

SUBSTITUTION1

The challenge gives us an encrypted message and source code.

```
HO IFSBZXUFABDS, A QCVQZHZCZHXO IHBDKF HQ A GKZDXW XR KOIFSBZHOU HO JDHID COHZQ XR
BNAHOZKYZ AFK FKBNAIKW JHZD ZDK IHBDKFZKYZ, HO A WKRHOKW GAOOKF, JHZD ZDK DKNB XR A
PKS; ZDK "COHZQ" GAS VK QHOUNK NKZZKFQ (ZDK GXQZ IXGGXO), BAHFQ XR NKZZKFQ, ZFHBKZQ XR
NKZZKFQ, GHYZCFKQ XR ZDK AVXMK, AOW QX RXFZD. ZDK FKIKHMKF WKIHBDKFQ ZDK ZKYZ VS
BKFRXFGHOU ZDK HOMKFQK QCVQZHZCZHXO BFXIKQQ ZX KYZFAIZ ZDK XFHUHOAN GKQQAUK. DKFK
HQ ZDK RNAU: DIGCQ-IZR{ODAONCO_NHPKQ_ZX_BNAS_IFSBZXUFAG}
```

```
from fnmatch import translate
```

```
from random import shuffle
```

```
msg = ''
```

```
with open('../Secret/msg.txt') as file:
```

```
    msg = file.read().upper()
```

```
ALPHABET = b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

```
SUB_ALPHABET = list(b'ABCDEFGHIJKLMNOPQRSTUVWXYZ')
```

```
shuffle(SUB_ALPHABET)
```

```
translate_dict = {}
```

```
for u, v in zip(ALPHABET, SUB_ALPHABET):
```

```
    translate_dict[u] = v
```

```
with open('msg_enc.txt', 'w+') as file:
```

```
    file.write(msg.translate(translate_dict))
```

By looking at the source, we can tell that is a substitution cipher. I use this website to solve it <https://www.guballa.de/substitution-solver>

Input

Cipher Text:

```
HO IFSBZXUFABDS, A QCVQZH ZCZH XO IHBDKF HQ A GKZDXW XR KOIFSBZHOU HO
JDHID COHZQ XR BNAHOZKYZ AFK FKBNAIKW JHZD ZDK IHBDKFZKYZ, HO A
WKRHOKW GAOOKF, JHZD ZDK DKNB XR A PKS; ZDK "COHZQ" GAS VK QHOUNK
NKZZKFQ (ZDK GXQZ IXGGXO), BAHFQ XR NKZZKFQ, ZFHBKZQ XR NKZZKFQ,
GHYZCFKQ XR ZDK AVXMK, AOW QX RXFZD. ZDK FKIHKMKF WKIHBDKFQ ZDK ZKYZ
VS BKFRXFGHOU ZDK HOMKFQK QCVQZH ZCZH XO BFXIKQQ ZX KYZFAIZ ZDK
XFHUHOAN GKQQAUK. DKFK HQ ZDK RNAU: DIGCQ-
IZR{ODAONCO_NHPKQ_ZX_BNAS_IFSBZXUFAG}
```

Language:

Break Cipher

Clear Cipher Text

Result

Clear text [\[hide\]](#)

Key	abcdefghijklmnopqrstuvwxyz	This clear text ...
	awiwkrudhtpngoxblfqzcmjyse	... maps to this cipher text

Clear text:

```
IN CRYPTOGRAPHY, A SUBSTITUTION CIPHER IS A METHOD OF ENCRYPTING IN
WHICH UNITS OF PLAINTEXT ARE REPLACED WITH THE CIPHERTEXT, IN A
DEFINED MANNER, WITH THE HELP OF A KEY; THE "UNITS" MAY BE SINGLE
LETTERS (THE MOST COMMON), PAIRS OF LETTERS, TRIPLETS OF LETTERS,
MIXTURES OF THE ABOVE, AND SO FORTH. THE RECEIVER DECIPHERS THE TEXT
BY PERFORMING THE INVERSE SUBSTITUTION PROCESS TO EXTRACT THE
ORIGINAL MESSAGE. HERE IS THE FLAG: HCMUS-
CTF{NHANLUN_LIKES_TO_PLAY_CRYPTOGAM}
```

Bingo!!!!. Flag : HCMUS-
CTF{NHANLUN_LIKES_TO_PLAY_CRYPTOGAM}

SUBSTITUTION2

MOTRVZLGEYDQCWBHDHDLJZDQDESRSAGGUYNLYDWOFGTDFGOZMGAQKZMFEGTFESLWBYWRYR
MESHFETMMZUJQDVYIJLHFSMNQLJIKCREGTODKGGBUHFESGQOYHFQSUZXBRDYFRDJKOTQOZUMSRMR
FDSAUSUYGMZMFBFAHDXGUNRBMFIDIKETGGTUJMAYEFBMIVQUEYMAZJYHRKIMSEJKDMYKKJQHNCR
DKMGUYMKYLOTQORMKOHFJMYLGROFGRFSDTAMNQLQKRDOCHDUESLWBYWRREQQFXKSOZMAZPIBE
YIJKFBANKZTPAQDVZSOTOYUZNURZFBTGUTOQLYLDQDQMRGUTJKJMGQPHIDNQHODKMKZMMZMBE
RMJTXLHWUYPJIMVFOOSPUYLDJJRREEQFXMMSUZMAZHIBEYKJKFUZVZYEBESRFKFDLJZVAMQBRRG
KHJFAUDKTMRNMTDTFGJMAZPIBEKMTRVTFMKZMMRMBFDGYPHAYKZLWCFBSCSFQVSWZMZIESYHTDA
GMFQ_JVYYRB_QFX_PKKEYKVSQJREHA_HD_JZGBMR_FBFG_HG

```
from random import shuffle

ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
n = len(ALPHABET)
sbox = [i for i in range(n)]
shuffle(sbox)

def transform(msg, offset):
    msg = ALPHABET.index(msg)
    return (sbox[msg] + offset) % n

msg = ''
with open('../Secret/msg.txt') as file:
    msg = file.read().upper()

special_symbols = "`~!@#$%&*()-=+[]\\';',./{}|:~<>? "
for ch in special_symbols:
    msg = msg.replace(ch, '')

msg_enc = ''
offset = 1
for i in range(0, len(msg), 5):
    for j in range(5):
        print(j)
        # Sorry the code looks so ugly, but you know what it does :)
        if msg[i+j] == '_':
            msg_enc += '_'
        else:
            msg_enc += ALPHABET[transform(msg[i+j], offset)]
    offset = (offset * 3 + 4) % n

with open('msg_enc.txt', 'w+') as file:
    file.write(msg_enc)
```

The second challenge is similar to the first one, but this time, the whole message is divided into 5 character – blocks, and characters in each block are encrypted with the same offset, so the method above won't work with this one. We can notice that, assume the sbx is random (It doesn't have to be the same as original), if we reverse the code, it will return some random message – which is the substitution encryption. So my idea behind this is to first reverse the code then solve it using substitution cipher.

My reverse source code:

```
from random import shuffle

ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
n = len(ALPHABET)
sbx = [i for i in range(n)]
shuffle(sbx)

def transform(msg, offset):
    msg = ALPHABET.index(msg)
    return (sbx[msg] + offset) % n

msg = ''
with open('../Secret/msg.txt') as file:
    msg = file.read().upper()

special_symbols = "`~!@#$%&*()-=+[\\;','./{}|:~\"<>? "
for ch in special_symbols:
    msg = msg.replace(ch, '')

msg_enc = ''
offset = 1
for i in range(0, len(msg), 5):
    for j in range(5):
        print(j)
        # Sorry the code looks so ugly, but you know what it does :)
        if msg[i+j] == '_':
            msg_enc += '_'
        else:
            msg_enc += ALPHABET[transform(msg[i+j], offset)]
    offset = (offset * 3 + 4) % n

