哈爾濱工業大學

实验报告

实 验（三）

题　　目 Binary Bomb

二进制炸弹

专　　业 计算学部

学　　号 1190202128

班　　级 1903002

学　　生 林搏海

指 导 教 师 郑贵滨

实 验 地 点 G704

实 验 日 期 2021.4.23

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

熟练掌握计算机系统的 ISA 指令系统与寻址方式

熟练掌握 Linux 下调试器的反汇编调试跟踪分析机器语言的方法

增强对程序机器级表示、汇编语言、调试器和逆向工程等的理解

1.2 实验环境与工具

1.2.1 硬件环境

I7-9750H CPU　8g 内存　983.58G 硬盘　GTX1650 显卡

1.2.2 软件环境

VirtualBox 6.1.18　Ubuntu20.04.2LTS

1.2.3 开发工具

VIM8.1.2269　Visual Studio Code1.54.3

1.3 实验预习

上实验课前，必须认真预习实验指导书（PPT 或 PDF）

了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

请写出 C 语言下包含字符串比较、循环、分支（含 switch）、函数调用、递归、指针、结构、链表等的例子程序 sample.c。

生成执行程序 sample.out。

用 gcc –S 或 CodeBlocks 或 GDB 或 OBJDUMP 等，反汇编，比较。

列出每一部分的 C 语言对应的汇编语言。

修改编译选项-O (缺省 2)、O0、O1、O2、O3，-m32/m64。再次查看生成的汇

编语言与原来的区别。

注意 O1 之后无栈帧，EBP 做别的用途。-fno-omit-frame-pointer 加上栈指针。

GDB 命令详解 –tui 模式 ^XA 切换 layout 改变等等

有目的地学习: 看 VS 的功能 GDB 命令用什么？

第 2 章 实验环境建立

2.1 Ubuntu 下 CodeBlocks 反汇编（10 分）

CodeBlocks 运行 hellolinux.c。反汇编查看 printf 函数的实现。

要求：C、ASM、内存(显示 hello 等内容)、堆栈（call printf 前）、寄存器同时在一个窗口。

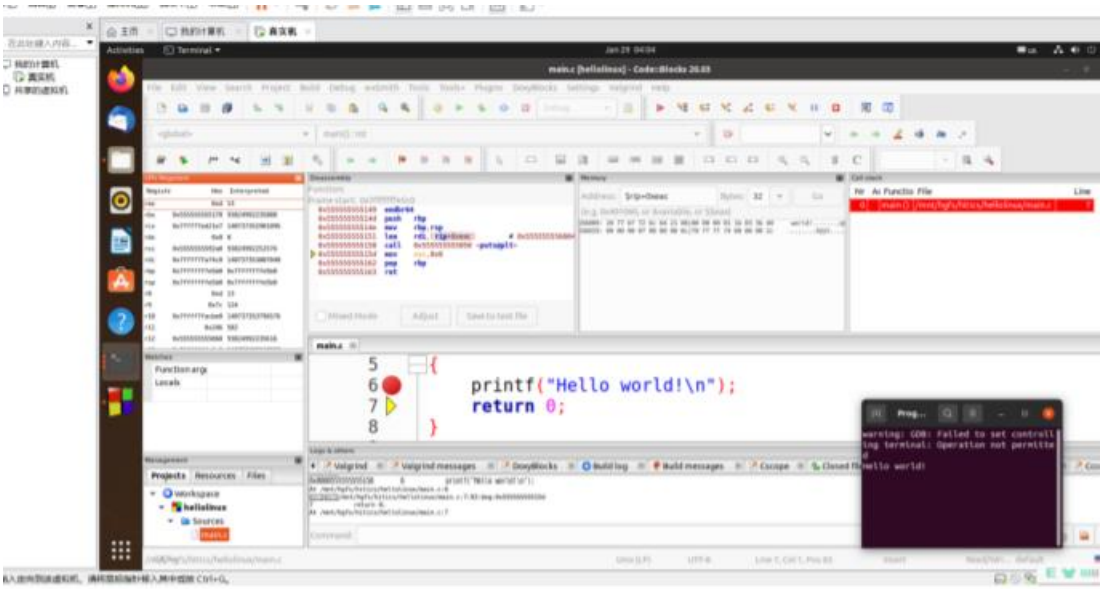


图 2-1 Ubuntu 下 CodeBlocks 反汇编截图

2.2 Ubuntu 下 EDB 运行环境建立（10 分）

用 EDB 调试 hellolinux.c 的执行文件，截图，要求同 2.1

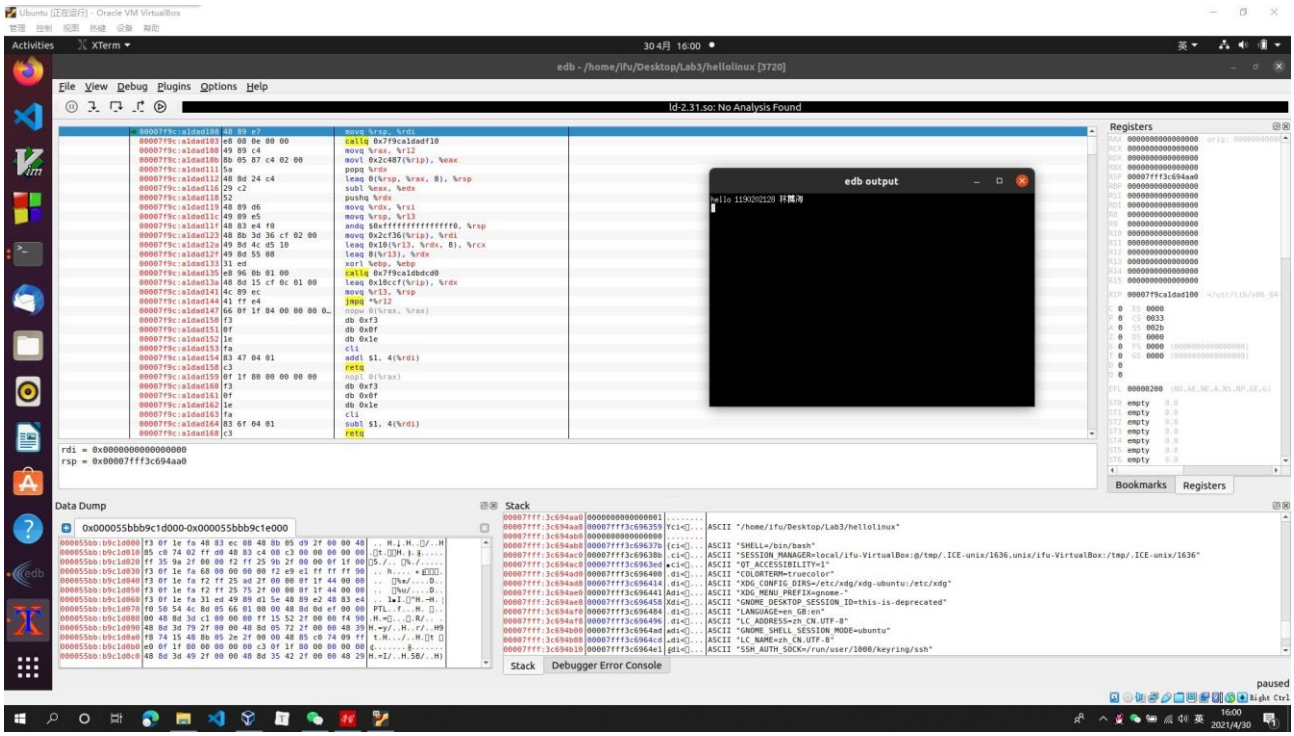


图 2-2　Ubuntu 下 EDB 截图

第 3 章 各阶段炸弹破解与分析

每阶段 15 分（密码 10 分，分析 5 分），总分不超过 80 分

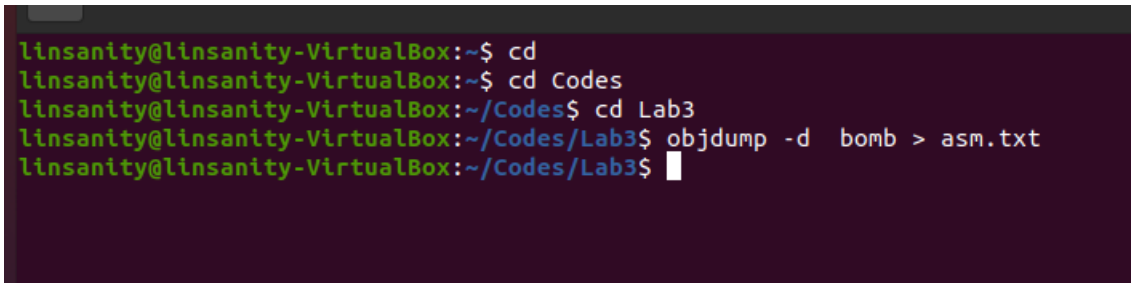
3.1 阶段 1 的破解与分析

密码如下：

Public speaking is very easy.

破解过程：

首先，我们使用 cd 指令进入可执行文件 bomb 所在的目录，然后使用 objdump -d bomb > asm.txt 指令将其反汇编并且将反汇编文件重定向到 asm.txt（如下图）



得到的反汇编代码文本文件如图所示：

```
1
2 bomb:     文件格式 elf64-x86-64
3
4
5 Disassembly of section .init:
6
7 0000000000401000 <_init>:
8   401000:       f3 0f 1e fa             endbr64
9   401004:       48 83 ec 08             sub    $0x8,%rsp
10  401008:       48 8b 05 e9 3f 00 00    mov    0x3fe9(%rip),%rax        # 404ff8 <__gmon_start__>
11  40100f:       48 85 c0                test   %rax,%rax
12  401012:       74 02                   je     401016 <_init+0x16>
13  401014:       ff d0                   callq  *%rax
14  401016:       48 83 c4 08             add    $0x8,%rsp
15  40101a:       c3                      retq
16
17 Disassembly of section .plt:
18
19 0000000000401020 <.plt>:
20  401020:       ff 35 e2 3f 00 00       pushq  0x3fe2(%rip)        # 405008 <_GLOBAL_OFFSET_TABLE_+0x8>
21  401026:       ff 25 e4 3f 00 00       jmpq   *0x3fe4(%rip)        # 405010 <_GLOBAL_OFFSET_TABLE_+0x10>
22  40102c:       0f 1f 40 00             nopl   0x0(%rax)
23
24 0000000000401030 <getenv@plt>:
```

可以看到文件首行并非 main 函数字段，搜索 main 字段，定位 main 函数部分的反汇编代码如下：

```
227
228 00000000004012a6 <main>:
229   4012a6:       55                      push   %rbp
230   4012a7:       48 89 e5                mov    %rsp,%rbp
231   4012aa:       53                      push   %rbx
232   4012ab:       48 83 ec 08             sub    $0x8,%rsp
233   4012af:       83 ff 01                cmp    $0x1,%edi
234   4012b2:       0f 84 ed 00 00 00       je     4013a5 <main+0xff>
235   4012b8:       48 89 f3                mov    %rsi,%rbx
