

# 北京邮电大学

## 实验报告



题目： 拆解二进制炸弹

班 级： \_\_\_\_\_

学 号： \_\_\_\_\_

姓 名： \_\_\_\_\_

学 院： \_\_\_\_\_

2023 年 11 月 13 日

# 目录

一、实验目的 .....	3
二、实验环境 .....	3
三、实验内容 .....	3
四、实验步骤及实验分析 .....	3
(一) 准备工作 .....	3
(二) 阶段 1 .....	6
(三) 阶段 2 .....	7
(四) 阶段 3 .....	9
(五) 阶段 4 .....	12
(六) 阶段 5 .....	14
(七) 阶段六 .....	16
(八) 隐藏阶段 .....	20
五、总结体会 .....	24
六、诚信声明 .....	24

## 一、实验目的

- 1、理解 C 语言程序的机器级表示。
- 2、初步掌握 GDB 调试器的用法。
- 3、阅读 C 编译器生成的 x86-64 机器代码，理解不同控制结构的基本指令模式，过程的实现。

## 二、实验环境

- 1、远程登陆工具：MobaXterm（服务器：10.120.11.12，x86-64 版本）
- 2、操作系统：Linux
- 3、调试工具：GDB
- 4、反汇编工具：Objdump

## 三、实验内容

登录 bupt1 服务器，在 home 目录下可以找到 Evil 博士专门为你量身定制的一个 bomb，当运行时，它会要求你输入一个字符串，如果正确，则进入下一关，继续要求你输入下一个字符串；否则，炸弹就会爆炸，输出一行提示信息并向计分服务器提交扣分信息。因此，本实验要求你必须通过反汇编和逆向工程对 bomb 执行文件进行分析，找到正确的字符串来解除这个的炸弹。

本实验通过要求使用课程所学知识拆除一个“binary bombs”来增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。“binary bombs”是一个 Linux 可执行程序，包含了 5 个阶段（或关卡）。炸弹运行的每个阶段要求你输入一个特定字符串，你的输入符合程序预期的输入，该阶段的炸弹就被拆除引信；否则炸弹“爆炸”，打印输出“BOOM!!!”。炸弹的每个阶段考察了机器级程序语言的一个不同方面，难度逐级递增。

为完成二进制炸弹拆除任务，需要使用 gdb 调试器和 objdump 来反汇编 bomb 文件，可以单步跟踪调试每一阶段的机器代码，也可以阅读反汇编代码，从中理解每一汇编语言代码的行为或作用，进而设法推断拆除炸弹所需的目标字符串。实验 2 的具体内容见实验 2 说明。

## 四、实验步骤及实验分析

### （一）准备工作

## 1、解压

阅读实验二说明，按指引登入服务器。

通过 ls 命令找到目录下的 bomb102.tar 文件，输入命令 tar -xvf bomb102.tar 解压，得到 bomb102 文件夹及其中的三个文件：README，bomb.c 和 bomb 可执行文件。

```
2022211073@bupt1:~$ tar -xvf bomb102.tar
bomb102/README
bomb102/bomb.c
bomb102/bomb
```

图 1:解压获得三个文件

## 2、初步查看源文件

输入命令 cat README，查看 README 文件内容。为简略说明。

```
2022211073@bupt1:~/bomb102$ cat README
This is bomb 102.

It belongs to 2022211073 (bupt2022211073
)
```

图 2:README 文件

输入命令 vi bomb.c，查看 bomb.c 源代码内容。查看后输入命令:wq!退出 vi 编辑器。

```
*****
* Dr. Evil's Insidious Bomb, Version 1.1
* Copyright 2011, Dr. Evil Incorporated. All rights reserved.
*
* LICENSE:
*
* Dr. Evil Incorporated (the PERPETRATOR) hereby grants you (the
* VICTIM) explicit permission to use this bomb (the BOMB). This is a
* time limited license, which expires on the death of the VICTIM.
* The PERPETRATOR takes no responsibility for damage, frustration,
* insanity, bug-eyes, carpal-tunnel syndrome, loss of sleep, or other
* harm to the VICTIM. Unless the PERPETRATOR wants to take credit,
* that is. The VICTIM may not distribute this bomb source code to
* any enemies of the PERPETRATOR. No VICTIM may debug,
* reverse-engineer, run "strings" on, decompile, decrypt, or use any
* other technique to gain knowledge of and defuse the BOMB. BOMB
* proof clothing may not be worn when handling this program. The
* PERPETRATOR will not apologize for the PERPETRATOR's poor sense of
* humor. This license is null and void where the BOMB is prohibited
* by law.
*****/

#include <stdio.h>
#include <stdlib.h>
#include "support.h"
#include "phases.h"

/*
 * Note to self: Remember to erase this file so my victims will have no
 * idea what is going on, and so they will all blow up in a
 * spectacular fiendish explosion. -- Dr. Evil
 */

FILE *infile;

int main(int argc, char *argv[])
{
    char *input;
```

```

/* Note to self: remember to port this bomb to Windows and put a
 * fantastic GUI on it. */

/* When run with no arguments, the bomb reads its input lines
 * from standard input. */
if (argc == 1) {
    infile = stdin;
}

/* When run with one argument <file>, the bomb reads from <file>
 * until EOF, and then switches to standard input. Thus, as you
 * defuse each phase, you can add its defusing string to <file> and
 * avoid having to retype it. */
else if (argc == 2) {
    if (!(infile = fopen(argv[1], "r"))) {
        printf("%s: Error: Couldn't open %s\n", argv[0], argv[1]);
        exit(8);
    }
}

/* You can't call the bomb with more than 1 command line argument. */
else {
    printf("Usage: %s [<input_file>]\n", argv[0]);
    exit(8);
}

/* Do all sorts of secret stuff that makes the bomb harder to defuse. */
initialize_bomb();

printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
printf("which to blow yourself up. Have a nice day!\n");

/* Hmm... Six phases must be more secure than one phase! */
input = read_line();          /* Get input */
phase_1(input);               /* Run the phase */
phase_defused();              /* Drat! They figured it out!
 * Let me know how they did it. */
printf("Phase 1 defused. How about the next one?\n");

```

```

/* The second phase is harder. No one will ever figure out
 * how to defuse this... */
input = read_line();
phase_2(input);
phase_defused();
printf("That's number 2. Keep going!\n");

/* I guess this is too easy so far. Some more complex code will
 * confuse people. */
input = read_line();
phase_3(input);
phase_defused();
printf("Halfway there!\n");

/* Oh yeah? Well, how good is your math? Try on this saucy problem! */
input = read_line();
phase_4(input);
phase_defused();
printf("So you got that one. Try this one.\n");

/* Round and 'round in memory we go, where we stop, the bomb blows! */
input = read_line();
phase_5(input);
phase_defused();
printf("Good work! On to the next...\n");

/* This phase will never be used, since no one will get past the
 * earlier ones. But just in case, make this one extra hard. */
input = read_line();
phase_6(input);
phase_defused();

/* Wow, they got it! But isn't something... missing? Perhaps
 * something they overlooked? Mua ha ha ha ha! */

return 0;

```

图 3~5: bomb.c 文件内容

观察 bomb.c 源文件，发现每一个阶段之前会调用 read\_line 函数输入一行字符，再作为参数传到 phase\_x 函数中。猜测这一函数作用为判断输入是否合法（若合法，炸弹不爆炸）。另外，每个阶段结束前都会执行 phase\_defused 函数，作用未知。

