# Cryptography Homework 5b—RSA and AES Together

## Required reading

Crypto 5 slides

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.

## Overview

This is the procedure we will follow.  
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 securely (more or less) 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
* 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 don’t get it confused with your own.

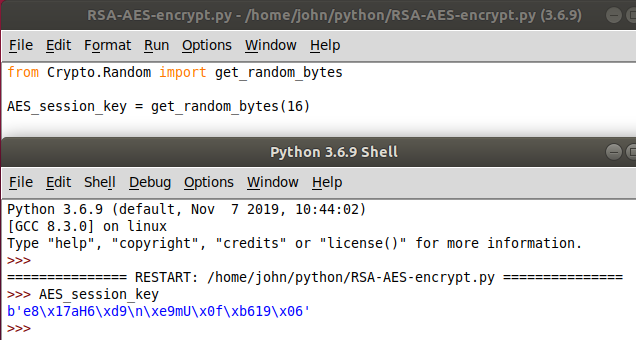
Don’t lose sight of the fact that this is a lab on encryption. If netcat/ncat or Python http.server isn’t working for you, fall back to flash drives or email.

## Encrypt a message

We are using the section in the PyCryptodome example, “Encrypt data with RSA,” with slight modifications. You will create an AES session key and encrypt it with your partner’s RSA public key. Then you will encrypt your message with your AES session key. Finally, you will put your **encrypted** session key, the encrypted AES message, the tag, and the nonce into a file for your partner.

### Create and Encrypt the Session Key

First, create an AES session key. In Lab 3 on AES, you had the option of using text you could easily type. Here, the session key will be encrypted and sent with the file so we might as well use a 16-byte random key. Start a new file and create a key.



Now we need to import the other person’s (I’ll say Bob’s) key and use it to encrypt the AES session key. Note that we will be using a module called “PKCS1\_OAEP”. If you remember from the slides in Crypto 5, Public Key Encryption—RSA Math, RSA encryption has many vulnerabilities if it is not protected by a padding and hashing algorithm like PKCS1-OAEP.

from Crypto.Random import get\_random\_bytes

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1\_OAEP

AES\_session\_key = get\_random\_bytes(16)

#read and import Bob's RSA public key

with open('/home/john/python/Bob\_public.pem') as fh:

Bob\_pub = fh.read()

Bob\_pub\_key = RSA.import\_key(Bob\_pub)

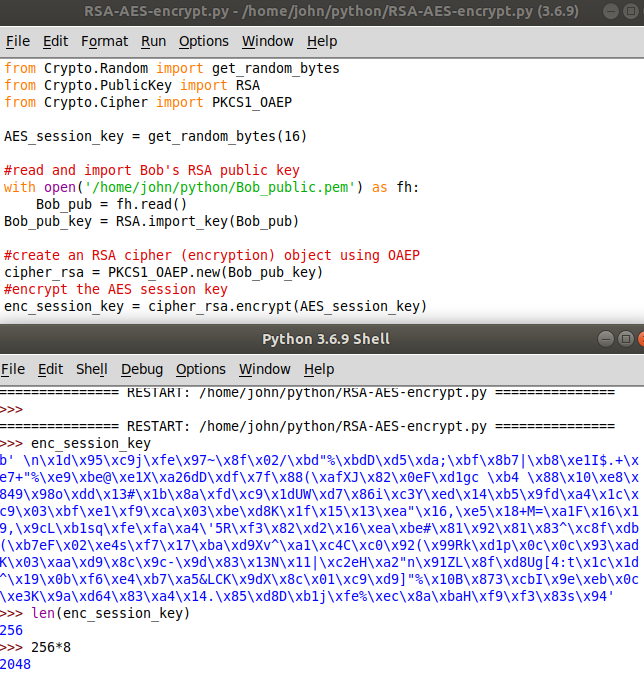
#create an RSA cipher (encryption) object using OAEP

cipher\_rsa = PKCS1\_OAEP.new(Bob\_pub\_key)

#encrypt the AES session key

enc\_session\_key = cipher\_rsa.encrypt(AES\_session\_key)

Although the AES session key is only 16 bytes long, you can see (enter enc\_session\_key in the Idle shell) that the encrypted session key is much longer. This is because PKCS1 OAEP pads the message so that it fills an entire block. In RSA encryption the block size is the same as the key length, which is 256 bytes or 2048 bits in our case.



