Solving the challenge

- ## Step1

"I found data left by Noopsy who was analysing a security lock before getting captured. Noopsy might have been able to glitch its RNG.
This document was also laying around:
https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186.pdf

Can you help me recover the secret to clone a security badge and open the secure door to free Noopsy."
(File: SignatureLog.txt)

- ### Solve

The signatures in the log file have 2 components, from this it can be deducted the signatures could be ECDSA ones. They are 256 bits long which is consistent with the ECDSA assumption.
When analysing the different signatures, there are 5 cases where the first component of the signature (r) is the same:

line 10
Sign("Approve access for ID 16400215")
>> 0xa53c9ec6c45a6d1d0347b09cb36f3bea52ac37d8e22f4d7cc344db033901bdc6
0xa2022a236027f16a528774474569946190622 8baaa1124a5f5bfcd0aaeb34b43
line 136
Sign("Approve access for ID 16400201")
>> 0xa53c9ec6c45a6d1d0347b09cb36f3bea52ac37d8e22f4d7cc344db033901bdc6
0x43e149561f9246162d275e6a7354d53a9ce23a727c965a5fdc61c983df53bc4
line 145
Sign("Approve access for ID 16400294")
>> 0xa53c9ec6c45a6d1d0347b09cb36f3bea52ac37d8e22f4d7cc344db033901bdc6
0x4a785c3f9055c48ac136118d3e2d7159a9123fe3b3e47aca74ae7b48a2ddb49
line 212
Sign("Approve access for ID 16400262")
>> 0xa53c9ec6c45a6d1d0347b09cb36f3bea52ac37d8e22f4d7cc344db033901bdc6

0x13b526ba5fa132026f105b70bfb1b07dc3394e233d89adb50d9a022fbed54dc

line 241

Sign("Approve access for ID 16400262")

>> 0xa53c9ec6c45a6d1d0347b09cb36f3bea52ac37d8e22f4d7cc344db033901bdc6

0x13b526ba5fa132026f105b70bfb1b07dc3394e233d89adb50d9a022fbed54dc

r repeating on different signatures happens when the nonce is reused (which is also hinted in the text file saying the RNG might have been glitched).

Going from this guess, the private key of the ECDSA computation can be recovered using two message signatures generated with different messages and the same nonce:

- $k = (s1-s2)^{-1} * (H(M1) - H(M2))$
- $privateKey = r^{-1} * ((s1 * k) - H(M1))$

To be able to perform these computations, the attacker needs to find which curve was used and which hash was used. The NIST documentation *'laying around'* gives P-256 in its documentation (https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-186.pdf) and secp256k1 in the appendix (with a link to https://www.secg.org/sec2-v2.pdf).

The correct curve is secp256k1.

The sha used is SHA-256 which should be the default assumption as dealing with 256-bit data.

Under these assumptions, it is possible to compute:

```
import hashlib

m1 = 'Approve access for ID 16400201'
m2 = 'Approve access for ID 16400294'

hm1 = hashlib.sha256(m1.encode()).digest()
hm2 = hashlib.sha256(m2.encode()).digest()

s1 = 0x43e149561f9246162d275e6a7354d53a9ce23a727c965a5fdc61c983df53bc4
s2 = 0x4a785c3f9055c48ac136118d3e2d7159a9123fe3b3e47aca74ae7b48a2ddb49

n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

k = (pow((s1-s2), -1, n)*(int.from_bytes(hm1) - int.from_bytes(hm2)))%n
```

>> k = 0x7a2dfed52190212bfc81f258b6878fc95f874c57787b7131aa76a5d001d2df37

```
import hashlib

m1 = 'Approve access for ID 16400201'

hm1 = hashlib.sha256(m1.encode()).digest()

r = 0xa53c9ec6c45a6d1d0347b09cb36f3bea52ac37d8e22f4d7cc344db033901bdc6
s1 = 0x43e149561f9246162d275e6a7354d53a9ce23a727c965a5fdc61c983df53bc4

k = 0x7a2dfed52190212bfc81f258b6878fc95f874c57787b7131aa76a5d001d2df37

n = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141

privateKey = (pow(r, -1, n)*((s1 * k) - int.from_bytes(hm1)))%n
```

>> privateKey =
0x2a20207a317020506135735730724 43a207343345f643363527950372320202a

Printing privateKey as ascii gives:
* z1p Pa5sW0rD: sC4_d3cRyP7# *

Step 1 is completed.

