

北京邮电大学

实验报告



题目： 拆解二进制炸弹（高阶）

班 级： 2022211320

学 号： 2022211683

姓 名： 张晨阳

学 院： 计算机学院（国家示范性软件学院）

2023 年 11 月 7 日

一、实验目的

1. 理解 C 语言程序的机器级表示。
2. 初步掌握 GDB 调试器的用法。
3. 阅读 C 编译器生成的 x 86-64 机器代码，理解不同控制结构生成的基本指令模式，过程的实现。

二、实验环境

1. Windows PowerShell (10.120.11.12)
2. Linux
3. Objdump 命令反汇编
4. GDB 调试工具
5. Visual Studio Code 1.83.1

三、实验内容

登录 bupt 1 服务器，在 home 目录下可以找到 Evil 博士专门为你量身定制的一个 bomb，当运行时，它会要求你输入一个字符串，如果正确，则进入下一关，继续要求你输入下一个字符串；否则，炸弹就会爆炸，输出一行提示信息并向计分服务器提交扣分信息。因此，本实验要求你必须通过反汇编和逆向工程对 bomb 执行文件进行分析，找到正确的字符串来解除这个的炸弹。

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

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

四、实验步骤及实验分析

第一阶段

- 有了初级 bomb 的经验和教训，我很快进入了阶段一的分析；
- 首先设置阶段一的断点以及爆炸函数的断点，然后进入 `phase_1`，显示汇编代码；

```
(gdb) b phase_1
Breakpoint 1 at 0x401038
(gdb) b explode_bomb
Breakpoint 2 at 0x401928
(gdb) r
Starting program: /students/2022211683/bomb136/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
12345

Breakpoint 1, 0x00000000401038 in phase_1 ()
(gdb) disas
Dump of assembler code for function phase_1:
0x00000000401028 <+0>: stp    x29, x30, [sp, #-16]!
0x0000000040102c <+4>: mov    x29, sp
0x00000000401030 <+8>: adrp   x1, 0x402000 <submitr+900>
0x00000000401034 <+12>: add    x1, x1, #0x6e8
=> 0x00000000401038 <+16>: bl     0x401614 <strings_not_equal>
0x0000000040103c <+20>: cbnz   w0, 0x401048 <phase_1+32>
0x00000000401040 <+24>: ldp    x29, x30, [sp], #16
0x00000000401044 <+28>: ret
0x00000000401048 <+32>: bl     0x401918 <explode_bomb>
0x0000000040104c <+36>: b      0x401040 <phase_1+24>
End of assembler dump.
```

图1-阶段一开始

- 本阶段的关键之处就在光标所停 `strings_not_equal` 函数处，分析可得这个函数是用来比较输入字符与正确字符是否相等的，进入该函数并显示内部汇编代码：

```
(gdb) si
0x00000000401614 in strings_not_equal ()
(gdb) disas
Dump of assembler code for function strings_not_equal:
=> 0x00000000401614 <+0>: stp    x29, x30, [sp, #-48]!
0x00000000401618 <+4>: mov    x29, sp
0x0000000040161c <+8>: stp    x19, x20, [sp, #16]
0x00000000401620 <+12>: str    x21, [sp, #32]
0x00000000401624 <+16>: mov    x20, x0
0x00000000401628 <+20>: mov    x19, x1
0x0000000040162c <+24>: bl     0x4015e8 <string_length>
0x00000000401630 <+28>: mov    w21, w0
0x00000000401634 <+32>: mov    x0, x19
0x00000000401638 <+36>: bl     0x4015e8 <string_length>
0x0000000040163c <+40>: cmp    w21, w0
0x00000000401640 <+44>: b.eq   0x401658 <strings_not_equal+68> // b.none
0x00000000401644 <+48>: mov    w0, #0x1 // #1
0x00000000401648 <+52>: ldp    x19, x20, [sp, #16]
0x0000000040164c <+56>: ldr    x21, [sp, #32]
0x00000000401650 <+60>: ldp    x29, x30, [sp], #48
0x00000000401654 <+64>: ret
0x00000000401658 <+68>: ldrb   w0, [x20]
0x0000000040165c <+72>: cbz    w0, 0x40169c <strings_not_equal+136>
0x00000000401660 <+76>: ldrb   w1, [x19]
0x00000000401664 <+80>: cmp    w1, w0
0x00000000401668 <+84>: b.ne   0x4016a4 <strings_not_equal+144> // b.any
0x0000000040166c <+88>: mov    x0, #0x1 // #1
0x00000000401670 <+92>: ldrb   w1, [x20, x0]
0x00000000401674 <+96>: cbz    w1, 0x401694 <strings_not_equal+128>
0x00000000401678 <+100>: add    x0, x0, #0x1
0x0000000040167c <+104>: add    x2, x19, x0
0x00000000401680 <+108>: ldurb  w2, [x2, #-1]
--Type <RET> for more, q to quit, c to continue without paging--c
0x00000000401684 <+112>: cmp    w2, w1
0x00000000401688 <+116>: b.eq   0x401670 <strings_not_equal+92> // b.none
0x0000000040168c <+120>: mov    w0, #0x1 // #1
0x00000000401690 <+124>: b      0x401648 <strings_not_equal+52>
0x00000000401694 <+128>: mov    w0, #0x0 // #0
0x00000000401698 <+132>: b      0x401648 <strings_not_equal+52>
0x0000000040169c <+136>: mov    w0, #0x0 // #0
0x000000004016a0 <+140>: b      0x401648 <strings_not_equal+52>
0x000000004016a4 <+144>: mov    w0, #0x1 // #1
0x000000004016a8 <+148>: b      0x401648 <strings_not_equal+52>
End of assembler dump.
```

