lab2_report

答案：

phase_1：

Border relations with Canada have never been better.

phase_2:：

1 2 4 8 16 32

phase_3：

0 207
1 311
2 707
3 256
4 389
5 206
6 682
7 327

phase_4：

0 0
1 0
3 0
7 0

phase_5：

答案是 6 个字符，有许多组合，组合如下：

第一个字符：	)	9	y	Y	i	I
第二个字符：	/	?	_	o	O	
第三个字符：	.	>	^	~	n	N
第四个字符：	%	5	u	U	e	E
第五个字符：	&	6	v	V	f	F
第六个字符：	‘	7	w	W	g	G

密码就可以为：**)/.%&’**

phase_6：

4 3 2 1 6 5

secret_phase：

解锁：需要在输入 phase_4 的密码时在后面输入”**DrEvil**”字符串才能解锁 secret_phase

密码：**22**

推演过程：

phase_1：

当对函数产生的反汇编代码检查时，发现其代码很少，最主要功能是掉用strings_not_equal 函数，如果该函数返回值为 0 救跳转到最后不执行函数 explode_bomb，否则执行 explode_bomb，使炸弹爆炸，程序运行结束。后面直接去检查 strings_not_equal 函数的反汇编代码。输入字符串"a"使程序开始执行。

函数首先进行压栈腾出几个寄存器，将存第一参数寄存器%rdi 的值拷给了一份给%rbx，将存第二参数寄存器%rsi 的值拷给了一份给%rbp，然后以%rdi 的值作为参数调用函数 string_length 来计算字符串长度。通过检查调用后寄存器%rax 的值得到结果是 1，可以确定第一参数是输入的字符串； 检查第二次调用 string_length 后%rax 的值发现是 52，则密码的长度有 52 位。于是重新执行程序，输入 52 个'a'使输入字符串长度与密码长度相等，使该函数能继续进行下去（不相等函数会直接返回 1）。

输入字符串与密码在调用完 string_length 后分别存在%rbx 与%rbp 中，储存的是地址，后面的代码是输入字符串与密码一位一位的比较，检查%rbp 的值为 0x402400，因此直接输入 print (char*) 0x402400 检查这个地址储存的字符串，得到的就是密码："Border relations with Canada have never been better."。再观察 phase_1 函数的反汇编代码，发现在第二行 0x402400 就已经被传入寄存器%rsi 作为函数 strings_not_equal 的第二个参数了。

phase_2：

通过 phase_2 的反汇编代码，可以看到调用了函数<read_six_numbers>，可以推测密码应该是 6 个数字，这个函数的功能是读取 6 个数字，把读入的数字从最后一个开始依次放入栈中，即栈顶对应的数字是第一个输入的数字，如果输入的数字不足 6 个就引爆炸弹（输入字母不计算为数字），多于 6 个就只取前 6 个。

在输入完后，输入的值都保存在栈中，phase_2 中用了一个循环来限制输入，将汇编翻译成一下就是：（用 () 表示取寄存器存储的地址指向的值）

```
if((%rsp)!=1)//用于判断输入的第一个数是否为 1，不是 1 就引爆炸弹
    explode_bomb();
%rbx=%rsp+0x04;//初始化，%rbx 存的是第二个数的地址（int 类型占用 4byte）
%rbp=%rsp+0x18;//%rbp 存的是最后一个数的地址+4（0x18=24）
do        //循环
{
    %eax=(%rbx-0x4);//%eax 存的是%rbx 存的地址中的数的前一个数
    %eax<<=1;
    if(%eax!=(%rbx));//判断%eax*2 是否等于(%rbx)，不等于就引爆炸弹
        explode_bomb();
    %rbx=%rbx+0x4;//更新语句，使%rbx 存储下一个数的地址
}while(%rbp!=%rbx);//循环判断条件
```

.....	
x6	<- %rbp
x5	<- 最后一个数的地址
x4	
x3	
x2	
x1	<- %rbx（初始时寄存器%rbx 指向的地址） <- %rsp

因此可以推测得到后一个数是前一个数的两倍且第一个数必须是 1，因此可得到密码。

phase_3:

phase_3 是一个关于 switch 的解密，直接查看 phase_3 函数的反汇编代码，可以发现在调用函数__isoc99_sscanf@plt 前，将一个立即数传入储存第一个参数的寄存器%rsi 中，并且可以推测该函数为标准输入函数，因此利用 print (char*) 0x4025cf 查看这个地址储存的东西，

```
0x0000000000400f51 <+14>:    mov    $0x4025cf,%esi
0x0000000000400f56 <+19>:    mov    $0x0,%eax
0x0000000000400f5b <+24>:    callq  0x400bf0 <__isoc99_sscanf@plt>
```

