LAB2 二进制炸弹

17307130178 宁晨然

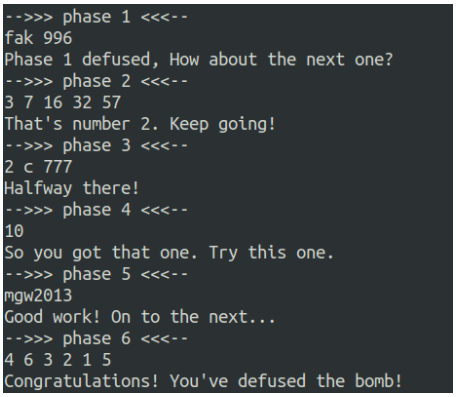
一、概述

本次实验被称为“拆炸弹”，实质其实是对汇编语言的理解。实验目的为认识汇编语言，从六个小程序的汇编代码反演出源程序过程，从而找出密码。实验准备阶段需要把书浏览一遍，记住基础的汇编代码指令；然后加强一遍运用 gdb 调试。

注：本次实验我使用的工具为 pwndbg，操作与 gdb 一样，但功能更丰富，可以查看更多 stack 内存的信息，界面也挺人性化。然后汇编语言格式使用的是 Att 格式。

本报告中左边为汇编语言，右边为对应的汇编语言的解释。

二、实验结果




```
Phase 1: string comparision
    ANS: fak 996
Phase 2: loops
    ANS: 3 7 16 32 57
Phase 3: conditionals/switches
    ANS:  0/1/2 c 777
          3/4/5 c 888
          6/7/8 c 999
Phase 4: recursive calls and the stack discipline
    ANS: 10
Phase 5: pointers/others
    ANS: mgw2013
Phase 6: linked lists/pointers/structs
    ANS: 4 6 3 2 1 5
```

三、实验过程

由于这次的核心在于在 terminal 中的 gdb 中进行调试和查看汇编代码，实际上在电脑屏幕上长时间看这么复杂的汇编代码会很降低工作效率，所以我决定先把整个可执行文件的汇编代码汇编到文件中，打印出来查看。命令为：objdump -d 17307130178_ > 17307130178

（一）、总体思路：

查看代码可以发现汇编代码肯定远远比源 c 代码多得多，所以篇幅很长、冗杂，有些不知所措从哪里开始看。Disassembly 代码基本可以分成四个结构：

①section:存放了<.init>等函数，猜测与初始化、头文件、库函数有关。

②main：主函数，主界面调出在此处，每个 phase 也从这里引入。

③phase1-6：存放 6 个谜题，分别对应六个主题的炸弹，主界面也说明清楚了。

④others：一些重要、经常使用的函数，如 string 比较函数、print 函数等。

从 PA1（上）的实验得出一个结论是：

hint1:程序执行从 main 函数开始。

所以前面<_init>/<.plt>几乎都没看，直接看 main 函数，加上实际执行可以发现这个 main 函数先进行了一堆压栈操作，再 call <initialize_bomb>，再输出初始界面。先不管这些介绍性的操作，应该研究意义不大。再看每个 phase 调用前的操作，可以看到有一些

<gets@plt>/<__isoc99_scanf@plt>的函数调用，可以根据名称猜测这就是对应的 c 语言的 gets()/scanf()函数从键盘读入数据，再在 phase 中操作比较。如果输入错误炸弹就会爆炸，正确就可以继续进行操作。下面分别讲述每个 phase。

（二）、Phase 详解

(1) Phase 1: string comparision　　ANS: fak 996

根据名称猜测这是字符串比较的函数。先查看 main 函数调用前。

```
0x00000000004008b9 <+220>:   lea    -0x220(%rbp),%rax
0x00000000004008c0 <+227>:   mov    %rax,%rdi
0x00000000004008c3 <+230>:   callq  0x4006b0 <gets@plt>
0x00000000004008c8 <+235>:   lea    -0x220(%rbp),%rax
0x00000000004008cf <+242>:   mov    %rax,%rdi
0x00000000004008d2 <+245>:   mov    $0x0,%eax
0x00000000004008d7 <+250>:   callq  0x401341 <phase_1>
```

①main:调用<gets@plt>函数从键盘读入字符串，并且保存在 rdi 中，意思是 phase_1 函数中刚开始的 rdi 就是我输入的字符串指针 char*

```
Dump of assembler code for function phase_1:
   0x0000000000401341 <+0>:     push   %rbp
   0x0000000000401342 <+1>:     mov    %rsp,%rbp
   0x0000000000401345 <+4>:     sub    $0x10,%rsp
   0x0000000000401349 <+8>:     mov    %rdi,-0x8(%rbp)
   0x000000000040134d <+12>:    mov    -0x8(%rbp),%rax
   0x0000000000401351 <+16>:    mov    $0x603200,%esi
   0x0000000000401356 <+21>:    mov    %rax,%rdi
   0x0000000000401359 <+24>:    callq  0x40138f <strings_equal>
   0x000000000040135e <+29>:    test   %eax,%eax
   0x0000000000401360 <+31>:    jne    0x401373 <phase_1+50>
   0x0000000000401362 <+33>:    mov    $0x0,%eax
   0x0000000000401367 <+38>:    callq  0x40137a <explode_bomb>
   0x000000000040136c <+43>:    mov    $0x0,%eax
   0x0000000000401371 <+48>:    jmp    0x401378 <phase_1+55>
   0x0000000000401373 <+50>:    mov    $0x1,%eax
   0x0000000000401378 <+55>:    leaveq
   0x0000000000401379 <+56>:    retq
End of assembler dump.
```

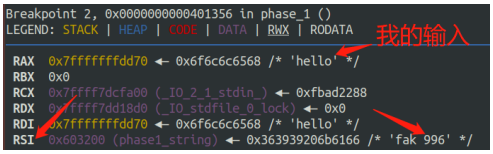
②phase_1:先有压栈操作，然后把 rdi 中字符串传给 rax，内存 0x603200 传给 esi，调用 strings_equal 进行字符串比较。