- ## Step2

Unzip Secret.zip with the password from the previous step: sC4_d3cRyP7#

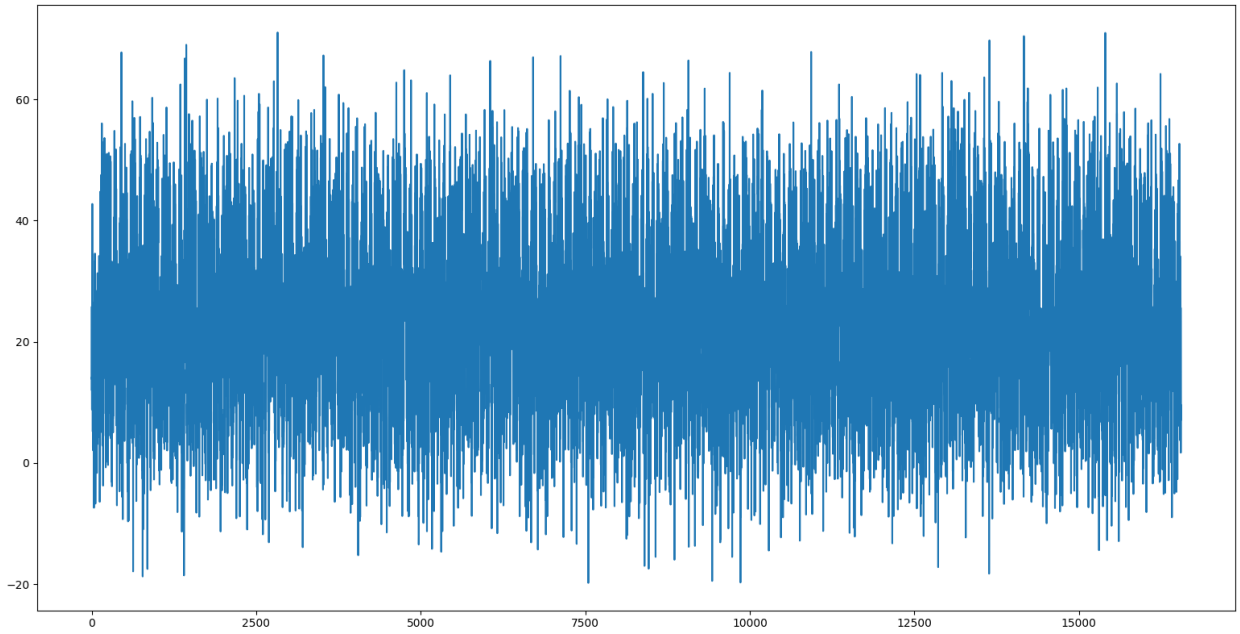
You get:

Yay! We made it past the first step to rescue Noopsy.
Wasn't that a good sign? But now we go to the next task when we have the time!
The comms seem stuck with one single operation now.
Some things add up and some don't! Remember, patience is key for rescuing Noopsy!
(File: traces_ECC.npy)

