

**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 ：**

**专业班级 ：**

**学 号 ：**

**姓 名 ：**

**指导教师 ：**

**年 月 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。

**5、第4级 kaboom**

**一个函数的栈帧的地址通常并不是固定的，随程序运行实例的不同而不同，即每次运行有一个随机的、不固定的值。在此种条件下，要求getbuf（getbufn）函数返回cookie的值，而不是返回值1，并且能正确回到调用函数处继续执行。**

**三、实验记录及问题回答**

**（1）实验任务的实验记录**

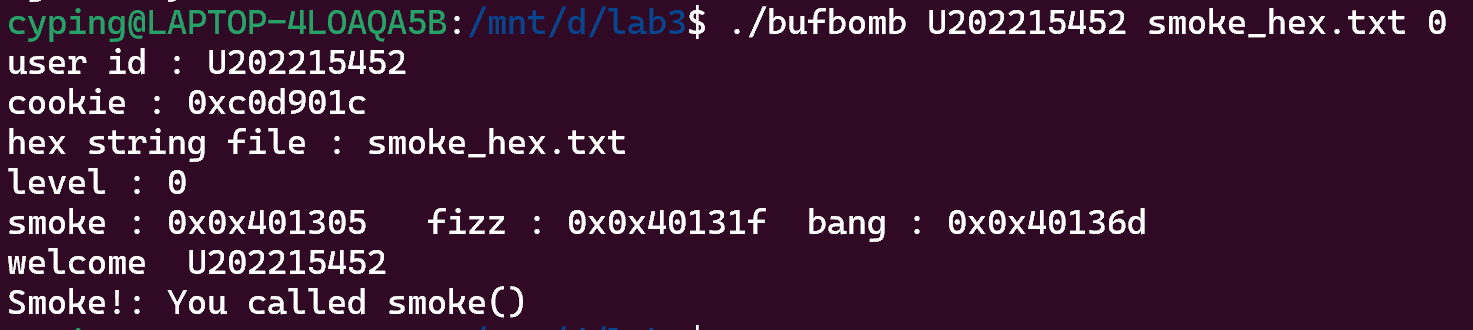


图1 第0级smoke通关截图

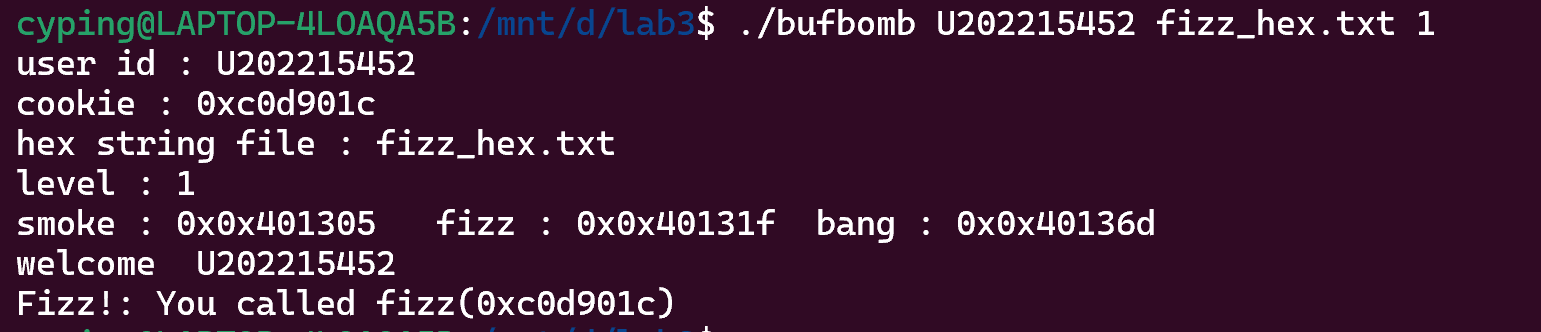


图2 第1级fizz通关截图

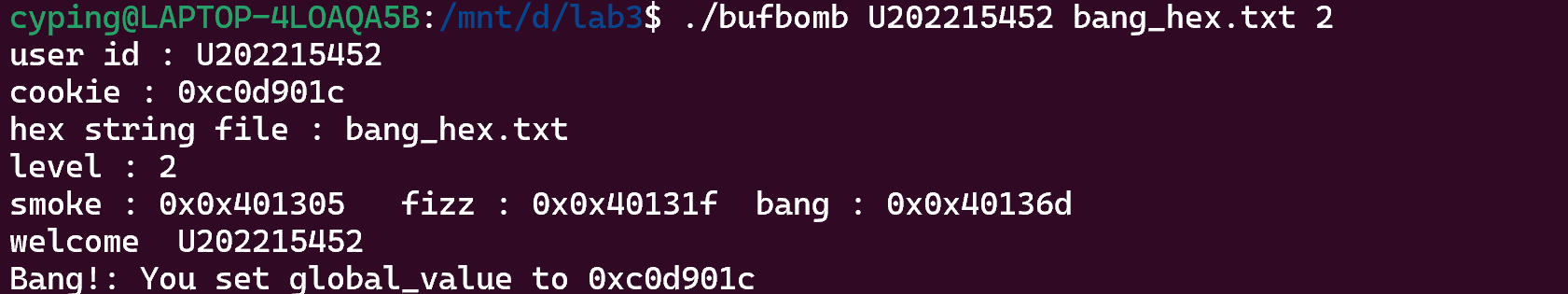


图3 第2级bang通关截图

