Bomb Lab Writeup

2026-05-02 · Write Up · C, Programming, CSAPP

前言

> 邪恶的**邪恶博士**在我们的班级机器上放置了大量的“二进制炸弹”。二进制炸弹是由一系列阶段组成的程序。每个阶段都要求你在 stdin 上键入一个特定的字符串。如果你输入了正确的字符串，那么这个阶段就被**拆除**，炸弹进入下一个阶段。否则炸弹会**爆炸**，并打印出“BOOM!!!”，然后终止。当每一个阶段都被拆除时，炸弹才算拆除。
>
> 有太多炸弹要我们处理，所以我们给每个学生一个炸弹来拆除。这是你的任务，你别无选择，只能接受，就是在截止时间前拆除你的炸弹。祝你好运，欢迎加入拆弹小组！

开始

本 WriteUp 所有操作均在

```
Linux sakimidare-arch 7.0.2-arch1-1-lily #1 SMP PREEMPT_DYNAMIC Thu, 30 Apr 2026 05:29:46 +0000 x86_64 GNU/Linux
```

上进行。

我拿到了一个 `bomb.tar`，目录结构如下：

```
.
├── bomb
├── bomb.c
└── README

1 directory, 3 files
```

我们的目标是尽可能多地拆除炸弹（获取正确的拆弹密钥）。

首先来看 `bomb.c`：

```
/***************************************************************************
 * 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
 * spectaculary 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;
}
```

`support.h` 和 `phases.h` 并没有提供，源码中的 `initialize_bomb()`，`read_line()`，`phase_x()`，`phase_defused()` 未提供定义。这就需要我们用反汇编推出这几个函数内部的逻辑并猜出密钥。

推测原型:

```
void initialize_bomb();
char* read_line();
```

```
void phase_x(char*);
void phase_defused();
```

反汇编

首先用 `readelf` 看一眼这个 ELF 文件的布局：

```
readelf -a bomb
ELF 头:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  类别:                              ELF64
  数据:                              2 补码，小端序 (little endian)
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI 版本:                          0
  类型:                              EXEC (可执行文件)
  系统架构:                          Advanced Micro Devices X86-64
  版本:                              0x1
  入口点地址:               0x400c90
  程序头起点:               64 (bytes into file)
  Start of section headers:          18616 (bytes into file)
  标志:             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         36
  Section header string table index: 33

节头:
  [号] 名称              类型             地址              偏移量
       大小              全体大小         旗标    链接    信息    对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.bu[...] NOTE             0000000000400274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000400298  00000298
       0000000000000030  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000004002c8  000002c8
       0000000000000300  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           00000000004005c8  000005c8
       000000000000016d  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           0000000000400736  00000736
       0000000000000040  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000400778  00000778
       0000000000000060  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             00000000004007d8  000007d8
       0000000000000060  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000400838  00000838
```

```
     0000000000000288  0000000000000018   A       5    12     8
  [11] .init             PROGBITS         0000000000400ac0  00000ac0
     000000000000000e  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000400ad0  00000ad0
     00000000000001c0  0000000000000010  AX       0     0     16
  [13] .text             PROGBITS         0000000000400c90  00000c90
     0000000000001614  0000000000000000  AX       0     0     16
  [14] .fini             PROGBITS         00000000004022a4  000022a4
     0000000000000009  0000000000000000  AX       0     0     4
  [15] .rodata           PROGBITS         00000000004022b0  000022b0
     00000000000004e5  0000000000000000   A       0     0     16
  [16] .eh_frame_hdr     PROGBITS         0000000000402798  00002798
     0000000000000104  0000000000000000   A       0     0     4
  [17] .eh_frame         PROGBITS         00000000004028a0  000028a0
     0000000000000454  0000000000000000   A       0     0     8
  [18] .init_array       INIT_ARRAY       0000000000602df8  00002df8
     0000000000000008  0000000000000000  WA       0     0     8
  [19] .fini_array       FINI_ARRAY       0000000000602e00  00002e00
     0000000000000008  0000000000000000  WA       0     0     8
  [20] .jcr              PROGBITS         0000000000602e08  00002e08
     0000000000000008  0000000000000000  WA       0     0     8
  [21] .dynamic          DYNAMIC          0000000000602e10  00002e10
     00000000000001d0  0000000000000010  WA       6     0     8
  [22] .got              PROGBITS         0000000000602fe0  00002fe0
     0000000000000008  0000000000000008  WA       0     0     8
  [23] .got.plt          PROGBITS         0000000000602fe8  00002fe8
     00000000000000f0  0000000000000008  WA       0     0     8
  [24] .data             PROGBITS         00000000006030e0  000030e0
     0000000000000660  0000000000000000  WA       0     0     32
  [25] .bss              NOBITS           0000000000603740  00003740
     00000000000006d0  0000000000000000  WA       0     0     32
  [26] .comment          PROGBITS         0000000000000000  00003740
     0000000000000053  0000000000000001  MS       0     0     1
  [27] .debug_aranges    PROGBITS         0000000000000000  00003793
     0000000000000030  0000000000000000           0     0     1
  [28] .debug_info       PROGBITS         0000000000000000  000037c3
     00000000000007a3  0000000000000000           0     0     1
  [29] .debug_abbrev     PROGBITS         0000000000000000  00003f66
     000000000000021f  0000000000000000           0     0     1
  [30] .debug_line       PROGBITS         0000000000000000  00004185
     0000000000000161  0000000000000000           0     0     1
  [31] .debug_str        PROGBITS         0000000000000000  000042e6
     00000000000002f3  0000000000000001  MS       0     0     1
  [32] .debug_loc        PROGBITS         0000000000000000  000045d9
     0000000000000188  0000000000000000           0     0     1
  [33] .shstrtab         STRTAB           0000000000000000  00004761
     0000000000000153  0000000000000000           0     0     1
  [34] .symtab           SYMTAB           0000000000000000  000051b8
     0000000000000eb8  0000000000000018          35    57     8
  [35] .strtab           STRTAB           0000000000000000  00006070
     00000000000006b6  0000000000000000           0     0     1
Key to Flags:
```

```
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)

There are no section groups in this file.
```

对我们有用的部分是 `.text` 和 `.rodata` 段。前者保存机器码，后者保存用到的常量（包括字符串）。

`initialize_bomb()`

```
objdump -d bomb -disassemble=initialize_bomb
objdump -d bomb --disassemble=initialize_bomb

bomb:     文件格式 elf64-x86-64

Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:

00000000004013a2 <initialize_bomb>:
  4013a2:       48 83 ec 08             sub    $0x8,%rsp
  4013a6:       be a0 12 40 00          mov    $0x4012a0,%esi
  4013ab:       bf 02 00 00 00          mov    $0x2,%edi
  4013b0:       e8 db f7 ff ff          call   400b90 <signal@plt>
  4013b5:       48 83 c4 08             add    $0x8,%rsp
  4013b9:       c3                      ret

Disassembly of section .fini:
```

`%edi` `%esi` 按照惯例来讲应该是函数的两个参数。`%rsp` 是栈指针，指向当前栈顶。上述汇编从栈中弹出一个四字（不用 `popq` 大概是为了不污染 `%rax`），并赋了两个魔数给两个寄存器，并调用了 `400b90` 这个地方的函数 `<signal@plt>`。我们继续去反编译 `signal`。

```
0000000000400b90 <signal@plt>:
  400b90:       ff 25 c2 24 20 00       jmp    *0x2024c2(%rip)        # 603058
<signal@GLIBC_2.2.5>
  400b96:       68 0b 00 00 00          push   $0xb
  400b9b:       e9 30 ff ff ff          jmp    400ad0 <.plt>

0000000000400ad0 <.plt>:
  400ad0:       ff 35 1a 25 20 00       push   0x20251a(%rip)        # 602ff0
<_GLOBAL_OFFSET_TABLE_+0x8>
  400ad6:       ff 25 1c 25 20 00       jmp    *0x20251c(%rip)        # 602ff8
<_GLOBAL_OFFSET_TABLE_+0x10>
  400adc:       0f 1f 40 00             nopl   0x0(%rax)
```

坏事了，这个函数并不在我们的二进制文件里，而是动态链接到了 glibc 里面的 `signal` 函数。

查阅相关资料，发现 `signal` 函数原型如下：

```
#include <signal.h>

typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

代入`%edi`，`%esi`，发现调用的函数是：

```
sighandler_t signal(0x2, 0x4012a0);
```

通过查表：

```
$ man signal

SIGNAL(7)                                                  Linux Programmer's Manual
SIGNAL(7)

NAME
     signal - 有效信号的清单

描述 (DESCRIPTION)
     下面 列出 Linux 支持的 信号. 某些 信号 依赖于 体系结构(architecture).

     首先, POSIX.1 描述了 下列 信号.

