

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**专业班级： CS2011**

**学 号： U202011675**

**姓 名： 徐锦慧**

**指导教师： 张宇**

**报告日期： 2022年6月14日**

**计算机科学与技术学院**

**目录**

[1 实验2：拆弹实验 3](#_Toc45)

[1.1 实验概述 3](#_Toc45)

[1.2 实验要求 3](#_Toc12776)

[1.3 实验内容 3](#_Toc26710)

[1.3.1 阶段1 拆除<phase\_1> 3](#_Toc13085)

[1.3.2 阶段2 拆除<phase\_2> 6](#_Toc21957)

[1.3.3 阶段3 拆除<phase\_3> 8](#_Toc15690)

[1.3.4 阶段4 拆除<phase\_4> 10](#_Toc20207)

[1.3.5 阶段5 拆除<phase\_5> 13](#_Toc23886)

[1.3.6 阶段6 拆除<phase\_6> 15](#_Toc21627)

[1.3.7 阶段7 寻找隐藏区域的入口 18](#_Toc907)

[2 实验3：缓冲区溢出攻击 2](#_Toc45)4

[2.1 实验概述 2](#_Toc45)4

[2.2 实验要求 2](#_Toc12776)4

[2.3 实验内容 2](#_Toc26710)4

[2.3.1 阶段1 smoke 2](#_Toc13085)4

[2.3.2 阶段2 fizz 27](#_Toc21957)

[2.3.3 阶段3 bang 28](#_Toc15690)

[2.3.4 阶段4 boom 31](#_Toc20207)

[2.3.5 阶段5 nitro 33](#_Toc23886)

[3 实验总结 38](#_Toc1603)

[3.1 实验2小结 38](#_Toc1603)

[3.2 实验3小结 38](#_Toc1603)

[参考文献 40](#_Toc1603)

# 1 实验2：拆弹实验

# 1.1 实验概述

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。

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输入一个特定的字符串，若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。实验的目标是需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，但只有在第4阶段的解之后附加一特定字符串后才会出现。

实验语言：C语言。

实验环境：linux。

# 1.2 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. 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。

# 1.3 实验内容

## 1.3.1 阶段1 拆除<phase\_1>

1.实验描述

找到与输入的字符串进行比较的存储的字符串的首地址，进而得到存储的字符串，得到结果。

2.实验设计

通过objdump对C语言文件进行反汇编，根据反汇编代码一步一步分析，并使用gdb调试，具体见实验过程。

3.实验过程

通过objdump对bomb进行反汇编并将汇编代码输出到bomb.asm中，如图1.1.1所示。

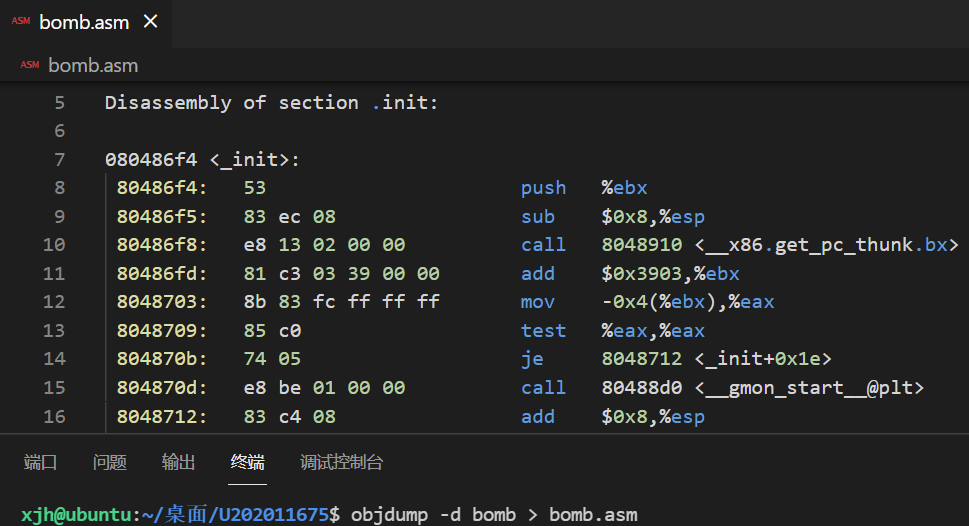


图1.1.1 对bomb进行反汇编

查看汇编源代码asm.txt文件， 在main函数中找到如下语句，这里为phase\_1函数在main()函数中被调用的位置，如图1.1.2所示。

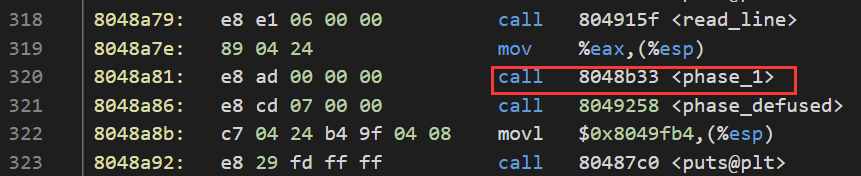


图1.1.2 查找main函数中的phase\_1

在反汇编文件中继续查找phase\_1函数的具体实现，如图1.1.3所示。

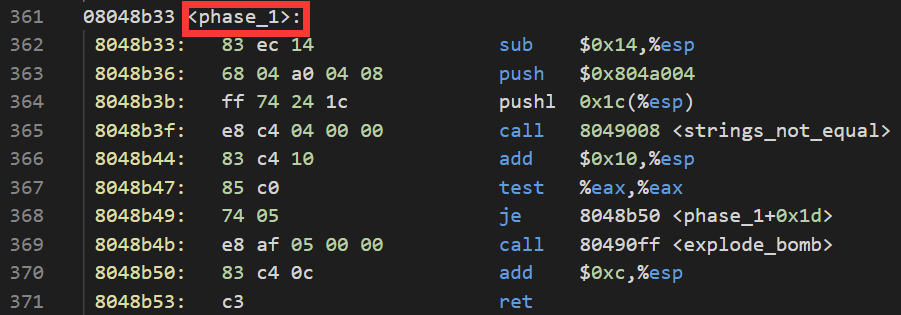


图1.1.3 查找phase\_1

通过分析<phase\_1>函数, 可以发现在调用 <string\_not\_equal>函数之前, push了两个地址，在main()函数调用了read\_line()，如图1.1.4所示。

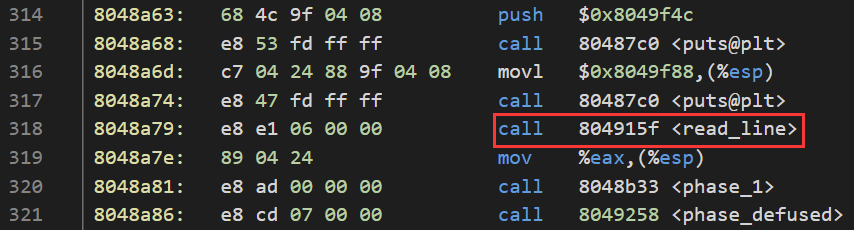


图1.1.4 查找read\_line()

%eax里存储的是调用read\_line()函数返回值，也是用户输入的字符串首地址。而在phase\_1中，调用strings\_not\_equal前利用堆栈传了两个参数，画出栈帧图，如图1.1.5所示。

|  |
| --- |
| 用户输入的字符串 |
| 返回地址 |
|  |
|  |
|  |
|  |
|  |
| 0x804a004 |
| 用户输入的字符串地址 |

图1.1.5 栈桢图

推测拆弹密码字符串的存储地址为0x804a004，因为调用strings\_not\_equal前有语句：push $0x804a004。用gdb查看这个地址存储的数据内容，可以得到第一个密码为 “There are rumors on the internets.”，如图1.1.6所示。

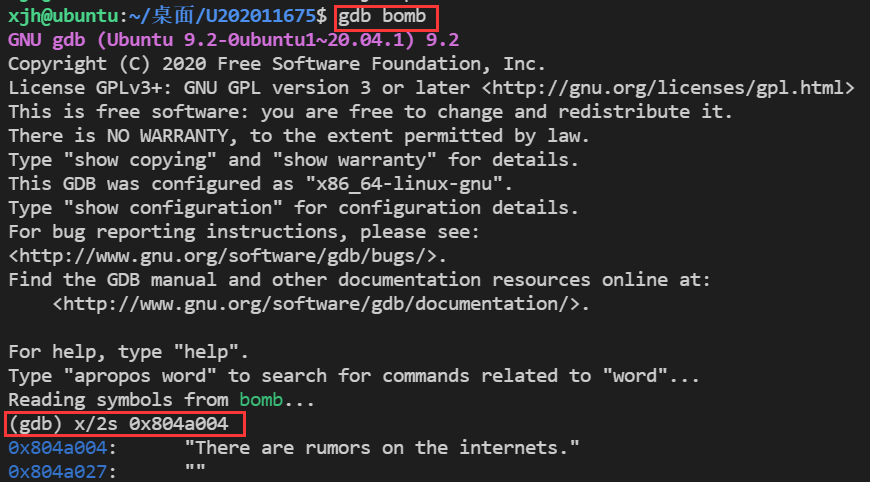


图1.1.6 用gdb查看0x804a004处数据

4.实验结果

阶段1的测试结果如图1.1.7所示，可知阶段1能够被正确拆弹。

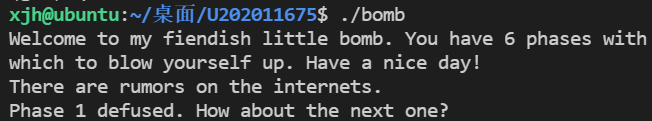


图1.1.7 阶段1测试结果

## 1.3.2 阶段2 拆除<phase\_2>

1.实验描述

找到与输入的字符串进行比较的存储的字符串的首地址，进而得到存储的字符串，完成炸弹2的拆除。

2.实验设计

通过objdump对C语言文件进行反汇编，根据反汇编代码一步一步分析，并使用gdb调试，具体见实验过程。

3.实验过程

观察函数<phase\_2>, 通过<read\_six\_numbers>可以知道一共有6个数字, 如图1.2.1所示。若第一个数和0不相等，则会跳转到引爆语句；若第二个数和1不相等，也会执行引爆语句。

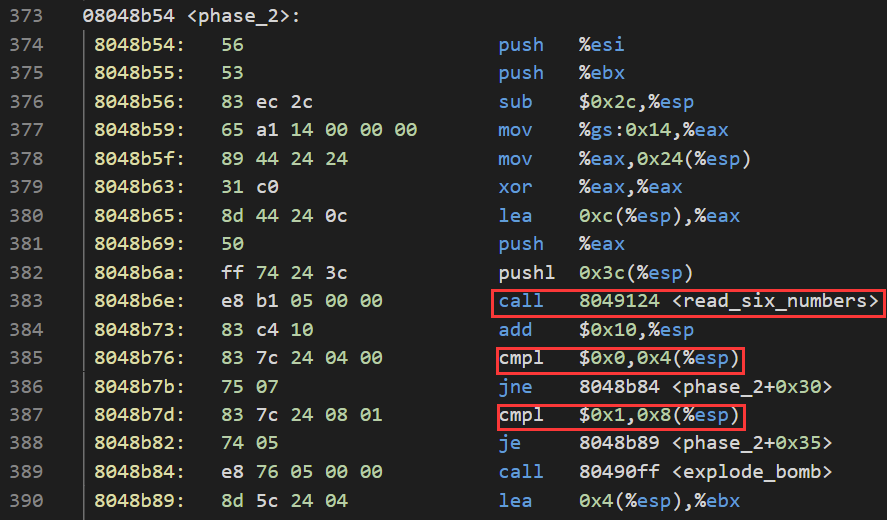


图1.2.1 判断前2个字符

程序接着连续两次跳转, 到达第二个会引起爆炸的地方, 如图1.2.2所示。可以看到又对eax和输入的第3个数进行了一次比较, 其中eax的值是输入的前2个数之和，而0x8(%ebx)则是输入的第3个数字, 所以第3个数字为1。