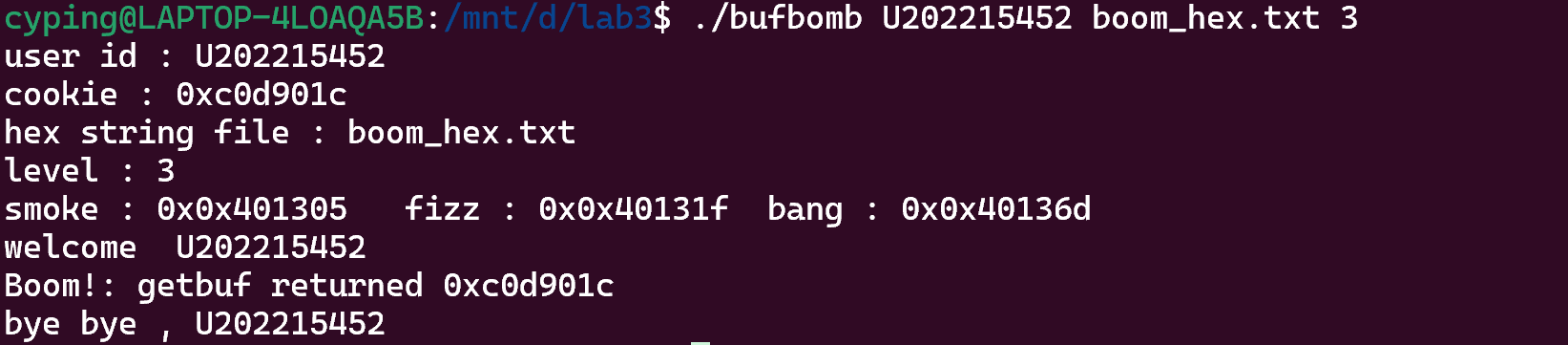
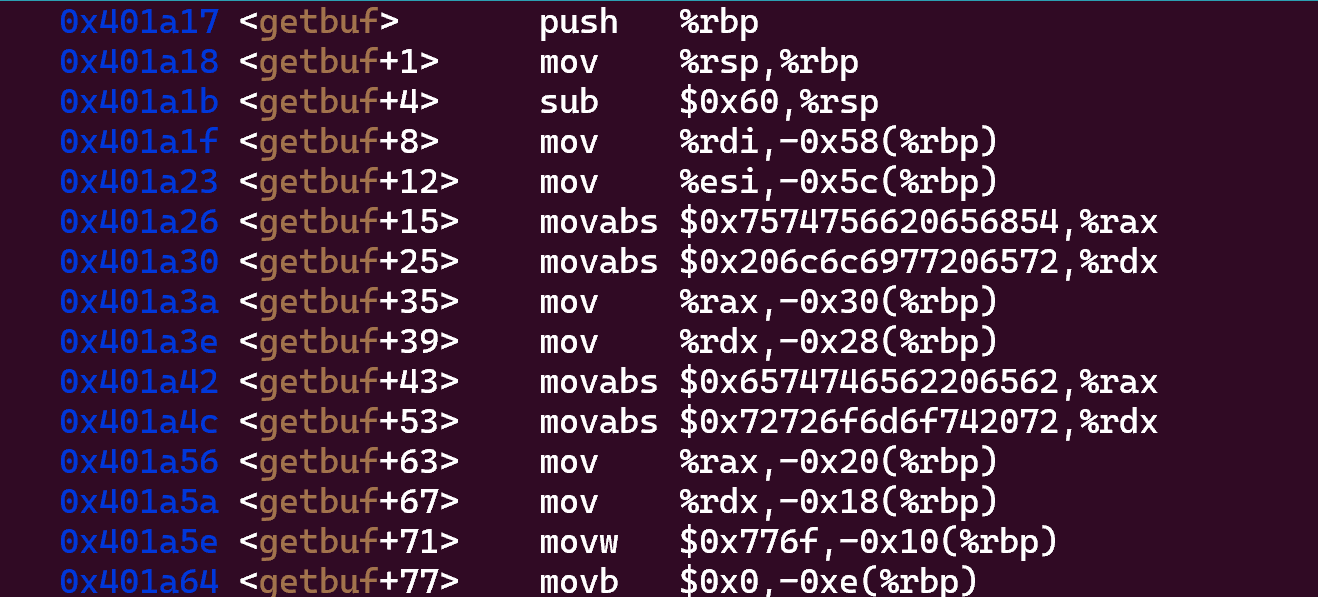


图4 第3级boom通关截图

**（2）缓冲区溢出攻击中字符串产生的方法描述**

**第0级 smoke**

首先使用layout指令查看getbuf的反汇编代码，由<getbuf+4>可知为getbuf函数在栈内分配了0x60的空间，又由<getbuf+88> lea -0x50(%rbp), %rax可知为输入字符串分配了0x50即80字节的空间。攻击字符串的功能应该是覆盖getbuf里面的buf数组，进而溢出覆盖rbp和rbp下方的返回地址。所以要输入的字符串总共为80+8+8=96个字节。



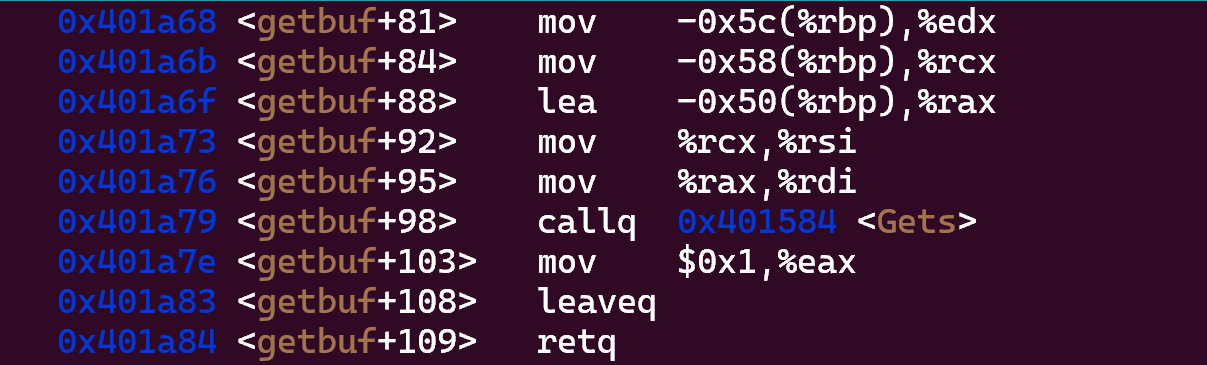


图1-1. getbuf函数反汇编代码

其中，最后8个字节对应返回地址，应该修改为smoke函数的地址，我们可以查看smoke函数的地址，为0x401305，由于地址要扩展为8个字节且为小端存储，为“05 13 40 00 00 00 00 00”。所以最终的攻击字符串（存储在smoke\_hex.txt文件中）如下所示。