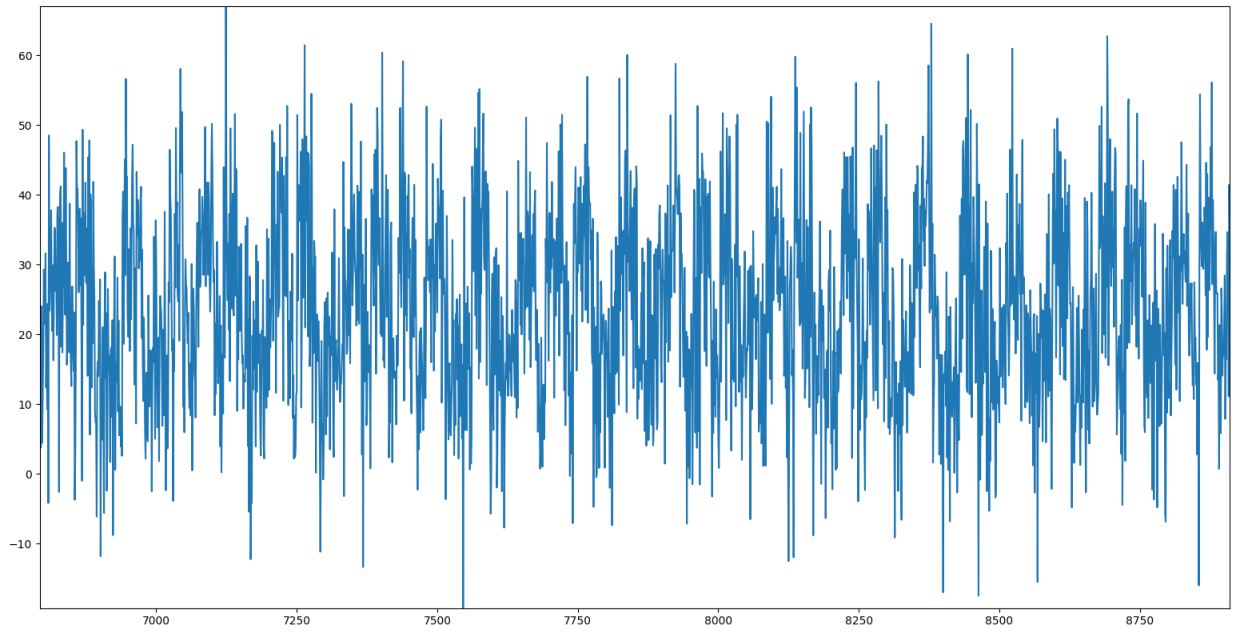
○ Solve

The file provided is a ".npy", which is a python numpy format. The name of the file hints that it contains side channel traces.
Once loaded in python, the numpy array has a size of [20,16551], the file contains 20 traces. It is possible to view these traces (e.g. with matplotlib)

```
import numpy as np
import matplotlib.pyplot as plt

traces = np.load('traces_ECC.npy', allow_pickle = True)

plt.plot(traces[0])
plt.show()
```

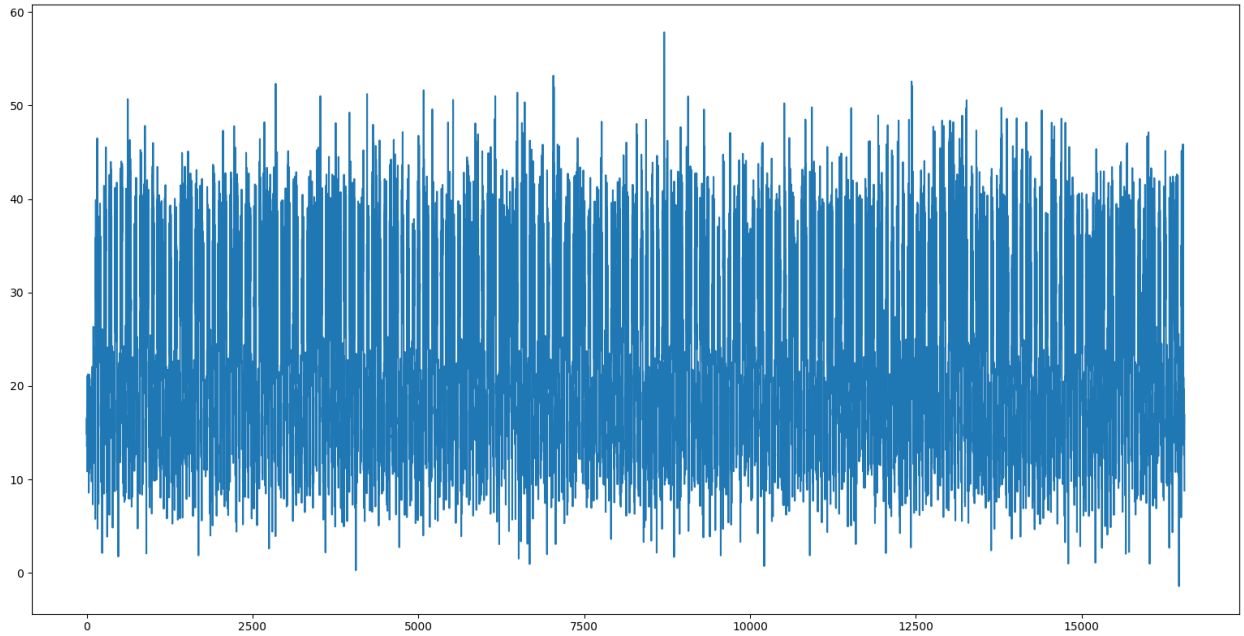


Zooming in this trace, some patterns can be kind of distinguished:

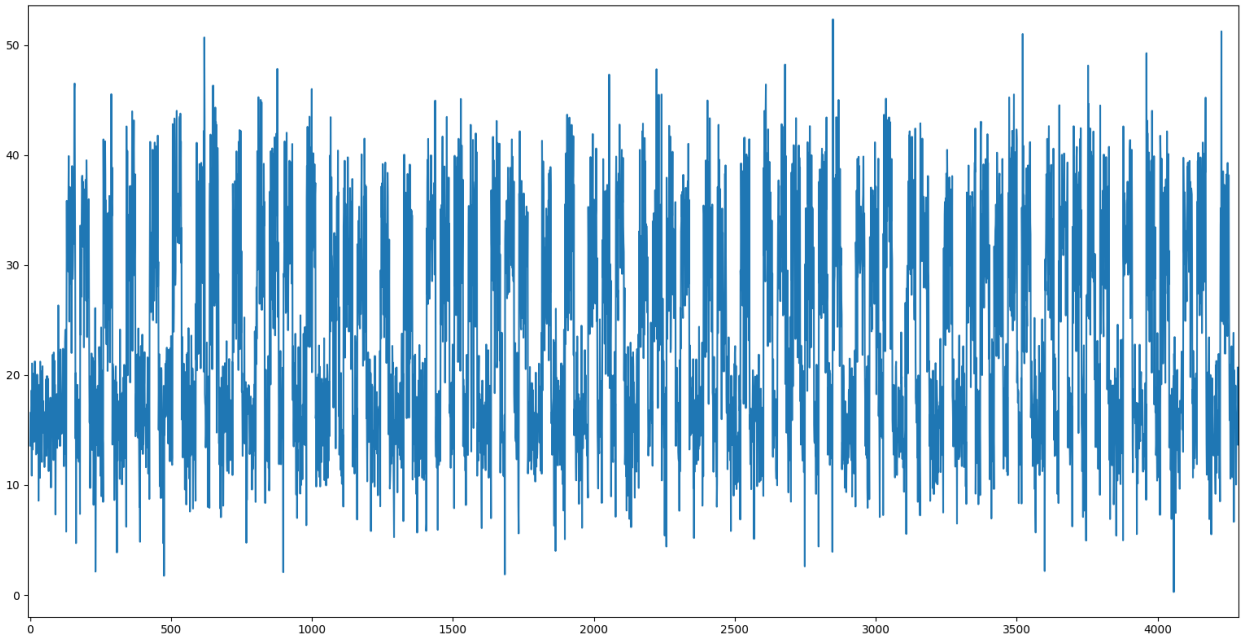


The introduction text says '*The comms seem stuck with one single operation now*', it is possible to improve the signal to have better patterns by averaging the traces.

```
import numpy as np
import matplotlib.pyplot as plt

traces = np.load('traces_ECC.npy', allow_pickle = True)

av_trace = np.mean(traces, axis = 0)

plt.plot(av_trace)
plt.show()
```



Zooming in, it is now quite clear there are different patterns in the trace:



The file name being traces_ECC.npy, it can be deduced this trace is probably representing a scalar multiplication. The different patterns remind of the classical square and multiply SPA on RSA, which can be applied to Double and Add on scalar multiplication.
There are two options:

- Iterative algorithm, index increasing:

```
let bits = bit_representation(s)  # the vector of bits (from LSB to MSB) representing s
let res = 𝒪 # point at infinity
let temp = P  # track doubled P val
for bit in bits:
    if bit == 1:
        res = res + temp  # point add
    temp = temp + temp  # double
return res
```

- Iterative algorithm, index decreasing:

```
let bits = bit_representation(s)  # the vector of bits (from LSB to MSB) representing s
let i = length(bits) - 2
let res = P
while (i >= 0):  # traversing from second MSB to LSB
    res = res + res  # double
    if bits[i] == 1:
        res = res + P  # add
    i = i - 1
return res
```

In both cases, the double operation is always performed and the add operation is performed only if the bit is 1. The average trace shows that the high patterns sometimes follow each other when the low pattern is always between high patterns. Therefore, the low pattern has to represent an add operation and the high pattern represent a double operation.
Considering this, the trace can be used to recover the bits with two options depending on LSB to MSB or MSB to LSB:

- LSB to MSB:
  - (low_pattern + high_pattern): bit = 1
  - high_pattern: bit = 0
- MSB to LSB:
  - (high_pattern + low_pattern): bit = 1
  - high_pattern: bit = 0

The first pattern is a low_pattern so the only option is that the computation is LSB to MSB. The bits can then be recovered by reading the patterns on the trace one by one (‘*patience is key*’), alternatively a program with a threshold/pattern recognition can be written.

The recovered bits are: (lsb to msb)

10111110110011001110001001110010001011000001001011000010000110101100110011111010110011000011001000001010101100101000110011001010111110100010110011

11101000101010110010101010101001010010111110100100101000001100011000101101111011001010000010100000110001110011

Then reverse those bits (to msb to lsb) convert to bytes and check the ascii:

```
bitstring =
'10111110110011001110001001110010001011000001001011000010000110101100110011111010110011000011001000001010101100101000110011001010111110100001011001111101000101011001010101010100101001011111010010010100000110001100010110111101100101000001010000011000111001'

int(bitstring[::-1], 2).to_bytes((len(bitstring[::-1]) + 7) // 8, byteorder='big')
```

>> b'N0PS{F0R_JUST_4_S1MPL3_3XCH4NG3}'

Step 2 is completed.

Flag is: N0PS{F0R_JUST_4_S1MPL3_3XCH4NG3}