北京郵電大學

实验报告

题目：高阶二进制炸弹拆解

班　　级：2021211301

学　　号：2021210967

姓　　名：王心雨

学　　院：计算机学院

2022 年 11 月 17 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 x86-64 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

三、实验内容

登录 bupt1 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。

1. 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

建议按照：准备工作、阶段 1、阶段 2、…等来组织内容
各阶段需要有操作步骤、运行截图、分析过程的内容

- **准备工作**

1. ls 打开文件目录，发现新炸弹压缩包为 bomb57.tar，解压文件，重新列出目录；查看 bomb.c 文件内容，找到与拆炸弹相关的函数；发现与普通版 lab2 类似，发现共有 6 个实验，其中 phase_6 为隐藏关卡；

```
    input = read_line();             /* Get input                   */
    phase_1(input);                  /* Run the phase               */
    phase_defused();                 /* Drat!  They figured it out!
                                      * Let me know how they did it. */
    printf("Phase 1 defused. How about the next one?\n");

    /* The second phase is harder.  No one will ever figure out
     * how to defuse this... */
    input = read_line();
    phase_2(input);
    phase_defused();
    printf("That's number 2.  Keep going!\n");

    /* I guess this is too easy so far.  Some more complex code will
     * confuse people. */
    input = read_line();
    phase_3(input);
    phase_defused();
    printf("Halfway there!\n");
```

```
    /* Oh yeah?  Well, how good is your math?  Try on this saucy problem! */
    input = read_line();
    phase_4(input);
    phase_defused();
    printf("So you got that one.  Try this one.\n");

    /* Round and 'round in memory we go, where we stop, the bomb blows! */
    input = read_line();
    phase_5(input);
    phase_defused();
    printf("Good work!  On to the next...\n");

    /* This phase will never be used, since no one will get past the
     * earlier ones.  But just in case, make this one extra hard. */
    input = read_line();
    phase_6(input);
    phase_defused();

    /* Wow, they got it!  But isn't something... missing?  Perhaps
     * something they overlooked?  Mua ha ha ha ha! */

    return 0;
}
```

2. 退出 bomb.c 将 bomb 文件反汇编，得到了 phase1-5 的汇编代码；我们可以看到，高阶版本与普通版本的不同之处在于汇编指令采用了 ARM 指令；所以我们的目标是尽可能读懂汇编代码，分析关键步骤；【截图略，下面各实验 disas 都显示】

- **阶段一**

1. 先打开 bomb57 文件，输入 gdb bomb57 编译器；进入首先在 exlode_bomb 函数处打断点，保证炸弹不炸开，类似于普通版 lab2 实验，再次在 phase_1 处打断点，运行；使用 disas 查看 phase_1 函数的汇编代码；
2. 根据 string_not_equal 函数，推测阶段一是输入还是一个字符串；可知现在程序已经运行到了 strings_not_equal 的函数，我们只需查看此时 x1 寄存器的值，即可直到传进子函数的参数；输入 print $x1 指令，查看此时地址内容为字符串：“Only you can give me that feeling.”

```
(gdb) b explode_bomb
Breakpoint 1 at 0x401878
(gdb) b phase_1
Breakpoint 2 at 0x401038
(gdb) r
Starting program: /students/2021210967/bomb57/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
test

Breakpoint 2, 0x0000000000401038 in phase_1 ()
(gdb) disas phase_1
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+1076>
   0x0000000000401034 <+12>:    add     x1, x1, #0x638
=> 0x0000000000401038 <+16>:    bl      0x401564 <strings_not_equal>
   0x000000000040103c <+20>:    cbnz    w0, 0x401048 <phase_1+32>
   0x0000000000401040 <+24>:    ldp     x29, x30, [sp], #16
   0x0000000000401044 <+28>:    ret
   0x0000000000401048 <+32>:    bl      0x401868 <explode_bomb>
   0x000000000040104c <+36>:    b       0x401040 <phase_1+24>
End of assembler dump.
(gdb)
0x402638:       "Only you can give me that feeling."
(gdb)
```