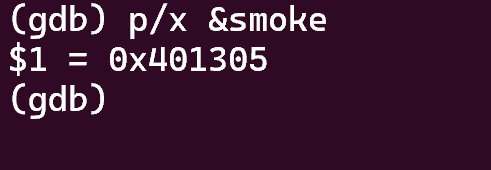


图1-2. 查看smoke函数地址

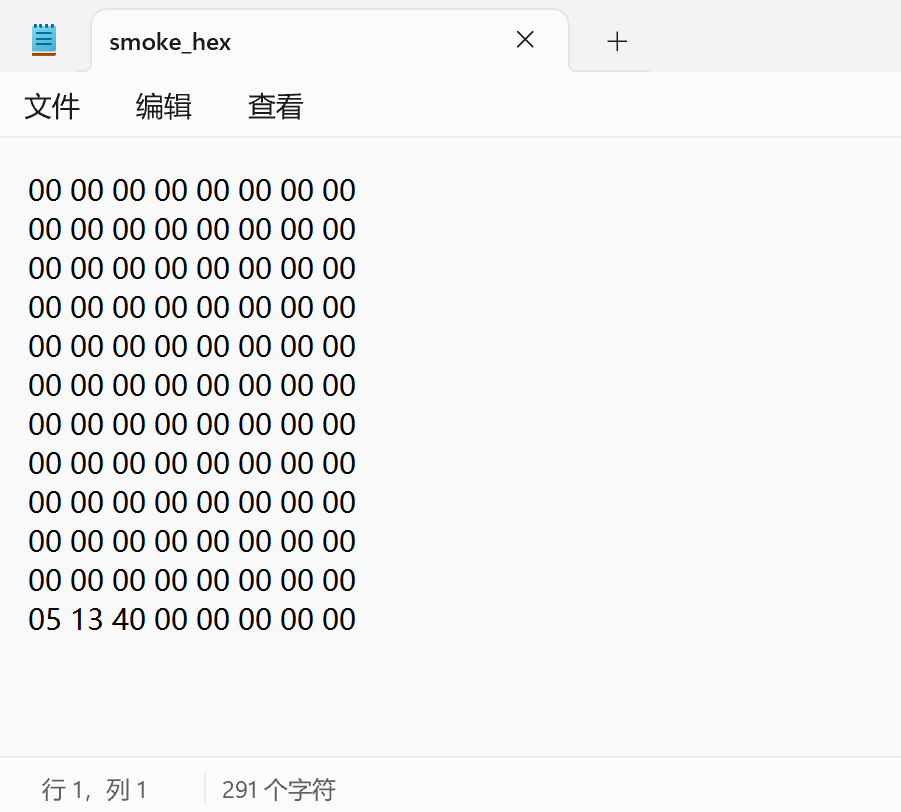


图1-3. smoke\_hex.txt文件内容

**第1级 fizz**

对于 64位程序，使用的是寄存器%edi 来传递int型的参数val。直接修改%edi 是很困难的。一种巧妙的办法是，不要跳到fizz函数的起始地址，而直接跳到if (val==cookie) 处。此时val 的值已存放在栈中地址为-0x4（%rbp）处。只要%rbp-0x4 与cookie 对应同一个单元，则if的条件就会成立。如果只是简单绕开if语句的检查，之后的打印语句打印val的值会发现它与cookie并不一致。

因此，这一关在前一关的基础上，最后8个字节即返回地址应该为if(val==cookie)语句的地址，而倒数第2个8个字节对应的%rbp应该满足%rbp-4与cookie对应同一单元这一条件。

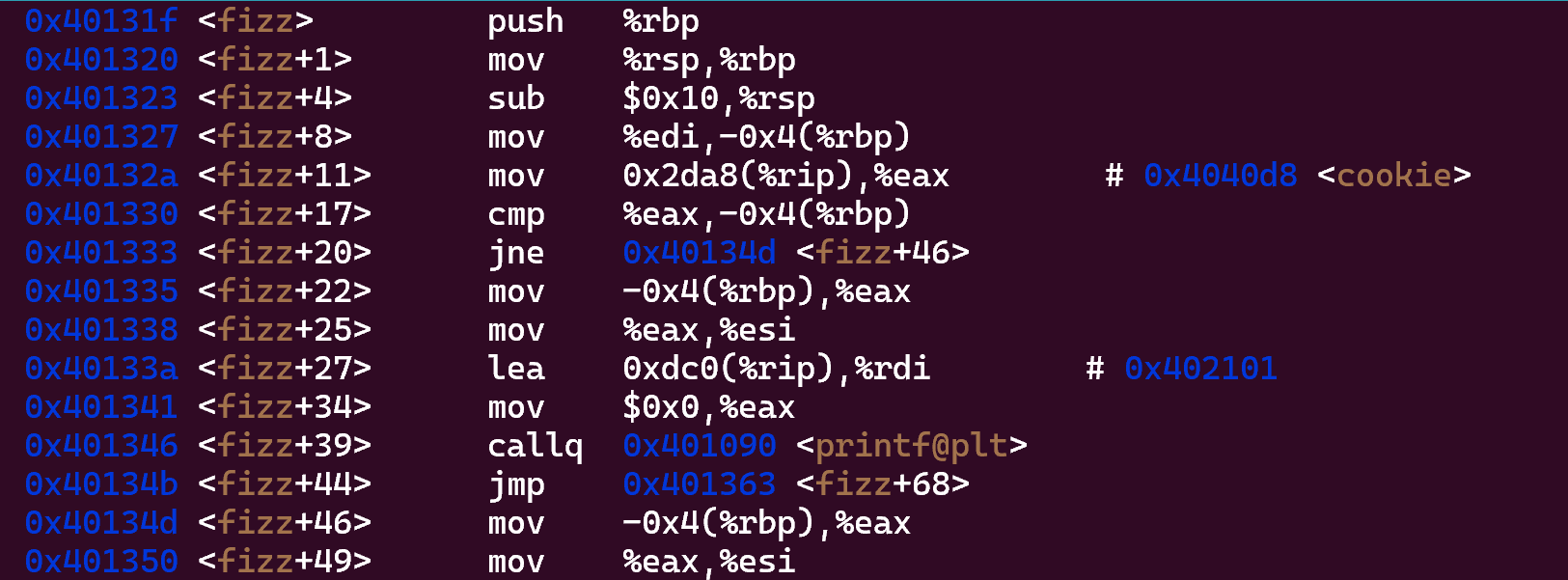


图2-1. fizz函数反汇编代码

调用查看fizz函数的反汇编代码可以看到<cookie>的地址为0x4040d8，要使得%rbp-4后对应cookie的地址，则%rbp应存入0x4040dc。同时，从该图中我们可以得知要跳转的地址为0x40132a，所以最后8个字节存入的内容为0x40132a。

