

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**专业班级： CS2003班**

**学 号： U202015375**

**姓 名： 汪宇飞**

**指导教师： 胡侃**

**报告日期： 2022年 6月 21日**

**计算机科学与技术学院**

**目录**

[实验2： 3](#_Toc73983883)

[实验3： 19](#_Toc73983884)

[实验总结 36](#_Toc73983885)

# 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。

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。

实验的目标是你要拆除尽可能多的炸弹阶段。

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：含switch语句
* 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 实验内容

拆弹技术：为了完成二进制炸弹拆除任务，你需要

① 使用gdb调试器和objdump来反汇编炸弹的可执行文件；

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”出拆除炸弹所需的目标字符串。

⑤ 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

此实验中每个人会分发一个不同的binary bomb二进制可执行文件及其相关文件，获得其反汇编代码后通过观察运行和测试，获得六个阶段的密码。

### 2.2.1 阶段1 phasse\_1

1. 任务描述：

根据实验指引，在反汇编文件中找到对应的代码，对phase\_1的函数进行分析从而得出phase\_1的内容。

1. 实验设计：

根据所给的实验指引步骤，在反汇编文件中找到对应的代码部分，获得相关信息，并使用gdb进行调试，访问内存得到字符串phase\_1的内容。

1. 实验过程：

（1）调用objdump –d bomb > asm.txt对bomb进行反汇编并将汇编代码输出到asm.txt中。

（2）查看汇编源代码asm.txt文件。首先，查找“main”，找到main函数的位置。然后，在main函数中找到如下语句（这里为phase1函数在main()函数中被调用的位置）：

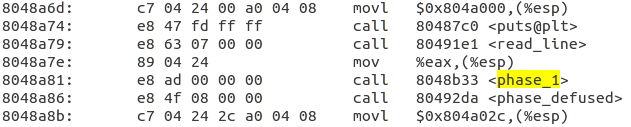


图2.1 main函数中的phase\_1

（3）根据所查询到的地址找到phase\_1函数，如图2.2所示。

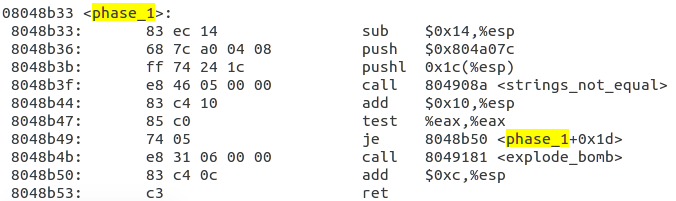


图2.2 phase\_1函数

由图中可知，strings\_not\_equal函数调用前push了地址0x804a07c，从而可以推断出该地址保存着phase\_1的内容。通过gdb调试获取内存如图2.3 所示。根据字符串末尾为00，并对照ASCII码表可以得出phase\_1的内容。

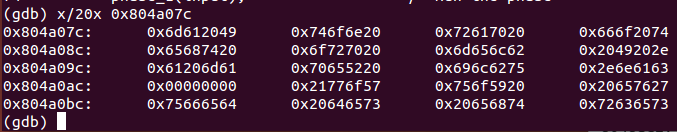


图2.3 gdb获取内存

1. 实验结果：

由以上分析得出phase\_1：I am not part of the problem. I am a Republican。如图2.4所示。



图2.4 phase\_1输入测试

### 2.2.2 阶段2 phase\_2

1.任务描述：

在反汇编文件中找到对应的代码，对phase\_2的函数进行分析从而得出phase\_2的内容。

2.实验设计：

在反汇编文件中找到对应的代码部分，获得相关信息，并使用gdb进行调试，分析phase\_2函数所使用的循环结构，得出该所需要输入的值。

3.实验过程：

在asm.txt中找到phase\_2函数的代码，开始进行分析。

如图2.5所示，在0x8048b54到0x8048b6e部分，开辟了大小为0x2c的空间并通过<read\_six\_numbers>函数读入了六个数字，那么可知这一关的输入也就是六个数字。

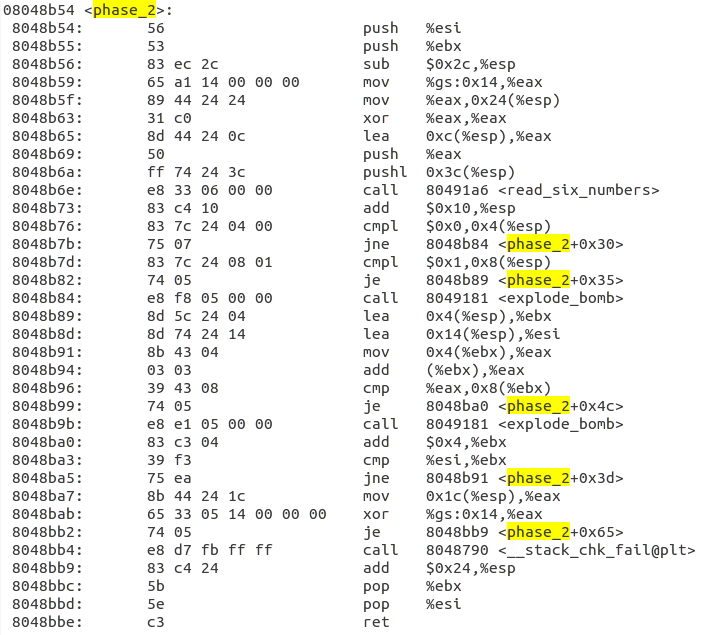


图2.5 0x8048b54到0x8048b6e部分

在明确了输入类型之后继续推断，如图2.6所示，在输入数字之后首先是cmpl $0x1,0x4(%esp)，并且如果不相等则会爆炸，也即输入的第一个数字应该是0。接下来是cmpl $0x1,0x8(%esp)，如果不相等也会爆炸，所以第二个数字应该是1。

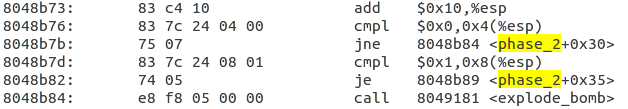


图2.6 0x8048b73到0x8048b84部分

随后如图2.7所示，将ebx赋值为0x4(%esp)也即为0，将eax赋值为0x4(%ebx)也即下一个数字，二者相加后与0x8(%ebx)也即下下个数字相比较，若不相等则爆炸，也即第三个数字为前两个数字之和，再之后为循环操作直到比较完第六个数字，从而得出数列为斐波那契数列。

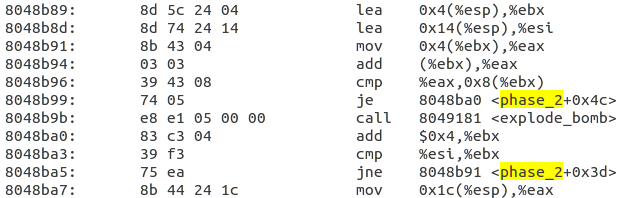


图2.7 0x8048b89到0x8048ba7部分

4.实验结果：

由以上分析可知，输入的六个数为0 1 1 2 3 5。如图2.8所示。



图2.8 phase\_2输入测试

### 2.2.3 阶段3 phase\_3

1.任务描述：

在反汇编文件中找到对应的代码，对phase\_3的函数进行分析从而得出phase\_3的内容。

2.实验设计：

在反汇编文件中找到对应的代码部分，获得相关信息，并使用gdb进行调试，分析phase\_3中所出现的switch结构，得出所需要的输入值。

3.实验过程：

在asm.txt中找到phase\_3函数的代码。

如图2.9所示，在0x8048bbf到0x8048be6部分，函数开辟了大小为0x28的空间，随后push了0x804a0d6之后调用了输入函数输入了数据，用gdb调试查看0x804a0d6处的数据如图2.10所示，可得“%d %c %d”，即分别输入了应该数字、一个字符和一个数字。

