Computer Architecture

Bomb Lab Report

2016 Spring

李琛
微电子科学与工程
13307130163@fudan.edu.cn

Table of Content

1 introduction

1.1 Overview

A "binary bombïs a program provided to students as an object code file. When run, it prompts the user to type in 6 different strings. If any of these is incorrect, the bomb “explodes,” printing an error message and logging the event on a grading server. Students must “defuse” their own unique bomb by disassembling and reverse engineering the program to determine what the 6 strings should be. The lab teaches students to understand assembly language, and also forces them to learn how to use a debugger. It’s also great fun. A legendary lab among the CMU undergrads.

1.2 Preparation

在拆除炸弹之前，首先进行一些前期准备：

- 通过 putty 登陆服务器，发现我本次 Lab 的实验文件：**bomb51.tar**
- 执行 **tar xvf bomb51.tar** 解压文件，得到四个文件：bomb bomb.c ID README
- 其中 ID README 分别是学生的编号和这次的说明文档。

  于是查看 **bomb.c**, 发现其头文件声明如下：

```
1 #include <stdio.h>
2 #include "support.h"
3 #include "phases.h"
```

  phase.h 包含了这次炸弹的全部关卡，而它并没有在 Lab 中给出。因此我们只能从可执行文件 **bomb** 下手

- 执行 **objdump -d bomb > bomb.txt**，反编译可执行文件，将汇编代码输出到 **bomb.txt**
- 反编译之后的代码非常长，不过在仔细研究之后，发现其中有六个函数 <**phase_1..6**>，分别对应六个关卡。因此，这次 Lab 的关键就是破解这六个关卡对应的汇编代码，分析这六个函数的功能。

2 phase_1

- 第一关的汇编代码如下：

```
08048c10 <phase_1>:
 8048c10:       83 ec 1c                sub    $0x1c,%esp
 8048c13:       c7 44 24 04 ec 99 04    movl   $0x80499ec,0x4(%esp)
 8048c1a:       08
 8048c1b:       8b 44 24 20             mov    0x20(%esp),%eax
 8048c1f:       89 04 24                mov    %eax,(%esp)
 8048c22:       e8 1d 04 00 00          call   8049044 <strings_not_equal>
 8048c27:       85 c0                   test   %eax,%eax
 8048c29:       74 05                   je     8048c30 <phase_1+0x20>
 8048c2b:       e8 a3 09 00 00          call   80495d3 <explode_bomb>
 8048c30:       83 c4 1c                add    $0x1c,%esp
 8048c33:       c3                      ret
```

- 这个函数较短，分析发现，它调用了一个名为 **strings_not_equal** 函数。

  分析 **strings_not_equal** 这个函数，发现它实现这样一个功能：

  比较两个字符串，相等返回 0，不相等返回 1

- 分析 call 之后的程序，发现如果返回值（即 *%eax*）不为 0 的话，会调用 **explode_bomb** 函数，炸弹会被引爆。因此，我们输入的字符串应该与某个字符串相同。

- 这样，破解 phase_1 的方法就很清晰了：

  输入存在地址 **0x80499ec** 中的字符串

- 于是现在的问题是找出存放在地址 0x80499ec 中的字符串。

  从调用 strings_not_equal 的语句向上看：

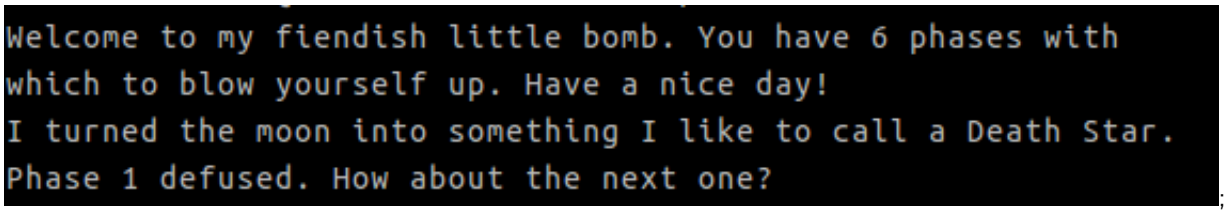
```
 8048c13:       c7 44 24 04 ec 99 04    movl   $0x80499ec,0x4(%esp)
 8048c1a:       08
 8048c1b:       8b 44 24 20             mov    0x20(%esp),%eax
 8048c1f:       89 04 24                mov    %eax,(%esp)
```

  这表示，这个函数的两个参数，一个是程序自身地址 **0x80499ec**，另一个 **0x20(%esp)** 就是我们的输入参数。

- 我们首先需要知道 **0x80499ec** 地址里存放的数据。于是使用 **gdb**，输入命令 **p (char *) 0x80499ec**, 查看它存放的数据：

```
(gdb) p (char *) 0x80499ec
$1 = 0x80499ec "I turned the moon into something I like to call a Death Star."
```

- 第一关的答案水落石出：输入 **I turned the moon into something I like to call a Death Star.** ，顺利过关。



;

主要复习知识点：常量的存储、函数的参数传递

3 phase_2

- 第二关的汇编代码如下：

```
08048c34 <phase_2>:
 8048c34:       53                      push   %ebx
 8048c35:       83 ec 38                sub    $0x38,%esp
 8048c38:       8d 44 24 18             lea    0x18(%esp),%eax
 8048c3c:       89 44 24 04             mov    %eax,0x4(%esp)
 8048c40:       8b 44 24 40             mov    0x40(%esp),%eax
 8048c44:       89 04 24                mov    %eax,(%esp)
 8048c47:       e8 8c 0a 00 00          call   80496d8 <read_six_numbers>
 8048c4c:       83 7c 24 18 01          cmpl   $0x1,0x18(%esp)
 8048c51:       74 05                   je     8048c58 <phase_2+0x24>
 8048c53:       e8 7b 09 00 00          call   80495d3 <explode_bomb>
 8048c58:       bb 01 00 00 00          mov    $0x1,%ebx
 8048c5d:       89 d8                   mov    %ebx,%eax
 8048c5f:       83 c3 01                add    $0x1,%ebx
 8048c62:       89 da                   mov    %ebx,%edx
 8048c64:       0f af 54 84 14          imul   0x14(%esp,%eax,4),%edx
 8048c69:       39 54 84 18             cmp    %edx,0x18(%esp,%eax,4)
 8048c6d:       74 05                   je     8048c74 <phase_2+0x40>
 8048c6f:       e8 5f 09 00 00          call   80495d3 <explode_bomb>
 8048c74:       83 fb 06                cmp    $0x6,%ebx
 8048c77:       75 e4                   jne    8048c5d <phase_2+0x29>
 8048c79:       83 c4 38                add    $0x38,%esp
 8048c7c:       5b                      pop    %ebx
 8048c7d:       c3                      ret
```