最终的攻击字符串如下。

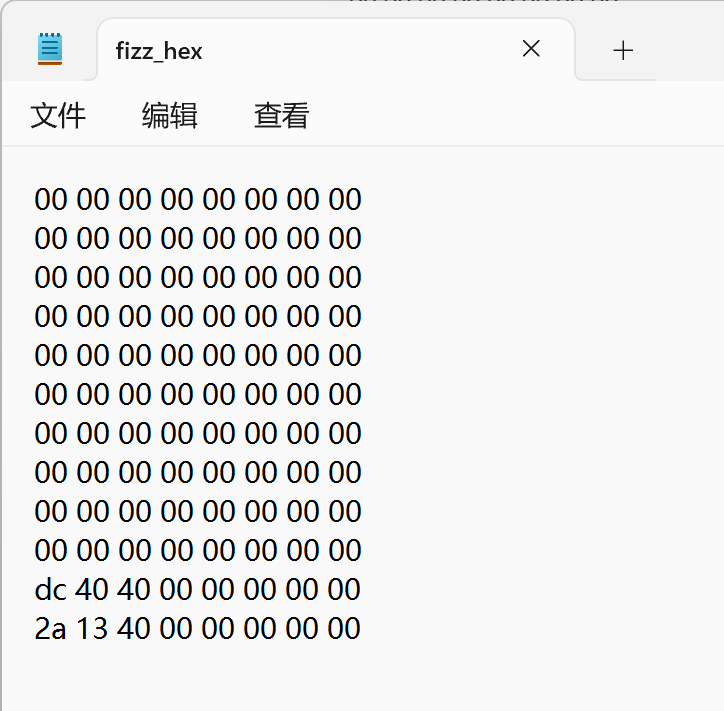


图2-2. fizz\_hex.txt文件内容

**第2级 bang**

全局变量global\_value 并没有存储在栈中，要真正的修改它的值为cookie，无法像前一关一样直接使用攻击字符串来更改，只能编写一段代码来修改global\_value的值，并且要让这段代码得到执行。

首先通过观察反汇编代码，可以知道<global\_value>的地址为0x4040dc，<cookie>的地址为0x4040d8，可以查看该地址对应的值，也可以通过前几关的输出得到cookie的值为0xc0d901c。

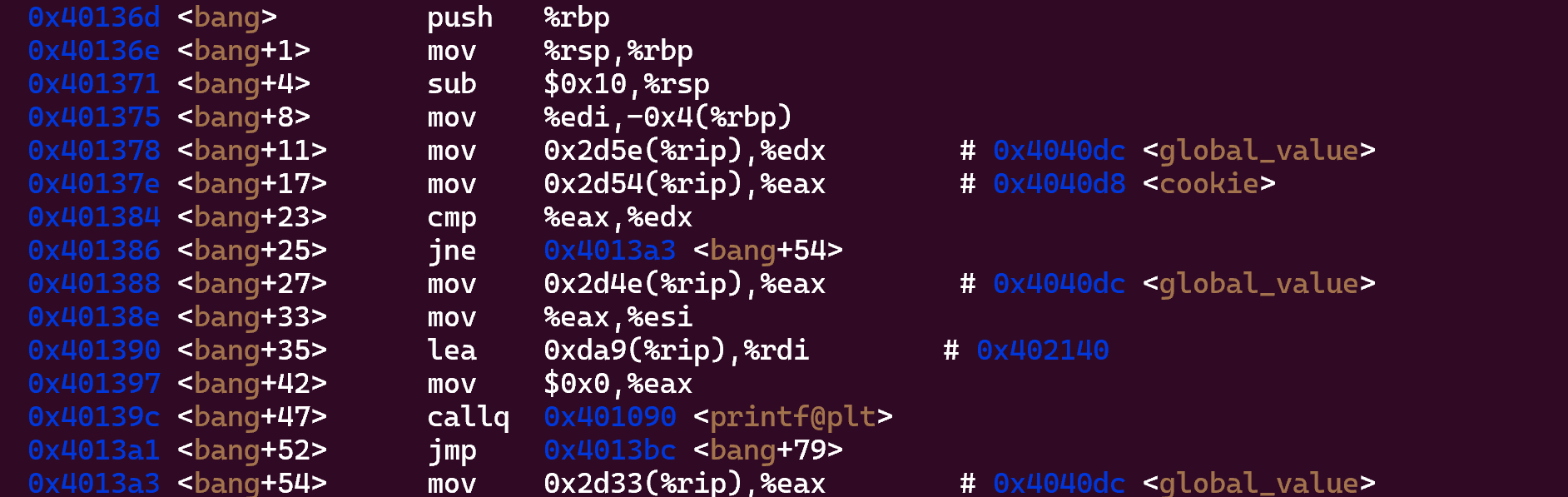


图3-1. bang函数反汇编代码

在bang.S文件中写出如下语句：

movl $0xc0d901c,0x4040dc # 将cookie的值赋给global\_value的地址

push $0x40136d # 将bang的首地址入栈

ret # 返回

将bang.S文件编译生成bang.o，得到指令编码放入攻击字符串，并将字符串最后8个字节修改为buf的首地址，结果发现在跳入bang函数后会报错./stdio-common/vprintf-internal.c:No such file or director。

经过进一步排查，发现只有在跳到<bang+0>才会printf报错，调到<bang+1>就可以了。猜测可能是因为在进入printf函数后，system会检查每个栈帧的返回地址是否有效，所以当跳到<bang+0>时，第一次push的%rbp被当作了返回地址。所以，我们最好在攻击指令中push一个合法的返回地址，如test( )函数中call getbuf的下一句。

