




































# CTF writeup for CS2107 AY24/25 Sem2, Assignment 1

Written by: Low Yu Xuan (Bachelor of Science in Mathematics)  
Student number: A0272353H

## Easy 1: Birthday emoji cipher.

### Step 1: Initial Pattern Recognition

- Found two clear patterns to work with:
  - Beginning message: "

```
a='😂😂😂😂😂😂😂😂' # Known emojis for 'cs2107'
b='cs2107' # Known characters
c='😂😂😂😂😂😂😂😂😂😂😂😂😂😂😂😂😂😂' # Known emojis for 'happy birthday'
d='happy birthday' # Known characters
dic = {} # Dictionary for substitutions

# Building the substitution dictionary
for i in range(len(a)):
    dic[a[i]] = b[i]
for i in range(len(c)):
    dic[c[i]] = d[i]
```

One could see that after the initial substitution, you would obtain bits of the words. Of which you can proceed to play 'wordle' and guess the remaining substitutions.

```
happy birthday!! h💯p💯y💯e💯r💯 d💯a💯y💯's💯 🌱💯s💯,💯 🌱💯d💯,💯 a💯d💯 a💯 th💯 th💯i💯s💯 y💯e💯.💯 y💯e💯 d💯e💯s💯r💯 th💯 b💯e💯s💯t💯,💯 s💯 a💯a💯 s💯e💯r💯 | Python
```

Final flag cs2107{skibidi\_emoji\_language\_go\_brrrrrr}

## Easy 2:

The challenge provided a hash: e82d0df0e1001e2c33bc12900faf06e5

The task was to crack this hash to reveal the original password.

Solution:

1. Hash Analysis:
  - Hash appeared to be an MD5 hash (32 characters, hexadecimal)
  - Mode 0 in hashcat corresponds to MD5
2. Command Used:

bash

Copy

```
hashcat -m 0 e82d0df0e1001e2c33bc12900faf06e5 rockyou.txt --show
```

Breaking down the command:

- `-m 0`: Specifies MD5 hash mode
  - The hash value: e82d0df0e1001e2c33bc12900faf06e5
  - `rockyou.txt`: A commonly used wordlist for password cracking
  - `--show`: Displays the cracked hash results
3. Tools Used:
    - Hashcat: A popular password cracking tool
    - rockyou.txt: A well-known password dictionary containing millions of real-world passwords
  4. Methodology:
    - Used a dictionary attack approach
    - Leveraged the rockyou.txt wordlist, which contains common passwords from data breaches

- Hashcat computes MD5 hashes of each word in rockyou.txt and compares them with our target hash

Google “Hashcat” for more details.

Flag = cs2107{cs2107711923}

## Easy 3:

Probably the easiest question, as you could directly see the 1 to 1 bijective mapping at each stage. You realistically only need to look at the first row and the last row of the map.

```
import string as str_lib

string = 'P K D F U B L I S Z H N X T A Y G J Q
O M C V R E W'
string = string.split()
alphabet = list(str_lib.ascii_uppercase)

dic = zip (alphabet, string)
dic = dict(dic)
cipher = 'KUO_OISQ_OAAH_EAM_P_VISNU'
plain = ''
for i in range(len(cipher)):
    if cipher[i] in dic:
        plain += dic[cipher[i]]
    else:
        plain += '_'
plain
```

## Medium 1:

Notice that since AES pads in blocks of 16, the vulnerability lies when each byte of the folder is mapped into a block sized output for AES, since the encrypt function pads every single byte into length 16.

```

thing = ''# Flag is the secret and will be in the format CS2107{50m3th1ng}

# 3 things to import
from Crypto.Cipher import AES
from random import randbytes
from Crypto.Util.Padding import pad

# Randomly generated key!
cipher = AES.new(randbytes(16),AES.MODE_ECB)
secret = "
'''
print(len(thing))

```

You could notice that the plaintext's first 265 letters are always the same. Hence it suffices to find the corresponding dictionary mapping from any byte to a block of encrypted AES block.

```

with open('output', 'rb') as file:
    binary_data = file.read()
    s2_blocks = [s2[i:i+16] for i in range(0, 264*16, 16)]

source = ''# Flag is the secret and will be in the format CS2107{50m3th1ng}

# 3 things to import
from Crypto.Cipher import AES
from random import randbytes
from Crypto.Util.Padding import pad

# Randomly generated key!
cipher = AES.new(randbytes(16),AES.MODE_ECB)
secret = "[REDACTED]"
code = open("herring.py", "rb").read()

with open("test_output", "wb") as f:
    for i in code:
        f.write(cipher.encrypt(pad(bytes([i]),16)))'''

dic = {}
for i in range(264):
    if s2_blocks[i] not in dic:

```

```

        dic[s2_blocks[i]] = source[i]
    else:
        continue
dic

```

By doing this, we essentially have a bijective(likely) mapping from any byte of the pt to the AES ct.

```

{b'i\x8cb\xa5\xe0^\x19\xe7\xb0\xbd|\x8eIp\xeeR': '#',
 b'\xaa\xa8\xdfy\x8bq\xfa\xcb\xfd#M\x81(\xfb}\xf9': ' ',
 b']\xc9\xa5xeb*\xb6\x0b\x024\x0c\xfd\xa3\xaa\xa1\xaa': 'F',
 b'x\x9ee\xf3\x8f\x91j4\x1a6psXe\x07\x9c': 'l',
 b'\xe5bL B\x83R\xd2IC\xbd\x1a\x0b\x92\x1d': 'a',
 b'\x95\x15\x89\xb2\xe0u\xad.\xa1Bf\xf3u\xf5\x1aG': 'g',
 b'\xe0\x1cY\xba\xbd$\xe4\xad\x85\x85\xbb\xeb\x06\xa9\xf4': 'i',
 b'"\xfb\x9d\x10!\x1ePI\xd5\xc4VA\x94\xbdxi': 's',
 b'(\xe6\x15\xa7\t\xe7i8\xd5\x18\xa8u\xcb^\x1c\xc0': 't',
 b'\xdf>S\x00\xb7\x10;5\xd0a&\xca\x04\xfd\xc5\xc8': 'h',
 b'z\x0f\xc6,\xe5\xf6T\x83Fz\x97\x0f l\xee\x13': 'e',
 b'A\x94\xff+Cb&\x8b\x9f\xb9Ej\x8a\x81\xfa\xc7': 'c',
 b' /\x1d\xde\xc7\nb\xc4\x94\x19\xa3\xc0\x00'\xb6[\xaf': 'r',
 b'\x07\x9e\x88,KY\x11&\xaf\x8e\x8a\xef<s\n\xf1': 'n',
 b'\xd2n+%PZ\xd2\xdf>\xb9\x82M:* \xc6\xb9': 'd',
 b'\xca\xc5\xce\xec\xf2v\xb2\xddV\x84\x99\xbd\\\xc9P\xd5': 'w',
 b'E\xf8\x97@p\x1eAE\x10\x99\x8bk\x87o\xed\x0b': 'b',
 b'\xa4|\xfa7j\xef\xceE\x03-]\x02\xad\xe6\xd9d': 'f',
 b'\xaa@\xd3n\x97\xdfg\xb9\xdc\E\x81\xbd\xe2\x90\xbd': 'o',
 b'\x9a\x13\xb1\x8c\x8fA\xefP\x01\xc5\x97\x15\x9f\x92\xab\xf2': 'm',
 b'E\xfaU\xa0&\x11=!\x0fy\xba#\xaf\x14p\x0e': 'C',
 b'\x83^\x1e\x9a\xc0"\xbblx!\xa1k\x0c\x13a': 'S'.

```

^ Example of mappings.

Using this, you can easily decrypt the ciphertext.

```

decrypted = 'CS2107{byt3_by_byt3_3ncrypt10n_15_k1ll1ng_my_AES!!}'

```

## Medium 2:

Notice that  $\text{isqrt}(pq) + 1 = \text{Arithmetic mean of } P \text{ and } Q$ .

My initial idea was knowing that the geometric mean and arithmetic mean was pretty close and hence the entropy wasn't a lot to bash out. But the above observation speeds things up pretty fast since we have  $pq=N$  already. With this, we can obtain  $P+Q$  and construct  $\phi(N) = (P-1)(Q-1)$ . From here, we can compute the decryption key since finding the inverse is fast in the finite field under  $Z$ .

AI Declaration: Asked ChatGPT what are good ways to factorize  $N=pq$ , where  $p$  and  $q$  are very near each other.

```
from math import isqrt
def fermat_factor(n):
    a = isqrt(n)
    if a % 2 == 0:
        a += 1

    while True:
        b2 = a*a - n
        b = isqrt(b2)
        if b*b == b2:
            p = a + b
            q = a - b
            if isPrime(p) and isPrime(q):
                return q, p
        a += 1

p, q = fermat_factor(integer)
```

This was the original idea, which was slower but relied on the fact that prime gaps aren't very big.