图2-strings_not_equal函数

- 与初级 bomb 类似，`string_length` 函数应该是获取输入字符数量的函数；
- 所以 `strings_not_equal` 函数也是先比较字符个数，再比较内容；
- 观察接下来的指令，是在比较寄存器 `x2` 和 `x19` 的内容，查看两个寄存器的值；

```
(gdb) x /s $x2
0x420728 <input_strings>: "12345"
(gdb) x /s $x19
0x4024d8 <__libc_csu_init>: "\375{\274\251\375\003"
```

图3-寄存器初步分析

- `x2` 确实是输入的字符串，继续分析发现函数前部分有一个 `mov x19, x1` 指令，将寄存器 `x1` 的内容存到了寄存器 `x19` 中，但还未执行到该语句，故正确字符应查看寄存器 `x1` 的值；

```
(gdb) x /s $x1
0x4026e8: "There are many handsome guys and beautiful girls on ShaHe campus."
```

图4-查看正确寄存器

- 故第一阶段正确答案为 **There are many handsome guys and beautiful girls on ShaHe campus.**
- 输入答案，通过，进入第二阶段。

第二阶段

- 同理设置断点，输入阶段一答案以及测试密码，进入阶段二；
- 显示汇编代码，开始阶段二的分析；

```
(gdb) b phase_2
Breakpoint 1 at 0x401060
(gdb) b explode_bomb
Breakpoint 2 at 0x401928
(gdb) r ans.txt
Starting program: /students/2022211683/bomb136/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?
123456

Breakpoint 1, 0x00000000401060 in phase_2 ()
(gdb) disas
Dump of assembler code for function phase_2:
0x00000000401050 <+0>: stp x29, x30, [sp, #-64]!
0x00000000401054 <+4>: mov x29, sp
0x00000000401058 <+8>: stp x19, x20, [sp, #16]
0x0000000040105c <+12>: add x1, x29, #0x28
=> 0x00000000401060 <+16>: bl 0x401954 <read_six_numbers>
0x00000000401064 <+20>: ldr w0, [x29, #40]
0x00000000401068 <+24>: cbnz w0, 0x401078 <phase_2+40>
0x0000000040106c <+28>: ldr w0, [x29, #44]
0x00000000401070 <+32>: cmp w0, #0x1
0x00000000401074 <+36>: b.eq 0x40107c <phase_2+44> // b.none
0x00000000401078 <+40>: bl 0x401918 <explode_bomb>
0x0000000040107c <+44>: add x19, x29, #0x28
0x00000000401080 <+48>: add x20, x19, #0x10
0x00000000401084 <+52>: b 0x401094 <phase_2+68>
0x00000000401088 <+56>: add x19, x19, #0x4
0x0000000040108c <+60>: cmp x19, x20
0x00000000401090 <+64>: b.eq 0x4010b4 <phase_2+100> // b.none
0x00000000401094 <+68>: ldr w0, [x19]
0x00000000401098 <+72>: ldr w1, [x19, #4]
0x0000000040109c <+76>: add w0, w0, w1
0x000000004010a0 <+80>: ldr w1, [x19, #8]
0x000000004010a4 <+84>: cmp w1, w0
0x000000004010a8 <+88>: b.eq 0x401088 <phase_2+56> // b.none
0x000000004010ac <+92>: bl 0x401918 <explode_bomb>
0x000000004010b0 <+96>: b 0x401088 <phase_2+56>
0x000000004010b4 <+100>: ldp x19, x20, [sp, #16]
0x000000004010b8 <+104>: ldp x29, x30, [sp, #64]
0x000000004010bc <+108>: ret
```

