Info

Dit document is geschreven door Tom Stuurman en Abe Bruinsma

Code + comentaar

Week 4 Crypto

Contents

[Code 1 – Base64.py 2](#_Toc161328800)

[Code 2 – Binary XOR.py 3](#_Toc161328801)

[Code 3 – ECB Mode AES.py 4](#_Toc161328802)

[3- Bonus 4](#_Toc161328803)

[Code 4 – CBC Mode AES.py 5](#_Toc161328804)

[Code 5 – Breaking ECB.py 7](#_Toc161328805)

# Code 1 – Base64.py

import base64

def string\_to\_b64(asciiString):

    # encode the ascii string to b64

    b64String = base64.b64encode(asciiString.encode("ascii"))

    # return the b64 string

    return b64String

# Laat deze asserts onaangetast!

assert type(string\_to\_b64("foo")) == bytes

assert string\_to\_b64("Hello World") == b'SGVsbG8gV29ybGQ='

def b64\_to\_string(b64String):

    # Decode the b64 string to ascii

    asciiString = base64.b64decode(b64String).decode("ascii")

    # give the ascii string back

    return asciiString

# Laat deze asserts onaangetast!

assert type(b64\_to\_string("SGVsbG8gV29ybGQ=")) == str

assert b64\_to\_string("SGVsbG8gV29ybGQ=") == "Hello World"

# Code 2 – Binary XOR.py

from base64 import b64encode

def fixed\_length\_xor(text, key):

    # The function fixed\_length\_xor performs a binary XOR operation on the text and the key, byte by byte.

    xor\_output = bytes([a ^ b for a, b in zip(text, key)])

    print(xor\_output)

    return xor\_output

# Leave these asserts untouched!

assert type(fixed\_length\_xor(b'foo',b'bar')) == bytes

assert b64encode(fixed\_length\_xor(b'foo',b'bar')) == b'BA4d'

def repeating\_key\_xor(text, key):

# The key is repeatedly multiplied until its length is equal to or greater than the length of the text.

# This is done by multiplying the key with the integer division of the length of the text by the length of the key, plus one.

# This ensures that the key is at least as long as the text.

# Then the key is cut to the length of the text, making it exactly as long as the text.

    key = (key \* (len(text) // len(key) + 1))[:len(text)]

    print(key)

# The function fixed\_length\_xor is then called with the text and the key.

# This function performs a binary XOR operation on the text and the key, byte by byte.

# The result of this operation is stored in the variable xor\_output.

    xor\_output = fixed\_length\_xor(text, key)

    print(xor\_output)

    return xor\_output

# Leave these asserts untouched!

assert type(repeating\_key\_xor(b'all too many words',b'bar')) == bytes

assert b64encode(repeating\_key\_xor(b'all too many words',b'bar'))\

   == b'Aw0eQhUdDUEfAw8LQhYdEAUB'

# Code 3 – ECB Mode AES.py

from base64 import b64decode

from Crypto.Cipher import AES

def ECB\_decrypt\_from\_file(filename, key, output\_filename):

    # Reads the cipher text from the file

    with open(filename, 'rb') as f:

        ciphertext = f.read()

    # A new AES-cipher is created with the key and the ECB mode

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

    # The cipher text is decoded and then decrypted with the cipher's decrypt function

    plaintext = cipher.decrypt(b64decode(ciphertext))

    # Write the plaintext to another file

    with open(output\_filename, 'w') as f:

        f.write(plaintext.decode('utf-8'))

# Use the function

ECB\_decrypt\_from\_file('file3.txt', b'SECRETSAREHIDDEN', 'output.txt')

## 3- Bonus

from Crypto.Cipher import AES

from base64 import b64encode

def ECB\_encrypt(plaintext, key):

    # Encrypt the plaintext with the key and the ECB mode

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

    # The ciphertext is created with the encrypt function of the cipher

    ciphertext = cipher.encrypt(plaintext)

    return ciphertext

# Example use of the ECB\_encrypt function:

plaintext = b"Hello, this is a secret message!"  # Example plaintext

key = b'SECRETSAREHIDDEN'  # The same key as in the decryption function

# Encrypt the plaintext

ciphertext = ECB\_encrypt(plaintext, key)

