doi :10.3969/j.issn.1671-1122.2011.12.002

基于FUZZING测试技术的Windows内核安全漏洞挖掘方法研究及应用

姚洪波[1]，尹亮[2]，文伟平[2]

（1. 中国石油大学计算机与通信工程学院，山东东营 257062；2. 北京大学软件与微电子学院，北京 102600）

摘 要：随着技术的进步，Windows操作系统日益完善，多种内存保护技术的结合使得传统的基于缓冲区溢出攻击越来越困难，在这种情况下，内核漏洞往往可以作为突破安全防线的切入点。该论文首先分析了现有Windows内核漏洞挖掘方法，阐述了Windows内核下进行Fuzzing测试的原理和步骤，针对Windows win32k.sys对窗口消息的处理、第三方驱动程序对IoControlCode的处理、安全软件对SSDT、ShadowSSDT函数的处理，确定数据输入路径，挖掘出多个内核漏洞，验证了该方法的有效性。

关键词：安全漏洞；漏洞挖掘；漏洞补丁

中图分类号：TP393.08 **文献标识码：**A **文章编号：**1671-1122（2011）12-0009-08

Based on the FUZZING Lead to a New Mining Method Based on Windows Kernel Vulnerability

YAO Hong-bo[1],YIN Liang[2], WEN Wei-ping[2]

(1. *School of Computer & Communication Engineering, China University of Petroleum, Dongying Shandong* 257061*, China;*
2. *Department of Information Security, SSM, Peking University, Beijing* 102600*, China*)

Abstract: With advances in technology, Windows operating system has improved steadily. Combining many memory protection technologies made the traditional buffer-overflow-based attacks to be more useless. In this case, the kernel vulnerability can be used to break through the security line of defense as a starting point. This paper researches the existing mining Windows kernel vulnerability, then proposes a methods on how to find Windows kernel vulnerability based on Fuzzing, summarizes the existing Fuzzing technology, selects three kernel Fuzzing goal which are Windows win32k.sys processing of window messages, third-party driver for IoControlCode processing, security software on the SSDT, ShadowSSDT function of processing, after the analysis of the three principles, Fuzzing data are designed and data input path are identified. Finally, using this method found in case of Windows operating system unknown vulnerabilities verify the validity of the method.

Key words: security vulnerabilities; digging vulnerabilities; vulnerability patches

0 引言

随着技术的进步，Windows操作系统在安全性方面日益完善，尤其是自Vista操作系统应用以来，GS Stack Protection、SafeSEH、Heap Protection、DEP、ASLR等技术在Windows操作系统中得到了广泛的应用[1-3]，这些技术的结合使得传统的基于缓冲区溢出攻击越来越困难，特别是近几年来，微点、360安全卫士等主动防御软件得到了广泛的安装、使用，利用普通应用软件的漏洞进行攻击已经难以奏效，在这种情况下，容易被人忽略的内核漏洞往往可以作为突破安全防线的切入点。

Fuzzing技术最早是在1989年由Wisconsin大学的Barton Miller教授提出，是一种验证程序安全、发现软件漏洞的重要手段[4]，在传统的应用软件漏洞挖掘中得到了广泛应用，可以发现应用软件中多种类型漏洞，如堆栈溢出、整型溢出、格式化字符串溢出等缓冲区溢出漏洞，IE浏览器、Word、Excel、Flash等软件的特殊文件格式漏洞，字符过滤不严等可执行脚本漏洞。Fuzzing技术的应用不需要测试目标是否开源，并且易于自动化，能精确地找到漏洞的所在，正成为越来越流行的应用软件漏洞挖掘技术。

但是，现有的Fuzzing技术主要应用于应用软件的漏洞挖掘，较多的是针对微软的IE浏览器、Word、Excel、Adobe的Flash，并没有应用到Windows内核漏洞上。通过总结Fuzzing技术在应用软件漏洞挖掘上的应用，本文提出将Fuzzing技术应用

收稿时间：2011-11-12

基金项目：国家自然科学基金资助项目[61170282]

作者简介：姚洪波（1976-），女，山东，讲师，硕士，主要研究方向：软件工程；尹亮（1986-），男，湖南，硕士研究生，主要研究方向：漏洞分析和漏洞挖掘；文伟平（1976-），男，湖南，副教授，博士，主要研究方向：网络攻击与防范、恶意代码研究、信息系统逆向工程和可信计算技术等。

到 Windows 内核漏洞的挖掘中，由于 Windows 内核、第三方公司驱动程序的开发人员在内核、驱动程序设计开发的过程中，很难考虑到应用层所有可能的输入情况，如果在应用层向内核、驱动程序中输入不符合常规的数据，那么内核、驱动程序代码、功能逻辑等缺陷就有可能被检测出来。

1 针对 Windows 内核的 Fuzzing 安全漏洞挖掘方法

针对 Windows 内核漏洞挖掘的 Fuzzing 技术主要原理是通过给内核、驱动程序提供大量畸形的输入数据，使得内核、驱动程序发生异常，引发系统蓝屏崩溃，生成系统转储文件，里面记录有函数的调用顺序、系统崩溃时的 CPU 状态、寄存器状态以及堆栈状态等信息，通过分析这些信息，了解异常发生的位置和原因，进一步结合静态分析技术和动态调试技术判断异常函数的代码、逻辑缺陷，从而挖掘出内核、驱动程序的未知漏洞。Fuzzing 技术具有应用广泛、自动化强等特点，只要内核、驱动程序中的某个功能模块与应用层存在相互通信的接口，就可以利用 Fuzzing 技术进行漏洞挖掘，其挖掘过程也可以通过自动化实现，一旦开始 Fuzzing 测试后，可以不需要人工干预即可完成指定功能、指定范围的测试。

1.1 测试对象选取

Fuzzing 技术的第一步是确定 Fuzzing 目标，通过分析、总结已经公开披露的 Windows 内核漏洞，结合 Fuzzing 技术的实现原理，本文将选取以下三部分做为 Fuzzing 目标。