通过下面分析可知 strings_equal 正如名字所说就是字符串比较，如果字符串相等 eax 为 1，就可以跳过炸弹爆炸。

即答案就在 0x603200，并且成功过关返回值 eax 为 1。

```
Dump of assembler code for function strings_equal:
   0x000000000040138f <+0>:     push   %rbp
   0x0000000000401390 <+1>:     mov    %rsp,%rbp
   0x0000000000401393 <+4>:     sub    $0x10,%rsp
   0x0000000000401397 <+8>:     mov    %rdi,-0x8(%rbp)
   0x000000000040139b <+12>:    mov    %rsi,-0x10(%rbp)
   0x000000000040139f <+16>:    mov    -0x10(%rbp),%rdx
   0x00000000004013a3 <+20>:    mov    -0x8(%rbp),%rax
   0x00000000004013a7 <+24>:    mov    %rdx,%rsi
   0x00000000004013aa <+27>:    mov    %rax,%rdi
   0x00000000004013ad <+30>:    callq  0x400690 <strcmp@plt>
   0x00000000004013b2 <+35>:    test   %eax,%eax
   0x00000000004013b4 <+37>:    jne    0x4013bd <strings_equal+46>
   0x00000000004013b6 <+39>:    mov    $0x1,%eax
   0x00000000004013bb <+44>:    jmp    0x4013c2 <strings_equal+51>
   0x00000000004013bd <+46>:    mov    $0x0,%eax
   0x00000000004013c2 <+51>:    leaveq
   0x00000000004013c3 <+52>:    retq
End of assembler dump.
```

③strings_equal:实质又是调用了 strcmp 函数，比较 rsi 和 rdi 中存放的字符串。关键在于 **test %eax,%eax** 的意思是测试 eax 是否为 0。结合 phase_1 说明 strings_equal 函数，字符相等则返回 1，不相等返回 0。



此时有两种方法，1 是直接查看 0x603200，发现在运行前直接查看有问题。2 是在运行过程中查看内存或者 rsi，所以我把 breakpoint 设置在了 0x401356，在比较 rdi 和 rsi 之前，可以用 x/s $rsi 输出此时 rsi 存放的字符串，发现为'fak 996'即第一题密码。代码如下:

```
bool phase_1(char * str1){
    if(strcmp(str1,str2)) explode(),return false;
    return true;
}
```

(2) Phase 2: loops　　ANS: 3 7 16 32 57

根据题目猜测这次涉及循环，那肯定很多 jump 指令。猜测源代码是 for/while 结构。

```
0x00000000004008f4 <+279>:   movl   $0x0,-0x268(%rbp)
0x00000000004008fe <+289>:   jmp    0x40093a <main+349>
0x0000000000400900 <+291>:   mov    $0x0,%eax
0x0000000000400905 <+296>:   jmpq   0x400b45 <main+872>
0x000000000040090a <+301>:   lea    -0x260(%rbp),%rax
0x0000000000400911 <+308>:   mov    -0x268(%rbp),%edx
0x0000000000400917 <+314>:   movslq %edx,%rdx
0x000000000040091a <+317>:   shl    $0x2,%rdx
0x000000000040091e <+321>:   add    %rdx,%rax
0x0000000000400921 <+324>:   mov    %rax,%rsi
0x0000000000400924 <+327>:   mov    $0x4017ed,%edi
0x0000000000400929 <+332>:   mov    $0x0,%eax
0x000000000040092e <+337>:   callq  0x4006d0 <__isoc99_scanf@plt>
0x0000000000400933 <+342>:   addl   $0x1,-0x268(%rbp)
0x000000000040093a <+349>:   cmpl   $0x4,-0x268(%rbp)
0x0000000000400941 <+356>:   jle    0x40090a <main+301>
0x0000000000400943 <+358>:   lea    -0x260(%rbp),%rax
0x000000000040094a <+365>:   mov    %rax,%rdi
0x000000000040094d <+368>:   mov    $0x0,%eax
0x0000000000400952 <+373>:   callq  0x4012a0 <phase_2>
```

①main:先从 phase1 返回后，测试 rax?=1，然后进入一个 for 循环。从左图可以看出在 main 函数用了-0x268(%rbp)地址存放 i，初始化为 0，循环条件为 i<=4，并且通过移位、取地址等操作，打印内存 0x4006d0 为"%d"，即 scanf 格式输入方式。

