

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**专业班级：**

**学 号：**

**姓 名：**

**指导教师：**

**报告日期： 2022年6月14日**

**计算机科学与技术学院**

**目录**

[1 实验3：缓冲区溢出攻击 2](#_Toc45)

[1.1 实验概述 2](#_Toc45)

[1.2 实验要求 2](#_Toc12776)

[1.3 实验内容 2](#_Toc26710)

[1.3.1 阶段1 smoke 2](#_Toc13085)

[1.3.2 阶段2 fizz 5](#_Toc21957)

[1.3.3 阶段3 bang 6](#_Toc15690)

[1.3.4 阶段4 boom 9](#_Toc20207)

[1.3.5 阶段5 nitro 11](#_Toc23886)

[1.4 实验小结 16](#_Toc1603)

[参考文献 17](#_Toc1603)

# 1 实验3：缓冲区溢出攻击

# 1.1 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。

实验分5个难度递增的等级，分别命名为Smoke、Fizz、Bang、Boom和Nitro，其中Smoke级最简单而Nitro级最困难。

实验语言：C语言。

实验环境：linux。

# 1.2 实验要求

1. 对目标程序实施缓冲区溢出攻击。

2. 通过造成缓冲区溢出来破坏目标程序的栈帧结构。

3. 使程序执行一些原来没有的行为。

# 1.3 实验内容

## 1.3.1 阶段1 smoke

1.任务描述：

构造一个攻击字符串作为bufbomb的输入，而在 getbuf()中造成缓冲区溢出，使得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：

利用 objdump 反汇编出的指令进行静态分析，利用 gdb 进行动态分析，结合对函数栈帧结构的理解，以及其建立与回收过程，通过缓冲区溢出的手段，覆写正常栈帧结构，以到达控制程序执行方向的目的。