236   4012bb:       83 ff 02                cmp    $0x2,%edi
237   4012be:       0f 85 14 01 00 00       jne    4013d8 <main+0x132>
238   4012c4:       48 8b 7e 08             mov    0x8(%rsi),%rdi
239   4012c8:       be 04 30 40 00          mov    $0x403004,%esi
240   4012cd:       e8 5e fe ff ff          callq  401130 <fopen@plt>
241   4012d2:       48 89 05 97 44 00 00    mov    %rax,0x4497(%rip)        # 405770 <infile>
242   4012d9:       48 85 c0                test   %rax,%rax
243   4012dc:       0f 84 d6 00 00 00       je     4013b8 <main+0x112>
244   4012e2:       e8 82 05 00 00          callq  401869 <initialize_bomb>
245   4012e7:       bf 88 30 40 00          mov    $0x403088,%edi
246   4012ec:       e8 6f fd ff ff          callq  401060 <puts@plt>
247   4012f1:       bf c8 30 40 00          mov    $0x4030c8,%edi
248   4012f6:       e8 65 fd ff ff          callq  401060 <puts@plt>
249   4012fb:       e8 66 06 00 00          callq  401966 <read_line>
250   401300:       48 89 c7                mov    %rax,%rdi
251   401303:       e8 f1 00 00 00          callq  4013f9 <phase_1>
252   401308:       e8 84 07 00 00          callq  401a91 <phase_defused>
253   40130d:       bf f8 30 40 00          mov    $0x4030f8,%edi
```

可以看到，main 函数中读取了首个输入的字符串，并将其首地址通过寄存器 rdi 传递到函数 phase_1 中，由此可以判断在 phase_1 中进行了字符串比较以此判断是否拆除炸弹 1。根据 main 函数的跳转地址找到 phase_1 函数部分（如下图）

```
305
306 00000000004013f9 <phase_1>:
307   4013f9:       55                      push   %rbp
308   4013fa:       48 89 e5                mov    %rsp,%rbp
309   4013fd:       be 4c 31 40 00          mov    $0x40314c,%esi
310   401402:       e8 05 04 00 00          callq  40180c <strings_not_equal>
311   401407:       85 c0                   test   %eax,%eax
312   401409:       75 02                   jne    40140d <phase_1+0x14>
313   40140b:       5d                      pop    %rbp
314   40140c:       c3                      retq
315   40140d:       e8 f6 04 00 00          callq  401908 <explode_bomb>
316   401412:       eb f7                   jmp    40140b <phase_1+0x12>
317
```

我们发现 phase_1 函数继续将我们输入的字符串首地址放在 rdi 中作为第一个参数传递到下一个函数 strings_not_equal 中，此外还通过寄存器 esi 传递了一个常数地址，可以推断出这个地址为正确答案字符串的地址，因此我们需要的字符串答案就存储在这个地址指向的空间。

分析至此，我们可以在 gdb 中打开 bomb 文件，通过 gdb 以字符串的形式打印地址 0x40314c 的值即可，而为了使程序在爆炸结束之前停下，我们可在 phase_1 函数第一行的位置放入断点，如下图所示：

```
linsanity@linsanity-VirtualBox:~/Codes/Lab3$ gdb bomb
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 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-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://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) b phase_1
Breakpoint 1 at 0x4013f9: file phases.c, line 20.
(gdb) r
Starting program: /home/linsanity/Codes/Lab3/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
test_strings

Breakpoint 1, phase_1 (input=0x405780 <input_strings> "test_strings") at phases.c:20
20      phases.c: 没有那个文件或目录.
(gdb) x/s 0x40314c
0x40314c:       "Public speaking is very easy."
(gdb)
```

因此我们得到第一题的答案”Public speaking is very easy.”

