lab2：Crypto

1. 必做题目（70 pts）

怎么晕晕的。

1.1 RSAParty (25 pts)

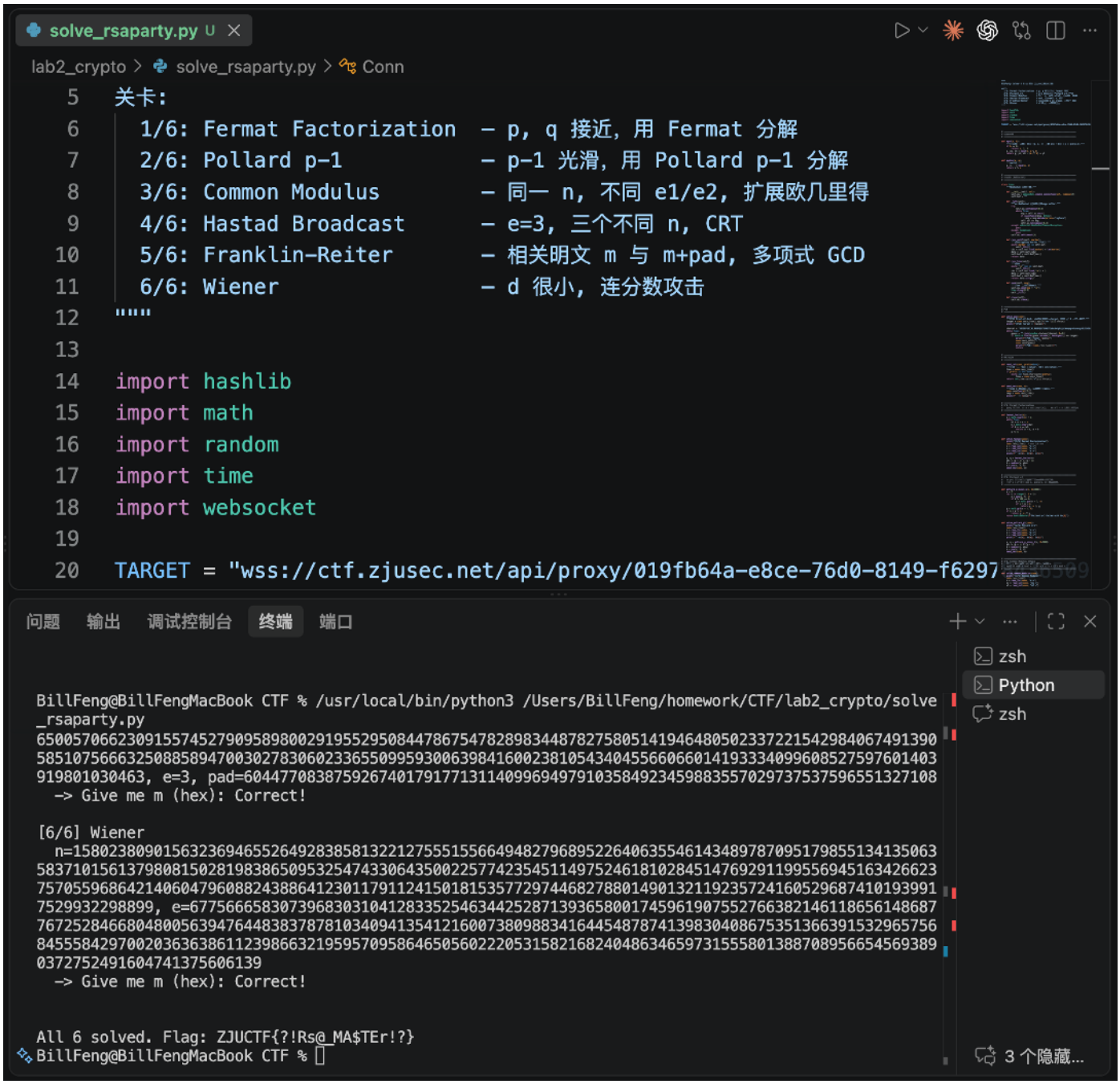

Image 1

共 6 关，每关要求解密服务器给出的 RSA 密文并返回明文。

本质上是分别利用6个数学漏洞解题。（好吧我晕晕的）

1.1.1 Fermat Factorization

这一关利用的漏洞：服务器用两个相近的素数 p, q 。

```
  def fermat(self):
        p = getPrime(512)
        gap = random.randint(0, 10000)
        q = p + gap
        while not isPrime(q):
            q += 1
        n = p * q
        e = 65537
        m = getRandomNBitInteger(256)
        c = pow(m, e, n)
```

$n = pq = \left(\frac{p+q}{2}\right)^2 - \left(\frac{p-q}{2}\right)^2$

令 $a = \frac{p+q}{2}$，$b = \frac{p-q}{2}$，则 $a^2 - n = b^2$。

从 $a = \lceil \sqrt{n} \rceil$ 开始，计算 $b^2 = a^2 - n$，当 b 为整数时得 $p = a - b$, $q = a + b$。

```
def fermat_factor(n):
    a = math.isqrt(n) + 1
    while True:
        b2 = a * a - n
        b = math.isqrt(b2)
        if b * b == b2:
            return a - b, a + b
        a += 1
```

算出 p,q 之后即可正常算出明文。

1.1.2 Pollard p-1

这一关利用的漏洞：$p-1$ 是光滑数（全部由小素数幂乘积组成）。

什么是光滑数？ 一个数如果分解后全是很小的质因数，就叫"光滑数"。比如 $12 = 2^2 \times 3$ 光滑，$9941 = 9941$（自己是素数）就不光滑。服务器故意生成 p 使得 $p-1$ 只含 ≤1000 的小素数。

```
def gen_smooth_prime(bits=512, bound=1000, power_cap=2 ** 16):
    primes = _small_primes(bound)
    while True:
        s = 1
        powers = {}
        while s.bit_length() < bits:
            q = random.choice(primes)
            if powers.get(q, 1) * q > power_cap:
                continue
            powers[q] = powers.get(q, 1) * q
            s *= q
        if s.bit_length() == bits and isPrime(s + 1):
            return s + 1
```

攻击思路 费马小定理说 $2^{p-1} \equiv 1 \pmod{p}$。如果 $p-1 \mid B!$（$B! = 1 \times 2 \times \cdots \times B$），那么 $2^{B!} \equiv 1 \pmod{p}$，即 $2^{B!} - 1$ 是 p 的倍数。

由于 $p-1$ 只含 ≤1000 的小素数，它一定能整除 2000!（因为 2000 以内包含了所有 ≤1000 的素数，且出现次数足够多）。所以计算 $a = 2^{2000!} \bmod n$，然后 $\gcd(a-1, n)$ 就会泄露 p。

```python
def pollard_p_minus_1(n, B=2000):
    a = 2
    for k in range(2, B + 1):
        a = pow(a, k, n)        # 逐步构造 a = 2^(k!) mod n
        if k % 100 == 0:
            g = math.gcd(a - 1, n)
            if 1 < g < n:
                return g, n // g  # g 就是 p!
```

算出 p, q 之后即可正常算出明文。

1.1.3 Common Modulus Attack

这一关利用的漏洞：共模攻击。同一 n 下用不同的 $e_1 = 3$, $e_2 = 17$ 加密同一明文 m。

```python
    def common_modulus(self):
        p = getPrime(512)
        q = getPrime(512)
        n = p * q
        e1, e2 = 3, 17
        m = getRandomNBitInteger(256)
        c1 = pow(m, e1, n)
        c2 = pow(m, e2, n)
```

