

北京郵電大學

实验报告



题目: 拆解二进制炸弹 Pro Max Ultra

班级: 2022211320

学号: 2022212408

姓名: 胡宇杭

学院: 计算机学院 (国家示范性软件学院)

2023 年 11 月 12 日

目录

| | |
|------------------|----|
| 1 实验目的 | 3 |
| 2 实验环境 | 3 |
| 3 实验内容 | 3 |
| 4 实验步骤及实验分析 | 4 |
| 4.1 准备工作 | 4 |
| 4.2 第一炸弹：字符串炸弹 | 4 |
| 4.3 第二炸弹：枯萎穿心攻击 | 7 |
| 4.4 第三炸弹：败者食尘 | 10 |
| 4.5 第四炸弹：递归炸弹 | 14 |
| 4.6 第五炸弹：真·二进制炸弹 | 16 |
| 5 总结体会 | 20 |

1 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 **ARM** 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；

2 实验环境

- 系统: **openEuler**
- 软件工具: **macOS Terminal**(10.99.0.230)、**gcc 7.3.0**、**GNU gdb 9.2-3.oe1**、**GNU objdump 2.34**

3 实验内容

- 登录 **kunpeng3** 服务器，在 **home** 目录下可以找到 **Dr. Evil** 专门为量身定制的一个 **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 **openEuler** 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。

4 实验步骤及实验分析

4.1 准备工作

1. 登陆 **kunpeng3** 服务器, 在自己的目录下找到 **bomb132.tar** 文件, 使用 **tar -xvf bomb132.tar** 命令解压缩该文件。

```
[[202212408@kunpeng1 ~]$ ls
bomb132 bomb132.tar
[[202212408@kunpeng1 ~]$ tar -xvf bomb132.tar
bomb132/README
bomb132/bomb.c
bomb132/bomb
[202212408@kunpeng1 ~]$ █
```

2. 使用 **cd bomb132** 命令移动到 **bomb132** 目录下并使用 **ls** 命令查看当前目录下的文件。可以发现, **bomb** 即是我们需要逆向的可执行文件, **bomb.c** 是提供的部分 C 代码。

```
[[202212408@kunpeng1 ~]$ cd bomb132
[[202212408@kunpeng1 bomb132]$ ls
bomb bomb.c README
```

3. 使用 **gdb bomb** 命令进入调试阶段。至此, 准备工作完毕, 开始正式拆弹。

```
[[202212408@kunpeng1 bomb132]$ gdb bomb
GNU gdb (GDB) openEuler 9.2-3.oe1
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 "aarch64-openEuler-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) █
```

4.2 第一炸弹：字符串炸弹

写在前面: 由于调试过程中需要用到大量 **disassemble**、**stepi**、**nexti** 指令, 全部截图显得十分冗余, 故只在 **第一炸弹**展示完整截图, 其他部分相关命令只进行文字描

述。其次，有时候会忘记截图，想起来的时候代码已经执行过几行了，所以看到进入函数后第一个 **disassemble** 命令指向的位置不是第一句，请不要感到意外。

1. 进入调试界面，我们首先使用 **list** 命令查看第一颗炸弹的位置，在 74 行发现了第一颗炸弹。于是我们使用 **break 74** 命令在该行设置一个断点。

```
[(gdb) list 74
69      printf("Welcome to my fiendish little bomb. You have 6 phases with\n");
70      printf("which to blow yourself up. Have a nice day!\n");
71
72      /* Hmm... Six phases must be more secure than one phase! */
73      input = read_line();           /* Get input                  */
74      phase_1(input);             /* Run the phase              */
75      phase_defused();            /* Drat! They figured it out!
76                           * Let me know how they did it. */
77      printf("Phase 1 defused. How about the next one?\n");
78
[(gdb) b 74
Breakpoint 1 at 0x400f54: file bomb.c, line 74.
```

2. 使用 **run** 命令运行程序，提示输入第一颗炸弹。由于我们此时并不知道密码是多少，因此随便填写。填写完成后，程序运行到断点位置暂停。

```
[(gdb) r
Starting program: /students/2022212408/bomb132/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
[test

Breakpoint 1, main (argc=<optimized out>, argv=<optimized out>) at bomb.c:74
74      phase_1(input);           /* Run the phase              */
```

3. 接下来我们使用 **stepi** 命令单步执行，进入 **phase_1(input)** 函数内部进行拆弹。

```
[(gdb) stepi
0x0000000000401028 in phase_1 ()
```

4. 使用 **disassemble** 命令查看第一颗炸弹的汇编代码。接下来对汇编代码进行分析：在 <+8> 位置，将 **\$0x402000** 移动到了 **x1** 寄存器中，并加上了 **0x680**，猜测该寄存器存放了我们输入的字符串作为参数传入 **strings_not_equal** 函数中，然后在 <+20> 位置判断函数返回值是否为 **0**。如果是，则继续执行；否则转跳到 <+32> 位置，引爆炸弹，**BOMB!!!**

```
[(gdb) disas
Dump of assembler code for function phase_1:
=> 0x0000000000401028 <+0>:    stp      x29, x30, [sp, #-16]!
  0x000000000040102c <+4>:    mov      x29, sp
  0x0000000000401030 <+8>:    adrp     x1, 0x402000 <submitr+1052>
  0x0000000000401034 <+12>:   add      x1, x1, #0x680
  0x0000000000401038 <+16>:   bl       0x40157c <strings_not_equal>
  0x000000000040103c <+20>:   cbnz    w0, 0x401048 <phase_1+32>
  0x0000000000401040 <+24>:   ldp      x29, x30, [sp], #16
  0x0000000000401044 <+28>:   ret
  0x0000000000401048 <+32>:   bl       0x401880 <explode_bomb>
  0x000000000040104c <+36>:   b        0x401040 <phase_1+24>
End of assembler dump.]
```

