

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**专业班级： CS1903**

**学 号： U201914983**

**姓 名： 王家顺**

**指导教师： 张宇**

**报告日期： 2021年 6月 24 日**

**计算机科学与技术学院**

目录

[实验2：Binary Bombs 3](#_Toc75427443)

[2.1 实验概述 3](#_Toc75427444)

[2.2 实验内容 3](#_Toc75427445)

[2.2.1 阶段1 字符串比较 3](#_Toc75427446)

[2.2.2 阶段2 循环 4](#_Toc75427447)

[2.2.3 阶段3 条件/分支 5](#_Toc75427448)

[2.2.4 阶段4 递归调用和栈 6](#_Toc75427449)

[2.2.5 阶段5 指针 7](#_Toc75427450)

[2.2.6 阶段6 链表/指针/结构 8](#_Toc75427451)

[2.3 实验小结 11](#_Toc75427452)

[实验3：缓冲区溢出攻击 12](#_Toc75427453)

[3.1 实验概述 12](#_Toc75427454)

[3.2 实验内容 12](#_Toc75427455)

[3.2.1 阶段1 smoke 12](#_Toc75427456)

[3.2.2 阶段2 Fizz 13](#_Toc75427457)

[3.2.3 阶段3 Bang 14](#_Toc75427458)

[3.2.4 阶段4 boom 15](#_Toc75427459)

[3.2.5 阶段5 nitro 16](#_Toc75427460)

[3.3 实验小结 19](#_Toc75427461)

[实验总结 20](#_Toc75427462)

# 实验2：Binary Bombs

## 2.1 实验概述

**实验目的**

加深对AT&T汇编格式的理解和熟悉gdb等工具的使用。

**实验内容**

利用gdb，objdump等工具，跟踪调试每一阶段的机器代码，理解汇编语言代码的行为和作用，推断出拆除炸弹所需的目标字符串

## 2.2 实验内容

### 2.2.1 阶段1 字符串比较

**1. 任务描述**

找到炸弹的第一个字符串。

**2. 实验设计**

在反汇编代码中找到用于比较的字符串

**3. 实验过程**

在main函数中首先调用了phase\_1函数，则此函数即为判断第一个字符串的函数。

phase\_1函数中，调用了strings\_not\_equal函数，随后根据返回值判定是否引爆炸弹，则该函数功能为比较字符串。观察到0x804a0e4为该函数的一个参数，用gdb查看该单元内容，得到一个字符串，即为第一阶段的答案。



图 2.1 main函数中调用了phase\_1函数

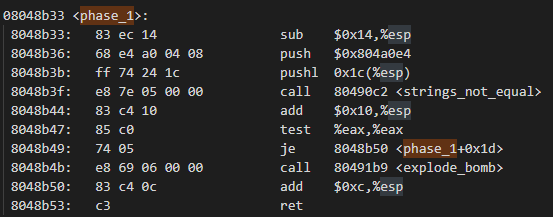


图 2.2 phase\_1中利用一个函数的返回值决定是否引爆炸弹



图 2.3 用gdb解析得到一个字符串

**4. 实验结果**

将得到的字符串进行测试，结果正确。

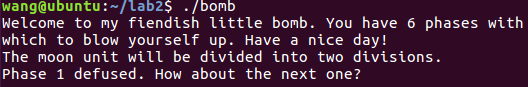


图 2.4 阶段1测试结果

### 2.2.2 阶段2 循环

**1. 任务描述**

找到炸弹程序的第二个字符串。

**2. 实验设计**

分析带有循环结构的phase\_2函数，得到正确的答案。

**3. 实验过程**

在函数的开头，先调用读取函数，读取输入的6个数字。然后比较第一个参数与0的大小关系。在循环体中，ebx和eax从零开始，在每次循环中，eax值加上ebx值，随后ebx值加1，与下一个参数进行比较，当每次比较结果都相等时循环结束，判定结果正确。根据上述分析，每次比较的eax值依次为0，1，3，6，10，15，即为本阶段答案。



