**深 圳 大 学 实 验 报 告**

**课 程 名 称： 计算机系统(2)**

**实验项目名称： 缓冲区溢出攻击实验**

**学 院： 计算机与软件学院**

**专 业： 计算机科学与技术/软件工程/物联网工程**

**指 导 教 师： 王毅**

**报告人： 沈晨玙 学号： 2019092121 班级： 国际班**

**实 验 时 间： 2021年6月24日**

**实验报告提交时间： 2021年6月24日**

**教务处制**

**一、 实验目标：**

1. 理解程序函数调用中参数传递机制；
2. 掌握缓冲区溢出攻击方法；
3. 进一步熟练掌握GDB调试工具和objdump反汇编工具。

**二、实验环境：**

1. 计算机（Intel CPU）
2. Linux 64位操作系统
3. GDB调试工具
4. objdump反汇编工具

**三、实验内容**

本实验设计为一个黑客利用缓冲区溢出技术进行攻击的游戏。我们仅给黑客（同学）提供一个二进制可执行文件bufbomb和部分函数的C代码，不提供每个关卡的源代码。程序运行中有3个关卡，每个关卡需要用户输入正确的缓冲区内容，否则无法通过管卡！

要求同学查看各关卡的要求，运用**GDB调试工具和objdump反汇编工具**，通过分析汇编代码和相应的栈帧结构**，**通过缓冲区溢出办法在执行了getbuf()函数返回时作攻击，使之返回到各关卡要求的指定函数中。第一关只需要返回到指定函数，第二关不仅返回到指定函数还需要为该指定函数准备好参数，最后一关要求在返回到指定函数之前执行一段汇编代码完成全局变量的修改。

实验代码bufbomb和相关工具（sendstring/makecookie）的更详细内容请参考“实验四 缓冲区溢出攻击实验.pptx”。

本实验要求解决关卡1、2、3，给出实验思路，通过截图把实验过程和结果写在实验报告上。

**四、实验步骤和结果**

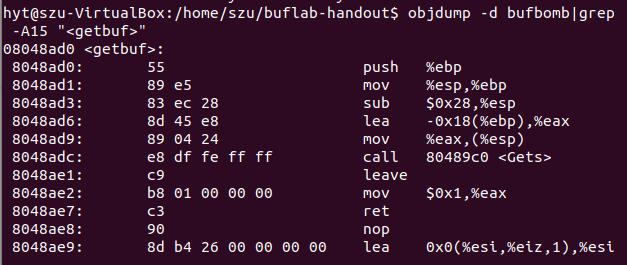
**因为本次实验用到的可执行文件是32位，而实验环境是64位的，需要先安装一个32位的库，在root权限下安装如下所示：**



