Challenge 1

解题思路

- 首先看到图片的水印以为是进行了隐写操作，想到用blind-watermark进行恢复，但是拿到一串乱码，还拿到cyberchef尝试解码了一下无果，就试着先看看别的方法
- 然后反过去看lab上的提示，使用 binwalk 检查文件末尾是否叠加了多余的文件，但是好像显示一切正常?

```
DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
0             0x0             JPEG image data, JFIF standard 1.01
26657         0x6821          JPEG image data, JFIF standard 1.01
```

- 然后接着试一下foremost的分离文件命令`foremost -i 1.jpg`(1是我保存后的命名)，产生了output文件夹，点进去发现flag出现了!

```
root@Petrichor:/mnt/c/Users/petrichor0/Desktop/misclab# foremost -i 1.jpg
Processing: 1.jpg
|*|
```

提交flag截图

Challenge 2

解题思路

- 首先发现链接打不开，但是右键链接发现可以将它保存到本地，便尝试保存为miao.png格式，之后用010editor打开，发现网页源码，并看到图片的地址！

```
<html>
<body>
    <script>
        for (;;) {
            alert('Miao~');
        }
    </script>
    <img src="img/miao~870F6C667A6CDC0D1F533859E72C48E0.jpg" />
</body>
</html>
```

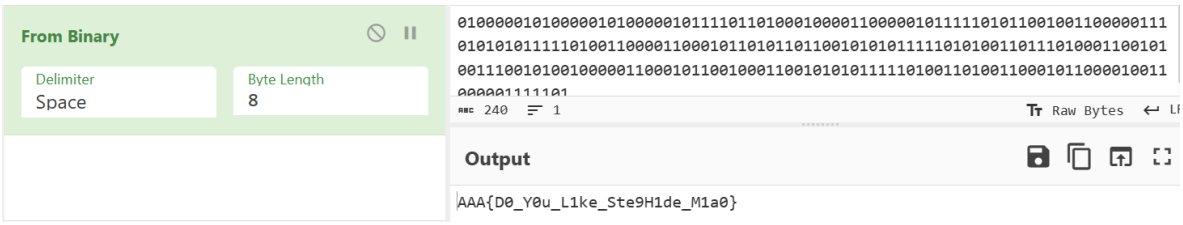
- 访问网站miao~870F6C667A6CDC0D1F533859E72C48E0.jpg (1188×852) (zjusec.com)成功得到图片
- 首先将该图片用exiftools打开，发现该图片有个加密的密钥，想到上课说找到类似密码的一般是工具题

```
Artist                                  : key:m1a0@888
```

便使用steghide

```
root@Petrichor:/mnt/c/Users/petrichor0# cd Desktop
root@Petrichor:/mnt/c/Users/petrichor0/Desktop# steghide extract -sf miao.jpg -p m1a0@888
wrote extracted data to "secret_file.txt".
```

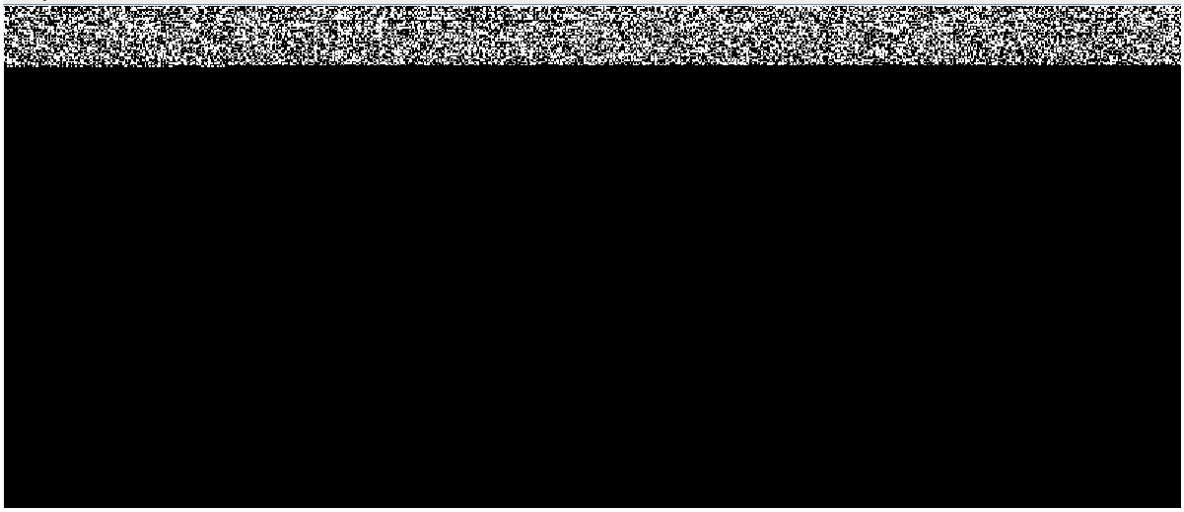
打开secret_file发现一串01序列，便用cyberchef打开，使用from binary工具得到flag！



