SSII

PSI-CONFIDENT

El presente informe documenta los procedimientos seguidos y por consiguiente los resultados obtenidos para resolver las cuestiones del proyecto PSI-Confident, el cual está destinado a garantizar la confidencialidad de los mensajes emitidos por una entidad en una comunicación cliente-servidor.

Fco. Javier Delgado Vallano

20 de Marzo del 2015

ÍNDICE

HISTORIAL DE VERSIONES

Versión	Fecha	Descripción
1.0	20/03/2015	Creación y estructuración del documento, desarrollo de planificación e introducción.
1.1	22/03/2015	Desarrollo de tarea 1 y tarea 3.
1.2	24/03/2015	Desarrollo de tarea 2 y de la conclusión.
1.3	25/03/2015	Desarrollo de reto y finalización del documento.

PLANIFICACIÓN

Tareas	Tiempo estimado	Tiempo total	Tiempo desviación
Informe	10 h.	9 h.	1 h.
Tarea 1	12 h.	15 h.	3 h.
Tarea 2	9 h.	8.5 h.	0.5 h.
Tarea 3	0.5 h.	0.5 h.	0 h.
Reto	3.5 h.	8 h.	4.5 h.

INTRODUCCIÓN

PSI-Confident es un proyecto que busca el garantizar la confidencialidad de los mensajes emitidos por una entidad en una comunicación cliente-servidor. Para ello se han barajado las opciones de *Python* y *Java* para desarrollar una aplicación, debido a su numerosa lista de librerías y gran comunidad que tienen a sus espaldas, pero finalmente fue el primero el elegido debido a ventajas como son:

- Código dinámico.
- Mayor eficiencia con respecto a *Java*.
- No necesita clases predefinidas, por lo que el número de líneas de código se ve reducido.

Mediante esta aplicación se intenta garantizar dicha confidencialidad haciendo uso de la las técnicas de cifrado, en concreto de la librería *pycrypto* del lenguaje *Python.*

El proceso consiste en encriptar y desencriptar, usando la técnica AES, Blowish o ARC4; los mensajes intercambiados entre cliente y servidor haciendo uso del modo *CFB* y una clave compartida por ambos.

Este modo, en esta versión del proyecto, es el único permitido hasta la fecha.

TAREA 1

La primera tarea a realizar consiste en desarrollar una comunicación confidencial entre cliente-servidor.

Para comenzar esta tarea se han realizado dos *scripts*, "cliente.py" y "servidor.py", los cuales serán tanto emisor como receptor respectivamente. El cliente será el encargado de emitir mensajes con la siguiente estructura:

Nombre de cuenta origen, Nombre de cuenta destino, cantidad transferida

Estos mensajes son cifrados mediante una técnica escogida, usando el modo *CBC* como se ha comentado anteriormente, el cual es escogido por trabajar con bloques de longitud fija; y por último, con una clave secreta compartida por ambos para realizar la el cifrado de clave secreta.

A continuación se muestra el código del *script* que toma la función de cliente, el cual tras haberse conectado a un socket (el del servidor) previamente especificado, envía al servidor el mensaje cifrado, y previo al cierre de la conexión recibe un mensaje del servidor notificando de si el proceso de descifrado ha sido satisfactorio o no.

```
# -*- coding: utf-8 -*-
import socket
import configparser
from Crypto.Cipher import AES
from Crypto.Cipher import Blowfish
from Crypto.Cipher import ARC4
import hashlib
import base64

config = configparser.ConfigParser()
config.read("configCliente.cfg")

size = int(config.get("S1", "blockSize"))
padding = config.get("S1","padding")*16

pad = lambda s: s + (size - len(s) % size) * padding

EncodeAES = lambda c, s: base64.b64encode(c.encrypt(pad(s)))
DecodeAES = lambda c, e: c.decrypt(base64.b64decode(e)).decode("ISO-8859-
1").rstrip(padding)

def cliente():

    IV = 16 * '\x00'
    password = config.get("S1", "keyClient").encode(encoding='utf_8')
    key = hashlib.sha256(password).digest()

    opciones = {
        1: 'AES',
        2 : 'Blowfish',
        3 : 'ARC4'
    }

    op_mode = {
        1 : "CBC",
        2 : "ECB",
        3 : "CFB"
    }

    cip = config.get("S1", "cipher")
    mode = config.get("S1", "mode")

    if cip == opciones.get(1):

        if mode == op_mode.get(1):
            m = AES.MODE_CBC
        elif mode == op_mode.get(2):
            m = AES.MODE_ECB
        else:
            m = AES.MODE_CFB

        cipher = AES.new(key,m,IV)

    elif cip == opciones.get(2):

        if mode == op_mode.get(1):
            m = Blowfish.MODE_CBC
        elif mode == op_mode.get(2):
            m = Blowfish.MODE_ECB
```

```
        else:
            m = Blowfish.MODE_CFB

        IV = 8 * "\x00"
        cipher = Blowfish.new(key, m ,IV)

    elif cip == opciones.get(3):
        cipher = ARC4.new(key)

    # Creando un socket TCP/IP
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Conecta el socket en el puerto cuando el servidor esté escuchando
    server_address = (config.get("S1", "ip"), int(config.get("S1",
"puerto")))

    print ('conectando a ' + server_address[0] + ' y puerto ' +
str(server_address[1]))

    sock.connect(server_address)

    try:

        # Enviando datos
        message = config.get("S1", "cuentaOrigen") + "," + config.get("S1",
"cuentaDestino") + "," + config.get("S1", "cantidad")
        print ('enviando..\n ' +  message)

        encoded = EncodeAES(cipher, message)
        print ('Encrypted string:', encoded)

        sock.sendall(encoded)

        # Buscando respuesta
        amount_received = 0
        amount_expected = len(message)

        while amount_received < amount_expected:
            data = sock.recv(1024)
            decoded = DecodeAES(cipher, data)
            amount_received += len(data)
            print ( 'recibiendo..\n' + str(decoded))

    finally:
        print ('cerrando socket\n\n')
        sock.close()
        del encoded
        del decoded
    del cipher


if __name__ == '__main__':

    cliente()
```

