管理支持文件的暴力删除，隐藏文件的枚举，文件解锁等功能，用于清除 Rootkit 病毒文件，注册表一项，基于注册表文件解析的编辑器，支持注册表离线编辑（包括删除，增加修改等功能），由于不使用任何 API 来操作注册表，可以防止通用注册表保护手段的干扰。

关键字：Anti-Rootkit 、Rootkit、反病毒、Windows 内核。

Abstract

Windows operating system is the current popular operating system. Based on this platform, there are endless variety of software and the corresponding Trojans also continue to progress. To counter the anti-virus software, these Trojans attempt to commandeer the computer's sovereignty, and further expand the malicious behavior. These malicious programs often have the capability of today's major anti-virus software, which gives users an immeasurable loss.

It is extremely necessary to explore the principles and behavior of the Trojans. However, since Windows operating system is not open, it becomes more difficult for people to know its principle and to develop the related Anti-Rootkits tools. Because of these reasons, making the Trojans against the current circulating virus became more valuable and challenging.

This paper focuses on the mainstream Windows platforms Rootkit technology and defense technology, and will eventually realize a relatively perfect anti-Rootkit software SnowShadow. The latest version of the current SnowShadow is 1.2, integrated with the process, file, registry, process, and dozens of functions. This software will use plenty of technology that Microsoft does not apply and many technologies are based on the operating system kernel code, thus using of this software has some risks. Use of this software can be achieved manually find and remove Rootkit virus, process management can be used for the end of stubborn process, enumerate hidden processes, document management support for the violence to delete the file, hidden file enumeration, file unlock and other functions, used to clear Rootkit virus file, a registry-based analysis of the registry file editor, support for offline editing the registry (including delete, modify etc added), because they do not use any API to manipulate the registry to prevent common registry protection Means of interference.

Keywords: Anti-Rootkit, Rootkit, Anti-Virus, Windows Kernel.

目 录

第一章 引言 1
1.1 相关概念 1
1.2 研究背景及意义 1
1.3 主要工作 2
第二章 Windows 应用层开发 3
2.1 PE 格式及加载重定向 3
2.2 原生 API 的使用 5
2.3 本软件开发基础 6
2.4 程序界面拟合 9
2.5 本章小结 9
第三章 Windows 驱动开发 10
3.1 Windows 驱动模型 10
3.2 开发准备 11
3.3 调试设置 12
3.4 本章小结 12
第四章 进程管理 14
4.1 进程与线程 14
4.2 进程隐藏技术 16
4.3 进程保护技术 18
4.4 用户态枚举隐藏进程 20
4.5 内核态枚举隐藏进程 22
4.6 强制终止进程 25
4.7 模块枚举及卸载 26
4.8 本章小结 29
第五章 内核钩子及回调函数 30
5.1 内核钩子分类 30
5.2 SSDT/ShadowSSDT 钩子枚举及卸载 30
5.3 内核普通函数钩子检测及恢复 32
5.4 中断钩子检测及恢复 33

5.5 FastCallEntry 钩子检测及恢复 36
5.6 Object 钩子检测及恢复 37
5.7 其它几个重要函数钩子检测及恢复 39
5.8 通知回调检测及卸载 39
5.9 注册表回调检测及卸载 41
5.10 本章小结 42
第六章 文件管理 43
6.1 文件隐藏及检测 43
6.2 FSD 钩子检测及恢复 43
6.3 过滤驱动检测及摘除 44
6.4 文件解锁及强制访问 45
6.5 磁盘级文件解析 47
6.6 本章小结 55
第七章 注册表 57
7.1 注册表以及隐藏 57
7.2 注册表文件格式 57
7.3 注册表文件解析 59
7.4 注册表文件访问方法 59
7.5 注册表解析进一步完善 60
7.6 本章小结 60
第八章 自我保护 61
8.1 基于 x86 的 inline Hook 引擎 ring0 及 ring3 编写 61
8.2 进程自我防护方法 62
8.3 应用层自我防护 64
8.4 防御消息洪水攻击 66
8.5 本章小结 66
第九章 软件签名系统设计与实现 67
9.1 为什么需要验证 67
9.2 RSA 加密算法 67
9.3 PE 文件的签名及验证实现 67
9.4 本章小结 70
第十章 结论 71

10.1 软件调试测试 71
10.2 全文总结 74
10.3 展望 74
参考文献 75
致谢 77
外文资料原文 78
外文资料译文 92

第一章 引言

1.1 相关概念

计算机病毒：

编制或者在计算机程序中插入的破坏计算机功能或者破坏数据，影响计算机使用并且能够自我复制的一组计算机指令或者程序代码被称为计算机病毒（Computer Virus）。具有破坏性，复制性和传染性。

特洛伊木马：

一种秘密潜伏的能够通过远程网络进行控制的恶意程序。控制者可以控制被秘密植入木马的计算机的一切动作和资源，是恶意攻击者进行窃取信息等的工具。

Rootkit 病毒：

Rootkit 是指其主要功能为隐藏其他程式行程的软体，可能是一个或一个以上的软体组合，广义而言，Rootkit 也可视为一项技术。最早 Rootkit 用于善意用途，但后来 Rootkit 也被骇客用在入侵和攻击他人的电脑系统上，电脑病毒、间谍软体等也常使用 Rootkit 来隐藏踪迹，因此 Rootkit 已被大多数的防毒软体归类为具危害性的恶意软体。Linux、Windows、Mac OS 等作业系统都有机会成为 Rootkit 的受害目标。

Rootkit 一般运行于操作系统内核空间，拥有和操作系统一样高的权限，为了实现伪装隐藏等恶意目的，其一般会破坏操作系统的完整性，改变原始系统内核指令流程。由于其运行于内核态，使的检测 Rootkit 变得比较困难。现今更多的是具有利益目的的 Rootkit，一般配合用户态的木马工作，主要要来窃取一些游戏帐号，银行卡帐号和其它私密信息，给用户造成有形和无形的损失，由于网络犯罪取证困难，造成的损失却不可估量，所以检测 Rootkit 日益变得重要。

1.2 研究背景及意义

Windows 操作系统是目前主流的操作系统，基于这个平台的各种软件层出不穷，同时也出现了各种类型的计算机病毒和木马程序。而且更为严重的是 Rootkits 的诞生，给系统安全带来了极大的破坏性。计算机病毒的产生是计算机技术和以计算机为核心的社会信息化进程发展到一定阶段的必然产物，是高技术犯罪,具有瞬时性、动态性和随机性。不易取证,风险小破坏大,从而刺激了犯罪意识和犯罪活动，是某些人恶作剧和报复心态在计算机应用领域的表现。 最初，某些程序员为

了展示自己强大精湛的电脑技术，制作了恶作剧程序。这些程序不符合正常软件的常规逻辑，具有病毒木马的雏形。而后，在巨大经济利益的驱使下，逐渐演变成现今破坏重要数据，劫持网络通讯，盗取用户网络银行、游戏账号密码，窥探用户隐私攫取暴利的恶意工具。而且为了提高生存能力，企图在受害者电脑内获得一席之地，霸占电脑主权，更进一步的扩展恶意行为，这些恶意程序更是采用底层的 Windows 内核技术，来对抗当今主流的杀毒软件。比如隐藏自己的进程、文件、注册表信息，让用户很难发现它们的存在。而它们依然在后台静悄悄的窥探着用户的一切操作，盗取用户的个人重要信息。这是十分可怕的，也给用户带来了不可估量的损失。 研发探究这些病毒木马的原理及其行为，制作出相应的安全工具来对抗它们，显得十分的必要。不像 Unix 等开源系统，Windows 操作系统是不开源的，这对研究其内部的原理增加了一定的难度，制作相应的 Anti-Rootkits 工具也并非易事。但正是由于这些原因，使得对抗当前流行的病毒木马变得更加具有价值和挑战性。

1.3 主要工作

本文主要研究一些主流 Rootkit 的特征和主流技术，并给出检测方案以及完整的代码实现，主要内容如下：

1. 研究 inline HOOK 以及各种内核钩子主流实现技术及检测、恢复技术。
2. 研究进程的主流隐藏技术以及枚举技术。
3. 研究注册表的文件格式及解析技术。
4. FSD 层次上文件的删除，枚举，文件解锁等技术细节和实现。
5. 研究进程自我保护技术，以及结束进程技术。
6. 研究 NTFS 文件系统的数据组织结构以及 FAT 文件系统的数据组织结构。
7. 研究各种系统高级信息的获取。
8. 研究使用 RSA 非对称加密算法实现的文件数字签名技术。
9. 内核级网络端口枚举。

第二章 Windows 应用层开发

2.1 PE 格式及加载重定向

可执行文件 (executable file) ，可移植可执行 (PE) 文件格式的文件，它可以加载到内存中，并由操作系统加载程序执行。它可以是 .exe 文件 .sys 文件 .com 文件 .bat 文件等。在本文内除特别说明外所说的 PE 文件主要指以下 3 类，.sys、.exe、.dll。其中 SYS 文件是运行于内核态的驱动文件，exe 就是常见的可执行程序，而 dll 则是动态链接库，这三类文件有着相同的文件结构，区别只是在一些域的具体值，比如表征执行环境的域。完整解析 PE 文件的各个结构的细节不是本论文的范畴，大致的 PE 整体结构图 2-1 如下：

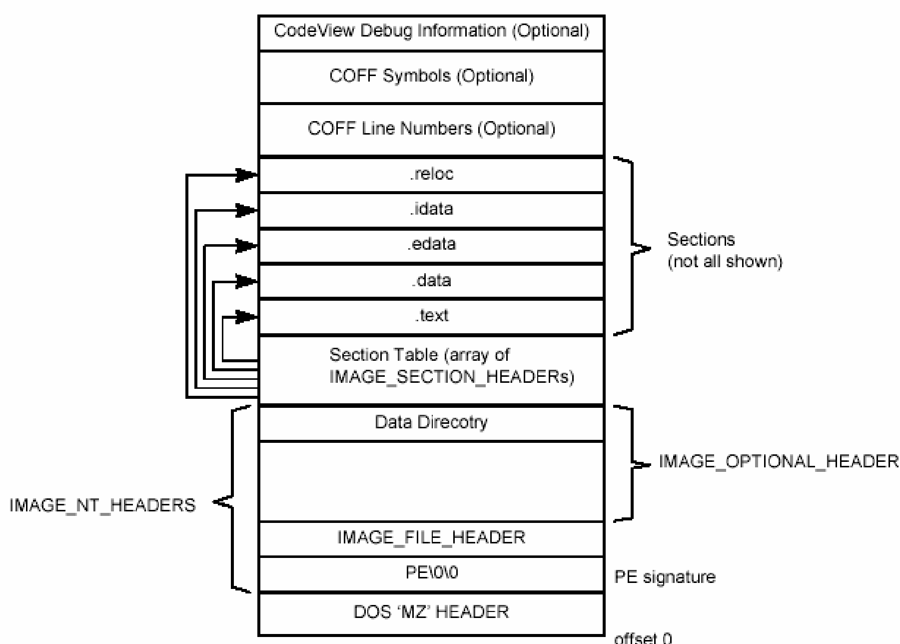

图 2-1 PE 结构示意图

为什么 PE 文件格式对反 Rootkit 很重要？具体来说，目前各种各样的钩子都是基于 PE 文件格式来进行的，期间还涉及到各种各样的 PE 文件逆向。存在于磁盘上的可执行文件和已经放到内存里的可执行镜像是不一样的，由于内存对齐和文件对齐以及加载基地址等影响，导致必须要对文件进行重定位。可执行程序在运行前必须要加载进内存，当然进程的创建是及其复杂而精细的过程，详细罗列不太现实，主要看加载进内存及重定位这个过程。

操作系统在加载可执行程序时，首先使用 NtCreateFile 打开相应的文件，之后创建文件映射 Section，接着进行重定位，重定位工作是由系统 ntdll.dll 在用户态完成的。而重定位需要的一些重要信息，如有那些指令使用直接选址等，这些信息保存在 PE 文件的重定位表里面。程序里的所有指令都是基于头部的 ImageBase 地址来说的，如果加载地址不在这个预定义的地址上时就必须执行重定位。

重定位表由一个个的重定位块组成，如果重定位表存在的话，必定是至少有一个重定位块。因为每个块只负责定位 0x1000 大小范围内的数据，因此如果要定位的数据范围比较大的话，就会有多个重定位块存在。

每个块的首部是如下定义：

```
typedef struct _IMAGE_BASE_RELOCATION {
DWORD      VirtualAddress;
DWORD      SizeOfBlock;
} IMAGE_BASE_RELOCATION;
```

把内存中需要重定位的数据按页的大小 0x1000 分为若干个块，而这个 VirtualAddress 就是每个块的起始 RVA.只知道块的 RVA 当然还不行，还要知道每一个需要重定位数据的具体地址。每个需重定位的数据其地址及定位方式用两个字节来表示，记为 RelocData，紧跟在 IMAGE_BASE_RELOCATION 结构之后。winodowsPE 文件重定位过程如图 2-2 所示：

默认加载地址是 0x400000

PE 转载器将其
加载在 0x600000 处

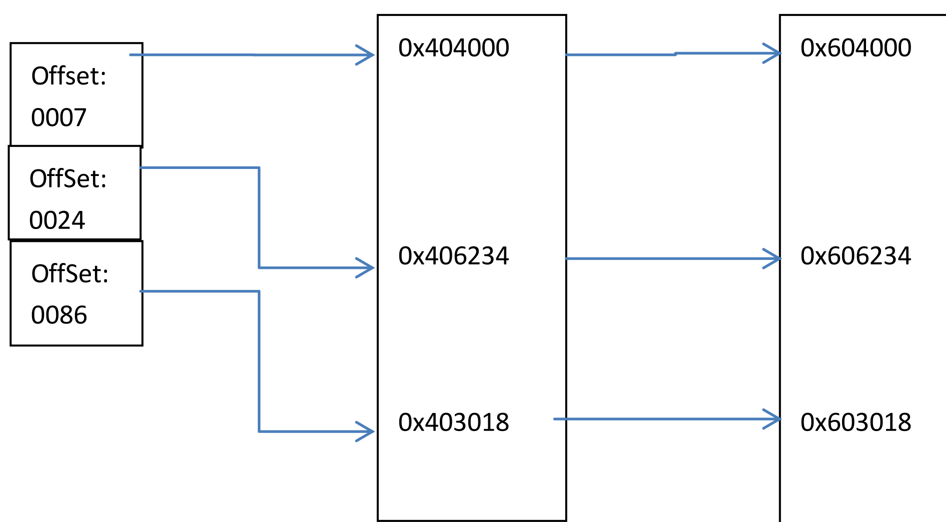

图 2-2 PE 重定位示意图

对于 ARK 来说，能够编程实现文件重定位是必备技能，因为一旦系统感染 Rootkit 之后，内存里面的很多数据已经不可信任了，而文件却相对可信些，我们需要通过对比文件和内存镜像内容的差异来查找内核被修改的蛛丝马迹。但是内存里的是经过重定位的，不能直接对比，需要对文件进行重定位后才能进行比对。

除了重定向外，PE 文件的导出引入表也是 ARK 关心的对象之一，导出表保存 PE 文件通过导出函数的方式提供对外 API 接口的相关信息。而操作系统内核导出了数量巨大的函数或者变量，Rootkit 可能会修改这些导出表的信息来实现函数的拦截，进而更改函数流程，以便实现恶意目的。导入表也是 PE 重要的格式，在 ring3 下感染导入表可以使受感染者加载恶意 dll，在内核里修改 IAT 表也可实现函数拦截。文件重定位技术在后面的 inline 钩子检测里经常用到，这里提及一个问题就是文件的读取方式，注意在 Windows 上读取文件有内存映射和直接使用 ZwReadFile 这类文件操作函数获取的区别，由于文件系统对文件处理的统一性，同一个文件一般系统没打算在内存中保留两个副本，如果使用内存映射，仅仅只是会使用内存中已经存在的这个副本，而不是新建一个映射，内存映射可以略去一些复杂的偏移转换，但是镜像可能和磁盘上的内容不一样，所以使用这种方法时要慎重而行。详细的 PE 文件重定位可以参考文献[1]。

2.2 原生 API 的使用

原生 API（native API）是 Windows 不公开的编程接口，大部分原生 API 来自 ntdll 动态链接库，Kernel32 动态链接库只是对 ntdll 的封装和包裹，图 2-3 是 kernel32 的 ReadFileEx 的大致调用流程，对于一个非常成熟的现代操作系统 Windows 来说，其调用层次是相当复杂的：

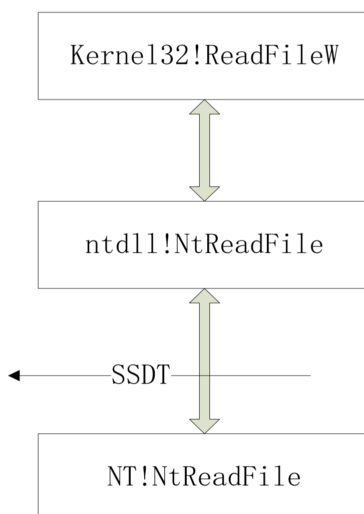

图 2-3 API 调用流程

当然了一些功能强大的原始 API 并没有在 Kernel32 里暴露，所以使用原生 API 可以实现一些很强大的功能。原生 API 的资料主要来自世界各地 Windows 爱好者对 windwos 进行的一些逆向工程和微软给驱动程序预留的编程接口 WDK。

原生 API 的数目和结构是不完全确定的，微软也不会保证原生 API 的版本兼容性，事实证明，Windows 的不同版本的 Native API 数目和功能都是有变化的，参数结构也有变化。典型的一个例子就是 NtQuerySystemInformation 函数，看一下此函数你就会发现 Native API 的强大之处。此函数是最重要的 native API 之一，可以查询系统个 70 多种信息，其中包括反 Rootkit 关心的进程集合信息，驱动信息，句柄信息等。但是使用 native API 是有风险的，比如我自己调试结果来看，对于进程集合的信息，在 XP 和 2000 上数据结构有些变化。所以正确使用 native API 不但要看前人的资料还要善于自我发掘。目前从网上可以找到比较相对完整的 native API 声明列表，但是对于不断发展中的 Windows 系统来说，这些资料只是冰山一角。要开开发底层应用，例如反 Rootkit，Nativa API 是必须要要知道的。

这里需要说的是，其实 Nativa API 就是对应内核的 SSDT 表（系统服务分派表），有一些函数对应了内核的导出函数，但是另外一些却没有导出，典型的就是 NtTerminateThread，当然了一种调用 SSDT 表函数的通用方法是在内核靠挂到一个一个非系统进程，读取 PEB 得到 ntdll 模块基地址，定位到其导出表，获取函数开头字节，得到相应的系统调用号。当然了目前许多成熟的杀毒软件都是使用对不同的系统硬编码系统需要使用的系统调用号。文献[2]给了一个相对完善的 Na ïve API 的参考，也可以参考在线文档 http://undocumented.ntinternals.net。

2.3 本软件开发基础

Rootkit 和反 Rootkit 的编写除了需要具备普通的 win32 编程外还学要知道一些基本的编程技巧。主要罗列几个如下：

1. 进程执行权限

在 Vsita 及以上操作系统中，普通的程序运行后只有普通的权限，普通权限有很多限制，比如不能打开系统关键进程，如 smss 等，一般需要将自身提升到调试权限，这样可以打开大部分的进程。但是你还是会发现在 Windows7 下无法打开 System 进程以及 audiodg.exe 进程，原因就是在 Vista 之后，增加了保护进程的概念，而常见的保护进程就是这两个，区分普通进程和保护进程的域在进程控制块里面，用户态无法访问到这个域。当然了，除了在进程控制块上有区别，在文件上也是有区别的，和普通的 PE 文件不同的是多了一个 Section。将自己的进程标志

为保护进程需要向微软申请，但是我们可以修改进程控制块里面的这个域来提升自己到保护进程。普通进程打开保护进程的权限只有有限的查询权限，也就是说就算是以 Xp 下操作进程的最小权限 PROCESS_QUERY_INFORMATION 也是打不开的，只有同时为保护进程的进程间才能实现打开操作。

对于反 Rootkit 工具来说，Vista 以上需要以管理员权限运行，才能操作大部分进程，这个可以通过编译时增加 UAC 请求级别来实现，在 VS2008 开发环境下修改如图 2-4 选项。这样在 Vista 以上，程序就会自动增加那个盾牌小图标，点击之后 UAC 就会提示用户是否运行。还有 UAC 隔离机制，UAC 隔离会导致传统进程间通信失败，包括一些消息发送也会失败，所以要在 VISTA 和 Windows 下系统下实现低权限的进程和管理员权限的进程通信有时需要降低内核对象的安全级别。

