

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**专业班级： ACM1901班**

**学 号： U201915035**

**姓 名： 邹雅**

**指导教师： 刘海坤**

**报告日期： 2021年 6月 8日**

**计算机科学与技术学院**

**目录**

[实验2： 2](#_Toc73983883)

[实验3： 4](#_Toc73983884)

[实验总结 5](#_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。

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。

**2.2 实验内容**

* 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。

**2.2.1 阶段1 拆除<phase\_1>**

1.任务描述：

第一阶段主要检测反汇编中的基础部分，通过对于字符串比较过程的解析来拆除炸弹的第一阶段。

2.实验设计：

利用objdump对bomb进行反汇编并将汇编代码输出到asm.txt中。vim查看汇编源代码。利用gdb来查看对应地址存放的内容。

3.实验过程：

观察phase\_1的汇编源代码

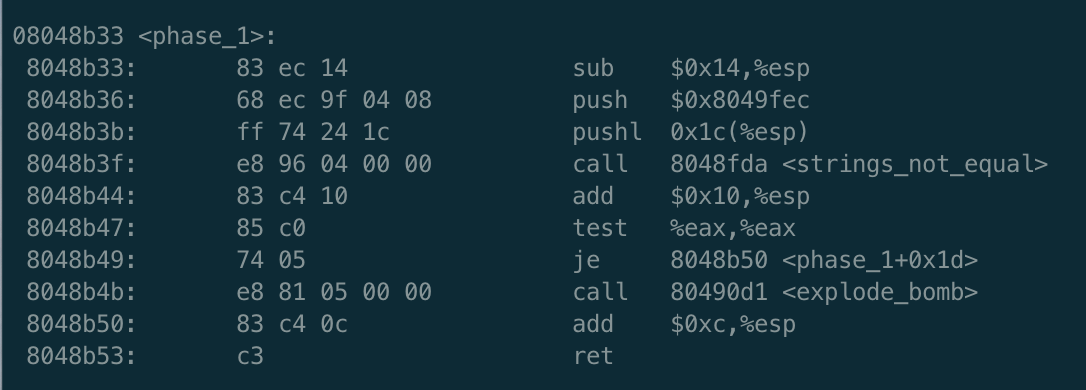


图2.2.1.1 phase\_1汇编源代码

可以看到在调用“strings\_not\_equal”之前push了一个地址0x8049fec，可见该字符串保存地址就是该地址。利用gdb查看该地址内容：

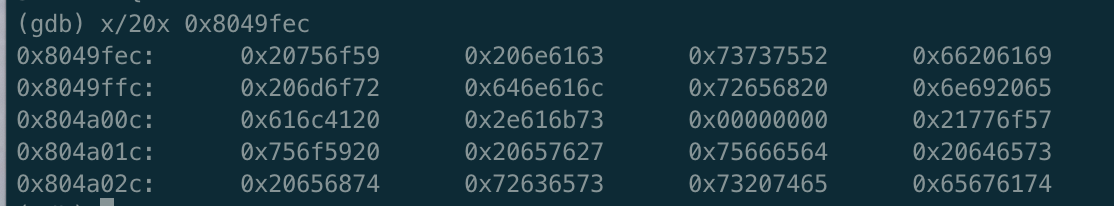


图2.2.1.2 0x8049fec地址内容

解析字符ASCII码值，得到结果为：You can Russia from land here in Alaska.

输入测试可见：

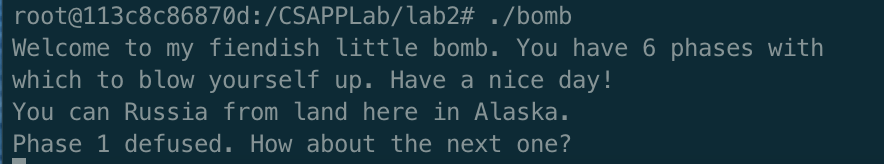


图2.2.1.3 phase\_1测试结果

4.实验结果：phase\_1的解为字符串：You can Russia from land here in Alaska.

**2.2.2 阶段2 拆除<phase\_2>**

1.任务描述：

phase\_2涉及汇编源程序的循环。

2.实验设计：

通过源代码观察分析源程序的循环思路、功能，进而推断结果。

3.实验过程：

观察phase\_2的汇编源代码（已添加必要英文注释）

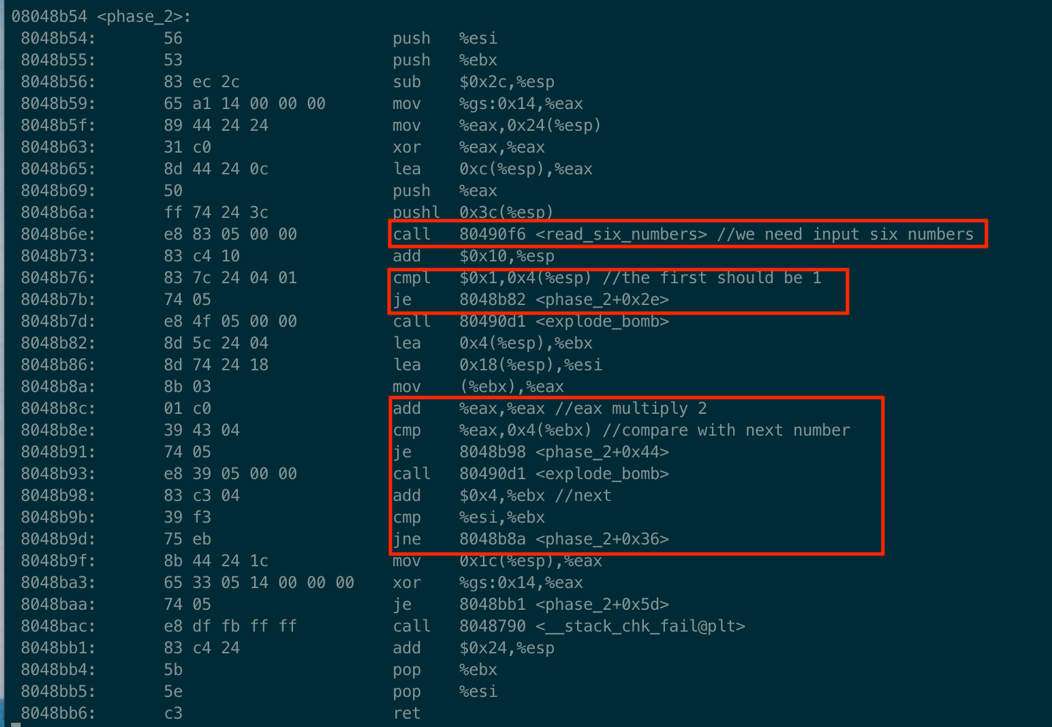


图2.2.2.1 phase\_2源代码

观察可得调用了“read\_six\_numbers”函数，可以知道本阶段的密码是六个数字。且把保存在堆栈中的第一个数字和0x1比较，如果不相等就爆炸，所以第一个是1.

在后续比较中，因为语句“add %eax,%eax”说明把第一个数变为两倍再做后续的比较，如此循环直到读完全部数字结束。由此可以得出这六个数字应该是：1 2 4 8 16 32.

输入测试可见：

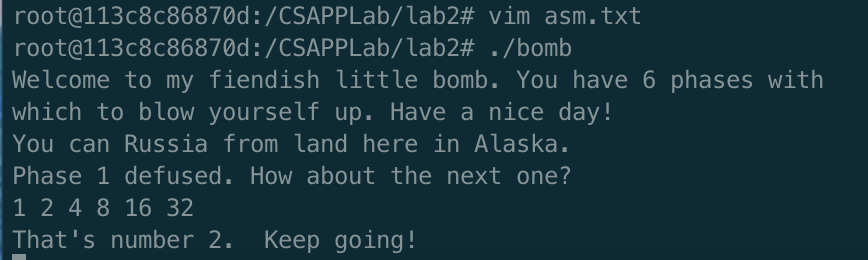


图2.2.2.2 phase\_2测试结果

4.实验结果：

phase\_2的解为1 2 4 8 16 32.

**2.2.3 阶段3 拆除<phase\_3>**

1.任务描述：

phase\_3涉及含switch语句的条件/分支语法。

2.实验设计：

通过源代码观察分析源程序的思路、功能，进而推断结果。利用gdb来查看对应地址存放的内容。

3.实验过程：

观察phase\_3的汇编源代码，截取关键语句进行分析说明。

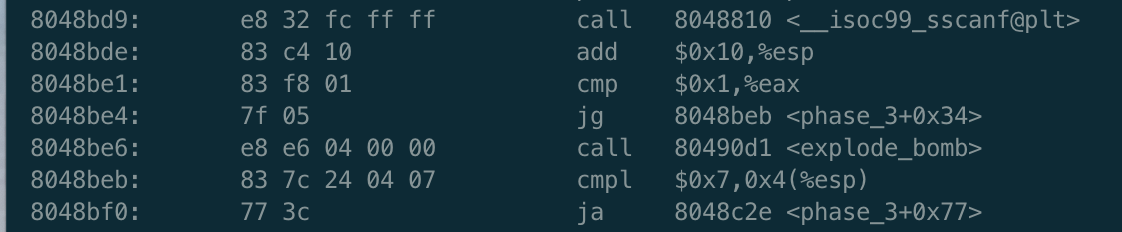


图2.2.3.1 phase\_3源代码1

比较scanf的输入结果，通过跳转指令的对应语句分析输入的参数数量应该大于1. ja跳转到的语句是调用“explode\_bomb”函数，所以推断出第一个参数应该要小于等于7.

