|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 课程编号 1502760001-01  题目类型 实验4 | |  |  |  | | --- | --- | --- | | **得分** | **教师签名** | **批改日期** | |  | 冯禹洪 |  | |

**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： 腾班**

**指 导 教 师： 冯禹洪**

**报告人： 叶茂林 学号： 2021155015 班级： 腾班**

**实 验 时 间： 2023.5.27**

**实验报告提交时间： 2023.5.29**

**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**

先在root权限下安装一个32位的库，如图1所示。

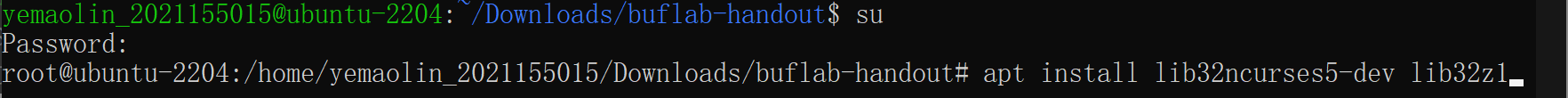


图1

然后安装sendmail，如图2所示。



图2

默认情况下，Linux内存地址会随机化，如图3所示。

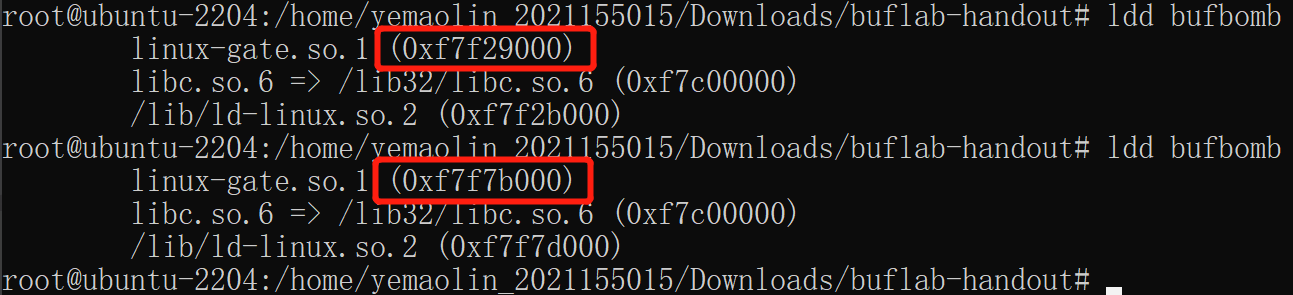


图3

为了实现简单缓冲区内存攻击，我们需要关闭Linux的内存地址随机化，如图4所示。

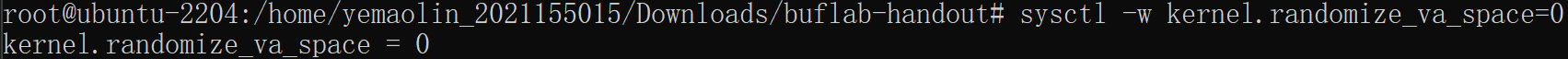


图4

这样Linux内存地址将不会随机化，如图5所示。

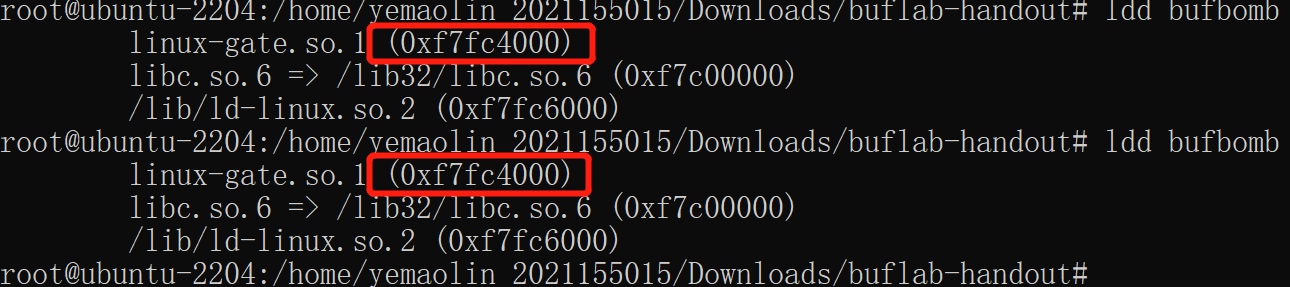


图5

**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。getbuf()函数代码如下：

***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**

利用objdump查看getbuf函数的汇编代码，如图6所示。