- 我们首先发现，它调用了一个 **read_six_numbers** 函数。经过分析，这个函数从输入中读入六个整数，并按地址从低到高存放在 **0x18(%esp)** - **(0x30%esp)** 这 24 个字节中。也就是说，**a[0]** 在 **0x18(%esp)**，**a[1]** 在 **0x1c(%esp)**，以此类推。
- 然后观察下面一段代码：

```
 8048c4c:       83 7c 24 18 01          cmpl   $0x1,0x18(%esp)
 8048c51:       74 05                   je     8048c58 <phase_2+0x24>
 8048c53:       e8 7b 09 00 00          call   80495d3 <explode_bomb>
```

- 这段代码将 **0x18(%esp)** 即 **a[0]** 与 1 进行比较，如果不等则炸弹爆炸。这一段说明，我们输入的第一个数必须为 1。
- 接下来是一个循环：

```
8048c58:        bb 01 00 00 00          mov    $0x1,%ebx
 8048c5d:       89 d8                   mov    %ebx,%eax
 8048c5f:       83 c3 01                add    $0x1,%ebx
 8048c62:       89 da                   mov    %ebx,%edx
 8048c64:       0f af 54 84 14          imul   0x14(%esp,%eax,4),%edx
 8048c69:       39 54 84 18             cmp    %edx,0x18(%esp,%eax,4)
 8048c6d:       74 05                   je     8048c74 <phase_2+0x40>
 8048c6f:       e8 5f 09 00 00          call   80495d3 <explode_bomb>
 8048c74:       83 fb 06                cmp    $0x6,%ebx
 8048c77:       75 e4                   jne    8048c5d <phase_2+0x29>
```

这个循环的循环变量是 %**ebx**，从 2 循环到 5。同时，%**eax** 始终为 %**ebx** - **1** 中间的判断语句，可以发现它是将 **0x14(%esp, %eax, 4) *** %**edx** 相乘，并与 **0x18(%esp, %eax, 4)** 比较。由于 a 数组的起始位置为 **0x18(%esp)**，这两个地址就是 **a[_eax - 1]** 和 **a[_eax]**。

因此，其对应的 c 语言代码如下：

```c
void phase_2() {
        if(num[0] != 1) explode_bomb();
        int b = 1;
        while(b <= 5) {
                a = b + 1;
                if(num[b - 1] * a != num[b])
                        explode_bomb();
                b++;
        }
}
```

- 这个函数表明，我们输入的数组 a 要满足以下条件：
  - 数组长度为 6
  - a[0] = 1
  - a[i] = a[i - 1] * (i + 1)
- 所以，第二关的答案就水落石出了：**1 2 6 24 120 720**

```
I turned the moon into something I like to call a Death Star.
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
```

考察知识点：数组的存储

4 phase_3

- 第三关的汇编代码如下：

```
08048c7e <phase_3>:
 8048c7e:       83 ec 2c                sub    $0x2c,%esp
 8048c81:       8d 44 24 1c             lea    0x1c(%esp),%eax
 8048c85:       89 44 24 0c             mov    %eax,0xc(%esp)
 8048c89:       8d 44 24 18             lea    0x18(%esp),%eax
 8048c8d:       89 44 24 08             mov    %eax,0x8(%esp)
 8048c91:       c7 44 24 04 c6 9d 04    movl   $0x8049dc6,0x4(%esp)
 8048c98:       08
 8048c99:       8b 44 24 30             mov    0x30(%esp),%eax
 8048c9d:       89 04 24                mov    %eax,(%esp)
 8048ca0:       e8 6b fc ff ff          call   8048910 <__isoc99_sscanf@plt>
 8048ca5:       83 f8 01                cmp    $0x1,%eax
 8048ca8:       7f 05                   jg     8048caf <phase_3+0x31>
 8048caa:       e8 24 09 00 00          call   80495d3 <explode_bomb>
 8048caf:       83 7c 24 18 07          cmpl   $0x7,0x18(%esp)
 8048cb4:       77 64                   ja     8048d1a <phase_3+0x9c>
 8048cb6:       8b 44 24 18             mov    0x18(%esp),%eax
 8048cba:       ff 24 85 5c 9a 04 08    jmp    *0x8049a5c(,%eax,4)
 8048cc1:       b8 00 00 00 00          mov    $0x0,%eax
 8048cc6:       eb 05                   jmp    8048ccd <phase_3+0x4f>
 8048cc8:       b8 8e 01 00 00          mov    $0x18e,%eax
 8048ccd:       2d 66 01 00 00          sub    $0x166,%eax
 8048cd2:       eb 05                   jmp    8048cd9 <phase_3+0x5b>
 8048cd4:       b8 00 00 00 00          mov    $0x0,%eax
 8048cd9:       05 92 00 00 00          add    $0x92,%eax
 8048cde:       eb 05                   jmp    8048ce5 <phase_3+0x67>
 8048ce0:       b8 00 00 00 00          mov    $0x0,%eax
 8048ce5:       2d e9 00 00 00          sub    $0xe9,%eax
 8048cea:       eb 05                   jmp    8048cf1 <phase_3+0x73>
 8048cec:       b8 00 00 00 00          mov    $0x0,%eax
 8048cf1:       05 28 01 00 00          add    $0x128,%eax
 8048cf6:       eb 05                   jmp    8048cfd <phase_3+0x7f>
 8048cf8:       b8 00 00 00 00          mov    $0x0,%eax
 8048cfd:       2d df 01 00 00          sub    $0x1df,%eax
 8048d02:       eb 05                   jmp    8048d09 <phase_3+0x8b>
 8048d04:       b8 00 00 00 00          mov    $0x0,%eax
 8048d09:       05 df 01 00 00          add    $0x1df,%eax
```

```
8048d0e:    eb 05                   jmp    8048d15 <phase_3+0x97>
8048d10:    b8 00 00 00 00          mov    $0x0,%eax
8048d15:    83 c0 80                add    $0xffffff80,%eax
8048d18:    eb 0a                   jmp    8048d24 <phase_3+0xa6>
8048d1a:    e8 b4 08 00 00          call   80495d3 <explode_bomb>
8048d1f:    b8 00 00 00 00          mov    $0x0,%eax
8048d24:    83 7c 24 18 05          cmpl   $0x5,0x18(%esp)
8048d29:    7f 06                   jg     8048d31 <phase_3+0xb3>
8048d2b:    3b 44 24 1c             cmp    0x1c(%esp),%eax
8048d2f:    74 05                   je     8048d36 <phase_3+0xb8>
8048d31:    e8 9d 08 00 00          call   80495d3 <explode_bomb>
8048d36:    83 c4 2c                add    $0x2c,%esp
8048d39:    c3                      ret
```