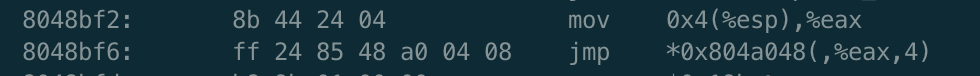


图2.2.3.2 phase\_3源代码2

这两句指令的意思是，根据输入的第一个参数来寻找地址0x804a048中保存的内容，并跳转到该内容对应的指令。通过gdb查看0x804a048的内容。

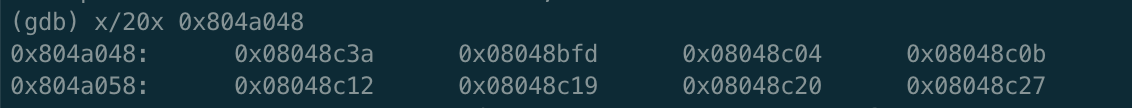


图2.2.3.3 0x804a048的内容

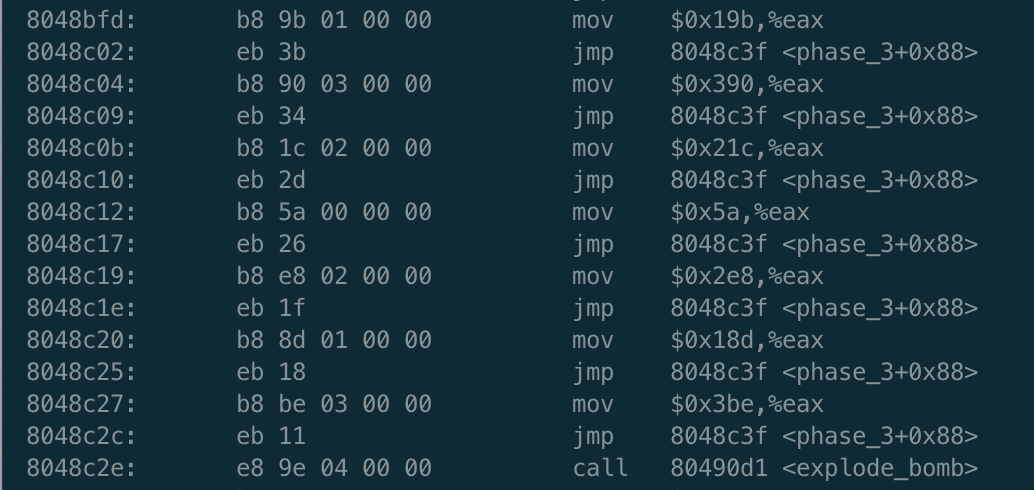


图2.2.3.4 phase\_3源代码3

可见这是一个switch语句。根据第一个参数来选择另一个参数的值。以第一个参数为1来分析，其他类似。当第一个参数为1时，程序跳转到0x08048bfd，eax等于0x19b.

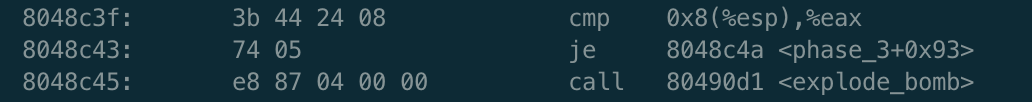


图2.2.3.5 phase\_3源代码4

上述跳转到0x8048c3f后是把第二个参数的值和eax中保存的值比较，不相等则爆炸。如此我们可以得到一组解为1 411，411为0x19b的十进制对应数值。类似我们可以得到剩余六组解2 912、3 540、4 90、5 744、6 397、7 958.

输入测试可见：

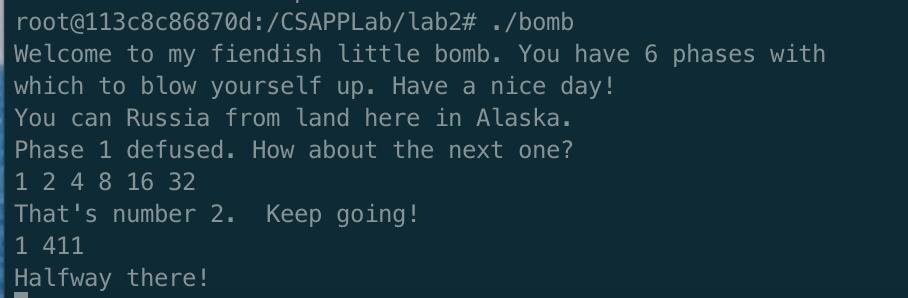


图2.2.3.6 phase\_3测试

4.实验结果：

由上分析可得，phase\_3有七组解，分别为1 411、2 912、3 540、4 90、5 744、6 397、7 958。

**2.2.4 阶段4 拆除<phase\_4>**

1.任务描述：

phase\_4涉及递归调用和栈的内容。

2.实验设计：

通过源代码观察分析源程序的思路、功能，进而推断结果。利用gdb来查看寄存器的值。

3.实验过程：

观察phase\_4的汇编源代码，截取关键语句进行分析说明。

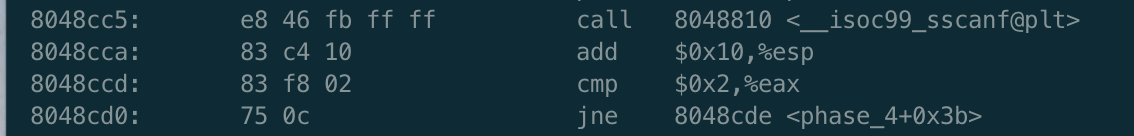


图2.2.4.1 phase\_4源代码1

jne的跳转语句为调用“explode\_bomb”函数，所以可以知道输入的一定是两个参数。

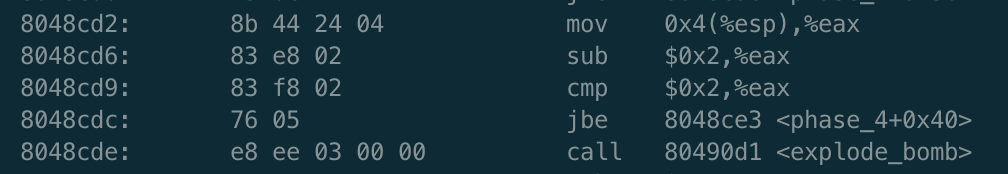


图2.2.4.2 phase\_4源代码2

输入的第二个参数先减2，再和2比较，如果小于等于2则满足条件，否则爆炸。根据数值的机器码表示，可以推出第二个参数应该是一个大于等于2且小于等于4的值，即可能是2、3、4.

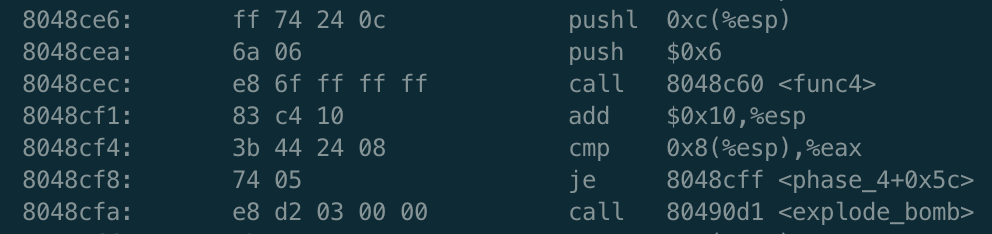


图2.2.4.3 phase\_4源代码3

把输入的第二个参数和6作为形式参数调用func4，调用完之后把返回值和输入的第一个参数比较，如果相等则满足条件，否则爆炸。

利用gdb直接观察寄存器的值来得到func4调用完之后的值。先输入第二个参数为2的情况，观察寄存器。

先在phase\_4函数打上断点，run到该位置时停下来，再利用disassemble指令来获取phase\_4下的汇编源代码来观察打断点的语句位置，在调用func4后的语句打上断点。

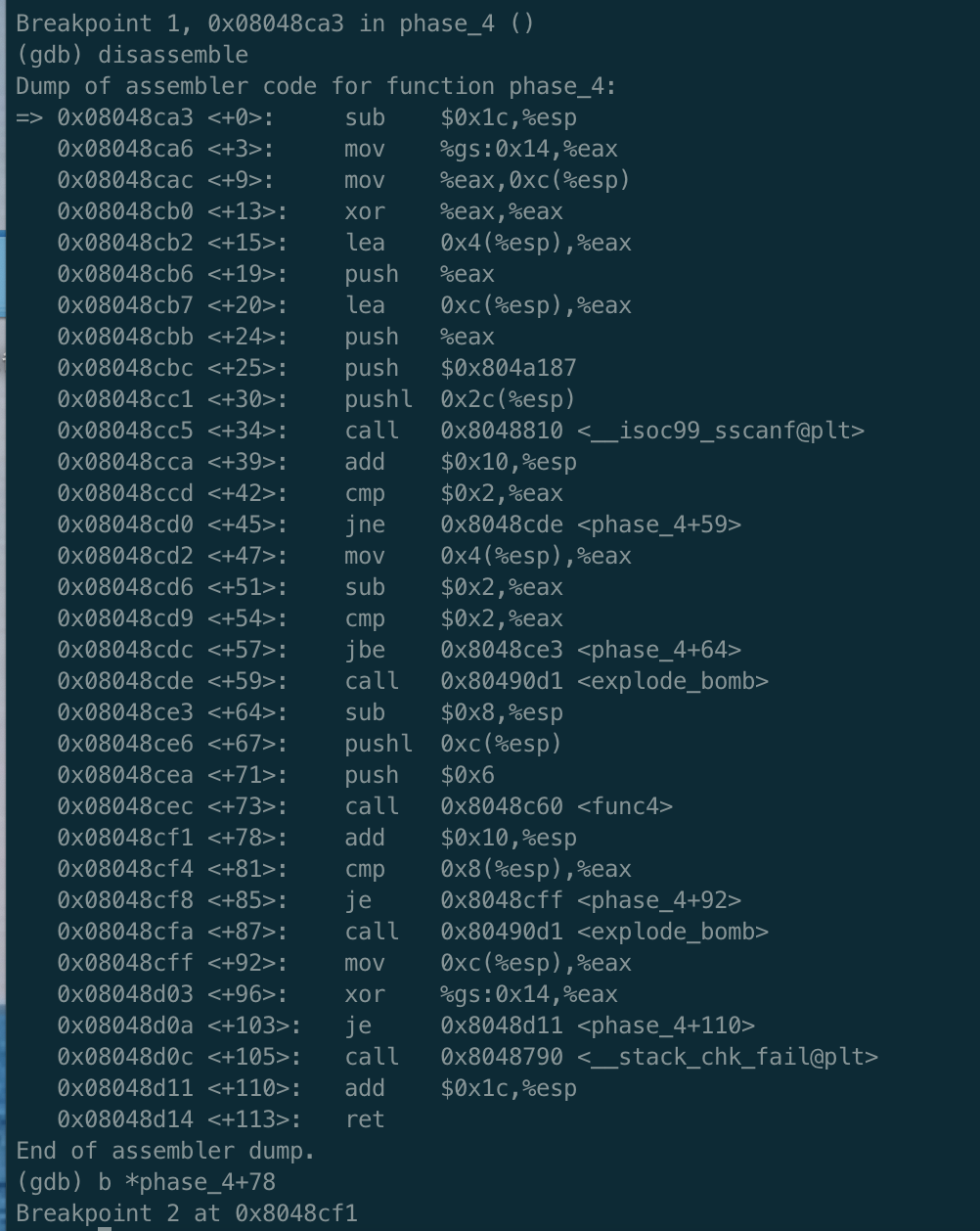


图2.2.4.4 disassemble语句观察并打断点

继续运行到第二个断点并观察寄存器的值。

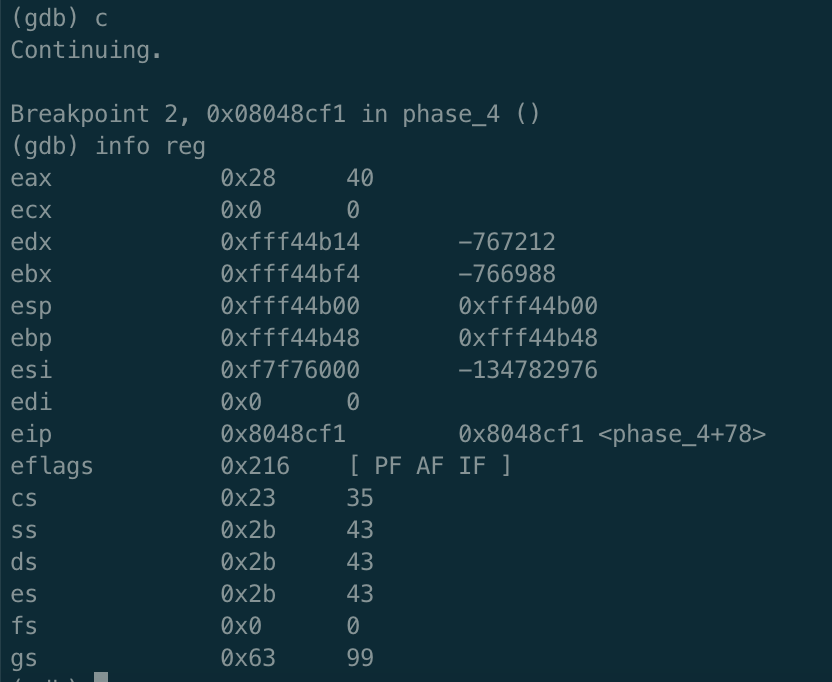


图2.2.4.5 观察寄存器的值

所以第二个参数为2时，第一个参数为40.同理可得另两组解为60 3、80 4.