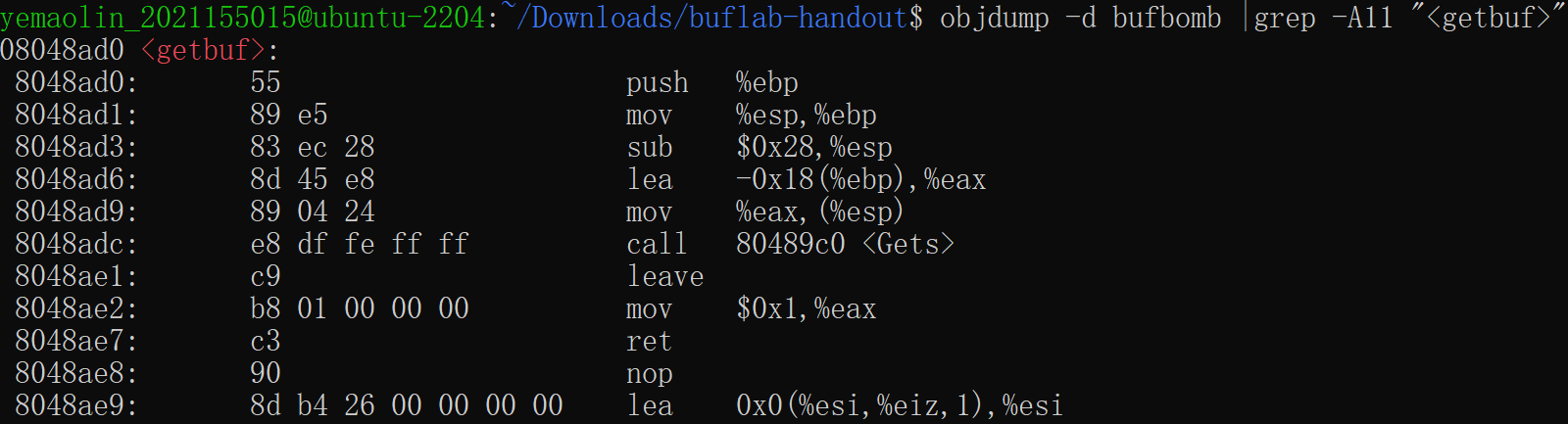


图6

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。

用objdump查看smoke函数的地址，如图7所示，地址为08048eb0。

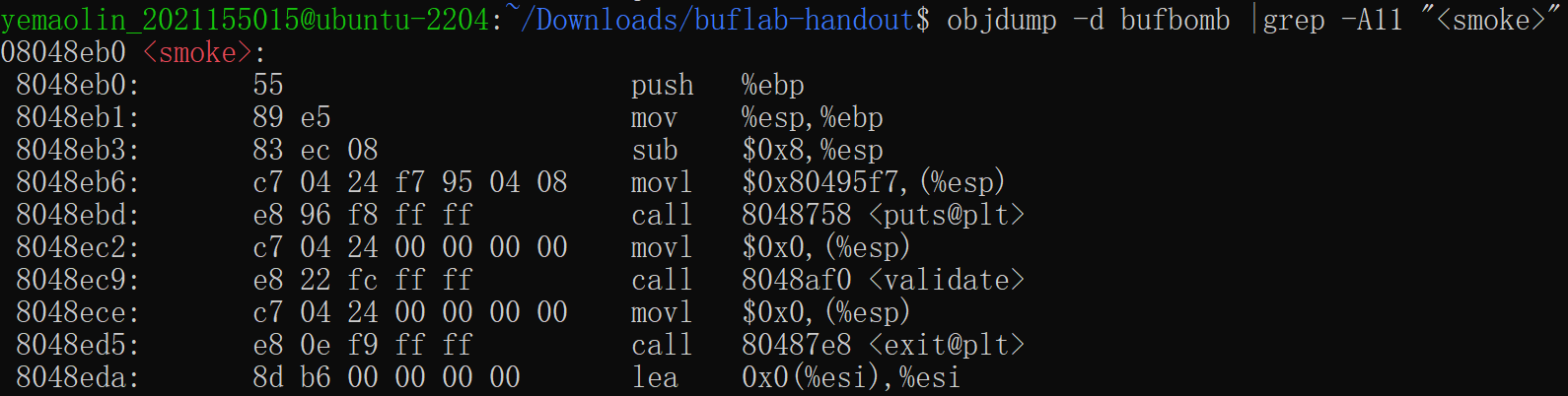


图7

又因为是小端机，所以低位在低地址，应该覆盖为b08e0408，其他字符用学号填充，用vim文本编辑器保存在exploit.txt文件里，如图8所示。

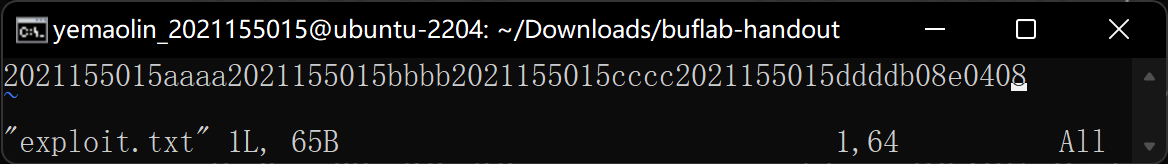


图8

**1.3 最终结果截图**

将exploit.txt文件由sendstring通过管道输入到bufbomb的标准输入设备中，如图9所示，成功调用了smoke函数。

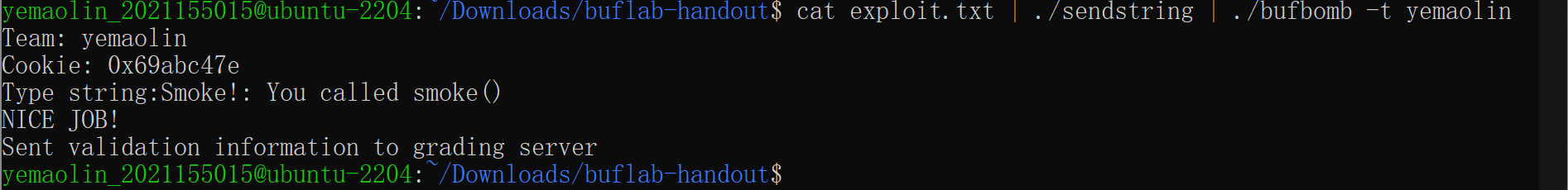


图9