重新在终端运行 bomb 程序验证得出答案的正确性（如下图）

```
linsanity@linsanity-VirtualBox:~/Codes/Lab3$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Public speaking is very easy.
Phase 1 defused. How about the next one?
```

3.2 阶段 2 的破解与分析

密码如下：5 6 8 11 15 20（任意一个间距依次为 1 2 3 4 5，首个数大于 0 的整数串）

破解过程：

首先查看 main 函数，发现在 main 函数中同样将我们输入的第二个答案串的首地址放在 rdi 传送到函数 phase_2，phase_2 函数反汇编代码如图：

```
318 0000000000401414 <phase_2>:
319   401414:       55                      push   %rbp
320   401415:       48 89 e5                mov    %rsp,%rbp
321   401418:       53                      push   %rbx
322   401419:       48 83 ec 28             sub    $0x28,%rsp
323   40141d:       48 8d 75 d0             lea    -0x30(%rbp),%rsi
324   401421:       e8 04 05 00 00          callq  40192a <read_six_numbers>
325   401426:       83 7d d0 00             cmpl   $0x0,-0x30(%rbp)
326   40142a:       78 07                   js     401433 <phase_2+0x1f>
327   40142c:       bb 01 00 00 00          mov    $0x1,%ebx
328   401431:       eb 0f                   jmp    401442 <phase_2+0x2e>
329   401433:       e8 d0 04 00 00          callq  401908 <explode_bomb>
330   401438:       eb f2                   jmp    40142c <phase_2+0x18>
331   40143a:       e8 c9 04 00 00          callq  401908 <explode_bomb>
332   40143f:       83 c3 01                add    $0x1,%ebx
333   401442:       83 fb 05                cmp    $0x5,%ebx
334   401445:       7f 17                   jg     40145e <phase_2+0x4a>
335   401447:       48 63 c3                movslq %ebx,%rax
336   40144a:       8d 53 ff                lea    -0x1(%rbx),%edx
337   40144d:       48 63 d2                movslq %edx,%rdx
338   401450:       89 d9                   mov    %ebx,%ecx
339   401452:       03 4c 95 d0             add    -0x30(%rbp,%rdx,4),%ecx
340   401456:       39 4c 85 d0             cmp    %ecx,-0x30(%rbp,%rax,4)
341   40145a:       74 e3                   je     40143f <phase_2+0x2b>
342   40145c:       eb dc                   jmp    40143a <phase_2+0x26>
343   40145e:       48 83 c4 28             add    $0x28,%rsp
344   401462:       5b                      pop    %rbx
345   401463:       5d                      pop    %rbp
346   401464:       c3                      retq
347
```

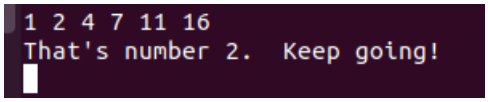
可以看到函数在运行时栈上开辟了 40 字节的空间并调用了 read_six_numbers 函数，推测可能将我们第二阶段输入的字符串转换为 6 个数存储在新开的栈空间上，我们根据跳转地址找到 read_six_numbers 的反汇编代码进行查看如下：

```
761 000000000040192a <read_six_numbers>:
762   40192a:       55                      push   %rbp
763   40192b:       48 89 e5                mov    %rsp,%rbp
764   40192e:       48 89 f2                mov    %rsi,%rdx
765   401931:       48 8d 4e 04             lea    0x4(%rsi),%rcx
766   401935:       48 8d 46 14             lea    0x14(%rsi),%rax
767   401939:       50                      push   %rax
768   40193a:       48 8d 46 10             lea    0x10(%rsi),%rax
769   40193e:       50                      push   %rax
770   40193f:       4c 8d 4e 0c             lea    0xc(%rsi),%r9
771   401943:       4c 8d 46 08             lea    0x8(%rsi),%r8
772   401947:       be 2b 33 40 00          mov    $0x40332b,%esi
773   40194c:       b8 00 00 00 00          mov    $0x0,%eax
774   401951:       e8 ba f7 ff ff          callq  401110 <__isoc99_sscanf@plt>
775   401956:       48 83 c4 10             add    $0x10,%rsp
776   40195a:       83 f8 05                cmp    $0x5,%eax
777   40195d:       7e 02                   jle    401961 <read_six_numbers+0x37>
778   40195f:       c9                      leaveq
779   401960:       c3                      retq
780   401961:       e8 a2 ff ff ff          callq  401908 <explode_bomb>
781
```

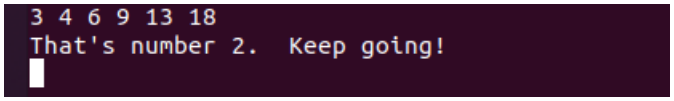
证实了我们的推测，继续回到 phase_2 代码，在 40142a 处的代码将第一个数

与 0 进行了比较，要求必须大于 0.

我们发现接下来进行了一个循环，一次判断 2，3,4,5,6 个数和与上一个数加上 1,2,3,4,5,是否相等，弱不相等就引爆炸弹，因此，最后得到了一个总共有 6 项，首项大于 0，并且相邻两项之间的差依次为 1,2,3,4,5 的数列，我们运行 bomb 程序，验证答案：



