

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**专业班级： CS1806**

**学 号： U201814788**

**姓 名： 刘美**

**指导教师： 刘海坤**

**报告日期： 2020年 06 月 11 日**

**计算机科学与技术学院**

**目录**

[实验2：Binary Bombs 3](#_Toc449875435)

[实验3：缓冲区溢出攻击 19](#_Toc449875436)

实验总结...............................................................................................36

# 

实验2： Binary Bombs

2.1 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。

2.2 实验内容

每个炸弹考察的内容如下：

\* 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

\* 阶段7：隐藏关卡

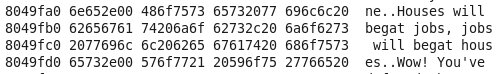
2.2.1 阶段1 phase\_1字符串比较

1.任务描述：输入字符串拆除炸弹

2.实验设计：观察汇编代码，结合数据段的内容实现拆弹

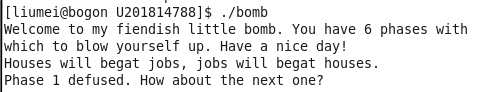
3.实验过程：

首先我们根据老师给的实验内容再结合phase\_1函数中的语句，确定这是一个比较字符串的函数，继续看这个函数发现在call strings\_not\_equal之前有将地址为0x8049fa4的内容压栈，猜测这一步应该就是为了调用字符串比较函数之前准备要传入的参数，所以查看0x8049fa4的内容得到语句：“House will begat jobs, jobs will begat houses.”，即为要输入的正确结果。



**图2-1 找到应该输入的语句**

1. 实验结果：



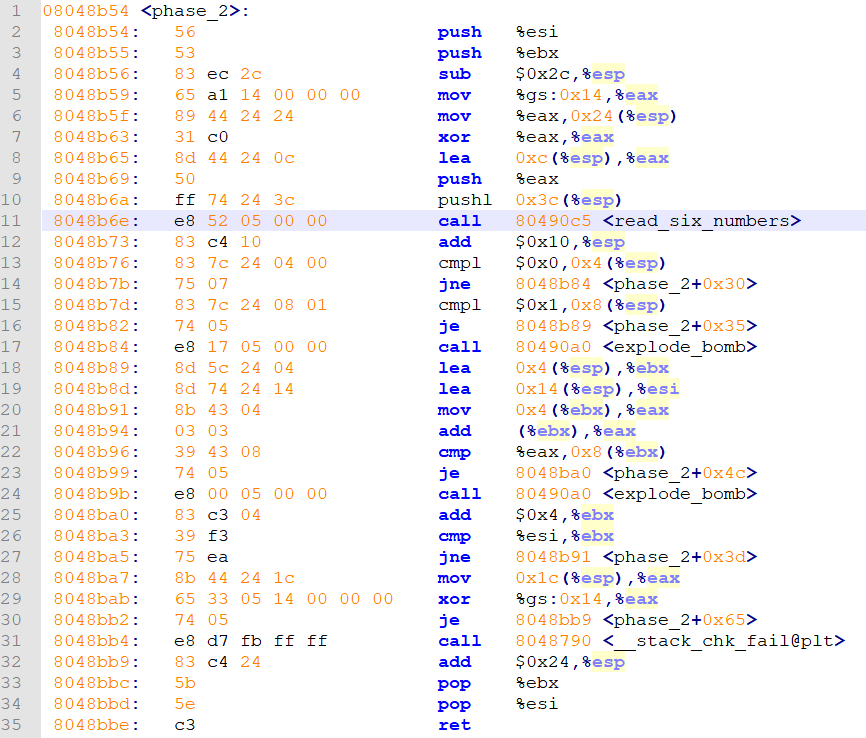
**图2-2 输入正确字符**

2.2.2 阶段2 phase\_2 循环

1.任务描述：输入字符串拆除炸弹

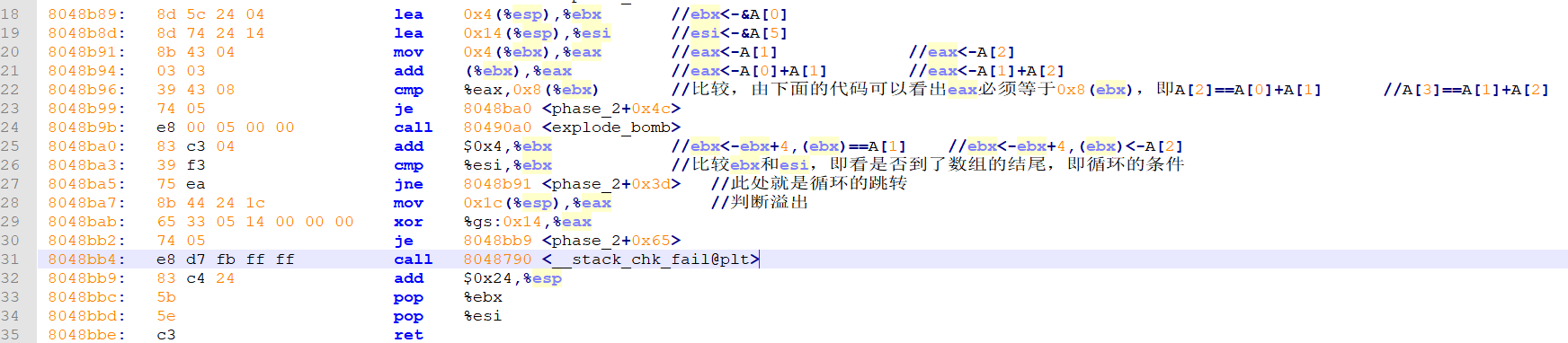
2.实验设计：通过观察反汇编的代码，按照代码逐行手动分析得到大致思路

3.实验过程：



**图2-3 阶段2的反汇编代码**

1. 由 第11行 call 80490c5<read\_six\_numbers> 语句我们得到需要输入六个字符
2. 查看整个函数，推测输入的六个字符应该是当做一个数字型数组进行相关的运算，不妨记这个数组名为A，按照实验的思路，逐步分析。
3. 我们不难得到数组的起始地址是0x4(eap),由第13、14、17行我们得知数组的第一个元素A[0]必须为0，由第15、16、17行我们得知A[1]的值必须为1。



**图2-4 按步骤分析的结果**

1. 由图1-4的分析结合循环语句的特点，我们知道输入的六个元素的数组是一个斐波那契数列，且起始的两个元素为0和1

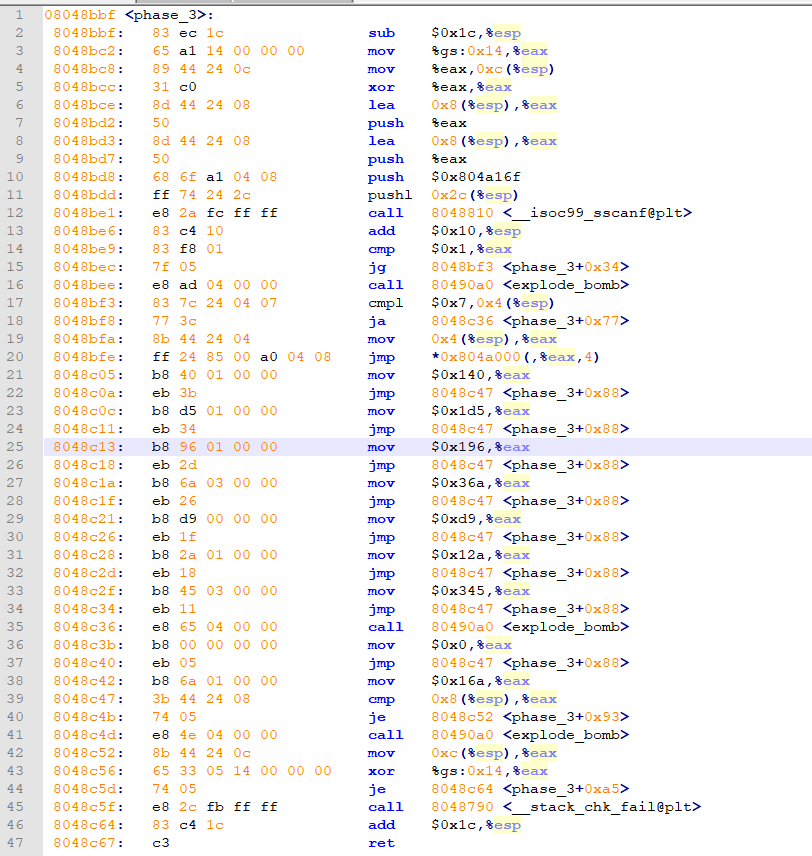
4. 实验结果：由上面的分析我们可以知道结果应该是0 1 1 2 3 5，输入判断