**步骤2 返回到fizz()并准备相应参数**

**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，如图10所示。

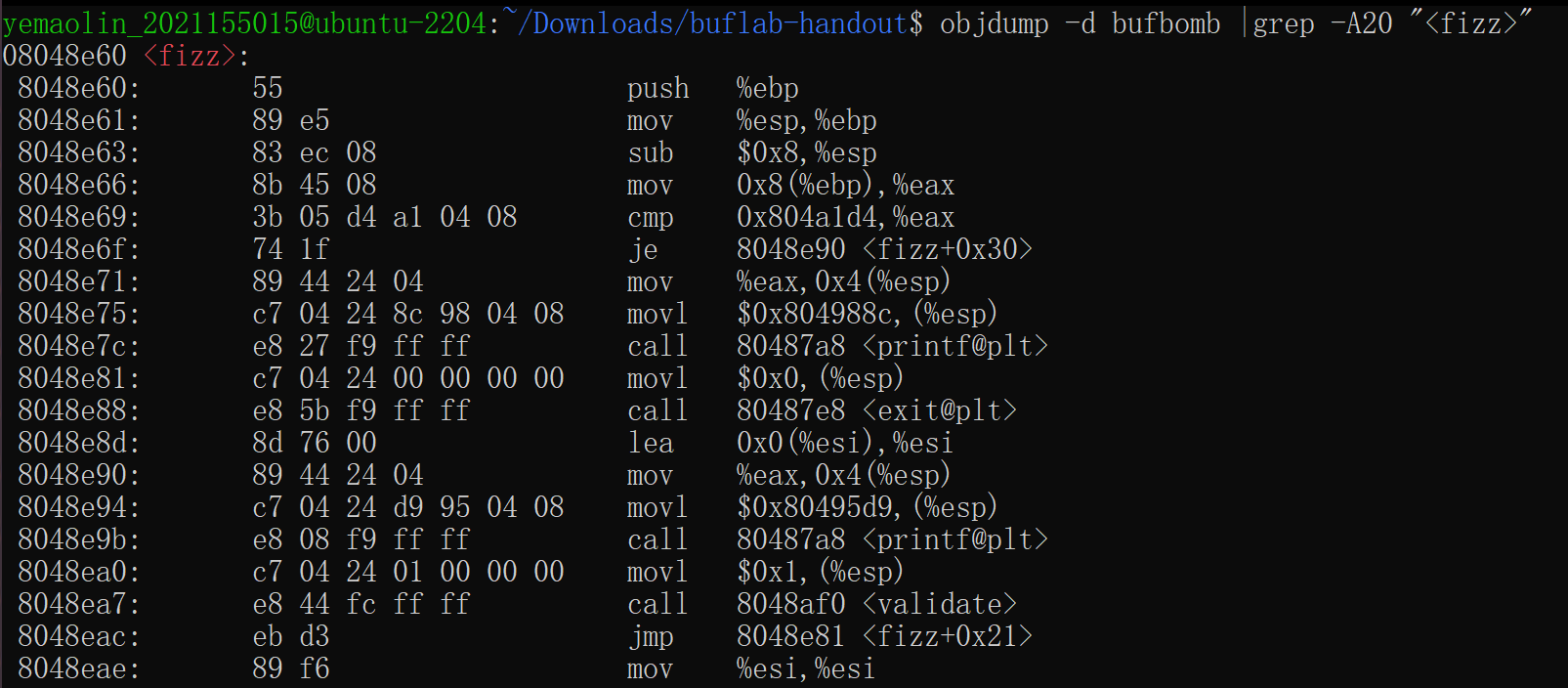


图10

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。

用objdump查看fizz函数地址，如图11所示，地址为08048e60。

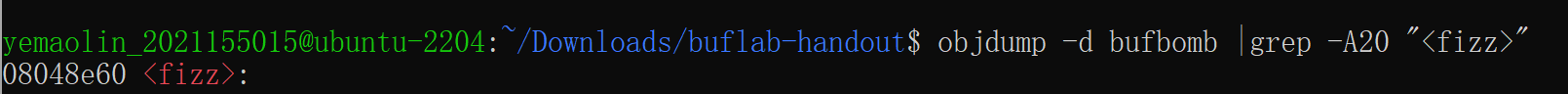


图11

用makecookie制作自己的cookie值，如图12所示。

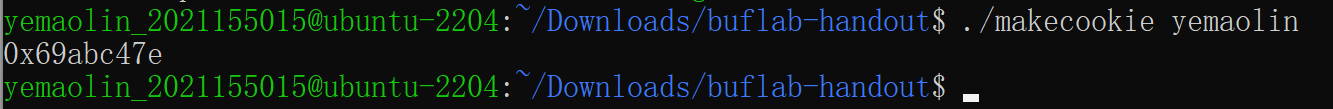


图12

又因为是小端机，所以低位在低地址，fizz地址应为608e0408，cookie值为应为7ec4ab69，其他字符用学号填充，用vim文本编辑器保存在exploit.txt文件里，如图13所示