- 首先看到它调用了 **sscanf** 函数，参数存放在 **0x8049dc6** 地址中。

  于是首先用 gdb 查看 **0x8049dc6** 地址中的参数：

```
(gdb) p (char *) 0x8049dc6
$3 = 0x8049dc6 "%d %d"
```

  "%d%d"代表读入了两个整数。调用 sscanf 结束后，将 sscanf 的返回值与 1 比较，若不大于 1 则炸弹爆炸。

  由此可以得到我们这关的目标是输入两个整数 **a b**。

- 输入的两个整数 **a b** 存放在 **0x18(%esp)** 和 **0x1c(%esp)** 中。接着往下看程序：

```
8048caf:    83 7c 24 18 07          cmpl   $0x7,0x18(%esp)
8048cb4:    77 64                   ja     8048d1a <phase_3+0x9c>
```

  这里可以发现，程序将 **0x18(%esp)**，即输入的第一个整数 **a** 与 7 进行比较，如果大于 7 则炸弹爆炸。同时，由于使用的是 **ja** 命令，即 **a** 为无符号数，**a** 应该不小于 0

  这样我们可以得出第一个限制条件：输入的第一个整数 $0 \leq a \leq 7$

  继续往下看，注意到这一段：

```
8048cb6:    8b 44 24 18             mov    0x18(%esp),%eax
8048cba:    ff 24 85 5c 9a 04 08    jmp    *0x8049a5c(,%eax,4)
```

- 这段具有很明显的 **switch** 语句的特征，而跳转表存放在 **0x8049a5c** 中。

于是调用 gdb，先查看对应的跳转表：

```
(gdb) p /x *0x8049a5c
$4 = 0x8048cc8
(gdb) p /x *(0x8049a5c + 4)
$5 = 0x8048cc1
(gdb) p /x *(0x8049a5c + 8)
$6 = 0x8048cd4
(gdb) p /x *(0x8049a5c + 12)
$7 = 0x8048ce0
(gdb) p /x *(0x8049a5c + 16)
$8 = 0x8048cec
(gdb) p /x *(0x8049a5c + 20)
$9 = 0x8048cf8
(gdb) p /x *(0x8049a5c + 24)
$10 = 0x8048d04
(gdb) p /x *(0x8049a5c + 28)
$11 = 0x8048d10
```

这样可以将接下来的代码划分模块：

```
// case 1:
8048cc1:        b8 00 00 00 00          mov    $0x0,%eax
8048cc6:        eb 05                   jmp    8048ccd <phase_3+0x4f>
// case 0:
8048cc8:        b8 8e 01 00 00          mov    $0x18e,%eax
8048ccd:        2d 66 01 00 00          sub    $0x166,%eax
8048cd2:        eb 05                   jmp    8048cd9 <phase_3+0x5b>
// case 2:
8048cd4:        b8 00 00 00 00          mov    $0x0,%eax
8048cd9:        05 92 00 00 00          add    $0x92,%eax
8048cde:        eb 05                   jmp    8048ce5 <phase_3+0x67>
// case 3:
8048ce0:        b8 00 00 00 00          mov    $0x0,%eax
8048ce5:        2d e9 00 00 00          sub    $0xe9,%eax
8048cea:        eb 05                   jmp    8048cf1 <phase_3+0x73>
// case 4:
8048cec:        b8 00 00 00 00          mov    $0x0,%eax
8048cf1:        05 28 01 00 00          add    $0x128,%eax
8048cf6:        eb 05                   jmp    8048cfd <phase_3+0x7f>
// case 5:
8048cf8:        b8 00 00 00 00          mov    $0x0,%eax
8048cfd:        2d df 01 00 00          sub    $0x1df,%eax
```

```
8048d02:        eb 05                   jmp    8048d09 <phase_3+0x8b>
// case 6:
8048d04:        b8 00 00 00 00          mov    $0x0,%eax
8048d09:        05 df 01 00 00          add    $0x1df,%eax
8048d0e:        eb 05                   jmp    8048d15 <phase_3+0x97>
// case 7:
8048d10:        b8 00 00 00 00          mov    $0x0,%eax
8048d15:        83 c0 80                add    $0xffffff80,%eax
```

- 于是可以得出对应的 c 语言代码：

```c
int fun(int a) {
        int ret = 0;
        switch (a) {
                case 1: ret = 0;
                case 0:
                        if (a != 1) ret = 0x18e;
                        ret -= 0x166;
                case 2: ret += 0x92;
                case 3: ret -= 0xe9;
                case 4: ret += 0x128;
                case 5: ret -= 0x1df;
                case 6: ret += 0x1df;
                case 7: ret += 0xffffff80;
        }
        return ret;
}
```

经过整理后的代码如下：

```c
int fun(int a) {
        int ret = 0;
        switch(a) {
                case 0 : ret = 121; break;
                case 1 : ret = -277; break;
                case 2 : ret = 81; break;
                case 3 : ret = -65; break;
                case 4 : ret = 168; break;
                case 5 : ret = -128; break;
                case 6 : ret = 351; break;
                case 7 : ret = -128; break;
        }
        return ret;
```

```
14 }
```

- 在 switch 语句段后，有这么一段：

```
1 8048d1f:      b8 00 00 00 00          mov    $0x0,%eax
2 8048d24:      83 7c 24 18 05          cmpl   $0x5,0x18(%esp)
3 8048d29:      7f 06                   jg     8048d31 <phase_3+0xb3>
4 8048d2b:      3b 44 24 1c             cmp    0x1c(%esp),%eax
5 8048d2f:      74 05                   je     8048d36 <phase_3+0xb8>
6 8048d31:      e8 9d 08 00 00          call   80495d3 <explode_bomb>
```

