**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）

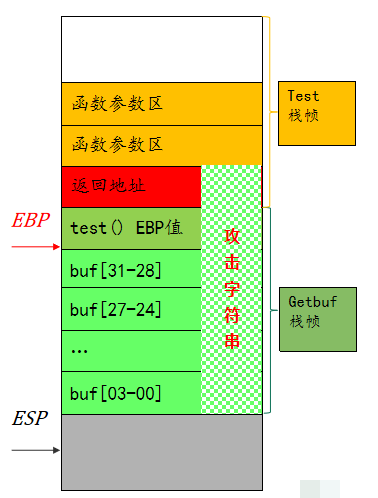


图3-1

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。

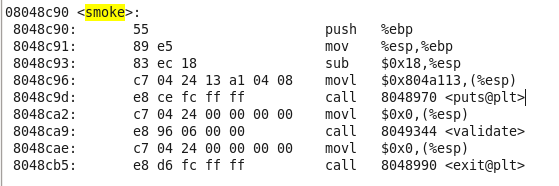


图3-2

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）。

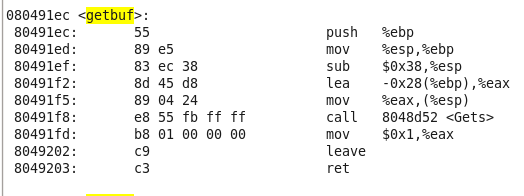


图3-3

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：

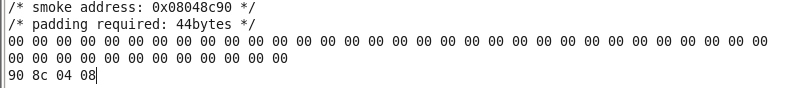


图3-4 输入的攻击字符串

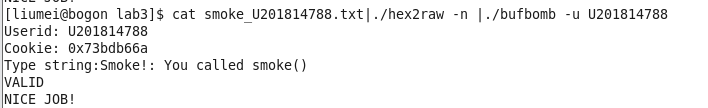


图3-5

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）

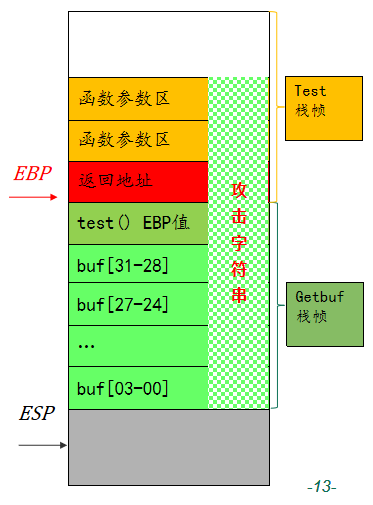


图3-6 栈帧示意图