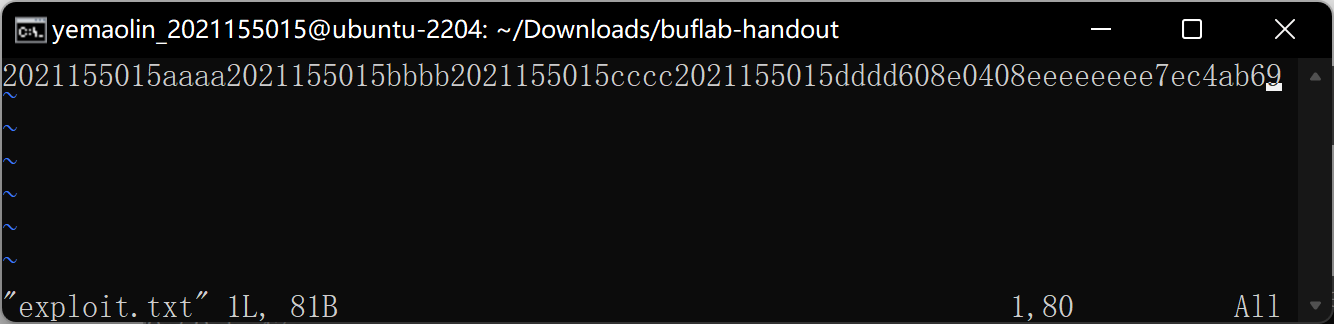
****

图13

**2.3 最终结果截图**

将exploit.txt文件由sendstring通过管道输入到bufbomb的标准输入设备中，如图14所示，成功调用了fizz函数，并传参成功。

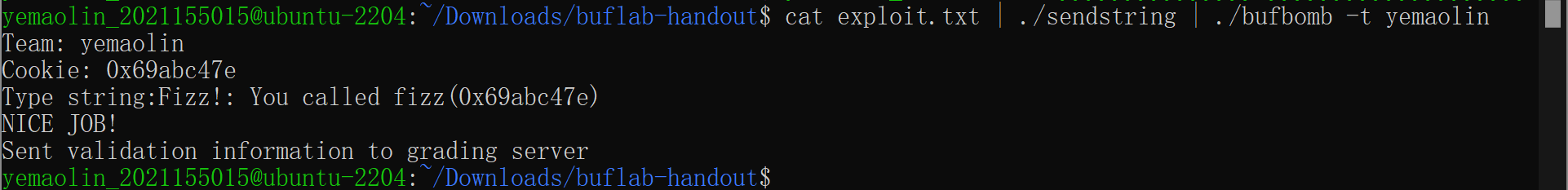


图14

**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，这一步我们在一开始已经做了。

用objdump查看bang()函数的汇编代码，如图15所示。

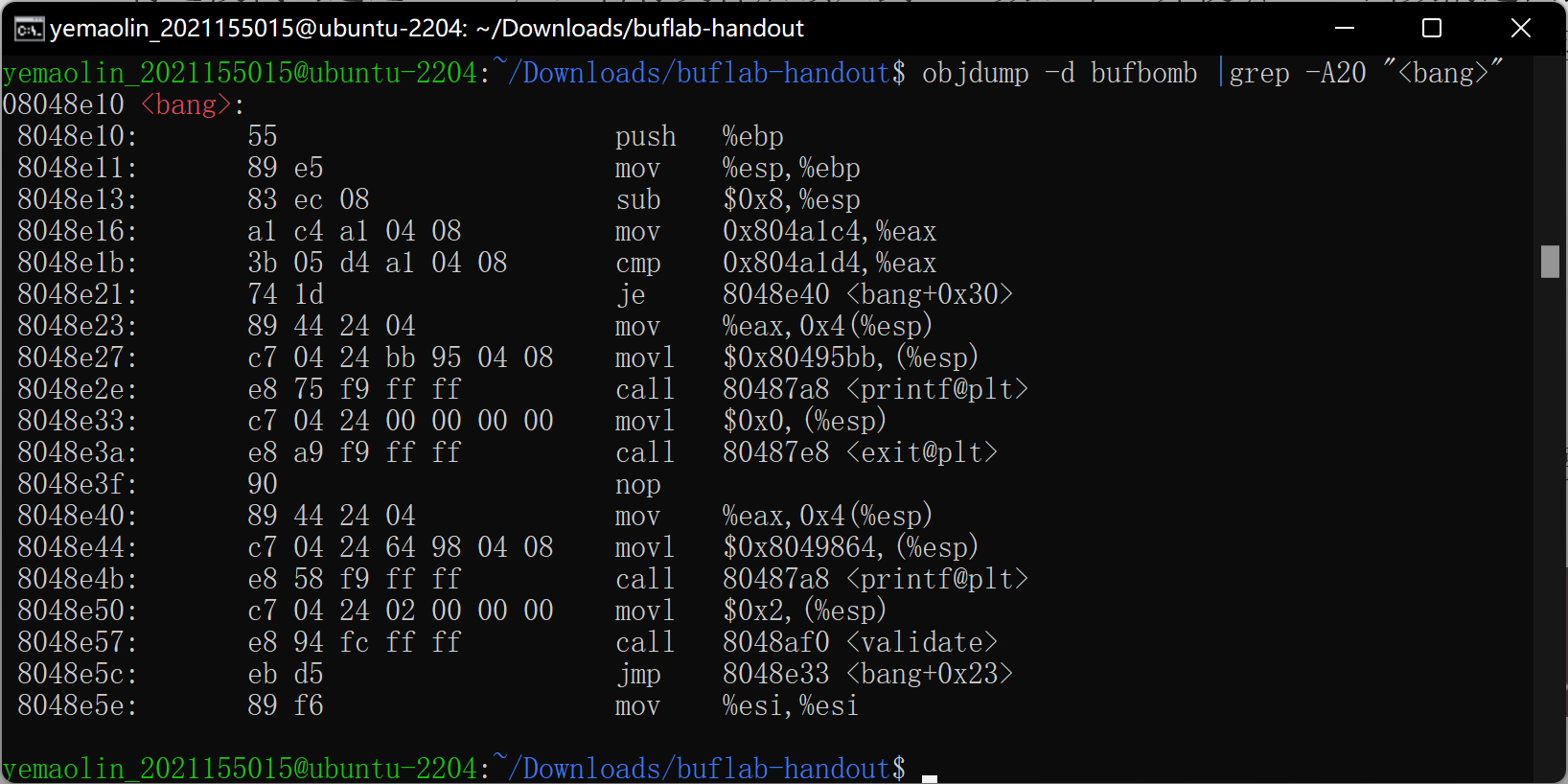