所以修改bang.S的内容如下：

movl $0xc0d901c,0x4040dc # 将cookie的值赋给global\_value的地址

push $0x401474 # 将合法地址入栈

mov $0x40136d,%rdx # 将bang函数首地址移入%rdx

jmp \*%rdx # 跳到bang的首指令

通过gcc将其编译成机器代码后，使用objdump将其反汇编获得需要的二进制机器指令字节序列如下图所示：

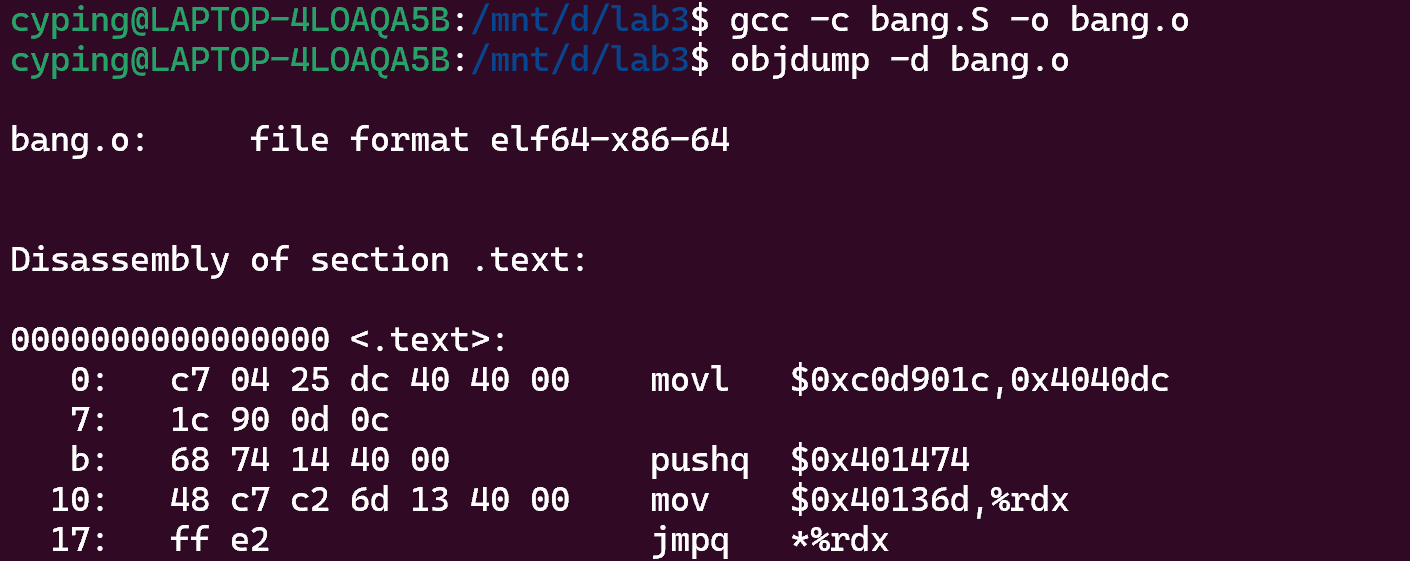


图3-2. bang.S文件编译生成目标文件

接下来就是要找到buf的首地址，在gdb模式下，我们可以通过在getbuf函数调用Gets前设置断点找到buf的首地址，如图，此时%rax中存储的即为buf的首地址，查看可得为0x7fffffffe3c0，于是将其写入攻击字符串的最后8个字节，在gdb模式下可以得出正确的结果。

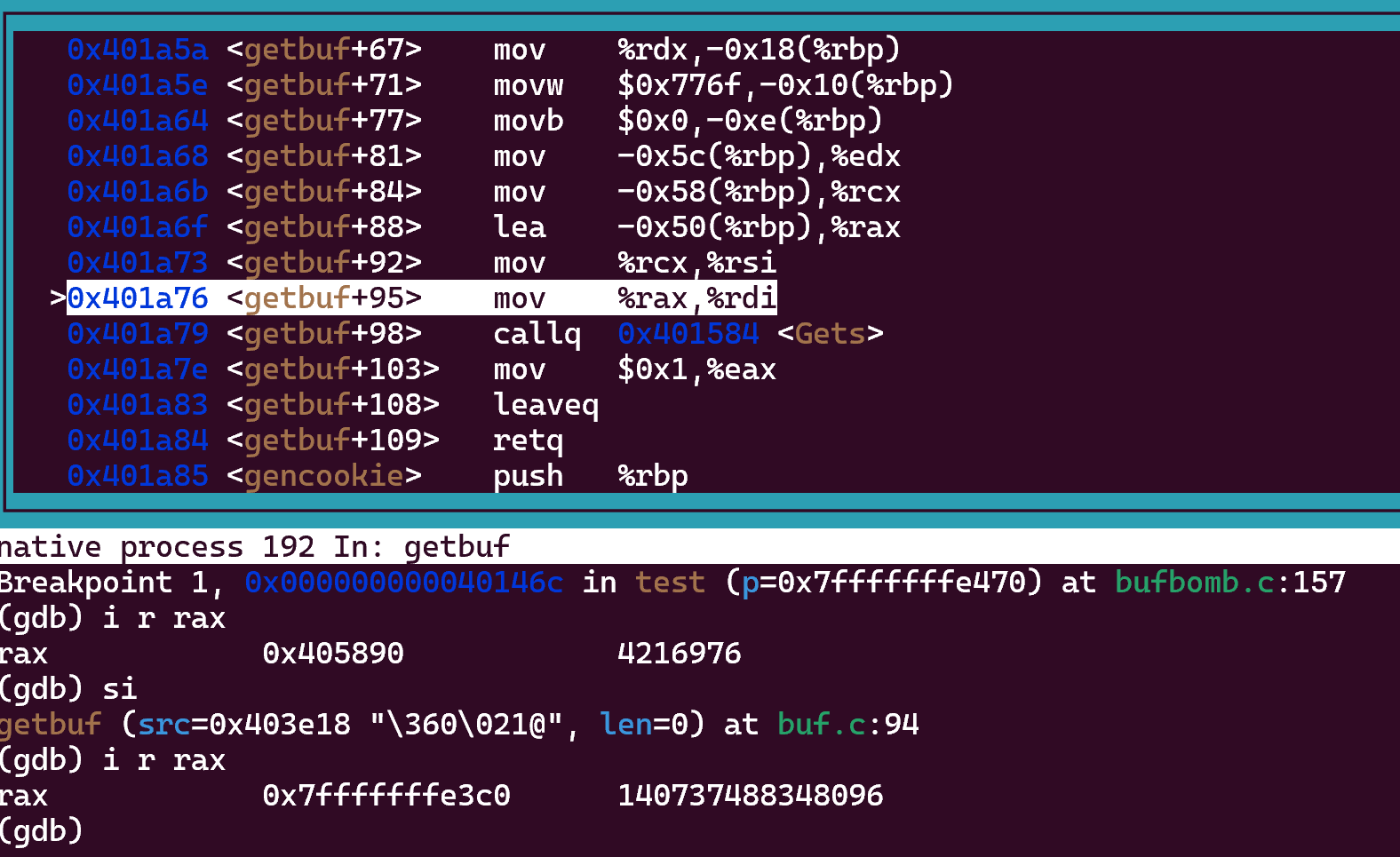


图3-3. 查看buf数组载gdb模式下的首地址

但是在直接运行模式下，buf的地址发生了改变，这时我们可以通过在getbuf函数中增加一个输出buf地址的语句来获得。

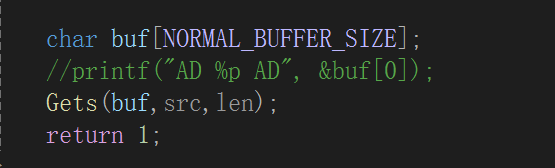


图3-4. 查看buf数组在直接运行模式下的首地址

由于不知道buf在直接运行模式下的首地址具体是多少，直接运行这一关的文件会得到Illegal instruction的结果，所以可以通过运行前几关的文件得到。如图，可以看到，buf首地址为0x7fffffffe400。

