華中科技大學

课 程 实 验 报 告

课程名称： 计算机系统基础

专业班级： CS2207

学　　号： U202215554

姓　　名： 付名扬

指导教师： 张宇

报告日期： 2024 年 3 月 20 日

计算机科学与技术学院

目录

I. Binary Bombs: 二进制炸弹

1.1 实验概述以及设计

- **实验目的:** 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。
- **实验目标:** 一个“**binary bombs**”（二进制炸弹，下文将简称为炸弹）是一个 Linux 可执行 C 程序，**包含了 6 个阶段（phase1~phase6）。**炸弹运行的每个阶段要求**输入一个特定的字符串**，输入符合程序预期的输入，该阶段的炸弹就被“拆除”，否则炸弹“爆炸”并打印输出 **"BOOM!!!"**字样。实验的目标是拆除尽可能多的炸弹层次。
- **实验内容:** 本次实验包含六个阶段，根据《Lab2：Binary Bomb 实验任务书》(以下简称实验 2 任务书)，分别考察了机器级语言程序的一个不同方面，难度逐级递增。各部分涉及的知识点如表 1.1 所示。

表 1.1 *考察的机器级语言程序的不同方面*

Phase_1	Phase_2	Phase_3	Phase_4	Phase_5	Phase_6
字符串	循环	条件分支	递归与栈	指针	链表结构

**NOTE:实验语言:C 语言;实验环境:Linux*

为了完成二进制炸弹拆除任务，需要使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件，并单步跟踪调试每一阶段的机器代码，从中理解每一汇编语言代码的行为或作用，进而设法“推断”出拆除炸弹所需的目标字符串。

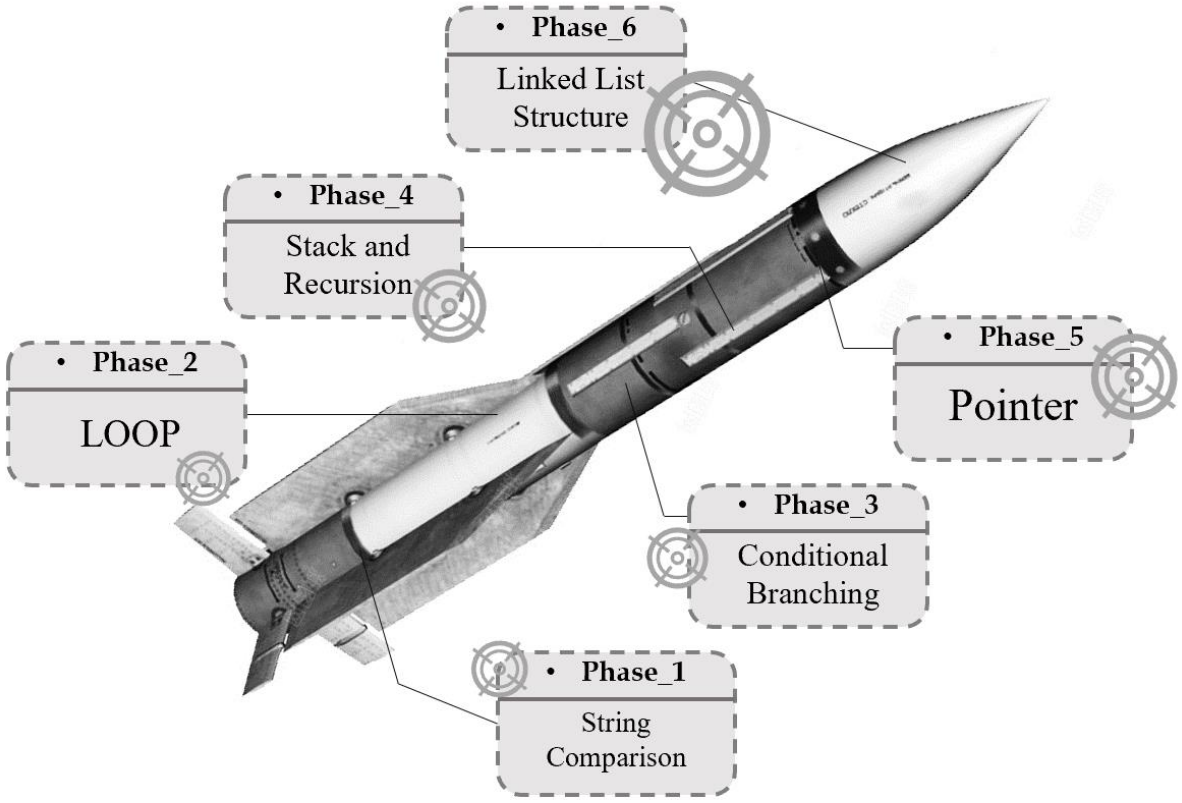

图 1.1 *有关二进制炸弹的概念美术。这里特地将炸弹分割为 6 个部分，并且将高阶段的部分放在尖端的位置上，因此可以生动形象地展示拆弹的层层递进的过程以及难度序列的递增。设计本图的目的是更好地理解实验的过程和结构。*

在进行本实验之前，需要进行一系列的准备工作。这里似乎并不要求学生进行呈现，但是考虑到实验的严谨性，笔者认为这一部分的工作需要进行描述。

首先 (实际上是在实验一之前。但是根据任务安排，实验一不需要进行报告) 利用 VMware 软件搭建了处于 Linux 环境下的虚拟机，在安装了 VMware Tools 之后，可以实现虚拟机与宿主机之间信息和文件的实时传递：使得宿主机上的文件系统在虚拟机内可访问，同时也支持在虚拟机内部访问宿主机上的文件系统，实现简单且高效的文件共享。

然后，在本次实验中 (即实验二，二进制炸弹)，每个学生将得到一个**不一样的的 binary bomb 二进制可执行程序及其相关文件，其中包含 bomb(可执行程序)，bomb.c(main 函数)**。这些文件将以学号进行打包，目的是为了使得每个同学具有不一样的测试结果，可能是为了防止舞弊行为。不过，在本次实验中，实际上同学们可以任意选取自己的文件包进行操作和实验。因此，笔者选取了编号为 **U201614500** 的文件包。

这些准备工作完成之后，实验正式开始的操作步骤如下：

① **打开命令行窗口，调用 objdump –D bomb > disassemble.txt** 可以将 bomb 进行反汇编，使得其汇编语言代码被保存在 txt 文本 disassemble.txt 中；

② **打开文本 disassemble.txt**，查找对应 phase_i (i = 1, 2, 3, 4, 5, 6)，找到不同阶段的相关函数；

③ 通过分析汇编语言代码，**推测每个阶段需要输入的字符串；**

接下来，运行./bomb：该程序打印出欢迎信息后，期待按行输入每一阶段用来拆除炸弹的字符串，并根据当前输入的字符串决定是通过相应阶段还是炸弹爆炸导致任务失败。也可将拆除每一阶段炸弹的字符串按行组织在一个文本文件中（比如：ans.txt），程序会自动读取文本文件中的字符串，并依次检查对应每一阶段的字符串来决定炸弹拆除成败。