     信号          值       动作    说明
     ──────────────────────────────────────────────────────────────────────
     SIGHUP         1        A     在控制终端上是挂起信号, 或者控制进程结束
     SIGINT         2        A     从键盘输入的中断
     SIGQUIT        3        C     从键盘输入的退出
     SIGILL         4        C     无效硬件指令
     SIGABRT        6        C     非正常终止, 可能来自 abort(3)
     SIGFPE         8        C     浮点运算例外
     SIGKILL        9       AEF    杀死进程信号
     SIGSEGV       11        C     无效的内存引用
     SIGPIPE       13        A     管道中止: 写入无人读取的管道
     SIGALRM       14        A     来自 alarm(2) 的超时信号
     SIGTERM       15        A     终止信号
     SIGUSR1   30,10,16      A     用户定义的信号 1
     SIGUSR2   31,12,17      A     用户定义的信号 2
     SIGCHLD   20,17,18      B     子进程结束或停止
     SIGCONT   19,18,25            继续停止的进程
     SIGSTOP   17,19,23     DEF    停止进程
     SIGTSTP   18,20,24      D     终端上发出的停止信号
     SIGTTIN   21,21,26      D     后台进程试图从控制终端(tty)输入
     SIGTTOU   22,22,27      D     后台进程试图在控制终端(tty)输出
...
```

发现`0x2`对应`SIGINT`。

`SIGFPE`是键盘上按下`Ctrl`+`C`时 Linux 向程序发送的信号。上面的函数捕获了这个信号并注册了一个函数，地址在`0x4012a0`。通过

```
typedef void (*sighandler_t)(int);
```

我们还能知道函数钩子的原型是`void foo(int);`。

接下来反汇编`0x4012a0`处的函数：

```
$ objdump -d --start-address=0x4012a0 --stop-address=0x4013a0 bomb

bomb:     文件格式 elf64-x86-64
```

```
Disassembly of section .text:

00000000004012a0 <sig_handler>:
  4012a0:       48 83 ec 08             sub    $0x8,%rsp
  4012a4:       bf c0 24 40 00          mov    $0x4024c0,%edi
  4012a9:       e8 62 f8 ff ff          call   400b10 <puts@plt>
  4012ae:       bf 03 00 00 00          mov    $0x3,%edi
  4012b3:       e8 98 f9 ff ff          call   400c50 <sleep@plt>
  4012b8:       be 82 25 40 00          mov    $0x402582,%esi
  4012bd:       bf 01 00 00 00          mov    $0x1,%edi
  4012c2:       b8 00 00 00 00          mov    $0x0,%eax
  4012c7:       e8 34 f9 ff ff          call   400c00 <__printf_chk@plt>
  4012cc:       48 8b 3d 6d 24 20 00    mov    0x20246d(%rip),%rdi        # 603740
<stdout@GLIBC_2.2.5>
  4012d3:       e8 08 f9 ff ff          call   400be0 <fflush@plt>
  4012d8:       bf 01 00 00 00          mov    $0x1,%edi
  4012dd:       e8 6e f9 ff ff          call   400c50 <sleep@plt>
  4012e2:       bf 8a 25 40 00          mov    $0x40258a,%edi
  4012e7:       e8 24 f8 ff ff          call   400b10 <puts@plt>
  4012ec:       bf 10 00 00 00          mov    $0x10,%edi
  4012f1:       e8 2a f9 ff ff          call   400c20 <exit@plt>
```

这个函数大量调用了 `puts()` `sleep()` `fflush()` `exit()` 等函数，猜测与炸弹密钥无关。但为了保险，我们先看看 `puts()` 都输出了什么东西。

查看 `0x4024c0` `0x400b10` 对应的汇编：

```
$ objdump -s -j .rodata bomb | grep -A 5 "4024c0"
 4024c0 536f2079 6f752074 68696e6b 20796f75  So you think you
 4024d0 2063616e 2073746f 70207468 6520626f   can stop the bo
 4024e0 6d622077 69746820 6374726c 2d632c20  mb with ctrl-c,
 4024f0 646f2079 6f753f00 43757273 65732c20  do you?.Curses,
 402500 796f7527 76652066 6f756e64 20746865  you've found the
 402510 20736563 72657420 70686173 65210000   secret phase!..
```

好吧确实没啥关系……不管了我们接着看下一个函数。

`phase_1()`

```
$ objdump -d bomb --disassemble=phase_1

bomb:     文件格式 elf64-x86-64

Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:

0000000000400ee0 <phase_1>:
  400ee0:       48 83 ec 08             sub    $0x8,%rsp
  400ee4:       be 00 24 40 00          mov    $0x402400,%esi
  400ee9:       e8 4a 04 00 00          call   401338 <strings_not_equal>
  400eee:       85 c0                   test   %eax,%eax
```

```
  400ef0:       74 05                   je     400ef7 <phase_1+0x17>
  400ef2:       e8 43 05 00 00          call   40143a <explode_bomb>
  400ef7:       48 83 c4 08             add    $0x8,%rsp
  400efb:       c3                      ret

Disassembly of section .fini:
```

查看 `0x402400` 处的汇编：

```
$ objdump -s -j .rodata bomb | grep -A 10 "402400"
 402400 426f7264 65722072 656c6174 696f6e73  Border relations
 402410 20776974 68204361 6e616461 20686176   with Canada hav
 402420 65206e65 76657220 6265656e 20626574  e never been bet
 402430 7465722e 00000000 576f7721 20596f75  ter.....Wow! You
 402440 27766520 64656675 73656420 74686520  've defused the
 402450 73656372 65742073 74616765 2100666c  secret stage!.fl
 402460 79657273 00000000 00000000 00000000  yers............
 402470 7c0f4000 00000000 b90f4000 00000000  |.@.......@.....
 402480 830f4000 00000000 8a0f4000 00000000  ..@.......@.....
 402490 910f4000 00000000 980f4000 00000000  ..@.......@.....
 4024a0 9f0f4000 00000000 a60f4000 00000000  ..@.......@.....
```

注意这一行：

```
 402430 7465722e 00000000 576f7721 20596f75  ter.....Wow! You
```

`00` 标志着 C 风格字符串的结尾。也就是说，这个字符串的内容是：

```
Border relations with Canada have never been better.
```

查看 `401338` 的汇编：

```
objdump -d bomb --start-address=0x401338 --stop-address=0x401400

bomb:     文件格式 elf64-x86-64

Disassembly of section .text:

0000000000401338 <strings_not_equal>:
  401338:       41 54                   push   %r12
  40133a:       55                      push   %rbp
  40133b:       53                      push   %rbx
  40133c:       48 89 fb                mov    %rdi,%rbx
  40133f:       48 89 f5                mov    %rsi,%rbp
  401342:       e8 d4 ff ff ff          call   40131b <string_length>
  401347:       41 89 c4                mov    %eax,%r12d
  40134a:       48 89 ef                mov    %rbp,%rdi
  40134d:       e8 c9 ff ff ff          call   40131b <string_length>
  401352:       ba 01 00 00 00          mov    $0x1,%edx
  401357:       41 39 c4                cmp    %eax,%r12d
  40135a:       75 3f                   jne    40139b <strings_not_equal+0x63>
  40135c:       0f b6 03                movzbl (%rbx),%eax
  40135f:       84 c0                   test   %al,%al
  401361:       74 25                   je     401388 <strings_not_equal+0x50>
  401363:       3a 45 00                cmp    0x0(%rbp),%al
  401366:       74 0a                   je     401372 <strings_not_equal+0x3a>
  401368:       eb 25                   jmp    40138f <strings_not_equal+0x57>
  40136a:       3a 45 00                cmp    0x0(%rbp),%al
```

```
  40136d:       0f 1f 00                nopl   (%rax)
  401370:       75 24                   jne    401396 <strings_not_equal+0x5e>
  401372:       48 83 c3 01             add    $0x1,%rbx
  401376:       48 83 c5 01             add    $0x1,%rbp
  40137a:       0f b6 03                movzbl (%rbx),%eax
  40137d:       84 c0                   test   %al,%al
  40137f:       75 e9                   jne    40136a <strings_not_equal+0x32>
  401381:       ba 00 00 00 00          mov    $0x0,%edx
  401386:       eb 13                   jmp    40139b <strings_not_equal+0x63>
  401388:       ba 00 00 00 00          mov    $0x0,%edx
  40138d:       eb 0c                   jmp    40139b <strings_not_equal+0x63>
  40138f:       ba 01 00 00 00          mov    $0x1,%edx
  401394:       eb 05                   jmp    40139b <strings_not_equal+0x63>
  401396:       ba 01 00 00 00          mov    $0x1,%edx
  40139b:       89 d0                   mov    %edx,%eax
  40139d:       5b                      pop    %rbx
  40139e:       5d                      pop    %rbp
  40139f:       41 5c                   pop    %r12
  4013a1:       c3                      ret
```

……

怎么这么长呀！

猜测一手，上面那个函数的意思是把输入进去的内容和先前看到的字符串做比对，如果相同表明拆弹成功。到底是不是这样呢？