### 3、初步查看汇编文件

输入命令 gdb bomb 进入 gdb 调试器，输入命令 disas phase\_defused，查看 phase\_defused 函数的汇编代码，粗略明确其作用。

```
Dump of assembler code for function phase_defused:
0x00000000004017e2 <+0>:  sub    $0x78,%rsp
0x00000000004017e6 <+4>:  mov     %fs:0x28,%rax
0x00000000004017ef <+13>: mov     %rax,0x68(%rsp)
0x00000000004017f4 <+18>:  xor     %eax,%eax
0x00000000004017f6 <+20>:  mov     $0x1,%edi
0x00000000004017fb <+25>:  callq   0x40153d <send_msg>
0x0000000000401800 <+30>:  cmpl    $0x6,0x202fa5(%rip)          # 0x6047ac <num_input_strings>
0x0000000000401807 <+37>:  jne     0x401876 <phase_defused+148>
0x0000000000401809 <+39>:  lea     0x10(%rsp),%r8
0x000000000040180e <+44>:  lea     0xc(%rsp),%rcx
0x0000000000401813 <+49>:  lea     0x8(%rsp),%rdx
0x0000000000401818 <+54>:  mov     $0x4029b7,%esi
0x000000000040181d <+59>:  mov     $0x6048b0,%edi
0x0000000000401822 <+64>:  mov     $0x0,%eax
0x0000000000401827 <+69>:  callq   0x400c40 <__isoc99_sscanf@plt>
0x000000000040182c <+74>:  cmp     $0x3,%eax
0x000000000040182f <+77>:  jne     0x401862 <phase_defused+128>
0x0000000000401831 <+79>:  mov     $0x4029c0,%esi
0x0000000000401836 <+84>:  lea     0x10(%rsp),%rdi
0x000000000040183b <+89>:  callq   0x401373 <strings_not_equal>
0x0000000000401840 <+94>:  test    %eax,%eax
0x0000000000401842 <+96>:  jne     0x401862 <phase_defused+128>
0x0000000000401844 <+98>:  mov     $0x402818,%edi
0x0000000000401849 <+103>: callq   0x400b70 <puts@plt>
0x000000000040184e <+108>: mov     $0x402840,%edi
0x0000000000401853 <+113>: callq   0x400b70 <puts@plt>
0x0000000000401858 <+118>: mov     $0x0,%eax
0x000000000040185d <+123>: callq   0x401289 <secret_phase>
0x0000000000401862 <+128>: mov     $0x402878,%edi
0x0000000000401867 <+133>: callq   0x400b70 <puts@plt>
0x000000000040186c <+138>: mov     $0x4028a8,%edi
0x0000000000401871 <+143>: callq   0x400b70 <puts@plt>
0x0000000000401876 <+148>: mov     0x68(%rsp),%rax
0x000000000040187b <+153>: xor     %fs:0x28,%rax
0x0000000000401884 <+162>: je      0x40188b <phase_defused+169>
0x0000000000401886 <+164>: callq   0x400b90 <__stack_chk_fail@plt>
0x000000000040188b <+169>: add     $0x78,%rsp
0x000000000040188f <+173>: retq
```

图 6:phase\_函数汇编代码

有两处是我注意到的地方：123 行，存在隐藏关炸弹函数 secret\_phase 的跳转入口。

```
0x000000000040185d <+123>:  callq   0x401289 <secret_phase>
```

图 7:phase\_函数 123 行汇编语句

30 行的比较语句，由 123 行猜测此处为拆弹完成个数=6 时不跳转到 return（148 行后内容基本为 return）。但后续需要验证。

```
0x0000000000401800 <+30>:  cmpl    $0x6,0x202fa5(%rip)          # 0x6047ac <num_input_strings>
0x0000000000401807 <+37>:  jne     0x401876 <phase_defused+148>
```

图 8:phase\_函数 30 行汇编语句

## (二) 阶段 1

### 1、查看汇编代码与初步分析



输入命令 `disas phase_1`，查看 `phase_1` 的汇编代码。

```
(gdb) disas phase_1
Dump of assembler code for function phase_1:
0x0000000000400f2d <+0>:    sub    $0x8,%rsp
0x0000000000400f31 <+4>:    mov    $0x402630,%esi
0x0000000000400f36 <+9>:    callq 0x401373 <strings_not_equal>
0x0000000000400f3b <+14>:   test   %eax,%eax
0x0000000000400f3d <+16>:   je     0x400f44 <phase_1+23>
0x0000000000400f3f <+18>:   callq 0x401647 <explode_bomb>
0x0000000000400f44 <+23>:   add    $0x8,%rsp
0x0000000000400f48 <+27>:   retq
End of assembler dump.
```

图 9: `phase_1` 汇编代码

观察分析，得知该汇编代码逻辑大概为：将 `0x402630` 处的值移入 `esi`，与 `edi` (`phase_1`) 的输入字符串一起传入函数 `strings_not_equal`。若返回为 0，说明相等，跳转到 23 行，否则炸弹爆炸。

## 2、查看所需字符串

从上文分析可知，需要知道 `0x402630` 处的值，倒推程序需要的输入。

输入命令 `x /s 0x402630`，以字符串形式查看该内存存储的字符串。

```
(gdb) x /s 0x402630
0x402630:      "In 2012, BUPT officially started the construction of new ShaHe campus."
```

图 10: `0x402630` 处的字符串

## 3、设置断点

在 (1) 的汇编指令中，注意到 `explode_bomb` 函数会使炸弹爆炸，因此可在此函数设置断点，防止输入错误时炸弹爆炸。

输入命令 `break explode_bomb`，设置断点。

```
(gdb) break explode_bomb
Breakpoint 1 at 0x401647
```

图 11: 设置断点

## 4、成功运行，拆除阶段一炸弹

综上分析，输入指令 `run` 在 `gdb` 中运行程序。输入制定字符串，显示阶段一炸弹拆除成功。输入 `ctrl-c`，终止程序，继续拆除下一阶段炸弹。

```
(gdb) run
Starting program: /students/2022211073/bomb102/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
In 2012, BUPT officially started the construction of new ShaHe campus.
Phase 1 defused. How about the next one?
^C
Program received signal SIGINT, Interrupt.
```

图 12: 拆除阶段 1 炸弹

## (三) 阶段 2

1、查看汇编代码与初步分析

输入命令 `disas phase_2`，查看该函数的汇编代码。

```
(gdb) disas phase_2
Dump of assembler code for function phase_2:
0x0000000000400f49 <+0>:    push    %rbp
0x0000000000400f4a <+1>:    push    %rbx
0x0000000000400f4b <+2>:    sub     $0x28,%rsp
0x0000000000400f4f <+6>:    mov     %fs:0x28,%rax
0x0000000000400f58 <+15>:   mov     %rax,0x18(%rsp)
0x0000000000400f5d <+20>:   xor     %eax,%eax
0x0000000000400f5f <+22>:   mov     %rsp,%rsi
0x0000000000400f62 <+25>:   callq   0x40167d <read_six_numbers>
0x0000000000400f67 <+30>:   cmpl    $0x0,(%rsp)
0x0000000000400f6b <+34>:   jns     0x400f72 <phase_2+41>
0x0000000000400f6d <+36>:   callq   0x401647 <explode_bomb>
0x0000000000400f72 <+41>:   mov     %rsp,%rbp
0x0000000000400f75 <+44>:   mov     $0x1,%ebx
0x0000000000400f7a <+49>:   mov     %ebx,%eax
0x0000000000400f7c <+51>:   add     0x0(%rbp),%eax
0x0000000000400f7f <+54>:   cmp     %eax,0x4(%rbp)
0x0000000000400f82 <+57>:   je      0x400f89 <phase_2+64>
0x0000000000400f84 <+59>:   callq   0x401647 <explode_bomb>
0x0000000000400f89 <+64>:   add     $0x1,%ebx
0x0000000000400f8c <+67>:   add     $0x4,%rbp
0x0000000000400f90 <+71>:   cmp     $0x6,%ebx
0x0000000000400f93 <+74>:   jne     0x400f7a <phase_2+49>
0x0000000000400f95 <+76>:   mov     0x18(%rsp),%rax
0x0000000000400f9a <+81>:   xor     %fs:0x28,%rax
0x0000000000400fa3 <+90>:   je      0x400faa <phase_2+97>
0x0000000000400fa5 <+92>:   callq   0x400b90 <__stack_chk_fail@plt>
0x0000000000400faa <+97>:   add     $0x28,%rsp
0x0000000000400fae <+101>:  pop     %rbx
0x0000000000400faf <+102>:  pop     %rbp
0x0000000000400fb0 <+103>:  retq
End of assembler dump.
```