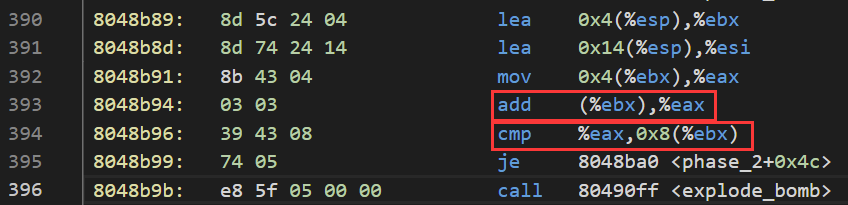


图1.2.2 判断第3个字符

接着就是一个循环，利用%esi和%ebx的比较来循环判断剩下的数字，如图1.2.3所示。当%esi和%ebx中的值相等时，循环结束，此时所有的数字都判断完毕，否则跳转至8048b91处，继续循环。每次循环都将前2个数字之和放到%eax中，与下一个数字比较。所以这6个数字构成了斐波那契数列，即第2阶段的答案为：0 1 1 2 3 5。

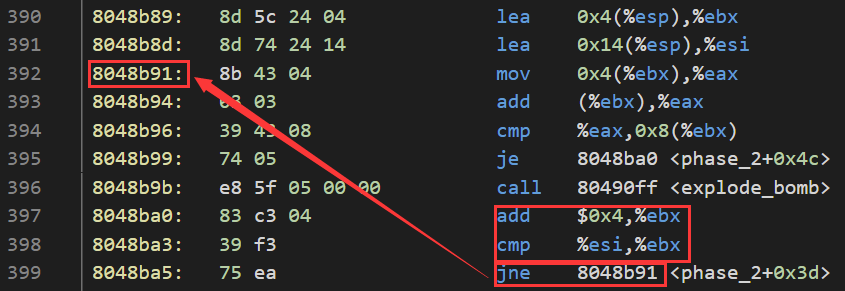


图1.2.3 判断剩余字符

4.实验结果

阶段2的测试结果如图1.2.4所示，可知阶段2能够被正确拆弹。

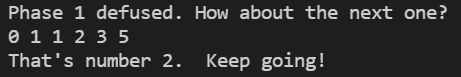


图1.2.4 阶段2测试结果

## 1.3.3 阶段3 拆除<phase\_3>

1.实验描述

找到与输入的字符串进行比较的存储的字符串的首地址，进而得到存储的字符串，完成炸弹3的拆除。

2.实验设计

通过objdump对C语言文件进行反汇编，根据反汇编代码一步一步分析，并使用gdb调试，具体见实验过程。

3.实验过程

先观察函数，发现调用了sscanf函数，如图1.3.1所示。

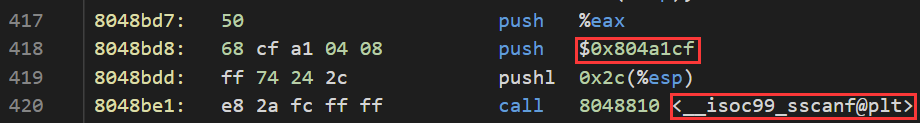


图1.3.1 调用sscanf函数

通过查看0x804a1cf的值, 可以发现要求输入的是两个整数，如图1.3.2所示。

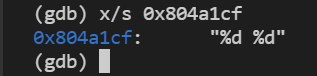


图1.3.2 判断输入类型

之后查看会引起的爆炸的语句, 根据比较可以说明第一个参数需要小于7，如图1.3.3所示。

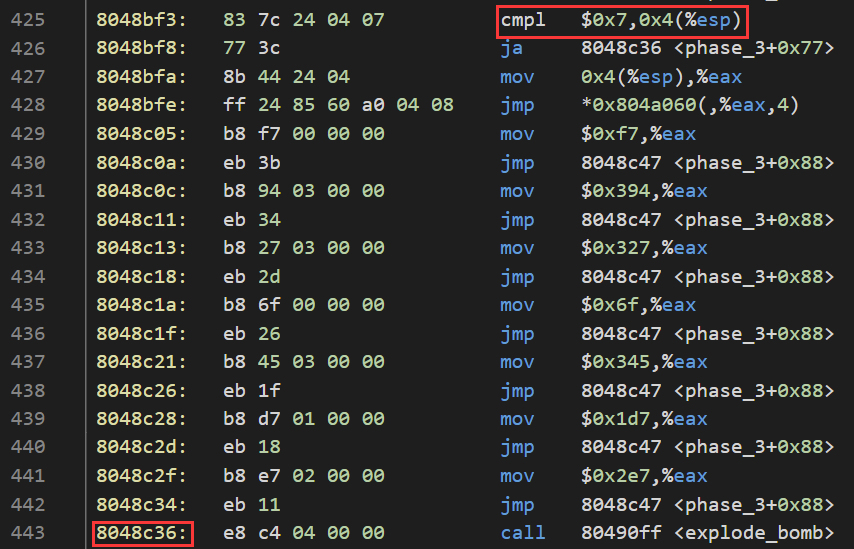


图1.3.3 判断第1个数

观察其后的跳转语句, 可以发现根据第一个输入数的不同, 跳转的语句也不同, 这里只讨论输入1的情况, 会跳转到8048c05这一行, 之后将0xf7送到eax中, 转换为十进制为247, 并将这个数与输入的第二个数字进行比较, 若二者不相等，则执行引爆语句，故第二个数即为247。

