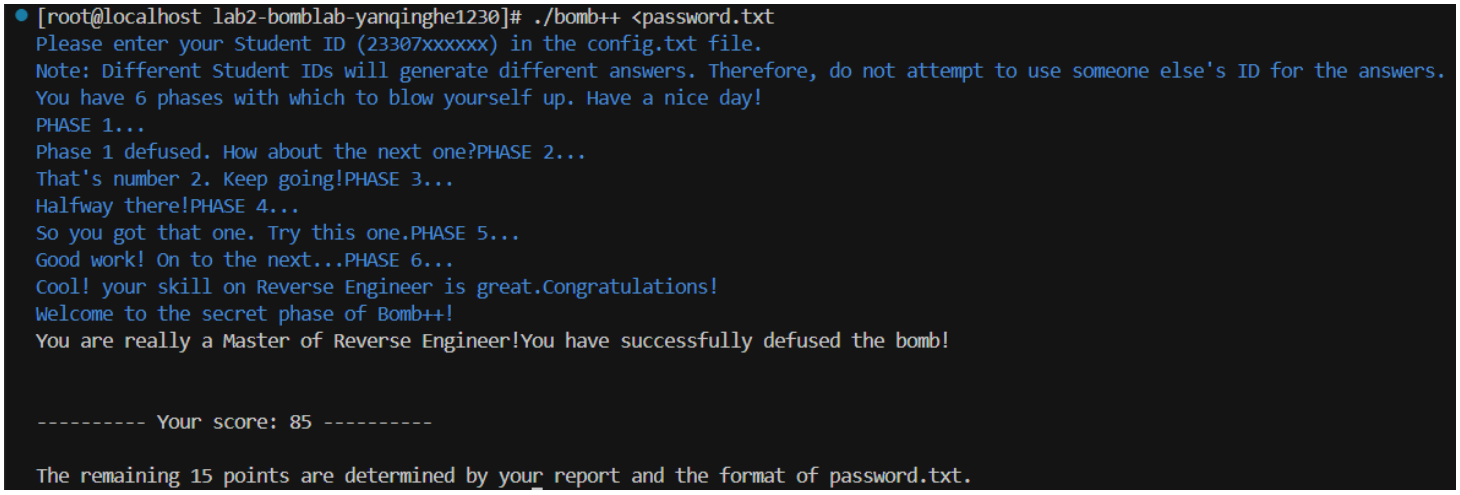
Bomblab Report

[redacted]

拆弹成功截图



注：在实验中许多变量的值与ID的hash值有关，故在本报告中，为了便于说明，将所有相关值命名为 tag

Phase_1

```
0x000000000008182c <+90>:    mov    -0x8(%rbp),%rdx
0x0000000000081830 <+94>:    mov    -0x18(%rbp),%rax
0x0000000000081834 <+98>:    mov    %rdx,%rsi
0x0000000000081837 <+101>:   mov    %rax,%rdi
0x000000000008183a <+104>:   call   0x8176f <string_not_equal>
0x000000000008183f <+109>:   xor    $0x1,%eax
0x0000000000081842 <+112>:   test   %al,%al
0x0000000000081844 <+114>:   je     0x8184b <phase_1+121>
0x0000000000081846 <+116>:   call   0x7de1f <explode_bomb>
```

不难发现，phase_1中控制炸弹引爆的核心在于 `string_not_equal` 函数，若函数返回值为 `1` ，则在与 `0x1` 进行异或操作后得到 `0` ，从而在 `test` 操作后将ZF置于 `1` ， `je` 语句满足条件跳过炸弹。
而 `-0x18(%rbp)` 中存放的是输入，作为参数之一传入 `string_not_equal` 函数中。因此只需要在调用该函数处打断点，查看此时 `%rsi` 存放的地址指向的字符串，便可以得到答案： `AI's unchecked growth risks losing human control`

Phase_2

首先我们需要确定输入格式：

```
0x00005555555d58d6 <+28>:    lea    -0x40(%rbp),%rdx
0x00005555555d58da <+32>:    mov    -0x48(%rbp),%rax
0x00005555555d58de <+36>:    mov    %rdx,%rsi
0x00005555555d58e1 <+39>:    mov    %rax,%rdi
0x00005555555d58e4 <+42>:    call   0x5555555d27d4 <read_six_numbers>
0x00005555555d58e9 <+47>:    lea    -0x40(%rbp),%rax
0x00005555555d58ed <+51>:    mov    %rax,-0x10(%rbp)
```