```
$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
```

嘿嘿，拆弹成功！

phase_2()

```
objdump -d bomb --disassemble=phase_2

bomb:     文件格式 elf64-x86-64

Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:

0000000000400efc <phase_2>:
  400efc:       55                      push   %rbp
  400efd:       53                      push   %rbx
  400efe:       48 83 ec 28             sub    $0x28,%rsp
  400f02:       48 89 e6                mov    %rsp,%rsi
  400f05:       e8 52 05 00 00          call   40145c <read_six_numbers>
  400f0a:       83 3c 24 01             cmpl   $0x1,(%rsp)
  400f0e:       74 20                   je     400f30 <phase_2+0x34>
  400f10:       e8 25 05 00 00          call   40143a <explode_bomb>
```

```
  400f15:       eb 19                   jmp    400f30 <phase_2+0x34>
  400f17:       8b 43 fc                mov    -0x4(%rbx),%eax
  400f1a:       01 c0                   add    %eax,%eax
  400f1c:       39 03                   cmp    %eax,(%rbx)
  400f1e:       74 05                   je     400f25 <phase_2+0x29>
  400f20:       e8 15 05 00 00          call   40143a <explode_bomb>
  400f25:       48 83 c3 04             add    $0x4,%rbx
  400f29:       48 39 eb                cmp    %rbp,%rbx
  400f2c:       75 e9                   jne    400f17 <phase_2+0x1b>
  400f2e:       eb 0c                   jmp    400f3c <phase_2+0x40>
  400f30:       48 8d 5c 24 04          lea    0x4(%rsp),%rbx
  400f35:       48 8d 6c 24 18          lea    0x18(%rsp),%rbp
  400f3a:       eb db                   jmp    400f17 <phase_2+0x1b>
  400f3c:       48 83 c4 28             add    $0x28,%rsp
  400f40:       5b                      pop    %rbx
  400f41:       5d                      pop    %rbp
  400f42:       c3                      ret

Disassembly of section .fini:
```

这个函数开始出现了比较多的跳转指令，猜测由 `if` 和 `while` 编译而来。

`<read_six_numbers>`

先去看看 `40145c` 的 `read_six_numbers` 函数吧，看看读入的六个数字存到哪里去。

```
objdump -d bomb --start-address=0x40145c --stop-address=0x40155c

bomb:     文件格式 elf64-x86-64

Disassembly of section .text:

000000000040145c <read_six_numbers>:
  40145c:       48 83 ec 18             sub    $0x18,%rsp
  401460:       48 89 f2                mov    %rsi,%rdx
  401463:       48 8d 4e 04             lea    0x4(%rsi),%rcx
  401467:       48 8d 46 14             lea    0x14(%rsi),%rax
  40146b:       48 89 44 24 08          mov    %rax,0x8(%rsp)
  401470:       48 8d 46 10             lea    0x10(%rsi),%rax
  401474:       48 89 04 24             mov    %rax,(%rsp)
  401478:       4c 8d 4e 0c             lea    0xc(%rsi),%r9
  40147c:       4c 8d 46 08             lea    0x8(%rsi),%r8
  401480:       be c3 25 40 00          mov    $0x4025c3,%esi
  401485:       b8 00 00 00 00          mov    $0x0,%eax
  40148a:       e8 61 f7 ff ff          call   400bf0 <__isoc99_sscanf@plt>
  40148f:       83 f8 05                cmp    $0x5,%eax
  401492:       7f 05                   jg     401499 <read_six_numbers+0x3d>
  401494:       e8 a1 ff ff ff          call   40143a <explode_bomb>
  401499:       48 83 c4 18             add    $0x18,%rsp
  40149d:       c3                      ret
```

我不行了这都啥啊！

看一眼 `sscanf()` 的原型：

```
int sscanf(const char *str, const char *format, ...);
```

发现不同于直接从 stdin 读取的 `scanf`，他接收 n+2 = 8 个参数。

已知 x86-64 汇编中函数传参的顺序是 `%rdi` `%rsi` `%rdx` `%rcx` `%r8` `%r9`，而我们读入六个数字需要传给 `sscanf()` 8 个参数。还余下两个怎么办？经过查找资料发现会从栈里从顶到底挨个读取数据。

第二个参数是 `%rsi`。盲猜一手 `0x4025c3` 那边存的应该是类似 `%d %d %d %d %d %d` 的东西。

```
4025c0 702e0025 64202564 20256420 25642025  p..%d %d %d %d %
4025d0 64202564 00457272 6f723a20 5072656d  d %d.Error: Prem
```

那问题来了，`%rdi` 在哪？

既然是我输入的字符串，我有理由猜测输入的字符串原封不动地躺在 `%edi` 里边 ~~大概吧 ()~~

这个地方是我蒙的不小心蒙对了（）

注意到有几行汇编非常有意思：

```
400efe:       48 83 ec 28             sub    $0x28,%rsp
400f02:       48 89 e6                mov    %rsp,%rsi
400f05:       e8 52 05 00 00          call   40145c <read_six_numbers>
...
40145c:       48 83 ec 18             sub    $0x18,%rsp
```

- `400efe` 行：开辟 `0x28` 字节大小空间；
- `400f02` 行：给当前栈指针做个备份，赋给 `%rsi`；
- `40145c` 行：开辟 `0x18` 字节大小空间。

```
401460:       48 89 f2                mov    %rsi,%rdx
401463:       48 8d 4e 04             lea    0x4(%rsi),%rcx
401478:       4c 8d 4e 0c             lea    0xc(%rsi),%r9
40147c:       4c 8d 46 08             lea    0x8(%rsi),%r8
401480:       be c3 25 40 00          mov    $0x4025c3,%esi
```

这几行涉及内存地址的计算。可以看到，所有参数对应的内存偏移量都是以 `%rsi`（也就是刚刚备份过的 `%rsp`）为基准的。`sscanf` 写入的所有数据都保存在 `phase_2` 的栈帧中。

```
401467:       48 8d 46 14             lea    0x14(%rsi),%rax
40146b:       48 89 44 24 08          mov    %rax,0x8(%rsp)
401470:       48 8d 46 10             lea    0x10(%rsi),%rax
401474:       48 89 04 24             mov    %rax,(%rsp)
```

这几行不同于上文的 `lea 0x...(%rsi) %寄存器` 是因为给参数使用的寄存器用完了，不得不从栈里读数据。这里的栈指针为分配好 `0x18` 字节之后的指针，确保不会覆盖之前读取好的数据。

于是我们就有了下面一张表：

（对不起我用 AI 整理了一下实在不好意思…）

次序	参数意义	存放位置	具体数据	汇编证据
1	输入字符串	`%rdi`	`(%rdi)`	(外部传入，直接透传)
2	格式化字符串	`%rsi`	`0x4025c3`	`mov $0x4025c3, %esi`
3	数字 1 的地址	`%rdx`	`(%rsi)`	`mov %rsi, %rdx`
4	数字 2 的地址	`%rcx`	`0x4(%rsi)`	`lea 0x4(%rsi), %rcx`
5	数字 3 的地址	`%r8`	`0x8(%rsi)`	`lea 0x8(%rsi), %r8`
6	数字 4 的地址	`%r9`	`0xc(%rsi)`	`lea 0xc(%rsi), %r9`
7	数字 5 的地址	`(%rsp)`	`0x10(%rsi)`	`lea 0x10(%rsi),%rax` `mov %rax, (%rsp)`
8	数字 6 的地址	`0x8(%rsp)`	`0x14(%rsi)`	`lea 0x14(%rsi),%rax` `mov %rax, 0x8(%rsp)`

注意到了 `401492` 的比较跳转指令正好跳过了 `401494` 的爆炸函数，所以我们要想办法满足 `jg` 的条件。

```
  40148a:       e8 61 f7 ff ff          call   400bf0 <__isoc99_sscanf@plt>
  40148f:       83 f8 05                cmp    $0x5,%eax
  401492:       7f 05                   jg     401499 <read_six_numbers+0x3d>
```

我们来查阅 `sscanf` 的返回值：

```
RETURN VALUE
  On success, these functions return the number of input items
  successfully matched and assigned; this can be fewer than provided
  for, or even zero, in the event of an early matching failure.

