Cryptography Homework 5a—Create an RSA Private/Public Key Pair in Python

In this lab you will create an RSA public/private key pair using the PyCryptodome library in Python. You will also use the Python dir command to find an object’s methods and properties.

# Required reading

Crypto 5 slides

# Generate an RSA key

We will use the example in <https://pycryptodome.readthedocs.io/en/latest/src/examples.html>, in the section “Generate public key and private key.” This will create a private key, stored in the file private.pem (or whatever you choose to name it), and a public key stored in public.pem.

Note that the previous example on the site, “Generate an RSA key,” generates the private key and protects the private key with a password. This is a common practice; to use the private key or see its contents you must use the password. We will assume you can keep your private key secure and will not put a password on our private key.

Use the following code to create an RSA key. It assumes that your installation of PyCryptodome in the earlier lab was successful.  
from Crypto.PublicKey import RSA  
key = RSA.generate(2048)  
  
Graphical user interface, text, email

Description automatically generated

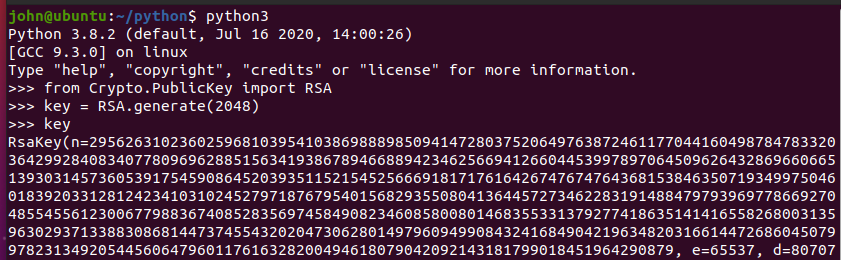
The code ran and created a key but we have not told it to print anything so there is no output. If we simply added print(key) to our code, it would just tell us we have a key object. There will be more on that later.  


## Examine the RSA key from an Interactive Prompt

Use the Linux or Windows terminal to examine the contents of the key variable. Note that this does not work from inside a script; you must be at an interactive prompt. We will discuss doing this within a script in a bit.

Windows Python interactive prompt  
Text

Description automatically generated  
<snip>

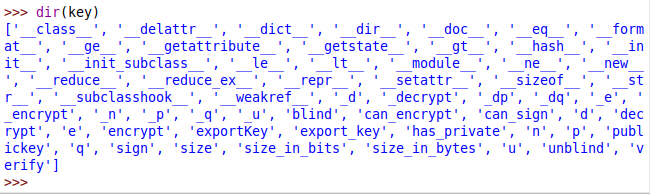
Linux Python interactive prompt  
  
<snip>

You should see:  
RsaKey(n= a really long number,  
 e=65537,  
 d= a really long number,  
 p= a really long number,  
 q= a really long number,  
 u= a really long number)

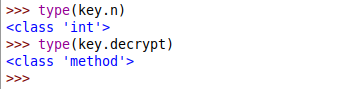
Those numbers (except for u) should be familiar to you, since you have studied the slides in the Cryptology 4 Module, Intro to Public Key Encryption. All of these are part of the private key, but only some should be included in the public key. See question 1 in the Hand In section.

* n is the modulus and should be about 2048 bits long (about 616 digits in base 10). It is created by multiplying the random primes p and q.
* e is the encryption exponent that will be used for the public key. According to NIST the public key should not be any smaller than 65537, so that is the number PyCryptodome chooses by default.
* d is the decryption exponent that will be used for the private key, and it should be long.
* p and q are the prime numbers that were chosen to create n. They each should be about 1024 bits long.
* u is the inverse of p mod q. u = p-1 mod q. It is used to make the RSA calculations faster. You may see the Chinese Remainder Theorem (CRT) associated with it, but that is beyond the scope of this course.

# Some Python

Execute the command dir(key).  


In Python, the dir(object) command shows all the properties and methods associated with the object. The terms that begin and end with double underscores (‘\_\_class\_\_’ for example) are normally for system use or for advanced users and are called “dunder” variables.

There are several terms in dir() that will be useful to us, like n, p, q, decrypt, encrypt, export\_key, size\_in\_bits, and more. To see what those terms are, use the type() command. For example, n is an integer property and decrypt is a method.  