发现函数调用了名为 `read_six_numbers` 的函数，猜测应该输入六个数字。这六个数字组成的数组首地址最终被存储在 `-0x10(%rbp)` 处。

继续查看该函数：

```
0x00005555555d281a <+70>:    lea    0x2cde06(%rip),%rsi    # 0x5555558a0627
0x00005555555d2821 <+77>:    mov    %rax,%rdi
0x00005555555d2824 <+80>:    mov    $0x0,%eax
0x00005555555d2829 <+85>:    call   0x5555555c9e80 <__isoc99_sscanf@plt>
```

根据 `scanf` 函数的签名，传入函数的第二个参数指定输入格式控制。因此在调用函数处打断点查看 `%rsi` 存放的值，发现其指向 `%d %d %d %d %d %d` ，故应该输入六个十进制整数。

接下来函数进入了一个循环：循环变量存放于 `-0x4(%rbp)` ，每次循环结束后加1，直到大于5时跳出循环。

```
0x00005555555d58f1 <+55>:    movl   $0x1,-0x4(%rbp) #为循环变量赋初值1
0x00005555555d58f8 <+62>:    jmp    0x5555555d5942 <phase_2+136>
...
0x00005555555d593e <+132>:   addl   $0x1,-0x4(%rbp)
0x00005555555d5942 <+136>:   cmpl   $0x5,-0x4(%rbp)
0x00005555555d5946 <+140>:   jle    0x5555555d58fa <phase_2+64>
```

循环中具体进行的操作如下：

1. 通过将循环变量左移两位，实现乘4（这是因为每个int类型的数据用4个字节存储）。将该值和数组首元素地址相加，实现通过循环遍历每个数组元素。

```
0x00005555555d58fa <+64>:    mov    -0x28(%rbp),%edx
0x00005555555d58fd <+67>:    mov    -0x4(%rbp),%eax
0x00005555555d5900 <+70>:    cltq
0x00005555555d5902 <+72>:    shl    $0x2,%rax
0x00005555555d5906 <+76>:    lea    -0x4(%rax),%rcx
0x00005555555d590a <+80>:    mov    -0x10(%rbp),%rax
0x00005555555d590e <+84>:    add    %rcx,%rax
0x00005555555d5911 <+87>:    mov    (%rax),%eax  #得到当前数组元素
```

2. 将当前数组元素和 `-0x28(%rbp)` 中存储的值相乘，再加上 `-0x24(%rbp)` 中存放的值，并将结果存放在 `-0x14(%rbp)`

```
0x00005555555d5913 <+89>:    imul   %eax,%edx
0x00005555555d5916 <+92>:    mov    -0x24(%rbp),%eax
0x00005555555d5919 <+95>:    add    %edx,%eax
0x00005555555d591b <+97>:    mov    %eax,-0x14(%rbp)
```

3. 将下一个数组元素和计算结果相比较，若不相等则引爆炸弹

```
0x00005555555d591e <+100>:   mov    -0x4(%rbp),%eax
0x00005555555d5923 <+105>:   lea    0x0(,%rax,4),%rdx
0x00005555555d592b <+113>:   mov    -0x10(%rbp),%rax
0x00005555555d592f <+117>:   add    %rdx,%rax
0x00005555555d5932 <+120>:   mov    (%rax),%eax #得到下一个数组元素
0x00005555555d5934 <+122>:   cmp    %eax,-0x14(%rbp)
0x00005555555d5937 <+125>:   je     0x5555555d593e <phase_2+132>
0x00005555555d5939 <+127>:   call   0x5555555d1e1f <explode_bomb>
```

在计算处打断点，可以看到 `-0x28(%rbp)` 中为 `-10`，`-0x24(%rbp)` 中为 `3` 。
该循环用于验证所输入的六个整数是否满足一定的递推关系： `a[i+1]=(-10)*a[i]+3` ，对于首个数字并没有要求。
为方便计算，输入： `1 -7 73 -727 7273 -72727`

Phase_3

同样先查看输入格式： `%d %d %c` ，需要输入两个十进制整数以及一个字符，分别存放在： `-0x15(%rbp)` 、 `-0x14(%rbp)` 、 `-0x10(%rbp)`

