AES encryption of files in Python with PyCrypto (https://eli.thegreenplace.net/2010/06/25/aes-encryption-of-files-in-python-with-pycrypto)

June 25, 2010 at 18:26 **Tags** Python (https://eli.thegreenplace.net/tag/python)

[**Update 15.11.2013**: passing IV is required in the new PyCrypto]

[**Update 03.08.2019**: port of this post's code to Python 3 (https://github.com/eliben/code-for-blog/tree/master/2010/aes-encrypt-pycrypto)]

The PyCrypto (http://www.pycrypto.org/) module seems to provide all one needs for employing strong cryptography in a program. It wraps a highly optimized C implementation of many popular encryption algorithms with a Python interface. PyCrypto can be built from source on Linux, and Windows binaries for various versions of Python 2.x were kindly made available by Michael Foord on this page (http://www.voidspace.org.uk/python/modules.shtml).

My only gripe with PyCrypto is its documentation. The auto-generated API doc (http://www.dlitz.net/software/pycrypto/apidoc/) is next to useless, and this overview (http://www.dlitz.net/software/pycrypto/doc/) is somewhat dated and didn't address the questions I had about the module. It isn't surprising that a few modules were created just to provide simpler and better documented wrappers around PyCrypto.

In this article I want to present how to use PyCrypto for simple symmetric encryption and decryption of files using the AES algorithm.

Simple AES encryption

Here's how one can encrypt a string with AES:

```
from Crypto.Cipher import AES

key = '0123456789abcdef'
IV = 16 * '\x00'           # Initialization vector: discussed later
mode = AES.MODE_CBC
encryptor = AES.new(key, mode, IV=IV)

text = 'j' * 64 + 'i' * 128
ciphertext = encryptor.encrypt(text)
```

Since the PyCrypto block-level encryption API is very low-level, it expects your key to be either 16, 24 or 32 bytes long (for AES-128, AES-196 and AES-256, respectively). The longer the key, the stronger the encryption.

Having keys of exact length isn't very convenient, as you sometimes want to use some mnemonic password for the key. In this case I recommend picking a password and then using the SHA-256 digest algorithm from `hashlib` to generate a 32-byte key from it. Just replace the assignment to `key` in the code above with:

```
import hashlib

password = 'kitty'
key = hashlib.sha256(password).digest()
```

Keep in mind that this 32-byte key only has as much entropy as your original password. So be wary of brute-force password guessing, and pick a relatively strong password (*kitty* probably won't do). What's useful about this technique is that you don't have to worry about manually padding your password - SHA-256 will scramble a 32-byte block out of any password for you.

The next thing the code does is set the block mode (http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation) of AES. I won't get into all the details, but unless you have some special requirements, CBC should be good enough for you.

We create a new AES encryptor object with `Crypto.Cipher.AES.new`, and give it the encryption key and the mode. Next comes the encryption itself. Again, since the API is low-level, the `encrypt` method expects your input to consist of an integral number of 16-byte blocks (16 is the size of the basic AES block).

The `encryptor` object has an internal state when used in the CBC mode, so if you try to encrypt the same text with the same encryptor once again - you will get different results. So be careful to create a fresh AES encryptor object for any encryption/decryption job.

Decryption

To decrypt the ciphertext, simply add:

```
decryptor = AES.new(key, mode, IV=IV)
plain = decryptor.decrypt(ciphertext)
```

And you get your plaintext back again.

A word about the initialization vector

The initialization vector (http://en.wikipedia.org/wiki/Initialization_vector) (IV) is an important part of block encryption algorithms that work in chained modes like CBC. For the simple example above I've ignored the IV (just using a buffer of zeros), but for a more serious application this is a grave mistake. I don't want to get too deep into cryptographic theory here, but it suffices to say that the IV is as important as the salt in hashed passwords, and the lack of correct IV usage led to the cracking of the WEP encryption (http://en.wikipedia.org/wiki/Wired_Equivalent_Privacy) for wireless LAN.

`PyCrypto` allows one to pass an IV into the `AES.new` creator function. For maximal security, the IV should be randomly generated for every new encryption and can be stored together with the ciphertext. Knowledge of the IV won't help the attacker crack your encryption. What can help him, however, is your reusing the same IV with the same encryption key for multiple encryptions.

Encrypting and decrypting files

The following function encrypts a file of any size. It makes sure to pad the file to a multiple of the AES block length , and also handles the random generation of IV.

```
import os, random, struct
from Crypto.Cipher import AES

def encrypt_file(key, in_filename, out_filename=None, chunksize=64*1024):
    """ Encrypts a file using AES (CBC mode) with the
        given key.

        key:
            The encryption key - a string that must be
            either 16, 24 or 32 bytes long. Longer keys
            are more secure.

        in_filename:
            Name of the input file

        out_filename:
            If None, '<in_filename>.enc' will be used.

        chunksize:
            Sets the size of the chunk which the function
            uses to read and encrypt the file. Larger chunk
            sizes can be faster for some files and machines.
            chunksize must be divisible by 16.
    """
    if not out_filename:
        out_filename = in_filename + '.enc'

    iv = ''.join(chr(random.randint(0, 0xFF)) for i in range(16))
    encryptor = AES.new(key, AES.MODE_CBC, iv)
    filesize = os.path.getsize(in_filename)

    with open(in_filename, 'rb') as infile:
        with open(out_filename, 'wb') as outfile:
            outfile.write(struct.pack('<Q', filesize))
            outfile.write(iv)

            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                elif len(chunk) % 16 != 0:
                    chunk += ' ' * (16 - len(chunk) % 16)

                outfile.write(encryptor.encrypt(chunk))
```

Since it might have to pad the file to fit into a multiple of 16, the function saves the original file size in the first 8 bytes of the output file (more precisely, the first `sizeof(long long)` bytes). It randomly generates a 16-byte IV and stores it in the file as well. Then, it reads the input file chunk by chunk (with chunk size configurable), encrypts the chunk and writes it to the output. The last chunk is padded with spaces, if required.

Working in chunks makes sure that large files can be efficiently processed without reading them wholly into memory. For example, with the default chunk size it takes about 1.2 seconds on my computer to encrypt a 50MB file. PyCrypto is fast!

Decrypting the file can be done with:

```
def decrypt_file(key, in_filename, out_filename=None, chunksize=24*1024):
    """ Decrypts a file using AES (CBC mode) with the
        given key. Parameters are similar to encrypt_file,
        with one difference: out_filename, if not supplied
        will be in_filename without its last extension
        (i.e. if in_filename is 'aaa.zip.enc' then
        out_filename will be 'aaa.zip')
    """
    if not out_filename:
        out_filename = os.path.splitext(in_filename)[0]

    with open(in_filename, 'rb') as infile:
        origsize = struct.unpack('<Q', infile.read(struct.calcsize('Q')))[0]
        iv = infile.read(16)
        decryptor = AES.new(key, AES.MODE_CBC, iv)

        with open(out_filename, 'wb') as outfile:
            while True:
                chunk = infile.read(chunksize)
                if len(chunk) == 0:
                    break
                outfile.write(decryptor.decrypt(chunk))

            outfile.truncate(origsize)
```

First the original size of the file is read from the first 8 bytes of the encrypted file. The IV is read next to correctly initialize the AES object. Then the file is decrypted in chunks, and finally it's truncated to the original size, so the padding is thrown out.

For comments, please send me ✉ an email (mailto:eliben@gmail.com).

↑ Back to top