

# 北京郵電大學

## 实验报告



题目: 拆解二进制炸弹

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 10 月 30 日

## 目录

## 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 **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；

## 2 实验环境

- 系统: **Linux Ubuntu 20.04.5**
- 软件工具: **macOS Terminal(10.120.11.12)**、**gcc 7.5.0**、**GNU gdb 9.2**、**GNU objdump 2.34**

## 3 实验内容

- 登录 **bupt1** 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 **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。

## 4 实验步骤及实验分析

### 4.1 准备工作

1. 登陆 **bupt1** 服务器，在自己的目录下找到 **bomb533.tar** 文件，使用 **tar -xvf bomb533.tar** 命令解压缩该文件。

```
[2022212408@bupt1:~$ tar -xvf bomb533.tar
bomb533/README
bomb533/bomb.c
bomb533/bomb
2022212408@bupt1:~$ ]
```

2. 使用 **cd bomb533** 命令移动到 **bomb533** 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。

```
[2022212408@bupt1:~$ cd bomb533
[2022212408@bupt1:~/bomb533$ ls
bomb  bomb.c  README  sol.txt
2022212408@bupt1:~/bomb533$ ]
```

PS：存在 **sol.txt** 文件是因为这部分拆炸弹前忘了截图，拆完回来补截图的。

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。

```
[2022212408@bupt1:~/bomb533$ gdb bomb
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 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) ]
```

### 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
73     input = read_line();          /* Get input           */
74     phase_1(input);              /* Run the phase       */
75     phase_defused();            /* Drat! They figured it out!
                                 * Let me know how they did it. */
76     printf("Phase 1 defused. How about the next one?\n");
77
78     /* The second phase is harder. No one will ever figure out
79     * how to defuse this... */
80     input = read_line();
81     phase_2(input);
82
[(gdb) list
83     phase_defused();
84     printf("That's number 2. Keep going!\n");
85
86     /* I guess this is too easy so far. Some more complex code will
87     * confuse people. */
88     input = read_line();
89     phase_3(input);
90     phase_defused();
91     printf("Halfway there!\n");
92
[(gdb) break 74
Breakpoint 1 at 0x400e8d: file bomb.c, line 74.
(gdb) ]
```

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) run
Starting program: /students/2022212408/bomb533/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 **disassemble** 命令查看当前汇编代码，并使用 **stepi** 命令单步执行，进入 **phase\_1(input)** 函数内部进行拆弹。

```
=> 0x0000000000400e8d <+151>:    mov    %rax,%rdi
0x0000000000400e90 <+154>:    callq  0x400f2d <phase_1>
0x0000000000400e95 <+159>:    callq  0x4018e8 <phase_defused>
[(gdb) stepi
0x0000000000400e90      74        phase_1(input);           /* Run the phase
[(gdb) stepi
0x0000000000400f2d 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 <+9> 位置，将 **\$0x402720** 移动到了 **%rsi** 寄存器中，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 <+16> 位置判断函数返回值是否为 0。如果是，则转跳到 <+23> 位置；否则继续执行，引爆炸弹，**BOMB!!!**

```
[gdb] disas
Dump of assembler code for function phase_1:
=> 0x0000000000400f2d <+0>:    sub    $0x8,%rsp
    0x0000000000400f31 <+4>:    mov    $0x402720,%esi
    0x0000000000400f36 <+9>:    callq  0x401479 <strings_not_equal>
    0x0000000000400f3b <+14>:   test   %eax,%eax
    0x0000000000400f3d <+16>:   je     0x400f44 <phase_1+23>
    0x0000000000400f3f <+18>:   callq  0x40174d <explode_bomb>
    0x0000000000400f44 <+23>:   add    $0x8,%rsp
    0x0000000000400f48 <+27>:   retq
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 <+9> 位置，然后使用 **x /s** 命令查看 **%esi** 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（我第一次拆的时候还没发现，血亏）。但图都截过了，那就当我们不知道这件事，继续看看 **strings\_not\_equal** 里面究竟有什么吧。

```
[gdb] stepi 2
0x0000000000400f36 in phase_1 ()
[gdb] x /s $esi
0x402720:      "Your job for this lab is to defuse your bomb."
(gdb) █
```

6. 继续使用 **stepi** 命令，进入到 **string\_not\_equal** 函数中，使用 **disassemble** 命令查看汇编代码。

```
[gdb] stepi 2
0x0000000000400f36 in phase_1 ()
[gdb] stepi
0x0000000000401479 in strings_not_equal ()
(gdb) █