2. 获取模块（文件）信息

作为一个 ARK，在给用户展现文件时，提供一个丰富的文件信息是很有必要的，所以一般的 ARK 不但要求能够给出文件的来源，日期等一些相关信息，甚至需要能够通过联网来获取信息。Windows 下普通的属性对话框如图 2-5 所示，对于 ARK 来说，在这里面主要关心的是创建日期，但是更为重要的数字签名信息。一般正规的软件文件都会带有有效的数字签名，而恶意软件一般没有数字签名，当然带有数字签名的恶意软件也是有的，这就要求在获取信息时能够然用户很好的通过互联网来获取文件足够多的信息。

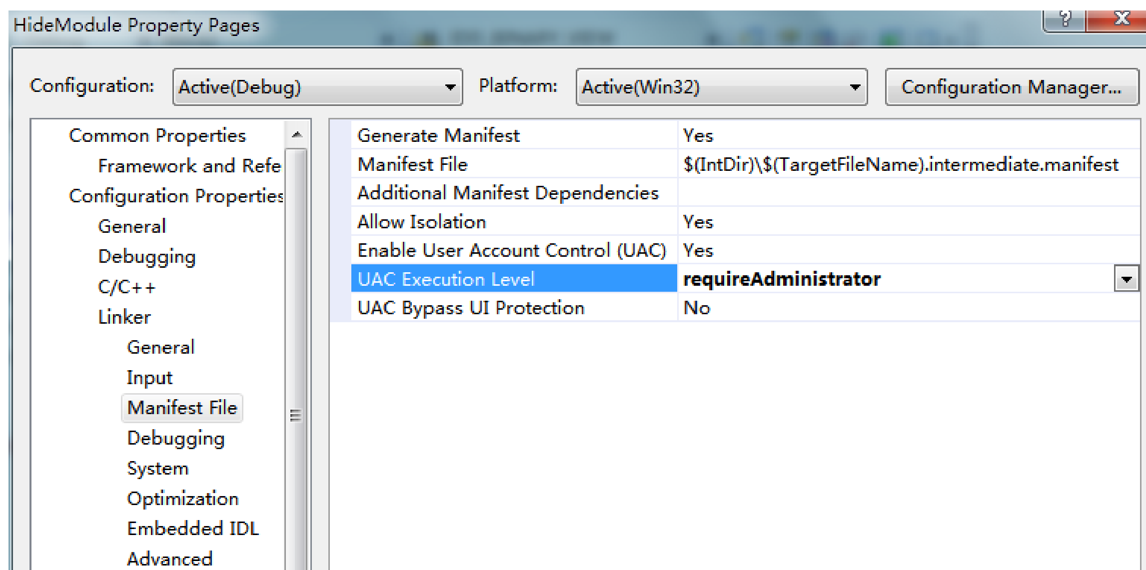

图 2-4 VS2008UAC 选项卡

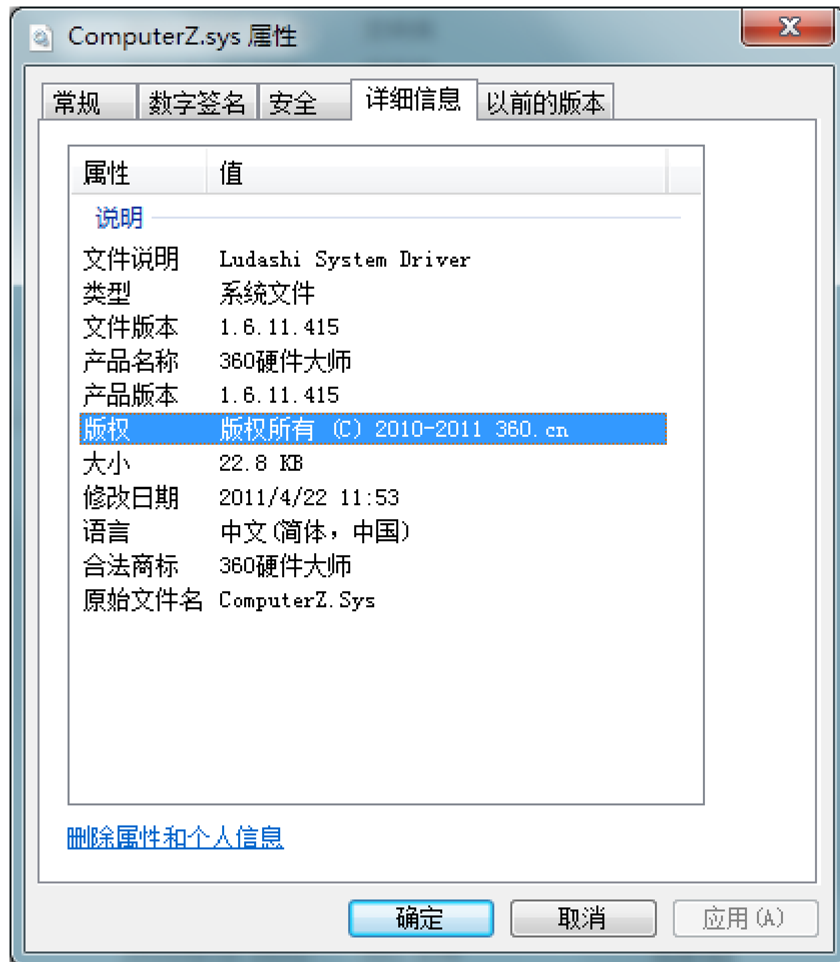

图 2-5 文件信息窗口

3. IDA 和 windbg 的使用

IDA 是进行逆向工程，发掘 Windows 内核数据结构的有力武器，在后面的各章里都会经常有它的身影出现。Windbg 主要用来查看运行中的内核已经进行驱动源代码级调试。

4．DIA 解析 pdb 文件

DIA 是微软提供的解析 pdb 文件的开发包，在获取一些内核函数和符号信息时非常有用。后面会讨论使用这个来获取内核的一些未导出函数的地址及长度方便内核调用。本软件使用这个技术来检测未导出函数的 inline 钩子，详细技术可参考 MSDN 有关 DIA 的细节。

4. 数字签名验证

数字签名技术可以用于快速排除 windows 自身文件及进程，有助于用户快速定位异常进程及文件，此技术在本软件里大量使用。

2.4 程序界面拟合

本软件涉及的内容较多，选择 Outlook 样式的展示风格能获得不错的展示效果。从 Win32 直接入手难度比较大，这里选择从 MFC 作为基础开始，开发整体界面元素，当前的界面风格如图 2-6。

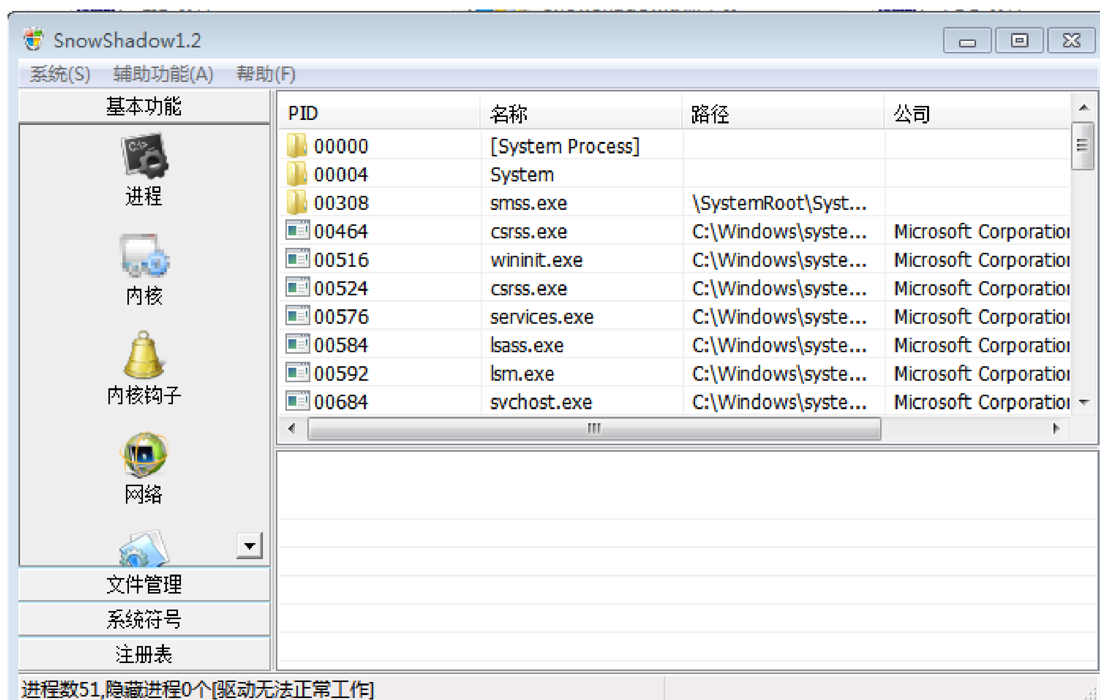

图 2-6 程序界面

这个界面简洁，方便找到各个功能。为了打造这个界面在 VS2008 下稍有难度，原因就是 VS2008 还没有提供 Outlook 风格的导航栏，Outlook 导航栏便于向里面添加多个功能。所以从国外开源的界面库里面把这个代码挖了来，由于是基于对话框的程序，为了支持窗口位置调整，需要处理窗口大小变化消息，合理摆放各个控件的位置。其次，列表框需要有高亮显示的功能，甚至树控件也有这种需求，后面需要用来区分签名进程和非签名进程，高亮显示有助于突出重点，在 ARK 里就是有助于快速排除正常文件，进程等。

2.5 本章小结

本章主要描述了在开发用本软件时用户态的一些基本技术，其中 PE 文件的加载及重定位很重要，这是实现任何一个反 Rootkit 工具最基本的东西，另外还提到了一些在反 Rootkit 技术里面用户态的一些技术细节，最后对交互界面 GUI 进行了初步的要求。

第三章 Windows 驱动开发

3.1 Windows 驱动模型

IA32 从 386 开始就进行了任务特权级别的划分，从内到外分为 4 个等级，分别是 ring0，ring1，ring2，ring3，如图 3-1，Windows 系统只用了其中的两层，就是下图中的 ring0 和 ring3，ring0 主要运行操作系统系统的一些核心组件和驱动模块，一般的 Rootkit 也是运行在这个环内。注意外层的软件是不能直接访问内环的资源的，所以为了清除 Rootkit，必须要进入 ring0，而进入 ring0 的唯一手段就是使用驱动程序，作为一个 ARK，驱动决定了其大部分功能，也是一个 ARK 内精华之所在，后面的所有章节都会以这个为基础而展开。

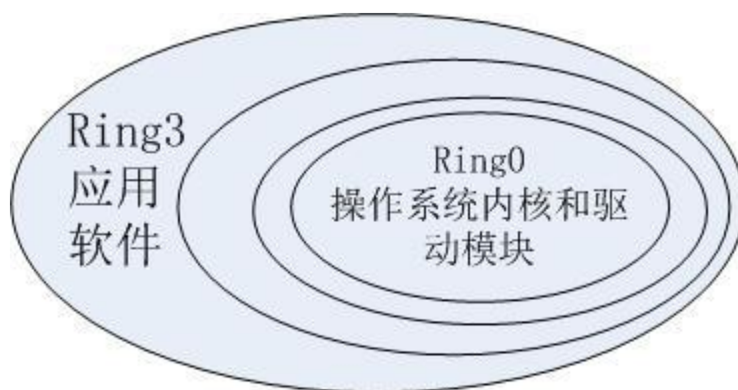

图 3-1 IA32 权限示意图

为了开发驱动，我们需要安装 WDK(Windows Driver Kits),以前也叫 DDK，WDK 里提供的开发工具几乎都是在控制台下运行的，所以在写代码阶段可以使用 VS 系列来进行代码编辑和编译。写这个软件时所使用的 WDK 是 7600 版本，集成开发环境是 VS2008。Windows 下驱动目前至少可以分为 3 大类，Nt 式驱动，以前遗留下来的格式，特点是简单快捷，但是不支持 PNP。WDM 驱动，即 Windows Driver Mode，支持 Windows 的电源管理和 PNP 以及 WMI，最新的模型是 WDF，WDF 隐藏了很多操作上的细节，使用面向对象的思想。 对于常见的 Rootkit 和大部分的 ARK 都是使用 NT 式驱动,因为里面涉及一些细节处理,WDF 很显然不太适合,而 WDM 过于臃肿，除了硬件驱动一般需要电源管理。WDK 能编译不同驱动模型的驱动，对于 Windows7 和 XP 等的这些版本，一般向后兼容，除非使用了新增加的 API，在使用这类 API 时推荐的一个做法是动态获取 API 地址，而不是静态连

接，静态连接时会放到 PE 文件的导入表里面，在加载时这个导入函数没有得到处理就会导致失败。在参考文献[3]可以找到通俗的驱动开发教程。

Windows 下进程高两 G 内存空间属于内核和驱动使用的空间，低两 G 是进程私有空间，在不同区域的内存还有不同的用途，32 位内存分布具体见图 3-2。

图 3-2 windows 内存空间分布示意图

其中本软件的驱动运行于高 2G 空间，交互界面运行于用户空间。熟习 Windows 的内核空间分布对于一些暴力搜索系统信息的技术来说很有用，可以避免许多不必要的工作。

3.2 开发准备

VS2008 开发驱动首先要将 WDK 的头文件和 lib 文件添加到 VC++目录里面，之后创建一个解决方案，基于控制台，配置环境如下：

1、在"调试信息格式"中选择 C7 兼容(/Z7)"选项。

2、在"警告等级"中选择"3 级(/W3)"。

3、在"将警告视为错误"中选择"是(/WX)"。

4、在"优化"选项中，在"优化"中选择"禁用(/Od)"。

5、在"预处理器"选项中，在"预处理器定义"中添加 WIN32=100;_X86_=1;
WINVER=0x501;
DBG=1"。并去除"从父级或项目默认属性继承"复选框的勾选。

6、在"高级"选项中，选择"调用约定"为"__stdcall (/Gz)"。

7、选择"链接器"并展开内部选项。

8、在"输入"选项中，添加"附加依赖项"ntoskrnl.lib 并去除"从父级或项目默认设置继承"复选框的勾选。

9、在"清单文件"选项中，选择"生成清单"为否。

10、在"调试"选项中，选择"生成调试信息"为"是 (/Debug)"。

11、在"系统"选项中，选择"子系统"为"本机 (/SUBSYSTEM:NATIVE)"。选择"驱动程序"为"驱动程序(/DRIVER)"。

12、在"高级"中，添加"入口点"为 DriverEntry。选择"随即基址"为"默认值"。选择"数据执行保护"为默认值。选择"目标计算机"为"MachineX86(/MACHINE:X86)"。

3.3 调试设置

驱动程序需要双机调试，比较经济的做法是在虚拟机里运行驱动，在本机运行内核调试器 windbg，虚拟机使用 Virtual PC.通信使用管道，主机上的 VirtualPC 需要设置如下：

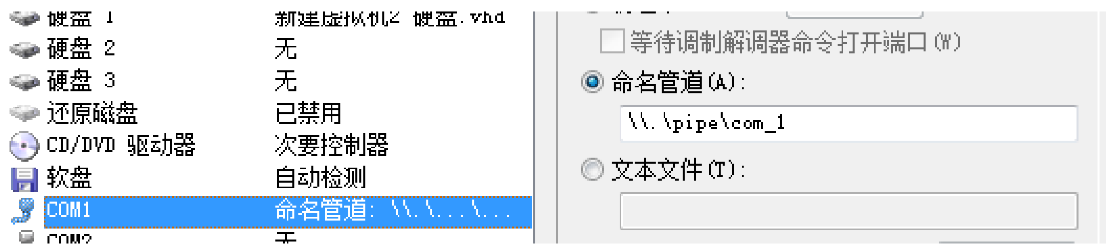

图 3-4VirtualPC 调试设置

之后要在虚拟机里面的操作系统里开启调试选项，调试版本速度要比发布版本慢，所以一般是重现增加一个启动配置。Windbg 设置启动参数设置如下：

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

对于 Virtual PC 里面的虚拟机可以使用 msconfig 来开启调试。切换到 BOOT.ini，选择高级选项就可以开启调试了，windows 7 和 windows XP 大致相同。

3.4 本章小结

本章主要描述开发本软件的核心驱动所用到的一些基础知识，windows 内核是 Rootkit 和反 Rootkit 的必争之地，谁拥有了制高点才能主宰这场战争，今年，Rootkit

的技术越做越底层，进入内核只是反 Rootkit 的第一步。

第四章 进程管理

4.1 进程与线程

进程和线程是现代操作系统的重要概念，Windows 也不例外，在 Windows 下，CPU 资源的调度单位是线程，而不是进程，进程充当线程的容器。真正的执行体单位是线程。进程在内核态进程表现为一个 EPROCESS 结构，线程则表现为 ETHREAD 结构，而进程控制块 PCB 位于 EPROCESS 的第一个域，线程控制块 TCB 位于 ETHREAD 的第一个域，Windows7 下其部分结构如图 4-1，图 4-2。

```
struct _EPROCESS // 0x260
{
    struct _KPROCESS Pcb; // +0x0(0x6c)
    struct _EX_PUSH_LOCK ProcessLock; // +0x6c(0x4)
    union _LARGE_INTEGER CreateTime; // +0x70(0x8)
    union _LARGE_INTEGER ExitTime; // +0x78(0x8)
    struct _EX_RUNDOWN_REF RundownProtect; // +0x80(0x4
    void* UniqueProcessId; // +0x84(0x4)
    struct _LIST_ENTRY ActiveProcessLinks; // +0x88(0x8
    ULONG QuotaUsage[0x3]; // +0x90(0xc)
    ULONG QuotaPeak[0x3]; // +0x9c(0xc)
    ULONG CommitCharge; // +0xa8(0x4)
    ULONG PeakVirtualSize; // +0xac(0x4)
    ULONG VirtualSize; // +0xb0(0x4)
    struct _LIST_ENTRY SessionProcessLinks; // +0xb4(0x
```

图 4-1 EPROCESS 部分结构

```
struct _ETHREAD // 0x258
{
    struct _KTHREAD Tcb; // +0x0(0x1c0)
    union _LARGE_INTEGER CreateTime; // +0x1c0(0x8)
    DWORD NestedFaultCount; // +0x1c0(0x4)
    DWORD ApcNeeded; // +0x1c0(0x4)
    union _LARGE_INTEGER ExitTime; // +0x1c8(0x8)
    struct _LIST_ENTRY LpcReplyChain; // +0x1c8(0x8)
    struct _LIST_ENTRY KeyedWaitChain; // +0x1c8(0x8)
    long ExitStatus; // +0x1d0(0x4)
    void* OfsChain; // +0x1d0(0x4)
    struct _LIST_ENTRY PostBlockList; // +0x1d4(0x8)
    struct _TERMINATION_PORT* TerminationPort; // +0x1dc(0x4)
    struct _ETHREAD* ReaperLink; // +0x1dc(0x4)
    void* KeyedWaitValue; // +0x1dc(0x4)
    ULONG ActiveTimerListLock; // +0x1e0(0x4)
    struct _LIST_ENTRY ActiveTimerListHead; // +0x1e4(0x8)
    struct _CLIENT_ID Cid; // +0x1ec(0x8)
    struct _KSEMAPHORE LpcReplySemaphore; // +0x1f4(0x14)
    struct _KSEMAPHORE KeyedWaitSemaphore; // +0x1f4(0x14)
```

图 4-2 ETHREAD 部分结构

对于各个版本的 Windows，EPROCESS 和 ETHREAD 都是经常在变化的。EPROCESS 里面有一个进程环境块，PEB，在 Windows7 在 EPROCESS 偏移 0x1b0 处，这个结构对反 Rootkit 稍有用处，其结构如图 4-3。

```
struct _PEB // 0x210
{
    BYTE InheritedAddressSpace; // +0x0(0x1)
    BYTE ReadImageFileExecOptions; // +0x1(0x1)
    BYTE BeingDebugged; // +0x2(0x1)
    BYTE SpareBool; // +0x3(0x1)
    void* Mutant; // +0x4(0x4)
    void* ImageBaseAddress; // +0x8(0x4)
    struct _PEB_LDR_DATA* Ldr; // +0xc(0x4)
    struct _RTL_USER_PROCESS_PARAMETERS* ProcessParameters;
```

图 4-3 PEB 部分结构

这个结构里面的 ProcessParameters 进去可以找到进程的启动参数，进程路径，窗口标题等信息。而 Ldr 进去是图 4-4 结构

```
struct _PEB_LDR_DATA // 0x28
{
    ULONG Length; // +0x0(0x4)
    BYTE Initialized; // +0x4(0x1)
    void* SsHandle; // +0x8(0x4)
    struct _LIST_ENTRY InLoadOrderModuleList; // +0xc(0x
    struct _LIST_ENTRY InMemoryOrderModuleList; // +0x14
    struct _LIST_ENTRY InInitializationOrderModuleList;
    void* EntryInProgress; // +0x24(0x4)
};
```

