[2023 JBUCTF] crypto

slide

Write-Up

문제 개요

제공 파일 : myCrypto.py, slide.py

slide.py

```python
from myCrypto import *
from Crypto.Random import get_random_bytes
from base64 import b64decode, b64encode

flag = open('/flag', 'rb').read()
key = get_random_bytes(16)
crypto = myCrypto(key)
enc_flag = crypto.encrypt(pad(flag, 16))

for i in range(10):
    print('encrypt [1]')
    print('slide   [2]')
    print('flag    [3]')
    print('exit    [4]')
    try:
        i = int(input('>> '))
        if i < 1 or i > 4:
            raise ValueError
    except Exception as e:
        print(e)
        exit()
    if i == 1:
        try:
            p = b64decode(input('Input text (base64) >> '))
            print(b64encode(crypto.encrypt(pad(p, crypto.block_size))).decode())
        except Exception as e:
            print(e)
            exit()
    elif i == 2:
        try:
            p = b64decode(input('Input text (base64) >> '))
            p = pad(p, crypto.block_size)
            states = [p[i: i+16] for i in range(0, len(p), crypto.block_size)]
            result = b''
            for state in states:
                state = byte2state(state)
                state = crypto.round(state)
                result += state2byte(state)
            print(b64encode(result).decode())
        except Exception as e:
            print(e)
            exit()
    elif i == 3:
        print(b64encode(enc_flag).decode())
    elif i == 4:
        exit()
```

myCrypto 클래스의 인스턴스인 crypto를 랜덤한 16byte키로 초기화 해서 생성한다.

그리고 flag를 myCrypto 클래스의 encrypt 함수로 암호화 한 결과를 enc_flag 변수에 저장한다.

1 입력

입력 받은 문자열을 base64 디코딩 후, pad함수로 패딩을 하고, myCrypto 클래스의 encrypt 함수로 암호화 한 결과를 base64 인코딩 해서 출력한다.

2 입력

1을 입력한 것과 동작이 비슷하지만 encrypt 함수가 아닌 round함수를 사용한 결과를 출력한다.

3 입력

enc_flag 값을 base64인코딩해서 출력한다.

4 입력

프로그램을 종료한다.

myCrypto.py

```python
def pad(byte, block_size):
    pad_byte =  bytes([block_size - (len(byte) % block_size)])
    return byte + pad_byte[0] * pad_byte

def unpad(byte, block_size):
    pad_bytes = byte[-1]
    if pad_bytes > block_size or pad_bytes == 0:
        raise ValueError
    for i in range(pad_bytes):
        if byte[-1] != pad_bytes:
            raise ValueError
    return byte[:-(pad_bytes)]

def byte2list(byte):
        return [byte[4*i + j] for i in range(4) for j in range(4)]

def list2byte(arr):
    result = b''
    for i in range(len(arr)):
        result += bytes([arr[i]])
    return result

def list2state(arr):
    return [arr[i+j] for i in range(4) for j in range(0, 16, 4)]

def state2list(state):
    return [state[i+j] for i in range(4) for j in range(0, 16, 4)]

def byte2state(byte):
    return list2state(byte2list(byte))

def state2byte(state):
    return list2byte(state2list(state))

def mult(x, y):
    result = 0
    while y:
        if y & 0x01:
            result ^= x
        y >>= 1
        x <<= 1
        if x & 0x100:
            x ^= 0x1b
    return result & 0xff

class myCrypto:
    block_size  = 16

    sbox = (
            0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
            0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
            0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
            0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
            0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
            0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
            0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
            0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
            0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
            0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x08, 0xDB,
            0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
            0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
            0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
            0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
            0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
            0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
            )

    inv_sbox = (
            0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
            0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
            0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
            0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
            0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
            0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
            0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
            0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
            0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
            0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
            0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
            0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
            0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
            0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
            0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
            0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D
            )

    def __init__(self, key):
        self.key = key
        if len(key) != 16:
            raise ValueError
```

```python
    def addRoundKey(self, state):
        key_state = byte2state(self.key)
        return [i^j for i, j in zip(state, key_state)]

    def mixColumns(self, state):
        return [mult(state[i], 2)^mult(state[i+4],3)^state[i+8]^state[i+12] for i in range(4)] + \
               [state[i]^mult(state[i+4], 2) ^mult(state[i+8], 3)^state[i+12] for i in range(4)] + \
               [state[i]^state[i+4] ^mult(state[i+8], 2)^mult(state[i+12], 3) for i in range(4)] + \
               [mult(state[i],3 )^state[i+4]^state[i+8]^mult(state[i+12], 2) for i in range(4)]

    def inv_mixColumns(self, state):
        return [mult(state[i], 14)^mult(state[i+4],11)^mult(state[i+8], 13)^mult(state[i+12], 9) for i in range(4)] + \
               [mult(state[i], 9)^mult(state[i+4], 14) ^mult(state[i+8], 11)^mult(state[i+12], 13) for i in range(4)] + \
               [mult(state[i], 13)^mult(state[i+4], 9) ^mult(state[i+8], 14)^mult(state[i+12], 11) for i in range(4)] + \
               [mult(state[i], 11)^mult(state[i+4], 13)^mult(state[i+8], 9)^mult(state[i+12], 14) for i in range(4)]

    def shiftRows(self, state):
        return state[:4] + \
               state[7:8] + state[4:7] + \
               state[10:12] + state[8:10] + \
               state[15:] + state[12:15]

    def inv_shiftRows(self ,state):
        return state[:4] + \
               state[5:8] + state[4:5] + \
               state[10:12] + state[8:10] + \
               state[13:] + state[12:13]

    def subBytes(self, state):
        return [myCrypto.sbox[i] for i in state]

    def inv_subBytes(self, state):
        return [myCrypto.inv_sbox[i] for i in state]

    def round(self, state):
        return self.addRoundKey(self.mixColumns(self.shiftRows(self.subBytes(state))))
    def inv_round(self, state):
        return self.inv_subBytes(self.inv_shiftRows(self.inv_mixColumns(self.addRoundKey(state))))
```

```python
    def encrypt(self, state):
        try:
            unpad(state, self.block_size)
        except ValueError as e:
            print(e)
            exit()

        blocks = [state[i: i+16] for i in range(0, len(state), self.block_size)]
        result = b''
        for block in blocks:
            state = byte2state(block)
            for i in range(20):
                state = self.round(state)
            result += state2byte(state)
        return result

    def decrypt(self, state):
        if len(state) % self.block_size != 0:
            return 0

        blocks = [state[i: i+16] for i in range(0, len(state), self.block_size)]
        result = b''
        for block in blocks:
            state = byte2state(block)
            for i in range(20):
                state = self.inv_round(state)
            result += state2byte(state)
        return result
```