1.1.1 Windows win32k.sys对窗口消息的处理

在 Windows 操作系统的设计中，视窗（Windows）、POSIX、OS/2 是内核所支持的三个子系统，视窗是其中首要的子系统，复制图形界面的操作。从 Windows NT 4.0 开始，微软将图形操作移到了内核里，以可安装的内核模块 win32k.sys 的形式对内核加以扩充，win32k.sys 提供视窗操作，也提供为此所需的“视窗间通信”，即 Windows 窗口消息，负责处理 Windows 的窗口消息的函数庞大而复杂，其中存在着不少漏洞，通过 Fuzzing 技术能有效地挖掘 win32k.sys 潜在的漏洞。

1.1.2 第三方驱动程序对IoControlCode的处理

应用程序与驱动程序的通信主要是依靠调用 DeviceIoControl 函数来实现的，当调用该函数向驱动发出 I/O 请求后，在内核中由操作系统将其转化为 IRP 的数据结构，并分发到对应驱动的派遣函数中。在调用 DeviceIoControl 函数时，需要指定一个 IoControlCode（IO 控制码），驱动程序根据 IoControlCode 调用对应的功能函数。从已公布的针对第三方驱动程序的漏洞来看，IoControlCode 类型的漏洞是最多的，它作为 Ring0/Ring3 通信的重要方式，很有可能会出现参数处理不当，或者内部逻辑、设计缺陷引发的问题。

1.1.3 安全软件对Native API的处理

Native API 是提供给 Ring3 调用，接收 Ring3 传入的参数，在 Ring0 完成最终功能的函数，现有的安全软件为了防御木马、病毒，对这些函数中某些重要的函数进行了 HOOK/Inline Hook，如果对参数的处理不当，或者有内部逻辑错误，则可能产生内核漏洞。由于这些函数的参数是从用户态传入的，是可控的，能很好地利用 Fuzzing 技术进行漏洞挖掘。

1.2 测试方法及流程

采用 Fuzzing 技术挖掘漏洞在很大程度上依赖于 Fuzzing 目标、数据格式，并没有通用的方法，通过对现有 Fuzzing 技术研究总结，本文提出在 Windows 内核漏洞的挖掘中 Fuzzing 的过程主要有以下几个步骤。

1.2.1 确定Fuzzing 的目标

从安全的角度出发，一个信任级别是对资源的许可的集合，信任边界是两个信任级别的交界，例如用户态是一个信任级别，内核态也是一个信任级别，无论是 Windows 内核还是第三方驱动程序，它们的输入都来自于用户态的应用程序，应用程序通过系统调用，使得程序流程从用户态转移到内核态，这种情况就可以认为是穿越了信任边界，因此在选取内核 Fuzzing 目标时，应该对用户态与内核交互的方式方法进行分类，综合选取 Fuzzing 目标。

1.2.2 分析数据格式，构造畸形数据

这一步在 Fuzzing 中比较重要，畸形数据的构造直接影响 Fuzzing 效果，这就需要针对已选取的 Fuzzing 目标，分析其输入数据的格式、取值范围，最理想的情况是构造出所有可能的输入数据，但在实际情况下，从时间和空间上综合考虑，只能选择具有代表性范围的数据。畸形数据可以是预定义的，可以通过对合法数据进行改变产生，也可以根据数据的格式动态产生。构造畸形数据应该由 Fuzzing 程序自动化完成。

1.2.3 确定数据输入路径

针对 Windows 内核漏洞的挖掘，需要考虑到在应用层中能够通过哪些系统调用与内核进行通信，那么畸形数据的输入路径就是这些系统调用，从而将畸形数据输入到内核、驱动程序中。

1.2.4 分析蓝屏后生成的系统转储文件

实时监控目标程序的运行，确定引起目标程序崩溃的畸形数据，这对于应用程序来说可能是难以实现的一个步骤，例如在对 Office Word 的 Fuzzing 测试中，如果生成的某个 Word 文件造成 Office Word 崩溃退出，需要在成千上万个 Word 文件中确定出这个文件，另外，引发异常时的 CPU、寄存器等状态难以获得，这些都将影响对 Fuzzing 结果的分析。而采用 Fuzzing 技术挖掘 Windows 内核漏洞时则非常简单，如果某个畸形的数据造成 Windows 内核的崩溃，Windows 会将崩溃时的系统信息存储到系统转储文件中，里面包含了出错时的寄存器信息、函数调用堆栈等，用于进行进一步分析。

1.2.5 确定漏洞利用的可能性

针对 Windows 内核的 Fuzzing 测试一般是在管理员用户下进行，如果通过 Fuzzing 引发了系统崩溃，那么需要进一步确定是否能在普通用户下触发崩溃，是只能造成拒绝服务，还是可以写任意内核地址，或者能读取任意内核地址造成信息泄漏等等。

这个步骤一般靠分析者结合静态分析技术和动态调试技术完成，往往需要有很深厚的 Windows 内核安全方面的知识和经验。

1.3 针对Windows win32k.sys处理窗口消息的漏洞挖掘

1.3.1 Windows窗口类

在 Windows 操作系统中，应用程序的每个窗口都是一个窗口类的成员，窗口类是一个属性集，Windows 在创建应用程序的窗口时使用它作为一个模板。每个窗口类有一个与之相应的窗口过程，由同类窗口所共享，窗口过程为该类的所有窗口处理消息，从而控制它们的特性和外观。

应用程序必须在它创建某类窗口之前注册这个窗口类，注册一个窗口类也就是把一个窗口过程、类风格及其他一些类属性与类名联系起来。如果应用程序在函数 CreateWindow 或 CreateWindowEx 中指定一个类名，操作系统就创建一个具有与这个类名相应的窗口过程、风格及其他属性的窗口。