3. 退出 gdb 重新输入断点，此时断点打到 phase_2，运行输入；显示阶段 1 已经解决，成功；

```
(gdb) b explode_bomb
Breakpoint 3 at 0x401878
(gdb) b phase_2
Breakpoint 4 at 0x401060
(gdb) r
Starting program: /students/2021210967/bomb57/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Only you can give me that feeling.
Phase 1 defused. How about the next one?
test
```

- **阶段三**

1. 类似以前的实验，我们快速打断点，disas 反汇编阶段 2 的函数；此时我们可以看到函数内部调用了 read_six_numbers 的函数，可以猜测输入的数为 6 个整数，类似于 lab2 的实验二；还可以看出，函数内部有一个次数为 6 的循环，从<phase_2+52>的指令开始；
2. 我们还可知，根据<+64>、<+68>、<+72>的指令，可知，w1 为此时比较的第 i 个数，而 w0 为比较的第 i+1 个数；如果 w0 与 w1 的左移 1 位（LSL——2 倍）相当，即 a[i+1]=2a[i]，则成功跳过 explode_bomb 函数，所以可知为等比数列；
3. 再根据第<+24>的 cmp 指令，我们知道第一个输入的数字为 1，所以最终输入数列为 1 2 4 8 16 32；

```
(gdb) disas phase_2
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      0x4018a4 <read_six_numbers>
   0x0000000000401064 <+20>:    ldr     w0, [x29, #40]
   0x0000000000401068 <+24>:    cmp     w0, #0x1
   0x000000000040106c <+28>:    b.ne    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      0x401868 <explode_bomb>
   0x0000000000401080 <+48>:    b       0x401070 <phase_2+32>
   0x0000000000401084 <+52>:    add     x19, x19, #0x4
   0x0000000000401088 <+56>:    cmp     x19, x20
   0x000000000040108c <+60>:    b.eq    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>:    b.eq    0x401084 <phase_2+52>  // b.none
   0x00000000004010a0 <+80>:    bl      0x401868 <explode_bomb>
   0x00000000004010a4 <+84>:    b       0x401084 <phase_2+52>
   0x00000000004010a8 <+88>:    ldp     x19, x20, [sp, #16]
   0x00000000004010ac <+92>:    ldp     x29, x30, [sp], #64
   0x00000000004010b0 <+96>:    ret
End of assembler dump.
(gdb)
```

4. 重新进入 gdb，输入答案 2，成功。

```
(gdb) b explode_bomb
Breakpoint 1 at 0x401878
(gdb) b phase_3
Breakpoint 2 at 0x4010cc
(gdb) r
Starting program: /students/2021210967/bomb57/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Only you can give me that feeling.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
test

Breakpoint 2, 0x00000000004010cc in phase_3 ()
```

- **阶段三**

1. 利用 disas 反汇编查看 phase_3 的汇编代码；当前运行到了 sscanf 函数，我们此时查看 x1 寄存内的内容，输入指令，x/s $x1，得到输入的格式为“%d %d”即两个整数；

```
(gdb) disas phase_3
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+1076>
```

```
(gdb) x/s $x1
0x402660:       "%d %d"
(gdb)
```

2. 此时分析第一个 cmp 指令，经过 sscanf 函数的返回值 w0 应为输入的整数的个数，如果小于等于 1 则触发炸弹；再看第二个 cmp，将 w1 即第一个输入的值与 3 进行比较，如果相等，跳转到<phase_4+204>，为将 w0 赋值为 0，接下来绝对跳转到<phase_4+132>；

```
=> 0x00000000004010cc <+24>:    bl      0x400d70 <__isoc99_sscanf@plt>
   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                        // #0
   0x0000000000401100 <+76>:    cmp     w1, #0x7
   0x0000000000401104 <+80>:    b.eq    0x401148 <phase_3+148>  // b.none
   0x0000000000401108 <+84>:    bl      0x401868 <explode_bomb>
   0x000000000040110c <+88>:    mov     w0, #0x0                        // #0
   0x0000000000401110 <+92>:    b       0x40114c <phase_3+152>
   0x0000000000401114 <+96>:    bl      0x401868 <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, #0x25e                      // #606
   0x000000000040112c <+120>:   cbnz    w1, 0x401108 <phase_3+84>
   0x0000000000401130 <+124>:   sub     w0, w0, #0x44
   0x0000000000401134 <+128>:   add     w0, w0, #0x3d8
```