发现答案正确，换一组输入：



同样正确，因此我们给出的答案是正确的。

3.3 阶段 3 的破解与分析

密码如下：

0 577

1 183

2 200

3 814

4 549

5 654

6 594

7 740

其中的任意一组

破解过程：

我们分析 main 函数，同样将首地址放入 rdi 传入 phase_3 函数，分析 phase-3

函数的反汇编代码：

```
0000000000401465 <phase_3>:
  401465:       55                      push   %rbp
  401466:       48 89 e5                mov    %rsp,%rbp
  401469:       48 83 ec 10             sub    $0x10,%rsp
  40146d:       48 8d 4d f8             lea    -0x8(%rbp),%rcx
  401471:       48 8d 55 fc             lea    -0x4(%rbp),%rdx
  401475:       be 37 33 40 00          mov    $0x403337,%esi
  40147a:       b8 00 00 00 00          mov    $0x0,%eax
  40147f:       e8 8c fc ff ff          callq  401110 <__isoc99_sscanf@plt>
  401484:       83 f8 01                cmp    $0x1,%eax
  401487:       7e 11                   jle    40149a <phase_3+0x35>
  401489:       8b 45 fc                mov    -0x4(%rbp),%eax
  40148c:       83 f8 07                cmp    $0x7,%eax
  40148f:       77 46                   ja     4014d7 <phase_3+0x72>
  401491:       89 c0                   mov    %eax,%eax
  401493:       ff 24 c5 80 31 40 00    jmpq   *0x403180(,%rax,8)
  40149a:       e8 69 04 00 00          callq  401908 <explode_bomb>
  40149f:       eb e8                   jmp    401489 <phase_3+0x24>
  4014a1:       b8 41 02 00 00          mov    $0x241,%eax
  4014a6:       39 45 f8                cmp    %eax,-0x8(%rbp)
  4014a9:       75 3f                   jne    4014ea <phase_3+0x85>
  4014ab:       c9                      leaveq
  4014ac:       c3                      retq
  4014ad:       b8 c8 00 00 00          mov    $0xc8,%eax
  4014b2:       eb f2                   jmp    4014a6 <phase_3+0x41>
  4014b4:       b8 2e 03 00 00          mov    $0x32e,%eax
  4014b9:       eb eb                   jmp    4014a6 <phase_3+0x41>
  4014bb:       b8 25 02 00 00          mov    $0x225,%eax
  4014c0:       eb e4                   jmp    4014a6 <phase_3+0x41>
  4014c2:       b8 8e 02 00 00          mov    $0x28e,%eax
  4014c7:       eb dd                   jmp    4014a6 <phase_3+0x41>
  4014c9:       b8 52 02 00 00          mov    $0x252,%eax
  4014ce:       eb d6                   jmp    4014a6 <phase_3+0x41>
  4014d0:       b8 e4 02 00 00          mov    $0x2e4,%eax
  4014d5:       eb cf                   jmp    4014a6 <phase_3+0x41>
  4014d7:       e8 2c 04 00 00          callq  401908 <explode_bomb>
  4014dc:       b8 00 00 00 00          mov    $0x0,%eax
  4014e1:       eb c3                   jmp    4014a6 <phase_3+0x41>
  4014e3:       b8 b7 00 00 00          mov    $0xb7,%eax
  4014e8:       eb bc                   jmp    4014a6 <phase_3+0x41>
  4014ea:       e8 19 04 00 00          callq  401908 <explode_bomb>
  4014ef:       eb ba                   jmp    4014ab <phase_3+0x46>
```

可以发现该函数一开始在运行时栈上分配了 16 字节的空间，并且将其中的 8 字节用于存放两个由我们的输入字符串拆分的数据，因此我推测我们应该输入两个整数。

在接下来的操作中，函数检查了 sscanf 函数的返回值，若小于等于 1，即输入数字只有一个以内则引爆炸弹。

接下来将第一个数据装入 rax 寄存器并检查其值，若大于 7 则引爆。

我们发现函数将 rax 中的第一个参数值作为偏移量，跳转向内存中某地址的空

间中存储的地址值，通过 x/20x 0x403180 的值我们可以查看到其具体跳转地址，在此之前我们需要设置断点将程序停在 phase_1 函数开头位置（具体操作如下图）

```
(gdb) x/72x 0x403180
0x403180:       0xa1    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0x403188:       0xe3    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0x403190:       0xad    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0x403198:       0xb4    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0x4031a0:       0xbb    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0x4031a8:       0xc2    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0x4031b0:       0xc9    0x14    0x40    0x00    0x00    0x00    0x00    0x00
0x4031b8:       0xd0    0x14    0x40    0x00    0x00    0x00    0x00    0x00
```

我们发现在这个 switch 语句中，由第一个参数的值（0~7）充当 case 判断，在每一个 case 中判断第二个参数的值是否与对应的值相等，如果相等则拆除炸弹，通过阅读反汇编代码，确定正确的输入组合如下：

0 577

1 183

2 200

3 814

4 549

5 654

6 594

7 740

将其分别代入 bomb 程序进行验证，发现其完全正确（截图仅展示第 1、2 组的情况）

```
0 577
Halfway there!
```

```
1 183
Halfway there!
```

3.4 阶段 4 的破解与分析

密码如下：

132 4

99 3

66 2

（其中的任意一组）

破解过程：

同样的，我们发现在 main 函数中将首地址放入 rdi 传到 phase_4，阅读 phase_4 汇编代码如图：

```
422 000000000040153c <phase_4>:
423   40153c:       55                      push   %rbp
424   40153d:       48 89 e5                mov    %rsp,%rbp
425   401540:       48 83 ec 10             sub    $0x10,%rsp
426   401544:       48 8d 4d fc             lea    -0x4(%rbp),%rcx
427   401548:       48 8d 55 f8             lea    -0x8(%rbp),%rdx
428   40154c:       be 37 33 40 00          mov    $0x403337,%esi
429   401551:       b8 00 00 00 00          mov    $0x0,%eax
430   401556:       e8 b5 fb ff ff          callq  401110 <__isoc99_sscanf@plt>
431   40155b:       83 f8 02                cmp    $0x2,%eax
432   40155e:       75 0d                   jne    40156d <phase_4+0x31>
433   401560:       8b 45 fc                mov    -0x4(%rbp),%eax
434   401563:       83 f8 01                cmp    $0x1,%eax
435   401566:       7e 05                   jle    40156d <phase_4+0x31>
436   401568:       83 f8 04                cmp    $0x4,%eax
437   40156b:       7e 05                   jle    401572 <phase_4+0x36>
438   40156d:       e8 96 03 00 00          callq  401908 <explode_bomb>
439   401572:       8b 75 fc                mov    -0x4(%rbp),%esi
440   401575:       bf 07 00 00 00          mov    $0x7,%edi
441   40157a:       e8 72 ff ff ff          callq  4014f1 <func4>
442   40157f:       39 45 f8                cmp    %eax,-0x8(%rbp)
443   401582:       75 02                   jne    401586 <phase_4+0x4a>
444   401584:       c9                      leaveq
445   401585:       c3                      retq
446   401586:       e8 7d 03 00 00          callq  401908 <explode_bomb>
447   40158b:       eb f7                   jmp    401584 <phase_4+0x48>
448
```

在该函数中，一开始现在运行时栈上分配了 16 字节空间，并用其中的 8 字节存放由输入字符串拆分得到的两个数据，因此推测输入字符串应该为两个整数，同时看到若不是两个整数则直接爆炸，同时我们发现若第二个必须大于 1 小于等于 4，即只能取 2,3,4 否则直接引爆炸弹。