  The value EOF is returned if the end of input is reached before
  either the first successful conversion or a matching failure
  occurs.
```

发现返回值 （也就是 `%eax` 里的值）是解析出的输入个数。之后 `40148f` 行将返回值与 5 进行比较，如果返回值大于 5，就跳过炸弹。

这个操作等效于以下 C 代码：

```c
if(sscanf("%d %d %d %d %d %d", &a1, &a2, &a3, &a4, &a5, &a6) ⩽ 5) {
    explode_bomb();
}
```

于是我们顺利分析了 `read_six_numbers` 的行为。

循环与跳转

调用完 `read_six_numbers` 函数后，紧接着 `phase_2` 做了以下行为：

```
  400f0a:       83 3c 24 01             cmpl   $0x1,(%rsp)
  400f0e:       74 20                   je     400f30 <phase_2+0x34>
  400f10:       e8 25 05 00 00          call   40143a <explode_bomb>
  400f15:       eb 19                   jmp    400f30 <phase_2+0x34>
  400f17:       8b 43 fc                mov    -0x4(%rbx),%eax
  400f1a:       01 c0                   add    %eax,%eax
  400f1c:       39 03                   cmp    %eax,(%rbx)
  400f1e:       74 05                   je     400f25 <phase_2+0x29>
  400f20:       e8 15 05 00 00          call   40143a <explode_bomb>
  400f25:       48 83 c3 04             add    $0x4,%rbx
  400f29:       48 39 eb                cmp    %rbp,%rbx
  400f2c:       75 e9                   jne    400f17 <phase_2+0x1b>
  400f2e:       eb 0c                   jmp    400f3c <phase_2+0x40>
  400f30:       48 8d 5c 24 04          lea    0x4(%rsp),%rbx
  400f35:       48 8d 6c 24 18          lea    0x18(%rsp),%rbp
  400f3a:       eb db                   jmp    400f17 <phase_2+0x1b>
  400f3c:       48 83 c4 28             add    $0x28,%rsp
  400f40:       5b                      pop    %rbx
  400f41:       5d                      pop    %rbp
  400f42:       c3                      ret
```

```
  400f0a:       83 3c 24 01             cmpl   $0x1,(%rsp)
  400f0e:       74 20                   je     400f30 <phase_2+0x34>
  400f10:       e8 25 05 00 00          call   40143a <explode_bomb>
  400f15:       eb 19                   jmp    400f30 <phase_2+0x34>
```

如果 `(%rsp)` 不是 `1`，直接引爆炸弹。之后跳转到 `400f30` 段落。

```
  400f30:       48 8d 5c 24 04          lea    0x4(%rsp),%rbx
  400f35:       48 8d 6c 24 18          lea    0x18(%rsp),%rbp
  400f3a:       eb db                   jmp    400f17 <phase_2+0x1b>
```

这时候遇到了 `0x...(%rsp)`！时刻记住上面推导出来的那张表，注意所有的 `%rsi` 均需换成 `%rsp`，因为栈帧已还原。

上述汇编做了个什么事呢？首先 `%rbx` 被赋予了 `%rsp + 0x4`。也就是**第二个数字的地址**。然后，`%rbp` 被赋予了 `%rsp + 0x18`，查表发现，刚好是最后一个数字**之后的地址**。随后跳转到 `400f17` 开始执行指令。

```
  400f17:       8b 43 fc                mov    -0x4(%rbx),%eax
  400f1a:       01 c0                   add    %eax,%eax
  400f1c:       39 03                   cmp    %eax,(%rbx)
  400f1e:       74 05                   je     400f25 <phase_2+0x29>
  400f20:       e8 15 05 00 00          call   40143a <explode_bomb>
  400f25:       48 83 c3 04             add    $0x4,%rbx
  400f29:       48 39 eb                cmp    %rbp,%rbx
  400f2c:       75 e9                   jne    400f17 <phase_2+0x1b>
```

- `400f17`：`%eax` 赋值为 `%rbx` 的 **前一个数字**；
- `f00f1a`：将 `%eax` 翻倍；
- `400f1c`：比较翻倍后的前一个数字与当前数字；
- `400f1e`：如果相等，跳过炸弹，否则引爆炸弹；
- `400f25`：`%rbx` 增加 `0x4`，指向**后一个数字**；
- `400f29`：检查 `%rbx` 是否与 `%rbp` 相等（指针是否指向了末尾）；
- `400f2c`：如果没有指向末尾，重复执行上述操作。

这不就是经典的 `for` 循环吗！

```
for(long *p = &a2; p < &a6 + 1; p++) {
    if(*(p - 1) * 2 != *p) explode_bomb();
}
```

显然可以看出，`phase_2` 是一个检查输入 **是否为等比数列** 的函数，初值为 `1`，公比是 `2`，项数为 `6`！

```
$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
```

`phase_3()`

```
objdump -d bomb --disassemble=phase_3

bomb:     文件格式 elf64-x86-64

Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:
```

```
0000000000400f43 <phase_3>:
  400f43:       48 83 ec 18             sub    $0x18,%rsp
  400f47:       48 8d 4c 24 0c          lea    0xc(%rsp),%rcx
  400f4c:       48 8d 54 24 08          lea    0x8(%rsp),%rdx
  400f51:       be cf 25 40 00          mov    $0x4025cf,%esi
  400f56:       b8 00 00 00 00          mov    $0x0,%eax
  400f5b:       e8 90 fc ff ff          call   400bf0 <__isoc99_sscanf@plt>
  400f60:       83 f8 01                cmp    $0x1,%eax
  400f63:       7f 05                   jg     400f6a <phase_3+0x27>
  400f65:       e8 d0 04 00 00          call   40143a <explode_bomb>
  400f6a:       83 7c 24 08 07          cmpl   $0x7,0x8(%rsp)
  400f6f:       77 3c                   ja     400fad <phase_3+0x6a>
  400f71:       8b 44 24 08             mov    0x8(%rsp),%eax
  400f75:       ff 24 c5 70 24 40 00    jmp    *0x402470(,%rax,8)
  400f7c:       b8 cf 00 00 00          mov    $0xcf,%eax
  400f81:       eb 3b                   jmp    400fbe <phase_3+0x7b>
  400f83:       b8 c3 02 00 00          mov    $0x2c3,%eax
  400f88:       eb 34                   jmp    400fbe <phase_3+0x7b>
  400f8a:       b8 00 01 00 00          mov    $0x100,%eax
  400f8f:       eb 2d                   jmp    400fbe <phase_3+0x7b>
  400f91:       b8 85 01 00 00          mov    $0x185,%eax
  400f96:       eb 26                   jmp    400fbe <phase_3+0x7b>
  400f98:       b8 ce 00 00 00          mov    $0xce,%eax
  400f9d:       eb 1f                   jmp    400fbe <phase_3+0x7b>
  400f9f:       b8 aa 02 00 00          mov    $0x2aa,%eax
  400fa4:       eb 18                   jmp    400fbe <phase_3+0x7b>
  400fa6:       b8 47 01 00 00          mov    $0x147,%eax
  400fab:       eb 11                   jmp    400fbe <phase_3+0x7b>
  400fad:       e8 88 04 00 00          call   40143a <explode_bomb>
  400fb2:       b8 00 00 00 00          mov    $0x0,%eax
  400fb7:       eb 05                   jmp    400fbe <phase_3+0x7b>
  400fb9:       b8 37 01 00 00          mov    $0x137,%eax
  400fbe:       3b 44 24 0c             cmp    0xc(%rsp),%eax
  400fc2:       74 05                   je     400fc9 <phase_3+0x86>
  400fc4:       e8 71 04 00 00          call   40143a <explode_bomb>
  400fc9:       48 83 c4 18             add    $0x18,%rsp
  400fcd:       c3                      ret
```

又是经典 `sscanf`。看看格式化字符串：

```
 4025c0 702e0025 64202564 20256420 25642025  p..%d %d %d %d %
 4025d0 64202564 00457272 6f723a20 5072656d  d %d.Error: Prem
```

竟然复用 `phase_2` 的字符串！但是起始位置不同。这次的格式化字符串是

```
%d %d
```

意味着我需要输入俩数字，存到 `(%rdx)` 和 `(%rcx)`（也就是 `0x8(%rsp)`，`0xc(%rsp)`）。

根据 `phase_2` 的经验，我们直接来看这段：

```
  400f6a:       83 7c 24 08 07          cmpl   $0x7,0x8(%rsp)
  400f6f:       77 3c                   ja     400fad <phase_3+0x6a>
  400f71:       8b 44 24 08             mov    0x8(%rsp),%eax
  400f75:       ff 24 c5 70 24 40 00    jmp    *0x402470(,%rax,8)
  400f7c:       b8 cf 00 00 00          mov    $0xcf,%eax
  400f81:       eb 3b                   jmp    400fbe <phase_3+0x7b>
```

