**CPS3232 Applied Cryptography: Assignment**



Name: JasmineXuereb

ID: 31997(G)

Course: BSc Mathematics and Computer Science

Table of Contents

[Introduction 3](#_Toc534293639)

[Task 1 - Block cipher modes and message authenticity 3](#_Toc534293640)

[Question a 3](#_Toc534293641)

[Question b 4](#_Toc534293642)

[Question c 6](#_Toc534293643)

[Question d 7](#_Toc534293644)

[Task 2 8](#_Toc534293645)

# Introduction

This project was implemented using the high-level general-purpose programming language *Python*.

# Task 1 – Block cipher modes and message authenticity

## Question a

A secure communication system that makes use of an AES library implementation using ECB mode was implemented. In order to do this, *Python*’s package *PyCryptodome* was used. The key is a randomly generated 16-byte string. It is assumed that the key was securely shared beforehand. Two functions were then implemented; *encrypt\_message\_EBC()* and *decrypt\_message\_ECB()*, where the first function is responsible for encrypting the message passed as parameter and returning a base-16 textual representation of the resulting ciphertext. The second function takes a key and a ciphertext as parameters and returns its plaintext.

import json

from base64 import b16encode

from base64 import b16decode

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad

from Crypto.Util.Padding import unpad

from Crypto.Random import get\_random\_bytes

data = b"Accnt:0123456789Accnt:9876543210Descr:1111111111111111111111111111111111111111111111111111111111Amount:123456789"

key = get\_random\_bytes(16)

#encrypt in ECB mode

def encrypt\_message\_ECB(key, plaintext):

cipher = AES.new(key, AES.MODE\_ECB)

ciphertext\_bytes = cipher.encrypt(pad(plaintext, AES.block\_size))

ciphertext = b16encode(ciphertext\_bytes).decode('utf-8')

return ciphertext

#it is assumed that the key was securely shared beforehand

def decrypt\_message\_ECB(key, ciphertext):

try:

ciphertext = b16decode(ciphertext) #encode the ciphertext using Base16

cipher = AES.new(key, AES.MODE\_ECB)

plaintext = unpad(cipher.decrypt(ciphertext), AES.block\_size)

return plaintext

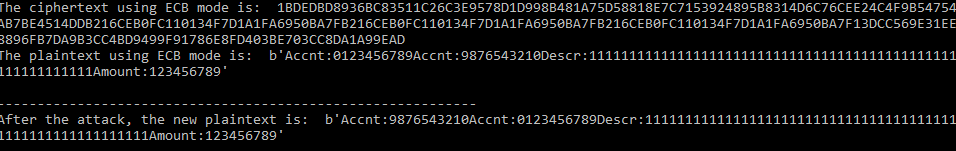
except Exception as e:

print("Incorrect decryption")

Then, once, the required system was implemented, the weakness of this deterministic approach was demonstrated through an attack that reverses the direction of the transfer. This was done by simply swapping the first and the second block. The code used to do this is shown below.

attacked\_ciphertext=''.join([ciphertext[32:64],ciphertext[0:32],ciphertext[64:]]) attacked\_plaintext = decrypt\_message\_ECB(key, attacked\_ciphertext)

After the attack was carried out, by modifying the ciphertext, the plaintext was also modified in a way that the direction of the transfer was reversed, with no visible signs that the ciphertext was modified as shown in the resulting output below!



## Question b

The AES system was modified by changing the mode from ECB to CBC. Even though this mode is stronger than ECB, it is till prone to similar exploits, more specifically it is prone to an attack that modifies that the initial figures of the transferred amount while still producing valid format field labels.

The decryption process in CBC mode of the first ciphertext block and the ith ciphertext block is performed by using the following two equations respectively.

Now, if the position of the target byte is known, then the ciphertext can be modified by modifying the corresponding ciphertext in the previous ciphertext block.

Suppose the amount to be transferred from one account to another is 123456789. Even though this amount is kept secret, if the first byte, which is 1, is known, then the ciphertext can be easily attacked in a way that the new plaintext after decryption shows the amount 923456789 instead!

The ASCII value of 1 is 0x31, while the ASCII value of 9 is 0x39. The corresponding byte of the previous ciphertext block has to be modified by XORing it with a new byte, *m*. *m* is obtained as following: where 0x31 is the actual plaintext hexadecimal value and 0x39 is the new hexadecimal value we want to change it to. After calculating *m*, bit flipping is performed, where the ciphertext is modified.

The following is the code used in order to be able to perform this attack.