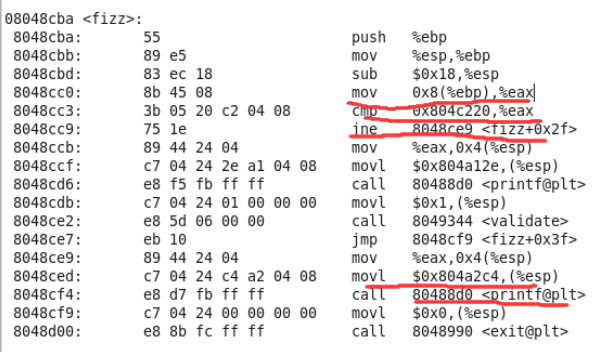


图3-7

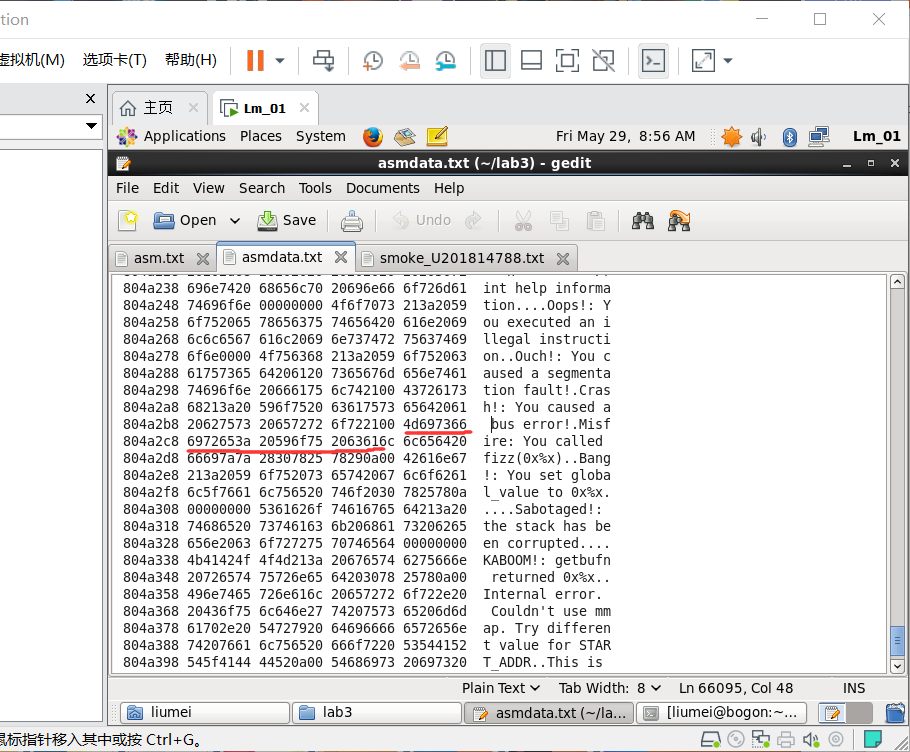


图3-8

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

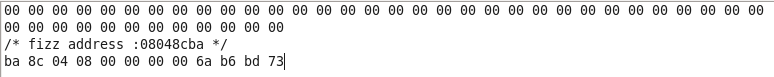


图3-10 输入的攻击字符串

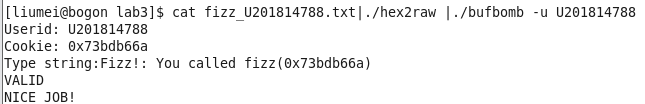


图3-11 测试结果

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。

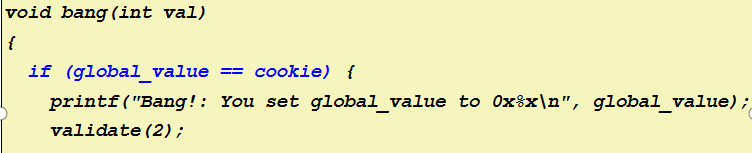


图3-12 bang c函数

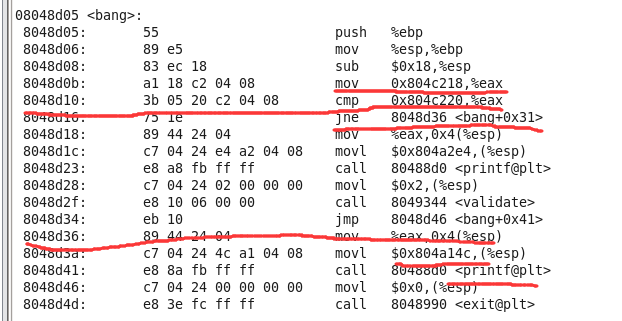


图3-13 bang反汇编代码

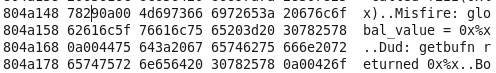


图3-14

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。

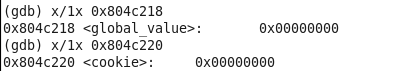


图3-15