Dump of assembler code for function strings_not_equal:
0x0000000000401479 <+0>:    push   %r12
0x000000000040147b <+2>:    push   %rbp
0x000000000040147c <+3>:    push   %rbx
0x000000000040147d <+4>:    mov    %rdi,%rbx
0x0000000000401480 <+7>:    mov    %rsi,%rbp
0x0000000000401483 <+10>:   callq  0x40145b <string_length>
0x0000000000401488 <+15>:   mov    %eax,%r12d
0x000000000040148b <+18>:   mov    %rbp,%rdi
=> 0x000000000040148e <+21>:   callq  0x40145b <string_length>
0x0000000000401493 <+26>:   mov    $0x1,%edx
0x0000000000401498 <+31>:   cmp    %eax,%r12d
0x000000000040149b <+34>:   jne    0x4014d9 <strings_not_equal+96>
```

7. 阅读汇编代码，我们发现函数调用了两次 `string_length` 函数，在第一次调用后将返回值复制至 `%r12d` 寄存器中，最后将 `%r12d` 和 `%rax` 寄存器中的内容相比较，即比较两个字符串的长度是否一致。如果不一致，则跳转到 `<+96>` 位置。可以看到，在 `<+96>` 位置将 `%edx` 寄存器的内容复制到 `%eax` 寄存器作为返回值，而在 `<+26>` 位置我们知道 `%edx` 存放的值为 `1`。所以如果两个字符串长度不等则返回 `1`，炸弹爆炸，所以两个 `string_length` 中必然有一个是密码。

```
0x00000000004014d9 <+96>:    mov    %edx,%eax
0x00000000004014db <+98>:    pop    %rbx
0x00000000004014dc <+99>:    pop    %rbp
0x00000000004014dd <+100>:   pop    %r12
0x00000000004014df <+102>:   retq
End of assembler dump.
```

8. 使用 `stepi` 指令分别移动到两处调用位置，分别查看 `%rdi` 寄存器中的内容，即可得出密码。

```
[gdb] x /s $rdi
0x402720:      "Your job for this lab is to defuse your bomb."
```

9. 重新运行程序，输入密码，第一炸弹被拆除。

```
which to blow yourself up. Have a nice day!
[Your job for this lab is to defuse your bomb.
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:
=> 0x0000000000400f49 <+0>:    push    %rbp
    0x0000000000400f4a <+1>:    push    %rbx
    0x0000000000400f4b <+2>:    sub     $0x28,%rsp
    0x0000000000400f4f <+6>:    mov     %fs:0x28,%rax
    0x0000000000400f58 <+15>:   mov     %rax,0x18(%rsp)
    0x0000000000400f5d <+20>:   xor     %eax,%eax
    0x0000000000400f5f <+22>:   mov     %rsp,%rsi
    0x0000000000400f62 <+25>:   callq   0x401783 <read_six_numbers>
    0x0000000000400f67 <+30>:   cmpl   $0x0,(%rsp)
    0x0000000000400f6b <+34>:   jne    0x400f74 <phase_2+43>
    0x0000000000400f6d <+36>:   cmpl   $0x1,0x4(%rsp)
    0x0000000000400f72 <+41>:   je     0x400f79 <phase_2+48>
    0x0000000000400f74 <+43>:   callq   0x40174d <explode_bomb>
    0x0000000000400f79 <+48>:   mov     %rsp,%rbx
    0x0000000000400f7c <+51>:   lea     0x10(%rsp),%rbp
    0x0000000000400f81 <+56>:   mov     0x4(%rbx),%eax
    0x0000000000400f84 <+59>:   add     (%rbx),%eax
    0x0000000000400f86 <+61>:   cmp     %eax,0x8(%rbx)
    0x0000000000400f89 <+64>:   je     0x400f90 <phase_2+71>
    0x0000000000400f8b <+66>:   callq   0x40174d <explode_bomb>
    0x0000000000400f90 <+71>:   add     $0x4,%rbx
    0x0000000000400f94 <+75>:   cmp     %rbp,%rbx
    0x0000000000400f97 <+78>:   jne    0x400f81 <phase_2+56>
    0x0000000000400f99 <+80>:   mov     0x18(%rsp),%rax
    0x0000000000400f9e <+85>:   xor     %fs:0x28,%rax
    0x0000000000400fa7 <+94>:   je     0x400fae <phase_2+101>
    0x0000000000400fa9 <+96>:   callq   0x400b90 <__stack_chk_fail@plt>
    0x0000000000400fae <+101>:  add     $0x28,%rsp
    0x0000000000400fb2 <+105>:  pop    %rbx
    0x0000000000400fb3 <+106>:  pop    %rbp
    0x0000000000400fb4 <+107>:  retq
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 `%eax` 寄存器的内容（`sscanf` 的返回值）与 5 做比较，并在判断为否时引爆炸弹。

```

0x0000000000401783 in read_six_numbers ()
(gdb) disas
Dump of assembler code for function read_six_numbers:
[=] 0x0000000000401783 <+0>:    sub    $0x8,%rsp
    0x0000000000401787 <+4>:    mov    %rsi,%rdx
    0x000000000040178a <+7>:    lea    0x4(%rsi),%rcx
    0x000000000040178e <+11>:   lea    0x14(%rsi),%rax
    0x0000000000401792 <+15>:   push   %rax
    0x0000000000401793 <+16>:   lea    0x10(%rsi),%rax
    0x0000000000401797 <+20>:   push   %rax
    0x0000000000401798 <+21>:   lea    0xc(%rsi),%r9
    0x000000000040179c <+25>:   lea    0x8(%rsi),%r8
    0x00000000004017a0 <+29>:   mov    $0x402a21,%esi
    0x00000000004017a5 <+34>:   mov    $0x0,%eax
    0x00000000004017aa <+39>:   callq 0x400c40 <__isoc99_sscanf@plt>
    0x00000000004017af <+44>:   add    $0x10,%rsp
    0x00000000004017b3 <+48>:   cmp    $0x5,%eax
    0x00000000004017b6 <+51>:   jg    0x4017bd <read_six_numbers+58>
    0x00000000004017b8 <+53>:   callq 0x40174d <explode_bomb>
    0x00000000004017bd <+58>:   add    $0x8,%rsp
[ 0x00000000004017c1 <+62>:   retq
End of assembler dump.
(gdb) █