5. 为了验证我们的猜想，我们先使用 **stepi** 命令移动到 <+16> 位置，然后使用 **x /s** 命令查看 **x1** 寄存器中究竟存放着什么，结果发现竟然就是我们梦寐以求的密码，第一颗炸弹成功解除！（更详细的方法和解释在基础版炸弹写过了，这里懒得写了）

```
[(gdb) stepi 3
0x0000000000401034 in phase_1 ()
[(gdb) stepi
0x0000000000401038 in phase_1 ()
[(gdb) x /s $x1
0x402680: "Houses will begat jobs, jobs will begat houses."]
```

6. 重新运行程序，输入密码，第一炸弹被拆除。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
[Houses will begat jobs, jobs will begat houses.
Phase 1 defused. How about the next one?]
```

4.3 第二炸弹：枯萎穿心攻击

- 我们在第二炸弹位置设置断点，进入函数内部，使用 `disassemble` 命令查看汇编代码。

```
[gdb] disas
Dump of assembler code for function phase_2:
=> 0x0000000000401050 <+0>:    stp      x29, x30, [sp, #-64]!
  0x0000000000401054 <+4>:    mov      x29, sp
  0x0000000000401058 <+8>:    stp      x19, x20, [sp, #16]
  0x000000000040105c <+12>:   add     x1, x29, #0x28
  0x0000000000401060 <+16>:   bl      0x4018bc <read_six_numbers>
  0x0000000000401064 <+20>:   ldr     w0, [x29, #40]
  0x0000000000401068 <+24>:   cmp     w0, #0x1
  0x000000000040106c <+28>:   bne    0x40107c <phase_2+44> // b.any
  0x0000000000401070 <+32>:   add     x19, x29, #0x28
  0x0000000000401074 <+36>:   add     x20, x19, #0x14
  0x0000000000401078 <+40>:   b      0x401090 <phase_2+64>
  0x000000000040107c <+44>:   bl      0x401880 <explode_bomb>
  0x0000000000401080 <+48>:   b      0x401070 <phase_2+32>
  0x0000000000401084 <+52>:   add     x19, x19, #0x4
  0x0000000000401088 <+56>:   cmp     x19, x20
  0x000000000040108c <+60>:   beq    0x4010a8 <phase_2+88> // b.none
  0x0000000000401090 <+64>:   ldr     w1, [x19]
  0x0000000000401094 <+68>:   ldr     w0, [x19, #4]
  0x0000000000401098 <+72>:   cmp     w0, w1, lsl #1
  0x000000000040109c <+76>:   beq    0x401084 <phase_2+52> // b.none
  0x00000000004010a0 <+80>:   bl      0x401880 <explode_bomb>
  0x00000000004010a4 <+84>:   b      0x401084 <phase_2+52>
[--Type <RET> for more, q to quit, c to continue without paging--
  0x00000000004010a8 <+88>:   ldp     x19, x20, [sp, #16]
  0x00000000004010ac <+92>:   ldp     x29, x30, [sp], #64
  0x00000000004010b0 <+96>:   ret
End of assembler dump.
```

- 由函数 `read_six_numbers` 可以推测这关的密码是 6 个符合一定顺序的数字，为了验证假设，我们进入函数内部查看其汇编代码。
- 在函数内部我们发现了提示中的关键函数 `sscanf`，在函数之后程序将 `w0` 寄存器的内容（`sscanf` 的返回值）与 5 做比较，并在其小于等于 5 时引爆炸弹。

```
0x00000000004018bc in read_six_numbers ()
[gdb] disas
Dump of assembler code for function read_six_numbers:
=> 0x00000000004018bc <+0>:    stp      x29, x30, [sp, #-16]!
  0x00000000004018c0 <+4>:    mov      x29, sp
  0x00000000004018c4 <+8>:    add     x7, x1, #0x14
  0x00000000004018c8 <+12>:   add     x6, x1, #0x10
  0x00000000004018cc <+16>:   add     x5, x1, #0xc
  0x00000000004018d0 <+20>:   add     x4, x1, #0x8
  0x00000000004018d4 <+24>:   add     x3, x1, #0x4
  0x00000000004018d8 <+28>:   mov     x2, x1
  0x00000000004018dc <+32>:   adrp    x1, 0x402000 <submitr+1052>
  0x00000000004018e0 <+36>:   add     x1, x1, #0x870
  0x00000000004018e4 <+40>:   bl      0x400d70 <__isoc99_sscanf@plt>
  0x00000000004018e8 <+44>:   cmp     w0, #0x5
  0x00000000004018ec <+48>:   ble    0x4018f8 <read_six_numbers+60>
  0x00000000004018f0 <+52>:   ldp     x29, x30, [sp], #16
  0x00000000004018f4 <+56>:   ret
  0x00000000004018f8 <+60>:   bl      0x401880 <explode_bomb>
End of assembler dump.
```

4. 我们移动至 `<+40>` 位置，使用 `x /s` 命令查看 `sscanf` 的第一个参数值 `x1`，发现函数确实是读入 6 个数字，并且中间用空格隔开。