En cuanto al servidor, el cual es el encargado de recibir dichos mensajes y de comprobar si puede desencriptarlos para posteriormente procesarlos. Por tanto, el servidor actúa de la siguiente manera:

- En primer lugar, tras haber recibido el mensaje cifrado, se procede a desencriptarlo.
- A continuación, se comprueba que se haya realizado el paso anterior correctamente. Llegado a este punto, caben dos posibilidades, una en la que se haya realizado esto correctamente y otra en la que se ha producido un fallo y por tanto no se puede leer el mensaje recibido.
  En caso de que no se haya podido descifrar el mensaje, el servidor notificará al cliente con un mensaje como el siguiente:

ERROR – No se pudo descifrar el mensaje

Además de esto, se creará un archivo de texto en el que se almacene tanto el momento en el que el resultado fue negativo junto con el mensaje y la notificación pertinente.

En cambio, si dicho resultado es satisfactorio se notificara con:

OK

Para verificar que el mensaje ha sido descifrado correctamente se procede a comprobar que el mensaje recibido sólo contiene dígitos separados por comas.

21000813610123456789,21000813610123452222,125

- Finalmente, el servidor envía al cliente la notificación de si el proceso ha sido o no satisfactorio.

A continuación se muestra el código del *script* que permitirá las funciones del servidor:

```
# -*- coding: utf-8 -*-
import socket
import configparser
import datetime
from Crypto.Cipher import AES
from Crypto.Cipher import Blowfish
from Crypto.Cipher import ARC4
import hashlib
import base64
import unicodedata

config = configparser.ConfigParser()
config.read("configServer.cfg")

outfile = open('confidencial.txt', 'w')

size = int(config.get("S1", "blockSize"))
padding = config.get("S1","padding")*16
```

```
pad = lambda s: s + (size - len(s) % size) * padding

EncodeAES = lambda c, s: base64.b64encode(c.encrypt(pad(s)))
DecodeAES = lambda c, e: c.decrypt(base64.b64decode(e)).decode("ISO-8859-
1").rstrip(padding)


def servidor():

    IV = 16 * '\x00'
    password = config.get("S1", "keyServer").encode(encoding='utf_8')
    key = hashlib.sha256(password).digest()

    opciones = {
        1 : 'AES',
        2 : 'Blowfish',
        3 : 'ARC4'
    }

    op_mode = {
        1 : "CBC",
        2 : "ECB",
        3 : "CFB"
               }


    cip = config.get("S1", "cipher")
    mode = config.get("S1", "mode")

    if cip == opciones.get(1):

        if mode == op_mode.get(1):
            m = AES.MODE_CBC
        elif mode == op_mode.get(2):
            m = AES.MODE_ECB
        else:
            m = AES.MODE_CFB

        cipher = AES.new(key,m,IV)

    elif cip == opciones.get(2):

        if mode == op_mode.get(1):
            m = Blowfish.MODE_CBC
        elif mode == op_mode.get(2):
            m = Blowfish.MODE_ECB
        else:
            m = Blowfish.MODE_CFB

        IV = 8 * "\x00"
        cipher = Blowfish.new(key, m ,IV)

    elif cip == opciones.get(3):
        cipher = ARC4.new(key)


    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Enlace de socket y puerto
```

```
    server_address = (config.get("S1", "ip"), int(config.get("S1",
"puerto")))
    print ( 'Empezando a levantar ' + server_address[0] + ' puerto ' +
str(server_address[1]))
    sock.bind(server_address)

    # Escuchando conexiones entrantes
    sock.listen(1)

    recibidos = []

    while True:
        # Esperando conexion
        print ('Esperando para conectarse\n')
        connection, client_address = sock.accept()

        try:
            print ('conexion desde ' + client_address[0])

            # Recibe los datos en trozos y reetransmite
            while True:
                data = connection.recv(1024)
                data = DecodeAES(cipher, data)
                recibidos.append(data)

                print ('recibido..\n' + data)

                notifi = "OK"

                data = str(data).replace(",", "")
                data = str(data).replace(" ", "")

                if (data.isdigit() == False):

                    outfile = open('confidencial.txt', 'a')
                    notifi = "ERROR - No se pudo descifrar el mensaje"
                    out = str(datetime.datetime.now()) + " - " + str(data) +
" - " + notifi + " \n"
                    outfile.write(out)
                    outfile.close()

                if data:

                    print ('Enviando mensaje de vuelta al cliente')
                    encoded = EncodeAES(cipher, notifi)
                    print ('Encrypted string:', encoded)
                    connection.sendall(encoded)



                else:
                    print ('No hay mas datos de ' + client_address[0])

                    break

                break

        except EOFError:
            ret = []
```

```
        finally:
            # Cerrando conexion

            connection.close()
    del encoded
    del data
    del cipher


if __name__ == '__main__':

    servidor()
```

Además de lo anteriormente descrito, el proyecto cuenta con dos archivos de configuración llamados “configCliente.cfg” y “configServer.cfg”. Estos archivos serán los encargados de almacenar las variables necesarias para ejecutar la comunicación.