图5-阶段二开始

- 依然关注 **read_six_numbers** 函数，其表明此阶段需要输入6个数字；
- 设置断点并进入该函数查看指令；

```
(gdb) c
Continuing.

Breakpoint 3, 0x0000000040197c in read_six_numbers ()
(gdb) disas
Dump of assembler code for function read_six_numbers:
0x00000000401954 <+0>: stp x29, x30, [sp, #-16]!
0x00000000401958 <+4>: mov x29, sp
0x0000000040195c <+8>: add x7, x1, #0x14
0x00000000401960 <+12>: add x6, x1, #0x10
0x00000000401964 <+16>: add x5, x1, #0xc
0x00000000401968 <+20>: add x4, x1, #0x8
0x0000000040196c <+24>: add x3, x1, #0x4
0x00000000401970 <+28>: mov x2, x1
0x00000000401974 <+32>: adrp x1, 0x402000 <submitr+900>
0x00000000401978 <+36>: add x1, x1, #0x908
=> 0x0000000040197c <+40>: bl 0x400d70 <__isoc99_sscanf@plt>
0x00000000401980 <+44>: cmp w0, #0x5
0x00000000401984 <+48>: b.le 0x401990 <read_six_numbers+60>
0x00000000401988 <+52>: ldp x29, x30, [sp, #16]
0x0000000040198c <+56>: ret
0x00000000401990 <+60>: bl 0x401918 <explode_bomb>
End of assembler dump.
```

图6-read_six_numbers函数

- 观察一连串对 **x1** 操作的指令，分析是与输入的六个数有关的寄存器，查看其内容；

```

Dump of assembler code for function read_six_numbers:
0x0000000000401954 <+0>:      stp     x29, x30, [sp, #-16]!
0x0000000000401958 <+4>:      mov     x29, sp
0x000000000040195c <+8>:      add     x7, x1, #0x14
0x0000000000401960 <+12>:     add     x6, x1, #0x10
0x0000000000401964 <+16>:     add     x5, x1, #0xc
0x0000000000401968 <+20>:    add     x4, x1, #0x8
0x000000000040196c <+24>:    add     x3, x1, #0x4
0x0000000000401970 <+28>:    mov     x2, x1
0x0000000000401974 <+32>:    adrp    x1, 0x402000 <submitr+900>
0x0000000000401978 <+36>:    add     x1, x1, #0x908
=> 0x000000000040197c <+40>:    bl      0x400d70 <__isoc99_sscanf@plt>
0x0000000000401980 <+44>:    cmp     w0, #0x5
0x0000000000401984 <+48>:    b.le    0x401990 <read_six_numbers+60>
0x0000000000401988 <+52>:    ldp     x29, x30, [sp], #16
0x000000000040198c <+56>:    ret
0x0000000000401990 <+60>:    bl      0x401918 <explode_bomb>
End of assembler dump.
(gdb) x /s $x1
0x402908:      "%d %d %d %d %d %d"

```

图7-发现输入格式

- 故第二阶段需要输入六个整数，以空格隔开；
- 回到 **phase_2**，分析接下来的指令（图 8 中高亮部分），得到第一个数需要等于 0；

```

0x0000000000401000 <+16>:    bl      0x401934 <read_six_numbers>
0x0000000000401064 <+20>:    ldr     w0, [x29, #40]
0x0000000000401068 <+24>:    cbnz    w0, 0x401078 <phase_2+40>
0x000000000040106c <+28>:    ldr     w0, [x29, #44]
0x0000000000401070 <+32>:    cmp     w0, #0x1
0x0000000000401074 <+36>:    b.eq    0x40107c <phase_2+44> // b.none
> 0x0000000000401078 <+40>:    bl      0x401918 <explode_bomb>