该实验的总体过程（实际上是整个准备工作的过程）如图 1.2 所示。

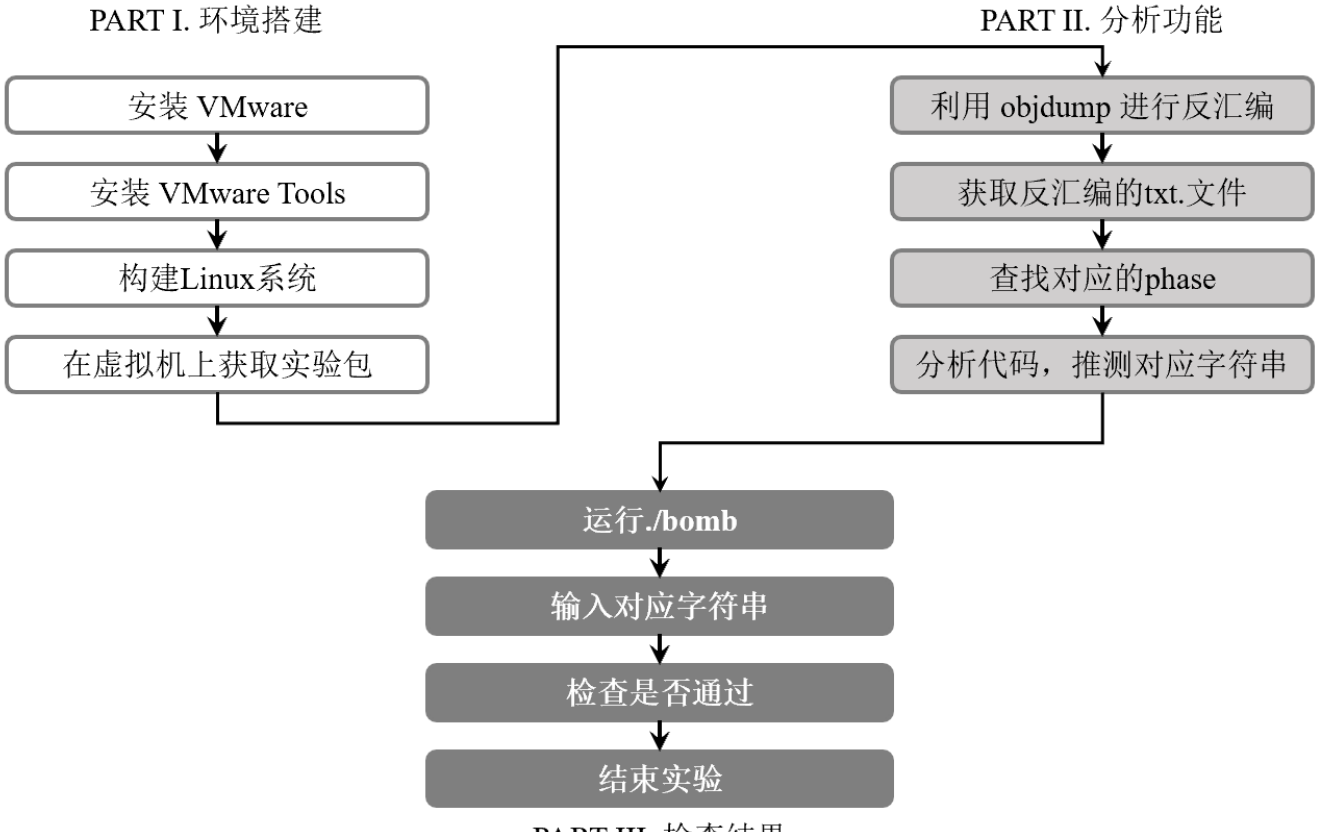

图 1.2 *该实验过程的简要概况*

1.2 Phase_1: String Comparison – 字符串比较

首先，需要在 disassemble.txt 中定位 <phase_1> 阶段的位置，如图 1.3 所示。

```
08048b42 <phase_1>:
 8048b42:       83 ec 14                sub    $0x14,%esp
 8048b45:       68 c4 a0 04 08          push   $0x804a0c4
 8048b4a:       ff 74 24 1c             push   0x1c(%esp)
 8048b4e:       e8 76 05 00 00          call   80490c9 <strings_not_equal>
 8048b53:       83 c4 10                add    $0x10,%esp
 8048b56:       85 c0                   test   %eax,%eax
 8048b58:       75 04                   jne    8048b5e <phase_1+0x1c>
 8048b5a:       83 c4 0c                add    $0xc,%esp
 8048b5d:       c3                      ret
 8048b5e:       e8 5b 06 00 00          call   80491be <explode_bomb>
 8048b63:       eb f5                   jmp    8048b5a <phase_1+0x18>
```

图 1.3 *对 <phase_1> 进行定位的结果*

这一部分代码的功能分析如下：

- 通过 **sub $0x14, %esp** 指令将栈指针 **%esp** 向下移动 0x14（20）个字节，为后续的参数和局部变量分配空间。
- 使用 **push $0x804a0c4** 指令将常数值 **0x804a0c4** 压入栈中，作为第一个参数。
- 接着，使用 **push 0x1c(%esp)** 指令将栈顶地址偏移 0x1c（28）处的值压入栈中，作为第二个参数。推测这可能是需要我们进行输入的一个字符串。
- 调用函数 **strings_not_equal**，对结果进行判断。

之后，按同样的方法，在 main 函数中找到调用函数<phase_1>的部分，如图 1.4 所示。

```
 8048a47:       e8 f6 00 00 00          call   8048b42 <phase_1>
 8048a4c:       e8 cc 08 00 00          call   804931d <phase_defused>
 8048a51:       c7 04 24 74 a0 04 08    movl   $0x804a074,(%esp)
 8048a58:       e8 63 fd ff ff          call   80487c0 <puts@plt>
 8048a5d:       e8 bc 07 00 00          call   804921e <read_line>
 8048a62:       89 04 24                mov    %eax,(%esp)
```

图 1.4 *在 main 函数中对 <phase_1> 进行定位的结果*

- **call 804921e <read_line>**：调用函数 read_line，该函数的返回值会存储在 %eax 寄存器中。
- **mov %eax, (%esp)**：将 %eax 寄存器中的值（即函数 read_line 的返回值）移动到栈顶位置（(%esp)）。这段代码的作用是调用函数 read_line 并将其返回值放置在栈顶位置。

可以推测出，系统给出的密码字符串被存储在地址 **0x804a0c4** 里面。接下来将采用 GDB 进行调试，从而找到这个地址代表的字符串的内容：（见下页所示）

输入指令：(gdb) x/40x 0x804a0c4

0x804a0c4: 0x6e656857 0x67204920 0x61207465 0x7972676e
0x804a0d4: 0x724d202c 0x6942202e 0x656c6767 0x726f7773
0x804a0e4: 0x67206874 0x20737465 0x65737075 0x00002e74
0x804a0f4: 0x21776f57 0x756f5920 0x20657627 0x75666564
0x804a104: 0x20646573 0x20656874 0x72636573 0x73207465
0x804a114: 0x65676174 0x64250021 0x20632520 0x00006425
0x804a124: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a134: 0x00000000 0x00000000 0x00000000 0x08048c1d
0x804a144: 0x08048c3f 0x08048c61 0x08048c83 0x08048ca2
0x804a154: 0x08048cbd 0x08048cd8 0x08048cf3 0x0000000a

再输入指令**(gdb) x/2s 0x804a0c4**，得到内容**:**

When I get angry, Mr. Bigglesworth gets upset.

