

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**专业班级： CS1906\_\_\_\_\_\_\_\_**

**学 号： U201915033\_\_\_\_**

**姓 名： 谭璇\_\_\_\_\_\_\_\_\_\_\_**

**指导教师： 胡侃\_\_\_\_\_\_\_\_\_\_\_**

**报告日期： 2021年 6月 22日**

**计算机科学与技术学院**

目录

[实验2: 3](#_Toc75410657)

[2.1 实验概述 3](#_Toc75410658)

[2.2 实验内容 3](#_Toc75410659)

[2.3 实验小结 13](#_Toc75410660)

[实验3: 13](#_Toc75410661)

[3.1 实验概述 13](#_Toc75410662)

[3.2 实验内容 13](#_Toc75410663)

[3.3实验小结 19](#_Toc75410664)

[实验总结 20](#_Toc75410665)

## 实验2:

### 2.1 实验概述

本实验的目的是更好地熟悉和掌握计算机中整数和浮点数的二进制编码表示。实验中，你需要解开一系列变成“难题”——使用有限类型和数量的运算操作实现一组给定功能的函数，在此过程中你将加深对数据二进制编码表示的了解。

本实验中,你要使用课程所学知识拆除一个“binary bombs”。一个“binary bombs”(二进制炸弹,下文将简称为炸弹)是一个Linux可执行 C 程序,包含了6个阶段 (phase1 phase6)。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。实验的目标是拆除尽可 能多的炸弹层次。六个层次如下：

• 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

实验语言：C;实验环境：Linux

### 2.2 实验内容

#### 2.2.1 阶段1 字符串比较

1.任务描述：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：根据反汇编代码一步一步分析，具体见实验过程。

3.实验过程：

将bomb反汇编输出到asm.txt文件中，在反汇编代码中查找phase\_1的位置：

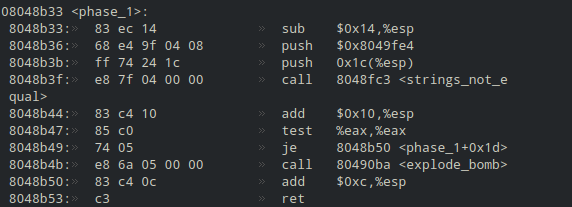


图2.2.1.1 phase\_1代码

从上面的语句中可以看出<strings\_not\_equal>所需要的两个变量是存在于%esp所指向的堆栈存储单元里。

在main()函数的汇编代码中，可以进一步找到：



图2.2.1.2 read\_line代码

这两条语句告诉我们%eax里存储的是调用read\_line()函数后返回的结果，也就是用户输入的字符串首地址，所以可以很容易地推断出和用户输入字符串相比较的字符串的存储地址为0x8049fe4，因为调用<strings\_not\_equal>前有语句：

8048b36: 68 e4 9f 04 08 push $0x8049fe4

可用gdb查看这个地址存储的数据：

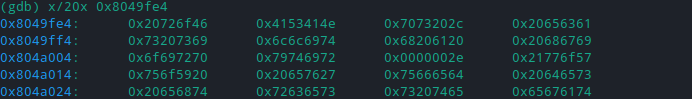


图2.2.1.3 存储数据

翻译过来之后就是：

For NASA, space is still a high priority.