```

图8-分析得第一个数

- 根据<+28><+32>处的指令，分析得到第二个数需要等于 1；
- 分析接下来的一连串的跳转指令，分析的 **w1** 依次存储第 3, 4, 5, 6 个数，**w0** 存储正确的值；
- 分析具体数据得到第 3 个数等于前两个数和，第 4 个数等于第 2 个数+第 3 个数，同理第 5, 6 个数；

```

0x0000000000401078 <+40>:    bl      0x401918 <explode_bomb>
0x000000000040107c <+44>:    add     x19, x29, #0x28
0x0000000000401080 <+48>:    add     x20, x19, #0x10
0x0000000000401084 <+52>:    b       0x401094 <phase_2+68>
0x0000000000401088 <+56>:    add     x19, x19, #0x4
0x000000000040108c <+60>:    cmp     x19, x20
0x0000000000401090 <+64>:    b.eq    0x4010b4 <phase_2+100> // b.none
0x0000000000401094 <+68>:    ldr     w0, [x19]
0x0000000000401098 <+72>:    ldr     w1, [x19, #4]
0x000000000040109c <+76>:    add     w0, w0, w1
0x00000000004010a0 <+80>:    ldr     w1, [x19, #8]
0x00000000004010a4 <+84>:    cmp     w1, w0
0x00000000004010a8 <+88>:    b.eq    0x401088 <phase_2+56> // b.none
0x00000000004010ac <+92>:    bl      0x401918 <explode_bomb>