```
(gdb) x /s $x1
0x402870:      "%d %d %d %d %d %d"
```

5. 现在让我们来分析一下 `phase_2` 中剩下的内容：

- 读入完成后程序将 `w0 + 40`（即 `w0 + 0x28`, `x1` 寄存器指向的地址）指向地址的值与 `0x1` 做比较，若不相等则引爆炸弹。

```
0x000000000040105c <+12>:    add    x1, x29, #0x28
0x0000000000401060 <+16>:    bl     0x4018bc <read_six_numbers>
0x0000000000401064 <+20>:    ldr    w0, [x29, #40]
0x0000000000401068 <+24>:    cmp    w0, #0x1
0x000000000040106c <+28>:    b.ne   0x40107c <phase_2+44> // b.any
```

- 接下来程序直接跳转至 `<+64>` 位置，分别把 `[x19]` 和 `[x19, #4]` 地址的值移动到 `w1` 和 `w0` 寄存器中，接下来将 `w0` 和 `w1` 左移一位后的值做比较，若为真则跳转回 `<+52>` 位置；若为否则引爆炸弹，推测 `x19` 寄存器指向的一片地址空间（即 数组）存放着我们输入的数据，使用 `x /6d` 命令查看，发现结果和我们的推测一样（此处是补截的图，所以输入为密码），与此同时，我们也知道了之前 `w0 + 40` 指向的值与 `0x1` 做比较是为了检查数组的第一个元素是否为 1。

```
0x0000000000401090 <+64>:    ldr    w1, [x19]
0x0000000000401094 <+68>:    ldr    w0, [x19, #4]
0x0000000000401098 <+72>:    cmp    w0, w1, lsl #1
0x000000000040109c <+76>:    b.eq   0x401084 <phase_2+52> // b.none
(gdb) x /6d $x19
0xffffffffffff988: 1          2          4          8
0xffffffffffff998: 16         32
```

- 接下来我们先看到 `<+52>` 位置，将 `x19` 加上 `0x4` 的偏移量，此时 `x19` 指向的元素从数组的第一个元素变为数组的第二个元素，接下来将 `x20` 与 `x19` 做比较，如果相等，则跳转到 `<+88>` 位置，程序结束；否则程序继续运行，再次执行上述判断。

```
0x0000000000401084 <+52>:    add      x19, x19, #0x4
0x0000000000401088 <+56>:    cmp      x19, x20
0x000000000040108c <+60>:    b.eq    0x4010a8 <phase_2+88> // b.none
```

- 回头看到 `<+36>` 位置，发现 `x20` 存放的是 `x19 + 0x14`，现在，我们可以看出这是一个 `do while` 循环。

```
0x0000000000401074 <+36>:    add      x20, x19, #0x14
```

6. 至此，我们基本了解了这段汇编代码的功能：每次迭代判断数组第 `k` 个元素是否为下一个元素的 $\frac{1}{2}$ 。如果不等，则引爆炸弹；否则移动到 `k + 1` 位置，直到 `x19 = x19 + 0x14` 时停止迭代，即当 `k = 5` 时停止循环。因此，所求密码是首项为 1，公比为 2，长度为 6 的等比数列。即 1 2 4 8 16 32。
7. 重新运行输入密码，成功拆除第二炸弹。

```
Phase 1 defused. How about the next one?
[1 2 4 8 16 32
That's number 2. Keep going!
```

4.4 第三炸弹：败者食尘

1. 查看第三炸弹的汇编代码如下：