扩展欧几里得求 $a \cdot e_1 + b \cdot e_2 = 1$，则 $m = c_1^a \cdot c_2^b \pmod{n}$。负指数时用模逆元处理。

（之前的题目涉及过共模攻击）

```python
g, a, b = egcd(e1, e2)
part1 = pow(c1, a, n) if a >= 0 else pow(modinv(c1, n), -a, n)
part2 = pow(c2, b, n) if b >= 0 else pow(modinv(c2, n), -b, n)
m = (part1 * part2) % n
```

1.1.4 Hastad Broadcast Attack

这一关利用的漏洞：$e = 3$，同一明文 m 用三个不同的 n_1, n_2, n_3 加密。

```python
    def hastad(self):
        m = getRandomNBitInteger(300)
        e = 3
        ns = []
        while len(ns) < 3:
            cand = getPrime(512)
            if cand not in ns:
                ns.append(cand)
        cs = [pow(m, e, n) for n in ns]
```

有三组 (n_i, c_i)，其中$c_i = m^3 \bmod n_i$。求 m。

中国剩余定理：

1. 令 $N = n_1 n_2 n_3$（三把锁的乘积，足够大）

2. 对每个 i，算 $N_i = N/n_i$，以及 $t_i = N_i$ 模 n_i 的逆元
3. 组合：$x = c_1N_1t_1 + c_2N_2t_2 + c_3N_3t_3 \pmod{N}$

因为 m 只有 300 bits，$m^3 < N$，所以 x 就是 m^3 的真值，直接开立方根得 m。

```
def crt(residues, moduli):
    M = 1
    for m in moduli: M *= m          # N = n1 * n2 * n3
    result = 0
    for r, m in zip(residues, moduli):
        Mi = M // m                    # Ni = N / ni
        result = (result + r * Mi * modinv(Mi, m)) % M
    return result

m = integer_cuberoot(crt(cs, ns))
```

1.1.5 Franklin-Reiter Related Message Attack

这一关利用的漏洞：已知两条相关消息 m 和 $m + pad$ 的密文。

```
    def franklin_reiter(self):
        p = getPrime(512)
        q = getPrime(512)
        n = p * q
        e = 3
        m = getRandomNBitInteger(400)
        pad = getRandomNBitInteger(256)

        c1 = pow(m, e, n)
        c2 = pow(m + pad, e, n)
```

构造多项式 $f_1(x) = x^3 - c_1$ 和 $f_2(x) = (x + pad)^3 - c_2$。两者有公共根 m，在 $\mathbb{Z}_n[x]$ 上求 $\gcd(f_1, f_2)$ 得到 $x - m$，常数项取反即 m。

```
f1 = [(-c1) % n, 0, 0, 1]
f2 = [(pad**3 - c2) % n, (3 * pad**2) % n, (3 * pad) % n, 1]
g = poly_gcd(f1, f2, n)
m = (-g[0]) % n   # 首一多项式 [const, 1] ⇒ x ≡ -const
```

1.1.6 Wiener Attack

这一关利用的漏洞：私钥 d 很小（200 bits），远小于 $n^{0.25}$（n 的四次方根）。

正常 RSA 是先选 e 再算 d，但服务器反过来：先选了一个很小的 d，再算出 e。

```python
    def wiener(self):
        p = getPrime(512)
        q = getPrime(512)
        n = p * q
        phi = (p - 1) * (q - 1)

        d = getPrime(200)          # d 只有 200 bits!
        e = pow(d, -1, phi)        # 从 d 反推 e

        m = getRandomNBitInteger(256)
        c = pow(m, e, n)
```

为什么 d 小就有问题？由 $ed \equiv 1 \pmod{\varphi}$，存在 k 使得 $ed - k\varphi = 1$。两边除以 $d\varphi$：

$\left|\frac{e}{\varphi} - \frac{k}{d}\right| = \frac{1}{d\varphi}$

d 小 $\rightarrow$ $\frac{1}{d\varphi}$ 极小 $\rightarrow$ $\frac{k}{d}$ 是 $\frac{e}{\varphi}$ 的极好近似。又因为 $\varphi \approx n$（差 $p+q-1$），所以 $\frac{k}{d}$ 也是 $\frac{e}{n}$ 的极好近似。

连分数恰好能找到一个数的所有最佳有理逼近。把 $\frac{e}{n}$ 展开：

$$\frac{e}{n} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \ldots}}$$

逐步截断得到收敛项 $\frac{k_1}{d_1}, \frac{k_2}{d_2}, \ldots$，真正的 $\frac{k}{d}$ 一定在其中。

Wiener 证明了：当 $d < \frac{1}{3}n^{0.25}$ 时（这关 d 200 bits，$n^{0.25} \approx 256$ bits，满足），遍历连分数收敛项一定能找到正确的 d。验证方法：对每个候选 (k, d)，算 $\varphi = \frac{ed-1}{k}$，然后检查能否解出 p, q。

```python
for k, d in convergents(continued_fraction(e, n)):
    if k == 0 or (e * d - 1) % k != 0:
        continue
    phi = (e * d - 1) // k
    s = n - phi + 1            # p + q
    disc = s * s - 4 * n       # (p - q)^2
    sqrt_disc = math.isqrt(disc)
    if sqrt_disc * sqrt_disc == disc:
        p = (s + sqrt_disc) // 2
        if p * (n // p) == n:
            m = pow(c, d, n)
```