窗口类的结构体定义如下所示：

表 1 窗口类的结构体定义

```
typedef struct {
    UINT    cbSize;                 // 结构体的字节数
    UINT    style;                  // 窗口类的风格
    WNDPROC    lpfnWndProc;         // 窗口过程
    int     cbClsExtra;
    int     cbWndExtra;
    HINSTANCE       hInstance;   // 该窗口类的窗口过程所属的应用实例
    HICON           hIcon;       // 该窗口类所用的像标
    HCURSOR         hCursor;     // 该窗口类所用的光标
    HBRUSH          hbrBackground;          // 该窗口类所用的背景刷
    LPCTSTR         lpszMenuName;           // 该窗口类所用的菜单资源
    LPCTSTR         lpszClassName;          // 该窗口类的名称
    HICON           hIconSm;     // 该窗口类所用的小像标
} WNDCLASSEX, *PWNDCLASSEX;
```

有三种类型的窗口类：系统类、应用程序全局类、应用程序本地类。系统类是由 Windows 注册的，有些系统类可以由所有的应用程序使用，有些只能由系统使用。表 2 列出的是可以由所有应用程序使用的系统类：

表 2 可以由所有应用程序使用的系统类

窗口类	说明
Button	按钮窗口类
ComboBox	组合框窗口类
Edit	文本控件窗口类
ListBox	下拉列表窗口类
MDIClient	多文档界面客户端窗口类
ScrollBar	滚动条窗口类
Static	静态控件窗口类

表 3 列出的是由系统使用的系统类：

表3 由系统使用的系统类

窗口类	说明
ComboLBox	组合框中包含的下拉列表窗口类
DDEMLEvent	动态数据交换事件窗口类
Message	纯消息窗口类
#32768	菜单窗口类
#32769	桌面窗口类
#32770	对话框窗口类
#32771	任务选择窗口类
#32772	图标窗口类

在 Windows 中，系统类由系统在应用程序第一次调用 User32.dll 或者是 GDI32.dll 中的函数时注册，应用程序不能销毁这些系统类，系统类窗口过程位于 win32k.sys，这些负责处理窗口消息的函数庞大而复杂，可能会存在安全问题。

1.3.2 确定消息输入路径

向一个窗口发送消息可以调用 SendMessage 和 PostMessage 函数。两个函数的定义如图 1 所示。

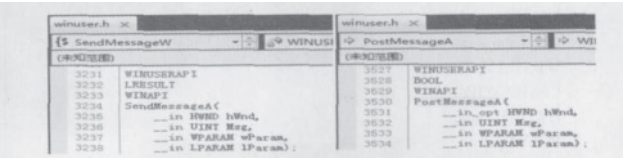

图 1 SendMessage和PostMessage函数定义

SendMessage 函数将指定的消息发送到一个或多个窗口，直到窗口程序处理完消息再返回，是同步消息投放函数。如果指定的窗口是由调用线程创建的，则窗口程序立即作为子程序调用。如果指定的窗口是由不同线程创建的，则系统切换到该线程并调用恰当的窗口程序。线程间的消息只有在线程执行消息检索代码时才被处理。发送线程被阻塞直到接收线程处理完消息为止。

参数 1：

hWnd：其窗口程序将接收消息的窗口的句柄。如果此参数为 HWND_BROADCAST，则消息将被发送到系统中所有顶层窗口，包括无效或不可见的非自身拥有的窗口、被覆盖的窗口和弹出式窗口，但消息不被发送到子窗口。

Msg：指定被发送的消息，0～0x400 的消息值由 Windows 保留。

wParam：通常是一个与消息有关的常量值

lParam：通常是一个指向内存中数据的指针。

返回值：返回值指定消息处理的结果，依赖于所发送的消息。

PostMessage 函数将一个消息放入到与指定窗口创建的线程相联系消息队列里，不等待线程处理消息就返回。消息队列里的消息通过调用 GetMessage 和 PeekMessage 取得。

参数 2：

hWnd：其窗口程序接收消息的窗口的句柄。可取有特定含义的两个值：

HWND.BROADCAST：消息被寄送到系统的所有顶层窗口，包括无效或不可见的非自身拥有的窗口、被覆盖的窗口和弹出式窗口。消息不被寄送到子窗口。

NULL：此函数的操作和调用参数 dwThread 设置为当前线程的标识符 PostThreadMessage 函数一样。

Msg：指定被寄送的消息，0～0x400 的消息值由 Windows 保留。

wParam：通常是一个与消息有关的常量值。

lParam：通常是一个指向内存中数据的指针。

返回值：如果函数调用成功，返回非零值；如果函数调用失败，返回值是零。

如果一个窗口采用系统类来创建，当其他应用程序调用 SendMessage 和 PostMessage 函数向该窗口发送消息时，消息将会插入到目标窗口的消息队列，目标窗口通过循环不断地从消息队列中取出消息，并调用 DispatchMessage 函数分发消

息给窗口处理函数。DispatchMessage 函数将经过 User32.dll 中的 NtUserDispatchMessage 函数进入内核，对应 win32k.sys 中的 NtUserDispatchMessage 函数。NtUserDispatchMessage 函数接受一个消息结构体指针做为参数，消息结构体定义如下：

```
typedef struct {
    HWND hwnd;                  // 窗口句柄
    UINT message;               // 消息标识
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG;
```

wParam、lParam 含义由具体的消息处理函数决定。

win32k!NtUserDispatchMessage 函数检查消息结构体地址，并复制到内核空间中，然后调用 xxxDispatchMessage 函数。如果窗口对应的窗口类为系统类，xxxDispatchMessage 函数将直接调用位于内核空间中的窗口过程。否则将调用 win32k!SfnDWORD 函数返回用户态进行处理。

1.3.3 畸形消息数据的构造

主要方式是，选择一个窗口类创建窗口，向这个窗口调用 SendMessage 和 PostMessage 函数发送消息，遍历由 Windows 保留的 0 ~ 0x400 消息值，对 wParam 和 lParam 畸形化处理。在通常情况下，lParam 为一个指向内存中数据的指针，因此也需要畸形化处理 lParam 指向的内存中数据。畸形化方案如表 4 所示。