4.实验结果：

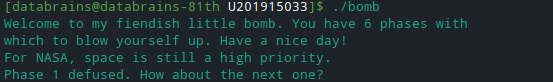


图2.2.1.4 实验结果

可见结果正确。

#### 2.2.2 阶段2 循环

1.任务描述：阶段二需要我们通过对于汇编中循环的分析来解析程序，从而推测这一阶段的解。

2.实验设计：观察分析phase\_2代码，使用gdb调试分析结果。

3.实验过程：由第一阶段可知，第二阶段应该从phase\_2开始。在反汇编代码中查找phase\_2的位置：

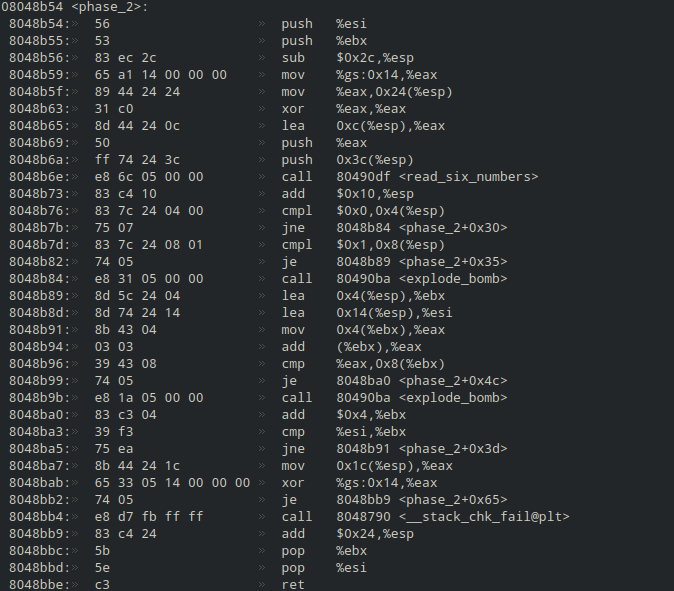


图2.2.2.1 phase\_2代码

由read\_six\_numbers可知是要输入6个数字，由：

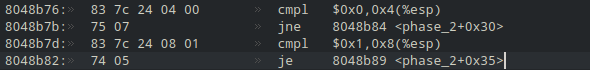


图2.2.2.2相关代码

可知，输入的第一个和第二个数字分别为0、1。观察下列代码可知，只有当输入的数为前两个数之和时才不会爆炸，故得到序列0、1、1、2、3、5。

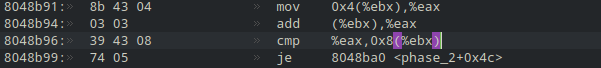


图2.2.2.3相关代码

4.实验结果：

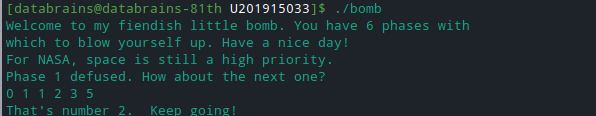


图2.2.2.4实验结果

可知结果正确。

#### 2.2.3 阶段3 条件/分支

1.任务描述：阶段三考察对于条件与分支判断的汇编语言解析，需要通过解析汇编条件判断的结构来拆除第三层。

2.实验设计：观察分析phase\_3代码，使用gdb调试分析结果。

3.实验过程：找到phase\_3代码如下：

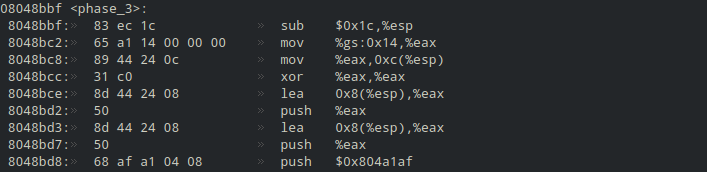


图2.2.3.1 phase\_3部分代码

由gdb观察0x804a1af地址的内容：



图2.2.3.2 0x804a1af地址内容

可知输入的是数字、数字。

由以下代码可知：



图2.2.3.3 phase\_3部分代码

输入的第一个数字一定小于7。

假设输入的第一个数为1,即(%eax)=1,所以：



图2.2.3.4 gdb调试结果

而地址为0x8048c05的内容为：8048c05: b8 21 03 00 00 mov $0x321,%eax

故第一个数为1,第二个数为0x321=(801)d。

4.实验结果：

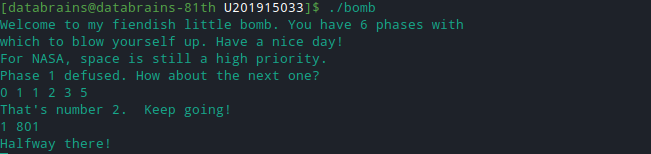


图2.2.3.5 实验结果

可知结果正确。

#### 2.2.4 阶段四 递归

1.任务描述：通过找到反汇编程序中的递归规律，确定输入的参数，从而破解炸弹。

2.实验设计：分析phase\_4和func4的代码，通过gdb调试，确定输入的参数。

3.实验过程：

观察主流程，通过8048ceb: 83 f8 02 cmp $0x2,%eax 发现参数个数为2。

通过8048cf0: 83 7c 24 04 0e cmpl $0xe,0x4(%esp)发现第一个参数不能大于14。

通过8048d14: 83 7c 24 08 15 cmpl $0x15,0x8(%esp)发现第二个参数为21。

而通过