1.2 EZDLP (15 pts)

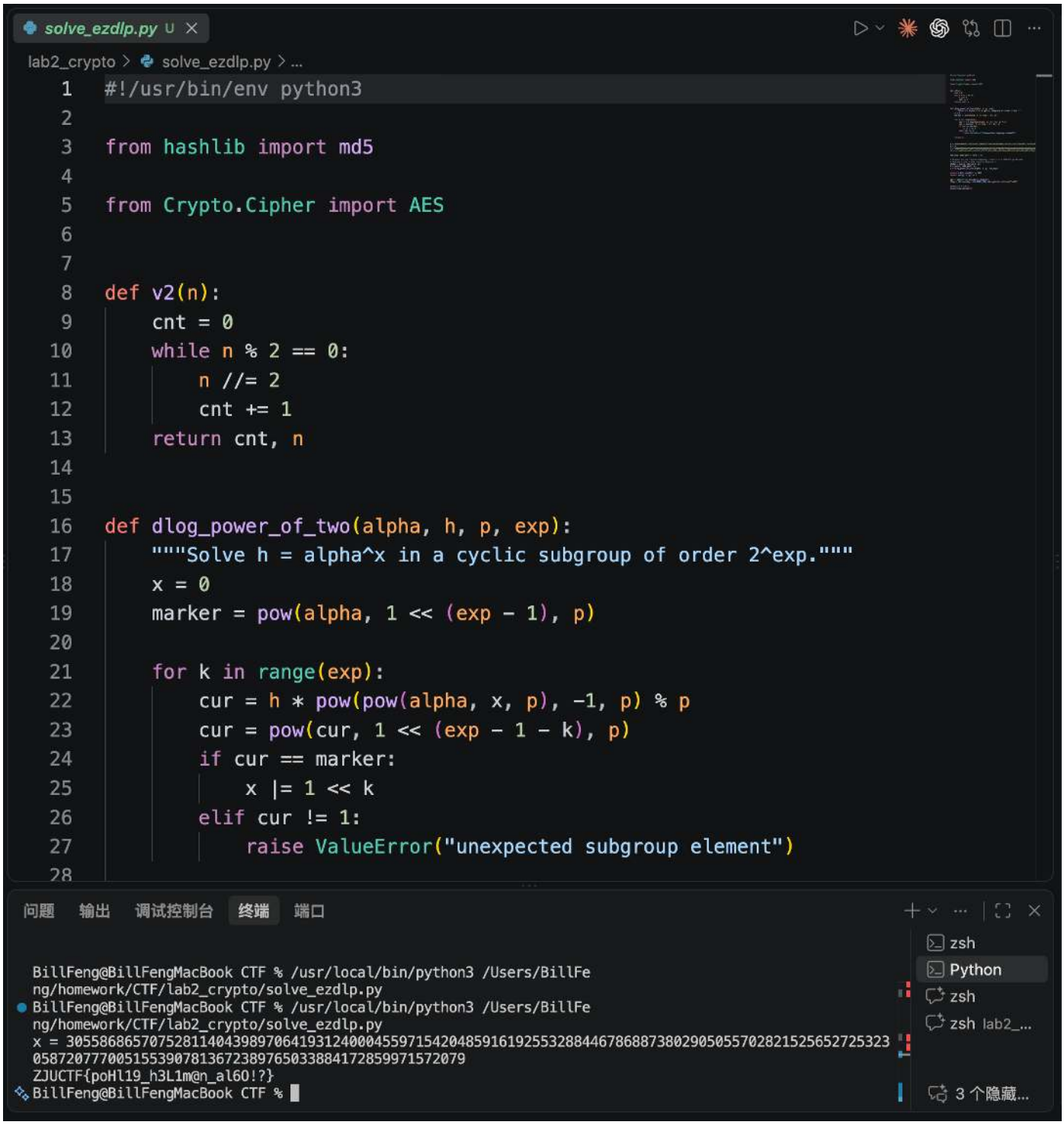

Image 2

这题的代码如下：

```
p = 960494008017250155494739990397196249930200062145145133132556398221074529657304218221253517153928380265486339083177542201148993799925721673833337786213881109579869080457126122337945518 09
x = getPrime(500)
g = 3
c = pow(g, x, p)

aes = AES.new(key = md5(str(x).encode()).digest(), mode = AES.MODE_ECB)
ct = aes.encrypt(pad(flag, 16))
```

`c = pow(g, x, p)`，也就是 $c = 3^x \bmod p$

题目告诉了我们 `p`、`g=3`、`c`，但没有告诉我们 `x`。我们要从 $3^x \bmod p = c$ 反推 `x`。这个问题叫离散对数问题。正常情况下，大素数模数下的离散对数很难直接求。

AES 是对称加密算法，可以理解成“同一把钥匙加密，也用同一把钥匙解密”。这里 AES 的钥匙是从 `x` 算出来的。`x -> 转成字符串 -> MD5 -> AES key`

```
key = md5(str(x).encode()).digest()
aes = AES.new(key=key, mode=AES.MODE_ECB)
ct = aes.encrypt(pad(flag, 16))
```

所以只要我恢复了 `x`，就能重新算出 AES key，然后解密密文 `ct` 得到 flag。

1.2.1 ?!pp!?

这题的突破口在 `p`（并非 AES 算法本身），对模素数 `p` 的乘法来说，指数会按 `p-1` 形成周期，所以我们先分解 `p-1`：

```
p - 1 = 2^518 * 111932680969824918166220667345 7
```

也就是说，`p-1` 里面有一个非常大的 2^{518} 因子。

然后注意题目里生成 `x` 的方式：

```
x = getPrime(500)
```

这说明 `x` 是 500 bit 的素数，所以它一定小于 2^{500} $x < 2^{500} < 2^{518}$

这一点非常关键。因为如果一个数本来就小于 2^{518}，那么它对 2^{518} 取余还是自己。

因此我不需要求完整群里的离散对数，只要求出 `x mod 2^518`，就已经拿到了真正的 `x`。

1.2.2 缩小问题

令 $r = 111932680969824918166220667345 7$

因为 p - 1 = 2^518 * r

所以我把原来的式子两边都提升到 `r` 次方 $c^r \equiv (3^r)^x \pmod{p}$

令

```
alpha = pow(3, r, p)
h = pow(c, r, p)
```

就变成 $h = \alpha^x \bmod p$

这样做的效果是：原来很大的问题被投影到了一个大小为 2^{518} 的循环子群里。

1.2.3 逐位恢复 x

二进制里的每一位只有两种可能：0 或 1。

所以脚本的想法就是：

```
判断 x 的第 0 位是 0 还是 1
判断 x 的第 1 位是 0 还是 1
判断 x 的第 2 位是 0 还是 1
...
一直判断到第 517 位
```

每判断一位，就把它加到答案 `x` 里。

核心代码：

```python
def dlog_power_of_two(alpha, h, p, exp):
    x = 0
    # marker 是用来判断当前二进制位是否为 1 的标记值
    marker = pow(alpha, 1 << (exp - 1), p)

    for k in range(exp):
        # pow(alpha, x, p) 表示已知低位对应的 alpha^x
        # 乘它的逆元，相当于从 h = alpha^真实x 中扣掉已经知道的低位
        cur = h * pow(pow(alpha, x, p), -1, p) % p

        # 把第 k 位"放大"到最高有效位置，方便判断这一位是 0 还是 1
        cur = pow(cur, 1 << (exp - 1 - k), p)

        # 如果结果等于 marker，说明第 k 位是 1，把这一位写进 x
        if cur == marker:
            x |= 1 << k

        # 正常情况下结果只会是 1 或 marker；如果都不是，说明参数或前面恢复出错
        elif cur != 1:
            raise ValueError("unexpected subgroup element")
    return x
```

找到的 `x` 满足 $3^x \bmod p = c$

1.2.4 AES

拿到 `x` 后，就可以重新算 AES key：

```
key = md5(str(x).encode()).digest()
```

然后解密：

```
flag = AES.new(key, AES.MODE_ECB).decrypt(ct).rstrip(b"\x00")
```

（这里 `.rstrip(b"\x00")` 是因为题目加密前用 `\x00` 把 flag 补齐到了 16 字节的倍数，解密后要把这些补位删掉。

1.3 KillerECC (15 pts)

题目提示了 `npm audit`，所以先看附件里的依赖。`package.json` 里用到了 `elliptic@^6.6.0`：

```
"dependencies": {
  "elliptic": "^6.6.0",
  "ws": "^8.21.1"
}
```

运行 `npm audit` 后发现 `elliptic <= 6.6.0` 有一个 critical 漏洞。

```
Elliptic's private key extraction in ECDSA upon signing a malformed input (e.g. a
string)
```

大意是，用 `elliptic@6.6.0` 做 ECDSA 签名时，如果传入特殊格式的字符串消息，可能导致私钥泄露。

服务端刚好把用户输入的 `msg` 原样传给了 `key.sign(msg)`：

```
function sign(key, msg) {
    const sig = key.sign(msg);
    return { r: sig.r.toString(10), s: sig.s.toString(10) };
}
```