****

图15

如图16所示，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。

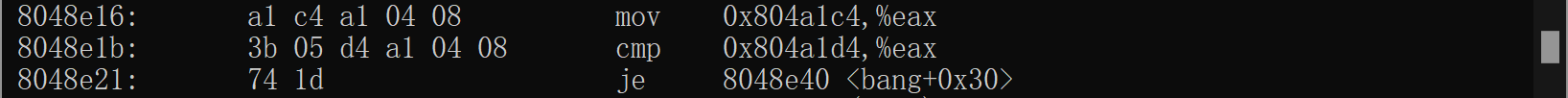


图16

用GDB查看这两个地址的值，如图17所示。



图17

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，如图18所示。

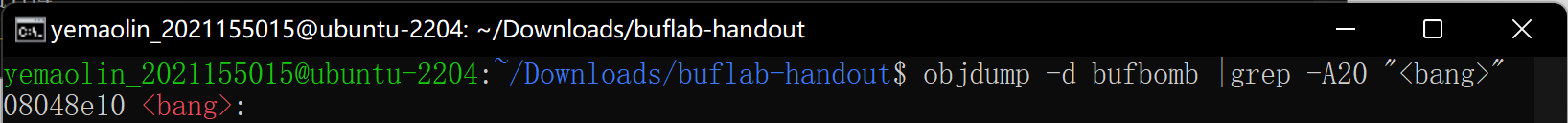
****

图18

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。

用vim编写汇编代码code.s文件，如图19所示。