3.实验过程：

首先，通过objdump对bomb进行反汇编并将汇编代码输出到bufbomb.asm中，如图1.1所示。

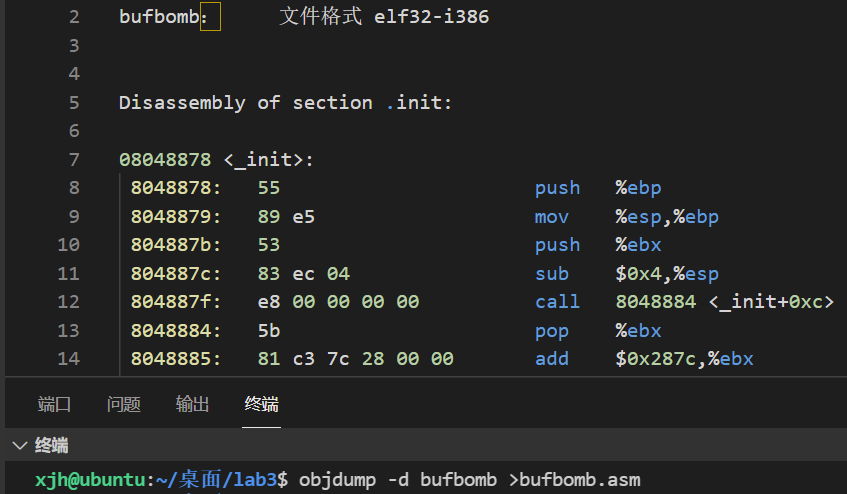


图1.1 生成bufbomb.asm

在bufbomb的反汇编源程序中查找smoke函数，记录smoke函数开始地址为0x8048c90，如图1.2所示。

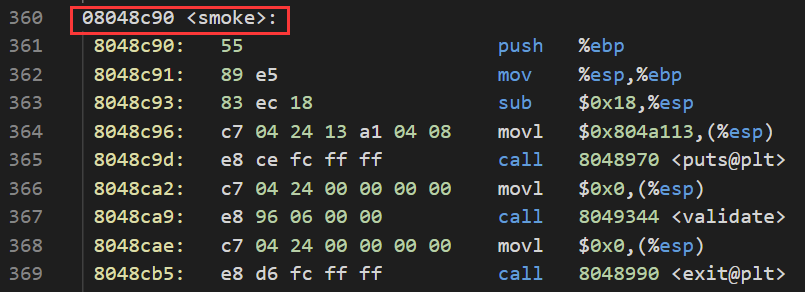


图1.2 smoke函数开始地址

再分析getbuf函数，可知getbuf函数栈帧大小为0x3c个字节，其中buf缓冲区总共为0x28个字节，如图1.3所示。

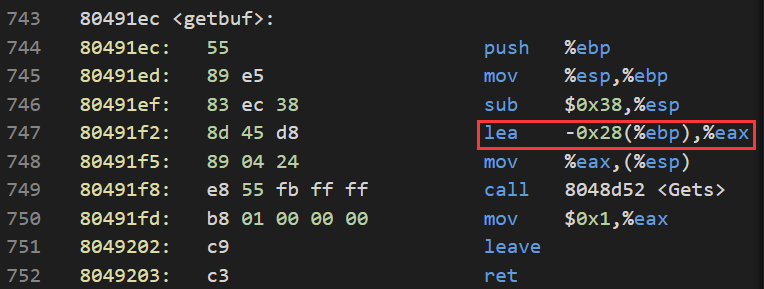


图1.3 getbuf函数

根据实验任务书所给出的getbuf栈帧图，可知要完成任务，应输入大小为48个字节的字符串，其中最后四个字节的内容应该填入smoke函数的首地址，使得执行完getbuf函数后转而执行smoke函数。

可以看到getbuf的栈帧是0x38+4个字节， buf缓冲区的大小是0x20个字节，还有8个空闲字节（共40个字节）。

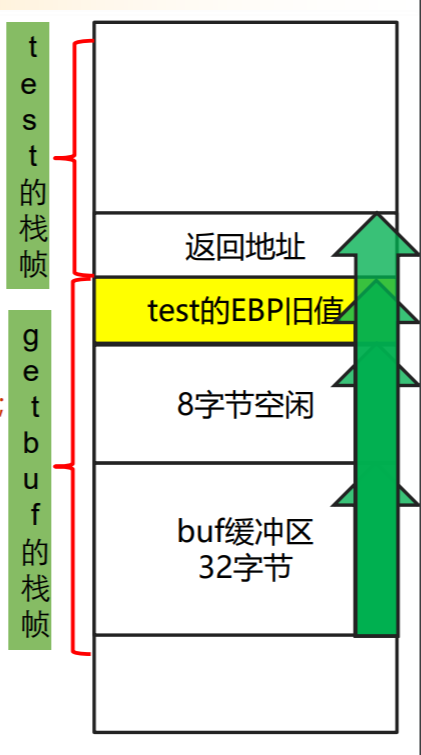


图1.4 getbuf栈帧结构

攻击字符串的功能是用来覆盖getbuf函数内的数组buf，进而溢出并覆盖ebp和ebp上面的返回地址，所以攻击字符串的大小应该是0x28+4+4=48个字节。攻击字符串的最后4个字节应是smoke函数的地址。则攻击字符串应为：00 00 00… …00 00 90 8c 04 08，如图1.5所示。

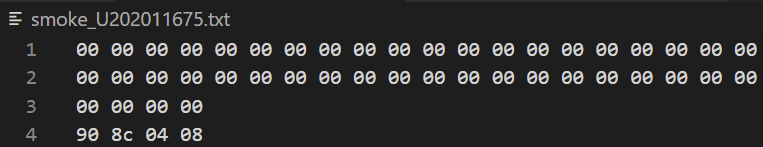


图1.5 结果字符串

4.实验结果：

将上述攻击字符串写入smoke\_U202011675.txt文件中，并在命令窗口中输入对应命令，实验结果如图1.6所示，阶段1成功通过。

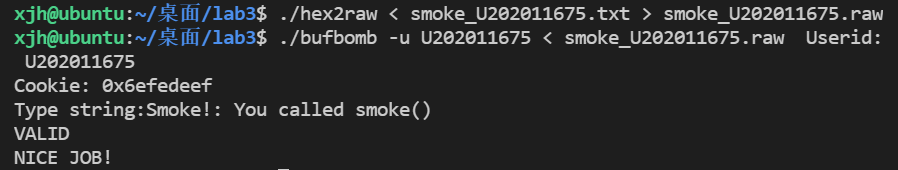


图1.6 阶段1测试结果

## 1.3.2 阶段2 fizz

1.任务描述

构造一个攻击字符串作为bufbomb的输入，在getbuf()中造成缓冲区溢出，使得本次getbuf()返回时不是返回到test函数继续执行，而是转向执行fizz()。与smoke阶段不同和且较难的地方在于：fizz函数需要一个输入参数，因此要将cookie值作为参数传递给fizz 函数，以便于fizz中val与cookie的比较能够成功。