1.3.1 签名

什么是数字签名？数字签名证明“这条消息确实是某个私钥持有者签出来的”。ECDSA 里有一对密钥，私钥记作 `d`，自己保存；公钥可以公开，别人拿公钥来验证签名。

对一条消息签名时，算法会先把消息变成一个数字，通常记作 `z`。签名过程还会用到一个临时数字 `k`，也叫 nonce。这个 `k` 每次签名都应该不一样，而且不能被别人知道，因为它会直接参与签名计算。

最终服务端返回的签名是两个数 `(r, s)`。其中 `r` 和 `k` 有关，所以如果两次签名的 `r` 一样，就很可能说明两次用了同一个 `k`。

ECDSA 每次签名都会用到一个 nonce，通常记为 `k`。签名结果是 `(r, s)`，其中 `s` 的计算公式是：

$s = k^{-1}(z + rd) \bmod n$

这里 `z` 是消息对应的数字，`d` 是私钥，`n` 是 secp256k1 的阶（？）。正常情况下，每次签名的 `k` 都不能重复。一旦两次签名复用了同一个 `k`，私钥就可以被代数算出来。

1.3.2 触发漏洞

`elliptic@6.6.0` 对字符串消息的处理不一致。向服务端请求两次签名：

```
sign 1
sign -1
```

这两个消息在 ECDSA 公式里的消息值分别是 `z1 = 1` 和 `z2 = -1`，但是生成 RFC6979 nonce 时会被转成同样的 32 字节数组。于是两次签名得到同一个 `k`，返回结果里的 `r` 也相同。

于是有两条签名：

$s_1 = k^{-1}(z_1 + rd) \bmod n$

$s_2 = k^{-1}(z_2 + rd) \bmod n$

两式相减：

$s_1 - s_2 = k^{-1}(z_1 - z_2) \bmod n$

这样就能先算出 `k`：

$k = (z_1 - z_2)(s_1 - s_2)^{-1} \bmod n$

再把 `k` 代回签名公式，就能恢复私钥：

$d = (s_1 k - z_1) r^{-1} \bmod n$

核心代码如下。先签 `1` 和 `-1`，确认 `r` 相同，再按公式算出私钥 `d`，最后提交十六进制私钥。

```python
N = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

r1, s1 = parse_sig(send_cmd(ws, "sign 1", need_sig=True))
r2, s2 = parse_sig(send_cmd(ws, "sign -1", need_sig=True))
assert r1 == r2

z1 = 1
z2 = -1

k = ((z1 - z2) % N) * inverse((s1 - s2) % N, N) % N
d = (s1 * k - z1) % N
d = d * inverse(r1, N) % N

send_cmd(ws, f"submit {d:x}")
```

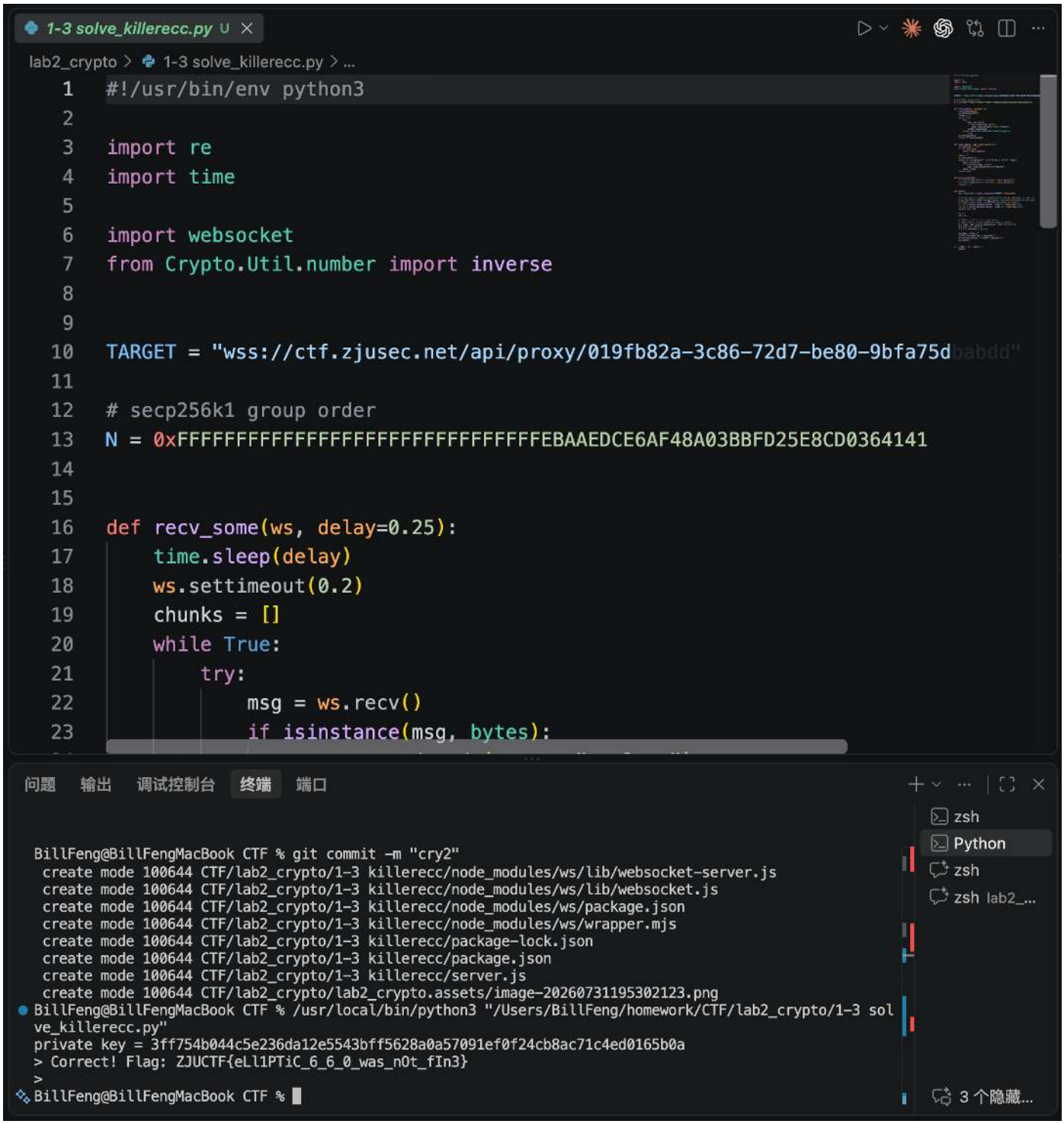

Image 3

1.4 EZCopper (15 pts)

```
m = bytes_to_long(flag)
assert len(flag) == 60

p = getPrime(512)
q = getPrime(512)
N = p*q

c1 = pow(m, p, N)
c2 = pow(m, q, N)
```

它的指数直接用了 `p` 和 `q`。这里先分别在模 `p` 和模 `q` 下看。因为 `N = p*q`，一个式子如果模 `N` 同余，那么拿去模 `p` 或模 `q` 也仍然同余。

对 `c1 = m^p mod N` 来说，有 $c_1 \equiv m^p \pmod{p}$。根据费马小定理，$m^p \equiv m \pmod{p}$，所以 $c_1 \equiv m \pmod{p}$，也就是 $p \mid (c_1 - m)$。