可知这是一个循环输入数组的过程：

a.-0x260(%rbp)为数组 a 首地址

b.数组类型每个位置占用 4 个字节

c.每次调用 scanf("%d",a+i)输入，则为 int 数组

d.最终将首地址存放在 rdi 中，数组由五个 int 构成

推测源代码如下：

```
for(int i=0;i<=4;i++)
    scanf("%d",a+i);
```

```
Dump of assembler code for function phase_2:
   0x00000000004012a0 <+0>:     push   %rbp
   0x00000000004012a1 <+1>:     mov    %rsp,%rbp
   0x00000000004012a4 <+4>:     sub    $0x20,%rsp
   0x00000000004012a8 <+8>:     mov    %rdi,-0x18(%rbp)
   0x00000000004012ac <+12>:    movl   $0x1,-0x4(%rbp)
   0x00000000004012b3 <+19>:    mov    -0x18(%rbp),%rax
   0x00000000004012b7 <+23>:    mov    (%rax),%edx
   0x00000000004012b9 <+25>:    mov    0x201f39(%rip),%eax        # 0x6031f8 <phase2_ini>
   0x00000000004012bf <+31>:    cmp    %eax,%edx
   0x00000000004012c1 <+33>:    je     0x4012d4 <phase_2+52>
   0x00000000004012c3 <+35>:    mov    $0x0,%eax
   0x00000000004012c8 <+40>:    callq  0x40137a <explode_bomb>
   0x00000000004012cd <+45>:    mov    $0x0,%eax
   0x00000000004012d2 <+50>:    jmp    0x40133f <phase_2+159>
   0x00000000004012d4 <+52>:    movl   $0x1,-0x4(%rbp)
   0x00000000004012db <+59>:    jmp    0x401334 <phase_2+148>
   0x00000000004012dd <+61>:    mov    -0x4(%rbp),%eax
   0x00000000004012e0 <+64>:    cltq
   0x00000000004012e2 <+66>:    lea    0x0(,%rax,4),%rdx
   0x00000000004012ea <+74>:    mov    -0x18(%rbp),%rax
   0x00000000004012ee <+78>:    add    %rdx,%rax
   0x00000000004012f1 <+81>:    mov    (%rax),%eax
   0x00000000004012f3 <+83>:    mov    -0x4(%rbp),%edx
   0x00000000004012f6 <+86>:    movslq %edx,%rdx
   0x00000000004012f9 <+89>:    shl    $0x2,%rdx
   0x00000000004012fd <+93>:    lea    -0x4(%rdx),%rcx
   0x0000000000401301 <+97>:    mov    -0x18(%rbp),%rdx
   0x0000000000401305 <+101>:   add    %rcx,%rdx
   0x0000000000401308 <+104>:   mov    (%rdx),%ecx
   0x000000000040130a <+106>:   mov    -0x4(%rbp),%edx
   0x000000000040130d <+109>:   lea    0x1(%rdx),%esi
   0x0000000000401310 <+112>:   mov    -0x4(%rbp),%edx
   0x0000000000401313 <+115>:   add    $0x1,%edx
   0x0000000000401316 <+118>:   imul   %esi,%edx
   0x0000000000401319 <+121>:   add    %ecx,%edx
   0x000000000040131b <+123>:   cmp    %edx,%eax
   0x000000000040131d <+125>:   je     0x401330 <phase_2+144>
   0x000000000040131f <+127>:   mov    $0x0,%eax
   0x0000000000401324 <+132>:   callq  0x40137a <explode_bomb>
   0x0000000000401329 <+137>:   mov    $0x0,%eax
   0x000000000040132e <+142>:   jmp    0x40133f <phase_2+159>
   0x0000000000401330 <+144>:   addl   $0x1,-0x4(%rbp)
   0x0000000000401334 <+148>:   cmpl   $0x4,-0x4(%rbp)
   0x0000000000401338 <+152>:   jle    0x4012dd <phase_2+61>
```

②phase_2:

同理先进行栈操作。首地址 rdi 放入栈中-0x18(%rbp)，每次通过首地址生成每个数组元素的地址来取值。

a.打印 0x6031f8 可知内存存放的是数字 3，又因为与 a[0]进行比较必须相等可知

a[0] = 3

b.-0x4(%rbp)存放的是 i，接下来由 0x401334 的 jump 条件 i<=4 和 0x4012d4 初始化 i=1，可知剩下循环四次判断 a[1]-a[4]

c.中间一系列操作在生成 a[i]的地址，并取值放入 eax 中；并且又生成了 a[i-1]的地址，取值放入 ecx 中。

d.生成(i+1)存入 esi 和 edi 并且相乘得到平方存入 edx，与 a[i-1]相加存入 ecx

e.比较 eax 和 ecx，此时由跳转条件可知必须相等，可得每次递推关系：

a[i] = a[i-1] + (i+1)^2

可以推测主程序为：

```
bool phase_2(int * a){
    if(a[0]!=3) explode(),return false;
    for(int i=1;i<=4;i++)
       if(a[i]!=a[i-1]+(i+1)*(i+1)) explode(),return false;
    return true;
}
```

所以算出结果为五个数字：3 7 16 32 57

本题的关键在于找好所有 jmp 操作，在草稿纸上把他们先用划线连接起来，可以明显发现判断条件与循环退出条件。

(3) Phase 3: conditionals/switches ANS:

0/1/2 c 777

3/4/5 c 888

6/7/8 c 999

由名字可猜测这次使用了 c 语言中的 switch case 语句，应该有多种情况组成。

```
0x0000000000400973 <+406>:   lea    -0x274(%rbp),%rax
0x000000000040097a <+413>:   mov    %rax,%rsi
0x000000000040097d <+416>:   mov    $0x401821,%edi
0x0000000000400982 <+421>:   mov    $0x0,%eax
0x0000000000400987 <+426>:   callq  0x4006d0 <__isoc99_scanf@plt>
0x000000000040098c <+431>:   lea    -0x275(%rbp),%rax
0x0000000000400993 <+438>:   mov    %rax,%rsi
0x0000000000400996 <+441>:   mov    $0x401825,%edi
0x000000000040099b <+446>:   mov    $0x0,%eax
0x00000000004009a0 <+451>:   callq  0x4006d0 <__isoc99_scanf@plt>
0x00000000004009a5 <+456>:   lea    -0x270(%rbp),%rax
0x00000000004009ac <+463>:   mov    %rax,%rsi
0x00000000004009af <+466>:   mov    $0x4017ed,%edi
0x00000000004009b4 <+471>:   mov    $0x0,%eax
0x00000000004009b9 <+476>:   callq  0x4006d0 <__isoc99_scanf@plt>
0x00000000004009be <+481>:   mov    -0x270(%rbp),%edx
0x00000000004009c4 <+487>:   mov    -0x274(%rbp),%eax
0x00000000004009ca <+493>:   lea    -0x275(%rbp),%rcx
0x00000000004009d1 <+500>:   mov    %rcx,%rsi
0x00000000004009d4 <+503>:   mov    %eax,%edi
0x00000000004009d6 <+505>:   callq  0x4011f3 <phase_3>
```

①main:在调用 phase_3 之前先格式化输入了三个东西，可以 x 通过打印内存 0x401821/0x401825/0x4017ed 查看得知为格式输入”%d %c %d”,分别保存在栈 -0x274(%rbp)/-0x275(%rbp)/-0x270(%rbp)当中。

-0x270(%rbp)	int	%edx	d2
-0x274(%rbp)	char	%rsi	c
-0x275(%rbp)	int	%edi	d1

并且保存方式如上。

```
Dump of assembler code for function phase_3:
   0x00000000004011f3 <+0>:     push   %rbp
   0x00000000004011f4 <+1>:     mov    %rsp,%rbp
   0x00000000004011f7 <+4>:     sub    $0x10,%rsp
   0x00000000004011fb <+8>:     mov    %edi,-0x4(%rbp)
   0x00000000004011fe <+11>:    mov    %rsi,-0x10(%rbp)
   0x0000000000401202 <+15>:    mov    %edx,-0x8(%rbp)
   0x0000000000401205 <+18>:    cmpl   $0x8,-0x4(%rbp)
   0x0000000000401209 <+22>:    ja     0x40126c <phase_3+121>
   0x000000000040120b <+24>:    mov    -0x4(%rbp),%eax
   0x000000000040120e <+27>:    mov    0x402308(,%rax,8),%rax
   0x0000000000401216 <+35>:    jmpq   *%rax
```