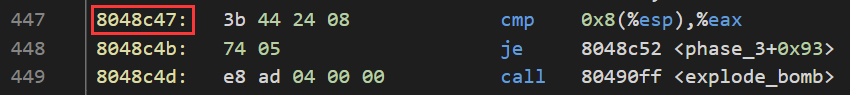


图1.3.4 判断第2个数

4.实验结果

阶段3的测试结果如图3.5所示，可知阶段3能够被正确拆弹。

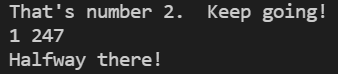


图1.3.5 阶段3测试结果

## 1.3.4 阶段4 拆除<phase\_4>

1.实验描述

找到与输入的字符串进行比较的存储的字符串的首地址，进而得到存储的字符串，完成炸弹4的拆除。

2.实验设计

通过objdump对C语言文件进行反汇编，根据反汇编代码一步一步分析，并使用gdb调试，具体见实验过程。

3.实验过程

首先观察函数phase\_4，发现调用了sscanf函数，如图1.4.1所示。

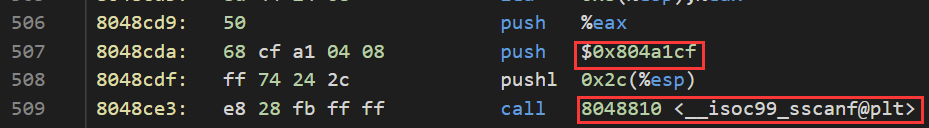


图1.4.1 调用sscanf函数

通过查看0x804a1cf的值, 可以发现要求输入的是两个整数，如图4.2所示。



图4.2 判断输入类型

之后继续观察函数, 将输入的第一个数和0xe进行比较, 可以得出第一个应该小于等于14。如图1.4.3所示。

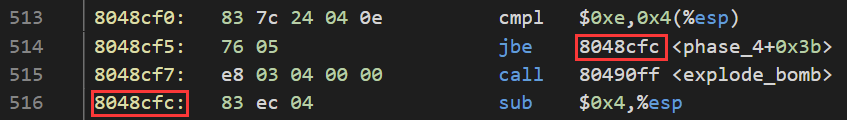


图1.4.3 判断第1个数

之后查看进入func4函数之前, 发现传入了3个参数, 第一个为输入的第2个数, 第2个为0, 第3个为14，如图1.4.4所示。

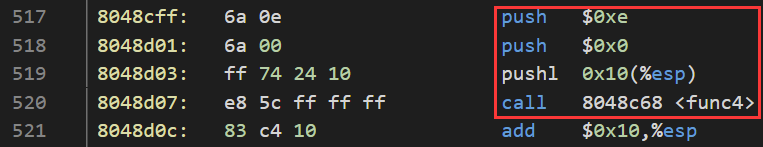


图1.4.4 调用func4函数

之后观察fun4函数，可以知道, ecx为输入的第1个数，ebx为0，esi为14, 如图1.4.5所示。

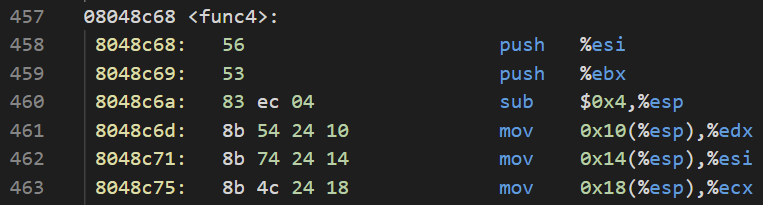


图1.4.5 判断func传入参数

继续查看func4函数体，如图1.4.6所示。

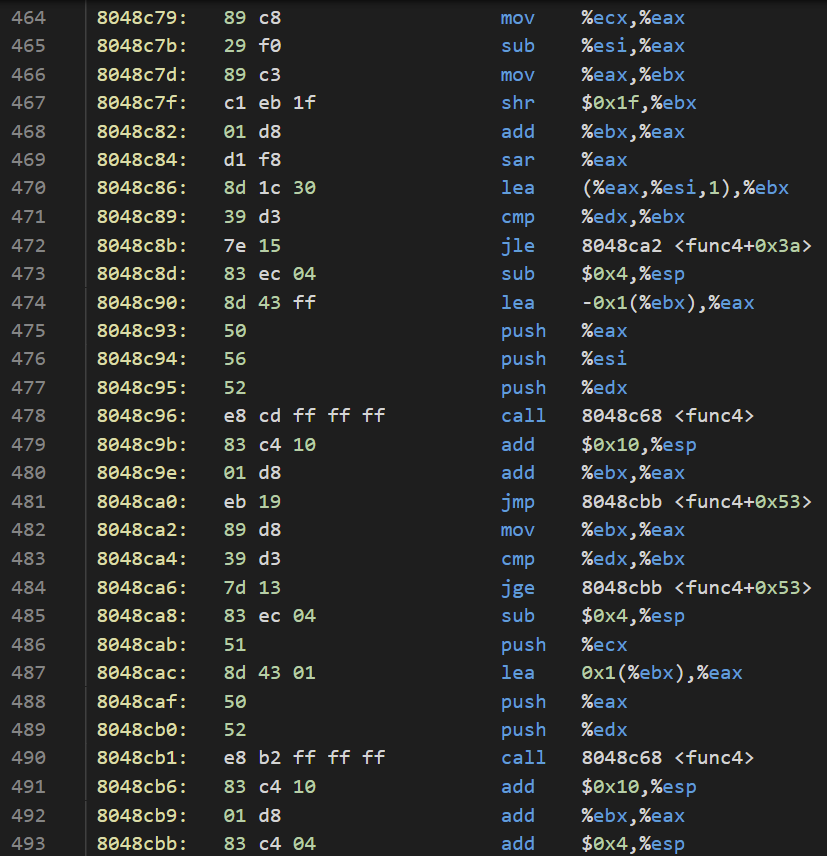


图1.4.6 func4具体实现

可以一步步将其函数写出, C语言程序如图1.4.7所示。

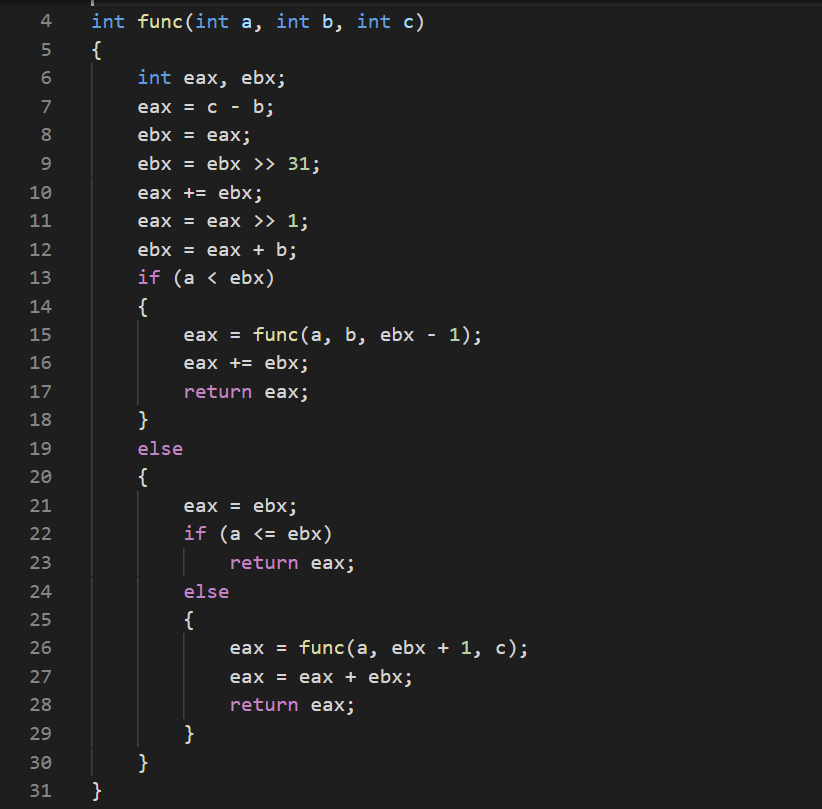


图1.4.7 对应C语言程序

之后查看func4函数执行后的程序，如图1.4.8所示。

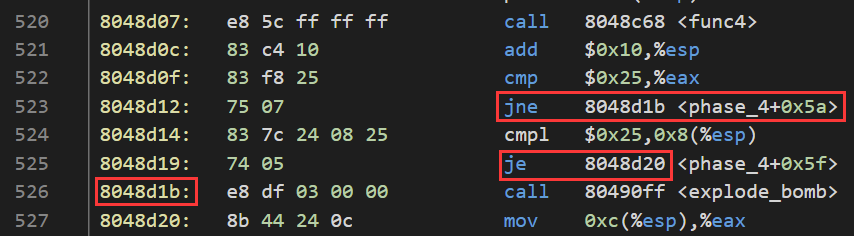


图1.4.8 判断func执行后结果

可以知道输入的第一个数经过运算后结果应该为0x25, 第二个数也应该为0x25, 而经过计算可以知道第一个输入为10时, 运算后为37, 可以知道两个数字分别为10，37。

4.实验结果

阶段4的测试结果如图1.4.9所示，可知阶段4能够被正确拆弹。

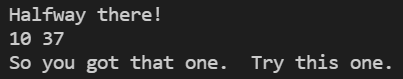


图1.4.9 阶段4测试结果

## 1.3.5 阶段5 拆除<phase\_5>

1.实验描述

找到与输入的字符串进行比较的存储的字符串的首地址，进而得到存储的字符串，完成炸弹5的拆除。

2.实验设计

通过objdump对C语言文件进行反汇编，根据反汇编代码一步一步分析，并使用gdb调试，具体见实验过程。

3.实验过程

首先观察函数phase\_5，发现在第545行处调用了sscanf函数，如图1.5.1所示。

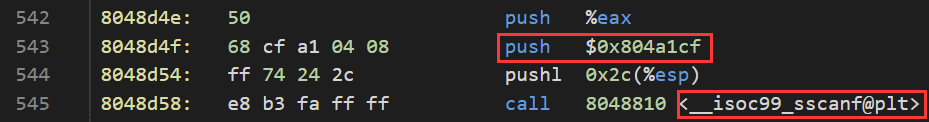


图1.5.1 调用sscanf函数

接着，通过gdb查看0x804a1cf处的值, 可以发现要求输入的是两个整数，如图1.5.2所示。



图1.5.2 判断输入类型

接下来看函数主体部分，如图5.3所示。可知先取用户所输入的第一个参数的低4位保存至eax中，若其为15，则直接引爆炸弹。在不为15的情况下，可知会从（0x804a080+eax）地址中取出数字赋给eax，并累加到寄存器ecx中，当累加到元素15时便跳出循环。

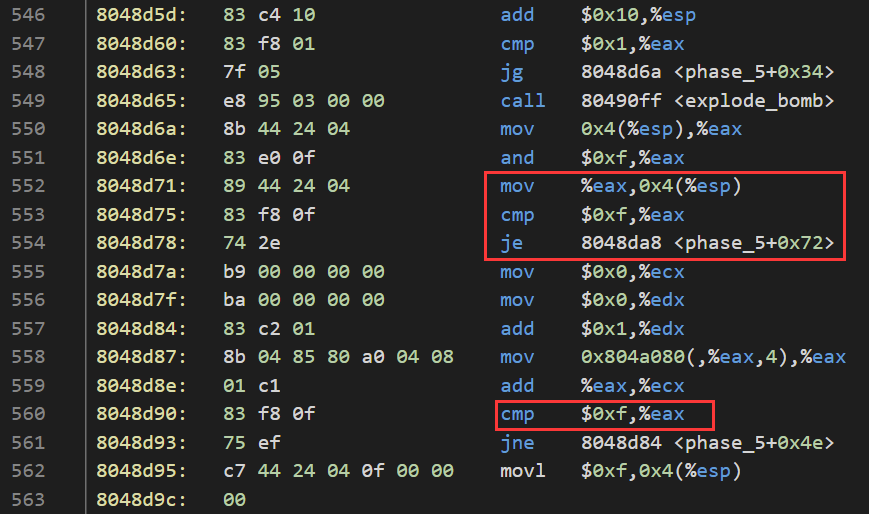


图1.5.3 phase\_5函数

再根据下图中代码可知，在跳出循环时，需要进行15次加法，并且累加结果需要与用户输入的第二个参数相等。

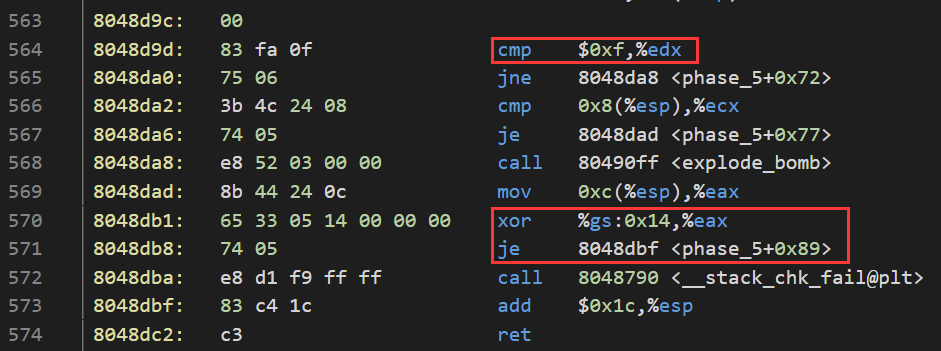


图1.5.4 phase\_5函数

因此猜测地址0x804a080是一个大小为15的数组的首地址，使用gdb查看地址内容，如下图。

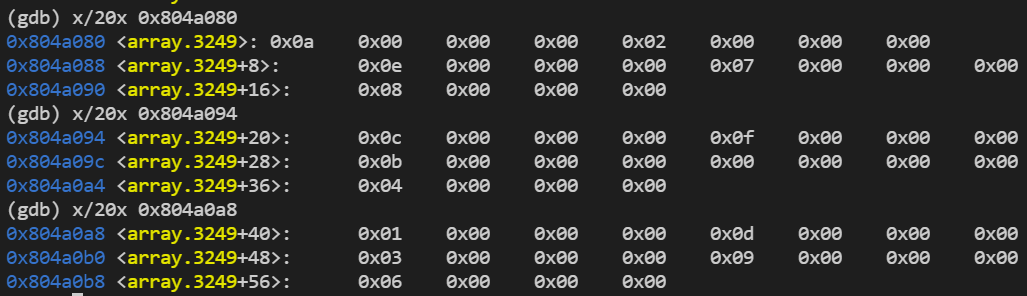


图1.5.5 0x804a4080处数据

可以得到第一个输入的数为下标, 来查找数组中该下标存储的值, 以该值为下标继续循环, 当第一次出现15为第15次循环时, 才能成功解除。

通过遍历可得, 当第一个数为5时, 恰好可以出现15时是第15次循环, 则第二个数的值为C + 3 + 7 + B + D + 9 + 4 + 8 + 0 + A + 1 + 2 + E + 6 + F = 115, 即两个数为5，115。

4.实验结果

阶段5的测试结果如图5.6所示，可知阶段5能够被正确拆弹。



图1.5.6 阶段5测试结果

## 1.3.6 阶段6 拆除<phase\_6>

1.实验描述

找到与输入的字符串进行比较的存储的字符串的首地址，进而得到存储的字符串，完成炸弹6的拆除。

2.实验设计

通过objdump对C语言文件进行反汇编，根据反汇编代码一步一步分析，并使用gdb调试，具体见实验过程。

3.实验过程

这一部分分为多个模块分析，首先可以得到图1.6.1所示的这一部分。

第590行将eax减1，eax中的值是esp位置存放的值。第591行将减一以后的值与5进行比较，小于等于5则跳过引爆代码。也就是说esp中存放的第一个数必须小于等于6。