同理，`c2 = m^q mod N`，放到模 `q` 下有 $c_2 \equiv m^q \pmod{q}$。再由费马小定理，$m^q \equiv m \pmod{q}$，所以 $c_2 \equiv m \pmod{q}$，也就是 $q \mid (c_2 - m)$。

现在 `(c1 - m)` 里有因子 `p`，`(c2 - m)` 里有因子 `q`。把它们乘起来，就同时含有 `p` 和 `q`，而 `N = p*q`，所以 $N \mid (c_1 - m)(c_2 - m)$。换成模运算就是 $(c_1 - m)(c_2 - m) \equiv 0 \pmod{N}$。

把未知的 `m` 暂时写成变量 `x`，就得到 $f(x) = (x - c_1)(x - c_2) \equiv 0 \pmod{N}$。真正的明文 `m` 就是这个方程的一个解。

但是普通的模方程并不好解，因为我们不知道 `p` 和 `q`，也不能直接分解 `N`。这时题目给了另一个重要条件

```
assert len(flag) == 60
```

flag 只有 60 字节，所以 `m` 不会太大，$m < 256^{60} = 2^{480}$。而 `N` 是 1024 bit，$N^{1/2}$ 大约是 2^{512}，所以 $m < 2^{480} < N^{1/2}$。

也就是说 `m` 是一个“小根”。题目名里的 `Copper` 就是在提示 Coppersmith。Coppersmith 可以找这种模方程里的小整数解。

这里我的理解是：我们不去分解 `N`，把“明文满足某个模方程”这件事变成小根问题。因为明文本身足够短，所以 Coppersmith 能把它找出来。

核心代码如下：

```
expr = (x - c1) * (x - c2)

reduced = 0
for power, coeff in enumerate(reversed(sp.Poly(expr, x).all_coeffs())):
    reduced += center_mod(int(coeff), N) * x ** power

m = small_root_by_coppersmith(reduced, N, 256 ** 60)
```

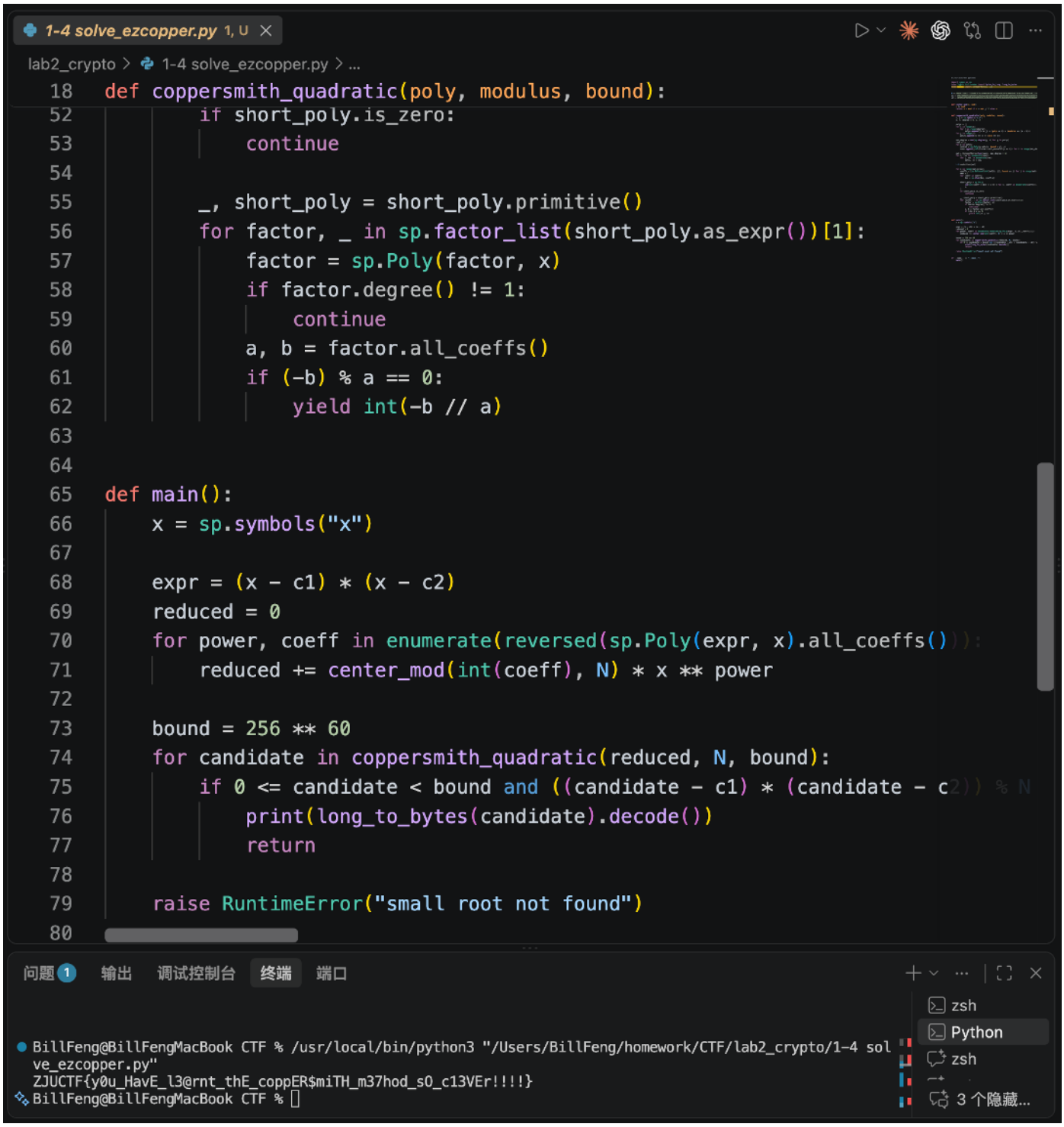

Image 4

2. 😭选做题目（最高 45 pts）

2.1 Regev (15 pts)

这题给的是一个很标准的 LWE？（Learning With Errors，容错学习问题） 形式

```
n = 100
m = 150
q = next_prime(2**20)

s = [random.randint(0, 1) for _ in range(n)]
e = [random.randint(-1, 1) for _ in range(m)]
b[i] = (A[i] * s + e[i]) % q
```

这里 `s` 是 100 维的 0/1 向量，误差 `e` 每一项都只有 `-1, 0, 1`。最后 AES key 是 `sha256(bytes(s))`，所以目标就是恢复 `s`。

如果没有误差，直接解线性方程 $As = b$ 就行。但这里多了一个很小的误差，所以更像是“找离 `b` 最近的格点”。把所有 $As + qz$ 组成一个格，真实的 `b` 满足 $b = As + e \pmod{q}$

也就是说 `b` 距离某个格点只差一个很短的误差向量 `e`。因为 `e` 的每一项都不超过 1，这个最近格点非常近，用 LLL/BKZ 规约后做 CVP 就能找到。