with open('msg_enc.txt', 'w+') as file:
    file.write(msg_enc)
```

Input

Cipher Text:

IYBCSBVLNRVRFNWAWAWERWBTDVQXBIRLQQRQAVRVNUAMBVMQYBQLXDIKYAQQBVFRVUS
RUBLBYQRAVFBQQBYWRVTLDQEAQRQADVWDIFBQQBYWDLLSYKAQXMRNAVUIYBCSBVLABW
ODYBDMBYQXBYBAWRLXRYRLQBYAQALTAWQYAESQADVDIFBQQBYWQXRQAWYDSUXFNQXBW
ROBIDYRFODWQRFFWROHFBWDIQXRQFRVUSRUBIDYAVWQVRVLBUAMBVRWBLQADVDIBVUFAW
XFRVUSRUBBQRRVTDYRBYQXBODWQLDOODVKXAFBJCGRVTPRYBYRYBFAZBKAWBQXBYDVRVT
RVRYBQXBODWQLDOODVHRAYWDIFBQQBYWQBYOBTEAUROWDYTAUYRHXWRVTWWBBQQRVTI
IRYBQXBODWQLDOODVYBHRQWQXBVDVWBVBHXYRWBBQRDAVWXYTFSYBHYBWBVQWQXBOD
WQIYBCSBVQFBQQBYWAVQNHARFBVUFAWXFRVUSRUBQBGQXLOS WLQIVD_WHRLBW_RVT_H
DFNRFXHREBQAL_AW_WBLSYB_AWVQ_AQ

Language: English ▼

Break Cipher

Clear Cipher Text

Result

Clear text [\[hide\]](#)

Key	abcdefghijklmnopqrstuvwxyz	This clear text ...
	reltbiuxapzfovdhcywqsmkgnj	... maps to this cipher text

Clear text:

FREQUENCYANALYSISISBASEDONTHEFACTTHATINANYGIVENSTRETCHOFWRITTENLANGU
AGECERTAINLETTERSANDCOMBINATIONSOFLETTERSOCURWITHVARYINGFREQUENCIES
MOREOVERTHEREISACHARACTERISTICDISTRIBUTIONOFLETTERSTHATISROUGHLYTHES
AMEFORALMOSTALLSAMPLESOFTHATLANGUAGEFORINSTANCEGIVENASECTIONOFENGLIS
HLANGUAGEETAANDOARETHEMOSTCOMMONWHILEZQXANDJARERARELIKewisETHERONAND
ANARETHEMOSTCOMMONPAIRSOFLETTERSTERMEDBIGRAMSORDIGRAPHSANDSSEETTANDF
FARETHEMOSTCOMMONREPEATSTHENONSENSEPHRASEETAOINSHRDLUREPRESENTSTHEMO
STFREQUENTLETTERSINTYPICALENGLISHLANGUAGETEXTHCMUSCTFNO_SPACES_AND_P
OLYALPHABETIC IS SECURE ISNT IT

Flag : HCMUS-

CTF{NO_SPACES_AND_POLYALPHABETIC_IS_SECURE_ISNT_IT}

OH SEED

```
import time
import random
import threading
import socketserver
```

```

n = 2**32-2 # 32 bits
FLAG_FILE = "flag.txt"
class Service(socketserver.BaseRequestHandler):
    def handle(self):
        self.flag = self.get_flag()
        l = self.gen_random()
        self.send("WELCOME TO RANDOM MACHINE!\n")
        self.send("Here is the first 665 random numbers.\n")
        self.send(" ".join(map(str, l[:-1])) + "\n")

        user_input = int(self.receive("Now it's your turn to guess the last
random number:\n"))
        #print(user_input, l[-1])
        if (user_input == l[-1]):
            self.send("I hope you're not guessing.\n")
            self.send("Here is your flag.\n")
            self.send(self.flag + "\n")
        else:
            self.send(f"Sorry {l[-1]} != {user_input}\n")

    def get_flag(self):
        with open(FLAG_FILE) as f:
            return f.readline()