#xor function

def xor(A, B):

return hex(int(A, 16) ^ int(B, 16))[2:].upper() #[2:] to remove 0x

def attack\_CBC(data):

b16 = json.loads(data)

ciphertext = b16['ct']

iv = b16['iv']

block\_to\_alter = ciphertext[160:192] #prev block to one containing amount

byte\_to\_change = block\_to\_alter[14:16] #start of amount value

#xor 1st bit of plaintext with the hex value you want in new plaintext

m = xor('0x31', '0x39') #0x31 is decimal 1, 0x39 is decimal 9

#perform bit flipping

new\_byte = xor(hex(int(m, 16)), hex(int(byte\_to\_change, 16)))

while len(new\_byte) < 2:

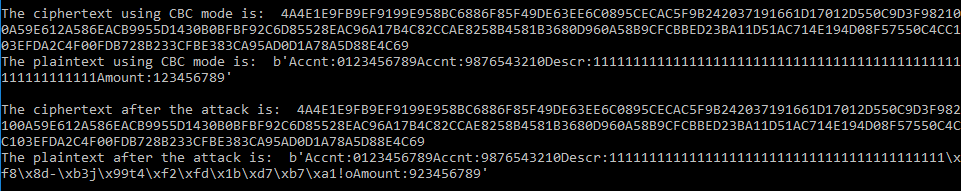
new\_byte = "".join(['0', new\_byte])

new\_ciphertext = ''.join([ciphertext[0:174], new\_byte, ciphertext[176:]])

result = json.dumps({'iv':iv, 'ct':new\_ciphertext})

return result

The following shows how the ciphertext and the plaintext before and after the attack.



Even though the amount was changed successfully, it can be noted that the previous plaintext block is garbage. This happens due to the fact that when flipping a byte in the ciphertext block Ci, not only the corresponding byte of the next plaintext block Pi+1 is affected, but also the corresponding **full** plaintext block which has the same index as the modified ciphertext block.

## Question c

The previous attack was possible since in CBC mode there is not message integrity. Hence, the system was hardened by providing transaction integrity. In order to protect the system from both content tampering and replay, *HMAC* was used alongside the CBC mode. *HMAC*, which stands for Hashed Message Authentication Code, works by hashing together the key with the message. *Python*’s *hmac* and *hashlib* packages were used for this task.

In the function that encrypts the plaintext, the plaintext passed as parameter is encrypted with AES in CBC mode and then it is signed with HMAC, using SHA256 as the cryptographic hash function. Then, when decrypting the ciphertext, before even trying to decrypt it, the HMAC-SHA256 signature is verified so as to check whether the ciphertext has been corrupted. If the signature is correct, then decryption is carried out, otherwise an authentication error is raised.

class AuthenticationError(Exception): pass

def compare\_mac(a, b):

a = a[1:] #since is byte string, start from second element

b = b[1:]

different = 0

#Compare using xor to mitigate timing attacks

for x, y in zip(a, b):

different |= x ^ y

return different == 0

#encrypt data with AES-CBC and sign it with HMAC-SHA256

def encrypt\_message\_CBC\_HMAC(plaintext, shared\_key, hmac\_key):

plaintext = pad(plaintext, AES.block\_size)

iv\_bytes = get\_random\_bytes(AES.block\_size)

cypher = AES.new(shared\_key, AES.MODE\_CBC, iv\_bytes)

encrypted\_data = cypher.encrypt(plaintext)

iv = iv\_bytes + encrypted\_data #secret prefix mac

signature = hmac.new(hmac\_key, iv, hashlib.sha256).digest()

return (encrypted\_data, iv\_bytes, signature)

#verify HMAC-SHA256 signature and decrypt data with AES-CBC

def decrypt\_message\_CBC\_HMAC(encrypted\_data, iv\_bytes, signature, shared\_key, hmac\_key):

iv = iv\_bytes + encrypted\_data

new\_hmac = hmac.new(hmac\_key, iv, hashlib.sha256).digest()

if not compare\_mac(new\_hmac, signature):

print("Incorrect decryption")

#raise AuthenticationError("message authentication failed")

cypher = AES.new(shared\_key, AES.MODE\_CBC, iv\_bytes)

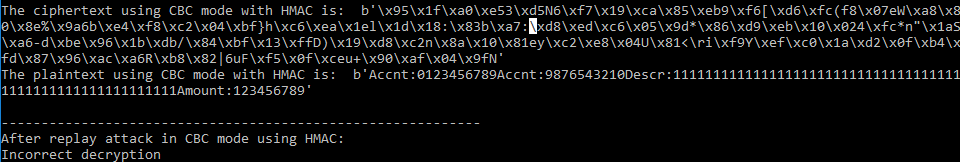
plaintext = cypher.decrypt(encrypted\_data)

return unpad(plaintext, AES.block\_size)

## Question d

Unlike in CBC mode without any message integrity, CBC mode with HMAC is not prone to attacks where the initial figures of the transferred amounts are modified, also known as tempering attacks, or attacks that reverse the direction of the transfer, known as replat attacks.

In order to demonstrate the strength of HMAC, a replay attack in CBC using HMAC was implemented, producing the result shown in the figure below. This shows that once the ciphertext had been corrupted, when trying to decrypt it, an authentication error was immediately raised!



Similarly, a tamper attack in CBC using HMAC was also implemented, producing the result shown below when trying to decrypt the corrupted cyphertext.