**还需要安装sendmail**



首先利用反汇编命令查看getbuf函数的汇编代码，以便分析getbuf在调用<Gets>时的栈帧结构，汇编代码如下：



**步骤1 返回到smoke()**

* 1. **解题思路**

本实验中，bufbomb中的test()函数将会调用getbuf()函数，getbuf()函数再调用gets()从标准输入设备读入字符串。

系统函数gets()未进行缓冲区溢出保护。其代码如下：

***int getbuf()***

***{***

***char buf[12];***

***Gets(buf);***

***return 1;***

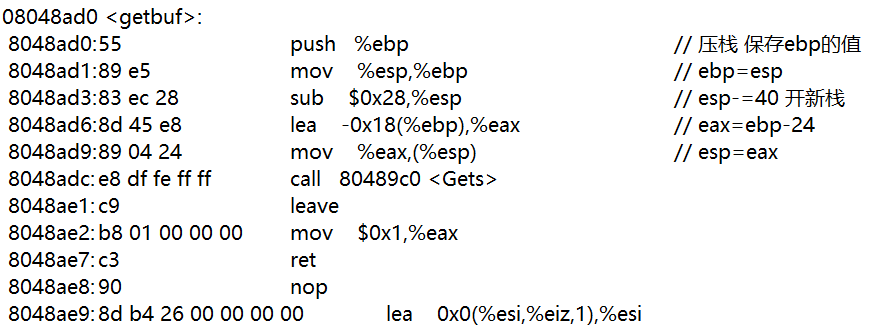
***}***

我们的目标是使getbuf()返回时，不返回到test()，而是直接返回到指定的smoke()函数。

为此，我们可以通过构造并输入大于getbuf()中给出的数据缓冲区的字符串而破坏getbuf()的栈帧，替换其返回地址，将返回地址改成smoke()函数的地址。

* 1. **解题过程**

对于第一关：

分析getbuf()函数的汇编代码，可以发现，getbuf()在保存%ebp的旧值后，将%ebp指向%esp所指的位置，然后将栈指针减去0x28来分配额外的20个字节的地址空间。字符数组buf的位置用%ebp下0x18(即24)个字节来计算。然后调用Gets（）函数，读取的字符串返回到%ebp-0x18，即%ebp-24。

**具体的栈帧结构如下：**

|  |  |
| --- | --- |
| **栈帧** |  |
| 返回地址 | 属于调用者的栈帧 |
| 保存的%ebp旧值 | %ebp |
| 20-23 |  |
| 16-19 |  |
| 12-15 |  |
| [11][10][9][8] |  |
| [7][6][5][4] |  |
| [3][2][1][0] | buf,%ebp-0x18 |
|  |  |
|  |  |
|  |  |
|  | %esp，%ebp-0x24 |

从以上分析可得，只要输入不超过11个字符，gets返回的字符串（包括末尾的null）就能够放进buf分配的空间里。长一些的字符串就会导致gets覆盖栈上存储的某些信息。

随着字符串变长，下面的信息会被破坏：

|  |  |
| --- | --- |
| **输入的字符数量** | **附加的被破坏的状态** |
| **0-11** | **无** |
| **12-23** | **分配后未使用的空间** |
| **24-27** | **保存的%ebp旧值** |
| **28-31** | **返回地址** |
| **32+** | **调用者test()中保存的状态** |

因此，我们要替换返回地址，需要构造一个长度至少为32的字符串，其中的第0~11个字符放进buf分配的空间里，第12~23个字符放进程序分配后未使用的空间里，第24~27个字符覆盖保存的%ebp旧值，第28-31个字符覆盖返回地址。

由于替换掉返回地址后，getbuf()函数将不会再返回到test()中，所以覆盖掉test()的%ebp旧值并不会有什么影响。也就是说我们构造的长度为32的字符串前28个字符随便是啥都行，而后面四个字符就必须能表示smoke()函数的地址。所以我们要构造的字符串就是“28个任意字符+smoke()地址”。任意的28个字符都用十六进制数00填充就行。

利用objdump查看smoke（）函数的地址如下：

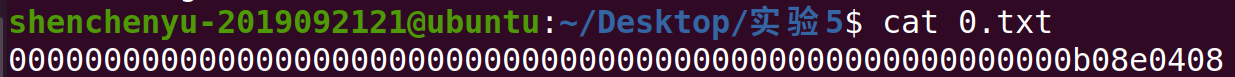


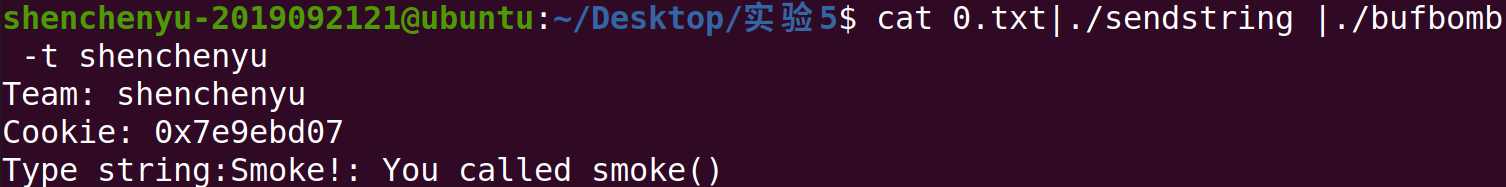
由此可得smoke()函数的地址是0x08048eb0,考虑小端法表示，对应的我们要构造的字符串是:

**00000000000000000000000000000000000000000000000000000000b08e0408**

将输入的字符串保存到0.txt中。

* 1. **最终结果截图**





**2.1 解题思路**

这一关要求返回到fizz()并传入自己的cookie值作为参数，破解的思路和第一关是类似的，构造一个超过缓冲区长度的字符串将返回地址替换成fizz()的地址，只是增加了一个传入参数，所以在读入字符串时，要把fizz()函数读取参数的地址替换成自己的cookie值，具体细节见解题过程。