图 4-4 PEB_LDR_DATA 完整结构

其中的三条链表来自如下结构的开头三条（图 4-5）。

```
struct _LDR_DATA_TABLE_ENTRY // 0x50
{
    struct _LIST_ENTRY InLoadOrderLinks; // +0x0(0x8)
    struct _LIST_ENTRY InMemoryOrderLinks; // +0x8(0x8)
    struct _LIST_ENTRY InInitializationOrderLinks; // +0
    void* DllBase; // +0x18(0x4)
    void* EntryPoint; // +0x1c(0x4)
    ULONG SizeOfImage; // +0x20(0x4)
    struct _UNICODE_STRING FullDllName; // +0x24(0x8)
    struct _UNICODE_STRING BaseDllName; // +0x2c(0x8)
    ULONG Flags; // +0x34(0x4)
    WORD LoadCount; // +0x38(0x2)
    WORD TlsIndex; // +0x3a(0x2)
    struct _LIST_ENTRY HashLinks; // +0x3c(0x8)
    void* SectionPointer; // +0x3c(0x4)
    ULONG CheckSum; // +0x40(0x4)
    ULONG TimeDateStamp; // +0x44(0x4)
    void* LoadedImports; // +0x44(0x4)
    void* EntryPointActivationContext; // +0x48(0x4)
    void* PatchInformation; // +0x4c(0x4)
};
```

图 4-5LDR_DATA_TABLE_ENTRY 完整结构

这个结构是进程已经加载的模块的列表，遍历这个列表可以得到进程模块列表，这个也是枚举进程模块的一个方法，后面还会说道。EPRCESSS 和 ETHREAD 里面还有很多很有意思的域，从里面可以取到一些对 ARK 很有意义的数据。例如在 EPROCESS 里面有个 ActiveProcessLinks 的成员，这个成员连接着当前活动的进程双向链表，遍历这个链表可以枚举进程，任务管理器的枚举进程就是来看这个链表获取到的。EPROCESS 里里面有好几条链表连接 着 ETHREAD 结构，ETHREAD 反过来也有成员指向 EPROCESS，所以得到 EPROCESS 结构之后，我们可以进一步得到这个进程的所有 ETHREAD 结构，同理找到一个 ETHREAD 结构也可以找到这个这个线程所在的进程，在后面还会有讲述。ETHREAD 和 EPROCESS 结构都是比较大型的结构，详细叙述要求大量篇幅，也不属于本论文的重点，所以本论文不展开细说了。

4.2 进程隐藏技术

进程隐藏是 Rootkit 一个最典型的特征，而且目前进程隐藏也很成熟了，常见的进程隐藏技术如下：

第一类：挂接系统调用实现。

第二类：直接修改内核数据结构隐藏进程。

第三类：将 Rootkit 制作成 dll。

网上主流的软件，例如用来隐藏调试器的 HideTools 主要使用第一种方法，主要使用 SSDT 钩子或者 Inline 挂钩 NtQuerySystemInformation 函数来实现隐藏，对于这一类的隐藏进程，可以在用户态可以通过使用 OpenProcess 使 PID 从 0-99999 来检测，为了对抗这种检测方法，一般的 Rootkit 在挂接 NtQuerySystemInformation 的同时还会挂接 NtOpenProcess、ObReferenceObjectByHandle 等函数来阻止打开进程。挂接函数来实现隐藏一个很明显的缺点就是会有钩子痕迹，在另一个方面暴露的 Rootkit 的存在。EPROCESS 里面有个 ActiveProcessLinks 链表，NtQuerySystemInformation 函数就是通过遍历这条链表来枚举进程的，所以后面一种方法就是将指定进程从这个双向链表上摘除掉。这这种方法要比第一种方法好，原因就是看不出来明显的修改痕迹。其修改示意图如下（实现对进程 2 的隐藏）：

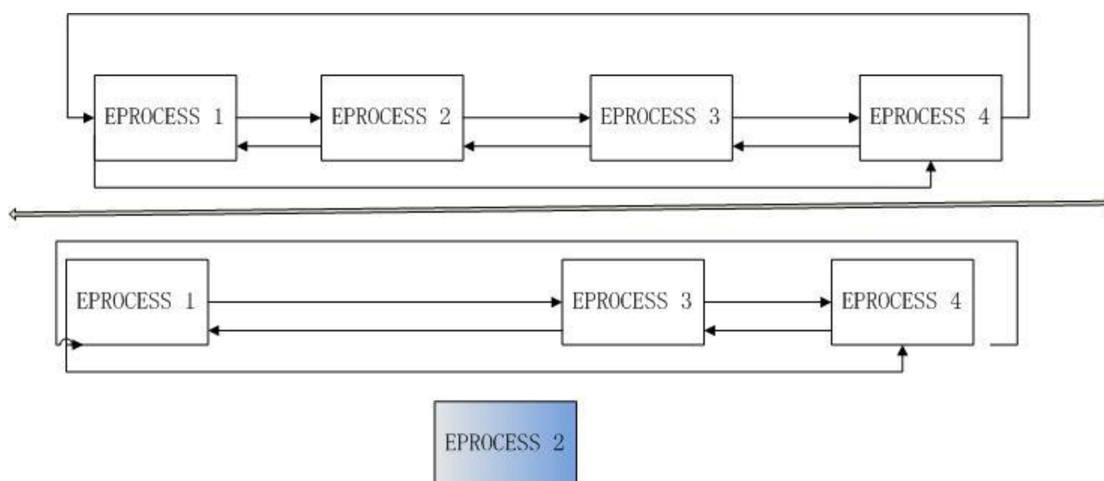

图 4-5 直接内核操作隐藏进程

上面部分是修改前的数据，下面是修改后的数据，通过将进程 EPROCESS 从这个双向链表上摘除来隐藏[4]。除此之外还有一种隐藏方法，启动一个傀儡进程 A，通过将其进程暂停后，加载自身到内存，并且进行重定位后，将傀儡进程的改变傀儡进程的上下文，使其指向自身，这就是进程注入的方法，这样也就进行了进程的隐藏。随着反 Rootkit 技术的发展，纯 exe 的木马其实已经很少了，更多的是 dll 形式的木马。对于 dll 的检测，稍后有详细叙述。

4.3 进程保护技术

现在再来说说进程保护，检测到隐藏进程之后我们要能终止 Rootkit 进程才有意义，如果检测到之后无法结束其进程则检测变得很没有意义。要知道怎么保护进程，先看看怎么结束进程，结束进程常见方法如下：

1) 使用 TerminateProcess 直接结束进程，对于进程句柄的获取，可以通过 OpenProcess 获得，或者枚举 Csrss 的句柄表，这个句柄表里面有所有 Ring3 的程序的句柄，之后复制这个句柄，接着就可以调用 TerminateProcess 结束进程了,需要注意 Vista 以后有两个 Csrss 进程。使用 TerminateThread 结束，过程类似 TerminateProcess，不同的是句柄只能通过 OpenThread 来获取了，线程结束完了，或者主线程结束了，进程也就退出了。
2) 使用 WriteProcessMemory 在进程里写入垃圾数据，之后进程自己崩溃退出。
3) 使用 CreateRemoteThread 创建远线程，在远线程里结束进程。
4) 使用 NtUnmapViewOfSection 卸载掉进程的重要 dll 比如 kernel32 或者 ntdll，进程异常退出。
5) 远程关闭进程里面的句柄，有时也会导致异常退出。
6) 消息攻击，比如发送 WM_CLOSE 消息结束进程，还有其它的消息攻击方式我们这里暂时不讨论，留到第八章再来讨论。

以 TernimateProcess 为例，我们来看一下 Terminate 的调用流程，首先 TernimateProcess 会进入到 Ntdll 里面的 NtTerminateProcess 里面，NtTerminateProcess 经过服务调用，通过 SSDT 进入到内核的 NtTerminateProcess 里面，在 NtTerminateProcess 里面，首先通过 ObReferenceObjectByHandle 从句柄得到相应进程的 EPROCESS,之后使用循环使用 PsGetNextProcessThread 函数来获取这个进程的每一个线程，之后对每个线程依次调用 PspTerminateThreadByPointer 来结束进程，如下图 4-6 所示。PspTerminateThreadByPointer 使用 KeInitializeApc 初始化一个 APC，之后使用 KeInsertQueueApc 来插入这个 APC（异步过程调用）,Apc 对应的函数是 PspExitNormalApc，这个函数里线程进行了自杀，大致代码如图 4-7 所示。注意 Windows 系统的 APC 很特别，当线程有 APC 请求时，线程获得 CPU 时间片后首先执行 APC 函数，这就是为什么在上面插入 APC 之后还要调用 KeForceResumeThread 的原因，注意之所说线程是自杀是因为 Windows 的 APC 是在 ACP 所在的那个线程上下文里执行的。同样 NtTerminateThread 有类似的流程。从上面的流程可知，可以在以拦截以上各个函数的调用来保护自己的进

程。也就是说可以在 SSDT 层挂钩 NtTerminateProcess、Thread 等函数，也可以更进一步挂接 PspTerminateThreadByPointer 函数，或者 KeInsertQueueApc 函数。其中常用的 inline 挂钩的原理如图 4-8。可以在 Detour 函数里直接返回错误而终止服务调用。当然这里没有对 SSDT 钩子进行更多的叙述，毕竟 SSDT 钩子现在已经被用得很烂了。

```
PAGE:004ABCBB 53                          push    ebx
PAGE:004ABCBC C7 45 08 22 01 00 00        mov     [ebp+Handle], 122h
PAGE:004ABCC3 E8 3A B6 FF FF              call    _PsGetNextProcessThread@8 ; Ps(
PAGE:004ABCC8 8B F0                       mov     esi, eax
PAGE:004ABCCA 85 F6                       test    esi, esi
PAGE:004ABCCC 74 1E                       jz      short loc_4ABCEC
PAGE:004ABCCE 83 65 08 00                 and     [ebp+Handle], 0
PAGE:004ABCD2
PAGE:004ABCD2                 loc_4ABCD2:                         ; CODE XREF: Nt
PAGE:004ABCD2 3B F7                       cmp     esi, edi
PAGE:004ABCD4 74 09                       jz      short loc_4ABCDF
PAGE:004ABCD6 FF 75 0C                    push    [ebp+ExitStatus]
PAGE:004ABCD9 56                          push    esi
PAGE:004ABCDA E8 48 77 FF FF              call    _PspTerminateThreadByPointer@8
PAGE:004ABCDF
PAGE:004ABCDF                 loc_4ABCDF:                         ; CODE XREF: Nt
PAGE:004ABCDF 56                          push    esi
PAGE:004ABCE0 53                          push    ebx
PAGE:004ABCE1 E8 1C B6 FF FF              call    _PsGetNextProcessThread@8 ; Ps(
```

图 4-6 NtTerminateProcee 部分反汇编代码

```
KeInitializeApc(v7, a1, 0, PsExitSpecialApc, PspExitApcRundown, PspExitNormalApc, 0, a2);
if ( (unsigned __int8)KeInsertQueueApc(v8, v8, 0, 2) )
{
  KeAlertThread(a1, 0);
  KeForceResumeThread();
}
```

图 4-7 PspTerminateThreadByPointer 伪代码

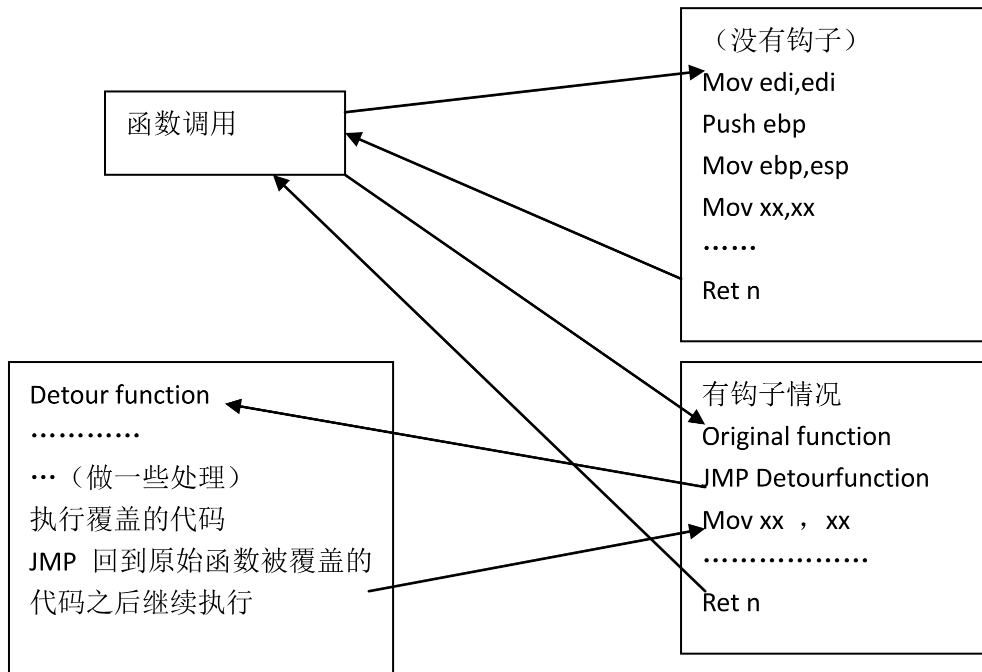

图 4-8 inline 钩子示意图

除了使用上述函数挂钩方法可以保护进程，在 Windows7 下，微软新增加了两个函数: PsSetCreateProcessNotifyRoutineEx、ObjectPostCallback 这两个函数都可以用来实现进程保护。当然了前面讨论过受保护进程的概念，将其设置为受保护进程也可以用来实现进程的保护。

4.4 用户态枚举隐藏进程

目前在用户态枚举隐藏进程的主流技术有如下几个：

方法一：从进程 ID 从 4-99999 依次调用 OpenProcess，可以称为用户态的暴力搜索，其实这个方案近接遍历了 PspCidTable，不过如果挂接 NtOpenProcess 就可以过滤掉这种暴力枚举方法了。

方法二：使用 NtQuerySystemInformation 来获取所有类型为进程的句柄，之后复制这类句柄到本进程，获取进程路径，从而近接得到进程名和路径。

方法三：从 Csrss 进程里面获取，Csrss 里面保存了 win32 子系统的所有进程的信息，当然从内核态起来的进程不包括在内。这个方法主要来自对 csrsswalker 的逆向。保存进程信息的结构是一个双向链表结构如图 4-9 所示。其中的 CLIENT_ID 结构如下：struct _CLIENT_ID // 0x8

```
{
    void* UniqueProcess; // +0x0(0x4)
    void* UniqueThread; // +0x4(0x4)
};
```

看看两个结构就发现有很多有用的信息，比如说进程句柄，ID 等。遍历这个双向链表就可枚举所有 win32 子系统下的进程。这个链表有一个链表头，叫做 CsrssRootProcess，导出函数 CsrLockProcessByClientId 直接访问了这个链表头，所以枚举应该很简单了。当然了线程也是是可以枚举到的，线程信息对应的结构如图 4-10，所以枚举到线程也可以近接得到进程的。不同之处在于，线程数目比较多，放在单一的一个链表里查询速度会很慢，所以有一个长度是 256 的数组来保存链表头，这个数组名字为 CsrThreadHashTable，注意每个元素是一个 LIST_ENTRY。这样遍历这个 HashTable 里面的所有链表就可以找到所有线程信息了。HashTable 地址的获取可以从 CsrLockThreadByClientId 里面去获得。

```
struct _CSR_PROCESS // 0x60
{
    struct _CLIENT_ID ClientId; // +0x0(0x8)
    struct _LIST_ENTRY ListLink; // +0x8(0x8)
    struct _LIST_ENTRY ThreadList; // +0x10(0x8)
    struct _CSR_NT_SESSION* NtSession; // +0x18(0x4)
    void* ClientPort; // +0x1c(0x4)
    char* ClientViewBase; // +0x20(0x4)
    char* ClientViewBounds; // +0x24(0x4)
    void* ProcessHandle; // +0x28(0x4)
    ULONG SequenceNumber; // +0x2c(0x4)
    ULONG Flags; // +0x30(0x4)
    ULONG DebugFlags; // +0x34(0x4)
    ULONG ReferenceCount; // +0x38(0x4)
    ULONG ProcessGroupId; // +0x3c(0x4)
    ULONG ProcessGroupSequence; // +0x40(0x4)
    ULONG LastMessageSequence; // +0x44(0x4)
    ULONG NumOutstandingMessages; // +0x48(0x4)
    ULONG ShutdownLevel; // +0x4c(0x4)
    ULONG ShutdownFlags; // +0x50(0x4)
    struct _LUID Luid; // +0x54(0x8)
    void* ServerDllPerProcessData[0x1]; // +0x5c(0x4)
};
```

图 4-9 CSR_PROCESS 完整结构

```
struct _CSR_THREAD // 0x38
{
    union _LARGE_INTEGER CreateTime; // +0x0(0x8)
    struct _LIST_ENTRY Link; // +0x8(0x8)
    struct _LIST_ENTRY HashLinks; // +0x10(0x8)
    struct _CLIENT_ID ClientId; // +0x18(0x8)
    struct _CSR_PROCESS* Process; // +0x20(0x4)
    void* ThreadHandle; // +0x24(0x4)
    ULONG Flags; // +0x28(0x4)
    ULONG ReferenceCount; // +0x2c(0x4)
    ULONG ImpersonateCount; // +0x30(0x4)
};
```

图 4-10 CSR_THREAD 完整结构

很显然最后一种方法是比较新颖的，也比前面几种得到的信息多。

4.5 内核态枚举隐藏进程

在 EPROCESS 里面含有很多调链表，遍历这些链表可以得到进程列表，常用的几个链表如下:ActiveProcessLinks 链表，前面已经说过了，还有 SessionProcessLinks 链表也可以，PCB 里面的 ProcssListEntry，HandleTable 也可以。前面说过了对这些链表的摘除是可以实现隐藏进程的，但是有可能 Rootkti 没摘干净，所以可以用这些链表来做些参考。

另外一种方法是遍历线程调度链表得到线程集合，再从线程获取到进程。windows 执行的基本单位是线程，而不是进程，所以才有从进程链表上摘除自身的进程隐藏方法，这是虽然从进程链表上摘除了自身，但不会影响操作系统的调度，所以不影响程序运行。但是不能从线程链表上摘除自身，除非程序不想获得 cpu 时间了。在内核里和线程调度有关的结构是 TCB，在 ETHREAD 里的 KTHREAD 里，和线程调度有关的域有如下几个：

```
Typedef _KTHREAD {……
        LIST_ENTRY WaitListEntry；
        LIST_ENTRY QueueListEntry;
        LIST_ENTRY ThreadListEntry; …… };
```

在 windows xp 下，windows 线程分派器使用 pKiDispatcherReadyListHead 和 pKiWaitListHead 这两个结构来调度线程。它的线程分派主要利用 KTHREAD 中的 WaitListEntry 、QueueListEntry 和 ThreadListEntry 来完成，这三个成员变量代表了

三种不同的线程状态，是三个双向链表结构，ThreadListEntry 队列是处于准备状态的，其余两个处于等待状态，如图 4-12。pKiDispatcherReadyListHead 是指向一个有 32 个元素的 LIST_ENTRY 数组，Windows 线程有 32 个优先级，所以有 32 个节点。每个数组元素连接一个双向链表。如图 4-11 所示。基于线程调度链表的隐藏进程检测，就是遍历这个两个链表获取线程，再通过 ThreadProcess 域得到进程的 EPROCES。如何得到 pKiDispatcherReadyListHead 和 pKiWaitListHead？使用 KiDispatcherReadyListHead 的例程有：KeSetAffinityThread、KiFindReadyThread、KiReadyThread 、 KiSetPriorityThreadNtYieldExecution 、 KiScanReadyQueues 、 KiSwapThread。系统中用到 KiWaitInListHead 的例程有：
KeWaitForSingleObject() 、 KeWaitForMultipleObject() 、 KeDelayExecutionThread 、 KiOutSwapKernelStacks。