该字符串便是本题目的答案。本题的解题思路的流程如图 1.5 所示。

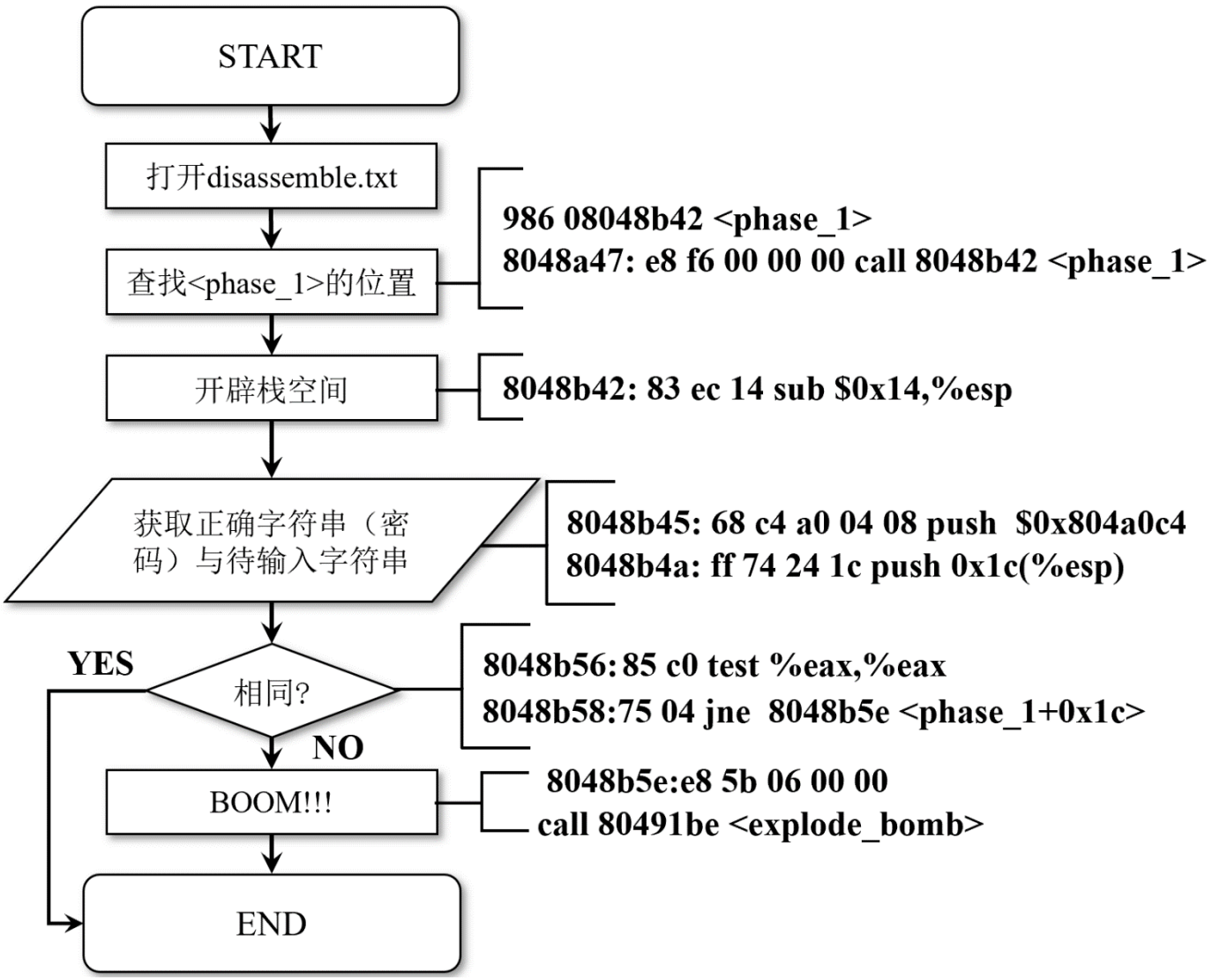

图 1.5 *<phase_1>阶段的流程图*

1.3 Phase_2: Loop – 循环

对<phase_2>进行定位的过程和方法与 1.2 小节中完全一致，因此本处和以后不再赘述。以下将简述该函数的整体思路。过程如图 1.6 所示。

a) **开辟栈空间**(栈针是 esp)，留出内存，为后续操作做准备。这一部分是初始化操作。
b) **读取用户输入的第一个数字**，并保存在栈针偏移 4 个字节的位置，即地址 0x4(%esp)中。之后将输入的值**与常量 0x1 进行比较**，如果**不相同，炸弹爆炸**。可以推断出：第一个输入的数是 1。
c) **将地址 0x4(%esp)保存在寄存器 ebx 中，将地址 0x18(%esp)保存在寄存器 esi 中。**0x4 的十进制数字是 4，而 0x18 的十进制数字是 24。后者是前者的 6 倍，可以认为是读取 6 个数字。
d) 地址 0x4(%esp)指向的数字是 0x1(刚刚读取的)，现在将这个数字保存至寄存器 eax 中。之后，**对寄存器 eax 的值**进行对自身的相加操作，即**实现二倍乘法**。之后 eax 保存的数字为 0x2。
e) 将此时 ebx 寄存器地址再向后偏移 4 个字节的地址 **0x4(%ebx)内部的存储数与刚刚完成自增的 eax 的存储数进行比较**，这里 0x4(%ebx)可以推断出是需要输入的数字。如果不相等，说明不合法，炸弹爆炸。
f) 最后将 **ebx 进行长度为 4 的地址偏移**，使得其保存的内容为之前的地址 0x4(%ebx)，再把这个地址的值与 esi 的保存的地址 0x18(%esp)进行比较，如果相等，说明已经输入了 6 个数字。如果不相等，则重复以上流程一直到相等。