2.实验设计：

利用 objdump 反汇编出的指令进行静态分析，利用 gdb 进行动态分析，结合对函数栈帧结构的理解，以及其建立与回收过程，通过缓冲区溢出的手段，覆写正常栈帧结构，以到达控制程序执行方向的目的。

3.实验过程：

本阶段和smoke类似，查看fizz的汇编代码，如图2.1所示。

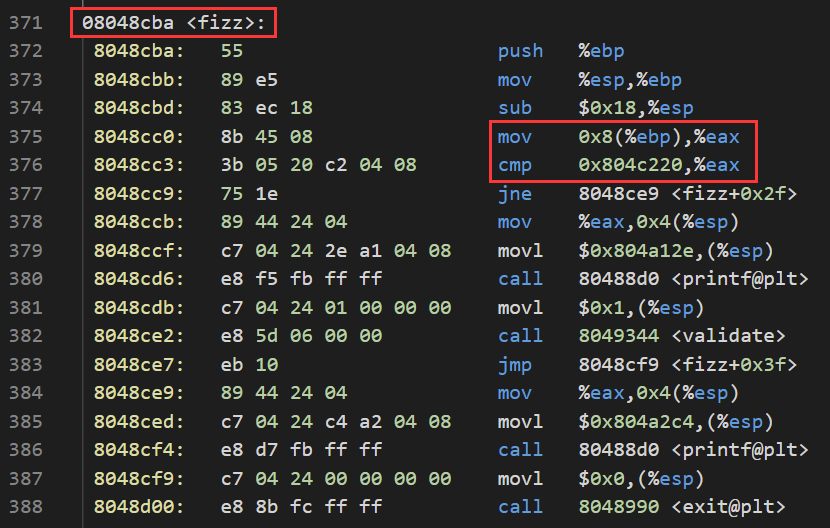


图2.1 fizz函数

fizz函数与smoke基本一致，ebp的上一位置ebp+4存放了调用者的返回地址，所以参数的地址应该为ebp+8的位置，我们只需要将自己的cookie放置在该位置即可。

而fizz函数开始的地址为08048cba，所以输入的前44个字节为非'\n'任意值，第45-48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，即07 ab 99 67，最终输入的56个字节如图2.2所示。

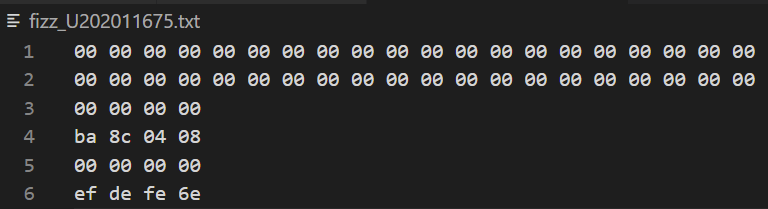


图2.2 结果字符串

4.实验结果

将上述攻击字符串写入fizz\_U202011675.txt文件中，并在命令窗口中输入对应命令，实验结果如图2.3所示，阶段2成功通过。

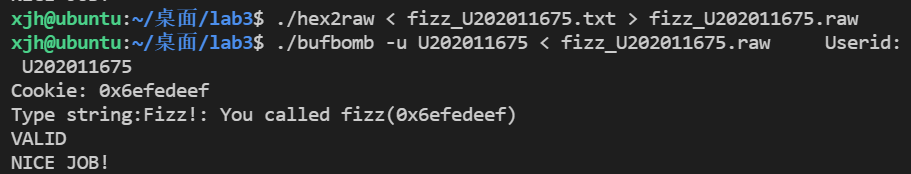


图2.3 阶段2测试结果

## 1.3.3 阶段3 bang

1.任务描述

本阶段的任务是设计包含攻击代码的攻击字符串，所含攻击代码首先将全局变量global\_value设置为你的cookie值，然后转向执行bang()。

2.实验设计：

利用 objdump 反汇编出的指令进行静态分析，利用 gdb 进行动态分析，结合对函数栈帧结构的理解，以及其建立与回收过程，通过缓冲区溢出的手段，覆写正常栈帧结构，以到达控制程序执行方向的目的。