将结果写入ans.txt文件，并把先前三个阶段的解写入，测试本阶段：

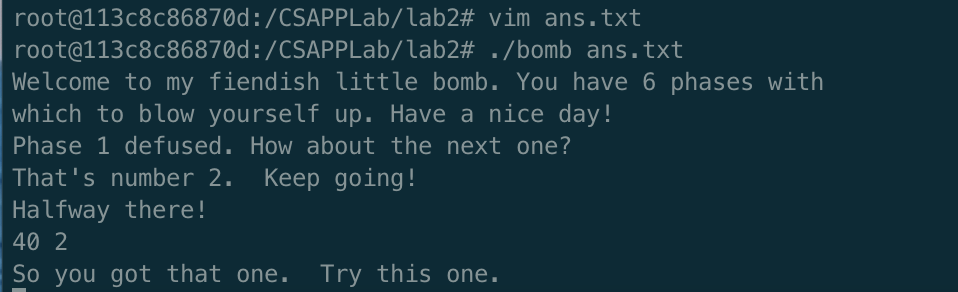


图2.2.4.6 phase\_4测试结果

4.实验结果：

phase\_4有三组解，分别为40 2、60 3、80 4。

**2.2.5 阶段5 拆除<phase\_5>**

1.任务描述：

phase\_5涉及指针的内容。

2.实验设计：

通过源代码观察分析源程序的思路、功能，进而推断结果。利用gdb来查看对应地址存放的内容。

3.实验过程：

观察phase\_5的汇编源代码，截取关键语句进行分析说明。

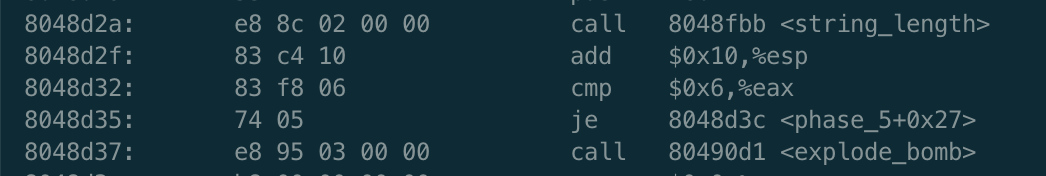


图2.2.5.1 phase\_5源代码1

可得应该输入一个字符串，字符串的长度为6.

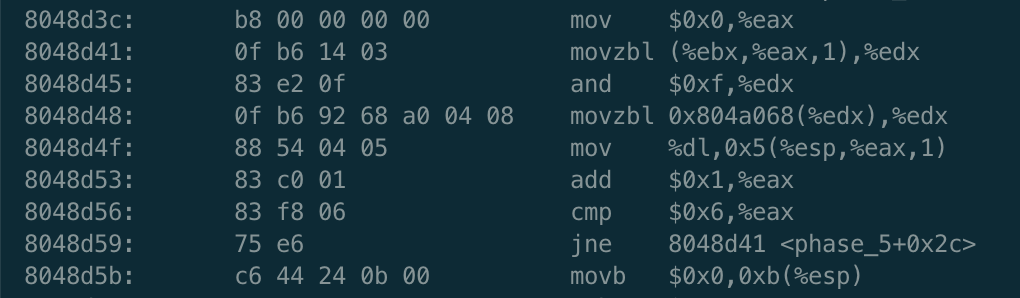


图2.2.5.2 phase\_5源代码2

这是理解整个函数的关键。此时的ebx指向的是输入的字符串在堆栈中的位置，通过eax把每个字符取出来，在“and $0xf,%edx”指令中，截取字符的低4位，以这个为偏移地址在基址0x804a068里取对应字符保存到堆栈中。

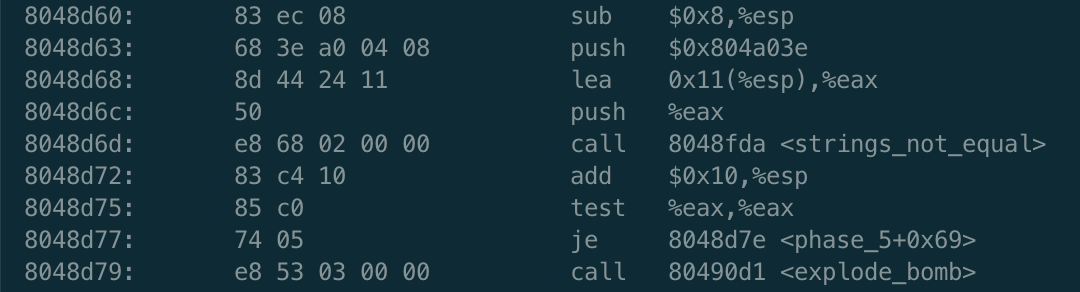


图2.2.5.3 phase\_5源代码3

把上述处理后的字符串地址和0x804a03e压入堆栈，再调用“strings\_not\_equal”，返回值为0时满足结果，否则爆炸，也就是这两个字符串必须要相等。

利用gdb来观察0x804a03e地址中的内容。

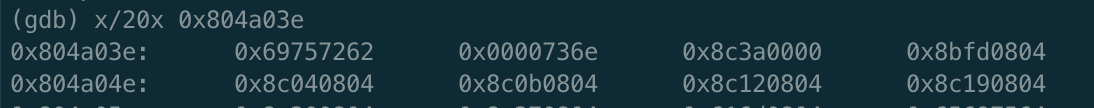


图2.2.5.4 0x804a03e地址中的内容

解析得这六个字符为“bruins”。利用gdb来观察0x804a068地址中的内容。

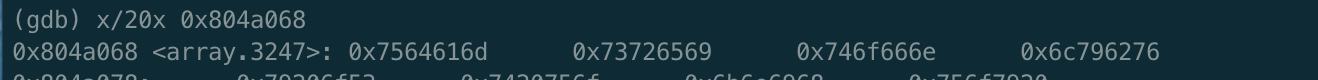


图2.2.5.5 0x804a068地址中的内容

在这块内存中找到对应“bruins”各个字符的位置，得出相对于0x804a068的偏移地址为13、6、3、4、8、7，这是输入字符串ASCII码值的低四位值，转化为ASCII码后其中一个解为mfcdhg（0x6d、0x66、0x63、0x64、0x68、0x67）。

输入测试结果：

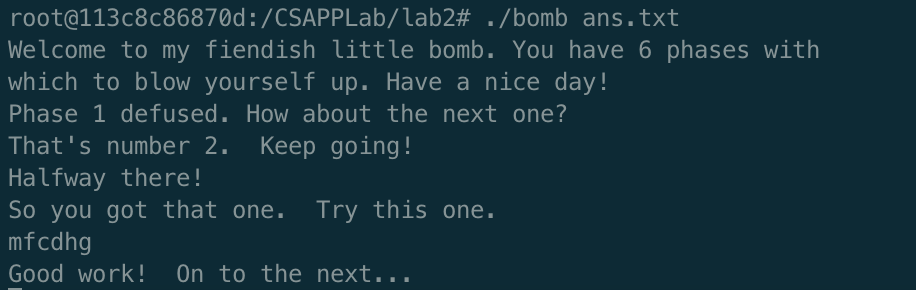


图2.2.5.6 phase\_5测试结果

4.实验结果：

phase\_5的解为mfcdhg。

**2.2.6 阶段6 拆除<phase\_6>**

1.任务描述：

phase\_6涉及链表/指针/结构的内容。

2.实验设计：

通过源代码观察分析源程序的思路、功能，进而推断结果。利用gdb来查看对应地址存放的内容。

3.实验过程：

观察phase\_6的汇编源代码，截取关键语句进行分析说明。

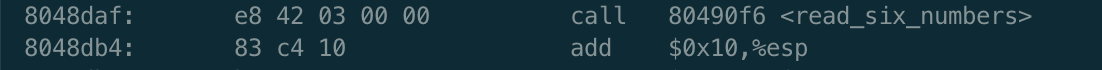


图2.2.6.1 phase\_6源代码1

由此可以得出输入为六个数字。

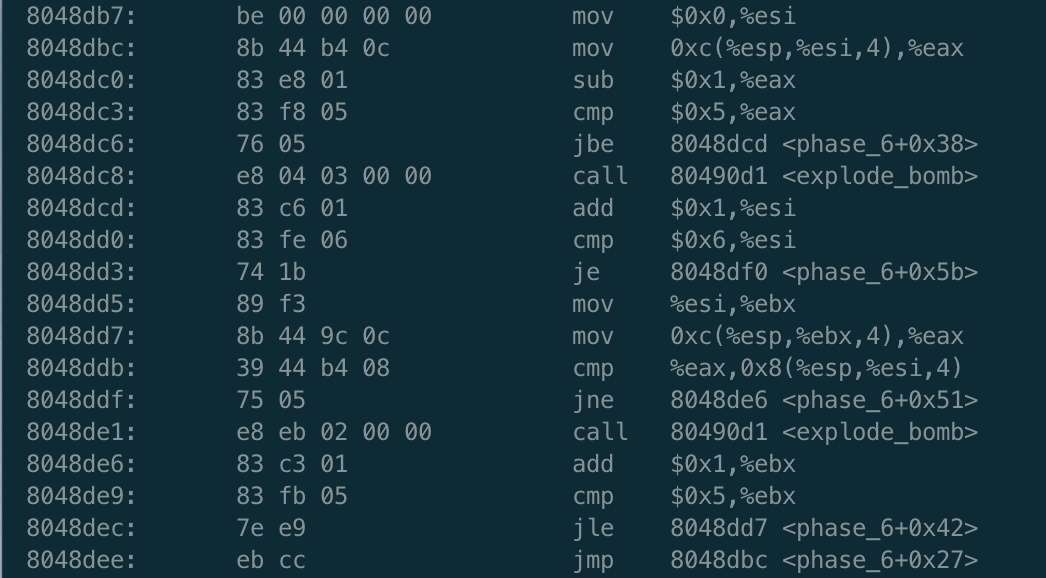


图2.2.6.2 phase\_6源代码2

循环判断。第一重输入的六个值都必须大于等于1且小于等于6，第二重循环判断六个值是否两两之间相等，如果相等则爆炸。所以可以推断这六个数是1、2、3、4、5、6的排序。

