Assignment 4 Q1

watIAM: sh2yap
student ID: 21111395

a)

1) Hardcoded IV: I would use a random IV each time, otherwise, it the AES-CBC will only reduce to a key problem.
2) The way e is generated is incorrect according to the RSA algorithm. It should be a coprime with $\Phi(n) = (p-1)(q-1)$, 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 $\Phi(n)$. If I were the hacker, I would check using $\gcd\big(e, \Phi(n)\big) = 1$
3) `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.

```
>>> import random
>>> random.seed(10)
>>> print(random.randint(1,10)
...
... )
10
>>> print(random.randint(1,10)
... )
1
>>> print(random.randint(1,10))
7
>>> random.seed(10)
>>> print(random.randint(1,10))
10
>>> print(random.randint(1,10))
1
>>> print(random.randint(1,10))
7
>>>
```

4) `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()`.
5) AES key generation also uses deterministic `random.randbytes`, instead of `os.urandom()`
6) 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.
7) The number of bits used for RSA key is small, therefore factorization is possible. I would at least 1024-bits.

b) Kindly Refer to the pseudocode and code below for the description. (For the full code, refer to appendix)

```
    # 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)
```

c) 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'))
```

Assignment 4 Q1

```
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
```

Assignment 4 Q1

```
    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()
```