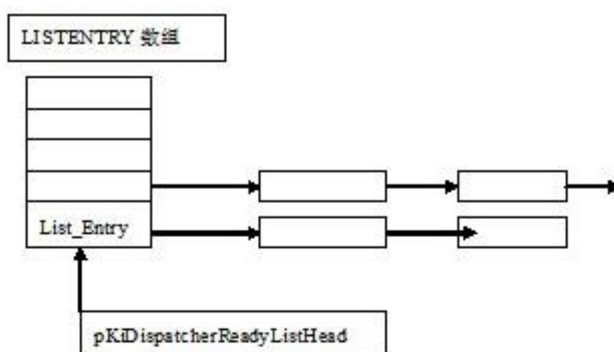

图 4-11

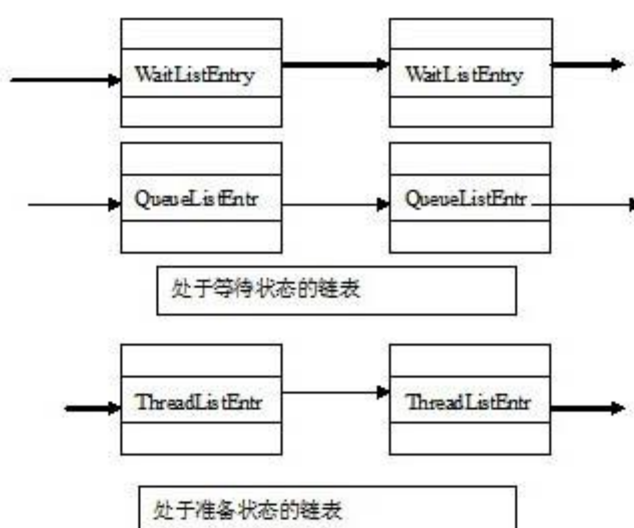

图 4-12

现在普遍使用的方法是遍历 PspCidTable，例如著名的 ARK 冰刃就是使用这种方法来查找隐藏进程，PspCidTale 里面不但有进程信息，还有线程信息，PspCidTable 其实是一个句柄表，与普通的句柄表的格式是完全一样的，但是它与每个进程私有的句柄表有所不同，PspCidTable 中存放的对象是系统中所有的进线程对象，其索引就是 PID 和 TID；而普通句柄表的索引是句柄(HANDLE),PspCidTable 表中存放的直接是对象体，而每个进程私有的句柄表则存放的是对象头（OBJECT_HEADER）；PspCidTable 是一个独立的句柄表，而每个进程私有的句柄表以一个双链链接起来的。

对于 PspCidTable 句柄表的获取有多种方式，常用的方式就是从使用过此句柄表的函数中进行特征搜索，即通过进行内存特征定位的方式来查找。常见的方法是从 PsLookupProcessByProcessId 里面去暴力搜索地址，但是因为 PsLookupProcessByProcessId 有可能会被挂钩，所以可以通过检索内核导出函数列表，找到 PsLookupProcessByProcessId，从文件读取 PspCidTable 的位置。如图 4-13 所示。

```
PAGE:0049CE24 64 A1 24 01 00 00        mov     eax, large fs:124h
PAGE:0049CE2A FF 75 08                 push    [ebp+arg_0]
PAGE:0049CE2D 8B F0                    mov     esi, eax
PAGE:0049CE2F FF 8E D4 00 00 00        dec     dword ptr [esi+0D4h]
PAGE:0049CE35 FF 35 E0 95 48 00        push    _PspCidTable
PAGE:0049CE3B E8 9B 1D FF FF           call    _ExMapHandleToPointer@8
PAGE:0049CE40 8B D8                    mov     ebx, eax
```

图 4-13PsLookupProcessByProcessId 部分反汇编代码

在上面的地址分配中就可以很清晰的看出，PspCidTable 句柄表的地址就是 0xff35 和 0xe8 之间的 4 个字节。就可以利用这个特征来寻找此句柄表。学习别人对句柄表的存储算法逆向并且实现手工解析是一件乏味而意义不大的工作，可以使用系统例程 ExEnumHandleTable 函数来实现句柄表的遍历。在 PspCidTable 里面的进程可能有死进程，怎么判断进程是死的有多种方法，比如看退出状态等。

还有一种检索隐藏进程的方法是暴力搜索内存，EPROCESS 和 ETHREAD 结构都存在于高 2G 的内存空间中。暴力搜索这高 2G 空间获取所有的 EPROCESS 结构或者 ETHREAD 结构，有多种方法来进行特征匹配，业内已经有比较成熟的匹配方法了。本软件使用查找所有 ETHREAD 结构的方法来进行暴力搜索线程，近

接找到相应进程。

4.6 强制终止进程

强制终止进程，方法有很多，比较标准的是遍历进程的所有线程，使用KeInsertQueueApc向相应线程投递自杀APC，当然应该结合后面几章的知识，在投递前恢复一些敏感函数钩子。还有一种就是内存填0，或者填充其它数据，这样，进程也会被系统自动清理掉，也可以使用MmUnMapViewOfSection卸载到进程模块，这样进程也会出错退出。涉及到的几个内核函数多为未文档化函数吗，函数定义列举如下:

```
NTKERNELAPI
NTSTATUS
MmUnmapViewOfSection (
                    IN PEPROCESS Process,
                    IN PVOID BaseAddress
                    );

NTKERNELAPI
BOOLEAN
KeInsertQueueApc (
                PRKAPC Apc,
                PVOID SystemArgument1,
                PVOID SystemArgument2,
                KPRIORITY Increment
                );
NTKERNELAPI
VOID
KeInitializeApc (
               PRKAPC Apc,
               PRKTHREAD Thread,
               KAPC_ENVIRONMENT Environment,
               PKKERNEL_ROUTINE KernelRoutine,
               PKRUNDOWN_ROUTINE RundownRoutine,
```

```
        PKNORMAL_ROUTINE NormalRoutine,
        KPROCESSOR_MODE ProcessorMode,
        PVOID NormalContext
        );
```

其原型可以参考 WRK 和 Windows 泄漏的源代码，ReactOs 源代码也有一定的参考价值。

4.7 模块枚举及卸载

从眼下来看，纯 exe 木马已经很少了，大部分木马只是一个动态链接库文件，这样就做到了无进程，所以对 dll 的枚举和检测将会是未来 Rootkit 检测的重点内容之一。

首先看一下常见的列举进程模块的方法：第一种是利用系统进程快照，这个方法，在列举模块时速度是非常快的，效率很高。方法比较简单，主要使用方法是利用 API CreateToolhelp32Snapshot，创建进程快照，Module32First 获得第一个模块 Module32Next 获取下一个模块，Module32Next 列举到最后一个模块时返回 false。这种方法得到的信息是比较全的，比如基址，模块名字等，第二种使用函数 EnumProcessModules，这个函数返回一个 HMODULE 的数组，再使用 GetModuleFileNameEx 或者其他函数获更加具体的信息。第三种方法，使用 Native debug API ：

```
void EnumProcessModuleEx(DWORD Pid)
{
    PDEBUG_BUFFER Buffer=RtlCreateQueryDebugBuffer(0,FALSE);
    status=RtlQueryProcessDebugInformation(Pid,PDI_MODULES ,Buffer);
    if(!NT_SUCCESS(status))
    {       return ;
    }
    ULONG count=*(PULONG)(Buffer->ModuleInformation);
    //开头一个 DWORD 返回模块数 PDEBUG_MODULE_INFORMATION ModuleInfo
    //=(PDEBUG_MODULE_INFORMATION)((ULONG)Buffer->ModuleInformation+4);
    for(long i=0;i<count;i++)
    {
        //PDEBUG_MODULE_INFORMATION ModuleInfo 里面有需要的模块路径、
```

```
        //模块名、基址等信息
    }
    RtlDestroyQueryDebugBuffer(Buffer);
    FreeLibrary(hMod);
}
```

方法四，使用 ZwQueryVirtualMemory 枚举进程内存来列模块，实验发现这种方法在 ring3 下还是比较强的，360 的模块用前面的方法是无法获取到的，但是用这个方法可以得到。核心代码如下：

```
for(unsigned int i=0;i<0x7fffffff;i=i+0x10000)
{
    if( ZwQueryVirtualMemory(hProcess,(DWORD)i,2,(DWORD)Out_Data,
        512,(DWORD)&retLength)>0)
    { if(!IsBadReadPtr((BYTE*)Out_Data->SectionFileName.Buffer,1))
    if(((BYTE*)Out_Data->SectionFileName.Buffer)[0]==0x5c)
    {
        if(wcscmp(wstr, Out_Data->SectionFileName.Buffer))
        {
            GetUserPath(Out_Data->SectionFileName.Buffer);
        }
        wcscpy_s(wstr,     Out_Data->SectionFileName.Buffer);
    }
}
```

这种方法需要注意的一点就是，取到得是 DOS 路径，形如\Device\HarddiskVolume1，使用 QueryDosDevice 可将用户态的盘符转换成 DOS 下的，进行反查即可实现转换。很显然这两种使用原生 api 的方法可以轻易地移植到内核态下面，冰刃的个别版本就是用 NtQueryVirtualMemory 来枚举进程模块的。第五种方法，遍历 PEB 里面的双向链表。见 4.1 节。

方法六，_EPROCESS 里面有个 VadRoot 成员，其偏移在 xp 和 Windows7 下变化较大。但是实质没有多大变化，如下所示：

```
        struct _EPROCESS // 0x2c0
        {
                struct _KPROCESS Pcb; // +0x0(0x98)
```

```
        long ExitStatus; // +0x274(0x4)
        struct _MM_AVL_TABLE VadRoot; // +0x278(0x20)
        struct _ALPC_PROCESS_CONTEXT AlpcContext; // +0x298(0x10)
};
```

这个 VadRoot 的结构如图 4-14：

```
struct _MM_AVL_TABLE // 0x20
{
    struct _MMADDRESS_NODE BalancedRoot; // +0x0(0x14)
    ULONG DepthOfTree; // +0x14(0x4)
    ULONG Unused; // +0x14(0x4)
    ULONG NumberGenericTableElements; // +0x14(0x4)
    void* NodeHint; // +0x18(0x4)
    void* NodeFreeHint; // +0x1c(0x4)
};
```

图 4-14 VadRoot 结构

其中 BalancedRoot 是一个平衡二叉树，里面的成员实际上是这个结构（图 4-15）。

```
struct _MMVAD // 0x3c
{
    union u1; // +0x0(0x4)
    struct _MMVAD* LeftChild; // +0x4(0x4)
    struct _MMVAD* RightChild; // +0x8(0x4)
    ULONG StartingVpn; // +0xc(0x4)
    ULONG EndingVpn; // +0x10(0x4)
    union u; // +0x14(0x4)
    struct _EX_PUSH_LOCK PushLock; // +0x18(0x4)
    union u5; // +0x1c(0x4)
    union u2; // +0x20(0x4)
    struct _SUBSECTION* Subsection; // +0x24(0x4)
    struct _MSUBSECTION* MappedSubsection; // +0x24(0x4)
    struct _MMPTE* FirstPrototypePte; // +0x28(0x4)
    struct _MMPTE* LastContiguousPte; // +0x2c(0x4)
    struct _LIST_ENTRY ViewLinks; // +0x30(0x8)
    struct _EPROCESS* VadsProcess; // +0x38(0x4)
};
```

图 4-15MMVAD 完整结构

现在遍历这个二叉树就可以枚举到进程的所有模块了 。至于模块的名字和基地址等信息，从 Subsection 里面进去即可获取到了。遍历这个二叉树需要注意的是，内核堆栈的大小限制，在 Windows 上内核堆栈大小为 12kb，所以尽量不要使用二叉树的递归算法。二叉树的非递归遍历算法如下：

```
void    PreOrder(BiTree T, Status ( *Visit ) (ElemType e))
{
    InitStack(S);
    while ( T!=NULL || !StackEmpty(S))
    {
        while ( T != NULL )
        {
            Visit(T->data) ;
            Push(S,T);
            T = T->lchild;
        }
        if( !StackEmpty(S) )
        {   Pop(S,T);
            T = T->rchild;
        }
    }
}
```

第七种方法是暴力搜索进程的所有内存空间，查找 PE 文件标志，但是这个方法一般不能得到模块名字。

当出现可疑模块的时候，必须能够卸载可以模块才行，在用户态下，可以通过线程注入的的方法远线程使用 FreeLibrary 函数来卸载 dll，也可以先打开进程后使用 NtUnMapViewOfSection 来卸载 dll，同样在内核态也可以使用 NtUnMapViewOfSection 来卸载模块还可以使用 MmUnMapViewOfSection 或者 MiUnMapViewOfSection 来强制卸载模块。Dll 木马是未来的木马发展方向，所以对于一个 ARK 来说，检测模块比较重要。

4.8 本章小结

本章首先回顾了一下 windows 的进程线程的相关数据结构，接着总结了目前主流 Rootkit 的使用一些进程线程隐藏技术，并且给出了相应的隐藏进程检测方案，并且重点探讨了 windows 下的进程模块枚举技术，Rootkit Dll 化是当前的发展方向。

第五章 内核钩子及回调函数

5.1 内核钩子分类

内核钩子的命名主要有两类，一类是根据挂钩对象来命名的，一类是通过挂钩方式来命名的。通过挂钩对象来命名的例如：FSD 钩子，SSDT 钩子，Object 钩子，TCPIP 钩子等，通过挂钩方式来区分有 EAT 钩子，IAT 钩子，inline 钩子等。EAT 钩子的影响是全局的，对所有驱动模块都会产生影响，IAT 钩子只是影响被挂钩的那个模块，著名的主动防御软件东方微点里面大量使用了 IAT 和 EAT 钩子技术用来实现进程行为监视。Inline 钩子比 EAT 和 IAT 钩子稍微高级些，并且有很多技术细节上的研究，比如头部 HOOK 方式，push ret 方式，还有 Call 钩子方式等，实现的方法确实有很多花样。典型的 Head Hook 方式如图 4-8 所示。这种方式在中途修改了指令的跳转流程，跳转到恶意代码的函数里面，以便对参数进行修改或者对返回结果进行修改，在本论文的最后一章，还会深入讨论一种基于 x86 指令集的设计良好的 inline 钩子方案，很方便实现参数过滤和返回值过滤。

内存里面的 IAT 和 EAT 存储的是函数地址偏移，可以很容易的修改这个偏移来实现函数劫持，其挂钩方式有点类似 SSDT 钩子，但是又不完全像。

5.2 SSDT/ShadowSSDT 钩子枚举及卸载

SSDT 是 ring3 进入 ring0 的必经之门，在这里可以拦截到来自 ntdll 的所有系统调用，一个完整的系统调用 ReadFile 的执行流程如图 5-2

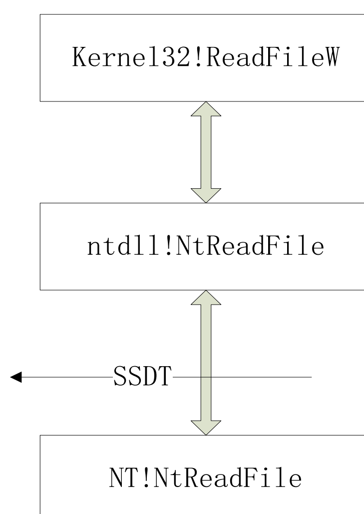

图 5-2

中间的黑线就是 SSDT 界面，过了这个界面就进入了 ring0 空间了，在这里可以拦截系统调用，并修改函数返回值等。SSDT 是一个跳转表，其中有参数和函数指针信息，所以挂钩 SSDT 只需要简单修改一下这个表里的函数指针即可，函数在这个 SSDT 表里面的索引称为系统调用号，这个表的函数排列顺序各个系统可能有差异。对于这个调用号，可以通过读取 ntdll 导出函数的开头是 Nt 的函数来获得，使用 IDA 打开 ntdll 定位到任意系统调用函数（ntdll 里面的一些函数不是系统调用），例如这里定位到 NtCreateFile 导出函数（图 5-3）

```
                              public NtCreateFile
                NtCreateFile  proc near              ; CODE XREF: sub_77EC826D+86↑p
                                                     ; sub_77EECB1A+21↑p ...
B8 42 00 00 00                mov     eax, 42h       ; NtCreateFile
BA 00 03 FE 7F                mov     edx, 7FFE0300h
FF 12                         call    dword ptr [edx]
C2 2C 00                      retn    2Ch
                NtCreateFile  endp
```

图 5-3 NtCreateFile 反汇编代码

第一句 mov eax,42h 中这个 42h 就是系统调用号，而且查看其它系统调用都会有相同的结构，所以我们可以遍历所有 ntdll 的 NT*导出函数，获取函数开头部分就可以得到函数名字和调用号了（导出表里有函数名字）。

对于原始函数地址的获取，通过反汇编 Windows 内核可知在如下位置使用了 SSDT，我们定位到这个位置就可以得到原始函数表地址了，这个函数位于 KiInitSystem()函数里面，直接硬编码会有难度，目前通用的查找方法是从重定位表里得到 KeServiceDescriptorTable（导出变量）被调用的相对地址，转换成可以访问的文件地址或者内存地址，如图 5-4。

```
NIT:005D5E24                mov     ds:_KiProcessOutSwapListHead, esi
NIT:005D5E2A                mov     ds:_KiStackInSwapListHead, esi
NIT:005D5E30                mov     ds:_KiGenericCallDpcMutex, 1
NIT:005D5E3A                mov     ds:dword_4823C4, esi
NIT:005D5E40                mov     ds:dword_4823C8, esi
NIT:005D5E46                mov     ds:byte_4823CC, 1
NIT:005D5E4D                mov     ds:byte_4823CE, 4
NIT:005D5E54                mov     ds:dword_4823D0, esi
NIT:005D5E5A                mov     ds:_KeServiceDescriptorTable, offset _KiServiceTable
NIT:005D5E64                mov     ds:dword_482484, esi
NIT:005D5E6A                mov     ds:dword_48248C, offset _KiArgumentTable
NIT:005D5E74                mov     eax, ecx
NIT:005D5E76
NIT:005D5E76 loc_5D5E76:                          ; CODE XREF: KiInitSystem()+13D↓j
NIT:005D5E76                mov     ds:dword_482488[eax], esi
```

图 5-4 KiInitSystem 函数部分反汇编代码

得到内存里的 SSDT 表是一件很容易的事情，比对 SSDT 函数指针是很简单的事情，但是现在更多的 Rootkit 使用 inline 钩子，得到文件里的原始地址和内存里的地址之后由于文件地址是没有被重定位的数据，所以还需要进行重定位，重定位后就可以比对内存数据和文件数据，如果不相同就说明被非法修改了。

对于 ShadowSSDT 钩子，和 SSDT 钩子是很类似的，不同之处在于 ShadowsSSDT 管理着 GUI 调用。可以在这里修改 GUI 调用来隐藏窗口等。其调用过程主旋律和 SSDT 一致，但是进入内核和返回的地址都有区别，前者通过系统快速调用进入内核，后者通过软中断自陷进入内核。只有线程调用了 GUI 函数后这个线程的 ServiceTable 才会指向 ShadowSSDT。不过在分发函数里调用的话是不会有问题的。获取 ShadowSSDT 地址的方法如下：

从 DriverEntry（PE 入口点）进去，可以找到设置 ShadowSSDT 的代码，提取出来就可以了，具体位置如图 5-5，XP 和 Windows7 通用。

```
:BF9AE612 57                    push    edi
:BF9AE613 68 10 90 99 BF        push    offset _W32pArgumentTable
:BF9AE618 FF 35 0C 90 99 BF     push    _W32pServiceLimit
:BF9AE61E 89 35 20 35 9A BF     mov     _countTable, esi
:BF9AE624 56                    push    esi
:BF9AE625 68 00 83 99 BF        push    offset _W32pServiceTable
:BF9AE62A FF 15 D8 B4 98 BF     call    ds:__imp__KeAddSystemServiceTable@20 ; KeAddSystemS
```

图 5-5 Shadow SSDT 引用位置

所以结构体如下：

```
typedef struct _RAW_SSSDT
{
    unsigned char PushOp;//0x68
    unsigned long TableRva;
    unsigned short CallOp;//0x15ff
    unsinged long KeIn;
}RAW_SSSDT,*PRAW_SSSDT;
```

至于界限，当然可以从这附近取得，但是不通用，所以读取时注意就可以了，函数表以 00000101 01000000 结尾

检查钩子方法和 SSDT 是一致的，修复方法是将其还原到原来文件里的情形。

5.3 内核普通函数钩子检测及恢复

Windows 内核有很多的导出的和没导出的函数，这些函数都会称为 Rootkit 挂

钩的对象。因此检测这些函数是必须的，但是完整的检测需要很高的技术和更多的代码，鉴于时间关系，当前版本实现的是导出函数的检测。使用长度反汇编引擎来获取函数长度，进行代码重定位，之后比对内存镜像和文件内容的差异，找出被挂勾的函数以及函数篡改地址。本软件使用的长度反汇编是开源的 LDsam，这个引擎代码量比较少，稳定性高，比较适合在驱动中使用。