整个 Fuzzing 过程如下图 2 所示。

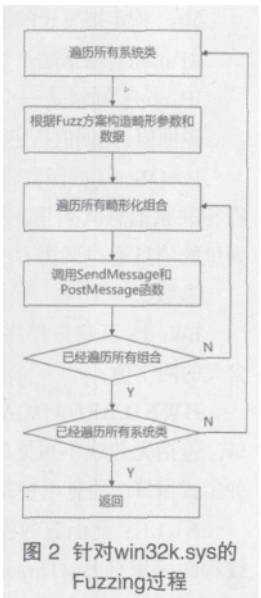

图 2 针对win32k.sys的Fuzzing过程

1.4 针对第三方驱动程序处理 IoControlCode 的漏洞挖掘

1.4.1 驱动对象、设备对象

在编写驱动程序时，需要填写一个名为 DRIVER_OBJECT（驱动对象）的结构体，每个驱动程序都会有唯一的一个驱动对象与之对应。IO 管理器负责加载驱动程序，驱动对象是在加载驱动程序时由 Windows 对象管理程序创建。驱动程序需要在 DriverEntry 函数中初始化。DRIVER_OBJECT 中含有派遣函数指针，这些派遣函数用来处理发送到驱动的各种请求，由 IO 管理器决定对某个请求调用哪个派遣函数。

每个驱动程序会创建一个或多个 DEVICE_OBJECT（设备对象）。每个设备对象都会有一个指针指向下一个设备对象，形成一个设备链。如图 3 所示。

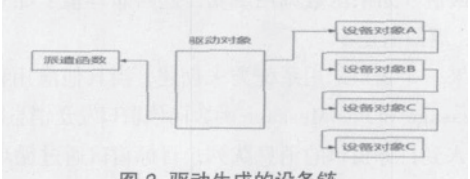

图 3 驱动生成的设备链

表4 针对win32k.sys的参数、数据畸形化方案

参数	方案	说明
hWnd	遍历所有系统类	利用每个系统类生成一个窗口
Msg	遍历 0 ~ 0x400	遍历由 Windows 保留的消息值
wParam	固定	选择一个固定值
	随机	随机生成一个值
lParam	固定	选择一个固定的内存地址
	随机	随机生成一个内存地址
	非法用户态地址	一个无法访问的用户态内存地址
	非法内核态地址	一个无法访问的内核态内存地址
	合法用户态地址	
	合法内核态地址	
输入数据	随机	随机生成数据
	固定字节填充	输入数据用固定字节填充
	固定字填充	输入数据用固定字填充
	固定双字填充	输入数据用固定双字填充

驱动程序之间、驱动和应用程序之间的都是通过给设备发送或者接收发给设备的请求来通信的，如果需要和应用程序之间通信，还需要为设备生成应用程序可以访问的符号链接，其格式为“\DosDevices\Name”，其中 Name 是由驱动程序任意设定的。应用程序可以调用 CreateFile 函数，指定符号链接，打开对应的设备对象，代码如图 4 所示。

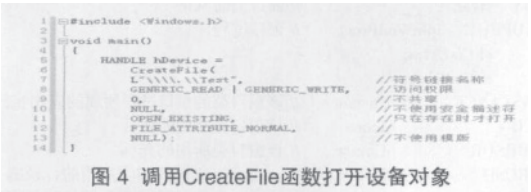
图 4 调用CreateFile函数打开设备对象

1.4.2 确定应用程序和驱动的通信途径

在打开驱动设备后，如果应用程序需要同驱动通信，或者调用驱动的派遣函数，需要调用 DeviceIoControl 函数，其定义如图 5 所示。

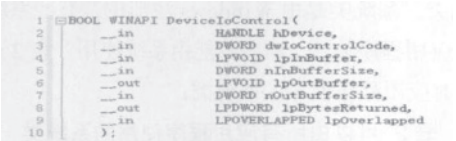
图 5 DeviceIoControl函数定义

hDevice，需要通信的设备句柄；

dwIoControlCode，IO 控制码；

lpInBuffer，输入缓冲区指针；

nInBufferSize，输入缓冲区字节数；

lpOutBuffer，输出缓冲区指针；

nOutBufferSize，输出缓冲区字节数；

lpBytesReturned，返回输出字节数；

lpOverlapped，异步调用时指向一个 OVERLAPPED 结构体。

第二个 IoControlCode 很重要，它的值一般由宏 CTL_CODE 构造而来，图 6 是这个宏的定义。

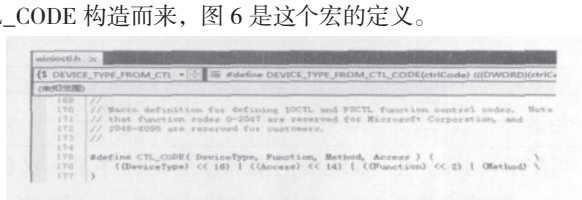
图 6 宏CTL_CODE 定义

可以看出 IoControlCode 以下四部分构成：

DeviceType，设备类型；

Access，对设备的访问权限；

Function，设备 IoControl 的功能号，0~-x7FF 为微软保留，0x800~0xFFF 由程序员指定；

Method，表示通信过程中的内存访问方式，主要有以下四种，如图 7 所示。

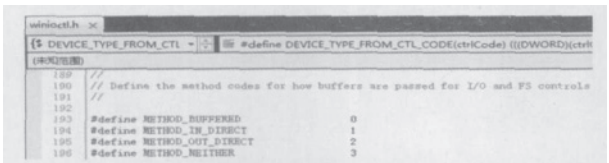

图 7 通信过程中的四种内存访问方式

METHOD_BUFFERED 为缓冲方式，应用程序指定的输入、输出缓冲区的内容都要经过 Windows 内核的缓冲。Windows 内核会对比 nInBufferSize 指定的输入缓冲区、nOutBufferSize 指定的输出缓冲区大小，取较大的一个在内核空间中分配一段内存（SystemBuffer），将位于用户空间中的输入缓冲区内容复制到位于内核空间中的 SystemBuffer，然后传递给驱动程序，驱动程序将 SystemBuffer 当作输入数据读取，在处理完成后，将处理结果写入到 SystemBuffer 中。从驱动程序中返回时，Windows 内核负责将 SystemBuffer 中的内容复制回应用程序指定的输出缓冲区中。这种方式避免了驱动程序在内核模式下直接读写用户态空间内存，整个过程比较安全。

