Hybrid AES with RSA Mini Project

INTRODUCTION

AES is a short form for **Advanced Encryption Standard** which used to be called *Rijndael* (for its inventor names, *Rijmen* and *Daemen*, pronounced like "rain-dull"). It was one of the 15 candidates in the AES competition held by NIST from 1997 to 2000 to specify "an unclassified, publicly disclosed encryption algorithm capable of protecting sensitive information well into the next century," as stated in the 1997 announcement of the competition in the Federal Register.

The AES competition was kind of a "Got Talent" competition for cryptographers, where anyone could participate by submitting a cipher or breaking other contestant ciphers. AES is a *symmetric block cipher* algorithm, means it process blocks of data using a secret key with size of 128, 192 or 256-bits for encryption and decryption process and is called **encryption key**.

AES was selected for this mini project because of the strong encryption algorithm and one of the most widely adopted block cipher around the world that proven to be very secure for many years. To demonstrate AES encryption in Python, we will use *pycryptodome* library that provides many functions of encryption algorithm including AES with mode of operations for symmetric ciphers including CBC and CTR.

Another encryption algorithm to be used in this project was RSA (*Rivest Shamir Adleman*), an *asymmetric encryption*, which uses two keys: one is **public key** that can be used by anyone who wants to encrypt messages for you. Another one is **private key** that is required in order to decrypt messages encrypted using the public key. Theoretically, by combining two encryption algorithm we're able to create very strong system that provide safe and secure message encryption and decryption.

```
import secrets
import random
import sys
from Crypto.Cipher import AES
from Crypto import Random
```

The first part of code will import all of the necessary libraries needed to run the encryption program. *Secrets*, *random* and *sys* are three modules contains in **Python Standard Library**. Specifically, *secrets* module is used for generating cryptographically strong random numbers suitable for managing data such as passwords, account authentication, security tokens, and related secrets.

Random module implements *pseudo-random number generators* (PRNG) for various distributions, which makes *secrets* in particular should be used in preference to the default pseudo-random number generator in the *random* module, which is designed for modelling and simulation, not security or cryptography.

Sys is a module to provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter, that we will used later.

```
def gcd(a, b):
    "Euclidean algorithm"
    while b != 0:
        temp = a % b
        a = b
        b = temp
    return a
```

This is Python function to return GCD of a and b value. Essentially, *Euclidean algorithm* calculates the **greatest common divisor** (GCD) of two natural numbers a and b. The greatest common divisor is the largest natural number that divides both a and b without leaving a remainder.

```
def multiplicativeInverse(a, b):
    "Euclidean extended algorithm"
    x = 0
    y = 1
    lx = 1
    ly = 0
    oa = a
    ob = b
    while b != 0:
        q = a // b
        (a, b) = (b, a % b)
        (x, lx) = ((lx - (q * x)), x)
        (y, ly) = ((ly - (q * y)), y)
    if lx < 0:
        lx += ob
    if ly < 0:
        ly += oa
    return lx
```

The function of this Python code is to compute the secret exponent $d, 1 < d < \phi(n)$, such that $ed = 1 mod \phi(n)$ that will be called later at line 127. The *Extended Euclidean algorithm* is an extension to the previous *Euclidean algorithm* to calculate GCD of a and b. This function proves that the previous solution to GCD is correct because the GCD is the only number that can simultaneously satisfy this equation and divide the inputs.

```
def generatePrime(keysize):
    while True:
        num = random.randrange(2**(keysize-1), 2**(keysize))
        if isPrime(num):
            return num
```

This Python code that will be used later to generate random number using pseudo-random number generators (PRNG) from random module.

```
def isPrime(num):
    if (num < 2):
    # 0, 1, and negative numbers are not prime
    return False
    lowPrimes = [2, … … … … … … … … … ,997]
    if num in lowPrimes:
        return True
    for prime in lowPrimes:
        if (num % prime == 0):
    return False
    return millerRabin(num)
```

```
# Program Miller-Rabin primality test
def millerRabin(n, k = 7):
    # Other code section
```

This two Python function code will be used together to determine whether the number generated from PRNG were true prime number. By using a set of pre-defined prime number from 2 until 997 and *Miller-Rabin Primality Test*, an algorithm which determines whether a given number is prime by relies on an equality or set of equalities that hold true for prime values, then checks whether or not they hold for a number that we want to test for primality. The importance of this test is to determine whether that random number is true prime number. Because PRNG will generate very large unpredictable number that will be very tedious to be calculated by hand.

```
def KeyGeneration(size=8):
    p=generatePrime(size)
    q=generatePrime(size)
    if not (isPrime(p) and isPrime(q)):
```

```
        raise ValueError('Both numbers must be prime.')
    elif p == q:
        raise ValueError('p and q cannot be equal')
    n = p * q
    phi = (p-1) * (q-1)
    e = random.randrange(1, phi)
    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
        g = gcd(e, phi)
        d = multiplicativeInverse(e, phi)
    return ((n, e), (d, n))
```

The function of this Python code is the most important part in RSA cryptosystem specifically to generate public key for encryption and private key for decryption. First, the code will generate two large random prime number *p* and *q* derived from *generatePrime(keysize)* function at line 48. Then, the number will be tested if it is prime number derived from *isPrime(num)* function at line 54. If both numbers generated were not prime number and p equal to q, it will trigger *ValueError* and raise a message either "Both numbers must be prime" or "p and q cannot be equal".