En el caso del archivo de configuración del cliente, este contiene el socket compartido por el cliente y el servidor, el cifrado y el modo a utilizar, el padding, las componentes del mensaje a emitir por este, el tamaño de los bloques (necesario para los objetos *Cipher*) y la clave secreta compartida por ambos. Y en cuanto a la configuración del servidor, este almacena los datos ya comentados a excepción de los componentes del mensaje.

Por tanto, antes de poner en funcionamiento el proyecto debemos dirigirnos a estos archivos de configuración para definir dichos parámetros como podemos observar a continuación:

- Para el caso de “configCliente.cfg”:

```
[S1]
cipherType: AES, Blowfish, ARC4
cipher: AES
modeType: CFB
mode: CFB
padding: {
blockSize: 16
ip: localhost
puerto: 10000
cuentaOrigen: 21000813610123456789
cuentaDestino: 2100081361012345222 2
cantidad: 125
keyClient: claveServer
```

- Para el caso de “configServer.cfg”:

```
[S1]
cipherType: AES, Blowfish, ARC4
cipher: AES
modeType: CFB
mode: CFB
padding: {
blockSize: 16
ip: localhost
puerto: 10000
KeyServer: claveServer
```

En estos archivos podemos observar una única sección y los parámetros a completar para poner en funcionamiento nuestra comunicación:

- “S1”:
  - cipher: Tipo de cifrado a utilizar.
  - mode: Modo de cifrado a usar.
  - padding: Padding uilizado en el cifrado de los mensajes intecambiados.
  - ip: Dirección ip del servidor.
  - puerto: Puerto abierto del servidor.
  - cuentaOrigen: Número de cuenta origen.
  - cuentaDestino: Número de cuenta destino.
  - cantidad: Cantidad de dinero a traspasar desde la cuenta origen a la cuenta destino.
  - keyClient: Clave secreta compartida por cliente y servidor.
  - KeyServer: Clave sercreta compartida por cliente y servidor.

Por último, centrándonos en la clave secreta y a la vez compartida por el cliente y el servidor, ya que en el presente proyecto no nos centramos en los mecanismos de negociaciones de claves, suponemos dichas claves previamente negociadas.

TAREA 2

La segunda tarea propuesta para el presente proyecto, consiste en analizar la complejidad computacional de los algoritmos de cifrado/descifrado para el almacenamiento de archivos.

Para ello se han realizado varios *scripts*, uno por técnica de cifrado, en el que se probará su uso en archivos de diferente tamaño, haciendo posible de este modo el análisis tanto temporal como espacial. En estos *scripts* se realiza la comprobación del tiempo que tardan estas técnicas en cifrar y descifrar los diferentes archivos en modo de cifrado por bloque "CBC", así como también la variación del tamaño de los archivos resultantes de estas operaciones.

Los tamaños de estos archivos son de 10Kb, 1Mb, 10Mb y 20Mb; todos ellos aproximado.

A continuación se mostrará el código de estos *scripts*.

En primer lugar, tenemos el código de cifrado AES con clave simétrica:

```
from Crypto import Random
from Crypto.Cipher import AES
import time
import os

def pad(s):

    return s + b"\0" * (AES.block_size - len(s) % AES.block_size)

def encrypt(message, key, key_size=256):

    message = pad(message)
    iv = Random.new().read(AES.block_size)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return iv + cipher.encrypt(message)

def decrypt(ciphertext, key):

    iv = ciphertext[:AES.block_size]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext[AES.block_size:])
    return plaintext.rstrip(b"\0")

def encrypt_file(file_name, key):
    tami = os.path.getsize(file_name)
    print( "Tamaño inicial : " + str(tami) )
    inicio = time.time()*1000
    with open(file_name, 'rb') as fo:
        plaintext = fo.read()
    fo.close()
    enc = encrypt(plaintext, key)
    name = file_name.replace(".","")
    with open(name + ".enc", 'wb') as fo:
```

```
            fo.write(enc)
        fo.close()
        fin = time.time()*1000
        outfile = name + ".enc"
        tamf = os.path.getsize(outfile)
        tiempo_total = fin - inicio
        print( "Tiempo de ejecucion : " + str(tiempo_total) )
        print( "Tamaño final : " + str(tamf) + "\n")

    def decrypt_file(file_name, key):
        inicio = time.time()*1000
        tami = os.path.getsize(file_name)
        print( "Tamaño inicial : " + str(tami) )
        with open(file_name, 'rb') as fo:
            ciphertext = fo.read()
        fo.close()
        dec = decrypt(ciphertext, key)
        name = file_name[:-4] + ".txt"
        with open(file_name[:-4] + ".txt", 'wb') as fo:
            fo.write(dec)
        fo.close()
        fin = time.time()*1000
        tamf = os.path.getsize(name)
        tiempo_total = fin - inicio
        print( "Tiempo de ejecucion : " + str(tiempo_total) )
        print( "Tamaño final : " + str(tamf) )



    key =
    b'\xbf\xc0\x85)\x10nc\x94\x02)j\xdf\xcb\xc4\x94\x9d(\x9e[EX\xc8\xd5\xbfI{\xa2
    $\x05(\xd5\x18'


    if __name__ == '__main__':

        print("Archivo de 10Mb")
        encrypt_file('10MB.txt', key)
        decrypt_file('10MBtxt.enc', key)

        print("\nArchivo de 1Mb")
        encrypt_file('1MB.txt', key)
        decrypt_file('1MBtxt.enc', key)

        print("\nArchivo de 20Mb")
        encrypt_file('20MB.txt', key)
        decrypt_file('20MBtxt.enc', key)

        print("\nArchivo de 10Kb")
        encrypt_file('10KB.txt', key)
        decrypt_file('10KBtxt.enc', key)
```