恢复钩子则很简单，直接将原始代码写回去即可。在本论文的最后阶段，笔者已经实现了未导出函数的在线检测，通过 DIA 接口获取未导出函数的地址地址偏移以及长度。在线检测的好处是可以获取到函数友好地名字。目前虽有部分公开的工具能够离线检测未导出函数的钩子，但是因为没有函数名字，参考意义其实不大。

5.4 中断钩子检测及恢复

中断处理机制从第一台 PC 诞生至今都存在，可见它的重要性，所以一些恶意软件可能会对中断函数进行挂钩，从而进行一些恶意操作，比如拦截键盘输入等等。下面我门就来编程恢复中断钩子。首先是获取原始中断函数地址，众所周知，Windows 内核在加载到内存时首先执行 KiSystemStartup 函数，这个函数在初始化完必要的寄存器，处理器个数，和任务段之后就开始初始化中断了，因为接下来的许多操作都要依赖中断。所以我们可以在 KiSystemStartup 里面获取到原始中断函数表的地址，通过对 IDA 可以看到设置中断的代码如下（Windows xp 和 Windows7 下变化不大）：

```
INIT:005D5490   8B 7D F0         mov     edi, [ebp+var_10]
INIT:005D5493   BE 4C B2 5D 00   mov     esi, offset _IDT
INIT:005D5498   B9 00 08 00 00   mov     ecx, 800h
INIT:005D549D   C1 E9 02         shr     ecx, 2
INIT:005D54A0   F3 A5            rep movsd
```

其中的_IDT 就是指向中断函数表在文件中的位置了，而 KiSystemStartup 就是 PE 文件的入口点，和其它驱动不一样（其它驱动入口一般都是初始化异常处理结构）。这个特点为我们准确定位上面的代码带来了很大便利。映射一份内核进来，找到 PE 入口点，往下找就可以发现上面的代码了。文件 TDT 表里面的内容，文件里的 IDT 表和内存中的是不一样的

从 IDA 的结果来看，大致格式如下:

```
Struct    _RAW_IDT_TABLE
```

```
    {
        ULONG    IsrAddressRva;
        ULONG Other;
    };
```

获取原始地址函数如下，注意虚拟机不能体现函数作用。

```
NTSTATUS    GetRawIdtData(PVOID Buffer,ULONG BufferSize)
{
    MODULE_INFO                   KernelM={0};
    UCHAR                         *pKernelFile=NULL;
    ULONG                         i=0;
    int                           Strlen=0;
    PIMAGE_NT_HEADERS             Nt_head=NULL;
    UCHAR                         * EntryPointer=NULL;
    PRAW_IDT                      pRawData=NULL;
    ULONG                         pBaseAddress=0;
    PKIDTRAWENTRY                 DTr=NULL;
    KIDTENTRYS *                  DTMe=NULL;
    KGDT                          Gdt={0};
    PISR_INFO                     pInfo=(PISR_INFO)Buffer;
    ULONG                         ImageBase=0;
    if(GetKernelBase(&KernelM)==0)
    {
        KdPrint(("[GetRawIdtData] Open Kernel Failed!\n"));
        return STATUS_ACCESS_DENIED;
    }
    if(BufferSize<(256*sizeof(KIDTENTRYS)))
        return    STATUS_BUFFER_TOO_SMALL;
    pKernelFile=(UCHAR*)MapViewOfImage(KernelM.FullPath);
    if(pKernelFile==NULL)
    {
        return STATUS_UNSUCCESSFUL;
```

```
}
Nt_head=(PIMAGE_NT_HEADERS)(pKernelFile+((PIMAGE_DOS_HEADER)(pKernelFile)
)->e_lfanew);
EntryPointer=pKernelFile+Nt_head->OptionalHeader.AddressOfEntryPoint;
while(1)
{
搜索特征码来获取文件里的表偏移，代码省略
}
if(!pBaseAddress)
{
    UnMapViewOfImage(pKernelFile);
    return STATUS_UNSUCCESSFUL;
}
ImageBase=GetImageBase(pKernelFile);
int Delta=(ULONG)pKernelFile-ImageBase;
pBaseAddress+=Delta;
DTr=(PKIDTRAWENTRY)(pBaseAddress);
_asm sidt Gdt;
DTMe=(PKIDTENTRYS)MAKELONG(Gdt.LowIDTbase,Gdt.HiIDTbase);
for(DWORD index=0;index<256;index++)
{
    if(DTMe[index].Present)
pInfo[index].CurrAddr=MAKELONG(DTMe[index].OffsetLow,DTMe[index].OffsetHigh);
    pInfo[index].CurrAddr=0;
    pInfo[index].Index=index;
    if(DTr[index].Offset!=0x00000)
    pInfo[index].OrigAddr=DTr[index].Offset+(ULONG)KernelM.Base-ImageBase;
    else
    pInfo[index].OrigAddr=0;
}
if(pKernelFile)
    UnMapViewOfImage(pKernelFile);
```

```
    return STATUS_SUCCESS;
}
```

5.5 FastCallEntry 钩子检测及恢复

从 ring3 通过 SYSENETER 进入 ring0 的具体细节可参考 intell 手册。大致技术如下：从用户级到特权级的跳转，可以通过 SYSENETER 指令来实现，在进行特权级切换时，需要知道切换到的新堆栈的地址，以及相应过程的第一条指令的位置，各个寄存器内容等，为此 IA-32 有一组特殊寄存器来实现上述数据的保存，称为 MSR(Model Specific Register)。

SYSENTER_CS_MSR: New code segment selector 0x174

SYSENTER_ESP_MSR: New Stack Pointer 0x175

SYSENTER_EIP_MSR: New Instruction Pointer 0x176

后面的数据可以认为是寄存器的编号，对这三个寄存器的访问使用专门指令 rdmsr/wrmsr，编号由 ecx 指定，数据由 eax 传入或传出。Ring3 调用 SYSENETER 后，系统会把上面几个寄存器内容装入相应的寄存器，而 SYSENTER_EIP_MSR 中保存的地址就是 KiFastCallEntry，ring3 到 ring0 切换时，这个寄存器里的内容会被放到 EIP（指令指针）里面，之后切换到 ring0，从而执行 KiFastCallEntry，所以 KiFastCallEntry 是用户态通往 SSDT 的必经之门，在这里可以拦截到所有 SSDT 函数的调用，由于其重要性，所以非常有必要单独关注一下这个未导出函数。Windows xp 和 Windows7 都无一例外的在 1 号中断里面使用了这个函数，而且第一个 call 就是，这样获取这个未导出函数地址就比较通用和准确了，网上一些工具（例如 ruk）是通过暴力搜索 KiLoadFastSyscallMachineSpecificRegisters 来获取这个地址的，这种方法首先效率低（KiLoadFastSyscallMachineSpecificRegisters 是在 text 节，还是比较大的），其次又不通用，很难保证在其它操作系统上搜索的准确性。从上面对 ring3 进入 ring0 的简单分析可知，KiFastCallEntry 至少有两个挂钩点，挂钩点一，SYSENTER_EIP_MSR 值修改挂钩，挂钩点二，KiFastCallEntry 的 inline hook，360 安全卫士的挂钩点就是后面这种情况。明白了原理检测和恢复就比较简单了，获取原始地址函数如下，函数参数是 1 号中断的地址

```
ULONG GetKiFastCallEntryOrigEntry(UCHAR * Trap1)
{   UCHAR* p=Trap1;
    for(int i=0;i<1000;i++)
    {   if(*p==0x81)//第一个 call 就是了
```

```
            {return *(ULONG*)(p+2);}//绝对 call
            p++;
    }       return 0;
}
```

获取当前 KiFastCallEntry 地址代码如下

```
    _asm
    {
              mov ecx,0x176
             rdmsr
             mov KiFmAddr,eax
    }
```

5.6 Object 钩子检测及恢复

随着 inline 钩子的普及，越来越多的 Rootkit 已经把目标转向了更深层次的钩子，Object 钩子就是就是其中一种。Object 是指内核的一些对象，典型的包括文件对象，驱动对象，设备对象，事件，回调，等等，种类有很多，具体体现是，用户态使用 CreateXXX 来创建，使用 CloseXX 来销毁的对象。不同的对象具有不同的对象 Body 但是头部却是相同的，都是如图 5-6 结构，其中的 Body 就是具体的对象体，其中的 ObjectType 的结构如图 5-7，这个里面有一个 TypeInfo 结构，里面有操作这类对象的各个函数指针，如图 5-8。在这里面可以看到一些常见的对象操作函数指针，DumpProcedure，复制对象，其中 ParseProcedure 是对应了 CreateXXX 操作，比如在文件对象里，这个函数指针就是 IopParseFile，CreateFile 函数打开或创建文件最终会来到这个 函数里面。很显然这这里修改函数指针就可以实现挂钩，当然了进行 inline 钩子也是行的。

对于这类函数没有很好方法来获取原始地址，一般 是对内核文件进行暴力特征匹配，或者解析 pdb 文件获取地址，但是 pdb 文件需要联网，自身带一个大型 pdb 文件是不行的，所以目前的实现主要还是使用暴力特征匹配。

```
struct _OBJECT_HEADER // 0x20
{
    long PointerCount; // +0x0(0x4)
    long HandleCount; // +0x4(0x4)
    void* NextToFree; // +0x4(0x4)
    struct _OBJECT_TYPE* Type; // +0x8(0x4)
    BYTE NameInfoOffset; // +0xc(0x1)
    BYTE HandleInfoOffset; // +0xd(0x1)
    BYTE QuotaInfoOffset; // +0xe(0x1)
    BYTE Flags; // +0xf(0x1)
    struct _OBJECT_CREATE_INFORMATION* ObjectCreateInfo; //
    void* QuotaBlockCharged; // +0x10(0x4)
    void* SecurityDescriptor; // +0x14(0x4)
    struct _QUAD Body; // +0x18(0x8)
};
```

图 5-6 ObjectHeader 完整结构

```
struct _OBJECT_TYPE // 0x190
{
    struct _ERESOURCE Mutex; // +0x0(0x38)
    struct _LIST_ENTRY TypeList; // +0x38(0x8)
    struct _UNICODE_STRING Name; // +0x40(0x8)
    void* DefaultObject; // +0x48(0x4)
    ULONG Index; // +0x4c(0x4)
    ULONG TotalNumberOfObjects; // +0x50(0x4)
    ULONG TotalNumberOfHandles; // +0x54(0x4)
    ULONG HighWaterNumberOfObjects; // +0x58(0x4)
    ULONG HighWaterNumberOfHandles; // +0x5c(0x4)
    struct _OBJECT_TYPE_INITIALIZER TypeInfo; // +0x60(0x4c)
    ULONG Key; // +0xac(0x4)
    struct _ERESOURCE ObjectLocks[0x4]; // +0xb0(0xe0)
};
```

图 5-7 ObjectType 完整结构

```
struct _OBJECT_TYPE_INITIALIZER // 0x4c
{
    WORD Length; // +0x0(0x2)
    BYTE UseDefaultObject; // +0x2(0x1)
    BYTE CaseInsensitive; // +0x3(0x1)
    ULONG InvalidAttributes; // +0x4(0x4)
    struct _GENERIC_MAPPING GenericMapping; // +0x8(0x10)
    ULONG ValidAccessMask; // +0x18(0x4)
    BYTE SecurityRequired; // +0x1c(0x1)
    BYTE MaintainHandleCount; // +0x1d(0x1)
    BYTE MaintainTypeList; // +0x1e(0x1)
    enum _POOL_TYPE PoolType; // +0x20(0x4)
    ULONG DefaultPagedPoolCharge; // +0x24(0x4)
    ULONG DefaultNonPagedPoolCharge; // +0x28(0x4)
    void (__stdcall * DumpProcedure)(void*, struct _OBJECT_DUMP_CONTROL*); // +0x2c(0x4)
    long (__stdcall * OpenProcedure)(enum _OB_OPEN_REASON, struct _EPROCESS*, void*, ULONG,
ULONG); // +0x30(0x4)
    void (__stdcall * CloseProcedure)(struct _EPROCESS*, void*, ULONG, ULONG, ULONG); //
+0x34(0x4)
    void (__stdcall * DeleteProcedure)(void*); // +0x38(0x4)
    long (__stdcall * ParseProcedure)(void*, void*, struct _ACCESS_STATE*, char, ULONG,
struct _UNICODE_STRING*, struct _UNICODE_STRING*, void*, struct
_SECURITY_QUALITY_OF_SERVICE*, void**); // +0x3c(0x4)
    long (__stdcall * SecurityProcedure)(void*, enum _SECURITY_OPERATION_CODE, ULONG*,
void*, ULONG*, void**, enum _POOL_TYPE, struct _GENERIC_MAPPING*, char); // +0x40(0x4)
    long (__stdcall * QueryNameProcedure)(void*, BYTE, struct _OBJECT_NAME_INFORMATION*,
ULONG, ULONG*); // +0x44(0x4)
    BYTE (__stdcall * OkayToCloseProcedure)(struct _EPROCESS*, void*, void*, char); // +0x48
(0x4)
};
```

图 5-8 ObjectTypeInitializer 完整结构

5.7 其它几个重要函数钩子检测及恢复

对于 ARK 来说，为了获得一些可靠的数据，首先应该考虑几个重要函数的钩子恢复，一个就是 IofCompleteRequest，这个函数在驱动每次完成 IRP 时被调用，在这里可以修改返回数据，例如 Ak922 就是通过挂钩这个函数来实现文件隐藏的。还有另外一个函数是 IofCallDriver，这个函数被用来实现驱动间的相互调用，在这里可以实现驱动调用转向控制。这几个函数挂钩恢复要专门处理，因为这两个函数都有好几个挂钩点，根据不同的操作系统而异。

5.8 通知回调检测及卸载

PsSetCreateProcessNotifyRoutine 是一个监视进程创建的函数，这个函数提供了一种监视进程创建的绿色方法，毕竟是有文档的东西，使用起来比较安全。当然这个函数在使用时时有限制的，XP 下只能安装 8 个监视函数（32 位的系统）。我们可以想到，系统应该有一个表来保存这几个函数的地址，以便系统调用，OK 现在我们使用 windbg 去看看，反汇编之后发现一个和 PspCreateProcessNotifyRoutine 结构：

```
8062cc40 56              push    esi
8062cc41 8d049d60165680  lea     eax,nt!PspCreateProcessNotifyRoutine (80561660)[ebx*4]
8062cc48 50              push    eax
8062cc49 e8b17d0100      call    nt!ExDereferenceCallBackBlock (806449ff)
8062cc4e 56              push    esi
```

PspCreateProcessNotifyRoutine 是一个 EX_FAST_REF 的数组，XP 下一共有 8 个元素（Vista 以后增加到 64 个）。这个结构在 wrk 里面可以找到到，其结构是

```
typedef struct _EX_FAST_REF
{
    union
    {
        PVOID Object;
        ULONG_PTR RefCnt:3;
        ULONG_PTR Value;
    };
} EX_FAST_REF, *PEX_FAST_REF;
```

注意里面低三位是引用计数，value 是一个指针，指向一个叫做 EX_CALLBACK_ROUTINE_BLOCK,的结构，这个结构里面就有回调函数的指针了，完整结构如下：

```
typedef struct _EX_CALLBACK_ROUTINE_BLOCK
{
    EX_RUNDOWN_REF RundownProtect;
    PEX_CALLBACK_FUNCTION Function;
    PVOID Context;
} EX_CALLBACK_ROUTINE_BLOCK, *PEX_CALLBACK_ROUTINE_BLOCK;
```

为了得到这个指针，可以用移位运算：PEX_CALLBACK_ROUTINE_BLOCK Point =(PEX_CALLBACK_ROUTINE_BLOCK)((FastRef->Value>>3)<<3);

找到了上面这个地址，进行一下数组枚举就得到所有回调函数地址了，至于卸载则很简单，可以使用相同的函数来进行卸载。除了上面这个之后如下两个监视的机理一模一样，PsSetCreateThreadNotifyRoutine，PsSetLoadImageNotifyRoutine。同样找到函数地址之后可以使用如下两个函数来实现卸载 PsRemoveCreateThreadNotifyRoutine，PsRemoveLoadImageNotifyRoutine。

5.9 注册表回调检测及卸载

注册表回调可以用来保护注册表内容。经过我自己对 XP 和 Windows7 相关代码的不完全逆向，总结规律如下：

从 CmUnRegisterCallback 进去在 XP 下有一个_CmpCallBackVector 数组，数目为 100，数组元素为

```
typedef struct _EX_FAST_REF
{
    union
    {
        PVOID Object;
        ULONG_PTR RefCnt:3;
        ULONG_PTR Value;
    };
} EX_FAST_REF, *PEX_FAST_REF;
```

其中的 Object 指向的结构为

```
typedef struct _EX_CALLBACK_ROUTINE_BLOCK
{
    EX_RUNDOWN_REF RundownProtect;
    PEX_CALLBACK_FUNCTION Function;
    PVOID Context;//
} EX_CALLBACK_ROUTINE_BLOCK, *PEX_CALLBACK_ROUTINE_BLOCK;
```

而 Context 结构指向

```
typedef struct _CM_ARRY_XP
{
LARGE_INTEGER Cookie;//
LIST_ENTRY          ListEntry1;//8
ULONG UnKnown1;//16
ULONG UnKnown2;//20
ULONG UnKnown3;//24
ULONG UnKnown4;//28
ULONG UnKnown5;//32
```

```
LIST_ENTRY ListEntry2;//36
ULONG     UnKnown;//44
ULONG     Context;//48
ULONG     UnKnown6;
}CM_CONTEXT,*PCM_CONTEXT;
```

取得 Cookie 就可以实现卸载了……在 XP 这个 Cookie 就是注册回调时的系统时间，上面那种方式很明显寻址效率超低，所以 Windows7 下就做了改进了，没有使用数组而是一个链表：

_CallbackListHead，这个链表连接在下面结构的第一项里（就是 ListEntryHead）：

```
typedef struct _CM_NOTIFY_ENTRY
{
LIST_ENTRY ListEntryHead;///初始化时指向自己
ULONG     UnKnown1;//+8
ULONG     UnKnown2;//12
LARGE_INTEGER Cookie;//16
ULONG     Context;//+24
ULONG     Function;//+28
ULONG     Altitudes;//32
PVOID      BufferPointer;//36
LIST_ENTRY ListEntry1;//40 初始化时指向自己,没搞清楚到底指哪去
}CM_NOTIFY_ENTRY,*PCM_NOTIFY_ENTRY;
```

要卸载只需要遍历这个双向链表，取得 Cookie 就可以了实现卸载了，辛辛苦苦取得这个 Cookie 是因为使用 CmUnRegisterCallback 来安全卸除回调函数时需要使用 Cookie。

5.10 本章小结

本章主要总结了 Rootkit 里面需要关注几种常见钩子的检测及恢复技术，同时各种系统回调也是 Rootkit 实现再感染的以及自保护隐藏等恶意目的帮凶之一。本章所述内容是整个 Rootkit 实现中难度较大的部分也是最重要的部分，同时也是影响整个软件的稳定性的重要因素，这一部分的代码质量至关重要。限于论文篇幅许多细节已经被笔者略去了，在文献[4]已经文献[17]对各种钩子的实现有详尽的描述，还有简单的检测方法。

第六章 文件管理

6.1 文件隐藏及检测

文件隐藏有如下一些常见方法：

1) 用户态挂钩 ZwQueryDirectoryFile 实现隐藏。
2) SSDT 层挂钩 ZwQueryDirectoryFile 实现隐藏。
3) 挂钩 IofCompleteRequest 实现隐藏。
4) 挂钩 FSD 实现隐藏。
5) 挂钩 IofCallDriver 实现隐藏。
6) 挂钩系统总线隐藏。
7) Object 钩子实现隐藏。
8) Object 劫持实现隐藏。
9) 其它方法，例如 Disk 分发函数钩子。

本软件使用的检测方法有两种，一种是通过向 FSD（File System Driver）发送查询 IRP 来实现检测的可以检测到上述前面 5 种的隐藏方法，由于前面有对 Object 钩子的检测，综合起来检测力度还可以。FSD 层次的文件枚举流程如下：

1) 打开文件系统驱动。
2) 构造查询目录 IRP 请求报文。
3) 填写 IRP 请求包一些重要字段。
4) 调用文件系统驱动分发函数。这里是调用而不是 IofCallDriver，这样可以防止 IofCallDriver 的干扰。

