

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**实验名称： 机器级语言理解**

**院 系 ：计算机科学与技术**

**专业班级 ： CS2306**

**学 号 ： U202315696**

**姓 名 ： 彭冲**

**指导教师 ： 朱虹**

**2025 年 4 月 10 日**

**一、实验目的与要求**

通过逆向分析一个二进制程序（称为“二进制炸弹”）的构成和运行逻辑，加深对理论课中关于程序的机器级表示各方面知识点的理解，增强反汇编、跟踪、分析、调试等能力。

实验环境：Ubuntu，GCC，GDB等

**二、实验内容**

**任务1** 二进制炸弹拆除

作为实验目标的二进制炸弹（binary bombs）可执行程序由多个“关”组成。每一个“关”（阶段）要求输入一个特定字符串，如果输入满足程序代码的要求，该阶段即通过，否则程序输出失败。实验的目标是设法得到得出解除尽可能多阶段的字符串。

为了完成二进制炸弹的拆除任务，需要通过反汇编和分析跟踪程序每一阶段的机器代码，从中定位和理解程序的主要执行逻辑，包括关键指令、控制结构和相关数据变量等等，进而推断拆除炸弹所需要的目标字符串。

实验源程序及相关文件 bomb.rar

bomb.c 主程序

phases.o 各个阶段的目标程序

support.c 完成辅助功能的目标程序

support.h 公共头文件

**阶段1： 串比较 phase\_1(char \*input);**

要求输出的字符串(input) 与程序中内置的某一特定字符串相同。提示：找到与input串相比较的特定串的地址，查看相应单元中的内容，从而确定input 应输入的串。

**阶段2：循环 phase\_2(char \*input);**

要求在一行上输入 6个整数数据，与程序自动产生的 6个数据进行比较，若一致，则过关。提示：将输入串input拆分成 6个数据由函数 read\_six\_numbers(input, numbers) 完成。之后是各个数据与自动产生的数据的比较，在比较中使用了循环语句。