En segundo lugar, tenemos el código de cifrado Blowfish con clave simétrica:

```
from Crypto import Random
from Crypto.Cipher import Blowfish
import time
import os

def pad(s):

    return s + b"\0" * (Blowfish.block_size - len(s) % Blowfish.block_size)

def encrypt(message, key, key_size=256):

    message = pad(message)
    iv = Random.new().read(Blowfish.block_size)
    cipher = Blowfish.new(key, Blowfish.MODE_CBC, iv)
    return iv + cipher.encrypt(message)

def decrypt(ciphertext, key):

    iv = ciphertext[:Blowfish.block_size]
    cipher = Blowfish.new(key, Blowfish.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext[Blowfish.block_size:])
    return plaintext.rstrip(b"\0")

def encrypt_file(file_name, key):
    tami = os.path.getsize(file_name)
    print( "Tamaño inicial : " + str(tami) )
    inicio = time.time()*1000
    with open(file_name, 'rb') as fo:
        plaintext = fo.read()
    fo.close()
    enc = encrypt(plaintext, key)
    name = file_name.replace(".","")
    with open(name + ".enc", 'wb') as fo:
        fo.write(enc)
    fo.close()
    fin = time.time()*1000
    outfile = name + ".enc"
    tamf = os.path.getsize(outfile)
    tiempo_total = fin - inicio
    print( "Tiempo de ejecucion : " + str(tiempo_total) )
    print( "Tamaño final : " + str(tamf) + "\n")

def decrypt_file(file_name, key):
    inicio = time.time()*1000
    tami = os.path.getsize(file_name)
    print( "Tamaño inicial : " + str(tami) )
    with open(file_name, 'rb') as fo:
        ciphertext = fo.read()
    fo.close()
    dec = decrypt(ciphertext, key)
    name = file_name[:-4] + ".txt"
    with open(name, 'wb') as fo:
```

```
        fo.write(dec)
    fo.close()
    fin = time.time()*1000
    tamf = os.path.getsize(name)
    tiempo_total = fin - inicio
    print( "Tiempo de ejecucion : " + str(tiempo_total) )
    print( "Tamaño final : " + str(tamf) )


key =
b'\xbf\xc0\x85)\x10nc\x94\x02)j\xdf\xcb\xc4\x94\x9d(\x9e[EX\xc8\xd5\xbfI{\xa2
$\x05(\xd5\x18'


if __name__ == '__main__':

    print("Archivo de 10Mb")
    encrypt_file('10MB.txt', key)
    decrypt_file('10MBtxt.enc', key)

    print("\nArchivo de 1Mb")
    encrypt_file('1MB.txt', key)
    decrypt_file('1MBtxt.enc', key)

    print("\nArchivo de 20Mb")
    encrypt_file('20MB.txt', key)
    decrypt_file('20MBtxt.enc', key)

    print("\nArchivo de 10Kb")
    encrypt_file('10KB.txt', key)
          decrypt_file('10KBtxt.enc', key)
```

En tercer lugar, tenemos el código de cifrado DES3 con clave simétrica, en el que debemos tener la consideración de utilizar una clave de tamaño de 16 o 24 bytes:

```
from Crypto import Random
from Crypto.Cipher import DES3
import time
import os

def pad(s):

    return s + b"\0" * (DES3.block_size - len(s) % DES3.block_size)

def encrypt(message, key, key_size=256):

    message = pad(message)
    iv = Random.new().read(8)
    cipher = DES3.new(key, DES3.MODE_CBC, iv)
    return iv + cipher.encrypt(message)

def decrypt(ciphertext, key):

    iv = Random.new().read(8)
```

```
    cipher = DES3.new(key, DES3.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext[DES3.block_size:])
    return plaintext.rstrip(b"\0")


def encrypt_file(file_name, key):
    tami = os.path.getsize(file_name)
    print( "Tamaño inicial : " + str(tami) )
    inicio = time.time()*1000
    with open(file_name, 'rb') as fo:
        plaintext = fo.read()
    fo.close()
    enc = encrypt(plaintext, key)
    name = file_name.replace(".","")
    with open(name + ".enc", 'wb') as fo:
        fo.write(enc)
    fo.close()
    fin = time.time()*1000
    outfile = name + ".enc"
    tamf = os.path.getsize(outfile)
    tiempo_total = fin - inicio
    print( "Tiempo de ejecucion : " + str(tiempo_total) )
    print( "Tamaño final : " + str(tamf) + "\n")

def decrypt_file(file_name, key):
    inicio = time.time()*1000
    tami = os.path.getsize(file_name)
    print( "Tamaño inicial : " + str(tami) )
    with open(file_name, 'rb') as fo:
        ciphertext = fo.read()
    fo.close()
    dec = decrypt(ciphertext, key)
    name = file_name[:-4] + ".txt"
    with open(name, 'wb') as fo:
        fo.write(dec)
    fo.close()
    fin = time.time()*1000
    tamf = os.path.getsize(name)
    tiempo_total = fin - inicio
    print( "Tiempo de ejecucion : " + str(tiempo_total) )
    print( "Tamaño final : " + str(tamf) )


#key must be either 16 or 24 bytes long
key = b'-8B key--8B key-'

if __name__ == '__main__':

    print("Archivo de 10Mb")
    encrypt_file('10MB.txt', key)
    decrypt_file('10MBtxt.enc', key)

    print("\nArchivo de 1Mb")
    encrypt_file('1MB.txt', key)
    decrypt_file('1MBtxt.enc', key)

    print("\nArchivo de 20Mb")
    encrypt_file('20MB.txt', key)
    decrypt_file('20MBtxt.enc', key)
```

```
    print("\nArchivo de 10Kb")
    encrypt_file('10KB.txt', key)
          decrypt_file('10KBtxt.enc', key)
```

