# De la friture sur la ligne

**Catégorie**: Divers
**Difficulté**: Introduction
**Points**: 100

## Sommaire



## Ressources

- Numpy librairie: https://numpy.org/doc/stable/user/index.html
- Bit de parité: https://oger.perso.math.cnrs.fr/fdls/bitparite.pdf

## Introduction

## Annalyse

Pour ce challenge un ficher (challenge.zip) nous ai donné, une fois unzip on se retrouve avec - challenge.py - channel_1 - channel_2 - channel_3 - channel_4 - channel_5 - channel_6 - channel_7 - channel_8

En inspectant le fichier challenge.py on comprend que c'est un script qui a servi a transformer le ***flag.png*** en binaire reparti parmi les 8 fichiers channels, donc l'idée c'est de faire la même manipulation mais dans le sens inverse pour se faire regardons étape par étape comment challenge.py transforme le flag en 8 fichiers text contenant la version binaire du flag.

Étape 1

Le script démarre avec la function **encode_file** qui prend en paramètre le chemin de notre **flag.png**. Il va en effet pour chaque **7** bits de notre PNG calculer un bit de parité et l'ajouté puis continué jusqu'au dernier

```
def encode_data(d):
    return list(d)+[sum([e for e in d])%2]

def encode_file(f):
    # Read a file and convert it to binary
```

```
    _bytes = np.fromfile(f, dtype = "uint8")
    bits = np.unpackbits(_bytes)
    output = []
    # Encode it for more data integrity safety ;)
    for i in range(0,len(bits),7):
        encoded = encode_data(bits[i:i+7])
        output += encoded.copy()
    return np.array(output,dtype="uint8")
```

On arrive donc à la boucle qui est la parti la plus intéressante puisqu'elle va **encoder** notre "flag.png" (donc notre array *bits*) en ajoutant un *bit de parité* pour chaque 7 elements

> [!NOTE]
> *Lors de la transmission de données avec un bit de parité, l'émetteur compte le nombre d'une des données transmises. Si le compte est impair, le bit de parité est défini à 1 pour que le nombre total d'un soit même. Si le compte est pair, le bit de parité est fixé à 0.*

Étape 2

Ensuite le script continue avec la 2eme fonction **transmit** qui lui va découper chaque octets notre *flag encodé* en 8 channels (array) pour ensuite la dispatcher dans 8 fichiers differents. Cependant le *channel 4* va être complétement friturisé

```
def save_channel(data,channel):
    with open("channel_"+str(channel),"w+") as f:
        f.write(''.join(data.astype(str)))

def transmit(data):
    # Time to send it !
    # Separate each bits of each bytes
    to_channel_1 = data[0::8]
    to_channel_2 = data[1::8]
    to_channel_3 = data[2::8]
    to_channel_4 = data[3::8]
    to_channel_5 = data[4::8]
    to_channel_6 = data[5::8]
    to_channel_7 = data[6::8]
    to_channel_8 = data[7::8]
    # Send it to good channel (I hope)
    from_channel_1 = good_channel(to_channel_1)
    from_channel_2 = good_channel(to_channel_2)
    from_channel_3 = good_channel(to_channel_3)
    from_channel_4 = bad_channel(to_channel_4)  # Oups :/
    from_channel_5 = good_channel(to_channel_5)
    from_channel_6 = good_channel(to_channel_6)
    from_channel_7 = good_channel(to_channel_7)
```

```
    from_channel_8 = good_channel(to_channel_8)
    # It's up to you now ;)
    save_channel(from_channel_1,1)
    save_channel(from_channel_2,2)
    save_channel(from_channel_3,3)
    save_channel(from_channel_4,4)
    save_channel(from_channel_5,5)
    save_channel(from_channel_6,6)
    save_channel(from_channel_7,7)
    save_channel(from_channel_8,8)

def good_channel(data):
    return data
def bad_channel(data):
    return (data+np.random.randint(low=0,high=2,size=data.size,dtype='uint8'))%2
```

## Solution

Donc on a bien compris on doit faire la même manipulation mais dans le sens sauf que pour avoir notre flag en clair il faut trouver un moyen de *réparer* ce fameux channel 4 qui a été *friturisé*. Heureusement pour nous notre chère admin **acmo0** a eu la présence d'esprit d'encodé chaque 7 bits avec un bit de parité (dont ils se trouvent tous dans le channel 8) avant de transmettre le flag. C'est ces bits de parité qui va nous aider a *réparer* ce channel 4.