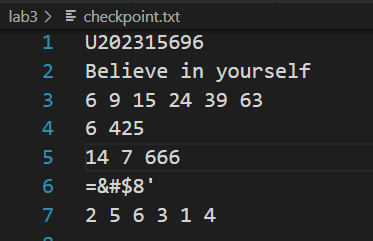
**阶段3：条件分支 phase\_3(char \*input);**

要求输入两个整数数据，与程序中给定的数据比较，相等则过关。提示：在自动生成数据时，使用了 switch … case 语句。

**三、实验记录及问题回答**

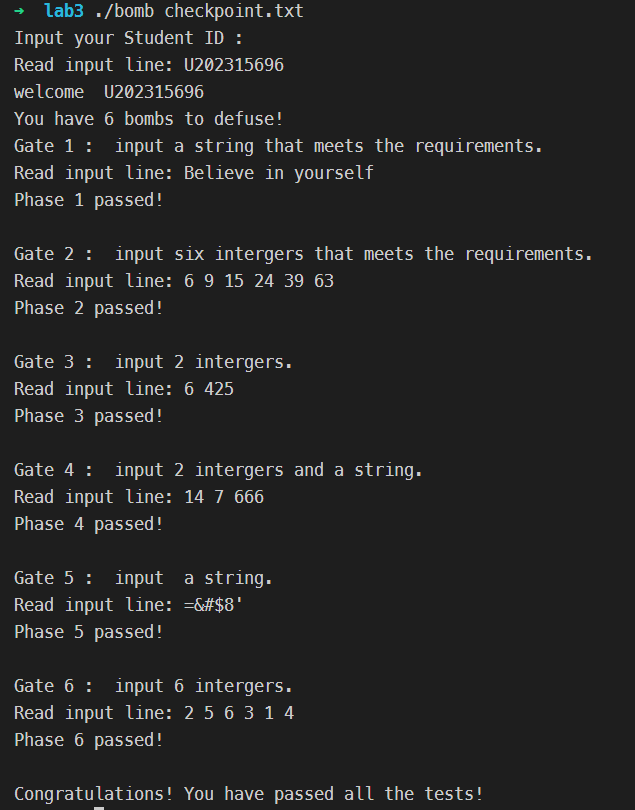
**（1）实验任务1 的实验记录**

**输入：**



图表 1 bomb程序从文件输入

**输出：**



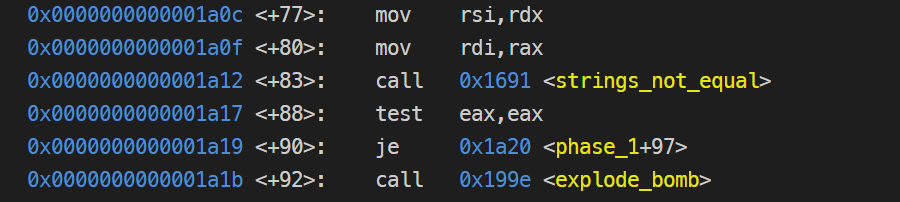
图表 2 bomb程序通关输出

**（2）拆除炸弹的过程中关键操作**

首先，由任务书可知，phases.o文件的编译未启用-g选项，因此不包含调试符号，无法在各phase函数内直接添加断点，需另寻他法。

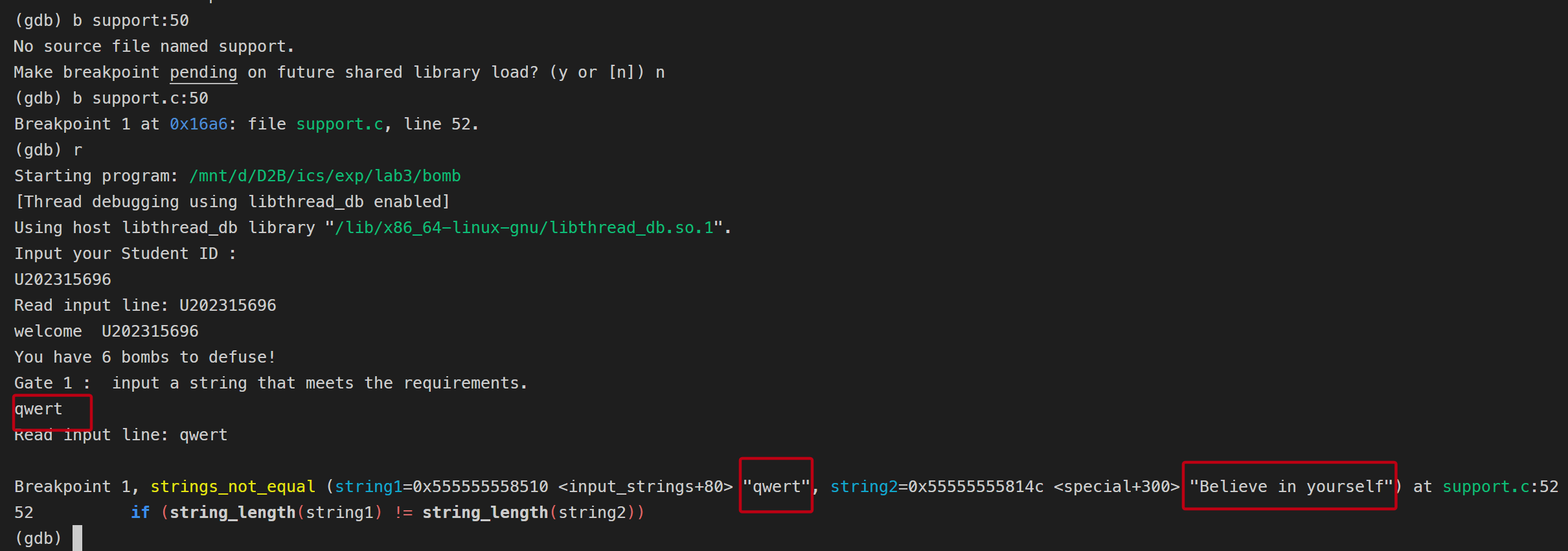
1. **串比较 phase\_1(char \*input)**
   1. **动态调试跟踪法**