```

4. 我们移动至 +34 位置，使用 `x /s` 命令查看 `sscanf` 的第一个参数值 `%esi`，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 $esi
0x402a21:      "%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：

- 读入完成后紧接着是两个判断，分别把 **0** 和 **1** 与 (`%rsp`) 和 `0x4(%rsp)` (这里是将 `%rsp` 所指向地址存放的值) 做比较，若为否则引爆炸弹，推测 `%rsp` 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 /6` 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）。

所以这两个比较说明我们输入的第一和第二个数为 **0** 和 **1**。

```

0x0000000000400f67 <+30>:    cmpl   $0x0,(%rsp)
0x0000000000400f6b <+34>:    jne    0x400f74 <phase_2+43>
0x0000000000400f6d <+36>:    cmpl   $0x1,0x4(%rsp)
0x0000000000400f72 <+41>:    je     0x400f79 <phase_2+48>
0x0000000000400f74 <+43>:    callq 0x40174d <explode_bomb>
0x0000000000400f79 <+48>:    mov    %rsp,%rbx
(gdb) x /6 $rsp
0x7ffffffffea70: 0          1          1          2
0x7ffffffffea80: 3          5
(gdb) █

```

- 接下来我们先看到 <+75> 位置，在 `%rbp` 和 `%rbx` 内容不相等时转跳回 <+56> 位置，所以推测这是一个 `do while` 循环，用来判断输入的后四个

数是否满足要求。

```
0x0000000000400f94 <+75>:    cmp    %rbp,%rbx
0x0000000000400f97 <+78>:    jne    0x400f81 <phase_2+56>
```

- 现在回过头来看之后的汇编代码，我们先把 **数组** 指针 **%rsp** 复制到 **%rbx** 中（即现在 **%rbx** 也是数组指针），然后将数组的第四个元素复制到 **%rbp** 寄存器中。接下来执行至 **<+56>** 位置，进入循环。

```
0x0000000000400f79 <+48>:    mov    %rsp,%rbx
0x0000000000400f7c <+51>:    lea    0x10(%rsp),%rbp
0x0000000000400f81 <+56>:    mov    0x4(%rbx),%eax
```

- 前两行的操作是将 **%rbx** 指向的数组中的元素和下一个元素相加并存储到 **%eax** 寄存器中，接着判断相加得到的结果与数组的下下个元素是否相等。如果不相等，则引爆炸弹。由于第一次迭代时 **%rbx** 指向数组第一个元素，因此数组的第三个元素应该是第一和第二个元素的和。最后，将 **0x4** 加到 **%rbx** 中，即将 **%rbx** 指向数组的第二个元素，判断 **%rbx** 是否等于 **%rbp** (**0x10%rsp**)。如果不等，则继续循环。

```
0x0000000000400f81 <+56>:    mov    0x4(%rbx),%eax
0x0000000000400f84 <+59>:    add    (%rbx),%eax
0x0000000000400f86 <+61>:    cmp    %eax,0x8(%rbx)
0x0000000000400f89 <+64>:    je     0x400f90 <phase_2+71>
0x0000000000400f8b <+66>:    callq 0x40174d <explode_bomb>
0x0000000000400f90 <+71>:    add    $0x4,%rbx
```