3. <phase_4+132>——<phase_4+148>为对 w0 的加减操作，经过五次-+-+-，可得此时 w0 的值为-0x47=-71;此时看<phase_4+168>的 cmp 指令，将 w1+24 位的指令即第二个输入的数进行比较，如果相等，跳过 explode_bomb 函数，最终结果为 3 -71；

```
--Type <RET> for more, q to quit, c to continue without paging--c
   0x0000000000401138 <+132>:   sub     w0, w0, #0x47
   0x000000000040113c <+136>:   add     w0, w0, #0x47
   0x0000000000401140 <+140>:   sub     w0, w0, #0x47
   0x0000000000401144 <+144>:   add     w0, w0, #0x47
   0x0000000000401148 <+148>:   sub     w0, w0, #0x47
   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      0x401868 <explode_bomb>
   0x0000000000401168 <+180>:   ldp     x29, x30, [sp], #32
   0x000000000040116c <+184>:   ret
```

```
   0x0000000000401180 <+204>:   mov     w0, #0x0                        // #0
   0x0000000000401184 <+208>:   b       0x401138 <phase_3+132>
```

4. 重新进入 gdb，输入阶段三的结果，正确，通过；

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) b explode_bomb
Breakpoint 1 at 0x401878
(gdb) b phase_4
Breakpoint 2 at 0x40120c
(gdb) r
Starting program: /students/2021210967/bomb57/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Only you can give me that feeling.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
3 -71
Halfway there!
```

- **阶段四**

1. Disas 反汇编 phase_4，可知此时运行到了 sscanf 函数，此时类似于阶段 3，查看 x1 的地址内容，依旧是“%d %d”，证明阶段 4 我们仍旧要输入两个整数，

```
(gdb) disas phase_4
Dump of assembler code for function phase_4:
   0x00000000004011f4 <+0>:     stp     x29, x30, [sp, #-32]!
   0x00000000004011f8 <+4>:     mov     x29, sp
   0x00000000004011fc <+8>:     add     x3, x29, #0x18
   0x0000000000401200 <+12>:    add     x2, x29, #0x1c
   0x0000000000401204 <+16>:    adrp    x1, 0x402000 <submitr+1076>
   0x0000000000401208 <+20>:    add     x1, x1, #0x660
=> 0x000000000040120c <+24>:    bl      0x400d70 <__isoc99_sscanf@plt>
   0x0000000000401210 <+28>:    cmp     w0, #0x2
   0x0000000000401214 <+32>:    b.ne    0x401224 <phase_4+48>  // b.any
   0x0000000000401218 <+36>:    ldr     w0, [x29, #28]
   0x000000000040121c <+40>:    cmp     w0, #0xe
   0x0000000000401220 <+44>:    b.ls    0x401228 <phase_4+52>  // b.plast
   0x0000000000401224 <+48>:    bl      0x401868 <explode_bomb>
   0x0000000000401228 <+52>:    mov     w2, #0xe                        // #14
   0x000000000040122c <+56>:    mov     w1, #0x0                        // #0
   0x0000000000401230 <+60>:    ldr     w0, [x29, #28]
   0x0000000000401234 <+64>:    bl      0x4011a0 <func4>
   0x0000000000401238 <+68>:    cmp     w0, #0x1
   0x000000000040123c <+72>:    b.ne    0x40124c <phase_4+88>  // b.any
   0x0000000000401240 <+76>:    ldr     w0, [x29, #24]
   0x0000000000401244 <+80>:    cmp     w0, #0x1
   0x0000000000401248 <+84>:    b.eq    0x401250 <phase_4+92>  // b.none
   0x000000000040124c <+88>:    bl      0x401868 <explode_bomb>
   0x0000000000401250 <+92>:    ldp     x29, x30, [sp], #32
   0x0000000000401254 <+96>:    ret
End of assembler dump.
```