图 13: phase\_2 汇编代码

观察发现，其代码逻辑大概为：读入六个数字，对读入的数字进行一系列操作，不满足条件则引爆炸弹，满足条件则拆除成功。

输入命令 `disas read_six_numbers`，查看子函数的汇编代码。

发现该函数内部调用了 `sscanf` 函数，且前几行出现了与阶段一第 4 行汇编代码类似的地址 `0x402961`。输入 `x/s 0x402961` 查看存储的内容。

```
(gdb) x /s 0x402961
0x402961:    "%d %d %d %d %d %d"
```

图 14: 0x402961 处的字符串

分析这里是 `scanf` 函数调用之前的格式化字符串语句，因此 `phase_2` 需要输入六个整数。

2、进一步分析 phase\_2 函数

观察分析图 14，注意到下面的行数：

编号	汇编代码行数	操作含义	备注
1	25	调用函数，读取六个数字	该阶段炸弹输入为六个数字
2	30—36	将第一个输入数 <code>n1</code> 与 0 比较 若 <code>n1 &gt;= 0</code> ，跳转到 41 行，不爆炸；反之执行到 36 行，爆炸	第一个输入数 <code>n1 &gt;= 0</code>
3	41—74	<code>%rbp</code> 作为遍历数组的指针； <code>%ebx</code> 作为计数	总体分析：



		器 ( $\leq 6$ )，也是两个数间的差值	通过循环实现了判断这六个数是否为一个差值为 1, 2, 3, 4, 5 的数列
	41-44	令 %rbp 指向第一个输入数 (%rsp)，%ebx=1 开始计数	循环前的初始化
	49-51	%eax=该输入数+%ebx	循环语句
	54-59	比较下一个输入数与 %eax 若相等就跳转到 64 行继续，若不相等执行 59 行爆炸	判断是否爆炸 0x4(%rbp) 为下一个输入数 (int 型，偏移量为 4)
	64-67	%ebx++ (计数)，%rbp 指向下一个输入数	循环语句
	71-74	计数器 < 6，跳转到 49 行继续循环	判断循环结束

### 3、成功运行，拆除阶段二炸弹

综上所述，输入 0 1 3 6 10 15 (第一个输入数  $\geq 0$  且差值为 1, 2, 3, 4, 5)，阶段一炸弹拆除成功。

```
Starting program: /students/2022211073/bomb102/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
In 2012, BUPT officially started the construction of new ShaHe campus.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
^C
Program received signal SIGINT, Interrupt.
```

图 15: 拆除阶段 2 炸弹

## (四) 阶段 3

### 1、查看汇编代码与初步分析

输入命令 `disas phase_3`，查看该函数的汇编代码。

```

(gdb) disas phase_3
Dump of assembler code for function phase_3:
0x000000000400fb1 <+0>:    sub    $0x18,%rsp
0x000000000400fb5 <+4>:    mov    %fs:0x28,%rax
0x000000000400fbe <+13>:   mov    %rax,0x8(%rsp)
0x000000000400fc3 <+18>:   xor    %eax,%eax
0x000000000400fc5 <+20>:   lea    0x4(%rsp),%rcx
0x000000000400fca <+25>:   mov    %rsp,%rdx
0x000000000400fcd <+28>:   mov    $0x40296d,%esi
0x000000000400fd2 <+33>:   callq 0x400c40 <__isoc99_sscanf@plt>
0x000000000400fd7 <+38>:   cmp    $0x1,%eax
0x000000000400fda <+41>:   jg     0x400fe1 <phase_3+48>
0x000000000400fdc <+43>:   callq 0x401647 <explode_bomb>
0x000000000400fe1 <+48>:   cmpl   $0x7,(%rsp)
0x000000000400fe5 <+52>:   ja     0x401022 <phase_3+113>
0x000000000400fe7 <+54>:   mov    (%rsp),%eax
0x000000000400fea <+57>:   jmpq   *0x4026a0(,%rax,8)
0x000000000400ff1 <+64>:   mov    $0x90,%eax
0x000000000400ff6 <+69>:   jmp    0x401033 <phase_3+130>
0x000000000400ff8 <+71>:   mov    $0x398,%eax
0x000000000400ffd <+76>:   jmp    0x401033 <phase_3+130>
0x000000000400fff <+78>:   mov    $0x23a,%eax
0x000000000401004 <+83>:   jmp    0x401033 <phase_3+130>
0x000000000401006 <+85>:   mov    $0x41,%eax
0x00000000040100b <+90>:   jmp    0x401033 <phase_3+130>
0x00000000040100d <+92>:   mov    $0x159,%eax
0x000000000401012 <+97>:   jmp    0x401033 <phase_3+130>
0x000000000401014 <+99>:   mov    $0x25a,%eax
0x000000000401019 <+104>:  jmp    0x401033 <phase_3+130>
0x00000000040101b <+106>:  mov    $0x20d,%eax
0x000000000401020 <+111>:  jmp    0x401033 <phase_3+130>
0x000000000401022 <+113>:  callq 0x401647 <explode_bomb>
0x000000000401027 <+118>:  mov    $0x0,%eax
0x00000000040102c <+123>:  jmp    0x401033 <phase_3+130>
0x00000000040102e <+125>:  mov    $0x26f,%eax
0x000000000401033 <+130>:  cmp    0x4(%rsp),%eax
0x000000000401037 <+134>:  je     0x40103e <phase_3+141>

0x000000000401039 <+136>:  callq 0x401647 <explode_bomb>
0x00000000040103e <+141>:  mov    0x8(%rsp),%rax
0x000000000401043 <+146>:  xor    %fs:0x28,%rax
--Type <RET> for more, q to quit, c to continue without paging--c
0x00000000040104c <+155>:  je     0x401053 <phase_3+162>
0x00000000040104e <+157>:  callq 0x400b90 <__stack_chk_fail@plt>
0x000000000401053 <+162>:  add    $0x18,%rsp
0x000000000401057 <+166>:  retq
End of assembler dump.