METHOD_NEITHER 对输入、输出缓冲区都不缓冲，在驱动程序中直接读写用户态空间内存，这种方式并不安全，因此驱动程序在读取输入缓冲区数据时需要调用 ProbeForRead 检测用户态空间内存是否可读，在把处理结果写入输出缓冲区时要调用 ProbeForWrite 检测是否可写，在确保没有发生异常时再进行读写操作。

METHOD_IN_DIRECT 和 METHOD_OUT_DIRECT，系统对输入缓冲区进行缓冲，对输出缓冲区不缓冲，而在内核中进行锁定，在驱动程序完成 IO 请求之前无法访问，相比 METHOD_NEITHER 来说比较安全。

1.4.3 构造畸形数据，执行输入

在正常情况下，应用程序同驱动通信时，DeviceIoControl 函数参数都是自己程序指定的，不会是畸形数据；驱动程序在编写时，也主要考虑的是功能的实现，对畸形数据的处理、代码逻辑方面可能存在安全问题，因此，可以通过构造畸形的 DeviceIoControl 函数参数来 Fuzz 某一驱动程序的处理，监控其是否存在安全漏洞。主要方式是选取需要 Fuzz 的驱动程序，主动发送一系列的 IoControlCode，按照拟定的 Fuzzing 策略分别构造畸形的参数和数据，组合成一组参数，然后调用 DeviceIoControl 函数，将畸形数据发送给驱动程序处理，如果驱动程序中存在有安全漏洞，直接结果是造成系统蓝屏崩溃，然后利用 WinDBG 对蓝屏崩溃生成的 DMP 文件进行进一步分析。

根据 IoControlCode 中 Method 值的不同，分为以下两种情况：

1）Method 不为 METHOD_NEITHER

这种情况下输入缓冲区和输出缓冲区都有系统保护，修改输入缓冲区和输出缓冲区的地址没有意义，此时需要对输入数据、输入长度、输出长度进行畸形化处理。

2）Method 为 METHOD_NEITHER

这种情况下驱动程序可能会没有检查输入缓冲区和输出缓冲区而直接访问，因此需要对输入缓冲区地址、输出缓冲区地址、输入数据、输入长度、输出长度进行畸形化处理。最终的畸形化方案如表 5 所示。

表 5 针对IoControlCode的参数、数据畸形化方案

	方案	说明
输入地址	固定	选择一个固定的内存地址
	随机	随机生成一个内存地址
	非法用户态地址	一个无法访问的用户态内存地址
	非法内核态地址	一个无法访问的内核态内存地址
	合法用户态地址	
	合法内核态地址	
输入数据	随机	随机生成数据
	固定字节填充	输入数据用固定字节填充
	固定字填充	输入数据用固定字填充
	固定双字填充	输入数据用固定双字填充
输入长度	固定	选择固定长度
	随机	随机生成一个长度值
输出地址	固定	选择一个固定的内存地址
	随机	随机生成一个内存地址
	非法用户态地址	一个无法访问的用户态内存地址
	非法内核态地址	一个无法访问的内核态内存地址
	合法用户态地址	
	合法内核态地址	
输出长度	固定	选择固定长度
	随机	随机生成一个长度值

整个 Fuzzing 过程如图 8 所示。

图 8 针对IoControlCode的Fuzzing过程

1.5 针对主动防御软件处理Native API的漏洞挖掘

1.5.1 Native API概述

操作系统的主要功能是为应用程序的运行创建良好的环境，为了达到这个目的，操作系统内核提供一系列具备预定功能的多内核函数，通过一组称为系统调用接口（System Call Interface）呈现给用户。系统调用把应用程序的请求传给内核，调用相应的的内核函数完成所需的处理，然后将处理结果返回给应用程序，如果没有系统调用和内核函数，用户将不能往磁盘上读写文件、不能与底层驱动程序通信。像 UNIX 一类传统的操作系统的系统调用接口都很好地文档化了，定义很明确。在 Windows 中，这些系统调用接口叫 Native API，微软没有给出 Native API 的明确定义，也没有给出相关文档。

在 Windows NT 和 Windows 2000 中，应用程序可以通过调用 int 2Eh 从 Ring3 切换到 Ring0，而自从 Windows XP 以后，则采用 sysenter 和 sysexist 指令进行系统调用和返回。在进行系统调用时，应用程序需要将系统服务号放置在 EAX 寄存器

中，当切换到内核空间后，执行流程将进入 nt!KiSystemService 函数中，nt!KiSystemService 函数根据 EAX 寄存器中保存的系统服务号，在系统服务分发表中查找对应的系统调用。

系统服务号唯一确定一个系统调用，它有 32 位，从第 0 位到第 11 位是索引，表示将要调用哪个系统服务，第 12、13 位表示要调用哪个分发表中的函数。在 Windows 里面有两个分发表，分别是 System Service Dispatch Table（SSDT）、Shadow System Service Table（Shadow SSDT）。Windows 所有的系统调用都位于这两个表中。

1.5.2 确定Native API参数的输入路径

以下将以 Kernel32.dll 中的一个导出函数 WriteFile 为例，分析程序控制流程怎样进入到位于 ntoskrnl.exe 中的系统调用，以及在调用 WriteFile 函数时发生了些什么，这些将为确定设计 Fuzzing 数据和确定输入路径提供理论依据。