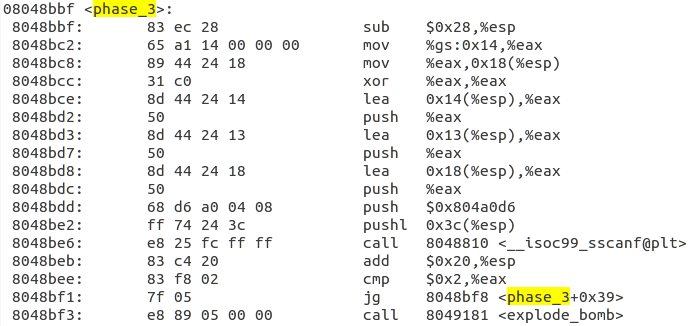


图2.9 0x8048bbf到0x8048bf3部分



图2.10 0x804a0d6处内容

接着如图2.11所示，由cmpl $0x7,0x4(%esp)如果大于则爆炸可知，第一个数据应该是0~7中的一个数字。



图2.11 0x8048bf8到0x8048c07部分

从而可以推测出接下来的jmp \*0x804a0e8(,%eax,4)即为switch跳转语句。通过gdb调试查看内存，可以得出0x804a0e8处的数据如图2.12所示。

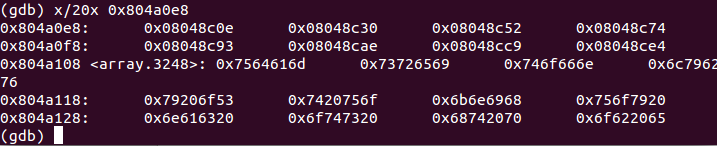


图2.12 0x804a0e8处内容

即存放了第一个数字取0~7时所跳转的地址，每个地址所进行的操作相同，以第一个数字为0时为例分析。

如图2.13所示，在第一个数字为0的情况下，首先将eax赋值为0x6c，然后将0x26c与0x8(%esp)即第三个数字相比较，若不相等则爆炸，所以第三个数字为0x26c即620。

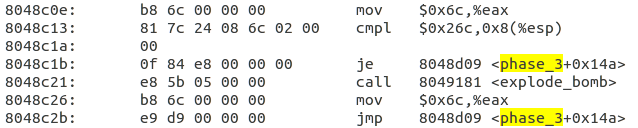


图2.13 0x8048c0e到0x8048c2b部分

最后如图2.14所示，将0x3(%esp)即输入的一个字符与al相比较，此时al中所存放的即0x6c，为字符“l”。由此得出第一个数字为零时所输入应该为“0 l 620”。

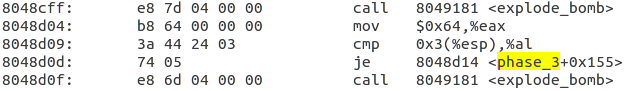


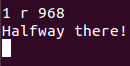
图2.14 0x8048cff到0x8048d0f部分

4.实验结果：

由上述分析可得第一个数字为0~7时输入分别为“0 l 620”“1 r 968”“2 t 674”“3 y717”“4 e 686”“5 u 308”“6 e 338”“7 o 171”。如以下图所示。



2.15 第一个输入数字为0



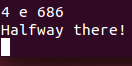
2.16 第一个输入数字为1



2.17 第一个输入数字为2



2.18 第一个输入数字为3



2.19 第一个输入数字为4



2.20 第一个输入数字为5



2.21 第一个输入数字为6



2.22 第一个输入数字为7

### 2.2.4 阶段4 phase\_4

1.任务描述：

在反汇编文件中找到对应的代码，对phase\_4的函数进行分析从而得出phase\_4的内容。

2.实验设计：

在反汇编文件中找到对应的代码部分，获得相关信息，并使用gdb进行调试，分析phase\_4中所出现的递归函数情况，得出所需要输入的值。

3.实验过程：

在asm.txt中找到phase\_4函数的代码并从函数中可看出phase\_4调用了func4函数，在asm.txt中找到func4的代码。

对于func4分析，如图2.23所示，由mov 0x10(%esp),%ebx和mov 0x14(%esp),%edi可知，函数有两个int型参数，分别设为x和y。



图2.23 0x8048d2a到0x8048d31部分

接下来如图2.24所示，由test语句可知，如果x为0，则将eax也设置为0；由cmp语句可知，如果x为1，则eax设置为y。



图2.24 0x8048d35到0x8048d3e部分

如图2.25所示，在x大于1时，函数首先将edi即x进行push操作，然后将eax赋值为y-1后也push并递归调用func4函数，即func4(x,y-1)。同理推断随后是func4(x,y-2)，最后将这两个递归调用的函数与y相加，作为返回值存入eax中。

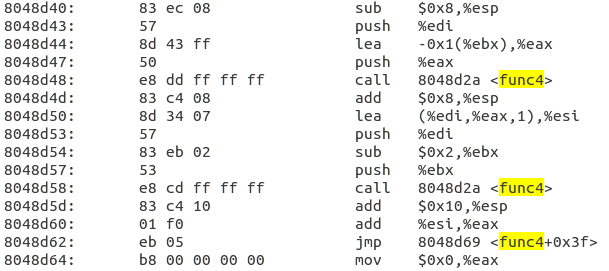


图2.25 0x8048d40到0x8048d64部分

由上述分析可得func4函数的C语言形式，如图2.26所示。

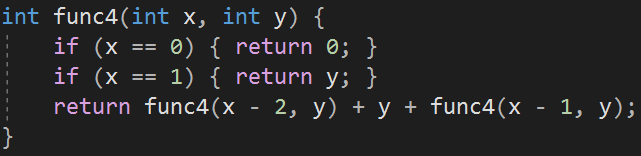


图2.26 func4函数的C语言形式

得出了func4的作用后，再次回到phase\_4函数进行分析，如图2.27所示，在0x8048d6d到0x8048d8f部分，函数开辟了大小为0x1c的空间，随后push %eax和push $0x804a227之后调用了输入函数。

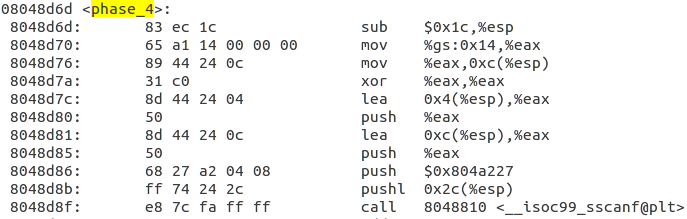


图2.27 0x8048d6d到0x8048d8f部分

通过gdb调试观察该地址的内容如图2.28所示。



图2.28 0x804a227处内容

即要求输入的是两个int整型数，接下来如图2.29所示，通过pushl 0xc(%esp)可知所输入的第二个整型数将作为为func4的参数y，通过pushl 0x9可知func4的参数x为常量9，又由0x8048d9c至0x8048da8部分可知，y的范围是2~4。