- 通过分析，我们了解这段汇编代码实现的功能是判断数组的第 **k** 个元素与第 **k + 1** 个元素相加的结果是否等于第 **k + 2** 个元素。如果不等，则引爆炸弹。直到 **k = 5** 时停止迭代，因此，所求密码是前两项为 **0、1**，长度为 **6** 的斐波那契数列。即 **0 1 1 2 3 5**。

## 7. 重新运行输入密码，成功拆除第二炸弹。

```
which to blow yourself up. Have a nice day!
[Your job for this lab is to defuse your bomb.
Phase 1 defused. How about the next one?
[0 1 1 2 3 5
That's number 2. Keep going!
[test
```

## 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:
=> 0x0000000000400f5 <>0:    sub    $0x28,%rsp
 0x0000000000400fb9 <+4>:   mov    %fs:0x28,%rax
 0x0000000000400fc2 <+13>:  mov    %rax,%rax(%rsp)
 0x0000000000400fc7 <+18>:  xor    %eax,%eax
 0x0000000000400fc9 <+20>:  lea    0x14(%rsp),%r8
 0x0000000000400fd0 <+25>:  lea    0xf(%rsp),%rcx
 0x0000000000400fd3 <+30>:  lea    0x10(%rsp),%rdx
 0x0000000000400fd8 <+35>:  mov    $0x402776,%esi
 0x0000000000400fe2 <+40>:  callq 0x400c40 <__isoc99_sscanf@plt>
 0x0000000000400fe2 <+45>:  cmp    $0x2,%eax
 0x0000000000400fe5 <+48>:  jg    0x400fec <phase_3+55>
 0x0000000000400fe7 <+50>:  callq 0x40174d <explode_bomb>
 0x0000000000400fe7 <+55>:  cmpl   $0x7,0x10(%rsp)
 0x0000000000400ff1 <+60>:  ja    0x401bf3 <phase_3+318>
 0x0000000000400ff7 <+66>:  mov    0x18(%rsp),%eax
 0x0000000000400ffb <+70>:  jmpq   *0x027990,%rax,8
 0x0000000000401082 <+77>:  mov    $0x74,%eax
 0x0000000000401087 <+82>:  cmpl   $0x3de,0x14(%rsp)
 0x000000000040108f <+90>:  je    0x401bf4 <phase_3+328>
 0x0000000000401015 <+94>:  callq 0x40174d <explode_bomb>
 0x000000000040101a <+101>: mov    $0x74,%eax
 0x0000000000401021 <+106>: jmpq   0x401bf4 <phase_3+328>
 0x0000000000401024 <+111>: mov    $0x62,%eax
 0x0000000000401028 <+116>: cmpl   $0x246,0x14(%rsp)
 0x0000000000401031 <+124>: je    0x401bf4 <phase_3+328>
 0x0000000000401037 <+130>: callq 0x40174d <explode_bomb>
 0x000000000040103c <+135>: mov    $0x62,%eax
 0x0000000000401041 <+140>: jmpq   0x401bf4 <phase_3+328>
 0x0000000000401046 <+145>: mov    $0x61,%eax
 0x000000000040104b <+150>: cmpl   $0x1f5,0x14(%rsp)
 0x0000000000401053 <+158>: je    0x401bf4 <phase_3+328>
 0x0000000000401059 <+164>: callq 0x40174d <explode_bomb>
```

### 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 字符 数字组成的，并且在输入小于两个字符时引爆炸弹。

```
0x0000000000400fdd <+40>:    callq 0x400c40 <__isoc99_sscanf@plt>
0x0000000000400fe2 <+45>:    cmp    $0x2,%eax
0x0000000000400fe5 <+48>:    jg    0x400fec <phase_3+55>
0x0000000000400fe7 <+50>:    callq 0x40174d <explode_bomb>

(gdb) x /s $esi
0x402776:      "%d %c %d"
```

3. 继续往后，我们发现了三个 **lea** 指令，根据上个炸弹，我们可以知道输入的三个字符分别存放在 **0x10(%rsp)**、**0xf(%rsp)**、**0x14(%rsp)** 中。