```

图 16-17: phase\_3 汇编代码

发现该函数汇编代码与阶段 2 的 read\_six\_numbers 函数有相似处，即调用了 sscanf 函数，且前一行出现了地址 0x402961。

```

0x000000000400fcd <+28>:    mov    $0x40296d,%esi
0x000000000400fd2 <+33>:    callq 0x400c40 <__isoc99_sscanf@plt>

```

图 18: 调用 sscanf 函数的相似处

输入命令 x /s 0x402961，查看内存 0x402961 处存储的内容。

```

(gdb) x /s 0x40296d
0x40296d:      "%d %d"

```

图 19: 0x402961 处的字符串

因此，该阶段需要输入两个整数。

此外，在 48 行出现了某变量与 7 的比较，在  $\leq 7$  的情况下这个值还作为偏移量在 57 行的跳转表中使用。分析得知阶段 3 存在 switch 语句，且  $\text{case} \leq 7$ 。

```

0x000000000400fe1 <+48>:    cmpl   $0x7,(%rsp)
0x000000000400fe5 <+52>:    ja     0x401022 <phase_3+113>
0x000000000400fe7 <+54>:    mov    (%rsp),%eax
0x000000000400fea <+57>:    jmpq   *0x4026a0(,%rax,8)

```

图 20: 跳转表

## 2、查看跳转表

输入指令 `x /8xg 0x4026a0`，以 16 进制查看 8 个内存单元的值（每个内存单元位 64 位，取决于机器种类；`xg` 为双字，八个字节）。

```
(gdb) x /8xg 0x4026a0
0x4026a0: 0x00000000000040102e 0x000000000000400ff1
0x4026b0: 0x000000000000400ff8 0x000000000000400fff
0x4026c0: 0x000000000000401006 0x00000000000040100d
0x4026d0: 0x000000000000401014 0x00000000000040101b
```

图 21: 跳转表内容

分析可知，每个内存单元对应的地址为 `switch` 函数该 `case` 的入口，标注于下图代码中。

```
1 0x000000000000400ff1 <+64>: mov $0x90,%eax
0x000000000000400ff6 <+69>: jmp 0x401033 <phase_3+130>
2 0x000000000000400ff8 <+71>: mov $0x398,%eax
0x000000000000400ffd <+76>: jmp 0x401033 <phase_3+130>
3 0x000000000000400fff <+78>: mov $0x23a,%eax
0x000000000000401004 <+83>: jmp 0x401033 <phase_3+130>
4 0x000000000000401006 <+85>: mov $0x41,%eax
0x00000000000040100b <+90>: jmp 0x401033 <phase_3+130>
5 0x00000000000040100d <+92>: mov $0x159,%eax
0x000000000000401012 <+97>: jmp 0x401033 <phase_3+130>
6 0x000000000000401014 <+99>: mov $0x25a,%eax
0x000000000000401019 <+104>: jmp 0x401033 <phase_3+130>
7 0x00000000000040101b <+106>: mov $0x20d,%eax
0x000000000000401020 <+111>: jmp 0x401033 <phase_3+130>
0x000000000000401022 <+113>: callq 0x401647 <explode_bomb>
0x000000000000401027 <+118>: mov $0x0,%eax
0x00000000000040102c <+123>: jmp 0x401033 <phase_3+130>
0 0x00000000000040102e <+125>: mov $0x26f,%eax
0x000000000000401033 <+130>: cmp 0x4(%rsp),%eax
0x000000000000401037 <+134>: je 0x40103e <phase_3+141>
```

图 22: `switch` 函数入口

## 3、分析 `switch` 函数源代码

观察图 22 得知，其实不需要逐个分析出各个 `case` 的源代码内容。

取第一个输入数 (`%rsp`) 为 0，由图 20 可知该值会被传入跳转表，使程序跳转到 `case 0` 语句 (125 行)。

`case 0` 语句中比较 623 (16 进制: `0x26f`) 与第二个输入数 (`0x4(%rsp)`)，若相等，跳转到 141，阶段三结束。

```
0 0x00000000000040102e <+125>: mov $0x26f,%eax
0x000000000000401033 <+130>: cmp 0x4(%rsp),%eax
0x000000000000401037 <+134>: je 0x40103e <phase_3+141>
```

图 23: `case 0` 语句

## 4、成功运行，拆除阶段三炸弹

尝试输入 0 623 (第一个输入数  $\leq 7$ ，623 为 `case 0` 所需的返回值)，阶段三炸弹拆除成功。

```
Starting program: /students/2022211073/bomb102/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
In 2012, BUPT officially started the construction of new ShaHe campus.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 623
Halfway there!
^C
Program received signal SIGINT, Interrupt.
```

图 24: 拆除阶段 3 炸弹

## (五) 阶段 4

### 1、查看汇编代码与初步分析

输入命令 `disas phase_4`，查看该函数的汇编代码。

```
(gdb) disas phase_4
Dump of assembler code for function phase_4:
0x0000000000401093 <+0>:    sub    $0x18,%rsp
0x0000000000401097 <+4>:    mov    %fs:0x28,%rax
0x00000000004010a0 <+13>:   mov    %rax,0x8(%rsp)
0x00000000004010a5 <+18>:   xor    %eax,%eax
0x00000000004010a7 <+20>:   mov    %rsp,%rcx
0x00000000004010aa <+23>:   lea    0x4(%rsp),%rdx
0x00000000004010af <+28>:   mov    $0x40296d,%esi
0x00000000004010b4 <+33>:   callq 0x400c40 <__isoc99_sscanf@plt>
0x00000000004010b9 <+38>:   cmp    $0x2,%eax
0x00000000004010bc <+41>:   jne    0x4010c9 <phase_4+54>
0x00000000004010be <+43>:   mov    (%rsp),%eax
0x00000000004010c1 <+46>:   sub    $0x2,%eax
0x00000000004010c4 <+49>:   cmp    $0x2,%eax
0x00000000004010c7 <+52>:   jbe    0x4010ce <phase_4+59>
0x00000000004010c9 <+54>:   callq 0x401647 <explode_bomb>
0x00000000004010ce <+59>:   mov    (%rsp),%esi
0x00000000004010d1 <+62>:   mov    $0x7,%edi
0x00000000004010d6 <+67>:   callq 0x401058 <func4>
0x00000000004010db <+72>:   cmp    0x4(%rsp),%eax
0x00000000004010df <+76>:   je     0x4010e6 <phase_4+83>
0x00000000004010e1 <+78>:   callq 0x401647 <explode_bomb>
0x00000000004010e6 <+83>:   mov    0x8(%rsp),%rax
0x00000000004010eb <+88>:   xor    %fs:0x28,%rax
0x00000000004010f4 <+97>:   je     0x4010fb <phase_4+104>
0x00000000004010f6 <+99>:   callq 0x400b90 <__stack_chk_fail@plt>
0x00000000004010fb <+104>:  add    $0x18,%rsp
0x00000000004010ff <+108>:  retq
End of assembler dump.
```

图 25: phase\_4 汇编代码

观察发现, 与前 3 个阶段相似地, 发现 phase\_4 中调用了 `sscanf` 函数。输入命令 `x /s 0x40296d`, 查看该内存处的存储内容, 发现需要输入两个整数。

```
0x00000000004010af <+28>:    mov    $0x40296d,%esi
0x00000000004010b4 <+33>:    callq 0x400c40 <__isoc99_sscanf@plt>
```

图 26: 调用 `sscanf` 函数

```
(gdb) x /s 0x40296d
0x40296d:      "%d %d"
```

图 27: 0x40296d 处的字符串

另外注意到, phase\_4 调用了子函数 `func4`。输入 `disas func4`, 查看该子函数的汇编代码。



```

(gdb) disas func4
Dump of assembler code for function func4:
0x0000000000401058 <+0>: test %edi,%edi
0x000000000040105a <+2>: jle 0x401087 <func4+47>
0x000000000040105c <+4>: mov %esi,%eax
0x000000000040105e <+6>: cmp $0x1,%edi
0x0000000000401061 <+9>: je 0x401091 <func4+57>
0x0000000000401063 <+11>: push %r12
0x0000000000401065 <+13>: push %rbp
0x0000000000401066 <+14>: push %rbx
0x0000000000401067 <+15>: mov %esi,%ebp
0x0000000000401069 <+17>: mov %edi,%ebx
0x000000000040106b <+19>: lea -0x1(%rdi),%edi
0x000000000040106e <+22>: callq 0x401058 <func4>
0x0000000000401073 <+27>: lea 0x0(%rbp,%rax,1),%r12d
0x0000000000401078 <+32>: lea -0x2(%rbx),%edi
0x000000000040107b <+35>: mov %ebp,%esi
0x000000000040107d <+37>: callq 0x401058 <func4>
0x0000000000401082 <+42>: add %r12d,%eax
0x0000000000401085 <+45>: jmp 0x40108d <func4+53>
0x0000000000401087 <+47>: mov $0x0,%eax
0x000000000040108c <+52>: retq
0x000000000040108d <+53>: pop %rbx
0x000000000040108e <+54>: pop %rbp
0x000000000040108f <+55>: pop %r12
0x0000000000401091 <+57>: repz retq
End of assembler dump.