图 2.5 先调用函数读取6个数字

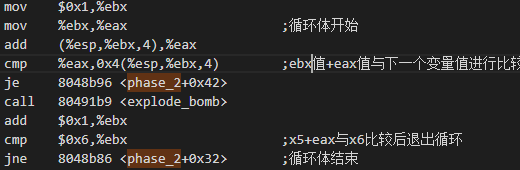


图 2.6 循环体的内容

**4. 实验结果**

将得到的答案进行测试，结果正确。

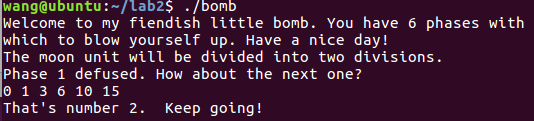


图 2.7 阶段3的测试结果

### 2.2.3 阶段3 条件/分支

**1. 任务描述**

找到炸弹程序的第三组字符串

**2. 实验设计**

分析和理解phase\_3中分支程序的结构，得出正确的结果。

**3. 实验过程**

实验开头调用了scanf函数，用gdb分析传递的参数可知，答案为数字-字符-数字。

根据跳转指令，跳转到的位置取决于第一个参数的值，用gdb可得地址0x804a160及其后64个字节的内容（按4字节数解析），一共有8个有效地址，这8个有效地址均在phase\_3函数中，每个地址后都对应着对第二个、第三个参数的分比较。可知，阶段3的密码共有8组，分别为0，t，746；1，t，564；2，m，981；3，n，115；4，m，216；5，c，982；6，q，984；7，k，852。



图 2.8 scanf接受的参数



图 2.9 跳转指令部分

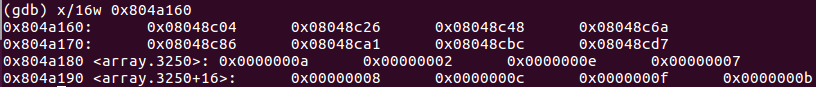


图 2.10 0x804a160处共有8个有效地址

**4. 实验结果**

分别对上述8组答案进行测试，结果均正确



图 2.11~2.18 8组答案的测试结果

### 2.2.4 阶段4 递归调用和栈

**1. 任务描述**

找到炸弹程序的第四组字符串

**2. 实验设计**

分析phase\_4和func4中的递归程序，得到正确的答案。

**3. 实验过程**

在phase\_4中，用gdb读取scanf的参数，得到输入值为两个数字。

由后续代码可知，函数phase\_4调用了func4函数，传递了参数0x0和0xe，得到返回值为5；且答案的第二个参数为5。

对函数func4进行分析可知，func4实现的是二分查找的过程。二分查找的high值存储于esi中，low值存储于ebx中，目标值为ecx，当前查找到的值存储与edx中，eax的值为（high-low）/2。若edx>ecx，返回值为2eax；若edx<ecx，返回值为2eax+1；若edx=ecx，则返回值为0。再根据最终返回值为5，可断定eax中值经过0-1-2-5的变化过程，通过反推可知，答案的第一个参数为10。最终确定答案的两个参数为5，10。



图 2.19 scanf获得的参数

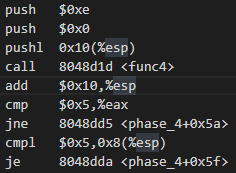


图 2.20 phase\_4函数对两个参数的处理

**4. 实验结果**

将得到的答案进行测试，可知答案正确。



图 2.21 阶段4的测试结果

### 2.2.5 阶段5 指针

**1. 任务描述**

找到炸弹程序的第5组字符串。

**2. 实验设计**

分析phase\_5中的指针调用，得到正确的答案。

**3. 实验过程**

在phase\_5中，用gdb读取到scanf的参数，可知答案为两个数字。

在调用scanf后，phase\_5将第一个参数的低4位保留，高28位置零，同时当处理后的值为0xf的时候引爆炸弹。故第一个参数为低4为某个取值的任意数。

接下来进入一个循环，ecx存储所有eax值的和，edx统计循环次数，eax取值为地址（0x804a180+4\*eax）中的值。当循环次数为15次且最终eax结果为15时，所得ecx值即为第2个参数的值。