速览该函数后发现，根据第一个输入值，函数会跳转到不同的地址，再对后续输入进行比对，即switch语句。

而需要跳转到哪一个case，则取决于 tag 值。

```
0x00005555555d595c <+16>:    mov    0x42302e(%rip),%eax        # 0x5555559f8990 <ID_hash>
0x00005555555d5962 <+22>:    and    $0x7,%eax
0x00005555555d5965 <+25>:    mov    %eax,-0x8(%rbp)
```

tag 即存放在 -0x8(%rbp) 处，打断点查看该值为 2 ， 对应的分支语句如下：

```
0x00005555555d5a53 <+263>:   movb   $0x65,-0x1(%rbp)
0x00005555555d5a57 <+267>:   mov    -0x14(%rbp),%eax
0x00005555555d5a5a <+270>:   cmp    $0x8,%eax
0x00005555555d5a5d <+273>:   jne    0x5555555d5a69 <phase_3+285>
0x00005555555d5a5f <+275>:   cmpl   $0x2,-0x8(%rbp)
...
0x00005555555d5b17 <+459>:   movzbl -0x15(%rbp),%eax
0x00005555555d5b1b <+463>:   cmp    %al,-0x1(%rbp)
0x00005555555d5b1e <+466>:   je     0x5555555d5b25 <phase_3+473>
0x00005555555d5b20 <+468>:   call   0x5555555d1e1f <explode_bomb>
```

因此第二个输入值为 0x8 ， 第三个输入为ASCII码中对应 0x65 的字符： e 。
再次阅读函数，发现该分支不能直接跳转得到，而是经过如下计算：

```
0x00005555555d59e2 <+150>:   mov    %eax,%eax
0x00005555555d59e4 <+152>:   lea    0x0(,%rax,4),%rdx
0x00005555555d59ec <+160>:   lea    0x2cad69(%rip),%rax        # 0x5555558a075c
0x00005555555d59f3 <+167>:   mov    (%rdx,%rax,1),%eax
0x00005555555d59f6 <+170>:   cltq
0x00005555555d59f8 <+172>:   lea    0x2cad5d(%rip),%rdx        # 0x5555558a075c
0x00005555555d59ff <+179>:   add    %rdx,%rax
0x00005555555d5a02 <+182>:   notrack jmp *%rax
```

由前序程序得知第一个输入值的范围是 (0,0x22] ， 而该输入值会决定最终 %rdx 将指向哪一个整数，该整数与 %rax 中存放的地址相加，得到最终跳转的目标地址。
经过尝试，发现第一个输入为 5 的时候能跳转到所需要的分支。
最终得到答案： 5 8 e

Phase_4

首先确定输入格式： %lld ， 需要输入一个六十四位的十进制整数，存放在 -0x10(%rbp)

通过算数右移，该函数将输入整数的高位和低位分别存放在 `-0x4(%rbp)` 、 `-0x8(%rbp)`

```
0x00005555555d5bae <+62>:    mov    -0x10(%rbp),%rax
0x00005555555d5bb2 <+66>:    sar    $0x20,%rax
0x00005555555d5bb6 <+70>:    mov    %eax,-0x4(%rbp)
0x00005555555d5bb9 <+73>:    mov    -0x10(%rbp),%rax
0x00005555555d5bbd <+77>:    mov    %eax,-0x8(%rbp)
```

接下来进行了一系列比较，限制了高位和低位的范围： `(0,0xa]` ,再将高位传入了 `_ZL3CIEi` 进行递归运算：

```
0x00005555555d5b34 <+12>:    mov    %edi,-0x14(%rbp)
0x00005555555d5b37 <+15>:    cmpl   $0x0,-0x14(%rbp)
0x00005555555d5b3b <+19>:    jne    0x5555555d5b44 <_ZL3CIEi+28>
0x00005555555d5b3d <+21>:    mov    $0x1,%eax
0x00005555555d5b42 <+26>:    jmp    0x5555555d5b6e <_ZL3CIEi+70>
...
0x00005555555d5b6e <+70>:    leave
```