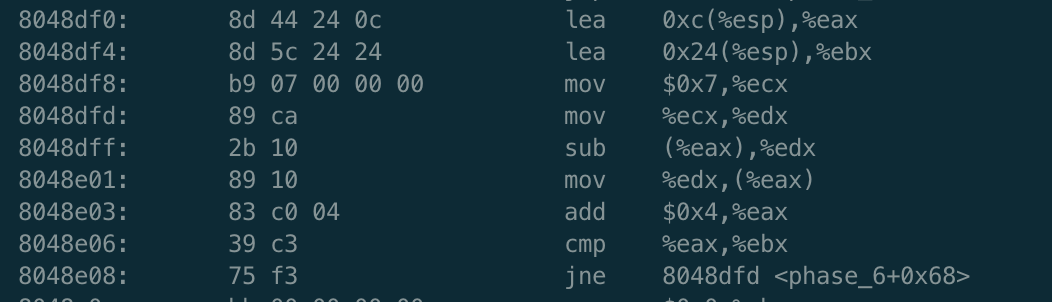


图2.2.6.3 phase\_6源代码3

循环读取每一个数，计算用7减去读入的数并保存至堆栈中，为后续表述方便称之为第二组数据。

后续一部分涉及到一个地址0x804c13c，我们先去查看这个地址中的内容。

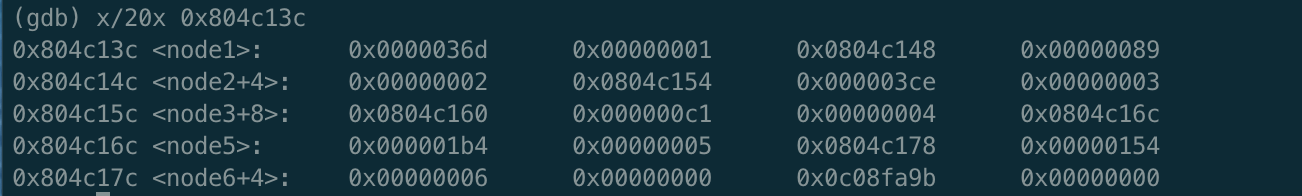


图2.2.6.4 0x804c13c地址中的内容

这是名为node的struct构成的链表内容。每个节点占12个字节，前四个字节保存了一个数值，第二个四个字节保存了索引顺序，第三个四个字节保存了next指针的值。

阅读剩下的汇编源代码（代码较长在此省略截图），可得其语意为把链表按照第一个数值进行排序，排序后的索引值与上述处理过的第二组数据进行比较，如果相等则满足条件，否则爆炸。

将0x36d、0x089、0x3ce、0x0c1、0x1b4、0x154进行排序，得到排序后的索引值为3 1 5 6 4 2。由此可以得到输入的数据应为4 6 2 1 3 5。

输入测试结果：

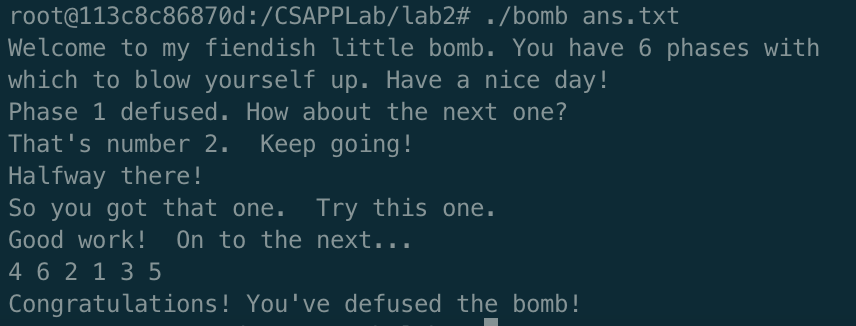


图2.2.6.5 phase\_6测试结果

4.实验结果：

phase\_6的解为4 6 2 1 3 5。

**2.2.7 阶段7 拆除<secret\_phase>**

1.任务描述：隐藏关卡

2.实验设计：

通过源代码观察分析源程序的思路、功能，进而推断结果。利用gdb来查看对应地址存放的内容。

3.实验过程：

在phase\_6下看到两个函数fun7和secret\_phase，说明存在隐藏关卡，检索secret\_phase发现其入口在phase\_defused中。

在phase\_defused有如下语句：

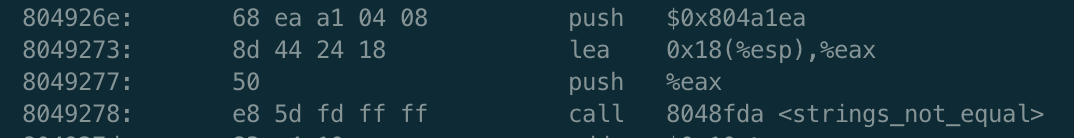


图2.2.7.1 phase\_defused源代码

可以看到涉及一个地址0x804a1ea,访问该地址的内容。

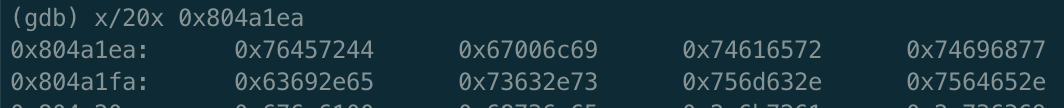


图2.2.7.2 0x804a1ea内容

解析该字符串为“DrEvil”,说明这是进入secret\_phase的字符串，根据提示将该字符串加在phase\_4的解后面，即可进入secret\_phase。

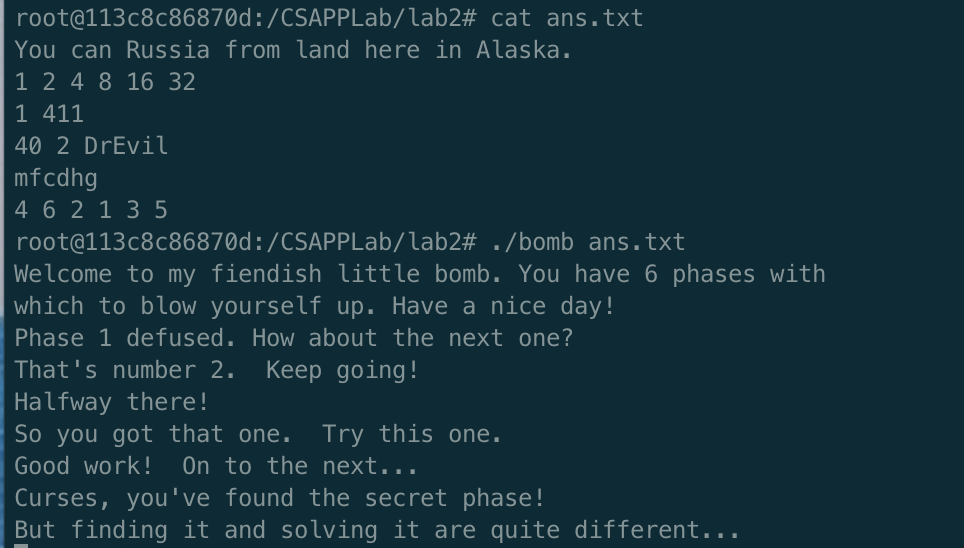


图2.2.7.3 进入隐藏关卡

观察secret\_phase的代码并分析。secret\_phase首先读入一行字符串，然后转化为数字保存。接着以这个数字为参数和另一个地址0x804c088调用fun7，并要求fun7的返回值为5.

查看0x804c088地址中的内容。

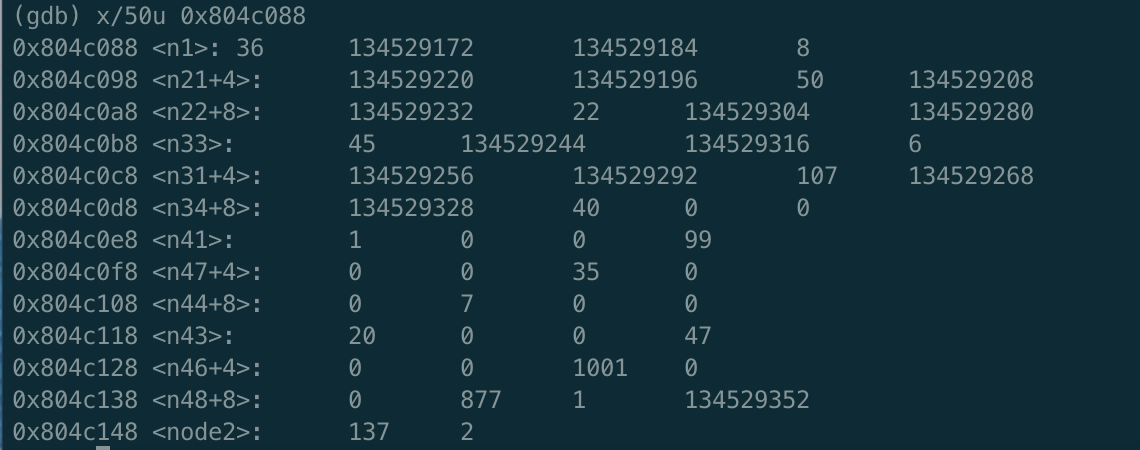


图2.2.7.4 0x804c088内容

这是一个二叉树结构，如下图所示：

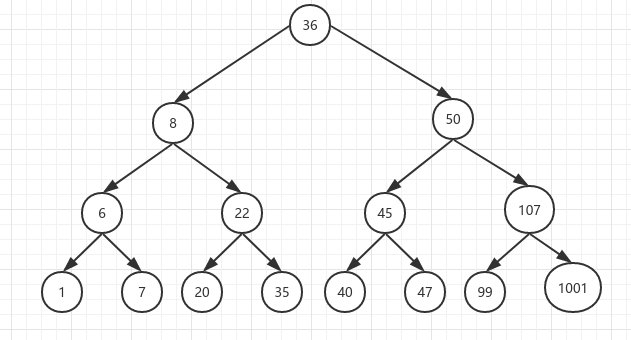


图2.2.7.5 二叉树内容

接着观察fun7的源代码.