```
((gdb) disas
Dump of assembler code for function phase_3:
=> 0x00000000004010b4 <+0>:    stp    x29, x30, [sp, #-32]!
0x00000000004010b8 <+4>:    mov    x29, sp
0x00000000004010bc <+8>:    add    x3, x29, #0x18
0x00000000004010c0 <+12>:   add    x2, x29, #0x1c
0x00000000004010c4 <+16>:   adrp   x1, 0x402000 <submitr+1052>
0x00000000004010c8 <+20>:   add    x1, x1, #0x6b0
0x00000000004010cc <+24>:   bl     0x400d70 <__isoc99_sscanf_pplt>
0x00000000004010d0 <+28>:   cmp    w0, #0x1
0x00000000004010d4 <+32>:   b.le   0x401114 <phase_3+96>
0x00000000004010d8 <+36>:   ldr    w1, [x29, #28]
0x00000000004010dc <+40>:   cmp    w1, #0x3
0x00000000004010e0 <+44>:   b.eq   0x401180 <phase_3+204> // b.none
0x00000000004010e4 <+48>:   b.le   0x40111c <phase_3+104>
0x00000000004010e8 <+52>:   cmp    w1, #0x5
0x00000000004010ec <+56>:   b.eq   0x401190 <phase_3+220> // b.none
0x00000000004010f0 <+60>:   b.lt   0x401188 <phase_3+212> // b.tstop
0x00000000004010f4 <+64>:   cmp    w1, #0x6
0x00000000004010f8 <+68>:   b.eq   0x401198 <phase_3+228> // b.none
0x00000000004010fc <+72>:   mov    w0, #0x0
0x0000000000401100 <+76>:   cmp    w1, #0x7
0x0000000000401104 <+80>:   b.eq   0x401148 <phase_3+148> // b.none
0x0000000000401108 <+84>:   bl     0x401880 <explode_bomb>
[--Type <RET> for more, q to quit, c to continue without paging--]
0x000000000040110c <+88>:   mov    w0, #0x0
0x0000000000401110 <+92>:   b     0x40114c <phase_3+152>
0x0000000000401114 <+96>:   bl     0x401880 <explode_bomb>
0x0000000000401118 <+100>:  b     0x4010d8 <phase_3+36>
0x000000000040111c <+104>:  cmp    w1, #0x1
0x0000000000401120 <+108>:  b.eq   0x401170 <phase_3+188> // b.none
0x0000000000401124 <+112>:  b.gt   0x401178 <phase_3+196>
0x0000000000401128 <+116>:  mov    w0, #0x1a1
0x000000000040112c <+120>:  cbnz  w1, 0x401188 <phase_3+84>
0x0000000000401130 <+124>:  sub    w0, w0, #0x39f
0x0000000000401134 <+128>:  add    w0, w0, #0x94
0x0000000000401138 <+132>:  sub    w0, w0, #0x1da
0x000000000040113c <+136>:  add    w0, w0, #0x1da
0x0000000000401140 <+140>:  sub    w0, w0, #0x1da
0x0000000000401144 <+144>:  add    w0, w0, #0x1da
0x0000000000401148 <+148>:  sub    w0, w0, #0x1da
0x000000000040114c <+152>:  ldr    w1, [x29, #28]
0x0000000000401150 <+156>:  cmp    w1, #0x5
0x0000000000401154 <+160>:  b.gt   0x401164 <phase_3+176>
0x0000000000401158 <+164>:  ldr    w1, [x29, #24]
0x000000000040115c <+168>:  cmp    w1, w0
0x0000000000401160 <+172>:  b.eq   0x401168 <phase_3+180> // b.none
0x0000000000401164 <+176>:  bl     0x401880 <explode_bomb>
[--Type <RET> for more, q to quit, c to continue without paging--]
0x0000000000401168 <+180>:  ldp    x29, x30, [sp], #32
0x000000000040116c <+184>:  ret
0x0000000000401170 <+188>:  mov    w0, #0x0
0x0000000000401174 <+192>:  b     0x401130 <phase_3+124>
0x0000000000401178 <+196>:  mov    w0, #0x0
0x000000000040117c <+200>:  b     0x401134 <phase_3+128>
0x0000000000401180 <+204>:  mov    w0, #0x0
0x0000000000401184 <+208>:  b     0x401138 <phase_3+132>
0x0000000000401188 <+212>:  mov    w0, #0x0
0x000000000040118c <+216>:  b     0x40113c <phase_3+136>
0x0000000000401190 <+220>:  mov    w0, #0x0
0x0000000000401194 <+224>:  b     0x401140 <phase_3+140>
0x0000000000401198 <+228>:  mov    w0, #0x0
0x000000000040119c <+232>:  b     0x401144 <phase_3+144>
End of assembler dump.
```

2. 我们还是如法炮制地查看 `sscanf` 的输入，发现这段密码是由 `数字 数字` 组成的，并且在输入小于一个字符时引爆炸弹。

```

0x00000000004010c4 <+16>:    adrp    x1, 0x402000 <submitr+1052>
0x00000000004010c8 <+20>:    add     x1, x1, #0x6b0
0x00000000004010cc <+24>:    bl      0x400d70 <_isoc99_sscanf@plt>
0x00000000004010d0 <+28>:    cmp     w0, #0x1
0x00000000004010d4 <+32>:    b.le   0x401114 <phase_3+96>
[(gdb) x /s $x1
0x4026b0:      "%d %d"

```

3. 继续往后，程序将某一地址存放的值存入 `w1` 寄存器，并将其与 `0x3` 比较，推测 `[x29, #28]` 位置存放的是我们传入的变量，使用 `print` 命令查看，发现是我们传入的第一个变量。

```

0x00000000004010d8 <+36>:    ldr     w1, [x29, #28]
[(gdb) p $w1
$2 = 4

```

4. 接下来是一系列判断语句，我们对其进行归纳并列出对应条件下的转跳位置。

| | | | | | | | |
|---------------------------------------------------------|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| <+116> <+188> <+196> <+204> <+212> <+220> <+228> <+148> | | | | | | | |


