bomblab 报告

姓名：施素注

学号：2024201620

总分	phase_1	phase_2	phase_3	phase_4	phase_5	phase_6	secret_phase
7	1	1	1	1	1	1	1

scoreboard 截图：

2024201619	1	0	0	0	0	0	0
2024201620	1	0	0	0	0	0	0
2024201621	0	6	7	0	2	1	0

解题报告

phase_1

```
    1439:   48 8d 35 40 1d 00 00    lea    0x1d40(%rip),%rsi
    1440:   e8 7a 08 00 00          call   1cbf <strings_not_equal>
    1445:   85 c0                   test   %eax,%eax
    1447:   75 05                   jne    144e <phase_1+0x19>
# 附上题目答案
# Open the book, turn to page 617 : Scientific Witchery
```

讲解题目思路

<string_not_equal>这个函数名就已经将phase展示差不多了，他会将读入的string与目标string进行比对，如果不完全相同就返回1，完全相同就返回0。函数结果存在%eax中，再由test进行更新ZF，jne在ZF = 1时会跳到<explode_bomb>导致炸弹爆炸。

为了得到目标string，利用gdb，然后`x/s 0x555555557180`得到%rsi寄存器指向的内存的目标字符串。

phase_2

```
14e5:   48 83 f8 03             cmp    $0x3,%rax
14e9:   75 ed                   jne    14d8 <phase_2+0x83> # 其中一个循环
……
14d8:   8b 14 87                mov    (%rdi,%rax,4),%edx
14db:   0f af 14 c6             imul   (%rsi,%rax,8),%edx  # 矩阵乘法
# 附上题目答案
# 783982 1252667 703755 1086747
```

讲解题目思路

这道题目的代码主要是实现矩阵$A_{2\times3}B_{3\times2}=result_{2\times2}$的乘法。

代码主体是三层循环，是主要由%r8d（0-1），%r11d（0-1），%rax(0-2)这三个寄存器存储循环变量。可以知道这是$2\times3\times2$的矩阵乘法。`(%rdi,%rax,4),%edx`，`(%rsi,%rax,8),%edx`这两个每次乘法偏移量不同也印证了矩阵A是第一个矩阵，矩阵B的列数为2。

一开始我把矩阵A，B的元素都用gdb读了出来，想着算出来。但是后来想到算法已经算过一次矩阵乘法把答案存在result里面了，直接读取result就行了。

`x/4d 0x7fffffffdab0`读取result得到的四个元素就是答案。

phase_3

```
15b3:   8d 83 3a 01 00 00       lea    0x13a(%rbx),%eax # eax = %rbx + 0x13   此时%rbx(%ebx)在case里面赋值了是0 ,  m = 3
15b9:   2d 3a 01 00 00          sub    $0x13a,%eax # eax -= 0x13a,m = 4
15be:   05 3a 01 00 00          add    $0x13a,%eax # eax += 0x13a
15c3:   2d 3a 01 00 00          sub    $0x13a,%eax # eax -= 0x13a
15ce:   39 44 24 04             cmp    %eax,0x4(%rsp) # n 等于 eax 不然就爆炸
..............
15ee:   c3                      ret    # 安全退出程序
...........

1604:   bb 00 00 00 00          mov    $0x0,%ebx            # m = 3
1609:   eb a8                   jmp    15b3 <phase_3+0x6f>
160b:   b8 00 00 00 00          mov    $0x0,%eax            # m = 4
1610:   eb a7                   jmp    15b9 <phase_3+0x75>

# 附上题目答案
# 5 -314
```

讲解题目思路

这题代码的核心是switch跳转。sscanf输入两个整数，第一个数m要在0-7之间,才能进入switch，但是又要求不能大于5，不然也炸。第二个数n不能为负数。

经由switch-case跳转后，m如果是0-2，他带入的case又会跳转到无论如何都会爆炸的地方，所以m只能是3，4，5了。第二个数n最终要和%eax一致，但是这个%eax具体是多少又取决于m带进入的case。