```

图 28: func4 汇编代码

观察 func4，在 22 行、37 行存在 func4 函数的调用，说明这是一个递归函数。

## 2、进一步分析 phase\_4 函数与 func4 函数

我对 phase\_4 函数进行了更为细致的观察，发现 43-78 行是需要重点分析的内容：

编号	汇编代码行数	操作含义	备注
1	43-54	如果第二个输入数 $n2-2 \leq 2$ ，不爆炸，跳转到 59 行；反之爆炸 因此 $n2$ 要 $\leq 4$	与阶段二的 read_six_numbers 函数不同，sscanf 函数将先输入的数放在栈底，后输入的放在栈顶。所以 43 行的 (%rsp) 是第二个输入的数
2	59-67	将 7、第二个输入数分别作为第一个、第二个参数传入 func4	
3	72-78	比较 func4 返回值 rax 和第一个输入数 若相等，跳转到 83 行，拆除炸弹；反之炸弹爆炸	0x4 (%rsp) 为第一个输入数

在仔细分析 phase\_4 函数之前，发现 func4 的汇编代码并不长，我本想阅读汇编语句将源代码解释出来，从而代入参数计算 eax 应有的值。但后来我意识到，func4 的传入参数只需要传入常数 7 与  $n2$ ，与  $n1$  无关。因此，只要我输入一个合法的  $n2$  ( $\leq 4$ )，并在 func4 的返回处（代码位置 0x00000000004010db）设置断点，便可以直接查看其返回值，无需花时间重写 func4 源代码。

## 3、设置断点



输入命令 `break *0x00000000004010db`，设置断点以查看 `%eax` 返回值。

```
(gdb) break *0x00000000004010db
Breakpoint 4 at 0x4010db
```

图 29:设置断点

输入命令 `info break`，检查断点信息。

```
(gdb) info break
Num      Type           Disp Enb Address              What
1        breakpoint     keep y   0x0000000000401647 <explode_bomb>
4        breakpoint     keep y   0x00000000004010db <phase_4+72>
```

图 30:检查断点信息

#### 4、查看 func4 返回值

输入命令 `run` 运行程序，在 `phase_4` 处输入 100 4（100 为任意 `int` 型数）。执行到该断点后，输入命令 `print $eax`，打印出 `func4` 返回值，为 132。132 即为需要的第一个输入数。

```
Breakpoint 4, 0x00000000004010db in phase_4 ()
(gdb) print $eax
$5 = 132
```

图 31:打印 `%eax` 的值

#### 5、成功运行，拆除阶段四炸弹

尝试输入 132 4（第一个输入数为 `func4` 返回值，第二个输入数  $\leq 4$ ），阶段四炸弹拆除成功。

```
Starting program: /students/2022211073/bomb102/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
In 2012, BUPT officially started the construction of new ShaHe campus.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 623
Halfway there!
132 4
So you got that one. Try this one.
^C
Program received signal SIGINT, Interrupt.
```

图 32:拆除阶段 4 炸弹

### (六) 阶段 5

#### 1、查看汇编代码与初步分析

输入命令 `disas phase_5`，查看该函数的汇编代码。

```

(gdb) disas phase_5
Dump of assembler code for function phase_5:
0x0000000000401100 <+0>:      push    %rbx
0x0000000000401101 <+1>:      mov     %rdi,%rbx
0x0000000000401104 <+4>:      callq  0x401355 <string_length>
0x0000000000401109 <+9>:      cmp     $0x6,%eax
0x000000000040110c <+12>:     je      0x401113 <phase_5+19>
0x000000000040110e <+14>:     callq  0x401647 <explode_bomb>
0x0000000000401113 <+19>:     mov     %rbx,%rax
0x0000000000401116 <+22>:     lea     0x6(%rbx),%rdi
0x000000000040111a <+26>:     mov     $0x0,%ecx
0x000000000040111f <+31>:     movzbl (%rax),%edx
0x0000000000401122 <+34>:     and     $0xf,%edx
0x0000000000401125 <+37>:     add     0x4026e0(,%rdx,4),%ecx
0x000000000040112c <+44>:     add     $0x1,%rax
0x0000000000401130 <+48>:     cmp     %rdi,%rax
0x0000000000401133 <+51>:     jne     0x40111f <phase_5+31>
0x0000000000401135 <+53>:     cmp     $0x34,%ecx
0x0000000000401138 <+56>:     je      0x40113f <phase_5+63>
0x000000000040113a <+58>:     callq  0x401647 <explode_bomb>
0x000000000040113f <+63>:     pop     %rbx
0x0000000000401140 <+64>:     retq
End of assembler dump.

```

图 33: phase\_5 汇编代码

观察发现，此处调用了 string\_length 函数计算输入字符串长度。若返回值为 6，跳转到 19 行，不爆炸；反之爆炸。因此我们要输入长度为 6 的字符串。

```

0x0000000000401104 <+4>:      callq  0x401355 <string_length>
0x0000000000401109 <+9>:      cmp     $0x6,%eax
0x000000000040110c <+12>:     je      0x401113 <phase_5+19>
0x000000000040110e <+14>:     callq  0x401647 <explode_bomb>

```

图 34: string\_length 函数

## 2、分析 phase\_5 函数

重点分析 19-58 行，这是一个循环：

编号	汇编代码行数	操作含义	备注
1	19-22	%rax 指向字符串首 %rdi=%rax+6	循环前的初始化 %rax, %rdi, %rbx 在这里都是存储内存地址的指针，其中 %rbx 存储了输入字符串的指针地址，%rax, %rdi 用于循环次数的计数，共循环六次（48 行）
2	26	%ecx=0	循环前的初始化 %ecx 用于存储循环中的加法和 sum（37 行），最后进行比较（53 行）
3	31	从 %rax 访问字符串，将字符串每个字节的值由 ascii 翻译成数字，截断存入 %edx	循环语句
4	34	与 0xf 进行与运算，只保留 %edx 的低四位，其他位置 0	循环语句

分析到 37 行时，发现此处访问了内存为 0x4026e0+4\*%rdx 的值。每次偏移四个字节，猜测这是一个 int 型数组。

输入命令 x/16xw 0x4026e0（xw 为单字，四个字节），查看该数组结构内容。

```
(gdb) x /16xw 0x4026e0
0x4026e0 <array.3599>: 0x00000002      0x0000000a      0x00000006      0x00000001
0x4026f0 <array.3599+16>: 0x0000000c      0x00000010      0x00000009      0x00000003
0x402700 <array.3599+32>: 0x00000004      0x00000007      0x0000000e      0x00000005
0x402710 <array.3599+48>: 0x0000000b      0x00000008      0x0000000f      0x0000000d
```

图 35: 0x4026e0 处的数组

可知, array[0]=2, array[1]=10, array[2]=6, array[3]=1 (十进制)。

### 3、进一步分析 phase\_5 函数

继续分析 38-58 行:

编号	汇编代码行数	操作含义	备注
5	37	%ecx+=array[%rdx]	循环语句
2	44	%rax++	循环语句
3	48-51	比较%rax 和%rdi, 若次数不等于 6 就返回 32 行继续循环	条件判断
4	53-58	若 6 次后%ecx 的和等于 0x34, 炸弹成功拆除	跳出循环, 判断输入值是否满足要求

由上文分析可推断: 程序要求输入六个字符, 这六个字符经过 ascii 表转换变为六个整数值。这六个值和 0xf 进行与运算后, 仅保留了低四位。接着, 这六个数作为下标访问 array 数组并求和, 其和等于 52 (0x34) 时炸弹成功拆除。