아래의 함수들은 16byte 또는 길이가 16인 값을 인자로 갖는다.

패딩 참고

Wikipedia Padding (cryptography) : https://url.kr/v76dwc
IBM PKCS padding method : https://url.kr/rblvmy

def pad(byte, block_size) : PKCS#7 Padding

def unpad(byte, block_size) : PKCS#7 UnPadding

def byte2list(byte) : Byte To List

def list2byte(arr) : List To Byte

def list2state(arr) : List To State

def state2list(state) : State To List

def byte2state(byte) : Byte To State

def state2byte(state) : State To Byte

def mult(x, y) : GF(2^8) Multiplication

myCrypto 클래스 함수

def addRoundKey(self, state)

def mixColumns(self, state)

def inv_mixColumns(self, state)

def shiftRows(self, state)

def inv_shiftRows(self ,state)

def subBytes(self, state)

def inv_subBytes(self, state)

위 함수들은 AES의 연산을 구현한 함수들이다. 자세한 내용은 아래 링크를 참고하자.

Advanced Encryption Standard (AES) : https://url.kr/9nw6t2

def round(self, state) : addRoundKey(mixColumns(shiftRows(subBytes(state))))를 계산한다.

def inv_round(self, state) : round함수의 역연산이다.

def encrypt(self, state) : 입력 받은 값을 16byte씩 나눠서 round함수를 20번 수행한다.

def decrypt(self, state) 입력 받은 값을 16byte씩 나눠서 inv_round 함수를 20번 수행한다.

문제 풀이

2번을 입력 시 원하는 문자열을 한번의 round함수를 거친 값을 알 수 있다.

입력한 문자열을 A라 할 때, A에 round함수를 적용하면

subBytes, shiftRows, mixColumns, addRoundKey 순으로 함수가 적용된다.

따라서 A에 round함수가 적용된 값을 B라 하면

B = addRoundKey(mixColumns(shiftRows(subBytes(A)))) 이다.

이때 subBytes, shiftRows, mixColumns 함수는 addRoundKey 함수처럼 key와 같은 비밀 값을 사용하지 않는 함수이므로 mixColumns(shiftRows(subBytes(A)))는 계산이 가능하다.

이 값을 A' 이라고 하면 B = addRoundKey(A') 이다.

addRoundKey연산은 key와 매개변수 값을 xor연산을 하는 함수이므로

B = addRoundKey(A') = A' xor Key이고, B = A' xor key 이므로 B xor A' = key이다.

즉 key를 알아낼 수 있다.

이제 구한 key값으로 myCrypto 인스턴스를 생성해서 enc_flag를 복호화 하면 flag값이 나온다.

주의 사항 : 문제 풀이 코드 작성시 state, byte 변환을 신경 써서 해야 한다.

exploit.py

```python
from myCrypto import *
from pwn import *
from base64 import b64encode, b64decode

p = remote('172.17.0.2', 10008)

def xor_bytes(a, b, block_size):
    result = b''
    for i in range(block_size):
        result += bytes([a[i] ^ b[i]])
    return result

BLOCK_SIZE = 16
key = b"B"*16
crypto = myCrypto(key)
p1 = b'A'*16

p.recvuntil(b'>> ')
p.sendline(b'2')
p.recvuntil(b'Input text (base64) >> ')
p.sendline(b64encode(p1))
p2 = p.recvline()[:-1]
p2 = b64decode(p2)[:16]

p1 = crypto.mixColumns(crypto.shiftRows(crypto.subBytes(byte2state(p1))))

key = xor_bytes(p1, byte2state(p2), 16)

p.recvuntil(b'>> ')
p.sendline(b'3')
enc_flag = p.recvline()[:-1]
enc_flag = b64decode(enc_flag)

crypto = myCrypto(key)
flag = unpad(crypto.decrypt(enc_flag), BLOCK_SIZE)
print(f'flag : {flag.decode()}')

```

FLAG

scpCTF{83e19dccdeba1b0f16fabf44165da7d23dbe82a4d7f983db4631fb519129}