中国科学技术大学

实验报告

计算机系统详解

Bomb Lab

学生姓名: 朱云沁

学生学号: PB20061372

完成时间: 二〇二二年四月二十五日

目录

一、 简介

1. 实验目的

- 熟悉 x86-64 架构汇编语言.
- 掌握二进制调试器的使用.
- 学习反汇编与逆向工程方法.

2. 实验要求

使用 `objdump`、`gdb`、`strings` 等工具, 通过反汇编与逆向工程方法, 分析可执行文件 `bomb` 的工作原理, 破解该程序要求输入的若干不同字符串, 从而拆除“二进制炸弹”.

该二进制炸弹共分 7 个关卡 (含 1 个隐藏关卡).

3. 实验环境

本实验所有命令均在以下环境执行[1]:

Machine	ASUS FH5900V Notebook PC
Processor	Intel(R) Core(TM) i7-6700HQ CPU @ 2.60Hz
Memory	4GB DDR4 2133MHz
System	Windows 10 家庭中文版, 64 位, 基于 x64
WSL Distro	Ubuntu 20.04.4 LTS
Packages	GNU Binutils 2.34, GDB 9.2.0

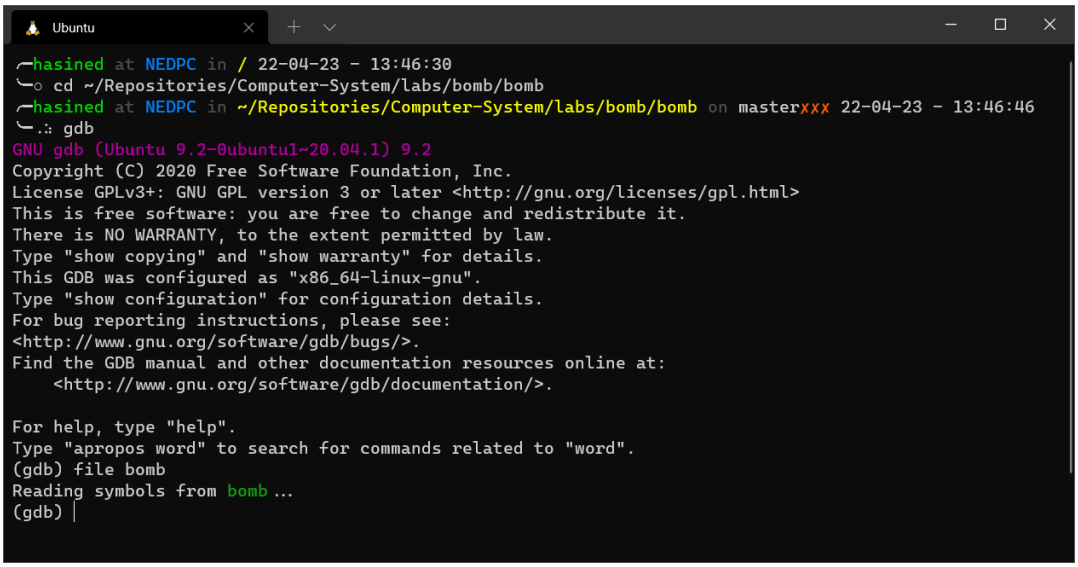

图 1: 在 WSL 中运行 gdb

[1]本实验需要在 x86-64 环境下用 gdb 调试 glibc 程序. 由于笔者的 MacBook 为 ARM 架构, 最初计划在 Docker 容器中部署完整 Linux 实验环境 (参见 Data Lab 实验报告). 调查发现, 用于仿真 x86-64 架构的 QEMU 并未实现 ptrace, 无法直接在容器内调试, 仅支持一种在主机上运行 gdb 并远程连接的调试方法 (详细描述转见 issue: https://github.com/docker/for-mac/issues/5191 及 QEMU 官方文档: https://qemu.readthedocs.io/en/latest/system/gdb.html). 然而, 在基于 ARM 的 Mac 机器上安装 gdb 并非常规做法, 缺乏良好支持. 笔者决定放弃远程调试的方案, 转移到原生 x86-64 的机器下进行本实验.

二、 实验成果

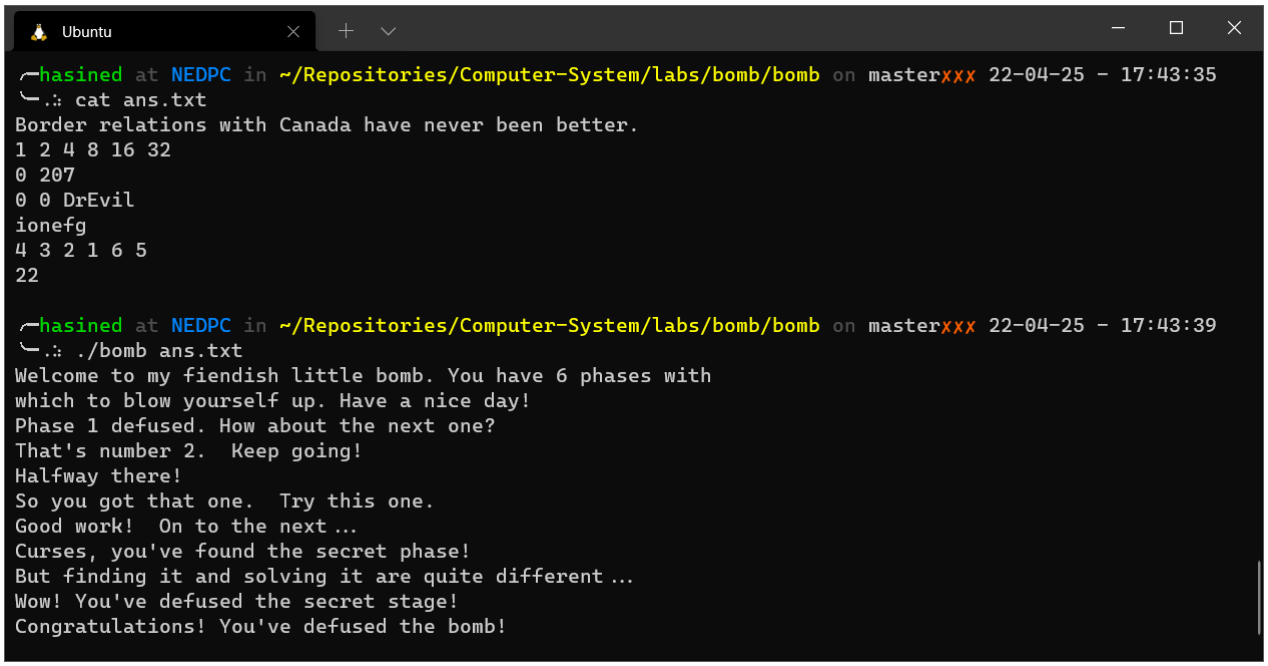

图 2: 在 WSL 中拆除炸弹

三、 实验过程

1. 准备工作

根据实验材料中的提示, 首先利用 GNU Binutils 提供的 `objdump` 工具反汇编 `bomb` 文件. 在工作路径中执行命令

```
$ objdump -d bomb > disas.txt
```

从而将结果存入 `disas.txt` 中, 以备后续参阅. 输出包括了 `.init`, `.plt`, `.text`, `.fini` 等 4 个节的反汇编结果, 涵盖了主要的机器指令. 输出还注明了文件格式为 `elf64-x86-64`, 因此可使用 `readelf` 工具查看该文件头信息

```
$ readelf -h bomb
```

有如下结果

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
```

```
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x400c90
  Start of program headers:          64 (bytes into file)
  Start of section headers:          18616 (bytes into file)
  Flags:                             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
```

可见，该 ELF 文件采用小端格式补码表示数据，依赖于 Unix 的二进制接口，需在 x86-64 机器中运行. 映证了搭建相应实验环境的必要性. 上述结果还指出程序入口点地址为 0x400c90，即 .text 节的起始地址.

自然地想到用 nm 工具打印代码段的符号

```
$ nm bomb | grep -w T | sort > syms.txt
```

得到按地址升序排列的结果，见附录 syms.txt. 其中包括了 bomb.c 中出现的函数名，如 main, phase_defused 等. 也包括了被隐藏的函数名，如 secret_phase, read_six_numbers 等. 显然，该二进制炸弹有隐藏关卡.

另一方面，根据提示，可以使用 strings 工具打印 bomb 文件中所有字符串. 此处我们使用 readelf 工具，以字符串形式导出 .rodata 节的内容

```
$ readelf -p .rodata bomb > rodata.txt
```

结果见附录 rodata.txt. 观察发现，其中用于交互或报错的居多，部分串在 bomb.c 文件中直接出现. 而 “Border relations with Canada have never been better.” “flyers” 等串较为突兀，应当引起注意.

下面，利用 gdb 工具逐关卡破解炸弹程序. 进入 gdb 并载入 bomb 文件，应有如图 1 所示输入输出. 在各关卡的入口处设置断点，以防炸弹爆炸

```
(gdb) break phase_1
Breakpoint 1 at 0x400ee0
(gdb) break phase_2
Breakpoint 2 at 0x400efc
```

```
(gdb) break phase_3
Breakpoint 3 at 0x400f43
(gdb) break phase_4
Breakpoint 4 at 0x40100c
(gdb) break phase_5
Breakpoint 5 at 0x401062
(gdb) break phase_6
Breakpoint 6 at 0x4010f4
(gdb) break secret_phase
Breakpoint 7 at 0x401242
```

2. phase_1

运行程序, 输入任意字符串, 触发 phase_1 处断点, 进而反汇编当前函数

```
(gdb) run
Starting program: /home/hasined/Repositories/Computer-System/labs/bomb/bomb/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
My ID is PB20061372

Breakpoint 1, 0x0000000000400ee0 in phase_1 ()
(gdb) disas
Dump of assembler code for function phase_1:
=> 0x0000000000400ee0 <+0>:     sub    $0x8,%rsp
   0x0000000000400ee4 <+4>:     mov    $0x402400,%esi
   0x0000000000400ee9 <+9>:     callq  0x401338 <strings_not_equal>
   0x0000000000400eee <+14>:    test   %eax,%eax
   0x0000000000400ef0 <+16>:    je     0x400ef7 <phase_1+23>
   0x0000000000400ef2 <+18>:    callq  0x40143a <explode_bomb>
   0x0000000000400ef7 <+23>:    add    $0x8,%rsp
   0x0000000000400efb <+27>:    retq
End of assembler dump.
```

分析汇编代码主体部分, 得知大意为:

1. 将 0x402400 移入 esi 中.
2. 调用 strings_not_equal.
3. 若 eax 为 0, 跳转至函数尾部; 否则调用 explode_bomb.

显然, explode_bomb 为实现炸弹爆炸的函数. 出于安全起见, 在该函数入口设置断点

```
(gdb) break explode_bomb
Breakpoint 8 at 0x40143a
```

从而在后续实验过程中避免非预期的爆炸.

根据代码逻辑, 在 0x400eee 处设置断点, 程序运行到此处时修改 eax 为 0, 即可通过 phase_1 . 输入输出如下

```
(gdb) break *0x400eee
Breakpoint 9 at 0x400eee
(gdb) continue
Continuing.