另外一种方法是磁盘级文件解析，见 6.5 节。

6.2 FSD 钩子检测及恢复

将文件系统驱动用 IDA 打开，等待一会就反汇编好了，停在程序入口处，入口处一般是异常处理程序初始化，从入口往下找，就可以看到 DriverEntry 函数，在这个函数里可以看到如下代码：

```
C7 46 7C 24 CE 02 00    mov         dword ptr [esi+7Ch], offset _NtfsFsdLockControl@8
C7 46 68 03 B4 0B 00    mov         dword ptr [esi+68h], offset _NtfsFsdDirectoryControl@8
C7 46 50 24 A2 02 00    mov         dword ptr [esi+50h], offset _NtfsFsdSetInformation@8
C7 46 38 C8 8B 0A 00    mov         dword ptr [esi+38h], offset _NtfsFsdCreate@8
```

```
C7 46 40 A6 3D 0A 00    mov         dword ptr [esi+40h], offset _NtfsFsdClose@8
C7 46 44 35 89 02 00    mov         dword ptr [esi+44h], offset _NtfsFsdRead@8
C7 46 48 7E 15 02 00    mov         dword ptr [esi+48h], offset _NtfsFsdWrite@8
```

这个地方就是设置分发例程的指令了，原始文件系统分发函数地址可以从这些指令中提取，接着重复的把各个版本的驱动拿过来反汇编一下，发现各个版本，不管是 fastfat 还是 ntfs 的驱动设置分发例程都有和上面类似的代码。故定义特征码如下：

```
typedef struct _ORIG_CODE
{
    unsigned short OpCode;//0x46c7
    unsigned char   index;//就是分发函数号码
    unsigned long   Rav;
}ORIG_CODE;
```

其中特征码格式不符合上述结构体的两个分发函数是 Clearnup 和 PNP 处理例程，但是这两个例程在 Rootkit 里面用途不大，可以单独考虑也可以省略（本软件直接省略）。

6.3 过滤驱动检测及摘除

文件过滤驱动处于文件系统上面，磁盘过滤驱动处在卷设备上面，过滤驱动优先获得了 IRP 的处理权，所以这两种驱动都可能被恶意用来隐藏文件和防止文件删除。安装了过滤驱动的系统，其驱动关系如图 6-1 下：

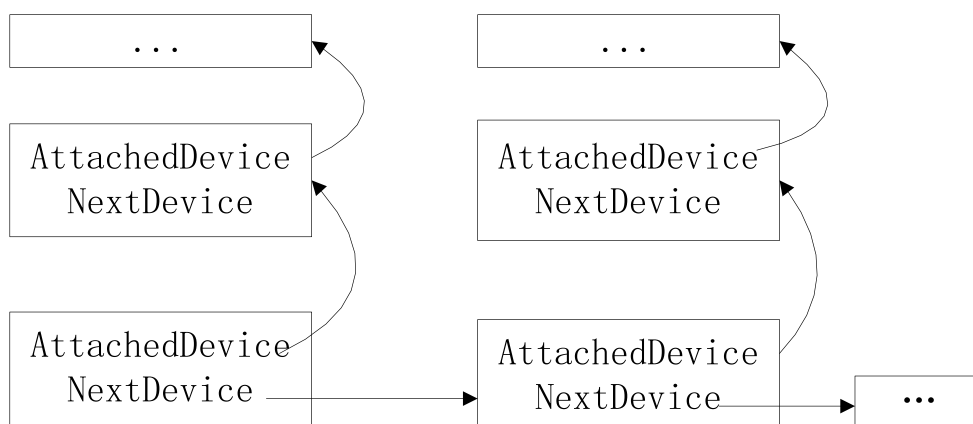

图 6-1 分层过滤驱动示意图

DriverObject 的 DeviceObject 指向驱动创建的第一个设备，而每个设备的 NextDevice 指向同一个驱动创建的下一个设备，设备对象里的 AttachedDevice 指向过挂载在这个设备上的过滤驱动，过滤驱动之上可能还会存在过滤驱动，使用函数安装过滤驱动时，总是最后挂载的设备在最上面，优先获得 Irp 处理权。

由上图可知，检测文件系统过滤驱动只需要遍历文件系统驱动的所有设备，逐个查看 AttachedDeive 域指向的内容就可以了，当然磁盘过滤驱动的情况是一样。要检测过滤驱动，首先要获得最下层的驱动对象，可以使用上面获取 ntfs 驱动对象的方法来取得文件系统驱动，也可以打开任意一个文件之后，使用 IoGetBaseFileSystemDeviceObject 来获取最下层设备对象，再从设备对象获取驱动对象。注意这个函数调用之后，不可以减少对象引用计数，因为这个函数只是返回了文件对象里面的 Vbp 里面的 DeviceObject 指针，也不可使用 IoGetRelatedDeviceObject，因为这个是返回设备栈上最顶层的设备。

摘除比较简单，直接把 AttachedDevice 设置为空指针就可以了。

6.4 文件解锁及强制访问

有时候需要操作被占用的文件，例如别人是以非共享打开的，在强制删除文件时一般都要进行文件解锁。文件被锁定的情况主要是 3 种，一种是被进程加载了，另外两种是被进程打开或者映射了，本文主要关注文件被独占打开的情况，解锁流程一般是先枚举系统句柄表，找到所有类型是文件的句柄，复制句柄到本本进程，如果需要解锁则远程关闭。但是枚举打开的文件是一个效率很低的事情，而且如果枚举到命名管道，就会使线程阻塞。所以我使用如下代码实现句柄信息查询：

```
{
    if(Param==NULL)
        return 0;
    NTSTATUS Status=0;
    ULONG RetLen;
    PVOID Buffer=AllocateBuffer(0x1000);
    PUNICODE_STRING Name=(PUNICODE_STRING)Buffer;
    __try
    {
        while(!Param->m_bStop)
```

```
        {
            ULONG FileType=GetFileType(Param->hHandle);
            RtlZeroMemory(Buffer,0x1000);
            if(FileType==FILE_TYPE_DISK)//只处理这种情况
            {
    Status=ZwQueryObject(Param->hHandle,ObjectNameInformation,Buffer,0x1000,&RetLen);
                if(NT_SUCCESS(Status))
                {
                    if(Name->Length<MAX_PATH*2&&Name->Length>1)
                    {
                        RtlZeroMemory(Param->FileName,MAX_PATH);
                        RtlCopyMemory(Param->FileName,Name->Buffer,Name->Length);
                        Param->RetCode=1;
                    }
                }
            }
            SetEvent(Param->hEvent);
            SuspendThread(GetCurrentThread());
        }
        FreeBuffer(Buffer);
        return 0;
    }
    __except(1)
    {
        if(Buffer)
            FreeBuffer(Buffer);
        return 1;
    }
}
```

代码首先检查文件类型是不是 FILE_TYPE_DISK，因为管道那些在 Windows 下也会当成文件来处理，不是普通文件类型的文件很容易造成查询线程僵死。还有就

是使用一个线程循环查询，而不是每个句柄都创建一个线程，上述代码有助于显著提高查询效率。

强制访问文件这个功能是为了支持后面对注册表解析的支持，Windows 在运行后，注册表文件被系统进程独占，其他进程没有访问权限，所以需要强制获取访问权限，访问权限保存在进程的句柄信息表里面，每一个句柄元素表示如图 6-2。其中的 GrantedAccess 记录的就是访问权限了，在不同版本的 Windows 下，句柄表在 EPROCESS 里面的位置不同，但是结构却是相同的。只需要获取进程 EPROCESS，得到句柄表偏移，枚举进程句柄表，得到 HANDLE_TABLE_ENTRY，修改句柄 GrantedAccess 字段就可以提升文件访问权限了。当然如果仅仅只是为了看看文件里有些什么东西，本软件还提供了一个强制复制文件功能，此功能通过 Irp 查询得到文件的扇区位置，使用直接复制扇区的方法实现文件强力复制，之后得到文件的访问权限。

```
struct _HANDLE_TABLE_ENTRY // 0x8
{
    void* Object; // +0x0(0x4)
    ULONG ObAttributes; // +0x0(0x4)
    struct _HANDLE_TABLE_ENTRY_INFO* InfoTable; // +0x0(0x4)
    ULONG Value; // +0x0(0x4)
    ULONG GrantedAccess; // +0x4(0x4)
    WORD GrantedAccessIndex; // +0x4(0x2)
    WORD CreatorBackTraceIndex; // +0x6(0x2)
    long NextFreeTableEntry; // +0x4(0x4)
};
```

图 6-2 句柄元素结构

6.5 磁盘级文件解析

本软件虽然在 FSD 层次上做了隐藏文件的检测，但是对于日益猖獗的 Rootkit 技术来说还是远远不够，于是在此基础上增加了磁盘级的文件解析。Windows 支持的文件系统比较多，无法实现一一支持，所以这中磁盘级的解析目前仅仅支持 Ntfs 已经 FAT 两种文件系统，详细解释 Ntfs 文件系统以及 FAT 文件系统的各种数据结构需要超过一本普通教材的篇幅，这里需要说明的一个问题是文件时间的获取和设置，通常有这三种的时间：DOS 时间(DOS time)，系统时间(system time)，文件时间(File Time)。在用户态可以使用如下几个 API 来实现转换：

FileTimeToDosDateTime

DosDateTimeToFileTime

FileTimeToSystemTime

SystemTimeToFileTime

FAT 文件系统已经是 DOS 时代的产物了，所以很多东西受到限制，比如单文件大小不能超过 4G 等。FAT 使用的时间是 DOS 时间，FAT 文件系统主要用于各类移动存储设备，一些古董级别的 PC 也使用 FAT32 文件系统。FAT 的解析异常的简单，卷开头的是 512 字节的 PBP，这个是 FAT32 和 Ntfs 共有的东西，当然字段是不一样的。FAT 的 bpb 如下：

```
typedef struct _FATBPB
{
    BYTE     JMPCode[3];            /*引导跳转代码*/
    CHAR     System_ID[8];          /*厂商标志和版本号*/
    WORD     BytesPerSector;        /*每扇区字节数*/
    BYTE     SectorsPerCluster;     /*每簇扇区数*/
    WORD     ReservedSectors;       /*保留扇区数*/
    BYTE     FatNum;                /*FAT 的个数*/
    WORD     RootEntry;             /*根目录项数*/
    WORD     TotalSectors;          /*分区总扇区数(分区小于 32M 时)*/
    BYTE     Media;                 /*分区介质标识*/
    WORD     SectorsPerFAT16;       /*每个 FAT 占的扇区数*/
    WORD     SectorsPerTrack;       /*每道扇区数*/
    WORD     Heads;                 /*磁头数*/
    DWORD    HiddenSectorNum;       /*隐含扇区数*/
    DWORD    BigTotalSectors;       /*分区总扇区数(分区大于 32M 时)*//*end of 23 Bytes*/
    DWORD    SectorsPerFAT32;       /*每个 FAT 占的扇区数，FAT32 用*/
    WORD     Extendumber;           //扩展号
    WORD     VersionNumber;         //版本号
    DWORD    RootDirClusterNumber;  //根目录簇号
    WORD     FileInfoSector;        //文件信息扇区
    WORD     backupsector;          //备份扇区
    char   Reserved[12];            //保留
```

```
    //add by clivebi
    unsigned char    Drive;                      // 64
    unsigned char    Res4;                       // 65
    unsigned char    ExtBootSignature;                   // 66
    unsigned long    VolumeID;                   // 67
    unsigned char    VolumeLabel[11];
    unsigned char    SysType[8];         // 71
    unsigned char    Res2[420];                  // 90
    unsigned short Signature1;                   // 510

} FATBPB,*PFATBPB;
```

其中含有一些卷相关的参数信息，这些信息比较重要，通过其中的根目录簇号可以定位到根目录，之后遍历目录表可以得到目录信息。详细信息参考 FAT 有关文档。

NTFS 的解析稍微复杂，开头的 BPB 内容如下：

```
typedef struct _BPB
{
    UCHAR    Jump[3];
    UCHAR    Format[8];// 文件系统 NTFS 的为 "NTFS" 4-7 字节
    USHORT BytesPerSector;//每扇区字节数，12-13 字节,512 字节
    UCHAR    SectorsPerCluster;
//每簇扇区数，第 14 字节,通常为 8，8*512=4096=0x1000=4KB 字节
    USHORT BootSectors;//dos 保留扇区
    UCHAR    FATNumber;//FAT 区个数
    USHORT Mbz2;
    USHORT Reserved1;
    UCHAR    MediaType;
    USHORT Mbz3;
    USHORT SectorsPerTrack;
    USHORT NumberOfHeads;
    ULONG    PartitionOffset;
    ULONG    Reserved2[2];
```

```
    ULONGLONG TotalSectors; //扇区总数  41-48 字节
    ULONGLONG MftStartLcn;//$MFT 的起始逻辑簇号 LCN 49-56 字节
    ULONGLONG Mft2StartLcn;
    ULONG ClustersPerFileRecord;
   //65-68  当小于 0X80 时为每 MFT 记录簇数，
    //当大于等于 0x80 时，比如为 0xF6(通常为这个数)，则 1 << (0x100-0xF6)=1024 字节（两
扇区）为每 MFT 的字节数
    ULONG ClustersPerIndexBlock;//69-72 字节,每索引块簇数
    ULONGLONG VolumeSerialNumber;//卷序列号 73-80
}NTFS_BOOTSECTOR,*PNTFS_BOOTSECTOR;
```

从比 BPB 得到 MFT 表起始地址，之后遍历 MFT 即可。有时可能需要从物理磁盘上获取各个卷的位置信息，物理磁盘的开头即 MBR 内容如下：

```
typedef struct _PARTITION_ENTRY
{
    UCHAR active;          /*0x80*/
    UCHAR StartHead;
    UCHAR StartSector;
    UCHAR StartCylinder;
    UCHAR PartitionType;
    UCHAR EndHead;
    UCHAR EndSector;
    UCHAR EndCylinder;
    ULONG StartLBA;       //根据这个可以得到该分区的物理扇区的偏移
    ULONG TotalSector;
} PARTITION_ENTRY, *PPARTITION_ENTRY;
typedef struct _MBR_SECTOR
{
    UCHAR                  BootCode[446];
    PARTITION_ENTRY Partition[4];
    USHORT                   Signature;//结束标志
} MBR_SECTOR, *PMBR_SECTOR;
```

从物理磁盘上读取第一个扇区得到的就是上面的数据，这里说的物理磁盘在Windows 下的符号连接一般如下图 6-3 所示：

PhysicalDrive0	SymbolicLink	\Device\Harddisk0\DR0
PhysicalDrive1	SymbolicLink	\Device\Harddisk1\DR1

图 6-3

而逻辑卷的符号连接通常如图 6-4 所示：

C:	SymbolicLink	\Device\HarddiskVolume1

图 6-4

在我电脑上 C 盘连接的对象是\Device\HarddiskVolume1,而后者就是我的第一个物理磁盘的第一个分区（卷）。

解析时可以选择直接打开物理磁盘还是打开分区进行解析，一般来说结构都一样，但前者更可靠些。读写磁盘使用给 ATAPI 发送 srb 请求来实现，这个也是机器狗病毒使用来穿透还原软件的核心技术，其函数如下：

```
NTSTATUS    AtapiReadWriteDiskLow (
                    PDEVICE_OBJECT dev_object,
                    ULONG MajorFunction,
                    PVOID buffer,
                    ULONG DiskPos, //Number Of Sector
                    int BlockCount //Count Of Sectors
                    )
{
    NTSTATUS status;
    PSCSI_REQUEST_BLOCK srb;
    PSENSE_DATA sense;
    KEVENT Event;
    PIRP irp;
    PMDL mdl;
    IO_STATUS_BLOCK isb;
    PIO_STACK_LOCATION isl;
    PVOID psense;
    int count = 8;
```

```
while(1){
    srb =
(PSCSI_REQUEST_BLOCK)RtlAllocateNonPagedBuffer(sizeof(SCSI_REQUEST_BLOCK));
    if(!srb)
        break;
    sense =(PSENSE_DATA) RtlAllocateNonPagedBuffer(sizeof(SENSE_DATA));
    psense = sense;
    if(!sense)
        break;
    srb->Length = sizeof(SCSI_REQUEST_BLOCK);
    srb->Function = 0;
    srb->DataBuffer = buffer;
    srb->DataTransferLength = BlockCount<<9;//sector size*number of sector
    srb->QueueAction = SRB_FLAGS_DISABLE_AUTOSENSE;
    srb->SrbStatus = 0;
    srb->ScsiStatus = 0;
    srb->NextSrb = 0;
    srb->SenseInfoBuffer = sense;
    srb->SenseInfoBufferLength = sizeof(SENSE_DATA);
    if(MajorFunction == IRP_MJ_READ)
        srb->SrbFlags = SRB_FLAGS_DATA_IN;
    else
        srb->SrbFlags = SRB_FLAGS_DATA_OUT;
    if(MajorFunction == IRP_MJ_READ)
        srb->SrbFlags |= SRB_FLAGS_ADAPTER_CACHE_ENABLE;
    srb->SrbFlags |= SRB_FLAGS_DISABLE_AUTOSENSE;
    srb->TimeOutValue = (srb->DataTransferLength >> 10) + 1;
    srb->QueueSortKey = DiskPos;
    srb->CdbLength = 10;
    srb->Cdb[0] = 2 * ((UCHAR)MajorFunction+ 17);
    srb->Cdb[1] = srb->Cdb[1] & 0x1F | 0x80;
```

```
        srb->Cdb[2] = (unsigned char)(DiskPos>>0x18)&0xFF;      //
        srb->Cdb[3] = (unsigned char)(DiskPos>>0x10)&0xFF;      //
        srb->Cdb[4] = (unsigned char)(DiskPos>>0x08)&0xFF;      //
        srb->Cdb[5] = (UCHAR)DiskPos;           //填写 sector 位置
        srb->Cdb[7] = (UCHAR)(BlockCount>>0x08);
        srb->Cdb[8] = (UCHAR)BlockCount;
        KeInitializeEvent(&Event,NotificationEvent, 0);
        irp = IoAllocateIrp(dev_object->StackSize,0);
        mdl = IoAllocateMdl(buffer, BlockCount<<9, 0, 0, irp);
        irp->MdlAddress = mdl;
        if(!mdl)
        {
            //ExFreePool(srb);
            //ExFreePool(psense);
            RtlFreeNonPagedBuffer(srb);
            RtlFreeNonPagedBuffer(psense);
            IoFreeIrp(irp);
            KdPrint(("InsufficientRes"));
            return STATUS_INSUFFICIENT_RESOURCES;
        }
        MmProbeAndLockPages(
mdl,0,(MajorFunction == IRP_MJ_READ?IoReadAccess:IoWriteAccess));
        srb->OriginalRequest = irp;
        irp->UserIosb = &isb;
        irp->UserEvent = &Event;
        irp->IoStatus.Status = 0;
        irp->IoStatus.Information = 0;
        irp->Flags = IRP_SYNCHRONOUS_API|IRP_NOCACHE;
        irp->AssociatedIrp.SystemBuffer = 0;
        irp->Cancel = 0;
        irp->RequestorMode = 0;
        irp->CancelRoutine = 0;
```

```
        irp->Tail.Overlay.Thread = PsGetCurrentThread();
        isl = IoGetNextIrpStackLocation(irp);
        isl->DeviceObject = dev_object;
        isl->MajorFunction = IRP_MJ_SCSI;
        isl->Parameters.Scsi.Srb = srb;
        isl->CompletionRoutine = IrpCompletionRoutine_0;
        isl->Context = srb;
        isl->Control                                    =
SL_INVOKE_ON_CANCEL|SL_INVOKE_ON_SUCCESS|SL_INVOKE_ON_ERROR;
        status = IofCallDriverEx(dev_object,irp);
        KdPrint(("IoCallDriverBack"));
        KeWaitForSingleObject(&Event,Executive, 0, 0, 0);

        if(srb->SenseInfoBuffer != psense && srb->SenseInfoBuffer)
            ExFreePool(srb->SenseInfoBuffer);

        RtlFreeNonPagedBuffer(srb);
        RtlFreeNonPagedBuffer(psense);

        if ( status >= 0 || !count )
            return status;

        KdPrint(("Send XXX Failed..%08x\r\n", status));
        KeStallExecutionProcessor(1u);
        --count;
    }
    return STATUS_INSUFFICIENT_RESOURCES;
}
```

使用此函数的难点是获得 ATAPI 驱动中处理相关请求的设备对象（DEVICE_OBJECT）,ATAPI 会生成不少于 4 个的设备对象，但是类型是 FILE_DEVICE_DISK 通常只有一个，而这个设备就是我们需要的设备了。

```
NTSTATUS GetAtapiObjectWeNeed(PDEVICE_OBJECT * ppObj)
{
    NTSTATUS Status=STATUS_UNSUCCESSFUL;
    UNICODE_STRING Name;
    RtlInitUnicodeString(&Name,L"atapi");
    PDRIVER_OBJECT pDriverObject;
    *ppObj=0;
    Status=ObReferenceObjectByName(&Name,OBJ_CASE_INSENSITIVE,NULL,
    FILE_ALL_ACCESS,*IoDriverObjectType,KernelMode,0,(PVOID*)&pDriverObject);
    if(NT_SUCCESS(Status))
    {
        PDEVICE_OBJECT Seek=pDriverObject->DeviceObject;
        Status=STATUS_UNSUCCESSFUL;
        while(Seek)
        {
            if(Seek->DeviceType==FILE_DEVICE_DISK)
            {
                *ppObj=Seek;
                Status= STATUS_SUCCESS;
                break;
            }
            Seek=Seek->NextDevice;
        }
        ObDereferenceObject(pDriverObject);
    }
    return Status ;
}
```