    def gen_random(self):
        random.seed(time.time() + random.randint(0, 999999) + 1312 +
hash(self.flag))
        results = [random.randrange(0, n) for i in range(666)]
        return results

    def send(self, string: str):
        self.request.sendall(string.encode("utf-8"))

    def receive(self, prompt):
        self.send(prompt)
        return self.request.recv(1000).strip().decode("utf-8")

class ThreadedService(socketserver.ThreadingMixIn,
                        socketserver.TCPServer,
                        socketserver.DatagramRequestHandler,):
    pass

def main():
    port = 20202

```

```

host = "192.168.1.8"

service = Service
server = ThreadedService((host, port), service)
server.allow_reuse_address = True
server_thread = threading.Thread(target=server.serve_forever)
server_thread.daemon = True
server_thread.start()

print("Server started on " + str(server.server_address) + "!")
# Now let the main thread just wait...
while True:
    time.sleep(10)

if __name__ == "__main__":
    main()

```

In this challenge, you are given 665 random numbers and you have to guess the value of 666th number. In order to solve this, you have to know a little about how RNG(random number generator) in python work. Basically, the RNG algorithm is based on Mersenne Twister, which is not cryptographically secure. Here is some information I found about MT:

<https://crypto.stackexchange.com/questions/12426/is-a-mersenne-twister-cryptographically-secure-if-i-truncate-the-output>

https://en.wikipedia.org/wiki/Mersenne_Twister

Disadvantages [edit]

- Relatively large state buffer, of 2.5 KiB, unless the TinyMT variant (discussed below) is used.
- Mediocre throughput by modern standards, unless the SFMT variant (discussed below) is used.^[42]
- Exhibits two clear failures (linear complexity) in both Crush and BigCrush in the TestU01 suite. The test, like Mersenne Twister, is based on an \mathbf{F}_2 -algebra.^[39] There are a number of other generators that pass all the tests (and numerous generators that fail badly).^[clarification needed]
- Multiple instances that differ only in seed value (but not other parameters) are not generally appropriate for Monte-Carlo simulations that require independent random number generators, though a method for choosing multiple sets of parameter values.^{[43][44]}
- Poor diffusion: can take a long time to start generating output that passes randomness tests, if the initial state is highly non-random—particularly if the initial state has many zeros. A consequence is that two instances of the generator, started with initial states that are almost the same, will usually output nearly the same sequence for many iterations, before eventually diverging. The 2002 update of the MT algorithm has improved initialization, so that beginning with such a state is very unlikely.^[45] The GPU version (MTGP) is said to be even better.^[46]
- Contains subsequences with more 0's than 1's. This adds to the poor diffusion property to make recovery from many-zero states difficult.
- Is not cryptographically secure, unless the CryptMT variant (discussed below) is used. The reason is that observing a sufficient number of iterations (624 in the case of MT19937, since this is the state vector from which future iterations are produced) allows one to predict all future iterations.

And I also found that there is a python module which can crack python's RNG. Here is the full code:

```

from pwn import remote
from randcrack import RandCrack

```

```

rc = RandCrack()
r = remote("103.245.250.31", 30620)
res = r.recvuntil(b"guess the last random
number:\n").decode().split('\n')[2].strip()
leak = [int(i) for i in res.split(' ')]
print(leak)
for i in range(624):
    rc.submit(leak[i])
    # Could be filled with random.randint(0,4294967294) or
    random.randrange(0,4294967294)

for i in range(665 - 624):
    print(f"Leak: {leak[624 + i]}")
    print("Cracker result: {}".format(rc.predict_randrange(0, 4294967295 - 2)))
    print("*"*20)
ans = rc.predict_randrange(0, 4294967295 - 2)
print("Cracker result: {}".format(ans))
r.sendline(str(ans).encode())