Breakpoint 9, 0x0000000000400eee in phase_1 ()
(gdb) set $eax=0
(gdb) continue
Continuing.
Phase 1 defused. How about the next one?
```

每个关卡都可以用修改寄存器值的手段作弊通过, 往后不再使用类似方法, 而是以正常途径得出答案.

合理猜测, strings_not_equal 函数用于比较两字符串, 两字符串相等时返回 0. 结合 x86-64 System V 的调用习惯, 可知作为实参的字符串指针有一个位于 esi 中, 此处传入 0x402400; 另一指针应当位于 edi 或 rdi 中; 返回值存于 eax 中.

出于验证目的, 可以将被调函数 strings_not_equal 反汇编

```
(gdb) disas strings_not_equal
```

或者在准备工作的 disas.txt 文件中查找, 往后不再特别指出. 由汇编代码[2], 可知函数逻辑

1. 调用 string_length, 分别得到 rdi, rsi 中字符串的长度.
2. 比较两字符串长度, 若不相等, 返回 1.
3. 逐个比较每一位, 若不相等, 返回 1; 否则返回 0.

与假设相符. 为了确认实参 rdi 的含义, 将 main 函数反汇编, 发现在调用 phase_1 前有如下语句

```
0x0000000000400e32 <+146>:   callq  0x40149e <read_line>
0x0000000000400e37 <+151>:   mov    %rax,%rdi
```

[2]见附录 strings_not_equal.txt

对应到 bomb.c 中为

```
input = read_line();             /* Get input                   */
phase_1(input);                  /* Run the phase               */
```

可知 rdi 存入了 read_line 函数的返回值, 即所输入字符串的指针.

至此, phase_1 得到解决. 该关卡要求输入的字符串与 0x402400 指向的字符串严格相等. 检查得到答案

```
(gdb) x/s 0x402400
0x402400:       "Border relations with Canada have never been better."
```

重新运行程序并输入 “Border relations with Canada have never been better.” , 成功通过 phase_1. 输入输出如下

```
(gdb) disable 1
(gdb) disable 9
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/hasined/Repositories/Computer-System/labs/bomb/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?
```

事实上, 在准备工作的 rodata.txt 中已经找到了答案.

3. phase_2

来到 phase_2 的断点处并反汇编

```
Breakpoint 2, 0x0000000000400efc in phase_2 ()
(gdb) disas
Dump of assembler code for function phase_2:
=> 0x0000000000400efc <+0>:     push   %rbp
   0x0000000000400efd <+1>:     push   %rbx
   0x0000000000400efe <+2>:     sub    $0x28,%rsp
   0x0000000000400f02 <+6>:     mov    %rsp,%rsi
   0x0000000000400f05 <+9>:     callq  0x40145c <read_six_numbers>
   ...                          ...    ...
   0x0000000000400f3c <+64>:    add    $0x28,%rsp
   0x0000000000400f40 <+68>:    pop    %rbx
```

```
   0x0000000000400f41 <+69>:    pop    %rbp
   0x0000000000400f42 <+70>:    retq
End of assembler dump.
```

发现该关卡以栈帧顶部指针为参数, 调用 read_six_numbers. 合理猜测, 该函数从输入字符串读出 6 个数字并返回到栈帧中. 出于验证目的, 将其反汇编[3], 梳理得其功能为

1. 调用动态链接的 sscanf, 传入的参数依次为: rdi, 0x4025c3, rsi, rsi+4, rsi+8, rsi+12, rsi+16, rsi+20.
2. 若 sscanf 的返回值 eax 不大于 5, 调用 explode_bomb, 否则跳转至函数尾部.

应当注意, sscanf 的前 6 个参数通过寄存器传递, 而后 2 个参数通过栈帧传递. sscanf 是用于格式化输入数据的标准库函数, 在此处, 作为输入源的字符串由 read_line 获得, 作为格式控制的字符串位于地址 0x4025c3, 可用 gdb 查看

```
(gdb) x/s 0x4025c3
"%d %d %d %d %d %d"
```

返回的 6 个整数位于 phase_2 函数的栈帧中, 地址依次为rsp, rsp+4, rsp+8, rsp+12, rsp+16 和 rsp+20. 若输入字符串所含整数少于 6 个, 炸弹爆炸.

省略号部分的汇编代码如下

```
0x0000000000400f0a <+14>:    cmpl   $0x1,(%rsp)
0x0000000000400f0e <+18>:    je     0x400f30 <phase_2+52>
0x0000000000400f10 <+20>:    callq  0x40143a <explode_bomb>
0x0000000000400f15 <+25>:    jmp    0x400f30 <phase_2+52>
0x0000000000400f17 <+27>:    mov    -0x4(%rbx),%eax
0x0000000000400f1a <+30>:    add    %eax,%eax
0x0000000000400f1c <+32>:    cmp    %eax,(%rbx)
0x0000000000400f1e <+34>:    je     0x400f25 <phase_2+41>
0x0000000000400f20 <+36>:    callq  0x40143a <explode_bomb>
0x0000000000400f25 <+41>:    add    $0x4,%rbx
0x0000000000400f29 <+45>:    cmp    %rbp,%rbx
0x0000000000400f2c <+48>:    jne    0x400f17 <phase_2+27>
0x0000000000400f2e <+50>:    jmp    0x400f3c <phase_2+64>
0x0000000000400f30 <+52>:    lea    0x4(%rsp),%rbx
0x0000000000400f35 <+57>:    lea    0x18(%rsp),%rbp
0x0000000000400f3a <+62>:    jmp    0x400f17 <phase_2+27>
```

其大意为

[3]见附录 read_six_numbers.txt

1. 若第 1 个整数不为 1, 炸弹爆炸.
2. 进入循环结构: 对每个 $i = 2, 3, \cdots, 6$, 若第 i 整数不是第 $i-1$ 整数的 2 倍, 炸弹爆炸.

故要求输入的 6 个整数为 “1 2 4 8 16 32” .

重新运行程序, 输入答案, 在 `add $0x28,%rsp` 处设置断点

```
(gdb) break *0x400f3c
Breakpoint 10 at 0x400f3c
(gdb) continue
Continuing.

Breakpoint 10, 0x0000000000400f3c in phase_2 ()
(gdb) x/6wd $rsp
0x7fffffffdf60: 1       2       4       8
0x7fffffffdf70: 16      32
(gdb) continue
Continuing.
That's number 2.  Keep going!
```

顺利到达断点处, 并且栈帧中的数据与前面的分析相符.

4. phase_3

将 `phase_3` 反汇编, 发现调用了 `sscanf`, 代码结构与 `read_six_numbers` 相似

```
Dump of assembler code for function phase_3:
=> 0x0000000000400f43 <+0>:     sub    $0x18,%rsp
   0x0000000000400f47 <+4>:     lea    0xc(%rsp),%rcx
   0x0000000000400f4c <+9>:     lea    0x8(%rsp),%rdx
   0x0000000000400f51 <+14>:    mov    $0x4025cf,%esi
   0x0000000000400f56 <+19>:    mov    $0x0,%eax
   0x0000000000400f5b <+24>:    callq  0x400bf0 <__isoc99_sscanf@plt>
   0x0000000000400f60 <+29>:    cmp    $0x1,%eax
   0x0000000000400f63 <+32>:    jg     0x400f6a <phase_3+39>
   0x0000000000400f65 <+34>:    callq  0x40143a <explode_bomb>
   ...                          ...    ...
   0x0000000000400fc9 <+134>:   add    $0x18,%rsp
   0x0000000000400fcd <+138>:   retq
End of assembler dump.
```

检查其格式, 发现要求输入 2 个整数

```
(gdb) x/s 0x4025cf
0x4025cf:       "%d %d"
```

目标是找出 2 个整数应当满足的条件. 观察省略号部分的代码, 发现有大量重复结构