运行gdb，对phase\_1函数反汇编，观察到调用函数string\_not\_equal

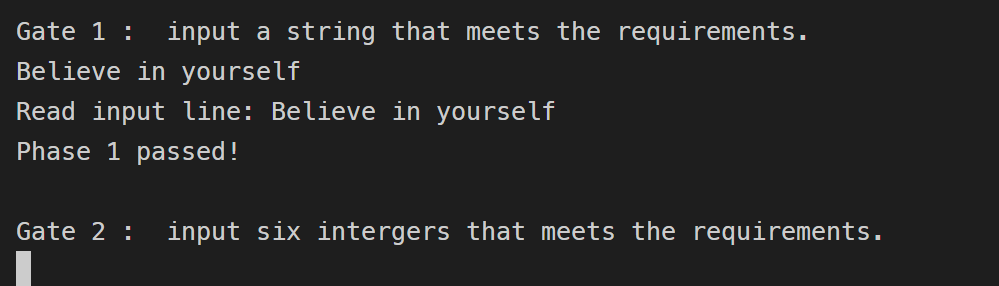


猜测string\_not\_equal比较的是输入串和目标串。

因此，在string\_not\_equal函数内设置断点，并运行到此处：

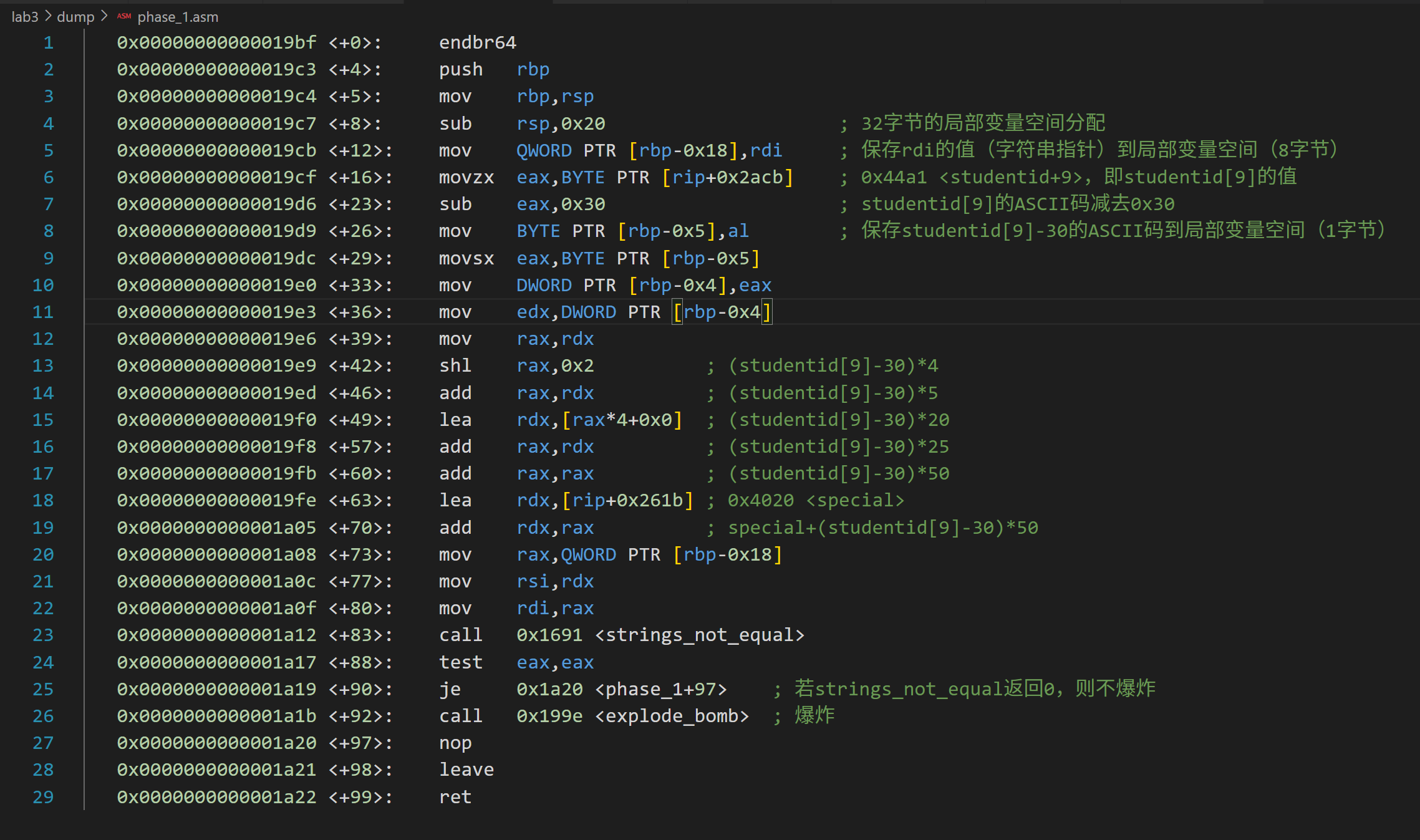


可以看到，形参string1为手动输入的字符串，而string2为Believe in yourself，猜测此串为目标字符串，经验证成立。



* 1. **反汇编静态分析法**

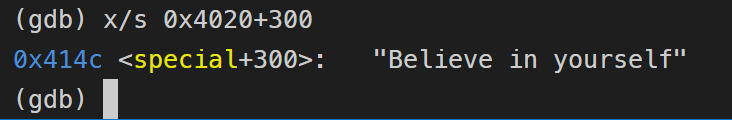
分析函数phase\_1的反汇编代码（分析见下图中代码注释）



分析可得，传入string\_not\_equal的两个串的地址：

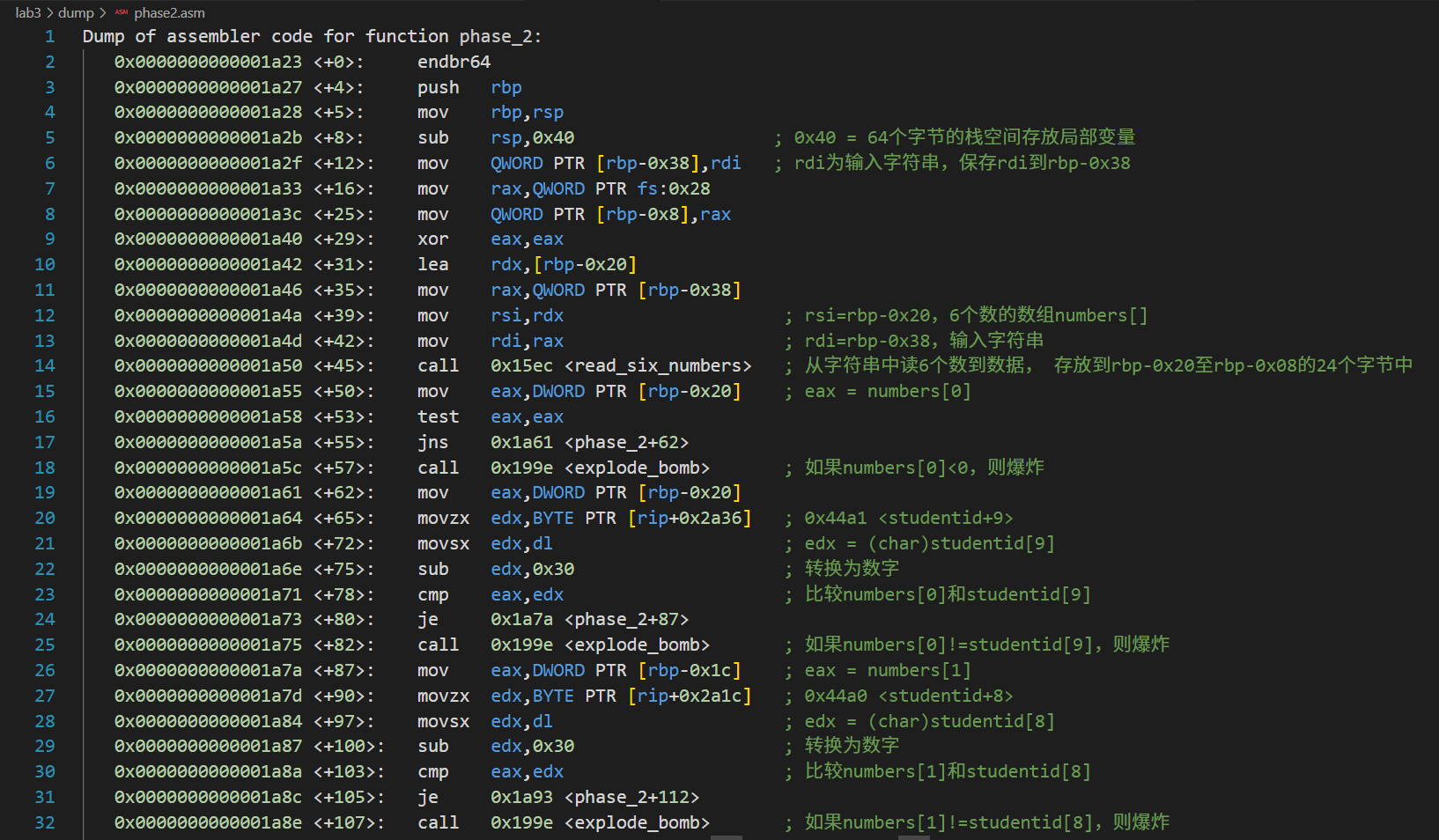
* + 1. rdi = QWORD PTR[rbp-18]为传入phase\_1的串，也即输入的串；
    2. rsi = special + (studentid[9]-30)\*50 = special+300 = 0x4020+300

