

# LAB2 实验分析

使用gdb bomb命令可以实时调试程序。结合break function、break \*地址、disas、x/s \$地址命令实时查看程序内的内容，同时用info registers和info frame查看寄存器信息和栈帧信息，可以拆除炸弹。

## Phase 1

在Phase\_1打下断点，使用info register可以得到寄存器信息：

```
(gdb) info register
rax                0x603780                6305664
rbx                0x0                    0
rcx                0x3                    3
rdx                0x1                    1
rsi                0x603780                6305664
rdi                0x603780                6305664
rbp                0x402210                0x402210 <__libc_csu_init>
rsp                0x7fffffffde28         0x7fffffffde28
r8                 0x604674                6309492
r9                 0x7ffff7fba540          140737353852224
r10                0x3                    3
r11                0x7ffff7e015c0          140737352046016
r12                0x400c90                4197520
r13                0x7fffffffdf10          140737488346896
r14                0x0                    0
r15                0x0                    0
rip                0x400ee0                0x400ee0 <phase_1>
eflags             0x206                [ PF IF ]
cs                 0x33                    51
ss                 0x2b                    43
ds                 0x0                    0
es                 0x0                    0
fs                 0x0                    0
gs                 0x0                    0
```

得到汇编代码如下：

```
Dump of assembler code for function phase_1:
=> 0x0000000000400ee0 <+0>:      sub    $0x8,%rsp
      0x0000000000400ee4 <+4>:      mov    $0x402400,%esi
      0x0000000000400ee9 <+9>:      callq  0x401338 <strings_not_equal>
      0x0000000000400eee <+14>:     test   %eax,%eax
      0x0000000000400ef0 <+16>:     je     0x400ef7 <phase_1+23>
      0x0000000000400ef2 <+18>:     callq  0x40143a <explode_bomb>
      0x0000000000400ef7 <+23>:     add    $0x8,%rsp
      0x0000000000400efb <+27>:     retq
```

发现程序调用了`strings_not_equal`函数，该函数应该是用于比较两个字符串是否相等的。可以发现`%eax`是输入的字符串数据，调用程序会将如果返回值`%eax`为0的话就结束函数`phase_1`。其中所以使用命令`x/s 0x402400`获取存储的字符串，得到结果：

```
Border relations with Canada have never been better.
```

第一个炸弹输入这一行就可以通过了。

## Phase 2

同样获取这一阶段的汇编代码：

```
(gdb) disas
Dump of assembler code for function phase_2:
=> 0x0000000000400efc <+0>:      push    %rbp
    0x0000000000400efd <+1>:      push    %rbx
    0x0000000000400efe <+2>:      sub     $0x28,%rsp
    0x0000000000400f02 <+6>:      mov     %rsp,%rsi
    0x0000000000400f05 <+9>:      callq  0x40145c <read_six_numbers>
    0x0000000000400f0a <+14>:     cmpl    $0x1,(%rsp)
    0x0000000000400f0e <+18>:     je      0x400f30 <phase_2+52>
    0x0000000000400f10 <+20>:     callq  0x40143a <explode_bomb>
    0x0000000000400f15 <+25>:     jmp     0x400f30 <phase_2+52>
    0x0000000000400f17 <+27>:     mov     -0x4(%rbx),%eax
    0x0000000000400f1a <+30>:     add     %eax,%eax
    0x0000000000400f1c <+32>:     cmp     %eax,(%rbx)
    0x0000000000400f1e <+34>:     je      0x400f25 <phase_2+41>
    0x0000000000400f20 <+36>:     callq  0x40143a <explode_bomb>
    0x0000000000400f25 <+41>:     add     $0x4,%rbx
    0x0000000000400f29 <+45>:     cmp     %rbp,%rbx
    0x0000000000400f2c <+48>:     jne     0x400f17 <phase_2+27>
    0x0000000000400f2e <+50>:     jmp     0x400f3c <phase_2+64>
    0x0000000000400f30 <+52>:     lea     0x4(%rsp),%rbx
    0x0000000000400f35 <+57>:     lea     0x18(%rsp),%rbp
    0x0000000000400f3a <+62>:     jmp     0x400f17 <phase_2+27>
    0x0000000000400f3c <+64>:     add     $0x28,%rsp
    0x0000000000400f40 <+68>:     pop     %rbx
    0x0000000000400f41 <+69>:     pop     %rbp
    0x0000000000400f42 <+70>:     retq
End of assembler dump.
```