```
0x0000000000400f6a <+39>:    cmpl   $0x7,0x8(%rsp)
0x0000000000400f6f <+44>:    ja     0x400fad <phase_3+106>
0x0000000000400f71 <+46>:    mov    0x8(%rsp),%eax
0x0000000000400f75 <+50>:    jmpq   *0x402470(,%rax,8)
0x0000000000400f7c <+57>:    mov    $0xcf,%eax
0x0000000000400f81 <+62>:    jmp    0x400fbe <phase_3+123>
0x0000000000400f83 <+64>:    mov    $0x2c3,%eax
0x0000000000400f88 <+69>:    jmp    0x400fbe <phase_3+123>
0x0000000000400f8a <+71>:    mov    $0x100,%eax
0x0000000000400f8f <+76>:    jmp    0x400fbe <phase_3+123>
0x0000000000400f91 <+78>:    mov    $0x185,%eax
0x0000000000400f96 <+83>:    jmp    0x400fbe <phase_3+123>
0x0000000000400f98 <+85>:    mov    $0xce,%eax
0x0000000000400f9d <+90>:    jmp    0x400fbe <phase_3+123>
0x0000000000400f9f <+92>:    mov    $0x2aa,%eax
0x0000000000400fa4 <+97>:    jmp    0x400fbe <phase_3+123>
0x0000000000400fa6 <+99>:    mov    $0x147,%eax
0x0000000000400fab <+104>:   jmp    0x400fbe <phase_3+123>
0x0000000000400fad <+106>:   callq  0x40143a <explode_bomb>
0x0000000000400fb2 <+111>:   mov    $0x0,%eax
0x0000000000400fb7 <+116>:   jmp    0x400fbe <phase_3+123>
0x0000000000400fb9 <+118>:   mov    $0x137,%eax
0x0000000000400fbe <+123>:   cmp    0xc(%rsp),%eax
0x0000000000400fc2 <+127>:   je     0x400fc9 <phase_3+134>
0x0000000000400fc4 <+129>:   callq  0x40143a <explode_bomb>
```

其中使用间接比例变址寻址方式的 `jmpq *0x402470(,%rax,8)` 一行值得重视. 加之紧随其后有大量相同的 `jmp` 操作, 有充分的理由猜测这是 `switch` 语句的编译结果.

首先, 我们查看 0x402470 指向的跳转表

```
(gdb) x/8g 0x402470
0x402470:       0x400f7c <phase_3+57>   0x400fb9 <phase_3+118>
0x402480:       0x400f83 <phase_3+64>   0x400f8a <phase_3+71>
0x402490:       0x400f91 <phase_3+78>   0x400f98 <phase_3+85>
0x4024a0:       0x400f9f <phase_3+92>   0x400fa6 <phase_3+99>
```

然后, 根据跳转结果, 将汇编代码翻译为 C 语言

```
sscanf(input, "%d %d", &index, &number);
switch (index) {
case 0:
    eax = 207;      // mov 0xcf(%rsp),%eax
    break;          // jmp 0x400fbe <phase_3+123>
case 1:
    eax = 311;      // mov 0x137(%rsp),%eax
    break;
case 2:
    eax = 707;      // mov 0x2c3(%rsp),%eax
    break;          // jmp 0x400fbe <phase_3+123>
case 3:
    eax = 256;      // mov 0x100(%rsp),%eax
    break;          // jmp 0x400fbe <phase_3+123>
case 4:
    eax = 389;      // mov 0x185(%rsp),%eax
    break;          // jmp 0x400fbe <phase_3+123>
case 5:
    eax = 206;      // mov 0xce(%rsp),%eax
    break;          // jmp 0x400fbe <phase_3+123>
case 6:
    eax = 682;      // mov 0x2aa(%rsp),%eax
    break;          // jmp 0x400fbe <phase_3+123>
case 7:
    eax = 327;      // mov 0x147(%rsp),%eax
    break;          // jmp 0x400fbe <phase_3+123>
default:
    explode_bomb(); eax = 0;
}
if (eax != number) explode_bomb();
```

其中, index, number 分别为 rsp+8, rsp+12 指向的整数, 两者应当满足的映射关系已十分显然: 当 index = 0 时, number 必须为 0xcf, 即 207; 当 index = 1 时, number = 0x137, 即 311; 依此类推. 故本关卡共有 8 种可行的答案, 部分试验结果如下

```
(gdb) disable 3
(gdb) break *0x400e86
Breakpoint 11 at 0x400e86: file bomb.c, line 94.
(gdb) jump *0x400e5b
Continuing at 0x400e5b.
That's number 2.  Keep going!
```

```
0 207
Halfway there!

Breakpoint 11, main (argc=<optimized out>, argv=<optimized out>) at bomb.c:94
94          input = read_line();
(gdb) jump *0x400e5b
Continuing at 0x400e5b.
That's number 2.  Keep going!
1 311
Halfway there!

...

Breakpoint 11, main (argc=<optimized out>, argv=<optimized out>) at bomb.c:94
94          input = read_line();
(gdb) jump *0x400e5b
Continuing at 0x400e5b.
That's number 2.  Keep going!
7 327
Halfway there!
```

一个有趣的事实是, 在使用 jump 命令反复试验的过程中, 炸弹提前输出了 2 次 "Congratulations! You've defused the bomb!"

5. phase_4

将 phase_4 反汇编

```
Dump of assembler code for function phase_4:
   0x000000000040100c <+0>:     sub    $0x18,%rsp
   0x0000000000401010 <+4>:     lea    0xc(%rsp),%rcx
   0x0000000000401015 <+9>:     lea    0x8(%rsp),%rdx
   0x000000000040101a <+14>:    mov    $0x4025cf,%esi
   0x000000000040101f <+19>:    mov    $0x0,%eax
   0x0000000000401024 <+24>:    callq  0x400bf0 <__isoc99_sscanf@plt>
   0x0000000000401029 <+29>:    cmp    $0x2,%eax
   0x000000000040102c <+32>:    jne    0x401035 <phase_4+41>
   0x000000000040102e <+34>:    cmpl   $0xe,0x8(%rsp)
   0x0000000000401033 <+39>:    jbe    0x40103a <phase_4+46>
   0x0000000000401035 <+41>:    callq  0x40143a <explode_bomb>
   ...                          ...    ...
   0x000000000040105d <+81>:    add    $0x18,%rsp
```

```
   0x0000000000401061 <+85>:    retq
End of assembler dump.
```

发现前 6 行与 phase_3 完全一致, 同样都要求输入 2 个整数. 区别在于输入数据数目严格为 2, 并且第 1 个无符号整数不得大于 14.

省略号部分如下

```
0x000000000040103a <+46>:    mov    $0xe,%edx
0x000000000040103f <+51>:    mov    $0x0,%esi
0x0000000000401044 <+56>:    mov    0x8(%rsp),%edi
0x0000000000401048 <+60>:    callq  0x400fce <func4>
0x000000000040104d <+65>:    test   %eax,%eax
0x000000000040104f <+67>:    jne    0x401058 <phase_4+76>
0x0000000000401051 <+69>:    cmpl   $0x0,0xc(%rsp)
0x0000000000401056 <+74>:    je     0x40105d <phase_4+81>
0x0000000000401058 <+76>:    callq  0x40143a <explode_bomb>
```

发现调用了名为 func4 的函数, 传入的参数 edi, esi, edx 分别为输入的第 1 个整数, 0 和 14. 若返回值不为 0, 炸弹爆炸.

根据最后 3 行, 输入的第 2 个整数必须为 0. 因此本关卡可通过枚举第 1 个整数解决.

注意到 func4 的汇编代码[4]涉及递归, 为了清晰, 将其翻译为 C 语言进行分析.

```c
int func4(int edi, int esi, int edx) {
    int eax = edx - esi;
    eax += eax >> 31;
    eax >>= 1;
    int ecx = esi + eax;
    if (edi < ecx)
        return func4(edi, esi, ecx - 1) * 2;
    else if (edi > ecx)
        return func4(edi, ecx + 1, edx) * 2 + 1;
    else
        return 0;
}
```

类似于二分查找, 但似乎逻辑更为复杂. 稍加分析, 知 eax += eax >> 31 与 eax >>= 1 两句实为 C 语言整数除法 eax /= 2 的编译结果, 使得被除数为负整数时向上取整. 于是, 优化代码结构, 并重写变量名, 得到

[4]见附录 func4.txt

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

初始时, low = 0, high = 14. 根据回溯过程, 将 0~14 画为一棵二叉搜索树

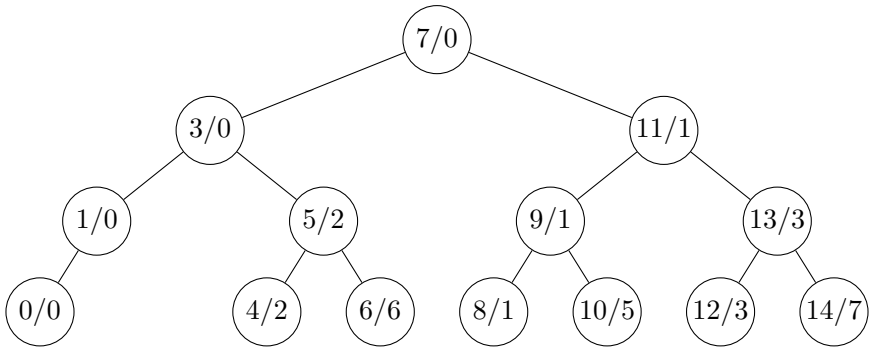

图 3: phase_4 的二叉树表示 (节点中为参数 x 与对应返回值)

显然, 只有每层最左的节点的对应返回值为 0, 符合要求. 故本关卡共有 4 种答案, 分别为 “7 0” “3 0” “1 0” “0 0” . 输入 “0 0” , 得到结果如下

```
Halfway there!
0 0

Breakpoint 4, 0x000000000040100c in phase_4 ()
(gdb) continue
Continuing.
So you got that one.  Try this one.
```

6. phase_5

将 phase_5 反汇编得

```
Dump of assembler code for function phase_5:
=> 0x0000000000401062 <+0>:     push   %rbx
   0x0000000000401063 <+1>:     sub    $0x20,%rsp
   0x0000000000401067 <+5>:     mov    %rdi,%rbx
```

