Lab Assessment

CSD3005-DATA PRIVACY-[redacted]

Bachelor of Technology in Computer Science & Engineering
with specialisation in Cybersecurity and Digital Forensics

NAME : Saket Upadhyay

REG. NO. : [redacted]

SLOT : [redacted]

FACULTY : [redacted]

Semester : Winter Interim Semester 2020

School of Computer Science and Engineering

VIT Bhopal

Table of Contents

TITLE

"Studying Efficient Encryption Strategy of PGP / GPG"

AIM

The aim of this experiment is to understand the key encryption strategy of PGP, which makes it faster, while still using Public Key Encryption. I want to see the time efficiency of RSA full data encryption v/s RSA key-only encryption v/s AES full data encryption.

THEORY

Advanced Encryption Standard

AES, also, known by its original name *Rijndael*, is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.

AES is a block cipher based on substitution–permutation network, and is efficient in both software and hardware. It is a variant of Rijndael, with a fixed block size of 128 bits, and a key size of 128, 192, or 256 bits.

Rivest–Shamir–Adleman

RSA is an asymmetric key encryption, which is not usually used to encrypt large chunks of data.

It is a relatively slow algorithm. Because of this, it is not commonly used to *directly encrypt user data*. More often, RSA is used to transmit shared keys for symmetric key cryptography, which are then used for bulk encryption-decryption.

But, in theory (and practically), we can still use it to encrypt large data by dividing it into smaller chunks, or blocks, using RSA as a type of block cipher.

An RSA user creates and publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers are kept secret. Messages can be encrypted by anyone, via the public key, but can only be decoded by someone who knows the prime numbers.

PGP & PGP Strategy

PGP encryption uses a serial combination of hashing, data compression, symmetric-key cryptography, and finally public-key cryptography; each step uses one of several supported algorithms. Each public key is bound to a username or an e-mail address.

The GNU Privacy Guard. GnuPG is a complete and free implementation of the OpenPGP standard as defined by RFC4880 (also known as PGP)

Generally, the standard recommends RSA as described in PKCS#1 v1.5 for encryption and signing, as well as AES-128, CAST-128 and IDEA. Beyond these, many other algorithms are supported.

Here, by PGP Strategy I mean the approach of PGP and OpenGP (GPG), where they used AES,CAST or IDEA for large data encryption and then uses RSA to encrypt the keys of AES,CAST or IDEA.

PROCEDURE

The main goal of the experiment being comparing RSA and AES for bulk data encryption and understanding why PGP's approach is efficient, we will be creating a standardised test data file with random printable ASCII characters, which we will then encrypt using three approaches as follows :

1. RSA only
2. AES only
3. AES for bulk and RSA for AES's keys (PGP strategy)

Comparing the time taken for key generation, encryption and decryption by each approach.

The details of the experiment are:

Data File Size	724 KB (741,600 bytes)
P-ASCII Characters in the file	741601
No. of Lines	1

RSA Properties

Encoding	Base64
Encryption	PKCS1_v1_5
Key Size	1024 bits

AES Properties

Checksum	SHA256
Length	32
Salted	Yes
Iteration	1,00,000
Mode	CBC

File Checksum: MD5

PROGRAM

The program I made is in python3.