**图 2-5 输入结果正确**

2.2.3阶段三 phase\_3 条件/分支switch语句

1. 任务描述：输入字符实现拆弹
2. 实验设计：通过对switch的了解和对实验反汇编代码的分析得到结果
3. 实验过程：

****

**图2-6 实验反汇编代码**

1. 首先我们知道这是一个switch方面的问题，那么结合所学知识，我们的关键应该是要找到跳转表。我们先看代码
2. 由第12行调用了sscanf函数我们知道这是需要我们输入字符，因为sscanf函数的返回结果是成功赋值的变量的个数，所以结合14、15、16、17、18、35行我们知道输入的字符的个数应该限制在1-7之间。由第19行和函数调用时的栈帧我们得知输入的第一个为0x4(esp)即eax，再由第21到38行中的各种jmp语句，推测输入的第一个参数应该为对应switch的选择项。所以我们查看一下第20行的\*0x804a000(,eax,4)的内容。

****

**图2-7 数据段的内容**

1. 由上面的的图片我们可以发现，在0x804a000开始的32字节的数据存放的是反汇编中的机器指令的地址，所以上面的推测是正确的。不妨设输入的第一个参数为0，那么由图1-8 我们得知应该跳转到0x80048c42即第38行，执行完这一行之后，eax被赋值为0x16a=362。
2. 在由第39、40、41行我们可以退出输入的第二个参数必须和此时eax的值即362相等。所以我们推测输入为两个数：0 362
3. 实验结果：

****

**图2-8 实验结果正确**

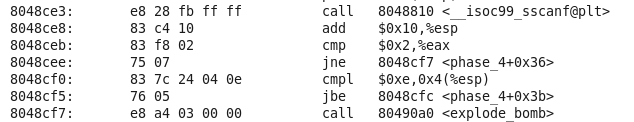
通过上面的分析我们可以得知此题的答案是不唯一的，其他答案的获得和本答案的方法一致，不做赘述。

2.2.4 阶段四phase\_4递归调用和栈

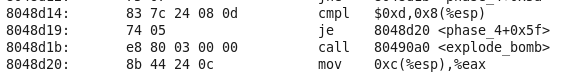
1. 任务描述：输入字符实现拆弹

2. 实验设计：可以通过观察反汇编代码结合递归倒推得到结果实现的功能，也可以直接将func4函数改写为c函数得到结果。

3. 实验过程：

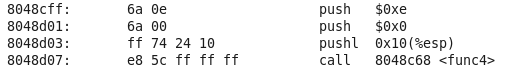


**图2-9**

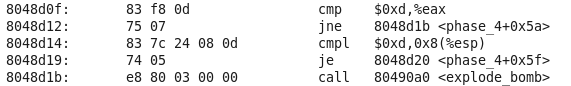
****

**图2-10**

1. 由图1-10 我们可以得到需要输入两个字符（两个数字字符），且第一个字符必须小于等于13，由图1-11我们可知输入的第二个参数必须是13。

****

**图2-11**

****

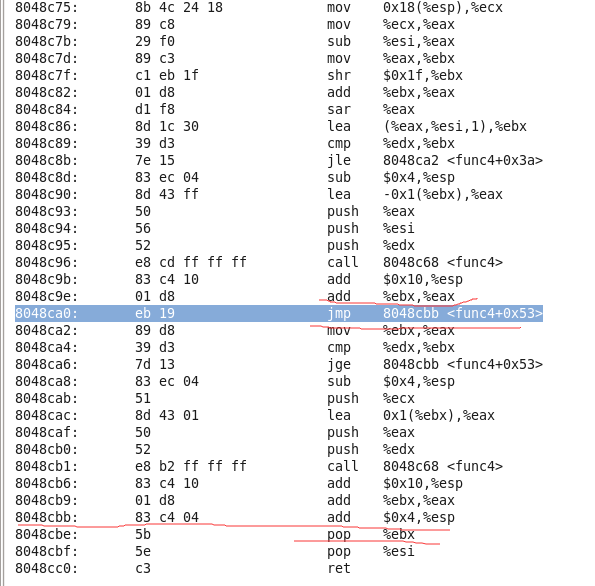
**图2-12**

1. 由图1-12 我们可知将主函数将0xe，0x0和输入的第一个参数传入了func4函数并调用。再由图1-13 我们可知调用完func4函数之后得到的返回结果必须为0xd。下面我们来看递归函数。



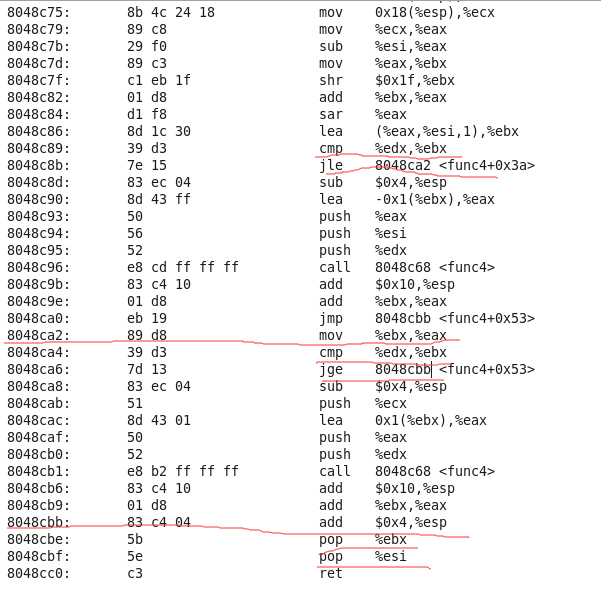
**图2-13**

1. 图1-14中edx，esi，ecx分别对应的输入的第一个数、0x0、0xe。



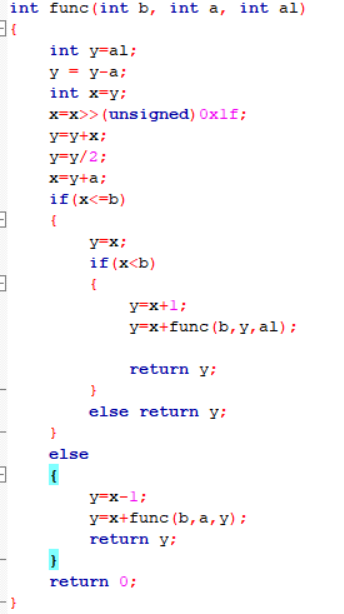
**图2-14**

1. 由图1-15 我们可以得知返回结果应该是ebx+eax

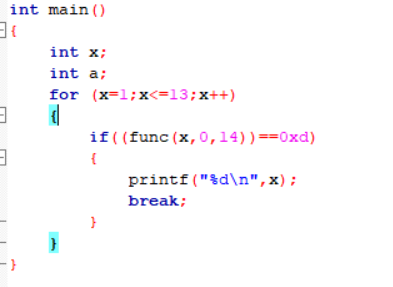


**图2-15**

1. 由图1-16 我们分析得到最后一次进入func4函数的执行状况如划线部分表示，即最终ebx==edx,且返回结果为eax=0xd==ebx。然后采用倒推的思想可以得到结果为2。
2. 更简单的是，我们可以将func4函数转换为c函数，并在主函数main函数中对输入的第一个函数进行试探。这里我采用的就是c函数的方法。