```
   0x0000000000040106a <+8>:     mov    %fs:0x28,%rax
   0x0000000000401073 <+17>:    mov    %rax,0x18(%rsp)
   0x0000000000401078 <+22>:    xor    %eax,%eax
   0x000000000040107a <+24>:    callq  0x40131b <string_length>
   0x000000000040107f <+29>:    cmp    $0x6,%eax
   0x0000000000401082 <+32>:    je     0x4010d2 <phase_5+112>
   0x0000000000401084 <+34>:    callq  0x40143a <explode_bomb>
   0x0000000000401089 <+39>:    jmp    0x4010d2 <phase_5+112>
   ...                          ...    ...
   0x00000000004010d9 <+119>:   mov    0x18(%rsp),%rax
   0x00000000004010de <+124>:   xor    %fs:0x28,%rax
   0x00000000004010e7 <+133>:   je     0x4010ee <phase_5+140>
   0x00000000004010e9 <+135>:   callq  0x400b30 <__stack_chk_fail@plt>
   0x00000000004010ee <+140>:   add    $0x20,%rsp
   0x00000000004010f2 <+144>:   pop    %rbx
   0x00000000004010f3 <+145>:   retq
End of assembler dump.
```

注意到开头部分调用了 `phase_1` 中的 `string_length` 函数并与 6 作比较, 该关卡的有效输入应是长度为 6 的字符串. 难点在于段寄存器 `fs` 在此处的使用. 调查发现, `fs:0x28` 是 GCC 用于检查栈缓冲区溢出而设置的警惕标志 (canary), 若函数末尾部分检查到该值发生变化, 则发生溢出, 调用 `__stack_chk_fail`. 这些语句的出现说明该关卡的函数体中极有可能声明了某个字符数组并进行操作.

为了验证猜想, 继续参看省略号部分的汇编代码

```
0x000000000040108b <+41>:    movzbl (%rbx,%rax,1),%ecx
0x000000000040108f <+45>:    mov    %cl,(%rsp)
0x0000000000401092 <+48>:    mov    (%rsp),%rdx
0x0000000000401096 <+52>:    and    $0xf,%edx
0x0000000000401099 <+55>:    movzbl 0x4024b0(%rdx),%edx
0x00000000004010a0 <+62>:    mov    %dl,0x10(%rsp,%rax,1)
0x00000000004010a4 <+66>:    add    $0x1,%rax
0x00000000004010a8 <+70>:    cmp    $0x6,%rax
0x00000000004010ac <+74>:    jne    0x40108b <phase_5+41>
0x00000000004010ae <+76>:    movb   $0x0,0x16(%rsp)
0x00000000004010b3 <+81>:    mov    $0x40245e,%esi
0x00000000004010b8 <+86>:    lea    0x10(%rsp),%rdi
0x00000000004010bd <+91>:    callq  0x401338 <strings_not_equal>
0x00000000004010c2 <+96>:    test   %eax,%eax
0x00000000004010c4 <+98>:    je     0x4010d9 <phase_5+119>
```

```
0x00000000004010c6 <+100>:   callq  0x40143a <explode_bomb>
0x00000000004010cb <+105>:   nopl   0x0(%rax,%rax,1)
0x00000000004010d0 <+110>:   jmp    0x4010d9 <phase_5+119>
0x00000000004010d2 <+112>:   mov    $0x0,%eax
0x00000000004010d7 <+117>:   jmp    0x40108b <phase_5+41>
```

梳理大意如下:

1. 前半部分为循环结构, 依次取出输入字符数组 rbx 中的 6 个字符.
2. 每次循环, 取当前字符模 16 的余数为偏移量, 0x4024b0 为基地址, 从内存中取出另一字符, 存入以 rsp+10 为首元素地址的字符数组中.
3. 循环结束后, 调用 strings_not_equal, 将以 rsp+10 为首元素地址的字符数组与位于 0x40245e 的字符串做比较, 若不相等则调用 explode_bomb.

此外, 还有一些与栈缓冲区溢出相关的额外处理. 根据上述逻辑, 我们只需找出字符数组 rbx 与字符数组 rsp+10 间各字符的映射关系, 使得字符数组 rsp+10 的内容与地址 0x40245e 处的内容相同即可.

首先, 查看 0x4024b0 的内容

```
(gdb) x/16bc 0x4024b0
0x4024b0 <array.3449>:  109 'm' 97 'a'  100 'd' 117 'u' 105 'i' 101 'e' 114 'r'
↪  115 's'
0x4024b8 <array.3449+8>:        110 'n' 102 'f' 111 'o' 116 't' 118 'v' 98 'b'
↪  121 'y' 108 'l'
```

可见 0x4024b0 处实际也为字符数组, 作用域为全局.

其次, 查看 0x40245e 的内容

```
(gdb) x/s 0x40245e
0x40245e:       "flyers"
```

知最后要求 rsp+10 处的内容为 “flyers” , 对应数组 0x4024b0 中的下标为 9, 15, 14, 5, 6, 7.

由于映射过程中有模 16 运算, 本关卡的可行解较多, 此处统一取 ASCII 在 96~111 之间的字符, 得到一种可行解的 ASCII 为 105, 111, 110, 101, 102, 103. 写成字符串为”ionefg”.

```
(gdb) jump *0x400ea2
Continuing at 0x400ea2.
ionefg

Breakpoint 5, 0x0000000000401062 in phase_5 ()
(gdb) continue
```

```
Continuing.
Good work!  On to the next...
```

7. phase_6

出于篇幅考虑, phase_6 的完整汇编代码置于附录 phase_6.txt.

该关卡的输入部分为

```
0x0000000000401100 <+12>:    mov    %rsp,%r13
0x0000000000401103 <+15>:    mov    %rsp,%rsi
0x0000000000401106 <+18>:    callq  0x40145c <read_six_numbers>
```

参照 phase_2, 读入的 6 个 32 位整数分别位于 r13, r13+4, r13+8, r13+12, r13+16, r13+20 中, 往后记为 $x[i], i = 0, \cdots, 5$.

随后的 <+32> 到 <+93> 是一个二重循环结构, 用 C 语言表达为

```
for(r12d = 0; r12d < 6; r12d++) {
    if((unsigned)([r12d] - 1) > 5) explode_bomb();
    for(ebx = r12d + 1; ebx < 6; ebx++)
        if(x[r12d] == x[ebx]) explode_bomb();
}
```

可见要求输入的 6 个整数为 1~6 且互不相同.

<+108> 到 <+121> 也是一个循环结构, 用 C 语言表达为

```
for(rax = r14; rax != rsi; rax += 4)
    *rax = 7 - *rax
```

注意此处 r14, rsi 已分别被修改为 $x[0]$ 和 $x[5]$ 的地址, 因此等价于 $x[i] \leftarrow 7 - x[i], i = 0, \cdots, 5$.

<+128> 到 <+181> 也是二重循环结构, 用 C 语言表达为

```
for(rsi = 0; rsi != 24; rsi += 4) {
    ecx = *(x + rsi);
    if(ecx > 1){
        rdx = 0x6032d0;
        for(eax = 1; eax != ecx; eax++) rdx = *(rdx + 8);
    } else
        rdx = 0x6032d0;
    *(rsp + 32 + 2 * rsi) = rdx;
}
```

ecx遍历了 $x[0]$~$x[5]$ 的值. 0x6032d0 较为突兀, 极有可能是地址. rsp+32 处有一数组, 每元素占 8 字节, 可能也是地址. 若尝试查看 0x6032d0 处的数据

```
(gdb) x/24wx 0x6032d0
0x6032d0 <node1>:       0x0000014c      0x00000001      0x006032e0
↪  0x00000000
0x6032e0 <node2>:       0x000000a8      0x00000002      0x006032f0
↪  0x00000000
0x6032f0 <node3>:       0x0000039c      0x00000003      0x00603300
↪  0x00000000
0x603300 <node4>:       0x000002b3      0x00000004      0x00603310
↪  0x00000000
0x603310 <node5>:       0x000001dd      0x00000005      0x00603320
↪  0x00000000
0x603320 <node6>:       0x000001bb      0x00000006      0x00000000
↪  0x00000000
```

发现符号为 node1 ~node6, 可能是某种数据结构, 记“节点”为 $node[i-1], i=1,\cdots,6$. 每个节点包含 3 个数据项, 其中第一个数据项可视作整数, 我们记为 $node[i-1].data$, 第 2 个数据项恰为元素序号 i, 记为 $node[i-1].index$, 第 3 个数据项为指向 $node[i]$ 的指针, 记为 $node[i-1].next$. 那么 rdx = *(rdx + 8) 一句等价于寻找后继节点. 我们断言地址 0x6032d0 处存放的是单向链表, 如图 4 所示.

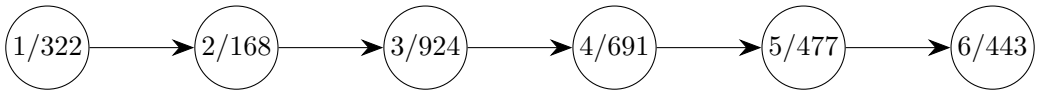

图 4: phase_6 的单链表 (节点中内容为“$index/data$”)

记数组 rsp+32 的元素为 $addr[i], i=0,\cdots,5$, 那么上述循环等价于将 $addr[i]$ 赋值为 $node[x[i]-1]$ 的地址. 说明输入的数据起到索引的作用.

有了上述一系列记号, <+201>~<+235> 的循环结构可以用 C 语言表达为

```
node* rcx = addr;
for(node* rax = addr + 8; rax != addr + 40; rax += 8) {
    rcx -> next = rax;
    rcx = rax;
}
```

等价于 $*(addr[i-1]).next \leftarrow addr[i], i=1,\cdots,5$, 使得 $node[x[i-1]]$ 的后继节点为 $node[x[i]]$

最后一个循环结构 <+235> <+257> 用 C 语言表达为

```
node* rbx = addr;
for(ebp = 5; ebp != 0; ebp--) {
    node* rax = rbx -> next;
    if(rax -> data < rbx -> data)
        explode_bomb();
    rbx = rbx -> next;
}
```

或者说, 要求 $node[x[i]] \geq node[x[i+1]]$, 否则炸弹爆炸.

综合以上分析, 通关思路已经显然易见. 记输入的原始整数为 $d[i], i = 0, \cdots, 5$, 那么当且仅当 $node[6 - d[i]].data \geq node[6 - d[i+1]].data$ 时, 炸弹不会爆炸. 由于

$$node[2].data < node[3].data < node[4].data < node[5].data < node[0].data < node[1].data$$

故答案为

$$d[0] = 4, d[1] = 3, d[2] = 2, d[3] = 1, d[4] = 6, d[5] = 5$$

输入答案, 顺利通过. 输入输出如下

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/hasined/Repositories/Computer-System/labs/bomb/bomb/bomb

Breakpoint 12, printf (__fmt=<optimized out>) at
↪  /usr/include/x86_64-linux-gnu/bits/stdio2.h:105
105     printf (const char *__restrict __fmt, ...)
(gdb) jump *0x400ebe
Line 107 is not in `printf'.  Jump anyway? (y or n) y
Continuing at 0x400ebe.
4 3 2 1 6 5

Breakpoint 6, 0x0000000000004010f4 in phase_6 ()
(gdb) continue
Continuing.
[Inferior 1 (process 6159) exited normally]
```

此处, 由于是直接跳转至 `phase_6`, 程序并未输出通关信息.