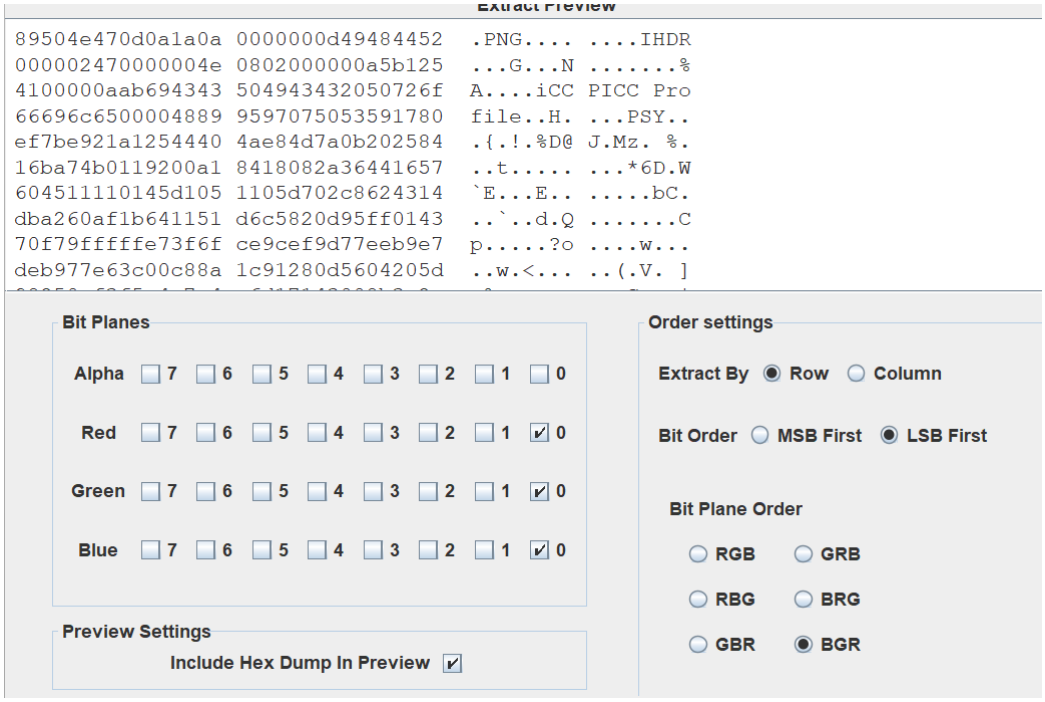
提交flag成功截图

Challenge 3

由题目提示可得该题为LSB隐写，于是将图片用stegsolve打开，观察red green blue的plane 0发现图片上方有隐藏的信息



于是打开Data Extract窗口，勾选bit plane中rgb的第0位，选中lsb first然后挨个尝试，突然发现如下截图的情况在preview窗口中出现了PNG.....IHDR！说明可以得到一个新的png文件!