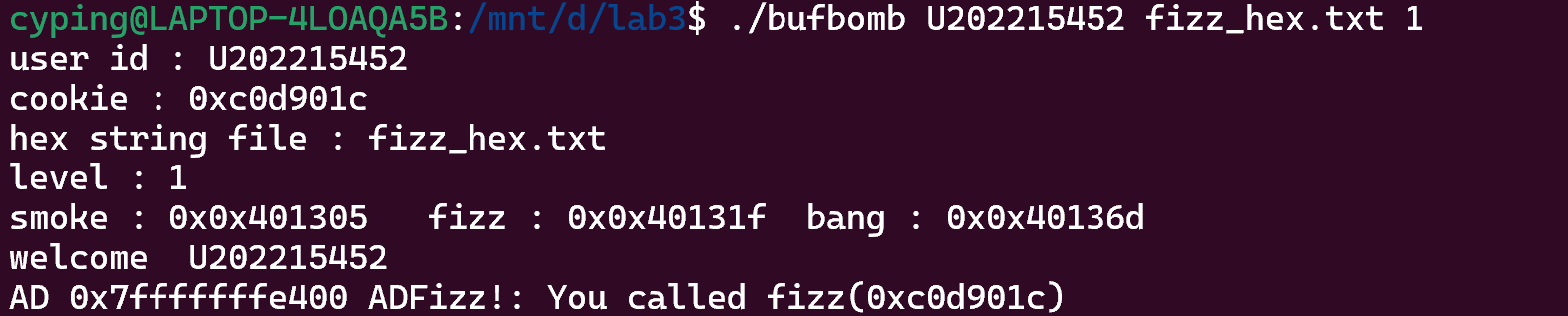


图3-5. 运行查看buf数组在直接运行模式下的首地址

除此以外，为了保证这个地址不会改变，我们还应该关闭Linux上的地址空间随机化。Linux上地址空间随机化（Address Space Layout Randomization）（ASLR）分为0/1/2三级，可以看到初始状态下为2级，即完全的随机化，应该将其修改为0级，即没有随机化，关闭ASLR。