```
(gdb) x/s $x1
0x402660:       "%d %d"
```

2. 对 func4 进行反汇编，可以类比 lab2 实验，可知其函数内仍然存在循环，并且根据 func4 所传的参数，最终能得到第二个与参数 1 相关的参数；在 func4 被调用后的，<phase_4+68>指令，可以看到将 w0 函数返回值与 1 进行比较，所以我们可以判定第二个输入的数为 1；

```
(gdb) disas func4
Dump of assembler code for function func4:
   0x00000000004011a0 <+0>:     stp     x29, x30, [sp, #-16]!
   0x00000000004011a4 <+4>:     mov     x29, sp
   0x00000000004011a8 <+8>:     sub     w3, w2, w1
   0x00000000004011ac <+12>:    add     w3, w3, w3, lsr #31
   0x00000000004011b0 <+16>:    add     w3, w1, w3, asr #1
   0x00000000004011b4 <+20>:    cmp     w3, w0
   0x00000000004011b8 <+24>:    b.gt    0x4011d0 <func4+48>
   0x00000000004011bc <+28>:    mov     w1, #0x0                        // #0
   0x00000000004011c0 <+32>:    b.lt    0x4011e0 <func4+64>  // b.tstop
   0x00000000004011c4 <+36>:    mov     w0, w1
   0x00000000004011c8 <+40>:    ldp     x29, x30, [sp], #16
   0x00000000004011cc <+44>:    ret
   0x00000000004011d0 <+48>:    sub     w2, w3, #0x1
   0x00000000004011d4 <+52>:    bl      0x4011a0 <func4>
   0x00000000004011d8 <+56>:    lsl     w1, w0, #1
   0x00000000004011dc <+60>:    b       0x4011c4 <func4+36>
   0x00000000004011e0 <+64>:    add     w1, w3, #0x1
   0x00000000004011e4 <+68>:    bl      0x4011a0 <func4>
   0x00000000004011e8 <+72>:    lsl     w0, w0, #1
   0x00000000004011ec <+76>:    add     w1, w0, #0x1
   0x00000000004011f0 <+80>:    b       0x4011c4 <func4+36>
End of assembler dump.
(gdb)
```

3. 并且根据汇编代码，我们可以知道第一个参数必须小于等于 14，否则触发 bomb 函数；因此我们从 0 开始尝试：打断点到 explode_bomb 防止爆炸，打断点到+68 指令，查看此时的 x0 寄存器的值是（即 func4 函数的返回值）否等于第二个输入值 1，如果等于，证明寻找到了答案；
4. 从输入 1 1，2 1，3 1…开始尝试；尝试结束即 kill 掉，del 删掉断点，重新打断点运行；得到返回值分别为：1 1 0；2 1 4；3 1 0；4 1 2；5 1 2；6 1 6；7 1 0；8 1 1，得到 8 为第一个解；

```
(gdb) b explode_bomb
Breakpoint 1 at 0x401878
(gdb) b *0x401238
Breakpoint 2 at 0x401238
(gdb) r
Starting program: /students/2021210967/bomb57/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
```

```
Halfway there!
1 1

Breakpoint 2, 0x0000000000401238 in phase_4 ()
(gdb) info reg
x0             0x0                 0
```

```
Halfway there!
2 1

Breakpoint 2, 0x0000000000401238 in phase_4 ()
(gdb) info reg
x0             0x4                 4
x1             0x4                 4
```

```
Halfway there!
3 1

Breakpoint 2, 0x0000000000401238 in phase_4 ()
(gdb) info reg
x0             0x0                 0
```

```
Halfway there!
4 1

Breakpoint 2, 0x0000000000401238 in phase_4 ()
(gdb) info reg
x0             0x2                 2
x1             0x2                 2
```

```
Halfway there!
5 1

Breakpoint 2, 0x0000000000401238 in phase_4 ()
(gdb) info reg
x0             0x2                 2
```