于是save bin，打开该png文件，便得到flag

Challenge 4: Palette Stego

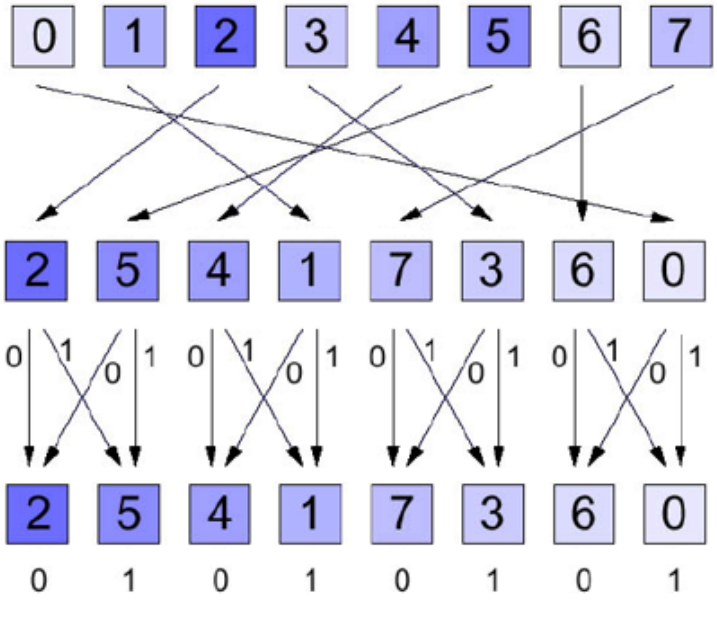
PILE chunk

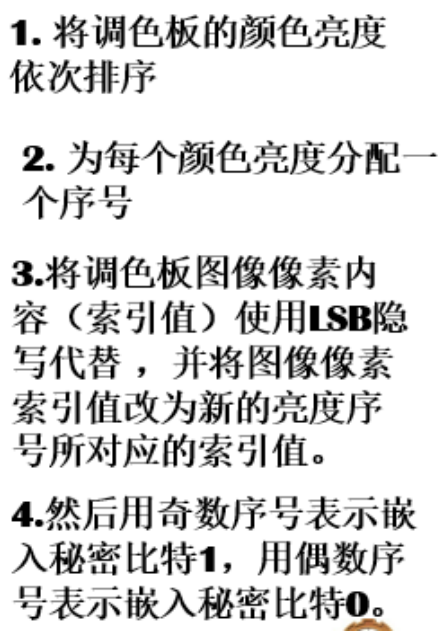
PILE（调色板图像）含有一个不超过256种颜色的调色板，并定义了每种颜色对应的R,G,B各颜色分量值，图像内容中的每个像素是不超过8比特信息的一个索引值，其指向的调色板中的对应颜色就是该像素中的真实颜色。

EZStego隐写

步骤

- 将调色板的颜色亮度依次排序，其中颜色的亮度由不同的颜色分量线性叠加而成，表达式为 $Y = 0.299R + 0.587G + 0.114B$
- 为每个颜色分配一个亮度序号
- 将调色板图像像素内容使用LSB隐写代替，并将图像像素索引值改为新的亮度序号所对应的索引值
- 用奇数序号表示嵌入秘密比特1，用偶数序号表示嵌入秘密比特0





有关本道题

该题好像用zsteg就能直接找到flag?

使用命令行 zsteg -a chal.png | grep AAA（因为知道flag的格式，开头肯定是AAA(●'◡'●)

```
root@Petrichor:/mnt/c/Users/petrichor0/Desktop# zsteg -a chal.png | grep AAA
b1,g,lsb,xy         .. text: "...AAA{gOoD_joB_P4lEtTE_M0D3_c@N_al$0_57E9o!}......AAA{gOoD_joB_P4lEtTE_M0D3_c@N_al$0_57E9
o!}......AAA{gOoD_joB_P4lEtTE_M0D3_c@N_al$0_57E9o!}......AAA{gOoD_joB_P4lEtTE_M0D3_c@N_al$0_57E9o!}......AAA{gOoD_joB_P4
lEtTE_M0D3_c@N_al$0_57E9o!}......AAA{gOoD_joB_"
b1,g,msb,XY         .. text: "$la_N@c_3D0M_ETtEl4P_Boj_DoOg{AAA......}!o9E75_0$la_N@c_3D0M_ETtEl4P_Boj_DoOg{AAA.....}!o
9E75_0$la_N@c_3D0M_ETtEl4P_Boj_DoOg{AAA......}!o9E75_0$la_N@c_3D0M_ETtEl4P_Boj_DoOg{AAA......}!o9E75_0$la_N@c_3D0M_ETtEl
4P_Boj_DoOg{AAA......}!o9E75_0$la_N@c_3D0M_ETt"
b1,g,msb,Xy         .. text: "A......}!o9E75_0$la_N@c_3D0M_ETtEl4P_Boj_DoOg{AAA......}!o9E75_0$la_N@c_3D0M_ETtEl4P_Boj_D
oOg{AAA...g{AAA......}!o9E75_0$la_N@c_3D0M_ETtEl4P_Boj_DoOg{AAA......}!o9E75_0$la_N@c_3D0M_ETtEl4P_Boj_DoOg{AA_DoOg{AAA.
.....}!o9E75_0$la_N@c_3D0M_ETtEl4P_Boj_DoOg{AA"
b1,g,lsb,xY         .. text: "_al$0_57E9o!}......AAA{gOoD_joB_P4lEtTE_M0D3_c@N_al$0_57E9o!}......AAA{gOoD_joB_P4lEtTE_M0
D3_c@N_al$_c@N_al$0_57E9o!}......AAA{gOoD_joB_P4lEtTE_M0D3_c@N_al$0_57E9o!}......AAA{gOoD_joB_P4lEtTE_M0D3_c@NM0D3_c@N_a
l$0_57E9o!}......AAA{gOoD_joB_P4lEtTE_M0D3_c@N"
```