```

0x00000000004010d8 <+36>:    ldr     w1, [x29, #28]
0x00000000004010dc <+40>:    cmp     w1, #0x3
0x00000000004010e0 <+44>:    b.eq   0x401180 <phase_3+204> // b.none
0x00000000004010e4 <+48>:    b.le   0x40111c <phase_3+104>
0x00000000004010e8 <+52>:    cmp     w1, #0x5
0x00000000004010ec <+56>:    b.eq   0x401190 <phase_3+220> // b.none
0x00000000004010f0 <+60>:    b.lt   0x401188 <phase_3+212> // b.tstop
0x00000000004010f4 <+64>:    cmp     w1, #0x6
0x00000000004010f8 <+68>:    b.eq   0x401198 <phase_3+228> // b.none
0x00000000004010fc <+72>:    mov     w0, #0x0           // #0
0x0000000000401100 <+76>:    cmp     w1, #0x7
0x0000000000401104 <+80>:    b.eq   0x401148 <phase_3+148> // b.none
0x000000000040111c <+104>:   cmp     w1, #0x1
0x0000000000401120 <+108>:   b.eq   0x401170 <phase_3+188> // b.none
0x0000000000401124 <+112>:   b.gt   0x401178 <phase_3+196>

```

5. 其中, 当 **w1 < 0** 时, 程序首先会跳转至 **<+104>** 位置, 并运行至 **<+120>** 位置, 此时 **cbnz** 命令会判断 **w1** 是否为 **0**。如果不是, 则引爆炸弹, 故 **w1** 不能小于 **0**; 当 **w1 > 7** 时, 程序会一直运行至 **<+84>** 位置引爆炸弹。

```

0x000000000040111c <+104>:    cmp      w1, #0x1
0x0000000000401120 <+108>:    b.eq    0x401170 <phase_3+188> // b.none
0x0000000000401124 <+112>:    b.gt    0x401178 <phase_3+196>
0x0000000000401128 <+116>:    mov      w0, #0xa1           // #417
0x000000000040112c <+120>:    cbnz   w1, 0x401108 <phase_3+84>

```

6. 至此, 我们发现程序会在 **w1** 为不同值时执行不同的命令, 故此处应为一个 **switch** 语句。由于每个部分的功能都一样, 所以此处只解释 **w1 = 4** 对应的情况:

- 通过上述转跳表, 我们知道应跳转至 **<+212>** 位置, 然后程序将 **w0** 置 **0**, 并跳转至 **<+136>** 位置。

```

0x0000000000401188 <+212>:    mov      w0, #0x0           // #0
0x000000000040118c <+216>:    b       0x40113c <phase_3+136>

```

- 在 **<+136>** 位置, 程序首先进行 4 次加减法, 由于加上和减去的数一致, 因此最终 **w0** 的值仍为 **0**。

```

0x0000000000401138 <+132>:    sub     w0, w0, #0x1da
0x000000000040113c <+136>:    add     w0, w0, #0x1da
0x0000000000401140 <+140>:    sub     w0, w0, #0x1da
0x0000000000401144 <+144>:    add     w0, w0, #0x1da
0x0000000000401148 <+148>:    sub     w0, w0, #0x1da

```

- 接下来判断传入的第一个变量是否大于 **5**, 如果大于, 则引爆炸弹。
 $=> 0x000000000040114c <+152>: ldr w1, [x29, #28]$
 $0x0000000000401150 <+156>: cmp w1, #0x5$
 $0x0000000000401154 <+160>: b.gt 0x401164 <phase_3+176>$
- 继续进行, 我们发现其将 **[x29, #24]** 地址存放的值存入 **w1** 中, 使用 **print** 指令查看, 发现是我们传入的第二个变量。

```
[gdb] p $w1  
$2 = 0
```

- 然后比较 **w0** 和 **w1** 中的值是否相等，如果相等，就不会引爆炸弹。我们知道 **w0** 的值为 **0**，**w1** 是传入的第二个变量，因此第一个变量为 **4**，第二个变量为 **0**。

```
0x0000000000401158 <+164>: ldr    w1, [x29, #24]  
0x000000000040115c <+168>: cmp    w1, w0  
0x0000000000401160 <+172>: b.eq   0x401168 <phase_3+180> // b.none  
0x0000000000401164 <+176>: b1    0x401880 <explode_bomb>
```

7. 输入密码，解除第三炸弹。

```
[4 0  
Halfway there!
```

4.5 第四炸弹：递归炸弹

我愿称其为除第一炸弹以外最简单的炸弹，因为真的很容易卡 bug 来 skip。（而且和低阶炸弹基本一模一样，嘻嘻）

1. 还是先查看其汇编代码。