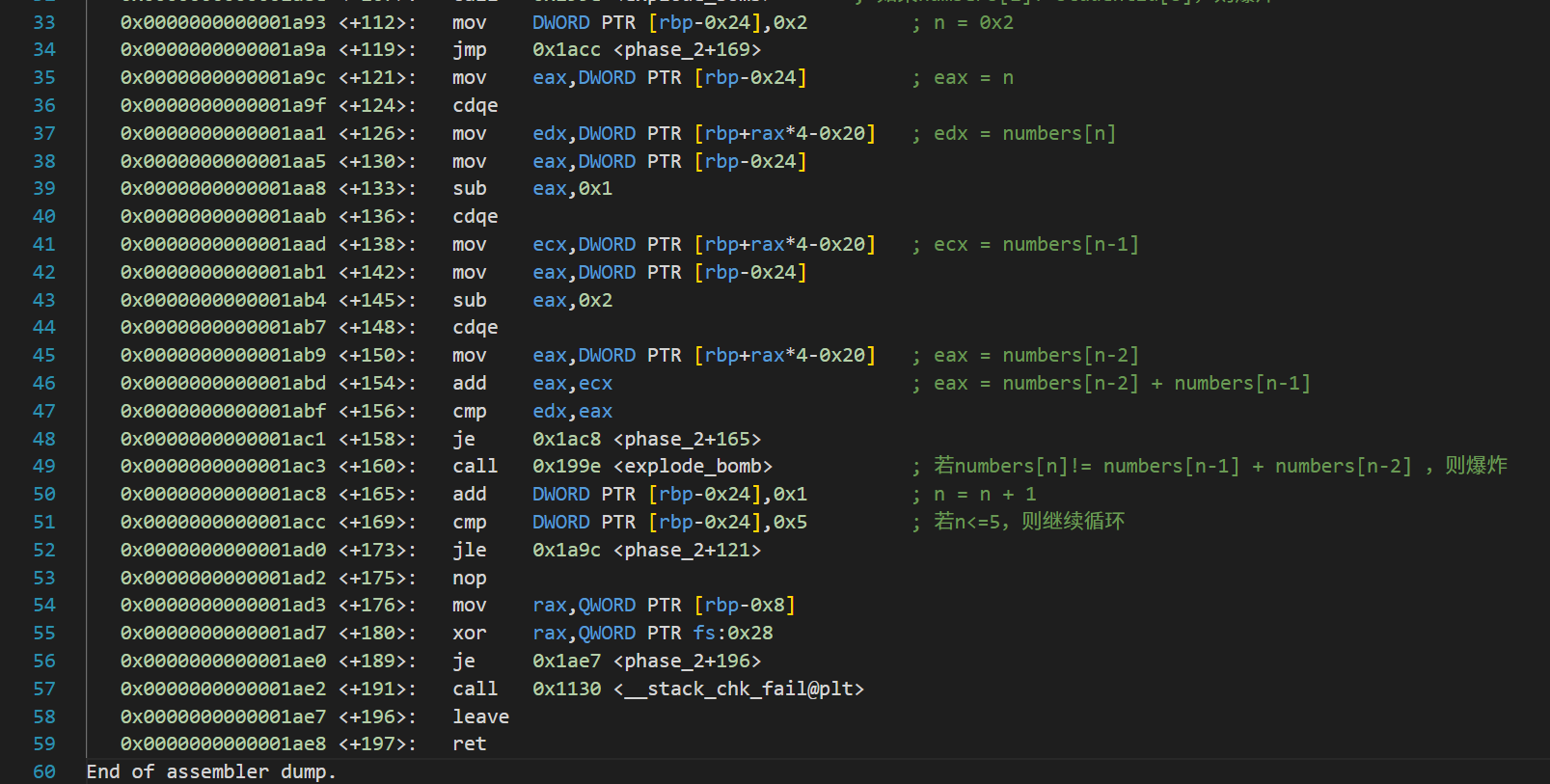
gdb中使用x/s命令查看该处，同样得到答案为Believe in yourself。



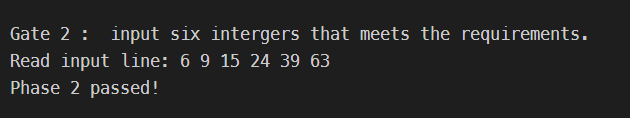
1. **循环 phase\_2(char \*input)**
   1. **反汇编静态分析法**

