Cryptography Homework 5c—Using RSA and AES together.

Public key encryption gives us a way to exchange keys securely, but it is very slow. Symmetric encryption is fast but has no method for secure key exchange. In practice, most systems use public key encryption to securely exchange a session key and then switch to symmetric encryption using that session key.

This exercise follows a variation of the technique that HTTPS in your web browser generally follows. It uses public key encryption (RSA in this case) to securely exchange a key (session key) that will be used in symmetric encryption. It then switches to AES symmetric encryption to transfer the data.

The lab follows the method used in the example, Encrypt data with RSA in the Pycryptodome documentation. <https://pycryptodome.readthedocs.io/en/latest/src/examples.html>

You will use the RSA public/private key pair you created in the last lab and give the public key to your partner. Your partner will create a session key, encrypt the session key with your public key, and give the encrypted session key to you. You should be able to decrypt the session key with your RSA private key. Finally, your partner uses AES and the session key to encrypt a file and send it to you. (Actually, your partner will give you the encrypted key and the AES data bundled into a single file.)

Note: If your course is entirely online, you can send the message to yourself instead of to a partner; your choice.

A picture containing indoor, object

Description automatically generated

# RSA key pair

You each should have your own RSA public/private key pair that you generated in the last lab. The first step will be to get your public key to your partner. Remember that the security of this entire process relies on you and your partner trading your public keys. If an attacker can substitute their key for your partner’s, the attacker can conduct a Man in the Middle (MitM) attack against you. We will talk about certificates as a means of exchanging public keys (more or less) securely in a later lesson. For the time being you can choose the method you want to use to get your public key to your partner. The methods range from easy/boring, to fun/more difficult.

* Sneakernet. Trade public keys using a flash drive. If you are working by yourself, just give yourself the files you need ;-)
* Email your public keys to each other.
* Simple Web Server. Python has a built-in module that contains a very simple web server, but there are a couple limitations. First, you must know how to open a port on your firewall so that your partner can connect. Second, if you use a VM, the VM Network Adapter must be in bridged mode so that it has its own IP address. The server will publish any files in the directory you run the command from. If you can handle all that, the command is simple.  
  Windows Python 3  
  python -m http.server 8000  
  Ubuntu  
  python3 -m http.server 8000  
  where 8000 is the port number to use, your choice. If you omit the port, the module will use 80 which is a little easier for your partner.
* Netcat. Netcat has the same firewall and VM Network Adapter problems as the Python http.server. We used netcat in Networking Lab 1, Physical and Datalink. You can find instructions there.

It may be wise to rename your partner’s public key (Bob\_public.pem, or something) so you do not get it confused with your own.

Do not lose sight of the fact that this is a lab on encryption. If netcat/ncat or Python http.server isn’t working for you, fall back to flash drives or email.

# Use Functions

Good coding practice uses functions to separate code into modules, so we will do that. We will have two functions, one for encryption and one for decryption.

We have used all the modules you will need in previous labs—encryption with RSA was covered in Lab 5b, and encryption with RSA was covered in Lab 3c. You should be able to copy code from your earlier labs for most of this lab.

## Main program

A normal main will offer an interface that allows the user to select encode or decode, enter a path to the public or private key to be used, and enter a message or path to the message (plaintext for encryption, ciphertext for decryption. You can do that interface coding if you wish, and have time, but for this lab you may hard code this information in the main program. A template that includes the main portion, but removes the code from the encrypt and decrypt functions is available in RSA-AES-encrypt-decrypt-stuff-missing.py

The main program will read the message and the key, and then call the encryption or decryption function as appropriate. For encryption, the key should be the receiver's public key in pem format (we will let the encryption function import it into an RSA object.) For decryption, the key should be the receiver's private key in pem format.

When we call the encrypt function with the plaintext and public RSA key, it will return a JSON string containing the AES key (encrypted with RSA), and the nonce, tag, and ciphertext from the AES encryption.

When we call the decrypt function with the JSON string and the private RSA key, it will return the decrypted message plaintext.