以此类推，继续分析后面的循环后可以发现，这部分主要是确定读入的6个数字都小于等于6,并且两两之间不相等。

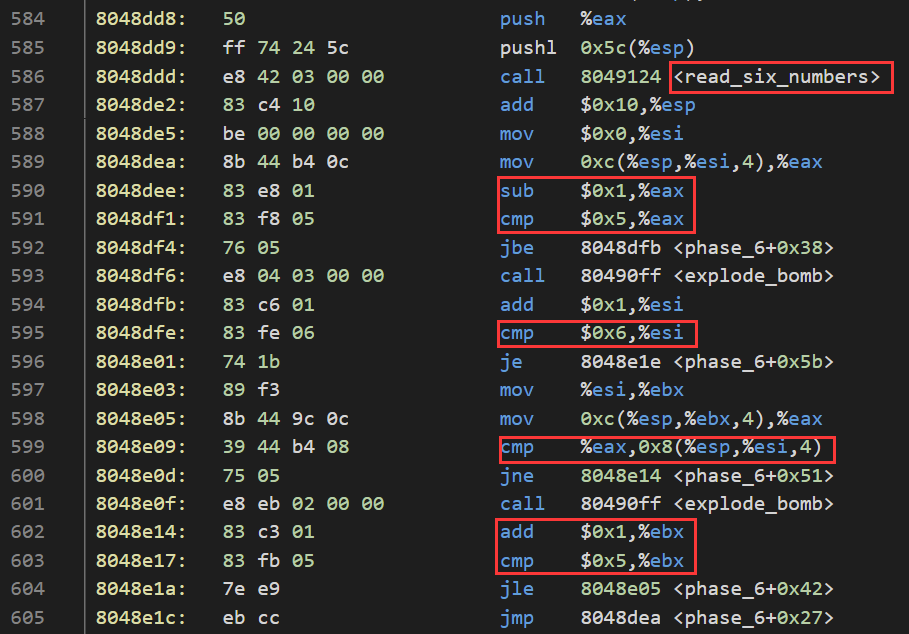


图1.6.1 判断输入类型

接着分析下一部分。第608行将0x7赋给ecx，第609行又将ecx复制给edx。下一步将edx减去eax存放的地址指向的值，接下来又将edx的值赋回eax存放的地址指向的值。第612行将eax的值加4，也就是指向了下一个int值，接着与之前设定的ebx进行的比较，如果不相等则重复这个过程。这段代码总结起来就是将栈中的6个值（假设为x）变为7-x。

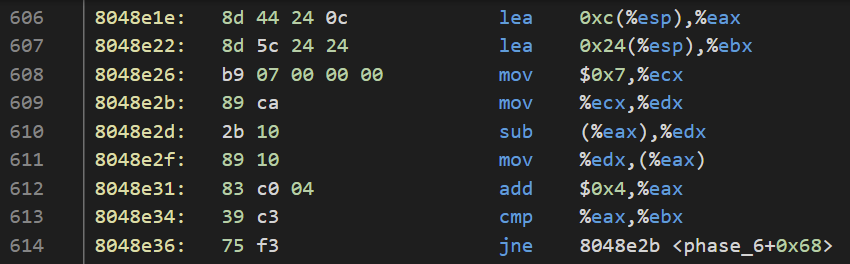


图1.6.2 将6个值从x变为7-x

继续观察，发现在函数中用到了0x804c13c处的数据，如图1.6.3所示。

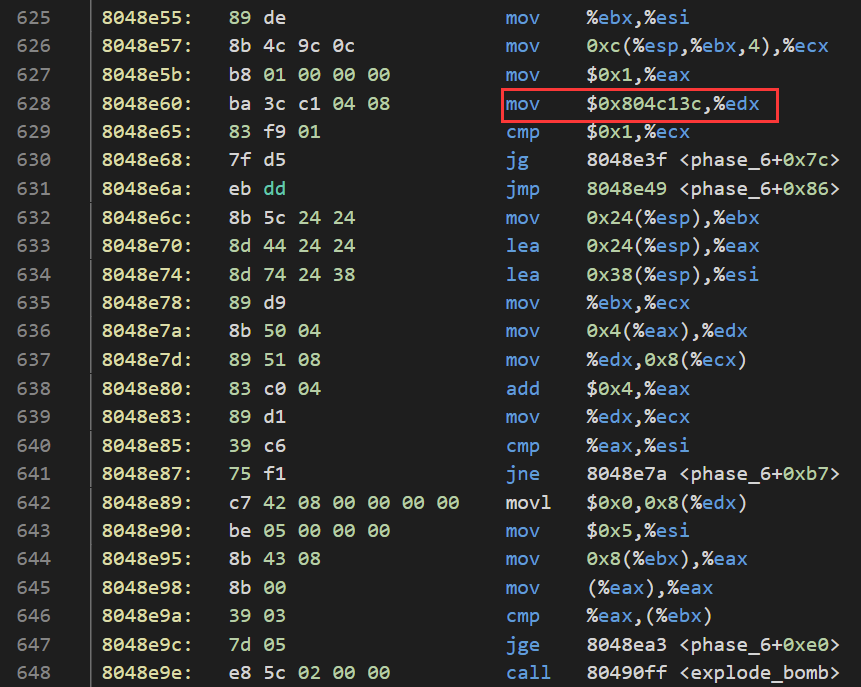


图1.6.3 将0x804c13c处赋值给ebx

通过gdb查看该地址, 可以得到一个链表，如图1.6.4所示。

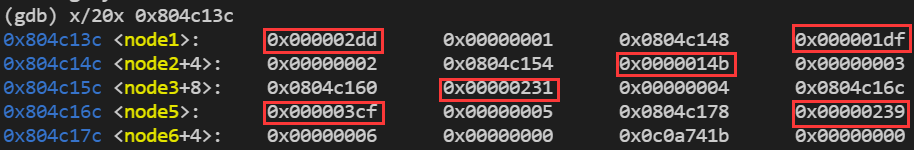


图1.6.4 0x804c13c处数据

继续分析<phase\_6>。第643行将esi赋上0x5，第644行中ebx的值是之前的esp+0x20，那么ebx+0x8这个地址中存放的值就是下一个节点的地址，赋给了eax。

第645行将eax代表的节点的数据取出放入eax，再与ebx代表的节点的数据的值的低4位进行比较，如果前一个节点的数据的低4字节大于等于后一个节点的，则跳过引爆代码。

接着，使ebx指向下一个节点,减小esi这个循环变量再进行判断，保证循环进行5次。也就是说，我们需要使新的链表中前一个节点存放的数据值的低4字节都大于后一个节点的。

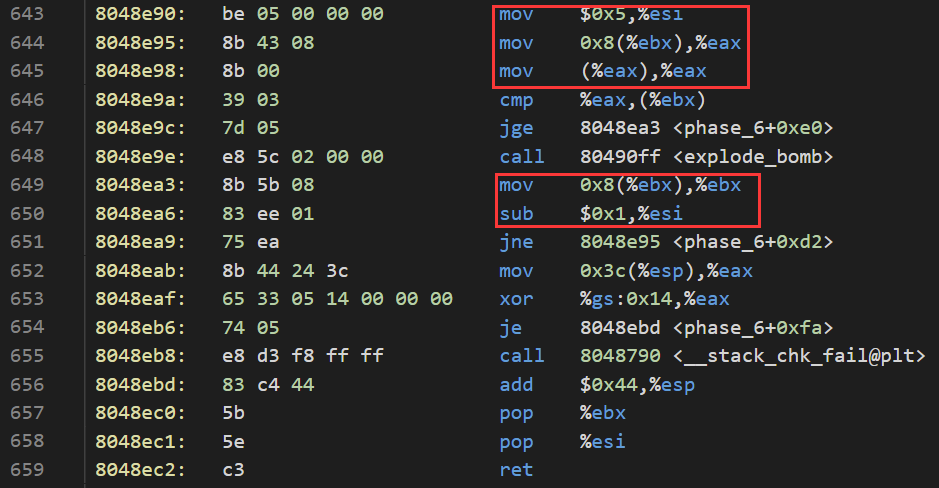


图1.6.5 将链表降序排序

综上，我们可以得出该函数的功能：将输入的6个数降序排序，并对降序后的链表进行模7取补。根据用gdp调试出的链表数据，我们可以得到链表降序后的值为： 5 1 6 4 2 3, 与7做差后为：2 6 1 3 5 4。

4.实验结果

阶段6的测试结果如图1.6.6所示，可知阶段6能够被正确拆弹。

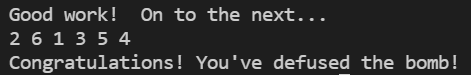


图1.6.6 阶段6测试结果

## 1.3.7 阶段7 隐藏阶段

1.实验描述

找到进入隐藏阶段的字符串，并拆除隐藏阶段炸弹。

2.实验设计

通过objdump对C语言文件进行反汇编，根据反汇编代码一步一步分析，并使用gdb调试，具体见实验过程。

3.实验过程

搜索整个文件,发现只有一处调用了<secret\_phase>, 在<phase\_defused>中，其中需要在0x804c3cc地址中的值等于6的时候才会触发，如图1.7.1所示。

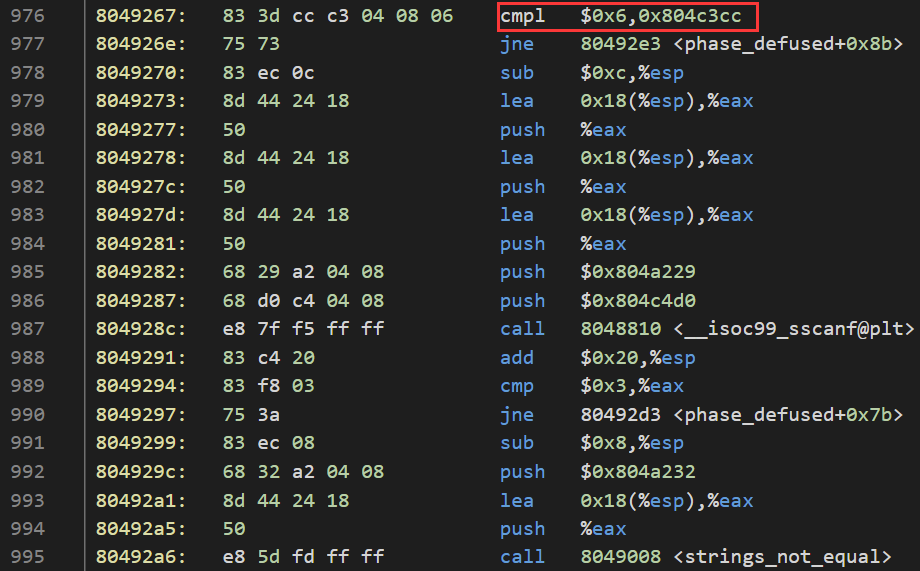


图1.7.1 调用<secret\_phase>

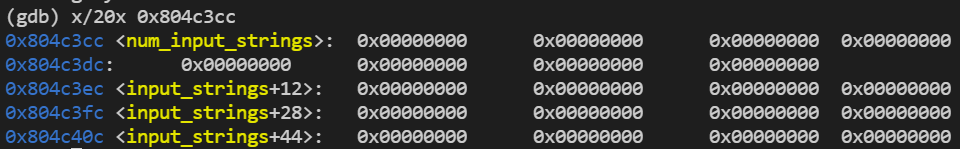


图1.7.2 查看0x804c3cc处数值

经过实验分析, 该地址储存的为当前进行到的phase的序号，也即，只有当第六个阶段完成之后才会触发

而在<phase\_defused>中还有一段判断字符串是否相等的代码, 查看与phase1基本相同，通过查看该位置的值, 可以得到一个一段值为 “DrEvil”的字符串，如图1.7.3所示。

