PyCrypto

The PyCrypto package is probably the most well known 3rd party cryptography package for Python. Sadly PyCrypto's development stopping in 2012. Others have continued to release the latest version of PyCryto so you can still get it for Python 3.5 if you don't mind using a 3rd party's binary. For example, I found some binary Python 3.5 wheels for PyCrypto on Github (https://github.com/sfbahr/PyCrypto-Wheels).

Fortunately there is a fork of the project called PyCrytodome that is a drop-in replacement for PyCrypto. To install it for Linux, you can use the following pip command:

```
pip install pycryptodome
```

Windows is a bit different:

```
pip install pycryptodomex
```

If you run into issues, it's probably because you don't have the right dependencies installed or you need a compiler for Windows. Check out the PyCryptodome website for additional installation help or to contact support.

Also worth noting is that PyCryptodome has many enhancements over the last version of PyCrypto. It is well worth your time to visit their home page and see what new features exist.

Encrypting a String

Once you're done checking their website out, we can move on to some examples. For our first trick, we'll use DES to encrypt a string:

```
from Crypto.Cipher import DES
```

```
key = b'abcdefgh'
def pad(text):
        while len(text) % 8 != 0:

            text += b' '
        return text
des = DES.new(key, DES.MODE_ECB)
text = b'Python rocks!'
padded_text = pad(text)
encrypted_text = des.encrypt(text)
#Traceback (most recent call last):
#  File "<pyshell#35>", line 1, in <module>
#    encrypted_text = des.encrypt(text)
#  File "C:\Programs\Python\Python35-32\lib\site-packages\Crypto\Cipher\blockalgo.py", line 2
#    return self._cipher.encrypt(plaintext)
#ValueError: Input strings must be a multiple of 8 in length
```

```
from Crypto.Cipher import DES
key = b'abcdefgh'
def pad(text):
        while len(text) % 8 != 0:
            text += b' '
        return text
des = DES.new(key, DES.MODE_ECB)
text = b'Python rocks!'
padded_text = pad(text)

encrypted_text = des.encrypt(padded_text)
print (encrypted_text)
#b'>\xfc\x1f\x16x\x87\xb2\x93\x0e\xfcH\x02\xd59VQ'
```

This code is a little confusing, so let's spend some time breaking it down. First off, it should be noted that the key size for DES encryption is 8 bytes, which is why we set our key variable to a size letter string. The string that we will be encrypting must be a multiple of 8 in length, so we create a function called **pad** that can pad any string out with spaces until it's a multiple of 8. Next we create an instance of DES and some text that we want to encrypt. We also create a padded version of the text. Just for fun, we attempt to encrypt the original unpadded variant of the string which raises a **ValueError**. Here we learn that we need that padded string after all, so we pass that one in instead. As you can see, we now have an encrypted string!

Of course the example wouldn't be complete if we didn't know how to decrypt our string:

```
print (des.decrypt(encrypted_text))
#b'Python rocks!    '
```

Fortunately, that is very easy to accomplish as all we need to do is call the **decrypt** method on our des object to get our decrypted byte string back. Our next task is to learn how to encrypt and decrypt a file with PyCrypto using RSA. But first we need to create some RSA keys!

Creating an RSA key

If you want to encrypt your data with RSA, then you'll need to either have access to a public / private RSA key pair or you will need to generate your own. For this example, we will just generate our own. Since it's fairly easy to do, we will do it in Python's interpreter:

```
from Crypto.PublicKey import RSA
code = 'nooneknows'
key = RSA.generate(2048)
encrypted_key = key.exportKey(passphrase=code, pkcs=8,
        protection="scryptAndAES128-CBC")
with open('my_private_rsa_key.bin', 'wb') as f:
        f.write(encrypted_key)
with open('my_rsa_public.pem', 'wb') as f:
        f.write(key.publickey().exportKey())
```

First we import **RSA** from **Crypto.PublicKey**. Then we create a silly passcode. Next we generate an RSA key of 2048 bits. Now we get to the good stuff. To generate a private key, we need to call our RSA key instance's **exportKey** method and give it our passcode, which PKCS standard to use and which encryption scheme to use to protect our private key. Then we write the file out to disk.

Next we create our public key via our RSA key instance's **publickey** method. We used a shortcut in this piece of code by just chaining the call to **exportKey** with the publickey method call to write it to disk as well.

Encrypting a File

Now that we have both a private and a public key, we can encrypt some data and write it to a file. Here's a pretty standard example:

```
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes
from Crypto.Cipher import AES, PKCS1_OAEP

with open('encrypted_data.bin', 'wb') as out_file:
    recipient_key = RSA.import_key(
        open('my_rsa_public.pem').read())
    session_key = get_random_bytes(16)

    cipher_rsa = PKCS1_OAEP.new(recipient_key)
    out_file.write(cipher_rsa.encrypt(session_key))

    cipher_aes = AES.new(session_key, AES.MODE_EAX)
    data = b'blah blah blah Python blah blah'
    ciphertext, tag = cipher_aes.encrypt_and_digest(data)

    out_file.write(cipher_aes.nonce)
    out_file.write(tag)
    out_file.write(ciphertext)
```

The first three lines cover our imports from PyCryptodome. Next we open up a file to write to. Then we import our public key into a variable and create a 16-byte session key. For this example we are going to be using a hybrid encryption method, so we use PKCS#1 OAEP, which is Optimal asymmetric encryption padding. This allows us to write a data of an arbitrary length to the file. Then we create our AES cipher, create some data and encrypt the data. This will return the encrypted text and the MAC. Finally we write out the nonce, MAC (or tag) and the encrypted text.

As an aside, a nonce is an arbitrary number that is only used for crytographic communication. They are usually random or pseudorandom numbers. For AES, it must be at least 16 bytes in length. Feel free to try opening the encrypted file in your favorite text editor. You should just see gibberish.

Now let's learn how to decrypt our data:

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

code = 'nooneknows'

with open('encrypted_data.bin', 'rb') as fobj:
    private_key = RSA.import_key(
        open('my_rsa_key.pem').read(),
        passphrase=code)

    enc_session_key, nonce, tag, ciphertext = [ fobj.read(x)
```

```
                                                 for x in (private_key.size_in_bytes(),
                                                 16, 16, -1) ]


    cipher_rsa = PKCS1_OAEP.new(private_key)
    session_key = cipher_rsa.decrypt(enc_session_key)

    cipher_aes = AES.new(session_key, AES.MODE_EAX, nonce)
    data = cipher_aes.decrypt_and_verify(ciphertext, tag)

print(data)
```

If you followed the previous example, this code should be pretty easy to parse. In this case, we are opening our encrypted file for reading in binary mode. Then we import our private key. Note that when you import the private key, you must give it your passcode. Otherwise you will get an error. Next we read in our file. You will note that we read in the private key first, then the next 16 bytes for the nonce, which is followed by the next 16 bytes which is the tag and finally the rest of the file, which is our data.

Then we need to decrypt our session key, recreate our AES key and decrypt the data.

You can use PyCryptodome to do much, much more. However we need to move on and see what else we can use for our cryptographic needs in Python.