CPS3232: Applied Cryptography

Hands-on Crypto

Marc Ferriggi

Supervised by Dr Mark Vella

Department of Computer Science
Faculty of ICT
University of Malta

January, 2019

A assignment submitted in partial fulfilment of the requirements for the degree of B.Sc. (hons.) Conputer Science AND Statistics and Operations Research.

Statement of Originality

I, the undersigned, declare that this is my own work unless where otherwise acknowledged and referenced.

Candidate Marc Ferriggi

Signed ____________________

Date January 10, 2019

Contents

List of Figures

List of Tables

1

Block Cipher Modes and Message Authenticity

1.1 Introduction

In this task, a secure communication system will be modelled that makes use of an AES library for an e-Banking application for money transfers. The text-based protocol that's used by this e-Banking company has the following message format:

`Accnt:`*[10 bytes]*`Accnt:`*[10 bytes]*`Descr:`*[58 bytes]*`Amount:`*[9 bytes]*

The following libraries and functions were imported in order to run all the code presented in this chapter:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
from Crypto.Util.Padding import unpad
from Crypto.Random import get_random_bytes
from base64 import b16encode
from base64 import b16decode
import os
import json
import hmac
import hashlib
```

The required data was then initialized as follows:

```
#Initialize Data
data = b"Accnt:0003336669Accnt:0123456789Descr:00000000000000
              00000000000000000000000000000000000000000000A
```

```
                mount:023456789"
key = get_random_bytes(16)
print(b16encode(key).decode('utf-8'))
```

1.2 ECB Mode Implementation

The first task was to implement the e-Banking system using AES in ECB mode and demonstrate the weakness of using this approach through an attack which swaps the accounts thus reversing the direction of the transfer.

The data was encrypted in ECB mode using the following code:

```
ECB_cipher = AES.new(key, AES.MODE_ECB) #Create a new AES Cypher
 #encrypt the padded data
ciphertext_bytes = ECB_cipher.encrypt(pad(data, AES.block_size))
#Encode the cyphertext to 16 bit encoding to have a more clear output
#use the decode function to remove the b'...' in the output
ciphertext = b16encode(ciphertext_bytes).decode('utf-8')
print(ciphertext)
```

Output: 4CDEB5515885FEC7390A48D4F9747BD53A309C3CA17838A5A1FD18A9A433737E9 D35FE123067946375B9B59FAD318E1CE9C1DC732606DAA7644898E4D577B90DE9C1DC732606 DAA7644898E4D577B90DE9C1DC732606DAA7644898E4D577B90DBD18D9394E2B939E4EA7D9E 8C14C0B04FAF69C2C9877EE7A8805287F5B34C232

The ciphertext was then immediately decrypted to ensure it was encrypted successfully:

```
decoded_ciphertext = b16decode(ciphertext)
decrypted_text = unpad(ECB_cipher.decrypt(decoded_ciphertext),
                                AES.block_size)
print(decrypted_text)
```

Output: b'Accnt:0003336669Accnt:0123456789Descr:00000000000000000000000000000000 0000000000000000000000000Amount:023456789'

The attack was then demonstrated by Oscar by simply swapping the desired bits of the ciphertext, and once Bob decodes the message he gets a valid transaction and is thus none the wiser that the accounts have been swapped.

```
#Swap the account numbers to reverse the direction of the transfer
modified_ciphertext = ''.join([ciphertext[32:64],
```

```
        ciphertext[0:32], ciphertext[64:]])
#decrypt message:
modified_result = unpad(ECB_cipher.decrypt(b16decode
        (modified_ciphertext)), AES.block_size)
print(modified_result.decode('utf-8'))
```

Output: Accnt:0123456789Accnt:0003336669Descr:0000000000000000000000000000000000000000000000000000000000Amount:023456789

1.3 CBC Mode Implementation

The encryption was then implemented using CBC mode and another attack was demonstrated by modifying the amount since the implementation does not verify for valid format field labels.

The data was encrypted in CBC mode using the following code:

```
iv = os.urandom(16) #initializing vector
#Create new AEC Cipher in CBC mode
CBC_cipher = AES.new(key, AES.MODE_CBC,iv)
#Encrypt padded data
ciphertext_bytes = CBC_cipher.encrypt(pad(data, AES.block_size))
#Encode the cyphertext to 16 bit encoding to have a more clear output
#use the decode function to remove the b'...' in the output
CBC_ciphertext = b16encode(ciphertext_bytes).decode('utf-8')
#Save as Json in order to simulate passing the whole thing over the network
json_data = json.dumps({'iv':b16encode(iv).decode('utf-8'),
         'ciphertext':CBC_ciphertext})
```

The ciphertext was again immediately decrypted to ensure it was encrypted successfully:

```
#Read the data from the json file
b16 = json.loads(json_data)
iv = b16decode(b16['iv'])
#Create a new Cipher for decryption using the same key and iv
CBC_cipher2 = AES.new(key, AES.MODE_CBC, iv)
#Decode, unpad and display the results
CBC_dec_ciphertext = b16decode(b16['ciphertext'])
CBC_decoded_plaintext = unpad(CBC_cipher2.decrypt(
        CBC_dec_ciphertext),AES.block_size)
print(CBC_decoded_plaintext)
```

Output: b'Accnt:0003336669Accnt:0123456789Descr:000000000000000000000000000000000000000000000000000000000Amount:023456789'

An XOR function was then defined to be used in the attack:

```
def xor(A, B):
        return hex(int(A, 16) ^ int(B, 16))[2:].upper()
```

The attack as explained previously was then demonstrated as follows:

```
#Attacker can also reda the data from the json file
Oscar_b16 = json.loads(json_data)
Oscar_ciphertext = Oscar_b16['ciphertext']
Oscar_iv = Oscar_b16['iv']
block_to_alter = Oscar_ciphertext[160:192] #block containing amount
byte_to_change = block_to_alter[14:16] #amount

#xor 1st bit of plaintext with the hex value you want in new plaintext
m = xor('0x30', '0x39') #0x30 is decimal 0, 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])
attacked_ciphertext = ''.join([Oscar_ciphertext[0:174],
                 new_byte, Oscar_ciphertext[176:]])

result = json.dumps({'iv':Oscar_iv, 'ciphertext':attacked_ciphertext})
```

Finally, the attack was decoded by Bob and as can be seen in the output, the amount was successfully altered.

```
#Create a new Cipher for decryption using the same key and iv
CBC_cipher_Oscar = AES.new(key, AES.MODE_CBC,b16decode(Oscar_iv))
#Decode, unpad and display the results
CBC_dec_ciphertext = b16decode(attacked_ciphertext)
CBC_decoded_plaintext = unpad(CBC_cipher_Oscar.decrypt(
                CBC_dec_ciphertext),AES.block_size)
print(CBC_decoded_plaintext)
```

*Output:*b'Accnt:0003336669Accnt:0123456789Descr:00000000000000000000000000000000000000000\xbbs\x14\x81\x92xC\x975\xbfb\xdf\xaa\\\xc8DAmount:923456789'

1.4 Hardened Implementation

The system was then hardened in order to provide transaction integrity. Message authentication codes were used to protect from content tampering and a challenge-response was st up to protect from replay attacks.

Initially, a function was set up to compare the MACs:

```
def compare_mac(a, b):
        a = a[1:]
        b = b[1:]
        different = 0

        for x, y in zip(a, b):
                different |= x ^ y
        return different == 0
```

The data was then encrypted in CBC mode but with added MAC using HMAC-SHA256 standard.

```
hmac_key = get_random_bytes(16)
plaintext = pad(data, AES.block_size)
iv_bytes = get_random_bytes(AES.block_size)
HMAC_cipher = AES.new(key, AES.MODE_CBC, iv_bytes)
encrypted_data = HMAC_cipher.encrypt(plaintext)
iv = iv_bytes + encrypted_data #secret prefix mac
signature = hmac.new(hmac_key, iv, hashlib.sha256).digest()
```

Finally, the signature was verified and the data decrypted if it wasn't tampered with.

```
new_hmac = hmac.new(hmac_key, iv, hashlib.sha256).digest()
if not compare_mac(new_hmac, signature):
print("Incorrect decryption")
cipher = AES.new(key, AES.MODE_CBC, iv_bytes)
dec_plaintext = cipher.decrypt(encrypted_data)
print(unpad(dec_plaintext, AES.block_size).decode('utf-8'))
```

Output: Accnt:0003336669Accnt:0123456789Descr:0000000000000000000000000000000000000000000000000000000Amount:023456789

1.5 Demonstrating Failed Attacks

Two failed attacks were then demonstrated to show the new system's security. The first attack attempts to tamper the data, while the second failed attack attempts to perform a replay attack.

The attack function attempts the data tampering similar to the first attack shown when the data was encrypted in CBC mode.

```
#Like Task B's attack
def attack(data):
        b16 = json.loads(data)
        ciphertext = b16['ct']
        iv = b16['iv']
        block_to_alter = ciphertext[160:192]
        byte_to_change = block_to_alter[14:16]

        m = xor('0x30', '0x39')

        #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 *decrypt_message* function performs the required checks before decrypting the data in order to ensure the data hasn't been tampered with, similar to the decryption step shown in the previous section.

```
#Like Task C's Decryption
def decrypt_message(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("Attack!")
```

```
        cypher = AES.new(shared_key, AES.MODE_CBC, iv_bytes)
        plaintext = cypher.decrypt(encrypted_data)
        return unpad(plaintext, AES.block_size)
```

The tamper attack was done as follows:

```
iv = b16encode(iv_bytes).decode('utf-8')
ciphertext = b16encode(encrypted_data).decode('utf-8')
ct_json = json.dumps({'iv': iv, 'ct': ciphertext})
altered_ciphertext = attack(ct_json)
altered_ciphertext = b16decode(json.loads(altered_ciphertext)['ct'])
decrypt_message(altered_ciphertext, iv_bytes, signature, key, hmac_key)
```

Output: Attack!