flag即为AAA{{gOoD_joB_P4lEtTE_M0D3_c@N_al $_c@N_al$ 0_57E9o!}}

Challenge 5：Spectrogram

generate.py做的事情

- 首先是定义一些与音频处理相关的参数
- 第一个大部分就是计算音频信号的梅尔频谱图（梅尔频谱图在音频处理中用于表示频率的能量分布，用于语音识别和音频分类任务）
    - 输入音频信号`y`的梅尔频谱图，进行量化处理（感觉是为了降低频谱图的动态范围，使数值更集中，有利于后续处理?
    - 得到表示频谱能量的矩阵
- 第二个大部分就是根据刚刚产生的梅尔频谱图数据生成一个GIF图像
    - 首先将梅尔频谱图矩阵转置，使得每一个列对应每一个时间帧，便于处理
    - 接下来对每一列频谱图数据进行处理，首先生成列表包含min_db到max_db的数值并从大到小排序，然后处理每一行的像素数据。处理完之后生成二维数组，表示一帧图像
    - 最后使用Kronecker乘积扩展每个像素，然后将每一帧的图像数据保存在gif_data列表中，得到GIF动画。

我的思路

首先很抱歉！我的代码一直跑不起来，先是在我的本机（python 3.10.6)试了numpy librosa不同的版本号无果，再用anaconda环境，尝试了3.7,3.8,3.9,以及base的3.11也都在vscode或者spydier下报错，所以无法完整做出这道题目/(ㄒoㄒ)/~所以在报告中阐述一下我的大致思路/(ㄒoㄒ)/

- 首先将GIF中每一帧图像数据存储在一个列表里，还原每一帧图像的频谱信息
- 然后合并频谱信息，得到完整的梅尔频谱图
- 逆转梅尔频谱图得到原始频谱图
- 将频谱图转换回音频信号

Challenge 6:Huffman Stego

学习到的相关知识

JPEG中的范式哈夫曼编码

- 哈夫曼压缩：将数据拆分成一个个符号，统计每个符号出现的频率，根据频率构建出二叉树，并根据二叉树为每个符号值分配二进制位编码
- 想要正确解码，要保存编码后的二进制位以及编码树信息
- 关键问题：如何应用尽量少的数据保存编码树信息

自适应的哈夫曼编码

- 使用事先约定好的编码树，编码树信息直接嵌在编码器和解码器的源代码当中，这样压缩数据中就不需要再保存
- 自适应的哈夫曼编码
    - 初始化模型，获取符号，编码符号后再动态更新模型

范式哈夫曼编码

- 最小编码长度的第一个编码必须从0开始
- 相同长度编码必须是连续的
- 编码长度为j的第一个符号可以从编码长度为$j-1$的最后一个符号所得知，即$c_j = 2(c_{j-1}+1)$

JPEG文件格式与DHT

每一个分区的基本格式如下

```
0xFF+Tag
data length
data
```

当tag为0xc4，表示为DHT，用于保存哈夫曼编码表

用 Symbol表示编码前的原始值，用 Code表示哈夫曼编码后的二进制数据。哈夫曼编码后是个二进制位串，用Code Length来表示二进制位数。

数据

```
                     0   2   2   2   1
 3   2   5   2   4   5   5   0   3   0
 0   1   2   0   3   4  11  21   5  12
31  13  41   6  22  32  51  61  14  71
23  81  91  a1  15  42  b1  c1  d1   7
33  52  e1  f0  24  62  f1
```

描述了表格

```
Code length  | Number | Symbol
-------------+--------+----------
 1 bit       | 0      |
 2 bits      | 2      | 0x01 0x02
 3 bits      | 2      | 0x00 0x03
 4 bits      | 2      | 0x04 0x11
 5 bits      | 1      | 0x21
 6 bits      | 3      | 0x05 0x12 0x31
 7 bits      | 2      | 0x13 0x41
 8 bits      | 5      | 0x06 0x22 0x32 0x51 0x61
 9 bits      | 2      | 0x14 0x71
10 bits      | 4      | 0x23 0x81 0x91 0xa1
11 bits      | 5      | 0x15 0x42 0xb1 0xc1 0xd1
12 bits      | 5      | 0x07 0x33 0x52 0xe1 0xf0
13 bits      | 0      |
14 bits      | 3      | 0x24 0x62 0xf1
15 bits      | 0      |
16 bits      | 0      |
……
```

而由code位数还原为code的方法，则由编码的三条规则给出，其中第三条要进行修正，为
$c_j = (c_{j-k} + 1) << k$

总结

JPEG哈夫曼编码的隐写原理：在 JPEG编解码的过程中会使用到计算出来的哈夫曼编码，这些编码与 DHT 块有关。而在实际解码的过程中，即使有编码没有被使用到，也不会认为出错了。所以可以在哈夫曼编码树上插入实际没有用的编码，从而隐藏信息。