**2.2 解题过程**

首先还是利用objdunp查看并分析fizz()函数的汇编代码：



从汇编代码可知，fizz()函数被调用时首先保存%ebp旧值并分配新的空间，然后读取%ebp-0x8地址处的内容作为传入的参数，要求传入的参数是自己的cookie值。也就是说传入的参数其实是存在%ebp-0x8处的，具体的栈帧结构如下：

|  |  |
| --- | --- |
| **栈帧** |  |
| 传入的参数 | %ebp+0x8 |
|  | %ebp+0x4 |
| 保存的%ebp旧值 | %ebp |
|  |  |
|  | %esp |

对应到getbuf()函数中的栈帧结构如下：

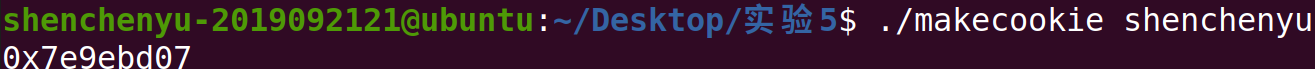
|  |  |
| --- | --- |
| **栈帧** |  |
|  | 需要替换成cookie传入fizz() |
|  | 任意替换 |
| 返回地址 | 属于调用者的栈帧 |
| 保存的%ebp旧值 | %ebp，需要替换成fizz()的地址 |
|  | 任意替换 |
|  | 任意替换 |
|  | 任意替换 |
| [11][10][9][8] |  |
| [7][6][5][4] |  |
| [3][2][1][0] | buf,%ebp-0x18 |
|  |  |
|  |  |
|  |  |
|  | %esp，%ebp-0x24 |

由以上结构不难判断出，我们需要读入buf的字符串为“28个任意字符+fizz()的地址+4个任意的字符+自己的cookie值”，每个字符还是用十六进制数表示。

为此，首先利用objdump得到fizz()的地址为0x08048e60:

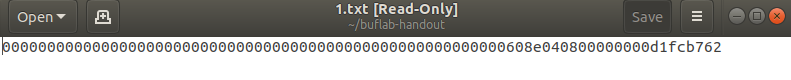


再利用makecookie生成自己的cookie值为0x7e9ebd07：

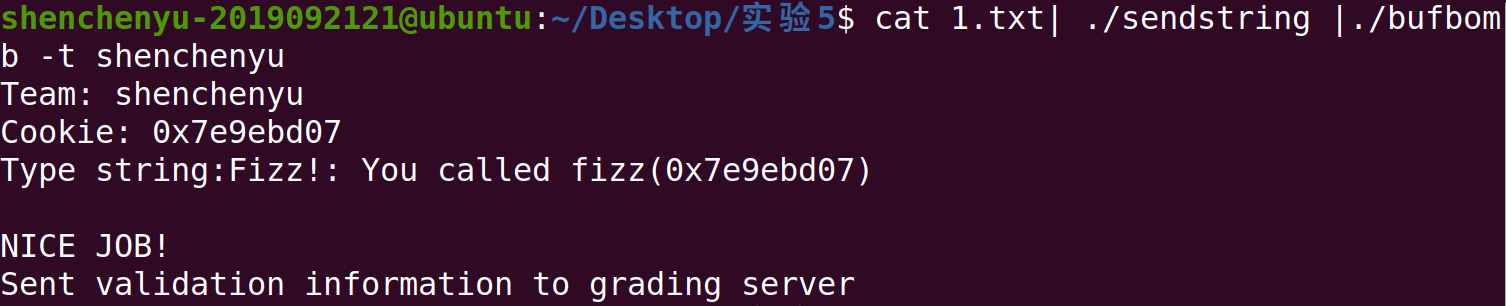


任意的字符依旧用00表示，则最后我们得到构造出来的字符串为：**00000000000000000000000000000000000000000000000000000000608e04080000000007bd9e7e**