②phase_3:

栈操作后，分别导入了之前的三个参数，并且 cmpl 判断了 d1，即满足

d1 <= 8

然后生成地址 0x402308(,%rax,8) 即取 0x402308+8*d1 地址中的值作为地址跳转，相当于 switch(d1)

```
0x0000000000401218 <+37>:    cmpl   $0x309,-0x8(%rbp)
0x000000000040121f <+44>:    je     0x401232 <phase_3+63>
0x0000000000401221 <+46>:    mov    $0x0,%eax
0x0000000000401226 <+51>:    callq  0x40137a <explode_bomb>
0x000000000040122b <+56>:    mov    $0x0,%eax
0x0000000000401230 <+61>:    jmp    0x40129e <phase_3+171>
0x0000000000401232 <+63>:    jmp    0x40127d <phase_3+138>
0x0000000000401234 <+65>:    cmpl   $0x378,-0x8(%rbp)
0x000000000040123b <+72>:    je     0x40124e <phase_3+91>
0x000000000040123d <+74>:    mov    $0x0,%eax
0x0000000000401242 <+79>:    callq  0x40137a <explode_bomb>
0x0000000000401247 <+84>:    mov    $0x0,%eax
0x000000000040124c <+89>:    jmp    0x40129e <phase_3+171>
0x000000000040124e <+91>:    jmp    0x40127d <phase_3+138>
0x0000000000401250 <+93>:    cmpl   $0x3e7,-0x8(%rbp)
0x0000000000401257 <+100>:   je     0x40126a <phase_3+119>
0x0000000000401259 <+102>:   mov    $0x0,%eax
0x000000000040125e <+107>:   callq  0x40137a <explode_bomb>
0x0000000000401263 <+112>:   mov    $0x0,%eax
0x0000000000401268 <+117>:   jmp    0x40129e <phase_3+171>
0x000000000040126a <+119>:   jmp    0x40127d <phase_3+138>
0x000000000040126c <+121>:   mov    $0x0,%eax
0x0000000000401271 <+126>:   callq  0x40137a <explode_bomb>
0x0000000000401276 <+131>:   mov    $0x0,%eax
0x000000000040127b <+136>:   jmp    0x40129e <phase_3+171>
0x000000000040127d <+138>:   mov    -0x10(%rbp),%rax
0x0000000000401281 <+142>:   movzbl (%rax),%eax
0x0000000000401284 <+145>:   cmp    $0x63,%al
0x0000000000401286 <+147>:   je     0x401299 <phase_3+166>
```

case 1

case 2

case 3

③case 语句:

可以查看内存地址 0x402308+8*d1 可得 d1=0/1/2case1;3/4/5case2;6/7/8case3;

```
pwndbg> x/100x 0x402308
0x402308:       0x18    0x12    0x40    0x00
0x402310:       0x18    0x12    0x40    0x00
0x402318:       0x18    0x12    0x40    0x00
0x402320:       0x34    0x12    0x40    0x00
0x402328:       0x34    0x12    0x40    0x00
0x402330:       0x34    0x12    0x40    0x00
0x402338:       0x50    0x12    0x40    0x00
0x402340:       0x50    0x12    0x40    0x00
0x402348:       0x50    0x12    0x40    0x00
```

如左图，所以分三种情况讨论即可。

case1:由 cmpl 语句可知 d2 必须为 0x309，并且合格后跳转至 char c 的判断，c 必须为 0x63（对于三个情况均相同）

case2:同理 d2 = 0x378

case3:同理 d2 = 0x3e7

需要转换为十进制和 ascii 码，得到答案

0/1/2 c 777

3/4/5 c 888

6/7/8 c 999

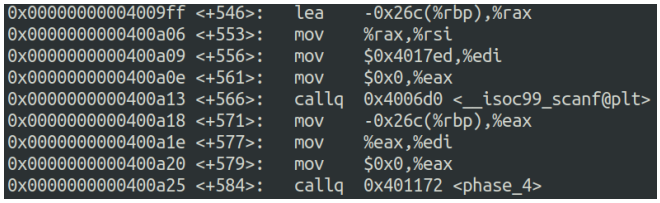
这道题主要是要看懂跳转条件和跳转过程，然后每个情况分别讨论就会很简单了。打成 c 程序可得:

```
phase_3(int d1,char c,int d2){
```

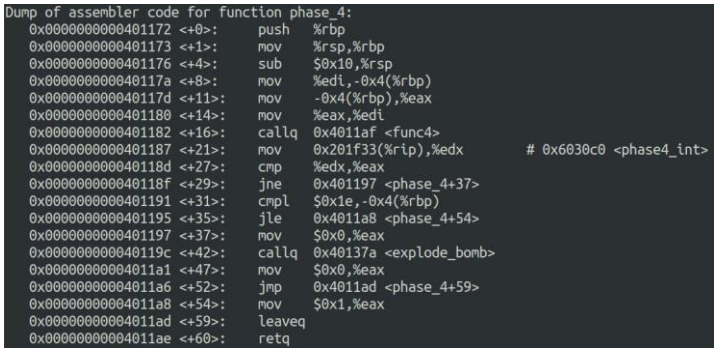
```
    if(d1>8) explode();
    switch(d1)
    {
        case 0:
        case 1:
        case 2: if(d2!=777) explode();break;
        case 3:
        case 4:
        case 5: if(d2!=888) explode();break;
        case 6:
        case 7:
        case 8: if(d2!=999) explode();break;
    }
    if(c!='?') explode();
      return 1;
}
```

(4) Phase 4:recursive calls and the stack discipline ANS:

根据题目猜测这是关于函数递归、函数调用的题目。

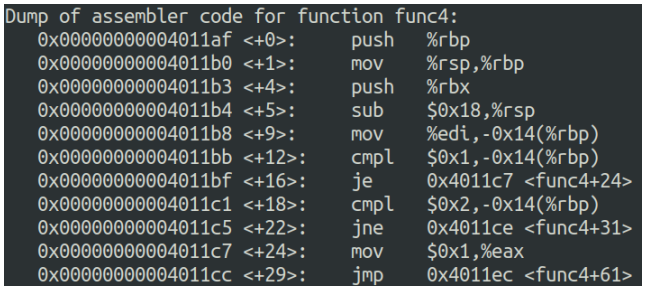


①main:这次很简单，同样取得内存 0x4017ed 为"%d"可知格式化输入一个 int 类型整数，并且保存在 edi 中传入参数，设为 int d



②phase_4:传入参数 d 后，将其保存入栈-0x4(%rbp)和 eax 当中，edi 也保存 d。

调用 func4 结束后，返回值为 eax，比较从 0x6030c0 取得的值与 eax，说明 func4 返回值必须为0x37，即 55。并且进行了 cmpl 0x1e 和 d，要求 d<=30



③func4:传入参数 d，栈操作后，先判断 d?=1，若相等则返回，eax=1。再判断 d?=2，若相等则返回 eax=1。若不相等，则开始嵌套调用 func4

```
0x00000000004011ce <+31>:    mov    -0x14(%rbp),%eax
0x00000000004011d1 <+34>:    sub    $0x1,%eax
0x00000000004011d4 <+37>:    mov    %eax,%edi
0x00000000004011d6 <+39>:    callq  0x4011af <func4>
0x00000000004011db <+44>:    mov    %eax,%ebx
0x00000000004011dd <+46>:    mov    -0x14(%rbp),%eax
0x00000000004011e0 <+49>:    sub    $0x2,%eax
0x00000000004011e3 <+52>:    mov    %eax,%edi
0x00000000004011e5 <+54>:    callq  0x4011af <func4>
0x00000000004011ea <+59>:    add    %ebx,%eax
0x00000000004011ec <+61>:    add    $0x18,%rsp
0x00000000004011f0 <+65>:    pop    %rbx
0x00000000004011f1 <+66>:    pop    %rbp
0x00000000004011f2 <+67>:    retq
```

④嵌套调用 func4:
a.调用 func4(d-1)保存返回值在 ebx
b.调用 func4(d-2)保存返回值在 eax
c.eax = eax + ebx 并且返回
相当于返回 func4(d-1)+func4(d-2)
可以看出 func4 的功能就是斐波那契数列的生成函数。
可以找到 func4(10) = 55
以上分析后，写成程序。

```
bool phase_4(int d){
    if(func4(d)!=55) explode(),return false;
    if(d>30) explode(),return false;
    return true;
}
int func4(int d){
    if(d==1||d==2) return 1;
    else return func4(d-1) + func4(d-2);
}
```

这道题关键在于看懂 func4 的函数调用，返回值保存、传递。根据它的步骤写出原来的代码更能方便理解递归的调用过程。

(5) Phase 5: pointers/others ANS: mgw2013

根据题目名称猜测跟指针相关。

```
0x00000000004004a4e <+625>:   callq  0x4006a0 <getchar@plt>
0x00000000004004a53 <+630>:   lea    -0x120(%rbp),%rax
0x00000000004004a5a <+637>:   mov    %rax,%rdi
0x00000000004004a5d <+640>:   callq  0x4006b0 <gets@plt>
0x00000000004004a62 <+645>:   lea    -0x120(%rbp),%rax
0x00000000004004a69 <+652>:   mov    $0x7,%esi
0x00000000004004a6e <+657>:   mov    %rax,%rdi
0x00000000004004a71 <+660>:   mov    $0x0,%eax
0x00000000004004a76 <+665>:   callq  0x40108c <phase_5>
```

①main:这次使用了 gets 函数，说明应该是获取了字符串并且保存在 rdi 中。开头的 getchar 应该是为了清理上题的回车。
注意：置 esi=7;edi=char* str1;

②phase_5:

```
Dump of assembler code for function phase_5:
   0x000000000040108c <+0>:     push   %rbp
   0x000000000040108d <+1>:     mov    %rsp,%rbp
   0x0000000000401090 <+4>:     push   %rbx
   0x0000000000401091 <+5>:     sub    $0x38,%rsp
   0x0000000000401095 <+9>:     mov    %rdi,-0x38(%rbp)
   0x0000000000401099 <+13>:    mov    %esi,-0x3c(%rbp)
   0x000000000040109c <+16>:    mov    %fs:0x28,%rax
   0x00000000004010a5 <+25>:    mov    %rax,-0x18(%rbp)
   0x00000000004010a9 <+29>:    xor    %eax,%eax
   0x00000000004010ab <+31>:    movl   $0x0,-0x24(%rbp)
   0x00000000004010b2 <+38>:    jmp    0x401124 <phase_5+152>
```

```
0x0000000000401124 <+152>:   mov    -0x24(%rbp),%eax
0x0000000000401127 <+155>:   cmp    -0x3c(%rbp),%eax
0x000000000040112a <+158>:   jl     0x4010b4 <phase_5+40>
0x000000000040112c <+160>:   lea    -0x20(%rbp),%rax
0x0000000000401130 <+164>:   mov    $0x6030e0,%esi
0x0000000000401135 <+169>:   mov    %rax,%rdi
0x0000000000401138 <+172>:   callq  0x40138f <strings_equal>
0x000000000040113d <+177>:   test   %eax,%eax
0x000000000040113f <+179>:   jne    0x401152 <phase_5+198>
0x0000000000401141 <+181>:   mov    $0x0,%eax
0x0000000000401146 <+186>:   callq  0x40137a <explode_bomb>
0x000000000040114b <+191>:   mov    $0x0,%eax
0x0000000000401150 <+196>:   jmp    0x401157 <phase_5+203>
0x0000000000401152 <+198>:   mov    $0x1,%eax
0x0000000000401157 <+203>:   mov    -0x18(%rbp),%rbx
0x000000000040115b <+207>:   xor    %fs:0x28,%rbx
0x0000000000401164 <+216>:   je     0x40116b <phase_5+223>
0x0000000000401166 <+218>:   callq  0x400670 <__stack_chk_fail@plt>
0x000000000040116b <+223>:   add    $0x38,%rsp
0x000000000040116f <+227>:   pop    %rbx
0x0000000000401170 <+228>:   pop    %rbp
0x0000000000401171 <+229>:   retq
```