```
0x0000000000400fc9 <+20>:    lea      0x14(%rsp),%r8
0x0000000000400fce <+25>:    lea      0xf(%rsp),%rcx
0x0000000000400fd3 <+30>:    lea      0x10(%rsp),%rdx
```

补充说明：由于函数的前六个参数按顺序存放在 **%rdi** **%rsi** **%rdx** **%rcx** **%r8** **%r9** 中，因此整个工作流为将对应地址放入寄存器，**sscanf** 将输入读入寄存器所指向的内存空间，其中 **(%rdx)** **(%rcx)** **(%r8)** 按顺序分别存放三个输入。

4. 接下来，我们看到了熟悉的语句：**jmpq \*0x402790(%rax,8)**，说明从 **<+70>** 位置开始是一个 **switch** 语句。

```
0x0000000000400ffb <+70>:    jmpq    *0x402790(%rax,8)
```

5. **switch** 语句执行前，判断如果输入的第一个数字大于 **7**，则引爆炸弹。说明输入的第一个数字和 **switch** 语句的转跳最多到 **7**。

```
0x0000000000400fec <+55>:    cmpl    $0x7,0x10(%rsp)
0x0000000000400ff1 <+60>:    ja      0x4010f3 <phase_3+318>
```

6. 使用 **x /20wx** 查看 **switch** 语句的转跳表，确定代码的整体框架：

0	1	2	3	4	5	6	7
<b>&lt;+77&gt;</b>	<b>+111&gt;</b>	<b>&lt;+145&gt;</b>	<b>&lt;+179&gt;</b>	<b>&lt;+210&gt;</b>	<b>&lt;+237&gt;</b>	<b>&lt;+264&gt;</b>	<b>&lt;+291&gt;</b>
<b>[(gdb) x /20wx 0x402790</b>							
<b>0x402790:</b>	<b>0x00401002</b>	<b>0x00000000</b>	<b>0x00401024</b>	<b>0x00000000</b>			
<b>0x4027a0:</b>	<b>0x00401046</b>	<b>0x00000000</b>	<b>0x00401068</b>	<b>0x00000000</b>			
<b>0x4027b0:</b>	<b>0x00401087</b>	<b>0x00000000</b>	<b>0x004010a2</b>	<b>0x00000000</b>			
<b>0x4027c0:</b>	<b>0x004010bd</b>	<b>0x00000000</b>	<b>0x004010d8</b>	<b>0x00000000</b>			

7. 由于每个部分的功能都一样，所以在此只解释 **0** 对应的情况：

首先将立即数 **0x74** 复制到 **%eax** 中，然后判断输入的第三个数字是否等于 **0x3de**，如果相等，就跳转到 **<+328>** 位置；否则引爆炸弹。

在 **<+328>** 位置，判断输入的第二个字符是否与 **%al** 寄存器储存的内容（即 **%eax** 寄存器的最低 **8** 位）相同。如果相同，**SAFE!!!**，否则 **BOMB!!!**

```

0x0000000000401002 <+77>:    mov    $0x74,%eax
0x0000000000401007 <+82>:    cmpl   $0x3de,0x14(%rsp)
0x000000000040100f <+90>:    je     0x4010fd <phase_3+328>
0x0000000000401015 <+96>:    callq  0x40174d <explode_bomb>
0x00000000004010fd <+328>:   cmp    0xf(%rsp),%al
0x0000000000401101 <+332>:   je     0x401108 <phase_3+339>
0x0000000000401103 <+334>:   callq  0x40174d <explode_bomb>

```

8. 综上，我们可以总结出每一种情况对应的输入，任选其一作为密码即可。

input1	input2(char)	input3
0	0x74	0x3de
1	0x62	0x246
2	0x61	0x1f5
3	0x74	0x8a
4	0x79	0xe1
5	0x74	0x1aa
6	0x73	0x2f1
7	0x63	0x15b

9. 输入密码，解除第三炸弹

```

Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Your job for this lab is to defuse your bomb.
Phase 1 defused. How about the next one?
[0 1 1 2 3 5
That's number 2. Keep going!
[0 t 990
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。