分析见下图注释。



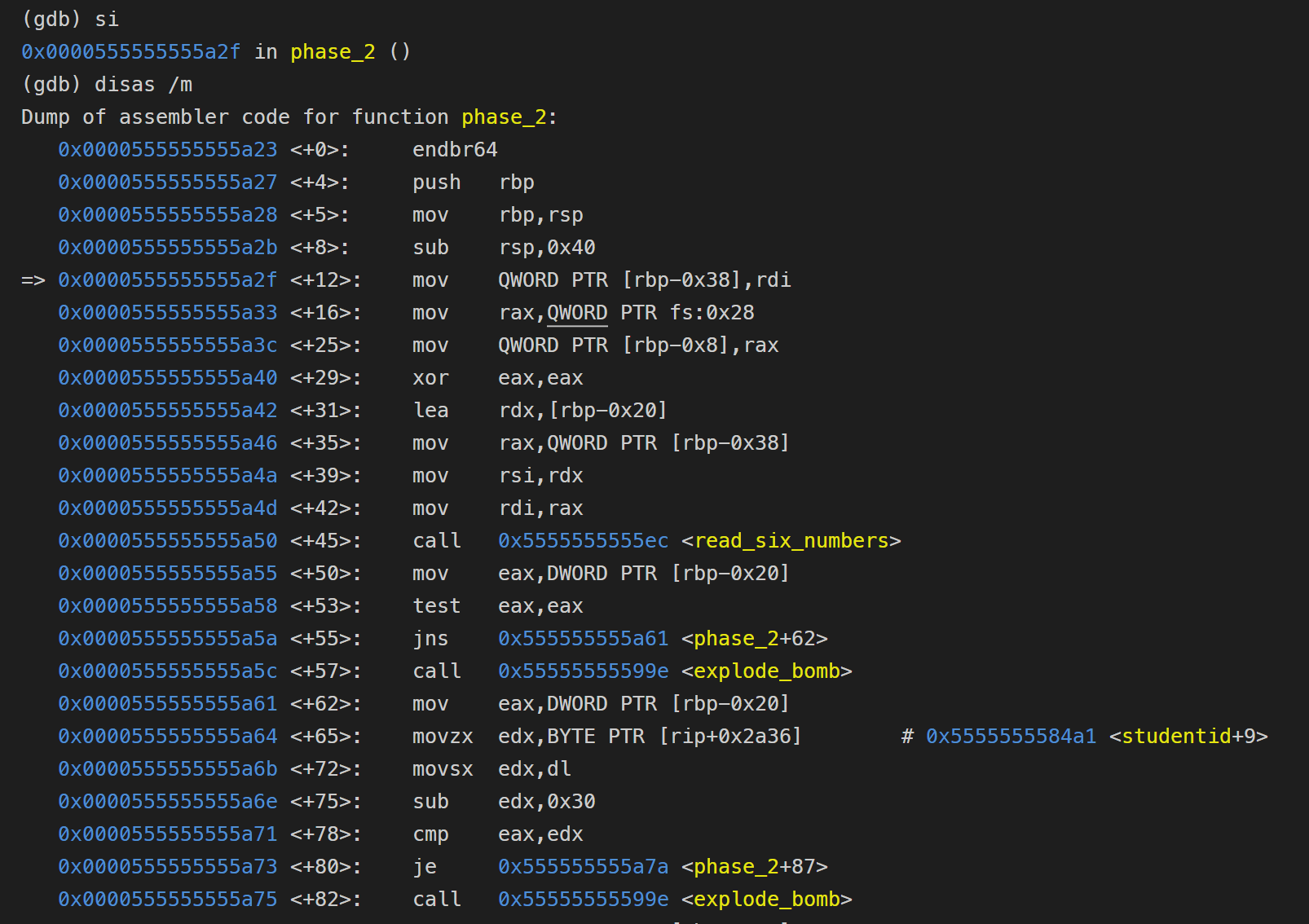


可知，前两个数分别为studentid[9]和studentid[8]，此后为4次循环，通过类似斐波那契数列的算法，由前两个数的和可递推出下一个数。则答案为6 9 15 24 39 63，经验证正确。

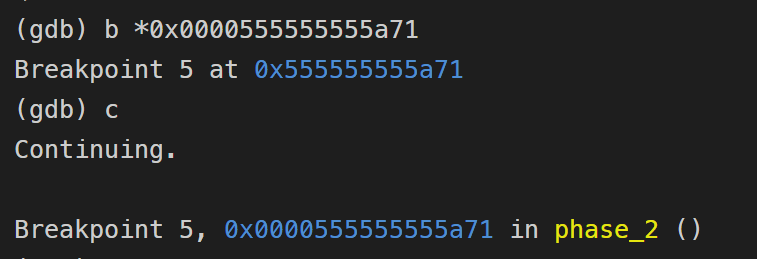


* 1. **动态调试法**

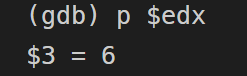
通过stepi命令步入phase\_2函数体，再使用disassemble /m查看反汇编以及当前位置：



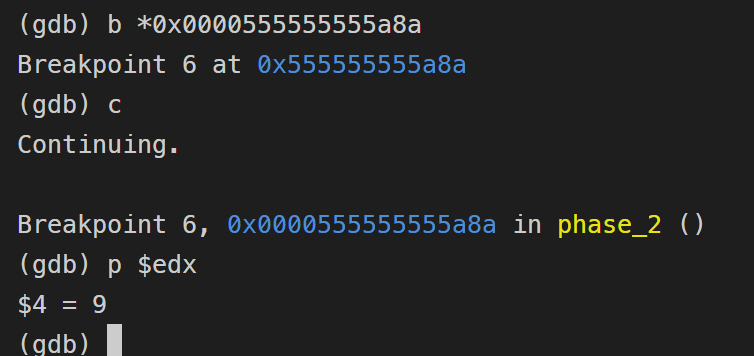
上图中，0x0000555555555a71处调用cmp指令，且会跳转到explode\_bomb，则使用break命令在此处打断点，并使用continue指令运行到此处。



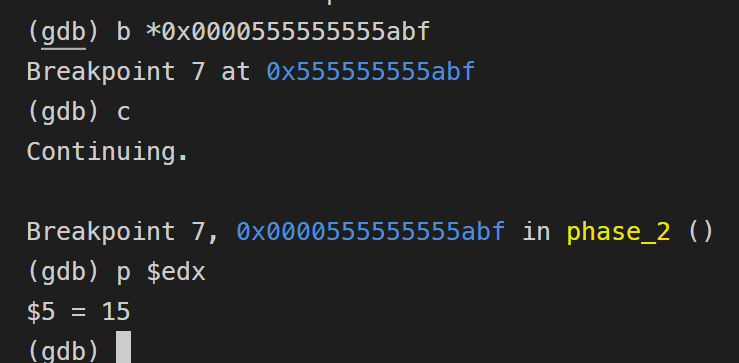
使用print命令查看edx值为6，即第1个数为9.