3.实验过程：

先观察bang的代码，如图3.1所示。bang函数的首地址0x08048d05，在bang函数中，会将全局变量global\_value和cookie进行比较。

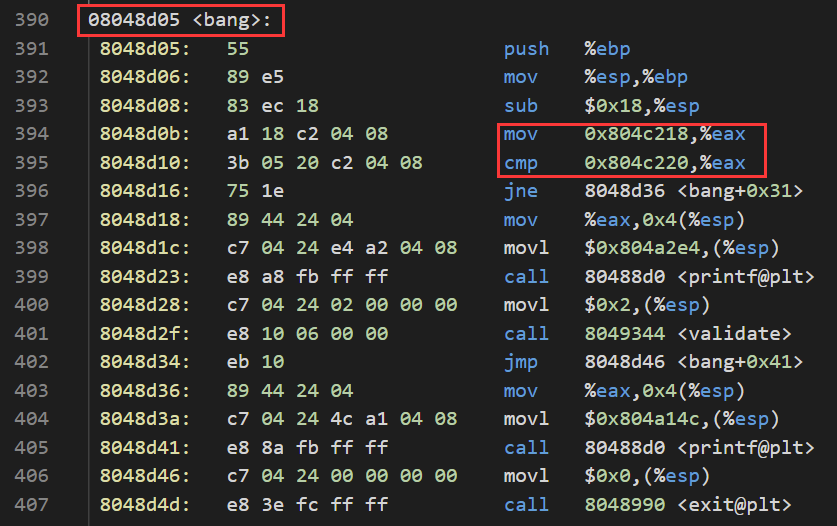


图3.1 bang函数

global\_value的地址是0x804c218，cookie的地址是0x804c220，global\_value在c代码中显示为0，所以需要修改global\_value的值使其与cookie一致。汇编代码为：

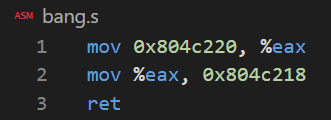


图3.2 bang.s代码

将bang.s进行反汇编，如图3.3所示。

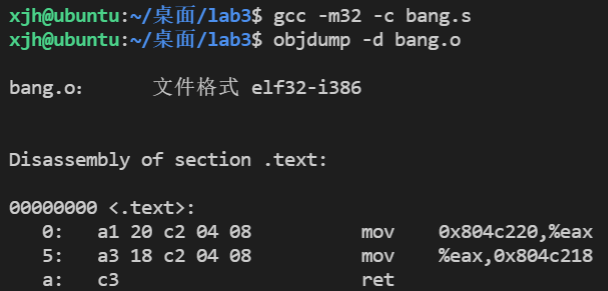


图3.3 攻击代码及其机器指令

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

设置断点查看cookie为 0x6efedeef时buf的首地址，如图3.4所示。

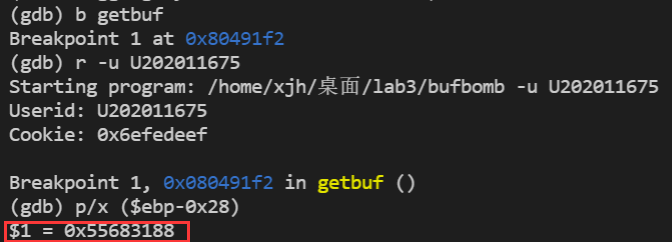


图3.4 查看buf首址

可知buf首址为0x55683188，综合之前的指令序列，45-48字节放buf首址，49-52放bang函数首址，得到结果字符串：



图3.5 结果字符串

4.实验结果

将上述攻击字符串写入bang\_U202011675.txt文件中，并在命令窗口中输入对应命令，实验结果如图3.6所示，阶段3成功通过。

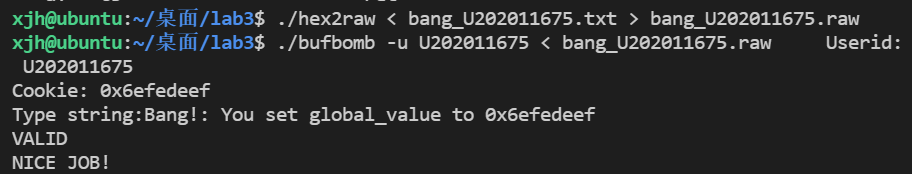


图3.6 阶段3测试结果

## 1.3.4 阶段4 boom

1.任务描述

本阶段的实验任务是构造一个攻击字符串，使得getbuf函数不管获得什么输入，都能将正确的cookie值返回给test函数，而不是返回值1。除此之外，攻击代码应还原任何被破坏的状态，将正确返回地址压入栈中，并执行ret指令从而真正返回到test函数。

2.实验设计：

利用 objdump 反汇编出的指令进行静态分析，利用 gdb 进行动态分析，结合对函数栈帧结构的理解，以及其建立与回收过程，通过缓冲区溢出的手段，覆写正常栈帧结构，以到达控制程序执行方向的目的。