RSA_only.py

```
import hashlib
from Crypto.Cipher import PKCS1_v1_5
from Crypto.PublicKey import RSA
from Crypto.Random import new as Random
from base64 import b64encode
from base64 import b64decode
import random
import time
from os import system

DataList = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U',
            'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
            'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '0',
'1', '2', '3', '4', '5', '6', '7', '8', '9']


class RSA_Cipher:
    def generate_key(self, key_length):
        assert key_length in [1024, 2048, 4096]
        rng = Random().read
        self.key = RSA.generate(key_length, rng)

    def save_key(self):
        PK = self.key
        with open("RSAPrivateKeyFile", 'wb') as RKF:
            RKF.write(PK.export_key('PEM'))

    def load_key(self):
        with open("RSAPrivateKeyFile", 'rb') as PKF:
            self.key = RSA.import_key(PKF.read())

    def encrypt(self, data):
        plaintext = b64encode(data.encode())
        rsa_encryption_cipher = PKCS1_v1_5.new(self.key)
        ciphertext = rsa_encryption_cipher.encrypt(plaintext)
        return b64encode(ciphertext).decode()
```

```
    def decrypt(self, data):
        ciphertext = b64decode(data.encode())
        rsa_decryption_cipher = PKCS1_v1_5.new(self.key)
        plaintext = rsa_decryption_cipher.decrypt(ciphertext, 16)
        return b64decode(plaintext).decode()


def generatedata(charnum):
    data = ""
    for i in range(charnum):
        RC = random.choice(DataList)
        data = data + RC
    return data


def generatedatafile(charnum):
    with open("DataFile2", 'w') as DF:
        for i in range(charnum):
            RC = random.choice(DataList)
            DF.write(RC)


def encryptD2file():
    EncryptedDataList = []
    RSACipherObject = RSA_Cipher()
    RSACipherObject.load_key()

    with open("DataFile2", 'r') as DF2:
        Fdata = DF2.read()
    s = 0
    e = s + 80

    for i in range(10000):
        # print("Encrypting Block "+str(i))
        if Fdata[s:e] == '':
            break

EncryptedDataList.append(RSACipherObject.encrypt(Fdata[s:e]))
        s = s + 80
        e = e + 80
    try:
        system("del EncD2File")
    except Exception:
        print("NF : Ignore")
    with open("EncD2File", "a+") as EF:
        for line in EncryptedDataList:
            EF.write(line + "\n")
```

```
def decryptD2file():
    RSACipherObject = RSA_Cipher()
    RSACipherObject.load_key()
    with open("EncD2File", 'r') as EF:
        ETs = EF.readlines()
    try:
        system("del DecD2File")
    except Exception:
        print("NF: Ignore")
    with open("DecD2File", 'a+') as DF:
        t = 0
        for i in ETs:
            # print("Decrypting Block " + str(t))
            DF.write(RSACipherObject.decrypt(i))
            t += 1


if __name__ == '__main__':
    MainTimeStart = time.time()
    RSACipherObject = RSA_Cipher()
    # Generate Keys
    # RSACipherObject.generate_key(1024)
    # RSACipherObject.save_key()
    print("Generating Key | ", end='')
    Mark = time.time()
    RSACipherObject.generate_key(1024)
    print("Key Generation Runtime = " + str(time.time() - Mark))

    print("Encrypting File | ", end='')
    Mark = time.time()
    encryptD2file()
    print("Encryption Runtime = " + str(time.time() - Mark))

    print("Decrypting File | ", end='')
    Mark = time.time()
    decryptD2file()
    print("Decryption Runtime = " + str(time.time() - Mark))

    print("Integrity Check = ", end='')
    TarHash = hashlib.md5(open('DecD2File',
'rb').read()).hexdigest()
    OrgHash = hashlib.md5(open('DataFile2',
'rb').read()).hexdigest()
    if TarHash == OrgHash:
        print("PASS! [MD5 : " + OrgHash + "]")
    else:
        print("Fail! [MD5 : " + OrgHash + "]")
```

```
    MainTimeEnd = time.time()
    print("Total Time = " + str(MainTimeEnd - MainTimeStart))
```

AES_CBC.py

```
import base64
import hashlib
import time
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import os
from cryptography.fernet import Fernet
import random

DataList = ['B', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U',
           'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
           'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '0',
'1', '2', '3', '4', '5', '6', '7', '8', '9']


def genrandpass(n):
    passw = ""
    for i in range(n):
        passw += (random.choice(DataList))
    return passw


def genrandomkey():
    password_provided = genrandpass(40)
    password = password_provided.encode()  # Convert to type bytes
    salt = os.urandom(16)
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )
    key = base64.urlsafe_b64encode(kdf.derive(password))  # Can only
use kdf once
    # print(key.decode())
    with open("AES_KEY", 'w') as ak:
```

```
        ak.write(key.decode())


def encryptAESDF2():
    with open('AES_KEY', 'r') as DF2:
        key = DF2.readline()

    with open('DataFile2', 'r') as DF2:
        dat = DF2.readline()

    message = dat.encode()

    f = Fernet(key.encode())
    encrypted = f.encrypt(message)  # Encrypt the bytes. The
returning object is of type bytes
    with open("EncAESD2File", 'w') as ef:
        ef.write(encrypted.decode())


def decryptAESD2F():
    with open('AES_KEY', 'r') as DF2:
        key = DF2.readline()

    with open('EncAESD2File') as ef:
        encr = ef.read()

    f = Fernet(key.encode())
    decrypted = f.decrypt(encr.encode())
    with open('DecAESD2File', 'w') as df:
        df.write(decrypted.decode())


if __name__ == '__main__':
    MainTimeStart = time.time()
    print("Generating Key | ", end='')
    Mark = time.time()
    genrandomkey()
    print("Key Generation Runtime = " + str(time.time() - Mark))

    print("Encrypting File | ", end='')
    Mark = time.time()
    encryptAESDF2()
    print("Encryption Runtime = " + str(time.time() - Mark))
    print("Decrypting File | ", end='')
    Mark = time.time()
    decryptAESD2F()
    print("Decryption Runtime = " + str(time.time() - Mark))

    print("Integrity Check = ", end='')
```

```
    TarHash = hashlib.md5(open('DecAESD2File',
'rb').read()).hexdigest()
    OrgHash = hashlib.md5(open('DataFile2',
'rb').read()).hexdigest()
    if TarHash == OrgHash:
        print("PASS! [MD5 : " + OrgHash + "]")
    else:
        print("Fail! [MD5 : " + OrgHash + "]")
    MainTimeEnd = time.time()
    print("Total Time = " + str(MainTimeEnd - MainTimeStart))
```