图2.2.4.1 phase\_4部分代码

发现，将第一个参数带入func4之后，返回值为21。

用gdb查看0x804a1af可知，输入的参数为两个数字，如下图所示：



图2.2.4.2 0x804a1af代码查看结果

观察func4函数，用C语言编写相应的代码段：

#include<stdio.h>  
​  
int func(int o,int p,int q);  
int dx,ax,cx;  
int main(){  
    int a;  
    int i;  
    for(i=0;i<15;i++){  
        a = func(i,0,14);  
        printf("输入%d -> func输出%d\n",i,a);  
   }  
    return 0;  
}  
int func(int o,int p,int q){  
    int bx,si;  
    dx = o;  
    si = p;  
    cx = q;  
    ax = cx;  
    ax -= si;  
    bx = ax;  
    bx >>= 31;  
    ax += bx;  
    ax >>= 1;  
    bx = ax+si;  
    if(bx<=dx){  
        ax = bx;  
        if(bx>=dx){  
            return ax;  
       }  
        else{  
            ax = bx+1;  
            func(dx,ax,cx);  
            ax += bx;  
       }  
   }else{  
        ax = bx-1;  
        func(dx,si,ax);  
        ax+=bx;  
   }  
    return ax;  
}

运行程序可知，当输入的参数为6时，输出为0,如下图所示：



图2.2.4.3 0x8函数输出结果

由此可见，拆弹密钥为6 21。

4.实验结果

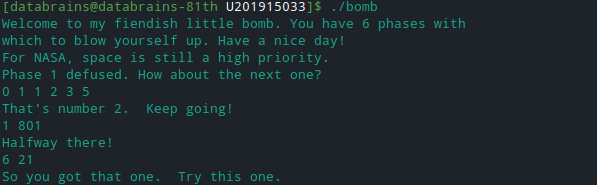
向炸弹中输入6 21，拆弹成功，如下图所示。

图2.2.4.4 实验结果

可知结果正确。

#### 2.2.5 阶段5 指针

1.任务描述：第五阶段最要考察对于指针的分析。通过分析汇编代码语句对应的 C 语言指针操作来对 炸弹进行拆除。

2.实验设计：此阶段实验与指针相关，又根据静态调试跟踪可知，需借助gdb的动态调试来查找相关地址中存放的数据的值，进而分析出最终的拆弹密码。

3.试验过程：首先跳转到 phase\_5 的代码处，代码如下：

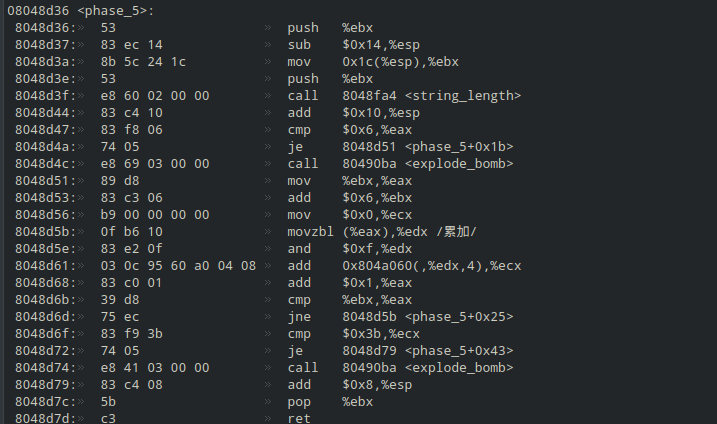


图2.2.5.1 phase\_5实验代码

对汇编代码进行分析，0x1c(%esp)处保存的是输入的数据，发现首先调用了string\_length函数，来查看输入的数据有几位。通过代码cmp $0x6,%eax发现，输入的数据为6位。具体代码如下图所示：