```

图9-分析后续数字

- 故得到最终答案为 **0 1 1 2 3 5**；
- 输入答案，通过，进入第二阶段。

第三阶段

- 同理设置断点，输入阶段一、二答案以及测试密码，进入阶段三；

- 显示汇编代码，开始阶段三的分析：

```
(gdb) b explode_bomb
Breakpoint 1 at 0x01028
(gdb) b phase_3
Breakpoint 2 at 0x010dc
(gdb) r ans.txt
Starting program: /students/2022211683/bomb136/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!
12345

Breakpoint 2, 0x0000000004010dc in phase_3 ()
(gdb) disas
Dump of assembler code for function phase_3:
0x0000000004010c0 <+0>: stp    x29, x30, [sp, #-32]!
0x0000000004010c4 <+4>: mov    x29, sp
0x0000000004010c8 <+8>: add    x4, x29, #0x18
0x0000000004010cc <+12>: add    x3, x29, #0x17
0x0000000004010d0 <+16>: add    x2, x29, #0x1c
0x0000000004010d4 <+20>: adrp   x1, 0x402008 <submit+90>
0x0000000004010d8 <+24>: add    x1, x1, #0x730
0x0000000004010dc <+28>: b       0x400070 <__isoc99_sscanf@plt>
=> 0x0000000004010e0 <+32>: cmp    w0, #0x2
0x0000000004010e4 <+36>: b.le   0x401130 <phase_3+112>
0x0000000004010e8 <+40>: ldr     w0, [x29, #28]
0x0000000004010ec <+44>: cmp    w0, #0x3
0x0000000004010f0 <+48>: b.eq   0x40119c <phase_3+228> // b.none
0x0000000004010f4 <+52>: b.le   0x401138 <phase_3+120>
0x0000000004010f8 <+56>: cmp    w0, #0x5
0x0000000004010fc <+60>: b.eq   0x4011d4 <phase_3+276> // b.none
0x000000000401100 <+64>: b.lt   0x4011b8 <phase_3+248> // b.tstop
0x000000000401104 <+68>: cmp    w0, #0x6
0x000000000401108 <+72>: b.eq   0x4011f0 <phase_3+304> // b.any
0x00000000040110c <+76>: cmp    w0, #0x7
0x000000000401110 <+80>: b.ne   0x40120c <phase_3+332> // b.any // #97
0x000000000401114 <+84>: mov    w0, #0x61
0x000000000401118 <+88>: ldr     w1, [x29, #24]
0x00000000040111c <+92>: cmp    w1, #0x33c
0x000000000401120 <+96>: b.eq   0x401214 <phase_3+340> // b.none
0x000000000401124 <+100>: bl     0x401918 <explode_bomb>
0x000000000401128 <+104>: mov    w0, #0x61
0x00000000040112c <+108>: b       0x401214 <phase_3+340> // #97
--Type <RET> for more, q to quit, c to continue without paging--c
0x000000000401130 <+112>: bl     0x401918 <explode_bomb>
0x000000000401134 <+116>: cmp    0x4018e8 <phase_3+40>
0x000000000401138 <+120>: cmp    w0, #0x1
0x00000000040113c <+124>: b.eq   0x4011d4 <phase_3+276> // b.none
0x000000000401140 <+128>: b.gt   0x4011b8 <phase_3+248>
0x000000000401144 <+132>: cbnz   w0, 0x40110c <phase_3+332>
0x000000000401148 <+136>: mov    w0, #0x70
0x00000000040114c <+140>: ldr     w1, [x29, #24]
0x000000000401150 <+144>: cmp    w1, #0x2ae
0x000000000401154 <+148>: b.eq   0x401214 <phase_3+340> // b.none
0x000000000401158 <+152>: bl     0x401918 <explode_bomb>
0x00000000040115c <+156>: mov    w0, #0x70
0x000000000401160 <+160>: b       0x401214 <phase_3+340> // #112
0x000000000401164 <+164>: mov    w0, #0x70
0x000000000401168 <+168>: ldr     w1, [x29, #24]
0x00000000040116c <+172>: cmp    w1, #0x6f
0x000000000401170 <+176>: b.eq   0x401214 <phase_3+340> // b.none
0x000000000401174 <+180>: bl     0x401918 <explode_bomb>
0x000000000401178 <+184>: mov    w0, #0x70
0x00000000040117c <+188>: b       0x401214 <phase_3+340> // #112
0x000000000401180 <+192>: mov    w0, #0x6c
0x000000000401184 <+196>: ldr     w1, [x29, #24]
0x000000000401188 <+200>: cmp    w1, #0x5e
0x00000000040118c <+204>: b.eq   0x401214 <phase_3+340> // b.none
0x000000000401190 <+208>: bl     0x401918 <explode_bomb>
0x000000000401194 <+212>: mov    w0, #0x6c
0x000000000401198 <+216>: b       0x401214 <phase_3+340> // #108
0x00000000040119c <+220>: mov    w0, #0x72
0x0000000004011a0 <+224>: ldr     w1, [x29, #24]
0x0000000004011a4 <+228>: cmp    w1, #0x238
0x0000000004011a8 <+232>: b.eq   0x401214 <phase_3+340> // b.none
0x0000000004011ac <+236>: bl     0x401918 <explode_bomb>
0x0000000004011b0 <+240>: mov    w0, #0x72
0x0000000004011b4 <+244>: b       0x401214 <phase_3+340> // #114
0x0000000004011b8 <+248>: mov    w0, #0x6b
0x0000000004011bc <+252>: ldr     w1, [x29, #24]
0x0000000004011c0 <+256>: cmp    w1, #0x2db
0x0000000004011c4 <+260>: b.eq   0x401214 <phase_3+340> // b.none
0x0000000004011c8 <+264>: bl     0x401918 <explode_bomb>
0x0000000004011cc <+268>: mov    w0, #0x6b
0x0000000004011d0 <+272>: b       0x401214 <phase_3+340> // #107
0x0000000004011d4 <+276>: mov    w0, #0x70
0x0000000004011d8 <+280>: ldr     w1, [x29, #24]
0x0000000004011dc <+284>: cmp    w1, #0x189
0x0000000004011e0 <+288>: b.eq   0x401214 <phase_3+340> // b.none
0x0000000004011e4 <+292>: bl     0x401918 <explode_bomb>
0x0000000004011e8 <+296>: mov    w0, #0x70
0x0000000004011ec <+300>: b       0x401214 <phase_3+340> // #112
0x0000000004011f0 <+304>: mov    w0, #0x77
0x0000000004011f4 <+308>: ldr     w1, [x29, #24]
0x0000000004011f8 <+312>: cmp    w1, #0x15c
0x0000000004011fc <+316>: b.eq   0x401214 <phase_3+340> // b.none
0x000000000401200 <+320>: bl     0x401918 <explode_bomb>
0x000000000401204 <+324>: mov    w0, #0x77
0x000000000401208 <+328>: b       0x401214 <phase_3+340> // #119
0x00000000040120c <+332>: bl     0x401918 <explode_bomb>
0x000000000401210 <+336>: mov    w0, #0x78
0x000000000401214 <+340>: ldrb    w1, [x29, #23]
0x000000000401218 <+344>: cmp    w1, w0
0x00000000040121c <+348>: b.eq   0x401224 <phase_3+356> // b.none
0x000000000401220 <+352>: bl     0x401918 <explode_bomb>
0x000000000401224 <+356>: ldp     x29, x30, [sp], #32
0x000000000401228 <+360>: ret
End of assembler dump.
```

