目录

1 实验二：Binary Bomb

1.1 实验概述

在本次实验中，我需要使用上课所学的内容拆除一个二进制炸弹（Binary Bomb）。二进制炸弹的拆除过程一共有六个阶段，分别是`phase_1`~`phase_6`。在拆除炸弹的每个阶段，我需要分别输入一个字符串，并且使得在每个阶段中二进制炸弹不会调用`explode_bomb`函数。在本次实验中，拆除炸弹的难度随着每个阶段递增。每个阶段考察的内容如下所示。

- 阶段 1：字符串比较
- 阶段 2：循环
- 阶段 3：条件/分支
- 阶段 4：递归调用和栈
- 阶段 5：指针
- 阶段 6：链表/指针/结构

除此之外，本实验还有一个隐藏阶段，需要在阶段四输入特定的字符串进行才会出现。本实验要求我熟练的掌握和使用 GDB 调试工具以及 OBJDUMP 工具。其中 GDB 调试工具用于调试程序，OBJDUMP 工具则用于显示二进制炸弹的反汇编代码。

1.2 实验内容

在本次实验中，拆除炸弹的过程主要分为七个阶段，其中第七个阶段是隐藏阶段，将在进行完六个主要阶段后开展。

为了便于后续实验能够顺利地进行，在开展实验之前，我首先需要使用`objdump`工具将可执行文件的反汇编代码保存下来。具体方法是使用如下语句：

```
objdump -D ./bomb > ./bomb.s
```

使用上述语句即可将反汇编之后输出的结果保存在`bomb.s`文件中了。其中`-D`选项表示将可执行文件中所有的节进行反汇编。

接着我还需要分析实验包中的`bomb.c`文件，便于后续拆除炸弹。`bomb.c`文件主要的代码部分如下所示：

```
input = read_line();
phase_1(input);
phase_defused();
printf("Phase 1 defused. How about the next one?\n");

```

```
input = read_line();
phase_2(input);
phase_defused();
printf("That's number 2.  Keep going!\n");

input = read_line();
phase_3(input);
phase_defused();
printf("Halfway there!\n");

input = read_line();
phase_4(input);
phase_defused();
printf("So you got that one.  Try this one.\n");

input = read_line();
phase_5(input);
phase_defused();
printf("Good work!  On to the next...\n");

input = read_line();
phase_6(input);
phase_defused();
```

分析上述代码可知，每一个phase函数的输入参数都一样，都是一个字符串input。而input字符串又是read_line函数的返回值，即从标准输入中送入程序的一个字符串。要将炸弹拆除，我只需要在六个阶段分别输入相应的字符串即可。

1.2.1 阶段 1 字符串匹配

1. 任务描述

   找出phase_1中使用的程序中保存的字符并输入相同的字符串以通过本关卡。

2. 实验设计

   在反汇编文件bomb.s中查找phase_1的汇编代码。找到程序中保存的字符串的地址并用gdb打印出相应的字符串。

3. 实验过程

   (a) 寻找phase_1函数的代码并查看字符串的地址

   在 vscode 中按下Ctrl+F按键，并在弹出的提示框中输入phase_1即可定位到phase_1的代码段。代码段如下所示：

```
08048b33 <phase_1>:
 8048b33:       83 ec 14                sub    $0x14,%esp
 8048b36:       68 24 a0 04 08          push   $0x804a024 // 参数：保存的字符串
 8048b3b:       ff 74 24 1c             push   0x1c(%esp) // 输入的字符串
 8048b3f:       e8 e6 04 00 00          call   804902a <strings_not_equal>
 8048b44:       83 c4 10                add    $0x10,%esp
 8048b47:       85 c0                   test   %eax,%eax
 8048b49:       74 05                   je     8048b50 <phase_1+0x1d>
 8048b4b:       e8 d1 05 00 00          call   8049121 <explode_bomb>
 8048b50:       83 c4 0c                add    $0xc,%esp
 8048b53:       c3                      ret
```

