Bomb Lab Writeup

王浩然 计23 2022010229

前置知识

gcc:

```
-o filename: 指定输出文件末
-S: 输出汇编代码
-c: 编译到对象文件(object file, .o / .obj)
-E: 对代码进行预处理
-g: 打开调试符号
-Werror: 把所有的warning当成error
-O{,1,2,3,s,fast,g}: 设置优化选项
```

objdump:

```
-d: 反汇编可指定的段(如.text)
-D: 反汇编所有的段，即使这个段存储的是数据，也当成指令来解析
-S: 把反汇编的代码和调试信息里的代码合在一起显示
-t: 显示函数的符号列表
--adjust-vma OFFSET: 把反汇编的地址都加上一个偏移，用于嵌入式开发
-M {intel, att} 用Intel或AT&T的风格显示X86汇编，默认AT&T
```

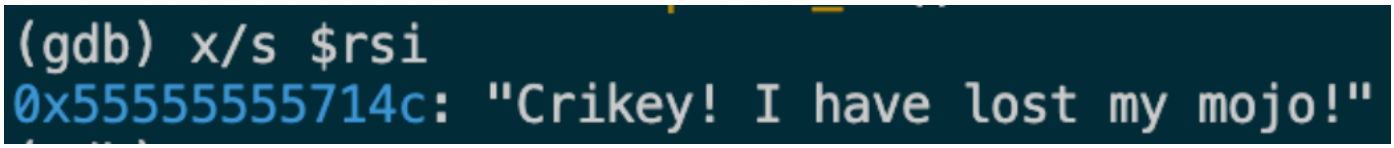
gdb:

```
gdb filename: 开始调试
gdb -q: 静默模式
run [arglist]: 开始运行
kill: 停止运行
quit: 退出gdb
break sum(*addr): 在sum函数(addr地址处)开头设置断点
delete 1: 删除断点1
clear sum: 删除sum函数入口的断点
stepi [instruction number]: 运行一条指令(区分step，后者以一条C代码为粒度)
continue: 运行到下一个端点
disas sum(*addr): 反汇编sum函数，addr地址处的函数
print /x /d/ t $rax: 将rax里的内容以16,10,2进制形式输出，$rax也可换成立即数
print *(int*)addr，将addr地址内存储的内容以int形式输出
x/2wd $rsp: 解析在rsp所指向位置的word，以十进制形式输出
info registers: 寄存器信息
info functions: 函数信息
info stack: 栈信息
set logging enabled on/off: 将gdb输出重定向到gdb.txt里
```

> Bomb Lab的最终目标是运行时不出发explode_bomb函数，也就是说我们要让每个函数正常ret，而不能jmp explode_bomb

Phase 1

阅读汇编代码可知，phase1调用`strings_not_equal`函数，结合返回值判断是否trigger bomb。若返回值为1，触发；否则不触发。而在`string_not_equal`中，若位于`rdi`和`rsi`的字符串相等，给`rax`赋值0，否则赋值1。这样我们的目标就很明确了：由于`rdi`中存储我们输入字符串的指针，`rsi`中存储的必然是给定的字符串，所以我们要找到`rsi`中存储的字符串。在phase1打断点后用指令`x/s $rsi`即可。
注意，`x/s $rsi`需要在`lea 0x1d28(%rip),%rsi`之后调用，否则`rsi`还未被赋值。



因此最终答案即为"Crikey! I have lost my mojo!"

Phase 2

第一次阅读时，了解此函数大概是读取六个数字，其符合要求则正常返回。再次阅读汇编代码，发现几个关键点：

```
# 在read_six_numbers
lea 0x186e(%rip),%rsi
mov $0x0,%eax
call 1140 <__isoc99_sscanf@plt>
```

而sscanf的定义为`int sscanf(const char *str, const char *format, ...)`，这表明rsi存储的是格式串format，即0x186e(%rip)中存储的就是格式串，读取后发现其为`%d %d %d %d %d %d`