这里给出了第二个限制条件：输入的第一个整数 $a \leq 5$

最后，程序将输入的第二个整数 **b** 与 switch 的结果进行比较，不同则炸弹爆炸。

- 因此，这关的答案就水落石出了：

输入一个整数 $a(0 \leq a \leq 5)$，和 a 经过 switch 之后的结果 b

选用 $\mathbf{a = 4, b = 168}$, 顺利过关

```
I turned the moon into something I like to call a Death Star.
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
4 168
Halfway there!
```

5 phase_4

- 第四关的汇编代码如下：

```
08048d64 <phase_4>:
 8048d64:       83 ec 2c                sub    $0x2c,%esp
 8048d67:       8d 44 24 1c             lea    0x1c(%esp),%eax
 8048d6b:       89 44 24 08             mov    %eax,0x8(%esp)
 8048d6f:       c7 44 24 04 c9 9d 04    movl   $0x8049dc9,0x4(%esp)
 8048d76:       08
 8048d77:       8b 44 24 30             mov    0x30(%esp),%eax
 8048d7b:       89 04 24                mov    %eax,(%esp)
 8048d7e:       e8 8d fb ff ff          call   8048910 <__isoc99_sscanf@plt>
 8048d83:       83 f8 01                cmp    $0x1,%eax
 8048d86:       75 07                   jne    8048d8f <phase_4+0x2b>
 8048d88:       83 7c 24 1c 00          cmpl   $0x0,0x1c(%esp)
 8048d8d:       7f 05                   jg     8048d94 <phase_4+0x30>
 8048d8f:       e8 3f 08 00 00          call   80495d3 <explode_bomb>
 8048d94:       8b 44 24 1c             mov    0x1c(%esp),%eax
 8048d98:       89 04 24                mov    %eax,(%esp)
 8048d9b:       e8 9a ff ff ff          call   8048d3a <func4>
 8048da0:       3d a7 41 00 00          cmp    $0x41a7,%eax
 8048da5:       74 05                   je     8048dac <phase_4+0x48>
 8048da7:       e8 27 08 00 00          call   80495d3 <explode_bomb>
 8048dac:       83 c4 2c                add    $0x2c,%esp
 8048daf:       c3                      ret
```

首先，它和 phase_3 一样，调用了 sscanf 函数。和上一个函数的处理方法一样，运用 gdb，发现 sscanf 的参数是**"%d"**，即读入一个整数 n，存放在 **0x1c(%esp)**。

-

```
 8048d88:       83 7c 24 1c 00          cmpl   $0x0,0x1c(%esp)
 8048d8d:       7f 05                   jg     8048d94 <phase_4+0x30>
 8048d8f:       e8 3f 08 00 00          call   80495d3 <explode_bomb>
```

读入后，将 n 与 0 比较，如果 $n \leq 0$ 炸弹就会爆炸。

因此，我们得到了第一个限制条件：$n > 0$

-

```
 8048d94:       8b 44 24 1c             mov    0x1c(%esp),%eax
 8048d98:       89 04 24                mov    %eax,(%esp)
 8048d9b:       e8 9a ff ff ff          call   8048d3a <func4>
```

随后，将 n 作为参数传入函数 **func4** 中。

-

```
8048da0:        3d a7 41 00 00          cmp    $0x41a7,%eax
8048da5:        74 05                   je     8048dac <phase_4+0x48>
8048da7:        e8 27 08 00 00          call   80495d3 <explode_bomb>
```

这段代码表示，如果函数的返回值不为 **0x41a7**，则炸弹爆炸。

这样就知道了第四关的要求：找出使 **func4** 的结果为 **0x41a7** 的输入值。

- 于是我们开始研究 **func4** 函数。

```

08048d3a <func4>:
8048d3a:        83 ec 1c                sub    $0x1c,%esp
8048d3d:        8b 54 24 20             mov    0x20(%esp),%edx
8048d41:        b8 01 00 00 00          mov    $0x1,%eax
8048d46:        85 d2                   test   %edx,%edx
8048d48:        7e 16                   jle    8048d60 <func4+0x26>
8048d4a:        83 ea 01                sub    $0x1,%edx
8048d4d:        89 14 24                mov    %edx,(%esp)
8048d50:        e8 e5 ff ff ff          call   8048d3a <func4>
8048d55:        8d 14 c5 00 00 00 00    lea    0x0(,%eax,8),%edx
8048d5c:        29 c2                   sub    %eax,%edx
8048d5e:        89 d0                   mov    %edx,%eax
8048d60:        83 c4 1c                add    $0x1c,%esp
8048d63:        c3                      ret
```

分析这一个函数，可以得出它的作用如下：

- 首先，函数将返回值置为 1。
- 如果传入参数 $x \leq 0$，则直接返回。
- 递归调用 **fun4**
- 将递归的结果乘 7 后作为返回值

对应的 C 语言代码如下：

```
int fun4(int x) {
        int ret = 1;
        if (x <= 0)
                return ret;
        return ret * 7;
}
```

因此，我们的问题变成：$\log_7(0x41a7) = ?$

经过计算得出，$7^5 = 16807 = 0x41a7$

输入 5，成功过关。

```
I turned the moon into something I like to call a Death Star.
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
4 168
Halfway there!
5 austinpowers
So you got that one.  Try this one.
```