```
Halfway there!
7 1

Breakpoint 2, 0x0000000000401238 in phase_4 ()
(gdb) info reg
x0             0x0                 0
```

```
Halfway there!
8 1

Breakpoint 2, 0x0000000000401238 in phase_4 ()
(gdb) info reg
x0             0x1                 1
```

5. 输入 8 1，成功通过，阶段 4 完成；

```
(gdb) b explode_bomb
Breakpoint 1 at 0x401878
(gdb) b phase_5
Breakpoint 2 at 0x401268
(gdb) r
Starting program: /students/2021210967/bomb57/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Only you can give me that feeling.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
3 -71
Halfway there!
8 1
So you got that one.  Try this one.
```

- **阶段 5**

1. disas 反汇编 phase_5 函数，此时运行到了 bl 跳转指令，在 string_length 函数，如果返回值等于 6，

则正确，否则触发 bomb 函数；因此，我们需要输入的是 6 个字符；

```
(gdb) disas phase_5
Dump of assembler code for function phase_5:
   0x0000000000401258 <+0>:     stp     x29, x30, [sp, #-48]!
   0x000000000040125c <+4>:     mov     x29, sp
   0x0000000000401260 <+8>:     str     x19, [sp, #16]
   0x0000000000401264 <+12>:    mov     x19, x0
=> 0x0000000000401268 <+16>:    bl      0x401538 <string_length>
   0x000000000040126c <+20>:    cmp     w0, #0x6
   0x0000000000401270 <+24>:    b.ne    0x4012c4 <phase_5+108>  // b.any
   0x0000000000401274 <+28>:    mov     x0, #0x0                        // #0
   0x0000000000401278 <+32>:    add     x3, x29, #0x28
   0x000000000040127c <+36>:    adrp    x2, 0x402000 <submitr+1076>
   0x0000000000401280 <+40>:    add     x2, x2, #0x628
   0x0000000000401284 <+44>:    ldrb    w1, [x19, x0]
   0x0000000000401288 <+48>:    and     w1, w1, #0xf
   0x000000000040128c <+52>:    ldrb    w1, [x2, w1, sxtw]
   0x0000000000401290 <+56>:    strb    w1, [x0, x3]
   0x0000000000401294 <+60>:    add     x0, x0, #0x1
   0x0000000000401298 <+64>:    cmp     x0, #0x6
   0x000000000040129c <+68>:    b.ne    0x401284 <phase_5+44>  // b.any
   0x00000000004012a0 <+72>:    strb    wzr, [x29, #46]
   0x00000000004012a4 <+76>:    adrp    x1, 0x402000 <submitr+1076>
   0x00000000004012a8 <+80>:    add     x1, x1, #0x668
   0x00000000004012ac <+84>:    add     x0, x29, #0x28
   0x00000000004012b0 <+88>:    bl      0x401564 <strings_not_equal>
   0x00000000004012b4 <+92>:    cbnz    w0, 0x4012cc <phase_5+116>
   0x00000000004012b8 <+96>:    ldr     x19, [sp, #16]
   0x00000000004012bc <+100>:   ldp     x29, x30, [sp], #48
   0x00000000004012c0 <+104>:   ret
   0x00000000004012c4 <+108>:   bl      0x401868 <explode_bomb>
   0x00000000004012c8 <+112>:   b       0x401274 <phase_5+28>
   0x00000000004012cc <+116>:   bl      0x401868 <explode_bomb>
   0x00000000004012d0 <+120>:   b       0x4012b8 <phase_5+96>
End of assembler dump.
(gdb)
```

2. 看得出下面几行指令为循环内的内容，可以知道，w1 取低四位，然后以 x2 内存为寻址基址对 w1 内容进行了扩展移位操作，基址偏移量为 x3；