函数的第一行sub $0x14,%esp首先为phase_1分配了0x14的栈帧空间。此时%esp+0x14即是函数的返回地址，而%esp+0x18则是phase_1函数的输入，即main.c文件中看到的input参数。

在函数的第二行中push $0x804a02将保存的字符串地址压入栈中，作为strings_not_equal函数的一个参数。此时%esp的值减少了了0x4，input的地址变为%esp+0x18+0x4 = %esp+0x1c。

接着，在函数的第三行中，push 0x1c(%esp)将input压入栈中，作为strings_not_equal函数的另一个参数。

(b) 使用gdb调试程序，并查看0x804a024地址下字符串的值。

首先使用以下命令进入gdb交互模式：

```
gdb ./bomb
```

接着使用以下命令查看0x804a024地址下字符串的值：

```
(gdb) x /s 0x804a024
0x804a024:      "I am just a renegade hockey mom."
```

由gdb输出的结果可知，”I am just a renegade hockey mom.” 即是我们需要输入的字符串。

4. 实验结果

将上述字符串通输入到ans.txt中并运行程序，通过了第一个关卡。

```
$ echo "I am just a renegade hockey mom." >> ans.txt
$ ./bomb ans.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
```

1.2.2 阶段 2 循环结构

1. 任务描述

分析phase_2代码，并从循环结构中分析出需要输入的数字以破解本关卡。

2. 实验设计

本阶段实验主要分为以下几个步骤：

(a) 找出需要输入的数字个数；

(b) 找到数字存放的位置；

(c) 找出所需要输入的数字具体的值。

3. 实验过程

(a) 找出需要输入的数字个数

查看phase_2反汇编代码可以发现以下用于读取数字的函数read_six_numbers，相关代码如下所示：

Code Listing 1: Read

```
8048b6e:        e8 d3 05 00 00          call   8049146 <read_six_numbers>
8048b73:        83 c4 10                add    $0x10,%esp
8048b76:        83 7c 24 04 01          cmpl   $0x1,0x4(%esp)
```

通过函数的名称很容易知道我们需要输入的数字个数是6个。

(b) 找到数字存放的位置

在read_six_numbers函数返回后，可以发现，在代码1中的地址0x8048b76处将0x4(%esp)与0x1作比较，因此我们可以合理推测出所读入的数字存放在0x4+%esp附近。

接着使用gdb验证上述猜想：

```
$ gdb ./bomb
(gdb) b *0x8048b76 // 上述代码中的cmpl 0x1, 0x4(%esp)语句处设置断点
Breakpoint 1 at 0x8048b76
(gdb) r ans.txt // ans中已经保存了第一关的答案
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
1 1 4 5 1 4 // 第二关的输入测试

Breakpoint 1, 0x08048b76 in phase_2 ()
(gdb) x /6uw 0x4+$esp // 通过观察0x4+$esp中的内容
0xffffc954:     1       1       4       5
0xffffc964:     1       4
```

通过观察0x4+$esp中的内容可以发现，我们输入的数字存放在以0x4+$esp为首地址的连续内存中。

(c) 找出所需要输入的数字具体的值

接着分析代码段，找出第一个数字的值：

```
8048b76:        83 7c 24 04 01          cmpl   $0x1,0x4(%esp) // 第一个数字
8048b7b:        74 05                   je     8048b82 <phase_2+0x2e>
8048b7d:        e8 9f 05 00 00          call   8049121 <explode_bomb>
8048b82:        8d 5c 24 04             lea    0x4(%esp),%ebx
```

上述代码段的逻辑十分简单，即：若第一个数字等于0x1则跳过explode_bomb函数。因此，我们需要输入的第一个数字是1。

分析接下来的循环结构代码，得出剩下数字的值：

```
8048b82:        8d 5c 24 04         lea    0x4(%esp),%ebx // 首地址
8048b86:        8d 74 24 18         lea    0x18(%esp),%esi // 尾地址
8048b8a:        8b 03               mov    (%ebx),%eax // loop start
8048b8c:        01 c0               add    %eax,%eax
8048b8e:        39 43 04            cmp    %eax,0x4(%ebx)
8048b91:        74 05               je     8048b98 <phase_2+0x44>
8048b93:        e8 89 05 00 00      call   8049121 <explode_bomb>
8048b98:        83 c3 04            add    $0x4,%ebx
8048b9b:        39 f3               cmp    %esi,%ebx
8048b9d:        75 eb               jne    8048b8a <phase_2+0x36> // loop end
```