### Encrypt the Message with AES

The next step is to encrypt our message. Create an AES object with the AES session key we made earlier, using EAX mode. Then encrypt the message with the AES and the session key (not the encrypted one) we created.

message = b'I don\'t suppose you\'ve seen anything weird around here? 13th Dr Who'

from Crypto.Cipher import AES

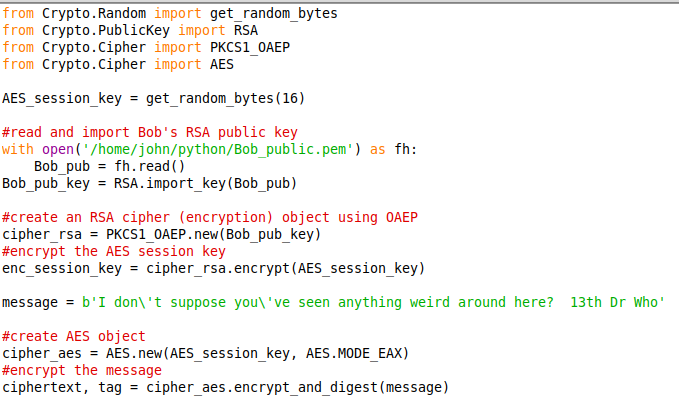
#create AES object

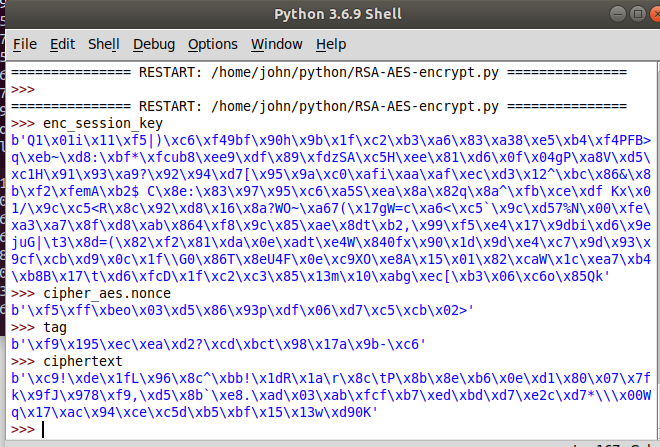
cipher\_aes = AES.new(AES\_session\_key, AES.MODE\_EAX)

#encrypt the message

ciphertext, tag = cipher\_aes.encrypt\_and\_digest(message)

We didn’t ask for output, but we can see the values of the variables by entering them in the Idle shell.





When we unpack the file, we will need to know how long the components are. We just checked the length of the encrypted\_session\_key (256 bytes) and we know the tag and nonce are 16 bytes long.

### Save the Message to a File

Finally, we save the encrypted session key, and the AES info (ciphertext, tag, and nonce) in a file and transmit the file to our partner. Be sure to save the encrypted session key to the file and not the raw AES session key! If you send the raw key, your security is shot. Note the “wb” where we opened the file; it means write, binary.

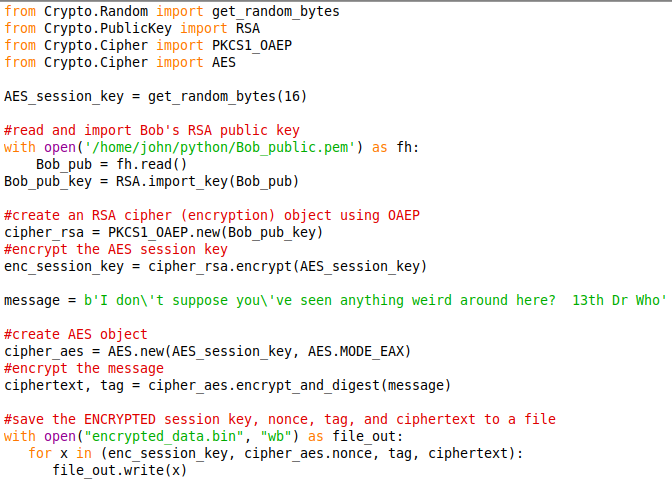
#save the ENCRYPTED session key, nonce, tag, and ciphertext to a file