a.进入比较之前，先将str放进-0x38(%rbp)，-0x3c(%rbp)保存 esi=7；跳转 b。
由此处和后面可知：

-0x38(%rbp)	原始字符串 char* str1
-0x24(%rbp)	循环次数 i 初始为 0
-0x20(%rbp)	变换后字符串 char* str2
-0x18(%rbp)	fs:0x28 文件分隔符 ^\

b.0x401124 部分，可知此处有一个循环条件 i<7。循环体部分在 c
经过 c 循环后，7 个字符保存在新位置，前面三位字符进行了变换。然后调用 strings_equal 语句说明str2需要等于内存 0x6030e0 的字符串 ics2013

```
Breakpoint *0x40108c
pwndbg> x/s 0x6030e0
0x6030e0 <phase5_string>:       "ics2013"
```

```
0x00000000004010b4 <+40>:    cmpl   $0x2,-0x24(%rbp)
0x00000000004010b8 <+44>:    jg     0x401107 <phase_5+123>
0x00000000004010ba <+46>:    mov    -0x24(%rbp),%eax
0x00000000004010bd <+49>:    movslq %eax,%rdx
0x00000000004010c0 <+52>:    mov    -0x38(%rbp),%rax
0x00000000004010c4 <+56>:    add    %rdx,%rax
0x00000000004010c7 <+59>:    movzbl (%rax),%eax
0x00000000004010ca <+62>:    and    $0x7b,%eax
0x00000000004010cd <+65>:    mov    %eax,%edx
0x00000000004010cf <+67>:    mov    -0x24(%rbp),%eax
0x00000000004010d2 <+70>:    cltq
0x00000000004010d4 <+72>:    mov    %dl,-0x20(%rbp,%rax,1)
0x00000000004010d8 <+76>:    mov    -0x24(%rbp),%eax
0x00000000004010db <+79>:    cltq
0x00000000004010dd <+81>:    movzbl -0x20(%rbp,%rax,1),%edx
0x00000000004010e2 <+86>:    mov    -0x24(%rbp),%eax
0x00000000004010e5 <+89>:    movslq %eax,%rcx
0x00000000004010e8 <+92>:    mov    -0x38(%rbp),%rax
0x00000000004010ec <+96>:    add    %rcx,%rax
0x00000000004010ef <+99>:    movzbl (%rax),%eax
0x00000000004010f2 <+102>:   cmp    %al,%dl
0x00000000004010f4 <+104>:   jne    0x401120 <phase_5+148>
```

```
0x0000000000401107 <+123>:   mov    -0x24(%rbp),%eax
0x000000000040110a <+126>:   movslq %eax,%rdx
0x000000000040110d <+129>:   mov    -0x38(%rbp),%rax
0x0000000000401111 <+133>:   add    %rdx,%rax
0x0000000000401114 <+136>:   movzbl (%rax),%edx
0x0000000000401117 <+139>:   mov    -0x24(%rbp),%eax
0x000000000040111a <+142>:   cltq
0x000000000040111c <+144>:   mov    %dl,-0x20(%rbp,%rax,1)
0x0000000000401120 <+148>:   addl   $0x1,-0x24(%rbp)
```

c.首先比较 i 与 2，可知这个循环由两个部分即 i<2 与 i>=2 时。

i<2 时，先生成 str1[i]取值，然后与 0x7b 按位与操作，保存在变换后的字符串地址中-0x20(%rbp,%rax,1)即

str2[i]=str1[i]&0x7b

然后又一次用-0x38(%rbp)中生成 str1[i]存放在 rcx 当中，与刚才获得的 str2[i]比较，必须保证 **str2[i]!=str1[i]**，否则爆炸。

i>=2 时，生成 str1[i]取值，直接保存在 str2[i]中。即 str1[i] = str2[i]

*需要解释一下关键句子:

mov %dl,-0x20(%rbp,%rax,1)

指: %rbp+%rax*1-0x20 的地址处存放%dl，即一个字节，就是 str1 字符串中的其中一个字符。

③字符串变换:

根据分析可知，前三个字符与 0x7b 取与运算后等于 ics 但是字符本身不能是 ics。则根据 ascii 表格可知，符合这个条件的只有 mgw:

0x7b	0111	1011
i	0110	1001
原先 m	0110	1101
c	0110	0011
原先 g	0110	0111
s	0111	0011
原先 w	0111	0111

后面四个字符原封不动落下来即可，所以答案为 mgw2013。

④对于 fs:0x28 的解释:

在-0x18(%rbp)放置了一个分隔符，在字符串变换输出到-0x20(%rbp)后，进行了-0x18(%rbp)与 fs:0x28 的比较，说明这个字符串的长度只能是 7，如果超出 7 的长度，会把 fs:0x28 覆盖掉，则会导致栈（溢出）错误，字符串没有结束符（其实 fs:0x28 就是它的结束）。这里限制了 str1 的长度在 7 以内。

表达为代码即:

```
bool phase_5(char* str1){
    char* str2 = (char*) malloc(sizeof(char)*8);
    char* str3 = “ics2013”;
    str2[7] = ‘\0’;
    for(int i=0;i<7;i++)
        if(i<2) {
            str2[i] = str1[i] & 0x7b;
            if(str2[i]==str1[i]) explode(),return false;
        }
        else str2[i] = str1[i];
```

```
    if(strcmp(str2,str3)) explode(),return false;
    if(strlen(str1)>7) explode(),return false;
    return true;
}
```