由0x18 = 24 = 6*sizeof(int)可知，0x18+%esp是第六个数字的地址。分析上述代码：进入循环前程序先将数组的首地址存放在%ebx中，将数组的尾地址存放在%esi中。进入循环后，程序将当前数字存放在%eax中，并将2*%eax与下一个数字（0x4(%ebx)）进行比较，若两者相等，则跳过explode_bomb。因此剩下的数字的值分别是前一个数字的两倍。

综合上述分析可知，由于第一个数字是1，因此接下来的每一个数字分别是2、4、8、16、32。

4. 实验结果

将第二关的答案输入ans.txt中并运行程序：

```
$ echo "1 2 4 8 16 32" >> ans.txt
$./bomb ans.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That is number 2.  Keep going!
```

顺利通过！

1.2.3 阶段 3 条件分支

1. 任务描述

   找出代码段中的条件分支，并通过输入正确的数字破解关卡。

2. 实验设计

   本阶段实验主要分为以下几个步骤：

   (a) 判断第一个参数的范围，并在该范围内随便选取一个数；

   (b) 使用 gdb 找出给定第一个参数后，第二个参数的值。

3. 实验过程

   (a) 判断第一个参数的范围

   在phase_3的开头部分有以下代码段：

```
8048bd9:        e8 32 fc ff ff          call   8048810 <__isoc99_sscanf@plt>
8048bde:        83 c4 10                add    $0x10,%esp
8048be1:        83 f8 01                cmp    $0x1,%eax // 参数需要多于一个
8048be4:        7f 05                   jg     8048beb <phase_3+0x34>
8048be6:        e8 36 05 00 00          call   8049121 <explode_bomb>
8048beb:        83 7c 24 04 07          cmpl   $0x7,0x4(%esp) // 第一个参数<=7
8048bf0:        77 66                   ja     8048c58 <phase_3+0xa1>
...
8048c58:        e8 c4 04 00 00          call   8049121 <explode_bomb>
```

   根据前面关卡的分析，很容易知道0x4+%esp是第一个参数的地址。在代码段中地址0x8048beb处将0x4(%esp)的值和0x7作比较，如果第一个参数比 7 大，就会跳转到0x8048c58处，即call explode_bomb语句处。因此我们可以确定第一个参数需要小于或等于7。接着，根据比较指令使用的是ja指令，可以知道第一个参数是无符号整形数，因此第一个参数还需要大于或等于0。下面从 $\{0, 1 \ldots 7\}$ 内尝试选取第一个参数，不妨选 1。

