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's number 2.  Keep going!
```

顺利通过!

1.2.3 阶段 3 条件分支

1. 任务描述
2. 实验设计
3. 实验过程
4. 实验结果

1.2.4 阶段 4 递归调用

1. 任务描述
2. 实验设计
3. 实验过程
4. 实验结果

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