图2.2.5.2 string\_length调用代码

对接下来的代码进行分析，通过以下代码：

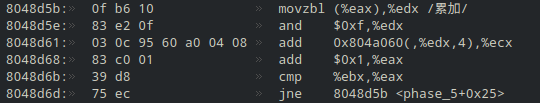


图2.2.5.3 循环部分代码

可知，在该部分进行了循环。(%eax)保存了输入的字符，通过movzbl (%eax),%edx 和 and $0xf,%edx两句代码可知，%edx保存了输入字符的ASCII码的二进制码的低4位，而通过add 0x804a060(,%edx,4),%ecx可知，将这低四位作为偏移量，也可以理解为数组下标。

而通过gdb调试可知，0x804a060及其之后地址保存的数据如下：



图2.2.5.5 循环部分代码

通过代码cmp $3b,%ecx可知，六次累加的结果为3bh，即偏移量为14\*4，10\*4，12\*4，8\*4，2\*4，6\*4时，累加数据为15，14，11，3，6，9，其累加和为3bh。

故可以选择ASCII码为4E，4A，4C，48，42，46的六个字符。

4.实验结果

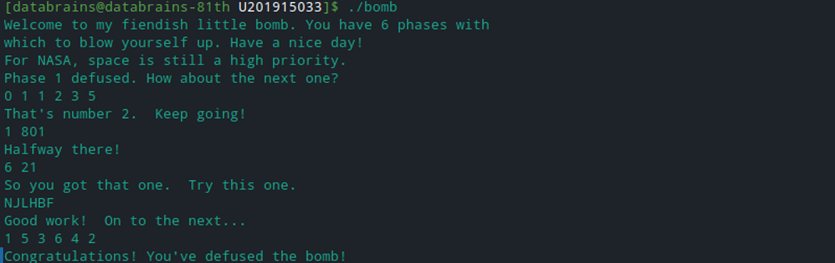


图2.2.5.6 循环部分代码

可见结果正确。

#### 2.2.6 阶段6 链表

1.任务描述：实验6较为复杂，要求通过对于汇编代码中的结构/链表操作进行分析并得出第六阶段的 结果，对于结构体在内存中的存储方式进行了考察。

2.实验分析：初步静态分析，此阶段代码有些过长，所以单纯通过静态调试跟踪有些困难，因此我首先找到几个循环体，通过静态调试跟踪和动态调试跟踪的方法来确认循环体的功能，最后将循环体结合起来，拆除此阶段炸弹。

3.试验过程：

首先，粗略的对于代码进行分段，将代码分为五个部分并逐个进行分析。

第一部分：调用 read\_six\_numbers 读入六个数字。进行循环，将每一个数减一以后与 5 进行比较，要求 ⩽ 5，推断出读入的六个数中每一个都 ⩽ 6。进行第二个循环，这是一个二重循环，检测每一个数是否相等，如果不相等则炸弹爆炸，推测出读入的六个数两两不相等。相关代码如下图所示：

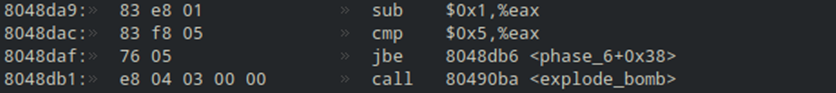


图2.2.6.1 phase\_6部分代码

第二部分：进行一个循环，对于读入的每一个数进行处理，将 7-< 读入的数 > 存入一个数组 的对应位置中，此列表在栈空间中并与读入的六个数的位置紧邻。如下图所示。

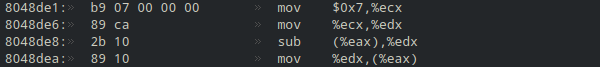


图2.2.6.2 phase\_6部分代码

第三部分：这一部分较为复杂，也是较为重要的一部分。首先对其进行观察，发现它涉及到地址 0x804c154，从这个地址开始反汇编，结果如下图所示。

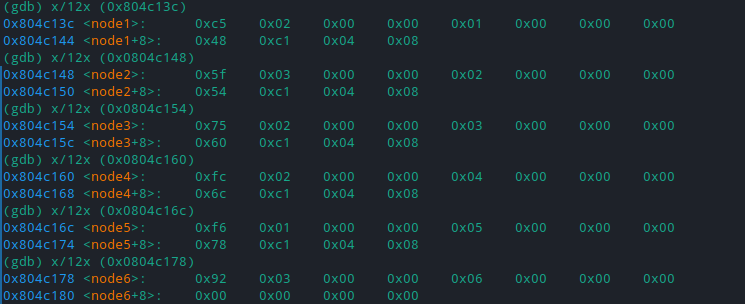


图2.2.6.2 反汇编结果

显然，这是一个链表，链表中的每一个节点为一个结构，占12 个字节，且第三个成员为 next 指针。经过分析后发现，这一部分进行的操作为对第二部分中的数组中的元素进行赋值。