   (b) 在给定第一个参数后，确定第二个参数的值

   接着分析代码段：

```
8048bf2:  8b 44 24 04           mov    0x4(%esp),%eax //将第一个参数赋给%eax
8048bf6:  ff 24 85 80 a0 04 08  jmp    *0x804a080(,%eax,4) // 根据%eax转跳
8048bfd:  b8 77 01 00 00        mov    $0x177,%eax
8048c02:  eb 05                 jmp    8048c09 <phase_3+0x52>
8048c04:  b8 00 00 00 00        mov    $0x0,%eax
8048c09:  2d ac 01 00 00        sub    $0x1ac,%eax
8048c0e:  eb 05                 jmp    8048c15 <phase_3+0x5e>
```

```
8048c10:  b8 00 00 00 00        mov    $0x0,%eax
8048c15:  05 fa 01 00 00        add    $0x1fa,%eax
8048c1a:  eb 05                 jmp    8048c21 <phase_3+0x6a>
8048c1c:  b8 00 00 00 00        mov    $0x0,%eax
8048c21:  2d c9 03 00 00        sub    $0x3c9,%eax
8048c26:  eb 05                 jmp    8048c2d <phase_3+0x76>
8048c28:  b8 00 00 00 00        mov    $0x0,%eax
8048c2d:  05 c9 03 00 00        add    $0x3c9,%eax
8048c32:  eb 05                 jmp    8048c39 <phase_3+0x82>
8048c34:  b8 00 00 00 00        mov    $0x0,%eax
8048c39:  2d c9 03 00 00        sub    $0x3c9,%eax
8048c3e:  eb 05                 jmp    8048c45 <phase_3+0x8e>
8048c40:  b8 00 00 00 00        mov    $0x0,%eax
8048c45:  05 c9 03 00 00        add    $0x3c9,%eax
8048c4a:  eb 05                 jmp    8048c51 <phase_3+0x9a>
8048c4c:  b8 00 00 00 00        mov    $0x0,%eax
8048c51:  2d c9 03 00 00        sub    $0x3c9,%eax
8048c56:  eb 0a                 jmp    8048c62 <phase_3+0xab>
8048c58:  e8 c4 04 00 00        call   8049121 <explode_bomb>
8048c5d:  b8 00 00 00 00        mov    $0x0,%eax
8048c62:  83 7c 24 04 05        cmpl   $0x5,0x4(%esp)
8048c67:  7f 06                 jg     8048c6f <phase_3+0xb8>
8048c69:  3b 44 24 08           cmp    0x8(%esp),%eax // 将第二个参数与%eax比较
8048c6d:  74 05                 je     8048c74 <phase_3+0xbd>
8048c6f:  e8 ad 04 00 00        call   8049121 <explode_bomb>
8048c74:  8b 44 24 0c           mov    0xc(%esp),%eax
```

上述代码首先根据%eax的值跳转0x804a080中存储的地址，接着进行一系列的跳转改变%eax的值。最后将第二个参数（0x8(%esp)）与%eax作比较，若两个数相等，则跳过explode_bomb。分析上述转跳表的逻辑看似是本关卡的必经之路,但我们很容易发现:虽然转跳表改变了%eax的值，我们只需要在最后保证第二个参数的值与转换后的%eax一样就行了。因此我们假定第一个参数为1，并在0x8048c69处打上断点，在断点处查看%eax的值即可。

```
gdb ./bomb
(gdb) b *0x8048c69
Breakpoint 1 at 0x8048c69
(gdb) r ans.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That is number 2.  Keep going!
1 0 // 测试输入，假设第一个参数为1

Breakpoint 1, 0x08048c69 in phase_3 ()
(gdb) p $eax
$1 = -891 // 第二个参数需为-891
```

使用gdb调试后可以和轻松的知道第二个参数为-891，而不需要分析分支转调表。

4. 实验结果

将第三关的答案输入ans.txt中并运行程序即可顺利通过:

```
$ echo "1 -891" >> ans.txt
$./bomb ans.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That is number 2.  Keep going!
Halfway there!
```

1.2.4 阶段 4 递归调用

1. 任务描述

   查看反汇编代码中递归函数的逻辑以及期望的返回值，用 C 语言复现递归函数并遍历所有输入找到期望的返回值。

2. 实验设计

   (a) 查看phase_4的反汇编代码，并确定输入参数的类型、范围以及数量；

   (b) 查看期待的递归函数func4的返回值；

   (c) 分析func4函数并使用 C 语言复现；

   (d) 遍历函数的输入，找出期望的返回值对应的输入。

3. 实验过程

   (a) 查看phase_4的反汇编代码，并确定输入参数的类型、范围以及数量

   查看汇编代码中读取输入的部分，如下所示：

```
8048cfb: 50              push   %eax
8048cfc: 68 ef a1 04 08  push   $0x804a1ef // 格式化字符串
8048d01: ff 74 24 2c     push   0x2c(%esp)
8048d05: e8 06 fb ff ff  call   8048810 <__isoc99_sscanf@plt>
8048d0a: 83 c4 10        add    $0x10,%esp
8048d0d: 83 f8 02        cmp    $0x2,%eax // 需要输入两个参数
```