1. 还是先查看其汇编代码。

```
(gdb) disas
Dump of assembler code for function phase_4:
=> 0x000000000040115d <+0>:    sub    $0x18,%rsp
 0x0000000000401161 <+4>:    mov    %fs:0x28,%rax
 0x000000000040116a <+13>:   mov    %rax,0x8(%rsp)
 0x000000000040116f <+18>:   xor    %eax,%eax
 0x0000000000401171 <+20>:   mov    %rsp,%rcx
 0x0000000000401174 <+23>:   lea    0x4(%rsp),%rdx
 0x0000000000401179 <+28>:   mov    $0x402a2d,%esi
 0x000000000040117e <+33>:   callq 0x400c40 <_isoc99_sscanf@plt>
 0x0000000000401183 <+38>:   cmp    $0x2,%eax
 0x0000000000401186 <+41>:   jne    0x401193 <phase_4+54>
 0x0000000000401188 <+43>:   mov    (%rsp),%eax
 0x000000000040118b <+46>:   sub    $0x2,%eax
 0x000000000040118e <+49>:   cmp    $0x2,%eax
 0x0000000000401191 <+52>:   jbe    0x401198 <phase_4+59>
 0x0000000000401193 <+54>:   callq 0x40174d <explode_bomb>
 0x0000000000401198 <+59>:   mov    (%rsp),%esi
 0x000000000040119b <+62>:   mov    $0x6,%edi
 0x00000000004011a0 <+67>:   callq 0x401122 <func4>
 0x00000000004011a5 <+72>:   cmp    0x4(%rsp),%eax
 0x00000000004011a9 <+76>:   je     0x4011b0 <phase_4+83>
 0x00000000004011ab <+78>:   callq 0x40174d <explode_bomb>
 0x00000000004011b0 <+83>:   mov    0x8(%rsp),%rax
 0x00000000004011b5 <+88>:   xor    %fs:0x28,%rax
 0x00000000004011be <+97>:   je     0x4011c5 <phase_4+104>
 0x00000000004011c0 <+99>:   callq 0x400b90 <__stack_chk_fail@plt>
 0x00000000004011c5 <+104>:  add    $0x18,%rsp
 0x00000000004011c9 <+108>:  retq
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 `0x4(%rsp)` 和 `(%rsp)` 中；在 `sscanf` 返回值不等于 **2** 时爆炸。

```
(gdb) x /s $esi
0x402a2d:      "%d %d"
```

3. 接着往下看，在 `<+43>` 位置将第二个元素拷贝到 `%eax` 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 `0x6` 和第二个元素传入 `%edi` 和 `%esi` 中，调用函数 `func4`。

```
0x0000000000401188 <+43>:   mov    (%rsp),%eax
 0x000000000040118b <+46>:   sub    $0x2,%eax
 0x000000000040118e <+49>:   cmp    $0x2,%eax
 0x0000000000401191 <+52>:   jbe    0x401198 <phase_4+59>
 0x0000000000401193 <+54>:   callq 0x40174d <explode_bomb>
 0x0000000000401198 <+59>:   mov    (%rsp),%esi
 0x000000000040119b <+62>:   mov    $0x6,%edi
 0x00000000004011a0 <+67>:   callq 0x401122 <func4>
```

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。

```
(gdb) disas
Dump of assembler code for function func4:
=> 0x0000000000401122 <+0>:    test    %edi,%edi
    0x0000000000401124 <+2>:    jle     0x401151 <func4+47>
    0x0000000000401126 <+4>:    mov     %esi,%eax
    0x0000000000401128 <+6>:    cmp     $0x1,%edi
    0x000000000040112b <+9>:    je      0x40115b <func4+57>
    0x000000000040112d <+11>:   push    %r12
    0x000000000040112f <+13>:   push    %rbp
    0x0000000000401130 <+14>:   push    %rbx
    0x0000000000401131 <+15>:   mov     %esi,%ebp
    0x0000000000401133 <+17>:   mov     %edi,%ebx
    0x0000000000401135 <+19>:   lea     -0x1(%rdi),%edi
    0x0000000000401138 <+22>:   callq   0x401122 <func4>
    0x000000000040113d <+27>:   lea     0x0(%rbp,%rax,1),%r12d
    0x0000000000401142 <+32>:   lea     -0x2(%rbx),%edi
    0x0000000000401145 <+35>:   mov     %ebp,%esi
    0x0000000000401147 <+37>:   callq   0x401122 <func4>
    0x000000000040114c <+42>:   add    %r12d,%eax
    0x000000000040114f <+45>:   jmp     0x401157 <func4+53>
    0x0000000000401151 <+47>:   mov     $0x0,%eax
    0x0000000000401156 <+52>:   retq
    0x0000000000401157 <+53>:   pop    %rbx
    0x0000000000401158 <+54>:   pop    %rbp
    0x0000000000401159 <+55>:   pop    %r12
    0x000000000040115b <+57>:   repz   retq
End of assembler dump.
```

- 首先是递归出口，当 `%edi` 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 `%eax`，因此此处的返回值一定为 0；接着将第二个元素复制到 `%eax` 中，当 `%edi` 寄存器中的值等于 1 时，返回第二个元素。