接下来，该函数将 7 和第二个参数一起传送到了函数 func4 中，阅读 func4 汇编代码如下图：

```
391 00000000004014f1 <func4>:
392   4014f1:       85 ff                   test   %edi,%edi
393   4014f3:       7e 3d                   jle    401532 <func4+0x41>
394   4014f5:       55                      push   %rbp
395   4014f6:       48 89 e5                mov    %rsp,%rbp
396   4014f9:       41 55                   push   %r13
397   4014fb:       41 54                   push   %r12
398   4014fd:       53                      push   %rbx
399   4014fe:       48 83 ec 08             sub    $0x8,%rsp
400   401502:       41 89 fc                mov    %edi,%r12d
401   401505:       89 f3                   mov    %esi,%ebx
402   401507:       83 ff 01                cmp    $0x1,%edi
403   40150a:       74 2c                   je     401538 <func4+0x47>
404   40150c:       8d 7f ff                lea    -0x1(%rdi),%edi
405   40150f:       e8 dd ff ff ff          callq  4014f1 <func4>
406   401514:       44 8d 2c 18             lea    (%rax,%rbx,1),%r13d
407   401518:       41 8d 7c 24 fe          lea    -0x2(%r12),%edi
408   40151d:       89 de                   mov    %ebx,%esi
409   40151f:       e8 cd ff ff ff          callq  4014f1 <func4>
410   401524:       44 01 e8                add    %r13d,%eax
411   401527:       48 83 c4 08             add    $0x8,%rsp
412   40152b:       5b                      pop    %rbx
413   40152c:       41 5c                   pop    %r12
414   40152e:       41 5d                   pop    %r13
415   401530:       5d                      pop    %rbp
416   401531:       c3                      retq
417   401532:       b8 00 00 00 00          mov    $0x0,%eax
418   401537:       c3                      retq
419   401538:       89 f0                   mov    %esi,%eax
420   40153a:       eb eb                   jmp    401527 <func4+0x36>
```

为更方便理解函数，我们将函数返回值记为 f(n,x),不难发现该函数在递归的调用自身，且 f(0,x)=0,f(1,x)=x，而当 n 的值大于 1 时，我们发现函数两次调用自身，得到 f(n,x)=f(n-1,x)+f(n-2,x)+x，对应到 phase_4 函数中的具体调用时，n 为传入的 7，x 为我们输入的第二个参数，进过简单计算可得 f(7,x)=33x，因此我们知道在 phase_4 中调用 func4 后返回了第二个参数乘以 33 的乘积，继续阅读 phase_4 发现，函数将返回值和第一个输入数据进行比较，相等则拆除炸弹，考虑到之前所述的参数 2 的取值 2,3,4，我们可以得到答案的组合为：

66,2

99,3

132,4

分别在 bomb 程序中验证得到：

```
Halfway there!
66 2
So you got that one.  Try this one.
```

```
Halfway there!
99 3
So you got that one.  Try this one.
```

```
Halfway there!
132 4
So you got that one.  Try this one.
```

因此我们得出的答案正确

3.5 阶段 5 的破解与分析

密码如下：

5 115

21 115

37 115

……

（第一个数据可以是任何模 16 为 5 的数）

破解过程：

同样的，我们发现在 main 函数中将首地址放入 rdi 传到 phase_5，阅读 phase_5 汇编代码如图：

```
449 000000000040158d <phase_5>:
450   40158d:       55                      push   %rbp
451   40158e:       48 89 e5                mov    %rsp,%rbp
452   401591:       48 83 ec 10             sub    $0x10,%rsp
453   401595:       48 8d 4d f8             lea    -0x8(%rbp),%rcx
454   401599:       48 8d 55 fc             lea    -0x4(%rbp),%rdx
455   40159d:       be 37 33 40 00          mov    $0x403337,%esi
456   4015a2:       b8 00 00 00 00          mov    $0x0,%eax
457   4015a7:       e8 64 fb ff ff          callq  401110 <__isoc99_sscanf@plt
458   4015ac:       83 f8 01                cmp    $0x1,%eax
459   4015af:       7e 2e                   jle    4015df <phase_5+0x52>
460   4015b1:       8b 45 fc                mov    -0x4(%rbp),%eax
461   4015b4:       83 e0 0f                and    $0xf,%eax
462   4015b7:       89 45 fc                mov    %eax,-0x4(%rbp)
463   4015ba:       b9 00 00 00 00          mov    $0x0,%ecx
464   4015bf:       ba 00 00 00 00          mov    $0x0,%edx
465   4015c4:       8b 45 fc                mov    -0x4(%rbp),%eax
466   4015c7:       83 f8 0f                cmp    $0xf,%eax
467   4015ca:       74 1a                   je     4015e6 <phase_5+0x59>
468   4015cc:       83 c2 01                add    $0x1,%edx
469   4015cf:       48 98                   cltq
470   4015d1:       8b 04 85 c0 31 40 00    mov    0x4031c0(,%rax,4),%eax
471   4015d8:       89 45 fc                mov    %eax,-0x4(%rbp)
472   4015db:       01 c1                   add    %eax,%ecx
473   4015dd:       eb e5                   jmp    4015c4 <phase_5+0x37>
474   4015df:       e8 24 03 00 00          callq  401908 <explode_bomb>
475   4015e4:       eb cb                   jmp    4015b1 <phase_5+0x24>
476   4015e6:       83 fa 0f                cmp    $0xf,%edx
477   4015e9:       75 05                   jne    4015f0 <phase_5+0x63>
478   4015eb:       39 4d f8                cmp    %ecx,-0x8(%rbp)
479   4015ee:       74 05                   je     4015f5 <phase_5+0x68>
480   4015f0:       e8 13 03 00 00          callq  401908 <explode_bomb>
481   4015f5:       c9                      leaveq
482   4015f6:       c3                      retq
483
```

我们发现，函数在运行时栈上分配了 16 字节的空间，然后使用了两个连续 4 字节空间拆分和存储我们读入的字符串，因此我们可以推断该阶段需要我们输入两个整数。之后检测返回值，若只收到 1 个或者 0 个参数则引爆炸弹，这也证明了这一点。

之后我们发现，函数将输入数据 1 放入了 rax 并将其对 0xf 做与运算，即函数将输入数据 1（下记为 x1，输入数据 2 记为 x2）取 16 的模。接下来函数进行了跳转，并且将一个值为 0 的计数器与 15 进行比较，猜测接下来将要进行一个 15 步的循环体，我们发现在每一次循环中，rax 的值都被更新为地址 0x4031c0 加上 rax 值作为偏移量的地址至指向的内存值，我们通过 x/50x 0x4031c0 查看开始余 0x4031c0 的内存空间存储的值，如下图所示：

```
(gdb) x/20x 0x4031c0
0x4031c0 <array.3401>:  0x0000000a      0x00000002      0x0000000e      0x00000007
0x4031d0 <array.3401+16>:       0x00000008      0x0000000c      0x0000000f      0x0000000b
0x4031e0 <array.3401+32>:       0x00000000      0x00000004      0x00000001      0x0000000d
0x4031f0 <array.3401+48>:       0x00000003      0x00000009      0x00000006      0x00000005
0x403200:       0x21776f57      0x756f5920      0x20657627      0x75666564
(gdb)
```

我们发现，此处存储了一个特殊的数组，不妨记为 A，该数组长度为 16，rax 中的值记录了数组的偏移量 i，而数组的每一个值的取值范围都在 0~15，因此我们可以将这个数组看成一个循环链表，数组每个元素的值对应了下一个链表位置的索引，即 A[0]→A[10]→A[1]→A[2]→A[14]→A[6]→A[15]→A[5]→A[12]→A[3]→A[7]→A[11]→A[13]→A[9]→A[4]→A[8]→A[0]