   根据上述代码分析，可以使用gdb查看位于0x804a1ef格式化字符串：

```
$ gdb ./bomb
(gdb) x /s 0x804a1ef
0x804a1ef:      "%d %d"
```

   可以确定，本关卡要求输入的参数为两个整形数字。

   继续分析代码：

```
8048d12: 83 7c 24 04 0e cmpl $0xe,0x4(%esp) // 第一个参数<= 0xe
8048d17: 76 05          jbe  8048d1e <phase_4+0x3b> // jbe:第一个参数为无符号数
8048d19: e8 03 04 00 00 call 8049121 <explode_bomb>
8048d1e: 83 ec 04       sub  $0x4,%esp
```

   可以确定第一个参数（0x4(%esp)）的范围是 $\{x \in \mathcal{Z} | 0 \leq x \leq 14\}$（0xe=14）。

   根据以下代码可以直接确定第二个参数的值：

```
8048d36: 83 7c 24 08 1b cmpl   $0x1b,0x8(%esp) // 第二个参数为27
8048d3b: 74 05          je     8048d42 <phase_4+0x5f>
8048d3d: e8 df 03 00 00 call   8049121 <explode_bomb>
8048d42: 8b 44 24 0c    mov    0xc(%esp),%eax
```

   当0x8(%esp)（即第二个参数）的值为0x1b=27时，跳过explode_bomb。因此，第二个参数为27。

   (b) 查看期待的递归函数func4的返回值

   找到调用函数func4后使用返回值%eax的代码段：

```
8048d29: e8 5c ff ff ff  call   8048c8a <func4>
8048d2e: 83 c4 10        add    $0x10,%esp
8048d31: 83 f8 1b        cmp    $0x1b,%eax // func4 returns 27
8048d34: 75 07           jne    8048d3d <phase_4+0x5a>
...
8048d3d: e8 df 03 00 00  call   8049121 <explode_bomb>
```

分析上述代码段可知，函数func4需要返回0x1b=27才能跳过爆炸。

(c) 分析函数func4接收的参数

找到调用函数func4前的部分代码：

```
8048d1e: 83 ec 04        sub    $0x4,%esp
8048d21: 6a 0e           push   $0xe
8048d23: 6a 00           push   $0x0
8048d25: ff 74 24 10     push   0x10(%esp) // 输入字符串中的第一个参数
8048d29: e8 5c ff ff ff  call   8048c8a <func4>
```

分析上述代码可知，func4一共接收三个参数，分别是0x10(%esp)、0x0以及0xe。我们输入的第一个参数的位置本来是0x4+%esp，但由于在调用func4函数之前%esp的值减少了0x4，并且还将两个数（0xe与0x0）进行了压栈，因此0x10(%esp)即是我们输入字符串中的第一个参数(0x10=0x4+0x4+0x4+0x4)。

考虑到C调用约定中函数参数使用反向压栈的方式，调用func4函数的C语句为：

```
func4(param1, 0, 14);
```

其中，param1是我们从输入字符串的第一个参数。

(d) 分析func4函数并使用 C 语言复现

分析参数在func4中存放的位置：

```
8048c8f: 8b 54 24 10     mov    0x10(%esp),%edx // p1=param1
8048c93: 8b 74 24 14     mov    0x14(%esp),%esi // p2=0
8048c97: 8b 4c 24 18     mov    0x18(%esp),%ecx // p3=14
```

从上述代码中不难看出，输入的三个参数分别存放在%edx、%esi以及%ecx中。

接着分析参数在func4中的计算过程：

```
8048c9b: 89 c8     mov    %ecx,%eax // %eax=p3
8048c9d: 29 f0     sub    %esi,%eax // %eax=p3-p2
8048c9f: 89 c3     mov    %eax,%ebx // %ebx=p3-p2
8048ca1: c1 eb 1f  shr    $0x1f,%ebx // %ebx>>=31,即%ebx=(p3-p2<0?1:0)
8048ca4: 01 d8     add    %ebx,%eax // %eax=p3-p2+(p3-p2<0?1:0)
8048ca6: d1 f8     sar    %eax // %eax=(p3-p2+(p3-p2<0?1:0))/2
8048ca8: 8d 1c 30  lea    (%eax,%esi,1),%ebx // %ebx=(p3-p2+(p3-p2<0?1:0))/2+p2
```