C:\Users\29393\Documents\Tencent Files\2939361916\Image\C2C\1$0%9ZL@PH}US~RTJE{54$G.png

图1.7.3 隐藏阶段字符串

根据实验所给出的提示，将其加在第四个阶段的答案之后, 即可进入隐藏phase，如图1.7.4所示。

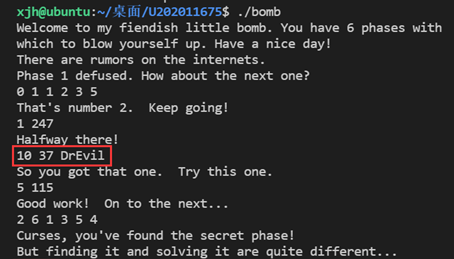


图1.7.4 进入隐藏阶段

接下来就对<secret\_phase>进行分析，如图1.7.5所示。

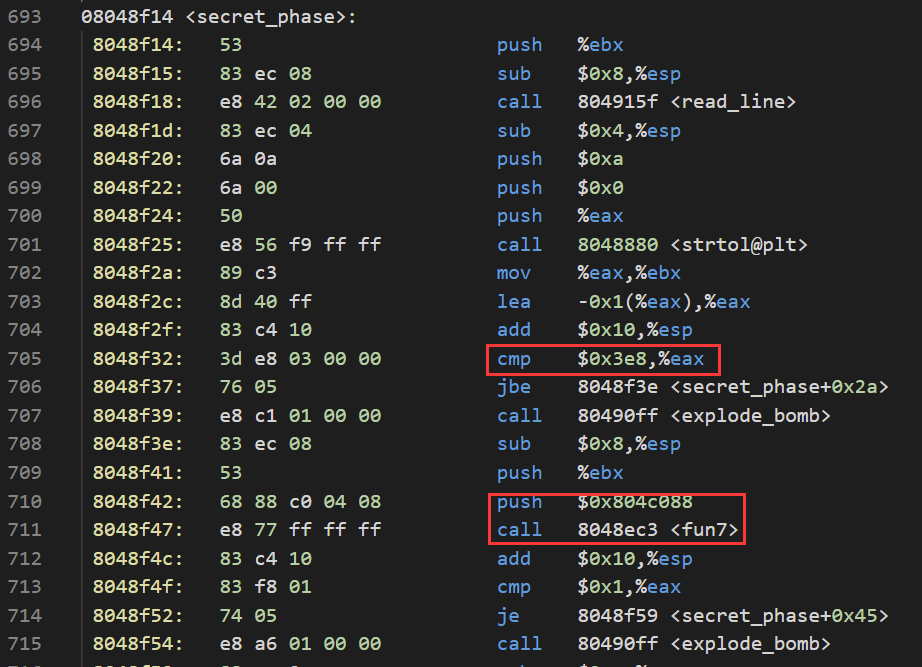


图1.7.5 secret\_phase函数

可以得到这一段是将eax，也就是将字符串参数转化为十进制数字并储存在ebx中, 并且可以知道值小于0x3e8, 即为十进制的1000。

在其后就调用了一个递归函数fun7,最初传进的参数中有一个固定地址0x804c088。通过gdb查看该处值，如图1.7.6所示。

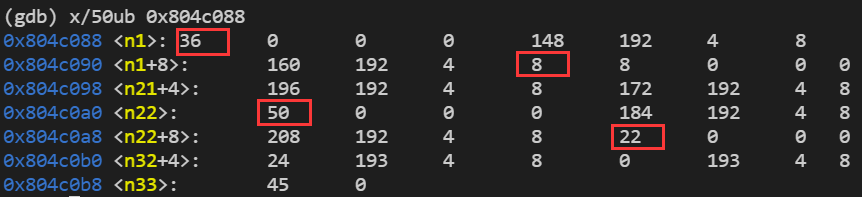


图1.7.6 0x804c088处内容

可以推测出这是一个二叉树型结构，每12个字节的内存单元中，前4个字节存储结点值，后8个字节分别存储左子树地址与右子树地址，可以得到图下图所示的二叉树（用16进制表示）：

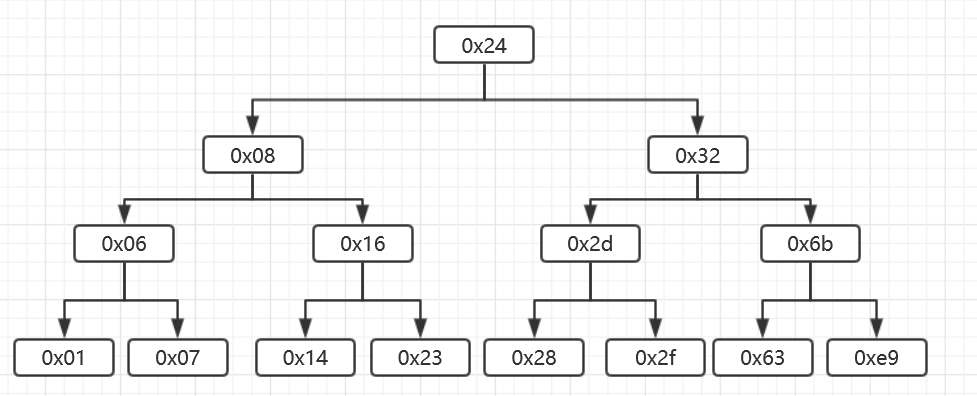


图1.7.7 二叉树结构

继续分析fun7函数，如图1.7.8所示。第666、667两行先对我们输入的这个数作一个判断，如果等于0直接跳到第688行，返回-1，这显然不是我们想要的结果。

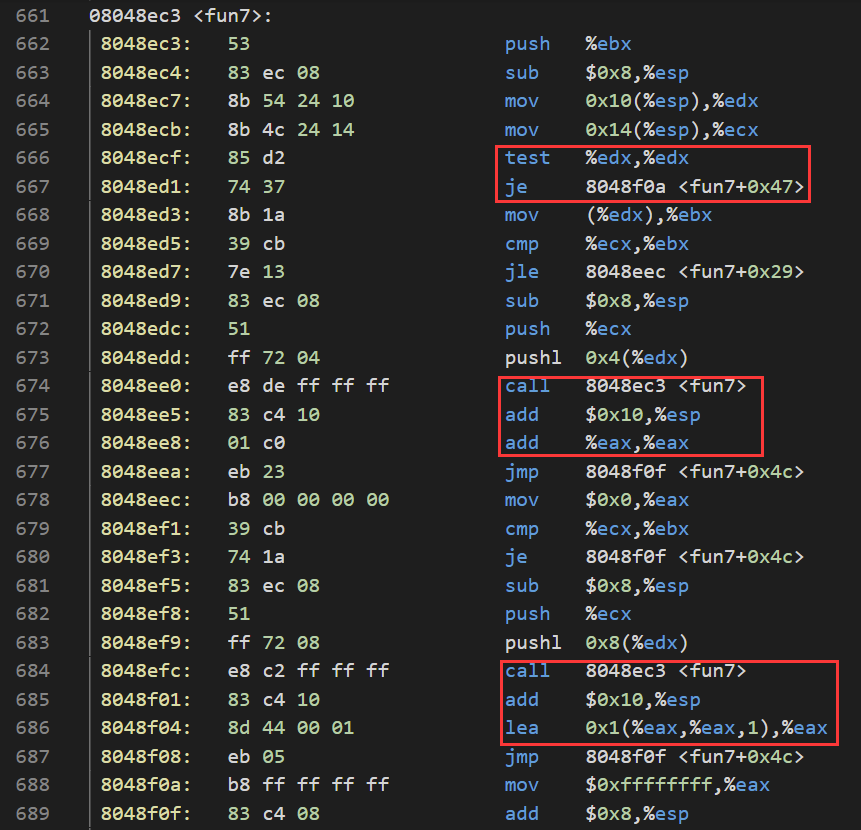


图1.7.8 fun7函数

而如果第669行的比较中树节点的值不大于我们读入的数，就会跳到第678行，将edx移到它的右子树的位置，接着调用fun7，在返回后令eax = 2 \* eax + 1。否则，代码会进行到第671行，令edx移到它的左子树的位置，接下来调用fun7在返回后令eax = 2 \* eax，下面跳至返回处。

总结上面的过程：edx指向一个树的节点，令edx节点的值与我们读入的值进行比：

1. 如果两者相等：返回0。
2. 如果前者大于后者：rdi移至左子树，返回2 \* eax。
3. 如果后者大于前者：rdi移至右子树，返回2 \* eax + 1。

而我们需要返回1，应该在最后一次调用返回0，倒数第二次调用返回2 \*eax + 1。所以，这个数应该在第2层，且比父节点大。故唯一的答案为0x32，转换成10进制为50。

4.实验结果

阶段7的测试结果如图1.7.8所示，可知阶段7能够被正确拆弹。

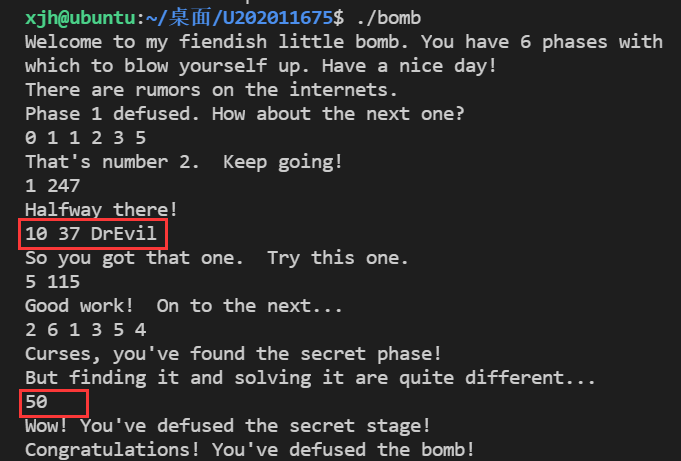


图1.7.8 阶段7测试结果

# 2 实验3：缓冲区溢出攻击

# 2.1 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。

实验分5个难度递增的等级，分别命名为Smoke、Fizz、Bang、Boom和Nitro，其中Smoke级最简单而Nitro级最困难。

实验语言：C语言。

实验环境：linux。

# 2.2 实验要求

1. 对目标程序实施缓冲区溢出攻击。

2. 通过造成缓冲区溢出来破坏目标程序的栈帧结构。

3. 使程序执行一些原来没有的行为。

# 2.3 实验内容

## 2.3.1 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：

利用 objdump 反汇编出的指令进行静态分析，利用 gdb 进行动态分析，结合对函数栈帧结构的理解，以及其建立与回收过程，通过缓冲区溢出的手段，覆写正常栈帧结构，以到达控制程序执行方向的目的。