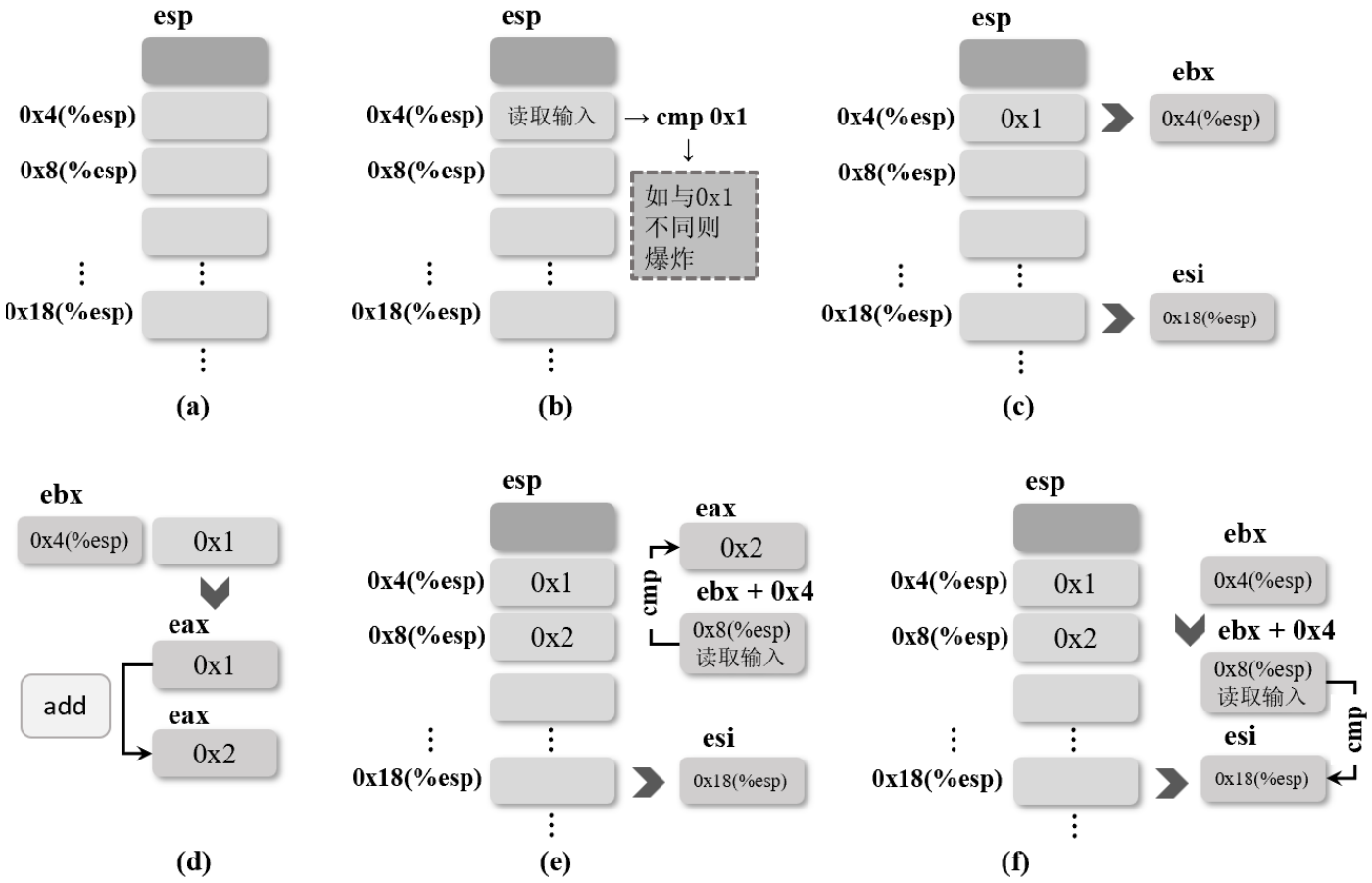

图 1.6 *<phase_2>阶段过程图解*

通过上方的分析，函数的功能可以简要概况为：通过开辟栈空间并保存用户输入的数字，进行初始化操作。然后，逐个比较输入的数字与预设常量的值，如果不符合条件则触发炸弹爆炸。接着，使用寄存器保存地址偏移后的数字，并进行相应的比较操作，以判断输入是否合法。最终，通过不断重复以上流程，确定是否已输入了六个数字。该函数展示了对输入数据的处理、逻辑判断和循环比较的过程，体现了程序的控制流程和条件执行逻辑。

考虑到该函数涉及到循环流程控制，因此如果单单采用图解的模式可能难以完整呈现其逻辑。而流程图非常适合展现循环的条件判断。因此绘制了<phase_2>函数的主要逻辑的流程图。

该函数的关键逻辑的代码以及相应的流程如图 1.7 所示。

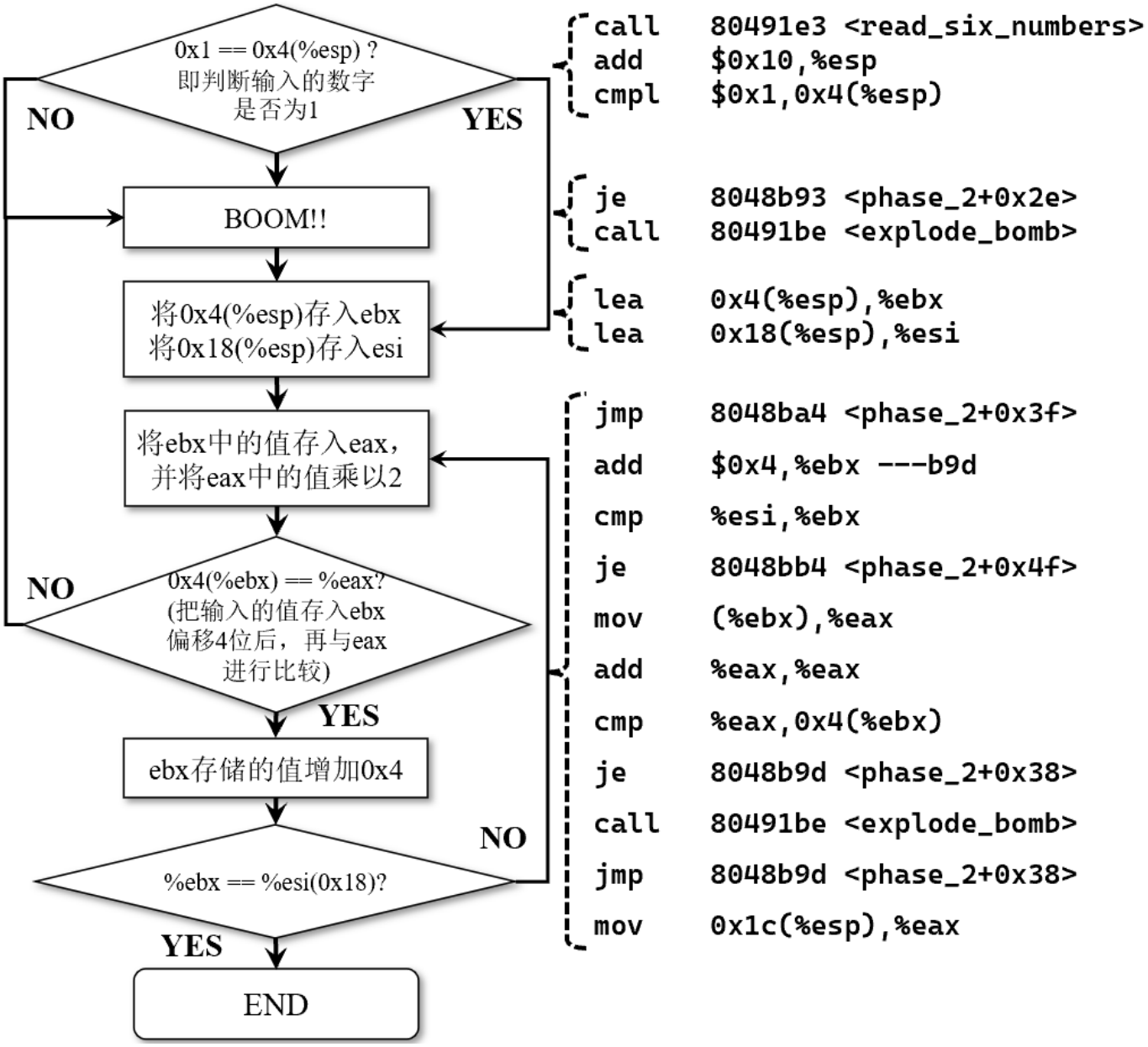

图 1.7 *<phase_2>阶段的流程图。这里采用了左边流程-右边代码的形式，目的是方便进行实时审阅和浏览。但考虑到代码的字面顺序和函数的逻辑顺序可能不一样，因此参考价值有限。*

1.3 Phase_3: Conditional Branching – 条件分支

在对 <phase_3> 进行定位之后，发现该部分的汇编代码相比 <phase_1>、<phase_2> 要长上许多。且其中有大量重复性比较高的部分，推测这是进入不同状态的分支。这些被推测为分支的部分如图 1.8 所示。

```
8048c29: 00
8048c2a: 0f 84 e8 00 00 00    je     8048d18 <phase_3+0x14c>
8048c30: e8 89 05 00 00       call   80491be <explode_bomb>
8048c35: b8 78 00 00 00       mov    $0x78,%eax
8048c3a: e9 d9 00 00 00       jmp    8048d18 <phase_3+0x14c>
8048c3f: b8 76 00 00 00       mov    $0x76,%eax
8048c44: 81 7c 24 08 b1 02 00 cmpl   $0x2b1,0x8(%esp)
8048c4b: 00
8048c4c: 0f 84 c6 00 00 00    je     8048d18 <phase_3+0x14c>
8048c52: e8 67 05 00 00       call   80491be <explode_bomb>
8048c57: b8 76 00 00 00       mov    $0x76,%eax
8048c5c: e9 b7 00 00 00       jmp    8048d18 <phase_3+0x14c>
8048c61: b8 6d 00 00 00       mov    $0x6d,%eax
8048c66: 81 7c 24 08 bc 00 00 cmpl   $0xbc,0x8(%esp)
8048c6d: 00
8048c6e: 0f 84 a4 00 00 00    je     8048d18 <phase_3+0x14c>
8048c74: e8 45 05 00 00       call   80491be <explode_bomb>
8048c79: b8 6d 00 00 00       mov    $0x6d,%eax
8048c7e: e9 95 00 00 00       jmp    8048d18 <phase_3+0x14c>
8048c83: b8 73 00 00 00       mov    $0x73,%eax
8048c88: 81 7c 24 08 a0 00 00 cmpl   $0xa0,0x8(%esp)
```

```
8048c8f: 00
8048c90: 0f 84 82 00 00 00    je     8048d18 <phase_3+0x14c>
8048c96: e8 23 05 00 00       call   80491be <explode_bomb>
8048c9b: b8 73 00 00 00       mov    $0x73,%eax
8048ca0: eb 76                jmp    8048d18 <phase_3+0x14c>
8048ca2: b8 64 00 00 00       mov    $0x64,%eax
8048ca7: 81 7c 24 08 ff 01 00 cmpl   $0x1ff,0x8(%esp)
8048cae: 00
8048caf: 74 67                je     8048d18 <phase_3+0x14c>
8048cb1: e8 08 05 00 00       call   80491be <explode_bomb>
8048cb6: b8 64 00 00 00       mov    $0x64,%eax
8048cbb: eb 5b                jmp    8048d18 <phase_3+0x14c>
8048cbd: b8 6f 00 00 00       mov    $0x6f,%eax
8048cc2: 81 7c 24 08 c7 02 00 cmpl   $0x2c7,0x8(%esp)
8048cc9: 00
8048cca: 74 4c                je     8048d18 <phase_3+0x14c>
8048ccc: e8 ed 04 00 00       call   80491be <explode_bomb>
8048cd1: b8 6f 00 00 00       mov    $0x6f,%eax
8048cd6: eb 40                jmp    8048d18 <phase_3+0x14c>
8048cd8: b8 72 00 00 00       mov    $0x72,%eax
8048cdd: 81 7c 24 08 9d 03 00 cmpl   $0x39d,0x8(%esp)
```