# Print the ciphertext in base64 encoded format

print(b64encode(ciphertext))

# Code 4 – CBC Mode AES.py

from base64 import b64decode

from Cryptodome.Cipher import AES

from Cryptodome.Util.Padding import unpad

from pyfiglet import Figlet

from colorama import Fore, init

def print\_big\_and\_green(text):

    init(autoreset=True)  # Initialize colorama for colored output

    f = Figlet(font='slant')  # Choose a font for the output

    print(Fore.GREEN + f.renderText(text))  # Print the text in green

def fixed\_length\_xor(block, xor\_with):

    # XOR each byte of the block with the corresponding byte of xor\_with

    return bytes([a ^ b for a, b in zip(block, xor\_with)])

def ECB\_decrypt(ciphertext, key):

    # A variable is created where the cipher is decrypted

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

    return cipher.decrypt(ciphertext)

def CBC\_decrypt(ciphertext, key, IV):

    # Split the ciphertext into blocks of 16 bytes each

    blocks = [ciphertext[i:i+16] for i in range(0, len(ciphertext), 16)]

    # Create a list to store the plaintext blocks

    plaintext\_blocks = []

    # Loop through the blocks

    for i in range(len(blocks)):

        # Decrypt the block using ECB mode

        decrypted = ECB\_decrypt(blocks[i], key)

        # If it is the first block

        if i == 0:

            # XOR the decrypted block with the IV (Initialization Vector)

            plaintext\_blocks.append(fixed\_length\_xor(decrypted, IV))

        else:

            # XOR the decrypted block with the previous ciphertext block

            plaintext\_blocks.append(fixed\_length\_xor(decrypted, blocks[i-1]))

    # Join the blocks together to get the complete plaintext

    plaintext = b''.join(plaintext\_blocks)

    return plaintext

# Leave this block of code untouched & at the bottom of your code!

a\_ciphertext = b64decode('e8Fa/QnddxdVd4dsL7pHbnuZvRa4OwkGXKUvLPoc8ew=')

a\_key = b'SECRETSAREHIDDEN'

a\_IV = b'WE KNOW THE GAME'

def clear\_screen():

    # Clears the terminal for a clean output

    print("\033c", end="")

def main():

    clear\_screen()

    # In the main function, the functions are called and the results are printed

    plaintext = CBC\_decrypt(a\_ciphertext, a\_key, a\_IV)

    print (plaintext)

    # The plaintext is unpadded to its original size

    #plaintext = unpad(plaintext, AES.block\_size)

    print\_big\_and\_green(plaintext.decode())

    # Assert that the first 18 bytes of the plaintext match a known value

    # assert plaintext[:18] == b64decode('eW91IGtub3cgdGhlIHJ1bGVz')

    # Print the plaintext in big and green letters

    # print\_big\_and\_green(plaintext.decode())

if \_\_name\_\_ == "\_\_main\_\_":

     main()

# Code 5 – Breaking ECB.py

from base64 import b64decode

from Cryptodome.Cipher import AES

from secrets import token\_bytes

# Function for PKCS7 padding

def pkcs7\_pad(plaintext, blocksize):

    """

    Appends the plaintext with n bytes, making it an even multiple of blocksize.

    Byte used for appending is byteform of n.

    Parameters

    ----------

    plaintext : bytes

        plaintext to be appended

    blocksize : int

        blocksize to conform to

    Returns

    -------

    plaintext : bytes

        plaintext appended with n bytes

    """

    # Determine how many bytes to append

    n = blocksize - len(plaintext) % blocksize

    # Append n\*(byteform of n) to plaintext

    # n is in a list as bytes() expects iterable

    plaintext += (n \* bytes([n]))

    return plaintext

# Function to encrypt plaintext using AES-ECB

def ECB\_oracle(plaintext, key):

    """

    Appends a top-secret identifier to the plaintext

    and encrypts it under AES-ECB using the provided key.