Second, it will calculate the value of both $n = pq$ and $\phi(n) = (p-1)(q-1)$ respectively. Third, it will select random integer $e, where 1 < e < \phi(n)$ such that $gcd(e, \phi(n)) = 1$ based on what were derived from *gcd(a, b)* function at line 16. Fourth it will use Extended Euclid's algorithm to compute the unique integer $d, where 1 < d < \phi(n)$, such that $ed = 1 mod \phi(n)$ based on the *multiplicativeInverse(a, b)* function at line 29. Finally, it will return public key *(n,e)* and private key *(d,n)*.

To see the number of public key *(n,e)* and private key *(d,n)* generated, copy some section of the code shown below into a new file, name and save it as *gen-rsa-key.py*

```
import sys
import random
from Crypt import Random
def gcd(a, b): …
def multiplicativeInverse(a, b): …
def generatePrime(keysize): …
def millerRabin(n, k = 7): …
def isPrime(num): …
def KeyGeneration(size=8): …
print(KeyGeneration(size=8))
```

Run the code and the number of public key *(n,e)* and private key *(d,n)* generated below are both true large random number.

```
python .\gen-rsa-key.py
((29999, 4547), (3683, 29999))
python .\gen-rsa-key.py
((21509, 19957), (9757, 21509))
python .\gen-rsa-key.py
((47053, 23477), (42293, 47053))
python .\gen-rsa-key.py
((32387, 14275), (22123, 32387))
python .\gen-rsa-key.py
((33389, 5123), (22955, 33389))
python .\gen-rsa-key.py
((33043, 18857), (1993, 33043))
python .\gen-rsa-key.py
((44929, 13461), (27141, 44929))
```

```
def encrypt(pk, plaintext):
    #1) obtain (n,e)
    n, e = pk
    #2)message space [0,n-1]
    #3)compute c=m^e(mod n)
    c = [(ord(char) ** e) % n for char in plaintext]
    print(c)
    #4) send "C" to the other party
return c
```

This is Python function to encrypt the input with public key and return the ciphertext produced to the other party (receiver).

```
def decrypt(pk, ciphertext):
    d, n = pk
    #5)m=c^d (mod n)
    m = [chr((char ** d) % n) for char in ciphertext]
return m
```

This is Python function to decrypt the cipher key received with private key and return the output produced.

```
def encryptAES(cipherAESe,plainText):
    return cipherAESe.encrypt(plainText.encode("utf-8"))
def decryptAES(cipherAESd,cipherText):
    dec= cipherAESd.decrypt(cipherText).decode('utf-8')
return dec
```

This is Python function that takes the previous encrypt and decrypt function using public key and private key respectively to be combine later with AES encryption and decryption. The format will be displayed in UTF-8 encoding type so that it will output nicely.

```
def main():
    print("HYBRID CRYPTOGRAPHIC SCHEME\n")
    pub,pri=KeyGeneration()
```

The first part of the function is to combine AES with RSA. The first part is to obtain RSA public key *(n,e)* and private key *(d,n)* from *KeyGeneration()* function at line 106.

```
key = secrets.token_hex(16)
print("AES Symmetric Key: ")
print(key)
KeyAES=key.encode('utf-8')
```

The second part is to generate 16-bytes AES symmetric key with secrets module. *Secret.token_hex()* will return a random text string in hexadecimal. The string has *n* random bytes, each byte converted to two hexadecimal digits. As usual, the format will be processed in UTF-8 so that it will output nicely.

```
plainText = input("Enter the plaintext: ")
cipherAESe = AES.new(KeyAES,AES.MODE_GCM)
nonce = cipherAESe.nonce
print("\nEncrypting the message with AES......")
cipherText=encryptAES(cipherAESe,plainText)
print("\nAES cipher text: ")
print(cipherText)
```

Next, user will input some plaintext. The *cipherAESe* variable will be used by *encryptAES()* function at line 151 to encrypt the plaintext using AES symmetric key in GCM mode of operations. Then the cipherText will called back by *encryptAES()* function to displayed back the final ciphertext

```
cipherKey=encrypt(pub,key)
print("\nEncrypting the AES symmetric key with RSA......")
print(cipherKey)
```

CipherKey called back the *encrypt()* function at line 133 to encrypt AES symmetric key with RSA public key so that the key can be transfer securely to the other party (receiver). In this case Alice send the cipher key used and ciphertext to Bob.

```
decriptedKey=''.join(decrypt(pri,cipherKey))
print("\nDecrypting the AES Symmetric Key...")
print(decriptedKey)
```

The receiver use Alice private key derived from *decrypt()* function at line 143 to decrypt the cipher key to get back the AES symmetric key used to encrypt the plaintext.

```
decriptedKey=decriptedKey.encode('utf-8')
cipherAESd = AES.new(decriptedKey, AES.MODE_GCM, nonce=nonce)
decrypted=decryptAES(cipherAESd,cipherText)
print("\nDECRYPTED CIPHER TEXT: ")
print(decrypted)
```

The receiver used the AES symmetric key to decrypt back the message contained in the data encapsulation segment.