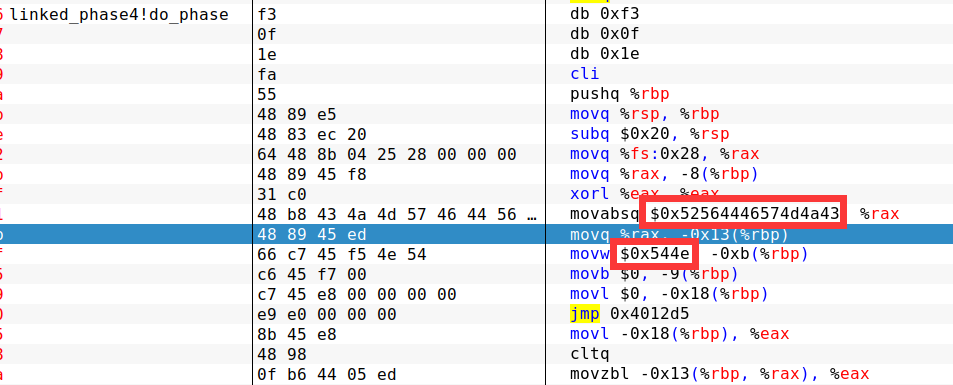


这时phase4的源码。这一关要求我们修改switch跳转表使得程序输出自己的学号。我们首先需要知道cookie数组中保存的元素值，就先用原始的phase4.o和main.o链接得到可执行文件，然后用EDB调试它，观察cookie数组的元素。



与上一关相同，汇编代码仍然采用立即数的方式给cookie数组赋值，由于是采用小端序存储，cookie数组中的元素分别为：{0x43, 0x4a, 0x4d, 0x57, 0x46, 0x44, 0x56, 0x52, 0x4e, 0x54}，用ASCII码转成字符就是：{‘C’, ‘J’, ‘M’, ‘W’, ‘F’, ‘D’, ‘V’, ‘R’, ‘N’, ‘T’}

直接运行一下这个程序，可以看到最后的输出是：

Nr}C012;`6



由此可以推断出之前的跳转表：

switch (c){

    case 'C':

{

        c = 'N';

        break;

    }

    case 'J':

    {

        c = 'r';

        break;

    }

    case 'M':

    {

        c = '}';

        break;

    }

    case 'W':

    {

        c = 'C';

        break;

    }

    case 'F':

    {

        c = '0';

        break;

    }

    case 'D':

    {

        c = '1';

        break;

    }

    case 'V':

    {

        c = '2';

        break;

    }

    case 'R':

    {

        c = ';';

        break;

    }

    case 'N':

    {

        c = '`';

        break;

    }

    case 'T':

    {

        c = '6';

        break;

    }

    }

序列化并转成ASCII码表示：

switch (c)

    {

    case 'C':

    {

        c = 0x4e;

        break;

    }

    case 'D':

    {

        c = 0x31;

        break;

    }

    case 'F':

    {

        c = 0x30;

        break;

    }

    case 'J':

    {

        c = 0x72;

        break;

    }

    case 'M':

    {

        c = 0x7d;

        break;

    }

    case 'N':

    {

        c = 0x60;

        break;

    }

    case 'R':

    {

        c = 0x3d;

        break;

    }

    case 'T':

    {

        c = 0x36;

        break;

    }

    case 'V':

    {

        c = 0x32;

        break;

    }

    case 'W':

    {

        c = 0x43;