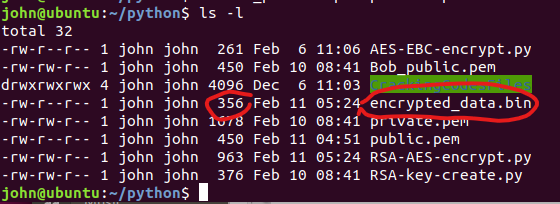
with open("encrypted\_data.bin", "wb") as file\_out:

for x in (enc\_session\_key, cipher\_aes.nonce, tag, ciphertext):

file\_out.write(x)

Again, there will not be output in the shell (no errors, hopefully) but there should be a file created. Make sure it has data in it.





### Transmit the encrypted message to your partner

You can use any of the methods you used to get your partner’s public key: sneakernet, email, Python HTTP, or netcat. Note: Any time you re-run the encryption script (fix errors, whatever) the session key, encrypted session key, nonce, tag, and ciphertext will all change, even though the plaintext is the same. That’s good, it’s the way encryption is supposed to be. You are creating the session key with a random number generator, RSA PKS1-OAEP is adding randomness to the encrypted session key, and AES is adding randomness with the nonce.

## Decrypt the Message

### Extract the Data from Your Partner’s File

First, the person receiving encrypted\_data.bin must extract the encrypted session key, the AES nonce, AES tag, and AES ciphertext from the file. They must also read their own private key file into memory.

from Crypto.PublicKey import RSA

#import the private key

with open('/home/john/python/private.pem') as fh:

private\_key = RSA.import\_key(fh.read())

#import the encrypted message

with open('/home/john/python/encrypted\_data.bin', 'rb') as fh:

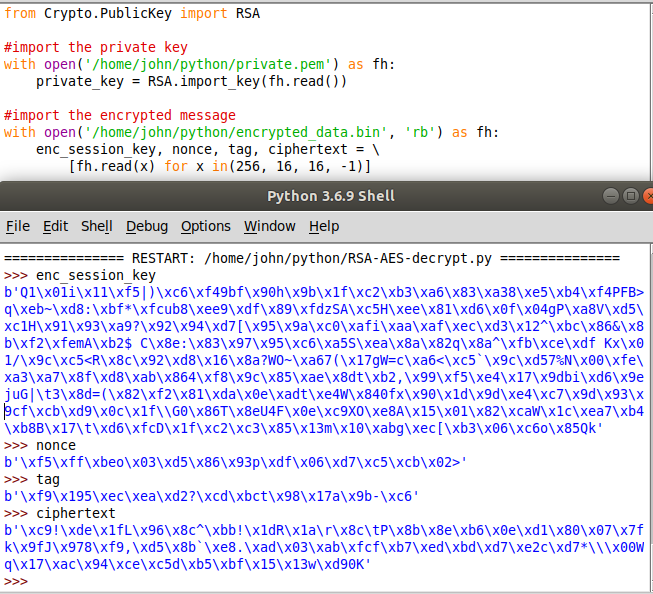
enc\_session\_key, nonce, tag, ciphertext = \

[fh.read(x) for x in(256, 16, 16, -1)]

Reading the private key into memory is straight-forward. The file contains just one item, the key, and it is base64 text so it can be opened with the default arguments. We use the RSA.import\_key() method to convert the key into the binary form that the RSA modules need.

Since the encrypted session key, nonce, tag, and ciphertext are all packed together in one file, reading them is more complicated. The encrypted\_data.bin file is binary data, so we need to open it with ‘rb’. We know that the encrypted session key is 256 bytes long (one block, the same size as the RSA key) because we looked at that when we created the encrypted session key. The for loop reads the first 256 bytes for enc\_session\_key, the next 16 bytes for nonce, the next 16 bytes for tag, and whatever is left goes into ciphertext.

For troubleshooting, we can check the values we have against the values our partner saw when they were encrypting the message.



Once you have extracted enc\_session\_key, nonce, tag, and ciphertext from encrypted.bin, you have two steps remaining. You need to use RSA and the private key to decrypt enc\_session\_key to get the AES\_session\_key, and then use AES\_session\_key, nonce, tag, and ciphertext in AES to decrypt the message.

### Decrypt the AES session key

The AES session key was encrypted with RSA’s PKCS1\_OAEP, so that’s how we should decrypt it.