```
  400f83:       b8 c3 02 00 00          mov    $0x2c3,%eax
  400f88:       eb 34                   jmp    400fbe <phase_3+0x7b>
  400f8a:       b8 00 01 00 00          mov    $0x100,%eax
  400f8f:       eb 2d                   jmp    400fbe <phase_3+0x7b>
  400f91:       b8 85 01 00 00          mov    $0x185,%eax
  400f96:       eb 26                   jmp    400fbe <phase_3+0x7b>
  400f98:       b8 ce 00 00 00          mov    $0xce,%eax
  400f9d:       eb 1f                   jmp    400fbe <phase_3+0x7b>
  400f9f:       b8 aa 02 00 00          mov    $0x2aa,%eax
  400fa4:       eb 18                   jmp    400fbe <phase_3+0x7b>
  400fa6:       b8 47 01 00 00          mov    $0x147,%eax
  400fab:       eb 11                   jmp    400fbe <phase_3+0x7b>
  400fad:       e8 88 04 00 00          call   40143a <explode_bomb>
  400fb2:       b8 00 00 00 00          mov    $0x0,%eax
  400fb7:       eb 05                   jmp    400fbe <phase_3+0x7b>
  400fb9:       b8 37 01 00 00          mov    $0x137,%eax
  400fbe:       3b 44 24 0c             cmp    0xc(%rsp),%eax
  400fc2:       74 05                   je     400fc9 <phase_3+0x86>
  400fc4:       e8 71 04 00 00          call   40143a <explode_bomb>
  400fc9:       48 83 c4 18             add    $0x18,%rsp
  400fcd:       c3                      ret
```

`400f6f` 会直接把炸弹搞炸，所以 `0x8(%rsp)` 不应该大于 `0x7`。

```
  400f71:       8b 44 24 08             mov    0x8(%rsp),%eax
  400f75:       ff 24 c5 70 24 40 00    jmp    *0x402470(,%rax,8)
```

这是干啥的?

看看 `0x402470` 附近有啥吧。

怎么反编译出来没这个地址?

```
00000000004022a0 <__libc_csu_fini>:
  4022a0:       f3 c3                   repz ret
  4022a2:       90                      nop
  4022a3:       90                      nop

Disassembly of section .fini:

00000000004022a4 <_fini>:
  4022a4:       48 83 ec 08             sub    $0x8,%rsp
  4022a8:       48 83 c4 08             add    $0x8,%rsp
  4022ac:       c3                      ret
```

????

我没招了。

（激情拷打 Gemini）

> 你看到的 402470 是 跳转表（Jump Table） 的基地址。
>
> 这正是 C 语言中 switch-case 语句编译后的典型模样。因为 switch 的分支太多，编译器为了效率，不会写一堆 if-else，而是直接在内存里建了一张“地址表”，根据你输入的数字直接查表跳转。

```
$ gdb bomb
GNU gdb (GDB) 17.1
Copyright (C) 2025 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

```
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) x/8gx 0x402470
0x402470:       0x0000000000400f7c      0x0000000000400fb9
0x402480:       0x0000000000400f83      0x0000000000400f8a
0x402490:       0x0000000000400f91      0x0000000000400f98
0x4024a0:       0x0000000000400f9f      0x0000000000400fa6
```

gdb 显示的地址为大端序，实际上 .rodata 以小端序存放地址。

```
$ objdump -s -j .rodata bomb --start-address=0x402470

bomb:     文件格式 elf64-x86-64

Contents of section .rodata:
 402470 7c0f4000 00000000 b90f4000 00000000  |.@.......@.....
 402480 830f4000 00000000 8a0f4000 00000000  ..@.......@.....
 402490 910f4000 00000000 980f4000 00000000  ..@.......@.....
 4024a0 9f0f4000 00000000 a60f4000 00000000  ..@.......@.....
```

行吧，那么得出这么一张表：

`%rax`	地址
0	`0x400f7c`
1	`0x400fb9`
2	`0x400f83`
3	`0x400f8a`
4	`0x400f91`
5	`0x400f98`
6	`0x400f9f`
7	`0x400fa6`

凭直觉，我猜测这可能是个多解问题。假设我们的 `0x8(%rsp)` 是 `0`，那么跳转到 `0x400f7c`。那么会依次执行：

```
mov    $0xcf,%eax
# jmp    400fbe <phase_3+0x7b>
cmp    0xc(%rsp),%eax
# je     400fc9 <phase_3+0x86>
```

如果 `je` 不满足条件。就会直接触发下一行的炸弹。

`0xcf` == `0d207`。 所以第二个数字应该输入 `207`！

为了验证猜想，我们把所有可能的答案都列出来：

0x8(%rsp)	0xc(%rsp)
0	207
1	311
2	707
3	256
4	389
5	206
6	682
7	327

```
./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 207
Halfway there!
```

phase_4()

```
$ objdump -d bomb --disassemble=phase_4

bomb:     文件格式 elf64-x86-64

Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:

000000000040100c <phase_4>:
  40100c:       48 83 ec 18             sub    $0x18,%rsp
  401010:       48 8d 4c 24 0c          lea    0xc(%rsp),%rcx
  401015:       48 8d 54 24 08          lea    0x8(%rsp),%rdx
  40101a:       be cf 25 40 00          mov    $0x4025cf,%esi
  40101f:       b8 00 00 00 00          mov    $0x0,%eax
  401024:       e8 c7 fb ff ff          call   400bf0 <__isoc99_sscanf@plt>
  401029:       83 f8 02                cmp    $0x2,%eax
  40102c:       75 07                   jne    401035 <phase_4+0x29>
  40102e:       83 7c 24 08 0e          cmpl   $0xe,0x8(%rsp)
  401033:       76 05                   jbe    40103a <phase_4+0x2e>
  401035:       e8 00 04 00 00          call   40143a <explode_bomb>
  40103a:       ba 0e 00 00 00          mov    $0xe,%edx
  40103f:       be 00 00 00 00          mov    $0x0,%esi
  401044:       8b 7c 24 08             mov    0x8(%rsp),%edi
  401048:       e8 81 ff ff ff          call   400fce <func4>
  40104d:       85 c0                   test   %eax,%eax
  40104f:       75 07                   jne    401058 <phase_4+0x4c>
```

```
  401051:       83 7c 24 0c 00          cmpl   $0x0,0xc(%rsp)
  401056:       74 05                   je     40105d <phase_4+0x51>
  401058:       e8 dd 03 00 00          call   40143a <explode_bomb>
  40105d:       48 83 c4 18             add    $0x18,%rsp
  401061:       c3                      ret
```

经典 `sscanf`。这次的格式化字符串是 `0x4025cf`。还是 `phase_3` 的格式，意味着输入俩整数进去，保存到 `0x8(%rsp)` 和 `0xc(%rsp)`。

此时我们的研究重心到了：

```
  40102e:       83 7c 24 08 0e          cmpl   $0xe,0x8(%rsp)
  401033:       76 05                   jbe    40103a <phase_4+0x2e>
  401035:       e8 00 04 00 00          call   40143a <explode_bomb>
  40103a:       ba 0e 00 00 00          mov    $0xe,%edx
  40103f:       be 00 00 00 00          mov    $0x0,%esi
  401044:       8b 7c 24 08             mov    0x8(%rsp),%edi
  401048:       e8 81 ff ff ff          call   400fce <func4>
  40104d:       85 c0                   test   %eax,%eax
  40104f:       75 07                   jne    401058 <phase_4+0x4c>
  401051:       83 7c 24 0c 00          cmpl   $0x0,0xc(%rsp)
  401056:       74 05                   je     40105d <phase_4+0x51>
  401058:       e8 dd 03 00 00          call   40143a <explode_bomb>
  40105d:       48 83 c4 18             add    $0x18,%rsp
```

`jbe` 需满足 `0xe` < `0x8(%rsp)`。即第一个数需要大于 `14`。

又已知函数传参的顺序是 `%edi`，`%esi`，`%edx`。在上面的汇编中，分别被赋值成 `0x8(%esp)`，`$0`，`$14`。查看 `func4` 的汇编：

```
objdump -d bomb --disassemble=func4

bomb:     文件格式 elf64-x86-64

Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:

0000000000400fce <func4>:
  400fce:       48 83 ec 08             sub    $0x8,%rsp
  400fd2:       89 d0                   mov    %edx,%eax
  400fd4:       29 f0                   sub    %esi,%eax
  400fd6:       89 c1                   mov    %eax,%ecx
  400fd8:       c1 e9 1f                shr    $0x1f,%ecx
  400fdb:       01 c8                   add    %ecx,%eax
  400fdd:       d1 f8                   sar    $1,%eax
  400fdf:       8d 0c 30                lea    (%rax,%rsi,1),%ecx
  400fe2:       39 f9                   cmp    %edi,%ecx
  400fe4:       7e 0c                   jle    400ff2 <func4+0x24>
  400fe6:       8d 51 ff                lea    -0x1(%rcx),%edx
  400fe9:       e8 e0 ff ff ff          call   400fce <func4>
  400fee:       01 c0                   add    %eax,%eax
  400ff0:       eb 15                   jmp    401007 <func4+0x39>
  400ff2:       b8 00 00 00 00          mov    $0x0,%eax
```

```
  400ff7:       39 f9                   cmp    %edi,%ecx
  400ff9:       7d 0c                   jge    401007 <func4+0x39>
  400ffb:       8d 71 01                lea    0x1(%rcx),%esi
  400ffe:       e8 cb ff ff ff          call   400fce <func4>
  401003:       8d 44 00 01             lea    0x1(%rax,%rax,1),%eax
  401007:       48 83 c4 08             add    $0x8,%rsp
  40100b:       c3                      ret
```