取前五个值为 array[1], 第六个值为 array[0],  $10*5+2=52$ , 恰好凑齐。1 对应 0001, 0 对应 0000, 查询 ascii 表找到两个适合的值:

二进制	字符
01110000	p
01110001	q

### 4、成功运行, 拆除阶段五炸弹

尝试输入 qqqqqp (恰好凑齐 52), 阶段五炸弹拆除成功。

```
Starting program: /students/2022211073/bomb102/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
In 2012, BUPT officially started the construction of new ShaHe campus.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 623
Halfway there!
132 4
So you got that one. Try this one.
qqqqqp
Good work! On to the next...
^C
Program received signal SIGINT, Interrupt.
```

图 36: 拆除阶段 5 炸弹

## (七) 阶段六

### 1、查看汇编代码与初步分析

输入命令 `disas phase_6`, 查看该函数的汇编代码。

```
(gdb) disas phase_6
Dump of assembler code for function phase_6:
0x0000000000401141 <+0>:      push    %r13
0x0000000000401143 <+2>:      push    %r12
0x0000000000401145 <+4>:      push    %rbp
0x0000000000401146 <+5>:      push    %rbx
0x0000000000401147 <+6>:      sub     $0x68,%rsp
0x000000000040114b <+10>:     mov     %fs:0x28,%rax
0x0000000000401154 <+19>:     mov     %rax,0x58(%rsp)
0x0000000000401159 <+24>:     xor     %eax,%eax
0x000000000040115b <+26>:     mov     %rsp,%rsi
0x000000000040115e <+29>:     callq  0x40167d <read_six_numbers>
0x0000000000401163 <+34>:     mov     %rsp,%r12
0x0000000000401166 <+37>:     mov     $0x0,%r13d
0x000000000040116c <+43>:     mov     %r12,%rbp
0x000000000040116f <+46>:     mov     (%r12),%eax
0x0000000000401173 <+50>:     sub     $0x1,%eax
0x0000000000401176 <+53>:     cmp     $0x5,%eax
0x0000000000401179 <+56>:     jbe     0x401180 <phase_6+63>
0x000000000040117b <+58>:     callq  0x401647 <explode_bomb>
0x0000000000401180 <+63>:     add     $0x1,%r13d
0x0000000000401184 <+67>:     cmp     $0x6,%r13d
0x0000000000401188 <+71>:     je      0x4011c7 <phase_6+134>
0x000000000040118a <+73>:     mov     %r13d,%ebx
0x000000000040118d <+76>:     movslq  %ebx,%rax
0x0000000000401190 <+79>:     mov     (%rsp,%rax,4),%eax
0x0000000000401193 <+82>:     cmp     %eax,0x0(%rbp)
0x0000000000401196 <+85>:     jne     0x40119d <phase_6+92>
0x0000000000401198 <+87>:     callq  0x401647 <explode_bomb>
0x000000000040119d <+92>:     add     $0x1,%ebx
0x00000000004011a0 <+95>:     cmp     $0x5,%ebx
0x00000000004011a3 <+98>:     jle     0x40118d <phase_6+76>
0x00000000004011a5 <+100>:    add     $0x4,%r12
0x00000000004011a9 <+104>:    jmp     0x40116c <phase_6+43>
0x00000000004011ab <+106>:    mov     0x8(%rdx),%rdx
0x00000000004011af <+110>:    add     $0x1,%eax
0x00000000004011b2 <+113>:    cmp     %ecx,%eax
```



```

0x000000000004011b4 <+115>: jne 0x4011ab <phase_6+106>
0x000000000004011b6 <+117>: mov %rdx,0x20(%rsp,%rsi,2)
0x000000000004011bb <+122>: add $0x4,%rsi
--Type <RET> for more, q to quit, c to continue without paging--c
0x000000000004011bf <+126>: cmp $0x18,%rsi
0x000000000004011c3 <+130>: jne 0x4011cc <phase_6+139>
0x000000000004011c5 <+132>: jmp 0x4011e0 <phase_6+159>
0x000000000004011c7 <+134>: mov $0x0,%esi
0x000000000004011cc <+139>: mov (%rsp,%rsi,1),%ecx
0x000000000004011cf <+142>: mov $0x1,%eax
0x000000000004011d4 <+147>: mov $0x6042f0,%edx
0x000000000004011d9 <+152>: cmp $0x1,%ecx
0x000000000004011dc <+155>: jg 0x4011ab <phase_6+106>
0x000000000004011de <+157>: jmp 0x4011b6 <phase_6+117>
0x000000000004011e0 <+159>: mov 0x20(%rsp),%rbx
0x000000000004011e5 <+164>: lea 0x20(%rsp),%rax
0x000000000004011ea <+169>: lea 0x48(%rsp),%rsi
0x000000000004011ef <+174>: mov %rbx,%rcx
0x000000000004011f2 <+177>: mov 0x8(%rax),%rdx
0x000000000004011f6 <+181>: mov %rdx,0x8(%rcx)
0x000000000004011fa <+185>: add $0x8,%rax
0x000000000004011fe <+189>: mov %rdx,%rcx
0x00000000000401201 <+192>: cmp %rsi,%rax
0x00000000000401204 <+195>: jne 0x4011f2 <phase_6+177>
0x00000000000401206 <+197>: movq $0x0,0x8(%rdx)
0x0000000000040120e <+205>: mov $0x5,%ebp
0x00000000000401213 <+210>: mov 0x8(%rbx),%rax
0x00000000000401217 <+214>: mov (%rax),%eax
0x00000000000401219 <+216>: cmp %eax,(%rbx)
0x0000000000040121b <+218>: jle 0x401222 <phase_6+225>
0x0000000000040121d <+220>: callq 0x401647 <explode_bomb>
0x00000000000401222 <+225>: mov 0x8(%rbx),%rbx
0x00000000000401226 <+229>: sub $0x1,%ebp
0x00000000000401229 <+232>: jne 0x401213 <phase_6+210>
0x0000000000040122b <+234>: mov 0x58(%rsp),%rax
0x00000000000401230 <+239>: xor %fs:0x28,%rax
0x00000000000401239 <+248>: je 0x401240 <phase_6+255>
0x0000000000040123b <+250>: callq 0x400b90 <__stack_chk_fail@plt>
0x00000000000401240 <+255>: add $0x68,%rsp

0x00000000000401244 <+259>: pop %rbx
0x00000000000401245 <+260>: pop %rbp
0x00000000000401246 <+261>: pop %r12
0x00000000000401248 <+263>: pop %r13
0x0000000000040124a <+265>: retq
End of assembler dump.