有一个`read_six_numbers`的函数，猜测会从输入中读取六个数字。所以可以随便输入6个数字测试以一下。查看`read_six_numbers`的汇编代码：

```
Dump of assembler code for function read_six_numbers:
0x000000000040145c <+0>:      sub     $0x18,%rsp
0x0000000000401460 <+4>:      mov     %rsi,%rdx
0x0000000000401463 <+7>:      lea     0x4(%rsi),%rcx
0x0000000000401467 <+11>:     lea     0x14(%rsi),%rax
0x000000000040146b <+15>:     mov     %rax,0x8(%rsp)
```

```

0x0000000000401470 <+20>: lea    0x10(%rsi),%rax
0x0000000000401474 <+24>: mov    %rax, (%rsp)
0x0000000000401478 <+28>: lea    0xc(%rsi),%r9
0x000000000040147c <+32>: lea    0x8(%rsi),%r8
0x0000000000401480 <+36>: mov    $0x4025c3,%esi
0x0000000000401485 <+41>: mov    $0x0,%eax
0x000000000040148a <+46>: callq  0x400bf0 <__isoc99_sscanf@plt>
0x000000000040148f <+51>: cmp    $0x5,%eax
0x0000000000401492 <+54>: jg     0x401499 <read_six_numbers+61>
0x0000000000401494 <+56>: callq  0x40143a <explode_bomb>
0x0000000000401499 <+61>: add    $0x18,%rsp
0x000000000040149d <+65>: retq
End of assembler dump.

```

通过观察寄存器的信息，可以发现输入的数据存储在了 %rsp 下的各个位置，分别为 %rsp + 1, %rsp + 2..... 等等（可以通过 `(gdb) print /d *0x7fffffffddfd0 + 4 $5 = 5` 来检查，更多的是通过函数的名称猜测出来的，因为不太确定 scanf 的机制，猜测这里 scanf 的返回值应该存储在 eax 内，为成功读取到的数据个数，`cmp $0x5,%eax` 这句会使成功读取的数据少于5的时候炸弹爆炸。）

对于 phase\_2 的汇编代码，第一条 cmp 语句可以看出在比较输入的的第一个数字，如果不为1就爆炸。之后的结构中有许多的跳转语句，可以判断这是一个循环。循环中把后面一个数字传入 %rbx 中，再把前一个数字传入 %eax 中。`add %eax,%eax` 一句再讲前一个数字乘以二，如果相等的话就可以跳过这个+36处的爆炸点。紧接着将 %rbx 指向下一个整数，比较是否和 %rsp 相等，也就是达到了最后一个整数的情况，如果没有就继续循环，达到了就跳到+64处，炸弹解除。所以只要每一个数都是前一个的两倍就可以了，答案为：

1 2 4 8 16 32

## Phase 3

接下来观察 phase\_3 的汇编代码：

Dump of assembler code for function phase\_3:


```

=> 0x0000000000400f43 <+0>: sub    $0x18,%rsp
0x0000000000400f47 <+4>: lea    0xc(%rsp),%rcx
0x0000000000400f4c <+9>: lea    0x8(%rsp),%rdx
    x/s 0x4025cf 可以得到 "%d %d", 这就是需要输入的两个数据
0x0000000000400f51 <+14>: mov    $0x4025cf,%esi
0x0000000000400f56 <+19>: mov    $0x0,%eax
0x0000000000400f5b <+24>: callq  0x400bf0 <__isoc99_sscanf@plt>
    %scanf 如果返回值为1也就是读取成功的个数为1的话就会爆炸，所以猜测这里需要至少读取两个数据才能跳过下一个爆炸点。
0x0000000000400f60 <+29>: cmp    $0x1,%eax
0x0000000000400f63 <+32>: jg     0x400f6a <phase_3+39>
0x0000000000400f65 <+34>: callq  0x40143a <explode_bomb>
0x0000000000400f6a <+39>: cmpl   $0x7,0x8(%rsp)
    %rsp + 8 如果大于7，则炸弹会爆炸。观察发现 %rsp + 8 存储的是第一个输入的整数。由于 ja 是无符号比较，所以输入的值必须大于等于0，否则也一定会爆炸。
0x0000000000400f6f <+44>: ja     0x400fad <phase_3+106>
0x0000000000400f71 <+46>: mov    0x8(%rsp),%eax
    应该是个 switch 语句，根据 rax 的值去查找跳转表对应的值。rax 是输入的的第一个整