6 phase_5

- 第五关的汇编代码如下：

```
08048db0 <phase_5>:
 8048db0:       53                      push   %ebx
 8048db1:       83 ec 28                sub    $0x28,%esp
 8048db4:       8b 5c 24 30             mov    0x30(%esp),%ebx
 8048db8:       65 a1 14 00 00 00       mov    %gs:0x14,%eax
 8048dbe:       89 44 24 1c             mov    %eax,0x1c(%esp)
 8048dc2:       31 c0                   xor    %eax,%eax
 8048dc4:       89 1c 24                mov    %ebx,(%esp)
 8048dc7:       e8 5f 02 00 00          call   804902b <string_length>
 8048dcc:       83 f8 06                cmp    $0x6,%eax
 8048dcf:       74 05                   je     8048dd6 <phase_5+0x26>
 8048dd1:       e8 fd 07 00 00          call   80495d3 <explode_bomb>
 8048dd6:       b8 00 00 00 00          mov    $0x0,%eax
 8048ddb:       0f be 14 03             movsbl (%ebx,%eax,1),%edx
 8048ddf:       83 e2 0f                and    $0xf,%edx
 8048de2:       0f b6 92 7c 9a 04 08    movzbl 0x8049a7c(%edx),%edx
 8048de9:       88 54 04 15             mov    %dl,0x15(%esp,%eax,1)
 8048ded:       83 c0 01                add    $0x1,%eax
 8048df0:       83 f8 06                cmp    $0x6,%eax
 8048df3:       75 e6                   jne    8048ddb <phase_5+0x2b>
 8048df5:       c6 44 24 1b 00          movb   $0x0,0x1b(%esp)
 8048dfa:       c7 44 24 04 52 9a 04    movl   $0x8049a52,0x4(%esp)
 8048e01:       08
 8048e02:       8d 44 24 15             lea    0x15(%esp),%eax
 8048e06:       89 04 24                mov    %eax,(%esp)
 8048e09:       e8 36 02 00 00          call   8049044 <strings_not_equal>
 8048e0e:       85 c0                   test   %eax,%eax
 8048e10:       74 05                   je     8048e17 <phase_5+0x67>
 8048e12:       e8 bc 07 00 00          call   80495d3 <explode_bomb>
 8048e17:       8b 44 24 1c             mov    0x1c(%esp),%eax
 8048e1b:       65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 8048e22:       74 05                   je     8048e29 <phase_5+0x79>
 8048e24:       e8 27 fa ff ff          call   8048850 <__stack_chk_fail@plt>
 8048e29:       83 c4 28                add    $0x28,%esp
 8048e2c:       5b                      pop    %ebx
 8048e2d:       c3                      ret
```

- ```
  8048dc7:    e8 5f 02 00 00          call   804902b <string_length>
  8048dcc:    83 f8 06                cmp    $0x6,%eax
  8048dcf:    74 05                   je     8048dd6 <phase_5+0x26>
  8048dd1:    e8 fd 07 00 00          call   80495d3 <explode_bomb>
  ```

  首先，我们发现它调用了 **string_length** 这个函数，并且将返回值与 6 进行比较，不相等的话炸弹会爆炸。查看 **string_length** 的代码发现，它的功能为返回一个字符串的长度。

  因此，我们得到了第一个限制条件：需要输入一个长度为 6 的字符串。

- ```
  8048dd6:    b8 00 00 00 00          mov    $0x0,%eax
  8048ddb:    0f be 14 03             movsbl (%ebx,%eax,1),%edx
  8048ddf:    83 e2 0f                and    $0xf,%edx
  8048de2:    0f b6 92 7c 9a 04 08    movzbl 0x8049a7c(%edx),%edx
  8048de9:    88 54 04 15             mov    %dl,0x15(%esp,%eax,1)
  8048ded:    83 c0 01                add    $0x1,%eax
  8048df0:    83 f8 06                cmp    $0x6,%eax
  8048df3:    75 e6                   jne    8048ddb <phase_5+0x2b>
  ```

  这里是一个循环，循环变量是 %*eax*，范围从 0 到 5。

- ```
  8048ddb:    0f be 14 03             movsbl (%ebx,%eax,1),%edx
  8048ddf:    83 e2 0f                and    $0xf,%edx
  ```

  movsbl 这个语句的意思为，将位于 *Mem*[%*ebx* + %*eax*] 处地址的数据最低位取出，进行符号位扩展后送入 %*edx* 中，结合后面一句 **and** $0*xf*, %*edx*，以及 %ebx 即为输入字符串的首地址，可以发现这句话的意思是取出输入字符串第 %**eax** 位 ASCII 码的后四位。

- ```
  8048ddf:    83 e2 0f                and    $0xf,%edx
  8048de2:    0f b6 92 7c 9a 04 08    movzbl 0x8049a7c(%edx),%edx
  8048de9:    88 54 04 15             mov    %dl,0x15(%esp,%eax,1)
  ```

  这个语句将刚刚取出的值作为下标，在从 **0x8049a7c** 开始的数组中进行寻值，并将结果构成新的字符串压入栈中。于是，我们调用 gdb 命令，查看 **0x8049a7c** 地址对应的值。

  ```
  (gdb) p (char *) 0x8049a7c
  $1 = 0x8049a7c "isrveawhobpnutfgS
  ```

  这里相当于是一个密码加密的过程，即将每一个原先的字母用一个一一对应的字母替换。

- ```
  8048df5:    c6 44 24 1b 00          movb   $0x0,0x1b(%esp)
  8048dfa:    c7 44 24 04 52 9a 04    movl   $0x8049a52,0x4(%esp)
  ```
  ```

```
8048e01:       08
8048e02:       8d 44 24 15             lea    0x15(%esp),%eax
8048e06:       89 04 24                mov    %eax,(%esp)
8048e09:       e8 36 02 00 00          call   8049044 <strings_not_equal>
8048e0e:       85 c0                   test   %eax,%eax
8048e10:       74 05                   je     8048e17 <phase_5+0x67>
8048e12:       e8 bc 07 00 00          call   80495d3 <explode_bomb>
```

这里又向栈中压入了一个 **0x8049a52** 地址中的数据，并且调用 **strings_not_equal** 函数。根据之前的分析，如果传入的两个字符串不等，炸弹就会爆炸。因此，我们生成的字符串应该与 **0x8049a52** 地址中的相同。

- 于是调用 gdb，查看 **0x8049a52** 地址中的内容：

```
(gdb) p (char *) 0x8049a52
$2 = 0x8049a52 "saints"
```

- 因此，我们只需要生成"saints"这个字符串即可。根据索引表，"saints"这六个字母对应的索引值分别为 1，5，0，11，13，1。

  我们只需要输入六个 ASCII 码低 4 位为这六个值的字符即可。于是选用 **a, e, 0, k, m, a**。

输入 **ae0kma**，顺利过关。

```
I turned the moon into something I like to call a Death Star.
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
4 168
Halfway there!
5 austinpowers
So you got that one.  Try this one.
ae0kma
Good work!  On to the next...
```

7 phase_6

- 第六关的汇编代码如下：