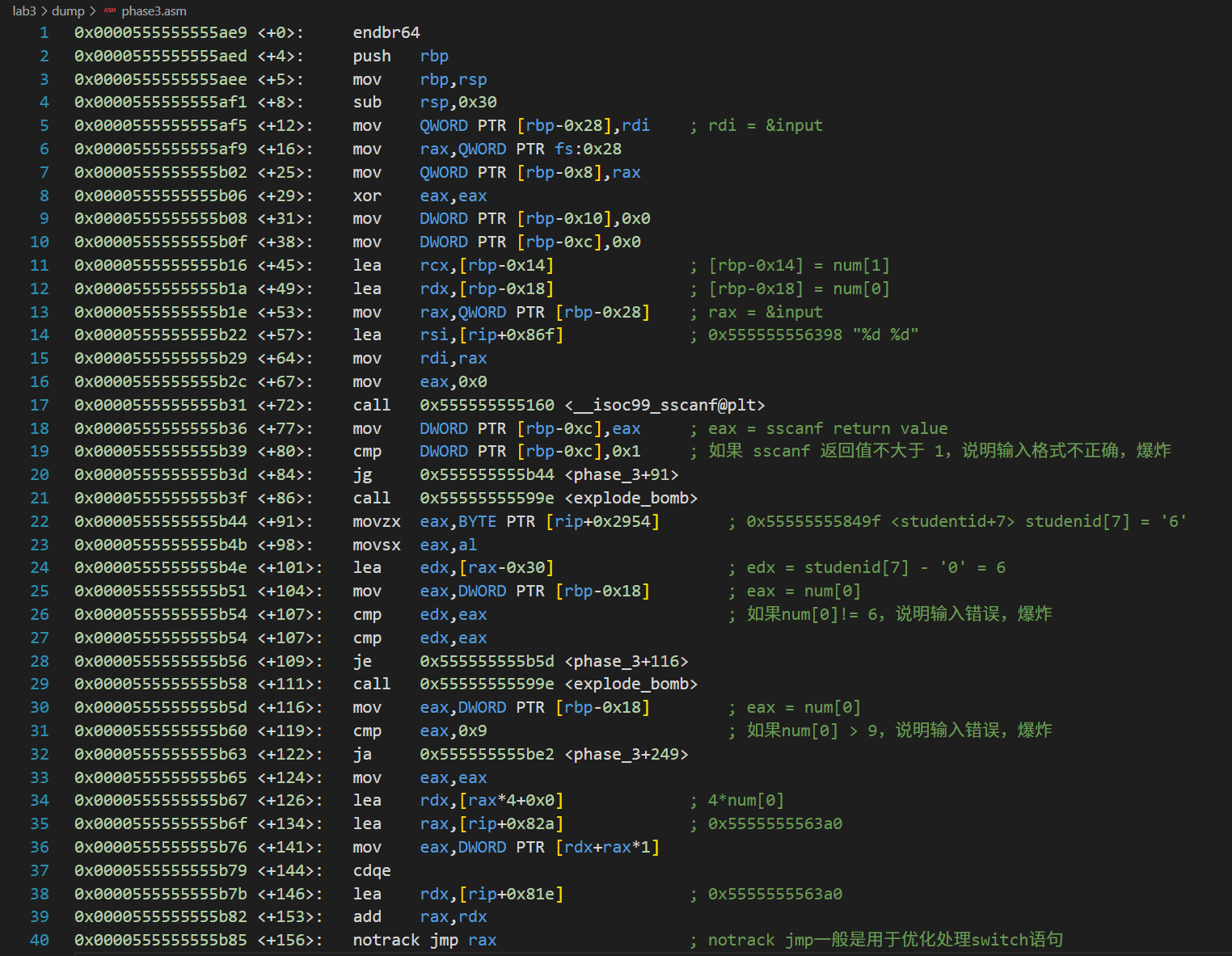
使用set命令修改rax，即将输入的第一个数改为6后，打断点继续运行到下一个cmp处，得到edx值为9。即第2个数为9.



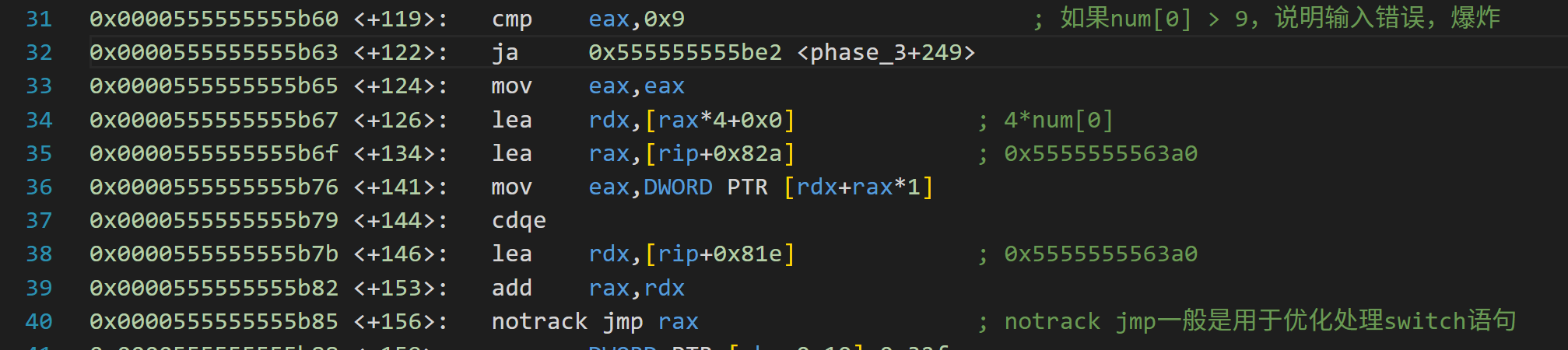
同上，使用set命令修改rax，并运行到下一处cmp，得到rdx值。重复，便可得到全部6个数的答案。



1. **条件分支 phase\_3(char \*input);** 
   1. **反汇编静态分析法**



分析见上图中注释，注意到第40行的notrack jmp指令通常出现在对switch语句的优化中，当switch语句的分支数量较多且值的范围跨度较小时，编译器可能会生成跳转表来优化执行效率。