## Medium 3:

This question was rather simple as you could just notice that you can produce two different plaintext and encrypt them, and when you multiply them, the secret key would be cancelled.

E.g. let's say u have  $c_1$  and  $c_2$ , where  $p_1$  and  $p_2$  differ by 1 only

When you take their difference, u obtain  $key[i \% \text{len key}] (1)$

And since u can factor out the key

Hence if you let  $m_1[i]$  and  $m_2[i]$  differ by 1, a lot of nice cancellations occur and you can compute the flag.

```
def flagGenerator(A,B,prev):  
    inv = pow(B-A, -1, 1114111) # 65-FLAG  
    #1114005 = (65 - flag) * key  
    flag = (65 + prev - (A * inv)) % 1114111  
    return flag  
chr(flagGenerator(765664,765716,186186))
```

Hence, it suffices to query something like

A, B -> then `flagGenerator(A,B,0)` generates `flag[0]`

AA, AB -> `flagGenerator(AA,AB, A)` generates `flag[1]`

AAA,AAB -> `flagGenerator(AAA,AAB,AA)` generates `flag[2]`

And so on.

From here, one should be able to obtain `CS2107{w0w_y0u_c4n_mult1ply_by_0}`

## Hard 1:

Import `randcrack`, and feed it 312 x 2 samples after breaking each input into two blocks.

The only note is that `randbytes` takes in little endian encoding, which is where you might need to be careful. One might want to find out how `randcrack` works here

<https://github.com/tna0y/Python-random-module-cracker>.

```

import random, time
from randcrack import RandCrack

random.seed(time.time())

rc = RandCrack()

for i in range(624):
    rc.submit(int.from_bytes(random.randbytes(4), 'little'))
    #rc.submit(int(random.randbytes(4).hex(), 16))
    # Could be filled with random.randint(0,4294967294) or random.randrange(0,4294967294)

print("Random result: {}\nCracker result: {}".format(random.randbytes(8), rc.predict_getrandbits(64).to_bytes(8, 'little')))

Random result: b'\xb1\xf9\x84\x95\xe2T\xee\x0c'
Cracker result: b'\xb1\xf9\x84\x95\xe2T\xee\x0c'

```

Example of how to use randcrack. Note the `int.from_bytes()` method having little endian encoding.

AI declaration(ChatGPT) : “How can I write a script to repeatedly input into a server that i SSH’ed into?”

```

for i in range(312):
    channel.send('0\n')
    time.sleep(0.1)

    response = ''
    while 'Your guess: ' not in response:
        if channel.recv_ready():
            response += channel.recv(1024).decode()

    hex_value = response.split('was ')[1].split('.')[0]
    samples.append(hex_value)
    print(f"Sample {i+1}: {hex_value}")

    # Convert to bits and feed to predictor
    byte = bytes.fromhex(hex_value)

    #bits = hex_to_bits(hex_value)
    first_4 = byte[:4]
    second_4 = byte[4:]
    rc.submit(int.from_bytes(first_4, 'little'))
    rc.submit(int.from_bytes(second_4, 'little'))

```

You can then proceed to call the output of the PRNG generator 312 times, noting to split the output into 2 before pushing it into randcrack.

Doing this, you should obtain the flag CS2107{untwist\_me!}



## Hard 2:

When you NC into the server, it outputs the server time. Since this is the basis for the seed, you can simply copy this and init the srand with srand(server\_timestamp). Following this, you can use this to generate the AES key that was used in the encryption scheme.

Obtaining this symmetric key, one should be able to decrypt the encrypted message.

AES\_ECB\_decrypt(...); (Note that encryption was done using ECB, so you must decrypt using ECB too)

AI Declaration(ChatGPT): “How can I use this AES.c file code to decrypt, given that i have the key?”

```
// Initialize AES context
struct AES_ctx ctx;
AES_init_ctx(&ctx, key);

// Convert hex encrypted flag to bytes
unsigned char encrypted[32] = {0};
unsigned char decrypted[32] = {0};
unsigned char block1[16] = {0};
unsigned char block2[16] = {0};

// Convert hex to bytes
hex_to_bytes(encrypted_hex, encrypted, 32);

// Split into blocks and decrypt separately
memcpy(block1, encrypted, 16);
memcpy(block2, encrypted + 16, 16);

// Decrypt each block
AES_ECB_decrypt(&ctx, block1);
AES_ECB_decrypt(&ctx, block2);
```