lab 4

杨乙　　21307130076　　信息安全

Task1

1. 缓冲区大小分析

对二进制程序进行反汇编，函数执行前的入栈指令如下（分析见注释）：

```
stp     x29, x30, [sp, #-144]!          // 栈开拓 144B，上一个函数 FP 和 LR 寄存器写入当前栈顶
mov     x29, sp                         // 当前栈指针保存至 x29

adrp    x0,  0 <__abi_tag-0x278>        // check 函数地址保存至 [sp + 120]
add     x0,  x0,  #0x7d4
str     x0,  [sp, #120]

mov     w0,  #0x2                       // [sp + 140] 存储变量 v_140 = 2
str     w0,  [sp, #140]

str     xzr, [sp, #128]                 // [sp + 128] 存储变量 v_128 = 0
str     xzr, [sp, #24]                  // [sp + 24] 为数组 a 起始位置，且 a[0] = 0

mov     x0,  #0x1
str     x0,  [sp, #32]                  // a[1] = 1

mov     x0,  #0x1
str     x0,  [sp, #40]                  // a[2] = 1
b       928 <main+0x90>
```

通过分析，缓冲区（数组 `a`）从 `sp + 24` 起始，到 `sp + 120` 结束，因此缓冲区大小为：

```
120 - 24 = 96 字节
```

若存储 64 位整数，则数组长度为：

```
96 / 8 = 12
```

2. 越界写入的指令：

`main` 函数执行过程的汇编指令：

```
ldr     w0,  [sp, #140]                 // w0 为 2（v_140）
sub     w0,  w0,  #0x1                  // w0 为 1
```

```
sxtw    x0,  w0                         // w0 符号扩展，写入 x0
lsl     x0,  x0,  #3                    // x0 * 8
add     x1,  sp,  #0x18                 // x1 值为数组 a 起始位置
ldr     x1,  [x1, x0]                   // x1 值为：a[v_140 - 1]

ldr     w0,  [sp, #140]
sub     w0,  w0,  #0x2
sxtw    x0,  w0
lsl     x0,  x0,  #3
add     x2,  sp,  #0x18
ldr     x0,  [x2, x0]                   // x0 值为：a[v_140 - 2]

add     x0,  x1,  x0                    // v_128 = a_24[v_140 - 1] + a_24[v_140
- 2]
str     x0,  [sp, #128]

ldr     w0,  [sp, #140]                 // v_140 自增 1
add     w1,  w0,  #0x1
str     w1,  [sp, #140]

sxtw    x0,  w0                         // w0 保存 v_140 原来的值
sl      x0,  x0,  #3
add     x1,  sp,  #0x18
ldr     x2,  [sp, #128]
str     x2,  [x1, x0]
ldr     x1,  [sp, #128]                 // a[v_140原] = v_128

mov     x0,  #0xe7ff
movk    x0,  #0x4876, lsl  #16
movk    x0,  #0x17,   lsl  #32
cmp     x1,  x0                         // 比较 a[v_140原] 和 0x00174876e7ff
b.le    8d0 <main+0x38>                 // 小于等于，循环

add     x0,  sp,  #0x18
ldr     x1,  [sp, #120]
blr     x1                              // 跳转至 check 函数
mov     w0,  #0x0
ldp     x29, x30, [sp],  #144           // 返回
ret
```

通过分析，`main` 函数的伪代码实现如下：

```
int64 a[12];
a[0] = 0;
a[1] = a[2] = 1;

int64 i;
for (i = 2; a[i] < 99999999999; i++) {
    a[i] = a[i - 1] + a[i - 2];
}
```