图10-

阶段三开始

- 同理阶段二，查看寄存器 **x1** 的内容，发现该阶段需要输入一个整数，一个字符，一个整数，用空格隔开；

```
(gdb) x /s $x1
0x402730: "%d %c %d"
```

图11-输入格式

- 由图 12 所示指令可知，第一个输入需要大于 2，否则直接爆炸；

```
=> 0x0000000004010dc <+28>: bl     0x400d70 <__isoc99_sscanf@
0x0000000004010e0 <+32>: cmp    w0, #0x2
0x0000000004010e4 <+36>: b.le   0x401130 <phase_3+112>
0x0000000004010e8 <+40>: ldr     w0, [x29, #28]
```

图12-第一个整数范围

- 接下来是一连串 **switch** 语句，有了初阶炸弹的经验，我们不需要将所有种类全部求出，只需分析一种正确答案即可，此处我们选择第一个输入为 3 的情况；
- 分析可知：当第一个输入为 3 时，跳转到<+220>处，我们继续分析<+220>处的指令；

```
0x000000000401198 <+216>: b       0x401214 <phase_3+340>
0x00000000040119c <+220>: mov     w0, #0x72 // #114
0x0000000004011a0 <+224>: ldr     w1, [x29, #24]
0x0000000004011a4 <+228>: cmp     w1, #0x238
0x0000000004011a8 <+232>: b.eq    0x401214 <phase_3+340> // b.none
0x0000000004011ac <+236>: bl      0x401918 <explode_bomb>
```

图13-第三个输入的确定

- 由图 13 可知，当第一个输入为 3 的时候，第三个输入需要等于 **0x238**，即 **568**；
- 然后跳转到<+340>处，分析第二个输入；

```

0x0000000000401210 <+336>:  mov    w0, #0x78                // #120
0x0000000000401214 <+340>:  ldrb   w1, [x29, #23]
0x0000000000401218 <+344>:  cmp    w1, w0
0x000000000040121c <+348>:  b.eq   0x401224 <phase_3+356> // b.none
0x0000000000401220 <+352>:  bl     0x401918 <explode_bomb>

```

图14-第二个输入的确切

- 由第三个输入处的指令可知，此时 **w0** 存储 **0x72**，转换成字符则为 **r**，故第二个字符是 **r**；
- 输入 **3 r 568**，通过，进入第四阶段。

第四阶段

- 同理设置断点，输入阶段一、二、三答案以及测试密码，进入阶段四；
- 显示汇编代码，开始阶段四的分析；

```

(gdb) b explode_bomb
Breakpoint 1 at 0x401928
(gdb) b phase_4
Breakpoint 2 at 0x401298
(gdb) r ans.txt
Starting program: /students/2022211683/bomb136/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!
12345

Breakpoint 2, 0x0000000000401298 in phase_4 ()
(gdb) disas
Dump of assembler code for function phase_4:
0x0000000000401280 <+0>:  stp    x29, x30, [sp, #-32]!
0x0000000000401284 <+4>:  mov    x29, sp
0x0000000000401288 <+8>:  add    x3, x29, #0x18
0x000000000040128c <+12>: add    x2, x29, #0x1c
0x0000000000401290 <+16>: adrp   x1, 0x402000 <submitr+900>
0x0000000000401294 <+20>: add    x1, x1, #0x740
=> 0x0000000000401298 <+24>: bl     0x40d70 <__isoc99_sscanf@plt>
0x000000000040129c <+28>: cmp    w0, #0x2
0x00000000004012a0 <+32>: b.ne   0x4012b0 <phase_4+48> // b.any
0x00000000004012a4 <+36>: ldr     w0, [x29, #28]
0x00000000004012a8 <+40>: cmp    w0, #0xe
0x00000000004012ac <+44>: b.ls   0x4012b4 <phase_4+52> // b.plast
0x00000000004012b0 <+48>: bl     0x401918 <explode_bomb>
0x00000000004012b4 <+52>: mov    w2, #0xe                // #14
0x00000000004012b8 <+56>: mov    w1, #0x0                // #0
0x00000000004012bc <+60>: ldr     w0, [x29, #28]
0x00000000004012c0 <+64>: bl     0x40122c <func4>
0x00000000004012c4 <+68>: cmp    w0, #0x1b
0x00000000004012c8 <+72>: b.ne   0x4012d8 <phase_4+88> // b.any
0x00000000004012cc <+76>: ldr     w0, [x29, #24]
0x00000000004012d0 <+80>: cmp    w0, #0x1b
0x00000000004012d4 <+84>: b.eq   0x4012dc <phase_4+92> // b.none
0x00000000004012d8 <+88>: bl     0x401918 <explode_bomb>
0x00000000004012dc <+92>: ldp     x29, x30, [sp], #32
0x00000000004012e0 <+96>: ret
End of assembler dump.