第四部分：对于第三部分的排序后的链表进行了重新的链接。

第五部分：对于重新链接的链表进行了检查，要求保证链表从首至尾是降序有序的。整合上面的信息，输入的数据应该是对于原链表中的数据02c5、035f、02fc、0275、01f6、0392进行排序，并用 7 减去排序后的索引，然后进行倒序的结果。推算出应为1 5 3 6 4 2，将 “1 5 3 6 4 2” 输入炸弹。

4.实验结果

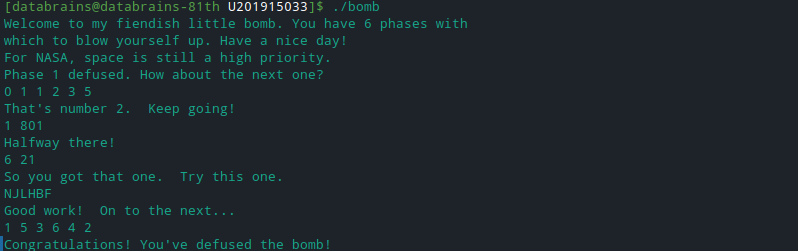


图2.2.6.4 实验结果

可知结果正确。

### 2.3 实验小结

本次实验熟悉了obj、gdb的各种操作，我对C语言如何转换为汇编有了更深的理解。对数据在计算机中的存储有了更加清晰的认识，更加加深了对汇编代码的理解，对循环、分支、数组、指针结构在机器内部的存储有了更深刻的认识。

## 实验3:

### 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。

### 3.2 实验内容

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。

#### 3.2.1 阶段1 smoke

1.任务描述：将getbuf函数执行return后执行test函数改为执行smoke函数

2.实验设计：根据反汇编代码求得buf的长度，确定填充的字节数，然后将smoke函数的首地址填入buf的末尾。

3.实验过程：首先通过gdb工具对bufbomb文件进行反汇编，查看getbuf的汇编代码：

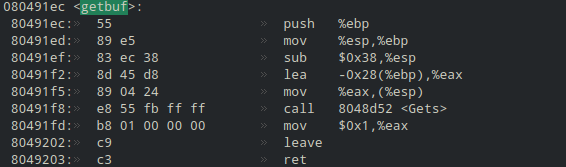


图3.2.1.1 getbuf代码

可见lea把buf的指针地址-0x28(%ebp)传给了Gets(),所以buf离返回地址有0x28+4=44字节的距离，因此只要在buf出开始填44个字节的非‘\n’数。