AES_CBC_RSA.py

```
import base64
import hashlib
from Crypto.Cipher import PKCS1_v1_5
from Crypto.PublicKey import RSA
from Crypto.Random import new as Random
from base64 import b64encode
from base64 import b64decode
import time
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import PBKDF2HMAC
import os
from cryptography.fernet import Fernet
import random

global global_aes_key
DataList = ['B', 'A', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U',
            'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f',
'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p',
            'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '0',
'1', '2', '3', '4', '5', '6', '7', '8', '9']


class RSA_Cipher:
    def generate_key(self, key_length):
        assert key_length in [1024, 2048, 4096]
        rng = Random().read
        self.key = RSA.generate(key_length, rng)

    def save_key(self):
        PK = self.key
        with open("RSAPrivateKeyFile", 'wb') as RKF:
```

```
            RKF.write(PK.export_key('PEM'))

    def load_key(self):
        with open("RSAPrivateKeyFile", 'rb') as PKF:
            self.key = RSA.import_key(PKF.read())

    def encrypt(self, data):
        plaintext = b64encode(data.encode())
        rsa_encryption_cipher = PKCS1_v1_5.new(self.key)
        ciphertext = rsa_encryption_cipher.encrypt(plaintext)
        return b64encode(ciphertext).decode()

    def decrypt(self, data):
        ciphertext = b64decode(data.encode())
        rsa_decryption_cipher = PKCS1_v1_5.new(self.key)
        plaintext = rsa_decryption_cipher.decrypt(ciphertext, 16)
        return b64decode(plaintext).decode()


def genrandpass(n):
    passw = ""
    for i in range(n):
        passw += (random.choice(DataList))
    return passw


def genrandomkey(ret=0):
    global global_aes_key
    password_provided = genrandpass(40)
    password = password_provided.encode()  # Convert to type bytes
    salt = os.urandom(16)
    kdf = PBKDF2HMAC(
        algorithm=hashes.SHA256(),
        length=32,
        salt=salt,
        iterations=100000,
        backend=default_backend()
    )
    key = base64.urlsafe_b64encode(kdf.derive(password))  # Can only
use kdf once
    # print(key.decode())
    with open("AES_KEY", 'w') as ak:
        ak.write(key.decode())
    if ret != 0:
        global_aes_key = str(key.decode())


def encryptAESDF2():
    with open('AES_KEY', 'r') as DF2:
```

```
        key = DF2.readline()

    with open('DataFile2', 'r') as DF2:
        dat = DF2.readline()

    message = dat.encode()

    f = Fernet(key.encode())
    encrypted = f.encrypt(message)  # Encrypt the bytes. The
returning object is of type bytes
    with open("EncAESD2File", 'w') as ef:
        ef.write(encrypted.decode())


def decryptAESD2F():
    with open('AES_KEY', 'r') as DF2:
        key = DF2.readline()

    with open('EncAESD2File') as ef:
        encr = ef.read()

    f = Fernet(key.encode())
    decrypted = f.decrypt(encr.encode())
    with open('DecAESD2File', 'w') as df:
        df.write(decrypted.decode())


if __name__ == '__main__':
    global global_aes_key
    MainTimeStart = time.time()
    RSACipherObject = RSA_Cipher()
    print("Generating RSA Key | ", end='')
    Mark = time.time()
    RSACipherObject.generate_key(1024)
    print("RSA Key Generation Runtime = " + str(time.time() - Mark))

    print("Generating AES Key | ", end='')
    Mark = time.time()
    genrandomkey(1)
    print("AES Key Generation Runtime = " + str(time.time() - Mark))

    print("Encrypting AES KEYS | ", end='')
    Mark = time.time()
    AES_RSA_KEY = RSACipherObject.encrypt(global_aes_key)
    print("KEY Encryption Runtime = " + str(time.time() - Mark))

    print("Encrypting File | ", end='')
    Mark = time.time()
    encryptAESDF2()
```

```
    print("Encryption Runtime = " + str(time.time() - Mark))
    print("Decrypting AES Key | ", end='')
    Mark = time.time()
    RSACipherObject.decrypt(AES_RSA_KEY)
    print("Key Decryption Runtime = " + str(time.time() - Mark))

    print("Decrypting File | ", end='')
    Mark = time.time()
    decryptAESD2F()
    print("File Decryption Runtime = " + str(time.time() - Mark))

    print("Integrity Check = ", end='')
    TarHash = hashlib.md5(open('DecAESD2File',
'rb').read()).hexdigest()
    OrgHash = hashlib.md5(open('DataFile2',
'rb').read()).hexdigest()
    if TarHash == OrgHash:
        print("PASS! [MD5 : " + OrgHash + "]")
    else:
        print("Fail! [MD5 : " + OrgHash + "]")
    MainTimeEnd = time.time()
    print("Total Time = " + str(MainTimeEnd - MainTimeStart))
```