```
[(gdb) disas
Dump of assembler code for function phase_4:
=> 0x000000000401204 <+0>:    stp    x29, x30, [sp, #-32]!
 0x000000000401208 <+4>:    mov    x29, sp
 0x00000000040120c <+8>:    add    x3, x29, #0x1c
 0x000000000401210 <+12>:   add    x2, x29, #0x18
 0x000000000401214 <+16>:   adrp   x1, 0x402000 <submitr+1052>
 0x000000000401218 <+20>:   add    x1, x1, #0x6b0
 0x00000000040121c <+24>:   bl     0x400d70 <__isoc99_sscanf@plt>
 0x000000000401220 <+28>:   cmp    w0, #0x2
 0x000000000401224 <+32>:   b.ne   0x401238 <phase_4+52> // b.any
 0x000000000401228 <+36>:   ldr    w0, [x29, #28]
 0x00000000040122c <+40>:   sub    w0, w0, #0x2
 0x000000000401230 <+44>:   cmp    w0, #0x2
 0x000000000401234 <+48>:   b.ls   0x40123c <phase_4+56> // b.plast
 0x000000000401238 <+52>:   bl     0x401880 <explode_bomb>
 0x00000000040123c <+56>:   ldr    w1, [x29, #28]
 0x000000000401240 <+60>:   mov    w0, #0x5           // #5
 0x000000000401244 <+64>:   bl     0x4011a0 <func4>
 0x000000000401248 <+68>:   ldr    w1, [x29, #24]
 0x00000000040124c <+72>:   cmp    w1, w0
 0x000000000401250 <+76>:   b.eq   0x401258 <phase_4+84> // b.none
 0x000000000401254 <+80>:   bl     0x401880 <explode_bomb>
 0x000000000401258 <+84>:   ldp    x29, x30, [sp], #32
 0x00000000040125c <+88>:   ret
End of assembler dump.
```

2. 依然是先看 `sscanf` 传入参数用的几个寄存器，发现这次的密码是 `数字 数字` 的形式。同时由上题可知，读入的两个数字分别被放在 `[x29, #24]` 和 `[x29, #28]` 中；在 `sscanf` 返回值不等于 `2` 时爆炸。

```
[(gdb) x /s $x1
0x4026b0:      "%d %d"
```

3. 接着往下看，在 `<+36>` 位置将第二个元素拷贝到 `w0` 寄存器中，减去 `0x2` 后与 `0x2` 做比较。如果大于 `0x2`，则炸弹爆炸。所以我们输入的第二个数应该小于等于 `4`。然后将 `func4` 的参数立即数 `0x5` 和第二个元素传入 `w0` 和 `w1` 中，调用函数 `func4`。

```
0x000000000401228 <+36>:   ldr    w0, [x29, #28]
0x00000000040122c <+40>:   sub    w0, w0, #0x2
0x000000000401230 <+44>:   cmp    w0, #0x2
0x000000000401234 <+48>:   b.ls   0x40123c <phase_4+56> // b.plast
0x000000000401238 <+52>:   bl     0x401880 <explode_bomb>

[(gdb) p $w0
$1 = 3
```

4. 进入 `func4` 内部，查看其汇编代码，发现其是一个递归函数，我们逐步分析其功能。

```
0x00000000004011a0 in func4 ()
[(gdb) disas
Dump of assembler code for function func4:
=> 0x00000000004011a0 <+0>:    cmp    w0, #0x0
  0x00000000004011a4 <+4>:    b.le   0x4011fc <func4+92>
  0x00000000004011a8 <+8>:    stp    x29, x30, [sp, #-48]!
  0x00000000004011ac <+12>:   mov    x29, sp
  0x00000000004011b0 <+16>:   str    x19, [sp, #16]
  0x00000000004011b4 <+20>:   mov    w19, w0
  0x00000000004011b8 <+24>:   mov    w0, w1
  0x00000000004011bc <+28>:   cmp    w19, #0x1
  0x00000000004011c0 <+32>:   b.ne   0x4011d0 <func4+48> // b.any
  0x00000000004011c4 <+36>:   ldr    x19, [sp, #16]
  0x00000000004011c8 <+40>:   ldp    x29, x30, [sp], #48
  0x00000000004011cc <+44>:   ret
  0x00000000004011d0 <+48>:   stp    x20, x21, [x29, #24]
  0x00000000004011d4 <+52>:   mov    w20, w1
  0x00000000004011d8 <+56>:   sub    w0, w19, #0x1
  0x00000000004011dc <+60>:   bl    0x4011a0 <func4>
  0x00000000004011e0 <+64>:   add    w21, w0, w20
  0x00000000004011e4 <+68>:   mov    w1, w20
  0x00000000004011e8 <+72>:   sub    w0, w19, #0x2
  0x00000000004011ec <+76>:   bl    0x4011a0 <func4>
  0x00000000004011f0 <+80>:   add    w0, w21, w0
  0x00000000004011f4 <+84>:   ldp    x20, x21, [x29, #24]
[--Type <RET> for more, q to quit, c to continue without paging--
 0x00000000004011f8 <+88>:   b     0x4011c4 <func4+36>
  0x00000000004011fc <+92>:   mov    w0, #0x0                                // #
  0x0000000000401200 <+96>:   ret
End of assembler dump.
```

- 首先是递归出口，当 `w0` 寄存器中的值小于等于 `0` 时直接返回。注意在函数的最后将 `0x0` 赋值给了 `w0`，因此此处的返回值一定为 `0`；接着先把 `w0` 复制到 `w19` 中，再将第二个元素复制到 `w0` 中，当 `w19` 寄存器（即原来的 `w0`）中的值等于 `1` 时，返回第二个元素。