3.实验过程：

首先观察函数test的汇编代码，如图4.1所示。

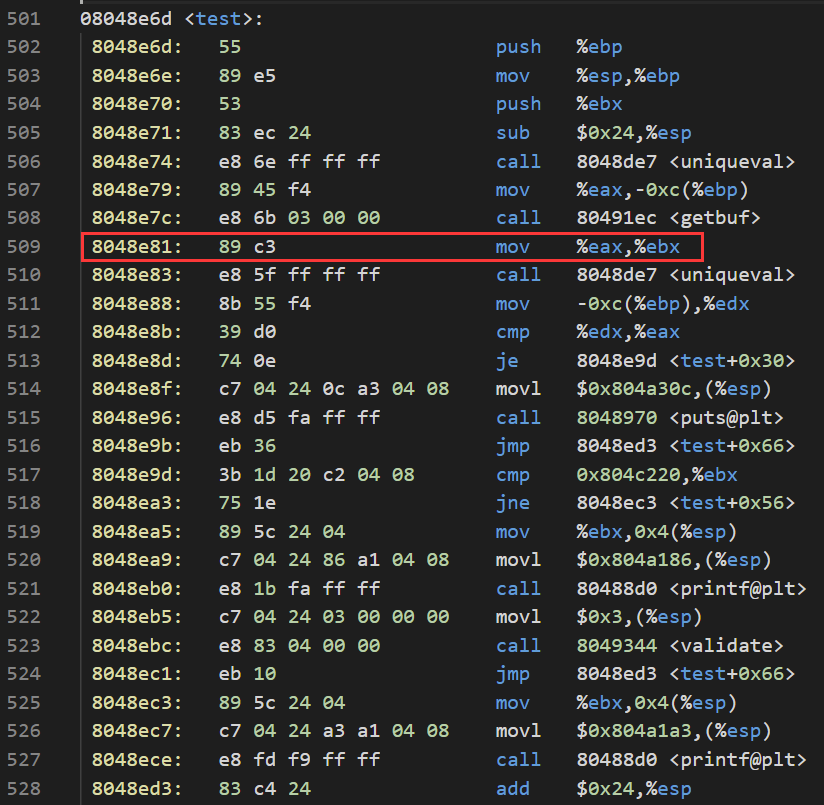


图4.1 test函数

要返回至test，应该返回至调用getbuf之后的一步，地址为0x08048e81，返回的值要为cookie的值，写出汇编代码：

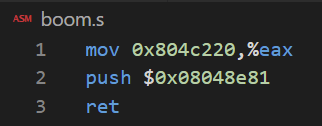


图4.2 boom.s代码

将boom.s进行反汇编，如图4.3所示。

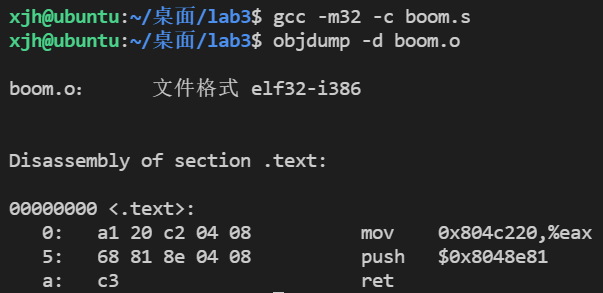


图4.3 攻击代码及其机器指令

得到指令序列a1 20 c2 04 08 68 81 8e 04 08 c3。

覆盖getbuf返回地址的时候会覆盖保存的寄存器ebp的值，所以通过设置断点来查看ebp的地址，如图4.4所示。

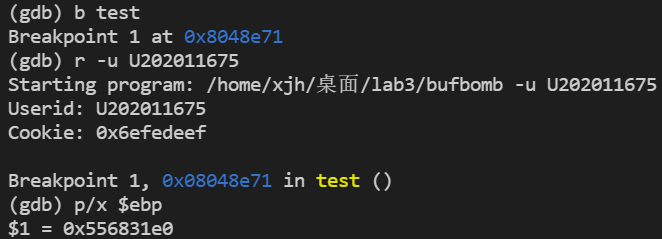


图4.4 查看ebp寄存器值

得到的地址为0x556831e0。

所以得到的序列指令放在前11个字节，41-44位存储ebp地址的，最后四位存储buf首址，如图4.5所示。



图4.5 结果字符串

4.实验结果

将上述攻击字符串写入boom\_U202011675.txt文件中，并在命令窗口中输入对应命令，实验结果如图4.6所示，阶段4成功通过。