图 1.8 *<phase_3>的汇编语言代码，有部分省略，只展示了这些类似于条件分支的高重复度的代码。*

接下来，对代码进行逐步分析。

```
8048bea: 68 1a a1 04 08       push   $0x804a11a
```

初始化时，首先将 0x804a11a 入栈。推测这是需要输入的部分。利用 gdb 进行调试，输入

```
(gdb) x/s 0x804a11a
```

得到的调试结果为

```
0x804a11a:  "%d %c %d"
```

因此，推测需要输入 3 次，类型分别是：有符号整数、字符、有符号整数。

在完成前方的初始化以及提前优化过程后，首先发现

```
8048c00: 83 7c 24 04 07       cmpl   $0x7,0x4(%esp)
8048c05: 0f 87 03 01 00 00    ja     8048d0e <phase_3+0x142>
…
8048d0e: e8 ab 04 00 00       call   80491be <explode_bomb>
```

该部分可以理解为：将 0x7 与栈针偏移 4 个字节的地址里的值进行比较，如果 0x7 > 0x4(%eax)，则跳转到地址 0x8048d0e 的指令，这个指令是函数 <explode_bomb> ，即炸弹爆炸。因此推断第一个需要输入的有符号整数 ≤ 7。

之后，执行了如下指令

```
8048c0b: 8b 44 24 04          mov    0x4(%esp),%eax
8048c0f: ff 24 85 40 a1 04 08 jmp    *0x804a140(,%eax,4)
```

mov 0x4(%esp), %eax：这条指令将栈指针偏移 4 个字节处的值加载到寄存器 **%eax** 中。**0x4(%esp)** 表示栈指针 **%esp** 向上偏移 4 个字节处的地址，将这个地址处的值加载到 **%eax** 寄存器中。**jmp *0x804a140(,%eax,4)**：这条指令是间接跳转指令，它通过寄存器 **%eax** 中的值进行跳转。具体来说，它会将 **0x804a140** 地址处的内容加上 **%eax** 寄存器中的值乘以 4（因为是按照 4 字节的偏移）得到的新地址作为跳转目标，然后执行跳转操作。**这一部分实际上已经可以看出来条件分支的结构：即跳转的位置随着输入的值改变而改变。**

接下来便进入了图 1.8 所示的条件分支里。推测可以允许输入的第一个有符号整数的取值集合为{0, 1, 2, 3, 4, 5, 6}，但在进入分支之前，首先需要留意*0x804a140 的意义。我们需要先看看这个地址里面到底放了什么。因此利用 gdb 进行调试。首先考虑输入的值为 0x0 的情况。输入的指令为

```
(gdb) p/x *0x804a140
```

获得的调试结果为

```
$1 = 0x8048c1d
```

因此，从地址 0x8048c1d 便开始执行输入值为 0x0 时的后续指令。掌握了这个规律，后面的取值便可以利用 gdb 进行快速地寻址了。可以通过汇编语言代码的结构，很容易地推断出取值为其他数时的分支所处位置。比如，希望查看输入值为 0x2 时的后续指令，那么只需要把指令(gdb) p/x *0x804a140 的地址直接加上 0x8 即可。即输入的是(gdb) p/x *0x804a148.

基于上述的分析，确定本函数实际上执行了一个 Switch 语句，每一个不同的 case 有不同的正确答案。这里将选取输入数字为 0x0 时的情况，进行进一步分析。

首先，根据上方获得的调试结果，定位到地址 0x8048c1d 部分的指令

```
8048c1d: b8 78 00 00 00       mov    $0x78,%eax
8048c22: 81 7c 24 08 66 01 00 cmpl   $0x166,0x8(%esp)
```

mov $0x78, %eax：这条指令将立即数 **0x78** 移动到寄存器 **%eax** 中，即将 **%eax** 的值设置为 **0x78**。**cmpl $0x166, 0x8(%esp)**：这条指令比较了栈指针 **%esp** 向上偏移 8 个字节处的值和立即数 **0x166** 的大小关系。**0x8(%esp)** 表示栈指针 **%esp** 向上偏移 8 个字节处的地址，比较这个地址处的值和立即数 **0x166**。

这里，可以推断第二个输入的数据的机器数为 0x78。在类型为字符的条件下，其值为 ’x’；而下方的 cmpl 语句比较的值为 0x166，换算为有符号整数位 358。因此，推断其中一个正确答案为

0 x 358

同理，可以得到其他情况的答案。在符合要求的取值集合里的答案如表 1.2 所示。

表 1.2 *<phase_3>不同输入的答案*

CASE	%c	字符	%d	带符号整数	ANS
0x0	**0x78**	**x**	**0x166**	**358**	**0 x 358**
0x1	**0x76**	**V**	**0x2b1**	**689**	**1 v 689**
0x2	**0x6d**	**m**	**0xbc**	**188**	**2 m 188**
0x3	**0x73**	**s**	**0xa0**	**160**	**3 s 160**
0x4	**0x64**	**d**	**0x1ff**	**511**	**4 d 511**
0x5	**0x6f**	**o**	**0x2c7**	**711**	**5 o 711**
0x6	**0x72**	**r**	**0x39d**	**925**	**6 r 925**

该函数实现了类似于 if-else/switch 语句的功能。可以将 Switch 语句与电路中的开关进行类比，这有助于理解其功能和工作原理：

- **开关功能类比：** Switch 语句就像电路中的开关一样，可以根据开关的位置（开或关）选择不同的电路路径。同样地，Switch 语句根据表达式的值（或者说开关的状态）选择不同的代码执行路径。
- **多路选择类比：** 在电路中，开关可以连接到不同的电路分支，从而实现多路选择。类似地，Switch 语句可以连接到多个 case 标签，根据表达式的值选择不同的代码分支执行。
- **默认路径类比：** 在电路中，如果没有任何开关被打开，电流通常会流向一个默认的路径。类似地，Switch 语句中的默认分支就像一个备用路径，用于处理所有 case 条件都不匹配的情况。
- **执行顺序类比：** 在电路中，电流会根据开关的位置顺序流向不同的部件。在 Switch 语句中，程序会根据 case 标签的顺序依次比较，直到找到与表达式值匹配的标签为止，并执行对应的代码。