Y en cuarto y último lugar, tenemos el código de cifrado RSA con claves pública y privada, en el que debemos tener en cuenta que sólo puede cifrar/descifrar elementos de 128 caracteres:

```
from Crypto.PublicKey import RSA
import os
import time


def generate_RSA(bits):

    new_key = RSA.generate(bits, e=65537)
    pub = open("public_key.der","w")
    pri = open("private_key.der","w")
    public_key = new_key.publickey().exportKey("DER")
    pub.write(str(public_key))
    private_key = new_key.exportKey("DER")
    pri.write(str(private_key))
    pub.close()
    pri.close()
    return private_key, public_key

def encrypt_file( file, public):

    tami = os.path.getsize(file)
    print( "Tamaño inicial: " + str(tami) )
    inicio = time.time()*1000
    with open(file, 'rb') as fo:
        file_to_encrypt = fo.read()
    fo.close()
    o = RSA.importKey(public)
    to_join = []
    step = 0

    while 1:
        # Read 128 characters at a time.
        s = file_to_encrypt[step*128:(step+1)*128]
        if not s:
            break
        # Encrypt with RSA and append the result to list.
        # RSA encryption returns a tuple containing 1 string, so i fetch the
string.
        to_join.append(o.encrypt(s, 0)[0])
        step += 1

    encrypted = b''.join(to_join)
    name = file.replace(".","")
    nam = name + ".enc"
    with open(nam, 'wb') as fo:
        fo.write(encrypted)
    fo.close()
    fin = time.time()*1000
    tamf = os.path.getsize(nam)
```

```
    tiempo_total = fin - inicio
    print( "Tiempo de ejecucion : " + str(tiempo_total) )
    print( "Tamaño final: " + str(tamf) + "\n" )

def decrypt_file( file, private):

    tami = os.path.getsize(file)
    print( "Tamaño inicial: " + str(tami) )
    inicio = time.time()*1000
    with open(file, 'rb') as fo:
        file_to_decrypt = fo.read()
    fo.close()
    o = RSA.importKey(private)
    to_join = []
    step = 0

    while 1:
        # Read 128 characters at a time.
        s = file_to_decrypt[step*128:(step+1)*128]
        if not s:
            break
        # Encrypt with RSA and append the result to list.
        # RSA encryption returns a tuple containing 1 string, so i fetch the
string.
        to_join.append(o.decrypt(s))
        step += 1

    decrypted = b''.join(to_join)
    name = file.replace(".","")
    nam = name[:-4] + ".txt"
    with open(nam, 'wb') as fo:
        fo.write(decrypted)
    fo.close()
    fin = time.time()*1000
    tamf = os.path.getsize(nam)
    tiempo_total = fin - inicio
    print( "Tiempo de ejecucion : " + str(tiempo_total) )
    print( "Tamaño final: " + str(tamf) + "\n" )


if __name__ == '__main__':

    g = generate_RSA(2048)

    print("\nArchivo de 10Kb")
    encrypt_file('10KB.txt', g[1])
    decrypt_file('10KBtxt.enc', g[0])

    print("\nArchivo de 1Mb")
    encrypt_file('1MB.txt', g[1])
    decrypt_file('1MBtxt.enc', g[0])

    print("\nArchivo de 10Mb")
    encrypt_file('10MB.txt',g[1] )
    decrypt_file('10MBtxt.enc', g[0])

    print("\nArchivo de 20Mb")
    encrypt_file('20MB.txt', g[1])
    decrypt_file('20MBtxt.enc', g[0])
```

Por tanto, tras haber realizado la ejecución de cada uno de estos *scripts*, se procede a realizar dos tablas en las que se recogen tanto los tiempos, en milisegundos y sin contar con el tiempo necesario para verificar la integridad de la información; y la diferencia de tamaño ("+" aumento | "-" reducción), en bytes, entre los ficheros de entradas y los de salida al ejecutar tanto el cifrado como el descifrado.

En primer lugar, tenemos la tabla que recoge en este caso los datos resultantes de haber realizado el cifrado de los archivos de entrada.

Cifrado Tamaño	3DES (ms)	3DES (b)	BlowFish (ms)	Blowfish (b)	AES (ms)	AES (b)	RSA (ms)	RSA (b)
10 KB	0.0	+10	0.0	+10	0.0	+19	38.716	+102424
1 MB	78.125	+14	15.613	+14	15.625	+30	2250.590	+1122110
10 MB	515.630	+11	156.232	+11	103.307	+19	20040.050	+10500083
20 MB	1000.010	+14	406.251	+14	234.376	+22	40355.856	+20681558