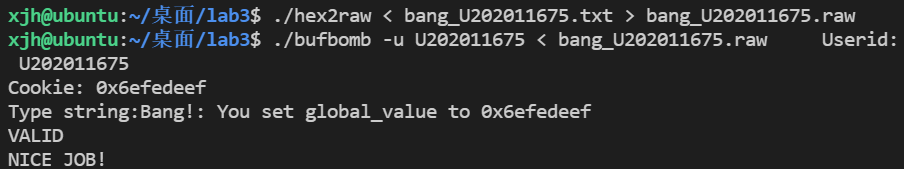


图4.6 阶段4测试结果

## 1.3.5 阶段5 nitro

1.任务描述

与阶段4类似，构造一攻击字符串使得getbufn函数（注，在nitro阶段，bufbomb将调用testn函数和getbufn函数，见bufbomb.c）返回cookie值至testn函数，而不是返回值1。

2.实验设计：

利用 objdump 反汇编出的指令进行静态分析，利用 gdb 进行动态分析，结合对函数栈帧结构的理解，以及其建立与回收过程，通过缓冲区溢出的手段，覆写正常栈帧结构，以到达控制程序执行方向的目的。

3.实验过程：

首先观察getbufn函数的反汇编源程序，如图5.1所示。

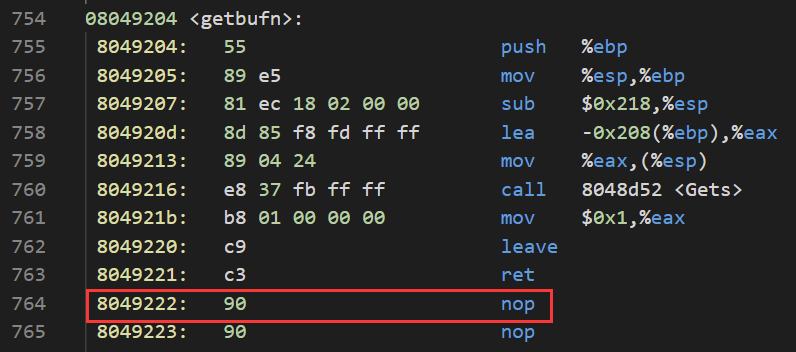


图5.1 getbufn函数

可知，在testn中每次调用getbufn后，栈顶指针esp的值并不会发生改变，但若输入的字符串造成了缓冲区溢出，则在执行leave指令后恢复至寄存器ebp中的值已不是原ebp值，因此需要进一步确定如何恢复该值。

继续观察testn的反汇编源程序，如图5.2所示。

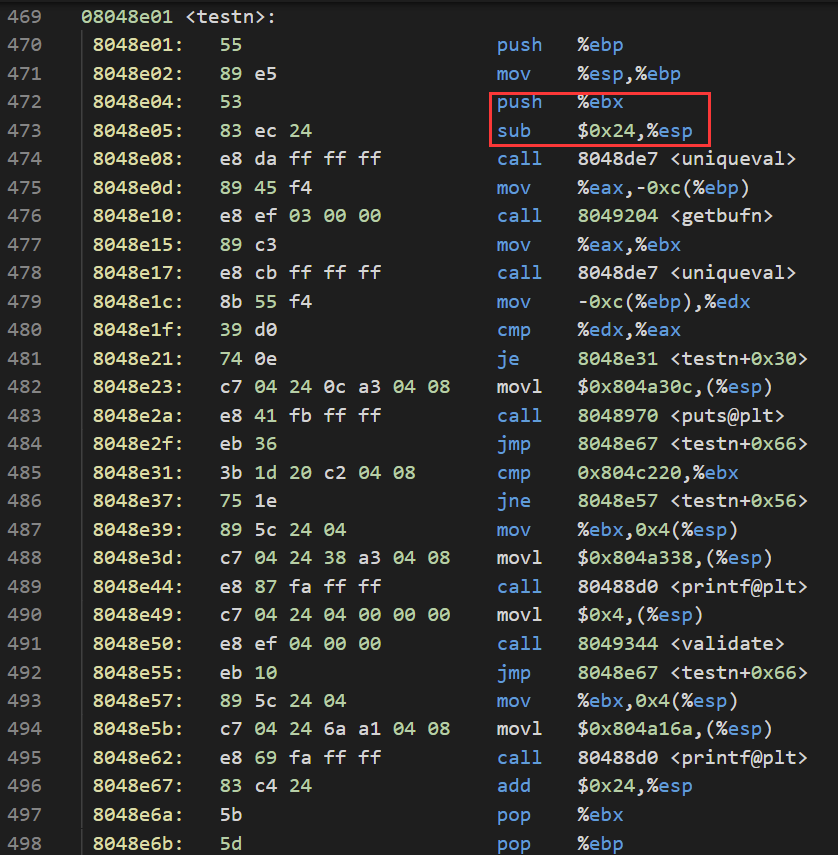
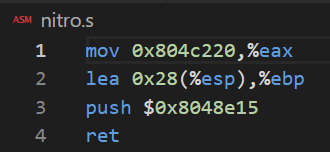