具体是如何跳转的呢？分析上图中第31行-40行这十行汇编代码。

L31: cmp eax, 0x9

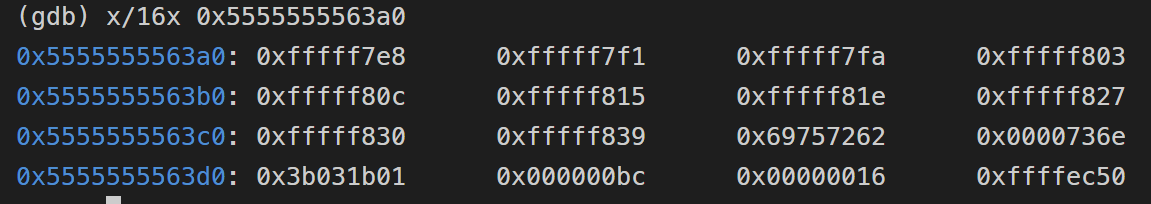
L32: ja 0x555555555be2 <phase\_3+249>

这两句是确保eax值在0~9之间。

L34: lea rdx, [rax\*4+0x0]

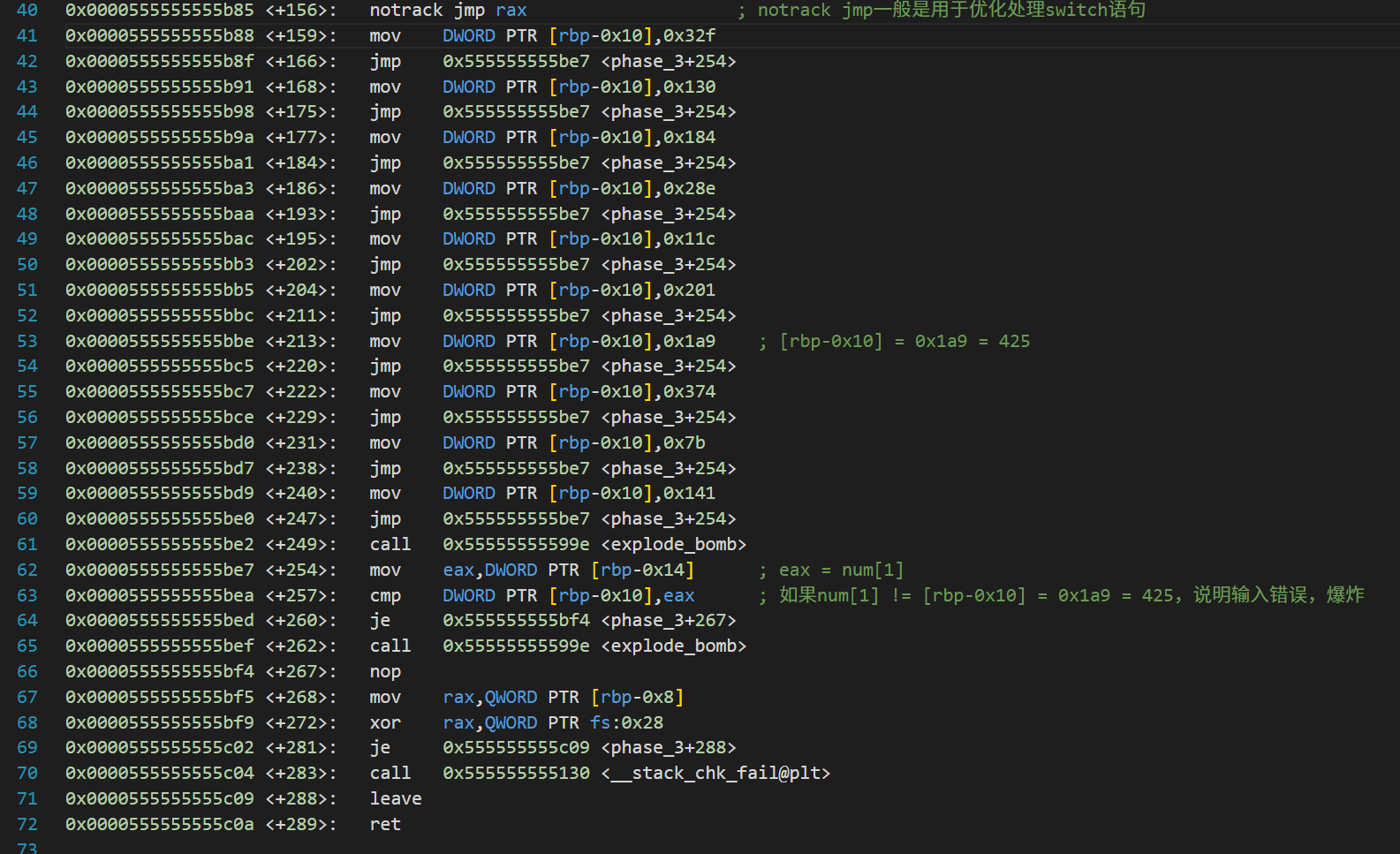
L35: lea rax, [rip+0x82a]

查看rip+0x82a处的值，如下图：

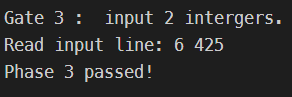


发现每4字节呈现一定规律，前10个数较为相似，且相邻间差值均为9，猜测这10个数即为跳转表，结合反汇编代码第41~第60行，发现每两行的指令占用9个字节，完美对应。

继续分析，跳转表中的值加上基址，便得到了跳转地址，即mov DWORD PTR [rbp-0x10],0x1a9处。其中，[rbp-0x10]后续发现与[rbp-0x14]（即输入的第二个数）进行比较，则0x1a9即为第二个数答案。