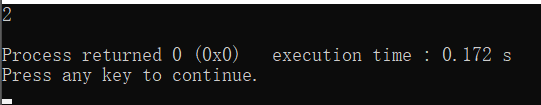
****

**图2-16 func4反汇编代码改写为c函数**

****

**图2-17 编写一个main函数得到输入第一个参数的值**

4.实验结果：



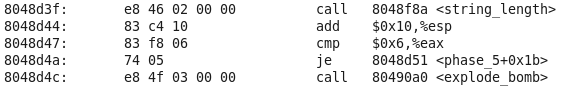
**图2-18 main函数得到的结果**



**图2-19 测试得到的结果正确**

2.2.5阶段五 phase\_5指针

1. 任务描述：输入字符实现拆弹
2. 实验设计：通过观察反汇编结合gdb单步调试得到正确的结果
3. 实验过程

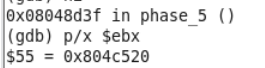


**图2-20**

1. 由图1-21 我们可以知道这里需要用户输入长度为6的字符串

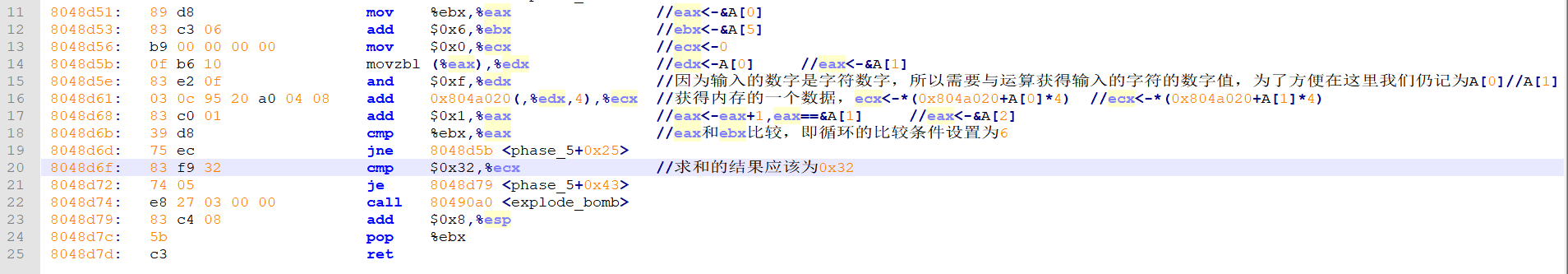


**图2-21**



**图2-22**

1. 由图1-22和图1-23我们可以知道ebx存放的是输入的字符串数组的首地址，我们不妨记数组的名称为A。
2. 通反汇编的代码我们可以分析得到，输入的字符串数组的元素的值应该作为0x804a020为起始地址存放的一堆数据（视为另一个数组）的下标值，函数实现的功能是对输入下标对应的数据求和，如果求和结果为0x32则不会爆炸。下面图1-24列出了两次分析的结果



**图2-23**

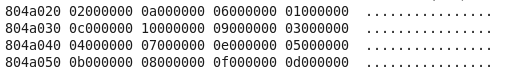


图2-24

1. 由上面的分析我们知道需要对图1-25的数据挑出六个求和为50，答案不唯一，这里我们选择下标为4 4 4 4 3 3的数据
2. 实验结果：



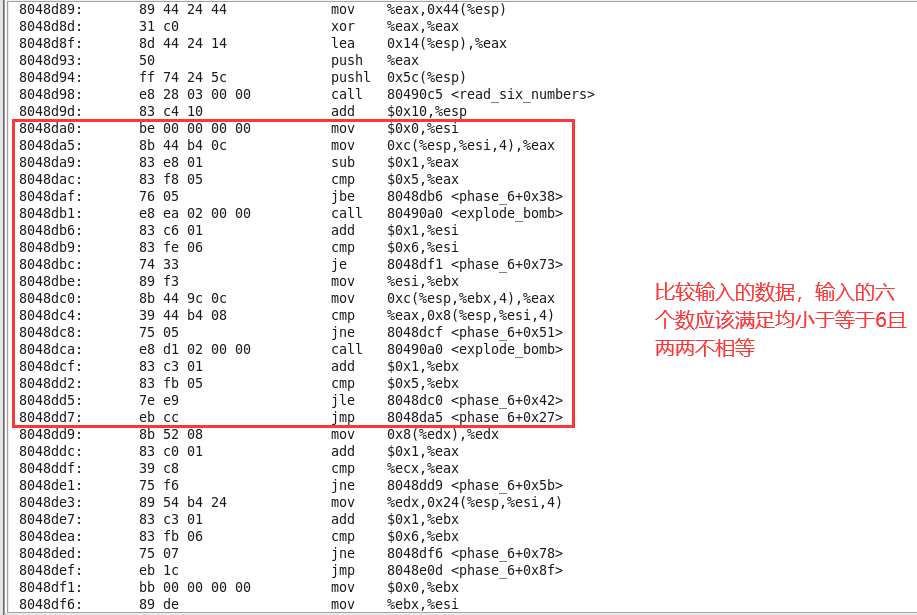
**图2-25 输入的答案正确**

**2.2.6 阶段六 phase\_6**

1. 任务描述：输入正确的字符实现拆弹

2. 实验设计：通过观察反汇编代码结合gdb单步调试得到正确的结果

3. 实验过程：



**图2-26**

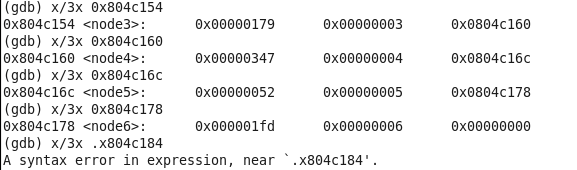
1. 由图1-27 我们确定输入的数字应改为1、2、3、4、5、6的一个排列组合。因为实验报告给出这个函数涉及到链表、结构，所以我们猜测这个结构中具有我们输入的值的变量以及只想下一个结构体变量的指针（地址值），所以我们希望可以找到这6个结构体变量并查看其中的内容。
2. 观察代码的时候我发现有这样一条特殊的语句，于是利用gdb查看这个地址的内容



**图2-27**

发现这就是结构变量的所在位置，而且这是对应的第一个结构体变量node1，结构体变量中存有三个变量，第一个变量为未知的值，第二个变量为输入数字值，第三个变量为下一个结构体变量的地址。我们从这个结构体变量的敌营可以发现结构体的有效大小为12个字节，所以我们一次找出剩余的五个结构体。





**图2-28**

1. 找到结构体变量之后这个函数就变得简单了一些。不妨输入字符1 2 3 4 5在通过gdb的调试过程中发现函数将输入的数字一次赋值给生成的六个节点的第二个变量，而输入的数字应该按照节点的第一个变量的值从大到小对应的编号输入。即得到结果为2 4 6 3 1 5。

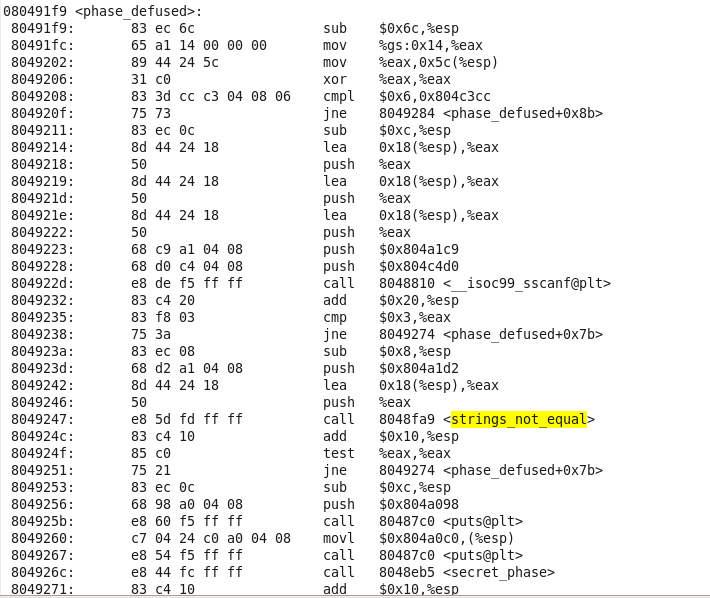
2.2.7 阶段7 隐藏关卡secret\_phase

1. 任务描述：找到打开隐藏关卡的条件并解答隐藏关卡

2. 实验设计：通过观察反汇编代码结合gdb进行单步调试理解代码解决炸弹

3. 实验过程：

（1） 由老师给出的任务书我们知道需要在第四关的后面加上一段特定字符才能找到隐藏关卡，我推测既然需要加入特定的字符，那么一定设计到对输入字符的比较过程，所以我们在asm.txt中查找strings\_not\_equal函数分别在哪些地方被调用了，在去排除和判定。

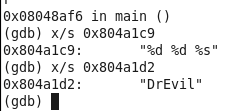


**图2-29 找到一个疑似和隐藏关卡有关的函数**

1. 在图1-30中显示找到了一个和隐藏关卡有关的函数，因为这里不仅调用了strings\_not\_equal函数，在这之前还看到sscanf函数，并且在sscanf之前还有push 0x804a1c9，push 0x804c4d0的敏感语句（第二个push后面的地址很大，在数据段中查不到，所以推测他就是我们输入的字符串的地址，而第一个push后面的地址可以在数据段中显示），这些语句是为调用sscanf函数提供参数使用的，所以我们查看0x804a1c9的内容。