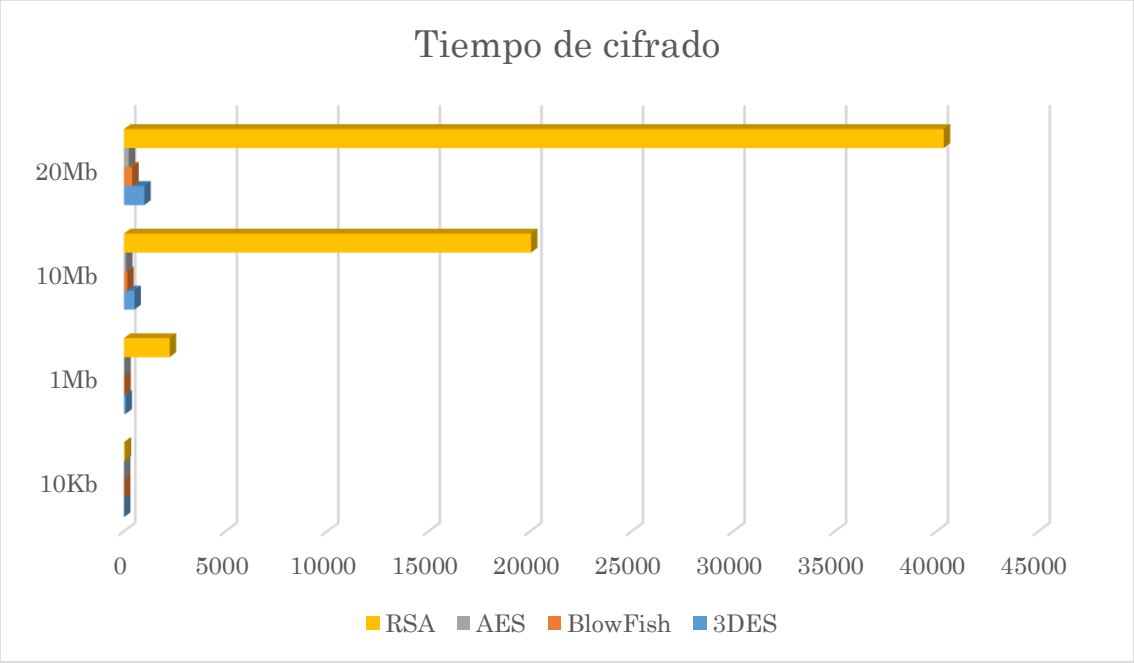
Y en segundo lugar, tenemos la tabla que recoge en este caso los datos resultantes de haber realizado el descifrado de los archivos de entrada.

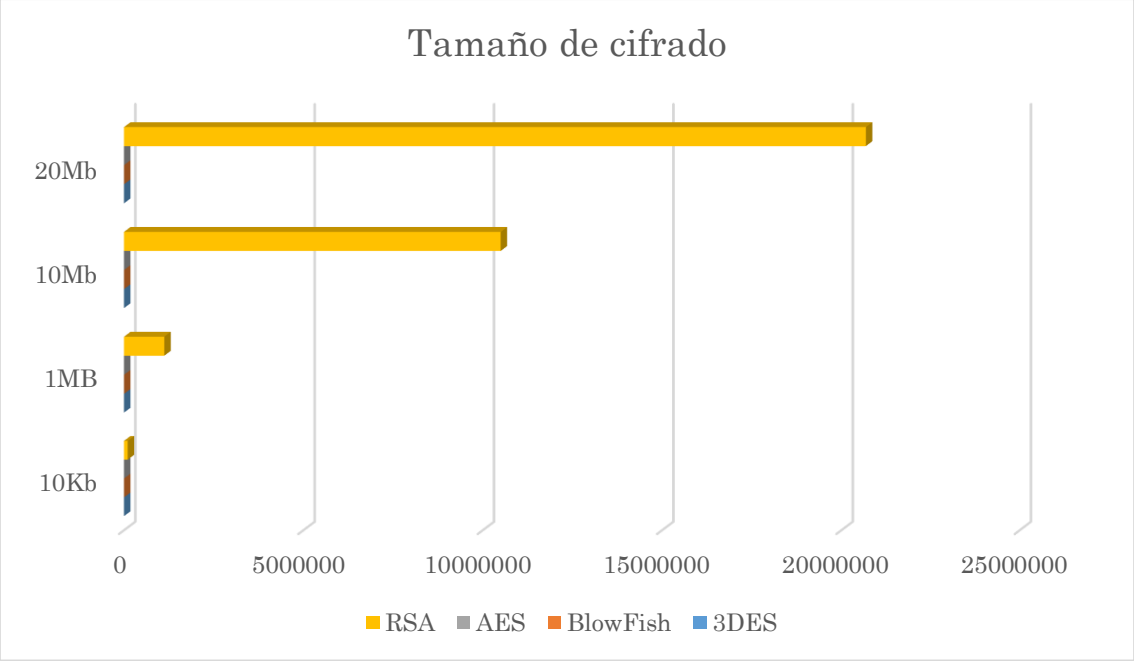
Descifrado Tamaño	3DES (ms)	3DES (b)	BlowFish (ms)	Blowfish (b)	AES (ms)	AES (b)	RSA (ms)	RSA (b)
10 KB	0.0	-10	0.0	-10	0.0	-19	2142.247	+20475
1 MB	46.893	-14	15.641	-14	0.0	-30	222382.718	+2244096
10 MB	484.376	-11	140.638	-11	109.375	-19	1990222.201	+20999936
20 MB	991.932	-14	281.237	-14	187.50	-22	3871486.317	+41362944

Una vez obtenido los resultados, como conclusión de las técnicas probadas para el cifrado obtenemos que el peor método de cifrado es RSA, tanto por el enorme tiempo de ejecución como por el gran tamaño de los archivos de salida con respecto a los de entrada.

Además, podemos comprobar que el método más veloz es AES, sobre todo para el cifrado de archivos de mayor tamaño; y también por último, en cuanto a la variación de tamaño de los archivos de salida, vemos que las técnicas de 3DES y BlowFish son las que denotan una menor variación.

A continuación, podemos observar las gráficas obtenidas a través de los datos recogidos en la primera tabla, es decir, la de cifrado.