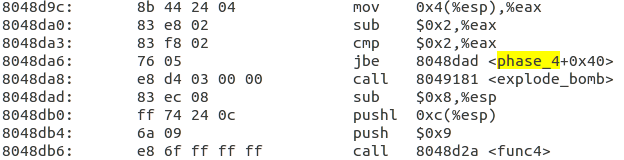


图2.29 0x8048d9c到0x8048db6部分

如图2.30所示，从调用后的cmp 0x8(esp),%eax语句可以得知所输入的第一个整型数应该等于func4函数的返回值。



图2.30 0x8048dbe到0x8048dc4部分

以第二个数字即func4的y参数为2时为例分析，在VS2019中编写相应的程序并运行，程序和运行结果如图2.31所示。

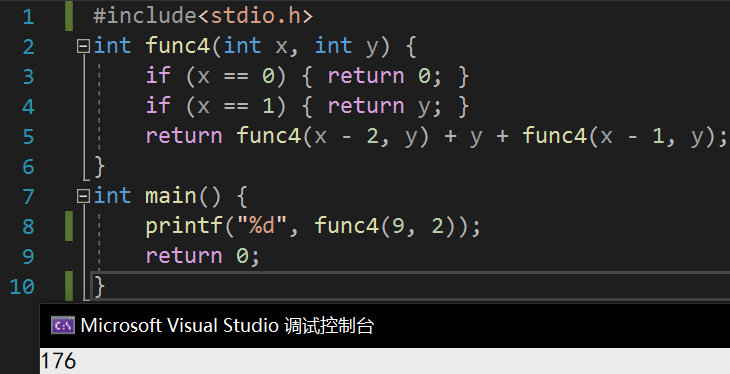


图2.31 func4对应C语言程序和运行结果

4.实验结果：

由上述分析可得第二个数字为2~4时输入分别为“176 2”“264 3”“352 4”，如以下图所示。



图2.32 第一个数字为2时的结果



图2.33 第一个数字为3时的结果



图2.34 第一个数字为4时的结果

### 2.2.5 阶段5 phase\_5

1.任务描述：

在反汇编文件中找到对应的代码，对phase\_5的函数进行分析从而得出phase\_5的内容。

2.实验设计：

在反汇编文件中找到对应的代码部分，获得相关信息，并使用gdb进行调试，分析phase\_5中所使用的指针用法，得出所需要输入的值。

3.实验过程：

在asm.txt中找到phase\_5函数的代码。

如图2.35所示，在0x8048ddf到0x8048e01部分，开辟了大小为0x24的空间并通过<string\_length>函数获得了字符串的大小，并与0x6作比较，若果不相等则爆炸，从而可以推断出输入为大小为六的字符串。

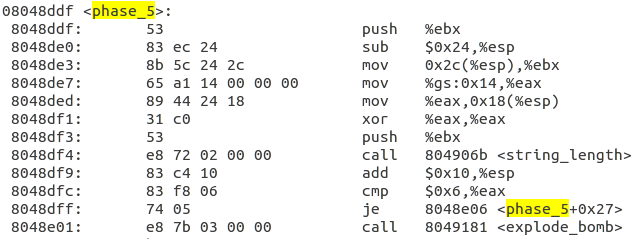


图2.35 0x8048ddf到0x8048e01部分

随后如图2.36所示，由movzbl (%ebx,%eax,1),%edx和and $0xf,%edx可推出保留每个字符的低四位，随后通过movzbl 0x804a1d8 (%edx),%edx语句进行查找，由所输入的六个字符的第四位为指针进行查找，得出新的字符串。接着将该字符串与0x804a0df处的字符串相比较，若不相同则爆炸。

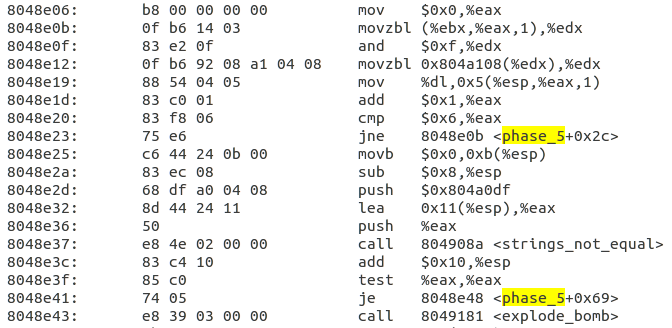


图2.36 0x8048e06到0x8048e43部分

通过gdb调试查看0x804a1d8和0x804a0df处的内容，如图2.37所示。

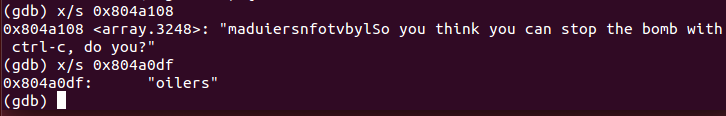


图2.37 0x804a1d8和0x804a0df处的内容

可知需要输入的字符串应该满足其ASCII码的低四位与oilers相对应，这样的字符串有多种组合，其中一种如表2.1所示。

表2.1 oilers对应字符串

|  |  |  |
| --- | --- | --- |
| **原字符** | **ASCII码低四位** | **对应字符** |
| o | 0xa | j |
| i | 0x4 | d |
| l | 0xf | o |
| e | 0x5 | e |
| r | 0x6 | f |
| s | 0x7 | g |

4.实验结果：

由以上的分析，取多种组合中的一种为“jdoefg”，如图2.38所示。



图2.38 phase\_5输入测试

### 2.2.6 阶段6 phase\_6

1.任务描述：

在反汇编文件中找到对应的代码，对phase\_6的函数进行分析从而得出phase\_6的内容。

2.实验设计：

在反汇编文件中找到对应的代码部分，获得相关信息，并使用gdb进行调试，分析phase\_6中所使用的链表结构，得出所需要的输入。

3.实验过程：

在asm.txt中找到phase\_6函数的代码。

如图2.39所示，在0x8048e5f到0x8048e79部分，开辟了大小为0x2c的空间并通过<read\_six\_numbers>读入了六个数字，也即所要求输入的数据为六个数字。由于phase\_6函数体较大且运用了多个分支跳转语句，因此可以结合前五关的经验进行逐段分析代码功能简化分析过程。

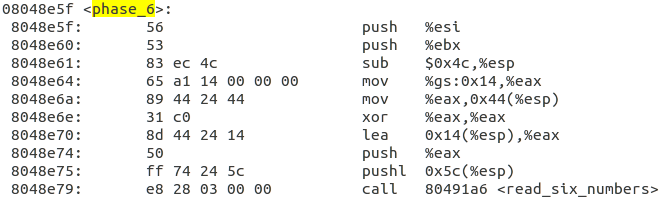


图2.39 0x8048e5f到0x8048e79部分

如图2.40所示，从0x8048e81到0x8048eb8部分，函数在读入六个数字之后首先逐个判断数字是否小于等于6，如果大于6则会爆炸。随后再将当前的数字与前面的数字逐个判断是否相等，如果相等则也会爆炸。从而可以推断出所需要输入的数据应该为1~6的排列。

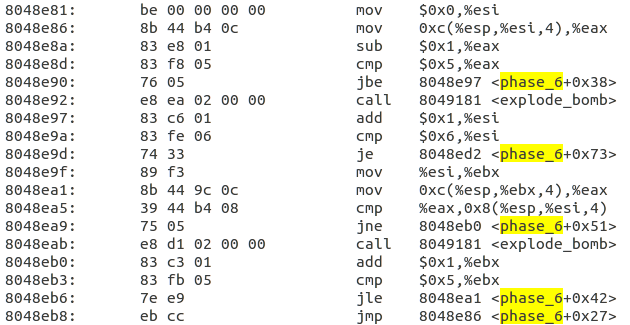


图2.40 0x8048e81到0x8048eb8部分

如图2.41所示，从0x8048eba到0x8048eec部分，函数读入以0x804c13c为首地址的一片内存的对应数据，并且将1~6与之对应起来，通过gdb调试查看0x804c13c处的内容如图2.42所示。

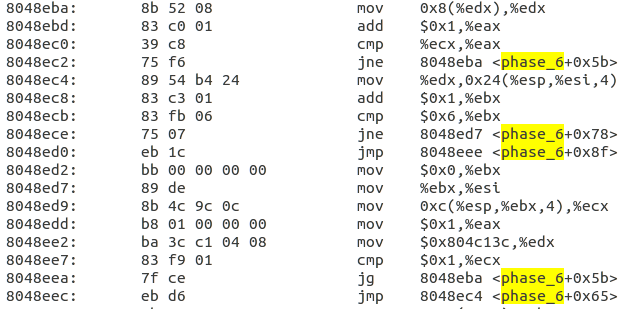


图2.41 0x8048eba到0x8048eec部分

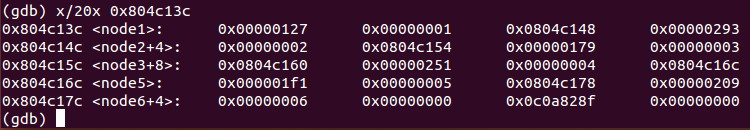


图2.42 0x804c13c处的内容

由图中可见，该区域储存了对应的数据以及数据所在的地址，根据函数进行分析后可以得出该部分存放的是六个结构体，结构体如表2.2所示。

表2.2 结构体组成

|  |  |
| --- | --- |
| 序号 | 含义 |
| 1 | 该结构体的数据 |
| 2 | 该结构体的序号 |
| 3 | 下一个结构体的地址 |