图5.2 testn函数

根据testn的汇编指令，可以看出ebp = esp + 0x28，结合前面的分析，可知可以将esp的值加上0x28即为原ebp值。同时，攻击字符串中所包含的返回地址应为0x8048e15。由此，编写相应攻击代码，如图5.3所示。



5.3 nitro.s代码

将boom.s进行反汇编，如图5.4所示。

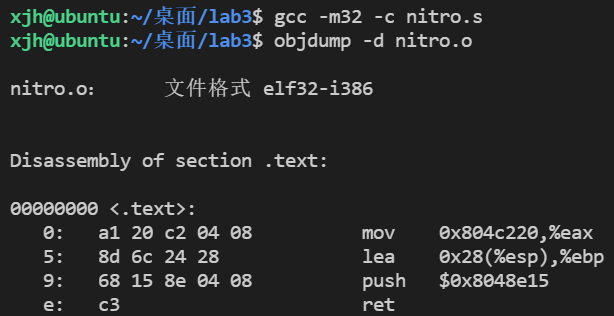


图5.4 攻击代码及其及其指令

由于五次调用getbufn时，写入字符串的首地址是变化的，无法通过gdb调试确定唯一值，因此可以选择让程序跳转到一个大致的位置，并且使用nop指令（其机器码为0x90）填充（CPU在执行nop指令时除程序计数器自加不进行任何其他操作），这样便可以使程序从该大致位置开始执行nop指令，直至执行到我们预先写好的攻击代码。同时，应将代码放在高位地址处，保证无论缓冲区的首地址为何，都能确保程序所跳转至的地址低于攻击代码存放地址。下一步，通过gdb调试工具查看五次循环的ebp值如图5.5所示。

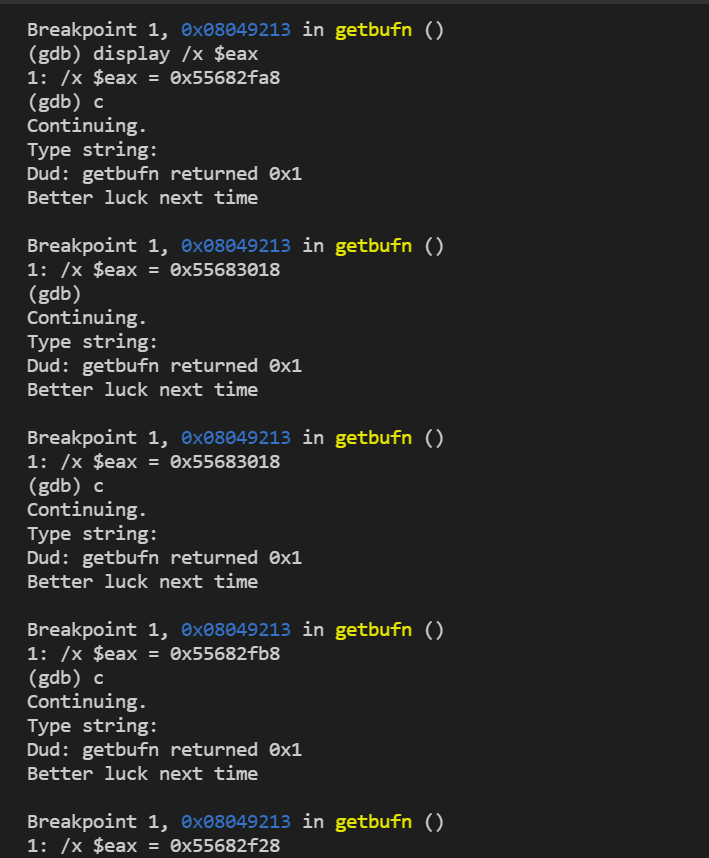


图5.5 5次循环的ebp值

由此可以确定缓冲区的首地址的最大值为0x55683018。并且由分析可知，返回地址应取buf首地址的最大值。因为字符串的存放位置是随机的，若返回地址不取buf首地址的最大值，而在某一次循环中攻击字符串存放的首地址为最大值，则程序便会执行从返回地址到攻击字符串存放的起始位置这一段空间中的机器指令，造成错误。