    Parameters

    ----------

    plaintext : bytes

        plaintext to be encrypted

    key : bytes

        16-byte key to be used in decryption

    Returns

    -------

    ciphertext : bytes

        encrypted plaintext

    """

    # Append secret identifier to plaintext

    plaintext += b64decode('U2F5IG5hIG5hIG5hCk9uIGEgZGFyayBkZXNlcnRlZCB3YXksIHNheSBuYSBuYSBuYQpUaGVyZSdzIGEgbGlnaHQgZm9yIHlvdSB0aGF0IHdhaXRzLCBpdCdzIG5hIG5hIG5hClNheSBuYSBuYSBuYSwgc2F5IG5hIG5hIG5hCllvdSdyZSBub3QgYWxvbmUsIHNvIHN0YW5kIHVwLCBuYSBuYSBuYQpCZSBhIGhlcm8sIGJlIHRoZSByYWluYm93LCBhbmQgc2luZyBuYSBuYSBuYQpTYXkgbmEgbmEgbmE=')

    # Apply PKCS7 padding

    plaintext = pkcs7\_pad(plaintext, len(key))

    # Create AES cipher object and encrypt plaintext

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

    ciphertext = cipher.encrypt(plaintext)

    return ciphertext

# Generate a random key

key = token\_bytes(16)

# Function to find the block length used by the ECB oracle

def find\_block\_length():

    """

    Finds the block length used by the ECB oracle.

    Returns

    -------

    blocksize : integer

        blocksize used by ECB oracle

    """

    plaintext = b''

    ciphertext = ECB\_oracle(plaintext, key)

    blocksize = 0

    # Incrementally increase plaintext size until ciphertext size changes

    for i in range(1, 100):

        plaintext += b'A'

        new\_ciphertext = ECB\_oracle(plaintext, key)

        if len(new\_ciphertext) != len(ciphertext):

            blocksize = len(new\_ciphertext) - len(ciphertext)

            break

    return blocksize

# Function to get the target ciphertext for analysis

def get\_target\_ciphertext(blocksize):

    padding = b'A' \* (blocksize - 1)  # Create a padding of blocksize - 1 bytes

    target\_ciphertext = ECB\_oracle(padding, key)

    return target\_ciphertext

# Function to break AES ECB encryption

def break\_ECB(blocksize):

    recovered = b'' # initialize the recovered plaintext

    target\_ciphertext = get\_target\_ciphertext(blocksize) # get the target ciphertext

    blocks = [target\_ciphertext[i:i + blocksize] for i in range(0, len(target\_ciphertext), blocksize)] # split the ciphertext into blocks

    for block\_num, block in enumerate(blocks):

        decrypted\_block = b''

        for byte\_num in range(blocksize):

            comparison\_made = False

            for i in range(256):

                padding = b'A' \* (blocksize - byte\_num - 1) # create padding

                data = padding + recovered + decrypted\_block + bytes([i]) # create data

                cipher\_block = ECB\_oracle(data, key)[block\_num \* blocksize:(block\_num + 1) \* blocksize] # get the cipher block

                padding\_block = ECB\_oracle(padding, key)[block\_num \* blocksize:(block\_num + 1) \* blocksize] # get the padding block

                if cipher\_block == padding\_block: # compare the cipher block with the padding block

                    decrypted\_block += bytes([i]) # add the byte to the decrypted block

                    comparison\_made = True

                    break

            if not comparison\_made:

                break

        recovered += decrypted\_block

    return recovered

# Function to clear terminal screen

def clear\_screen():

    """

    Clears the terminal screen.

    This function uses an escape sequence to clear the terminal screen. The escape sequence "\033c" is

    recognized by many terminal types as the command to clear the screen.

    Returns

    -------

    None

    """

    print("\033c", end="")

# Main function to execute the code

def main():

    clear\_screen()

    # Find block size

    blocksize = find\_block\_length()

    # Break ECB encryption

    secret = break\_ECB(blocksize)

    # Print the recovered secret

    print(secret.decode('utf-8'))

if \_\_name\_\_ == "\_\_main\_\_":

    main()