哦我的上帝这是什么诡异的递归……

（拿出草稿纸）

```
  400fd2:       89 d0                   mov    %edx,%eax
  400fd4:       29 f0                   sub    %esi,%eax
  400fd6:       89 c1                   mov    %eax,%ecx
  400fd8:       c1 e9 1f                shr    $0x1f,%ecx
  400fdb:       01 c8                   add    %ecx,%eax
  400fdd:       d1 f8                   sar    $1,%eax
  400fdf:       8d 0c 30                lea    (%rax,%rsi,1),%ecx
```

询问 Gemini 之后发现这部分实现了一个取平均数的操作。~~（原谅我）~~

`shr $0x1f,%ecx` 这条指令极其炫技。想想 `0x1f` 是什么？31 对吧。对一个有符号双字右移 31 位得到了什么？符号位！所以这条指令取了 `%ecx` (也就是 `%eax` 的备份) 的符号位。

`add %ecx,%eax` 更神奇。我们不妨分情况讨论：

1. `%eax` 是正数：符号位为 0，直接右移。
2. `%eax` 是负奇数：加一再右移（结果无区别）；
3. `%eax` 是负偶数：加一再右移（结果 + 1）。

发现了吗？这一部分实现了 C 标准的向 0 取整，确保右移和数学除法没有区别。

执行完之后 `%rax` 里面是差值的一半，而 `%ecx` 里面正好就是平均数（`low + (low + high) >> 1`）。

之后的条件跳转就不细说了，整体逻辑可以简化为下面的函数：

```
int func4(int target, int low, int high) {
    int mid = low + (high - low) / 2;
    if (mid > target) {
        return 2 * func4(target, low, mid - 1);
    } else if (mid < target) {
        return 2 * func4(target, mid + 1, high) + 1;
    } else {
        return 0;
    }
}
```

剩下下面几行：

```
  40104d:       85 c0                   test   %eax,%eax
  40104f:       75 07                   jne    401058 <phase_4+0x4c>
  401051:       83 7c 24 0c 00          cmpl   $0x0,0xc(%rsp)
  401056:       74 05                   je     40105d <phase_4+0x51>
  401058:       e8 dd 03 00 00          call   40143a <explode_bomb>
  40105d:       48 83 c4 18             add    $0x18,%rsp
  401061:       c3                      ret
```

显然，如果返回值不是 0，就会跳转到爆炸函数。`0xc(%rsp)` 如果不等于 0，也会触发爆炸。

那么显然，我们的 `0x8(%esp)` 应该被赋值成 7，`0xc(%esp)` 应该被赋值成 0。

```
./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 207
Halfway there!
7 0
So you got that one.  Try this one.
```

phase_5()

```
$ objdump -d bomb --disassemble=phase_5

bomb:     文件格式 elf64-x86-64

Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:

0000000000401062 <phase_5>:
  401062:       53                      push   %rbx
  401063:       48 83 ec 20             sub    $0x20,%rsp
  401067:       48 89 fb                mov    %rdi,%rbx
  40106a:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  401071:       00 00
  401073:       48 89 44 24 18          mov    %rax,0x18(%rsp)
  401078:       31 c0                   xor    %eax,%eax
  40107a:       e8 9c 02 00 00          call   40131b <string_length>
  40107f:       83 f8 06                cmp    $0x6,%eax
  401082:       74 4e                   je     4010d2 <phase_5+0x70>
  401084:       e8 b1 03 00 00          call   40143a <explode_bomb>
  401089:       eb 47                   jmp    4010d2 <phase_5+0x70>
  40108b:       0f b6 0c 03             movzbl (%rbx,%rax,1),%ecx
  40108f:       88 0c 24                mov    %cl,(%rsp)
  401092:       48 8b 14 24             mov    (%rsp),%rdx
  401096:       83 e2 0f                and    $0xf,%edx
  401099:       0f b6 92 b0 24 40 00    movzbl 0x4024b0(%rdx),%edx
  4010a0:       88 54 04 10             mov    %dl,0x10(%rsp,%rax,1)
  4010a4:       48 83 c0 01             add    $0x1,%rax
  4010a8:       48 83 f8 06             cmp    $0x6,%rax
  4010ac:       75 dd                   jne    40108b <phase_5+0x29>
  4010ae:       c6 44 24 16 00          movb   $0x0,0x16(%rsp)
  4010b3:       be 5e 24 40 00          mov    $0x40245e,%esi
  4010b8:       48 8d 7c 24 10          lea    0x10(%rsp),%rdi
  4010bd:       e8 76 02 00 00          call   401338 <strings_not_equal>
  4010c2:       85 c0                   test   %eax,%eax
  4010c4:       74 13                   je     4010d9 <phase_5+0x77>
  4010c6:       e8 6f 03 00 00          call   40143a <explode_bomb>
```

```
4010cb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
4010d0:       eb 07                   jmp    4010d9 <phase_5+0x77>
4010d2:       b8 00 00 00 00          mov    $0x0,%eax
4010d7:       eb b2                   jmp    40108b <phase_5+0x29>
4010d9:       48 8b 44 24 18          mov    0x18(%rsp),%rax
4010de:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
4010e5:       00 00
4010e7:       74 05                   je     4010ee <phase_5+0x8c>
4010e9:       e8 42 fa ff ff          call   400b30 <__stack_chk_fail@plt>
4010ee:       48 83 c4 20             add    $0x20,%rsp
4010f2:       5b                      pop    %rbx
4010f3:       c3                      ret
```

注意到有一行代码是

```
40106a:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
```

这是个啥（）

这个寻址方式我从来没见过，让我搜一手：

在 Stack OverFlow 上的帖子 Why does this memory address %fs:0x28 (fs[0x28]) have a random value? 中，有一位答主解释了这行代码的作用：

> Both the `FS` and `GS` registers can be used as base-pointer addresses in order to access special operating system data-structures. So what you're seeing is a value loaded at an offset from the value held in the `FS` register, and not bit manipulation of the contents of the `FS` register.
>
> Specifically what's taking place, is that `FS:0x28` on Linux is storing a special sentinel stack-guard value, and the code is performing a stack-guard check. For instance, if you look further in your code, you'll see that the value at `FS:0x28` is stored on the stack, and then the contents of the stack are recalled and an `XOR` is performed with the original value at `FS:0x28`. If the two values are equal, which means that the zero-bit has been set because `XOR`'ing two of the same values results in a zero-value, then we jump to the `test` routine, otherwise we jump to a special function that indicates that the stack was somehow corrupted, and the sentinel value stored on the stack was changed.
>
> If using GCC, this can be disabled with:
>
> `-fno-stack-protector`

中文翻译如下：

> `FS` 和 `GS` 寄存器都可以作为基址指针（Base-pointer addresses），用于访问操作系统特殊的内部数据结构。因此，你所看到的实际上是从 `FS` 寄存器保存的基地址加上一个偏移量后加载的数据，而不是对 `FS` 寄存器本身的内容进行位运算。
>
> 具体来说，在 Linux 系统上，`FS:0x28` 的位置存储了一个特殊的哨兵值，称为 栈保护值（Stack-guard value / Canary），这段代码正在进行栈保护检查。例如，如果你查看代码的后续部分，会发现程序先将 `FS:0x28` 处的值存入栈中；函数结束前，再将栈中的值取出，与 `FS:0x28` 原有的值进行 `XOR`（异或）运算。
>
> 如果两个值相等（这意味着异或运算的结果为 0，从而设置了零标志位 ZF），程序就会跳转到正常的测试或后续流程；否则，程序会跳转到一个特殊的函数，表明栈已经被破坏（缓冲区溢出），即存储在栈上的那个哨兵值被篡改了。

> 如果你使用 GCC 编译器，可以通过以下参数禁用此功能：
> `-fno-stack-protector`

观察后续代码，确实有将 `%fs:0x28` 与 `%rax` 异或运算的逻辑，如下：

```
  4010de:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
  4010e5:       00 00
  4010e7:       74 05                   je     4010ee <phase_5+0x8c>
  4010e9:       e8 42 fa ff ff          call   400b30 <__stack_chk_fail@plt>
```

我们继续往下看，看到下面这一段：

```
  401073:       48 89 44 24 18          mov    %rax,0x18(%rsp)
  401078:       31 c0                   xor    %eax,%eax
  40107a:       e8 9c 02 00 00          call   40131b <string_length>
  40107f:       83 f8 06                cmp    $0x6,%eax
  401082:       74 4e                   je     4010d2 <phase_5+0x70>
  401084:       e8 b1 03 00 00          call   40143a <explode_bomb>
```