Finally, the replay attack was also demonstrated:

```
altered_ciphertext = b"".join([encrypted_data[32:64],
           encrypted_data[0:32], encrypted_data[64:]])
decrypt_message(altered_ciphertext, iv_bytes, signature,
           key, hmac_key)
```

Output: Attack!

2

Transparent Access Control on the Blockchain

2.1 Introduction

In this task, a reference monitor was implemented where access request decisions took the form of Blockchain transactions. In particular, n parties were provided access to a controlled resource through a shared secret s. Fiat-Shamir's Zero Knowledge Proof was used in order not to disclose the shared secret s.

2.2 Implementing the Reference Monitor

The reference monitor was implemented in Jupyter Notebook. The RSA modulus was generated using PyCryptodome's *RSA.generate* function, where the number of bits was set to 2048 as this was deemed a sufficient length for 2017 and the public exponent e was set to be 65537 as this is the FIPS standard. The generated key (which contains the RSA modulus $n = p.q$) was then saved to a file protected by a password in order to simulate that only Rene the reference monitor has access to it.

```
%reset
from Crypto.PublicKey import RSA
import random

#Setting up FS
#Generate n:
key = RSA.generate(2048,e=65537)
#nbits = 2048 (Sufficient length for 2017), e -> FIPS Standard
#save the secure key to a file to only be accessed by Rene
passphrase = 'Str0ngPassw0rd!%'
```

```
f = open('mykey.pem','wb')
#Use a passphrase since
f.write(key.exportKey('PEM',passphrase=passphrase))
f.close()
n = key.n #assume n is public knowledge

#Rene will then generate s, which lies in integer ring (Z_n)
s = random.randint(1,key.n+1)
#Assume this is sent to Alice and Bob over a secure channel

#Delete Secrets:
del key
del passphrase
#Authorised parties will then be given mykey.pem and
#with knowledge of the passphrase will have access
#to the secure key
```

Once the RSA key was set up, Rene then set v by using the equation $v = s^2 \bmod n$.

```
#Rene:
#get the key
f = open('mykey.pem','r')
Rene_key = RSA.importKey(f.read(),passphrase='StrOngPasswOrd!%')
f.close()
#set v
v = (s**2)%Rene_key.n
```

The interactive protocol was then implemented setting $t = 100$, thus having a successful attack probability of 2^{-100}.

```
t = 100
for i in range(t):
        #Interactive Protocol
        #Alice
        #pick random r
        r = random.randint(1,n+1)
        x = (r**2)%n #compute x
        #x is now sent to Rene

        #Rene
        e = random.randint(0,1)
```

```
        #e is sent to Alice

        #Alice
        y = (r*(s**e)) % n
        #y is sent to Rene

        #Rene:
        if (y**2 % n != (x*(v**e))%n):
                print('Attack!')
                break
```

2.3 Attack 1

The implementation was then weakened in a way so that Alice always commits to the same $r \in Z_n$. An attack that discloses s which hinges on the fact that whenever $e = 0$ Alice sends $y = r \; mod \; n$ as a response was then shown. Given the fact that r wasn't changing, Oscar could simply extract r by intercepting y when $e = 0$. Once Oscar has r then, when $e = 1$, Oscar intercepts y and can extract s by using the equation $s = y.r^{-1}$ $mod \; n$. The modified code to the previous implementation can be seen below.

```
from sympy import mod_inverse

gotR = 0

t = 100
#Alice will pick a random r and keep it fixed:
r = random.randint(1,n+1)
for i in range(t):
        #Interactive Protocol
        #Alice:
        x = (r**2)%n #compute x
        #x is now sent to Rene

        #Rene
        e = random.randint(0,1)
        #e is sent to Alice on open channel, Oscar can intercept this

        #Alice
        y = (r*(s**e)) % n
        #y is sent to Rene on an open channel, Oscar can intercept this
```

```
#Oscar's Attack:
if e==0 and gotR == 0:
        oscar_r = y
        gotR = 1
if e==1 and gotR == 1:
        oscar_s=(y*mod_inverse(r,n)) % n
        print(s==oscar_s)
        break

#Rene:
if (y**2 % n != (x*(v**e))%n):
        print('Attack!')
        break
```

Output: True

2.4 Attack 2

3

Setting up Certificate Authority (CA) trees and HTTPS servers.