最后，根据本函数的分支结构，采用类似于绘制电路图的方法提供了概念图。如图 1.9 所示。

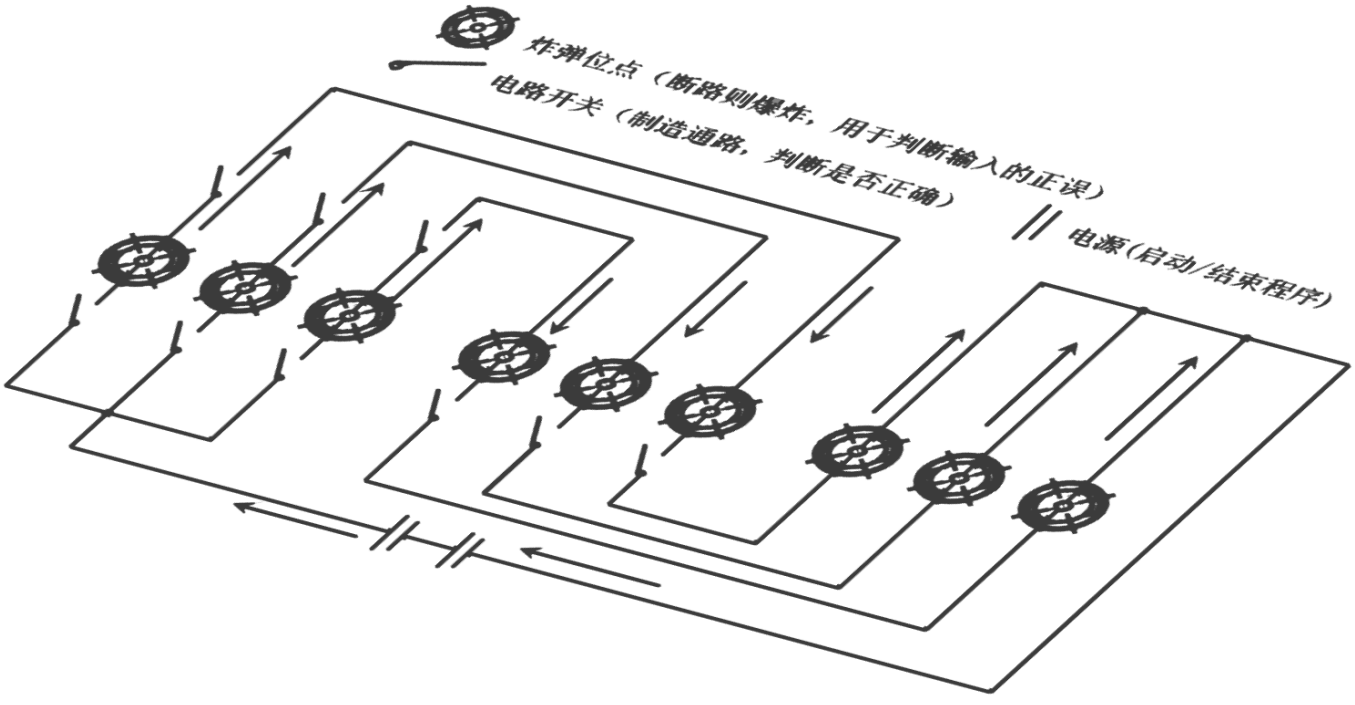

图 1.9 *<phase_3>的概念美术。这里把每一个“小型炸弹”设置为“断路则爆炸”的条件，每一个开关的闭合设置为“输入正确则闭合”。这样使得答案有多个，但每一组答案都必须完全正确才可通关。*

1.5 Phase_5: Pointer – 指针

按同样的方法对 <phase_5> 函数定位后，根据指令 **8048e09:68afa20408 push $0x804a2af** 可知，这里存储了读取的相关信息。可采用 gdb 进行调试，输入

```
(gdb) x/s 0x804a2af
```

可得

```
0x804a2af:  "%d %d"
```

这表明，输入的将会是两个带符号整数。

现在查看刚刚输入的两个数字读取的位置。首先，在函数 <phase_5> 出输入指令 **(gdb) b phase_5** ，在函数开始的位置设置断点。输入前方四个阶段的正确答案之后，我们来到 <phase_5> 的位置。暂时不关心答案的正确与否，可以先随意输入两个带符号整数。笔者在实际操作中选取了数字 11 与 12（十进制）。

利用断点开始的调试功能，持续执行“ni”指令进行逐步操作，并利用“i r”关注各个寄存器的存放的值。发现，在执行指令

```
8048e1f: 8b 44 24 04          mov    0x4(%esp),%eax
```

时，寄存器 eax 存储的值变为了刚刚输入的第一个数字：11。因此判定地址 **0x4(%esp)**存储了输入的第一个数字。

考虑利用这个地址“顺藤摸瓜”。首先查看栈顶 esp 的地址为 **0xffffd010**，则第一个数字 11 的地址为 **0xffffd014**。那么，猜想第二个数的地址为 **0x8(%esp)，即 0xffffd018**。借助此猜想，输入指令

```
(gdb) x/d 0xffffd018
```

从而得到调试结果为

```
0xffffd018: 12
```

12 正是刚刚输入的第二个数。因此猜想是正确的。

接下来，观察如下指令

```
8048e23: 83 e0 0f             and    $0xf,%eax
8048e26: 89 44 24 04          mov    %eax,0x4(%esp)
8048e2a: 83 f8 0f             cmp    $0xf,%eax
8048e2d: 74 2e                je     8048e5d <phase_5+0x6d>
…
8048e5d: e8 5c 03 00 00       call   80491be <explode_bomb>
```

这几条汇编指令的作用如下：

- **and $0xf, %eax**：将寄存器 eax 的值与 15 进行按位与运算，保留低四位，高位清零。
- **mov %eax, 0x4(%esp)**：将处理后的 eax 值存储到堆栈中偏移量为 4 的位置。
- **cmp $0xf, %eax**：比较处理后的 eax 值是否等于 15。
- **je 8048e5d <phase_5+0x6d>**：如果相等，则跳转到地址 0x8048e5d 处，炸弹爆炸。

这里实际上是对第一个输入的数字进行了模 16 的运算后再与 15 进行比较，如果模 16 运算后它为 15，则炸弹爆炸。因此，输入的第一个数字在进行模 16 运算后应该满足其值小于等于 14。

之后程序又引入两个清零后的寄存器，推测是计数器或累加器。指令为

```
8048e2f: b9 00 00 00 00       mov    $0x0,%ecx
8048e34: ba 00 00 00 00       mov    $0x0,%edx
```

引入清零的寄存器之后，紧接着执行

```
8048e39: 83 c2 01             add    $0x1,%edx
8048e3c: 8b 04 85 60 a1 04 08 mov    0x804a160(,%eax,4),%eax
8048e43: 01 c1                add    %eax,%ecx
8048e45: 83 f8 0f             cmp    $0xf,%eax
8048e48: 75 ef                jne    8048e39 <phase_5+0x49>
```

从 8048e48 部分看出，在满足寄存器 eax 里的值≠15 时，将会不断执行此代码块一直到其满足为止。与此同时，寄存器 edx 的值不断以 1 为增量去累加，而寄存器 ecx 的值不断加入寄存器 eax 里的值。因此 edx 是一个计数器，而 ecx 是一个累加器。

这里，最关键的部分便是指令 **mov　　0x804a160(,%eax,4),%eax**，这一指令的含义为：将地址 0x804a160＋4＊R[eax] 所指向的值，赋值给寄存器 eax，即 R[eax] ← M(0x804a160＋4＊R[eax])。这里特地选取常数 4，推测是整数类型的偏移。因此可以推测地址 0x804a160 开始，存放了一连串整数数组。