8. secret_phase

为了寻找进入隐藏关卡的方法，我们在 disas.txt 中搜索，发现 secret_phase 在 phase_defused 中被调用. 注意到汇编代码[5]中以下两行

```
0x00000000004015d8 <+20>:    cmpl   $0x6,0x202181(%rip)        # 0x603760
↪  <num_input_strings>
0x00000000004015df <+27>:    jne    0x40163f <phase_defused+123>
```

可知当全局变量 num_input_strings 的值为 6 时，才会进行与隐藏关卡相关的判定. 进一步调查，发现该变量在 read_line 函数[6]中被修改

```
0x000000000040151f <+129>:   mov    0x20223b(%rip),%edx        # 0x603760
↪  <num_input_strings>
...                          ...    ...
0x000000000040155e <+192>:   mov    0x2021fc(%rip),%eax        # 0x603760
↪  <num_input_strings>
0x0000000000401564 <+198>:   lea    0x1(%rax),%edx
0x0000000000401567 <+201>:   mov    %edx,0x2021f3(%rip)        # 0x603760
↪  <num_input_strings>
...                          ...    ...
0x00000000004015b3 <+277>:   add    $0x1,%edx
0x00000000004015b6 <+280>:   mov    %edx,0x2021a4(%rip)        # 0x603760
↪  <num_input_strings>
```

中间部分与输入过长时的错误处理有关，此处不赘述. 由末尾部分可知，每次读入一行字符串，都会使 num_input_strings 自增. 当 phase_6 调用结束后，应当有 num_input_strings = 6. 验证如下

```
(gdb) break *0x4015e1
Breakpoint 13 at 0x4015e1
(gdb) run
Starting program: /home/hasined/Repositories/Computer-System/labs/bomb/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
```

[5]见附录 phase_defused.txt

[6]见附录 read_line.txt

```
Halfway there!
0 0
So you got that one.  Try this one.
ionefg
Good work!  On to the next...
4 3 2 1 6 5

Breakpoint 13, 0x00000000004015e1 in phase_defused ()
(gdb) x/w &num_input_strings
0x603760 <num_input_strings>:   6
```

发现确实在前 6 个关卡通过后到达了断点处.

继续查看 phase_defused 的汇编代码, 发现调用了 sscanf, 以 0x402619 为源字符串, 以 0x402619 处的字符串为输入格式. 分别调查 2 个地址的内容

```
(gdb) x/s 0x402619
0x402619:       "%d %d %s"
(gdb) x/s 0x603870
0x603870 <input_strings+240>:   "0 0"
```

发现是从 input_strings+240 读入 2 个整数和 1 个字符串. 根据符号, input_strings 极有可能是输入字符串构成的数组. 由于 input_strings+240 的内容与 phase_4 的输入一致, 合理猜测, 数组为每个串分配的长度为 80 字节. 于是, 以 char[6][80] 为类型查看

```
(gdb) print (char[6][80]) input_strings
$1 = {"Border relations with Canada have never been better.", '\000' <repeats 27
↪  times>, "1 2 4 8 16 32", '\000' <repeats 66 times>,
  "0 207", '\000' <repeats 74 times>, "0 0", '\000' <repeats 76 times>,
  ↪  "ionefg", '\000' <repeats 73 times>,
  "4 3 2 1 6 5", '\000' <repeats 68 times>}
```

猜想得到验证. read_line 函数每次均将输入的字符串存入全局数组 input_strings 中. 对应到汇编代码, 主要为以下语句

```
0x0000000000401528 <+138>:   lea    (%rax,%rax,4),%rsi
0x000000000040152c <+142>:   shl    $0x4,%rsi
0x0000000000401530 <+146>:   add    $0x603780,%rsi
...                          ...    ...
0x0000000000401546 <+168>:   repnz scas %es:(%rdi),%al
```

总而言之, 要进入 secret_phase, 应当在 phase_4 输入额外的字符串. 前文提到,

phase_4 要求 sscanf 输入数据数目严格为 2, 与 phase_3 存在微妙的区别, 已经给予了我们一定的提示.

继续查看 phase_defused 的汇编代码, 发现调用了 strings_not_equal, 要求输入字符串为

```
(gdb) x/s 0x402622
0x402622:       "DrEvil"
```

重新运行程序, 在 phase_4 输入 “0 0 DrEvil” , 成功来到 secret_phase 的断点处.

```
Curses, you've found the secret phase!
But finding it and solving it are quite different...

Breakpoint 7, 0x0000000000401242 in secret_phase ()
```

反汇编得

```
Dump of assembler code for function secret_phase:
=> 0x0000000000401242 <+0>:     push   %rbx
   0x0000000000401243 <+1>:     callq  0x40149e <read_line>
   0x0000000000401248 <+6>:     mov    $0xa,%edx
   0x000000000040124d <+11>:    mov    $0x0,%esi
   0x0000000000401252 <+16>:    mov    %rax,%rdi
   0x0000000000401255 <+19>:    callq  0x400bd0 <strtol@plt>
   0x000000000040125a <+24>:    mov    %rax,%rbx
   0x000000000040125d <+27>:    lea    -0x1(%rax),%eax
   0x0000000000401260 <+30>:    cmp    $0x3e8,%eax
   0x0000000000401265 <+35>:    jbe    0x40126c <secret_phase+42>
   0x0000000000401267 <+37>:    callq  0x40143a <explode_bomb>
   0x000000000040126c <+42>:    mov    %ebx,%esi
   0x000000000040126e <+44>:    mov    $0x6030f0,%edi
   0x0000000000401273 <+49>:    callq  0x401204 <fun7>
   0x0000000000401278 <+54>:    cmp    $0x2,%eax
   0x000000000040127b <+57>:    je     0x401282 <secret_phase+64>
   0x000000000040127d <+59>:    callq  0x40143a <explode_bomb>
   0x0000000000401282 <+64>:    mov    $0x402438,%edi
   0x0000000000401287 <+69>:    callq  0x400b10 <puts@plt>
   0x000000000040128c <+74>:    callq  0x4015c4 <phase_defused>
   0x0000000000401291 <+79>:    pop    %rbx
   0x0000000000401292 <+80>:    retq
End of assembler dump.
```

大意如下:

1. 读入一行字符串, 调用标准库中 `strtol` 函数将其转化为长整型.
2. 检查输入的长整数是否小于 1001, 若为否, 炸弹爆炸.
3. 以输入的整数和 `0x6030f0` 为参数, 调用 `fun7` 函数.
4. 若 `fun7` 返回值为 2, 通过该关卡.

将 `fun7` 反汇编

```
Dump of assembler code for function fun7:
   0x0000000000401204 <+0>:     sub    $0x8,%rsp
   0x0000000000401208 <+4>:     test   %rdi,%rdi
   0x000000000040120b <+7>:     je     0x401238 <fun7+52>
   0x000000000040120d <+9>:     mov    (%rdi),%edx
   0x000000000040120f <+11>:    cmp    %esi,%edx
   0x0000000000401211 <+13>:    jle    0x401220 <fun7+28>
   0x0000000000401213 <+15>:    mov    0x8(%rdi),%rdi
   0x0000000000401217 <+19>:    callq  0x401204 <fun7>
   0x000000000040121c <+24>:    add    %eax,%eax
   0x000000000040121e <+26>:    jmp    0x40123d <fun7+57>
   0x0000000000401220 <+28>:    mov    $0x0,%eax
   0x0000000000401225 <+33>:    cmp    %esi,%edx
   0x0000000000401227 <+35>:    je     0x40123d <fun7+57>
   0x0000000000401229 <+37>:    mov    0x10(%rdi),%rdi
   0x000000000040122d <+41>:    callq  0x401204 <fun7>
   0x0000000000401232 <+46>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401236 <+50>:    jmp    0x40123d <fun7+57>
   0x0000000000401238 <+52>:    mov    $0xffffffff,%eax
   0x000000000040123d <+57>:    add    $0x8,%rsp
   0x0000000000401241 <+61>:    retq
End of assembler dump.
```