函数分析对应关系得到1~6对应的数据地址及其对应数据如表2.3所示。

表2.3 1~6对应的地址及其数据

|  |  |  |
| --- | --- | --- |
| 序号 | 对应地址 | 数据 |
| 1 | 0x0804c13c | 0x127 |
| 2 | 0x0804c148 | 0x293 |
| 3 | 0x0804c154 | 0x179 |
| 4 | 0x0804c160 | 0x251 |
| 5 | 0x0804c16c | 0x1f1 |
| 6 | 0x0804c178 | 0x209 |

得到了每个数字所对应的数据之后，如图2.43所示，在0x8048eee到0x8048f09部分，函数根据所输入的序列构建索引，从而生成链表结构。

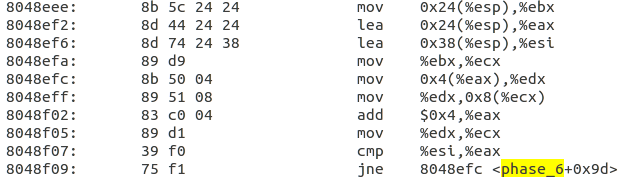


图2.43 0x8048eee到0x8048f09部分

随后如图2.44所示，在0x8048f0b到0x8048f2b部分，函数根据生成的链表进行逐个比较，如果后一个数据大于前一个数据则会爆炸，从而可以推断处输入的序列应该满足每一个数字所对应的数据应该由小到大排列。

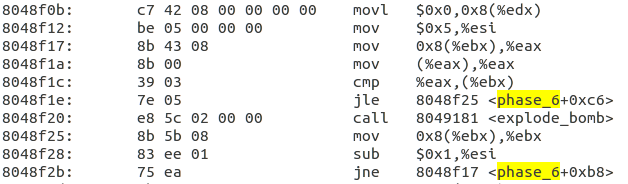


图2.43 0x8048e0b到0x8048f2b部分

4.实验结果：

根据上述分析，对照表2.3的内容可以得出输入的序列为：“1 3 5 6 4 2”，如图2.44所示。



图2.44 phase\_6输入测试

## 2.3 实验小结

这次实验是在Linux环境下通过分析反汇编文件并结合以gdb调试工具进行的，总体上来说还是比较富有挑战性的，分析求解出全部六个phase前前后后一共花费了我六七个小时，也是我做过的最有难度的实验之一。

实验方法无外乎是对着反汇编代码一行一行地分析其作用，加上通过gdb调试并观察内存从而得出答案，但是在实际分析的时候还是遇到了很多问题与困难的。譬如在分析phase\_4的时候函数涉及到了递归和栈的使用，多次调用函数导致esp改变较为频繁，从而为分析带来了很多阻碍，只能在分析时格外留心esp以及其他寄存器的变化从而得到正确结果。

除此之外，有的函数体十分巨大，比如phase\_6的函数体，十分地冗长，如果逐行去理解很容易被绕晕，所以在经过前几个关卡的练习后可以先对phase\_6的函数体进行大致区分并各自判断功能，从而简化流程加快理解速度。

总体而言，这次试验完成得比较成功，对于有多个解的关卡也均成功给出了所有可能的解，取得了比较让人满意的结果。

# 实验3： 缓冲区溢出攻击

## 3.1 实验概述

实验目的：加深对IA-32函数调用规则和栈结构的具体理解。

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。

实验安排：

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。实验环境为C语言和linux，要求熟练运用gdb、objdump、gcc等工具。

## 3.2 实验内容

离线下载实验程序包：lab3.tar

在本地目录将程序包解压：tar –xf lab3.tar

数据包中至少包含下面四个文件：

\* bufbomb：可执行程序，攻击所用的目标程序bufbomb。

\* bufbomb.c：C语言源程序，目标程序bufbomb的主程序。

\* makecookie：可执行程序，该程序基于你的学号产生一个唯一的由8个16进制数字组成的4字节序列（例如0x5f405c9a），称为“cookie”。

\* hex2raw：可执行程序，字符串格式转换程序。

另一个需要的文件是，用objdump工具反汇编bufbomb可执行目标程序，得到它的反汇编源程序。

需要精心设计一些字符串输入给bufbomb，有意造成缓冲区溢出，而使bufbomb程序完成一些有趣的事情。

本实验需要构造5个攻击字符串，对目标程序bufbomb分别实施5次缓冲区溢出攻击。5次攻击难度递增，分别命名为：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：

根据任务要求，跟随实验示例，构造一个攻击字符串作为bufbomb的输入，在getbuf()中造成缓冲区溢出，使得getbuf()返回时不是返回到test函数，而是转到smoke函数处执行。

3.实验过程：

根据实验要求，在终端输入“objdump -d bufbomb > asm.txt”对bufbomb反汇编并获得其反汇编文本文件，并在该反汇编文本文件中寻找smoke函数，如图3.1所示，函数开始地址为0x08048c90。

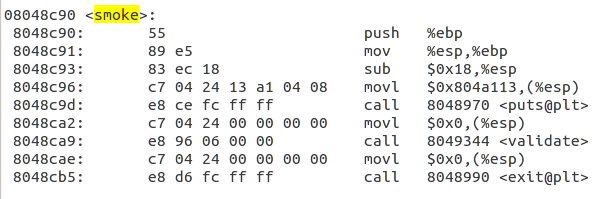


图3.1 smoke函数

同样的在bufbomb的反汇编源代码中找到getbuf函数，如图3.2所示。

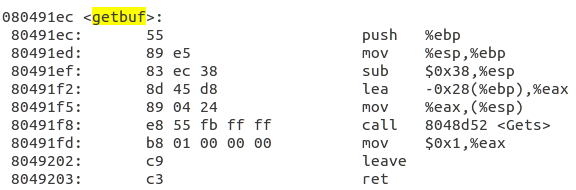


图3.2 getbuf函数

观察它的栈帧结构，如图3.3所示。可以看到getbuf的栈帧是0x38+4个字节，buf缓冲区的大小是0x20个字节，还有8个空闲字节（共40个字节）。

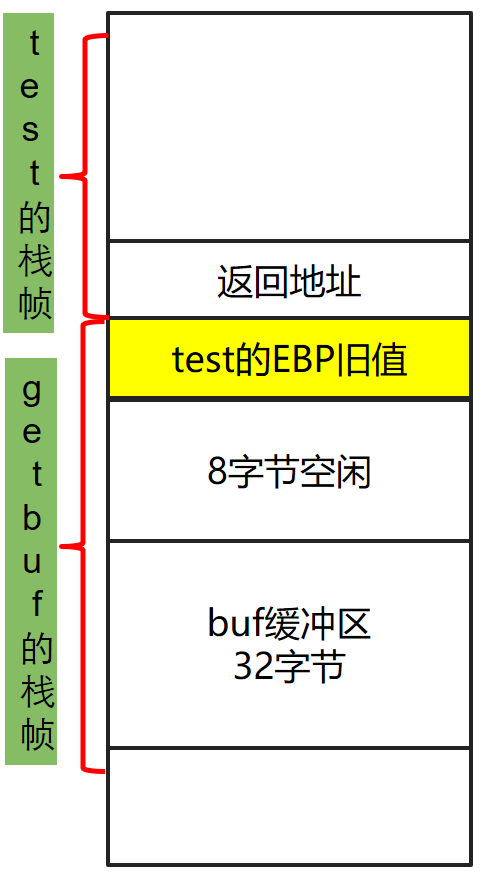


图3.3 getbuf函数栈帧结构

获得了smoke函数的首地址、getbuf函数的函数体及其栈帧结构后即可以开始设计攻击字符串。攻击字符串的功能是用来覆盖getbuf函数内的数组buf，进而溢出并覆盖ebp和ebp上面的返回地址，所以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个字节应是smoke函数的地址。这样的攻击字符串为：