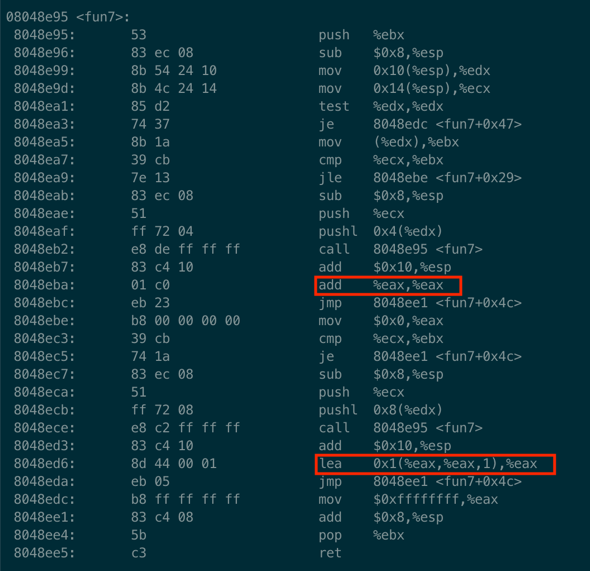


图2.2.7.6 fun7源代码

发现fun7的内容为对输入的数字在二叉树中进行搜索，如果搜索边为左子树的边则将返回值eax的值乘以2（对应“add %eax,%eax”语句），如果搜索边为右子树，则将返回值乘以2加1（对应“lea 0x1(%eax,%eax,1),%eax”语句）。

而我们所要得到的结果为5，那么（（0\*2+1）\*2）\*2+1=5，所有先到右子树再到左子树再到右子树，答案为47.

输入测试：

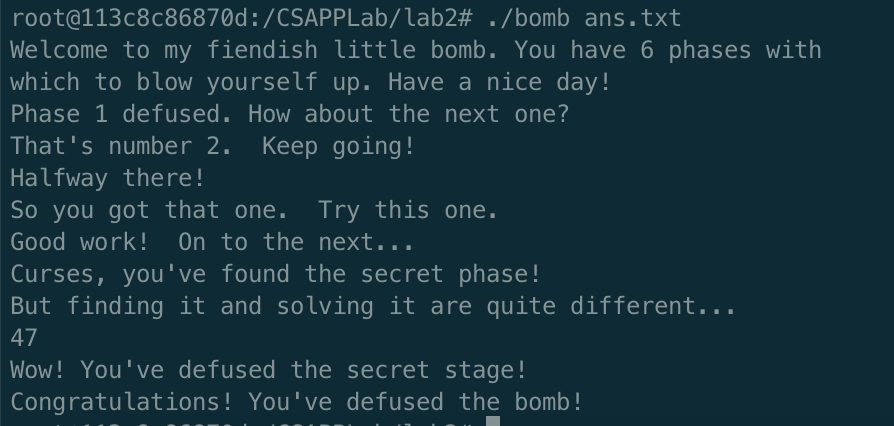


图2.2.7.7 secret\_phase测试

4.实验结果：

secret\_phase的解为47。

**1.3 实验小结**

通过本次实验，我对汇编语言有了更深程度的掌握，对C语言如何转换成汇编语言也有了更加深入的了解。特别是第四阶段的递归、第六阶段的链表和排序、隐藏阶段的二叉树，这些在机器级的表述让我学到了很多，对它们的工作方式也有了更深刻的了解。

本次实验中用到了objdump和gdb两个工具。其中objdump用于反汇编输出bomb程序的汇编源代码。整个实验做完我对gdb中的命令语句掌握了更多：

1. gdb bomb ——gdb运行某个程序
2. b +地址值 ——在某个地址处打断点
3. r ——run
4. ni ——单步执行机器语句
5. c ——continue
6. x/20x +地址值 ——查看某一个地址处的内存内容
7. info registers ——查看当前寄存器的值
8. disassemble ——查看当前所属地址下接着的汇编语句
9. q ——退出

# 实验3： 缓冲区溢出攻击

**3.1 实验概述**

加深对IA-32函数调用规则和栈帧结构的理解

**3.2 实验内容**

1.对目标程序实施缓冲区溢出攻击（buffer overflow attacks）

2.通过造成缓冲区溢出来破坏目标程序的栈帧结构

3.继而执行一些原来程序中没有的行为

**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：

利用objdump得到的反汇编代码推断出得缓冲区的长度，确定填充的字节数，然后将smoke函数的首地址覆盖返回地址.

3.实验过程：

观察反汇编代码getbuf函数的语句：

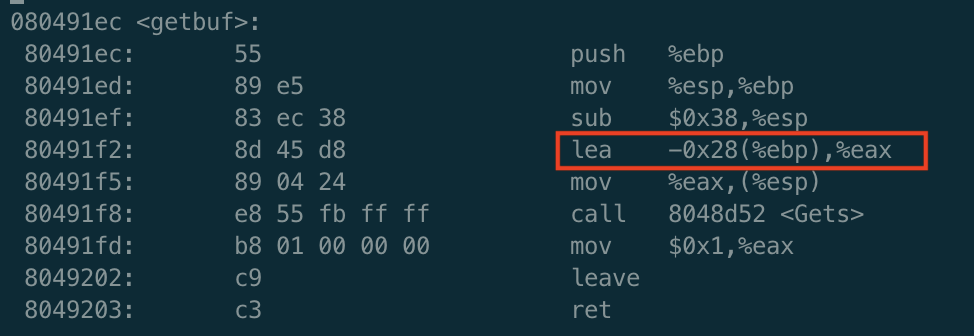


图3.2.1.1 getbuf函数反汇编代码

由此可以得出输入的字符串首地址距离此时ebp的指向有0x28（40）个字节，在push %ebp语句，原ebp占4个字节，由此可以得出返回地址距离输入的字符串首地址有44个字节。

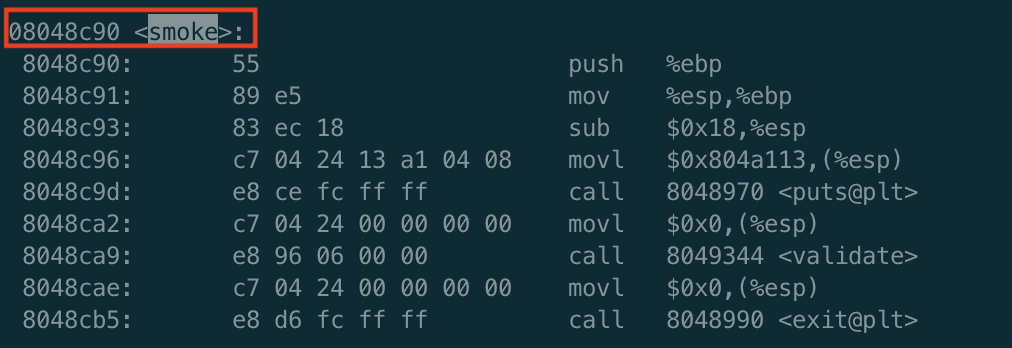


图3.2.1.2 smoke函数反汇编代码

观察得到smoke函数的地址为0x08048c90，于是构造44个任意字节加上该地址的文本文件smoke\_U201915035.txt，如下所示：

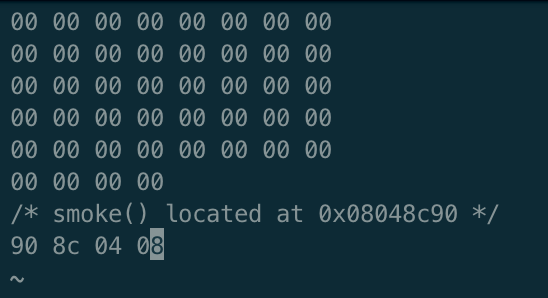


图3.2.1.3 smoke\_U201915035.txt文件内容

需要注意的是gets函数以换行，也就是0x0a作为结束的符号，所以填充前面40个字节的数据不能是0a。并且，x86使用的是小端方式，地址应该是90 8c 04 08这样保存。

4.实验结果：

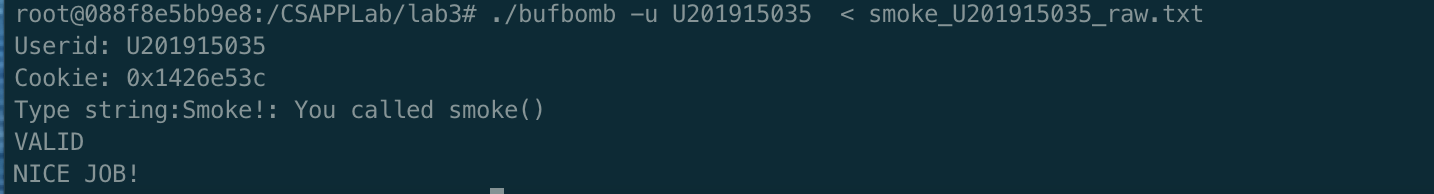


图3.2.1.4 smoke攻击成功截图

**3.2.2 阶段2 fizz**

1.任务描述：

构造攻击字串造成缓冲区溢出，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：

利用objdump得到的反汇编代码推断出得缓冲区的长度，确定填充的字节数，然后将fizz函数的首地址覆盖返回地址，并推断出fizz函数的参数在堆栈中存在的位置。

3.实验过程：

fizz阶段的实现基本和smoke阶段相同。

观察fizz函数，可以得到函数首地址以及函数参数所存放的位置。

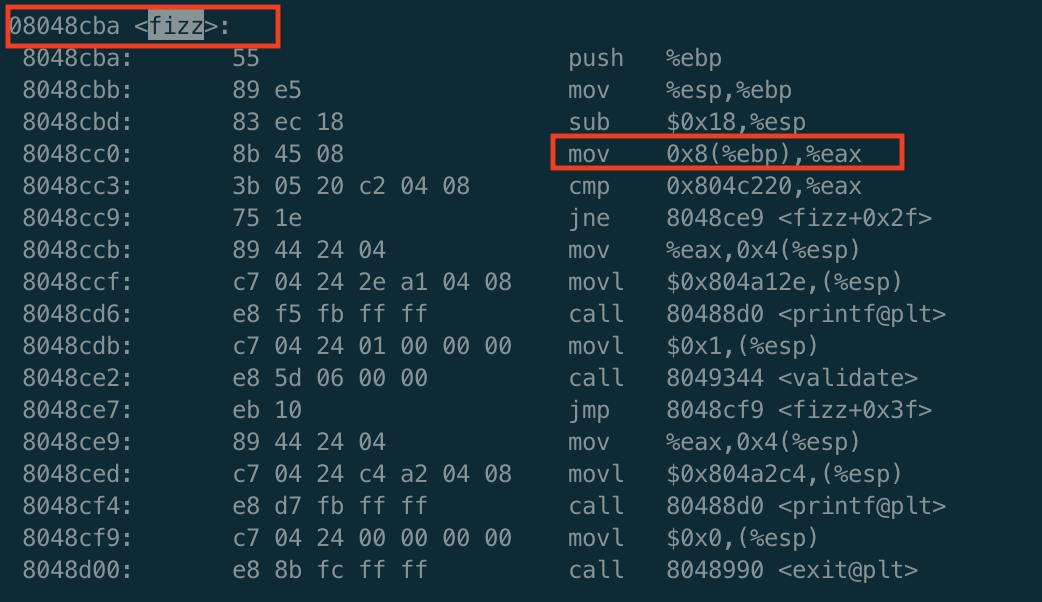


图3.2.2.1 fizz函数反汇编代码

因为fizz函数调用了exit退出程序所以不必再考虑fizz函数的返回地址问题。由“mov 0x8(%ebp),%eax” 语句，可以推断出，参数在此时ebp指向加0x8处。所以构建的文本文件fizz\_U201915035.txt中保存的是44个任意字节，加上fizz函数的起始地址，再加上4个任意字节和cookie值。