MLOYQ{BIIG)RYSZYY5@1[WO

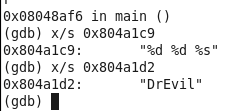
**图2-30**



**图2-31**

由图1-31 我们发现0x804a1c9的内容对应的是%d %d %s，所以可以确定推断是正确的，这就是在阶段四输入两个数字之后再输入一串字符。

1. 接着看代码，发现在strings\_not\_equal的前面有语句push 0x804ad12，所以推测这是传入比较字符串函数的参数，而且结合上面的分析可以知道应该是输入的字符串和设定的字符串比较，查看该位置的内容。



**图2-32**

由图1-33 我们可以推断触发隐藏关卡输入的字符应该就是DrEvil。

1. 完成出发隐藏关卡之后需要解答这个关卡。我们查看函数secret\_phase和func7。发现隐藏关卡也和第四关一样是一个递归过程。



**图2-33**

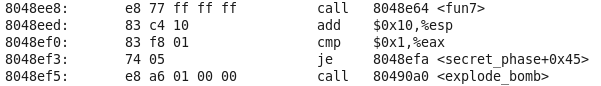
由图1-34结合gdb单步调试得到ebx的值我们可以得到第一次调用func7函数调用的参数是输入的值和0x804c088。又是push+地址的表达形式，所以我们查看这个地址存放的内容



**图2-34**

由图1-35 我们发现该地址存放的是0x24。

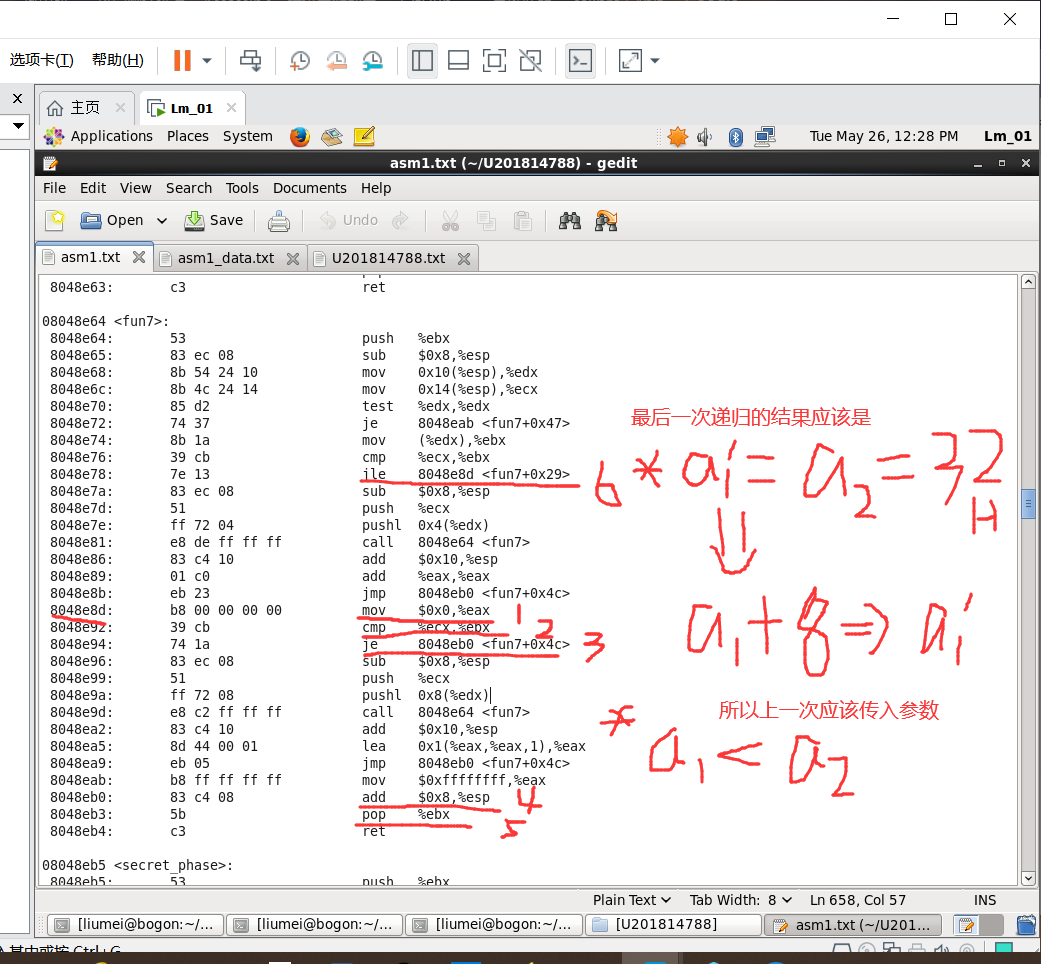
接着往下看这个递归函数的返回结果应该满足什么条件。



**图2-35**

由图1-36我们可以得知调用这个递归函数最终返回的结果应该是1。

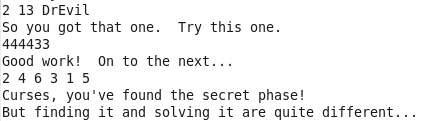
接着我们回到func7函数进行倒推分析。



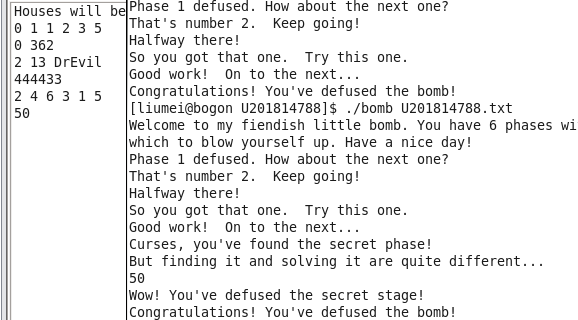
**图2-36**

图1-37中第一次edx=\*a1（是一个地址值），ebx=a2（输入的值），其表达的意思是，在递归过程中最后一次调用func7函数的步骤应该是6->1->2->3->4->5，这样才能满足最终eax的值为1(此时满足位置6的ebx<=ecx，和位置2的ebx==ecx)，从而倒推到上一次执行func7函数的时候也是在位置6的地方有ebx<=ecx，但是不满足位置2的ebx==ecx所以edx+8（a1’+8）和ecx（a2）一起压栈调用最后一次递归过程。因为我们不知道这个递归到底执行了几次，我们首先尝试最后一次递归是我的第二次调用func7函数（包括secret\_phase中调用的一次），此时最后一次递归时传入的参数ebx应该是第一次调用时传入的0x804c088+8=0x804c090，而因为最后一次递归是满足ecx==（ebx）而因为在整个递归过程中func7函数的一个参数ecx的值（即我们输入的值）是不变的，查看到0x804c090位置的值位0x32，所以推测输入的值即为0x32，不满足的话在进行一次倒推，即最一次递归应该是第三次调用func7函数。

4.实验结果：



**图2-37 输入的隐藏字符正确**



**图2-38 隐藏关卡通过**

所以其实隐藏关卡只需要倒推一次就好了。

2.3 实验小结

本次实验的代码全是反汇编的代码，为了实现不出发爆炸，需要我们能对反汇编语句具有较好的理解，并能够根据一些特殊的语句进行大胆的推测。通过本次实验学到的一些知识点可以做出如下的总结：

1. 注意一些特殊的语句。如果一个函数的代码太长不知道如何下手，我们的关注点可以首先放在一些call指令上，尤其是对于一些调用系统函数的call指令，可以让我们猜到这个函数需要干些什么比如（read\_six\_numbers就以为这需要输入六个字符，sccanf以及这之后的eax的比较可以看出需要输入几个字符）。除此之外，还要注意一些push+地址，mov 地址，寄存器 的指令，因为这些指令要么设计到和函数实现密切相关的字符串（设定的和输入的，一般输入的字符串的地址都很大）要么是作为之后的call指令的参数，不管哪一种注意这些代码并查看其中的内容都是很重要的。
2. 这次实验设计到的一个难点就是递归过程，递归过程就本次实验而言我用到了两种方法，一种是将递归函数转化为c函数，在编写一个c函数直接运行得到结果（阶段四），第二种就是倒推的方法（隐藏关卡），将递归的次数从1开始逐渐增大直至结果满足实验要求。总的来说，倒推的方法比较麻烦，一不留神就很容易搞混，但是如果代码比较繁琐但是递归次数少的话也挺实用。转化为c函数的方法的话对一些指令比较清楚（比如像阶段四的基本就是一些算术运算）的函数比较简单，如果指令稍微复杂一点转c语言复杂一点的话也比较难实现了。所以还是具体问题具体分析的较好。
3. 通过本次试验，我发现要想解决这类问题，还是要多加练习，想要快速解决这些问题需要积累大量的联系，锻炼对代码的敏感度，知道从哪方面下手才会切中要点。