(6) Phase 6: linked lists/pointers/structs ANS: 4 6 3 2 1 5

本题花费时间是最多的，接近 3 小时。根据题目可以猜测跟链表、指针、结构体相关。

```
0x0000000000400a9f <+706>:   movl   $0x0,-0x264(%rbp)
0x0000000000400aa9 <+716>:   jmp    0x400ae5 <main+776>
0x0000000000400aab <+718>:   mov    $0x0,%eax
0x0000000000400ab0 <+723>:   jmpq   0x400b45 <main+872>
0x0000000000400ab5 <+728>:   lea    -0x240(%rbp),%rax
0x0000000000400abc <+735>:   mov    -0x264(%rbp),%edx
0x0000000000400ac2 <+741>:   movslq %edx,%rdx
0x0000000000400ac5 <+744>:   shl    $0x2,%rdx
0x0000000000400ac9 <+748>:   add    %rdx,%rax
0x0000000000400acc <+751>:   mov    %rax,%rsi
0x0000000000400acf <+754>:   mov    $0x4017ed,%edi
0x0000000000400ad4 <+759>:   mov    $0x0,%eax
0x0000000000400ad9 <+764>:   callq  0x4006d0 <__isoc99_scanf@plt>
0x0000000000400ade <+769>:   addl   $0x1,-0x264(%rbp)
0x0000000000400ae5 <+776>:   cmpl   $0x5,-0x264(%rbp)
0x0000000000400aec <+783>:   jle    0x400ab5 <main+728>
0x0000000000400aee <+785>:   lea    -0x240(%rbp),%rax
0x0000000000400af5 <+792>:   mov    $0x6,%esi
0x0000000000400afa <+797>:   mov    %rax,%rdi
0x0000000000400afd <+800>:   callq  0x400fa3 <is_difference>
0x0000000000400b02 <+805>:   test   %eax,%eax
0x0000000000400b04 <+807>:   jne    0x400b17 <main+826>
0x0000000000400b06 <+809>:   mov    $0x0,%eax
0x0000000000400b0b <+814>:   callq  0x40137a <explode_bomb>
0x0000000000400b10 <+819>:   mov    $0x0,%eax
0x0000000000400b15 <+824>:   jmp    0x400b45 <main+872>
0x0000000000400b17 <+826>:   lea    -0x240(%rbp),%rax
0x0000000000400b1e <+833>:   mov    %rax,%rdi
0x0000000000400b21 <+836>:   mov    $0x0,%eax
0x0000000000400b26 <+841>:   callq  0x400df9 <phase_6>
```

```
Dump of assembler code for function is_difference:
   0x0000000000400fa3 <+0>:     push   %rbp
   0x0000000000400fa4 <+1>:     mov    %rsp,%rbp
   0x0000000000400fa7 <+4>:     mov    %rdi,-0x68(%rbp)
   0x0000000000400fab <+8>:     mov    %esi,-0x6c(%rbp)
   0x0000000000400fae <+11>:    lea    -0x50(%rbp),%rsi
   0x0000000000400fb2 <+15>:    mov    $0x0,%eax
   0x0000000000400fb7 <+20>:    mov    $0xa,%edx
   0x0000000000400fbc <+25>:    mov    %rsi,%rdi
   0x0000000000400fbf <+28>:    mov    %rdx,%rcx
   0x0000000000400fc2 <+31>:    rep stos %rax,%es:(%rdi)
   0x0000000000400fc5 <+34>:    movl   $0x0,-0x54(%rbp)
   0x0000000000400fcc <+41>:    jmpq   0x401079 <is_difference+214>
   0x0000000000400fd1 <+46>:    mov    -0x54(%rbp),%eax
   0x0000000000400fd4 <+49>:    cltq
   0x0000000000400fd6 <+51>:    lea    0x0(,%rax,4),%rdx
   0x0000000000400fde <+59>:    mov    -0x68(%rbp),%rax
   0x0000000000400fe2 <+63>:    add    %rdx,%rax
   0x0000000000400fe5 <+66>:    mov    (%rax),%eax
   0x0000000000400fe7 <+68>:    cmp    $0x6,%eax
   0x0000000000400fea <+71>:    jg     0x401006 <is_difference+99>
   0x0000000000400fec <+73>:    mov    -0x54(%rbp),%eax
   0x0000000000400fef <+76>:    cltq
   0x0000000000400ff1 <+78>:    lea    0x0(,%rax,4),%rdx
   0x0000000000400ff9 <+86>:    mov    -0x68(%rbp),%rax
   0x0000000000400ffd <+90>:    add    %rdx,%rax
   0x0000000000401000 <+93>:    mov    (%rax),%eax
   0x0000000000401002 <+95>:    test   %eax,%eax
   0x0000000000401004 <+97>:    jg     0x40100d <is_difference+106>
   0x0000000000401006 <+99>:    mov    $0x0,%eax
   0x000000000040100b <+104>:   jmp    0x40108a <is_difference+231>
```

```
0x0000000000401025 <+130>:   mov    -0x50(%rbp,%rax,4),%eax
0x0000000000401029 <+134>:   cmp    $0x1,%eax
0x000000000040102c <+137>:   jne    0x401035 <is_difference+146>
```

```
0x000000000040106d <+202>:   movl   $0x1,-0x50(%rbp,%rax,4)
0x0000000000401075 <+210>:   addl   $0x1,-0x54(%rbp)
0x0000000000401079 <+214>:   mov    -0x54(%rbp),%eax
0x000000000040107c <+217>:   cmp    -0x6c(%rbp),%eax
```

①main:可知-0x264(%rbp)充当了 i 的作用，初始化为 0，循环次数为 6。-0x240(%rbp)为数组的首地址，一共键入了六个数字，由 0x4017ed 可知内存格式化输入为"%d"。

然后调用了<is_difference>，说明这六个数字需要满足某些条件，最后再调用 phase_6。传入参数为下：

esi	6 循环次数
rdi	a 数组首地址
rax	a 数组首地址

②is_difference:该函数是测试每个元素是否各不相同。

a.根据左图矩形和后图可知

-0x50……	visited 数组访问记录
-0x54	i=a[5]初始化为 0
-0x68……	a 数组的首地址
-0x6c	6 数组大小

该程序使用了 a[5]作为计数器 i，并初始化为 0。经过一系列取值操作构造 a[i]，两个矩形分别正确条件为：

a[i]<=6

a[i]!=0

又由后面推测 a[i]取值范围为 1~6，且六个元素各不相同。

b.此处取值 a[i]方式过于重复，不多解释了。用-0x50(%rbp,%rax,4)生成了 visited[a[i]]，并且要求：

visited[a[i]]!=1

说明每个元素一开始没有访问过。访问之后会将对应的位置置 1.