由makecookie程序得到cookie值如下：

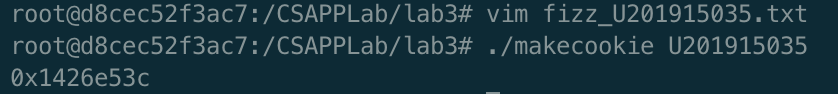


图3.2.2.2 cookie值的获取

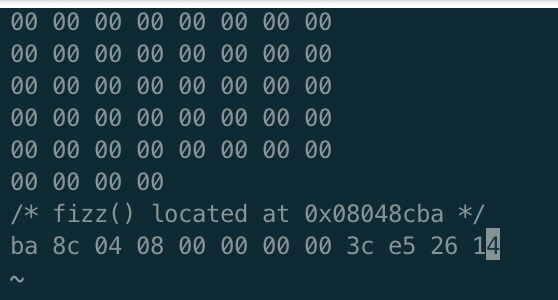


图3.2.2.3 fizz\_U201915035.txt文件内容

4.实验结果：

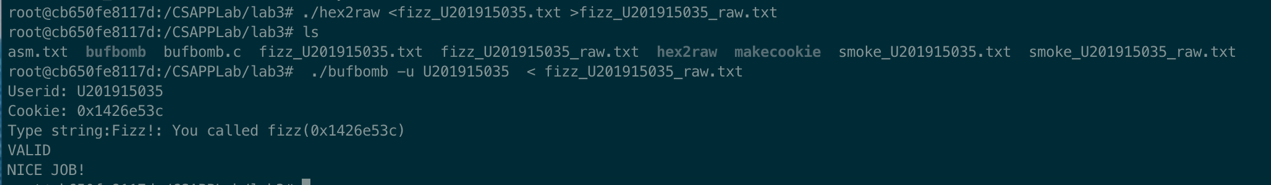


图3.2.2.4 fizz攻击成功截图

**3.2.3 阶段3 bang**

1.任务描述：

构造攻击字串，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：

利用objdump得到的反汇编代码推断出得缓冲区的长度，确定填充的字节数以及全局变量global\_value的地址。编写汇编代码将全局变量的内容修改为cookie值，并且利用覆盖返回地址的方式执行恶意代码。执行完恶意代码后转向bang函数。

3.实验过程：

观察bang函数的反汇编代码：

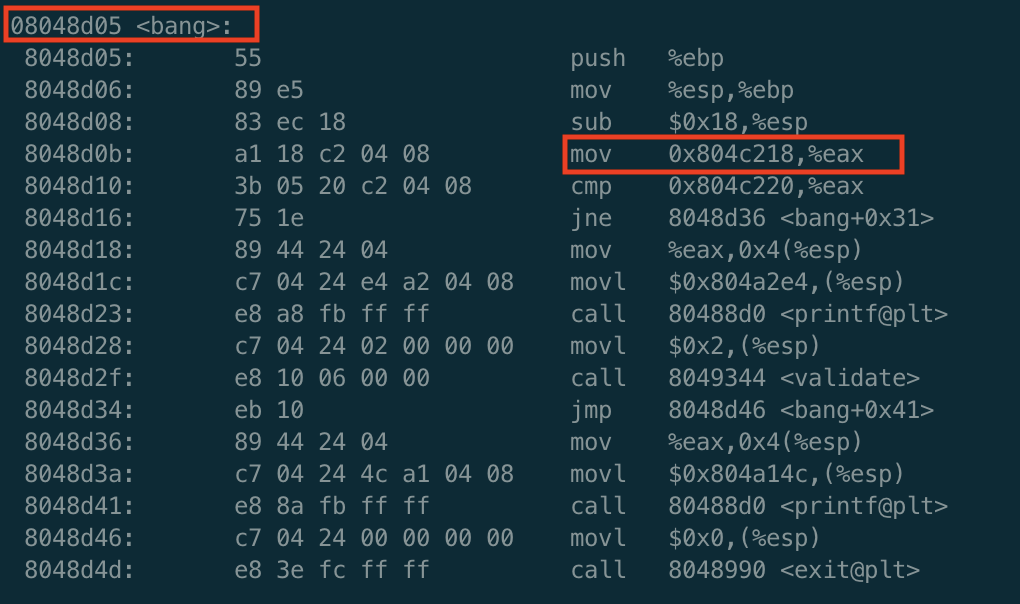


图3.2.3.1 bang函数反汇编代码

对比bang函数和上一个fizz函数可以发现0x804c220是全局变量中cookie所存放的位置。而另一个地址中0x804c218就是全局变量global\_value的地址，也是我们所要攻击的全局变量位置。

编写汇编代码并反汇编如下所示。

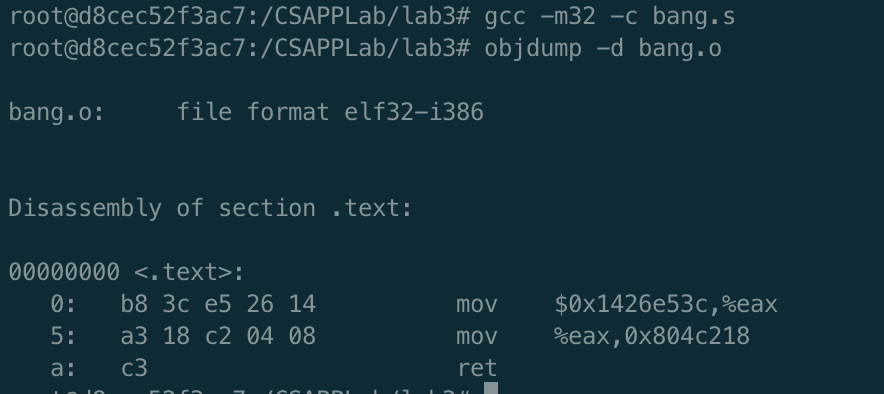


图3.2.3.2 反汇编结果

为了让程序能执行这段代码，就要得到这个代码首地址，并用之覆盖返回地址。而这段代码实质上是我们输入的字符串，保存在堆栈中，观察getbuf函数（见上图3.2.1.1），根据“lea -0x28(%ebp),%eax”以及“mov %eax,(%esp)”可以得出该地址应该是-0x28(%ebp)，利用gdb来得到ebp中的值。

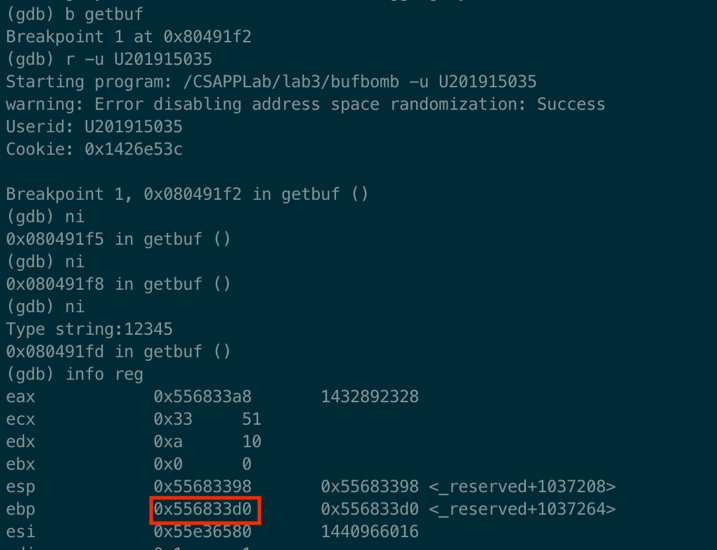


图3.2.3.3 ebp结果

所以返回地址应该是0x556833d0-0x28=0x556833a8，由此构建bang\_U201915035.txt文件内容，如下所示：

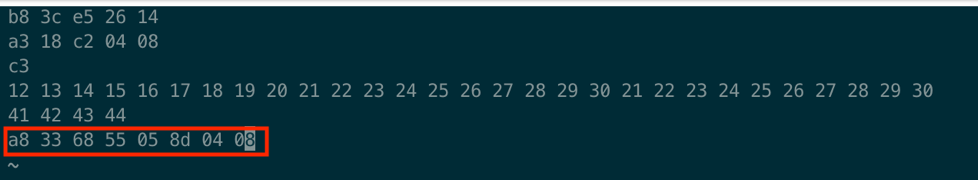


图3.2.3.4 bang\_U201915035.txt文件内容

注意这里我在堆栈返回地址0x556833a8后还加了一个地址0x08048d05，也就是bang函数的首地址，目的是执行到恶意代码ret时，把该地址送人EIP，让程序跳转到bang函数继续执行。

4.实验结果：

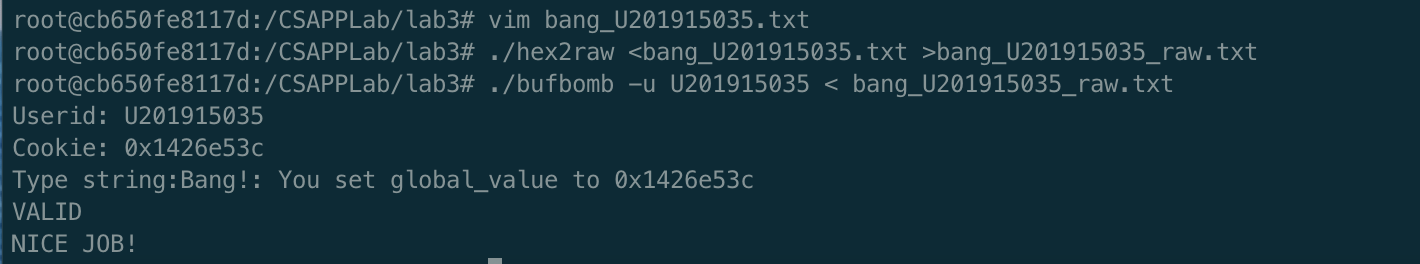


图3.2.3.5 bang攻击成功截图

**3.2.4 阶段4 boom**

1.任务描述：

Boom要求更高明的攻击，要求被攻击程序能返回到原调用函数test继续执行——即调用函数感觉不到攻击行为。 构造攻击字符串，使得getbuf都能将正确的cookie值返回给test函数，而不是返回值1。