r.interactive()

```

```

*****

```

```

Leak: 2562285664

```

```

Cracker result: 2562285664

```

```

*****

```

```

Cracker result: 4197466079

```

```

[*] Switching to interactive mode

```

```

I hope you're not guessing.

```

```

Here is your flag.

```

```

HCMUS-CTF{r4nd0m-1s-n0t-r4nd0m}

```

```

[*] Got EOF while reading in interactive

```

```


```

Flag : HCMUS-CTF{r4nd0m-1s-n0t-r4nd0m}

SIGNME

```

import hashlib
from Crypto.Util.number import *
from random import randint
from os import urandom
from base64 import b64decode, b64encode
from hashlib import sha256

with open("../flag.txt") as file:

```



```

FLAG = file.read()

class SignatureScheme:
    def __init__(self) -> None:
        self.N = len(FLAG)
        assert self.N == 32

        self.p =
99489312791417850853874793689472588065916188862194414825310101275999789178243
        self.x = randint(1, self.p - 1)
        self.g = randint(1, self.p - 1)
        self.y = pow(self.g, self.x, self.p)
        self.coef = [randint(1, self.p - 1) for _ in range(self.N)]

        self.sign_attempt = self.N

    def sign(self, pt):
        if self.sign_attempt == 0:
            print("Sorry, no more attempt to sign")
            return (0, 0)
        else:
            try:
                msg = b64decode(pt)

                if (len(msg) > self.N): # I know you are hecking :(((
                    return (0, 0)

                k = sum([coef * m for coef, m in zip(self.coef, msg)])
                if k % 2 == 0: # Just to make k and p-1 coprime :)))
                    k += 1

                r = pow(self.g, k, self.p)
                h = bytes_to_long(sha256(pt).digest())
                s = ((h - self.x * r) * inverse(k, self.p - 1)) % (self.p - 1)
                self.sign_attempt -= 1
                return (r, s)
            except:
                print('Please send message in base64 encoding')

    def verify(self, pt, r, s):
        if not 0 < r < self.p:
            return False
        if not 0 < s < self.p - 1:
            return False
        h = bytes_to_long(sha256(pt).digest())

```

```
        return pow(self.g, h, self.p) == (pow(self.y, r, self.p) * pow(r, s,
self.p)) % self.p
```

```
def get_flag(self):
```

```
    try:
```

```
        test = b64encode(urandom(self.N))
```

```
        print("Could you sign this for me: ", test.decode())
```

```
        r = b64decode(input('Input r: '))
```

```
        s = b64decode(input('Input s: '))
```

```
        if self.verify(test, bytes_to_long(r), bytes_to_long(s)):
```

```
            print("Congratulation, this is your flag: ", FLAG)
```

```
            exit(0)
```

```
        else:
```

```
            print("Sorry, that is not my signature")
```

```
            exit(-1)
```

```
    except Exception as e:
```

```
        print(e)
```

```
        print("Please send data in base64 encoding")
```

```
def menu(self):
```

```
    print(f"You have only {self.sign_attempt} attempts left")
```

```
    print("0. Get public key")
```

```
    print("1. Sign a message")
```

```
    print("2. Verify a message")
```

```
    print("3. Get flag")
```

```
    return int(input('Select an option: '))
```

```
def main(self):
```

```
    print("Welcome to our sign server")
```

```
    while True:
```

```
        option = self.menu()
```

```
        if option == 0:
```

```
            print("g =", self.g)
```

```
            print("p =", self.p)
```

```
        elif option == 1:
```

```
            msg = input("Input message you want to sign: ").encode()
```

```
            r, s = self.sign(msg)
```

```
            print("Signature (r, s): ", (r, s))
```

```
        elif option == 2:
```

```
            msg = input('Your message: ').encode()
```

```
            r = b64decode(input('Input r: '))
```

```
            s = b64decode(input('Input s: '))
```

```
            if self.verify(msg, bytes_to_long(r), bytes_to_long(s)):
```

```

        print("Valid signature")
    else:
        print("Invalid signature")

    elif option == 3:
        self.get_flag()

    else:
        print("Stay away you hecker :(((")

c = SignatureScheme()
c.main()

```

First, let's look at what we got in main:

```

def main(self):
    print("Welcome to our sign server")
    while True:
        option = self.menu()

        if option == 0:
            print("g =", self.g)
            print("p =", self.p)
        elif option == 1:
            msg = input("Input message you want to sign: ").encode()
            r, s = self.sign(msg)
            print("Signature (r, s): ", (r, s))

        elif option == 2:
            msg = input('Your message: ').encode()
            r = b64decode(input('Input r: '))
            s = b64decode(input('Input s: '))
            if self.verify(msg, bytes_to_long(r), bytes_to_long(s)):
                print("Valid signature")
            else:
                print("Invalid signature")

        elif option == 3:
            self.get_flag()

        else:
            print("Stay away you hecker :(((")

```

So in this challenge, you are given 4 options:

- 0 – Get public key (G and P)
- 1 – Sign the message, it will return R and S
- 2 – Validate signature R and S
- 3 – Get flag

In the get_flag function , we have:

```
def get_flag(self):
    try:
        test = b64encode(urandom(self.N))
        print("Could you sign this for me: ", test.decode())
        r = b64decode(input('Input r: '))
        s = b64decode(input('Input s: '))
        if self.verify(test, bytes_to_long(r), bytes_to_long(s)):
            print("Congratulation, this is your flag: ", FLAG)
            exit(0)
        else:
            print("Sorry, that is not my signature")
            exit(-1)
    except Exception as e:
        print(e)
        print("Please send data in base64 encoding")
```

Basically, it give us a random bytes of length N(N = 32), and request us to sign the message(R and S). If the verification succeed, it will print the flag- which is what we need.

Let's dive into the crypto scheme:

```
def __init__(self) -> None:
    self.N = len(FLAG)
    assert self.N == 32

    self.p =
99489312791417850853874793689472588065916188862194414825310101275999789178243
    self.x = randint(1, self.p - 1)
    self.g = randint(1, self.p - 1)
    self.y = pow(self.g, self.x, self.p)
    self.coef = [randint(1, self.p - 1) for _ in range(self.N)]

    self.sign_attempt = self.N

def sign(self, pt):
    if self.sign_attempt == 0:
```

```

        print("Sorry, no more attempt to sign")
        return (0, 0)
    else:
        try:
            msg = b64decode(pt)

            if (len(msg) > self.N): # I know you are hecking :(((
                return (0, 0)

            k = sum([coef * m for coef, m in zip(self.coef, msg)])
            if k % 2 == 0: # Just to make k and p-1 coprime :)))
                k += 1

            r = pow(self.g, k, self.p)
            h = bytes_to_long(sha256(pt).digest())
            s = ((h - self.x * r) * inverse(k, self.p - 1)) % (self.p - 1)
            self.sign_attempt -= 1
            return (r, s)
        except:
            print('Please send message in base64 encoding')

```

In this code, we have:

- g, P is the public key
- R, S is the signature of message
- X, Y, coef is the secret key

Also, you have limited attempts to sign the message(32 times).

Here is how it works:

Calculate R, S

$\min(32, \text{len}(\text{msg}))$

$$k = \sum_{i=0}^{\min(32, \text{len}(\text{msg}))} \text{msg}[i] * \text{coef}[i]$$

if $k \% 2 = 0$ then $k += 1$ (just to make k and $p-2$ coprime)

$$R = g^k \pmod{p}$$

$h = \text{hash}(\text{msg})$ // it's not important what kind of hash algorithm it use, so I will denote it as $\text{hash}()$

$$S = [(h - x * R) * k^{-1} \pmod{p-1}] \pmod{p-1}$$

Validate message (msg, R, S)

$$h = \text{hash}(\text{msg})$$

$$\text{return } g^h == y^R \cdot R^S \pmod{p}$$

To the verify function:

```
def verify(self, pt, r, s):
    if not 0 < r < self.p:
        return False
    if not 0 < s < self.p - 1:
        return False
    h = bytes_to_long(sha256(pt).digest())
    return pow(self.g, h, self.p) == (pow(self.y, r, self.p) * pow(r, s, self.p)) % self.p
```

Here is the proof:

Thứ ngày tháng năm

proof why it's true

$$y = g^x \pmod{p}$$
$$R = g^k \pmod{p} \text{ // } k \text{ is define in sign function}$$
$$h = \text{hash}(\text{msg})$$
$$s = (h - x \cdot R)k^{-1} \pmod{p-1}$$

$$\begin{aligned} g^h &= g^R \cdot R^s \pmod{p} \\ \Rightarrow g^h &= (g^x)^R \cdot (g^k)^s \pmod{p} \end{aligned}$$
$$\Rightarrow g^h = g^{xR} \cdot g^{k(h - xR)k^{-1}}$$
$$\Rightarrow g^h = g^{xR} \cdot g^{h - xR}$$
$$\Rightarrow g^h = g^{xR} \cdot g^h \cdot (g^{xR})^{-1}$$
$$\Rightarrow g^h = g^h \text{ (obviously)}$$

So, in order to solve the challenge, you have to calculate:

+ coef[i] (1)

+ X (2)

Knowing all values above, we can calculate R and S with any specific message.

(1)

$$R = \text{product}(g^{\text{msg}[i].\text{coef}[i]}), 0 \leq i < 32$$

Assume we want to find $\text{coef}[i]$, then our msg will be $b'\text{x00}' * i + b'\text{x02}' + b'\text{x00}' * (32 - i - 1)$. This will lead to $k = 2 * \text{coef}[i]$, $R = g^{2 * \text{coef}[i]}$. It's easy to find value $g^{\text{coef}[i]}$ but not possible to find $\text{coef}[i]$ as we have to solve dlp (not possible with p in this challenge). Still, there is no need to calculate it because we can simply find $R = (g^{\text{coef}[1]})^{\text{msg}[0]} \cdot (g^{\text{coef}[1]})^{\text{msg}[1]} \dots (g^{\text{coef}[31]})^{\text{msg}[31]}$

(2)

By the fact that k is the control value ($k = 0$ if msg is $b'\text{x00}' * 32$), it's not hard to find X :

$$S = (h - X * R) * k^{-1} \pmod{P - 1} \quad // k = 1$$

$$S = (h - X * R) \pmod{P - 1}$$

$$X = (h - S) * R^{-1} \pmod{P - 1}$$

There are 32 operation (1), 1 operation (2), which is total 33 (only 32 attempt). Is there other way which costs less than 32?

Let's look at the verify function again

```
def verify(self, pt, r, s):
    if not 0 < r < self.p:
        return False
    if not 0 < s < self.p - 1:
        return False
    h = bytes_to_long(sha256(pt).digest())
    return pow(self.g, h, self.p) == (pow(self.y, r, self.p) * pow(r, s, self.p)) % self.p
```

Notice that $Y^R \cdot R^S = (g^X)^R \cdot (g^k)^{(h - X \cdot R)k^{-1}}$, if we assume $k = k^{-1} = 1$ (case (2)), the equation becomes:

$$Y^R \cdot R^S = g^{XR} \cdot g^{h - XR} = g^h$$

This time, S can be calculated without k ($S = h - XR$ because $k = k^{-1} = 1$), and we also know the value R ($R = g^k = g$ in this case). Therefore, our signature will be:

- $R = g$
- $S = h - XR$ (remember that h can be calculated)

Problem solved!!!!

Code:


```

from pwn import *
from base64 import b64encode, b64decode
from Crypto.Util.number import inverse, bytes_to_long, long_to_bytes
from hashlib import sha256
s = remote('103.245.250.31', 31850)
#time to caculate x
def option_select(a : str, s : remote):
    print(s.recvuntil(b'Select an option: ').decode())
    s.sendline(a.encode())

R,S = 0, 0
G,P = 0,0
def get_publicKey(s : remote):
    global G,P
    option_select('0', s)
    G = int(s.recvline().decode()[3:-1])
    P = int(s.recvline().decode()[3:-1])
    print('G =',G)
    print('P =',P)

def get_x(s : remote):
    global G,P
    global R,S
    option_select('1', s)
    print(s.recvuntil(b'Input message you want to sign: ').decode())
    payload = b64encode(b'\x00'*32)
    s.sendline(payload)
    print(s.recvuntil(b'Signature (r, s): ').decode())
    R,S = eval(s.recvline()[:-1].decode())
    print((R,S))
    if R == G:
        print('correct path')