根据循环体内的规律，可推导出eax的取值依次为5，12，3，7，11，13，9，4，8，0，10，1，2，14，6，15，其中第一个5未被计入ecx中，则可知第二个参数值为 115，而第一个参数为低4位为1001的任意值。



图 2.22 scanf得到的参数

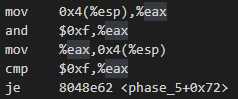


图 2.23 phase\_5对第一个参数的处理

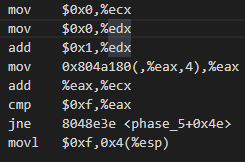


图 2.24 循环体的内容

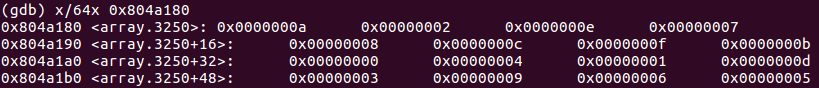


图 2.25 0x804a180后的值

**4. 实验结果**

任取符合条件的一组值进行测试，结果正确



图 2.25 阶段4的测试结果

### 2.2.6 阶段6 链表/指针/结构

**1. 任务描述**

找到炸弹程序的第六组字符串

**2. 实验设计**

分析phase\_6中对链表结构的操作，得到正确答案。

**3. 实验过程**

根据程序调用的read\_six\_numbers函数可知，本阶段答案为6个数字。

随后程序进入一个二重循环，其功能为，判断每个数字是否都小于6，且判断各个数字是否两两不相等。

接着，程序再次进入一个循环，其功能为，将每个输入值对7求补，并替换原来的输入值。

分析下一步之前，观察到程序中出现了一个地址，故用gdb对其进行解析。根据结果，从此处开始是6个由两个数字、一个地址组成的结构体。

由上可知，接下来程序进入一个循环，其功能为按照输入变量的顺序，将保存在存储单元中的一个链表重新排列，并将排序后的首地址保存在堆栈中，6个变量之后。随后程序通过循环，将每个链表的指针域，按照堆栈中排列的顺序指向下一个结点，并将最后一个结点指向0。这两步是建立按输入值顺序排列的一个新链表。