```

数，我们可以发现

```
0x0000000000400f75 <+50>: jmpq    *0x402470(,%rax,8)
0x0000000000400f7c <+57>: mov     $0xcf,%eax
0x0000000000400f81 <+62>: jmp     0x400fbe <phase_3+123>
0x0000000000400f83 <+64>: mov     $0x2c3,%eax
0x0000000000400f88 <+69>: jmp     0x400fbe <phase_3+123>
0x0000000000400f8a <+71>: mov     $0x100,%eax
0x0000000000400f8f <+76>: jmp     0x400fbe <phase_3+123>
0x0000000000400f91 <+78>: mov     $0x185,%eax
0x0000000000400f96 <+83>: jmp     0x400fbe <phase_3+123>
0x0000000000400f98 <+85>: mov     $0xce,%eax
0x0000000000400f9d <+90>: jmp     0x400fbe <phase_3+123>
0x0000000000400f9f <+92>: mov     $0x2aa,%eax
0x0000000000400fa4 <+97>: jmp     0x400fbe <phase_3+123>
0x0000000000400fa6 <+99>: mov     $0x147,%eax
0x0000000000400fab <+104>: jmp     0x400fbe <phase_3+123>
0x0000000000400fad <+106>: callq   0x40143a <explode_bomb>
0x0000000000400fb2 <+111>: mov     $0x0,%eax
0x0000000000400fb7 <+116>: jmp     0x400fbe <phase_3+123>
0x0000000000400fb9 <+118>: mov     $0x137,%eax
```

 这里会比较`%rsp+12`的值（可以发现就是我们输入的第二个值）和`%eax`中的数据是否相等，一样的话就跳过爆炸点。

```
0x0000000000400fbe <+123>: cmp     0xc(%rsp),%eax
0x0000000000400fc2 <+127>: je      0x400fc9 <phase_3+134>
0x0000000000400fc4 <+129>: callq   0x40143a <explode_bomb>
0x0000000000400fc9 <+134>: add     $0x18,%rsp
0x0000000000400fcd <+138>: retq
```

End of assembler dump.

分析已经插入在了汇编代码中。中间的`switch`语句可以通过输入不同的值和断点来探索出对应跳转的地方。比如我在这里第一个数字为2后，将断点设置在`print /x *(0x402470 + 16)`显示出来的位置，可以发现程序跳转到了 `=> 0x0000000000400f83 <+64>: mov $0x2c3,%eax`的语句。

这里的`0x2c3`就应该是第二个数的答案。转换为十进制为707。可以推测这个题目一共有八个解答，只要输入的第二个数字和第一个数字对应的跳转关系相对应即可。进一步多次解析，可以得到第三个炸弹答案表如下：

第一个数字	跳转到的语句	第二个数字
0	400f7c <+57>: mov \$0xcf,%eax	207
1	400fb9 <+118>: mov \$0x137,%eax	311
2	400f83 <+64>: mov \$0x2c3,%eax	707
3	400f8a <+71>: mov \$0x100,%eax	256
4	400f91 <+78>: mov \$0x185,%eax	389
5	400f98 <+85>: mov \$0xce,%eax	206
6	400f9f <+92>: mov \$0x2aa,%eax	682
7	400fa6 <+99>: mov \$0x147,%eax	327