将该字符串保存在1.txt中：

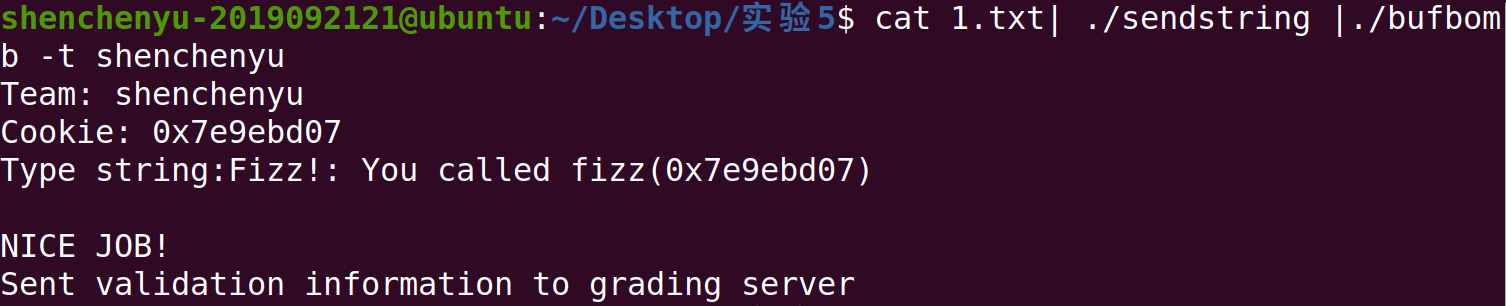


然后通过sendstring将1.txt转换成二进制格式，再通过管道输入到bufbomb中：



**2.3 最终结果截图**

从以上过程可得，最后成功利用缓冲区溢出漏洞，让getbuf()直接返回到fizz()函数并传入自己的cookie值作为参数，最终结果截图如下：



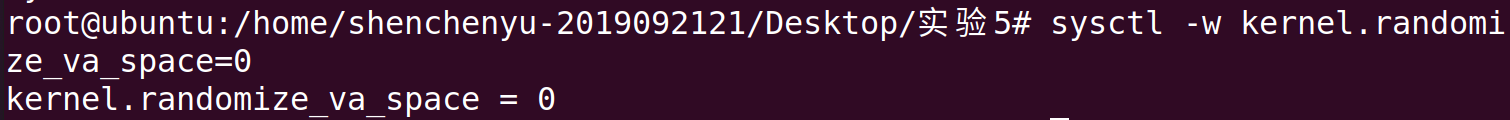
**步骤3 返回到bang()且修改global\_value**

* 1. **解题思路**

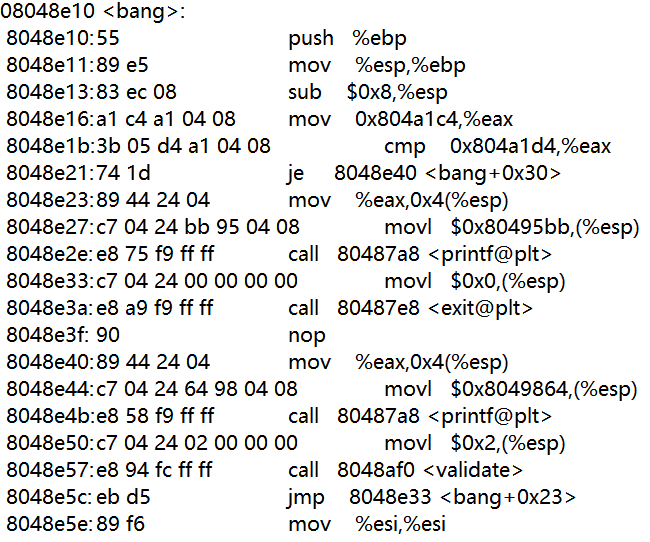
这一关要求先修改全局变量global\_value的值为自己的cookie值，再返回到band()。为此需要先编写一段代码，在代码中把global\_value的值改为自己的cookie后返回到band()函数。将这段代码通过GCC产生目标文件后读入到buf数组中，并使getbuf函数的返回到buf数组的地址，这样程序就会执行我们写的代码，修改global\_value的值并调用band()函数。具体细节见解题过程。

* 1. **解题过程**

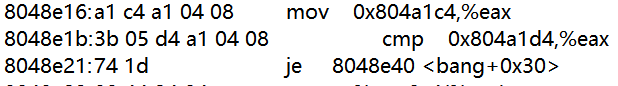
首先，为了能精确地指定跳转地址，先在root权限下关闭Linux的内存地址随机化：