最后，程序开始进行比较，确定新链表中的第一个值是否是按照降序排列如果是，则检查通过。

根据链表第一个变量降序排列的规则，可得输入值为4，2，5，6，1，3，考虑到这些数输入变量对7求补后的结果，原始的输入结果应为3，5，2，1，6，4。

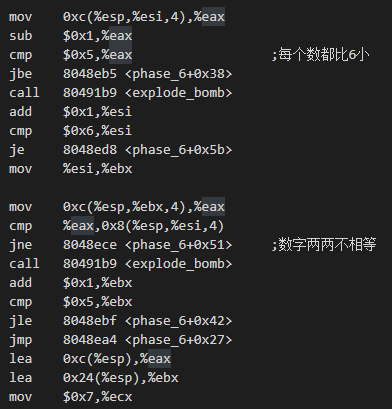


图 2.26 判断输入格式的二重循环

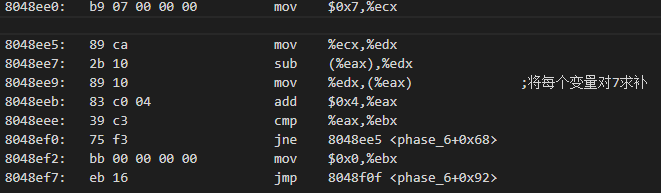


图 2.27 将每个变量对7求补的循环结构

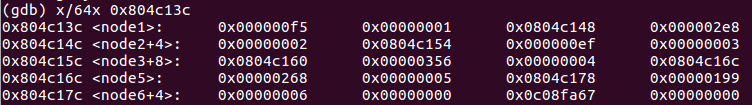


图 2.28 在gdb中查看链表

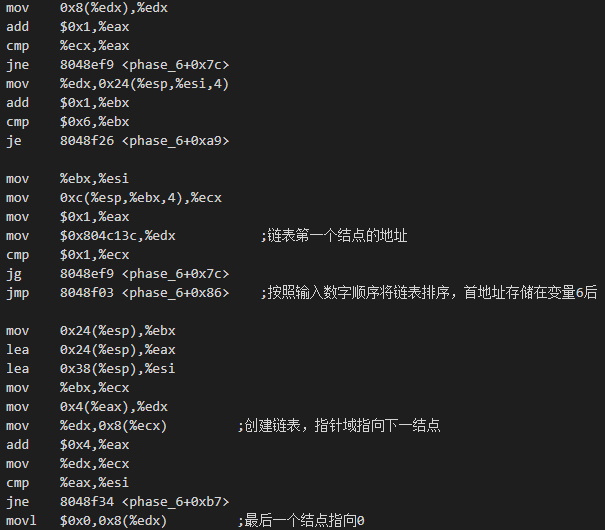


图 2.29 建立新链表

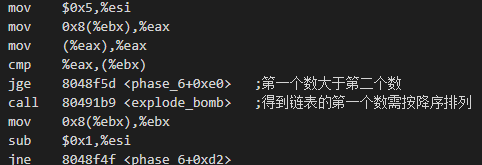


图 2.30 最后对输入值的正确性进行判断

**4. 实验结果**

将得到的答案进行测试，可知结果正确。



图 2.31 阶段6的测试结果

## 2.3 实验小结

本次实验为拆弹实验，即利用反汇编等技术分析程序比对输入值的操作，从而推断出正确的输入字符串。由于任务书中已给出分析的方向，因此在实验中采取的主要方法为静态分析，即分析反汇编代码和利用gdb读取存储单元来分析程序的功能。本次实验让我了解到破解程序的一些方法，也让我认识到linux系统下丰富的工具为程序破解跟踪提供的巨大方便。

# 实验3：缓冲区溢出攻击

## 3.1 实验概述

**实验目的：**

加深对IA-32函数调用规则和栈帧结构的理解

**实验内容：**

对目标程序实施缓冲区溢出攻击（buffer overflow attacks），通过造成缓冲区溢出来破坏目标程序的栈帧结构，继而执行一些原来程序中没有的行为。

## 3.2 实验内容

### 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getbuf函数调用的gets函数不判断字符串长度的特性，可利用gets函数造成的溢出，改变ebp对应存储区，即返回地址的值来使程序跳转至smoke处。