2.实验设计：

编写汇编代码，让返回值为cookie值，并且还原ebp的值，再转向test函数，使之如同没有被攻击。

3.实验过程：

返回值为cookie的修改过程如上一阶段所示，不同的是，此时要返回test函数，使之如同未被攻击。那也就是要恢复ebp的值，同时不能影响其他内存值。

通过gdb来观察原ebp的值，如下所示：

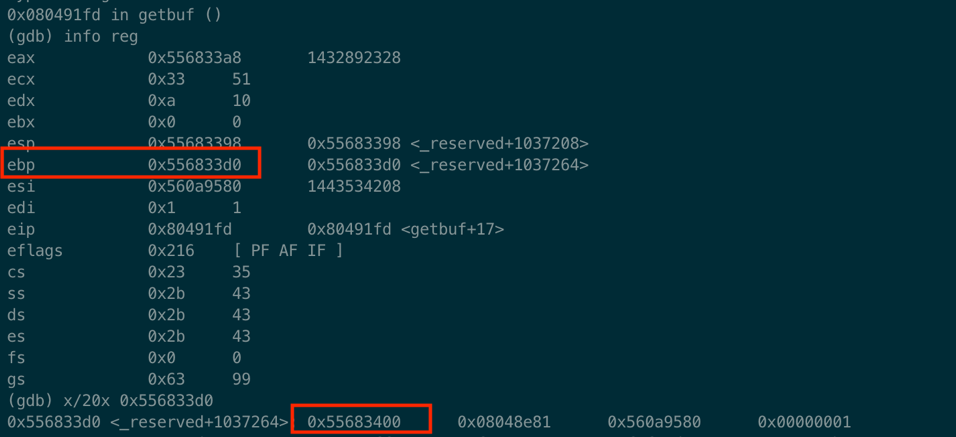


图3.2.4.1 原ebp观察截图

ebp指向地址中保存的值为原ebp的值，也就是0x55683400，我们让该部分内存为这个值。同时由于不能影响其他部分的地址，我们将汇编语言编写的部分改为如下所示：



图3.2.4.2 反汇编结果

向堆栈中push了地址0x8048e81，也就是要返回test函数中的地址。在执行ret时会把该值pop给EIP，那这个值是也是通过gdb观察得出的。我们可以推断，在原ebp上方（地址增加的方向）就是返回地址，当我们输入一个不会覆盖返回地址的字符串时，观察这个返回地址，就是我们执行完我们的代码后要返回的地址。gdb观察如下所示：

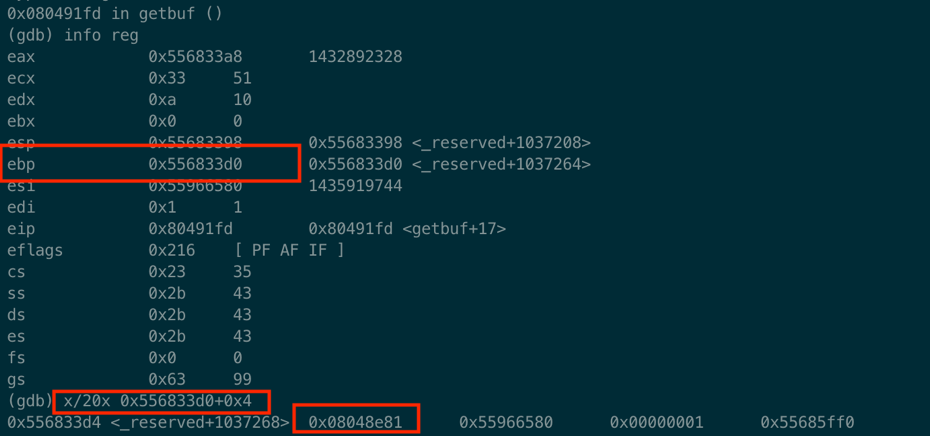


图3.2.4.3 返回地址的得到截图

由此构建了boom\_U201915035.txt文件，文件内容如下：

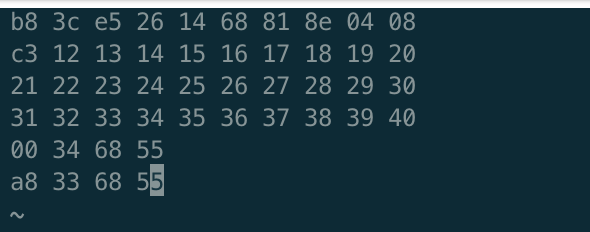


图3.2.4.4 boom\_U201915035.txt文件内容

4.实验结果：

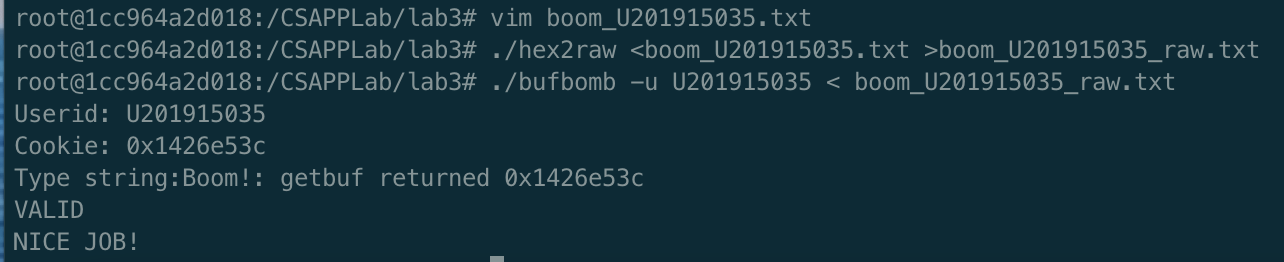


图3.2.4.1 boom攻击成功截图

**3.2.5 阶段5 nitro**

1.任务描述：

构造攻击字符串使getbufn函数，返回cookie值至testn函数，而不是返回值1。需要将cookie值设为函数返回值，复原被破坏的栈帧结构，并正确地返回到testn函数。其中的挑战在于5次执行栈（ebp）均不同，要想办法保证每次都能够正确复原栈帧被破坏的状态，并使程序能够正确返回到testn。

2.实验设计：

编写汇编代码，让返回值为cookie值，并且还原ebp的值，再转向test函数，使之如同没有被攻击。但是还原ebp的值时并不是使用一个固定的值，而是观察将要还原的ebp值和其他寄存器如esp之间的关系来达到目的。

3.实验过程：

查看getbufn函数：



图3.2.5.1 getbufn函数反汇编代码

观察可以得出所要占的字符串长度应为0x208也就是520个字节。

这个阶段最重要的挑战在于如何复原ebp的值，应为循环五次每一次的ebp都是不一样的，这时候我们需要观察ebp和别的寄存器esp之间的关系来得到确定的指令。观察testn函数：

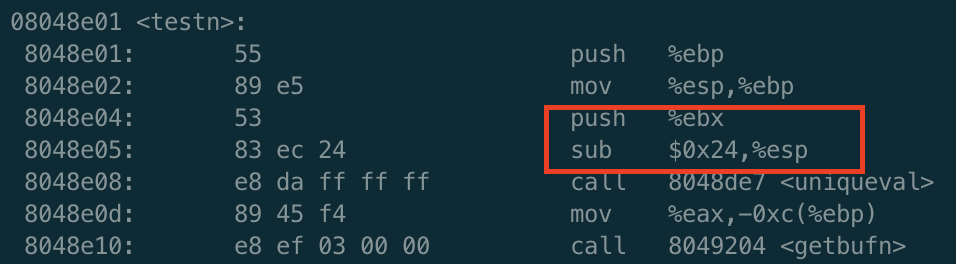


图3.2.5.2 testn函数反汇编代码

可以得出在原ebp基础上先push了一个ebx，也就是减了0x4个字节，再又减了0x24个字节，所以ebp=esp+0x28.由此编写汇编代码，其汇编代码如下所示：