    #this is the part where i caculate x using Magic
    h = bytes_to_long(sha256(payload).digest())
    try:
        XR = (h - S)%(P-1)
        X = (XR*inverse(R, P-1))%(P-1)
    except:
        print('error')
        exit(0)
    return int(X)

def sign_message(s : remote):
    global R,S

```

```

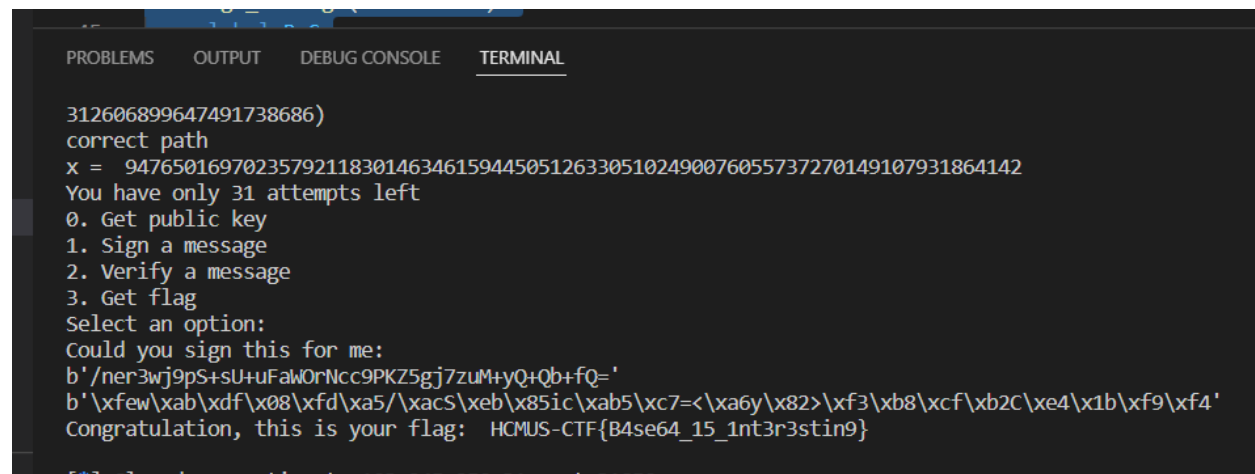
global G,P
x = get_x(s)
print('x = ',x)
option_select('3', s)

print(s.recvuntil(b'Could you sign this for me: ').decode())
msg_chall = s.recvline()[:-1]
print(msg_chall)
print(b64decode(msg_chall))
h = bytes_to_long(sha256(msg_chall).digest())
r = pow(G, 1, P)

chall_S = (h - x*r)*inverse(1, P-1)%(P-1)
s.sendlineafter(b'Input r: ', b64encode(long_to_bytes(R)))
s.sendlineafter(b'Input s: ', b64encode(long_to_bytes(chall_S)))
print(s.recvline().decode())

get_publicKey(s)
sign_message(s)

```



```

312606899647491738686)
correct path
x = 9476501697023579211830146346159445051263305102490076055737270149107931864142
You have only 31 attempts left
0. Get public key
1. Sign a message
2. Verify a message
3. Get flag
Select an option:
Could you sign this for me:
b'/ner3wj9pS+sU+uFawOrNcc9PKZ5gj7zuM+yQ+Qb+fQ='
b'\xfew\xab\xdf\x08\xfd\xa5/\xacS\xeb\x85ic\xab5\xc7=<\xa6y\x82>\xf3\xb8\xcf\xb2C\xe4\x1b\xf9\xf4'
Congratulation, this is your flag: HCMUS-CTF{B4se64_15_1nt3r3stin9}

```

Flag : HCMUS-CTF{B4se64_15_1nt3r3stin9}