```
=> 0x0000000000401122 <+0>:    test    %edi,%edi
    0x0000000000401124 <+2>:    jle     0x401151 <func4+47>
    0x0000000000401126 <+4>:    mov     %esi,%eax
    0x0000000000401128 <+6>:    cmp     $0x1,%edi
    0x000000000040112b <+9>:    je      0x40115b <func4+57>
    0x0000000000401151 <+47>:   mov     $0x0,%eax
    0x0000000000401156 <+52>:   retq
    0x0000000000401157 <+53>:   pop    %rbx
    0x0000000000401158 <+54>:   pop    %rbp
    0x0000000000401159 <+55>:   pop    %r12
    0x000000000040115b <+57>:   repz   retq
```

- 接着将 `%edi` 复制到 `%ebx` 中，令 `%edi` 减一，调用 `func4` 函数。然后将返回值 `%rax` 加上第二个元素并存储在 `%rdi` 中。将 `%rbx`（也就是 `%edi` 的初值）减 2 的值存放在 `%esi` 中，再次调用 `func4`，并将返回值加上之前的处理过的返回值 `%r12d`。

```

0x0000000000401131 <+15>:    mov    %esi,%ebp
0x0000000000401133 <+17>:    mov    %edi,%ebx
0x0000000000401135 <+19>:    lea    -0x1(%rdi),%edi
0x0000000000401138 <+22>:    callq  0x401122 <func4>
0x000000000040113d <+27>:    lea    0x0(%rbp,%rax,1),%r12d
0x0000000000401142 <+32>:    lea    -0x2(%rbx),%edi
0x0000000000401145 <+35>:    mov    %ebp,%esi
0x0000000000401147 <+37>:    callq  0x401122 <func4>
0x000000000040114c <+42>:    add    %r12d,%eax

```

- 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 + func4(n - 2);
6 }
7
8 // 调用 func(6, 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。

```

0x00000000004011a5 <+72>:    cmp    0x4(%rsp),%eax
0x00000000004011a9 <+76>:    je     0x4011b0 <phase_4+83>
0x00000000004011ab <+78>:    callq 0x40174d <explode_bomb>

```

- 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。

```

Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
[Your job for this lab is to defuse your bomb.
Phase 1 defused. How about the next one?
[0 1 1 2 3 5
That's number 2. Keep going!
[0 t 990
Halfway there!
[60 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

### 1. 老样子，先查汇编

```
(gdb) disas
Dump of assembler code for function phase_5:
=> 0x00000000004011ca <+0>:    push   %rbx
  0x00000000004011cb <+1>:    sub    $0x10,%rsp
  0x00000000004011cf <+5>:    mov    %rdi,%rbx
  0x00000000004011d2 <+8>:    mov    %fs:0x28,%rax
  0x00000000004011db <+17>:   mov    %rax,0x8(%rsp)
  0x00000000004011e0 <+22>:   xor    %eax,%eax
  0x00000000004011e2 <+24>:   callq  0x40145b <string_length>
  0x00000000004011e7 <+29>:   cmp    $0x6,%eax
  0x00000000004011ea <+32>:   je     0x4011f1 <phase_5+39>
  0x00000000004011ec <+34>:   callq  0x40174d <explode_bomb>
  0x00000000004011f1 <+39>:   mov    $0x0,%eax
  0x00000000004011f6 <+44>:   movzbl (%rbx,%rax,1),%edx
  0x00000000004011fa <+48>:   and    $0xf,%edx
  0x00000000004011fd <+51>:   movzbl 0x4027d0(%rdx),%edx
  0x0000000000401204 <+58>:   mov    %dl,(%rsp,%rax,1)
  0x0000000000401207 <+61>:   add    $0x1,%rax
  0x000000000040120b <+65>:   cmp    $0x6,%rax
  0x000000000040120f <+69>:   jne    0x4011f6 <phase_5+44>
  0x0000000000401211 <+71>:   movb   $0x0,0x6(%rsp)
  0x0000000000401216 <+76>:   mov    $0x40277f,%esi
  0x000000000040121b <+81>:   mov    %rsp,%rdi
  0x000000000040121e <+84>:   callq  0x401479 <strings_not_equal>
  0x0000000000401223 <+89>:   test   %eax,%eax
  0x0000000000401225 <+91>:   je     0x40122c <phase_5+98>
  0x0000000000401227 <+93>:   callq  0x40174d <explode_bomb>
  0x000000000040122c <+98>:   mov    0x8(%rsp),%rax
  0x0000000000401231 <+103>:  xor    %fs:0x28,%rax
  0x000000000040123a <+112>:  je     0x401241 <phase_5+119>
  0x000000000040123c <+114>:  callq  0x400b90 <__stack_chk_fail@plt>
  0x0000000000401241 <+119>:  add    $0x10,%rsp
  0x0000000000401245 <+123>:  pop    %rbx
  0x0000000000401246 <+124>:  retq
End of assembler dump.
```