```

图 37-39: phase\_6 汇编代码

该段代码较长。观察分析后得知前半段代码逻辑大概如下：

调用函数 read\_six\_numbers，说明要输入六个整数；之后，利用循环判断输入的六个数字  $\leq 6$  且不相等。若不满足条件则炸弹爆炸。

## 2、查看链表结点

在观察过程中，注意到存在一个这样的地址：

```
0x000000000004011d4 <+147>: mov $0x6042f0,%edx
```

图 40: phase\_6 的 147 行

输入 x/64d 0x6042f0，可以查看该地址开始的 64 个字节的无符号表示数。我一开始使用的是 d 表示，但是分析到后面，我发现 x/64u 0x6042f0 方式显示的数字，更好运算，因为不会存



在负数值，而汇编代码中是以 unsigned 表示的。

```
0x6042f0 <node1>: "i\002"
(gdb) x/64d 0x6042f0
0x6042f0 <node1>: 105 2 0 0 1 0 0 0
0x6042f8 <node1+8>: 0 67 96 0 0 0 0 0
0x604300 <node2>: -98 0 0 0 2 0 0 0
0x604308 <node2+8>: 16 67 96 0 0 0 0 0
0x604310 <node3>: -82 1 0 0 3 0 0 0
0x604318 <node3+8>: 32 67 96 0 0 0 0 0
0x604320 <node4>: 84 0 0 0 4 0 0 0
0x604328 <node4+8>: 48 67 96 0 0 0 0 0
```

图 41: 有符号整数形式查看 0x6042f0 处内容

扩大了查看的范围，发现这个内存中存在六个 node。

```
(gdb) x/128u 0x6042f0
0x6042f0 <node1>: 105 2 0 0 1 0 0 0
0x6042f8 <node1+8>: 0 67 96 0 0 0 0 0
0x604300 <node2>: 158 0 0 0 2 0 0 0
0x604308 <node2+8>: 16 67 96 0 0 0 0 0
0x604310 <node3>: 174 1 0 0 3 0 0 0
0x604318 <node3+8>: 32 67 96 0 0 0 0 0
0x604320 <node4>: 84 0 0 0 4 0 0 0
0x604328 <node4+8>: 48 67 96 0 0 0 0 0
0x604330 <node5>: 37 2 0 0 5 0 0 0
0x604338 <node5+8>: 64 67 96 0 0 0 0 0
0x604340 <node6>: 246 0 0 0 6 0 0 0
0x604348 <node6+8>: 0 0 0 0 0 0 0 0
0x604350 <user_password>: 114 57 114 82 115 99 69 73
0x604358 <user_password+8>: 99 76 71 115 105 77 73 80
0x604360 <user_password+16>: 109 114 113 119 0 0 0 0
0x604368 <userid>: 50 48 50 50 50 49 49 48
```

图 42: 无符号整数形式查看 0x6042f0 处内容

每个 node 为 16 个字节，根据汇编代码内容，猜测第一个数据类型（前四个字节）为 int，即结点的数据域。后四个数字是 1-6，为结点的编号。之后的八个字节是结点的 next 指针。

因此，我们可以猜测该汇编代码中使用了链表的数据结构。

3、进一步分析

确定链表后，之后的代码内容理解起来就比较容易了。汇编代码将链表按照数据域大小进行了升序排序，需输出升序排序后的结点编号。

计算数据域大小：令图 42 第一列数字为 n1，第二列为 n2，value=n1+256^8，得到数据如下：

编号	1	2	3	4	5	6
数据域的值	617	158	430	84	549	246
升序排序顺序	5	2	4	1	6	3

因此，得知按升序排序的规律，应输入 4 2 6 3 1 5。

4、成功运行，拆除阶段六炸弹

尝试输入 4 2 6 3 1 5（升序排序结点），阶段六炸弹拆除成功。

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /students/2022211073/bomb102/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
In 2012, BUPT officially started the construction of new ShaHe campus.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 623
Halfway there!
132 4
So you got that one. Try this one.
qqqqqq
Good work! On to the next...
4 2 6 3 5 1
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
[Inferior 1 (process 1390799) exited normally]
```

图 43:拆除阶段 6 炸弹

## (八) 隐藏阶段

### 1、寻找进入方式

在准备阶段, 我们注意到了 phase\_defused 函数, 现在经过了一段时间的拆弹后我们可以对其进一步分析:

```
(gdb) disas phase_defused
Dump of assembler code for function phase_defused:
0x00000000004017e2 <+0>:    sub    $0x78,%rsp
0x00000000004017e6 <+4>:    mov    %fs:0x28,%rax
0x00000000004017ef <+13>:   mov    %rax,0x68(%rsp)
0x00000000004017f4 <+18>:   xor    %eax,%eax
0x00000000004017f6 <+20>:   mov    $0x1,%edi
0x00000000004017fb <+25>:   callq 0x40153d <send_msg>
0x0000000000401800 <+30>:   cmpl   $0x6,0x202fa5(%rip) # 0x6047ac <num_input_strings>
0x0000000000401807 <+37>:   jne    0x401876 <phase_defused+148>
0x0000000000401809 <+39>:   lea    0x10(%rsp),%r8
0x000000000040180e <+44>:   lea    0xc(%rsp),%rcx
0x0000000000401813 <+49>:   lea    0x8(%rsp),%rdx
0x0000000000401818 <+54>:   mov    $0x4029b7,%esi
0x000000000040181d <+59>:   mov    $0x6048b0,%edi
0x0000000000401822 <+64>:   mov    $0x0,%eax
0x0000000000401827 <+69>:   callq 0x400c40 <__isoc99_sscanf@plt>
0x000000000040182c <+74>:   cmp    $0x3,%eax
0x000000000040182f <+77>:   jne    0x401862 <phase_defused+128>
0x0000000000401831 <+79>:   mov    $0x4029c0,%esi
0x0000000000401836 <+84>:   lea    0x10(%rsp),%rdi
0x000000000040183b <+89>:   callq 0x401373 <strings_not_equal>
0x0000000000401840 <+94>:   test   %eax,%eax
0x0000000000401842 <+96>:   jne    0x401862 <phase_defused+128>
0x0000000000401844 <+98>:   mov    $0x402818,%edi
0x0000000000401849 <+103>:  callq 0x400b70 <puts@plt>
0x000000000040184e <+108>:  mov    $0x402840,%edi
0x0000000000401853 <+113>:  callq 0x400b70 <puts@plt>
0x0000000000401858 <+118>:  mov    $0x0,%eax
0x000000000040185d <+123>:  callq 0x401289 <secret_phase>
0x0000000000401862 <+128>:  mov    $0x402878,%edi
0x0000000000401867 <+133>:  callq 0x400b70 <puts@plt>
0x000000000040186c <+138>:  mov    $0x4028a8,%edi
0x0000000000401871 <+143>:  callq 0x400b70 <puts@plt>
0x0000000000401876 <+148>:  mov    0x68(%rsp),%rax
0x000000000040187b <+153>:  xor    %fs:0x28,%rax
0x0000000000401884 <+162>:  je     0x40188b <phase_defused+169>

0x0000000000401886 <+164>:  callq 0x400b90 <__stack_chk_fail@plt>
0x000000000040188b <+169>:  add    $0x78,%rsp
0x000000000040188f <+173>:  retq
--Type <RET> for more, q to quit, c to continue without paging--c
End of assembler dump.
```

图 44-45:phase\_defused 函数汇编代码

观察到 69 行的 sscanf 函数, 输入指令 x/s 0x4029b7, 像之前一样访问前几行相关的地址值。

```
(gdb) x/s 0x4029b7
0x4029b7:      "%d %d %s"
```

图 46: 0x4029b7 处字符串

提示我们在输入两个整数之后还要输入一个多余的字符串。因此我们可以尝试第三或第四关时多输入这个字符串。

在 89 行还存在一个字符串比较函数，输入指令 x/s 0x4029c0，访问前几行相关的地址值。

```
(gdb) x/s 0x4029c0
0x4029c0:      "DrEvil"
```

图 47: 0x4029c0 处字符串

得知需要输入的字符串为 DrEvil。

## 2、尝试进入隐藏阶段

第一次尝试时，我在第三关之后输入了该字符串，但是并没有跳转；第二次在第四关后输入，成功弹出隐藏关提示。

```
(gdb) run
Starting program: /students/2022211073/bomb102/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
In 2012, BUPT officially started the construction of new ShaHe campus.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 623 DrEvil
Halfway there!
132 4
So you got that one. Try this one.
qqqqqp
Good work! On to the next...
4 2 6 3 5 1
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
[Inferior 1 (process 1436877) exited normally]
```

图 48: 第一次尝试进入（失败）