```
0x0000000000401284 <+44>:    ldrb    w1, [x19, x0]
0x0000000000401288 <+48>:    and     w1, w1, #0xf
0x000000000040128c <+52>:    ldrb    w1, [x2, w1, sxtw]
0x0000000000401290 <+56>:    strb    w1, [x0, x3]
0x0000000000401294 <+60>:    add     x0, x0, #0x1
0x0000000000401298 <+64>:    cmp     x0, #0x6
0x000000000040129c <+68>:    b.ne    0x401284 <phase_5+44>  // b.any
```

3. 此时我们退出，在进入 gdb 打断点在<+84>指令，类似于阶段 1，查看最终比对的六个字符的字符串是什么，任意输入一个字符串，abcdef；

```
(gdb) b explode_bomb
Breakpoint 3 at 0x401878
(gdb) b *0x4012ac
Breakpoint 4 at 0x4012ac
(gdb) r
Starting program: /students/2021210967/bomb57/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Only you can give me that feeling.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
3 -71
Halfway there!
8 1
So you got that one.  Try this one.
abcdef
```

4. 此时查看 x1 寄存器中存储的字符串，为 oilers；由此我们得到原字符串取低四位之后然后偏移寻址的结果；

```
(gdb) print $x1
$1 = 4204136
(gdb) x/s 4204136
0x402668:       "oilers"
(gdb)
```

5. 我们再次退出进入 gdb，打断点打在<+44>处，再次查看 x2 寄存器的内容，即我们需要寻址的字符串数组，可以得到 oilers 六个字符的基址偏移量为 10 4 15 5 6 7；则根据 ASCII 码，我们可以得到一个对应的 ASCII 码串为 106 100 111 101 102 103；对应的字符串为 jdoefg；

```
(gdb) b explode_bomb
Breakpoint 1 at 0x401878
(gdb) b *0x401284
Breakpoint 2 at 0x401284
(gdb) r
Starting program: /students/2021210967/bomb57/bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Only you can give me that feeling.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
3 -71
Halfway there!
8 1
So you got that one.  Try this one.
oilers

Breakpoint 2, 0x0000000000401284 in phase_5 ()
(gdb) print $x2
$1 = 4204072
(gdb) x/s 4204072
0x402628 <array.4327>:  "maduiersnfotvbylOnly you can give me that feeling."
(gdb) x/16c 4204072
0x402628 <array.4327>:  109 'm' 97 'a'  100 'd' 117 'u' 105 'i' 101 'e' 114 'r' 115 's'
0x402630 <array.4327+8>:        110 'n' 102 'f' 111 'o' 116 't' 118 'v' 98 'b'  121 'y' 108 'l'
(gdb)
```

6. 尝试运行发现正确，阶段 5 完成。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Only you can give me that feeling.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2.  Keep going!
3 -71
Halfway there!
8 1
So you got that one.  Try this one.
jdoefg
Good work!  On to the next...
```

五、总结体会

总结心得（包括实验过程中遇到的问题、如何解决的、过关或挫败的感受、实验投入的时间和精力、意见和建议等）

做 ARM 版的拆解二进制炸弹实验，我觉得是相当于重新理解了一遍 lab2 的过程；lab2 做的过程中心惊胆战，不敢随意动态调试，到最后做的两个实验的时候才知道可以 b explode_bomb 防止炸掉；而且对炸弹的调用以及具体的输入机制都没有太完全的理解。ARM 版 bomb 实验首先一个最大的挑战是读取 ARM 的指令，但是确实根据老师 ppt 和 x86 类似的汇编代码对比来理解会轻松很多，所以在读取 ARM 汇编指令这方面问题不算特别大；其余的问题就是，在做的过程当中，已经有了 lab2 的铺垫，所以做起来会得心应手一些，这次也尝试了退出 gdb，打不同断点，运行猜测结果的过程，尤其是第四个实验，我运行了 11 次，真的有点锻炼心理素质。实验五可能对我来说还是比较困难，好在通过了 ARM 版的，重新理解了一遍基址偏移量、寻址的汇编语言等等。具体的每个题目考察的内容在 lab2 已经提及，考的原理、题目都是类似的，大同小异；总之，华为鲲鹏实验在一定意义上是对 lab2 的一个总结吧，真的很喜欢 csapp 实验，闯关成功，good work 真的很开心！然后做这个实验还能锻炼耐力，感觉自己理解计算机内容已经更加专注和高效啦！