来自对话...

```
Q：什么是lll和cvp?

A：LWE 可以理解成“带一点误差的线性方程”。如果没有误差，我们只要解 A*s=b 就能得到秘密 s，但题目给的是 b=A*s+e，其中 e 很小，只会让结果偏一点点。于是 b 不是真正的正规点，而是离某个正规点非常近。我们把所有可能的 A*s+q*z 看成一堆有规律排列的格点，目标就是找到离 b 最近的那个格点。CVP 做的就是这件事，它负责找最近格点；LLL 则是先把格的表示整理得更好用，让后面的寻找不至于又慢又乱。因为这题误差特别小，所以最近的格点基本就是没加误差前的结果，找到它以后就能反推出秘密 s，再用 s 解密 flag。
```

```
L = lattice_generated_by_A_and_q(A, q)
closest = CVP.closest_vector(L, b)
e = b - closest

s = solve_mod(A0, closest[:n], q)
key = sha256(bytes(s)).digest()
flag = AES.new(key, AES.MODE_CBC, iv).decrypt(ct)
```

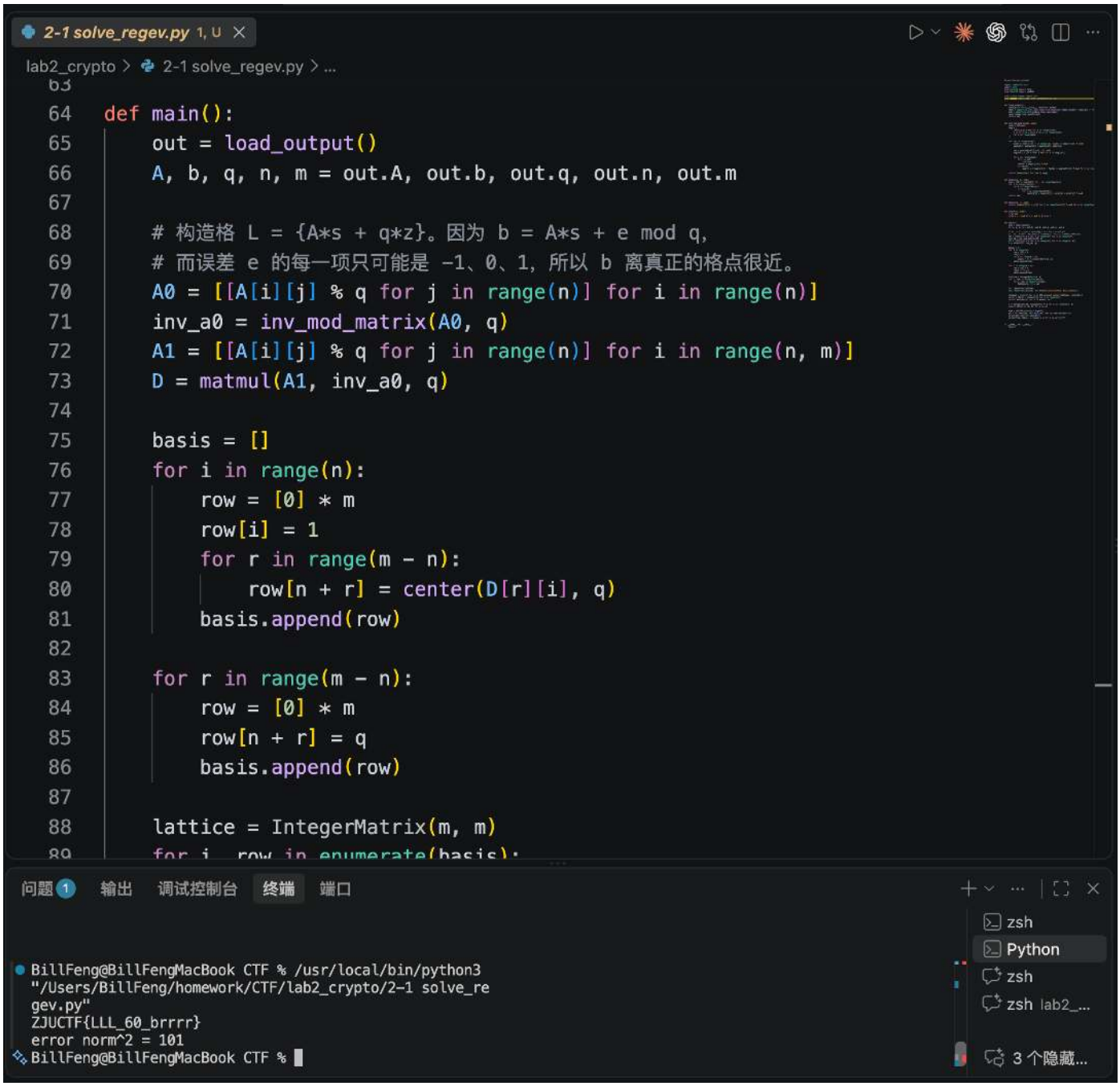

Image 5

ZJUCTF{LLL_60_brrrr}

2.2 EZHNP (15 pts)

ECDSA，但小 `k`

```
k = getPrime(240)
P = k * G
r = int(P.xy()[0]) % n
s = inverse(k, n) * (h + r * sk) % n
```

secp256k1 的阶 `n` 是 256 bit，而这里的 `k` 只有 240 bit。也就是说每个 nonce 的高 16 bit 都是 0。一次签名泄露不够，但题目给了 18 组同一消息的签名，刚好可以做 Hidden Number Problem。

从 ECDSA 公式出发：

$s_i k_i \equiv h + r_i sk \pmod{n}$

移一下项：

$k_i \equiv r_i s_i^{-1} sk + h s_i^{-1} \pmod{n}$

令 $a_i = r_i s_i^{-1}$，$c_i = h s_i^{-1}$，就有：

$k_i \equiv a_i sk + c_i \pmod{n}$

这里 `sk` 是私钥，也就是 flag 转成的整数；`k_i` 虽然未知，但我们知道它很小，小于 2^{240}。所以问题变成：找一个 `sk`，让所有 $a_i sk + c_i \pmod{n}$ 都落在很小的范围里。这就是 HNP。

脚本里把这些同余关系放进格里，然后用 CVP 找最近向量。为了让区间更对称，把 `[0, 2^240)` 平移到中心附近处理。

```
a = r * inverse(s, n) % n
c = h * inverse(s, n) % n

target = [-c[i] + 2**239 for i in range(t)]
closest = CVP.closest_vector(lattice, target)

sk = closest[0] * inverse(a[0], n) % n
```