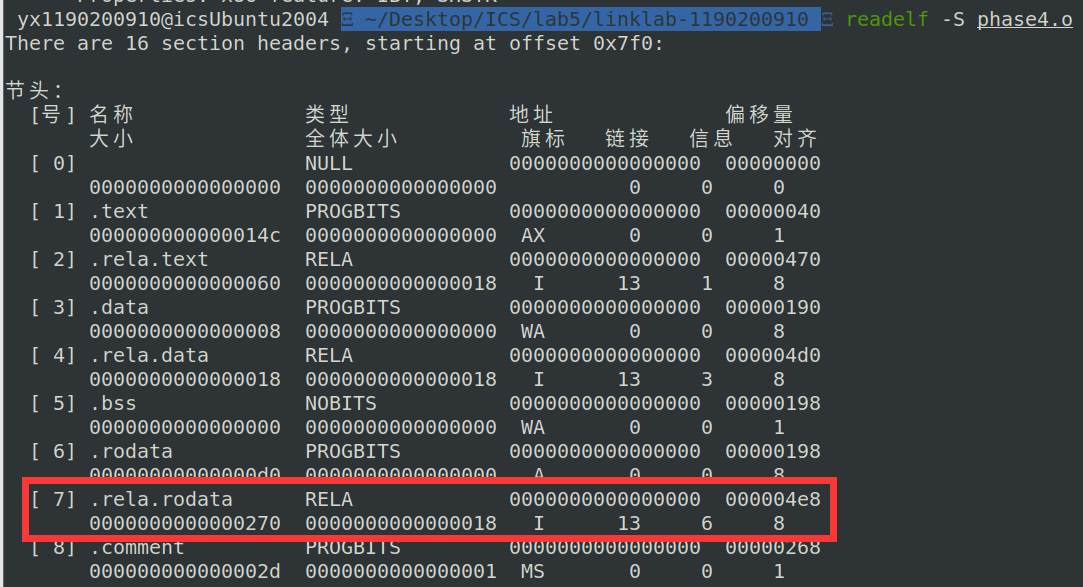
        break;

    }

    }

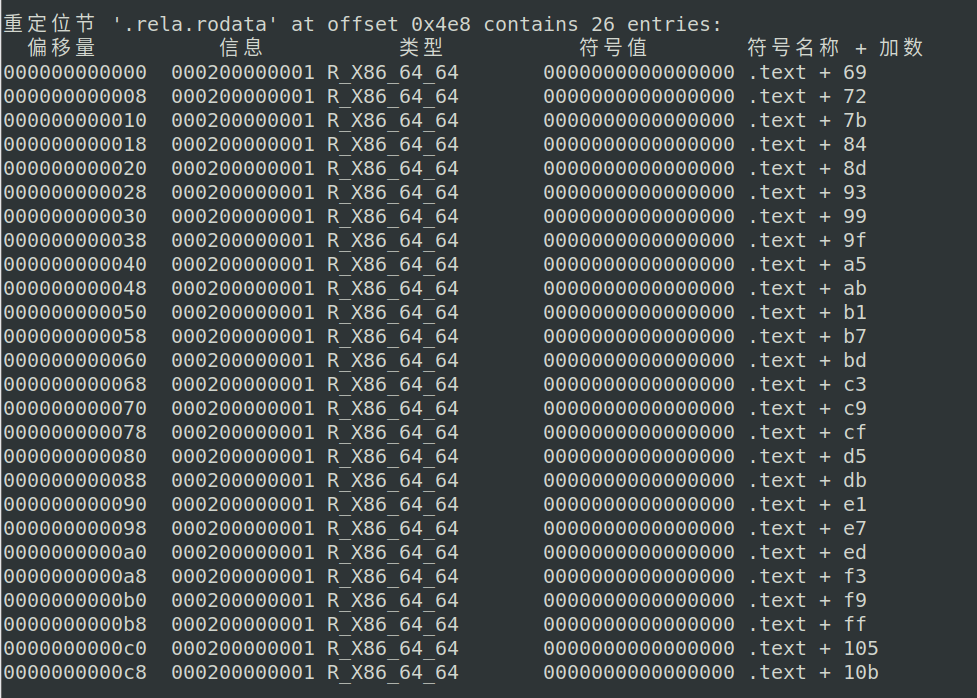
然后通过readelf读取phase4.o，在rodata节寻找有上面switch跳转表特征的部分，从而确定switch跳转表在.rodata节的偏移。