经过参数一系列的转化，最终得到了%ebx的值。这个值十分重要，因为这是我历经千辛万苦得出的结论，接下来的代码段中会根据%ebx的值进行转调并递归。

接着分析递归调用的转调代码：

```
8048cab: 39 d3           cmp    %edx,%ebx // p1>%ebx?
8048cad: 7e 15           jle    8048cc4 <func4+0x3a>
8048caf: 83 ec 04        sub    $0x4,%esp
8048cb2: 8d 43 ff        lea    -0x1(%ebx),%eax
8048cb5: 50              push   %eax
8048cb6: 56              push   %esi
```

```
8048cb7: 52              push   %edx
8048cb8: e8 cd ff ff ff  call   8048c8a <func4>
8048cbd: 83 c4 10        add    $0x10,%esp
8048cc0: 01 d8           add    %ebx,%eax // %eax=func4(p1, p2, ebx-1)+ebx;
8048cc2: eb 19           jmp    8048cdd <func4+0x53>
8048cc4: 89 d8           mov    %ebx,%eax
8048cc6: 39 d3           cmp    %edx,%ebx // p1<%ebx?
8048cc8: 7d 13           jge    8048cdd <func4+0x53>
8048cca: 83 ec 04        sub    $0x4,%esp
8048ccd: 51              push   %ecx
8048cce: 8d 43 01        lea    0x1(%ebx),%eax
8048cd1: 50              push   %eax
8048cd2: 52              push   %edx
8048cd3: e8 b2 ff ff ff  call   8048c8a <func4>
8048cd8: 83 c4 10        add    $0x10,%esp
8048cdb: 01 d8           add    %ebx,%eax// %eax=func4(p1, ebx+1, p3)+ebx;
```

分析上述代码，并结合前面对ebx的分析，不难得出func4的C语言代码：

```c
int func4(int p1, int p2, int p3)
{
    int ebx = (p3-p2+(p3-p2<0?1:0))/2+p2;
    if (p1 == ebx)
        return p1;
    if (p1 < ebx)
        return func4(p1, p2, ebx-1) + ebx;
    if (p1 > ebx)
        return func4(p1, ebx+1, p3) + ebx;
}
```

(e) 遍历函数的输入，找出期望的返回值对应的输入

根据3a中的分析可知，第一个参数的范围是 $\{0, 1 \dots 14\}$，第二个参数为0，第三个参数为14。因此我们只需要遍历第一个参数即可得出返回值为27时对应的输入。

在main函数中：

```c
int main(){
    for (int p1 = 0; p1 <= 14; ++p1){
        int ret = func4(p1, 0, 14);
        if (ret == 27){
            printf("p1 = %d, ret = %d\n", p1, ret);
            break;
        }
    }
    return 0;
}
```

将main函数与func4函数写入analyze_phase_4.c，编译并运行：

```
$ gcc analyze_phase_4.c -o main
$ ./main
p1 = 9, ret = 27
```

可以知道，输入的第一个参数为9。

4. 实验结果

将参数9与27输入到ans.txt中并运行程序：

```
$ echo "9 27" >> ans.txt
$ ./bomb ans.txt
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That is number 2.  Keep going!
Halfway there!
So you got that one.  Try this one.
```

历经千辛万苦终于顺利通过！

1.2.5 阶段 5 指针

1. 任务描述
2. 实验设计
3. 实验过程
4. 实验结果

1.2.6 阶段 6 链表/指针/结构

1. 任务描述
2. 实验设计
3. 实验过程
4. 实验结果

1.2.7 阶段 7 二叉查找树

1. 任务描述
2. 实验设计
3. 实验过程
4. 实验结果