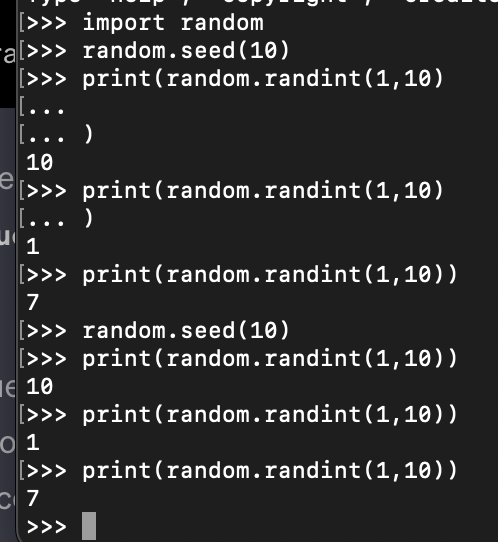
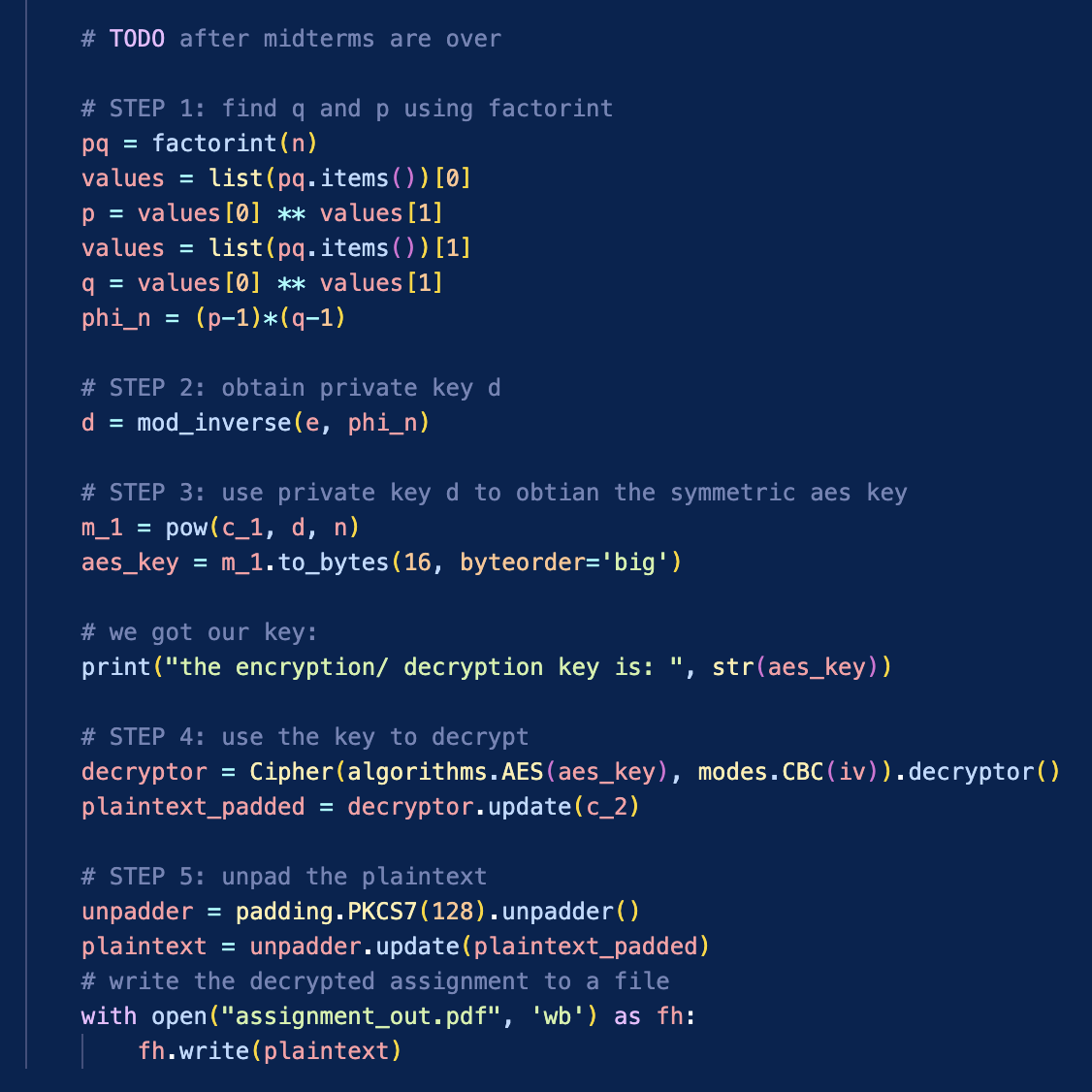
watIAM: sh2yap

student ID: 21111395

2. Hardcoded IV: I would use a random IV each time, otherwise, it the AES-CBC will only reduce to a key problem.
3. The way *e* is generated is incorrect according to the RSA algorithm. It should be a coprime with , but in the implementation, the hacker just used e = random.randint(2 \*\* 128, 2 \*\* 129 - 1). He doesn’t at least check that e is coprime with . If I were the hacker, I would check using
4. random.seed is a deterministic PRNG, and therefore will give the same result if the seed is the same. I would use os.urandom() instead, as it cannot be seeded and it is more random because it uses entropy from a variety of sources. The following picture illustrates how random.randint() gives the same sequence of numbers, if random is initialized with the same seed.