实际作用是用一个数组保存斐波那契数列的值。但是因为数组太小（12 个整数）而循环结束条件太大，会发生缓冲区溢出

Task 2

1. check 函数逻辑分析

`check` 函数的汇编指令（分析见注释）：

```
stp     x29, x30, [sp, #-64]!           // 栈开拓 64B，上一个函数 FP 和 LR 寄存器写入
当前栈顶
mov     x29, sp                         // 当前栈指针保存至 x29

str     x0,  [sp, #24]
ldr     x0,  [sp, #24]                  // [sp + 24] 存储变量 v_24 = a
ldr     x0,  [x0]
str     x0,  [sp, #56]                  // [sp + 56] 存储变量 v_56 = a[0]
ldr     x0,  [sp, #24]
ldr     x0,  [x0, #8]
str     x0,  [sp, #48]                  // [sp + 48] 存储变量 v_48 = a[1]
ldr     x0,  [sp, #24]
ldr     x0,  [x0, #16]
str     x0,  [sp, #40]                  // [sp + 40] 存储变量 v_40 = a[2]
b       870 <check+0x9c>                // 跳转

ldr     x1,  [sp, #40]
adrp    x0,  0 <__abi_tag-0x278>
add     x0,  x0, #0x978
bl      680 <printf@plt>
ldr     x1,  [sp, #56]
ldr     x0,  [sp, #48]
add     x0,  x1, x0
ldr     x1,  [sp, #40]
cmp     x1,  x0                         // 比较 v_40 和 v_48 + v_56
b.eq    840 <check+0x6c>                // 相等，跳过输出
adrp    x0,  0 <__abi_tag-0x278>        // 不相等，输出错误信息
add     x0,  x0,  #0x980
bl      670 <puts@plt>
b       890 <check+0xbc>

ldr     x0,  [sp, #24]
ldr     x0,  [x0]                       // x0 存储 a[i] 值
str     x0,  [sp, #56]                  // 写回 [sp + 56]
ldr     x0,  [sp, #24]                  // x0 存储 a[i] 地址
ldr     x0,  [x0, #8]
str     x0,  [sp, #48]                  // v_48 = a[i + 1]
ldr     x0,  [sp, #24]
ldr     x0,  [x0, #16]
str     x0,  [sp, #40]                  // v_40 = a[i + 2]
ldr     x0,  [sp, #24]
add     x0,  x0,  #0x8                  // ++i
str     x0,  [sp, #24]

adrp    x0,  0 <__abi_tag-0x278>
```

```
add     x0,  x0,  #0x7d4
ldr     x1,  [sp, #40]                  // v_40
cmp     x1,  x0                         // v_40 和 check 地址作比较
b.ne    808 <check+0x34>                // 不相等，循环

adrp    x0,  0 <__abi_tag-0x278>
add     x0,  x0,  #0x988
bl      670 <puts@plt>                  // 输出正确信息
ldp     x29, x30, [sp],  #64
ret
```

通过分析，`check` 函数对应的伪代码如下：

```
void check(int64* a) {
    int64 i = 0;
    int64 tmp;
    while (tmp != (int64)check) {
        if (a[i + 2] != a[i + 1] + a[i]) {
            puts("ERROR");
            return;
        }
        tmp = a[i + 2];
        ++i;
    }
    puts("DONE");
    return;
}
```

函数的原理是：对于数组依次判断第三个项是否等于前两个项之和，且第三个项等于 `check` 地址时退出循环。当发生缓冲区溢出时，`check` 地址会被覆盖，会导致直到全部遍历写入的数据，第三个项不等于前两个项时（此时第三个项是栈上某一块地址），打印错误信息。图示如下

```
             +-----------------------------+
             |       another address       |
             +-----------------------------+
  a ends--> |                             |
             |                             |
             +-----------------------------+
             |           check             |
             +-----------------------------+
             |                             |
             |                             |
a begins--> |                             |
             +-----------------------------+
```

2. 交叉引用列表

首先通过 `readelf -S lab4` 命令确定节头信息，以便确定交叉引用的类型：

段	开始	大小
.text	0x6c0	0x298
.data	0x20030	0x10
.bss	0x20040	0x8
.rodata	0x970	0x1d
.plt	0x610	0x80

