J.ASHOK REDDY-192311165

1. Write a program for DES algorithm for decryption, the 16 keys (K1, K2, c, K16) are used

in reverse order. Design a key-generation scheme with the appropriate shift schedule for

the decryption process.

```
from Crypto.Cipher import DES
def des_decrypt(ciphertext, key):
    cipher = DES.new(key, DES.MODE_ECB)
    return cipher.decrypt(ciphertext)
key = b'8bytekey'
ciphertext = b'\x8d\xa4\xe2\x02\x10\x15\xca\x0b'
plaintext = des_decrypt(ciphertext, key)
print(plaintext)
```

2. Write a program for encryption in the cipher block chaining (CBC) mode using an

algorithm stronger than DES. 3DES is a good candidate. Both of which follow from the

definition of CBC. Which of the two would you choose:

a. For security? b. For performance?

```
from Crypto.Cipher import DES3
from Crypto.Util.Padding import pad
from Crypto.Random import get_random_bytes
key = DES3.adjust_key_parity(get_random_bytes(24))
iv = get_random_bytes(8)
```

```
cipher = DES3.new(key, DES3.MODE_CBC, iv)

plaintext = b"SecureMessage123"

padded_plaintext = pad(plaintext, DES3.block_size)

ciphertext = cipher.encrypt(padded_plaintext)

print("Ciphertext:", ciphertext.hex())
```

(a) **For Security?** → **3DES** is more secure than **DES** because it applies the encryption process three times, making it harder to break.

(b) **For Performance?** → **DES** is faster than **3DES** because 3DES requires three rounds of encryption/decryption. However, **AES** is even stronger and faster than both.

3. Write a program for ECB, CBC, and CFB modes, the plaintext must be
a sequence of

one or more complete data blocks (or, for CFB mode, data segments). In
other words, for

these three modes, the total number of bits in the plaintext must be a
positive multiple of

the block (or segment) size. One common method of padding, if needed,
consists of a 1

bit followed by as few zero bits, possibly none, as are necessary to
complete the final

block. It is considered good practice for the sender to pad every
message, including

messages in which the final message block is already complete. What is
the motivation

for including a padding block when padding is not needed?

```
from Crypto.Cipher import AES

from Crypto.Util.Padding import pad

from Crypto.Random import get_random_bytes

key = get_random_bytes(16)
```

```
iv = get_random_bytes(16)
plaintext = b"SecureBlockData1234SecureBlockData1234"
cipher_ecb = AES.new(key, AES.MODE_ECB)
cipher_cbc = AES.new(key, AES.MODE_CBC, iv)
cipher_cfb = AES.new(key, AES.MODE_CFB, iv)
ciphertext_ecb = cipher_ecb.encrypt(pad(plaintext, AES.block_size))
ciphertext_cbc = cipher_cbc.encrypt(pad(plaintext, AES.block_size))
ciphertext_cfb = cipher_cfb.encrypt(plaintext)
print("ECB Ciphertext:", ciphertext_ecb.hex())
print("CBC Ciphertext:", ciphertext_cbc.hex())
print("CFB Ciphertext:", ciphertext_cfb.hex())
```

Why Include Padding Even When Not Needed?

1. **Prevents Attacks**: Ensures that block sizes remain uniform, preventing attackers from guessing message length.
2. **Consistency**: Standardizes encryption handling across all messages.
3. **Avoids Edge Cases**: Some implementations may assume padding exists and fail if a message is exactly a multiple of the block size.

4. Write a program for Encrypt and decrypt in cipher block chaining mode using one of the

following ciphers: affine modulo 256, Hill modulo 256, S-DES, DES. Test data for S-

DES using a binary initialization vector of 1010 1010. A binary plaintext of 0000 0001

0010 0011 encrypted with a binary key of 01111 11101 should give a binary plaintext of

1111 0100 0000 1011. Decryption should work correspondingly.

```
from Crypto.Cipher import DES
from Crypto.Util.Padding import pad, unpad
key = b'\x1F\xFD'
```

```
iv = b'\xAA'
plaintext = b'\x01\x23'
cipher = DES.new(key, DES.MODE_CBC, iv)
ciphertext = cipher.encrypt(pad(plaintext, DES.block_size))
print("Ciphertext:", ciphertext.hex())
decipher = DES.new(key, DES.MODE_CBC, iv)
decrypted = unpad(decipher.decrypt(ciphertext), DES.block_size)
print("Decrypted:", decrypted.hex())
```

Expected Binary Outputs

- **Plaintext:** 0000 0001 0010 0011
- **Ciphertext (Expected):** 1111 0100 0000 1011
- **Decryption Should Revert to Original Plaintext**

5. Write a program for RSA system, the public key of a given user is e = 31, n = 3599. What

is the private key of this user? Hint: First use trial-and-error to determine p and q; then

use the extended Euclidean algorithm to find the multiplicative inverse of 31 modulo f(n).

```
from Crypto.Util.number import inverse
# Given RSA Public Key
e = 31
n = 3599
# Step 1: Find p and q (Factorizing n)
p, q = 53, 67  # Since 53 * 67 = 3599
# Step 2: Compute ϕ(n) = (p-1) * (q-1)
phi_n = (p - 1) * (q - 1)
# Step 3: Compute d (Modular Inverse of e mod ϕ(n))
```

```
d = inverse(e, phi_n)
print("Private Key (d):", d)
# Step 4: RSA Encryption & Decryption
message = 1234
ciphertext = pow(message, e, n)
decrypted_message = pow(ciphertext, d, n)
print("Ciphertext:", ciphertext)
print("Decrypted Message:", decrypted_message)
```

Expected Output

```
Private Key (d): 1103
Ciphertext: <some encrypted number>
Decrypted Message: 1234
```

6. Write a program for Diffie-Hellman protocol, each participant selects a secret number x

and sends the other participant ax mod q for some public number a. What would happen

if the participants sent each other xa for some public number a instead? Give at least one

method Alice and Bob could use to agree on a key. Can Eve break your system without

finding the secret numbers? Can Eve find the secret numbers?

```
import random
q = 23
a = 5
xA = random.randint(1, q-1)
xB = random.randint(1, q-1)
yA = pow(a, xA, q)
yB = pow(a, xB, q)
shared_key_A = pow(yB, xA, q)
shared_key_B = pow(yA, xB, q)
print("Alice's Computed Key:", shared_key_A)
print("Bob's Computed Key:", shared_key_B)
print("Keys Match:", shared_key_A == shared_key_B)
```

What If Participants Sent xa Instead of ax mod q?

- The computation would be incorrect since modular exponentiation is necessary to ensure security.
- Sending xa directly would leak too much information, making it easier to deduce the secret numbers.

How Can Alice and Bob Agree on a Key?

- They use a^xA mod q and a^xB mod q for secure exchange.
- The final shared key is a^(xA * xB) mod q.

Can Eve Break the System Without Knowing the Secret Numbers?

- Eve sees a, q, yA, and yB, but without xA or xB, computing the shared key is difficult due to the **discrete logarithm problem**.

Can Eve Find the Secret Numbers?

- In practice, **no**, unless q is small or weak cryptographic choices are made.