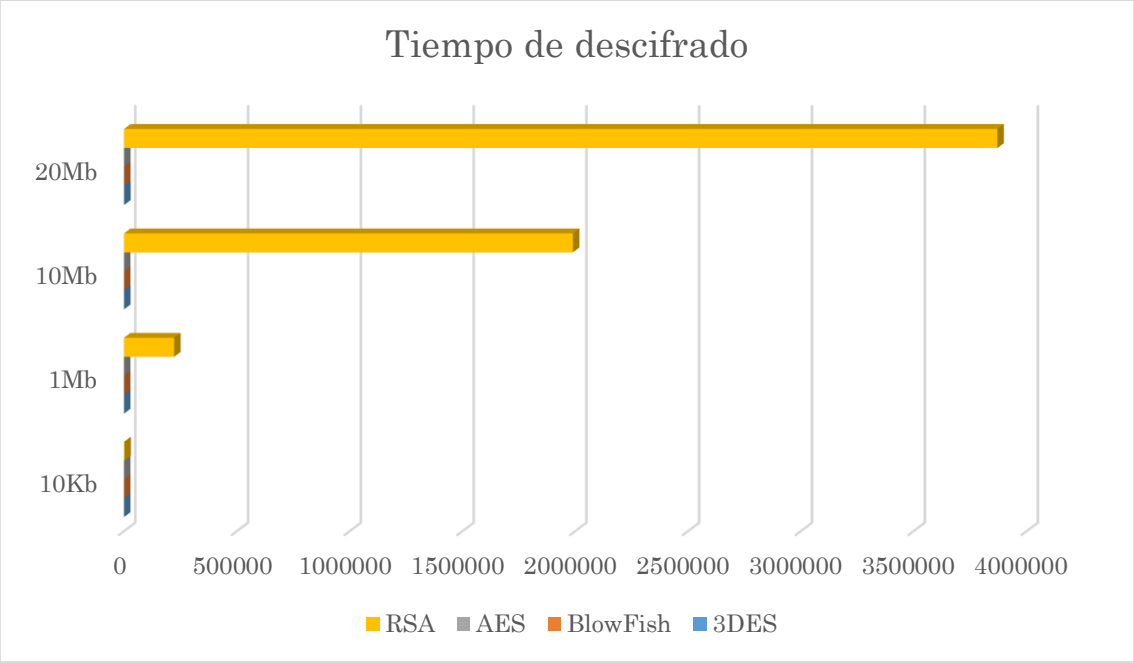


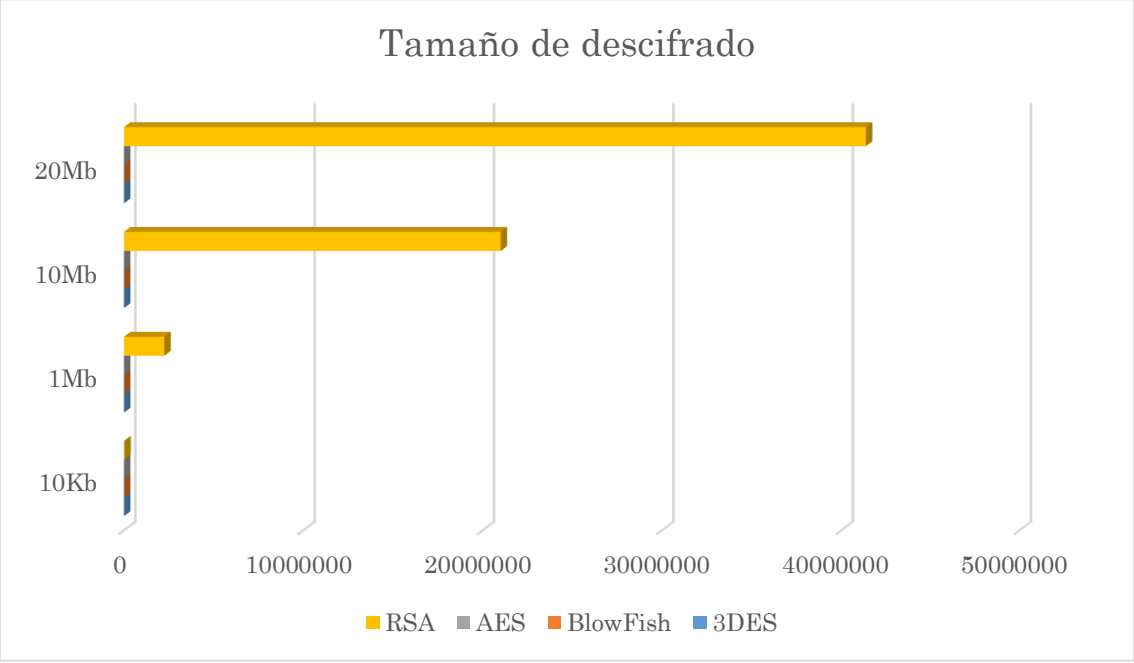


Ahora, centrándonos en los resultados obtenidos al descifrar, destacamos que al igual que para cifrar RSA es el que peores resultados muestra por a la enorme diferencia de variación de tamaño y tiempo de ejecución con respecto a los otros tipos de cifrados.

Comprobamos también, que al igual que antes, AES es el más eficiente en cuanto a tiempo, pero un poco menos que 3DES y BlowFish en cuanto a las variaciones de tamaño.

A continuación, podemos observar las gráficas obtenidas a través de los datos recogidos en la primera tabla, es decir, la de descifrado.





Finalmente, tras haber realizado el análisis de los datos obtenidos, se da respuesta a las siguientes cuestiones:

1. **¿qué algoritmo es más conveniente de forma general? Justificarlo.**

   El algoritmo más conveniente de usar es AES por su gran eficacia tanto en complejidad temporal como espacial.

2. **¿qué tamaño de clave es más recomendable y por qué? Justificarlo.**

   En cuanto al tamaño de la clave, el más recomendable es de 16 bytes, ya que es compatible con más algoritmos de cifrado/descifrado, aunque tenga en contra que su tamaño sea pequeño.