`check` 函数交叉引用列表：

交叉引用数	行数	指令	类型
0x870	804	b 870	c2c
0x978	80c、810	adrp x0, 0 add x0, x0, #0x978	c2d
0x680	814	bl 680	c2d
0x840	82c	b.eq 840	c2c
0x980	830、834	adrp x0, 0 add x0, x0, #0x980	c2d
0x670	838	bl 670	c2d
0x890	83c	b 890	c2c
0x7d4	870、874	adrp x0, 0 add x0, x0, #0x7d4	c2c
0x808	880	b.ne 808	c2c
0x988	888	adrp x0, 0 add x0, x0, #0x988	c2d
0x670	88c	bl 670	c2d

`main` 函数交叉引用列表：

交叉引用数	行数	指令	类型
0x7d4	8a4	adrp x0, 0 add x0, x0, #0x7d4	c2c
0x928	8cc	b 928	c2c
0x8d0	93c	b.le 8d0	c2c

Task 3

漏洞修复过程：

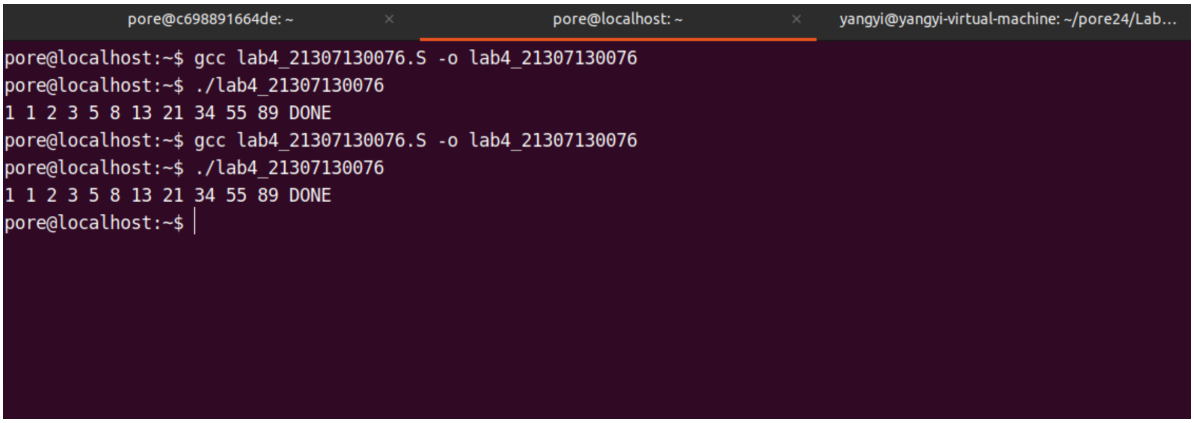
对 `check` 函数和 `main` 函数进行重汇编，交叉引用的重新符号化按照 lab4 网站上给出的方法即可。具体的修改见文件。漏洞修复的思路：之前已经分析得出缓冲区大小为 96B，即 12 个 64 位整数。因此可以增加边界检查：

```
main:

// ......

ldr w0, [sp, #140]
add w1, w0, #0x1
cmp w0, #12             // new
beq     .B7             // new

// ......

.B7:                    // new
add x0, sp, #0x18
ldr x1, [sp, #120]
blr x1
mov w0, #0x0
ldp x29, x30, [sp], #144
ret
```

汇编代码中每次将 `w0` 的值加 1 后，`w0` 中存储的是接下来最大的数组索引。因此可以判断 `w0` 是否等于 12，如果相等则结束循环。修改后文件的伪代码：

```
int64 a[12];
a[0] = 0;
a[1] = a[2] = 1;

int64 i;
for (i = 2; a[i] < 99999999999; i++) {
    if (i == 12) {
        break;
    }
    a[i] = a[i - 1] + a[i - 2];
}
```

重汇编的文件运行效果如下：


pore@c698891664de: ~
pore@localhost: ~
yangyi@yangyi-virtual-machine: ~/pore24/Lab...
pore@localhost:~$ gcc lab4_21307130076.S -o lab4_21307130076
pore@localhost:~$ ./lab4_21307130076
1 1 2 3 5 8 13 21 34 55 89 DONE
pore@localhost:~$ gcc lab4_21307130076.S -o lab4_21307130076
pore@localhost:~$ ./lab4_21307130076
1 1 2 3 5 8 13 21 34 55 89 DONE
pore@localhost:~$