Lab 2

Symmetric/Public Key Encryption

[Symmetric/Public Key Encryption]

Pycryptodome (https://pycryptodome.readthedocs.io/en/latest/src/introduction.html) is a widely used python package to encrypt/decrypt data. This lab will guide you on how to use Symmetric/Public Key Encryption in Pycryptodome.

First, install *Pycryptdome* in your UbuntuVM.

1. Install python3-pip for easier installation. Type the following commands in a Ubuntu terminal:
```
$ sudo apt-get update
$ sudo apt-get install build-essential python3-dev python3-
pip
$ pip3 install pycryptodome
```

2. Random key generation
   a. Pycryptodome provides a random generation function `get_random_bytes()`. For example, the following code can be used to generate random 16 bytes (128 bits).
```
>>> from Crypto.Random import get_random_bytes
>>> key = get_random_bytes(16)
>>> print(key)
```

3. Symmetric Encryption/Decryption with AES:
   a. AES encryption requires some parameters to be agreed between the encryption party and the decryption party. We let assume CBC (Cipher Block Chaining) mode and 128 bits (16 bytes) block size are used for encryption/decryption.
   b. The following code is to 1) initiate AES object with the CBC mode 2) pad data block into 16 bytes size and 3) encrypt the padded data. Additionally, the outputs 'iv' and 'ct' are encoded using based64 encoder. This encoder writes those outputs in more tractable format.
```
>>> from base64 import b64encode
>>> from Crypto.Cipher import AES
>>> from Crypto.Util.Padding import pad
>>> from Crypto.Random import get_random_bytes

>>> data = b"secret"

>>> key = get_random_bytes(16)
>>> cipher = AES.new(key, AES.MODE_CBC)
>>> ct_bytes = cipher.encrypt(pad(data,
AES.block_size))

>>> iv = b64encode(cipher.iv).decode('utf-8')
>>> ct = b64encode(ct_bytes).decode('utf-8')
>>> key = b64encode(key).decode('utf-8')
>>> print(iv, ct, key)
```

[Note] It should be noted that 'iv', called Initial Vector, is a parameter required for AES CBC mode. It is internally generated since it was not given when we set AES object was initiated. This 'iv' must be shared with the decryption party and can be shared using a public channel.

c. Decryption code can be written in the almost reverse way. Check the following code. Note that 'iv' must be given for decryption.

```
>>> from base64 import b64decode
>>> from Crypto.Cipher import AES
>>> from Crypto.Util.Padding import unpad

>>> try:
>>>    iv = b64decode(Output of Encryption)
>>>    ct = b64decode(Output of Encryption)
>>>    key = b64decode(key of encryption)

>>>    cipher = AES.new(key, AES.MODE_CBC, iv)
>>>    pt = unpad(cipher.decrypt(ct), AES.block_size)
>>>    print("The message was: ", pt)
>>> except ValueError:
>>>    print("Incorrect decryption")
>>> except KeyError:
>>>    print("Incorrect Key")
```

4. Public Key Encryption
    a. Pycryoptodome provides public key encryption like RSA.
    b. First, generate a *public* and *private* key pair for RSA:

```
>>> from Crypto.PublicKey import RSA

>>> key = RSA.generate(2048)
>>> private_key = key.export_key()
>>> file_out = open("private.pem", "wb")
>>> file_out.write(private_key)
>>> file_out.close()

>>> public_key = key.publickey().export_key()
>>> file_out = open("receiver.pem", "wb")
>>> file_out.write(public_key)
>>> file_out.close()
```

[Note] It should be noted that the private key is exported by key.export_key() and the public key was exported by key.publickey().export_key(). They are stored as a PEM file format for the future use.

c. RSA also needs to be specify its version. We use PKCS#1_OAEP for encryption and decryption. The following code 1) reads the public key from "receiver.pem" file, 2) generates 16 bytes random and 3) encrypts it using RSA algorithm

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Random import get_random_bytes
>>> from Crypto.Cipher import PKCS1_OAEP

>>> recipient_key =
RSA.import_key(open("receiver.pem").read())
>>> data = b"secret"
>>> file_out = open("encrypted_data.bin", "wb")

>>> cipher_rsa = PKCS1_OAEP.new(recipient_key)
>>> enc_data = cipher_rsa.encrypt(data)
>>> file_out.write(enc_data)
>>> file_out.close()
```

d. The decryption can be done using similarly, but using the private key (private.pem). The code 1) reads the private key from "private.pem", 2) decrypts the encrypted data.

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Cipher import AES, PKCS1_OAEP

>>> file_in = open("encrypted_data.bin", "rb")
>>> private_key =
RSA.import_key(open("private.pem").read())
>>> enc_data =
file_in.read(private_key.size_in_bytes())
>>> cipher_rsa = PKCS1_OAEP.new(private_key)
>>> data = cipher_rsa.decrypt(enc_data)
>>> print(data)
>>> file_in.close()
```

5. (Optional) Simulate the hybrid encryption explained in the lecture slides using the codes above. You may combine 3.b and 4.c for encryption and 3.d and 4.d for decryption.