递归完成的条件：传入的参数为0，此时返回1
若不为0，则右移一位再调用函数

```
0x00005555555d5b44 <+28>:    mov    -0x14(%rbp),%eax
0x00005555555d5b47 <+31>:    sar    $1,%eax
0x00005555555d5b49 <+33>:    mov    %eax,%edi
0x00005555555d5b4b <+35>:    call   0x5555555d5b28 <_ZL3CIEi>
```

对于函数返回值进行如下操作：
将当前参数和 `0x1` 按位与，若结果是0（即当前参数末位为0），则将返回值平方。反之，将返回值平方并左移三位。

```
0x00005555555d5b50 <+40>:    mov    %eax,-0x4(%rbp)
0x00005555555d5b53 <+43>:    mov    -0x14(%rbp),%eax
0x00005555555d5b56 <+46>:    and    $0x1,%eax
0x00005555555d5b59 <+49>:    test   %eax,%eax
0x00005555555d5b5b <+51>:    je     0x5555555d5b68 <_ZL3CIEi+64>
0x00005555555d5b5d <+53>:    mov    -0x4(%rbp),%eax
0x00005555555d5b60 <+56>:    imul   %eax,%eax
0x00005555555d5b63 <+59>:    shl    $0x3,%eax
0x00005555555d5b66 <+62>:    jmp    0x5555555d5b6e <_ZL3CIEi+70>
0x00005555555d5b68 <+64>:    mov    -0x4(%rbp),%eax
0x00005555555d5b6b <+67>:    imul   %eax,%eax
```

而炸弹不引爆的条件为：

```
0x00005555555d5bfe <+142>:   cmp    $0x40000000,%eax
```

综合下来，高位为 `0xa` ，低位无限制。

Phase_5

首先确定输入格式： `%s %d %u` ，需要输入一个字符串，一个十进制整数，和一个无符号整数，分别存放在 `-0x27(%rbp)` 、 `-0x2c(%rbp)` 、 `-0x30(%rbp)`
速览函数发现总共进行了三处 `strcmp` ，打断点查看发现分别是：behavior，ethics和growth。故所需要输入的字符串在这三者之中。

尝试输入behavior，进入对应的regulator函数中。
该函数总共执行了以下功能：

1. 确定第269行跳转的地址
2. 给某个检验变量赋值

```
0x00005555555d5d13 <+257>:   mov    -0x2c(%rbp),%edx
0x00005555555d5d16 <+260>:   mov    -0x18(%rbp),%rax
0x00005555555d5d1a <+264>:   mov    %edx,%esi
0x00005555555d5d1c <+266>:   mov    %rax,%rdi
0x00005555555d5d1f <+269>:   call   *%rcx
```

而在第269行跳转到的函数中，进行了第二个输入值的比对。

接下来在 `_ZN11AIRegulator18is_phase5_passableEj` 中，要求刚刚赋值的检验变量大于 `0x4a` 。经过尝试，发现只有growth对应的regulator函数中给检验变量的赋值满足条件。
同时，在 `_ZN11AIRegulator18is_phase5_passableEj` 中可以确定第三个输入的值，该值为 `tag` 值。
最后得到答案为 `growth 2034 1858`

Phase_6

首先确定输入格式：发现函数中再次调用了 `read_six_numbers` ， 故需要输入六个十进制整数，数组首地址存放在 `-0x18(%rbp)` 中。
接下来进入了一个循环：

```
0x00005555555d5ed6 <+49>:    movl   $0x0,-0x4(%rbp)
0x00005555555d5edd <+56>:    jmp    0x5555555d5f1d <phase_6+120>
...
0x00005555555d5ef5 <+80>:    cmp    $0x6,%eax
0x00005555555d5ef8 <+83>:    jg     0x5555555d5f14 <phase_6+111>
...
0x00005555555d5f10 <+107>:   test   %eax,%eax
0x00005555555d5f12 <+109>:   jns    0x5555555d5f19 <phase_6+116>
0x00005555555d5f14 <+111>:   call   0x5555555d1e1f <explode_bomb>
0x00005555555d5f19 <+116>:   addl   $0x1,-0x4(%rbp)
0x00005555555d5f1d <+120>:   cmpl   $0x5,-0x4(%rbp)
0x00005555555d5f21 <+124>:   jle    0x5555555d5edf <phase_6+58>
```