通过readelf -S看一下phase4.o的节头信息：



可以发现.rela.rodata节（只读节的重定位节）开始于文件偏移0x4e8的位置。

通过readelf -r读取phase4.o的重定位信息，可以看到在.rodata节有26个重定位条目：



这正好与switch跳转表的26个字母相对应，因此推断，这26个重定位条目就是switch跳转表，它开始于.rodata节偏移0的位置。

这些跳转应该就是从case ‘A’到case ‘Z’每一个跳转，跳转到的.text节位置。

根据cookie数组的内容{‘C’, ‘J’, ‘M’, ‘W’, ‘F’, ‘D’, ‘V’, ‘R’, ‘N’, ‘T’}，我们需要让跳转表满足以下要求：

case ‘C’跳转到c = 0x31

case ‘J’跳转到c = 0x31

case ‘M’跳转到c = 0x39

case ‘W’跳转到c = 0x30

case ‘F’跳转到c = 0x32

case ‘D’跳转到c = 0x30

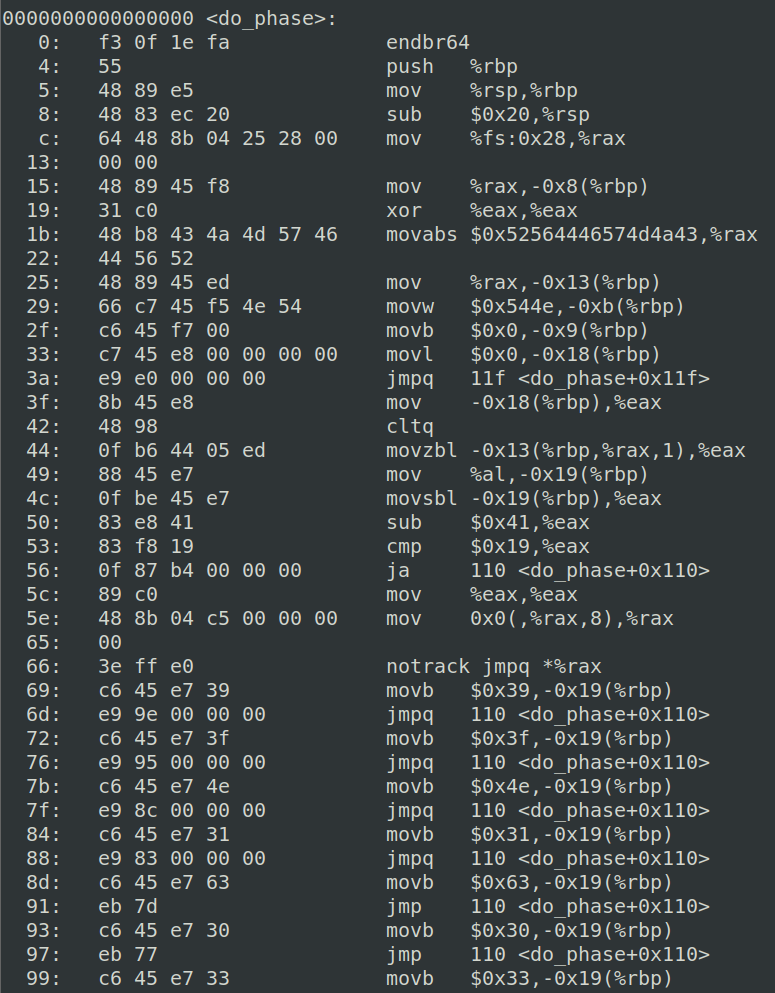
case ‘V’跳转到c = 0x30

case ‘R’跳转到c = 0x39

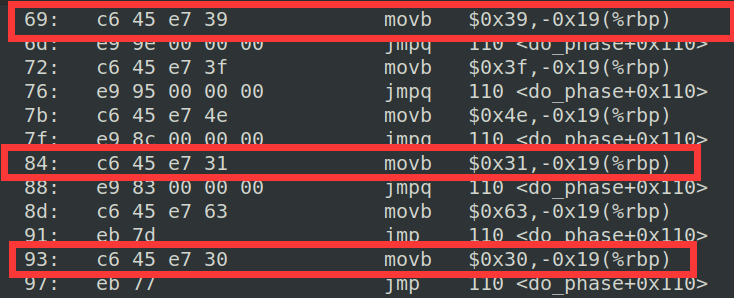
case ‘N’跳转到c = 0x31

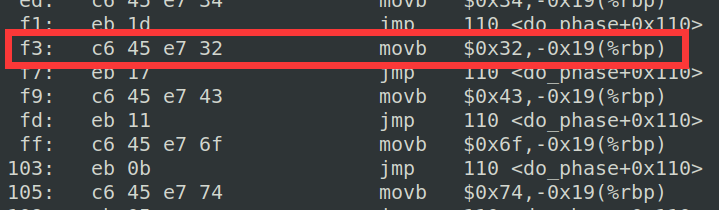
case ‘T’跳转到c = 0x30

通过objdump -d查看phase4.o中text节的反汇编指令：



由于do\_phase函数入口位于.text节偏移量为0的位置，因此跳转表中的相对.text节的偏移量就相当于反汇编指令中相对于do\_phase函数入口的偏移量。