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 **90 8c 04 08**

总共48个字节。前面44个字节可以为任意值，最后四个字节正确地设置为“90 8c 04 08”。

将上述攻击字符串写入攻击字符串文件中并命名为smoke.txt，如图3.4所示。

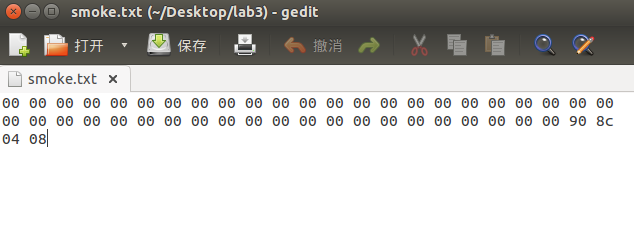


图3.4 smoke.txt

4.实验结果：

使用测试命令进行测试，如图3.5所示，测试成功。

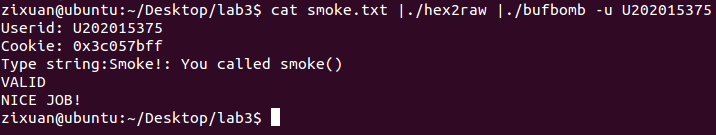


图3.5 smoke测试

### 3.2.2 阶段2 Fizz

1.任务描述：

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

2.实验设计：

根据任务要求，构造一个攻击字符串作为bufbomb的输入，在getbuf()中造成缓冲区溢出，使得getbuf()返回时不是返回到test函数，而是转到fizz函数处执行，并修改fizz函数的参数为自己的cookie。

3.实验过程：

在反汇编文本文件中寻找fizz函数，如图3.6所示，函数开始地址为0x08048cba。

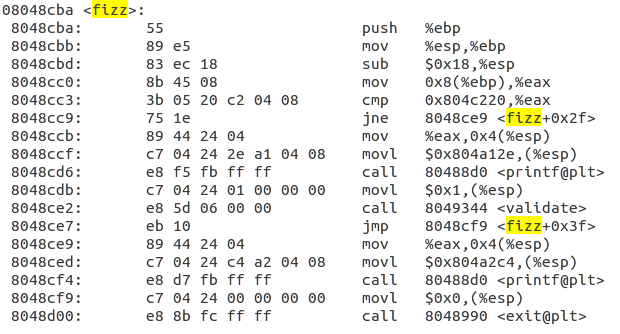


图3.6 fizz函数

在bufbomb.c中查找fizz函数，如图3.7所示，可知需要将其参数val修改为cookie。

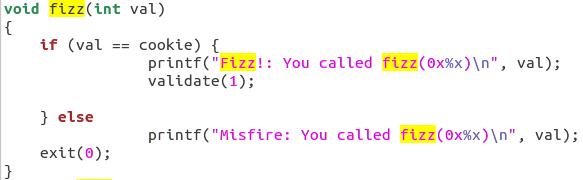


图3.7 fizz函数

使用makecookie工具，在终端输入“makecookie U202015375”语句获得自己的cookie，如图3.8所示。

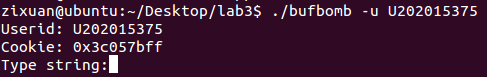


图3.8 获得cookie

在fizz函数中观察到有“mov 0x8(%ebp),%eax”语句，并随后有“cmp 0x804c220,%eax”语句，通过使用gdb调试观察0x804c220处的内容，如图3.9所示。



图3.9 0x804c220处内容

由图中可见0x804c220处即为所生成的cookie，从而可以推断出0x08(%ebp)即为fizz函数的参数val，所以在设计攻击字符串时需要将该处改为cookie。

综合分析getbuf函数和fizz函数，在getbuf函数ret之后栈的ebp和esp回到了调用getbuf函数之前的状态，此时PC为fizz函数首地址。在开始进行fizz函数时首先push了ebp并将ebp赋值为esp，也就是说0x08(%ebp)是getbuf函数ret之后esp的上上个单元，也即val的地址应该为返回地址的上上个单元。栈的变化如图3.10所示。

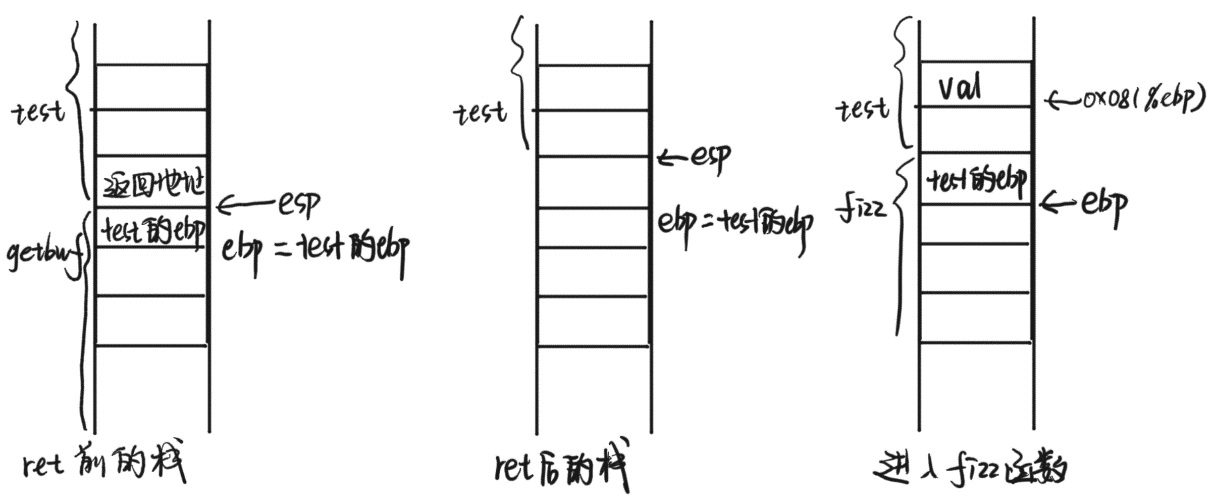


图3.10 栈的变化

分析完val应该存放的地址之后，即可以开始设计攻击字符串。与smoke的字符串类似，按照所分析出的结论设计的攻击字符串如图3.11所示。

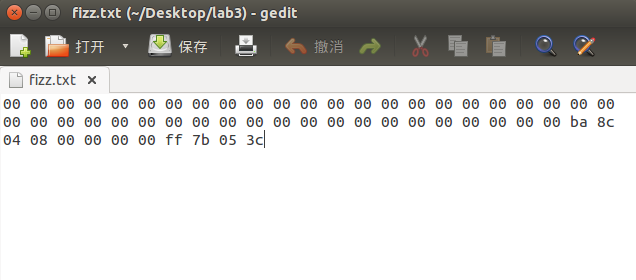


图3.11 fizz.txt

4.实验结果：

使用测试命令进行测试，如图3.12所示，测试成功。

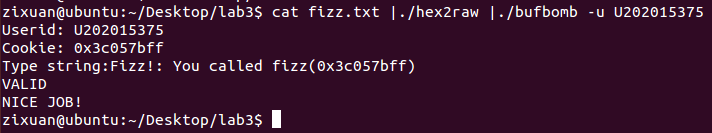


图3.12 fizz测试

### 3.2.3 阶段3 Bang

1.任务描述：

设计包含攻击代码的攻击字符串，所含攻击代码首先将全局变量global\_value设置为cookie值，然后转向执行bang()。

2.实验设计：

根据任务要求，构造一个攻击字符串作为bufbomb的输入，在getbuf()中造成缓冲区溢出，使得getbuf()返回时不是返回到test函数，而是转到bang函数处执行，并修改全局变量global\_value为自己的cookie。