图 9 为 Windows SDK 中 WriteFile 函数的定义

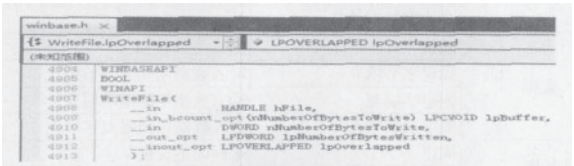

图9 WriteFile函数定义

在 WinDBG 中可以使用 uf 命令反汇编 WriteFile 函数，以下是反汇编部分指令：

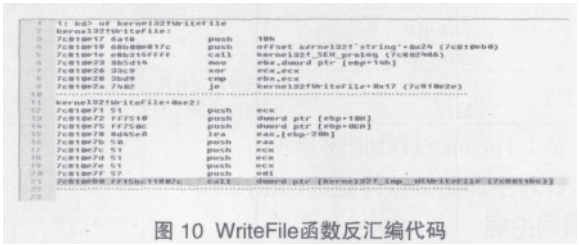

图 10 WriteFile函数反汇编代码

在图 10 中，最后一行可以看到，WriteFile 函数调用了 NtWriteFile 函数，而这个函数是从 ntdll.dll 中导入的，继续对 ntdll!ZwWriteFile 进行反汇编，如图 11 所示。

从反汇编结果可以看到，位于 ntdll 中的 ZwWriteFile 并不是真正的 Native API 的实现，它只是一个桩函数，将系统服务号 112h 放置到 EAX 寄存器中，然后调用了 ntdll!KiFastSystemCall 函数，接下来在 ntdll!KiFastSystemCall 函数中将用户态栈地址 ESP 放置到 EDX 中，然后调用 sysenter 指令进入内核。另外也注意到，在 ntdll!ZwWriteFile 最后面是一条指令 ret 24h，它表示从栈中弹出 24h 个字节，通常情况下，每个参数都视为 4 个字节，因此一共弹出了 9 个参数。

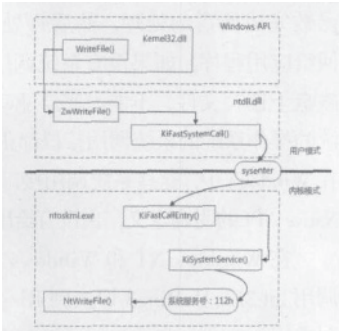

图11 ntdll!ZwWriteFile及后续函数的反汇编结果

在调用 sysenter 指令进入内核后，程序流程将进入位于 ntoskrnl.exe 中的 KiFastCallEntry 函数，KiFastCallEntry 函数进行一番处理后将进入到 KiSystemService 函数中，根据 EAX 中保存的系统服务号 112h，KiSystemService 函数将在 SSDT 表中查找到对应的系统调用 NtWriteFile；根据保存在 EDX 中的原用户态栈地址，取出参数传递给 NtWriteFile。整个流程如图 12 所示：

图 12 系统调用流程

1.5.3 主动防御软件对Native API的监控

在国内，360、微点等主动防御软件一般有三套防御体系：AD（Application Defend，应用程序防御体系）、RD（Registry Defend，注册表防御体系）、FD（File Defend，文件防御体系）。主动防御软件制定了一整套的行为规则，监视本地的运行程序、注册表读写操作、文件读写操作，恶意代码在运行时，总是会调用系统的一些其他的资源，例如修改注册表中的启动项，修改重要的系统文件等等，如果触发到主动防御软件已制定的规则，主动防御软件会通过对话框询问用户是否允许，用户根据自己的经验来判断该行为是否正确安全，是则放行并允许运行，否就不使之运行，如果用户选择否，那么恶意代码将无法更改重要文件，这更有效地防止了木马或者病毒的运行。

在上文中提到，WriteFile 是通过调用位于 ntdll.dll 中的桩函数 ntdll!ZwWriteFile 来实现系统调用，对恶意代码而言，桩函数的代码完全可以在自己的程序中实现，不需要借助于 ntdll.dll，所以为了有效地监视程序运行行为，主动防御软件就不能在用户态模块里面实施监控，只能在内核态中对 Windows Native API 实施监视。图 13 是腾讯公司出品的 QQ 电脑管家对 Windows Native API 的监视情况：

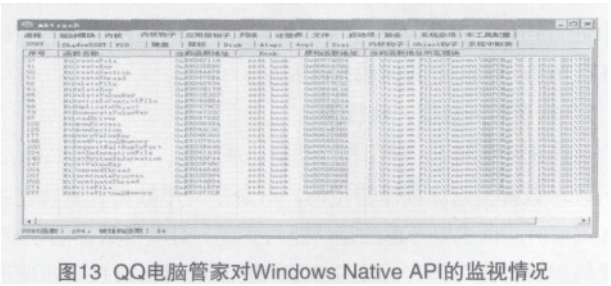

图13 QQ电脑管家对Windows Native API的监视情况

从图 13 可以看到，QQ 电脑管家监视了 24 个函数，这些函数囊括了进程、文件、注册表相关的 Native API。目前国内使用广泛的主动防御软件有 360 安全卫士、微点、金山卫士、QQ 电脑管家等等，各款主动防御软件按自己的要求或安全经验实现对这些 Native API 的监控，由于 Windows 的复杂、庞大，其中不可避免地会产生不当的处理参数，或者内部逻辑错误，部分 bug 可能会形成内核漏洞。

1.5.4 构造畸形Native API参数，执行输入

ntdll.dll 导出了所有 Native API 的桩函数，因此可以遍历 ntdll.dll 的导出表，查找所有以 Nt 开头的函数，获取桩函数的地址，通过指令识别，获取到 Native API 的服务号，然后根据桩函数的最后一条指令可以计算得到 Native API 的参数个数，虽然每个 Native API 的参数类型各不相同，但在实际的 Fuzzing 测试中，仍然可以将每一个参数视为一个指针，执行如下的 Fuzzing 方案，如表 6 所示。

表 6 针对Native API的参数、数据畸形化方案

	方案	说明
输入地址	固定	选择一个固定的内存地址
	随机	随机生成一个内存地址
	非法用户态地址	一个无法访问的用户态内存地址
	非法内核态地址	一个无法访问的内核态内存地址
	合法用户态地址	
	合法内核态地址	
输入数据	随机	随机生成数据
	固定字节填充	输入数据用固定字节填充
	固定字填充	输入数据用固定字填充
	固定双字填充	输入数据用固定双字填充
数据长度	固定	选择固定长度
	随机	随机生成一个长度值