**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。

实验语言：c；实验环境

3.2 实验内容

本实验需要你构造一些攻击字符串，对目标可执行程序 BUFBOMB 分别造成不同的缓冲

区溢出攻击。实验分 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）。

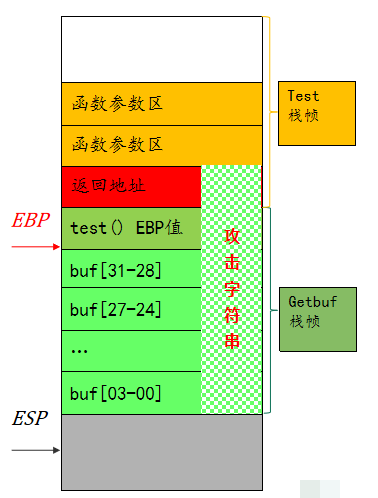
3.2.1 阶段1 Smoke

1.任务描述：构造一个攻击字符串bufbomb的输入，在getbuf中造成缓冲区的溢出是的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：通过对调用函数之间的栈帧的关系确定需要覆盖缓冲区的大小和具体的攻击字符串

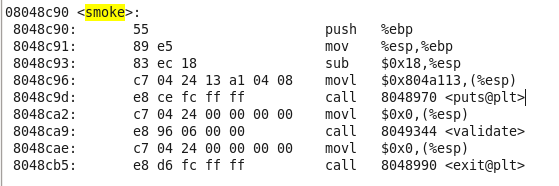
3.实验过程：

（1）



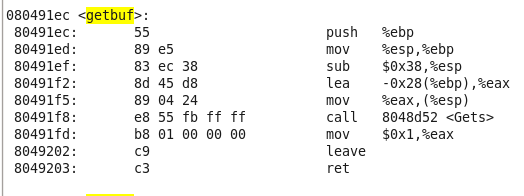
**图3-1**

1. 由上图3-1 test函数和getbuf函数的栈帧示意图我们可以知道，为了使getbuf执行完毕返回的时候不是返回到test函数而是smoke函数我们需要将原本的返回地址替换成smoke函数的起始地址0x8048c90，除此以外的缓冲区的值可以用0x0代替。



**图3-2**

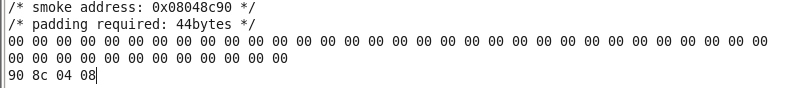
1. 确定好要输入的值之后确定攻击字符串的大小（缓冲区的大小）。



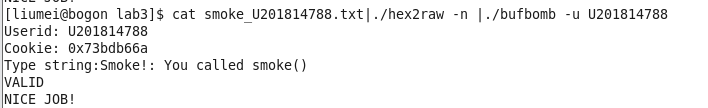
**图3-3**

如以上实例，你可以看到 getbuf 的栈帧是 0x38+4 个字节，而 buf 缓冲区的大小是 0x28 （40 个字节）。 再加上ebp的旧值和返回地址，攻击字符串的大小应该为40+4+4=48字节，且最后四个字节就是smoke的起始地址。

1. 实验结果：



**图3-4 输入的攻击字符串**



**图3-5**

由图3-5可知上述分析正确。

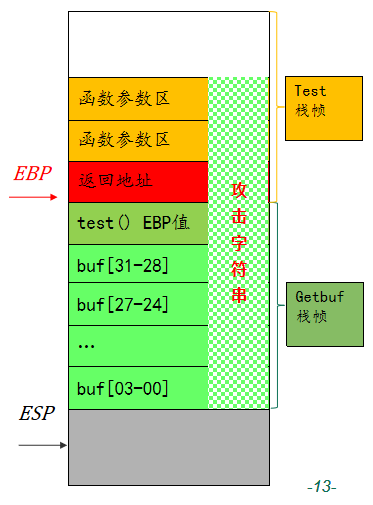
3.2.2 阶段Fizz

1.任务描述：输入一个参数，是的用makecookie得到的cookie值作为参数传递给fizz函数。

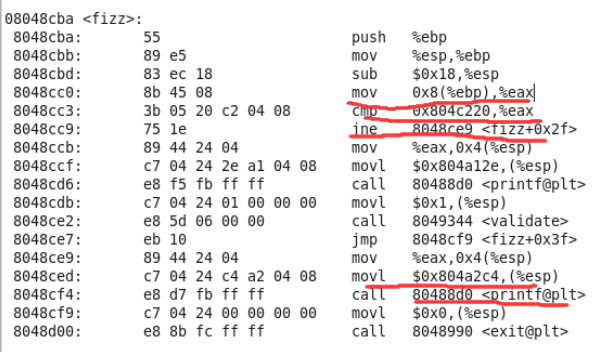
2.实验设计：结合栈帧示意图和对fizz函数的分析得到cookie放置在栈中的位置并得到攻击字符串。

3.实验过程：

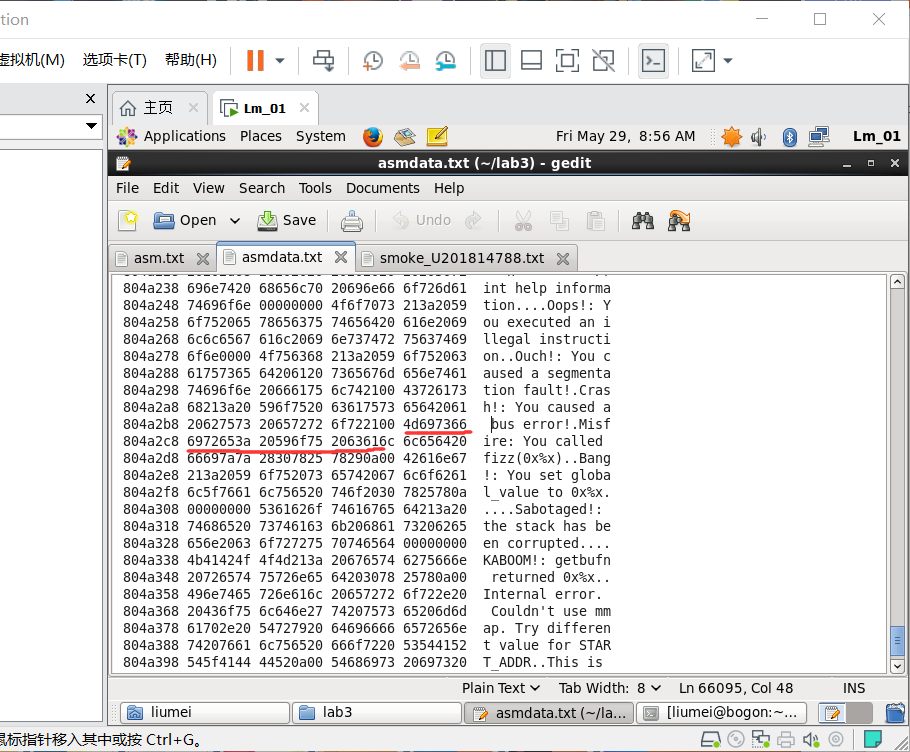
（1）



**图3-6 栈帧示意图**



**图3-7**



**图3-8**

和阶段一相同，攻击字符串的最后4个字节填入的是fizz函数的起始地址去覆盖原本的返回地址，结合图3-6我们可以知道，当getbuf函数执行完毕，fizz函数尚未执行的时候，栈顶指针位于图3-6返回地址上一格的位置处，再接着fizz函数将ebp入栈，此时在fizz栈帧中esp指向的是图3-6的返回地址处。结合fizz c函数中的

if (val == cookie)

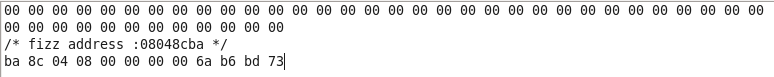
printf("Fizz!: You called fizz(0x%x)\n", val);

以及由图3-8可知划线部分的地址为0x804a2c4显示的内容是val==cookie时的输出语句，我们可以推测划线部分中比较语句中的eax，即0x8(%esp)的值即为cookie的值。这个地址我们由上面的分析可以得知正好为与栈顶指针往上两个的地方，所以需要输入的攻击字符串的大小为48+8=52，且最后一个字节的内容即为cookie。