1. seed variable is deterministic, only requires the datetime. If we wanted to get the same keys, all I need to do is to generate a key at the same time as my target. If seeds are ever needed, I would use os.urandom().
2. AES key generation also uses deterministic random.randbytes, instead of os.urandom()
3. p and q are prime numbers next to each other, and are therefore, related, not independently, randomly chosen. This would make it easier to factorize n. Therefore, if I was the hackers, I would use two randomly chosen p and q, where their prime differences are far apart.
4. The number of bits used for RSA key is small, therefore factorization is possible. I would at least 1024-bits.
5. Kindly Refer to the pseudocode and code below for the description. (For the full code, refer to appendix)



1. 38395 is the code.

**Appendix: Full Code**

import base64

import datetime

import json

import random

from cryptography.hazmat.primitives import padding

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

from sympy import mod\_inverse, factorint

from sympy.ntheory import isprime, nextprime

def bytes2string(b):

return base64.urlsafe\_b64encode(b).decode('utf-8')

def string2bytes(s):

return base64.urlsafe\_b64decode(s.encode('utf-8'))

def gen\_rsa\_pk():

bitlength = 256

seed = datetime.datetime.now().strftime("%Y-%m-%d %H:%M")

random.seed(seed)

p = random.randint(2 \*\* (bitlength - 1), 2 \*\* bitlength)

while not(isprime(p)): p = random.randint(2 \*\* (bitlength - 1), 2 \*\* bitlength)

q = nextprime(p)

n = p \* q

e = random.randint(2 \*\* 128, 2 \*\* 129 - 1) # this number is definitely coprime with n

return (n, e)

def do\_encryption():

# read the assignment file to be encrypted

with open("assignment\_in.pdf", 'rb') as fh:

plaintext = fh.read()

# generate the RSA public key

(n, e) = gen\_rsa\_pk()

# generate an AES key and convert it to an integer

aes\_key = random.randbytes(16)

aes\_key\_int = int.from\_bytes(aes\_key, byteorder='big')

# encrypt the AES key using RSA encryption

c\_1 = pow(aes\_key\_int, e, n)

# pad the plaintext to a multiple of the block length

padder = padding.PKCS7(128).padder() # 128 is the block size

padded\_data = padder.update(plaintext)

padded\_data += padder.finalize()

# encrypt the plaintext using AES

iv = b"1337c0487c068711"

cipher = Cipher(algorithms.AES(aes\_key), modes.CBC(iv)).encryptor()

aes\_ct = cipher.update(padded\_data) + cipher.finalize()

# output the data in a JSON data structure for easy parsing

output = {}

output["n"] = n

output["e"] = e

output["iv"] = bytes2string(iv)

output["c\_1"] = c\_1

output["c\_2"] = bytes2string(aes\_ct)

with open("encrypted\_assignment.json.txt", 'w') as fh:

fh.write(json.dumps(output))

def do\_decryption():

# Read and parse the JSON data structure

with open("encrypted\_assignment.json.txt", 'r') as fh:

input = json.loads(fh.read())

n = input["n"]

e = input["e"]

iv = string2bytes(input["iv"])

c\_1 = input["c\_1"]

c\_2 = string2bytes(input["c\_2"])

# TODO after midterms are over

# STEP 1: find q and p using factorint

pq = factorint(n)

values = list(pq.items())[0]

p = values[0] \*\* values[1]

values = list(pq.items())[1]

q = values[0] \*\* values[1]

phi\_n = (p-1)\*(q-1)

# STEP 2: obtain private key d

d = mod\_inverse(e, phi\_n)

# STEP 3: use private key d to obtian the symmetric aes key

m\_1 = pow(c\_1, d, n)

aes\_key = m\_1.to\_bytes(16, byteorder='big')

# we got our key:

print("the encryption/ decryption key is: ", str(aes\_key))

# STEP 4: use the key to decrypt

decryptor = Cipher(algorithms.AES(aes\_key), modes.CBC(iv)).decryptor()

plaintext\_padded = decryptor.update(c\_2)

# STEP 5: unpad the plaintext

unpadder = padding.PKCS7(128).unpadder()

plaintext = unpadder.update(plaintext\_padded)

# write the decrypted assignment to a file

with open("assignment\_out.pdf", 'wb') as fh:

fh.write(plaintext)

do\_decryption()