3.实验过程：

在反汇编文本文件中寻找bang函数，如图3.13所示，函数开始地址为0x08048d05。

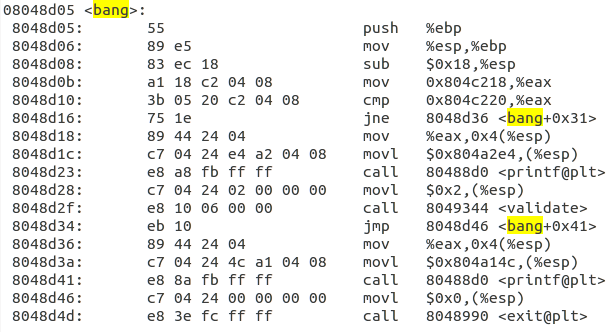


图3.13 bang函数

在bufbomb.c中查找bang函数，如图3.14所示，可知需要将全局变量global\_value修改为cookie。

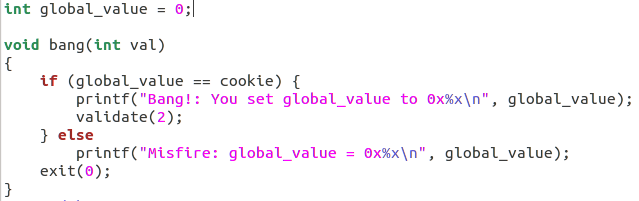


图3.14 fizz函数

要修改全局变量global\_value首先需要获得其地址，通过gdb调试获得global\_value的地址信息如图3.15所示。



图3.15 全局变量global\_value的地址信息

由于全局变量无法通过仅仅修改栈内容来改变，所以需要通过写入攻击代码来实现。编写汇编函数，将cookie赋给global\_value，随后将bang函数的首地址入栈并返回，如图3.16所示。



图3.16 bang.s

通过在终端输入“gcc -m32 -c bang.s”生成bang.o并输入“objdump -d bang.o”获得机器码如图3.17所示。

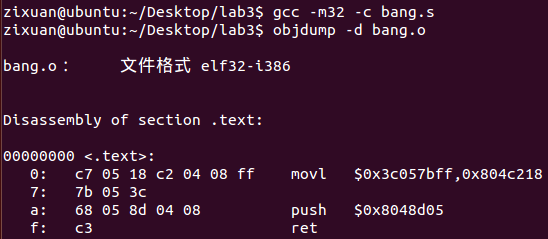


图3.17 bang.s

在编写号函数之后需要获得其入口地址以供跳转，所函数的入口地址即所输入的攻击字符串中函数所放置的位置，从而需要获得攻击字符串的位置。在getbuf函数中，调用Gets函数前，通过“mov %eax,(%esp)”语句可知eax中所存放的即字符串首地址，如图3.18所示。

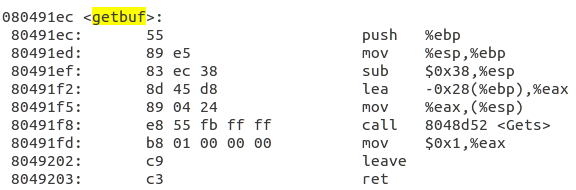


图3.18 0x80491f2到0x80491f8部分

通过gdb在调用Gets函数前设置断点并查看eax内容，如图3.19所示。

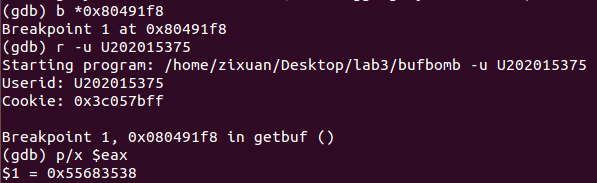


图3.19 查看eax内容

获得eax内容，即字符串首地址后，可以开始设计攻击字符串，将所获得的攻击函数的机器码放在攻击字符串开头，攻击字符串末尾放置字符串首地址即攻击函数入口，如图3.20所示。

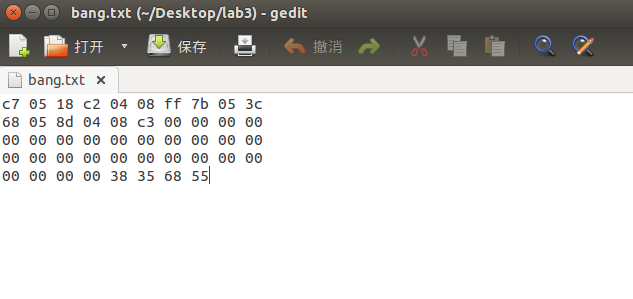


图3.20 bang.txt

4.实验结果：

使用测试命令进行测试，如图3.21所示，测试成功。

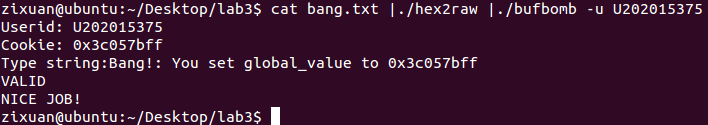


图3.21 bang测试

### 3.2.4 阶段4 Boom

1.任务描述：

构造这样一个攻击字符串，使得getbuf函数不管获得什么输入，都能将正确的cookie值返回给test函数，而不是返回值1。除此之外，攻击代码应还原任何被破坏的状态，将正确返回地址压入栈中，并执行ret指令从而真正返回到test函数。

2.实验设计：

根据任务要求，构造一个攻击字符串作为bufbomb的输入，在getbuf()中造成缓冲区溢出，使得getbuf函数返回到test函数时不是返回1，而是返回自己的cookie。

3.实验过程：

在反汇编文本文件中寻找test函数，如图3.22所示。

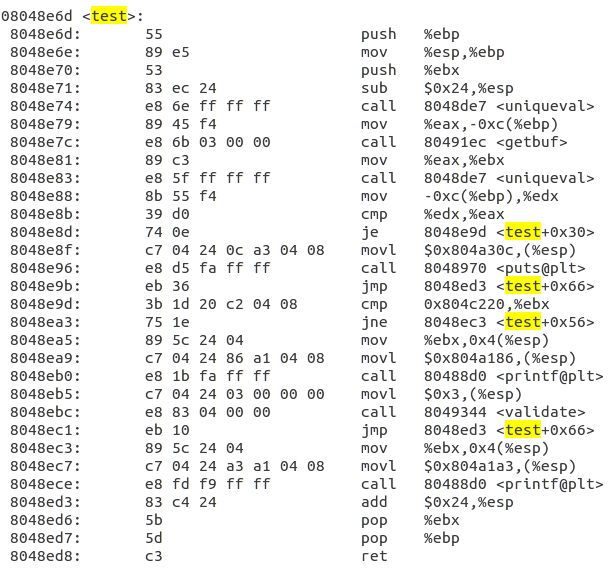


图3.22 test函数

由前面阶段的分析可知，getbuf的返回值存放在eax中，因此编写函数将cookie放入eax后返回到getbuf的下一条语句的首地址即可。编写的函数如图3.23所示。