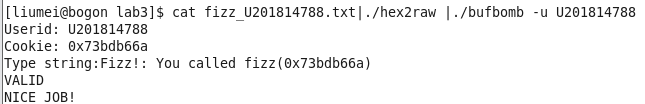
4.实验结果：

AM[X6LR15H~$G9TKEHU(GCW

**图3-9 得到cookie**



**图3-10 输入的攻击字符串**



**图3-11 测试结果**

由上图可知，上述分析正确

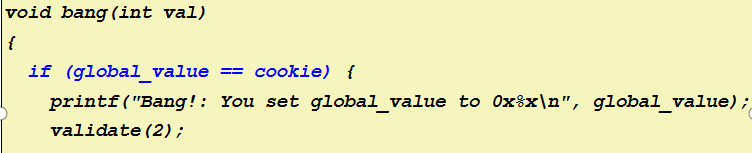
3.2.3 阶段3 Bang

1. 任务描述：构造攻击字串，是目标程序调用bang函数，要将函数中的全部变量global\_value篡改为cookie的值，是想赢判断成功需要在缓冲区中诸如恶意代码篡改全局变量的值。

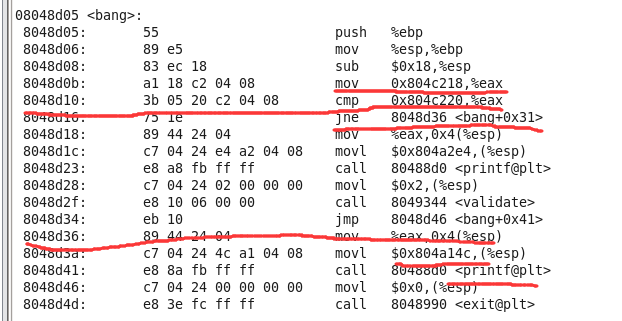
2. 实验设计：首先需要通过观看反汇编代码和c函数比较得到global的地址，然后需要编写一个汇编文件并反汇编得到篡改全局变量和返回到bang函数的攻击机器指令，然后将该机器指令入栈使得能够被执行。

3. 实验过程：

（1） 获得global\_value的地址。首先我们可以看一下bang c函数中对全局变量global\_value和cookie值进行了比较，如图3-12，那么在反汇编语句中肯定涉及到了原始的global，所以我们在看一下反汇编代码，如图3-13。



**图3-12 bang c函数**



**图3-13 bang反汇编代码**

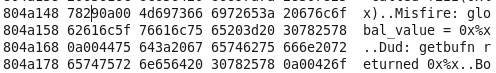
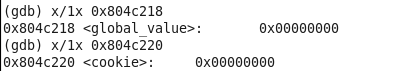


图3-14

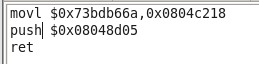
由阶段二我们知道地址0x804c220存放的数据就是我们的cookie值，而划线部分对cookie和另外一个数据进行了比较，且比较的结果如果不相等的话就会输出地址为0x804a14c的内容--Misfire：global\_value = 0x%x即输入攻击字符串错误的信息，所以我们可以推测地址为0x804c218的数据即为global\_value。



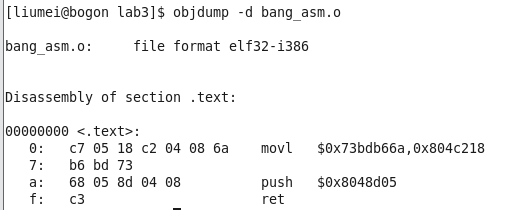
**图3-15**

由图3-15可知global的地址为0x804c218。

（2）编写汇编文件并进行反汇编。编写的汇编文件根据实验任务书我们可以知道首先将global\_value设置为我的cookie值0x73bdb66a，同时将bang函数的地址0x8048d05压入栈中，最后附一条ret语句。汇编文件如图3-16所示，反汇编得到的机器代码如图3-17所示。

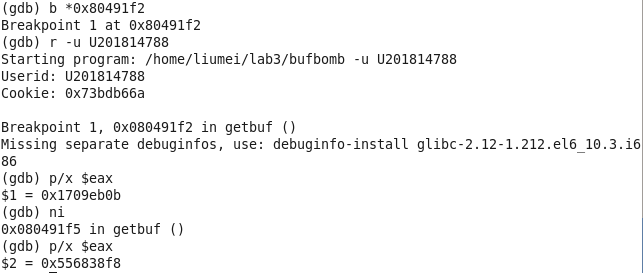


**图3-16 汇编代码**



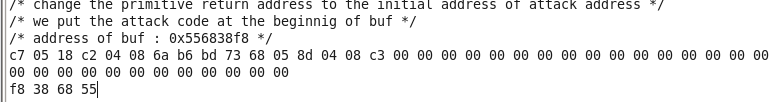
**图3-17 返回的机器指令**

1. 将得到的机器指令写入缓冲区，在这里我们将机器指令放在buf还中区的开始部分，然后将原返回地址改写为栈帧内的攻击机器指令的开始地址，即buf的首址。为了得到buf的首址，我们需要gdb调试，结果如图3-18所示。

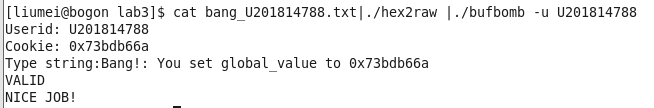


**图3-18 得到buf的首址为0x556838f8**

4. 实验结果：



**图3-19 攻击字符串**



**图3-20 测试结果**

由上图可知上述分析正确。

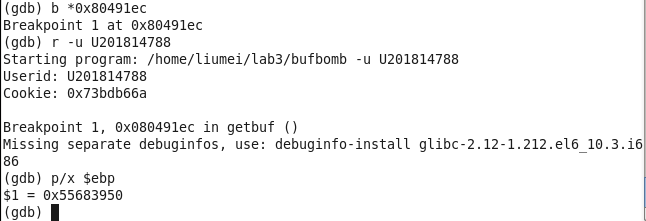
3.2.4 阶段4 Boom

1. 任务描述：构造攻击字符串，使得getbuf都能将正确的cookie值返回到test函数，而不是返回值1，除此以外还要能够还原任何被破坏的状态。

2. 实验设计：通过分析栈帧示意图，编写正确的汇编代码并反汇编得到机器指令，得到正确的攻击字符串。

3. 实验过程：

（1） 为了实现getbuf函数不论获得什么输入能够将cookie的值返回给test函数，我们编写的汇编文件必须实现：在getbuf执行完毕后将函数的返回值eax赋值为cookie的值，同时为了能够返回到test函数，我们需要将test函数中调用完getbuf函数之后的下一条语句的地址0x8048e81入栈。因为我们在写入攻击代码调用getbuf破坏的栈的状态在返回到test函数后体现出来的就是test函数中的ebp被修改，所以要恢复的话就是要恢复ebp的值。所以我们在写攻击的字符串的返回地址之前应该填入正确的ebp旧值。ebp的旧值可以通过gdb在getbuf处设置断点调试得到，如图3-21所示。