```
(gdb) run
Starting program: /students/2022211073/bomb102/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
In 2012, BUPT officially started the construction of new ShaHe campus.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 623
Halfway there!
132 4 DrEvil
So you got that one. Try this one.
qqqqqp
Good work! On to the next...
4 2 6 3 5 1
Curses, you've found the secret phase!
But finding it and solving it are quite different...
^C
Program received signal SIGINT, Interrupt.
```

图 49: 第二次尝试进入（成功）

## 3、查看汇编代码与初步分析



输入命令 `disas secret_phase`，查看该函数的汇编代码。

```
(gdb) disas secret_phase
Dump of assembler code for function secret_phase:
0x0000000000401289 <+0>:    push    %rbx
0x000000000040128a <+1>:    callq   0x4016bc <read_line>
0x000000000040128f <+6>:    mov     $0xa,%edx
0x0000000000401294 <+11>:   mov     $0x0,%esi
0x0000000000401299 <+16>:   mov     %rax,%rdi
0x000000000040129c <+19>:   callq   0x400c20 <strtol@plt>
0x00000000004012a1 <+24>:   mov     %rax,%rbx
0x00000000004012a4 <+27>:   lea     -0x1(%rax),%eax
0x00000000004012a7 <+30>:   cmp     $0x3e8,%eax
0x00000000004012ac <+35>:   jbe     0x4012b3 <secret_phase+42>
0x00000000004012ae <+37>:   callq   0x401647 <explode_bomb>
0x00000000004012b3 <+42>:   mov     %ebx,%esi
0x00000000004012b5 <+44>:   mov     $0x604110,%edi
0x00000000004012ba <+49>:   callq   0x40124b <fun7>
0x00000000004012bf <+54>:   cmp     $0x1,%eax
0x00000000004012c2 <+57>:   je      0x4012c9 <secret_phase+64>
0x00000000004012c4 <+59>:   callq   0x401647 <explode_bomb>
0x00000000004012c9 <+64>:   mov     $0x402678,%edi
0x00000000004012ce <+69>:   callq   0x400b70 <puts@plt>
0x00000000004012d3 <+74>:   callq   0x4017e2 <phase_defused>
0x00000000004012d8 <+79>:   pop     %rbx
0x00000000004012d9 <+80>:   retq
End of assembler dump.
```

图 50: secret\_phase 汇编代码

发现内部调用了子函数 `fun7`，输入指令 `disas fun7` 查看。

```
(gdb) disas fun7
Dump of assembler code for function fun7:
0x000000000040124b <+0>:    sub     $0x8,%rsp
0x000000000040124f <+4>:    test    %rdi,%rdi
0x0000000000401252 <+7>:    je      0x40127f <fun7+52>
0x0000000000401254 <+9>:    mov     (%rdi),%edx
0x0000000000401256 <+11>:   cmp     %esi,%edx
0x0000000000401258 <+13>:   jle     0x401267 <fun7+28>
0x000000000040125a <+15>:   mov     0x8(%rdi),%rdi
0x000000000040125e <+19>:   callq   0x40124b <fun7>
0x0000000000401263 <+24>:   add     %eax,%eax
0x0000000000401265 <+26>:   jmp     0x401284 <fun7+57>
0x0000000000401267 <+28>:   mov     $0x0,%eax
0x000000000040126c <+33>:   cmp     %esi,%edx
0x000000000040126e <+35>:   je      0x401284 <fun7+57>
0x0000000000401270 <+37>:   mov     0x10(%rdi),%rdi
0x0000000000401274 <+41>:   callq   0x40124b <fun7>
0x0000000000401279 <+46>:   lea     0x1(%rax,%rax,1),%eax
0x000000000040127d <+50>:   jmp     0x401284 <fun7+57>
0x000000000040127f <+52>:   mov     $0xffffffff,%eax
0x0000000000401284 <+57>:   add     $0x8,%rsp
0x0000000000401288 <+61>:   retq
End of assembler dump.
```

图 51: fun7 汇编代码

#### 4、进一步分析 fun7 和 secret\_phase 函数

这是一个递归函数，观察分析可得

假设传入了一个指针 root 与需要比对的值 x:

如果 root 指向的值=x, 直接返回 0;

如果 root 指向的值>x, root+=4, 返回值%eax 需要\*2;

如果 root 指向的值<x, root+=8, 返回值%eax 需要\*2+1。

另外，在 secret\_phase 汇编代码中存在这样一个特殊的地址:

```
0x00000000004012b5 <+44>:    mov     $0x604110,%edi
```

图 52: secret\_phase 汇编代码第 44 行

输入指令: x 地址。由第 37 行汇编代码可知，每次偏移量为 10，读取内存中的值。

```
(gdb) x 6041600
0x5c3000: Cannot access memory at address 0x5c3000
(gdb) x 0x604110
0x604110 <n1>: 0x00000024
(gdb) x 0x604120
0x604120 <n1+16>: 0x00604150
(gdb) x 0x604130
0x604130 <n21>: 0x00000008
(gdb) x 0x604150
0x604150 <n22>: 0x00000032
(gdb) x 0x604170
0x604170 <n32>: 0x00000016
(gdb) x 0x604190
0x604190 <n33>: 0x0000002d
(gdb) x 0x6041b0
0x6041b0 <n31>: 0x00000006
(gdb) x 0x6041d0
0x6041d0 <n34>: 0x0000006b
(gdb) x 0x6041f0
0x6041f0 <n45>: 0x00000028
(gdb) x 0x604210
0x604210 <n41>: 0x00000001
(gdb) x 0x604230
0x604230 <n47>: 0x00000063
(gdb) x 0x604250
0x604250 <n44>: 0x00000023
(gdb) x 0x604270
0x604270 <n42>: 0x00000007
(gdb) x 0x604290
0x604290 <n43>: 0x00000014
(gdb) x 0x6042b0
0x6042b0 <n46>: 0x0000002f
(gdb) x 0x6042d0
0x6042d0 <n48>: 0x000000e9
(gdb) x 0x6042f0
0x6042f0 <node1>: 0x00000269
```

图 53: 0x604110 内存的值

观察可得这是一个数组，该存储结构应为一个完全二叉树。

经过 fun7 函数逆向递归求解，程序需要的输入为 50，可以满足所有过程中的不等式。

## 5、成功运行，拆除隐藏阶段炸弹

尝试输入 50（升序排序结点），隐藏阶段炸弹拆除成功。



```
Starting program: /students/2022211073/bomb102/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
In 2012, BUPT officially started the construction of new ShaHe campus.
Phase 1 defused. How about the next one?
0 1 3 6 10 15
That's number 2. Keep going!
0 623
Halfway there!
132 4 DrEvil
So you got that one. Try this one.
qqqqqp
Good work! On to the next...
4 2 6 3 5 1
Curses, you've found the secret phase!
But finding it and solving it are quite different...
50
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
[Inferior 1 (process 1500912) exited normally]
```

图 54: 拆除 secret 阶段炸弹

## 五、总结体会

## 六、诚信声明

在完成本次实验过程中，我曾分别与以下各位同学就以下方面做过交流：

此外，我还参考了以下资料：

在我提交的程序中，还在对应的位置以注释形式记录了具体的参考内容。

我独立完成了本次实验除以上方面之外的所有工作，包括分析、设计、编码、调试与测试。

我清楚地知道，从以上方面获得的信息在一定程度上降低了实验的难度，可能影响起评分。

我从未使用他人代码，不管是原封不动地复制，还是经过某些等价转换。

我未曾也不会向同一课程（包括此后各届）的同学复制或公开我这份程序的代码，我有义务妥善保管好它们。

我编写这个程序无意于破坏或妨碍任何计算机系统的正常运行。

我清楚地知道，以上情况均为本课程纪律所禁止，若违反，对应的实验成绩将按照 0 分计。

(签名)