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。

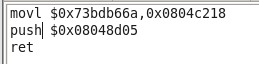


图3-16 汇编代码

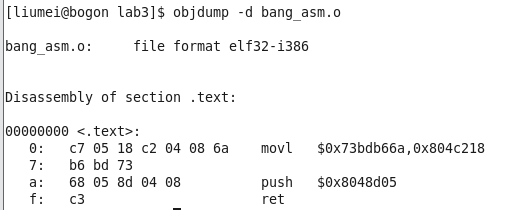


图3-17 返回的机器指令

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。

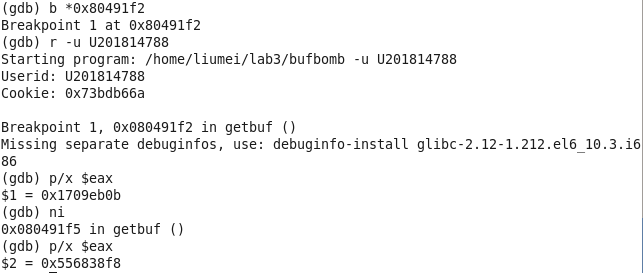


图3-18 得到buf的首址为0x556838f8

4. 实验结果：

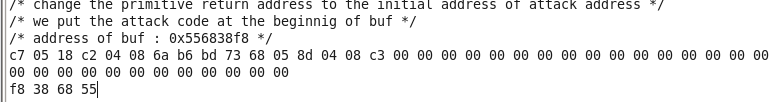


图3-19 攻击字符串

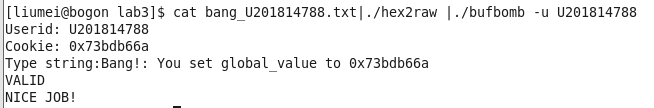


图3-20 测试结果

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。

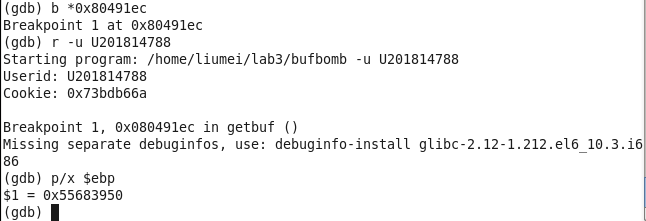


图3-21 得到ebp旧值

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

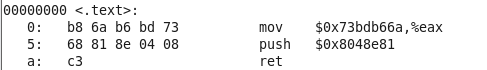