发现是"%d %d"，因此可以推测要求输入两个数字（phase_2 中也可以这样推测输入）。并且可以通过查看寄存器状态与栈得知输入的第一个数储存在地址%rsp+0x8 中，第二个数储存在地址%rsp+0xc 中。接下来检查输入的数是否为 2 个，是的话继续，否则引爆炸弹。

接下来就是 switch 语句，下图是实现从跳转表读取数据实现跳转的语句，%rax 中储存第一个输入的数，而 0x402470 中储存的值对应%rax 等于 0 时应该跳转的地址，0x402470+8 中储存的值对应%rax 等于 1 时应该跳转的地址，依次递增 8，可以通过 x/gx (0x402470)，x/gx (0x402470+8)……依次读出对应的跳转到的代码地址。

```
0x0000000000400f75 <+50>:    jmpq   *0x402470(,%rax,8)
```

将函数大致翻译成 c 代码就是:

```
int phase_3()
{
    int a,b;
    int val;
    val=scanf("%d %d",&a,&b);
    if(val<2)
        explode_bomb();
    switch(a)
    {
        case 0:val=207;break;
        case 1:val=311;break;
        case 2:val=707;break;
        case 3:val=256;break;
        case 4:val=389;break;
        case 5:val=206;break;
        case 6:val=682;break;
        case 7:val=327;break;
        default:
        {
            explode_bomb();
            val=0;
            break;
        }
    }
    if(val!=b)
        explode_bomb();
    return val;
}
```

就可以得到这个题的密码。

phase_4:

phase_4 函数的主要功能是读入两个 int 类型数字(确定输入的方法与 phase_3 中一样)，然后将第一个数字作为 func4 函数的一个参数调用 func4，如图：将返回的结果与 0 比较，如果结果不为 0 则引爆炸弹；如果结果为 0，再将输入的第二个数与 0 比较，如果输入第二个数不为 0 则引爆炸弹。因此可以直接确定第二个数为 0。

```
0x0000000000401048 <+60>:    callq  0x400fce <func4>
0x000000000040104d <+65>:    test   %eax,%eax
0x000000000040104f <+67>:    jne    0x401058 <phase_4+76>
0x0000000000401051 <+69>:    cmpl   $0x0,0xc(%rsp)
0x0000000000401056 <+74>:    je     0x40105d <phase_4+81>
0x0000000000401058 <+76>:    callq  0x40143a <explode_bomb>
```

同时可以由下面这两行代码

```
0x000000000040102e <+34>:    cmpl   $0xe,0x8(%rsp)
0x0000000000401033 <+39>:    jbe    0x40103a <phase_4+46>
```

确定输入的第一个数的范围是[0,14]。

接下来最重要的就是 func4 函数的实现，读取函数的反汇编代码翻译成 c 代码，就是如下:

```
int func4(int first_input,int a,int b)//第一次调用时 a=0，b=14
{
    int val=b-a;
    int x=val;
    val=(val+((unsigned)x>>31))>>1;
    x=val+a;
    if(first_input==x)
    {
        val=0;
        return val;
    }
    else if(first_input>x)
    {
        val=0;
        a=x+1;
        val=func4(first_input,a,b);
        val=2*val+1;
        return val;
    }
    else
    {
        b=x-1;
        val=func4(first_input,a,b);
        val*=2;
        return val;
```

```
    }
}
```
由这个函数就可以确定当输入的第一个数为 0，1，3，7 时输出才是 0，密码解出。

phase_5：

查看了 phase_5 的反汇编代码，其调用了 string_length 函数，并且将返回值与 6 比较，说明要输入 6 个字符。

接下来函数进入一个循环，在寄存器%eax 中储存索引值，寄存器%rbx 中储存输入字符串的地址，将输入的字符依次取出，与 0xf 做位与运算（结果最大为 15，最小为 0），再将运算结果加上 0x4024b0 存入栈中，储存的地址为%rsp+%rax+0x10。于是需要知道地址 0x4024b0 储存的是什么字符串，通过 print (char*) 0x4024b0 查看，为：

"maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"。

而做了位与运算后最大值 0x4024b0+0xf 对应的地址储存的字符是 l，最小值 0x4024b0 对应的地址为 m。

循环退出后，将前面循环处理过后得到的字符串末尾添加字符'/0'，然后将这个字符串作为调用 strings_not_equal 函数的第一参数，再将寄存器%esi 的值设置为 0x40245e 作为调用函数 strings_not_equal 的第二个参数。若最后返回的结果是不相等则炸弹爆炸，否则通过。可以知道 0x40245e 储存的是需要比较的字符串，查看后其为：" flyers"。

最后依次找到 f，l，y，e，r，s 在前一个字符串中对应的位置为 9，15，14，5，6，7。转换为二进制后分别为 1001，1111，1110，0101，0110，0111。只需要查 ASCII 码表找出第四位相对应的字符，再组合即可得到密码。

phase_6：

phase_6 是有关于链表的密码推演，可以从最开始的 phase_6 中的反汇编代码前面就知道要输入 6 个数字。

仅仅通过阅读反汇编代码再整理一下顺序，可以很容易的知道反汇编前<+121>前的代码的作用是：

1、通过两个循环来检测输入的 6 个数字是否有相同的，并检查范围是否为[1,6]的整数

2、通过一个循环将输入的数 n 设置为 7-n。

于是输入测试密码：2 3 4 5 6 1。经过上述两个过程后的栈的状态如图(1)。设输入为 k[n]。

接下来的一段反汇编代码跳转非常多，在这里用下标 list[n]表示初始时链表的连接顺序，即 list[0]->list[1]->……->list[6]。

接下来一段的反汇编：

```
<+123>:    mov    $0x0,%esi//这里%esi 存的是循环变量
<+128>:    jmp    0x401197 <phase_6+163>
<+130>:    mov     0x8(%rdx),%rdx//这里%rdx 储存的是链表表元的地址，这句话是使%rdx 储存下一个表元的地址（说明 data 占据的大小是 8 个 Byte）
<+134>:    add    $0x1,%eax//第二层循环的循环变量
<+137>:    cmp    %ecx,%eax
<+139>:    jne    0x401176 <phase_6+130>
/*第二层循环就是使%rdx 中储存 list[%ecx]的地址*/
<+141>:    jmp    0x401188 <phase_6+148>
```

```
<+143>:    mov     $0x6032d0,%edx//链表表头的地址
<+148>:    mov      %rdx,0x20(%rsp,%rsi,2)//向栈中依次添加表元的地址，每个地址占八个 Byte
<+153>:    add     $0x4,%rsi//第一层循环的循环变量的更新
<+157>:    cmp     $0x18,%rsi
<+161>:    je      0x4011ab <phase_6+183>
<+163>:    mov      (%rsp,%rsi,1),%ecx//初始化%ecx 或者更新%ecx，使%ecx 为 k[%rsi]
<+166>:    cmp     $0x1,%ecx
<+169>:    jle     0x401183 <phase_6+143>
<+171>:    mov     $0x1,%eax//这里是第二层循环的初始化
<+176>:    mov     $0x6032d0,%edx//第二层循环的初始化
<+181>:    jmp     0x401176 <phase_6+130>
```

翻译为加入 c 语法的代码就是就是:

```
for(%rsi=0;%rsi!=24;%rsi=%rsi+4)
{
    %ecx=(%rsp+%rsi)//即为 k[%rsi];
    %edx=0x6032d0;//链表表头地址
    if(%ecx>1)
    {
        %eax=1;
        do{
            %rdx=(%rdx+0x8);
            %eax=%eax+1;
        }while(%eax!=%ecx);
    }
    (%rsp+2*%rsi+0x20)=%rdx;//放入栈中
}
```

于是这段代码的意思是，将 list[k[0]]、list[k[1]]、list[k[2]]、list[k[3]]、list[k[4]]、list[k[5]]依次放入栈中，在栈中的地址依次为%rsp+0x20、%rsp+0x28、%rsp+0x30、%rsp+0x38、%rsp+0x40、%rsp+0x48。