我们进一步分析循环体内部的行为，我们发现循环体记录了数组按照链表顺序访问每一个节点值之和，并且在遇到 A[15]时停下，并且要求此时正好经过了 15 个数组元素（计数寄存器 edx 值为 0xf）， 否则引爆炸弹。此时将求得的和与我们输入的第二个数据进行比较，若相等则解除炸弹。

因此，根据我们得出的循环链表，我们得知输入的答案应该为 5 115。

运行 bomb 程序，发现正确。

```
So you got that one.  Try this one.
5 115
Good work!  On to the next...
```

同时我们注意到，在处理第一个参数时，我们对其进行了 mod16 的操作，因此理论上任何模 16 取 5 的整数都满足要求，我们将 21（21mod16=5）进行了测试，发现正确

```
21 115
Good work!  On to the next...
```

因此，我们得出的答案完全正确。

3.6 阶段 6 的破解与分析

密码如下：

6 1 4 3 5 2

破解过程：

同样的，我们发现在 main 函数中将首地址放入 rdi 传到 phase_6，阅读 phase_6 汇编代码的前半部分如图：

```
484 00000000004015f7 <phase_6>:
485   4015f7:       55                      push   %rbp
486   4015f8:       48 89 e5                mov    %rsp,%rbp
487   4015fb:       41 55                   push   %r13
488   4015fd:       41 54                   push   %r12
489   4015ff:       53                      push   %rbx
490   401600:       48 83 ec 58             sub    $0x58,%rsp
491   401604:       48 8d 75 c0             lea    -0x40(%rbp),%rsi
492   401608:       e8 1d 03 00 00          callq  40192a <read_six_numbers>
493   40160d:       41 bc 00 00 00 00       mov    $0x0,%r12d
494   401613:       eb 29                   jmp    40163e <phase_6+0x47>
495   401615:       e8 ee 02 00 00          callq  401908 <explode_bomb>
496   40161a:       eb 37                   jmp    401653 <phase_6+0x5c>
497   40161c:       83 c3 01                add    $0x1,%ebx
498   40161f:       83 fb 05                cmp    $0x5,%ebx
499   401622:       7f 17                   jg     40163b <phase_6+0x44>
500   401624:       49 63 c4                movslq %r12d,%rax
501   401627:       48 63 d3                movslq %ebx,%rdx
502   40162a:       8b 7c 95 c0             mov    -0x40(%rbp,%rdx,4),%edi
503   40162e:       39 7c 85 c0             cmp    %edi,-0x40(%rbp,%rax,4)
504   401632:       75 e8                   jne    40161c <phase_6+0x25>
505   401634:       e8 cf 02 00 00          callq  401908 <explode_bomb>
506   401639:       eb e1                   jmp    40161c <phase_6+0x25>
507   40163b:       45 89 ec                mov    %r13d,%r12d
508   40163e:       41 83 fc 05             cmp    $0x5,%r12d
509   401642:       7f 19                   jg     40165d <phase_6+0x66>
510   401644:       49 63 c4                movslq %r12d,%rax
511   401647:       8b 44 85 c0             mov    -0x40(%rbp,%rax,4),%eax
512   40164b:       83 e8 01                sub    $0x1,%eax
513   40164e:       83 f8 05                cmp    $0x5,%eax
514   401651:       77 c2                   ja     401615 <phase_6+0x1e>
515   401653:       45 8d 6c 24 01          lea    0x1(%r12),%r13d
516   401658:       44 89 eb                mov    %r13d,%ebx
517   40165b:       eb c2                   jmp    40161f <phase_6+0x28>
518   40165d:       be 00 00 00 00          mov    $0x0,%esi
519   401662:       eb 08                   jmp    40166c <phase_6+0x75>
520   401664:       48 89 54 cd 90          mov    %rdx,-0x70(%rbp,%rcx,8)
```

我们发现该函数在运行时栈上分配了 88 个字节的空间，并且使用了其中的 24 个字节的空间存放有输入字符串拆分得到的 6 个整数数据，因此可以推断我们需要输入六个整数数据（为了下文叙述方便，我们根据这些数据在内存上的地址从小到大依次称为 x1、x2、x3、x4、x5、x6）

接下来的汇编代码执行了双重循环操作，外层循环遍历了 xi（1≤i≤6）,外层循环检测 xi 的值必须≤6，否则引爆炸弹；内层循环遍历了 xj（i<j≤6），要求 xi≠xj，否则引爆炸弹。换言之，我们输入的 x1~x6 必须为互不相同的整数，且全部小于等于 6.

接下来我们阅读 phase_6 函数的下半部分，如下图：

```
519   401662:       eb 08                   jmp    40166c <phase_6+0x75>
520   401664:       48 89 54 cd 90          mov    %rdx,-0x70(%rbp,%rcx,8)
521   401669:       83 c6 01                add    $0x1,%esi
522   40166c:       83 fe 05                cmp    $0x5,%esi
523   40166f:       7f 1c                   jg     40168d <phase_6+0x96>
524   401671:       b8 01 00 00 00          mov    $0x1,%eax
525   401676:       ba d0 52 40 00          mov    $0x4052d0,%edx
526   40167b:       48 63 ce                movslq %esi,%rcx
527   40167e:       39 44 8d c0             cmp    %eax,-0x40(%rbp,%rcx,4)
528   401682:       7e e0                   jle    401664 <phase_6+0x6d>
529   401684:       48 8b 52 08             mov    0x8(%rdx),%rdx
530   401688:       83 c0 01                add    $0x1,%eax
531   40168b:       eb ee                   jmp    40167b <phase_6+0x84>
532   40168d:       48 8b 5d 90             mov    -0x70(%rbp),%rbx
533   401691:       48 89 d9                mov    %rbx,%rcx
534   401694:       b8 01 00 00 00          mov    $0x1,%eax
535   401699:       eb 12                   jmp    4016ad <phase_6+0xb6>
536   40169b:       48 63 d0                movslq %eax,%rdx
537   40169e:       48 8b 54 d5 90          mov    -0x70(%rbp,%rdx,8),%rdx
538   4016a3:       48 89 51 08             mov    %rdx,0x8(%rcx)
539   4016a7:       83 c0 01                add    $0x1,%eax
540   4016aa:       48 89 d1                mov    %rdx,%rcx
541   4016ad:       83 f8 05                cmp    $0x5,%eax
542   4016b0:       7e e9                   jle    40169b <phase_6+0xa4>
543   4016b2:       48 c7 41 08 00 00 00    movq   $0x0,0x8(%rcx)
544   4016b9:       00
545   4016ba:       41 bc 00 00 00 00       mov    $0x0,%r12d
546   4016c0:       eb 08                   jmp    4016ca <phase_6+0xd3>
547   4016c2:       48 8b 5b 08             mov    0x8(%rbx),%rbx
548   4016c6:       41 83 c4 01             add    $0x1,%r12d
549   4016ca:       41 83 fc 04             cmp    $0x4,%r12d
550   4016ce:       7f 11                   jg     4016e1 <phase_6+0xea>
551   4016d0:       48 8b 43 08             mov    0x8(%rbx),%rax
552   4016d4:       8b 00                   mov    (%rax),%eax
553   4016d6:       39 03                   cmp    %eax,(%rbx)
554   4016d8:       7e e8                   jle    4016c2 <phase_6+0xcb>
555   4016da:       e8 29 02 00 00          callq  401908 <explode_bomb>
556   4016df:       eb e1                   jmp    4016c2 <phase_6+0xcb>
557   4016e1:       48 83 c4 58             add    $0x58,%rsp
558   4016e5:       5b                      pop    %rbx
559   4016e6:       41 5c                   pop    %r12
560   4016e8:       41 5d                   pop    %r13
561   4016ea:       5d                      pop    %rbp
562   4016eb:       c3                      retq
```