同时，也可以得到smoke函数的首地址为0x8048c90，如下图所示：

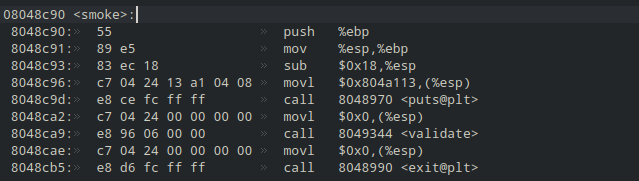


图3.2.1.2 smoke代码

此时，只需要构造一个长度为44+4字节，最后四个字节为smoke函数的起始地址的字符串即可，所以输入的字符串为：

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

4.实验结果：

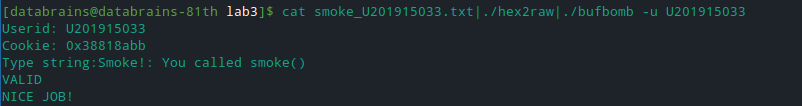


图3.2.1.3 实验结果

可见结果正确。

#### 3.2.2 阶段2 fizz

1.任务描述：将getbuf函数执行return后执行test函数改为执行fizz函数。

2.实验设计：跟阶段1类似，多了一个比较cookie的环节，所以要把cookie填入相应地址。

3.实验过程：查看fizz的汇编代码，如下图所示：

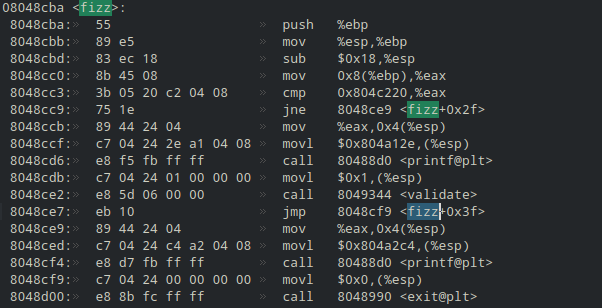


图3.2.2.1 fizz代码

可知val变量存储地址为fizz函数中的0x8(%ebp),而fizz函数在开始的地址为08048cba,所以输入的前44字节为非’\n’任意值，第45-58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，也就是bb 8a 81 38，所以输入的56个字节为

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 00 00 00 00 bb 8a 81 38

4.实验结果：

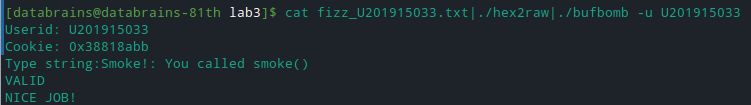


图3.2.2.2 实验结果

可见结果正确。

#### 3.2.3 阶段3 bang

1.任务描述：将getbuf函数执行return后执行test函数改为执行bang函数。

2.实验设计：找到cookie的地址和个globle\_value的地址，将globle\_value的值改为cookie值，再使函数成功跳至bang执行。

3.实验过程：通过观察bang函数可以得到cookie以及global\_value的地址分别为0x804c220和0x804c218，如下图所示：

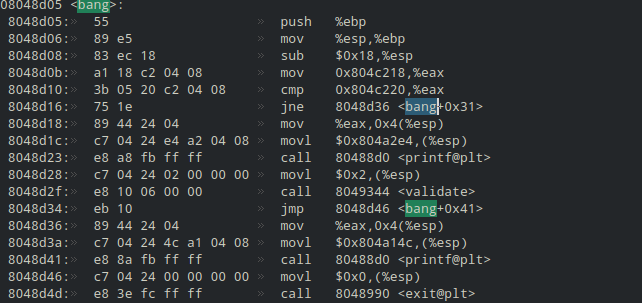


图3.2.3.1 bang代码

编写汇编代码，修改global\_value的值为cookie。如下图所示：



图3.2.3.2 bang\_asm代码

将这3行代码保存至example.s文件中，进行汇编和反汇编，查看结果如下图所示：

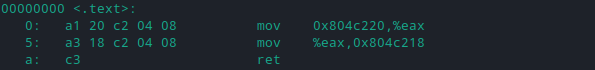


图3.2.3.3 反汇编bang\_asm.o代码

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

通过gdb调试，在getbuf函数中设置断点，如下图所示：

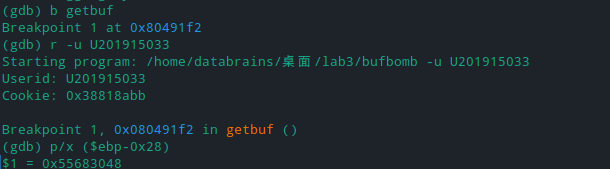
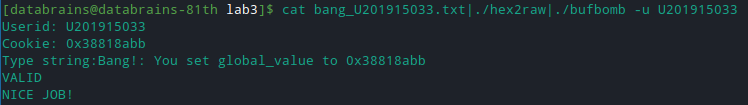


图3.2.3.4 查看地址

可以得到字符串首地址为0x55683358，综上所述，最终的攻击字符串应该为反汇编得到的指令，加上填充值(0)，最后加上字符串的首地址(0x55683358)，即a1 20 c2 04 08 a3 18 c2 04 08 c3 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 48 30 68 55 05 8d 04 08

