Securitatea Informaţiei - Tema 1

Cernat Monica- Roxana 3A6

1. **Nodul KM**

Nodul KM deţine 3 chei pe 128 de biţi.

- K1 reprezinta cheia cunoscută de către toate nodurile, cu ajutorul căreia se realizează comunicarea;
- K2 este asociată modului ECB;
- K3 este asociată modului OFB;

Cheile asociate fiecărui mod de comunicare sunt criptate cu ajutorul criptosistemului AES în modul ECB.

În momentul în care nodul A transmite către nodul KM că doreşte să iniţieze o conversaţie transmite şi modul prin care doreşte să realizeze comunicarea. Nodul KM transmite cheia corespunzătoare criptată.

```
def send_key(conn):
    conv_mode = so.recv_header(conn)
    if conv_mode == b"ecb":
        so.send_header(conn, encrypt_ecb_key())
        print('Sent ECB key')
    elif conv_mode == b"ofb":
        so.send_header(conn, encrypt_ofb_key())
        print('Sent OFB key')
    else:
        print('Received bad mode: ', conv_mode)
```

2. **Nodul A**

Cere introducerea de la tastatură a modului prin care se doreşte a realiza conversaţia, iar apoi cere de la nodul KM cheia corespunzătoare modului criptată, o decriptează, iar apoi continuă comunicarea cu nodul B.

```
def get_key(mode: bytes):
    sck = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sck.connect(("127.0.0.1", PORT_KM))
    so.send_header(sck, mode)
    print('Sent mode to KM, waiting for key')
    conv_key_enc = so.recv_header(sck)
    conv_key = shared_cipher.decrypt(conv_key_enc)
    print('Got key!')
    return conv_key
```

```
def ecb_conv(sck, key):
    cipher = em.ECB(key)

    print('Waiting for message... ')
    msg_enc = so.recv_header(sck)
    msg = cipher.decrypt(msg_enc)

    print('Message:', msg.decode('utf-8'))

    reply = input("Enter reply: ")
    reply_enc = cipher.encrypt(reply)
    so.send_header(sck, reply_enc)
```

```
def ofb_conv(sck, key):
    recv_iv = so.recv_header(sck)
    send_iv = so.recv_header(sck)

    recv_cipher = em.OFB(key, recv_iv)
    send_cipher = em.OFB(key, send_iv)

    print('Waiting for message... ')
    msg_enc = so.recv_header(sck)
    msg = recv_cipher.decrypt(msg_enc)

    print('Message:', msg.decode('utf-8'))

    reply = input("Enter reply: ")
    reply_enc = send_cipher.encrypt(reply)
    so.send_header(sck, reply_enc)
```

3. **Nodul B**

Aşteaptă să primească de la KM modul de comunicare preferat de către nodul A, iar după primirea acestuia şi a cheii corespunzătoare aşteaptă primirea unui mesaj de la nodul A. După primirea mesajului, îl decriptează şi aşteaptă introducerea de la tastatură a unui răspuns pe care îl transmite catre nodul A criptat prin modul anterior ales.

```
def ecb_conv(sck, key):
    cipher = em.ECB(key)

    print('Waiting for message... ')
    msg_enc = so.recv_header(sck)
    msg = cipher.decrypt(msg_enc)

    print('Message:', msg.decode('utf-8'))

    reply = input("Enter reply: ")
    reply_enc = cipher.encrypt(reply)
    so.send_header(sck, reply_enc)
```

```
def ofb_conv(sck, key):
    recv_iv = so.recv_header(sck)
    send_iv = so.recv_header(sck)

    recv_cipher = em.OFB(key, recv_iv)
    send_cipher = em.OFB(key, send_iv)

    print('Waiting for message... ')
    msg_enc = so.recv_header(sck)
    msg = recv_cipher.decrypt(msg_enc)

    print('Message:', msg.decode('utf-8'))

    reply = input("Enter reply: ")
    reply_enc = send_cipher.encrypt(reply)
    so.send_header(sck, reply_enc)
```

4. **Moduri de criptare**

ECB

- Dacă lungimea plaintextul nu este multiplu de 16, se adaugă atâţia octeti 0 până se ajunge la primul multiplu de 16.
- criptare: plaintextulse padează, iar apoi se împarte în blocuri de 16 octeti, iar fiecare este criptat independent cu ajutorul ajutorul criptosistemului AES
- decriptare: criptotextul se imparte în blocuri de 16 octeti, apoi fiecare bloc este decriptat cu ajutorul criptosisitemului AES, se concatenează blocurile iar apoi se îndepărtează padarea

```
class ECB:
    def __init__(self, key):
        self.key = key
        self.cipher = AES.new(self.key, AES.MODE_ECB)

    def encrypt(self, plain_text):
        if isinstance(plain_text, str):
            plain_text = bytes(plain_text, 'utf-8')
        plain_text = pad(plain_text)
        plain_blocks = [plain_text[i:i + 16] for i in range(0, len(plain_text), 16)]
        encr_blocks = [self.cipher.encrypt(block) for block in plain_blocks]
        encr = b''.join(encr_blocks)
        return encr

    def decrypt(self, crypto_text):
        crypto_blocks = [crypto_text[i:i + 16] for i in range(0, len(crypto_text), 16)]
        decr_blocks = [self.cipher.decrypt(block) for block in crypto_blocks]
        decr = b''.join(decr_blocks)
        return unpad(decr)
```

OFB

- pentru fiecare byte al plaintextului se face xor cu un byte al cheii, la fiecare pas se retine pozitia pe care se afla byte-ul din cheie cu care se realizeaza criptarea
- daca am ajuns la sfarcitul cheii, dar plaintextul inca nu s-a terminar, cheia se cripteaza din nou cu ajutuorul iv-ului iar pozitia retinuta se rezeteaza.
- daca plaintextul se termina inainte ca cheia sa se fi terminat, se retine pozitia, iar la mesajul urmator se reia criptare plaintextul cu cheia de pe pozitia anterior memorata.

```
class OFB:
    def __init__(self, key, iv):
        self.key = key
        self.cipher = AES.new(self.key, AES.MODE_ECB)
        self.iv = iv
        self.key_stream = self.cipher.encrypt(self.iv)
        self.pos = 0

    def encrypt(self, plain_text):
        if isinstance(plain_text, str):
            plain_text = bytes(plain_text, 'utf-8')
        cipher_text = b''

        for plain_byte in plain_text:
            if self.pos < 16:
                cipher_text += bytes([self.key_stream[self.pos] ^ plain_byte])
                self.pos += 1
            else:
                self.key_stream = self.cipher.encrypt(self.key_stream)
        return cipher_text

    def decrypt(self, ciphertext):
        return self.encrypt(iphertext)
```