#http://stackoverflow.com/questions/6425131/encrpyt-decrypt-data-in-python-with-salt

import os, random, struct, string, re

try:

import Crypto.Random

from Crypto.Cipher import AES

import hashlib

ENCRYPTION\_ENABLE = 1

except:

ENCRYPTION\_ENABLE = 0

class encryption():

# salt size in bytes

SALT\_SIZE = 32

# number of iterations in the key generation

NUMBER\_OF\_ITERATIONS = 20

# the size multiple required for AES

AES\_MULTIPLE = 16

def \_\_init\_\_(self, saltFile, password):

try:

with open(saltFile, 'rb') as saltfile:

self.salt = saltfile.read()

except:

with open(saltFile, 'wb') as saltfile:

self.salt = self.generateSalt()

saltfile.write(self.salt)

if password != None and password != '':

self.key = self.generateKey(password,)

#print self.key

def generateKey(self,password, iterations=NUMBER\_OF\_ITERATIONS):

if ENCRYPTION\_ENABLE == 0:

return

assert iterations > 0

key = str(password) + str(self.salt)

#print "iterations " + str(iterations)

for i in range(iterations):

key = hashlib.sha256(key).digest()

return key

def generateSalt(self,size=SALT\_SIZE):

return ''.join(random.SystemRandom().choice(string.letters + string.digits) for \_ in range(size))

def pad\_text(text, multiple):

extra\_bytes = len(text) % multiple

padding\_size = multiple - extra\_bytes

padding = chr(padding\_size) \* padding\_size

padded\_text = text + padding

return padded\_text

def unpad\_text(padded\_text):

padding\_size = ord(padded\_text[-1])

text = padded\_text[:-padding\_size]

return text

def encryptFilename(fileName):

if ENCRYPTION\_ENABLE == 0:

return

import base64

return base64.b64encode(fileName)

def decrypt(fileName):

if ENCRYPTION\_ENABLE == 0:

return

try:

import base64

return base64.b64decode(fileName)

except:

return ''

def decryptFile(self, 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 ENCRYPTION\_ENABLE == 0:

return

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)

# key = generate\_key(password, salt, NUMBER\_OF\_ITERATIONS)

decryptor = AES.new(self.key, AES.MODE\_ECB)

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)

def decryptStream(self,response, chunksize=24\*1024):

if ENCRYPTION\_ENABLE == 0:

return

# with open(in\_filename, 'rb') as infile:

origsize = struct.unpack('<Q', response.read(struct.calcsize('Q')))[0]

decryptor = AES.new(self.key, AES.MODE\_ECB)

with open(out\_filename, 'w') as outfile:

while True:

chunk = response.read(chunksize)

if len(chunk) == 0:

break

outfile.write(decryptor.decrypt(chunk))

outfile.truncate(origsize)

def decryptStreamChunk44(self,response, wfile, chunksize=24\*1024):

if ENCRYPTION\_ENABLE == 0:

return

# with open(in\_filename, 'rb') as infile:

origsize = struct.unpack('<Q', response.read(struct.calcsize('Q')))[0]

decryptor = AES.new(self.key, AES.MODE\_ECB)

currentSize = origsize

while True:

chunk = response.read(chunksize)

currentSize = currentSize + chunksize

if len(chunk) == 0:

break

if currentSize <= origsize:

wfile.write(decryptor.decrypt(chunk))

else:

deltaSize = currentSize - origsize

fixSize = chunk - deltaSize

wfile.write(decryptor.decrypt(chunk)[:])

def decryptStreamChunk(self,response, wfile, chunksize=24\*1024, startOffset=0):

if ENCRYPTION\_ENABLE == 0:

return

# with open(in\_filename, 'rb') as infile:

origsize = struct.unpack('<Q', response.read(struct.calcsize('Q')))[0]

decryptor = AES.new(self.key, AES.MODE\_ECB)

count = 0

while True:

chunk = response.read(chunksize)

count = count + 1

if len(chunk) == 0:

break

responseChunk = decryptor.decrypt(chunk)

if count == 1 and startOffset !=0:

wfile.write(responseChunk[startOffset:])

elif (len(chunk)) < (len(responseChunk.strip())):

wfile.write(responseChunk.strip())

else:

wfile.write(responseChunk)

def decryptStreamChunk2(self,response, wfile, chunksize=24\*1024, startOffset=0):

if ENCRYPTION\_ENABLE == 0:

return

# with open(in\_filename, 'rb') as infile:

origsize = struct.unpack('<Q', response.read(struct.calcsize('Q')))[0]

decryptor = AES.new(self.key, AES.MODE\_ECB)

while True:

chunk = response.read(chunksize)

if len(chunk) == 0:

break

wfile.write(decryptor.decrypt(chunk))

def encryptFile(self, 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 ENCRYPTION\_ENABLE == 0:

return

if not out\_filename:

out\_filename = in\_filename + '.enc'

# key = generate\_key(key, salt, NUMBER\_OF\_ITERATIONS)

# iv = ''.join(chr(random.randint(0, 0xFF)) for i in range(16))

encryptor = AES.new(self.key, AES.MODE\_ECB)

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))

def encryptString(self, stringDecrypted):

if ENCRYPTION\_ENABLE == 0:

return

# key = generate\_key(key, salt, NUMBER\_OF\_ITERATIONS)

# iv = ''.join(chr(random.randint(0, 0xFF)) for i in range(16))

encryptor = AES.new(self.key, AES.MODE\_ECB)

if len(stringDecrypted) == 0:

return

elif len(stringDecrypted) % 16 != 0:

stringDecrypted += ' ' \* (16 - len(stringDecrypted) % 16)

import base64

stringEncrypted = base64.b64encode(encryptor.encrypt(stringDecrypted))

stringEncrypted = re.sub('/', '---', stringEncrypted)

return stringEncrypted

def decryptString(self, stringEncrypted):

if ENCRYPTION\_ENABLE == 0:

return

decryptor = AES.new(self.key, AES.MODE\_ECB)

if len(stringEncrypted) == 0:

return

import base64

stringEncrypted = re.sub('---', '/', stringEncrypted)

stringDecrypted = decryptor.decrypt(base64.b64decode(stringEncrypted))

return stringDecrypted