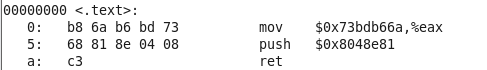


**图3-21 得到ebp旧值**

汇编代码和反汇编机器指令如图3-22所示。



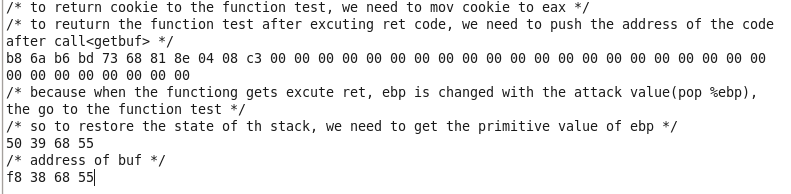
**图3-22汇编代码**



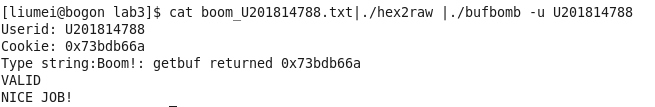
**图3-23 反汇编指令**

1. 和阶段3相同我们将恶意机器指令代码放在buf数组的开始，返回地址为buf的首址。

4. 实验结果：



**图3-24 攻击字符串**



**图3-25 测试结果**

由上图可知上述分析正确。

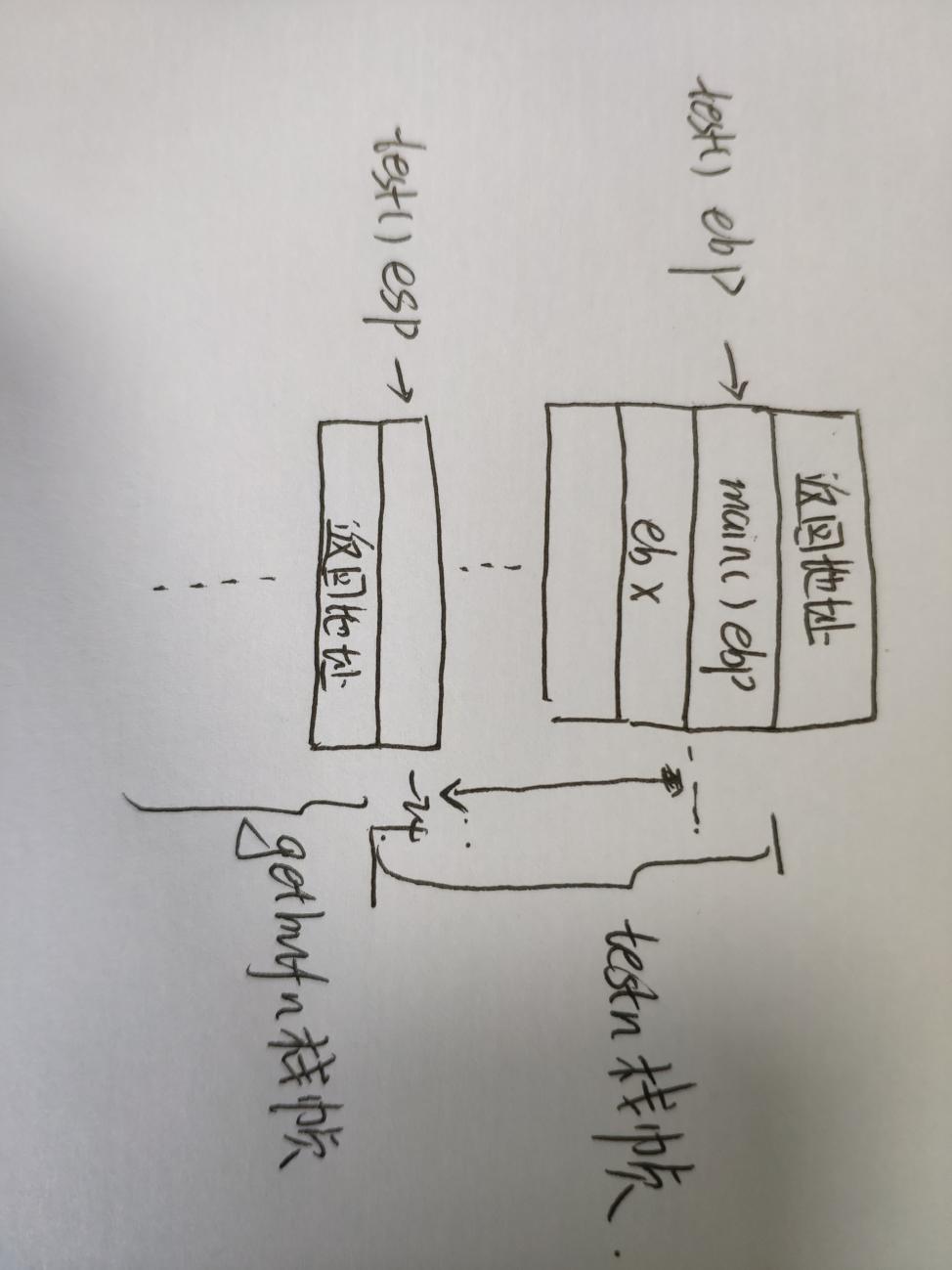
3.2.5 阶段5 Nitro

1. 任务描述：构造攻击字符串使得getbuf函数返回cookie值置testn函数，而不是返回值1，同时还要复原被破坏的栈帧的结构。

2. 实验设计：大致方法和阶段4相同，但是在阶段5当中由于getbuf的栈的地址是不确定的，所以不能将攻击字符串放在buf的开始，即返回地址需要自己确定，可通过gdb查看数据得到。

3. 实验过程：

（1） 和阶段4类似，我们首先需要写好汇编代码文件。在阶段4中我们的汇编代码文件并没有将ebp恢复旧值而是将其放在了攻击字符串里面实现，但是在阶段5中我们不能这样写。这是因为在阶段5中每次执行栈（ebp）均不同，所以不能直接在攻击字符串中写ebp的旧值。既然ebp是变化的，那我们如何获得ebp的旧值呢？我们看下面的图3-26：

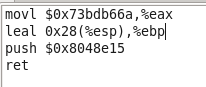


**图3-26**

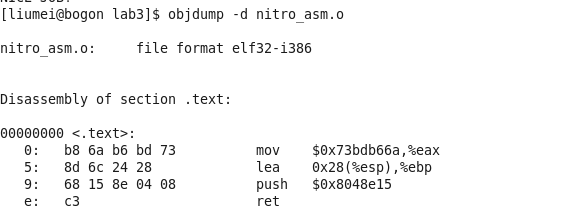
从图3-26我们可以知道，当getbuf函数执行完毕之后返回到testn函数时，esp位于testn栈帧的的栈顶位置，此时ebp和esp之间间隔0x24+4，即ebp=esp+0x28，所以ebp的旧值的获得;leal 0x28（%esp），%ebp。

除此以外值得注意的是这条语句必须写在push 返回地址语句的前面才能正确实现回到testn函数时ebp为原来的值没变。

汇编代码和反汇编指令如图3-27、3-28所示。



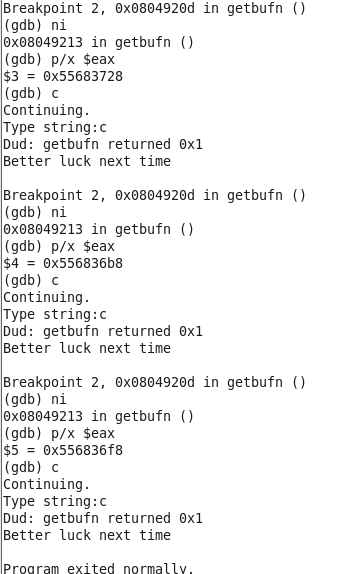
**图3-27 汇编代码**

****

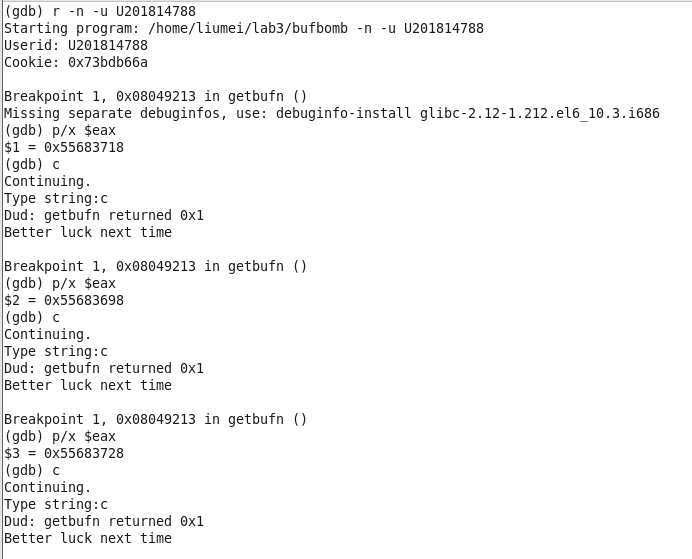
**图3-28 反汇编机器指令**

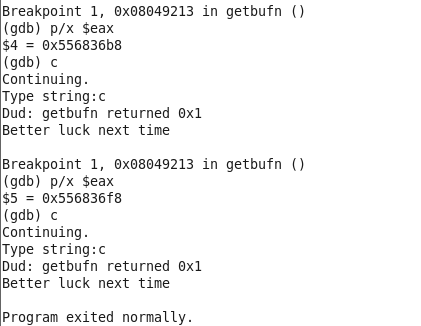
1. 得到机器指令之后，需要得到该指令在buf缓冲区存放的位置和返回地址。因为getbuf的栈在5次运行过程中在变化，所以buf的首址也在变化，此时我们就不能简单的直接写入buf的首址。通过gdb调试我们发现，在一次执行nitro函数的五次测试求解过程中，每一次的buf首址都不一样，但是每次执行nitro函数时得到的五次buf是一样的，也就是每一次执行nitro得到的buf地址均为：0x55683718, 0x55683698, 0x55683728, 0x556836b8, 0x556836f8。如图3-29和3-30所示。