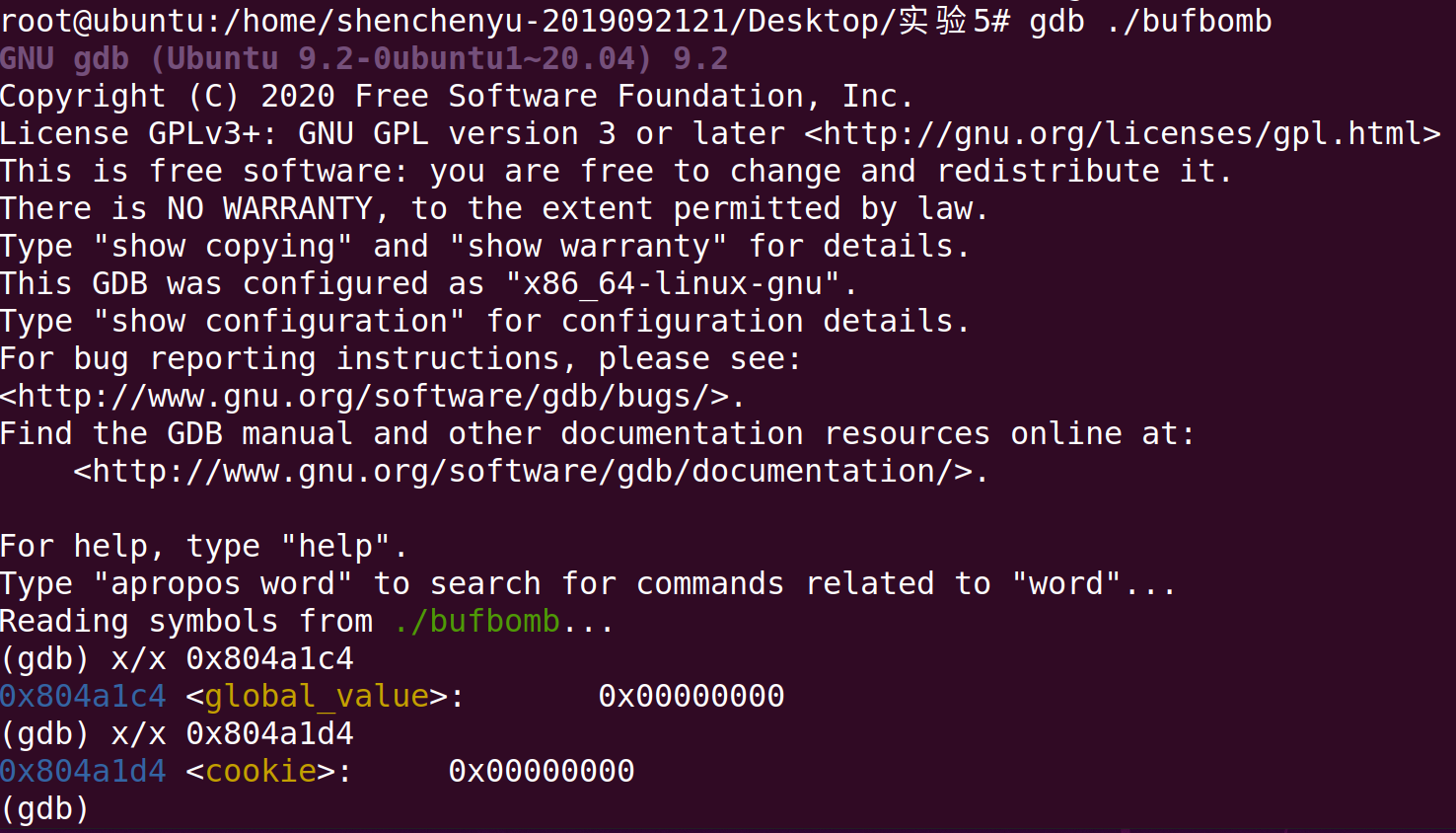
用objdump查看bang()函数的汇编代码如下：



很明显，bang()函数首先读取0x804a1c4和0x804a1d4的地址的内容并进行比较，要求两个地址中的内容相同：

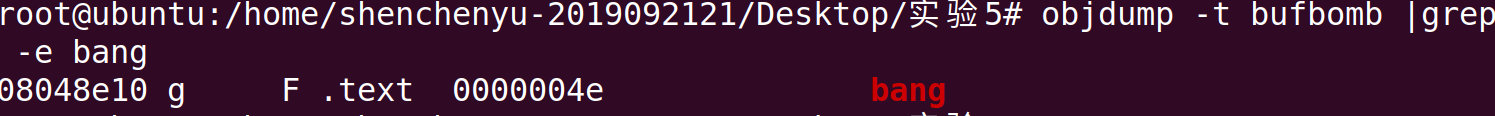


用gdb调试命令查看：



gdb可以发现，0x804a1c4就是全局变量global\_value的地址，0x804a1d4是cookie的地址。因此，我们只要在自己写的代码中，把地址0x804a1d4的内容存到地址0x804a1c4就行了。

再利用objdump得到bang()函数的入口地址为0x08048e10:

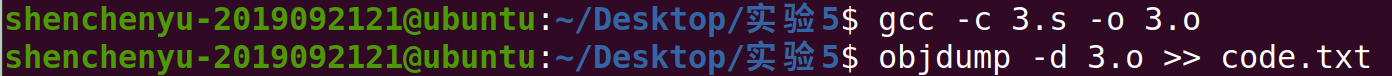


到这里，就可以确定我们自己写的代码要干的事情了。首先是将global\_value的值设置为cookie的值，也就是将0x804a1c4的值设置为0x804a1d4的值，然后将bang()函数的入口地址0x08048e10压入栈中，这样当函数返回的时候，就会直接取栈顶作为返回地址，从而调用bang()函数。接着函数返回，此时返回的地址就是上一条语句中压入栈中的地址，也就是bang()函数的入口地址了。