# Public and Private Keys

Once you create a key pair, you often need to use the private and public keys repeatedly. For that reason, you usually save the keys to files. The process of creating public and private keys from the key object is called exporting.

There are several formats for saving keys. The format that PyCryptodome uses by default is called pem, which is described here in this link:  
<https://serverfault.com/questions/9708/what-is-a-pem-file-and-how-does-it-differ-from-other-openssl-generated-key-file> .  
A PEM file contains binary data, but it is base64 encoded so that it may be sent through a text-only medium.

Execute the following code. It uses the “with open” method for saving the files so we will not need to worry about forgetting to close the files. We have not asked for output to the display yet, so there should not be any output to the console, although you can add a print statement to display a happy “mission accomplished” message if you wish. The code will create the files private.pem and public.pem.

Graphical user interface, text, application

Description automatically generated

Note that when we tried to print the key inside the script it just told us we have an RSA key object and gave us the memory address where it was stored. Not that useful.

You can open the private.pem and public.pem files in notepad from Windows or use  
cat private.pem  
in Linux to view the files. You should see that they are base64 encoded with text headers and footers (----END PUBLIC KEY----, etc.) If you remove the headers and footers, you can decode the base64, but you will end up with binary data.  
Text

Description automatically generated  
<snip>

## Print the RSA key details from a script.

If you try to print the key from inside a script, it will not give the same results as it did from the interactive prompt, above. It will just print the memory address of the key object.

The best way to print the parts of the key is to use the properties shown in dir(key), above, to print the properties of the key manually.

Text

Description automatically generated

print('N = {0},\n p = {1},\n q = {2},\n e = {3},\n d = {4}'.format(key.n, key.p, key.q, key.e, key.d))

This method shows string formatting that we have not used before. It was added in Python 3 and is quite handy. The string  
'N = {0},\n p = {1},\n q = {2},\n e = {3},\n d = {4}'  
has a method called format(). It will replace {0} with the first argument in format(), {1} with the second, {2} with the third, and so on (remember that Python starts counting at 0.) In our case, those arguments are key.n, key.p, key.q, and so on. The \n just adds the new line character so that the variables each start on their own line.

## Hand In

Create a key, then export and save both the public and private keys to files. To answer these questions, you will need the GCD and inverse functions from PyCryptodome for question 3.  
from Crypto.Util.number import GCD, inverse

1. The private\_key and public\_key variables both contain base64 encoded text. The private key is much longer than the public key. The private key is longer because it contains values that should not be in the public key. What does the private key need to have that the public key does not (and should not) have?
2. Verify your answer to question 1. You can see what your public key contains by reading in at an **interactive prompt** by pasting this code. It reads your public key file, converts it to a Python object (import\_key), and displays it.  
   from Crypto.PublicKey import RSA  
   fh = open('your public key filename ', 'rb')  
   pub = fh.read()  
   pubkey = RSA.import\_key(pub)  
   pubkey  
   What does your public key contain? Why does your private key contain more?
3. Check to see if the RSA key generation agrees with what was covered in the Crypto 4 module. Extract p, q, and d from the private key (private\_key.p, private\_key.q, etc.) You can do this by running printKey.py and then pasting p, q, etc., into an interactive Python prompt for the calculations, or you can write a Python script to do it.
   1. Compute n = p \* q. Is the result the same as private\_key.n ?
   2. Instead of using Φ = (p-1) \* (q-1), Pycryptodome uses a slightly more efficient equation, L = LCM( (p-1) , (q-1) ), where LCM stands for lowest common multiple. You can find LCM using the GCD function in this formula.  
      LCM(a, b) = a \* b // GCD(a, b)  
      Putting the two equations together gives you this, which you can compute in Python:  
        
      L = (p-1) \* (q-1) // GCD( (p-1) , (q-1) )  
        
      Compute L using the line above.
   3. Compute d = e-1 mod L. Is the result the same as private\_key.d ?   
      Note: In python, compute  
      d = inverse(e, L)

Note: The lecture notes use lambda = lcm( (p-1) \* (q-1) ). In Python lambda is a reserved word, so I’ve changed it to L.

Note: The Pycryptodome module computes the inverse of e using lcm( (p-1) \* (q-1) ), and not (p-1)\*(q-1) because it is more efficient.