接下来的代码为:

```
<+183>:mov      0x20(%rsp),%rbx//设置%rbx 指向第一个储存的表元，即%rbx=&list[k[0]]
<+188>:lea      0x28(%rsp),%rax// 设置%rax 储存第二个表元的栈的地址，即(%rax)=&list[k[1]]
<+193>:lea      0x50(%rsp),%rsi//设置末尾，循环中值量
<+198>:mov      %rbx,%rcx//拷贝%rbx 中的值到%rcx 中
/*值得注意的是，(%rbx+0x8)是取下一表元的地址，而(%rax+0x8)是取这个栈元素的前一个栈元素的内容（区别在于%rbx 指向表元，%rax 指向栈的一个元素）*/
<+201>:mov      (%rax),%rdx//设置%rbx 为%rax 指向元素中的内容
<+204>:mov      %rdx,0x8(%rcx)//设置%rcx 的 link 部分的值为%rbx 中的内容
<+208>:add      $0x8,%rax//使%rax 指向下一个栈元素
<+212>:cmp      %rsi,%rax
```

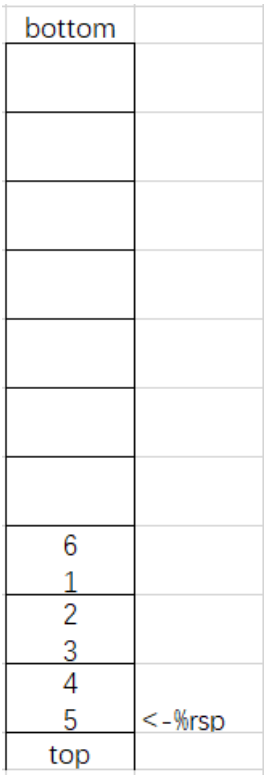
<+215>:je 0x4011d2 <phase_6+222>

<+217>:mov %rdx,%rcx//更新%rcx，意思相当于 a=a->link（a 指%rax,a->link 指%rdx）

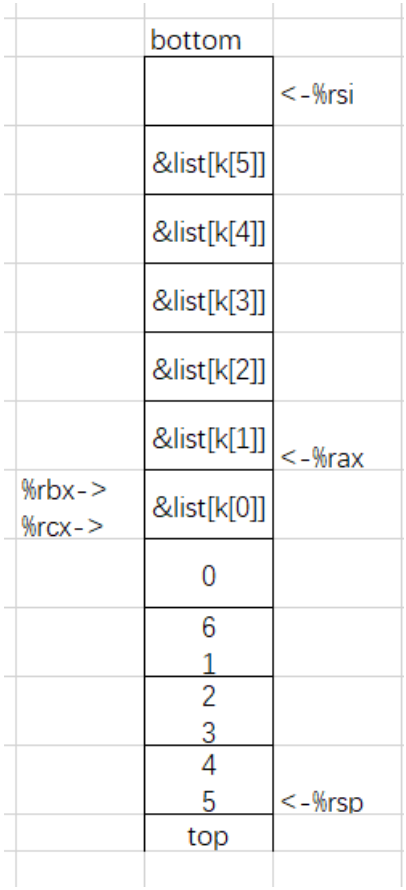
<+220>:jmp 0x4011bd <phase_6+201>

/*上述循环的意思是，使链表的序列由 list[0]->list[1]->……->list[5]变为 list[k[0]]-> list[k[1]] ->……-> list[k[3]]*/

最后一部分汇编就是要求链表的 list[k[0]]> list[k[1]]> list[k[2]]> list[k[3]]> list[k[4]]> list[k[5]]。由于输入的密码为 2 3 4 5 6 1，后来添加到栈中的地址依次为&list[5]，&list[4]，&list[3]，&list[2]，&list[1]，&list[6]，依次查值，分别为 477，691，924，168，332，443，因此有 list[3]> list[4]> list[5]> list[6]> list[1]> list[2]，在做 n=7-n 的变换即可得最后密码为 4 3 2 1 6 5。



图（1）



图（2）

secret_phase：

在主函数中没有发现任何对 secret_phase 函数的调用，而 phase_1 到 phase_6 里面也没有涉及，而主函数中还调用了 phase_defused 函数，因此去查看其反汇编代码，找到了对 secret_phase 函数的调用。在 phase_defused 函数中，只有解锁 phase_6 才可能开启 secret_phase。之后会调用 scanf 函数，查看其参数发现为“%d %d %s”，即输入两个数字一个字符串，但是执行过程中却没有任何输入机会。

```
0x00000000004015f0 <+44>:    mov    $0x402619,%esi
0x00000000004015f5 <+49>:    mov    $0x603870,%edi
0x00000000004015fa <+54>:    callq  0x400bf0 <__isoc99_sscanf@plt>
0x00000000004015ff <+59>:    cmp    $0x3,%eax
```