The main program will also import all the modules we will need.  
from Crypto.Random import get\_random\_bytes

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

from Crypto.Cipher import AES

import json, base64

## Encryption Function

The encryption function will receive the message (plaintext) and public key as arguments.  
def encryption\_whatever(plaintext, pub\_key\_pem):

It needs to accomplish the following tasks:

1. Create an AES session key, 16 bytes long, using get\_random\_bytes(16)
2. Use the public key in pem format to create an RSA key object with  
   RSA.import\_key(public\_key\_pem)
3. Create an RSA object with PKCS1\_OAEP.new(your RSA key object)
4. Encrypt the AES session key with  
   cipher\_rsa.encrypt(your AES session key)  
   You will use the unencrypted AES session key to encrypt your message, but send the encrypted key with the ciphertext so the recipient can decrypt the key using RSA.
5. Create an AES object using  
   AES.new(AES\_session\_key, AES.MODE\_EAX)
6. Encrypt the message with AES   
   ciphertext, tag = cipher\_aes.encrypt\_and\_digest(plaintext)
7. Package the ciphertext, nonce, tag, and encrypted AES session key into a Python3 dictionary.  
    enc\_msg\_dict = {  
    "encrypted\_session\_key" : base64.b64encode(enc\_session\_key).decode(),  
    "nonce" : base64.b64encode(cipher\_aes.nonce).decode(),  
    "tag" : base64.b64encode(tag).decode(),  
    "ciphertext": base64.b64encode(ciphertext).decode()  
    }  
   We did this in Lab 3c. This is an example—your variable names could be different.
8. Convert the dictionary to JSON string using json.dumps()
9. Return the JSON string.

## Decryption Function

The decryption function will receive the JSON string (AES key encrypted with RSA, nonce, tag, and ciphertext) and private RSA key in pem format as arguments.  
def decryption\_whatever(JSONstring, priv\_key\_pem):

It needs to accomplish the following tasks:

1. Use RSA.import\_key(rcvr\_private\_pem) to create an RSA key object
2. Convert the JSON string into a Python3 dictionary.  
   message = json.loads(enc\_msg\_json)  
   If you want to see the format the dictionary uses, add  
   print(message)
3. Extract the encrypted AES key, nonce, tag, and ciphertext from the dictionary. For example,  
   tag = base64.b64decode(message["tag"].encode()). Remember,  
   message["tag"] grabs the value (data) associated with the tag key in the dictionary  
   message["tag"].encode()changes the string to bytes  
   base64.b64decode(message["tag"].encode()) changes from base64 to binary  
     
   Don't forget to do the above line for the encrypted AES key (encrypted\_session\_key), nonce, and ciphertext as well
4. Decrypt the AES session key using a new object PKCS1\_OAEP.new(RSA private key object) and cipher\_rsa.decrypt(encrypted AES session key)
5. Decrypt the ciphertext with AES as you did in Lab 3
6. Return the decrypted message

# Put it together.

Give your RSA public key to your partner, along with an encrypted message. Get your partner's RSA public key and a message they have encrypted. Decrypt your partner's message.

# Note:

If your file does not decrypt properly check this.

1. Do the files from your partner actually have content? Use dir (Windows) or ls -l (Linux) to make sure the file length is not zero.
2. Are the files the same on both ends? You can check by taking a hash at both ends.  
   md5sum filename (for Linux)  
   Get-FileHash .\filename -Algorithm MD5 (for Windows PowerShell)

If the hashes are different, transfer the file again. Sometimes nc and ncat have trouble with binary data. Fix this by encoding the file with base64 before sending.

# Hand in, Part 1

Hand in screenshots of your encryption and decryption.

# Hand in, Part 2

Your foolish instructor has posted their private key, instructor\_priv.pem on Canvas, along with an encrypted file. The file (also on Canvas), poem.json was created using the procedure we just followed (Bob in the overview.) Use the private key to decrypt poem.json. What is the title, and who was the author of the poem?

You can use the same decryption script you used before, except that you will need to read the public key from instructor\_priv.pem and read poem.json.

Important: the file poem.json uses this structure:  
A black text on a white background

Description automatically generated

Check your decryption function to ensure that the key names (encrypted\_session\_key, nonce, tag, and ciphertext) are exactly the same as shown above. If your code uses AESkey where poem.json uses encrypted\_session\_key, you will get an error.

# Optional Reading

<SIDEBAR> We import some things from Crypto.PublicKey, some from Crypto.Random, and some from Crypto.Cipher. You may ask where those different names came from. We got them from the example, but is there any rhyme or reason to it? The answer can be found in two places. One is the documentation for PyCryptodome. <https://www.pycryptodome.org/src/api>

A screenshot of a computer

Description automatically generated

Another place is in the installation directories. The installation is well structured, so the directories correspond to the different packages.  
A screen shot of a computer code

Description automatically generated

The \_\_init\_\_.py file describes the modules.  
A screen shot of a computer

Description automatically generated

The windows installation really buries these files.  
A screenshot of a computer program

Description automatically generated

</SIDEBAR>