前两行表示把之前存的那个 Canary 值压到栈里，并且把 `%eax` 清空。这也好理解，毕竟做校验嘛，不能到处留这个值。下面一行又出现了 `string_length`。直觉告诉我们，`%rdi` 里面存的准是输入字符串，而且长度必须为 6，否则爆炸。

那问题来了，这个字符串长度包不包含 `\0`？`phase_1` 蒙对了这个函数，但这里可不能蒙了，我们必须使用 gdb 来调试一手：

```
$ gdb bomb
Reading symbols from bomb...
(gdb) b *0x40107a             # 在 call 40131b <string_length> 处打断点
Breakpoint 1 at 0x40107a
(gdb) r                       # 运行程序
Starting program: /home/sakimidare/bomb/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 207
Halfway there!
7 0
So you got that one.  Try this one.
123456                        # 输入一个 123456 看看效果

Breakpoint 1, 0x000000000040107a in phase_5 ()
(gdb) ni                      # 运行下一条指令 （即执行完整个 call）
0x000000000040107f in phase_5 ()
(gdb) i r eax                 # 立即查看寄存器 %eax 的值
eax            0x6                 6
(gdb)
```

我们得出结论：`string_length` 函数返回的长度不包含 `\0`。也就是说，我们应该输入一个长度为 6 的字符串。

```
  401089:       eb 47                   jmp    4010d2 <phase_5+0x70>
  ...
```

```
  4010d2:       b8 00 00 00 00          mov    $0x0,%eax
  4010d7:       eb b2                   jmp    40108b <phase_5+0x29>

  40108b:       0f b6 0c 03             movzbl (%rbx,%rax,1),%ecx
  40108f:       88 0c 24                mov    %cl,(%rsp)
  401092:       48 8b 14 24             mov    (%rsp),%rdx
  401096:       83 e2 0f                and    $0xf,%edx
  401099:       0f b6 92 b0 24 40 00    movzbl 0x4024b0(%rdx),%edx
  4010a0:       88 54 04 10             mov    %dl,0x10(%rsp,%rax,1)
  4010a4:       48 83 c0 01             add    $0x1,%rax
  4010a8:       48 83 f8 06             cmp    $0x6,%rax
  4010ac:       75 dd                   jne    40108b <phase_5+0x29>
```

此时，`%rbx` 是 `%rdi` 里面的值，也就是输入的字符串地址。注意到这里新出现了一个寄存器 `%cl`。这是干啥的?

~~(对不起写这一段的时候忘掉了 `%cl` 是 `rcx` 的低位)~~

那就是说 `40108b` 把内存里那个字符串地址加上 `%rax` 的偏移量，得到的那个字节抠低位放到 `(%rsp)` 和 `%rdx` 去呗!

那有个问题啊，`(%rsp)` 是刚刚从 `%cl` 复制过来的一个字节，`%rdx` 是一个双字。如果强行复制，那么不会导致高位（x86-64 是小端序）充满垃圾值吗?

还好，下一行的 `and` 直接只保留了低位，把垃圾值舍弃了。

看看 `0x4024b0` 里面是啥吧!

```
$ objdump -s -j .rodata bomb --start-address=0x4024b0

bomb:     文件格式 elf64-x86-64

Contents of section .rodata:
 4024b0 6d616475 69657273 6e666f74 7662796c  maduiersnfotvbyl
```

现在 `%edx` 中塞的是上文硬编码字符串偏移当前输入字符低两位的字符（好拗口）。例如第一个字符输入 `'H'`，`'H' = 0x48`。低两位是 `0x8`，那么就会往 `edx` 里面存入

```
maduiersnfotvbyl
        ^
```

`'n'`。

之后 `mov %dl,0x10(%rsp,%rax,1)` 往内存里压栈，`%rax` 自增 1，并判断 `%rax` 是否已自增到 6。如果自增到 6，那么跳出循环。

```
  4010ae:       c6 44 24 16 00          movb   $0x0,0x16(%rsp)
  4010b3:       be 5e 24 40 00          mov    $0x40245e,%esi
  4010b8:       48 8d 7c 24 10          lea    0x10(%rsp),%rdi
  4010bd:       e8 76 02 00 00          call   401338 <strings_not_equal>
  4010c2:       85 c0                   test   %eax,%eax
  4010c4:       74 13                   je     4010d9 <phase_5+0x77>
  4010c6:       e8 6f 03 00 00          call   40143a <explode_bomb>
  4010cb:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
  4010d0:       eb 07                   jmp    4010d9 <phase_5+0x77>
  4010d2:       b8 00 00 00 00          mov    $0x0,%eax
  4010d7:       eb b2                   jmp    40108b <phase_5+0x29>
  4010d9:       48 8b 44 24 18          mov    0x18(%rsp),%rax
  4010de:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
```

```
  4010e5:       00 00
  4010e7:       74 05                   je     4010ee <phase_5+0x8c>
  4010e9:       e8 42 fa ff ff          call   400b30 <__stack_chk_fail@plt>
  4010ee:       48 83 c4 20             add    $0x20,%rsp
  4010f2:       5b                      pop    %rbx
  4010f3:       c3                      ret
```

第一行给末尾加上 `\0`。

`40245e` 的内容是

```
 40245e 666c 79657273 00000000 00000000 0000 flyers..........
```

即 `flyers`。

那么答案呼之欲出：

按照之前说的方法，输入的字符串取 ASCII 最后一个字节，然后在固定字符串上做偏移。最后拼出来的字符串应该是 `flyers`。

所以一个可能的答案是

```
IONEFG
```

```
./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Border relations with Canada have never been better.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
0 207
Halfway there!
7 0
So you got that one.  Try this one.
IONEFG
Good work!  On to the next...
```