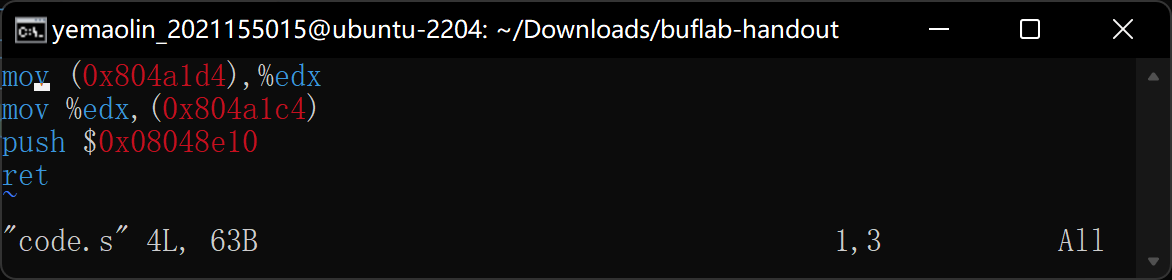


图19

用gcc编译汇编代码，用objdump反汇编输出到code.txt中，如图20所示。

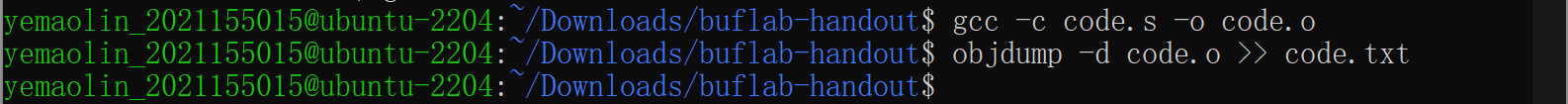


图20

用vim查看汇编代码对应的机器码，如图21所示。

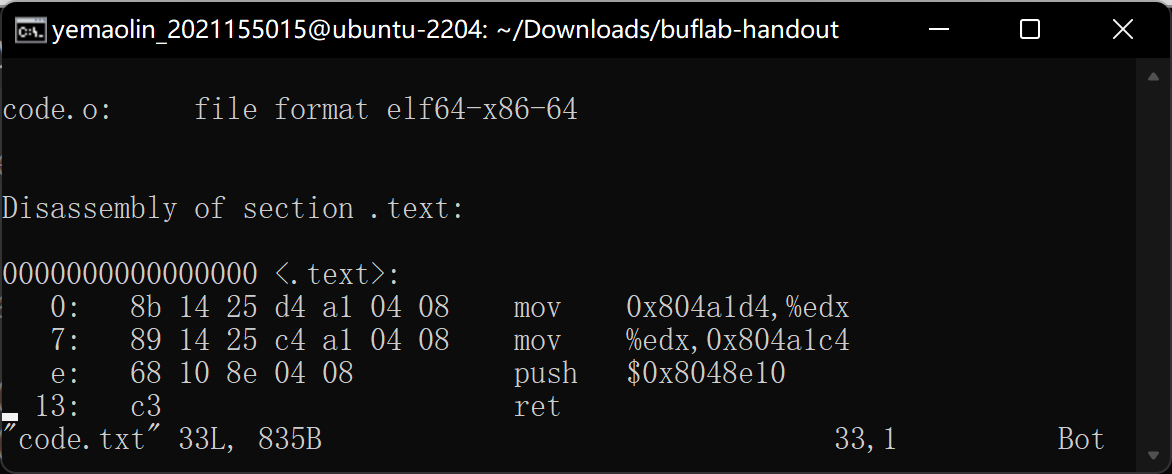


图21

为了使得我们写的代码可以执行，我们准备将这段机器码放进buf数组及缓冲区中，因此我们还需要知道buf数组的首地址，用GDB调试查看buf数组的首地址，如图22所示，则buf数组首地址为0xffffbd38-0x18,即0xffffbd20。