6.6 本章小结

本章主要讨论了 windows 下常见的几种文件隐藏方法以及检测对策，其中的过滤驱动技术被大量的还原软件以及只保护软件所使用，在正规产品中用途比较多，所以摘除还需谨慎，第二章所说的数字签名验证可以简单的实现这个目的。本软

件起初使用的向 FSD 发送 IRP 的方法来枚举文件，在论文完成的最后阶段，笔者实现了磁盘级的文件操作。有关 FAT 格式以及 NTFS 文件系统格式的相关内容参见[25~27]。

第七章 注册表

7.1 注册表以及隐藏

注册表（Registry）是 Microsoft Windows 中的一个重要的数据库，用于存储系统和应用程序的设置信息。早在 Windows 3.0 推出 OLE 技术的时候，注册表就已经出现。随后推出的 Windows NT 是第一个从系统级别广泛使用注册表的操作系统。但是，从 Microsoft Windows 95 开始，注册表才真正成为 Windows 用户经常接触的内容，并在其后的操作系统中继续沿用至今。注册表是为 Windows NT 和 Windows95 中所有 32 位硬件/驱动和 32 位应用程序设计的数据文件。Rootkit 一般要在注册表里存储信息或者设置开机自启动，为了自身存在不被发现，一般 Rootkit 都会进行各类函数挂钩或者从内核注册表链表上将自身节点摘除掉，以期达到隐藏相关注册表项的目的。Windows 平台下注册表访问流程大致如下（以 RegCreateKey 为例）：

Kernel32！RegCreateKey

Ntdll! ZwCreateKey

Nt!ZwCreateKey

Nt!CmpParseKey

……

可见，Windows 对注册表的访问是很复杂的，同时，微软在设计注册表时就没有打算让第三方软件随意访问注册表，所以一旦系统运行起来，System 就会独占注册表文件，任何其它进程对注册表文件的访问都会失败。从上述的注册表操作 API 调用过程来看，有很多种方法实现注册表隐藏，甚至可以摘除注册表文件缓存里面的键值节点来隐藏注册表。

7.2 注册表文件格式

由于注册表的重要性以及隐藏方法的多样性，使用一般的方法来检测注册表已经显得力不从心了，比如冰刃使用 CM 系列函数来实现注册表操作，现在已经可以绕过了，所以越来越多的 ARK 都把目光聚集在解析注册表文件实现注册表操作上。由目前公开的资料及逆向成果来看，注册表的整体结构如下下图所示。注册表文件包含许多存储块，这些块称为 bin 或者 Cell，块的饿大小是 0x1000，刚好是默认内存页面的大小，所以注册表文件通常是 0x1000 的整数倍。开始那个块是

基块，含有注册表的一些信息，例如版本号。Bin 是注册表的逻辑单元，有一个头部和多个 cell 组成。Cell 是包含数据的最小数据单元，通常其大小是 0x8。Cell 有多种类型分别用来存储不同的信息，如图 7-1。

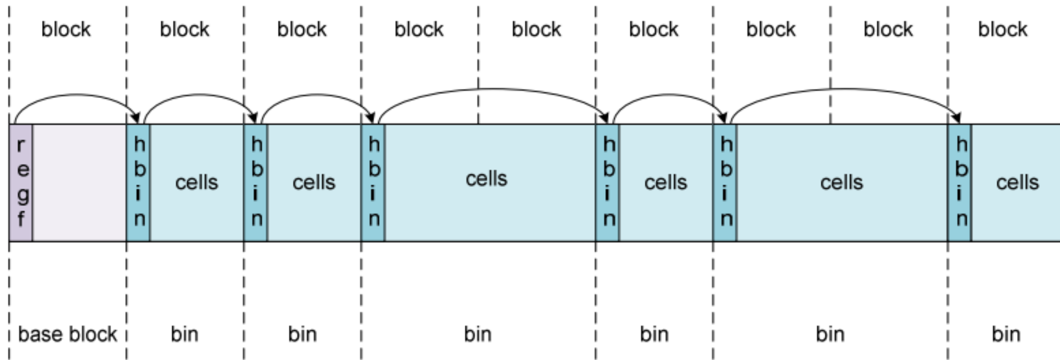

图 7-1 bin 大致结构

注册表的键组成了一棵树，如图 7-2，每个子节点都有一个指向父节点的指针。

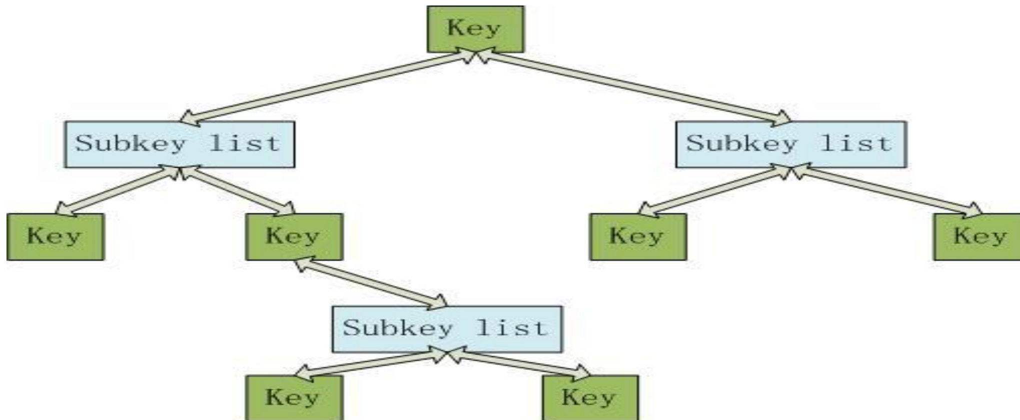

图 7-2 注册表键树状图

当然了 Key 里面还有很多指针指向其它信息，比如键值等，如图 7-3

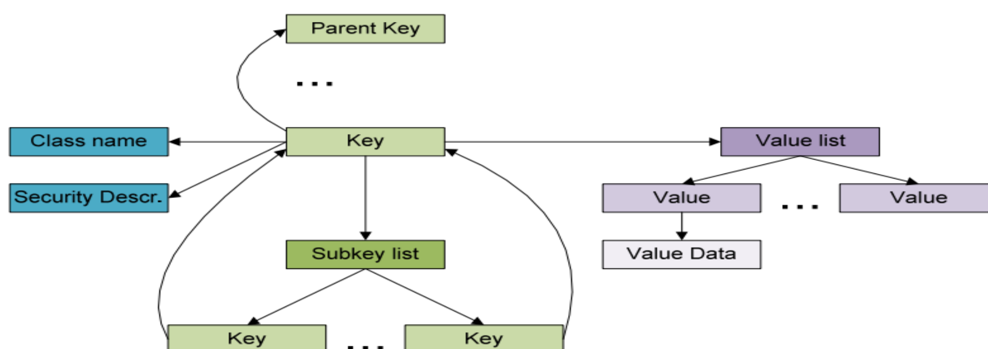

图 7-3 注册表键信息结构图

7.3 注册表文件解析

完整的开发注册表文件格式解析是一件很耗时的工作，虽然在上一节介绍了一下注册表格式的大体内容，但是这些内容离完整解析注册表还差很大的一截。所本软件在 Petter Nordahl-Hagen 的开源 The Offline NT Password Editor 工程里面的 ntreg access library 库的基础上进行二次开发。

这个库可以在参考文献列出的网站上找到源代码，但是直到最新版本也存在一些漏洞，一个典型就是无法解析含有斜杠的键值内容，还有一些其它零星的 bug 在本软件里得到了修正，并且开发了统一接口，便于编程调用。

7.4 注册表文件访问方法

前面提到，注册表文件在系统运行期间一直被 System 进程独占，这意味着其它进程无法访问到注册表，在 XP 下，可以在 ring3 通过复制 system 进程里的注册表文件句柄来获得访问权限，但是到了 vista 以后，已经不能以复制句柄的权限打开进程。所以在本软件里统一使用驱动来处理。

在前面文件管理有关章节提到过文件强制访问，首先以 READ_FILE_ATTRIBUTES(文件操作的最小权限)来打开注册表文件，注册表文件路径存储在这个位置：

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\hivelist

这里给的是 NT 下的路径，需要使用第四章提到的方法进行路径转换。在以最小权限打开文件后，需要把文件句柄传递到驱动里面，驱动通过修改句柄的权限来使应用层获得文件访问权限，修改句柄权限的实现在 6.4 节有详细描述。

7.5 注册表解析进一步完善

经过我多次实验，发现 Petter Nordahl-Hagen 的注册表访问库有一个很严重的问题就是不支持中文，注册表里面的键值存放方式是这样的：如果是和字符串兼容的类型，存放的方式是 UNICODE，二进制实际存放。键值名和子键名则很诡异，如果含有中文，存储为 UNICODE，否则存储为 ASCII，为了兼容我华夏文字，对 Petter Nordahl-Hagen 的库进行了大范围改造，主要改造的内容如下：首先编写一个超级字符串处理模块，这个字符串能同时处理 ASCII 和 UNICODE，实现常见的比较，连接，复制等操作，这个模块是整个改造过程的基础，主要定义如下：

```
typedef struct    tagSTRING
{
    unsigned long Size;
    unsigned long MaxSize;
    unsigned char *Buffer;
}POWERSTRING,*PPOWERSTRING;
```

第一个字段定义了 Buffer 里面存储的数据长度，而 MaxSize 定义 Buffer 最大可用的长度，基于这个结构设计了各种字符串处理函数，这样由于不需要以 ASCII 的\0，或者 Unicode 的\0 来区分长度了处理起来有很强的灵活性，更重要的是大部分字符串比较主要是用来判断相等或不等的情况，所以带了长度很方便比较。将 Petter Nordahl-Hagen 那个解析库里面的所有字符串处理函数替换成这个模块的字符串处理函数就可以很好的兼容中文了，当然了日文那些自然也不在话下。

7.6 本章小结

本章主要讨论了注册表的大致格式已经基于文件的访问方法，本论文附有一个注册表文件格式的文件的翻译，由于在第三方的库上进行二次开发，降低了很多难度，如果对注册表文件格式感兴趣，可以参考文献[5]、[6]和[7].

第八章 自我保护

8.1 基于 x86 的 inline Hook 引擎 ring0 及 ring3 编写

在 Windows32 位 PC 上，所有 API 接口都是使用 Stdcall 方式调用的，在内核里面除了 Stdcall 之外还有大量的 fastcall。标准调用下，调用 API 时通过堆栈传递参数，按 C 语言里面的来看，参数压入堆栈的数序是从左到右的，对于 fastcall 来说，前两个参数通过 ECX 和 EDX 寄存器传递，其余参数通过堆栈传递。所以对于 Stdcall 来说，给某个函数增加参数和修改参数只需要修改堆栈数据就行了，对于 fastcall 来说，除了修改堆栈，还要修改 ecx 和 edx。对于 inlineHook 前面章节已经有介绍了，这里主要说我编写的一段精巧的钩子函数，以便实现通用。

整个挂钩库的核心代码如下：

```
 void NAKED Stub()
 {    __asm
      {
           pop eax//取出调用者返回地址
           push OLD_CODE//把保存的原始函数开头代码压栈
           push eax//将调用者的返回地址放回去
           _emit 0xE9
           多个 nop//跳转到我们的接管函数
OLD_CODE:
           _emit 0xCC
           多个 nop//这里运行时写入被覆盖的地址，最后写入一个跳转指令，跳
转到将要被覆盖指令后面的指令处继续执行
         }
 }
```

我们在原始函数开头处跳转到上面这个 Stub 的地址，进入 Stub 前的堆栈构造如下：

调用者返回地址
原始函数参数 1
原始函数参数 2
原始函数参数……

堆栈上栈顶是调用这个 API 返回后将要执行的下一条指令的地址，依次下去是调

用这个 API 的各个参数，我们上面那个 Stub 的作用就是在要在调用者返回地址和原始函数参数 1 中间插入一个地址，而这个地址就是保存的将要被覆盖的原 API 开头的几个字节的地址（相当于一个函数指针），之后跳转到我们的中继函数处时，这个函数指针就成了中继函数的第一个参数了，结合上面那几个注释，进入我们中继函数时的堆栈情况如下：

调用者返回地址
原始函数开头地址（中继函数第 1 个参数）
原始函数参数 1 （中继函数第 2 个参数）
原始函数参数 2 （中继函数第 3 个参数）
原始函数参数……

从上图可知，我的中继函数的第一个参数其实就是原始函数的开头指令所在的地址了，这样后面我们的中继函数就没必要在面对使用裸函数和平衡堆栈等问题了，在上面这个 Stub 里面唯一使用的一个寄存器是 eax，选择这个寄存器是因为 API 返回值都是放在 eax 里面的，所以修改这个寄存器里的值一般没什么影响（也可以保存到一个全局变量里）。

对于 fastcall 的处理其实很简单，可以将其简单转换成 stdcall 调用处理，需要做的仅仅是先得到返回地址，再将 edx 和 ecx 一次入栈，添加原始函数地址后再将返回地址写到堆栈顶上。

上面这段精巧的 Stub 中继函数可以方便的移植到 ring0，只需要修改一下去除内存写保护已经多处理器挂钩安全就行了。对于多处理器，目前普遍采用的方法是使用 Interlocked 系列函数实现互锁修改地址。对于更多 CPU 的情况，一般使用投递空转 DPC 使其它核心或者 CPU 暂停执行。

8.2 进程自我防护方法

前面在进程保护那里已经说过了各种进程保护的防护了，现在我选择的挂钩点是 ObreferenceObjectByHandle 函数进程过滤，直接返回错误是一种很鲁莽的做法，因为可能有其它软件仅仅只是想看看你的进程名而已，对于这种请求应该放过。通过逆向各个版本下的系统内核文件可以知道，对于查询信息使用的权限是 0x40 。所以过滤函数如下即可：

```
NTSTATUS __stdcall Detour_ObReferenceObjectByHandle(
    DObReferenceObjectByHandle Fun,
    IN HANDLE    Handle,
    IN ACCESS_MASK    DesiredAccess,
    IN POBJECT_TYPE    ObjectType    OPTIONAL,
    IN KPROCESSOR_MODE    AccessMode,
    OUT PVOID    *Object,
    OUT POBJECT_HANDLE_INFORMATION    HandleInformation    OPTIONAL)
{
    NTSTATUS Status;
    PEPROCESS Eproc=0;
    BOOLEAN Denied=FALSE;
    if(*PsThreadType==ObjectType||*PsProcessType==ObjectType)
    {
        Status=Fun(Handle,DesiredAccess,ObjectType,AccessMode,Object,HandleInformation);
        if(NT_SUCCESS(Status)&&(*Object)!=NULL)
        {
          if(*PsThreadType==ObjectType)
            {
                Eproc=IoThreadToProcess((PETHREAD)(*Object));
                if(DesiredAccess!=0x40)
                {
                    Denied=TRUE;
                }

            }
            else
            {
                if(DesiredAccess!=0x400)
                    Denied=TRUE;
                Eproc=(PEPROCESS)(*Object);
```

```
            }
            ULONG Pid=(ULONG)PsGetProcessId(Eproc);
            PPROCESS_NODE Node=LookupPid(Pid);
            if(Denied&&Node&&Node.>Process!=PsGetCurrentProcess())
            {
                    KdPrint(("Process:%d
                        尝试操作保护进程!! 已拒绝\n",PsGetCurrentProcessId()));
                    ObDereferenceObject(*Object);
                    *Object=0;
                    return STATUS_ACCESS_DENIED;


            }
        }
        return Status;
    }
    return Fun(Handle,DesiredAccess,ObjectType,AccessMode,Object,HandleInformation);
}
```

上面的代码能够很好的保护进程不会被通过 TerminateXX 函数结束或者内存清零结束。

8.3 应用层自我防护

上面那个代码用来防止普通的远线程创建已经够用了，但是可能还是会被使用消息钩子等方式注入，所以这里还是需要防御消息钩子。消息钩子的加载时机是安装钩子后第一次消息到来。如果安装失败，每次消息道来系统都会尝试再次对我的进程加载钩子模块。但是消息钩子到底是使用哪个 API 来安装的呢？使用 Windbg 对 notepad 下以下断点：

Bp LoadLibraryA

Bp LoadLibraryW

Bp LoadLibraryExA

Bp LoadLibraryExW

会发现其实最后断在了 LoadLibraryExW 这里，所以拦截这个函数就可以了，但是注意除了钩子，正常的情况下也会调用这个函数的，所以还需要区分，消息钩子

一般来自 user32 里面。但是来自 User32 的也有可能是正常的模块，至于怎么过滤就仁者见仁了。为了方便得到调用者来自哪个模块，对上面的那个挂钩库做小小的改动。如下：

```
void NAKED Stub()
{
    __asm
    {
        pop eax
        push OLD_CODE
        push eax
        push eax
        _emit 0xE9
```

这样改动之后，返回地址就会以第一参数的形式传递给中继函数，过滤函数如下即可:

```
HMODULE __stdcall Detour_LoadLibraryEx(DWORD RetAddr,DLoadLibraryExW Fun, wchar_t* lpFileName,
                    HANDLE hFile,  DWORD dwFlags)
{
    HMODULE hModule=0;
    HMODULE Temp=GetModuleHandle(L"user32.dll");
    if(!Temp)
        return Fun(lpFileName,hFile,dwFlags);
    if(Info.SizeOfImage==0)
        GetModuleInformation(GetCurrentProcess(),Temp,&Info,sizeof(MODULEINFO));
    if(RetAddr>(DWORD)Info.lpBaseOfDll&&RetAddr<(DWORD)Info.lpBaseOfDll+Info.SizeOfImage)
    {//check if call this function in user32.dll module
        if(!CheckFileTrust(lpFileName))//Vierify Module
        {
            SetLastError(5);
            return NULL;
        }
    }
```

```
    return Fun(lpFileName,hFile,dwFlags);
}
```

8.4 防御消息洪水攻击

主要过滤一些 Shadow SSDT 里面的函数即可，常见的函数有如下几个，NtUserPostMessage、NtUserPostThreadMessage，NtUserFindWindowEx，例如著名的安全软件 QQ 电脑管家的的窗口保护挂钩了如下几个函数，如图 8-1

396	NtUserFindWindowEx	0x9B571308	ssdt hook	0x9FD89056	C:\Program Files\Tencent\QQPCM
423	NtUserGetForegroundW...	0x9B571562	ssdt hook	0x9FD75E0D	C:\Program Files\Tencent\QQPCM
495	NtUserMoveWindow	0x9B57177E	ssdt hook	0x9FD43FB8	C:\Program Files\Tencent\QQPCM
515	NtUserQueryWindow	0x9B571542	ssdt hook	0x9FDB07DA	C:\Program Files\Tencent\QQPCM
536	NtUserSendInput	0x9B571B50	ssdt hook	0x9FE4A27E	C:\Program Files\Tencent\QQPCM
560	NtUserSetParent	0x9B571610	ssdt hook	0x9FD68A0A	C:\Program Files\Tencent\QQPCM
578	NtUserSetWindowLong	0x9B5716C4	ssdt hook	0x9FDA7351	C:\Program Files\Tencent\QQPCM
579	NtUserSetWindowPlace...	0x9B571900	ssdt hook	0x9FD39E1D	C:\Program Files\Tencent\QQPCM
580	NtUserSetWindowPos	0x9B57183E	ssdt hook	0x9FD70140	C:\Program Files\Tencent\QQPCM
591	NtUserShowWindow	0x9B5719B4	ssdt hook	0x9FD75440	C:\Program Files\Tencent\QQPCM

图 8-1

8.5 本章小结

本章重点讨论了一个本软件所用到的 inline 引擎，基于这个 inline 引擎实现了自保护功能，关于进程的更多自保护实现已经在 4.3 节进行了详细论述，这里选择了一个不太底层的保护方法，大部分情况还是够用了。