## Phase 4

汇编代码及分析如下：

Dump of assembler code for function phase\_4:

```
=> 0x000000000040100c <+0>:      sub     $0x18,%rsp
0x0000000000401010 <+4>:      lea     0xc(%rsp),%rcx
0x0000000000401015 <+9>:      lea     0x8(%rsp),%rdx
    查看scanf的格式化输入（在0x4025cf）可得到"%d %d"，所以需要输入两个整形数据。
0x000000000040101a <+14>:     mov     $0x4025cf,%esi
0x000000000040101f <+19>:     mov     $0x0,%eax
0x0000000000401024 <+24>:     callq   0x400bf0 <__isoc99_sscanf@plt>
    必须输入两个数据，否则就会直接跳转到爆炸点。
0x0000000000401029 <+29>:     cmp     $0x2,%eax
0x000000000040102c <+32>:     jne     0x401035 <phase_4+41>
    比较0xe和输入的第一个数据的大小，数据必须要小于或等于0xe(14)才能跳过爆炸点
0x000000000040102e <+34>:     cmpl    $0xe,0x8(%rsp)
0x0000000000401033 <+39>:     jbe     0x40103a <phase_4+46>
0x0000000000401035 <+41>:     callq   0x40143a <explode_bomb>
0x000000000040103a <+46>:     mov     $0xe,%edx
0x000000000040103f <+51>:     mov     $0x0,%esi
0x0000000000401044 <+56>:     mov     0x8(%rsp),%edi
    调用fuc4函数，此时edx等于14，esi等于0，edi等于输入的第一个数x，
    相当于调用func4(x, 0, 14)
0x0000000000401048 <+60>:     callq   0x400fce <func4>
0x000000000040104d <+65>:     test    %eax,%eax
    测试func4返回值是否为0。如果不为0的话会直接跳到爆炸点。
0x000000000040104f <+67>:     jne     0x401058 <phase_4+76>
    比较rp+12，应该是第二个数字是否为0，如果为0的话可以跳过爆炸点。
0x0000000000401051 <+69>:     cmpl    $0x0,0xc(%rsp)
0x0000000000401056 <+74>:     je      0x40105d <phase_4+81>
0x0000000000401058 <+76>:     callq   0x40143a <explode_bomb>
0x000000000040105d <+81>:     add     $0x18,%rsp
0x0000000000401061 <+85>:     retq
End of assembler dump.
```

发现+60处会调用func4函数，反汇编func4如下。在func4内部还会再次调用func4，可以看出这是一个递归的函数过程，进一步分析：

(gdb) disas func4

Dump of assembler code for function func4:

```
0x0000000000400fce <+0>:      sub     $0x8,%rsp
0x0000000000400fd2 <+4>:      mov     %edx,%eax
0x0000000000400fd4 <+6>:      sub     %esi,%eax
0x0000000000400fd6 <+8>:      mov     %eax,%ecx
    将%ecx右移31位（可以用来判断eax的正负，比较第二个参数和第三个参数的大小）
0x0000000000400fd8 <+10>:     shr     $0x1f,%ecx
0x0000000000400fdb <+13>:     add     %ecx,%eax
    将eax算数右移一位，即除以2
0x0000000000400fdd <+15>:     sar     %eax
```

```

0x0000000000400fdf <+17>:    lea    (%rax,%rsi,1),%ecx
0x0000000000400fe2 <+20>:    cmp    %edi,%ecx
    如果%ecx小于%edi就跳转到36并把返回值设置为0。
0x0000000000400fe4 <+22>:    jle    0x400ff2 <func4+36>
    否则将rcx减去1传给edx。
    相当于递归调用
0x0000000000400fe6 <+24>:    lea    -0x1(%rcx),%edx
0x0000000000400fe9 <+27>:    callq  0x400fce <func4>
0x000000000040fee <+32>:    add    %eax,%eax
0x000000000040ff0 <+34>:    jmp    0x401007 <func4+57>
0x000000000040ff2 <+36>:    mov    $0x0,%eax
0x000000000040ff7 <+41>:    cmp    %edi,%ecx
    如果%ecx大于%edi就结束函数，否则继续调用下一层递归。
0x000000000040ff9 <+43>:    jge    0x401007 <func4+57>
    把%rcx + 1 传递到%esi中，作为下一个func4的参数。
0x000000000040ffb <+45>:    lea    0x1(%rcx),%esi
0x000000000040ffe <+48>:    callq  0x400fce <func4>
0x0000000000401003 <+53>:    lea    0x1(%rax,%rax,1),%eax
0x0000000000401007 <+57>:    add    $0x8,%rsp
0x000000000040100b <+61>:    retq

```

End of assembler dump.