3. **¿qué modo de cifrado por bloques es mejor para almacenar las imágenes de radiografías? ¿Importa de forma el tipo de *padding* usada en el cifrado y cuál recomienda usar? Justificarlo.**

   El mejor modo de cifrado por bloques para el manejo de imágenes es CTR, en el que el cifrado de cada bloque se realiza independientemente de los demás bloques y reduciendo hasta ser nula la localización de información en la frecuencia de aparición de los símbolos resultantes del cifrado.

   En cuanto al padding de cifrado, lo más destacable es el tamaño a usar, ya que este debe ser según la información recibida en el último bloque. Los más recomendados a usar son PKCS#5 y PKCS#7.

4. **Compruebe en todas las pruebas que el cifrado/descifrado conserva la integridad de todas las historias clínicas y radiografías. Informe sobre cómo lo ha realizado.**

   Para comprobar en las pruebas si se ha mantenido la integridad, se ha procedido a calcular el *hash* de la información contenida en cada archivo antes de ser cifrada y tras ser descifrada, para su posterior comparación, y de este modo comprobar que el resultado es el esperado. El resultado de la comparación se notifica mediante un mensaje que se mostrará en la consola.

   En cuanto a los resultados, en los casos de AES y BlowFish se mantiene la integridad. En cambio, en el caso de DES3 no se mantiene, por el hecho de que inserta una pequeña de caracteres al inicio der archivo.

   Finalmente, cabe destacar que la verificación de la integridad para el caso de RSA no se ha podido realizar debido a su mal funcionamiento.

Como nota última, cabe mencionar que los datos mostrados en cuando al algoritmo RSA al descifrar se corresponden a la presente implementación de este, que parece no funcionar correctamente.

TAREA 3

La tercera tarea propuesta consiste en aportar un *feedback* a la entidad financiera en la que se aporten posibles mejoras que se puedan realizar al producto entregado.

Algunas de las mejoras aplicables pueden ser:

- Desarrollar una interfaz de usuario que permita realizar la configuración inicial del programa de manera más visual, cómoda y sencilla para cualquier usuario.
- Proporcionar más técnicas de cifrado así como más modos.
- Verificar la integridad de los mensajes intercambiados.
- Modificar el servidor para que se inicie automáticamente con el inicio del sistema.
- Implementar el método de cifrado de clave híbrida.
- Almacenar la clave secreta cifrada y obligar al usuario a cambiarla cada cierto tiempo.
- Eliminar cada cierto tiempo los mensajes almacenados tanto en cliente como en servidor.

RETO

El reto propuesto para el presente proyecto consiste en cifrar mediante el algoritmo de AES, una serie de imágenes haciendo uso del modo de cifrado de ECB.

Para ello se ha confeccionado un *script* que nos permita realizar el cifrado de una determinada imagen para después ver la imagen resultante y comprobar de este modo las ventajas y desventajas de este modo de cifrado.

A continuación se muestra el código desarrollado para llevar a cabo lo anteriormente descrito. Este código no realiza correctamente el cifrado ya que, si que cifra la información contenida en la imagen pero no se puede visualizar la imagen resultante ya que también cifra los datos de la cabecera de la imagen.

Por tanto, el código utilizado es el siguiente:

```
from Crypto.Cipher import AES
import Crypto.Random
from PIL import Image
import base64


def mostrarImagen(imagen):

    im = Image.open(imagen)
    im.show()
```

```
def pad(s):

    return s + b"\0" * (AES.block_size - len(s) % AES.block_size)

def encrypt(message, key, key_size=256):

    message = pad(message)
    iv = Crypto.Random.new().read(AES.block_size)
    cipher = AES.new(key, AES.MODE_ECB, iv)
    return iv + cipher.encrypt(message)

def encrypt_file(file_name, key):

    with open(file_name, 'rb') as fo:
        plaintext = base64.b64encode(fo.read())
    fo.close()
    enc = encrypt(plaintext, key)
    name = file_name.replace(".","")
    with open(name + ".bmp", 'wb') as fo:
        fo.write(enc)
    fo.close()


key =
b'\xbf\xc0\x85)\x10nc\x94\x02)j\xdf\xcb\xc4\x94\x9d(\x9e[EX\xc8\xd5\xbfI{\xa2
$\x05(\xd5\x18'

if __name__ == "__main__":

    encrypt_file("us.bmp",key)
```

Como nota final, destacar que las pruebas se han realizado con imágenes con extensión “.bmp”.

Aun así, sin obtener los resultados esperados se puede comentar que el modo de cifrado de ECB es un cifrado por bloques que apenas se debe usar debido a que la frecuencia de repetición de caracteres en elementos grandes es elevada, por lo que resulta ser poco seguro. En cambio, con elementos pequeños y aleatorios que no presenten repeticiones en sus valores, puede resultar interesante. Además de todo esto, este modo permite acelerar la ejecución de los cifradores.

CONCLUSIÓN

La confidencialidad en la transmisión de mensajes consta de muchos procedimientos, los cuales van desde la negociación de las claves para permitir el cifrado y descifrado de estos hasta las propias técnicas de cifrado. Actualmente, con el avance de las tecnologías y de los ataques existentes, debemos tener especial cuidado al seleccionar los métodos y técnicas a utilizar para garantizar la confidencialidad.

Además de esto, cabe mencionar el aprendizaje de las técnicas existentes para mantener dicha confidencialidad, así como su funcionamiento y su implementación; e incluso también, la importancia y la dificultad de elegir un buen *padding* y el modo de cifrado más adecuado, de modo que gracias a estos el riesgo de ataque se vea enormemente reducido y la comunicación se convierta en un proceso más rápido y efectivo.