将已经获取到地址的桩函数视为输入路径，在构造完成畸形数据后可以直接调用，执行 Fuzzing 测试。整个 Fuzzing 过程如图 14 所示。

2 漏洞挖掘实例

2.1 Win32k 错误的参数允许信息泄露漏洞（CVE-2011-1886）挖掘过程及分析

在 2011 年 3 月份针对 win32k.sys 的 Fuzzing 中，成功引发了win32k.sys 蓝屏崩溃，通过进一步分析，发现此漏洞影响效果不仅限于拒绝服务，还可以使用该漏洞访问任何内核模式内存位置中的数据，造成信息泄漏。此漏洞已经通报给微软安全响应小组，得到微软的认可。微软在 2011 年 7 月 12 日发布修补补丁，安全公告：MS11-054，知识库编号：KB2555917，CVE 编号：CVE-2011-1886，以下是具体的挖掘、分析过程。

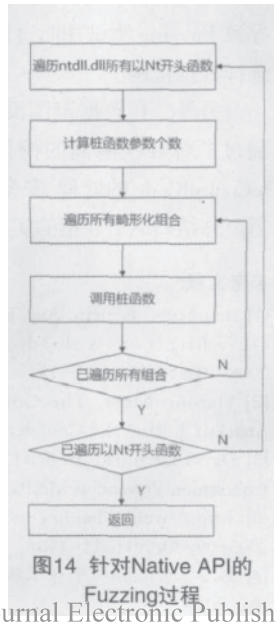

图14 针对Native API的Fuzzing过程

2.1.1 设置系统转储文件

系统转储文件描述的是整个系统，包括操作系统内核、工作在内核态里的驱动程序、各个用户进程的信息、寄存器信息、函数调用堆栈等等。为满足不同情况下的需要，Windows 定义了 3 种不同类型的系统转储文件：

1）完全内存转储。完全内存转储包含产生转储时物理内存中已有的所有数据，生成的转储文件默认为 C:\WINDOWS\MEMORY.DMP，其大小通常比物理内存的容量还要大；

2）核心内存转储。核心内存转储剔除了用户进程占用的内存，生成的转储文件体积比完全内存转储要小；

3）小内存转储（64 KB）。小内存转储大小默认为 64KB，保存在 C:\WINDOWS\MiniDump 文件夹下，以日期、序号的格式命名，可以保存多个小内存转储文件。

2.1.2 触发系统蓝屏崩溃

蓝屏是 Windows 中用于提示严重的系统级错误的一种方式，它会终止整个系统的运行，只有重新启动才能恢复到正常的桌面环境。利用 Fuzzing 技术挖掘 Windows 内核漏洞，目的是让内核代码中产生未处理的异常，或者违反某项 Windows 系统规则，而运行在内核模式下的代码是 Windows 系统信任的代码，对于这些代码中发生的错误，Windows 认为只有发生了严重的意外才会导致这些代码出错，所以都是通过蓝屏提示错误，让系统以可控的方式停止运行。

在 2011 年 3 月份针对 Windows XP SP3 操作系统 win32k.sys（文件版本为：5.1.2600.6064，xpsp_sp3_gdr.101231-1614）的 Fuzzing 中，成功引发了win32k.sys 一处蓝屏崩溃。

2.1.3 对系统转储文件的分析

微软没有公开系统转储文件格式，只能使用 WinDBG 调试器来分析系统转储文件。启动 WinDBG，点击菜单中的 File -> Open Crash Dump ... 即可打开 .DMP 文件，此时 WinDBG 会显示一些初步分析信息，如图 15 所示。

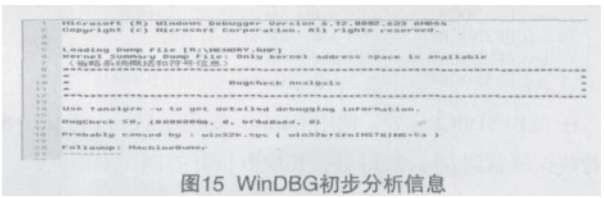
图15 WinDBG初步分析信息

从图 15 的第 6 行可以看到，转储文件类型为核心内存转储，只包含了内核模式下的内存信息。第 16 行显示的是停止码和参数，通过 WinDBG 的帮助手册可以了解到，0x50 表示 PAGE_FAULT_IN_NONPAGED_AREA（页面访问出错），它的第一个参数是 80000008，表示访问的内存地址，第二个参数 0，表示读取，第三个参数是 bf8d0a6d，表示在这一行代码出现的异常。第 18 行显示了 WinDBG 的初步判断结果，蓝屏可以是因为 win32k.sys 驱动引起的，有问题的代码距离 win32k!SfnINSTRING 函数 0x5a 字节，如图 16 所示。

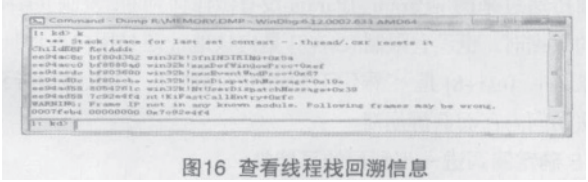
图16 查看线程栈回溯信息

从中可以看到崩溃时函数调用顺序，并显示出最后调用的函数为 win32k!SfnINSTRING，接下来应该结合静态分析技术和动态调试技术，分析该函数的代码、逻辑缺陷，判断是否

可以进一步利用。

2.1.4 手动分析相关函数代码

在系统类中，DDEMLEvent窗口类的窗口过程为win32k!xxxEventWndProc，它将调用到win32k!xxxDefWindowProc函数，根据消息标识的不同，win32k!xxxDefWindowProc将会调用到win32k!gapfnScSendMessage函数分发表中的某一函数。当消息为0x145或0x18D时，将会调用到win32k!gapfnScSendMessage函数分发表中的win32k!SfnINSTRING函数，以下是win32k!SfnINSTRING函数的部分代码分析：

```
1) .text:BF8D0A37          mov    esi, [ebp+lParam]
```