（0x402619 储存的即使字符串"%d %d %s"）

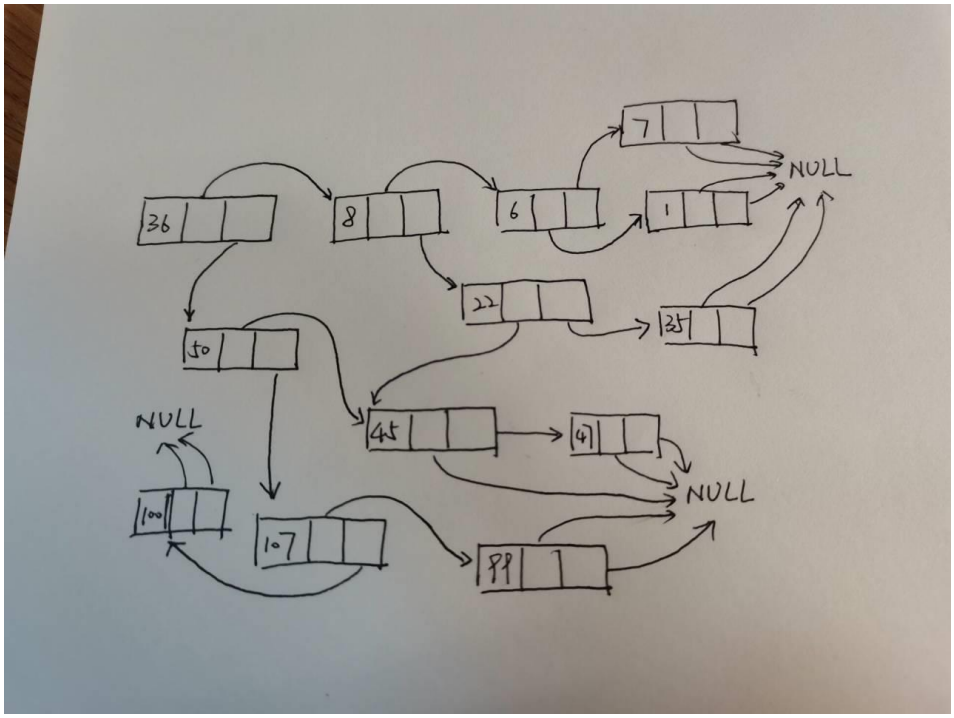
而地址 0x603870 的出现很奇怪，查看储存的内容，为"0 0"，我想到我 phase_4 输入的答案是 0 0，重新执行一遍并且将 phase_4 的答案换成 3 0，此时这个地址储存的内容变成了"3 0"，于是重新执行，并在 phase_4 的密码后面添加了一个字符串，查看内容发现这个地址也储存了这个字符串。继续往下，调用了 strings_not_equal 函数，对比的字符串是"DrEvil"，将 phase_4 后面的字符串修改成"DrEvil"，便开启的 secret_phase。

而 secret_phase 函数的内容很简单，就是调用 read_line 函数读取一个字符串，然后调用 strtol 函数将这个字符串转换成数字，并判断这个数字的大小是否为[1，1001]，再调用 fun7 函数（第一个参数为前面的得到的数字，第二个参数为一个地址），将返回值与 2 比较，如果返回值等于 2 即通过。

因此解密关键就是 fun7 函数，查看其内部反汇编，很容易发现传入的第二个参数是一个地址，是指向一个链表的表元，每个表元含有两个后继指针（下图的 next1 与 next2）。而且 fun7 还是一个递归函数。左图是对其的翻译。

```
int fun7(int input, node* p)
{
    if (p == 0)
        return -1;
    int val;
    if (input >= p->data)
    {
        val = 0;
        if (p->data == input)
            return val;
        p = p->next2;
        val = fun7(input, p);
        val = 2 * val + 1;
    }
    else
    {
        p = p->next1;
        val = fun7(input, p);
        val = 2 * val;
    }
    return val;
}
```

通过对每个链表表元的地址追踪与查找值，得到了他们的关系如下图。再通过编写代码，将 1 到 1001 的整数代入检查返回值是否为 2 即可找到最后的谜底：22。



reference：

https://baike.baidu.com/item/ASCII/309296?fromtitle=ascii%E7%A0%81%E8%A1%A8&fromid=19660475&fr=aladdin

https://m.jb51.net/article/71463.htm