由于跳转之后的地址仍然低于汇编攻击指令所在处，虽然每次攻击必然能够执行攻击指令，但会执行额外不必要的坏指令，所以需要用 nop 指令将这段空隙填充掉。

同时，根据前面的推导不难得出，攻击字符串的长度应为0x208+0x4+0x4=528个字节，先填充nop指令，后填充攻击代码指令，最后填充四个字节的返回地址，如图5.6所示。

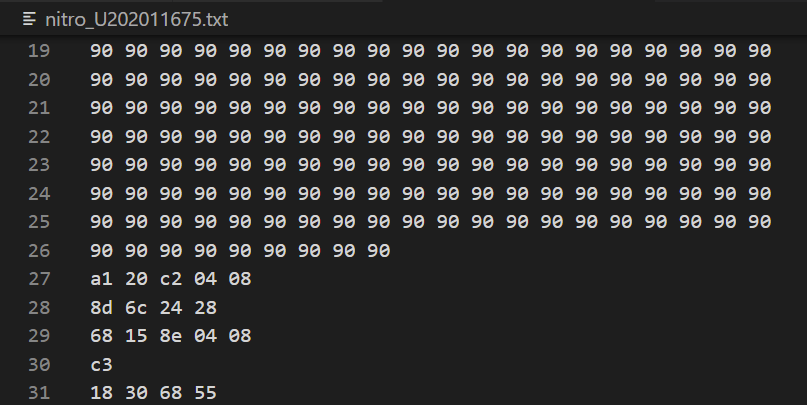


图5.6 结果字符串

4.实验结果

将上述攻击字符串写入nitro\_U202011675.txt文件中，并在命令窗口中输入对应命令，实验结果如图5.7所示，阶段5成功通过。

需要注意的是因为在Nitro模式下主程序需要读五次input以满足执行五次的需要，因此在执行./hex2raw程序时需要添加 -n flag以保证input string 被复制五次，每次以\n结尾来结束每次的gets()函数调用。

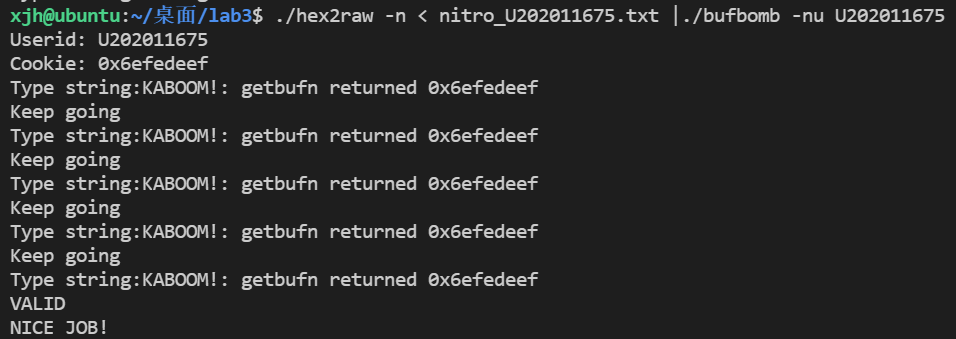


图5.7 阶段5测试结果

# 1.4 实验小结

这一次实验的主题是缓冲区溢出攻击，主要思想就是利用缓冲区溢出，将程序的运行流向转到原先设计好的函数或者自己新编写的函数之中。通过这次实验，我有了一下几点收获：

1. 通过深入理解缓冲区溢出buffer overflow的情形，体会到了体会几种可能危害到系统安全的场景
2. 加深了对IA-32函数调用规则和栈结构的具体理解，巩固了汇编指令集。
3. 巩固了对objdump、gcc等工具的掌握，可以熟练的将汇编代码用gcc编译成机器指令，再用objdump反汇编成二进制字节数据和汇编代码，来构造具有攻击代码的攻击字符串。
4. 学会了用gdb调试，能够熟练地用gdb指令设置断点来让程序暂停、观察断点处必要的内存单元内容、寄存器内容等。

# 参考文献

[1]袁春风. 计算机系统基础. 机械工业出版社，2014

[2]袁春风. 计算机系统基础. 机械工业出版社，2018

[3]深入理解计算机系统. 机械工业出版社，2016