在0xBF8D0A37处，将[ebp+lParam]的值赋给esi，这个值从用户态空间传递过来的。

```
2) .text:BF8D0AAD          push   1              ; int
   .text:BF8D0AAF          lea    eax, [ebp+var_21C]
   .text:BF8D0AB5          push   eax            ; int
   .text:BF8D0AB6          push   [ebp+AllocationSize] ; AllocationSize
   .text:BF8D0ABC          push   [ebp+var_220]  ; int
   .text:BF8D0AC2          push   30h            ; int
   .text:BF8D0AC4          call   _AllocCallbackMessage@20
   .text:BF8D0AC9          mov    ebx, eax
```

在0xBF8D0AC4处，调用AllocCallbackMessage分配一段内存空间，用于存储稍后返回用户空间的数据。这片内存空间地址赋给ebx

```
3) .text:BF8D0B13          lea    eax, [ebx+2Ch]
   .text:BF8D0B16          push   eax
   .text:BF8D0B17          mov    eax, [esi]
   .text:BF8D0CB5          inc    eax
   .text:BF8D0CB6          inc    eax
   .text:BF8D0CB7          push   eax
   .text:BF8D0CB8          push   dword ptr [esi+8]
   .text:BF8D0CBB          push   ebx
   .text:BF8D0CBC          call   _CaptureCallbackData@16
```

在0xBF8D0CBC处，调用CaptureCallbackData将[esi+8]处的数据复制到[ebx+2Ch]处，长度由[esi]控制。

```
4) .text:BF8D0BC8          lea    eax, [ebp+var_24C]
   .text:BF8D0BCE          push   eax
   .text:BF8D0BCF          lea    eax, [ebp+var_240]
   .text:BF8D0BD5          push   eax
   .text:BF8D0BD6          push   dword ptr [ebx]
   .text:BF8D0BD8          push   ebx
   .text:BF8D0BD9          push   1Ah
   .text:BF8D0BDB          call   ds:__imp__KeUserModeCallback@20
```

在0xBF8D0BDB处，调用nt!KeUserModeCallback返回用户空间，EBX指向的内存地址数据将会传递到用户空间中，这些数据包括[esi+8]处的数据。

在整个win32k!SfnINSTRING函数执行过程中，对用户态空间传递过来的wParam、lParam没有做任何判断，复制[esi+8]处的数据时，也没有限制[esi+8]是内核地址还是用户态地址。如果esi、[esi+8]是一个不可访问的内存地址，那么将会导致无法访问内存而蓝屏崩溃。

2.1.5 确定漏洞进一步利用的可能性

对于普通用户而言，也可以利用系统类DDEMLEvent建立一个窗口，因此，在普通用户下也可以引发系统蓝屏崩溃，可以确定这是一个Windows拒绝服务漏洞。

针对上文的分析，如果从用户态里面传入了一个可以访问的内核地址，由于调用nt!KeUserModeCallback可以返回用户空间，所以这个漏洞可以进一步用来读取任意内核地址里面的敏感数据。对于普通用户来说，一个典型的权限提升应用场景是：SAM文件保存了Windows用户名和密码等信息，普通用户无法读取磁盘上的SAM文件，也无法读取注册表HKEY_LOCAL_MACHINE\SAM\SAM下的内容。但在内核中保存有SAM文件的一份拷贝，可以通过上述的漏洞获取到位于内核中的完整的SAM文件，造成管理员用户名和密码的泄漏。

2.2 其他漏洞挖掘成果

在针对第三方驱动程序处理IoControlCode、针对安全软件处理Native API函数的Fuzzing中，成功挖掘到多个安全软件中的多处漏洞。已挖掘到的漏洞均已秘密通报给相关公司处理，受到相关公司的致谢。出于公司的保密性，本文不提供漏洞所在函数和漏洞细节。

3 结束语

鉴于Windows内核漏洞对系统具有巨大的危害性，本文通过掌握的Windows内核漏洞调试、分析技术，提出了将Fuzzing技术应用到挖掘Windows内核漏洞上的方法，对此方法进行了系统化地研究和阐述，应用到实际的漏洞挖掘工作中，成功实现了挖掘出多个Windows内核漏洞、多个安全软件内核漏洞，验证了本文理论的正确性、实用性。但是由于个人自身能力有限，还有许多问题有待于进一步的研究，主要有：

1）本文选取了Windows win32k.sys对窗口消息的处理、第三方驱动程序对IoControlCode的处理、安全软件对Native API的处理三个做为内核Fuzzing目标，虽然有Windows内核漏洞没有包含在这三部分中，但仍然可以利用Fuzzing来挖掘，这就需要在下一步研究中进一步扩大Fuzzing目标；

2）在设计Fuzzing数据时，数据的选择范围仍然过大，导致Fuzzing测试用时过长，在下一步研究中需要进一步精确选择数据范围。

另外，有数据范围没有覆盖到，在进行手工分析时发现漏过了对某些漏洞的挖掘，这种情况在第三方驱动程序对IoControlCode的处理、安全软件对Native API的处理上出现过，需要总结归纳出数据的实用范围。（责编 杨晨）

参考文献：

[1] Ben Nagy. Generic Anti Exploitation Technology for Windows[EB/OL]. http://download.csdn.net/detail/TO_YGY/334441, 2008-01-14/2011-11-11.

[2] Haroon Meer. The Complete History of Memory Corruption Attacks[C]. BlackHat Confidence USA, 2010.

[3] David Litchfield. Buffer Underruns, DEP, ASLR and improving the Exploitation Prevention Mechanisms (XPMs) on the Windows platform[EB/ol]..http://wenku.baidu.com/view/eb293e4d2b160b4e767fcf72.html, 2005-09-30/2011-11-11.

[4] 彭赓. Windows平台下软件安全漏洞研究[D]. 成都：电子科技大学，2010.