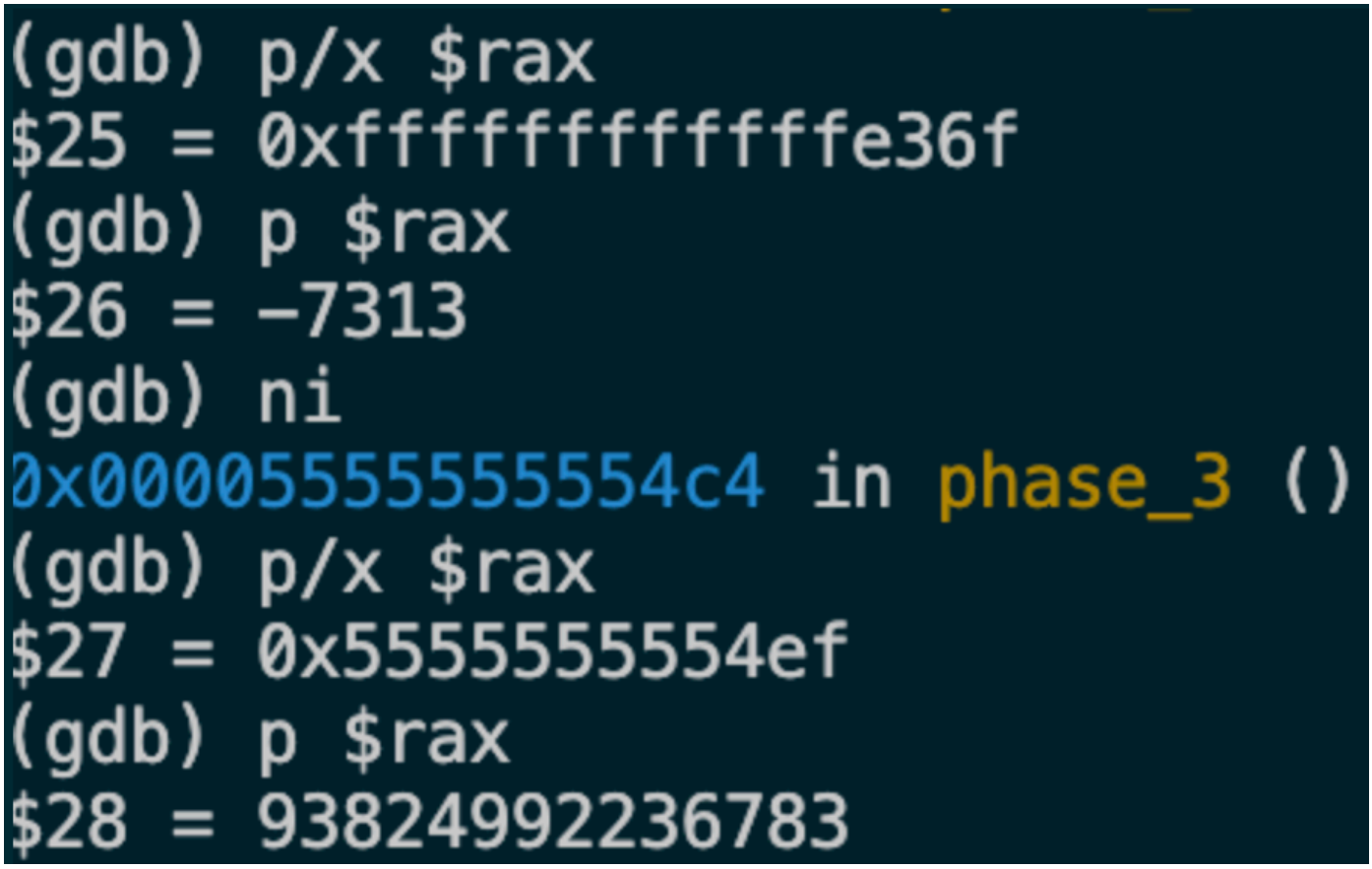
```
cmpl $0x1, (%rsp)
jne 1457 <phase_2+0x1e> # 这是explode_bomb地址
这意味着第一个数字必须为1
```

```
add %eax,%eax
cmp %eax,0x4(%rbx)
je 145e <phase_2+0x25> # 跳回上面开始比较的部分
这表明数字的比较过程是一个循环，每次比较下一个数是否是当前数的2倍
```

故需要输入2的幂次序列`1 2 4 8 16 32`
int sscanf(const char *str, const char *format, ...)