4.实验结果

图图3.2.3.5 实验结果

可见结果正确。

#### 3.2.4 阶段4 boom

1.任务描述： getbuf照常返回至test，但是返回值改为cookie值。

2.实验设计：返回test函数时的eax要赋值为cookie值，还要恢复被覆盖的ebp值。

3.实验过程：

观察test函数，查看调用getbuf函数之后的返回值位置，如下图所示：

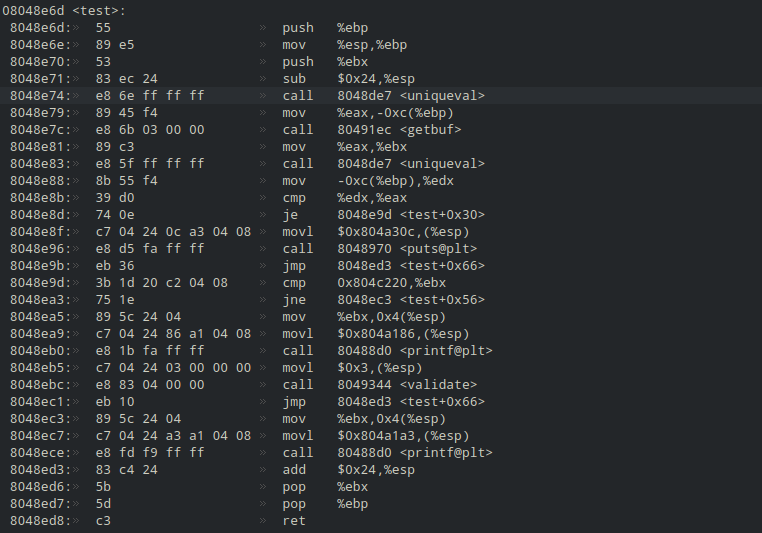


图3.2.4.1 test代码

要返回至test,应该返回至调用getbuf之后的一步，地址为0x0848e81，在0x8048e81处设置断点，观察正常情况下的ebp值为：



图3.2.4.2 ebp内容

可以得到正常情况下的ebp值为0x556833b0，综上所述，编写汇编代码：

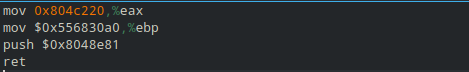
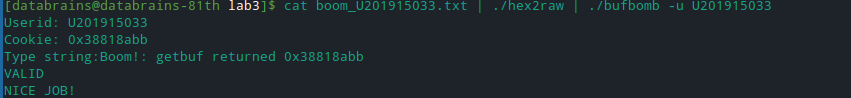


图3.2.4.3 boom\_asm代码

由此可以得到，攻击字符串为反汇编指令+填充值+字符串首地址，即

a1 20 c2 04 08 bd a0 30 68 55 68 81 8e 04 08 c3 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 48 30 68 55

4.实验结果

图3.2.4.3 实验结果

可见结果正确。

### 3.3实验小结

本次实验主要是切身实践了缓冲区溢出带来的种种有趣的现象，现在更加理解了数在机器内部的存储结构。

## 4 实验总结

后PC时代的到来，使得原先基于PC而建立起来的专业教学内容已经远远不能反映现代社会对计算机专业人才的培养要求，原先计算机专业人才培养强调“程序”设计也变为更加强调系统设计。

目前国内大学计算机专业课程设置，大多是按照计算机系统层次进行横向切分，每门课都仅限于在本层抽象，相互之间几乎没有关联，因而学生对整个计算机系统的认识过程就像瞎子摸象一样，很难形成对完整计算机系统的全面认识，本次实验脱胎于CMU的CSAPP实验，在我看来，这是国内计算机本科教育的一次进步。

世人常说北美IT，天下第一，原因有很多。不光是因为美国是现代计算机的起源之地，更重要的是美国名校对于计算机学科的教育理解是非常深刻的。如CMU的CSAPP、MIT的SICP、Stanford的CS144。作为一个想本科毕业参加工作的大学生，我觉得认真学习上述课程是非常有必要的。

这门课的实验相比其他的学科的实验更加具有趣味，不仅仅是实验本身，更有实验过程之中对未知的探索。在实验过程之中，收获了很多，也发现了自己的许多不足。尽管其间失败了无数次，但把全部炸弹都拆完后还是充满了成就感。感觉这份作业还是很有创意的，使我们对工具的掌握以及汇编级编程的能力都得到很好锤炼。

在此，我要感谢胡老师设计了一门这样的课程，深入浅出地为我们讲解计算机系统基础！感谢助教在我们学习过程中的付出！