`phase_6()`

```
$ objdump -d bomb --disassemble=phase_6

bomb:     文件格式 elf64-x86-64

Disassembly of section .init:

Disassembly of section .plt:

Disassembly of section .text:

00000000004010f4 <phase_6>:
  4010f4:       41 56                   push   %r14
  4010f6:       41 55                   push   %r13
  4010f8:       41 54                   push   %r12
  4010fa:       55                      push   %rbp
  4010fb:       53                      push   %rbx
  4010fc:       48 83 ec 50             sub    $0x50,%rsp
  401100:       49 89 e5                mov    %rsp,%r13
```

```
401103:       48 89 e6                mov    %rsp,%rsi
401106:       e8 51 03 00 00          call   40145c <read_six_numbers>
40110b:       49 89 e6                mov    %rsp,%r14
40110e:       41 bc 00 00 00 00       mov    $0x0,%r12d
401114:       4c 89 ed                mov    %r13,%rbp
401117:       41 8b 45 00             mov    0x0(%r13),%eax
40111b:       83 e8 01                sub    $0x1,%eax
40111e:       83 f8 05                cmp    $0x5,%eax
401121:       76 05                   jbe    401128 <phase_6+0x34>
401123:       e8 12 03 00 00          call   40143a <explode_bomb>
401128:       41 83 c4 01             add    $0x1,%r12d
40112c:       41 83 fc 06             cmp    $0x6,%r12d
401130:       74 21                   je     401153 <phase_6+0x5f>
401132:       44 89 e3                mov    %r12d,%ebx
401135:       48 63 c3                movslq %ebx,%rax
401138:       8b 04 84                mov    (%rsp,%rax,4),%eax
40113b:       39 45 00                cmp    %eax,0x0(%rbp)
40113e:       75 05                   jne    401145 <phase_6+0x51>
401140:       e8 f5 02 00 00          call   40143a <explode_bomb>
401145:       83 c3 01                add    $0x1,%ebx
401148:       83 fb 05                cmp    $0x5,%ebx
40114b:       7e e8                   jle    401135 <phase_6+0x41>
40114d:       49 83 c5 04             add    $0x4,%r13
401151:       eb c1                   jmp    401114 <phase_6+0x20>
401153:       48 8d 74 24 18          lea    0x18(%rsp),%rsi
401158:       4c 89 f0                mov    %r14,%rax
40115b:       b9 07 00 00 00          mov    $0x7,%ecx
401160:       89 ca                   mov    %ecx,%edx
401162:       2b 10                   sub    (%rax),%edx
401164:       89 10                   mov    %edx,(%rax)
401166:       48 83 c0 04             add    $0x4,%rax
40116a:       48 39 f0                cmp    %rsi,%rax
40116d:       75 f1                   jne    401160 <phase_6+0x6c>
40116f:       be 00 00 00 00          mov    $0x0,%esi
401174:       eb 21                   jmp    401197 <phase_6+0xa3>
401176:       48 8b 52 08             mov    0x8(%rdx),%rdx
40117a:       83 c0 01                add    $0x1,%eax
40117d:       39 c8                   cmp    %ecx,%eax
40117f:       75 f5                   jne    401176 <phase_6+0x82>
401181:       eb 05                   jmp    401188 <phase_6+0x94>
401183:       ba d0 32 60 00          mov    $0x6032d0,%edx
401188:       48 89 54 74 20          mov    %rdx,0x20(%rsp,%rsi,2)
40118d:       48 83 c6 04             add    $0x4,%rsi
401191:       48 83 fe 18             cmp    $0x18,%rsi
401195:       74 14                   je     4011ab <phase_6+0xb7>
401197:       8b 0c 34                mov    (%rsp,%rsi,1),%ecx
40119a:       83 f9 01                cmp    $0x1,%ecx
40119d:       7e e4                   jle    401183 <phase_6+0x8f>
40119f:       b8 01 00 00 00          mov    $0x1,%eax
4011a4:       ba d0 32 60 00          mov    $0x6032d0,%edx
4011a9:       eb cb                   jmp    401176 <phase_6+0x82>
4011ab:       48 8b 5c 24 20          mov    0x20(%rsp),%rbx
```

```
  4011b0:       48 8d 44 24 28          lea    0x28(%rsp),%rax
  4011b5:       48 8d 74 24 50          lea    0x50(%rsp),%rsi
  4011ba:       48 89 d9                mov    %rbx,%rcx
  4011bd:       48 8b 10                mov    (%rax),%rdx
  4011c0:       48 89 51 08             mov    %rdx,0x8(%rcx)
  4011c4:       48 83 c0 08             add    $0x8,%rax
  4011c8:       48 39 f0                cmp    %rsi,%rax
  4011cb:       74 05                   je     4011d2 <phase_6+0xde>
  4011cd:       48 89 d1                mov    %rdx,%rcx
  4011d0:       eb eb                   jmp    4011bd <phase_6+0xc9>
  4011d2:       48 c7 42 08 00 00 00    movq   $0x0,0x8(%rdx)
  4011d9:       00
  4011da:       bd 05 00 00 00          mov    $0x5,%ebp
  4011df:       48 8b 43 08             mov    0x8(%rbx),%rax
  4011e3:       8b 00                   mov    (%rax),%eax
  4011e5:       39 03                   cmp    %eax,(%rbx)
  4011e7:       7d 05                   jge    4011ee <phase_6+0xfa>
  4011e9:       e8 4c 02 00 00          call   40143a <explode_bomb>
  4011ee:       48 8b 5b 08             mov    0x8(%rbx),%rbx
  4011f2:       83 ed 01                sub    $0x1,%ebp
  4011f5:       75 e8                   jne    4011df <phase_6+0xeb>
  4011f7:       48 83 c4 50             add    $0x50,%rsp
  4011fb:       5b                      pop    %rbx
  4011fc:       5d                      pop    %rbp
  4011fd:       41 5c                   pop    %r12
  4011ff:       41 5d                   pop    %r13
  401201:       41 5e                   pop    %r14
  401203:       c3                      ret

Disassembly of section .fini:

  4010f4:       41 56                   push   %r14
  4010f6:       41 55                   push   %r13
  4010f8:       41 54                   push   %r12
  4010fa:       55                      push   %rbp
  4010fb:       53                      push   %rbx
  4010fc:       48 83 ec 50             sub    $0x50,%rsp
  401100:       49 89 e5                mov    %rsp,%r13
  401103:       48 89 e6                mov    %rsp,%rsi
  401106:       e8 51 03 00 00          call   40145c <read_six_numbers>
  40110b:       49 89 e6                mov    %rsp,%r14
  40110e:       41 bc 00 00 00 00       mov    $0x0,%r12d
  401114:       4c 89 ed                mov    %r13,%rbp
  401117:       41 8b 45 00             mov    0x0(%r13),%eax
  40111b:       83 e8 01                sub    $0x1,%eax
  40111e:       83 f8 05                cmp    $0x5,%eax
  401121:       76 05                   jbe    401128 <phase_6+0x34>
```

开幕雷击，压了一百万个栈。在栈中分配了 `0x50` 字节空间后把栈指针赋给了 `%r13`。然后又调用了 `read_six_numbers` 往内存里面存数字，内存布局还是见上一张表。存完数字之后把栈指针 `%rsp` 存到 `%r14` 里面去。

之后把 `%r12d` 清零，将 `(%r13)` 的值（未放数字时候的栈指针，也就是指向第一个数字的指针）放到 `%eax` 里。把 `%eax` 里面的数字减一，如果比 5 小，则炸弹不爆炸。所以第一个数字必须小于等于 6。

```
401128:       41 83 c4 01             add    $0x1,%r12d
40112c:       41 83 fc 06             cmp    $0x6,%r12d
401130:       74 21                   je     401153 <phase_6+0x5f>
401132:       44 89 e3                mov    %r12d,%ebx
401135:       48 63 c3                movslq %ebx,%rax
401138:       8b 04 84                mov    (%rsp,%rax,4),%eax
40113b:       39 45 00                cmp    %eax,0x0(%rbp)
40113e:       75 05                   jne    401145 <phase_6+0x51>
401140:       e8 f5 02 00 00          call   40143a <explode_bomb>
401145:       83 c3 01                add    $0x1,%ebx
401148:       83 fb 05                cmp    $0x5,%ebx
40114b:       7e e8                   jle    401135 <phase_6+0x41>
40114d:       49 83 c5 04             add    $0x4,%r13
401151:       eb c1                   jmp    401114 <phase_6+0x20>
401153:       48 8d 74 24 18          lea    0x18(%rsp),%rsi
```

此处 `%r12d` 自增 1，并与 6 比较，推测当循环来用。如果 `%12d` 没到 6，则把 `%r12d` 搬到 `%ebx` 和 `%rax` 里面去。`mov (%rsp,%rax,4),%eax` 显然用来读取刚刚输入的每一个数字。接着将 `%eax` 和 `(%rbp)`（也就是 `(%r13)`，输入的第一个数字）比较，如果不等则跳过炸弹。

之后 `%ebx` 自增，与 5 比较（其实等价于 `%12d` 与 6 比较）。如果小于等于 5，则跳转至 `401135` 开始循环。

这个地方有个问题：怎么两个循环变量？难道有个嵌套循环？

以下为 Gemini 的回答：

> 你的直觉非常敏锐。没错，这里确实是一个嵌套循环（Nested Loop）。
>
> 其核心逻辑是：验证你输入的 6 个数字是否互不相同（即没有重复值）。 我们可以把这段汇编翻译成更直近的高级语言逻辑，这样你就一眼能看出这两个变量的作用了：
>
> 逻辑结构拆解外层循环（`%r12d` 作为索引 `i`）从第 1 个数字遍历到第 6 个数字。内层循环（`%ebx` 作为索引 `j`）将当前第 `i` 个数字与其后面的所有数字进行对比。详细对照分析
>
> 1. 外层循环：选择当前要对比的基准代码段
>
> ```
> 401114:  mov    %r13,%rbp         # rbp 指向当前正在检查的输入数字
> 401117:  mov    0x0(%r13),%eax    # 取出当前数字
> ...
> 401128:  add    $0x1,%r12d        # i++
> 40112c:  cmp    $0x6,%r12d        # 是否检查完了全部 6 个数字?
> 401130:  je     401153            # 如果是，跳出循环进入下一阶段
> ```
>
> 1. 内层循环：检查后续数字是否重复代码段
>
> ```
> 401132:  mov    %r12d,%ebx        # j = i + 1 (开始位置)
> 401135:  movslq %ebx,%rax         # 将索引 j 转为 64 位
> 401138:  mov    (%rsp,%rax,4),%eax # 读取输入的第 j 个数字
> 40113b:  cmp    %eax,0x0(%rbp)    # 比较 input[i] 和 input[j]
> 40113e:  jne    401145            # 如果不相等，没问题，跳过炸弹
> 401140:  call   40143a <explode_bomb> # 如果相等，说明数字重复，炸弹爆炸!
> 401145:  add    $0x1,%ebx         # j++
> 401148:  cmp    $0x5,%ebx         # 检查 j 是否到头了
> 40114b:  7e e8  jle 401135        # 如果 j ≤ 5，继续比较下一个
> ```

> 为什么有两个循环变量? 为了保证你输入的 6 个数字是一个排列（Unique permutation），程序必须执行一个类似于“冒泡”式的两两比对：
>
> · 当 i = 0 时，它会拿第 1 个数字去和第 2, 3, 4, 5, 6 个数字比。
> · 当 i=1 时，它会拿第 2 个数字去和第 3, 4, 5, 6 个数字比。
> · 以此类推。总结 %r12d 是当前选定数字的序号（外层）。
>
> `%ebx` 是正在与其对比的后续数字的序号（内层）。`%rbp`（及 `%r13`）作为指针在栈空间（`%rsp`）上移动，方便直接通过内存地址读取“基准值”。如果你输入的数字里有两个 3，那么当外层循环到第一个 3，内层循环扫描到第二个 3 时，cmp 结果就会相等，触发 401140 处的 `explode_bomb`。

作者 · 许可证 CC BY-NC-SA 4.0