2. 可以看到程序在 <+24> 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。

```
0x00000000004011e2 <+24>:   callq  0x40145b <string_length>
  0x00000000004011e7 <+29>:   cmp    $0x6,%eax
  0x00000000004011ea <+32>:   je     0x4011f1 <phase_5+39>
  0x00000000004011ec <+34>:   callq  0x40174d <explode_bomb>
```

3. 继续往下看，发现在 <+69> 位置的跳转指令会跳转回 <+44> 位置，显然这也是一个 `do while` 循环，接下来分析循环的部分。

```
0x000000000040120f <+69>:   jne    0x4011f6 <phase_5+44>
```

4. 在循环外，程序先将 `0x0` 赋值给了 `%eax`，然后进入循环。循环开始时，将 `%rbx + %rax` 所指向地址的数据的末尾 8 位做 0 拓展传给了 `%edx` 寄存器，然后将 `%edx` 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 `%edx` 的最低 4 位。

```
0x00000000004011f1 <+39>:    mov     $0x0,%eax  
0x00000000004011f6 <+44>:    movzbl (%rbx,%rax,1),%edx  
0x00000000004011fa <+48>:    and    $0xf,%edx
```

5. 此外，我们注意到 `<+61>` 位置将 `%rax` 的值加 1，并在其值等于 6 时结束循环。因此 `%eax` 的作用为作为索引来遍历我们输入的字符串。

```
0x0000000000401207 <+61>:    add    $0x1,%rax  
0x000000000040120b <+65>:    cmp    $0x6,%rax  
0x000000000040120f <+69>:    jne    0x4011f6 <phase_5+44>
```

6. 接下来，和以前一样，程序将之前取出的末尾 4 位作为索引去访问一个地址，于是使用 `x /s` 命令查看该地址存放的数据，发现是一串不规律的字符串。

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



```
0x00000000004011fd <+51>:    movzbl 0x4027d0(%rdx),%edx  
[(gdb) x /s 0x4027d0  
0x4027d0 <array.3600>: "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
```

7. 然后程序将取得的结果的末尾 8 位存入 `%rsp + %rax` 所指的内存位置。不难发现，此处 `%rax` 的作用也是作为索引。

```
0x00000000004011fd <+51>:    movzbl 0x4027d0(%rdx),%edx
```

8. 接着将 `%rax` 加 1，并判断是否结束循环。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，取出某个字符并存储到 `%rsp` 所指的内存中，形成一个新的字符串。
9. 接着在 `<+76>` 位置将某个地址传入 `%esi`，并将 `%rsp` 传入 `%rdi`，紧接着调用了 `string_not_equal` 函数。说明程序在将之前生成的字符串与某个字符串做对比，并通过结果确定炸弹是否爆炸。于是我们使用 `x /s` 命令查看目标字符串。

```
0x0000000000401216 <+76>:    mov    $0x40277f,%esi
0x000000000040121b <+81>:    mov    %rsp,%rdi
0x000000000040121e <+84>:    callq 0x401479 <strings_not_equal>
0x0000000000401223 <+89>:    test   %eax,%eax
0x0000000000401225 <+91>:    je     0x40122c <phase_5+98>
0x0000000000401227 <+93>:    callq 0x40174d <explode_bomb>
[(gdb) x /s 0x40277f
0x40277f:      "oilers"
```

10. **oilers!!!** 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某一不规律的字符串中依次取出 `o i l e r s` 几个字符，对应下来为 `ZDOEFG`。至此 `Dr. Evil` 的阴谋彻底破灭，可喜可贺。

$$\xi \sum_{n=1}^{\infty} \lambda \int \sqrt[n]{dx} \sim \sim () \left( \sqrt[n]{\sigma \mu} \right)$$

## 5 总结体会

- 收获：

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：

建议下次炸弹部署时间是中国时间，不是北美时间；

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。