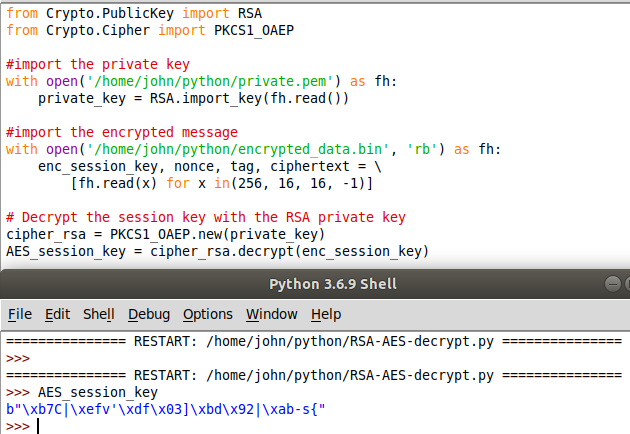
from Crypto.Cipher import PKCS1\_OAEP

# Decrypt the session key with the RSA private key

cipher\_rsa = PKCS1\_OAEP.new(private\_key)

AES\_session\_key = cipher\_rsa.decrypt(enc\_session\_key)

We didn’t ask for output, but you can check the value of AES\_session\_key at the Idle Shell if you like.



### Decrypt the message

The last step is to decrypt the message.

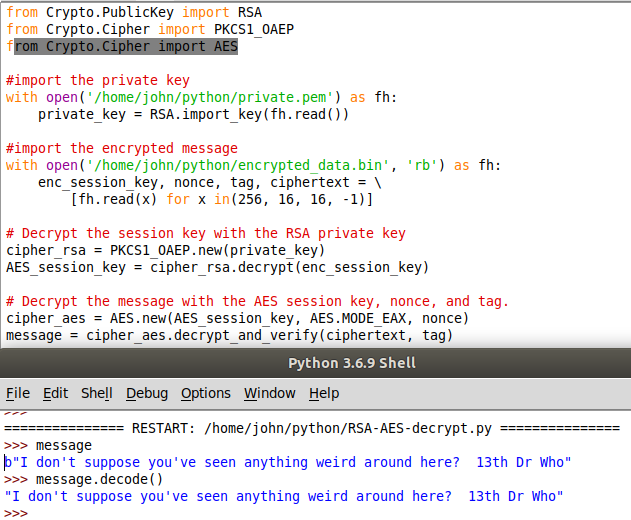
from Crypto.Cipher import AES

# Decrypt the message with the AES session key, nonce, and tag.

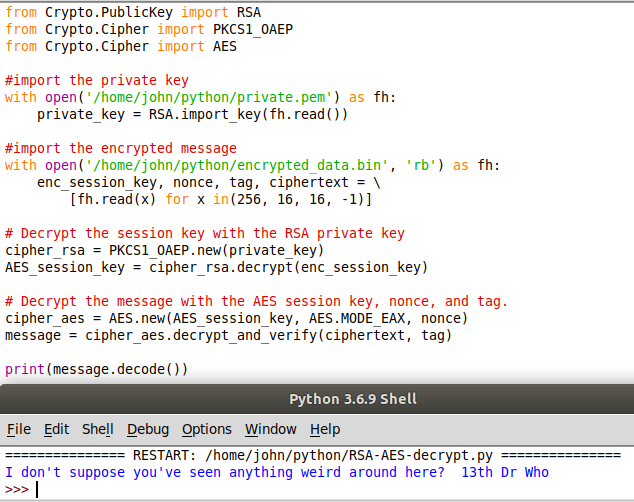
cipher\_aes = AES.new(AES\_session\_key, AES.MODE\_EAX, nonce)

data = cipher\_aes.decrypt\_and\_verify(ciphertext, tag)

Note that the message is output in bytes. It doesn’t look different from a string, other than the ‘b’ at the beginning. If you want a string output we can get that with message.decode(). The decode() method changes things from bytes to a UTF-8 string by default. To change from a string to bytes, use the encode() method.



Now that we are finished, we should print the message.  
print(message.decode())



# Note:

If your file does not decrypt properly check this.

1. Does the file encrypted\_data.bin 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 encrypted\_data.bin (for Linux)  
   Get-FileHash .\encrypted\_data.bin -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.