3.实验过程：

首先，通过objdump对bomb进行反汇编并将汇编代码输出到bufbomb.asm中，如图2.1.1所示。



图2.1.1 生成bufbomb.asm

在bufbomb的反汇编源程序中查找smoke函数，记录smoke函数开始地址为0x8048c90，如图2.1.2所示。

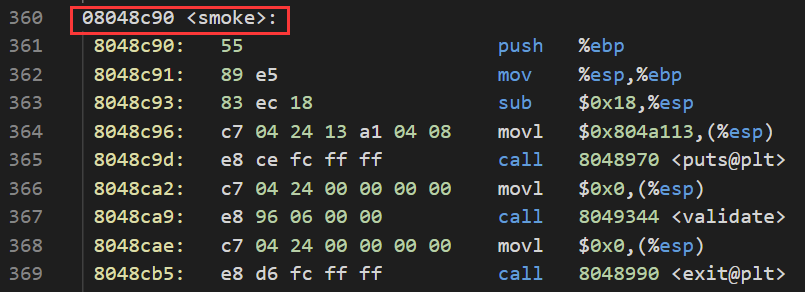


图2.1.2 smoke函数开始地址

再分析getbuf函数，可知getbuf函数栈帧大小为0x3c个字节，其中buf缓冲区总共为0x28个字节，如图2.1.3所示。

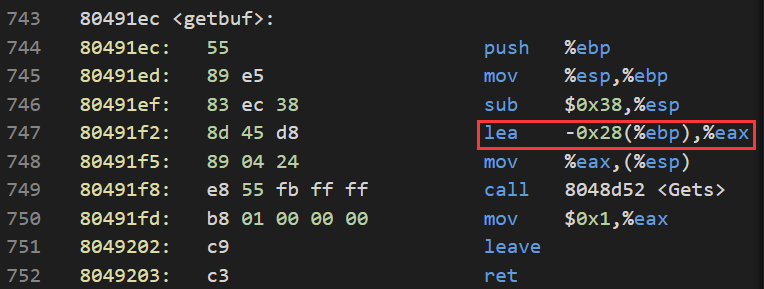


图2.1.3 getbuf函数

根据实验任务书所给出的getbuf栈帧图，可知要完成任务，应输入大小为48个字节的字符串，其中最后四个字节的内容应该填入smoke函数的首地址，使得执行完getbuf函数后转而执行smoke函数。

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）。

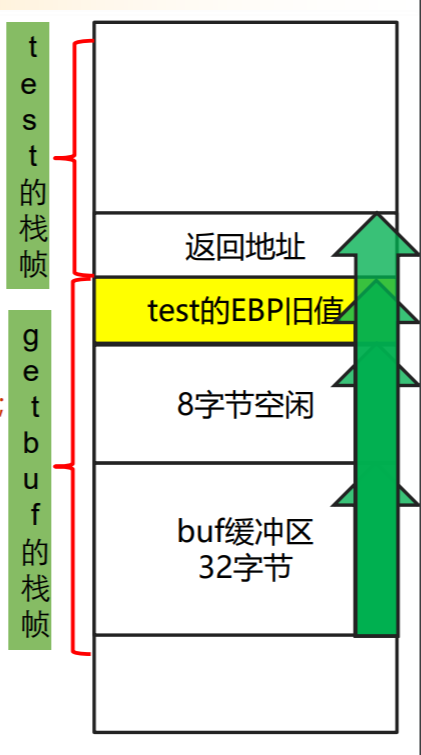


图2.1.4 getbuf栈帧结构

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。则攻击字符串应为：00 00 00… …00 00 90 8c 04 08，如图2.1.5所示。

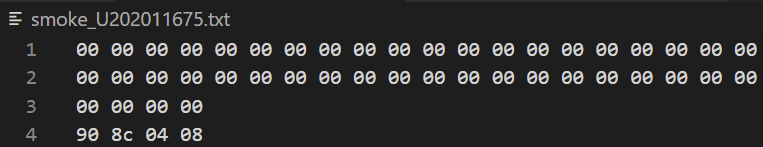


图2.1.5 结果字符串

4.实验结果：

将上述攻击字符串写入smoke\_U202011675.txt文件中，并在命令窗口中输入对应命令，实验结果如图2.1.6所示，阶段1成功通过。

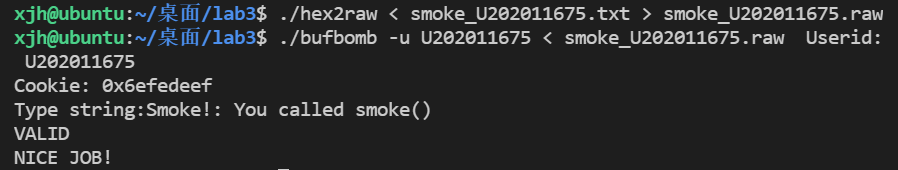


图2.1.6 阶段1测试结果

## 2.3.2 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()。与smoke阶段不同和且较难的地方在于：fizz函数需要一个输入参数，因此要将cookie值作为参数传递给fizz 函数，以便于fizz中val与cookie的比较能够成功。

2.实验设计：

利用 objdump 反汇编出的指令进行静态分析，利用 gdb 进行动态分析，结合对函数栈帧结构的理解，以及其建立与回收过程，通过缓冲区溢出的手段，覆写正常栈帧结构，以到达控制程序执行方向的目的。

3.实验过程：

本阶段和smoke类似，查看fizz的汇编代码，如图2.2.1所示。

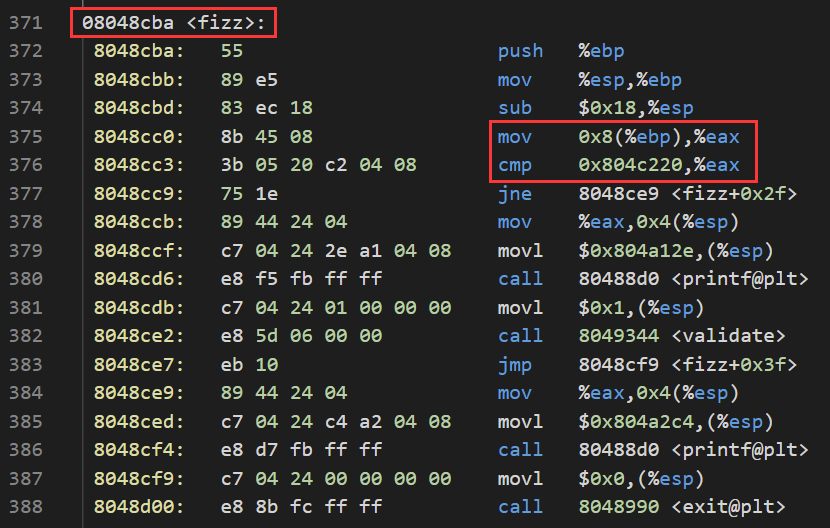


图2.2.1 fizz函数

fizz函数与smoke基本一致，ebp的上一位置ebp+4存放了调用者的返回地址，所以参数的地址应该为ebp+8的位置，我们只需要将自己的cookie放置在该位置即可。

而fizz函数开始的地址为08048cba，所以输入的前44个字节为非'\n'任意值，第45-48个字节存放fizz函数起始地址，即ba 8c 04 08，接下来4字节也是非'\n'值，最后为cookie值，即07 ab 99 67，最终输入的56个字节如图2.2.2所示。

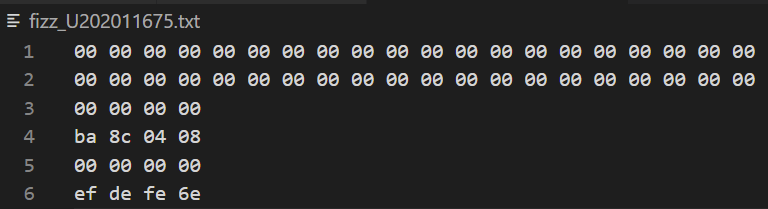


图2.2.2 结果字符串

4.实验结果

将上述攻击字符串写入fizz\_U202011675.txt文件中，并在命令窗口中输入对应命令，实验结果如图2.2.3所示，阶段2成功通过。

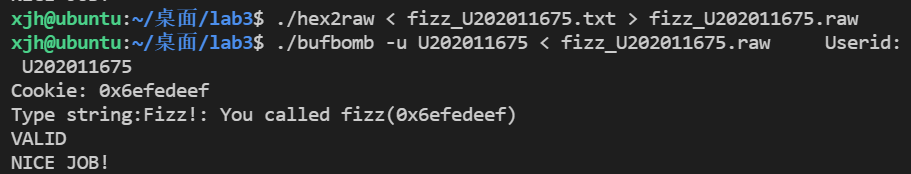


图2.2.3 阶段2测试结果

## 2.3.3 阶段3 bang

1.任务描述

本阶段的任务是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设置为你的cookie值，然后转向执行bang()。

2.实验设计：

利用 objdump 反汇编出的指令进行静态分析，利用 gdb 进行动态分析，结合对函数栈帧结构的理解，以及其建立与回收过程，通过缓冲区溢出的手段，覆写正常栈帧结构，以到达控制程序执行方向的目的。

3.实验过程：

先观察bang的代码，如图2.3.1所示。bang函数的首地址0x08048d05，在bang函数中，会将全局变量global\_value和cookie进行比较。

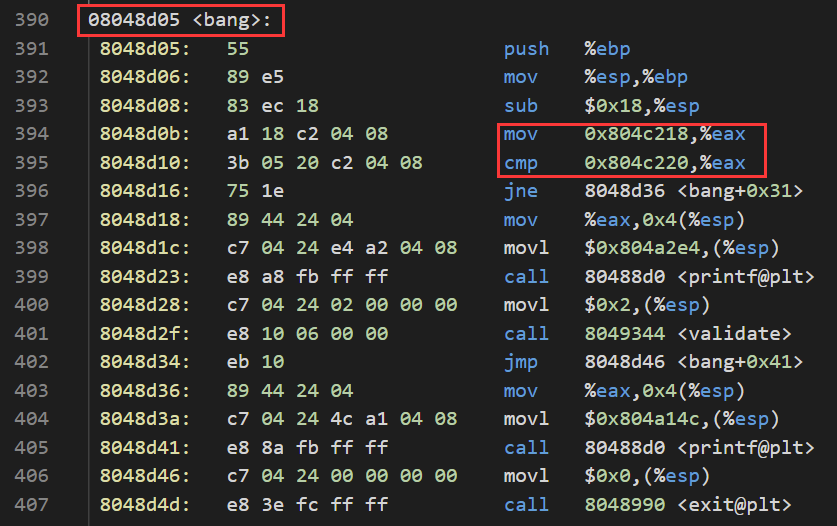


图2.3.1 bang函数

global\_value的地址是0x804c218，cookie的地址是0x804c220，global\_value在c代码中显示为0，所以需要修改global\_value的值使其与cookie一致。汇编代码为：