发现存在递归结构, 与 `func4` 极为相似. 为了帮助理解, 首先查看 `0x6030f0` 附近的内容

```
(gdb) x/120wd 0x6030f0
0x6030f0 <n1>:  36      0       6304016 0
0x603100 <n1+16>:       6304048 0       0       0
0x603110 <n21>: 8       0       6304144 0
0x603120 <n21+16>:      6304080 0       0       0
0x603130 <n22>: 50      0       6304112 0
0x603140 <n22+16>:      6304176 0       0       0
0x603150 <n32>: 22      0       6304368 0
0x603160 <n32+16>:      6304304 0       0       0
0x603170 <n33>: 45      0       6304208 0
```

```
0x603180 <n33+16>:      6304400 0       0       0
0x603190 <n31>: 6       0       6304240 0
0x6031a0 <n31+16>:      6304336 0       0       0
0x6031b0 <n34>: 107     0       6304272 0
0x6031c0 <n34+16>:      6304432 0       0       0
0x6031d0 <n45>: 40      0       0       0
0x6031e0 <n45+16>:      0       0       0       0
0x6031f0 <n41>: 1       0       0       0
0x603200 <n41+16>:      0       0       0       0
0x603210 <n47>: 99      0       0       0
0x603220 <n47+16>:      0       0       0       0
0x603230 <n44>: 35      0       0       0
0x603240 <n44+16>:      0       0       0       0
0x603250 <n42>: 7       0       0       0
0x603260 <n42+16>:      0       0       0       0
0x603270 <n43>: 20      0       0       0
0x603280 <n43+16>:      0       0       0       0
0x603290 <n46>: 47      0       0       0
0x6032a0 <n46+16>:      0       0       0       0
0x6032b0 <n48>: 1001    0       0       0
0x6032c0 <n48+16>:      0       0       0       0
```

根据符号和 `phase_5` 的经验, 我们猜测这是某种数据结构, 每个节点 32 个字节, 共有 15 个节点. 每个节点 *node* 有 3 个数据项, 第 1 个数据项可视为整数, 记为 *node.data*; 第 2、3 个数据项为指针, 不妨记为 *node.lchild* 和 *node.rchild*. 种种证据暗示我们, `secret_phase` 与 `phase_4` 有很强的关联性. 合理猜测, 这是一棵二叉树, 符号中的数字很可能是节点所在层数以及在该层的位置. 如图 5 所示.

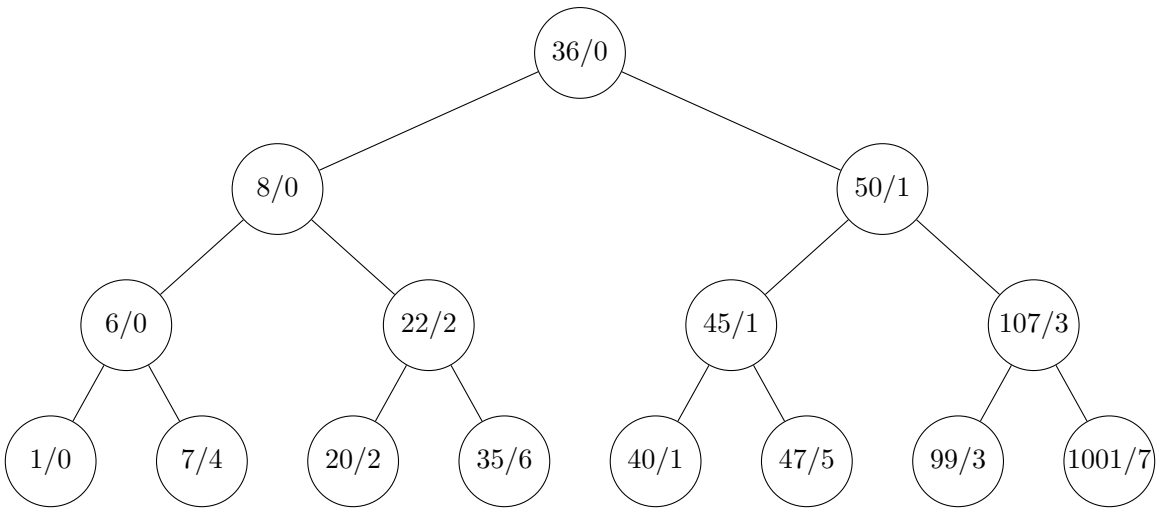

图 5: `secret_phase` 的二叉树表示 (节点中斜杠左侧内容为 *node.data*)

这是一棵二叉搜索树, 再次映证了与 phase_4 的关联性. 于是, 可以将汇编代码翻译为

```
int fun7(int x, node* p) {
    if (!p) return -1;
    if (x < p -> data)
        return fun7(x, p -> lchild) * 2;
    else if (x > p -> data)
        return fun7(x, p -> rchild) * 2 + 1;
    else
        return 0;
}
```

发现逻辑与 phase_4 完全一致, 仍是改写过的二分查找, 每次向左子树递归搜索, 返回值乘 2; 每次向右子树递归搜索, 返回值乘 2 加 1. 根据回溯过程, 若成功查找到 x, 则返回值如图 5 节点中斜杠右侧所示; 若未查找到 x, 返回负值.

要求返回值为 2, 故答案为 20 或 22.

```
(gdb) continue
Continuing.
22
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
[Inferior 1 (process 6228) exited normally]
```

四、 总结

完成 Bomb Lab, 主要有以下收获:

- 提高了阅读 x86-64 汇编代码的能力, 加深了对函数调用、参数传递的理解.
- 理解了 GCC 中 `for`、`switch`、负数除 2、乘 80 等语句的编译方式, 认识到编译器的优化对性能的重要性.
- 熟悉了链表、二叉树等数据结构的汇编表达.
- 学会了在终端中使用 gdb 调试汇编程序, 掌握了基本的反汇编、逆向工程方法.
- 学会了 GNU Binutils 中的实用命令, 借此理解了 ELF 文件的构造, 程序头、节头、符号表、程序段、数据段等概念.
- 理解了编译器对栈缓冲区溢出提供保护的方式.
- 认识到 Docker 容器和 QEMU 的局限性.
- 学会了在 LaTeX 中使用 `tikz` 宏包绘图.

本实验的所有材料已上传至 GitHub:
https://github.com/HasiNed/Computer-System

附录 A　部分输出结果

1. syms.txt

```
0000000000400ac0 T _init
0000000000400c90 T _start
0000000000400da0 T main
0000000000400ee0 T phase_1
0000000000400efc T phase_2
0000000000400f43 T phase_3
0000000000400fce T func4
000000000040100c T phase_4
0000000000401062 T phase_5
00000000004010f4 T phase_6
0000000000401204 T fun7
0000000000401242 T secret_phase
00000000004012f6 T invalid_phase
000000000040131b T string_length
0000000000401338 T strings_not_equal
00000000004013a2 T initialize_bomb
00000000004013ba T initialize_bomb_solve
00000000004013bc T blank_line
00000000004013f9 T skip
000000000040143a T explode_bomb
000000000040145c T read_six_numbers
000000000040149e T read_line
00000000004015c4 T phase_defused
0000000000401660 T sigalrm_handler
00000000004017ac T submitr
0000000000401f91 T init_timeout
0000000000401fb8 T init_driver
000000000040218d T driver_post
0000000000402210 T __libc_csu_init
00000000004022a0 T __libc_csu_fini
00000000004022a4 T _fini
```

2. rodata.txt

```

String dump of section '.rodata':
  [     4]  r
  [     6]  %s: Error: Couldn't open %s^J
  [    23]  Usage: %s [<input_file>]^J
  [    3d]  That's number 2.  Keep going!
  [    5b]  Halfway there!
  [    6a]  Good work!  On to the next...
```

```
[    88]  Welcome to my fiendish little bomb. You have 6 phases with
[    c8]  which to blow yourself up. Have a nice day!
[    f8]  Phase 1 defused. How about the next one?
[   128]  So you got that one.  Try this one.
[   150]  Border relations with Canada have never been better.
[   188]  Wow! You've defused the secret stage!
[   1ae]  flyers
[   1c0]  |^O@
[   1ca]  @
[   1d2]  @
[   1da]  @
[   1e2]  @
[   1ea]  @
[   1f2]  @
[   1fa]  @
[   200]  maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?
[   248]  Curses, you've found the secret phase!
[   270]  But finding it and solving it are quite different...
[   2a8]  Congratulations! You've defused the bomb!
[   2d2]  Well...
[   2da]  OK. :-)
[   2e2]  Invalid phase%s^J
[   2f4]  BOOM!!!
[   2fc]  The bomb has blown up.
[   313]  %d %d %d %d %d %d
[   325]  Error: Premature EOF on stdin
[   343]  GRADE_BOMB
[   34e]  Error: Input line too long
[   369]  %d %d %s
[   372]  DrEvil
[   379]  greatwhite.ics.cs.cmu.edu
[   393]  angelshark.ics.cs.cmu.edu
[   3ad]  makoshark.ics.cs.cmu.edu
[   3c8]  Program timed out after %d seconds^J
[   3f0]  Error: HTTP request failed with error %d: %s
[   420]  GET /%s/submitr.pl/?userid=%s&lab=%s&result=%s&submit=submit HTTP/1.0^M^J^M^J
[   470]  Error: Unable to connect to server %s
[   498]  %%%02X
[   49f]  %s %d %[a-zA-z ]
[   4b0]  changeme.ics.cs.cmu.edu
[   4c9]  AUTORESULT_STRING=%s^J
[   4df]  csapp
```