我们发现在下一个循环体中函数访问了地址 0x4052d0 之后的内容，因此我们可以查看该地址之后的内容，如下图所示：

```
(gdb) x/50x 0x4052d0
0x4052d0 <node1>:       0x00000102      0x00000001      0x004052e0      0x00000000
0x4052e0 <node2>:       0x000003b0      0x00000002      0x004052f0      0x00000000
0x4052f0 <node3>:       0x0000013f      0x00000003      0x00405300      0x00000000
0x405300 <node4>:       0x0000013a      0x00000004      0x00405310      0x00000000
0x405310 <node5>:       0x00000383      0x00000005      0x00405320      0x00000000
0x405320 <node6>:       0x000000d9      0x00000006      0x00000000      0x00000000
0x405330 <bomb_id>:     0x0000014c      0x00000000      0x00000000      0x00000000
```

不难看出此处存放了一个 6 个节点的链表，每一个链表节点大小为 16 字节，其中前 4 字节存放了一个值，之后四字节存放了节点编号，之后四字节存放

了指向下一个节点元素的指针，最后空出四字节用于对齐。

我们继续阅读 phase_6 函数的反汇编代码，我们发现在接下来的循环体中函数遍历了 x1~x6 的值，每次遍历确定了一个编号与 xi 相等的节点（到这时我们可以确定需要输入的 x1~x6 为互不相等的 1~6 中的数值），并把该链表节点的地址值放入之前开辟的运行时栈的空间上。

在接下来的循环体中，我们按照之前将地址存放进运行时栈的顺序读取节点的地址并访问该节点的数值，若存在某个节点的数值比之前节点的数值小则引爆炸弹，换言之，我们通过 phase_6 函数实现了对链表节点的重新排序，而排序依据就是我们输入的数据序列，当且仅当数据序列是按照节点值的升序排列时才可以拆除炸弹。

因此，根据我们得到的链表节点信息可以知道正确答案为 6 1 4 3 5 2，运行 bomb 文件有：

```
Good work!  On to the next...
6 1 4 3 5 2
Congratulations! You've defused the bomb!
```

因此，答案完全正确。

3.7 阶段 7 的破解与分析(隐藏阶段)

密码如下：

22

破解过程：

我们通过在汇编代码中搜索 secret_phase 字段，发现是在 phase_defused 函数中末尾被调用，如下图所示：

```
0000000000401a91 <phase_defused>:
  401a91:       83 3d d4 3c 00 00 06    cmpl   $0x6,0x3cd4(%rip)        # 40576c <num_input_strings>
  401a98:       74 01                   je     401a9b <phase_defused+0xa>
  401a9a:       c3                      retq
  401a9b:       55                      push   %rbp
  401a9c:       48 89 e5                mov    %rsp,%rbp
  401a9f:       48 83 ec 60             sub    $0x60,%rsp
  401aa3:       4c 8d 45 b0             lea    -0x50(%rbp),%r8
  401aa7:       48 8d 4d a8             lea    -0x58(%rbp),%rcx
  401aab:       48 8d 55 ac             lea    -0x54(%rbp),%rdx
  401aaf:       be 81 33 40 00          mov    $0x403381,%esi
  401ab4:       bf 70 58 40 00          mov    $0x405870,%edi
  401ab9:       b8 00 00 00 00          mov    $0x0,%eax
  401abe:       e8 4d f6 ff ff          callq  401110 <__isoc99_sscanf@plt>
  401ac3:       83 f8 03                cmp    $0x3,%eax
  401ac6:       74 0c                   je     401ad4 <phase_defused+0x43>
  401ac8:       bf c0 32 40 00          mov    $0x4032c0,%edi
  401acd:       e8 8e f5 ff ff          callq  401060 <puts@plt>
  401ad2:       c9                      leaveq
  401ad3:       c3                      retq
  401ad4:       be 8a 33 40 00          mov    $0x40338a,%esi
  401ad9:       48 8d 7d b0             lea    -0x50(%rbp),%rdi
  401add:       e8 2a fd ff ff          callq  40180c <strings_not_equal>
  401ae2:       85 c0                   test   %eax,%eax
  401ae4:       75 e2                   jne    401ac8 <phase_defused+0x37>
  401ae6:       bf 60 32 40 00          mov    $0x403260,%edi
  401aeb:       e8 70 f5 ff ff          callq  401060 <puts@plt>
  401af0:       bf 88 32 40 00          mov    $0x403288,%edi
  401af5:       e8 66 f5 ff ff          callq  401060 <puts@plt>
  401afa:       b8 00 00 00 00          mov    $0x0,%eax
  401aff:       e8 22 fc ff ff          callq  401726 <secret_phase>
  401b04:       eb c2                   jmp    401ac8 <phase_defused+0x37>
```

我们发现这个函数最开始将 6 与内存 0x40576c 中的值作比较，我们查看这个内存地址的值发现：

```
0x40576c <num_input_strings>:   0x00000006
```

其标记了关卡数目，只有到第六关才会触发隐藏关卡。

我们分别打印在其中提到的两个地址值，有：

```
(gdb) x/s 0x403381
0x403381:       "%d %d %s"
(gdb) x/s 0x405870
0x405870 <input_strings+240>:   "99 3"
(gdb)
```

一个是输入格式，一个是第四阶段函数 phase_4 的输入串，再结合对于返回值的检测，是与 3 进行比对，可以知道跳转到 phase_4 的方法是在输入“99 3”之后继续输入一个字符串。