RESULT

RSA Only:

```
"C:\Users\Saket Upadhyay\PycharmProjects\PGPStrategy\Scripts\python.exe"
Generating Key | Key Generation Runtime = 0.25003695487976074
Encrypting File | Encryption Runtime = 6.371272325515747
Decrypting File | Decryption Runtime = 27.22977614402771
Integrity Check = PASS! [MD5 : b3aa4393381491672fb0a3614972c3f2]
Total Time = 33.85108542442322

Process finished with exit code 0
```

Figure 1 : RSA only bulk encryption clocked at 33.85 seconds.

AES_CBC Only:

```
"C:\Users\Saket Upadhyay\PycharmProjects\PGPStrategy\Scripts\python.exe"
Generating Key | Key Generation Runtime = 0.24996113777160645
Encrypting File | Encryption Runtime = 0.04687643051147461
Decrypting File | Decryption Runtime = 0.031253814697265625
Integrity Check = PASS! [MD5 : b3aa4393381491672fb0a3614972c3f2]
Total Time = 0.3280913829803467

Process finished with exit code 0
```

Figure 2 : AES only file encryption clocked at 0.32 seconds.

AES_CBC + RSA :

```
"C:\Users\Saket Upadhyay\PycharmProjects\PGPStrategy\Scripts\python.exe"
Generating RSA Key | RSA Key Generation Runtime = 0.7656674385070801
Generating AES Key | AES Key Generation Runtime = 0.2531397342681885
Encrypting AES KEYS | KEY Encryption Runtime = 0.0
Encrypting File | Encryption Runtime = 0.048425912857055664
Decrypting AES Key | Key Decryption Runtime = 0.0
Decrypting File | File Decryption Runtime = 0.03125786781311035
Integrity Check = PASS! [MD5 : b3aa4393381491672fb0a3614972c3f2]
Total Time = 1.0984909534454346

Process finished with exit code 0
```

Figure 3 : AES+RSA bulk+key encryption clocked at 1.09 seconds

Mean Table:

$$Avg.Runtime = \frac{\sum_{i=1}^{5} RTi}{5}$$

Approach	RT 1	RT 2	RT 3	RT 4	RT 5	Avg. Runtime	Security Confidence†
RSA	34.53 s	33.28 s	34.20 s	34.81 s	34.89 s	34.34 s	High
AES	0.32 s	0.33 s	0.31 s	0.32 s	0.32 s	0.32 s	Low
AES+RSA	0.47 s	0.56 s	1.06 s	0.98 s	0.77 s	0.76 s	Medium - High

*RT=Run Time | † Theoretical Confidence

CONCLUSION

From the above results, we can see that **AES+RSA** approach is best for time efficiency and security. As it provides faster operation through AES_CBC 128 bits on bulk files, and enhanced security by encrypting AES's key with RSA 1024 bit key.

This is the reason why PGP is efficient, it encrypts the large bulk of data using IDEA symmetric encryption and IDEA's key is encrypted with RSA X bits [$X \in \{1024,2048,4096\}$].