那么，R[eax] ← M(0x804a160＋4＊R[eax])的含义便再明显不过了：将 eax 的值作为下一次的下标，将这个数组中此下标对应的值传入 eax 中。由于最开始 eax 只可以取 0～14，且最后一定变成 15（否则不可能跳出循环），因此只需要查看该数组的前 16 个值，即下标从 0 开始一直取到 15。在 gdb 中输入调试信息

```
(gdb)  p *0x804a160@16
```

得到

```
$1 = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5}
```

因此，该猜想是正确的。该地址存放的数组长度可以理解为 16，而 eax 存储其中的一个下标。该数组的信息如图 1.11 所示。

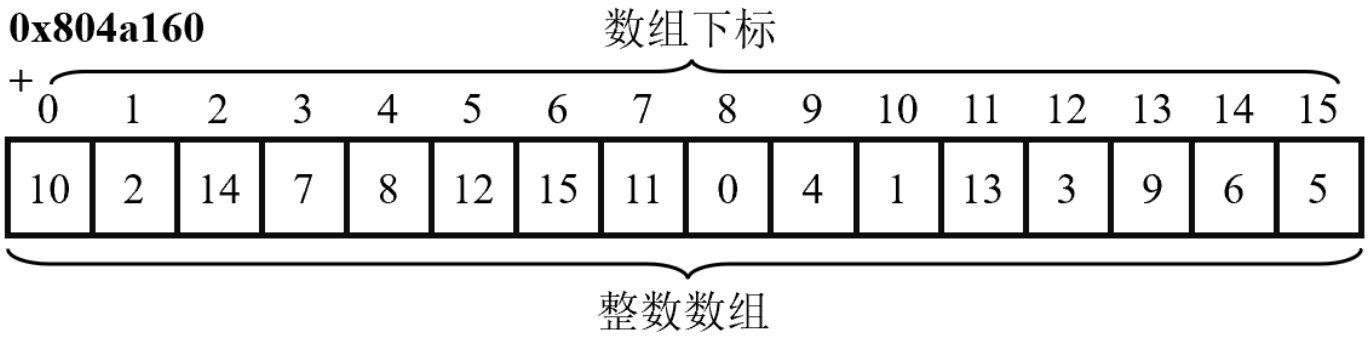

图 1.11 *以地址 0x804a160 为起始的数组*

此外，还有一个有趣的发现：数组的各个元素都互不相同，且取值也在 0~15 之间，与其下标构成了双射关系。这个关系如图 1.12 所示。

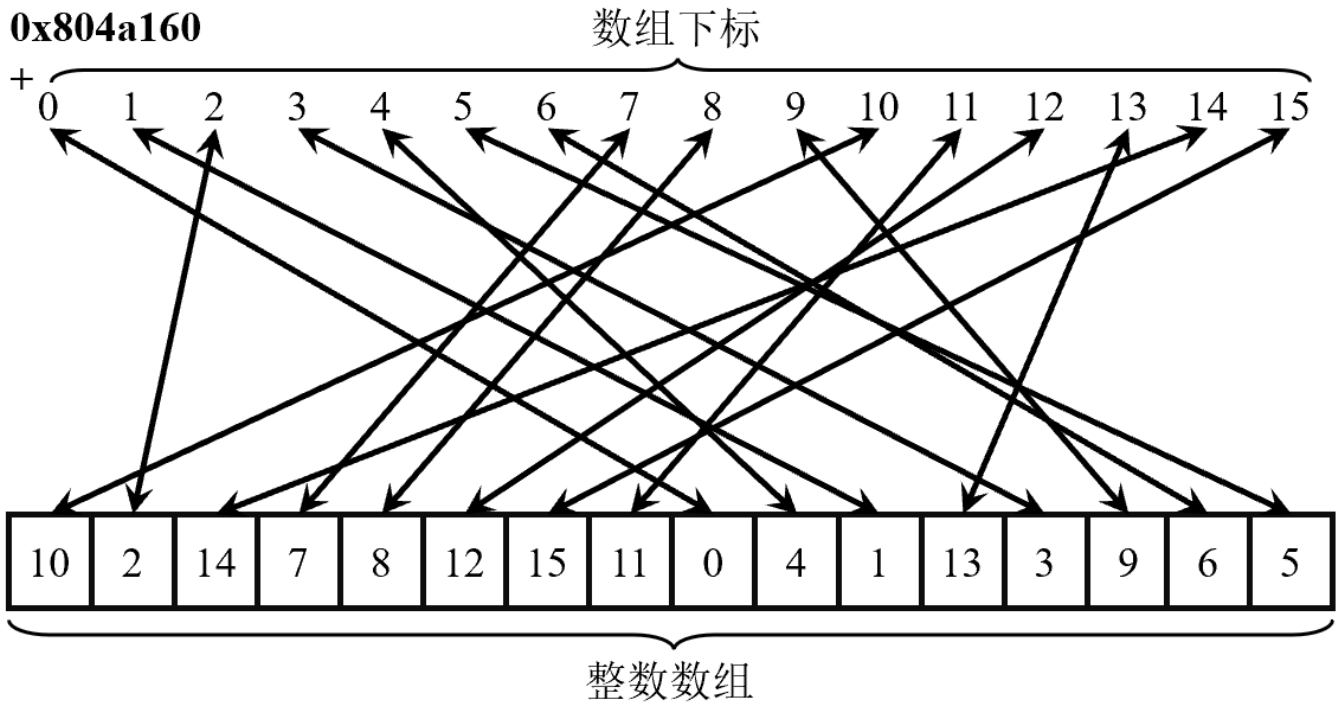

图 1.12 *数组下标以及其下标对应元素的双射关系*

因此，这里不存在死循环的情况。最后总可以跳转到代码块结束。

在这部分的功能完成之后，执行指令

```
8048e52: 83 fa 0f              cmp    $0xf,%edx
8048e55: 75 06                 jne    8048e5d <phase_5+0x6d>
8048e57: 3b 4c 24 08           cmp    0x8(%esp),%ecx
8048e5b: 74 05                 je     8048e62 <phase_5+0x72>
8048e5d: e8 5c 03 00 00        call   80491be <explode_bomb>
```

可以认为 8048e62 之后是程序结束

首先，比较了 0xf（即 15）与计数器 edx 的值。edx 是循环的次数，每循环一次便自增 1，而如果不相同则炸弹爆炸。因此上方循环必须执行 15 次，多一次和少一次都不行。

假设确实完成了 15 次循环后才退出，那么将对寄存器 ecx 存储的值与地址 0x8(%esp)中的值进行比较，实际上就是与输入的第二个数字进行比较。如果相同，那么程序结束；否则炸弹爆炸。

寄存器 ecx 的值是 15 次循环中每次对 eax 里的值进行累加的结果。因此应该是一个和数，它记录了 eax 在 15 次改变的过程中，每一次相加到最后的总和。因此，想要输入正确答案，**第一步是需要看看到底选取哪一个合适的下标，才可以使得程序在 15 次循环后恰好退出；第二步是计算出 15 次循环中，每一次涉及的下标对应的数组的值的总和是多少**。