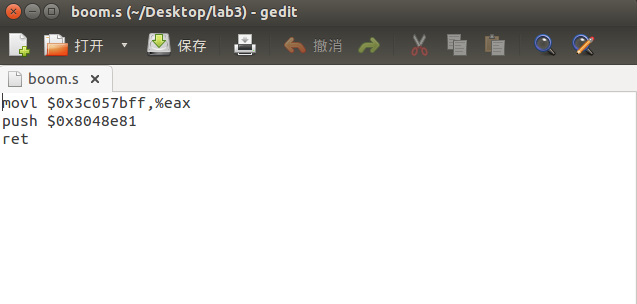


图3.23 boom.s

通过在终端输入“gcc -m32 -c boom.s”生成boom.o并输入“objdump -d boom.o”获得机器码如图3.24所示。

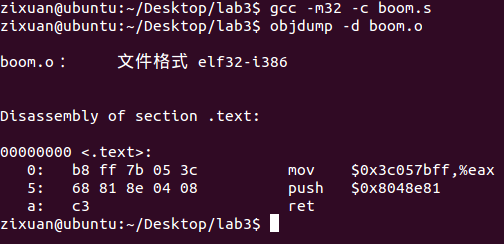


图3.24 boom.s

同时，按照题目要求，不能改变ebp的值，所以也需要获得旧的ebp放在返回地址前。通过gdb在getbuf函数入口处设置断点并查看ebp内容，如图3.25所示。

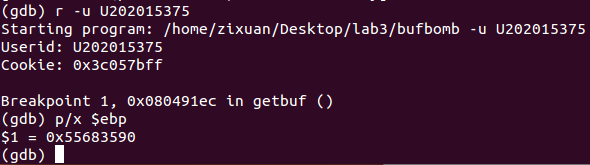


图3.25 旧的ebp

获得了函数的机器码和旧的ebp之后即可开始设计攻击字符串，和上一个阶段相似，机器码放在攻击字符串开头，攻击函数入口地址放在攻击字符串末尾，攻击函数入口地址前放置旧的ebp值，攻击字符串如图3.26所示。

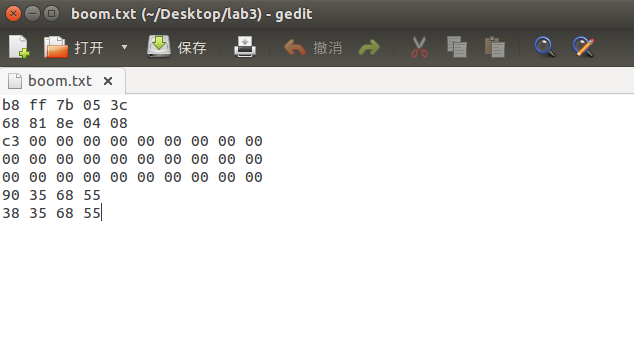


图3.26 boom.txt

4.实验结果：

使用测试命令进行测试，如图3.27所示，测试成功。

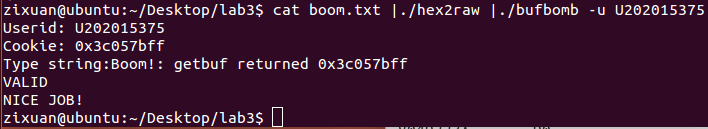


图3.27 boom测试

### 3.2.5 阶段5 Nitro

1.任务描述：

与阶段四类似，构造一攻击字符串使得getbufn函数返回cookie值至testn函数，而不是返回值1。

此时，需要将cookie值设为函数返回值，复原/清除所有被破坏的状态，并将正确的返回位置压入栈中，然后执行ret指令以正确地返回到testn函数。

2.实验设计：

根据任务要求，构造一个攻击字符串作为bufbomb的输入，在getbufn()中造成缓冲区溢出，使得getbufn函数返回到test函数时不是返回1，而是返回自己的cookie。

3.实验过程：

在反汇编文本文件中寻找testn函数和getbufn函数，分别如图3.28和图3.29所示。

在testn函数中观察可知，在开始部分通过push %ebp保存ebp内容后将esp赋给了ebp，随后esp减去0x28，即有%ebp=%esp+0x28，所以可以通过0x28(%esp)来指代ebp的内容。

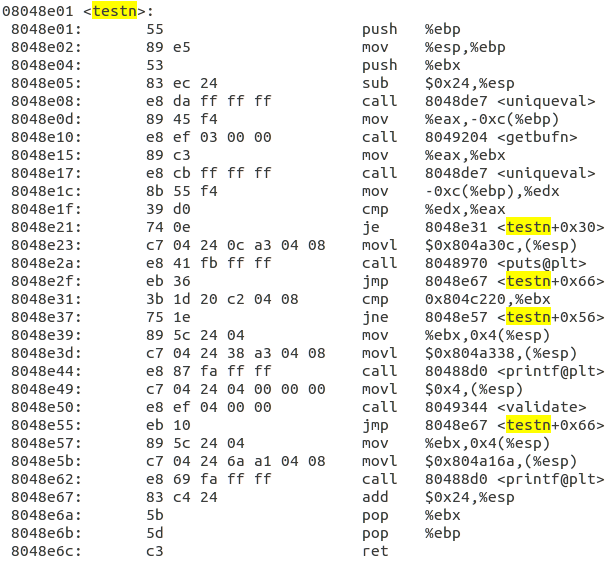


图3.28 testn函数

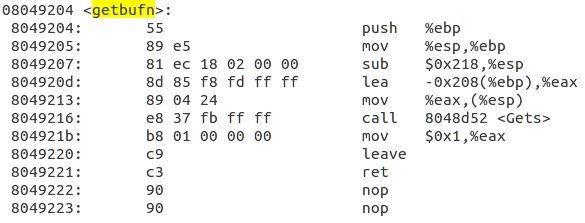


图3.29 getbufn函数

按上述可以编写攻击函数，如图3.30所示。

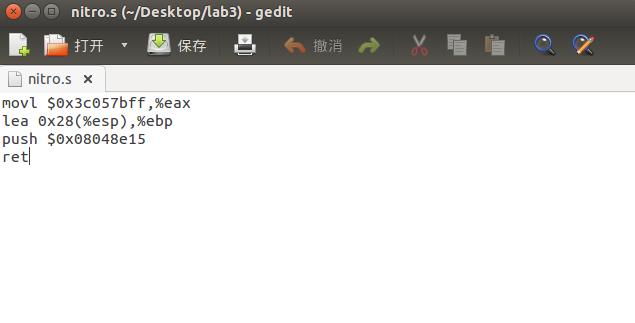


图3.30 nitro.s

通过在终端输入“gcc -m32 -c nitro.s”生成nitro.o并输入“objdump -d nitro.o”获得机器码如图3.31所示。

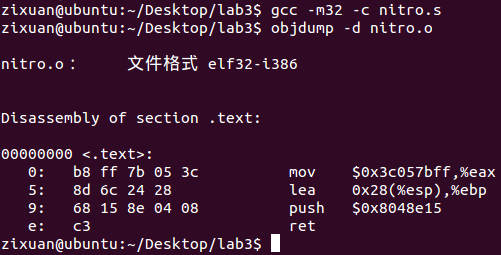


图3.31 nitro.s

由于getbufn函数每次调用时都会随机分配存储地址而编写的跳转地址是固定的，所以在攻击字符串中需要填充nop，从而使得能够成功跳转。

对于攻击函数的入口地址即返回地址，通过五次使用gdb调试工具查看ebp如下所示。选取其中最大的ebp，从而使得返回地址能够成功命中，最大的ebp为0x556835d0，减去0x208得0x556833C8，即获得返回地址。