第九章 软件签名系统设计与实现

9.1 为什么需要验证

Windows 程序是否能打开驱动进行命令控制不但和其安全描述符有关，还和进程的权限有关，但是这种基于 Windows 本身的安全描述符的验证机制有很多缺陷，首先是只要是用户运行的程序，一般都有打开我们驱动的权限，而我们驱动含有一些敏感功能，比如获取任意句柄的权限这个功能，自己在注册表解析部分使用，但是也可能被外人利用来非法获取任意句柄的最高权限，造成了一种安全隐患，还有我们的驱动必定含有某些漏洞，如果别人随意能够打开我们的驱动进行控制就可能通过发送非法控制信息触发我们驱动的程序漏洞，实现恶意目的。为了避免别的进程非法打开我们驱动获得驱动使用权，需要对调用者进行验证，通过进程名来验证是不安全的，所以必须使用专门的验证算法。

通常的验证算法可以使用验证文件 Hash 值的方法，但是这种方法有很多缺陷，很容易被攻破，例如一些软件的做法是使用记录进程文件 MD5 值的方法。在我这里使用标准 PE 数字签名技术。

9.2 RSA 加密算法

加密算法按加密和解密密钥是否是一个分为对称加密和非对称机密，RSA 是一种非对称加密，被广泛的用于数字签名领域，同样椭圆加密算法也可用于数字签名，本软件使用 RSA。

非对称加密算法一般有两个密钥：公钥和私钥，在数字签名里，一般使用私钥加密，使用公钥解密签名信息，从公钥无法推出私钥从而使数字签名不可伪造，除非对驱动进行专门 pach。本软件主要使用 SHA512 信息摘要算法已经 RSA 非对称加密算法来实现签名验证系统。通常大部分软件使用的是 Md5 和 RSA。其他可供选择的 Hash 算法还有 md4，SHA224，SHA256，SHA384，whirlpool，SHA1，ripemd 等，加密算法有 RSA（主要依赖大素数乘积分解困难），ECC（椭圆加密算法）。具体的加密流程不再论文里进行论述。

9.3 PE 文件的签名及验证实现

本文的签名及验证算法如下（公钥为 PUKEY，私钥为 PRKEY）:

首先看看 PE 文件的 DosStub 空间够不够保存签名信息，本文的签名信息长度是 128 字节，如果空间不够，就给 PE 文件专门增加一个区段。对除签名空间除外

的部分进行 SHA512 求 HASH 值，在这里可以使用其它算法增加干扰。之后使用 PRKEY 加密签名信息写入 PE 文件，验证时使用同样的加密 Hash 算法得到 PE 文件的 HASH 值，使用公钥 PUKEY 解密签名信息，看两个信息是否一致，如果是一致的则调用者合法。

其签名函数如下，验证函数类似。

```
BOOLEAN SignPEFile(wchar_t *FileName)
{
    ULONG         Space=0,HashPos=0;
    HANDLE              hFile=SOpenFile(FileName);
    LARGE_INTEGER FileSize={0},Move={0};
    BOOLEAN                Ret=0;
    unsigned char    *Buffer=0;
    unsigned char     Hash[80]={0};
    unsigned char     Encrypt[200]={0};
    unsigned int           Size=0;
    if(hFile==0)
        return FALSE;
    Space=GetSpacePeHead(hFile);
    CloseHandle(hFile);
    if(!Space)
    {
        return FALSE;
    }
    if(Space<MINSPACE)
    {
        CPELibrary Lib;//=new CPELibrary();
        Lib.OpenFile(FileName);
        Lib.AddNewSection(SECTION_NAME,512);
        DeleteFile(FileName);
        Lib.SaveFile(FileName);
    }
```

```
hFile=SOpenFile(FileName);
if(!GetFileSizeEx(hFile,&FileSize)||FileSize.HighPart)
{
    CloseHandle(hFile);
    return FALSE;
}
Buffer=(UCHAR*)malloc(FileSize.LowPart);
if(!Buffer)
{
    CloseHandle(hFile);
    return FALSE;
}
if(!ReadFile(hFile,Buffer,FileSize.LowPart,(DWORD*)&FileSize.HighPart,0)||(FileSize.LowPart!=FileSize.HighPart))
{
    free(Buffer);
    CloseHandle(hFile);
    return FALSE;
}
Ret=GetPeHash(Buffer,FileSize.LowPart,Hash,&HashPos);
if(Ret)
{
    RSAPrivateEncrypt(Encrypt,&Size,Hash,64,&PrivateKey);
    Move.QuadPart=HashPos;
    if(SetFilePointerEx(hFile,Move,&FileSize,FILE_BEGIN))
    {
        Ret=WriteFile(hFile,Encrypt,128,(DWORD*)&Size,0);
        Ret&=(Size==128);
    }else
    {
        Ret=0;
    }
```

```
    }
    free(Buffer);
    CloseHandle(hFile);
    return Ret;
}
```

其中带了 S 的函数都是表示可以在内核和应用层通用的函数，这样可以减少重复编码量。设置不同的编译开关就可以轻易实现代码复用。

9.4 本章小结

本章主要讨论了软件签名的实现技术，使用软件签名可以实现软件自校验等其他目的，使用这个签名技术可以很容易的防止驱动被非法程序调用，但是一旦程序自身受到感染，也就无法打开驱动了，这算是本签名的一个瑕疵。

第十章 结论

10.1 软件调试测试

本软件主要在 windows XP SP2 上完成全部功能测试，在 windows 7 下完成部分功能测试，主要的几个模块测试项目如下：

测试项目	测试预期	测试结果
进程管理	1、 能够检测到隐藏进程（HideTool） 2、 能够结束顽固进程（KV2010） 3、 能够枚举隐藏进程模块	成功检测 成功结束 成功检测
内核钩子	1、 两个系统表的钩子检测及恢复 2、 普通导出函数 inline 检测 3、 深度 inline 检测（狙剑的 ObInsertObjet 钩子）	正常 正常 正常
内核	1、 系统回调函数检测正常 2、 DPC 检测 3、 WorkItem 检测	正常 正常 正常
注册表	1、 基于文件操作的增删改 2、 隐藏注册表键值	正常 成功检测
文件	1、 隐藏文件检测 2、 正在运行进程文件强制删除 3、 文件解锁	成功 成功 正常

一些其他功能本表未列出。进程功能运行结果如图 10-1：

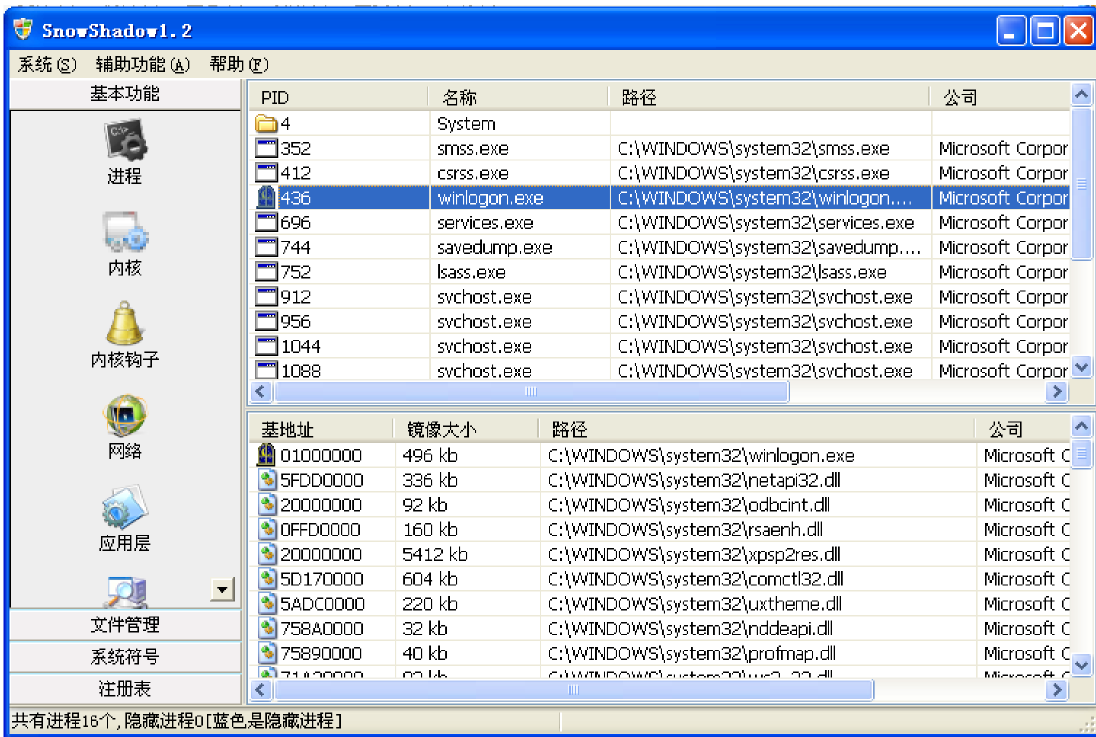

图 10-1

内核 DPC 检测运行结果如图 10-2

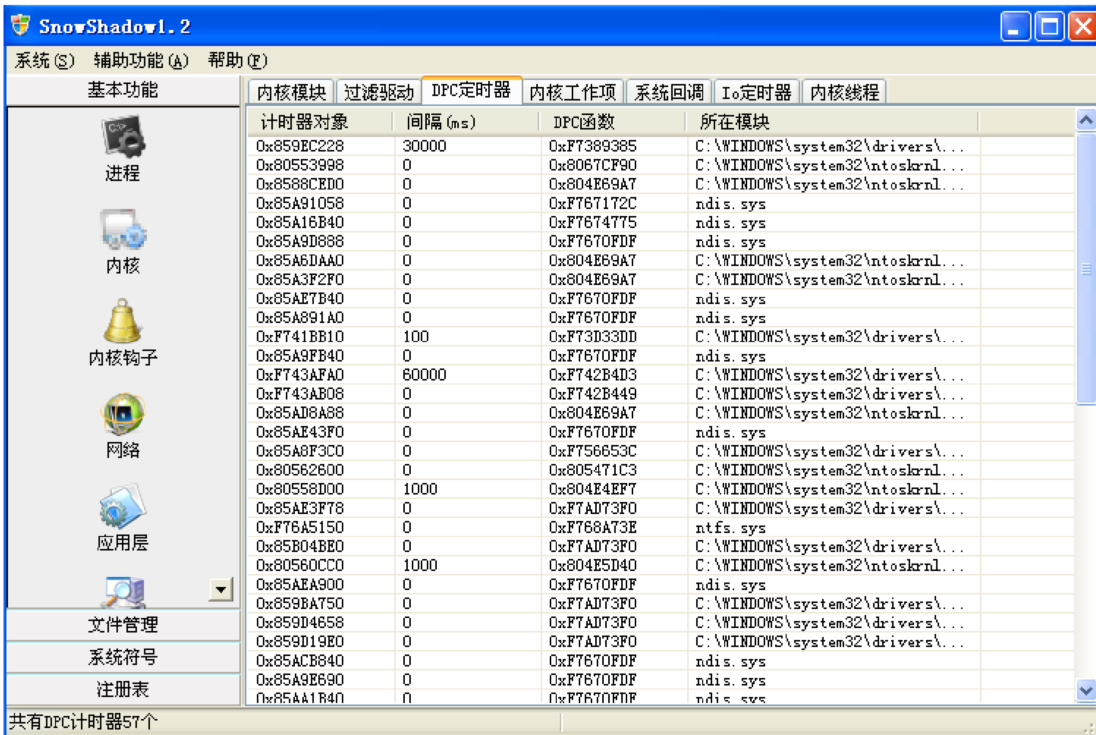

图 10-2

文件检测运行如果 10-2

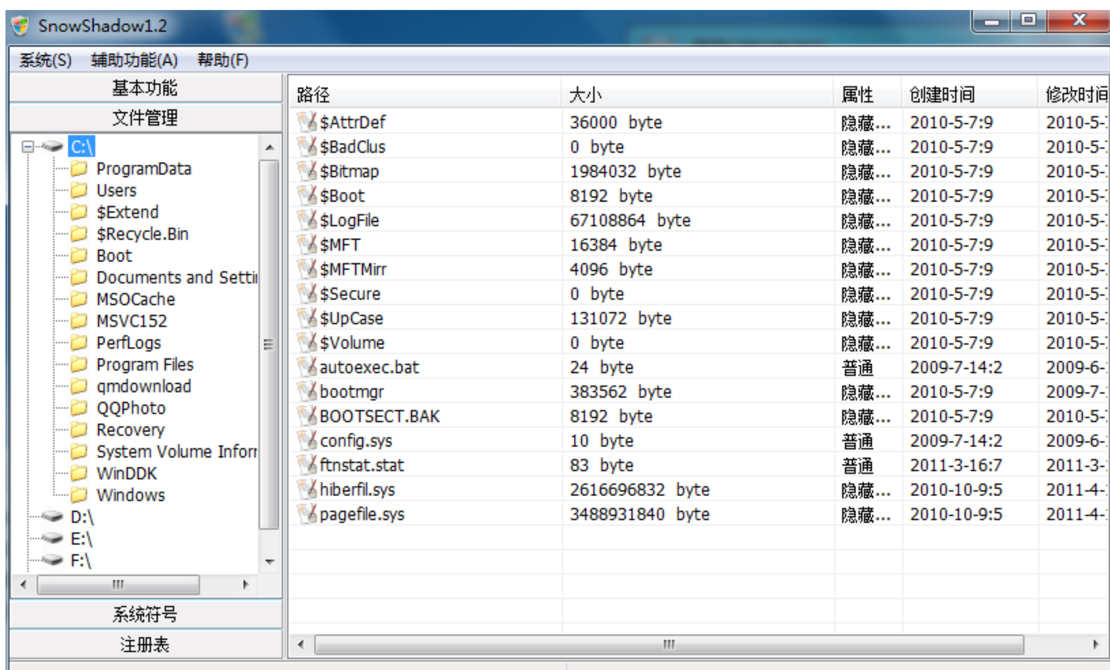

图 10-3

注册表编辑运行结果如图 10-4

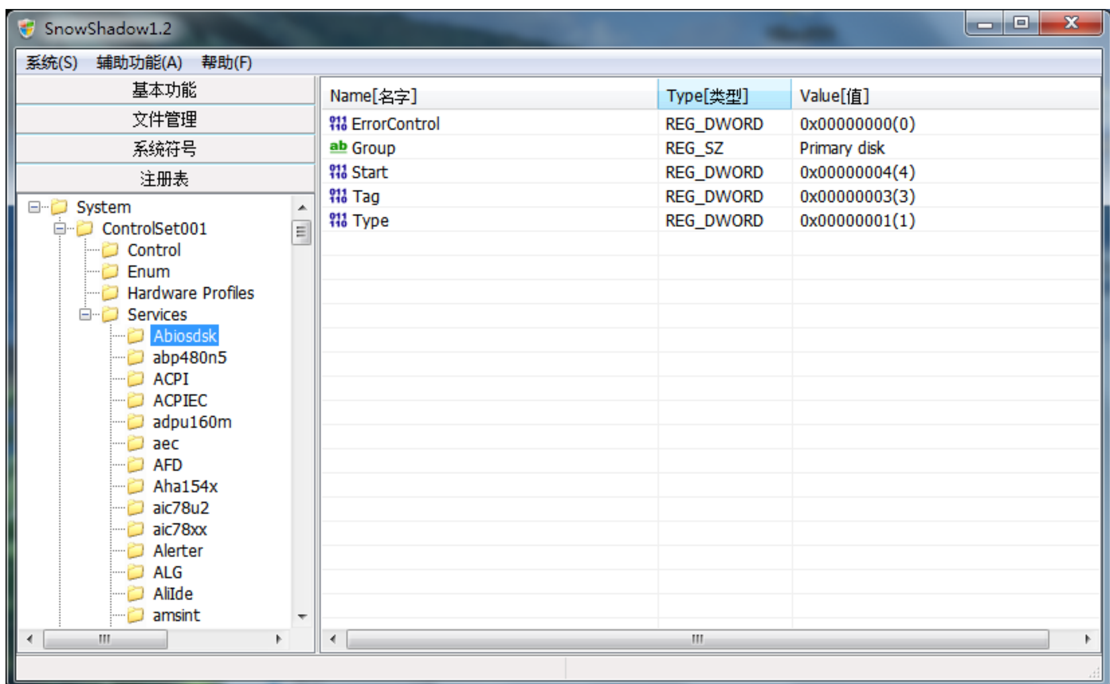

图 10-4

鉴于论文篇幅，不能对所有功能进行一一截图。

10.2 全文总结

本论文在作者开发反 Rootkit 工具 SnowShadow 中总结而来，能够胜任检测 Rootkit 的大部分工作，一些小的功能还不完善。目前的安全形势下，除了一些恶意软件，Rootkit 技术也被很多合法软件所采用。比如文件过滤驱动通常用来做安全软件的自保护以及透明加密，文件重定向等。在各种反游戏外挂，反调试中更是运用广泛，这些合法软件像恶意软件一样肆意修改 Windows 内核，导致 Windows 系统轻者性能下降，重者蓝屏，其中的 SSDT 修改更是被使用频繁，一些软件作者由于水平有限或者其他原因钩子函数编写有缺陷，很容易引起系统崩溃，这类软件的出现加大了 Windows 的崩溃概率。通过本软件可以修复大部分的内核修改 patch。

10.3 展望

本软件目前可以稳定运行于 Windows7 和 xp 的各个版本，暂时不支持其它版本的操作系统，在未来的版本里将会逐步兼容其它版本的操作系统。其次本软件的功能的方面还有许多地方需要完善：

1、 支持的操作系统版本还比较少，后面再慢慢改进。
2、 其它一些辅助功能还没有完善。
3、 限于论文篇幅，一些其它的功能没有写进论文，主要有网络端口检测，内核工作项检测，进程定时器检测，进程热键检测等。

Rootkit 和反 Rootkit 技术是一种相对的存在，所谓道高一尺，魔高一丈，反 Rootkit 是一条没有终点的路，大家都在争夺内核这块至高地，简单的防御 Rootkit 的方法是防止加载驱动程序。

参考文献

[1] 段刚《加密解密(第二版)》，电子工业出版社，2003
[2] Gary Nebbett,《Windows NT 2000 Native API Reference》，不详，不详
[3] 张帆、史彩成，《Windows 驱动开发技术详解》，电子工业出版社，2009
[4] Greg Hoglund,《Rootkits: Subverting the Windows Kernel》，清华大学出版社，2007
[5] The Internal Structure of the Windows RegistryMSc Thesis - Academic Year 2008-2009
Peter Norris BSc (Hons), MBCS February 2009
http://amnesia.gtisc.gatech.edu/~moyix/suzibandit.ltd.uk/MSc/
[6] The Windows NT Registry File Format by Timothy D. Morgan
http://www.sentinelchicken.com/research/registry_format/
[7] The Offline NT Password Editor (c) 1997-2010 Petter Nordahl-Hagen 2011
http://pogostick.net/~pnh/ntpasswd/editor.html
[8] 看雪论坛,2011
http://bbs.pediy.com/
[9] Debugman 论坛,2011
http://www.debugman.com/
[10] 谭文，《天书夜读-从汇编语言到 Windows 内核开发》，电子工业出版社，2009
[11] 张正秋，《Windows 应用程序捆绑核心编程》，清华大学出版社，2007
[12] 毛德操，《Windows 内核情景分析》，电子工业出版社，2009
[13] 谭文、杨潇等，《寒江独钓-Windows 内核安全编程》，电子工业出版社，2009
[14] 罗云彬，《Windows.环境下 32 位汇编语言程序设计》，电子工业出版社，2009
[15] 张银奎，《软件调试》，电子工业出版社，2008
[16] 不详，《Undocumented Windows NT》，不详，不详
[17] Ric Vieler，《Professional Rootkits》，不详，2007
[18] Gary Nebbett,《Windows NT 2000 Native API Reference》，不详，不详
[19] 黑客防线，《黑客防线月刊》，2008.1-2010.2
[20] Mark E. Russinovich, David A. Solomon《Microsoft® Windows® Internals》
[21] CodeProject 网站 http://www.codeproject.com/
[22] Chris Eagle 译者： 石华耀 段桂菊，《IDA Pro 权威指南》，人民邮电出版社，2010
[23] RAS 官方网站
http://www.rsa.com/

[24] DIA 开发包参考文档

[25] NTFS 相关 http://www.tuxera.com/community/ntfs-3g-download/

[26] NTFS Documentation, by Richard Russon, Yuval Fledel，2006

[27] Microsoft 《FAT 文件系统规范中文版》，microsoft，不详

致谢

值此论文完成之际，谨向唐勇导师致以崇高的敬意和衷心的感谢。导师严谨细致、一丝不苟的作风一直是我工作、学习中的榜样；他循循善诱的教导和不拘一格的思路给予我无尽的启迪。

本论文所的软件大量参考了看雪论坛以及 Debugman 论坛上的许多优秀帖子，参考的帖子太多，已经无法一一统计在此一并感谢。