结构比较复杂，尝试逐行翻译成C语言代码如下：

```

func4(x, 0, 14)
int func4(int a, int b, int c){
    // t in %eax , q in %ecx
    // a in %rdi, b in %rsi, c in %rdx
    int t = c;
    t = t - b;
    int q = t;
    q = q >> 31;
    t = t + q;
    t = t/2;
    q = t + b;
    if (q <= a){
        t = 0;
        if (q >= a){
            return t;
        }
        else{
            b = q + 1;
            func4(a, b, c);
        }
    }
    else{
        c = q - 1;
        func4(a, b, c);
        t = 2t;
    }
    return t;
}

```

分析发现第一次运行的时候，q会被赋值为7，而当x=7的时候可以直接跳过递归部分，解除炸弹。所以答案为：

7 0

## Phase 5

汇编代码如下：

(gdb) disas phase\_5

Dump of assembler code for function phase\_5:

```
0x0000000000401062 <+0>:      push    %rbx
0x0000000000401063 <+1>:      sub     $0x20,%rsp
0x0000000000401067 <+5>:      mov     %rdi,%rbx
0x000000000040106a <+8>:      mov     %fs:0x28,%rax
0x0000000000401073 <+17>:     mov     %rax,0x18(%rsp)
0x0000000000401078 <+22>:     xor     %eax,%eax
0x000000000040107a <+24>:     callq   0x40131b <string_length>
0x000000000040107f <+29>:     cmp     $0x6,%eax
0x0000000000401082 <+32>:     je      0x4010d2 <phase_5+112>
```

这里有一个爆炸点，上面的函数为string\_length，所以推断应该输入长度为6的字符串。

```
0x0000000000401084 <+34>:     callq   0x40143a <explode_bomb>
0x0000000000401089 <+39>:     jmp     0x4010d2 <phase_5+112>
0x000000000040108b <+41>:     movzbl  (%rbx,%rax,1),%ecx
0x000000000040108f <+45>:     mov     %cl, (%rsp)
0x0000000000401092 <+48>:     mov     (%rsp),%rdx
0x0000000000401096 <+52>:     and     $0xf,%edx
```

movzbl为做了0扩展的字节传送，0x4024b0 存储的是一个字符串。这里用%rdx的偏移量将字符串中的某一个字符传递到edx中。

```
0x0000000000401099 <+55>:     movzbl  0x4024b0(%rdx),%edx
0x00000000004010a0 <+62>:     mov     %dl,0x10(%rsp,%rax,1)
```

rax加一，作为循环的计数器。

```
0x00000000004010a4 <+66>:     add     $0x1,%rax
0x00000000004010a8 <+70>:     cmp     $0x6,%rax
0x00000000004010ac <+74>:     jne     0x40108b <phase_5+41>
```

```
0x00000000004010ae <+76>:     movb    $0x0,0x16(%rsp)
0x00000000004010b3 <+81>:     mov     $0x40245e,%esi
0x00000000004010b8 <+86>:     lea     0x10(%rsp),%rdi
0x00000000004010bd <+91>:     callq   0x401338 <strings_not_equal>
0x00000000004010c2 <+96>:     test    %eax,%eax
```

eax必须等于0，否则爆炸。

```
0x00000000004010c4 <+98>:     je      0x4010d9 <phase_5+119>
0x00000000004010c6 <+100>:    callq   0x40143a <explode_bomb>
0x00000000004010cb <+105>:    nopl    0x0(%rax,%rax,1)
0x00000000004010d0 <+110>:    jmp     0x4010d9 <phase_5+119>
0x00000000004010d2 <+112>:    mov     $0x0,%eax
0x00000000004010d7 <+117>:    jmp     0x40108b <phase_5+41>
0x00000000004010d9 <+119>:    mov     0x18(%rsp),%rax
```

```
0x00000000004010de <+124>: xor    %fs:0x28,%rax
0x00000000004010e7 <+133>: je     0x4010ee <phase_5+140>
0x00000000004010e9 <+135>: callq 0x400b30 <__stack_chk_fail@plt>
0x00000000004010ee <+140>: add    $0x20,%rsp
0x00000000004010f2 <+144>: pop    %rbx
0x00000000004010f3 <+145>: retq
End of assembler dump.
```

可以看到0x4024b0里存储的字符串数据为：

```
0x4024b0 <array.3449>: "maduiersnfotvbylSo you think you can stop the
bomb with ctrl-c, do you?"
```

已经凌晨2点了，拆不下去了，累了。

the end.