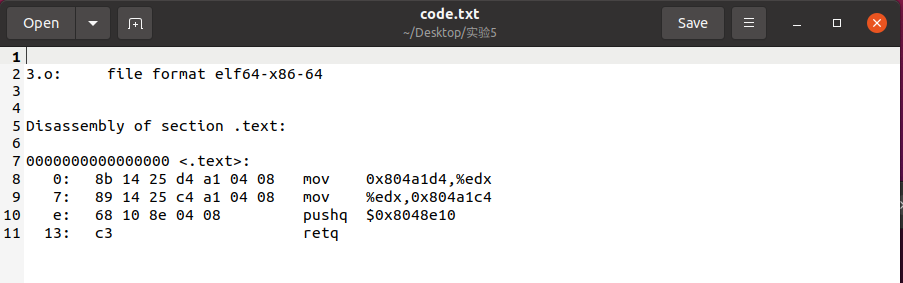
为此，新建一个名为3.s的文件，里面写相应的汇编代码：



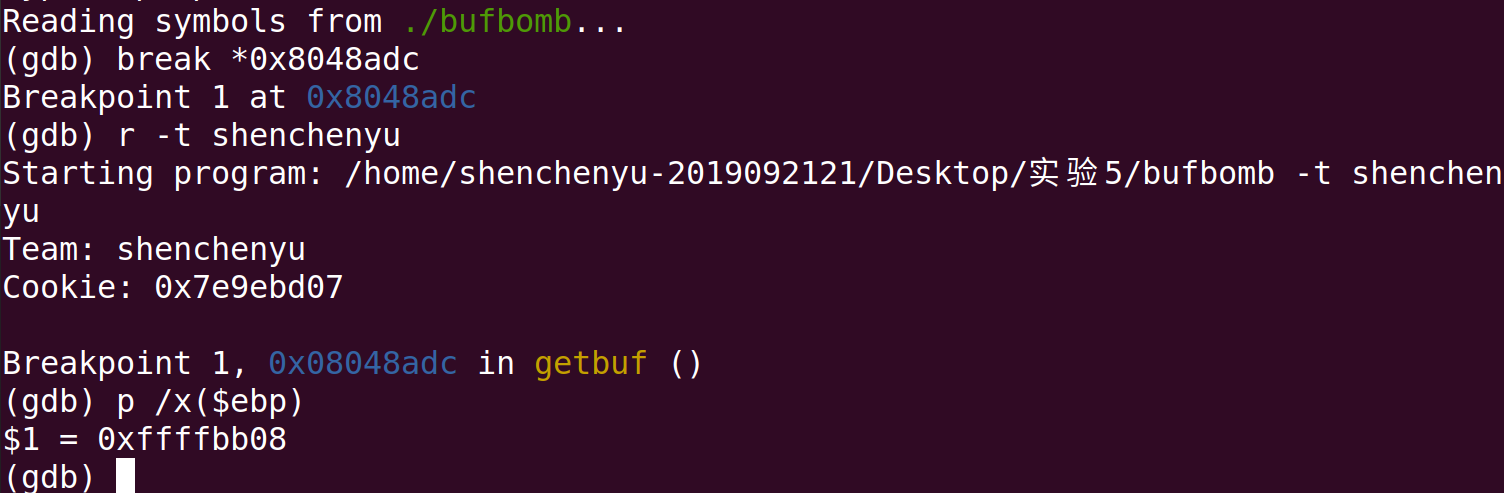
通过gcc编译该汇编代码，再反编译输出到code.txt中：



查看mycode.txt中生成的对应汇编代码的机器码如下：



在将这段机器码放入buf数组中后，为了能让getbuf()返回到buf数组处执行我们的代码，需要想办法得到buf数组的地址。为此，用gdb断点调试查看执行getbuf()时ebp的值为0xffffb148：

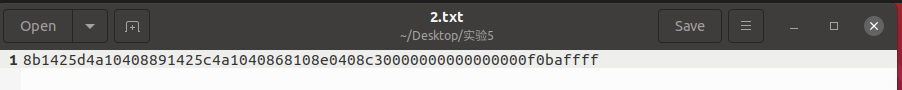


从第一关中对getbuf()函数栈帧结构的分析可知buf数组的首地址为%ebp-0x18，即0xffffbaf0

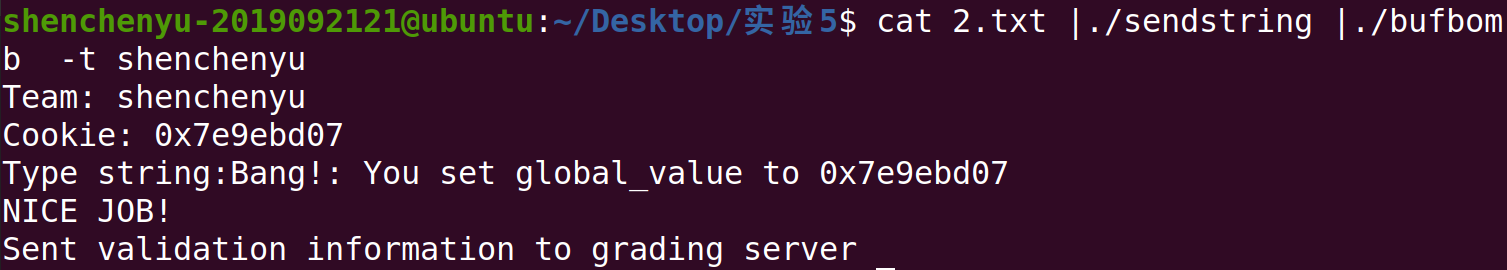
综上所述，最后我们要构造的字符串为自己写的汇编代码生成的机器码(20个字符)+8个任意字符+buf数组的首地址，任意字符依然取00，则对应的字符串为：

**8b1425d4a10408891425c4a1040868108e0408c30000000000000000f0baffff**

将该字符串保存在2.txt中：

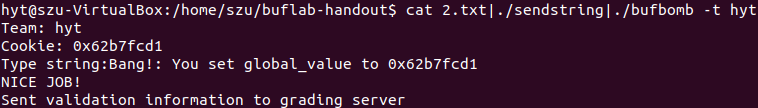


然后通过sendstring将2.txt转换成二进制格式，再通过管道输入到bufbomb中：



**3.3 最终结果截图**

根据以上解题过程，利用缓冲区溢出，让程序执行自己写的汇编代码对全局变量进行修改后，成功调用了bang（）函数。最后结果截图如下：



**五、实验总结与体会**

最终完成了实验，进一步了解了汇编代码&gdb的使用方式。

|  |
| --- |
| **指导教师批阅意见：**  **成绩评定：**  指导教师签字： 王毅    2020年 6月25日 |
| 备注： |