通过分析反汇编代码，可以看到，

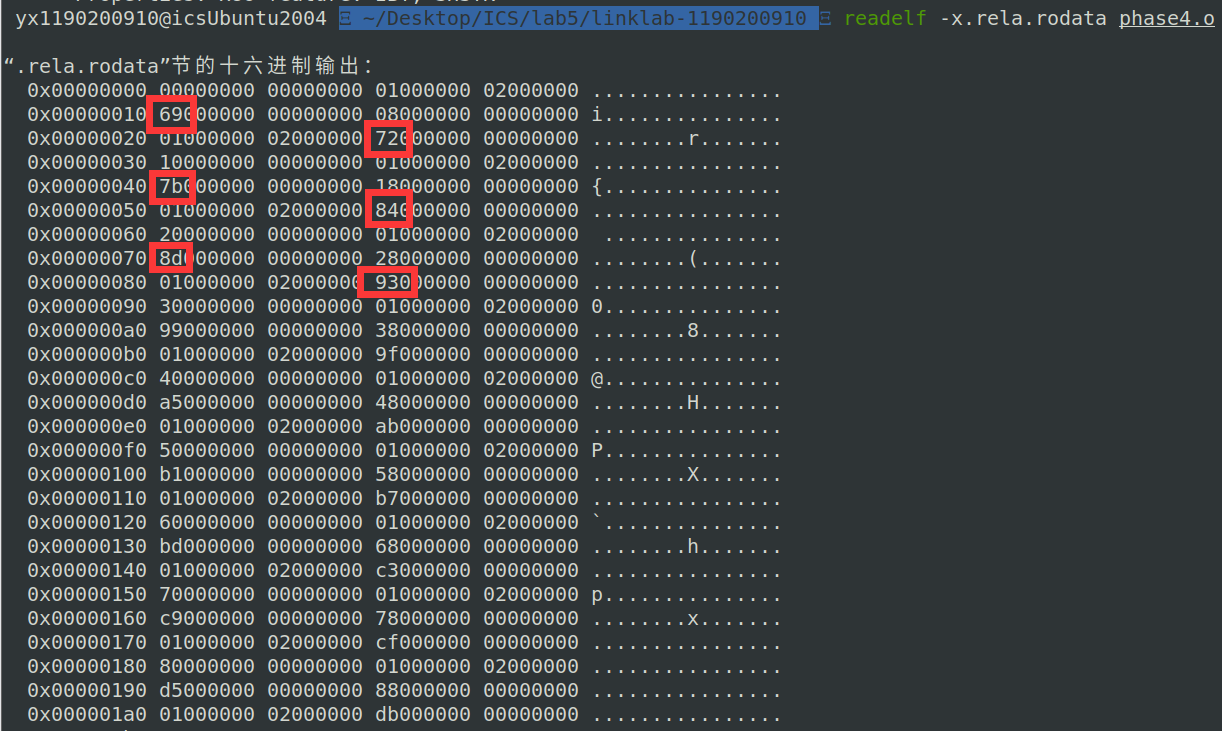
将c赋值为0x31位于偏移量为0x84的位置

将c赋值为0x39位于偏移量为0x69的位置

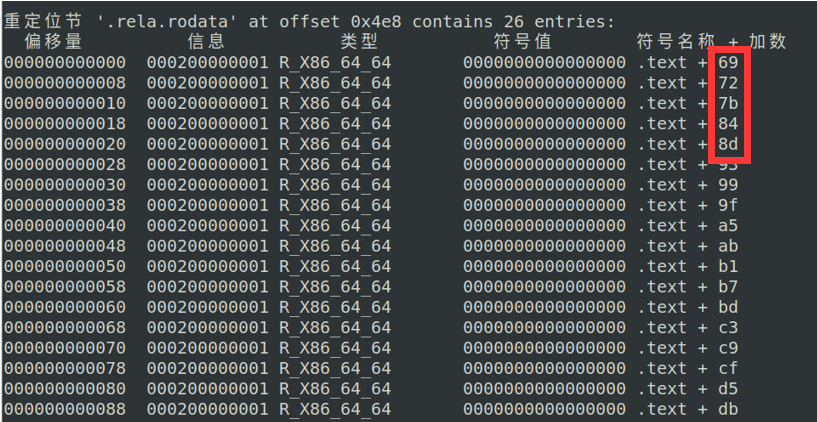
将c赋值为0x30位于偏移量为0x93的位置

将c赋值为0x32位于偏移量为0xf3的位置

通过readelf -x.rela.rodata phase4.o查看phase4.o的.rodata节内容：



其中的一些数字如0x69, 0x72, 0x84, 0x8d正好与下图中switch跳转表中跳转到.text节的偏移量对应，因此我们应该就是要修改这里的数据。



记住我们所需要的跳转表：

case ‘C’跳转到c = 0x31

case ‘J’跳转到c = 0x31

case ‘M’跳转到c = 0x39

case ‘W’跳转到c = 0x30

case ‘F’跳转到c = 0x32

case ‘D’跳转到c = 0x30

case ‘V’跳转到c = 0x30

case ‘R’跳转到c = 0x39

case ‘N’跳转到c = 0x31

case ‘T’跳转到c = 0x30

还要.text节的各个指令偏移：

c=0x31位于偏移量为0x84的位置

c=0x39位于偏移量为0x69的位置

c=0x30位于偏移量为0x93的位置

c=0x32位于偏移量为0xf3的位置

字符’C’是第3个字母，因此修改.rela.rodata节的第3个重定位目标使其跳转到.text节偏移0x84处。(7b🡪84)

字符’J’是第10个字母，因此修改.rela.rodata节的第10个重定位目标使其跳转到.text节偏移0x84处。(ab🡪84)

字符’M’是第13个字母，因此修改.rela.rodata节的第13个重定位目标使其跳转到.text节偏移0x69处。(bd🡪69)

字符’W’是第23个字母，因此修改.rela.rodata节的第23个重定位目标使其跳转到.text节偏移0x93处。(f9🡪93)

字符’F’是第6个字母，因此修改.rela.rodata节的第6个重定位目标使其跳转到.text节偏移0xf3处。(93🡪f3)

字符’D’是第4个字母，因此修改.rela.rodata节的第4个重定位目标使其跳转到.text节偏移0x93处。(84🡪93)