Phase 3

类似phase2可知phase3的输入格式串为`%d %c %d`
我刚开始在这里卡了很久，因为没理解后半段的结构，也不知道第二个`lea addr(%rip),%rdx`是什么意思，所以一直在进行无目的的比较，如下：



~~后来意识到自己太愚蠢了~~
实际上phase3是`switch`结构，由

```
cmpl $0x7,0xc(%rsp)
ja 15be <phase_3+0x140> # explode_bomb
```

可知，第一个参数不能超过7，也就是说第一个参数只能在{0,1,...,7}中取。
然后就是测试各个case对应的值是多少了，这里我选取第一个参数为6.
函数会跳转到`cmpl $0x1c2,0x8(%rsp)`，这意味着我们要让第三个参数取值为0x1c2，即450。
之后重新运行此程序，函数又跳转到`cmp %al,0x7(%rsp)`，由`p $al`可知al存储的值为120，而120对应的字符为x，故输入中字符应为x。
故答案为`6 x 450`

Phase 4

由`x/s $rsi`可知，格式串为`%d %d`，即输入是两个整数。

继续阅读后发现

```
call 15da <func4>
# 返回值必须是19
cmp $0x13,%eax
jne 165b # explode_bomb
# 第二个参数必须是19
cmpl $0x13,0x8(%rsp)
je 1660 # return without exception
```

阅读`func4`后可确定，`0x8(%rsp)`只出现在上面的代码段中，故确定第二个参数为19，那么第一个参数呢?

`func4`是递归函数，每次把`(r-l)>>1+l`存到`ebx`中（r是`(edx)`，l是`(esi)`），若`ebx`和`edi`相等，直接把`ebx`赋给`eax`返回，否则类似二分继续进行，最后把`ebx+func4(recurred)`赋给`eax`并返回。

手搓知，第一个参数为4

故答案为`4 19`

Phase 5

~~先是熟悉的味道，熟悉的操作~~，发现还是读入两个整数。然后借助`eax`只取第一个参数的最后四位，并且要求这后四位不能为1111。

关键代码是这一段

```
16aa: 83 c2 01             add    $0x1,%edx
16ad: 48 98                cltq
16af: 8b 04 86             mov    (%rsi,%rax,4),%eax
16b2: 01 c1                add    %eax,%ecx
16b4: 83 f8 0f             cmp    $0xf,%eax
16b7: 75 f1                jne    16aa <phase_5+0x45>
```

这是一段循环，只有在`eax`等于15时才退出，那么我们看看退出后会发生什么

```
# 要求edx=15
16c1: 83 fa 0f             cmp    $0xf,%edx
16c4: 75 06                jne    16cc <phase_5+0x67> # explode_bomb
# 要求ecx=第二个参数
16c6: 39 4c 24 08          cmp    %ecx,0x8(%rsp)
16ca: 74 05                je     16d1 <phase_5+0x6c> # return
```

而`edx`在循环中是计数器，这意味着循环要执行十五次。

循环做的事情是给`rsi`对应数组求和，把和保存在`ecx`里，这意味着第二个参数就是`rsi`数组的和。

注意第一次执行`mov (%rsi,%rax,4),%rax`，从这里可以看出我们输入的第一个参数是循环起始的下标。

那么`rsi`对应数组里存了啥呢？用`x/s`可知，其值如下

```
idx: {00, 01, 02, 03, 04, 05, 06, 07, 08, 09, 10, 11, 12, 13, 14, 15}
val: {10, 02, 14, 07, 08, 12, 15, 11, 00, 04, 01, 13, 03, 09, 06, 05}
```

为保证循环执行15次，我们需要保证当且仅当第15次循环执行完时，`eax`存的是15，由此倒推得

```
15 <- 06 <- 14 <- 02 <- 01 <-
10 <- 08 <- 04 <- 09 <- 13 <-
11 <- 07 <- 03 <- 12 <- 05
```

这又意味着第一次进入循环时，选用的数组下标为5，也就是`eax`初始为5，那么第一个参数取5即可。

而这个数组求和为120，但要注意第一次执行时其值不计入数组中，故第二个参数值为$120 - 5 = 115$

答案为`5 115`

Phase 6

这一段汇编代码太长了...