```
08048e91 <phase_6>:
 8048e91:       83 ec 1c                sub    $0x1c,%esp
 8048e94:       c7 44 24 08 0a 00 00    movl   $0xa,0x8(%esp)
 8048e9b:       00
 8048e9c:       c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
 8048ea3:       00
 8048ea4:       8b 44 24 20             mov    0x20(%esp),%eax
 8048ea8:       89 04 24                mov    %eax,(%esp)
 8048eab:       e8 f0 fa ff ff          call   80489a0 <strtol@plt>
 8048eb0:       a3 74 c1 04 08          mov    %eax,0x804c174
 8048eb5:       c7 04 24 74 c1 04 08    movl   $0x804c174,(%esp)
 8048ebc:       e8 6d ff ff ff          call   8048e2e <fun6>
 8048ec1:       8b 40 08                mov    0x8(%eax),%eax
 8048ec4:       8b 40 08                mov    0x8(%eax),%eax
 8048ec7:       8b 40 08                mov    0x8(%eax),%eax
 8048eca:       8b 40 08                mov    0x8(%eax),%eax
 8048ecd:       8b 40 08                mov    0x8(%eax),%eax
 8048ed0:       8b 40 08                mov    0x8(%eax),%eax
 8048ed3:       8b 15 74 c1 04 08       mov    0x804c174,%edx
 8048ed9:       39 10                   cmp    %edx,(%eax)
 8048edb:       74 05                   je     8048ee2 <phase_6+0x51>
 8048edd:       e8 f1 06 00 00          call   80495d3 <explode_bomb>
 8048ee2:       83 c4 1c                add    $0x1c,%esp
 8048ee5:       c3                      ret
```

- 首先发现它调用了 **strtol** 这个函数。观察这个函数的代码发现，它需要传入三个参数，这里分别为 *%eax*，**0** 和 **0xa**。**strtol** 的主要功能可以用以下 c 语言代码表示：

```
long strtol(const char *str, char **endptr, int base) {
        long ret = 0;
        for (; *str != endptr; str++)
                ret = ret * base + *str - '0';
        return ret;
}
```

这段代码的作用就是将输入的字符串转换为对应的 base 进制数表示。

-

```
8048eb0:    a3 74 c1 04 08          mov    %eax,0x804c174
8048eb5:    c7 04 24 74 c1 04 08    movl   $0x804c174,(%esp)
8048ebc:    e8 6d ff ff ff          call   8048e2e <fun6>
8048ec1:    8b 40 08                mov    0x8(%eax),%eax
8048ec4:    8b 40 08                mov    0x8(%eax),%eax
8048ec7:    8b 40 08                mov    0x8(%eax),%eax
8048eca:    8b 40 08                mov    0x8(%eax),%eax
8048ecd:    8b 40 08                mov    0x8(%eax),%eax
8048ed0:    8b 40 08                mov    0x8(%eax),%eax
8048ed3:    8b 15 74 c1 04 08       mov    0x804c174,%edx
8048ed9:    39 10                   cmp    %edx,(%eax)
```

这里调用了函数 **fun6**。由于传入的是一个地址 **0x804c174**，因此 fun6 的返回值是固定的，因此，经过一系列跳转后, 在 **cmp %edx, (%eax)** 中，(%eax) 的值也是一定的。运用 gdb 设置断点，查看得知此时 (%eax) 的值为 268。

- 由于 %edx 的值就是我们输入的字符串，答案就唾手可得了。

输入 **268**，顺利过关。

```
I turned the moon into something I like to call a Death Star.
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
4 168
Halfway there!
5 austinpowers
So you got that one.  Try this one.
ae0kma
Good work!  On to the next...
268
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

8 secret_phase

此时的炸弹还没有拆完。根据前人的经验，这里还有一个隐藏关等着破解。

8.1 find the secret phase

- 首先考虑如何进入隐藏关。由于每次拆弹成功都需要调用 **phase_defused** 函数，想进入新的关卡，突破口肯定在这个函数。函数的代码如下：

```
1
2  08049728 <phase_defused>:
3   8049728:        83 ec 7c                sub    $0x7c,%esp
4   804972b:        65 a1 14 00 00 00       mov    %gs:0x14,%eax
5   8049731:        89 44 24 6c             mov    %eax,0x6c(%esp)
6   8049735:        31 c0                   xor    %eax,%eax
7   8049737:        c7 04 24 01 00 00 00    movl   $0x1,(%esp)
8   804973e:        e8 cb fb ff ff          call   804930e <send_msg>
9   8049743:        83 3d 8c c7 04 08 06    cmpl   $0x6,0x804c78c
10  804974a:        75 76                   jne    80497c2 <phase_defused+0x9a>
11  804974c:        8d 44 24 1c             lea    0x1c(%esp),%eax
12  8049750:        89 44 24 0c             mov    %eax,0xc(%esp)
13  8049754:        8d 44 24 18             lea    0x18(%esp),%eax
14  8049758:        89 44 24 08             mov    %eax,0x8(%esp)
15  804975c:        c7 44 24 04 cc 9d 04    movl   $0x8049dcc,0x4(%esp)
16  8049763:        08
17  8049764:        c7 04 24 90 c8 04 08    movl   $0x804c890,(%esp)
18  804976b:        e8 a0 f1 ff ff          call   8048910 <__isoc99_sscanf@plt>
19  8049770:        83 f8 02                cmp    $0x2,%eax
20  8049773:        75 35                   jne    80497aa <phase_defused+0x82>
21  8049775:        c7 44 24 04 d2 9d 04    movl   $0x8049dd2,0x4(%esp)
22  804977c:        08
23  804977d:        8d 44 24 1c             lea    0x1c(%esp),%eax
24  8049781:        89 04 24                mov    %eax,(%esp)
25  8049784:        e8 bb f8 ff ff          call   8049044 <strings_not_equal>
26  8049789:        85 c0                   test   %eax,%eax
27  804978b:        75 1d                   jne    80497aa <phase_defused+0x82>
28  804978d:        c7 04 24 e8 9a 04 08    movl   $0x8049ae8,(%esp)
29  8049794:        e8 07 f1 ff ff          call   80488a0 <puts@plt>
30  8049799:        c7 04 24 10 9b 04 08    movl   $0x8049b10,(%esp)
31  80497a0:        e8 fb f0 ff ff          call   80488a0 <puts@plt>
32  80497a5:        e8 8d f7 ff ff          call   8048f37 <secret_phase>
```

```
80497aa:    c7 04 24 48 9b 04 08    movl   $0x8049b48,(%esp)
80497b1:    e8 ea f0 ff ff          call   80488a0 <puts@plt>
80497b6:    c7 04 24 74 9b 04 08    movl   $0x8049b74,(%esp)
80497bd:    e8 de f0 ff ff          call   80488a0 <puts@plt>
80497c2:    8b 44 24 6c             mov    0x6c(%esp),%eax
80497c6:    65 33 05 14 00 00 00    xor    %gs:0x14,%eax
80497cd:    74 05                   je     80497d4 <phase_defused+0xac>
80497cf:    e8 7c f0 ff ff          call   8048850 <__stack_chk_fail@plt>
80497d4:    83 c4 7c                add    $0x7c,%esp
80497d7:    c3                      ret
```