打印之后进行字符串比对的地址中的内容，有

```
(gdb) x/s 0x40338a
0x40338a:       "DrEvil"
(gdb)
```

因此我们确定，如果要触发隐藏关卡，需要在“99 3”之后输入“DrEvil”即

在第四阶段输入“99 3 DrEvil”即可，这样的话，运行 bomb 程序，在第六阶段结束之后，并未退出程序，而是显示如下图：

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

因此我们的隐藏关卡被正确触发。

接下来阅读 secret_phase 函数的反汇编代码部分如下图所示：

```
587 0000000000401726 <secret_phase>:
588   401726:       55                      push   %rbp
589   401727:       48 89 e5                mov    %rsp,%rbp
590   40172a:       53                      push   %rbx
591   40172b:       48 83 ec 08             sub    $0x8,%rsp
592   40172f:       e8 32 02 00 00          callq  401966 <read_line>
593   401734:       48 89 c7                mov    %rax,%rdi
594   401737:       e8 04 fa ff ff          callq  401140 <atoi@plt>
595   40173c:       89 c3                   mov    %eax,%ebx
596   40173e:       8d 40 ff                lea    -0x1(%rax),%eax
597   401741:       3d e8 03 00 00          cmp    $0x3e8,%eax
598   401746:       77 27                   ja     40176f <secret_phase+0x49>
599   401748:       89 de                   mov    %ebx,%esi
600   40174a:       bf f0 50 40 00          mov    $0x4050f0,%edi
601   40174f:       e8 98 ff ff ff          callq  4016ec <fun7>
602   401754:       83 f8 02                cmp    $0x2,%eax
603   401757:       75 1d                   jne    401776 <secret_phase+0x50>
604   401759:       bf 00 32 40 00          mov    $0x403200,%edi
605   40175e:       e8 fd f8 ff ff          callq  401060 <puts@plt>
606   401763:       e8 29 03 00 00          callq  401a91 <phase_defused>
607   401768:       48 83 c4 08             add    $0x8,%rsp
608   40176c:       5b                      pop    %rbx
609   40176d:       5d                      pop    %rbp
610   40176e:       c3                      retq
611   40176f:       e8 94 01 00 00          callq  401908 <explode_bomb>
612   401774:       eb d2                   jmp    401748 <secret_phase+0x22>
613   401776:       e8 8d 01 00 00          callq  401908 <explode_bomb>
614   40177b:       eb dc                   jmp    401759 <secret_phase+0x33>
```

我们发现 secret_phase 函数将一个输入值和一个地址传入了 fun7 函数，先打印传入的地址的内容可以看到：

```
(gdb) x/500x 0x4050f0
0x4050f0 <n1>:  0x24    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4050f8 <n1+8>:        0x10    0x51    0x40    0x00    0x00    0x00    0x00    0x00
0x405100 <n1+16>:       0x30    0x51    0x40    0x00    0x00    0x00    0x00    0x00
0x405108:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405110 <n21>: 0x08    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405118 <n21+8>:       0x90    0x51    0x40    0x00    0x00    0x00    0x00    0x00
0x405120 <n21+16>:      0x50    0x51    0x40    0x00    0x00    0x00    0x00    0x00
0x405128:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405130 <n22>: 0x32    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405138 <n22+8>:       0x70    0x51    0x40    0x00    0x00    0x00    0x00    0x00
0x405140 <n22+16>:      0xb0    0x51    0x40    0x00    0x00    0x00    0x00    0x00
0x405148:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405150 <n32>: 0x16    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405158 <n32+8>:       0x70    0x52    0x40    0x00    0x00    0x00    0x00    0x00
0x405160 <n32+16>:      0x30    0x52    0x40    0x00    0x00    0x00    0x00    0x00
0x405168:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405170 <n33>: 0x2d    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405178 <n33+8>:       0xd0    0x51    0x40    0x00    0x00    0x00    0x00    0x00
0x405180 <n33+16>:      0x90    0x52    0x40    0x00    0x00    0x00    0x00    0x00
0x405188:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405190 <n31>: 0x06    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405198 <n31+8>:       0xf0    0x51    0x40    0x00    0x00    0x00    0x00    0x00
0x4051a0 <n31+16>:      0x50    0x52    0x40    0x00    0x00    0x00    0x00    0x00
0x4051a8:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4051b0 <n34>: 0x6b    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4051b8 <n34+8>:       0x10    0x52    0x40    0x00    0x00    0x00    0x00    0x00
0x4051c0 <n34+16>:      0xb0    0x52    0x40    0x00    0x00    0x00    0x00    0x00
0x4051c8:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4051d0 <n45>: 0x28    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4051d8 <n45+8>:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4051e0 <n45+16>:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4051e8:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4051f0 <n41>: 0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x4051f8 <n41+8>:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405200 <n41+16>:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405208:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405210 <n47>: 0x63    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405218 <n47+8>:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405220 <n47+16>:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405228:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405230 <n44>: 0x23    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405238 <n44+8>:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405240 <n44+16>:      0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405248:       0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x405250 <n42>: 0x07    0x00    0x00    0x00    0x00    0x00    0x00    0x00
```

我们很容易发现这是一个二叉树的地址，并且该二叉树为二叉查找树，即根节点左边节点值都小于根节点值，根节点右边节点值都大于根节点值。因此 fun7 函数的数据结构为二叉树，阅读 fun7 汇编代码如下图：

```
564 00000000004016ec <fun7>:
565   4016ec:       48 85 ff                test   %rdi,%rdi
566   4016ef:       74 2f                   je     401720 <fun7+0x34>
567   4016f1:       55                      push   %rbp
568   4016f2:       48 89 e5                mov    %rsp,%rbp
569   4016f5:       8b 07                   mov    (%rdi),%eax
570   4016f7:       39 f0                   cmp    %esi,%eax
571   4016f9:       7f 09                   jg     401704 <fun7+0x18>
572   4016fb:       75 14                   jne    401711 <fun7+0x25>
573   4016fd:       b8 00 00 00 00          mov    $0x0,%eax
574   401702:       5d                      pop    %rbp
575   401703:       c3                      retq
576   401704:       48 8b 7f 08             mov    0x8(%rdi),%rdi
577   401708:       e8 df ff ff ff          callq  4016ec <fun7>
578   40170d:       01 c0                   add    %eax,%eax
579   40170f:       eb f1                   jmp    401702 <fun7+0x16>
580   401711:       48 8b 7f 10             mov    0x10(%rdi),%rdi
581   401715:       e8 d2 ff ff ff          callq  4016ec <fun7>
582   40171a:       8d 44 00 01             lea    0x1(%rax,%rax,1),%eax
583   40171e:       eb e2                   jmp    401702 <fun7+0x16>
584   401720:       b8 ff ff ff ff          mov    $0xffffffff,%eax
585   401725:       c3                      retq
586
```

我们发现 fun7 是一个递归调用自身的函数体，当查找值为根节点时返回 0，在左边时返回 2*fun7（root→left），查找值在根节点右边时返回 2*fun7（root→right）+1。

至此，结合二叉树具体结构（在此不给出），可以知道输入 22 时 fun7 返回 2，当返回 2 时 secret_phase 函数正确拆除炸弹，运行 bomb 程序发现我们的答案完全正确，如下图所示：

```
Curses, you've found the secret phase!
But finding it and solving it are quite different...
22
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

我们将所有的答案写入 ans.txt，并运行指令。./bomb ans.txt 可以得到：

```
linsanity@linsanity-VirtualBox:~/Codes/Lab3$ ./bomb ans.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
Good work!  On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

至此，炸弹拆除完毕。

第 4 章 总结

4.1 请总结本次实验的收获

更加深刻地体会了程序在执行时是如何传递参数与控制转移，深入理解了一些常见数据结构如何传递。
理解了汇编代码的基本操作，了解了常见的调试程序技巧。

4.2 请给出对本次实验内容的建议

建议针对 EDB 的安装做更加详细的说明

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.