图3-23 反汇编指令

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：

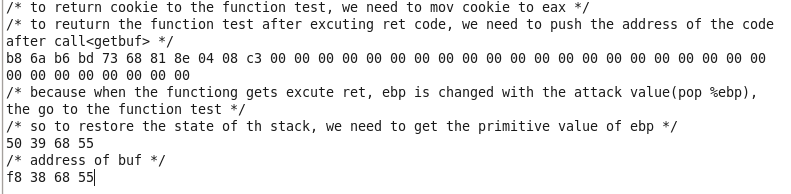


图3-24 攻击字符串

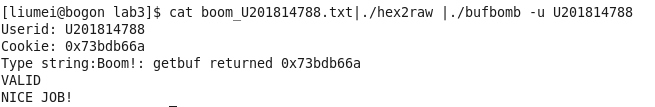


图3-25 测试结果

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：

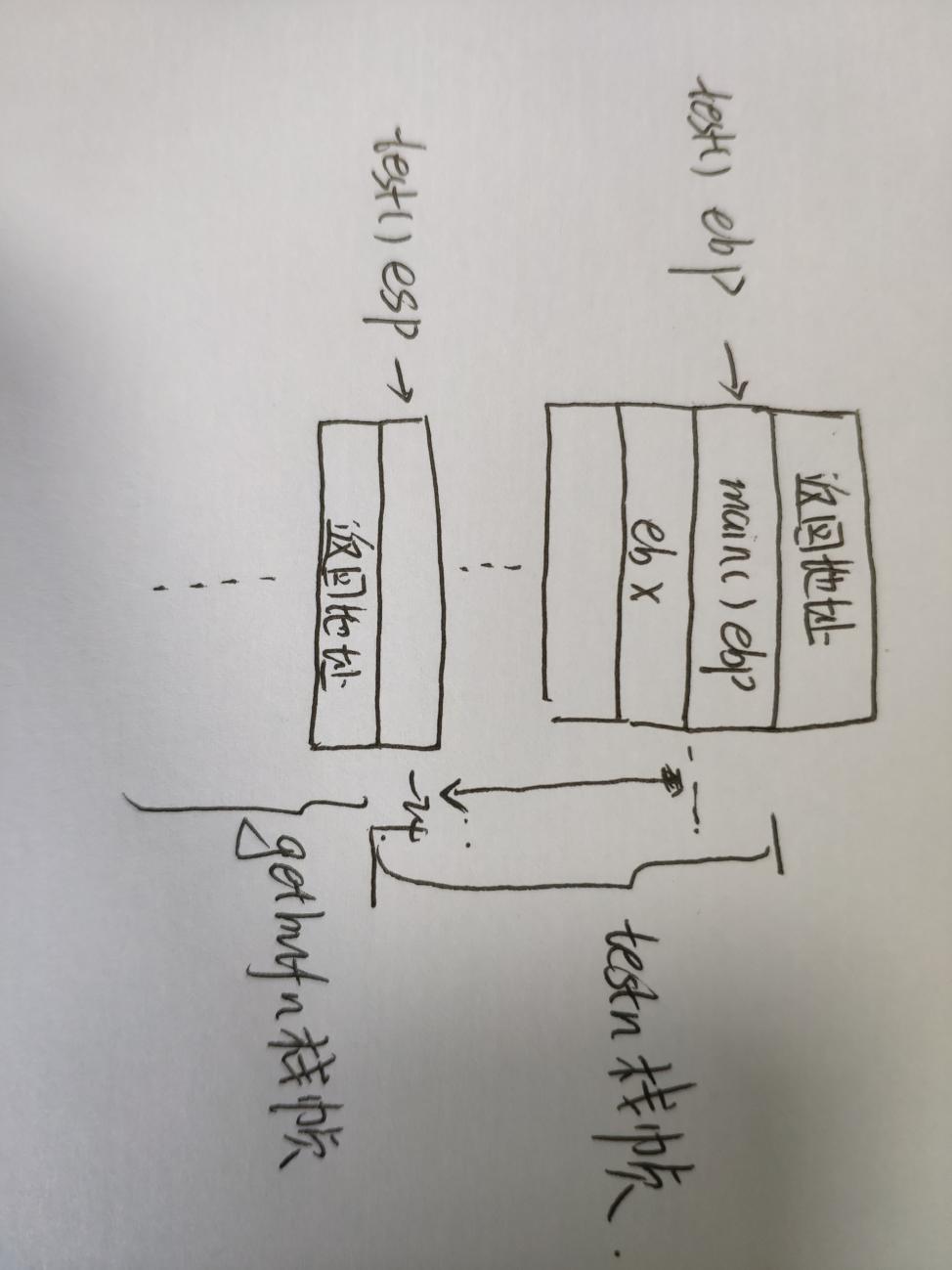


图3-26

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。

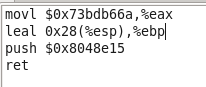


图3-27 汇编代码

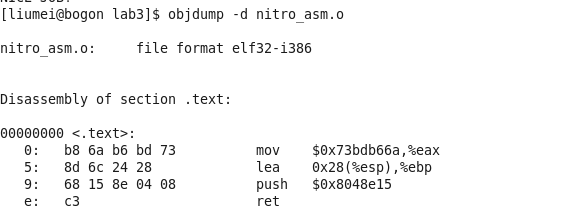


图3-28 反汇编机器指令

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。



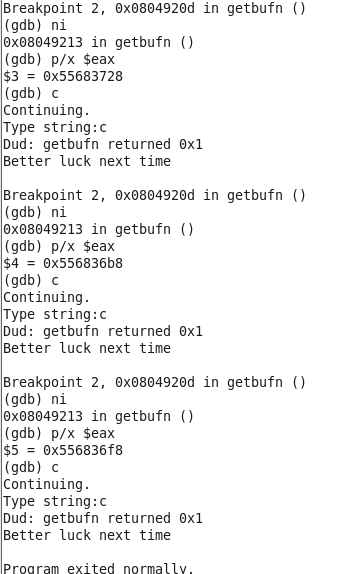
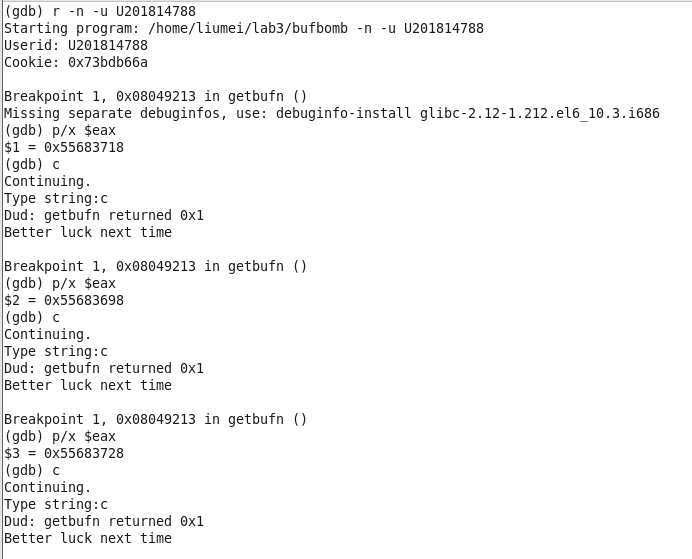