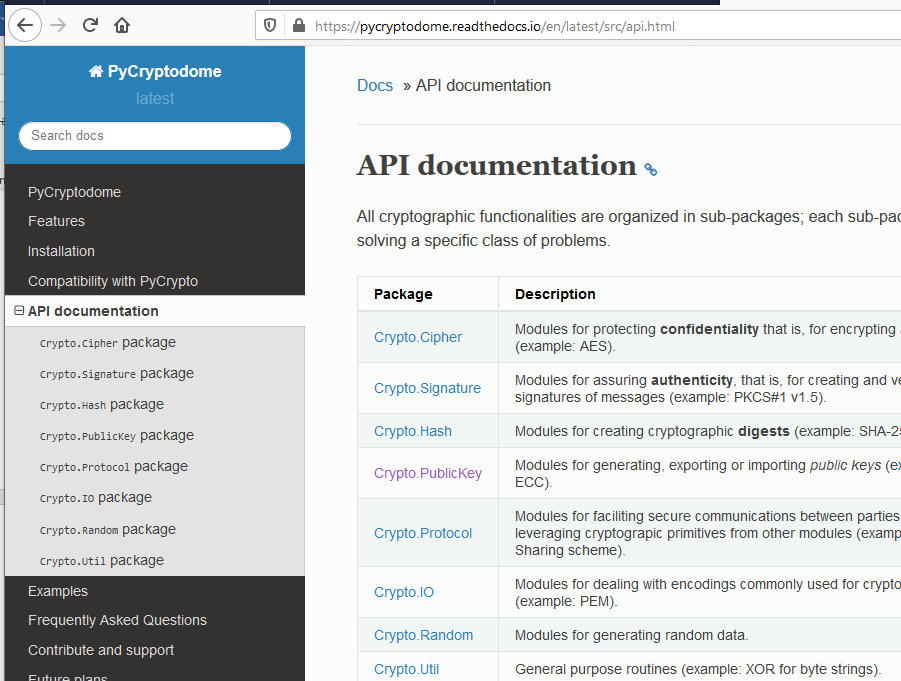
# Hand in, Part 2

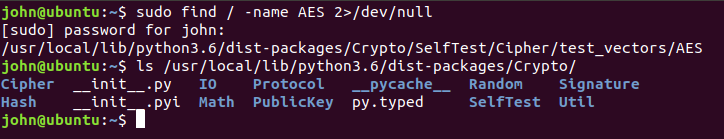
Your foolish instructor has posted their private key, instructor\_priv.pem on Canvas, along with an encrypted file. The file (also on Canvas), poem.bin was created using the procedure we just followed (Bob in the overview.) Use the private key to decrypt poem.bin. What is the title, and who was the author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

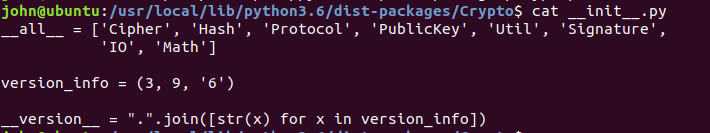
You can use the same decryption script you used before, except that you will need to read the key from instructor\_priv.pem and read poem.bin instead of encrypted\_data.bin.

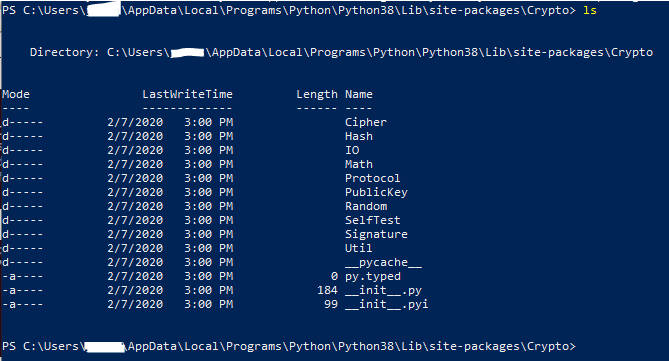
# Optional Reading

<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.



Another place is in the installation directories. The installation is well structured, so the directories correspond to the different packages.  


The \_\_init\_\_.py file describes the modules.  


The windows installation really buries these files.  


</SIDEBAR>