从`ret`语句倒着往前看，找到上一个跳转操作下面的一条指令，其为`1819    add $0x68,%rsp`，文件内搜索知，只有

```
cmp $0x5,%r14d
jg 17d0 <phase_6+0xf3>
mov %r14,%rbx
jmp 17bf <phase_6+0xe2>
mov 0x8(%rbx),%rbx
sub $0x1,%ebp
je 1819 <phase_6+0x13c>
```

这一段里的`je`会跳到`1819`号指令，条件为`r14`=5。又没头绪了，正着看。

`read_six_numbers`有2个参数，第一个参数为输入字符串，第二个参数是`rsp`

其返回值存储位置如下

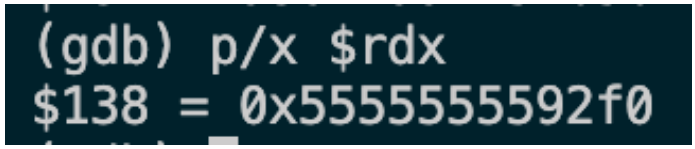
参数INDEX	存储位置	存储内容
1	%rdi	输入串
2	%rsi	格式串
3	%rdx	数组地址
4	%rcx	数组地址 + 4
5	%r8	数组地址 + 8
6	%r9	数组地址 + 12
7	(%rsp)	数组地址 + 16
8	8(%rsp)	数组地址 + 20

然后尝试阅读整个代码，看起来和链表操作很像。猜测phase6为**将一个已经存储的链表排序，要求排序好的链表中元素是按从大到小的顺序排列。**

梳理一下流程

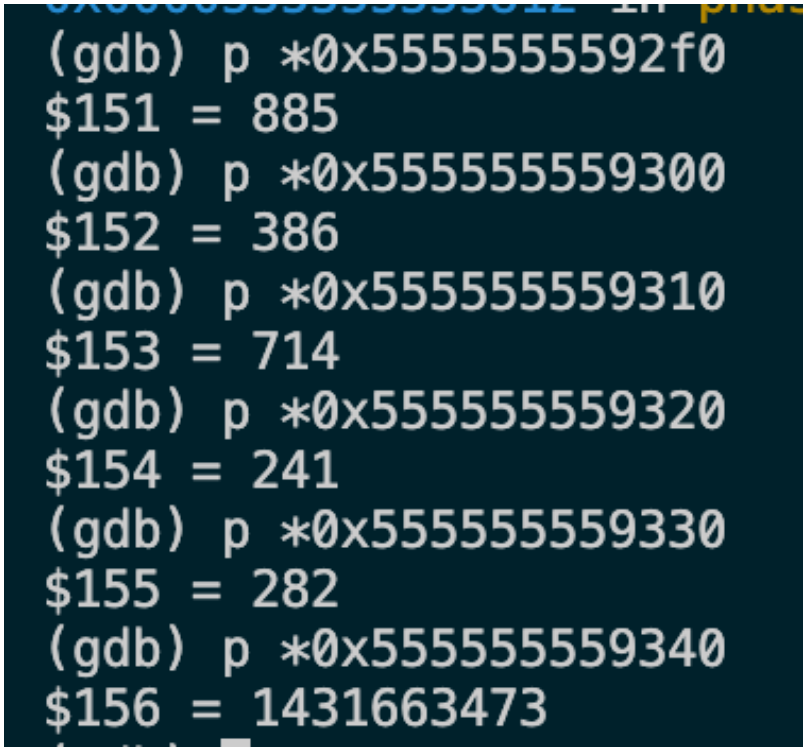
- 把输入的六个数字读到栈上，存在`rsp`到`rsp+0x18`，占用24个字节
- 循环检查数字的初值，需满足每个数字小于6且互不相同
- 把每个数字都用7减一遍，以降序排列
- 根据减法后得到的值，将列表中各节点首地址在栈中排序
- 基于排序结果，把链表连接起来，最后一个节点的next设置为0
- 检查排序后是否从大到小排序

我们需要找到链表在栈上的地址是什么。注意到`lea 0x3b9a(%rip),%rdx`，猜测这里的`rip`相对寻址就是给`rdx`赋链表初始值。在这条指令执行后读取`rdx`存储的值，我们知道列表首对应链表地址就是`0x5555555592f0`