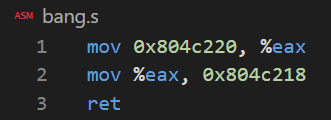


图2.3.2 bang.s代码

将bang.s进行反汇编，如图2.3.3所示。

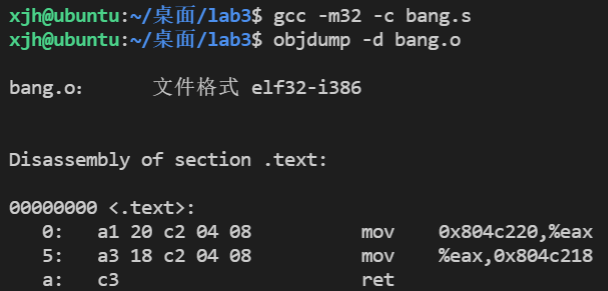


图2.3.3 攻击代码及其机器指令

得到指令序列：a1 20 c2 04 08 a3 18 c2 04 08 c3。

设置断点查看cookie为 0x6efedeef时buf的首地址，如图2.3.4所示。

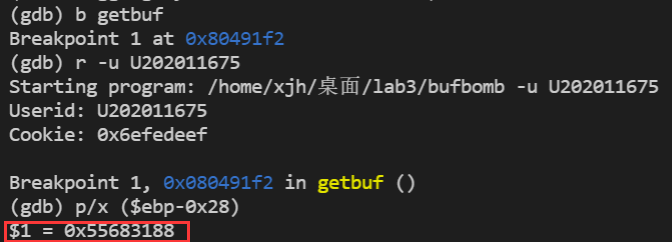


图2.3.4 查看buf首址

可知buf首址为0x55683188，综合之前的指令序列，45-48字节放buf首址，49-52放bang函数首址，得到结果字符串：



图2.3.5 结果字符串

4.实验结果

将上述攻击字符串写入bang\_U202011675.txt文件中，并在命令窗口中输入对应命令，实验结果如图2.3.6所示，阶段3成功通过。

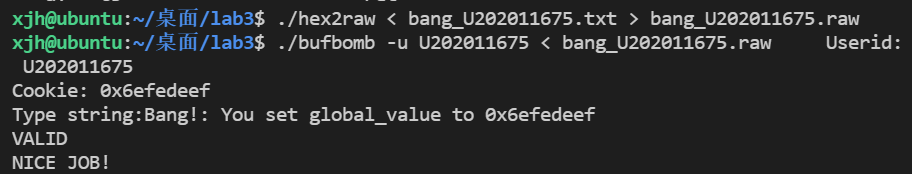


图2.3.6 阶段3测试结果

## 2.3.4 阶段4 boom

1.任务描述

本阶段的实验任务是构造一个攻击字符串，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：

利用 objdump 反汇编出的指令进行静态分析，利用 gdb 进行动态分析，结合对函数栈帧结构的理解，以及其建立与回收过程，通过缓冲区溢出的手段，覆写正常栈帧结构，以到达控制程序执行方向的目的。

3.实验过程：

首先观察函数test的汇编代码，如图2.4.1所示。

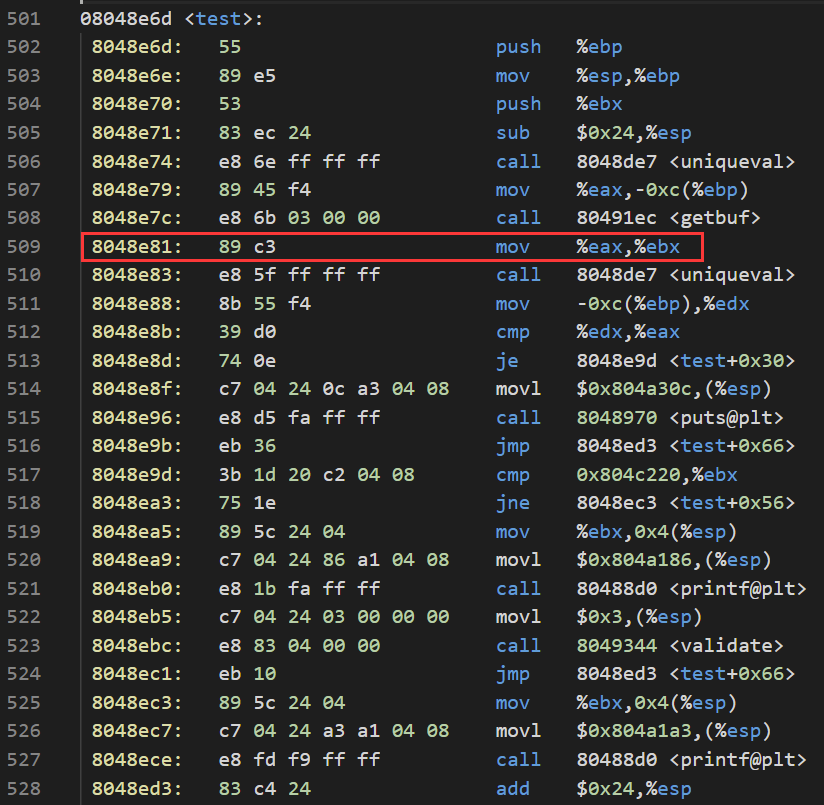


图2.4.1 test函数

要返回至test，应该返回至调用getbuf之后的一步，地址为0x08048e81，返回的值要为cookie的值，写出汇编代码：

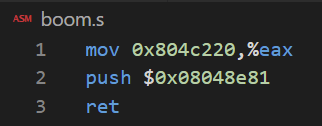


图2.4.2 boom.s代码

将boom.s进行反汇编，如图2.4.3所示。

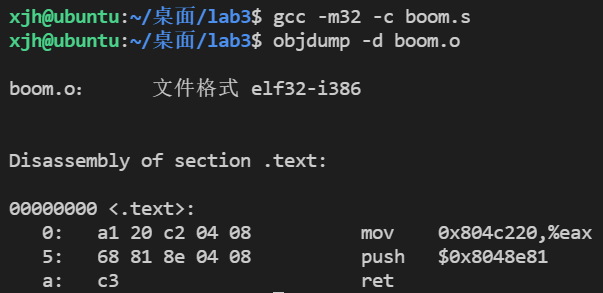


图2.4.3 攻击代码及其机器指令

得到指令序列a1 20 c2 04 08 68 81 8e 04 08 c3。

覆盖getbuf返回地址的时候会覆盖保存的寄存器ebp的值，所以通过设置断点来查看ebp的地址，如图2.4.4所示。

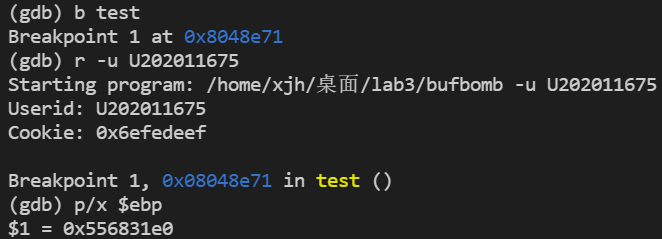


图2.4.4 查看ebp寄存器值

得到的地址为0x556831e0。

所以得到的序列指令放在前11个字节，41-44位存储ebp地址的，最后四位存储buf首址，如图2.4.5所示。



图2.4.5 结果字符串

4.实验结果

将上述攻击字符串写入boom\_U202011675.txt文件中，并在命令窗口中输入对应命令，实验结果如图2.4.6所示，阶段4成功通过。

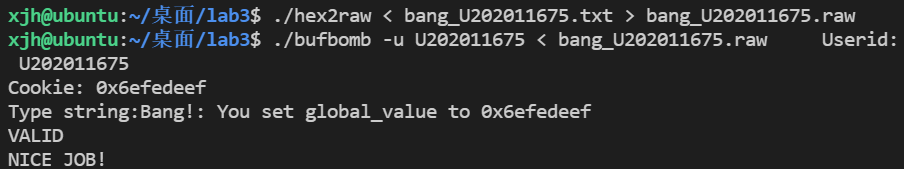


图2.4.6 阶段4测试结果

## 2.3.5 阶段5 nitro

1.任务描述

与阶段4类似，构造一攻击字符串使得getbufn函数（注，在nitro阶段，bufbomb将调用testn函数和getbufn函数，见bufbomb.c）返回cookie值至testn函数，而不是返回值1。

2.实验设计：

利用 objdump 反汇编出的指令进行静态分析，利用 gdb 进行动态分析，结合对函数栈帧结构的理解，以及其建立与回收过程，通过缓冲区溢出的手段，覆写正常栈帧结构，以到达控制程序执行方向的目的。

3.实验过程：

首先观察getbufn函数的反汇编源程序，如图2.5.1所示。

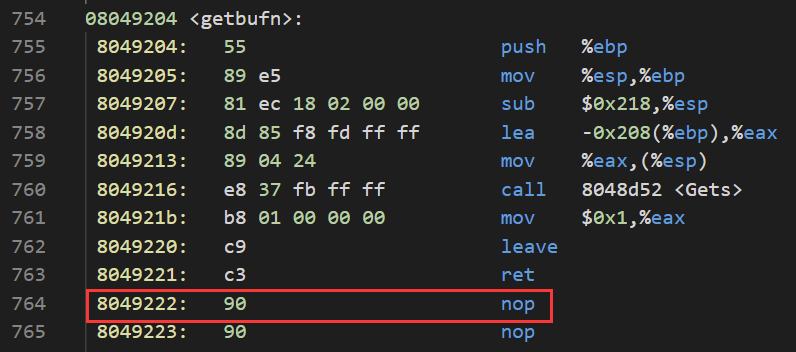


图2.5.1 getbufn函数

可知，在testn中每次调用getbufn后，栈顶指针esp的值并不会发生改变，但若输入的字符串造成了缓冲区溢出，则在执行leave指令后恢复至寄存器ebp中的值已不是原ebp值，因此需要进一步确定如何恢复该值。