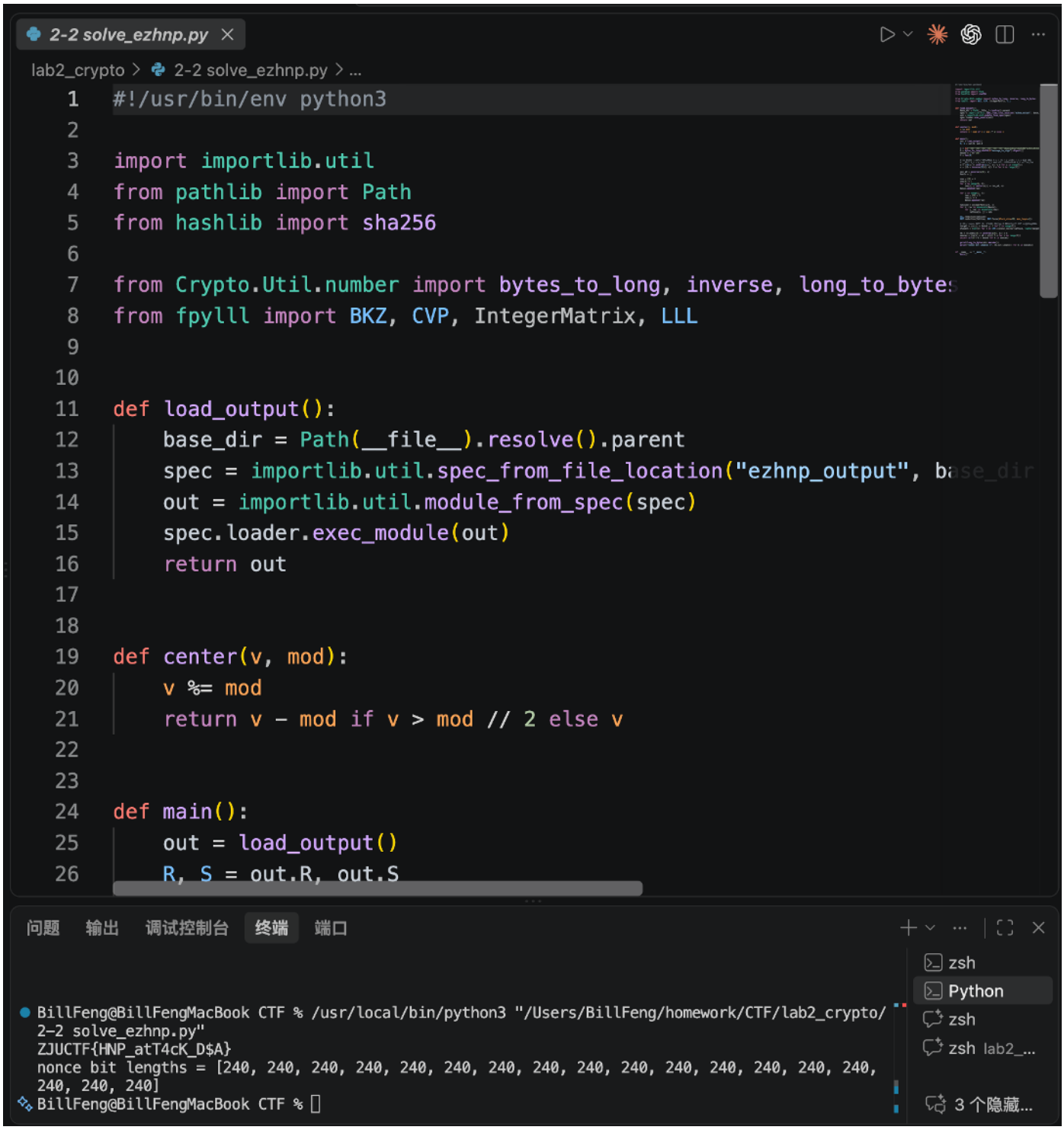

Image 6

ZJUCTF{HNP_atT4cK_D$A}

2.3 PRNG Study4 (15 pts)

连接 `nc zjusec.com 30094` 后选择 Level 4。源码里第 4 个 PRNG 看起来很像 Dual EC 那类生成器，它内部在 P-256 曲线上算点，然后把点的 x 坐标切成 64 bit 一段一段输出。

关键代码大概是这样：

```
self.Q = self.mul(random.getrandbits(bits // 2), self.P)
self.a = random.getrandbits(bits)

self.a = self.mul(self.a, self.P)[0]
self.b += (self.mul(self.a, self.Q)[0] >> 16) << self.cnt
```

这里 `bits = 64`，所以 `Q = dP` 里面的 `d` 只有 32 bit。服务一开始会泄露 8 个输出，前 4 个输出正好拼回 `Q` 的 x 坐标。因为 `d` 很小，可以用 baby-step giant-step 在大约 2^{16} 级别找回 `d`，这一步不是去解完整的 256 bit 椭圆曲线离散对数，而是利用题目把秘密乘数故意设小了。

后面的输出来自 `x(aQ) >> 16`，也就是只截掉了低 16 bit。泄露出来的 64 bit 块会跨过 240 bit 的内部输出，所以第 8 个泄露值其实一半属于上一块，一小段属于下一块。处理好这个拼接以后，把缺的 16 bit 暴力枚举，就能把可能的曲线点补出来。

因为已经知道 `Q = dP`，如果某个点是 `aQ`，那乘上 `d^{-1}` 就能得到 `aP`。而生成器下一轮的状态正是这个点的 x 坐标，所以状态就接上了。后面只要继续模拟 PRNG，再用服务给的 sha256 摘要筛掉错误候选，就可以预测全部 20 个数。

```
qx = chunks_to_int(known[:4])
d = bsgs_for_qx(qx)
inv_d = pow(d, -1, ORDER)

t1 = known[4] | (known[5] << 64) | (known[6] << 128)
t1 |= (known[7] & ((1 << 48) - 1)) << 192

for low in range(1 << 16):
    x = (t1 << 16) | low
    for r in lift_x(x):
        a = ecdh_x(inv_d, r)
        output = predict_rest(a)
```