但是奇怪的是，我这带入gdb跑的时候，发现m=0时不会进入case，后面的m都会进入case（m-1）,最后只有m = 5时可行，此时eax小于0，为-314，也就是对应的n.

phase_4

```
# 附上题目答案
# 31 AC
```

讲解题目思路

phase_4的关键是读懂func4_1和func4_2.

func4_1就是求$2^n - 1$

func4_2就是不断对调三个字符的位置，到特定点退出。

`(gdb) x/s  0x7fffffffda94`

`0x7fffffffda94: "AC"` 也可以通过gdb读取寄存器得到答案。。。

```
int func4_1(edi)
{
    if(edi == 0) return eax;
    eax = edi;
    if(edi == 1) return eax;
    edi -= 1;
    func4_1(edi);
    eax = 2 * rax + 1;
} // 也就是求 2^edi - 1
void func4_2(edi(5),esi(4),edx('A'),ecx('C'),r8d('B'),rbx(string)) {
    int eax = func4_1(edi - 1);
    // eax = 15 , 7 , 3 , 1
    // eax = 15   func4_2(4,4,B,A,C)
    // eax = 7    func4_2(3,4,C,B,A)
    // eax = 3    frbx = 'rcx' 'rdx' = A,C
    if (esi > eax) {
        if (esi == eax + 1) {
            rbx = 'r8d' 'rdx' = AC;
            return ;
        }
        else {
            func4_2(edi - 1,0,ecx('C'),r8d('B'),edx('A'),out);
        }
    }
    else {
        func4_2(edi - 1, esi,r8d('B'),edx('A'),ecx('C'), out);
    }
}
```

phase_5

```
# 附上题目答案
# .'$%)(
```

讲解题目思路

题目就是读入长度为6的字符串，然后这个字符串里面的字符都要经过一套操作最后得到一个数（0-15）之间。

```
   17dc:   0f be 04 13            movsbl (%rbx,%rdx,1),%eax # 先扩充到32位
   17e0:   83 c0 0f               add    $0xf,%eax # 加 0xf
   17e3:   83 e0 0f               and    $0xf,%eax # 然后和0xf与运算
```

然后得到的这个数其实就是中间给的一个字符串的其中一个字符的下标,提取出这个字符后在存在一个新的字符串里面。

`"maduiersnfotvbyl So you think you can stop the bomb with ctrl-c, do you?"` （这个字符串phase_3也读到过，估计是系统内存中和phase_3连在一起了）

这个字符串可以注意到前面的不规则字符正好长度为16，与eax的范围相符。后面的就是纯纯彩蛋了，可以忽略。

```
(gdb) x/s 0x5555555571f4
0x5555555571f4: "bruins"
```

可以知道`bruins`是目标字符串，对应下标为`[13][6][3][4][8][7]`。接着就要构建一个字符串在操作后能得到这些下标。然后其实字符的ascii码的最后四位等于对应下标加1就行了（因为加了0xf），我找的是`.'$%)(`，通过了这题。

phase_6

```
# 附上题目答案
# 6 2 1 3 4 5
```

讲解题目思路

这道题的基本思路是一个长度为6的链表，会根据输入的6个数字进行重排，重排完后的链表得是一个递减的序列。

但是对应输入的这六个数，程序会进行一些操作：

1，检查这六个数是否两两互不相等

```
    1886:   48 83 c3 01             add    $0x1,%rbx
    188a:   83 fb 05                cmp    $0x5,%ebx
    188d:   0f 8f a7 00 00 00       jg     193a <phase_6+0xff> # 当rbx > 5 跳出循环
    1893:   41 8b 44 9d 00          mov    0x0(%r13,%rbx,4),%eax # eax = 第 rbx 个
数
    1898:   39 45 00                cmp    %eax,0x0(%rbp) # rbp 也是 第 i 数  就是
输入的数互不相等
    189b:   75 e9                   jne    1886 <phase_6+0x4b> # 不相等就回,否则就爆
炸
    189d:   e8 82 06 00 00          call   1f24 <explode_bomb>
```