**3. 实验过程**

根据上述思路，可得到如图3.1所示的攻击代码，利用hex2raw将其转换后进行攻击测试。

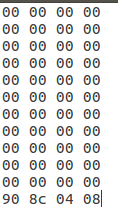


图 3.1 smoke的攻击代码

**4. 实验结果**

利用得到代码进行测试，结果正确。

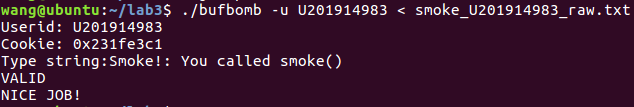


图 3.2 smoke的测试结果

### 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。

**2. 实验设计**

根据分析可知，在fizz函数中，ebp保存着函数调用前的ebp值，ebp+4保存着返回地址，则fizz函数的参数位于ebp+8处。

**3. 实验过程**

可以构造攻击字符串如图所示，其中0x08048cba是fizz函数的地址，0x231fe3c1是传递的cookie参数。

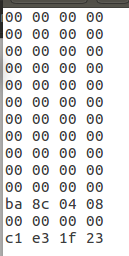


图 3.3 Fizz的攻击代码

**4. 实验结果**

使用攻击指令进行测试，结果正确。

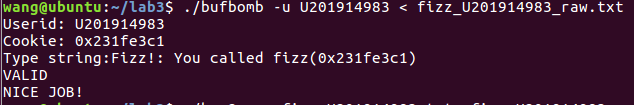


图 3.4 Fizz的测试结果

### 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，同时将程序中的全局变量global\_value的值篡改为cookie的值。

**2. 实验设计**

Getbuf函数会将读取到的字符串存储到一个缓冲区中，我们可以利用smoke中的方法使程序跳转至该缓冲区中，同时在缓冲区中填写相应的指令以达成目的。

**3. 实验过程**

利用objdump -D指令反汇编得到的反汇编代码（包含全局变量），可得到global\_value的地址为0x0804c218。通过push待跳转到的地址和ret指令，可使程序跳转至bang函数中。据此可得插入getbuf缓冲区中的指令。利用gcc将指令汇编的方法可得到其对应的机器指令。用这些机器指令替换smoke中开头部分的00即得攻击字符串。



图 3.5 变量global\_value的地址



图 3.6 bang的攻击代码

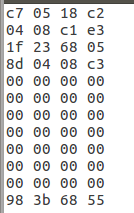


图 3.7 bang的攻击指令

**4. 实验结果**

使用得到的结果进行测试，发现结果正确。

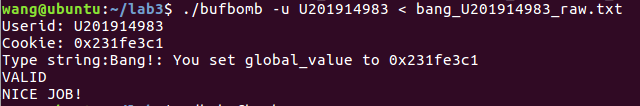


图 3.8 bang的测试结果

### 3.2.4 阶段4 boom

**1. 任务描述**

在bang的基础上，需要攻击完毕后使程序能够跳转回正确的位置继续执行，同时不会破坏原本ebp的值，避免破坏其他栈帧。

**2. 实验设计**

阶段3中，在攻击前，我们利用了gets函数不识别字符串长度的特性，修改了ebp的内容，从而使程序跳转至getbuf缓冲区内。为了恢复ebp的值，可在插入机器代码中加入相应指令，使ebp内容得到恢复。

**3. 实验过程**

在test函数调用getbuf函数前，读取ebp中的值（0x55683bf0），在攻击指令中插入恢复ebp值的语句即可。

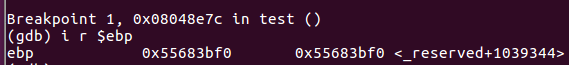


图 3.9 调用getbuf前ebp的值

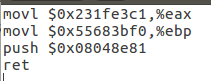


图 3.10 boom的攻击代码

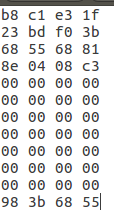


图 3.11 boom的攻击指令

**4. 实验结果**

使用得到的结果进行测试，可知结果正确。

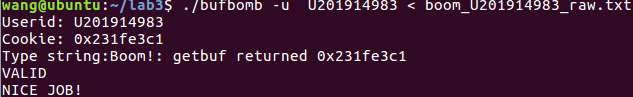


图 3.12 boom的测试结果

### 3.2.5 阶段5 nitro

**1. 任务描述**

在这一阶段，由于缺少特殊处理，getbuf函数的栈帧地址并不固定，需在此基础上构造攻击字符串，使getbufn函数返回cookie值给testn函数，同时恢复被破坏的栈帧结构。

2. 实验设计

在testn的开头可以看出，%ebp=%esp+28，可利用这一关系恢复ebp的值。

在getbufn的开头处，可见缓冲区大小为0x208，即520个字节。为防止改变程序的运行，我们向其中填充nop（90）指令，再额外填充四个字节的返回地址。

最后，需要在gdb中找到每次的跳转位置，并确定一个合理的地址。

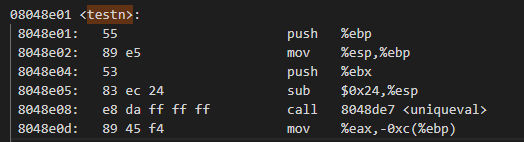


图 3.13 testn的开头内容

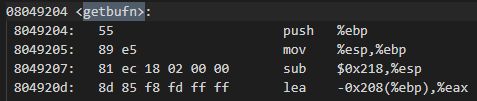


图 3.14 getbufn的开头内容

**3. 实验过程**

根据前述思路编写出攻击代码。

在gdb中进行调试，发现5次的缓冲区开始地址最小值为0x55683988，最大值为0x55683a28，差值为0xa0，这个值小于0x208。因此可将跳转地址设置为0x55683a28，即可确保不会执行到除nop（90）之外的其他指令。

最终得到文件内容如图所示（图中代码前还有500个90）.