```
=> 0x00000000004011a0 <+0>:    cmp    w0, #0x0
  0x00000000004011a4 <+4>:    b.le   0x4011fc <func4+92>
  0x00000000004011a8 <+8>:    stp    x29, x30, [sp, #-48]!
  0x00000000004011ac <+12>:   mov    x29, sp
  0x00000000004011b0 <+16>:   str    x19, [sp, #16]
  0x00000000004011b4 <+20>:   mov    w19, w0
  0x00000000004011b8 <+24>:   mov    w0, w1
  0x00000000004011bc <+28>:   cmp    w19, #0x1
  0x00000000004011c0 <+32>:   b.ne   0x4011d0 <func4+48> // b.any
  0x00000000004011c4 <+36>:   ldr    x19, [sp, #16]
  0x00000000004011c8 <+40>:   ldp    x29, x30, [sp], #48
  0x00000000004011cc <+44>:   ret
  0x00000000004011fc <+92>:   mov    w0, #0x0                                // #
  0x0000000000401200 <+96>:   ret
```

- 接着将 `w1` 复制到 `w20` 中，令 `w0 = w19 - 1`（`w19` 即原来的 `w0`），调用 `func4` 函数。然后将返回值 `w0` 加上 `w20`（即第二个元素）并存储在 `w21` 中。将 `w19`（也就是 `w0` 的初值）减 `2` 的值存放在 `w0` 中，再次调用 `func4`，并将返回值加上之前的处理过的返回值 `w21`。

```

0x00000000004011d4 <+52>:    mov      w20, w1
0x00000000004011d8 <+56>:    sub      w0, w19, #0x1
0x00000000004011dc <+60>:    bl       0x4011a0 <func4>
0x00000000004011e0 <+64>:    add      w21, w0, w20
0x00000000004011e4 <+68>:    mov      w1, w20
0x00000000004011e8 <+72>:    sub      w0, w19, #0x2
0x00000000004011ec <+76>:    bl       0x4011a0 <func4>
0x00000000004011f0 <+80>:    add      w0, w21, w0

```

- 根据上面描述，我们可以将其翻译成 C 代码：

```

1 int func4(int n, int e2)
2 {
3     if (n <= 0) return 0;
4     if (n == 1) return e2;
5     return func4(n - 1, e2) + e2 + func4(n - 2, e2);
6 }
7
8 // 调用 func(5, elem2);

```

- 分析完 **func4** 的功能，我们接着往下看。程序将第一个元素与函数返回值做比较。如果不相等，炸弹爆炸。

```

0x0000000000401248 <+68>:    ldr      w1, [x29, #24]
0x000000000040124c <+72>:    cmp      w1, w0
0x0000000000401250 <+76>:    b.eq    0x401258 <phase_4+84> // b.none
0x0000000000401254 <+80>:    bl       0x401880 <explode_bomb>
0x0000000000401258 <+84>:    ldp      x29, x30, [sp], #32

```

- 综上，我们的输入必须满足 **elem1 = func4(6, elem2)**，同时第二个输入要小于等于 **4**，至此第四炸弹解除。

Halfway there!
[36 3
So you got that one. Try this one.

skip 方法：只要知道递归边界，我可以直接传边界进去，这样可以省去对递归函数的分析；或者随便试一组数据，单步执行，在 **func4** 结束后查看 **%rax** 寄存器的值就可以找到对应的答案。

4.6 第五炸弹：真·二进制炸弹

- 老样子，先查汇编。（这题也和低阶的差不多，血赚）

```
(gdb) disas
Dump of assembler code for function phase_5:
=> 0x0000000000401260 <+0>:    stp      x29, x30, [sp, #-32]!
  0x0000000000401264 <+4>:    mov      x29, sp
  0x0000000000401268 <+8>:    str      x19, [sp, #16]
  0x000000000040126c <+12>:   mov      x19, x0
  0x0000000000401270 <+16>:   bl      0x401550 <string_length>
  0x0000000000401274 <+20>:   cmp      w0, #0x6
  0x0000000000401278 <+24>:   b.ne    0x4012c0 <phase_5+96> // b.any
  0x000000000040127c <+28>:   mov      x2, x19
  0x0000000000401280 <+32>:   add      x0, x19, #0x6
  0x0000000000401284 <+36>:   mov      w3, #0x0          // #
  0x0000000000401288 <+40>:   adrp    x4, 0x402000 <submitr+1052>
  0x000000000040128c <+44>:   add      x4, x4, #0x640
  0x0000000000401290 <+48>:   ldrb    w1, [x2], #1
  0x0000000000401294 <+52>:   and     x1, x1, #0xf
  0x0000000000401298 <+56>:   ldr     w1, [x4, x1, lsl #2]
  0x000000000040129c <+60>:   add     w3, w3, w1
  0x00000000004012a0 <+64>:   cmp     x2, x0
  0x00000000004012a4 <+68>:   b.ne    0x401290 <phase_5+48> // b.any
  0x00000000004012a8 <+72>:   cmp     w3, #0x3d
  0x00000000004012ac <+76>:   b.eq    0x4012b4 <phase_5+84> // b.none
  0x00000000004012b0 <+80>:   bl     0x401880 <explode_bomb>
  0x00000000004012b4 <+84>:   ldr     x19, [sp, #16]
  0x00000000004012b8 <+88>:   ldp     x29, x30, [sp], #32
  0x00000000004012bc <+92>:   ret
  0x00000000004012c0 <+96>:   bl     0x401880 <explode_bomb>
  0x00000000004012c4 <+100>:  b      0x40127c <phase_5+28>
End of assembler dump.
```

2. 可以看到程序在 <+16> 位置调用了 `string_length` 函数，并在之后将返回值与 `0x6` 比较。说明这次的输入是一个长度为 6 的字符串。