```

图15-阶段四开始

- 同理查看寄存器 **x1** 的内容，此阶段需要输入两个整数，中间用空格隔开；

```

(gdb) x /s $x1
0x402740:      "%d %d"

```

图16-阶段四输入格式

- 依次向下分析，第一个数需要 ≤ 14 ；

```

0x00000000004012a4 <+36>:  ldr     w0, [x29, #28]
0x00000000004012a8 <+40>:  cmp    w0, #0xe
0x00000000004012ac <+44>:  b.ls   0x4012b4 <phase_4+52> // b.plast
0x00000000004012b0 <+48>:  bl     0x401918 <explode_bomb>
0x00000000004012b4 <+52>:  mov    w2, #0xe                // #14
0x00000000004012b8 <+56>:  mov    w1, #0x0                // #0

```

图17-确定第一个数

- 根据<+80>处知道第二个输入必须是 27，而且<+68>处告诉我们该函数 **func4** 最后的返回值应该是 27；
- 设置断点 **func4** 并进入，查看 **func4** 汇编指令；

```

(gdb) b func4
Breakpoint 3 at 0x401238
(gdb) c
Continuing.

Breakpoint 3, 0x00000000401238 in func4 ()
(gdb) disas
Dump of assembler code for function func4:
0x0000000040122c <+0>: stp x29, x30, [sp, #-32]!
0x00000000401230 <+4>: mov x29, sp
0x00000000401234 <+8>: str x19, [sp, #16]
=> 0x00000000401238 <+12>: sub w19, w2, w1
0x0000000040123c <+16>: add w19, w19, w19, lsr #31
0x00000000401240 <+20>: add w19, w1, w19, asr #1
0x00000000401244 <+24>: cmp w19, w0
0x00000000401248 <+28>: b.gt 0x401260 <func4+52>
0x0000000040124c <+32>: b.lt 0x401270 <func4+68> // b.tstop
0x00000000401250 <+36>: mov w0, w19
0x00000000401254 <+40>: ldr x19, [sp, #16]
0x00000000401258 <+44>: ldp x29, x30, [sp], #32
0x0000000040125c <+48>: ret
0x00000000401260 <+52>: sub w2, w19, #0x1
0x00000000401264 <+56>: bl 0x40122c <func4>
0x00000000401268 <+60>: add w19, w19, w0
0x0000000040126c <+64>: b 0x401250 <func4+36>
0x00000000401270 <+68>: add w1, w19, #0x1
0x00000000401274 <+72>: bl 0x40122c <func4>
0x00000000401278 <+76>: add w19, w19, w0
0x0000000040127c <+80>: b 0x401250 <func4+36>
End of assembler dump.

```

图18-func4函数

- 该函数是一个递归函数，通过查看寄存器的值可知，**x0** 存储第一个输入数；
- 执行完整的递归函数，可以得到返回值为第一个输入的 2 倍再加 9，即 $2 \times a + 9$ ；
- 故为使返回值为 27，可求得第一个输入为 $(27 - 9) \div 2 = 9$ ；
- 输入 **9 27**，通过，进入第五阶段。