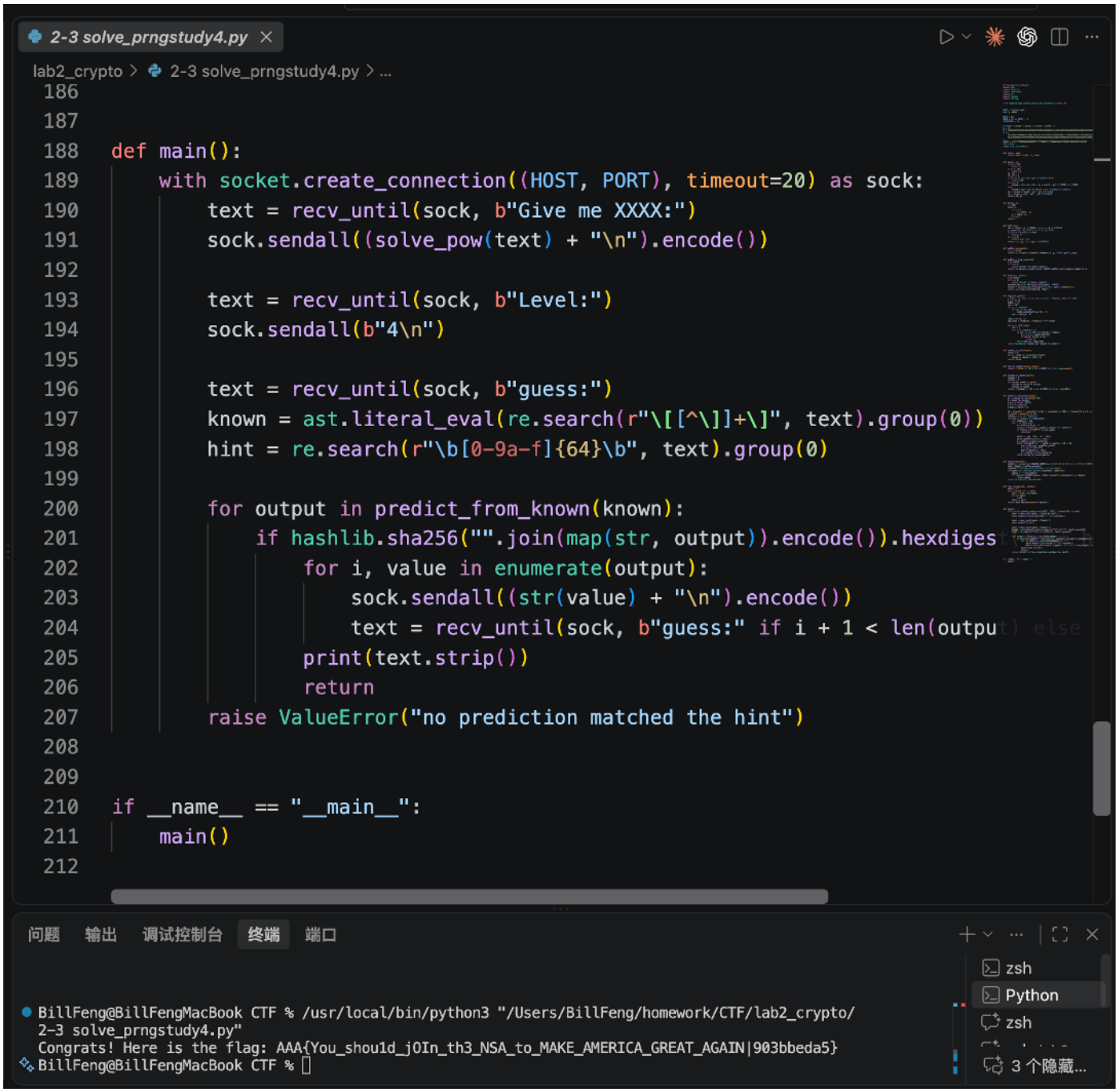

Image 7

```
AAA{You_shou1d_jOIn_th3_NSA_to_MAKE_AMERICA_GREAT_AGAIN|903bbeda5}
```

2.4 Democratic Signature Agency (15 pts)

`nc zjusec.com 30085`。菜单里可以看 Description，题目把部分签名代码直接给出来了

```python
def sign(message, private_key, p, q, g):
    global k
    r = pow(g, k, p) % q
    digest = bytes_to_long(hashlib.sha256(message).digest())
    s = pow(k, -1, q) * (private_key * r + digest) % q
    k += 1
    return r, s
```

DSA 里最重要的一点是 nonce `k` 必须保密且不可预测。这里它虽然没有重复，但每次只是 `k += 1`，所以拿两次签名就知道第二次的 nonce 是第一次的 `k+1`。这比真正随机差很多。

设两次签名分别是 `(r1, s1)` 和 `(r2, s2)`，两条式子就是：

$s_1 k \equiv x r_1 + h_1 \pmod{q}$

$s_2 (k+1) \equiv x r_2 + h_2 \pmod{q}$

这里 `x` 是私钥，`h1, h2` 是两个消息的 sha256。两个未知数是 `k` 和 `x`，两个方程刚好可以消元。先解出 `k`，再代回去得到私钥

```
r1_inv = pow(r1, -1, q)
k = (h2 - r2 * h1 * r1_inv - s2)
k *= pow(s2 - r2 * s1 * r1_inv, -1, q)
k %= q

private_key = (s1 * k - h1) * r1_inv % q
```

拿到私钥以后，就可以给目标字符串 `Plz give me the flag!` 自己签名。这里我直接选 `forge_k = 1`，算出 `(r, s)` 后交给 Verify。

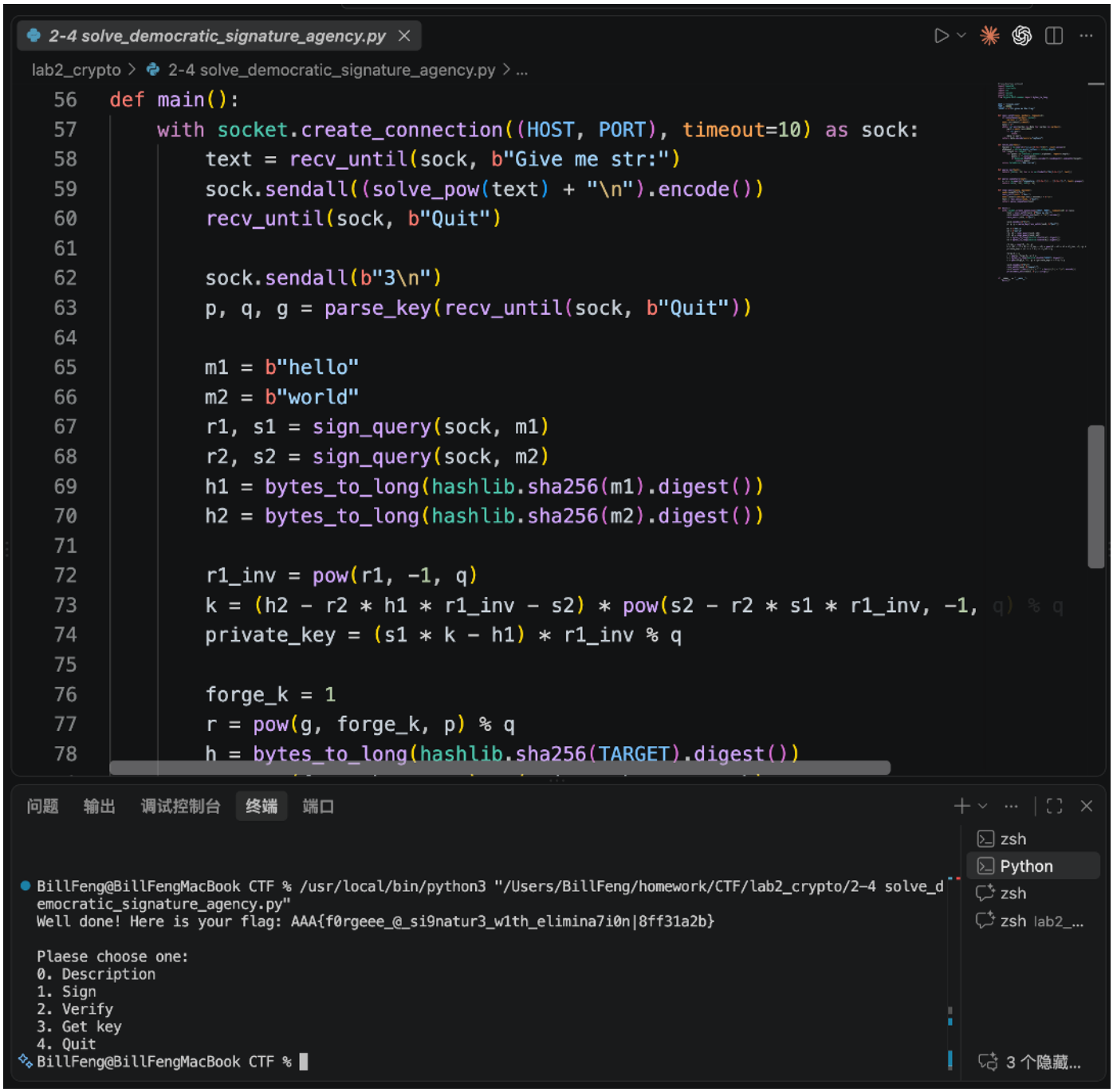

Image 8

```
AAA{f0rgeee_@_si9natur3_w1th_elimina7i0n|8ff31a2b}
```