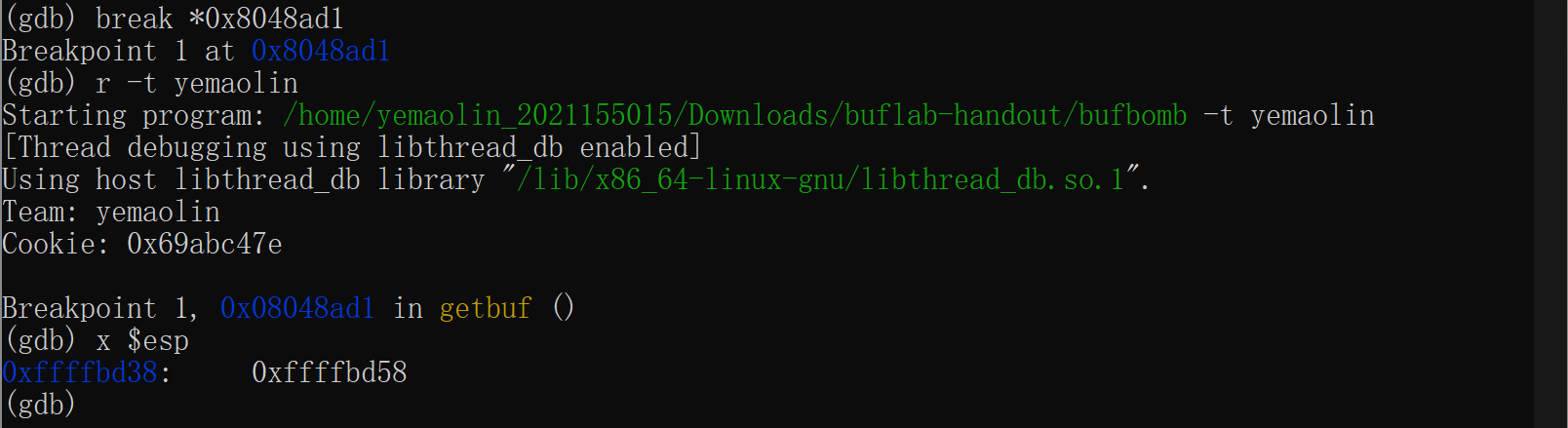
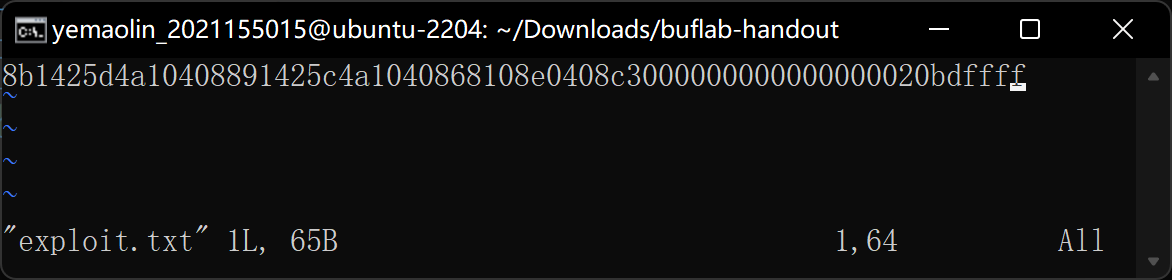


图22

又因为是小端机，数组首地址写为20bdffff，其他字符用0填充，如图23所示，用vim文本编辑器保存在exploit.txt文件里。

图23

**3.3 最终结果截图**

将exploit.txt文件由sendstring通过管道输入到bufbomb的标准输入设备中，如图24所示，成功修改全局变量global\_value的值为自己的cookie值，再返回到band()函数。

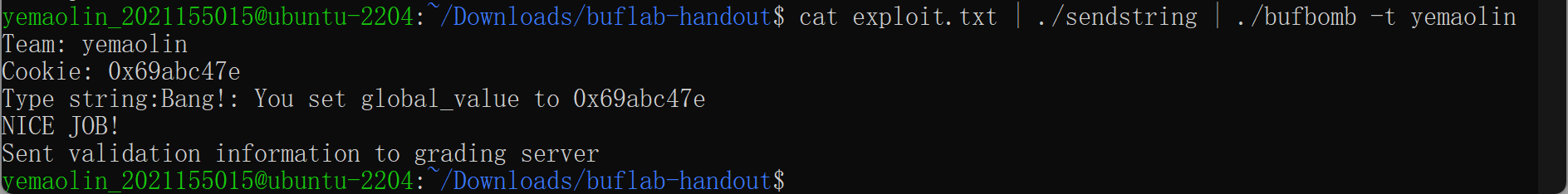


图24

**五、实验总结与体会**

在本次实验中，我通过分析汇编代码和相应的栈帧结构，运用GDB调试工具和objdump反汇编工具，利用gets函数未进行缓冲区溢出保护的漏洞，在程序执行完getbuf函数返回时通过覆盖程序返回地址来实现攻击。

在这个实验中，我不仅提高了自己的安全意识，同时也对于计算机程序安全性有了更深入的理解。通过这个实验，我也进一步掌握了GDB调试工具和objdump反汇编工具的使用技巧和调试过程。此外，我还学会了如何依靠自己的实践经验去解决问题，这将对我的未来学习和工作有很大的帮助。

通过本次实验，我我更好地理解了计算机底层架构中的栈、程序函数调用中的参数传递机制和缓冲区溢出攻击方法。同时，通过实践中的调试和反汇编，我掌握了一些常用的调试和分析工具及技巧，这将对我的未来学习和工作都是非常有益的。

|  |
| --- |
| **指导教师批阅意见：**  **成绩评定：**  指导教师签字：冯禹洪    2023年 月 日 |
| 备注： |