继续观察testn的反汇编源程序，如图2.5.2所示。

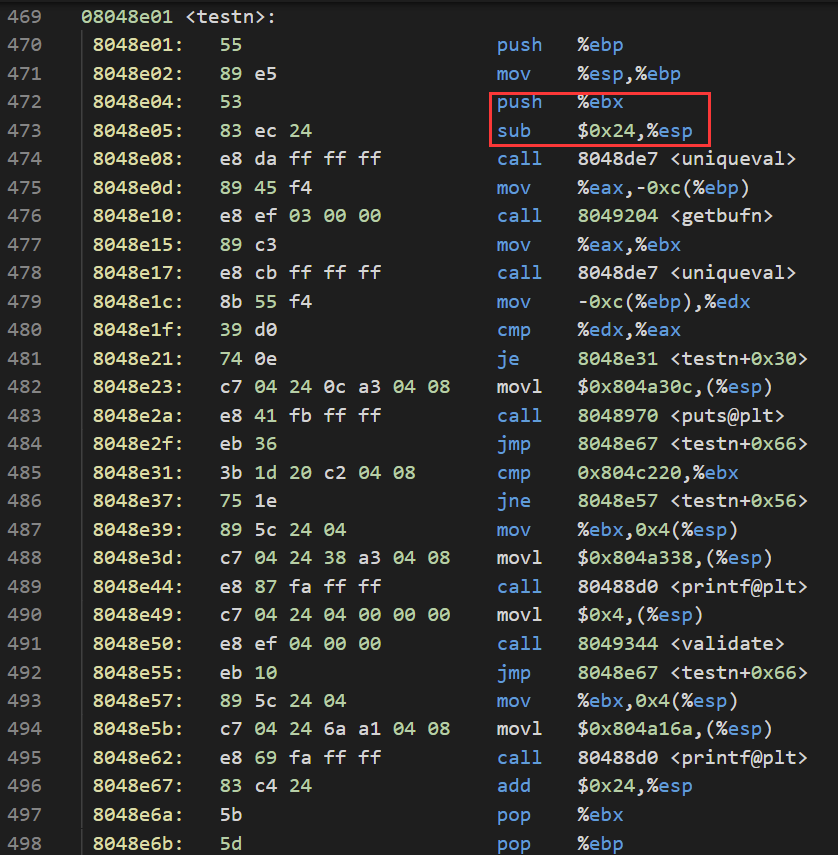


图2.5.2 testn函数

根据testn的汇编指令，可以看出ebp = esp + 0x28，结合前面的分析，可知可以将esp的值加上0x28即为原ebp值。同时，攻击字符串中所包含的返回地址应为0x8048e15。由此，编写相应攻击代码，如图2.5.3所示。

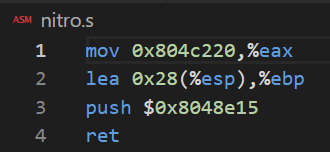


图2.5.3 nitro.s代码

将boom.s进行反汇编，如图2.5.4所示。

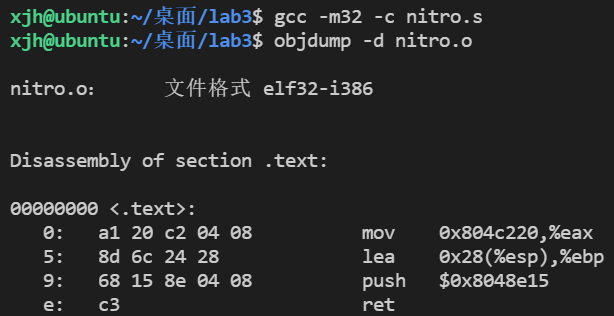


图2.5.4 攻击代码及其及其指令

由于五次调用getbufn时，写入字符串的首地址是变化的，无法通过gdb调试确定唯一值，因此可以选择让程序跳转到一个大致的位置，并且使用nop指令（其机器码为0x90）填充（CPU在执行nop指令时除程序计数器自加不进行任何其他操作），这样便可以使程序从该大致位置开始执行nop指令，直至执行到我们预先写好的攻击代码。同时，应将代码放在高位地址处，保证无论缓冲区的首地址为何，都能确保程序所跳转至的地址低于攻击代码存放地址。下一步，通过gdb调试工具查看五次循环的ebp值如图2.5.5所示。

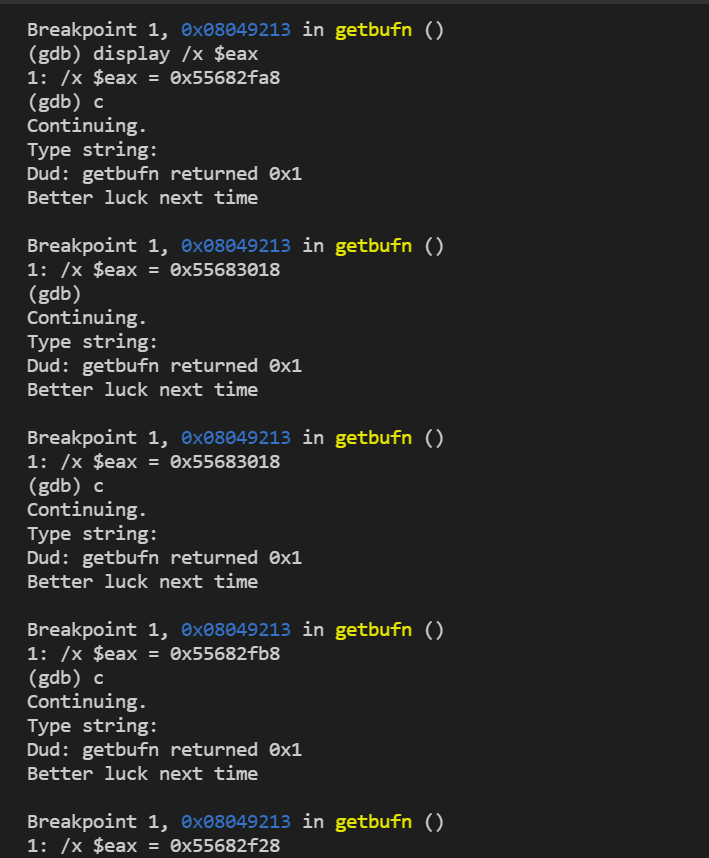


图2.5.5 5次循环的ebp值

由此可以确定缓冲区的首地址的最大值为0x55683018。并且由分析可知，返回地址应取buf首地址的最大值。因为字符串的存放位置是随机的，若返回地址不取buf首地址的最大值，而在某一次循环中攻击字符串存放的首地址为最大值，则程序便会执行从返回地址到攻击字符串存放的起始位置这一段空间中的机器指令，造成错误。

由于跳转之后的地址仍然低于汇编攻击指令所在处，虽然每次攻击必然能够执行攻击指令，但会执行额外不必要的坏指令，所以需要用 nop 指令将这段空隙填充掉。

同时，根据前面的推导不难得出，攻击字符串的长度应为0x208+0x4+0x4=528个字节，先填充nop指令，后填充攻击代码指令，最后填充四个字节的返回地址，如图2.5.6所示。

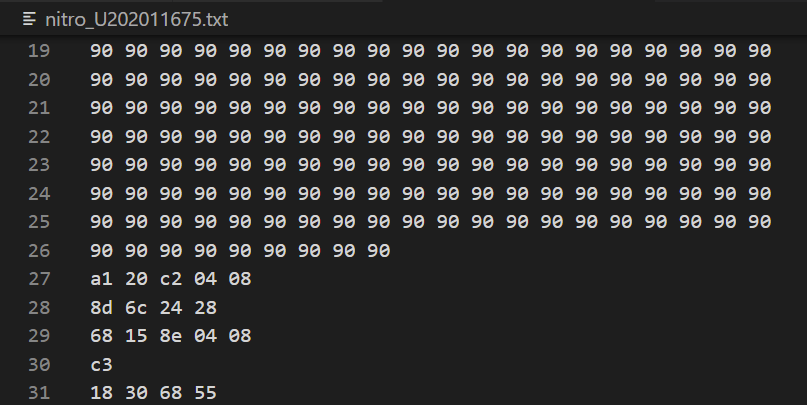


图2.5.6 结果字符串

4.实验结果

将上述攻击字符串写入nitro\_U202011675.txt文件中，并在命令窗口中输入对应命令，实验结果如图2.5.7所示，阶段5成功通过。

需要注意的是因为在Nitro模式下主程序需要读五次input以满足执行五次的需要，因此在执行./hex2raw程序时需要添加 -n flag以保证input string 被复制五次，每次以\n结尾来结束每次的gets()函数调用。

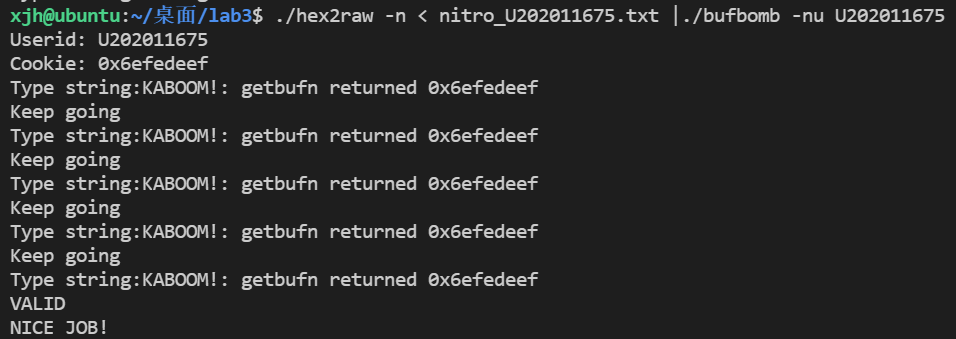


图2.5.7 阶段5测试结果

# 3 实验总结

# 3.1 实验2总结

本次实验主要的收获就是对于Linux下C语言与汇编语言的转化以及各种函数调用时的汇编操作是怎么样的。同时，由于这一次实验的汇编是AT&T语法的，和之前的Intel的语法还是有一些不同的，所以对汇编本身也有了更加深刻的理解。

在这次实验中还学习了gdb使用的各个方面，并且联想到了现代的编译器所提供的各种debug功能，比如说gdb中的display命令，就和IDE中常用的实时查看变量或表达式的值功能类似。

对于本次实验中的7个炸弹，每一个炸弹都考察了不同方面的内容，也加深了自己对于循环，分支，递归，指针以及栈等多方面内容的理解。在炸弹拆除过程中，第4阶段关于递归调用的炸弹花费了很多时间，需要对于参数传递以及数值运算的步骤都有一个清晰的理解认识，才能正确的还原递归过程。而对于第6个炸弹，初看汇编指令觉得十分复杂，存在多个循环，但在逐步的理解代码之后，发现可以将其划分成多个功能模块，得到最终的功能，成功地破解该炸弹。

总之，这7个炸弹的拆解过程不仅加深了对于课堂所学内容的巩固，也为后续更加系统深入的学习组成原理、操作系统等相关课程知识打下了基础。

# 3.2 实验3小结

这一次实验的主题是缓冲区溢出攻击，主要思想就是利用缓冲区溢出，将程序的运行流向转到原先设计好的函数或者自己新编写的函数之中。通过这次实验，我有了一下几点收获：

1. 通过深入理解缓冲区溢出buffer overflow的情形，体会到了体会几种可能危害到系统安全的场景
2. 加深了对IA-32函数调用规则和栈结构的具体理解，巩固了汇编指令集。
3. 巩固了对objdump、gcc等工具的掌握，可以熟练的将汇编代码用gcc编译成机器指令，再用objdump反汇编成二进制字节数据和汇编代码，来构造具有攻击代码的攻击字符串。
4. 学会了用gdb调试，能够熟练地用gdb指令设置断点来让程序暂停、观察断点处必要的内存单元内容、寄存器内容等。

# 参考文献

[1]袁春风. 计算机系统基础. 机械工业出版社，2014

[2]袁春风. 计算机系统基础. 机械工业出版社，2018

[3]深入理解计算机系统. 机械工业出版社，2016