- 观察这个函数，发现这里调用了 **secret_phase** 函数。

```
8049764:    c7 04 24 90 c8 04 08    movl   $0x804c890,(%esp)
804976b:    e8 a0 f1 ff ff          call   8048910 <__isoc99_sscanf@plt>
8049770:    83 f8 02                cmp    $0x2,%eax
8049773:    75 35                   jne    80497aa <phase_defused+0x82>
8049775:    c7 44 24 04 d2 9d 04    movl   $0x8049dd2,0x4(%esp)
804977c:    08
804977d:    8d 44 24 1c             lea    0x1c(%esp),%eax
8049781:    89 04 24                mov    %eax,(%esp)
8049784:    e8 bb f8 ff ff          call   8049044 <strings_not_equal>
8049789:    85 c0                   test   %eax,%eax
804978b:    75 1d                   jne    80497aa <phase_defused+0x82>
804978d:    c7 04 24 e8 9a 04 08    movl   $0x8049ae8,(%esp)
8049794:    e8 07 f1 ff ff          call   80488a0 <puts@plt>
8049799:    c7 04 24 10 9b 04 08    movl   $0x8049b10,(%esp)
80497a0:    e8 fb f0 ff ff          call   80488a0 <puts@plt>
80497a5:    e8 8d f7 ff ff          call   8048f37 <secret_phase>
```

这段语句是在第四关拆炸弹成功后执行，若要跳转到 **secret_phase** 语句，需要以下两个条件：

- 由 **cmp** $0x2, %*eax* 看出，需要输入两个参数
- 这里调用了 **strings_not_equal** 函数，输入的第二个参数要与 **0x8049dd2** 中的值一样。

- 于是调用 gdb，查看 **0x8049dd2** 中的值：

```
(gdb) p (char *) 0x8049dd2
$3 = 0x8049dd2 "austinpowers"
```

这样进入隐藏关的方法就很清楚了：在解锁第四关时输入 **5 austinpowers**，进入隐藏关。

8.2 solve the secret phase

- 隐藏关的代码如下：

```
08048f37 <secret_phase>:
 8048f37:       53                      push   %ebx
 8048f38:       83 ec 18                sub    $0x18,%esp
 8048f3b:       e8 d2 06 00 00          call   8049612 <read_line>
 8048f40:       c7 44 24 08 0a 00 00    movl   $0xa,0x8(%esp)
 8048f47:       00
 8048f48:       c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
 8048f4f:       00
 8048f50:       89 04 24                mov    %eax,(%esp)
 8048f53:       e8 48 fa ff ff          call   80489a0 <strtol@plt>
 8048f58:       89 c3                   mov    %eax,%ebx
 8048f5a:       8d 40 ff                lea    -0x1(%eax),%eax
 8048f5d:       3d e8 03 00 00          cmp    $0x3e8,%eax
 8048f62:       76 05                   jbe    8048f69 <secret_phase+0x32>
 8048f64:       e8 6a 06 00 00          call   80495d3 <explode_bomb>
 8048f69:       89 5c 24 04             mov    %ebx,0x4(%esp)
 8048f6d:       c7 04 24 c0 c0 04 08    movl   $0x804c0c0,(%esp)
 8048f74:       e8 6d ff ff ff          call   8048ee6 <fun7>
 8048f79:       83 f8 07                cmp    $0x7,%eax
 8048f7c:       74 05                   je     8048f83 <secret_phase+0x4c>
 8048f7e:       e8 50 06 00 00          call   80495d3 <explode_bomb>
 8048f83:       c7 04 24 2c 9a 04 08    movl   $0x8049a2c,(%esp)
 8048f8a:       e8 11 f9 ff ff          call   80488a0 <puts@plt>
 8048f8f:       e8 94 07 00 00          call   8049728 <phase_defused>
 8048f94:       83 c4 18                add    $0x18,%esp
 8048f97:       5b                      pop    %ebx
 8048f98:       c3                      ret
```

- 这里也调用了 **strtol** 函数，用处也相同：将输入的字符串转换为 10 进制整数。

```
 8048f40:       c7 44 24 08 0a 00 00    movl   $0xa,0x8(%esp)
 8048f47:       00
 8048f48:       c7 44 24 04 00 00 00    movl   $0x0,0x4(%esp)
 8048f4f:       00
 8048f50:       89 04 24                mov    %eax,(%esp)
 8048f53:       e8 48 fa ff ff          call   80489a0 <strtol@plt>
```

-

```
8048f58:        89 c3                   mov    %eax,%ebx
8048f5a:        8d 40 ff                lea    -0x1(%eax),%eax
8048f5d:        3d e8 03 00 00          cmp    $0x3e8,%eax
8048f62:        76 05                   jbe    8048f69 <secret_phase+0x32>
8048f64:        e8 6a 06 00 00          call   80495d3 <explode_bomb>
```

这里表示，我们输入的整数必须要不大于 **0x3e9**。

```
8048f69:        89 5c 24 04             mov    %ebx,0x4(%esp)
8048f6d:        c7 04 24 c0 c0 04 08    movl   $0x804c0c0,(%esp)
8048f74:        e8 6d ff ff ff          call   8048ee6 <fun7>
8048f79:        83 f8 07                cmp    $0x7,%eax
8048f7c:        74 05                   je     8048f83 <secret_phase+0x4c>
8048f7e:        e8 50 06 00 00          call   80495d3 <explode_bomb>
```