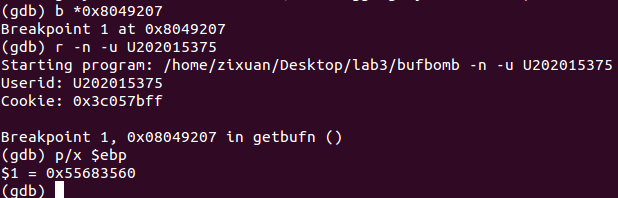


图3.32 第一次

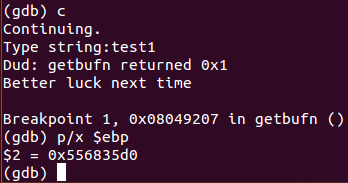


图3.33 第二次

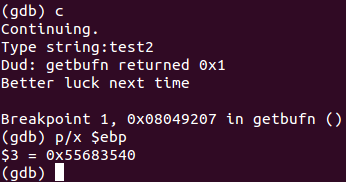


图3.34 第三次

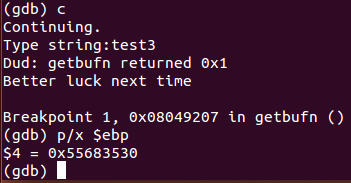


图3.35 第四次

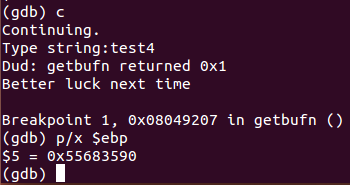


图3.36 第五次

观察getbufn函数，由“lea -0x208(%ebp),%eax”语句可知攻击字符串首地址为-0x208(%ebp)，返回地址为0x04(%ebp)，所以需要攻击字符串应该共0x4-(-0x208)+0x4=528个字节。其中509个nop、15个字节的机器码以及4个字节的返回地址。

完成上述分析后开始设计攻击字符串，如图3.37所示。

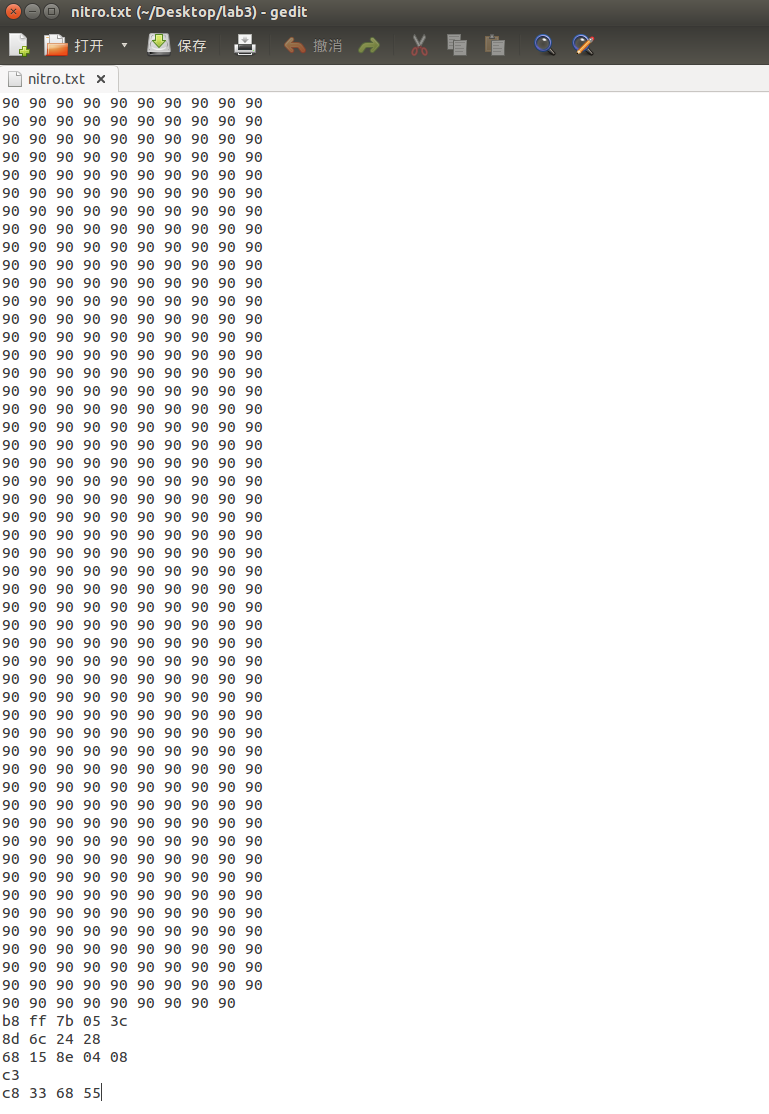


图3.37 nitro.txt

4.实验结果：

使用测试命令进行测试，如图3.38所示，测试成功。

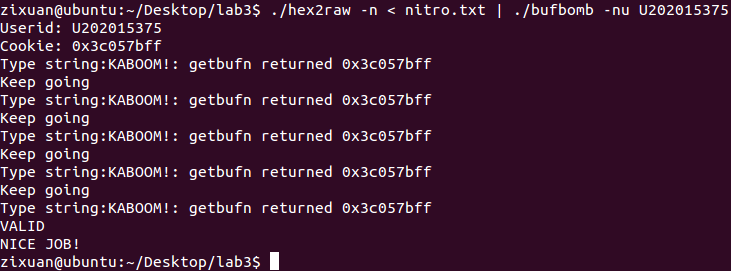


图3.38 nitro测试

## 3.3实验小结

本次实验是在linux环境下通过分析反汇编文件并利用gdb调试工具来分析栈帧并设计对应攻击字符串以及攻击函数来攻击bufbomb从而获得所期望的运行结果。总体而言是比较有难度的。

在实验二已经接触并熟悉了linux环境和gdb调试工具的基础上，实验三在实验开始前的准备工作减少了很多，在进行实验的过程中也能够比较熟练地进行各种相关操作。但是由于实验三本身的难度，在本次实验上花费的时间也并不比实验二少。

在实验三中，成功通关的关键在于能够理解并熟练掌握函数的调用返回与栈帧的变化等相关知识点，以及学会如何通过缓冲区溢出来改变栈帧内容。为了实现各个阶段的不同任务要求，需要仔细思考斟酌对于攻击字符串的设计，根据要求在函数中寻找线索，利用gdb调试工具寻找关键信息。

本次实验是我第一次实际体验到了对于缓冲区溢出的操作与实现，十分新颖却又有着一定程度上的困难。设计攻击字符串来实现缓冲区溢出，加深了我对于函数的调用返回、栈帧的变化等相关知识点的认知，也对与缓冲区溢出有了进一步的了解。

总体而言，这次实验较为圆满地完成，各个关卡均能成功通关，取得了较为让人满意的成果。

# 实验总结

本次计算机系统基础实验总体上来说是比较由挑战性、同时也十分有趣的。在我到目前为止所做过的实验中是最特别的实验之一。不管是linux环境还是对于拆炸弹实验和缓冲区溢出实验，都是头一次接触，是全新的体验。

由于之前从未接触过linux环境，所以在实验前的准备工作中，遇到了阻碍。诸如虚拟机软件VMware Workstation的安装、主机和虚拟机之间文件的传输、VMware Tools的安装与使用时所出现的问题的处理、linux系统的选择等一系列问题，都花费了我很多时间去完成和解决。但是在这些全部解决了之后，对于linux系统的使用和实验的进行让我感觉十分新颖，第一次实现虚拟机的安装与linux环境的使用无疑扩宽了我的视野、让我获得了新的技能。

在进行实验二的过程中，由于有过在汇编语言程序设计实验中类似的实验经验，我一开始以为做起实验二会比较得心应手，但是在做到phase\_4部分开始使用递归函数时我发现并非我想象得那么简单，在这一部分由于函数递归导致esp和ebp频繁地改变让我晕头转向，多次弄混堆栈的情况，好在最后思考清楚之后终于能够成功理解并通关。

在进行实验三的过程中，缓冲区溢出的操作不仅让我对函数的调用、堆栈的变化等有了更深层次的理解，也让我意识到了编写程序时对于溢出这方面的保护的重要性。

总体上来说，这三次实验虽然有一定的难度，但是以一种有趣的方式带领我进一步掌握了相关的知识点，对于linux环境的使用也有了更多了解，对于汇编语言、计算机系统等也有了更深的认识，总体而言是大有所益、大有收获的。