3. strings_not_equal.txt

```
Dump of assembler code for function strings_not_equal:
   0x0000000000401338 <+0>:     push   %r12
   0x000000000040133a <+2>:     push   %rbp
   0x000000000040133b <+3>:     push   %rbx
   0x000000000040133c <+4>:     mov    %rdi,%rbx
   0x000000000040133f <+7>:     mov    %rsi,%rbp
   0x0000000000401342 <+10>:    callq  0x40131b <string_length>
   0x0000000000401347 <+15>:    mov    %eax,%r12d
   0x000000000040134a <+18>:    mov    %rbp,%rdi
   0x000000000040134d <+21>:    callq  0x40131b <string_length>
   0x0000000000401352 <+26>:    mov    $0x1,%edx
   0x0000000000401357 <+31>:    cmp    %eax,%r12d
   0x000000000040135a <+34>:    jne    0x40139b <strings_not_equal+99>
   0x000000000040135c <+36>:    movzbl (%rbx),%eax
   0x000000000040135f <+39>:    test   %al,%al
   0x0000000000401361 <+41>:    je     0x401388 <strings_not_equal+80>
   0x0000000000401363 <+43>:    cmp    0x0(%rbp),%al
   0x0000000000401366 <+46>:    je     0x401372 <strings_not_equal+58>
   0x0000000000401368 <+48>:    jmp    0x40138f <strings_not_equal+87>
   0x000000000040136a <+50>:    cmp    0x0(%rbp),%al
   0x000000000040136d <+53>:    nopl   (%rax)
   0x0000000000401370 <+56>:    jne    0x401396 <strings_not_equal+94>
   0x0000000000401372 <+58>:    add    $0x1,%rbx
   0x0000000000401376 <+62>:    add    $0x1,%rbp
   0x000000000040137a <+66>:    movzbl (%rbx),%eax
   0x000000000040137d <+69>:    test   %al,%al
   0x000000000040137f <+71>:    jne    0x40136a <strings_not_equal+50>
   0x0000000000401381 <+73>:    mov    $0x0,%edx
   0x0000000000401386 <+78>:    jmp    0x40139b <strings_not_equal+99>
   0x0000000000401388 <+80>:    mov    $0x0,%edx
   0x000000000040138d <+85>:    jmp    0x40139b <strings_not_equal+99>
   0x000000000040138f <+87>:    mov    $0x1,%edx
   0x0000000000401394 <+92>:    jmp    0x40139b <strings_not_equal+99>
   0x0000000000401396 <+94>:    mov    $0x1,%edx
   0x000000000040139b <+99>:    mov    %edx,%eax
   0x000000000040139d <+101>:   pop    %rbx
   0x000000000040139e <+102>:   pop    %rbp
   0x000000000040139f <+103>:   pop    %r12
   0x00000000004013a1 <+105>:   retq
End of assembler dump.
```

4. read_six_numbers.txt

```
Dump of assembler code for function read_six_numbers:
   0x000000000040145c <+0>:     sub    $0x18,%rsp
   0x0000000000401460 <+4>:     mov    %rsi,%rdx
   0x0000000000401463 <+7>:     lea    0x4(%rsi),%rcx
   0x0000000000401467 <+11>:    lea    0x14(%rsi),%rax
   0x000000000040146b <+15>:    mov    %rax,0x8(%rsp)
   0x0000000000401470 <+20>:    lea    0x10(%rsi),%rax
   0x0000000000401474 <+24>:    mov    %rax,(%rsp)
   0x0000000000401478 <+28>:    lea    0xc(%rsi),%r9
   0x000000000040147c <+32>:    lea    0x8(%rsi),%r8
   0x0000000000401480 <+36>:    mov    $0x4025c3,%esi
   0x0000000000401485 <+41>:    mov    $0x0,%eax
   0x000000000040148a <+46>:    callq  0x400bf0 <__isoc99_sscanf@plt>
   0x000000000040148f <+51>:    cmp    $0x5,%eax
   0x0000000000401492 <+54>:    jg     0x401499 <read_six_numbers+61>
   0x0000000000401494 <+56>:    callq  0x40143a <explode_bomb>
   0x0000000000401499 <+61>:    add    $0x18,%rsp
   0x000000000040149d <+65>:    retq
End of assembler dump.
```

5. func4.txt

```
Dump of assembler code for function func4:
   0x0000000000400fce <+0>:     sub    $0x8,%rsp
   0x0000000000400fd2 <+4>:     mov    %edx,%eax
   0x0000000000400fd4 <+6>:     sub    %esi,%eax
   0x0000000000400fd6 <+8>:     mov    %eax,%ecx
   0x0000000000400fd8 <+10>:    shr    $0x1f,%ecx
   0x0000000000400fdb <+13>:    add    %ecx,%eax
   0x0000000000400fdd <+15>:    sar    %eax
   0x0000000000400fdf <+17>:    lea    (%rax,%rsi,1),%ecx
   0x0000000000400fe2 <+20>:    cmp    %edi,%ecx
   0x0000000000400fe4 <+22>:    jle    0x400ff2 <func4+36>
   0x0000000000400fe6 <+24>:    lea    -0x1(%rcx),%edx
   0x0000000000400fe9 <+27>:    callq  0x400fce <func4>
   0x0000000000400fee <+32>:    add    %eax,%eax
   0x0000000000400ff0 <+34>:    jmp    0x401007 <func4+57>
   0x0000000000400ff2 <+36>:    mov    $0x0,%eax
   0x0000000000400ff7 <+41>:    cmp    %edi,%ecx
   0x0000000000400ff9 <+43>:    jge    0x401007 <func4+57>
   0x0000000000400ffb <+45>:    lea    0x1(%rcx),%esi
   0x0000000000400ffe <+48>:    callq  0x400fce <func4>
   0x0000000000401003 <+53>:    lea    0x1(%rax,%rax,1),%eax
   0x0000000000401007 <+57>:    add    $0x8,%rsp
```

```
   0x000000000040100b <+61>:    retq
End of assembler dump.
```

6. phase_6.txt

```
Dump of assembler code for function phase_6:
=> 0x00000000004010f4 <+0>:     push   %r14
   0x00000000004010f6 <+2>:     push   %r13
   0x00000000004010f8 <+4>:     push   %r12
   0x00000000004010fa <+6>:     push   %rbp
   0x00000000004010fb <+7>:     push   %rbx
   0x00000000004010fc <+8>:     sub    $0x50,%rsp
   0x0000000000401100 <+12>:    mov    %rsp,%r13
   0x0000000000401103 <+15>:    mov    %rsp,%rsi
   0x0000000000401106 <+18>:    callq  0x40145c <read_six_numbers>
   0x000000000040110b <+23>:    mov    %rsp,%r14
   0x000000000040110e <+26>:    mov    $0x0,%r12d
   0x0000000000401114 <+32>:    mov    %r13,%rbp
   0x0000000000401117 <+35>:    mov    0x0(%r13),%eax
   0x000000000040111b <+39>:    sub    $0x1,%eax
   0x000000000040111e <+42>:    cmp    $0x5,%eax
   0x0000000000401121 <+45>:    jbe    0x401128 <phase_6+52>
   0x0000000000401123 <+47>:    callq  0x40143a <explode_bomb>
   0x0000000000401128 <+52>:    add    $0x1,%r12d
   0x000000000040112c <+56>:    cmp    $0x6,%r12d
   0x0000000000401130 <+60>:    je     0x401153 <phase_6+95>
   0x0000000000401132 <+62>:    mov    %r12d,%ebx
   0x0000000000401135 <+65>:    movslq %ebx,%rax
   0x0000000000401138 <+68>:    mov    (%rsp,%rax,4),%eax
   0x000000000040113b <+71>:    cmp    %eax,0x0(%rbp)
   0x000000000040113e <+74>:    jne    0x401145 <phase_6+81>
   0x0000000000401140 <+76>:    callq  0x40143a <explode_bomb>
   0x0000000000401145 <+81>:    add    $0x1,%ebx
   0x0000000000401148 <+84>:    cmp    $0x5,%ebx
   0x000000000040114b <+87>:    jle    0x401135 <phase_6+65>
   0x000000000040114d <+89>:    add    $0x4,%r13
   0x0000000000401151 <+93>:    jmp    0x401114 <phase_6+32>
   0x0000000000401153 <+95>:    lea    0x18(%rsp),%rsi
   0x0000000000401158 <+100>:   mov    %r14,%rax
   0x000000000040115b <+103>:   mov    $0x7,%ecx
   0x0000000000401160 <+108>:   mov    %ecx,%edx
   0x0000000000401162 <+110>:   sub    (%rax),%edx
   0x0000000000401164 <+112>:   mov    %edx,(%rax)
   0x0000000000401166 <+114>:   add    $0x4,%rax
   0x000000000040116a <+118>:   cmp    %rsi,%rax
   0x000000000040116d <+121>:   jne    0x401160 <phase_6+108>
```

```
   0x000000000040116f <+123>:   mov    $0x0,%esi
   0x0000000000401174 <+128>:   jmp    0x401197 <phase_6+163>
   0x0000000000401176 <+130>:   mov    0x8(%rdx),%rdx
   0x000000000040117a <+134>:   add    $0x1,%eax
   0x000000000040117d <+137>:   cmp    %ecx,%eax
   0x000000000040117f <+139>:   jne    0x401176 <phase_6+130>
   0x0000000000401181 <+141>:   jmp    0x401188 <phase_6+148>
   0x0000000000401183 <+143>:   mov    $0x6032d0,%edx
   0x0000000000401188 <+148>:   mov    %rdx,0x20(%rsp,%rsi,2)
   0x000000000040118d <+153>:   add    $0x4,%rsi
   0x0000000000401191 <+157>:   cmp    $0x18,%rsi
   0x0000000000401195 <+161>:   je     0x4011ab <phase_6+183>
   0x0000000000401197 <+163>:   mov    (%rsp,%rsi,1),%ecx
   0x000000000040119a <+166>:   cmp    $0x1,%ecx
   0x000000000040119d <+169>:   jle    0x401183 <phase_6+143>
   0x000000000040119f <+171>:   mov    $0x1,%eax
   0x00000000004011a4 <+176>:   mov    $0x6032d0,%edx
   0x00000000004011a9 <+181>:   jmp    0x401176 <phase_6+130>
   0x00000000004011ab <+183>:   mov    0x20(%rsp),%rbx
   0x00000000004011b0 <+188>:   lea    0x28(%rsp),%rax
   0x00000000004011b5 <+193>:   lea    0x50(%rsp),%rsi
   0x00000000004011ba <+198>:   mov    %rbx,%rcx
   0x00000000004011bd <+201>:   mov    (%rax),%rdx
   0x00000000004011c0 <+204>:   mov    %rdx,0x8(%rcx)
   0x00000000004011c4 <+208>:   add    $0x8,%rax
   0x00000000004011c8 <+212>:   cmp    %rsi,%rax
   0x00000000004011cb <+215>:   je     0x4011d2 <phase_6+222>
   0x00000000004011cd <+217>:   mov    %rdx,%rcx
   0x00000000004011d0 <+220>:   jmp    0x4011bd <phase_6+201>
   0x00000000004011d2 <+222>:   movq   $0x0,0x8(%rdx)
   0x00000000004011da <+230>:   mov    $0x5,%ebp
   0x00000000004011df <+235>:   mov    0x8(%rbx),%rax
   0x00000000004011e3 <+239>:   mov    (%rax),%eax
   0x00000000004011e5 <+241>:   cmp    %eax,(%rbx)
   0x00000000004011e7 <+243>:   jge    0x4011ee <phase_6+250>
   0x00000000004011e9 <+245>:   callq  0x40143a <explode_bomb>
   0x00000000004011ee <+250>:   mov    0x8(%rbx),%rbx
   0x00000000004011f2 <+254>:   sub    $0x1,%ebp
   0x00000000004011f5 <+257>:   jne    0x4011df <phase_6+235>
   0x00000000004011f7 <+259>:   add    $0x50,%rsp
   0x00000000004011fb <+263>:   pop    %rbx
   0x00000000004011fc <+264>:   pop    %rbp
   0x00000000004011fd <+265>:   pop    %r12
   0x00000000004011ff <+267>:   pop    %r13
   0x0000000000401201 <+269>:   pop    %r14
   0x0000000000401203 <+271>:   retq
End of assembler dump.
```