图3-29 第一次执行nitro



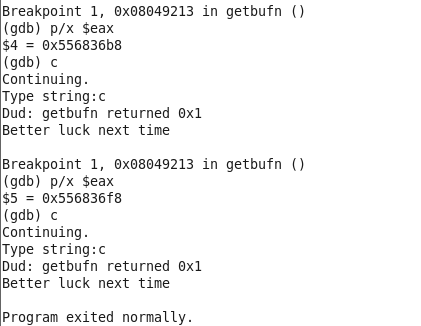
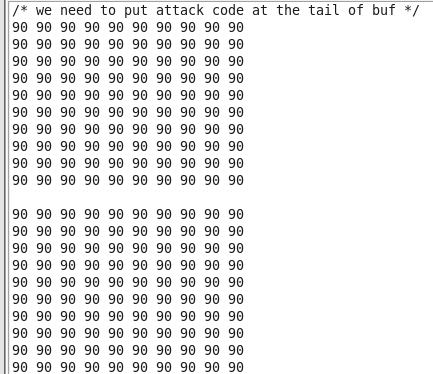


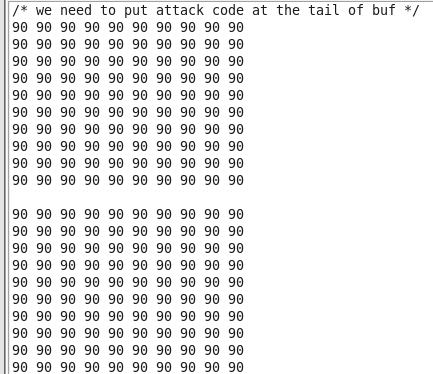
图3-30 第二次执行nitro

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：





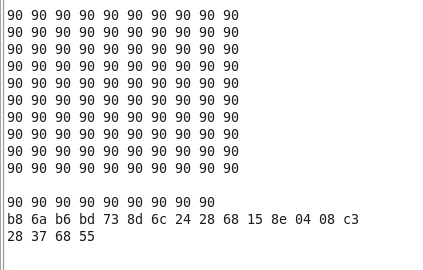


图3-31 攻击指令

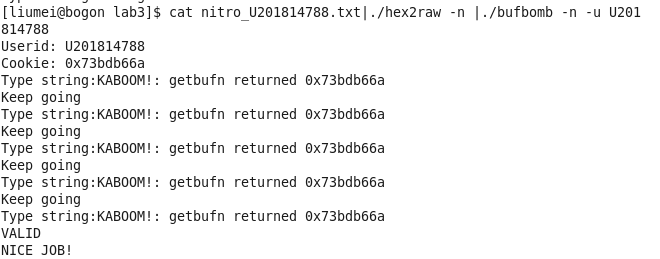


图3-32 测试结果

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：

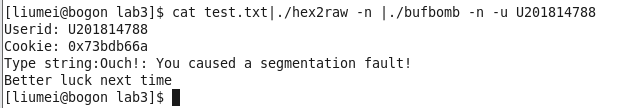


图3-33 错误结果

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）。

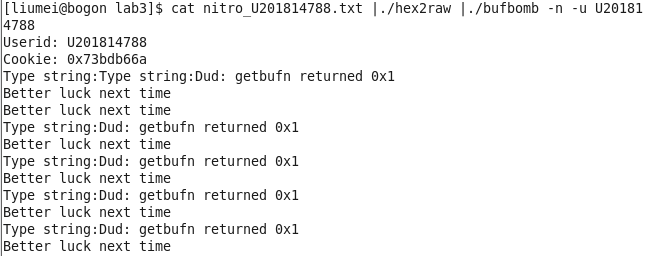


图3-34 错误命令

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。