这个循环每次都只用将num_i与$num_{i+1\sim6}$进行比较，而不是都从num_1开始比，所以就不用跳过与自身的比较。

2，将这六个数都进行$num' = 7 - num$的处理

```
    18a4:   48 8b 54 24 08          mov    0x8(%rsp),%rdx # 把指针给rdx了
    18a9:   48 83 c2 18             add    $0x18,%rdx # 第6个数字，也就是结束循环位置
    18ad:   b9 07 00 00 00          mov    $0x7,%ecx # 等于7
    18b2:   89 c8                   mov    %ecx,%eax
    18b4:   41 2b 04 24             sub    (%r12),%eax # eax =  7 - r12
    18b8:   41 89 04 24             mov    %eax,(%r12) # (%r12) 也就是输入的数变成了
7 - x;
    18bc:   49 83 c4 04             add    $0x4,%r12 # r12这个指针向后移动
    18c0:   4c 39 e2                cmp    %r12,%rdx # 没到最后一个数就跳回去
    18c3:   75 ed                   jne    18b2 <phase_6+0x77>
```

3，新的链表$list'[i] = list[num']$

```
    18c5:   be 00 00 00 00          mov    $0x0,%esi
    18ca:   8b 4c b4 10             mov    0x10(%rsp,%rsi,4),%ecx # 读取数组
    18ce:   b8 01 00 00 00          mov    $0x1,%eax
    18d3:   48 8d 15 36 39 00 00    lea    0x3936(%rip),%rdx        # 5210
<node1>
    18da:   83 f9 01                cmp    $0x1,%ecx # 比较1与ecx
    18dd:   7e 0b                   jle    18ea <phase_6+0xaf> # ecx小于等于1就跳
    18df:   48 8b 52 08             mov    0x8(%rdx),%rdx # 数组往下跳
    18e3:   83 c0 01                add    $0x1,%eax # +1
    18e6:   39 c8                   cmp    %ecx,%eax # 比较
    18e8:   75 f5                   jne    18df <phase_6+0xa4> # 不等于 就回去,读数
组指定下标 (系统内存里有个表，第i个数)
    18ea:   48 89 54 f4 30          mov    %rdx,0x30(%rsp,%rsi,8) # 存到链表中
```

这里还有个边际处理，$num' \leq 1$的话统一赋值$list'[i] = list[1]$,但是程序之前已经有对num的限制在$(1 \sim 6)$了。

最后得到的新链表得是递减的。

```
(gdb) x/24d 0x555555559210
0x555555559210 <node1>: 914     1       1431671328      21845
0x555555559220 <node2>: 192     2       1431671344      21845
0x555555559230 <node3>: 521     3       1431671360      21845
0x555555559240 <node4>: 686     4       1431671376      21845
0x555555559250 <node5>: 822     5       1431671136      21845
0x555555559260 <host_table>:    1431664120      21845   1431664125      21845
```

```
(gdb) x/4dw  93824992252256
0x555555559160 <node6>: 759     6       0       0
```

另外原链表的$node1 \sim 5$的内存是连在一起的，但是$node6$却不再一起，得再读取$node5$的$*next$才能读到$node6$

node	1	2	3	4	5	6
list	914	192	521	686	822	759
list'	914	822	759	686	521	192
num'	1	5	6	4	3	2
num	6	2	1	3	4	5

secret_phase

```
# 附上题目答案
# cipher (进入secret_phase,与phase_6的答案在同一行)
# 33022
```

在结束了6个phase后，还要输入密码才能进入secret_phase。

点击secret_phase在asm中查找匹配项，发现secret_phase还出现在phase_defused里面。看来进入secret_phase的关键在这里面。

```
(gdb) x/s 0x555555559978
0x555555559978 <input_strings+600>:     "6 2 1 3 4 5"
```