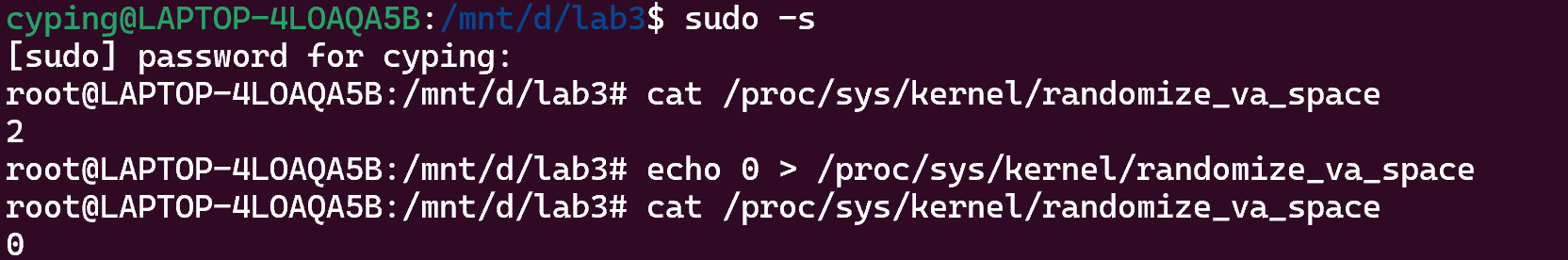


图3-6. 关闭ASLR

这样，我们就可以保证buf首地址不发生改变，依旧根据小端写入的原则，可以得到bang\_hex.txt文件内容如下（这个攻击字符串适用于自动运行模式）：

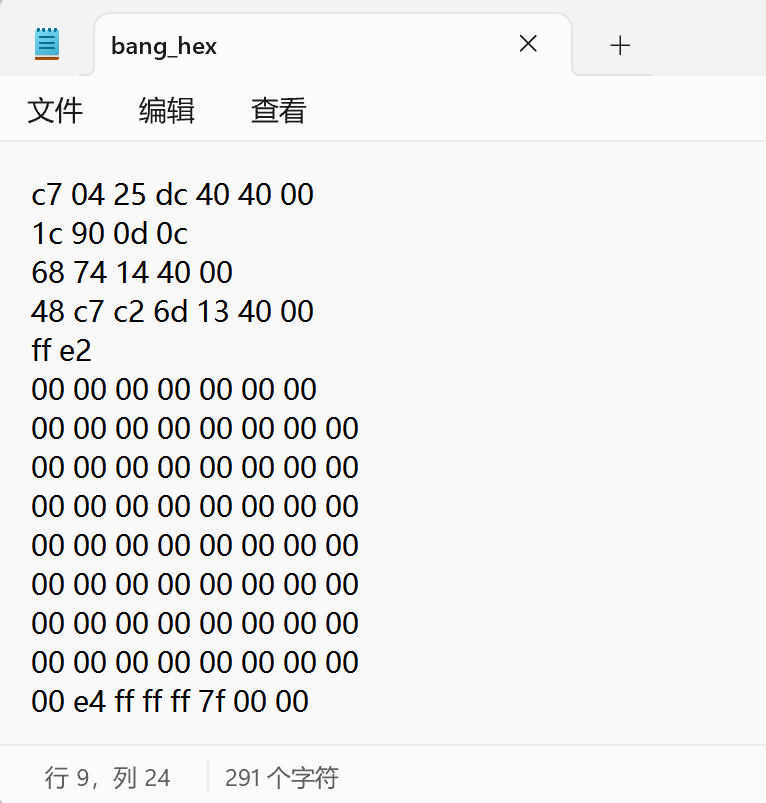


图3-7. bang\_hex.txt文件内容

**第3级 boom**

要实现boom的无感攻击，要实现以下几点：

（1）将%eax的值（即返回值）设置为cookie的值

（2）恢复%rbp为执行getbuf函数前的值

（3）将调用getbuf后下一条指令压栈，然后返回

第一步只需要知道cookie的值，我们在前面已经知道其值为0xc0d901c。第二步需要在getbuf前设置断点。

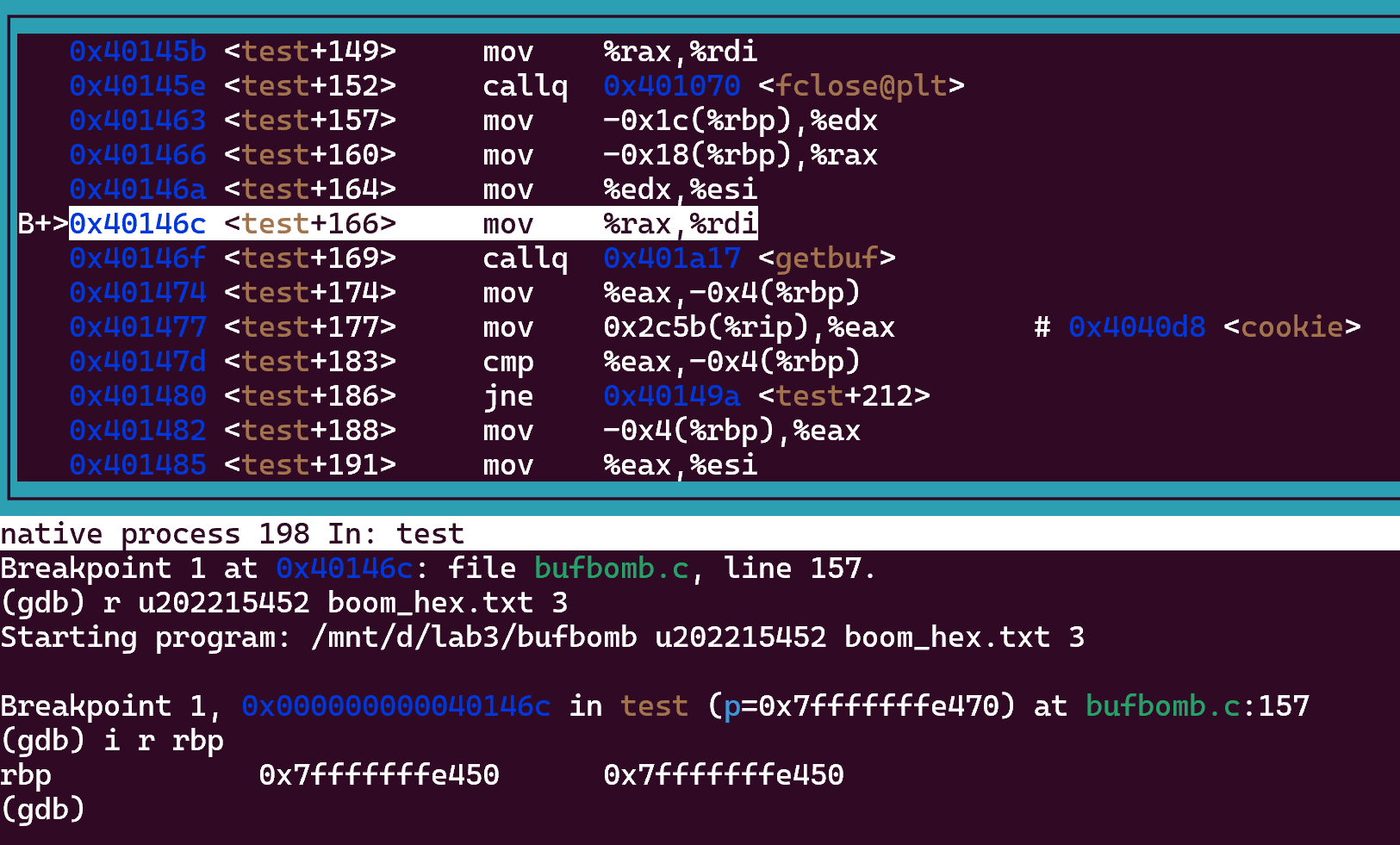


图4-1. 查看%rbp内的值

如图，可以看到在执行getbuf函数前%rbp的值为0x7fffffffe450，从这张图也可以看到执行完getbuf下一条指令地址为0x401474，所以可以编写boom.S文件内容如下：

movl $0xc0d901c,%eax # 将cookie的值写入%eax

mov $0x7fffffffe450,%rbp # 恢复%rbp的值

push $0x401474 # 将getbuf下一条指令地址值压入栈

ret # 返回

通过gcc将其编译成机器代码后，使用objdump将其反汇编获得需要的二进制机器指令字节序列如下图所示：

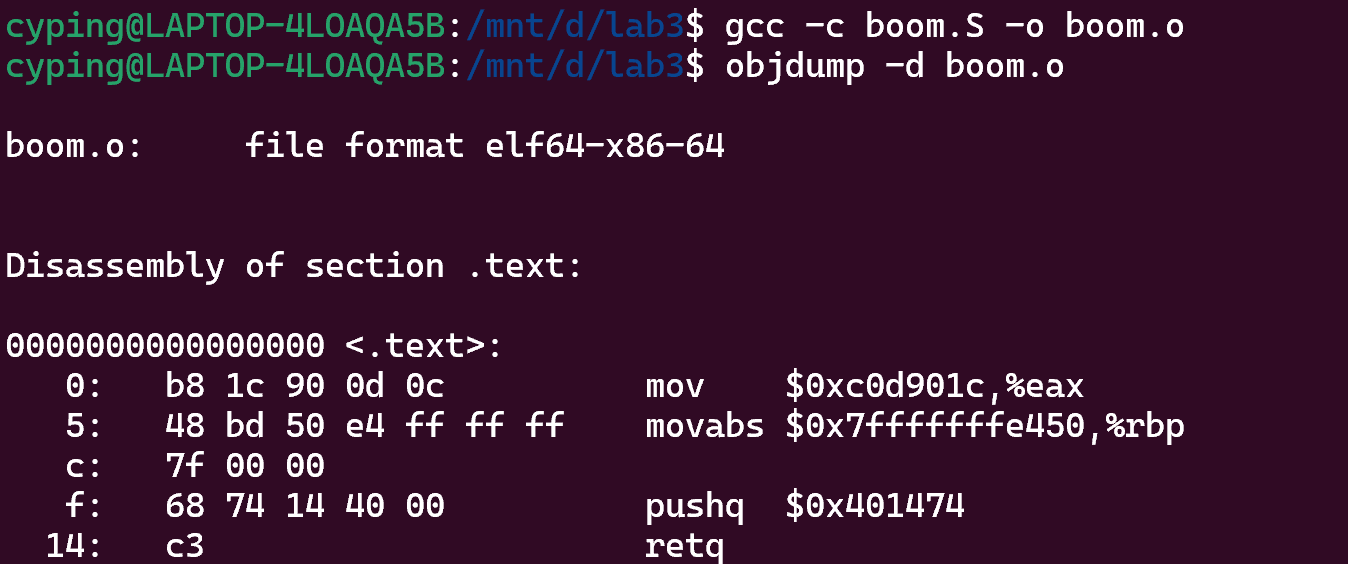


图4-2. boom.S文件编译生成目标文件

与前一关同理写入将机器指令字节码写入攻击字符串，并且将buf首地址写进最后8个字节，结果得到Segmentation fault，猜测是因为gdb内外模式下%rbp的值会发生变化。考虑到前一关buf的地址也存在这样的情况，我们可以对比两个值寻找规律：