第五阶段

- 同理设置断点，输入阶段一、二、三、四答案以及测试密码，进入阶段五；
- 显示汇编代码，开始阶段五的分析；

```

(gdb) b explode_bomb
Breakpoint 1 at 0x401928
(gdb) b phase_5
Breakpoint 2 at 0x4012f4
(gdb) r ans.txt
Starting program: /students/2022211683/bomb136/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.
123456

Breakpoint 2, 0x000000004012f4 in phase_5 ()
(gdb) disas
Dump of assembler code for function phase_5:
0x000000004012e4 <+0>: stp x29, x30, [sp, #-48]!
0x000000004012e8 <+4>: mov x29, sp
0x000000004012ec <+8>: str x19, [sp, #16]
0x000000004012f0 <+12>: mov x19, x0
=> 0x000000004012f4 <+16>: bl 0x4015e8 <string_length>
0x000000004012f8 <+20>: cmp w0, #0x6
0x000000004012fc <+24>: b.ne 0x401350 <phase_5+108> // b.any
0x00000000401300 <+28>: mov x0, #0x0 // #0
0x00000000401304 <+32>: add x3, x29, #0x28
0x00000000401308 <+36>: adrp x2, 0x402000 <submitr+900>
0x0000000040130c <+40>: add x2, x2, #0x6d8
0x00000000401310 <+44>: ldrb w1, [x19, x0]
0x00000000401314 <+48>: and w1, w1, #0xf
0x00000000401318 <+52>: ldrb w1, [x2, w1, sxtw]
0x0000000040131c <+56>: strb w1, [x0, x3]
0x00000000401320 <+60>: add x0, x0, #0x1
0x00000000401324 <+64>: cmp x0, #0x6
0x00000000401328 <+68>: b.ne 0x401310 <phase_5+44> // b.any
0x0000000040132c <+72>: strb wzr, [x29, #46]
0x00000000401330 <+76>: adrp x1, 0x402000 <submitr+900>
0x00000000401334 <+80>: add x1, x1, #0x748
0x00000000401338 <+84>: add x0, x29, #0x28
0x0000000040133c <+88>: bl 0x401614 <strings_not_equal>
0x00000000401340 <+92>: cbnz w0, 0x401358 <phase_5+116>
0x00000000401344 <+96>: ldr x19, [sp, #16]
0x00000000401348 <+100>: ldp x29, x30, [sp], #48
0x0000000040134c <+104>: ret
0x00000000401350 <+108>: bl 0x401918 <explode_bomb>
--Type <RET> for more, q to quit, c to continue without paging--c
0x00000000401354 <+112>: b 0x401300 <phase_5+28>
0x00000000401358 <+116>: bl 0x401918 <explode_bomb>
0x0000000040135c <+120>: b 0x401344 <phase_5+96>
End of assembler dump.

```

图19-阶段五开始

- 与阶段 1 相似，同理分析 `string_length` 函数后的 `cmp`，分析出正确答案由六个字符组成；
- 由初阶炸弹的经验可知，此题是通过输入值挑选字符组成正确答案；
- 设置断点进入 `strings_not_equal` 函数，查看此时寄存器的值；

```
(gdb) x /s $x1
0x402748: "devils"
(gdb) x /s $x0
0xffffffff988: "aduier"
(gdb) x /s $x2
0x4026d8 <array.4328>: "maduiersnfotvbylThere are many handsome guys and beautiful girls on ShaHe campus."
```

图20-答案字符串，取值字符串

- 可以发现，`x0` 存储的是根据我们的输入 `123456` 挑选出来的字符串，`x1` 存储的是答案字符串，`x2` 存储的是用来取数据的字符串；
- 对于答案 `devils`，其索引值为 2, 5, 12, 4, 15, 7，只需要输入的字符的低四位符合对应的索引即可；
- 在每个索引上加上 64（这样不会改变低四位），得到 66, 69, 76, 68, 79, 71，对应字符串 `BELDOG`；
- 输入 `BELDOG`，通过。

答案汇总与通过截图

```
(gdb) r ans.txt
Starting program: /students/2022211683/bomb136/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...
```

图21-5个阶段全部通过

```
[2022211683@kunpeng1 bomb136]$ cat ans.txt
There are many handsome guys and beautiful girls on ShaHe campus.
0 1 1 2 3 5
3 r 568
9 27
BELDOG
[2022211683@kunpeng1 bomb136]$
```

图22-全部答案

五、总结体会

总的来说，这次实验完成较好，有了初阶实炸弹的经验和教训，这次实验不论是完成时间还是正确率都达到了较好的水平，共用时 4 小时不到，并且没有触发任何一次爆炸，总体完成较好。

本次实验中最大的困难是对 ARM 的指令集不熟悉，比如 `ldr`, `bl` 等指令，还有源操作数和目的操作数顺序的不同，导致在第二阶段的破解花费了较长的时间，认真学习了相关资

料，查阅了文件之后，我才对 ARM 的指令集渐渐熟悉起来，第二阶段完成之后，后面几个阶段的实验的完成速度就有了大幅度的提高，并且由于高阶实验和初阶实验中涉及到的功能和知识点类似，我也成功规避了我在初阶实验中所犯的一些错误和弯路。

我相信在这两次实验中我所学到的东西、我所得到的锻炼达到了我的预期，特别是对 gdb 指令的使用，已经熟练了许多。希望后续的计算机系统基础的学习能够更进一步！