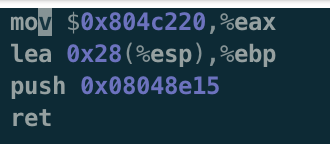


图3.2.5.3 汇编代码

而我们剩下的字节数我们需要用90，也就是nop指令的机器码来填充，也就是我们需要524-15=509个字节的90（15是我们写的代码的机器码字节数）。

其他返回地址的推断以及字符串堆栈地址的得到同上述阶段用gdb来得到即可，构建nitro\_U201915035.txt文件内容如下（省略了大量的90）：

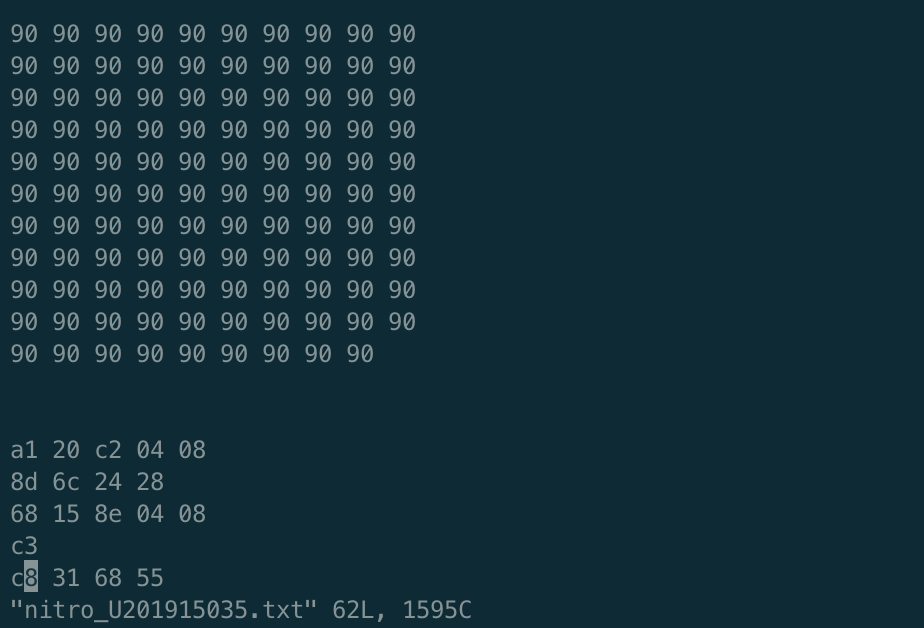


图3.2.5.4 nitro\_U201915035.txt文件内容

但是这样我们会得到的结果如下：

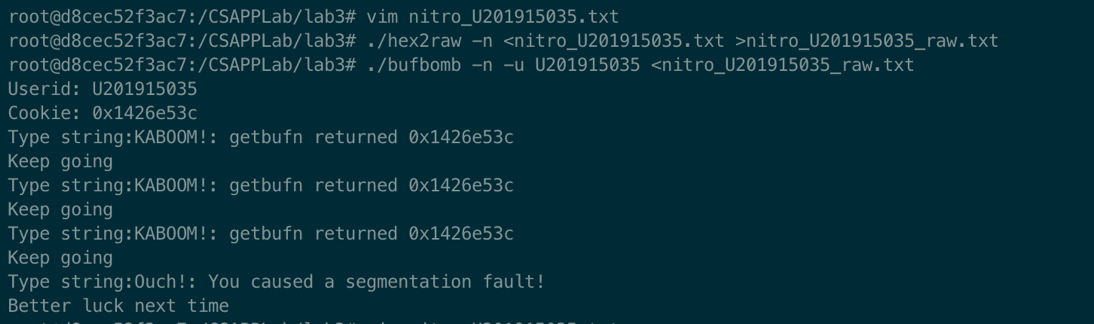


图3.2.5.5 nitro攻击失败截图

分析之后发现应该是堆栈返回地址的推断错误，我们利用gdb调试得到五次运行的值：

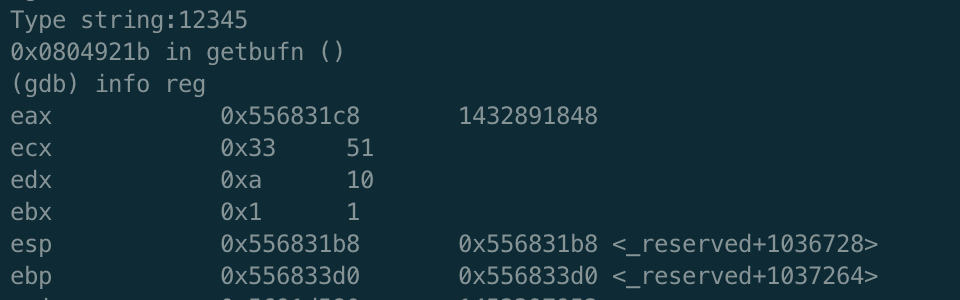


图3.2.5.6 gdb分析第一次进入getbufn

对应字符串地址为0x556831c8

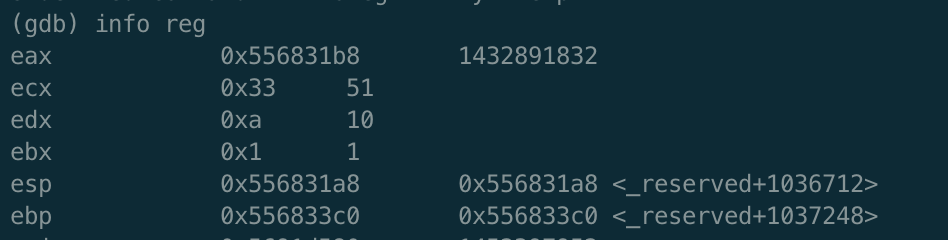


图3.2.5.7 gdb分析第二次进入getbufn

对应字符串地址为0x556831b8

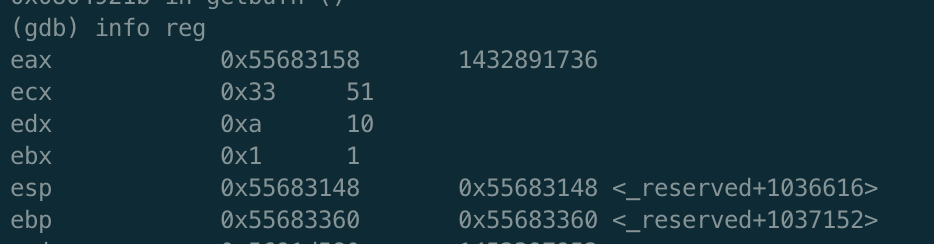


图3.2.5.8 gdb分析第三次进入getbufn

对应字符串地址为0x55683158

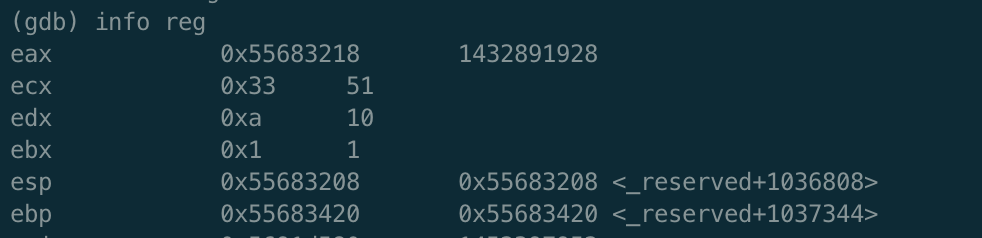


图3.2.5.9 gdb分析第四次进入getbufn

对应字符串地址为0x55683218

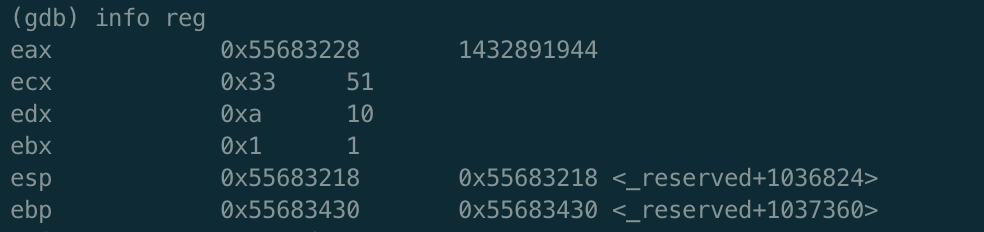


图3.2.5.10 gdb分析第五次进入getbufn

对应字符串地址为0x55683228

根据分析应该返回的堆栈地址是这几个地址中的最大值，也就是0x55683228，因为不能出现访问的地址内容是未定义的内容，这也是为什么要先写nop指令。

修改后的构建nitro\_U201915035.txt文件内容如下：

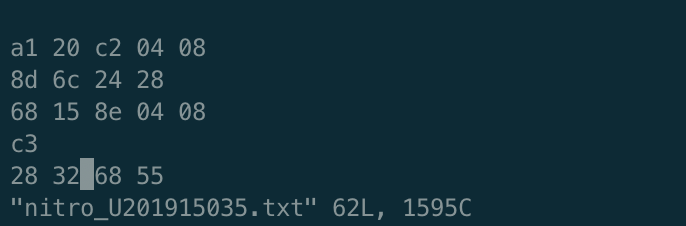


图3.2.5.11 修改后的nitro\_U201915035.txt文件内容

4.实验结果：

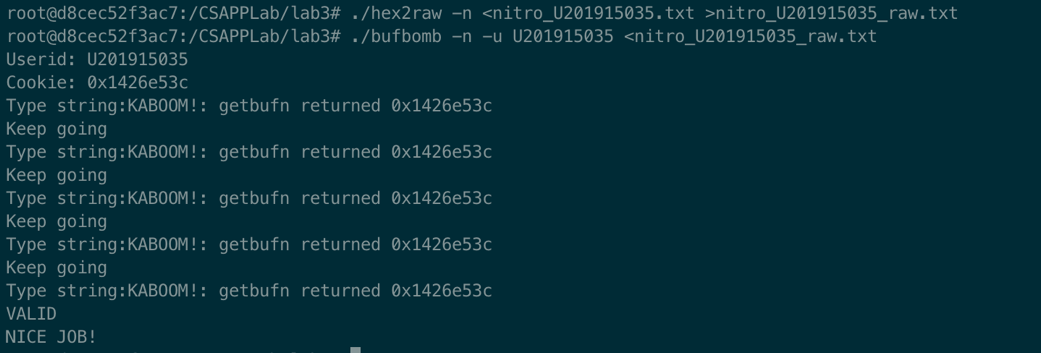


图3.2.5.12 nitro攻击成功截图

**3.3实验小结**

这一次实验的主题是缓冲区溢出攻击。整个实验的过程对我来说非常有趣。主要思想就是利用缓冲区溢出，将程序的运行转向原先设计好的函数或者自己新编写的函数之中。令我最惊叹的部分是可以执行写在堆栈里的机器指令，不过这也是处理之后的，因为其中一种防止代码注入攻击的方式就是把堆栈标记为不可执行。同时在第五个阶段我也了解到了开启了地址空间随机化的堆栈应该如何攻击。通过这次实验，我对于缓冲区的特点以及代码注入攻击有了更加深刻的认识。此外，我对开启了canary机制的程序应该如何攻击也有了兴趣，在课后我会去查一些资料来拓展知识。

# 实验总结

这门课的实验对我来说非常有意思，在实验的过程中，不断探索未知的东西，引起我的思索和好奇，让我对计算机系统底层的知识有了更加深入的了解。

在本次课程之前，我没有使用过Linux系统。在本次实验中，我是使用了docker配合i386/ubuntu来完成实验的，完全是在终端里的命令行操作，也是我第一次使用没有GUI的系统，但是整个过程也是十分流畅（在虚拟机上跑系统并不会有很好的体验），是一次全新的经验。在这门实验中我使用vim、gcc、gdb、objdump等工具，再用git配合完成文件传递。在经过3次实验之后，我对于Linux的工具和知识有了基本的了解和认识，比如能够熟练使用gdb中基础的命令。

在第一次实验之中，在对问题的反复思考中我开始对基本的一些位操作有了更加深刻的认识，比如说真正把～和！操作符区分开来，同时也明白了即使简单的操作符也能够组合实现复杂的逻辑功能，。

在第二次实验之中，我对之前很少接触的高级语言到汇编等底层语言的转化也有了更加清晰理解。我对高级语言里的变量、控制跳转、递归、循环等等在机器级别的表示的理解有了很大的提升，对汇编语言的掌握也更进了一步。

在第三次实验之中，我第一次了解到缓冲区溢出攻击这个概念，也了解到了三种相应的保护措施，并且我们对其中一种也就是地址空间随机化进行了攻击，让我对函数调用时堆栈里的变化以及esp、ebp的值的改变有了进一步的理解。同时也知道了在第二次实验中经常会看到的在函数起始时“mov %gs:0x14,%eax”“mov %eax,0x18(%esp)”和函数结尾的“xor %gs:0x14,%eax”这几条相关的命令的作用，因为我在第二次实验分析代码的时候就挺好奇这几个指令的意思，自己琢磨出来的是检验堆栈是否在函数结尾达到了和函数开头时相一致的状态。而在这次实验学习知识的过程中知道了这是canary机制，目的是防止这种缓冲区溢出攻击；但是在这次实验里我们并没有对这种机制进行攻击，应该是很难做到吧。

总的来说，这次实验为我打开了底层实现的大门，让我发现了其中的魅力，我认为这是真正对我学习计算机这门专业有意义的实验。