再注意到`mov 0x8(%rdx),%rdx`，这时候看看`0x8(%rdx)`存了什么，推测链表头之间指针相差$0x300 - 0x2f0 = 0x10$。

所以各个头的地址为`0x5555555592f0`，`0x555555559300`，`0x555555559310`，`0x555555559320`，`0x555555559330`，`0x555555559340`。我们打印修改后其取得的值，可得



~~我并不清楚这些值是怎么算出来的，~~但我知道他们的相对顺序代表输入数组大小顺序

按地址取值从大到小排序，则

```
0x555555559340 -> 修改后下标是6 -> 原来是1
0x5555555592f0 -> 修改后下标是1 -> 原来是6
0x555555559310 -> 修改后下标是3 -> 原来是4
0x555555559300 -> 修改后下标是2 -> 原来是5
0x555555559330 -> 修改后下标是5 -> 原来是2
0x555555559320 -> 修改后下表是4 -> 原来是3
```

但这么做感觉有些问题，因为`0x555555559340`的值太奇怪了！这是因为最后一个链表的首节点初始时被赋值为0，其值未定义。

不过我们可以确定前五个节点的相对顺序，然后枚举第六个节点的位置即可。将$6! = 120$的枚举次数降到了6次

最终答案为`6 4 1 5 2 3`

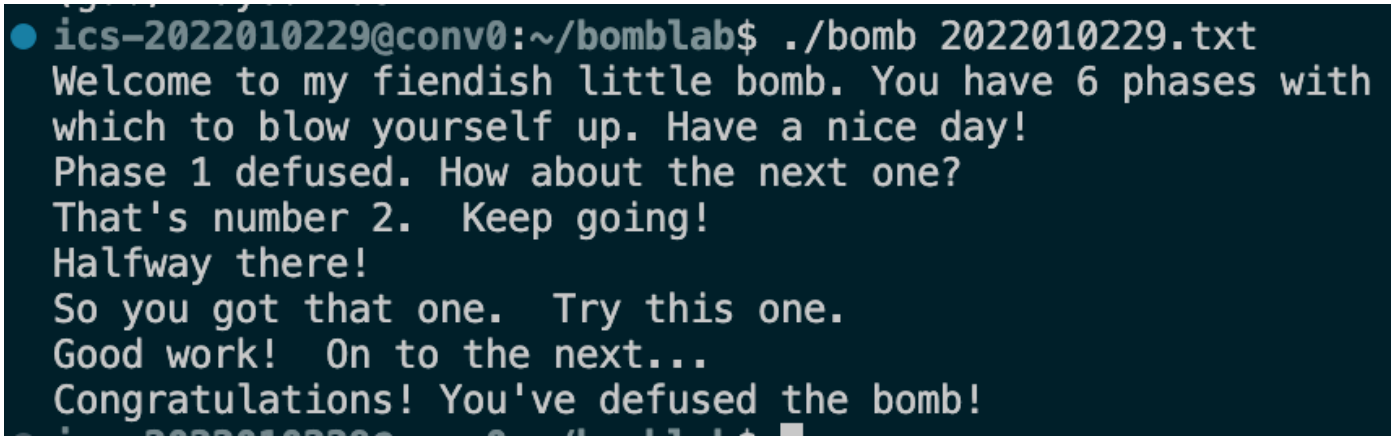
感想

真刺激！

对着一坨汇编代码看几个小时，发现其操作只是给数组排序...

阅读汇编的时候，不能一直一行一行抠，而要学会一段一段看。比如有一个phase中有`cltq`指令，这是把`eax`扩展到`rax`，但实际上后面的操作数只有`int`范围，用不到前32位。如果一直纠结`cltq`有啥用，就一定会陷进去。

为了避免如服务器重启等意外，我把通过测试的截图放在这里。



以及这个lab真的很棒（虽然是引用其他学校的），助教的文档也非常详细，尤其是贴心的gdb指令大全！让我刚上手gdb调试的时候轻松了很多，不至于连常用命令是什么都不知道，却还在瞎摸索.....

感谢张老师和助教！