c.左图将visited[a[i]]=1，然后i++，并且要求 i=a[5]<6，即循环条件。

根据上述解释可写出程序：

```
bool is_difference(int a[],int n){
    int vis[6] = {0};
    for(a[5]=0;a[5]<n;a[5]++){
```

```
        int i = a[5];
        if(a[i]>6||a[i]==0) return false;
        if(vis[a[i]]) return false;
        vis[a[i]] = 1;
    }
    return true;
}
```

根据 is_differece 程序可知，a[5]=5 时，i=5，要求 vis[5]=0，即原数组中 a[5]=5。

接下来传参进入 phase_6，传参方式为：

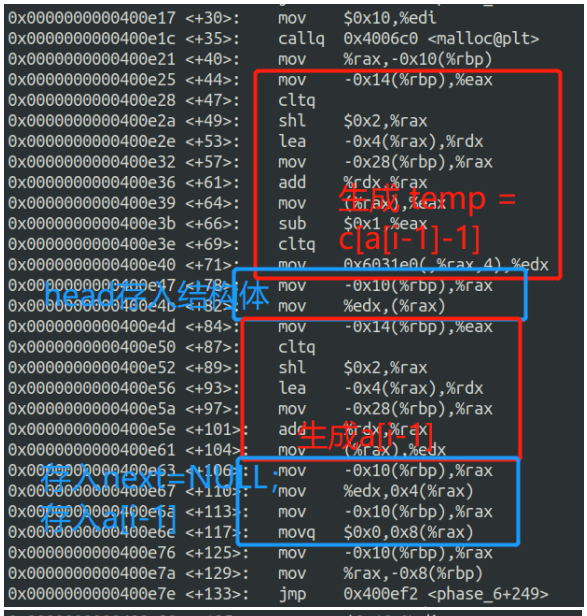
rdi	a
eax	0

③phase_6：

栈总体结构为：

-8	head
-10	head
-14	i=a[5]初始化为 1
-28……	a 数组

由于 cmpl $0x1,-0x14(%rbp)这句话，将循环中分成两种情况：



a.if(i==1)时

如左图所示，为结构体开了 4 个字节空间，又由后面(%rax)/0x4(%rax)/0x8(%rax)可推测结构体为：

```
struct node
{
    int temp;
    int a_i;
    node* next;
}
```

所以左图进行了第一个 head 表头的构造，并且把 head 存入-0x8(%rbp)

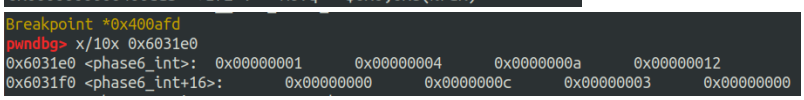
b.if(i!=1)

首先开空间，同样是构造新的结构体。

左图为链表的链接操作，构造好新的表元后，将其链接在尾表元的后方，顺序连接。

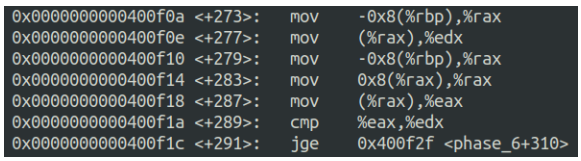
之后又进行生成 c[a[i-1]-1]，其中 c 为内存 0x6031e0 处的数组，打印如下。

其余操作如同 a。



c.比较

首先取出 p=head，p=head->next；



根据要求，有：

```
p->temp > p->next->temp
```

并且循环整个链表，至此整个 phase_6 很清晰了。写出程序：

```
bool phase_6(int a[],int n){
  for(a[5]=1;a[5]<=n;a[5]++){
      int i = a[5];
        if(i==1){
          node* head = (node*)malloc(sizeof(node));
          head->temp = c[a[i-1]-1];
          head->a_i = a[i-1];
          head->next = NULL;
        }
        else{
          node* temp = (node*)malloc(sizeof(node));
          temp->temp = c[a[i-1]-1];
          temp->a_i = a[i-1];
          temp->next = head;
          head = temp;
        }
  for(node* p1=head,p2=head->next;p2!=NULL;p1=p2,p2=p2->NULL)
      if(p1->temp<=p2->temp) explode(),return false;
  return true;
}
```

所以根据这个条件，结合内存中 c 数组和 c[a[i-1]-1]，可知 a 数组必须为 4 6 3 2 1 5

1	4	a	10	0	C
⑤	④	③	①	⑥	②

其中 a[5]=5 符合预期。

这个 phase 的关键就是找到链表操作的过程。

四、实验总结

本次实验从简单到难，从一开始的只是调用字符串匹配函数到后面复杂链表调用，一步一步的体会了整个阅读汇编/翻译成 c 语言的过程。通过这次实验，真正了解了每个指令的用途、方法，和使用情景。一开始真的特别蒙，因为很多指令都看不懂，只能一个一个的查看书籍、查看百度，最后才略微能够弄懂每个指令的意思。后来发现只是看懂指令很多东西都看不懂，很大程度上用名称猜测每个函数的作用、可能需要用的方法，还有结合标题、主题看出这个函数在怎么运作。特别是第六个 phase 让我花费了一个晚上和一个上午的时间，才搞清楚运作。本身链表的操作就格外复杂，放进汇编中更需要小心谨慎。

我还学会了使用分块的思想，有时候一块汇编讲述的是同一个功能，就能一次性把这个大块类搞定，然后再看跳转其他地方的情况。看汇编时，先把跳转找好、勾画出来，分情况讨论的地方也根据 c 语言的思想一个一个看，最后就基本能够理解代码框架了。

这是一场自学的盛宴吧。期待下次与你相见。

[reference]

课本/百度查看很多指令的解释

CSDN https://blog.csdn.net/miracle_ma/article/details/79968992

【just for reading】

gdb 查看内存信息
https://blog.csdn.net/yasi_xi/article/details/7322278
令人迷惑的 AT&T 的 jmp:直接跳转和间接跳转
https://blog.csdn.net/pi408637535/article/details/38458159
GDB 查看指定内存地址的内容——指令 x
https://blog.csdn.net/haifeng_gu/article/details/73928545
汇编语言程序设计
https://blog.csdn.net/ww506772362/article/details/75530723?locationNum=7&fps=1