**图3-29 第一次执行nitro**



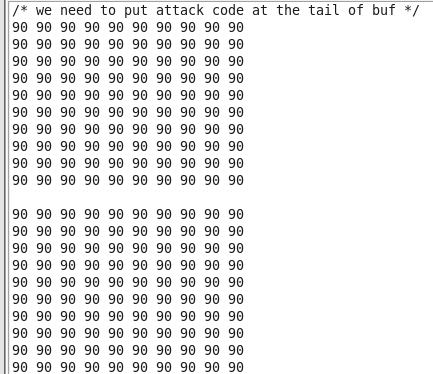


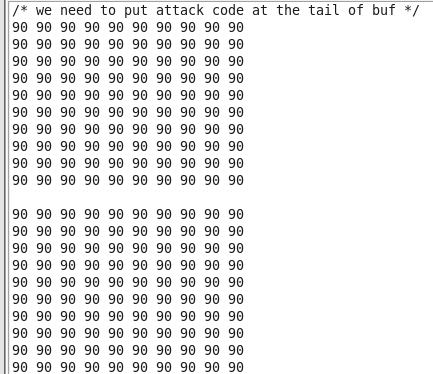
**图3-30 第二次执行nitro**

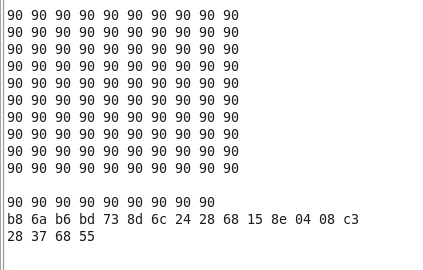
所以我们得到了buf首地址的大致范围。

1. 接下来确定攻击代码在buf缓冲区的位置和返回地址的具体的值。因为buf的首地址在变化，如果我们将攻击机器指令放在buf的开始位置，可能导致当我们输入的返回地址和实际的buf首址不同时，可能会返回到缓冲区靠后的位置使得攻击指令可能只执行一部分或者未被执行，或者返回到buf缓冲区之前的位置从其他程序开始执行，所以需要讲攻击指令代码放在buf的结尾处。放在结尾处之后，如果我们选择返回的地址buf小于实际的buf首址，会导致程序返回到缓冲区之前的其他程序部分执行，所以我们需要选择5次输出的buf首址中最大的一个0x5568728作为返回地址，此时实际的buf首址肯定小于等于返回地址，此时返回到缓冲区非首的位置，这样在执行恶意机器指令之前就只会执行nop指令（对程序无影响），最终正确执行攻击机器指令。

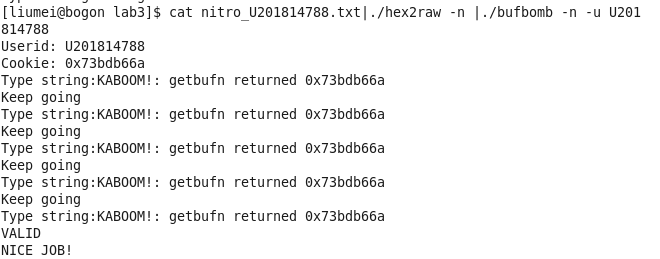
4. 实验结果：







**图3-31 攻击指令**

****

**图3-32 测试结果**

由上图可知上述分析正确。

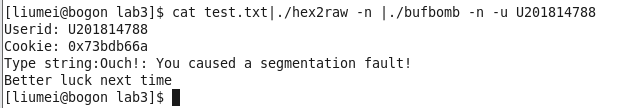
3.3 实验总结

本次实验的重点是缓冲区溢出的问题，其总体难度应该比实验二小一点。实验的难度逐级增加，恶意代码攻击性也更加强。实验实现攻击缓冲区总的来说就是利用gets函数的漏洞覆盖掉原始的返回地址，使程序返回到我们想要返回的位，除此以外，还可以通过在攻击字符串中加入机器指令去实现自己设计的函数。机器指令的获得可以在了解程序栈帧的基础上结合汇编和反汇编实现。

无感攻击的重点在于发生了攻击但是函数返回的时候仍然是返回到原本应该返回的位置，并且栈的状态也没有变化，但其实返回结果可能发生了改变。同时即使是在实际过程中调用函数的栈发生了改变也可以通过调试的方法获得来成功恢复栈的状态来掩人耳目。

恢复栈的状态的关键是恢复ebp的值，在稳定的栈中ebp的值可以直接通过调试获得，在随机的栈中，ebp的值要根据栈帧的具体情况靠和esp的相对位置决定（这个相对位置是一定的）。

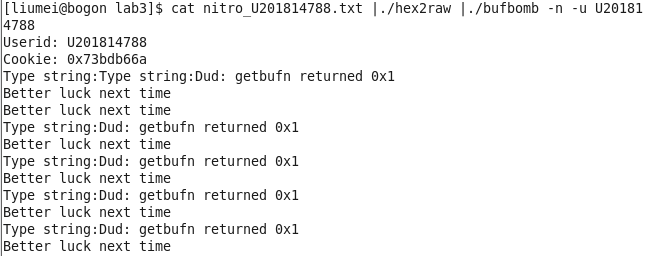
对于最后一个实验nitro，出现的问题比较多。首先是buf首址的获得，需要经过调试才能知道范围。对于攻击字符串我首先尝试的时候除了攻击机器指令，其他的我全部写的是00，结果显示：



**图3-33 错误结果**

这个错误输出的意思是我的缓冲区发生了溢出。事实上，我们应该期望得到的结果是返回缓冲区逐步执行指令一直到恶意机器指令所在的位置，而如果我们输入00的话，缓冲区的数据就只是作为数值存放而非机器指令，所以其实根本就无法执行我们的恶意机器指令，所以应该将数值00改为90，因为0x90正是指令nop的机器代码，可以执行。

注意测试nitro的时候hex2raw的后面也需要加入-n 否则我们输入的攻击字符串只会执行一次，其余四次的getbuf的返回值都为1（如图3-34）。



**图3-34 错误命令**

完成这次实验之后，我意识到了缓冲区溢出攻击会对程序造成多大的影响，所以在编写程序的时候，一定要注意尽量编写没有漏洞的正确代码。

实验总结

这门课程一共进行了三次实验，三次实验主要是针对书上的2、3章节的运用和分析。通过这三次实验，我对计算机系统有了更深的把握。在学这门课之前我是很少有想过关于一些溢出、阶段、缓冲区攻击等这些更加底层的知识的，向来是只要写好了程序就万事大吉了，至于是否存在漏洞，是否考虑到了全部的情况都很少有过分析。这门课的知识比较多，而这门实验正是给我们提供了一个很好的练手机会。

第一个实验让我对基本数据有了更深的把握。在这个实验中我需要从一些基本的运算操作符出发去设计一些平时写程序会用到的一些操作（这些操作太基本了，所以在写程序的时候基本都不怎么需要思考直接拿来用就是了，但实际上再真的设计起来需要认真的思考如何实现），在设计过程中我们需要仔细的去了解一些特殊的情况，往往设计的这个函数无法过关就是在一些特殊的数据上不成立。

第二个实验涉及到对反汇编语句的理解和运用。这个实验主要进行的是对反汇编代码的推测，理解代码的大致流程和实现的功能，因为有汇编课程的基础，反汇编的理解倒是没有那么难。在这个实验中还掌握了一小部分gdb工具的功能（单步调试等）。这个实验出现的几个难点还是有关递归方面，绕的弯有点多。想要加快过关的速度，还是需要多练多看。

第三个实验涉及到缓冲区攻击。写完这个实验后还是感谢蛮多的。大一的时候用c语言我虽然知道gets函数存在风险但是没想到这个风险带来的后果原来如此之大。包括我在有一次汇编实验中调用scanf函数时编译器也给我报了警告，现在我明白了对于这些输入函数，一定要十分注意缓冲区覆盖的问题。在编写一个函数的时候，为了保证安全性需要想办法避免这些漏洞，这是作为一个合格程序员的基本素养。

通过这三次实验，我明白作为一个程序员，不仅仅需要会写代码，还要对计算机底层的工作有一定的了解，这样才能帮助我们写出更加安全可靠的代码。