打印字符串，发现正好是phase_6的答案。

```
21d7:   48 8d 05 9a 37 00 00    lea    0x379a(%rip),%rax        # 5978
<input_strings+0x258>
21de:   48 8d 3c 06             lea    (%rsi,%rax,1),%rdi
21e2:   48 8d 35 08 14 00 00    lea    0x1408(%rip),%rsi        # 35f1
<array.0+0x3b1>
21e9:   e8 d1 fa ff ff          call   1cbf <strings_not_equal>
```

然后此时 rsi 为 12，rax为phase_6答案地址下标，然后又读取了一个字符串再与目标字符串比对。

这里有两个坑，第一个进入secret_phase的密码要和gdb在同一行，而不能换行，这也就是为什么读取字符串是从（%rax + 12）开始（6个数字加6个空格）。

第二个坑，目标字符串是“cipher”，翻译过来就是密码，我原本想着会是什么乱码，这个“cipher”是密码键值对的key，胡思乱想了半天，后来试了以下“cipher”就进到secret_phase才知道真的是这个。

接着是secret_phase

secret_phase函数本身不复杂，就是输入一个长度小于20的字符串，复杂的是他调用的func7。

func7一过来就洋洋洒洒赋了32个值，看傻了。

rsp	0	1	2	3	4	5	6	7
0	-2	-1	1	2	2	1	-1	-2
1	1	2	2	1	-1	-2	-2	-1
2	-1	0	0	1	1	0	0	-1
3	0	1	1	0	0	-1	-1	0

这就是一个表，改变x，y的。

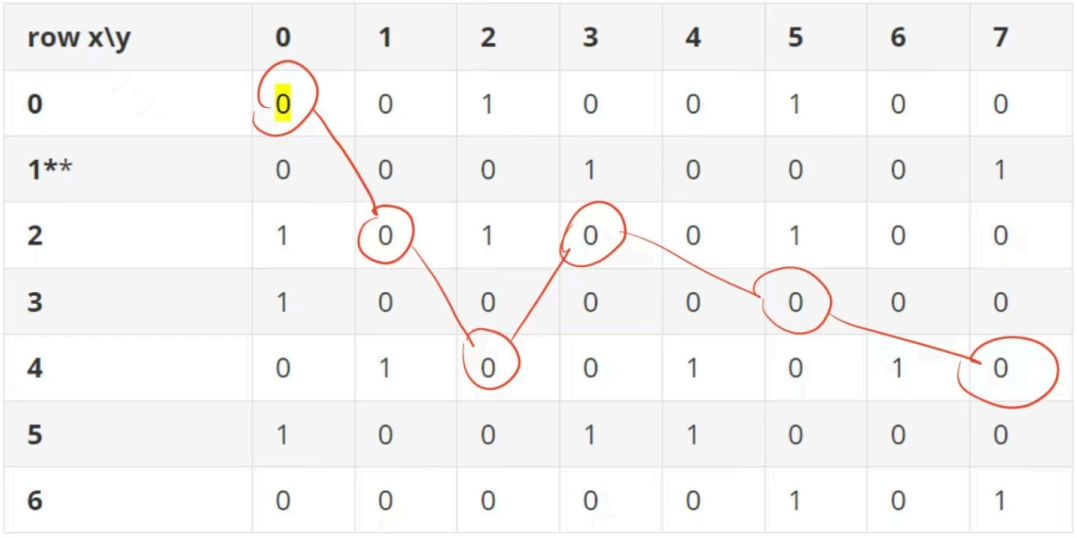
还有一个链表row，但他也可以化成表格的形式。下面展示读取row0，1的片段。

row由 check[8] 和 *next组成，注意指针在地址靠后的地方。check为[0]或[1]，当row$[x']$$[y']$为1时也不行，func7会返回0。当row$[x+table[x]]$$[y+table[y]]$为1时也不行，func7也会返回0。