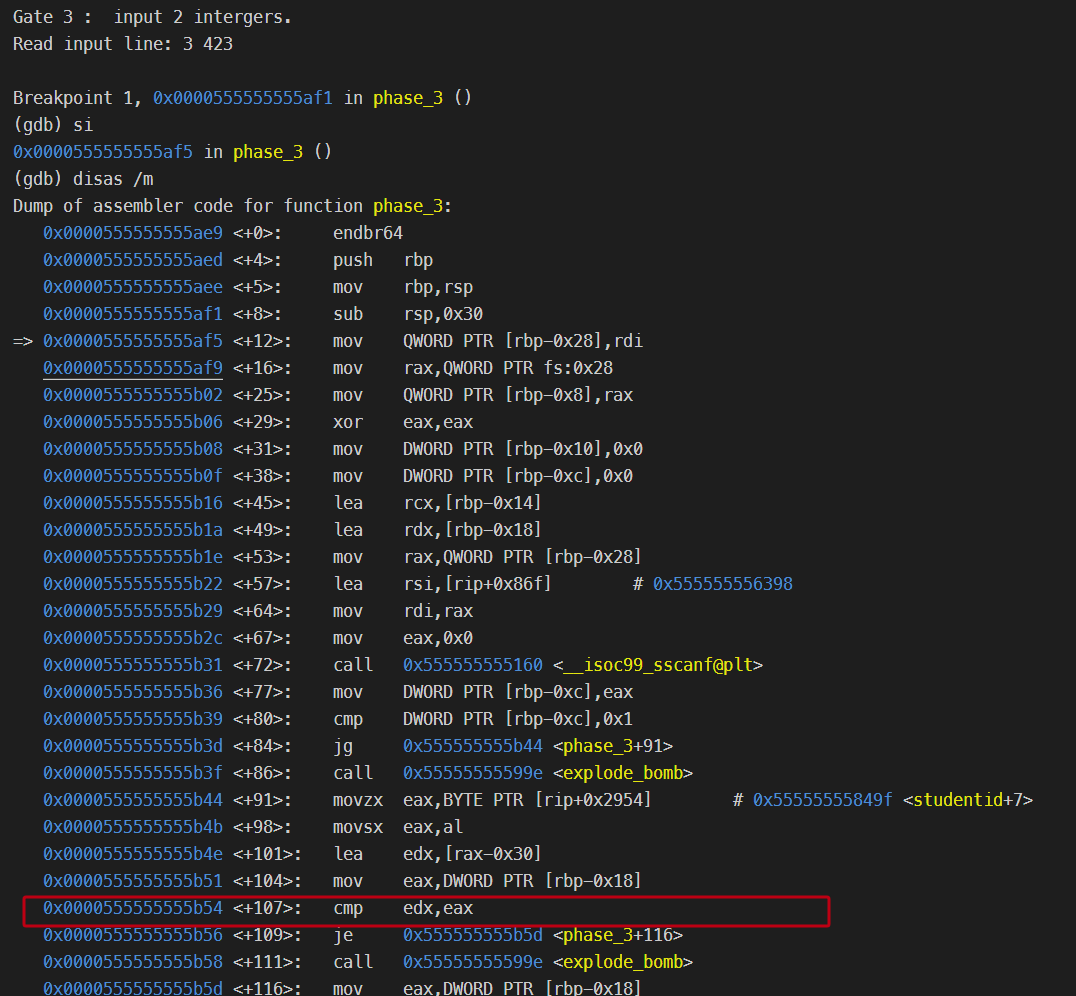


经验证，当输入的两个整数为6和425时成功通过。

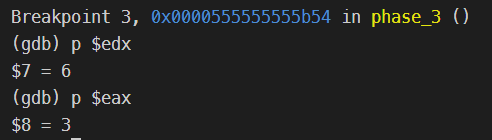


* 1. **动态调试法**

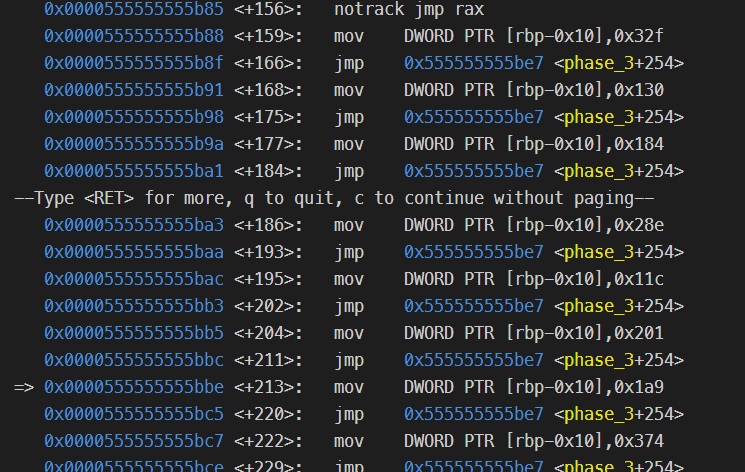
gdb步入函数phase\_3，结合静态分析，在图中cmp指令前设置断点



运行到此处后，打印edx和eax的值，发现分别为6和3。其中3是输入的第一个数，显然第一个数答案是6。

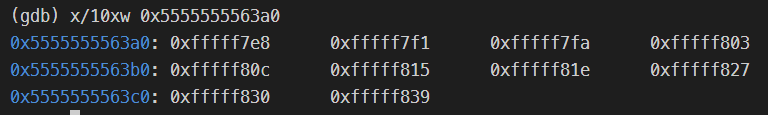


使用set指令修改eax的值，同时，也要修改堆栈中[rbp-0x18]的值。然后，在notrack jmp前打断点并运行到该处，接着，执行nexti指令，发现跳转到该处：



此处与静态分析结果一致，待比较值为0x1a9,即为第二个数的目标值。

对0x5555555563a0处进行分析，发现相邻差值为9，恰好为上图中两条指令的长度，则此处存放的是swich分支语句的跳转偏移表。



综上所述，此阶段的2个数为6和425。

**四、体会**

本实验采用破解闯关的模式，非常具有趣味性和挑战性，让我感觉非常上瘾，在游戏般的过程中感受到了课堂上的知识与实践的融合。

同时，也掌握了命令行环境下gdb调试工具的使用，让我对程序的执行和CPU的运行过程有了更深刻的理解。同样，也了解到linux操作系统的一些操作。

非常喜欢这样的实验课程设计，感谢老师们对课程的设计与指导！