① 计算合适的下标

由之前推断的关系：下一个 eax 的值，是以当前 eax 的值为下标所对应的数组的值。接下来需要寻找这个合适的下标，使得循环 15 次后恰好退出。

比较好的方法是逆推法。下面简述其步骤：

首先，定位数组里值为 15 的元素的下标 6，并找到值为 6 的元素，如图 1.13 所示

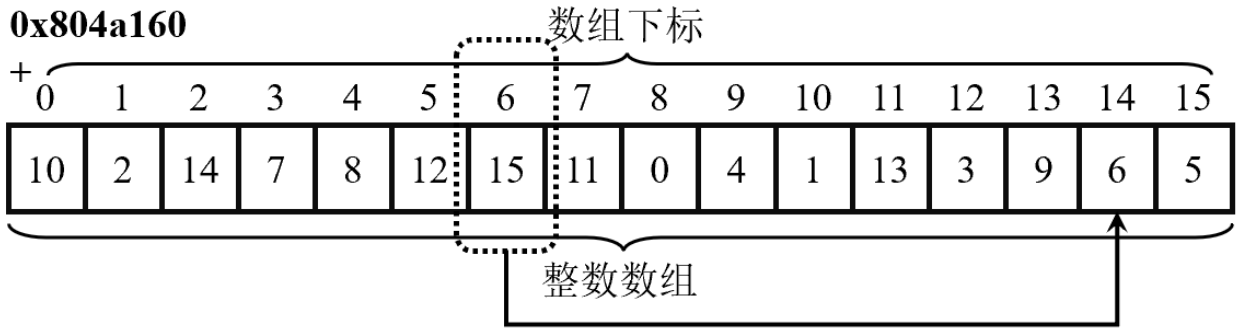

图 1.13 *第一次操作:在数组中寻找元素 15 的下标 6，将其作为元素进行定位。下标 14 对应元素 6。*

按同样方法操作第二次：元素 6 的下标为 14，那么在数组中寻找元素 14，如图 1.14 所示。

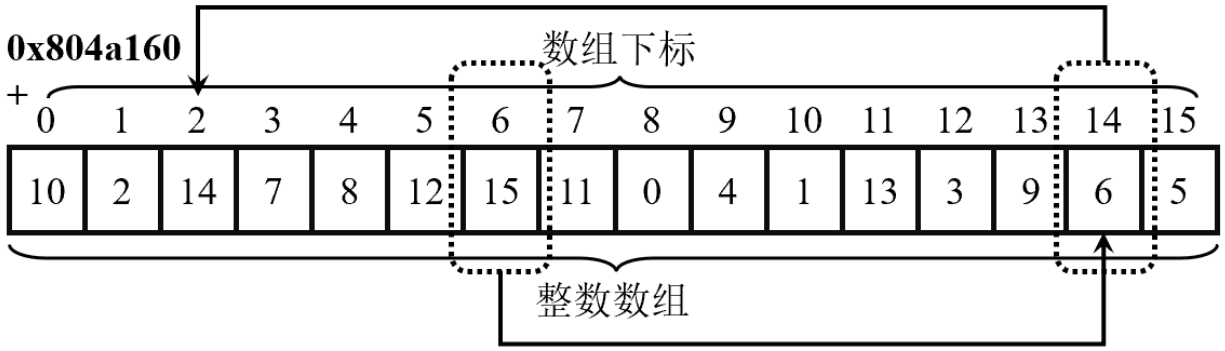

图 1.14 *第二次操作:在数组中寻找元素 14 的下标 6，将其作为元素进行定位。下标 2 对应元素 6。*

这是一个循环迭代的过程，刚刚展示的是逆过程。每一次操作为：

- 获取此时寄存器 eax 中的值；
- 将这个值作为元素，找到其对应的下标 i；
- 将下标 i 作为 eax 的新的值。

对于这个逆过程，为了加深理解，这里采取了类似于 C++语言语法格式的伪代码描述，将这个功能以 GetNextNumber 命名。这个不断迭代更新的算法流程如 Algorithm1.1 所示。

Algorithm1.1 The Process of GetNextNumber

```
1   GetNextNumber (%eax)
2       Let arr[16] = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5}; // Get the array.
3       Get %eax;   // Remove the value in register eax.
4       Let start = %eax;   // Record initial value.
5       While (%eax != 15);   // If the value in register eax is not 15, a loop operation is performed.
6           int currentValue = %eax;
7           int index = arr[currentValue];   // Use the value in eax as the next array subscript.
8           %eax = index;   // Update the value in register eax.
9       endWhile
10      return start;
11  end GetNextNumber
```

*格式: 行号 代码语句 //注释

② 计算总和

有了刚刚的顺序，这一部分就比较简单了。在每一次取到新的下标后，将下标对应的值进行累加即可。这里只需要将 Algorithm1.1 添加一段求和的代码，并且修改其返回值为二元组<start, sum>即可（sum 是总和）。修改后的算法流程 GetNextNumber_New 如 Algorithm1.2 所示。其中，增加的部分已经用加粗斜体注释 ***ADD*** 标出

Algorithm1.2 The Process of GetNextNumber_New

```
1  GetNextNumber_New (%eax)
2      Let arr[16] = {10, 2, 14, 7, 8, 12, 15, 11, 0, 4, 1, 13, 3, 9, 6, 5}; // Get the array.
3      Get %eax;   // Remove the value in register eax.
4      Let start = %eax;   // Record initial value.
5      Let sum = 0;   //Initialisation sum(ADD).
6      While (%eax != 15)   // If the value in register eax is not 15, a loop operation is performed.
7          int currentValue = %eax;
8          int index = arr[currentValue];   // Use the value in eax as the next array subscript.
9          sum += index; //Update the sum(ADD).
10         %eax = index;   // Update the value in register eax.
11     endWhile
12     return {start, sum};   //Return a binary(ADD).
13 end GetNextNumber_New
```

综合①②所述，本函数的主要流程可概况为：输入起始下标（第一个数字）→输入该下标对应的总和（第二个数字）→函数检索起始下标的正确性，如果不正确则炸弹爆炸→函数检验总和的正确性，如果不正确则炸弹爆炸→输入均正确，函数结束。主体流程如 Algorithm1.3 所示。

Algorithm1.3 The Process of Function <phase_5>

```
1  phase_5 (start, sum)
2      Get start;   // Input start subscript (first number).
3      Get sum;   // Enter the sum corresponding to this subscript (second number).
4      Get right_start;   // The function retrieves the correctness of the starting subscript.
5      if (start != right_start)   // If not, the bomb explodes.
6          explode_bomb();
7      end if
8      Let pair<int, int> ans = GetNextNumber_New(start);
9      if (sum !=ans.second)   // The function tests the correctness of the sum.
10         explode_bomb();   // If not, the bomb explodes.
11     end if
12     return;
13 end phase_5
```

1.7 实验小结

对本次实验使用的理论、技术、方法和结果进行总结。

实验 2：_____

2.1 实验概述

介绍本次实验的目的意义、目标、要求及安排等

2.2 实验内容

介绍本次实验的主要内容，如分阶段，则与 1.3~1.5 可以分阶段依次描述。

2.3 实验设计

给出解题思路分析和拟采用的技术和方法等

2.4 实验过程

给出实验过程的详细描述，分步骤说明实验的具体操作过程

2.5 实验结果

给出实验结果和必要的结果分析

2.6 实验小结

对本次实验使用的理论、技术、方法和结果进行总结。

实验 3：_____

3.1 实验概述

介绍本次实验的目的意义、目标、要求及安排等

3.2 实验内容

介绍本次实验的总体主要内容

3.2.1 阶段 1 XXXXXXX

1.任务描述：给出阶段 1 的任务描述

2.实验设计：给出解题思路分析和拟采用的技术和方法等

3.实验过程：详细描述实验的具体过程

4.实验结果：给出阶段 1 的实验结果和必要的结果分析

……

3.2.x 阶段 x XXXXXXX

1.任务描述：给出阶段 x 的任务描述

2.实验设计：给出解题思路分析和拟采用的技术和方法等

3.实验过程：详细描述实验的具体过程

4.实验结果：给出阶段 x 的实验结果和必要的结果分析

3.3 实验小结

对本次实验使用的理论、技术、方法和结果进行总结。描述一下通过实验你有哪些收获。

实验总结

全面总结实验成果，描述通过实验得到收获（400 字以上）