```
0x0000000000401270 <+16>:   bl      0x401550 <string_length>
0x0000000000401274 <+20>:   cmp     w0, #0x6
0x0000000000401278 <+24>:   b.ne    0x4012c0 <phase_5+96> // b.any
```

3. 继续往下看，发现在 <+68> 位置的跳转指令会跳转回 <+48> 位置，显然这也是一个 `do while` 循环，接下来分析循环的部分。

```
0x00000000004012a0 <+64>:   cmp     x2, x0
0x00000000004012a4 <+68>:   b.ne    0x401290 <phase_5+48> // b.any
```

4. 在循环外，程序先将 **x19** **x19 + 6** 拷贝到 **x2** 和 **x0** 中，再将 **0x0** 赋值给了 **w3**，然后我们先看循环部分的汇编代码。循环开始时，将 [**x2**] 所指向地址的数据的末尾 8 位做 0 拓展传给了 **w1** 寄存器，并让 **x2** 的值加一，然后将 **x1** 与 **0xf** 做按位与的操作。我们知道，**0xf** 的二进制表示为 **1111**，因此这句话执行的结果为保留 **x1** 的最低 4 位。

```
0x000000000000401290 <+48>: ldrb    w1, [x2], #1
0x0000000000401294 <+52>: and     x1, x1, #0xf
```

5. 此外，我们注意到 **<+64>** 位置在 **x2** 的值等于 **x0**（即最初的 **x2 + 6**）时结束循环。因此 **x2** 的作用为作为指针用来遍历我们输入的字符串。

```
0x0000000000004012a0 <+64>: cmp     x2, x0
0x00000000004012a4 <+68>: b.ne   0x401290 <phase_5+48> // b.any
```

6. 接下来，和以前一样，程序将之前取出的末尾 4 位作为索引去访问 **x4** 所指向的内存空间（因为输入为 **int** 类型，所以索引要乘上 4，即 **lsl #2**）。我们回过头来看 **<+40>** 位置，发现 **x4 = 0x402640**，于是使用 **x /16d** 命令查看该地址存放的数据，发现是 16 个不重复打乱的数字。

不知道为啥，**figure*** 环境只能插 3 张图，所以这里拿炸弹凑个数。

```
0x000000000000401298 <+56>: ldr     w1, [x4, x1, lsl #2]
0x000000000040129c <+60>: add     w3, w3, w1
0x0000000000401288 <+40>: adrp    x4, 0x402000 <submitr+1052>
0x000000000040128c <+44>: add     x4, x4, #0x640
(gdb) x /16d 0x402640
0x402640 <array.4325>: 2      10      6      1
0x402650 <array.4325+16>: 12     16      9      3
0x402660 <array.4325+32>: 4      7      14     5
0x402670 <array.4325+48>: 11     8      15     13
```

7. 然后程序将取得的结果存入 **w1** 寄存器中，并将其加到 **w3**。接着判断是否结束循环。经过上述分析，可以看出这段循环的作用是将输入字符串的每一字符的最低 4 位作为索引，取出一些数字并相加存放在 **w3** 中。

```
0x0000000000401298 <+56>: ldr      w1, [x4, x1, lsl #2]
0x000000000040129c <+60>: add      w3, w3, w1
0x00000000004012a0 <+64>: cmp      x2, x0
0x00000000004012a4 <+68>: b.ne    0x401290 <phase_5+48> // b.any
```

8. 接着在 <+72> 位置将之前得到的结果与 **0x3d** 做比较，如果相等则 **SAFE**。

```
0x00000000004012a8 <+72>: cmp      w3, #0x3d
0x00000000004012ac <+76>: b.eq    0x4012b4 <phase_5+84> // b.none
0x00000000004012b0 <+80>: bl     0x401880 <explode_bomb>
```

9. 于是，这次的密码就很清晰了：通过查找 **acsii** 码确定究竟输入哪些字符可以从之前的数中依次取出某些数，使其和为 **0x3d**，这里选择 **16 15 14 13 2 1**，对应下来为 **ONJEPS**。至此 **Dr. Evil** 的阴谋又双彻底破灭，可喜可贺。

5 总结体会

- 收获：

通过这次实验，我熟悉了 **arm** 架构下汇编代码的基本语句，大大强化了自己阅读汇编代码的能力。

剩下的和低阶一样

在一开始阅读汇编代码（尤其是第二个炸弹时），只知道从头到尾线性地去阅读，结果效率十分低下。在经过这次实验后，我知道了读汇编代码应该从整体入手，掌握大体框架后再去细致地看每一部分的内容；

同时，我也掌握了实际运用课程所学知识的能力，比如各个控制语句的汇编结构，函数参数传递的底层原理和过程等。

- 建议：

建议高阶炸弹可以和普通炸弹一块布置，像这样刚好卡着期中考试的时间真的很尴尬。

虽然可以使用文件避免重复输入，但好像 **gdb** 调试时并不支持，希望下次能把这个加上。