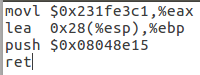


图 3.15 bitro的攻击代码

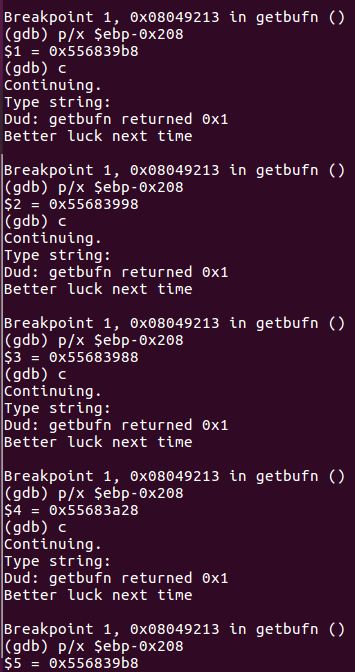


图 3.16 在gdb中查找5次的缓冲区地址



图 3.17 nitro的攻击指令（省略500个90）

**4. 实验结果**

利用所得代码进行测试，发现结果正确。

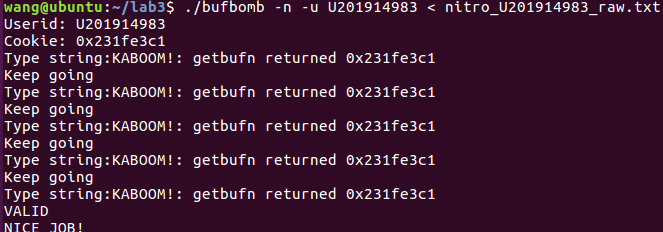


图 3.18 nitro的测试结果

## 3.3 实验小结

本次实验为缓冲区溢出攻击实验，是利用gets函数不比较字符串长度的特性进行攻击，通过大于缓冲区长度的字符串来破坏原有的堆栈结构，从而使程序跳转到指定的位置实施攻击。

在实验中，我们修改程序栈帧，改变返回地址、在getbuf缓冲区内填写攻击代码、甚至恢复栈帧来使攻击不被察觉。这让我对程序安全性有了新的认识。缓冲区溢出除了简单的报错之外，还为程序被恶意攻击破坏留下了隐患。

在阶段5，程序的栈帧随着程序运行实例的不同而变化，这为攻击提出了新的挑战。我们采用了nop指令进行填充，而非其他阶段中的任意值。另外，利用ebp和esp的关系而非一个固定值来还原栈帧。这些操作都需要以动态眼光来分析程序，启发我优化程序编写中的一些不良习惯。

# 实验总结

第一次实验，主要内容是数据的表示。我们利用位运算的方式，实现了任务要求中的多种功能。在这一过程中，我认识到，大量的运算操作都可以较为简单地利用位运算完成，可以提高计算机的执行效率。对于一些减法的操作、浮点数操作等，也使我更加深入地理解计算机中数据的存储和表示。

第二次实验，内容为拆弹实验。我们利用反汇编、静态分析和动态分析结合，寻找程序中处理和比对输入字符串的部分，从而分析出正确的拆弹字符串。在这一过程中，gdb工具强大的分析存储单元功能令我印象深刻。利用不同的格式指令，相同的内存单元可被解析成不同的格式（甚至结构体），从而方便程序的分析。这次实验也让我在实战中学习了汇编指令的AT&T表示。

第三次实验，内容为缓冲区攻击实验。这次实验让我理解了在VS下编程时，经常出现的函数不安全提示。利用这些函数可以实现类似的缓冲区溢出攻击，造成危险的结果。这次实验主要修改的是堆栈中返回地址、传入参数等信息，使我在实际操作中强化了函数调用过程中堆栈的变化。