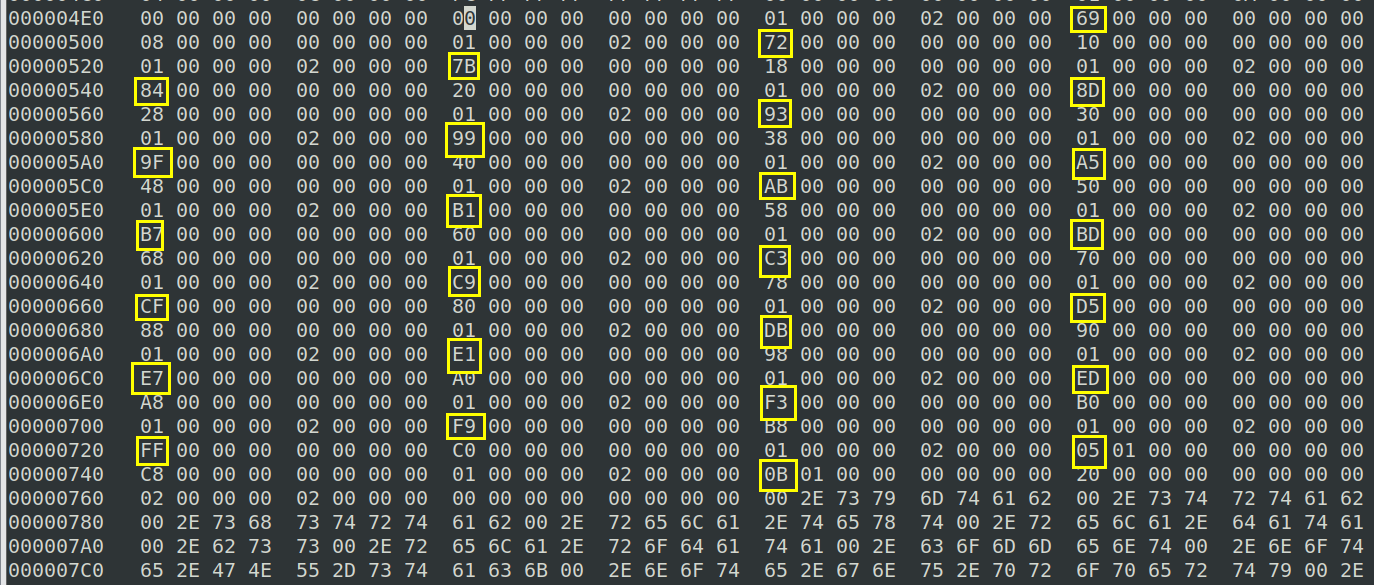
字符’V’是第22个字母，因此修改.rela.rodata节的第22个重定位目标使其跳转到.text节偏移0x93处。(f3🡪93)

字符’R’是第18个字母，因此修改.rela.rodata节的第18个重定位目标使其跳转到.text节偏移0x69处。(db🡪69)

字符’N’是第14个字母，因此修改.rela.rodata节的第14个重定位目标使其跳转到.text节偏移0x84处。(c3🡪84)

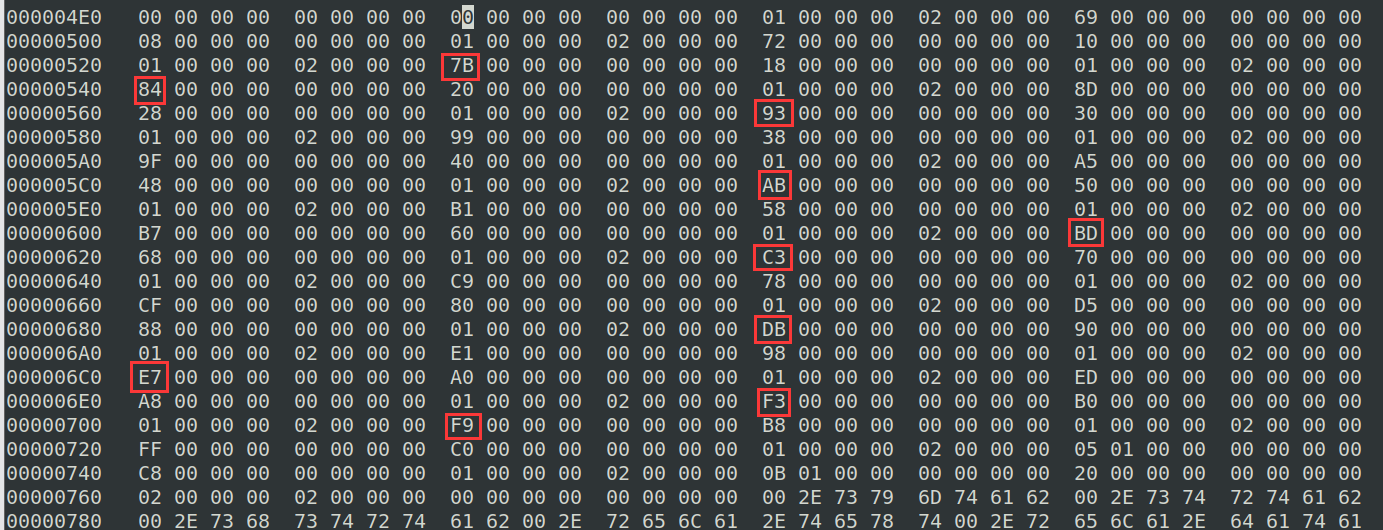
字符’T’是第20个字母，因此修改.rela.rodata节的第20个重定位目标使其跳转到.text节偏移0x93处。(e7🡪93)