这一段表示，程序将存在 **0x804c0c0** 处的数据作为参数，传入 **fun7** 函数中。如果 **fun7** 的返回值不为 7，炸弹爆炸。

- 因此，我们的目标就是：找出使 **fun7** 的返回值为 7 的输入参数。
- **fun7** 的代码如下：

```

08048ee6 <fun7>:
8048ee6:        53                      push   %ebx
8048ee7:        83 ec 18                sub    $0x18,%esp
8048eea:        8b 54 24 20             mov    0x20(%esp),%edx
8048eee:        8b 4c 24 24             mov    0x24(%esp),%ecx
8048ef2:        85 d2                   test   %edx,%edx
8048ef4:        74 37                   je     8048f2d <fun7+0x47>
8048ef6:        8b 1a                   mov    (%edx),%ebx
8048ef8:        39 cb                   cmp    %ecx,%ebx
8048efa:        7e 13                   jle    8048f0f <fun7+0x29>
8048efc:        89 4c 24 04             mov    %ecx,0x4(%esp)
8048f00:        8b 42 04                mov    0x4(%edx),%eax
8048f03:        89 04 24                mov    %eax,(%esp)
8048f06:        e8 db ff ff ff          call   8048ee6 <fun7>
8048f0b:        01 c0                   add    %eax,%eax
8048f0d:        eb 23                   jmp    8048f32 <fun7+0x4c>
8048f0f:        b8 00 00 00 00          mov    $0x0,%eax
8048f14:        39 cb                   cmp    %ecx,%ebx
8048f16:        74 1a                   je     8048f32 <fun7+0x4c>
8048f18:        89 4c 24 04             mov    %ecx,0x4(%esp)
8048f1c:        8b 42 08                mov    0x8(%edx),%eax
```

```
8048f1f:        89 04 24                mov    %eax,(%esp)
8048f22:        e8 bf ff ff ff          call   8048ee6 <fun7>
8048f27:        8d 44 00 01             lea    0x1(%eax,%eax,1),%eax
8048f2b:        eb 05                   jmp    8048f32 <fun7+0x4c>
8048f2d:        b8 ff ff ff ff          mov    $0xffffffff,%eax
8048f32:        83 c4 18                add    $0x18,%esp
8048f35:        5b                      pop    %ebx
```

这段代码的等价 c 语言表示如下：

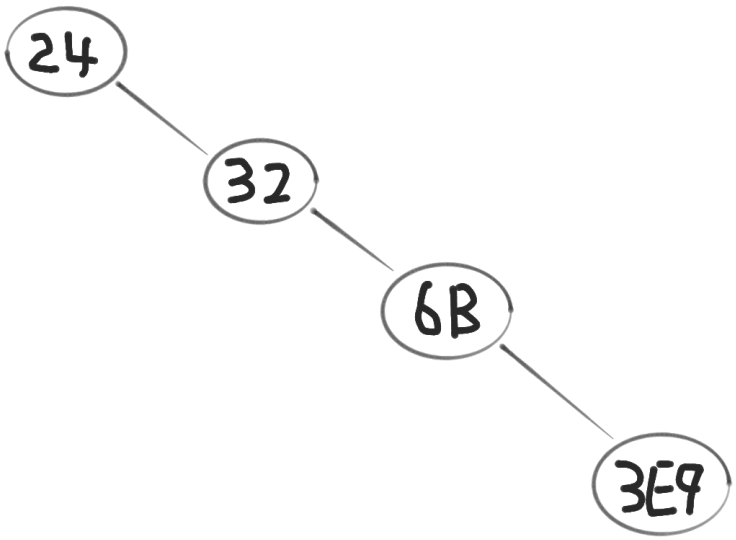
```c
int fun7(int *a, int b) {
        if (a == NULL)
                return -1;
        int ret;
        if (*a > b) {
                ret = fun7(*(a + 4), b);
                ret = ret * 2;
        }
        else
        if (*a == b) {
                ret = 0;
        }
        else {
                int ret = fun7(*(a + 8), b);
                ret = ret * 2 + 1;
        }
        return ret;
}
```

这个可以理解为一个在一个二叉查找树中查找值的过程，从根开始，如果向右走，当前位就是 1，向左走就是 0。由于输出需要为 7，因此需要向右走 3 次。

- 首先通过 gdb 得出这一条路上的所有节点：

```
(gdb) p /x *0x804c0c0
$4 = 0x24
(gdb) p /x *(0x804c0c0 + 8)
$5 = 0x804c0d8
(gdb) p /x *0x804c0d8
$6 = 0x32
(gdb) p /x *(0x804c0d8 + 8)
$7 = 0x804c108
(gdb) p /x *0x804c108
$8 = 0x6b
(gdb) p /x *(0x804c108 + 8)
$9 = 0x804c168
(gdb) p /x *0x804c168
$10 = 0x3e9
```

- 于是得出对应的二叉查找树：



- 所以我们输入的值就是 **0x3e9**，转换为十进制就是 1001。

输入 1001，问题解决。

```
I turned the moon into something I like to call a Death Star.
Phase 1 defused. How about the next one?
1 2 6 24 120 720
That's number 2.  Keep going!
4 168
Halfway there!
5 austinpowers
So you got that one.  Try this one.
ae0kma
Good work!  On to the next...
268
Curses, you've found the secret phase!
But finding it and solving it are quite different...
1001
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
Your instructor has been notified and will verify your solution.
```

9 conclusion

这次的 Bomb Lab 总计花了我 7 个小时，从晚上 6 点拆到凌晨 1 点，才算大功告成。不可否认，这个 lab 对我对于汇编的理解起到了很强的巩固作用。从一开始面对冗长汇编代码的手足无措，到一步步将炸弹抽丝剥茧，慢慢拆弹，到最后将隐藏关解决，这其中的每一关对我都是新的挑战。

这次 lab 的覆盖范围非常广泛：第一关的函数调用，第二关的数组的存储，第三关的 switch 跳转表的实现。。几乎每一关的知识点都不重合，设计者的用心良苦可见一斑。

同时，通过这次 lab，我也系统学习了 gdb 的使用。在平常的编程环境中，我们常常只需要在 IDE 中用鼠标设置断点，点开 watch 窗口，就可以轻松享用 IDE 的调试功能，却很少想过这些调试功能都是怎么实现的。通过这次一步步的手动输入 gdb 命令，我对于 IDE 的调试功能有了一个粗浅的了解。当然，对于主流 IDE 调试功能的更深层次的实现，是我接下来需要探索的。

陆游说过 :"纸上得来终觉浅，绝知此事要躬行。"对于 CSAPP 这门课来说，更是如此。通过这次 lab，通过大量的读代码，gdb 调试，我对于代码的底层实现有了新的认识。我想，这就是老师布置这次 lab 的用意之一吧。