该循环起到的作用是遍历数组，检查每个输入值是否在 `(0,6]` 的范围内。

接下来，函数给存放在 `-0x5(%rbp)` 的变量赋值，继续阅读发现该变量与下一个循环中调用的函数的返回值有关，且不引爆炸弹的条件为该变量值为 `1` 。将该变量命名为 `flag` 。

然后调用了 `build_stack` 函数建栈（？），发现z `stackbottom` 值为4，并将值为2的 `initialnodes` 压入栈中。

下一个循环中的核心是调用了 `maintain_monotonic_sequence` 函数。
在该函数中总共进行了三次比较：

1. 比较当前数组元素和栈底元素 `stackbottom` ， 若数组元素小于栈底元素，则不进行任何操作直接返回 `0`
2. 比较 `initialnodes` 和 `stackbottom` ， 若两者不相等则执行 `stack_pop` ， 将 `initialnodes` 退栈。这是为了在接下来能改变 `stackbottom` 的值（？）
3. 比较当前数组元素和栈顶元素。由 `2` 中的操作这里的比较对象一定为 `stackbottom` ， 由 `1` 中的操作数组元素一定不小于 `stackbottom` ， 故一定会执行 `stack_push` 。

在该函数中， stackbottom 被重新赋值为当前数组元素，并再次将 initialnodes 压入栈中。

调用函数完成后，会对 flag 值和新的返回值进行 and 操作，若返回值为0，则 flag 将被赋值为0，反之则为一。因此，只要某次循环中返回值为0， flag 将一直为0。
而返回值为0的条件是：当前数组元素小于栈底元素。因此，需要输入的第一个数字需要大于等于最初的 stackbottom ， 而数组需要满足单调递增关系。
输入为 4 4 5 5 6 6 ，符合条件。

Secret_Phase

触发条件

仔细阅读 main.cpp ， 发现 secret_phase 在一个 if 语句中被调用，因此目标是通过某种输入使 secret_key 的值被改变。
而在每次调用 phase 函数前，都会先调用 read_line 函数。阅读该函数发现，它起到的作用是给 input 规定一个字符数上限，对于超出该上限的输入不作额外操作，故它将溢出到之后定义的 secret_key 中。
因此，在某一行输入后输入一系列空格直到超出字符数上限即可。

Secret_Phase

该函数的主体是以下的一个循环，循环变量存放在 -0x8(%rbp) 中，用来遍历输入。

```
0x00005555555d5fa2 <+23>:    movl   $0x0,-0x8(%rbp)
0x00005555555d5fa9 <+30>:    jmp    0x5555555d6121 <secret_phase+406>
...
0x00005555555d611d <+402>:   addl   $0x1,-0x8(%rbp)
0x00005555555d6121 <+406>:   cmpl   $0x2,-0x8(%rbp)
0x00005555555d6125 <+410>:   jle    0x5555555d5fae <secret_phase+35>
```

遍历时，每次只移动一个字节，故输入应该为字符。结合循环的终止条件，可知输入为三个字符。

```
0x00005555555d5fae <+35>:    mov    -0x8(%rbp),%eax
0x00005555555d5fb1 <+38>:    movslq %eax,%rdx
0x00005555555d5fb4 <+41>:    mov    -0x18(%rbp),%rax
0x00005555555d5fb8 <+45>:    add    %rdx,%rax
0x00005555555d5fbb <+48>:    movzbl (%rax),%eax
```

在循环中，首先检查输入字符的ASCII码是否满足以下条件：

1. 范围在 [0x41,0x4a] 之间

2. 奇数

接下来是以 `-0x4(%rbp)` 中存放的值为输入的一系列if语句进行条件判断，将该值称为 `flag`

1. 值为0时
    i. 当前字符为0x43：将 `flag` 赋值为1
    ii. 当前字符为0x41、0x47：将 `flag` 赋值为4
2. 值为1时
    i. 当前字符为0x49：将 `flag` 赋值为2
    ii. 当前字符为0x43、0x45：将 `flag` 赋值为4
3. 值为2时
    i. 当前字符为0x45：将 `flag` 赋值为3
    ii. 当前字符为0x49：将 `flag` 赋值为2

循环完成后，需要 `flag` 值为3才能不引爆炸弹。
综合以上可以得出，输入字符的ASCII码分别为：0x43，0x49，0x45。对应的字符为： `CIE`