buf：在内0x7fffffffe3c0；在外0x7fffffffe400

rbp：在内0x7fffffffe450；在外0x7fffffffe490

于是就可以得到在自动运行模式下的攻击字符串如下：

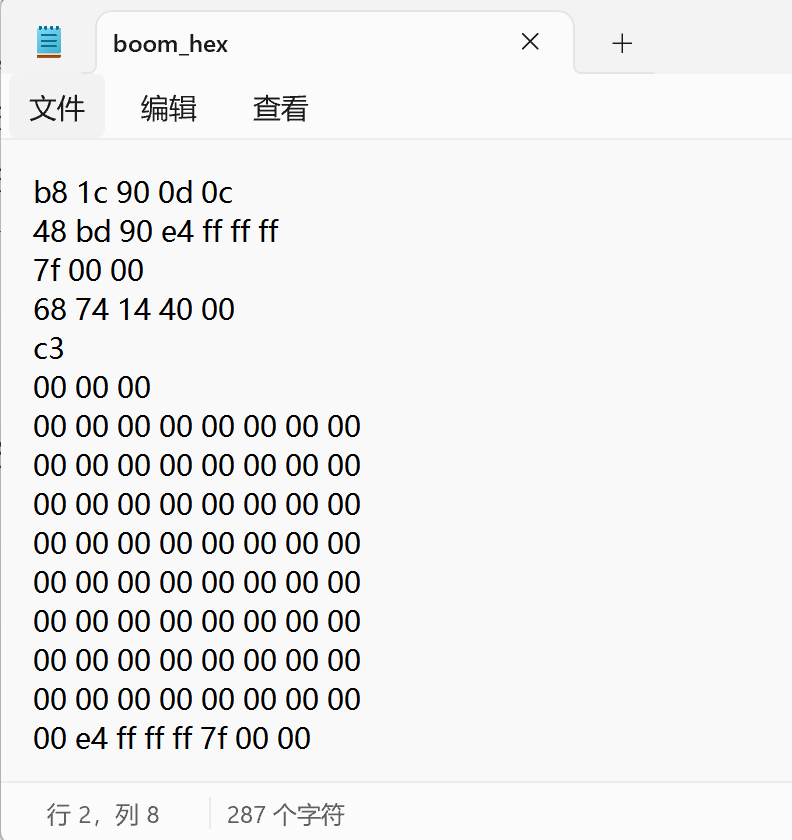


图3-7. boom\_hex.txt文件内容

1. **体会**

本次实验主要通过设计缓冲区攻击手段，进一步加深了我对栈帧结构的具体理解。与此同时，通过阅读大量的汇编代码的资料，加深了我对函数调用规则的具体理解；同时对于gdb、objdump、gcc等工具的频繁使用，使我熟练地掌握了这些工具的使用。为了完成实验目标，必须理解函数调用、返回、执行时，各个阶段下栈帧结构的变化。与PART 2类似，本次实验设计了多个关卡，难度逐渐递增，虽然很考验逻辑思考、代码阅读能力和对函数调用规则和栈帧结构的理解，也花费了大量的时间和精力，但也非常有挑战性和趣味性，使我在这些方面的能力在很大程度上得到了提高。

在本次实验中，贯穿始终的是对函数调用的栈帧结构的理解。为此在实验一开始就查找了相关资料。通用的栈帧结构如下：

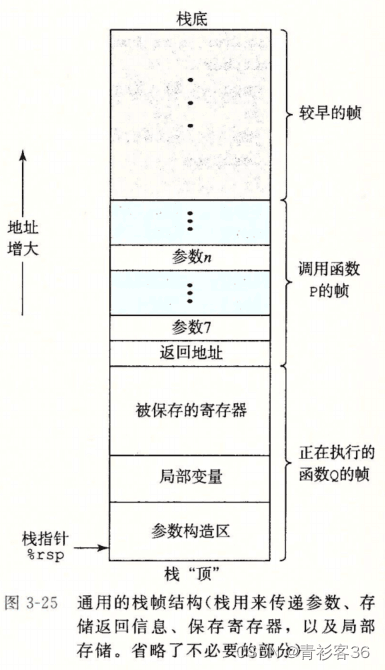


图4-1. 通用栈帧结构示意

而在本实验中，涉及到buf数组的结构如下所示：

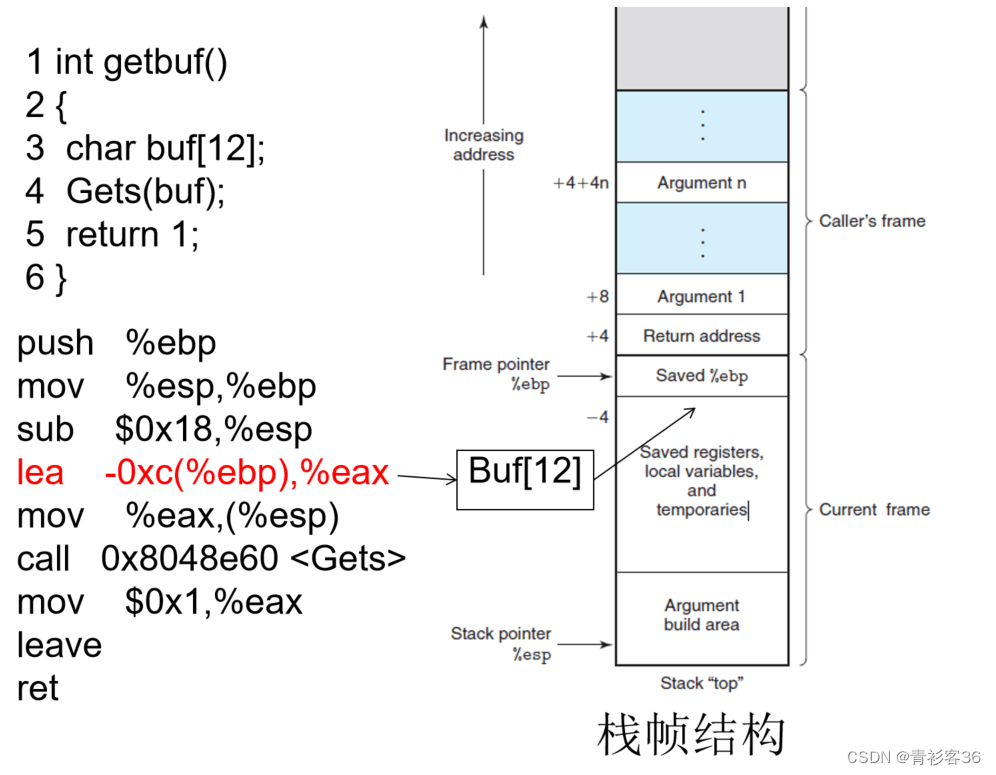


图4-2. 含有buf数组栈帧结构示意