# Task 2 – Transparent access control on the Blockchain

## Question a

The required reference monitor was implemented. For this task *Python*’s package *PyCryptodome* was also used. This package was used to generate the *RSA modulus*. The generated key, using the function *RSA.generate()*, was then saved to a file protected by a password, simulating the fact that only Rene has access to it. Rene can set *v* by simply using the equation . A shared secret *s*, which is an element of integer ring is then generated. Secret *s* is then sent to both Alice and Bob over an out-of-band secure channel.

from Crypto.PublicKey import RSA

import random

import math

from sympy import mod\_inverse

#setup FS

#generate a new RSA key pair and save it to a file, protected by a password

password = "mypassword"

key = RSA.generate(2048)

#scrypt key derivation function is used to thwart dictionary attacks

encrypted\_key = key.export\_key(passphrase=password, pkcs=8, protection="scryptAndAES128-CBC")

f = open("rsa\_key.bin", "wb") #save key to a file that can only be accessed by Rene

f.write(encrypted\_key)

n = key.n #n is public knowledge

print("Public Key: ", n)

#generate shared secret s where s is an element of the integer ring

s = random.randint(0, key.n-1)

print("Secret: ", s)

#Assume this is sent to Alice and Bob over a secure channel

#Delete secrets

del key

del password

#this is done by Rene - reads the private RSA key back from the rsa\_key.bin file which is protected by a password

encoded\_key = open("rsa\_key.bin", "rb").read()

key = RSA.import\_key(encoded\_key, passphrase='mypassword') #parties know the password

n = key.n #n is the public key

v = (s\*\*2)%n #v = s^2 mod n

Once this was completed, the interactive protocol was implemented, taking t as 100 – thus the interactive protocol is repeated 100 times with a successful attack probability of 2-100. A function *perform\_attack()* was implemented, which is responsible for generating a random number between 0 and 2-t so as to determine whether the attack was successful or not.

#attack with successful attack probability of 2^-t

def perform\_attack(t):

successful\_attack\_prob = 2 \*\* (-t)

attack = random.randint(0,2\*\*t)

if(attack < successful\_attack\_prob):

return 1

else:

return 0

#interactive protocol is repeated t times

t = 100

for i in range(t):

#Commitment Phase - Alice randomly picks r from integer ring

r = random.randint(0,n-1)

x = (r\*\*2)%n #x = s^2 mod n

#Alice sends x to Rene

#Challenge Phase - Rene randomly picks e from {0,1} and sends it to Alice

e = random.randint(0,1)

if (perform\_attack(t) == 1): #attack is successful

s = random.randint(0,n-1) #bad secret key

#Resoponse Phase - Alice replies with y = r s^2 mod n and sends y to Rene

y = (r\*(s\*\*e)) % n

#Rene verifies y^2 = x v^e mod n

#terminate and negate access if otherwise

if (y\*\*2 % n != (x\*(v\*\*e))%n): #y^2 mod n since we are in integer ring Z\_n

print('Attack!')

break

else:

continue

## 

## Question b

The implemented reference monitor was weakened in a way that Alice always commits to the same *r*. Since *r* is kept fixed, Oscar can simply intercept *y* when *e* is equal to 0, from which he can easily calculate *r* where *r = y mod n*. Now, once Oscar has knowledge of *r*, he can simply find *s* by intercepting *y* when *e* is equal to 1 by using the equation *s = y (r mod n)-1*. The following is the code for the modified interactive protocol.

As a matter of fact, such an implementation is very prone to attacks that disclose the secret *s* as shown in the figure below.

r\_obtained = 0 #0 if r is not known, 1 if r is known

t = 100

#Alice randomly picks r from integer ring and keeps it fixed

fixed\_r = random.randint(0,n-1)

for i in range(t):

#Alice computes x and sends it to Rene

x = (fixed\_r\*\*2)%n

#Challenge Phase - Rene randomly picks e from {0,1} and sends it to Alice

e = random.randint(0,1)

#Oscar can intercept e since Renee sends it to Alice via an open channel

#Resoponse Phase - Alice replies with y = r s^2 mod n and sends y to Rene via an open channel - Oscar can intercept y

y = (fixed\_r\*(s\*\*e)) % n

#Oscar's attack

if e==0 and r\_obtained == 0: #when e = 0, Alice sends y = r mod n as response

r\_found = y % n

r\_obtained = 1

if e==1 and r\_obtained == 1:

#given n, r, y are all known, Oscar can find s by s = y(r mod n)^(-1)

s\_found = (y \* mod\_inverse(r\_found, n)) % n

print("The shared secret has been discovered! It is ", s)

break

#Rene verifies y^2 = x v^e mod n

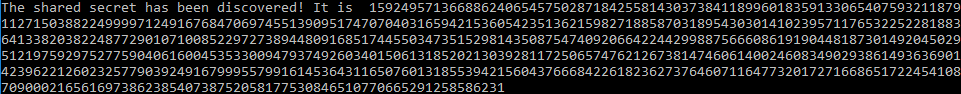
if (y\*\*2 % n != (x\*(v\*\*e))%n):

print('Attack!')

break

else:

continue



## Question c

The initial implementation of the implemented reference monitor was weakened further as specified.