然后通过hexedit编辑这个ELF。上面分析过.rela.rodata节位于文件偏移0x4e8的位置。

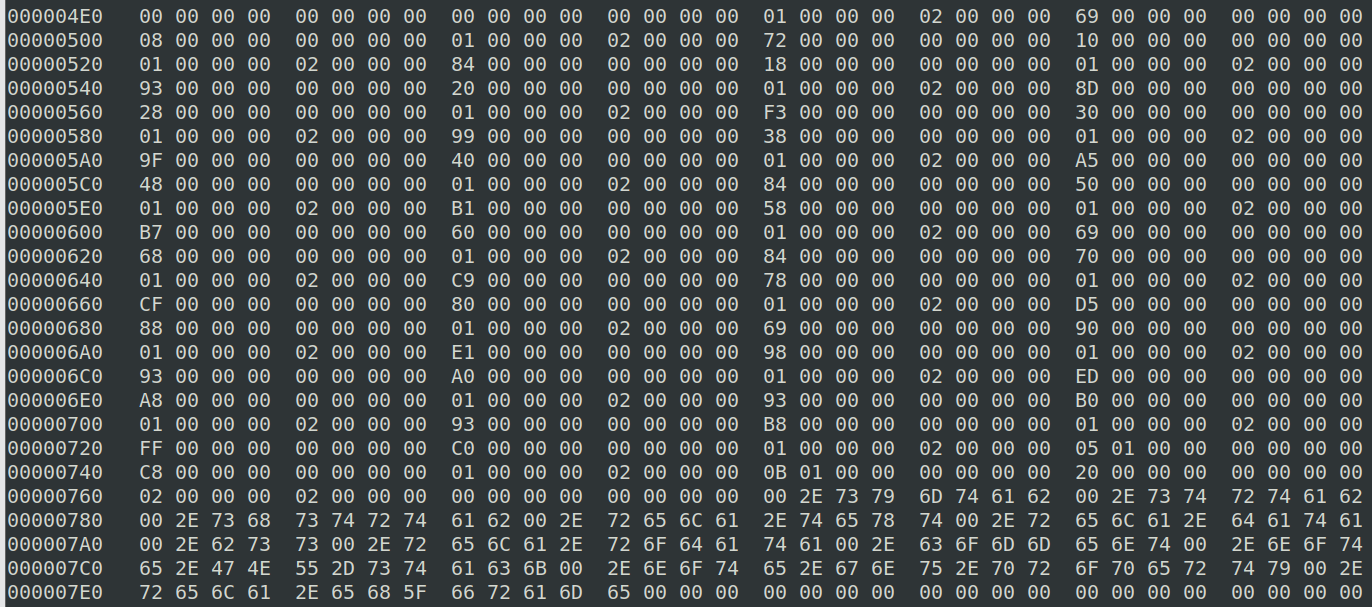


上图中标出的是所有偏移量的大小，小端序存储。

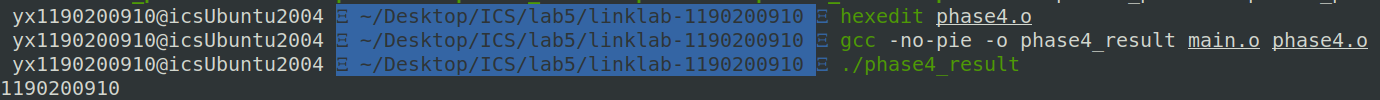
我们需要修改其中的10个。在下图中用红框标出。



修改后如下图所示。



然后保存退出。链接并运行程序：



可以看到正确输出了学号。