仔细观察$x \to x', y \to y'$都是2，1的组合，而$[x+table[x]]$$[y+table[y]]$都是0，1的组合。这就是中国象棋里面马的走法！马的跳往的地方不能为1，然后马也不能被蹩马脚。

```
(gdb) x/32tb 0x555555591a0
0x5555555591a0 <row0>:  00000000        00000000        00000001        00000000
00000000        00000001        00000000        00000000
0x5555555591a8 <row0+8>:        10110000        10010001        01010101
01010101
        01010101        01010101        00000000        00000000
0x5555555591b0 <row1>:  00000000        00000000        00000000        00000001
00000000        00000000        00000000        00000001
0x5555555591b8 <row1+8>:        11000000        10010001        01010101
01010101
        01010101        01010101        00000000        00000000
```

row x\y	0	1	2	3	4	5	6	7
0	0	0	1	0	0	1	0	0
1	0	0	0	1	0	0	0	1
2	1	0	1	0	0	1	0	0
3	1	0	0	0	0	0	0	0
4	0	1	0	0	1	0	1	0
5	1	0	0	1	1	0	0	0
6	0	0	0	0	0	1	0	1



然后ascii码里面数字对应的ascii码的后四位与他自身相等，33022就是路径的代码，输入之后就通过了。

反馈/收获/感悟/总结

花费的时间：10h+

难度：总体来说这次lab就是从汇编语言到C++语言，然后再读懂程序的一个过程。不过这个过程中可以借助gdb这个工具有的时候可以取巧。刚写这个lab的时候我们连jmp都还没讲到，写起来就比较难受，写到后面的时候已经搞懂这些汇编指令了，但是代码又开始复杂起来了😂。

不合理的地方：有一些的phase是程序跑完得到一些值，然后再判断你的答案和他程序结果一不一样。判断答案的方式就是一个test or cmp这样的方式，其实可以让程序能撑到检测答案的地方，然后知道答案是什么形式（int，string……）什么长度，然后用gdb读一下寄存器里面的判等的值（也就是答案）就行了，而不需要真正看懂所有汇编代码，或者说跑一遍程序就能通过，但是这种方式不能锻炼自己读汇编语言的能力。但是如果自己又人肉在脑海里跑程序的话，好像又太难了Orz，感觉需要把汇编代码转C++，然后再去跑得到答案才是最佳的做法。

有趣的地方：有的时候读字符串地址会读到一些彩蛋。

```
So you think you can stop the bomb with ctrl-c, do you?"
```

每写一个phase就像解一道迷，写起来还是很有意思的。尤其是secret_phase写到最后恍然大悟是走马的程序，给人眼前一亮的感觉。

用到的gdb指令

1. `gdb bomb` 用gdb运行文件
2. `b *phase_() +` 加断点
3. `layout asm,layout regs` 打开当前行视图，打开寄存器视图
4. `x/(读取数量)(读取类型)(读取字长) 地址` 按指定格式读取内存里面的数据
5. `info registers eax` 读取寄存器，`layout regs` 只显示64位寄存器的数据，用这个info读取低位寄存器比较方便。
6. `si` 运行一行
7. `c` 运行到断点处

收获：对寄存器的理解加深了，比如函数传参就是$\%edi \rightarrow \%esi \rightarrow \%edx \rightarrow \%ecx \ldots\ldots$，然后对汇编语言编写的逻辑的感受加深了了，比如他的jmp老是跳来跳去的。

参考的重要资料

更适合北大宝宝体质的 Bomb Lab 踩坑记 • Arthals' ink：这里面教了gdbinit来一次性输入答案和添加断点。减少了重复输入答案耗费的时间，避免了炸弹爆炸的风险。

CSAPP课本和老师的ppt帮助看懂汇编语言。

【Linux】GDB保姆级调试指南（什么是GDB？GDB如何使用？） gdb教程-CSDN博客