7. phase_defused.txt

```
Dump of assembler code for function phase_defused:
   0x00000000004015c4 <+0>:     sub    $0x78,%rsp
   0x00000000004015c8 <+4>:     mov    %fs:0x28,%rax
   0x00000000004015d1 <+13>:    mov    %rax,0x68(%rsp)
   0x00000000004015d6 <+18>:    xor    %eax,%eax
   0x00000000004015d8 <+20>:    cmpl   $0x6,0x202181(%rip)        # 0x603760
   ↪  <num_input_strings>
   0x00000000004015df <+27>:    jne    0x40163f <phase_defused+123>
   0x00000000004015e1 <+29>:    lea    0x10(%rsp),%r8
   0x00000000004015e6 <+34>:    lea    0xc(%rsp),%rcx
   0x00000000004015eb <+39>:    lea    0x8(%rsp),%rdx
   0x00000000004015f0 <+44>:    mov    $0x402619,%esi
   0x00000000004015f5 <+49>:    mov    $0x603870,%edi
   0x00000000004015fa <+54>:    callq  0x400bf0 <__isoc99_sscanf@plt>
   0x00000000004015ff <+59>:    cmp    $0x3,%eax
   0x0000000000401602 <+62>:    jne    0x401635 <phase_defused+113>
   0x0000000000401604 <+64>:    mov    $0x402622,%esi
   0x0000000000401609 <+69>:    lea    0x10(%rsp),%rdi
   0x000000000040160e <+74>:    callq  0x401338 <strings_not_equal>
   0x0000000000401613 <+79>:    test   %eax,%eax
   0x0000000000401615 <+81>:    jne    0x401635 <phase_defused+113>
   0x0000000000401617 <+83>:    mov    $0x4024f8,%edi
   0x000000000040161c <+88>:    callq  0x400b10 <puts@plt>
   0x0000000000401621 <+93>:    mov    $0x402520,%edi
   0x0000000000401626 <+98>:    callq  0x400b10 <puts@plt>
   0x000000000040162b <+103>:   mov    $0x0,%eax
   0x0000000000401630 <+108>:   callq  0x401242 <secret_phase>
   0x0000000000401635 <+113>:   mov    $0x402558,%edi
   0x000000000040163a <+118>:   callq  0x400b10 <puts@plt>
   0x000000000040163f <+123>:   mov    0x68(%rsp),%rax
   0x0000000000401644 <+128>:   xor    %fs:0x28,%rax
   0x000000000040164d <+137>:   je     0x401654 <phase_defused+144>
   0x000000000040164f <+139>:   callq  0x400b30 <__stack_chk_fail@plt>
   0x0000000000401654 <+144>:   add    $0x78,%rsp
   0x0000000000401658 <+148>:   retq
End of assembler dump.
```

8. read_line.txt

```
Dump of assembler code for function read_line:
   0x000000000040149e <+0>:     sub    $0x8,%rsp
   0x00000000004014a2 <+4>:     mov    $0x0,%eax
   0x00000000004014a7 <+9>:     callq  0x4013f9 <skip>
   0x00000000004014ac <+14>:    test   %rax,%rax
```

```
0x00000000004014af <+17>:    jne    0x40151f <read_line+129>
0x00000000004014b1 <+19>:    mov    0x202290(%rip),%rax        # 0x603748
↪ <stdin@@GLIBC_2.2.5>
0x00000000004014b8 <+26>:    cmp    %rax,0x2022a9(%rip)        # 0x603768 <infile>
0x00000000004014bf <+33>:    jne    0x4014d5 <read_line+55>
0x00000000004014c1 <+35>:    mov    $0x4025d5,%edi
0x00000000004014c6 <+40>:    callq  0x400b10 <puts@plt>
0x00000000004014cb <+45>:    mov    $0x8,%edi
0x00000000004014d0 <+50>:    callq  0x400c20 <exit@plt>
0x00000000004014d5 <+55>:    mov    $0x4025f3,%edi
0x00000000004014da <+60>:    callq  0x400ae0 <getenv@plt>
0x00000000004014df <+65>:    test   %rax,%rax
0x00000000004014e2 <+68>:    je     0x4014ee <read_line+80>
0x00000000004014e4 <+70>:    mov    $0x0,%edi
0x00000000004014e9 <+75>:    callq  0x400c20 <exit@plt>
0x00000000004014ee <+80>:    mov    0x202253(%rip),%rax        # 0x603748
↪ <stdin@@GLIBC_2.2.5>
0x00000000004014f5 <+87>:    mov    %rax,0x20226c(%rip)        # 0x603768 <infile>
0x00000000004014fc <+94>:    mov    $0x0,%eax
0x0000000000401501 <+99>:    callq  0x4013f9 <skip>
0x0000000000401506 <+104>:   test   %rax,%rax
0x0000000000401509 <+107>:   jne    0x40151f <read_line+129>
0x000000000040150b <+109>:   mov    $0x4025d5,%edi
0x0000000000401510 <+114>:   callq  0x400b10 <puts@plt>
0x0000000000401515 <+119>:   mov    $0x0,%edi
0x000000000040151a <+124>:   callq  0x400c20 <exit@plt>
0x000000000040151f <+129>:   mov    0x20223b(%rip),%edx        # 0x603760
↪ <num_input_strings>
0x0000000000401525 <+135>:   movslq %edx,%rax
0x0000000000401528 <+138>:   lea    (%rax,%rax,4),%rsi
0x000000000040152c <+142>:   shl    $0x4,%rsi
0x0000000000401530 <+146>:   add    $0x603780,%rsi
0x0000000000401537 <+153>:   mov    %rsi,%rdi
0x000000000040153a <+156>:   mov    $0x0,%eax
0x000000000040153f <+161>:   mov    $0xffffffffffffffff,%rcx
0x0000000000401546 <+168>:   repnz scas %es:(%rdi),%al
0x0000000000401548 <+170>:   not    %rcx
0x000000000040154b <+173>:   sub    $0x1,%rcx
0x000000000040154f <+177>:   cmp    $0x4e,%ecx
0x0000000000401552 <+180>:   jle    0x40159a <read_line+252>
0x0000000000401554 <+182>:   mov    $0x4025fe,%edi
0x0000000000401559 <+187>:   callq  0x400b10 <puts@plt>
0x000000000040155e <+192>:   mov    0x2021fc(%rip),%eax        # 0x603760
↪ <num_input_strings>
0x0000000000401564 <+198>:   lea    0x1(%rax),%edx
0x0000000000401567 <+201>:   mov    %edx,0x2021f3(%rip)        # 0x603760
↪ <num_input_strings>
0x000000000040156d <+207>:   cltq
```

```
   0x000000000040156f <+209>:   imul   $0x50,%rax,%rax
   0x0000000000401573 <+213>:   movabs $0x636e7572742a2a2a,%rdi
   0x000000000040157d <+223>:   mov    %rdi,0x603780(%rax)
   0x0000000000401584 <+230>:   movabs $0x2a2a2a64657461,%rdi
   0x000000000040158e <+240>:   mov    %rdi,0x603788(%rax)
   0x0000000000401595 <+247>:   callq  0x40143a <explode_bomb>
   0x000000000040159a <+252>:   sub    $0x1,%ecx
   0x000000000040159d <+255>:   movslq %ecx,%rcx
   0x00000000004015a0 <+258>:   movslq %edx,%rax
   0x00000000004015a3 <+261>:   lea    (%rax,%rax,4),%rax
   0x00000000004015a7 <+265>:   shl    $0x4,%rax
   0x00000000004015ab <+269>:   movb   $0x0,0x603780(%rcx,%rax,1)
   0x00000000004015b3 <+277>:   add    $0x1,%edx
   0x00000000004015b6 <+280>:   mov    %edx,0x2021a4(%rip)        # 0x603760
   ↪  <num_input_strings>
   0x00000000004015bc <+286>:   mov    %rsi,%rax
   0x00000000004015bf <+289>:   add    $0x8,%rsp
   0x00000000004015c3 <+293>:   retq
End of assembler dump.
```