Regardons étape par étape ce qu'il se passe dans ma solution

Étape 1

Pour faire simple cette fonction `decode_channel` va lire le channel (fichier de bits) qu'on lui donne en parametre et va ensuite creer un **Numpy array** avec. On récupère donc nos 8 channels avec `decode_transmission`.

```
def decode_channel(channel):
    with open("channel_" + str(channel), "rb") as f:
        data = np.fromfile(f, dtype='uint8')

    # Convert ASCII values to integers 0 and 1
    bits = data - 48

    return bits

def decode_transmission():
    # Load data from each channel
    from_channel_1 = decode_channel(1)
    from_channel_2 = decode_channel(2)
    from_channel_3 = decode_channel(3)
    from_channel_4 = decode_channel(4)
```

```
    from_channel_5 = decode_channel(5)
    from_channel_6 = decode_channel(6)
    from_channel_7 = decode_channel(7)
    from_channel_8 = decode_channel(8)

    # Reconstruct the transmitted data
    transmitted_data = np.zeros( (len(from_channel_1), 8), dtype=int )
    transmitted_data[:, 0] = from_channel_1
    transmitted_data[:, 1] = from_channel_2
    transmitted_data[:, 2] = from_channel_3
    transmitted_data[:, 3] = from_channel_4
    transmitted_data[:, 4] = from_channel_5
    transmitted_data[:, 5] = from_channel_6
    transmitted_data[:, 6] = from_channel_7
    transmitted_data[:, 7] = from_channel_8


    # Recover the original data by removing the checksum and converting back to bytes
    original_data = np.concatenate(transmitted_data)

    return original_data
```

Étape 2

Une fois tout nos channels reconstituer dans un seul **Numpy array** on va pouvoir réparer le channel 4 grace au bit de parité qui a été injecté pour chaque 7 bits (le fameux channel 8) et dans la foulé retrouver le contenu original du flag

```
def decode_file(bits):

    input = []
    for i in range(0,len(bits),8):
        decoded = decode_data( bits[i:i+8] )
        input += decoded.copy()

    _bytes = np.packbits(input)

    return _bytes
```

Donc ici 2 scénarios avec `check_parity`
si c'est `True` => le channel 4 n'a pas été perturbé
si c'est `False` => le channel 4 a besoin d'être réparé (on va juste flip le bit du chan4)

```
def decode_data(d):

    if ( check_parity(d) == False ):
```

```
        d[3] = (d[3] + 1) % 2

    return list( d[:-1] )
```

C'est ici qu'on va faire notre calcule de parité, exactement le même qui a été fait pour *encodé* notre flag, à la difference ici qu'on va comparer le resultat avec le bit de parité calculé avant la transmission des 8 channels.

```
def check_parity(data):
    # Calculate the sum of all bits except the parity bit
    sum_bits = sum(data[:-1])

    # Calculate the expected parity based on the sum
    expected_parity = sum_bits % 2

    # Check if the expected parity matches the actual parity bit
    return expected_parity == data[-1]
```

Et voila ! on a l'entiereté de notre flag, il ne nous reste plus qu'a creer un fichier PNG et le remplir de nos bits décodé
La solution en entier

```
import numpy as np



##############################
#          2nd PART          #
##############################

def check_parity(data):
    # Calculate the sum of all bits except the parity bit
    sum_bits = sum(data[:-1])

    # Calculate the expected parity based on the sum
    expected_parity = sum_bits % 2

    # Check if the expected parity matches the actual parity bit
    return expected_parity == data[-1]



def decode_data(d):

    if ( check_parity(d) == False ):
        d[3] = (d[3] + 1) % 2

    return list( d[:-1] )
```

```
def decode_file(bits):

    input = []
    for i in range(0,len(bits),8):
        decoded = decode_data( bits[i:i+8] )
        input += decoded.copy()

    _bytes = np.packbits(input)

    return _bytes




##############################
#          1st PART          #
##############################

def decode_channel(channel):
    with open("channel_" + str(channel), "rb") as f:
        data = np.fromfile(f, dtype='uint8')

    # Convert ASCII values to integers 0 and 1
    bits = data - 48

    return bits



def decode_transmission():
    # Load data from each channel
    from_channel_1 = decode_channel(1)
    from_channel_2 = decode_channel(2)
    from_channel_3 = decode_channel(3)
    from_channel_4 = decode_channel(4)
    from_channel_5 = decode_channel(5)
    from_channel_6 = decode_channel(6)
    from_channel_7 = decode_channel(7)
    from_channel_8 = decode_channel(8)
    # Reconstruct the transmitted data
    transmitted_data = np.zeros( (len(from_channel_1), 8), dtype=int )
    transmitted_data[:,0] = from_channel_1
```

```
    transmitted_data[:,1] = from_channel_2
    transmitted_data[:,2] = from_channel_3
    transmitted_data[:,3] = from_channel_4
    transmitted_data[:,4] = from_channel_5
    transmitted_data[:,5] = from_channel_6
    transmitted_data[:,6] = from_channel_7
    transmitted_data[:,7] = from_channel_8


    # Recover the original data by removing the checksum and converting back to bytes
    original_data = np.concatenate(transmitted_data)

    return original_data




##############################
#           START            #
##############################

if __name__ == "__main__":
    transmission = decode_transmission()
    flags = decode_file(transmission)

    with open("flag.png", "wb") as f:
        f.write(bytes(flags))
```

## Autheur

**Tondelier Jonathan**