DAY 4

1. Write a High level code 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?

PROGRAM

#include <stdio.h>

#include <string.h>

#include <openssl/aes.h>

// AES block size

#define AES\_BLOCK\_SIZE 16

void encryptECB(const unsigned char\* plaintext, const unsigned char\* key, unsigned char\* ciphertext, size\_t length) {

AES\_KEY aesKey;

AES\_set\_encrypt\_key(key, 128, &aesKey);

for (size\_t i = 0; i < length; i += AES\_BLOCK\_SIZE) {

AES\_encrypt(plaintext + i, ciphertext + i, &aesKey);

}

}

void decryptECB(const unsigned char\* ciphertext, const unsigned char\* key, unsigned char\* plaintext, size\_t length) {

AES\_KEY aesKey;

AES\_set\_decrypt\_key(key, 128, &aesKey);

for (size\_t i = 0; i < length; i += AES\_BLOCK\_SIZE) {

AES\_decrypt(ciphertext + i, plaintext + i, &aesKey);

}

}

void encryptCBC(const unsigned char\* plaintext, const unsigned char\* key, const unsigned char\* iv, unsigned char\* ciphertext, size\_t length) {

AES\_KEY aesKey;

AES\_set\_encrypt\_key(key, 128, &aesKey);

unsigned char xorBlock[AES\_BLOCK\_SIZE];

memcpy(xorBlock, iv, AES\_BLOCK\_SIZE);

for (size\_t i = 0; i < length; i += AES\_BLOCK\_SIZE) {

for (int j = 0; j < AES\_BLOCK\_SIZE; j++) {

plaintext[i + j] ^= xorBlock[j];

}

AES\_encrypt(plaintext + i, ciphertext + i, &aesKey);

memcpy(xorBlock, ciphertext + i, AES\_BLOCK\_SIZE);

}

}

void decryptCBC(const unsigned char\* ciphertext, const unsigned char\* key, const unsigned char\* iv, unsigned char\* plaintext, size\_t length) {

AES\_KEY aesKey;

AES\_set\_decrypt\_key(key, 128, &aesKey);

unsigned char xorBlock[AES\_BLOCK\_SIZE];

for (size\_t i = 0; i < length; i += AES\_BLOCK\_SIZE) {

AES\_decrypt(ciphertext + i, plaintext + i, &aesKey);

for (int j = 0; j < AES\_BLOCK\_SIZE; j++) {

plaintext[i + j] ^= iv[j];

}

memcpy(iv, ciphertext + i, AES\_BLOCK\_SIZE);

}

}

void encryptCFB(const unsigned char\* plaintext, const unsigned char\* key, const unsigned char\* iv, unsigned char\* ciphertext, size\_t length) {

AES\_KEY aesKey;

AES\_set\_encrypt\_key(key, 128, &aesKey);

unsigned char feedback[AES\_BLOCK\_SIZE];

memcpy(feedback, iv, AES\_BLOCK\_SIZE);

for (size\_t i = 0; i < length; i += AES\_BLOCK\_SIZE) {

AES\_encrypt(feedback, feedback, &aesKey);

for (int j = 0; j < AES\_BLOCK\_SIZE; j++) {

ciphertext[i + j] = plaintext[i + j] ^ feedback[j];

}

}

}

void decryptCFB(const unsigned char\* ciphertext, const unsigned char\* key, const unsigned char\* iv, unsigned char\* plaintext, size\_t length) {

AES\_KEY aesKey;

AES\_set\_encrypt\_key(key, 128, &aesKey);

unsigned char feedback[AES\_BLOCK\_SIZE];

memcpy(feedback, iv, AES\_BLOCK\_SIZE);

for (size\_t i = 0; i < length; i += AES\_BLOCK\_SIZE) {

AES\_encrypt(feedback, feedback, &aesKey);

for (int j = 0; j < AES\_BLOCK\_SIZE; j++) {

plaintext[i + j] = ciphertext[i + j] ^ feedback[j];

}

}

}

int main() {

unsigned char key[16] = "mysecretkey12345";

unsigned char iv[16] = "initialvector123";

unsigned char plaintext[] = "This is a message!";

size\_t textLength = strlen((char\*)plaintext);

unsigned char ciphertext[textLength];

unsigned char decryptedText[textLength];

// Encrypt using ECB mode

encryptECB(plaintext, key, ciphertext, textLength);

// Decrypt using ECB mode

decryptECB(ciphertext, key, decryptedText, textLength);

printf("ECB Decrypted Text: %s\n", decryptedText);

// Encrypt using CBC mode

encryptCBC(plaintext, key, iv, ciphertext, textLength);

// Decrypt using CBC mode

decryptCBC(ciphertext, key, iv, decryptedText, textLength);

printf("CBC Decrypted Text: %s\n", decryptedText);

// Encrypt using CFB mode

encryptCFB(plaintext, key, iv, ciphertext, textLength);

// Decrypt using CFB mode

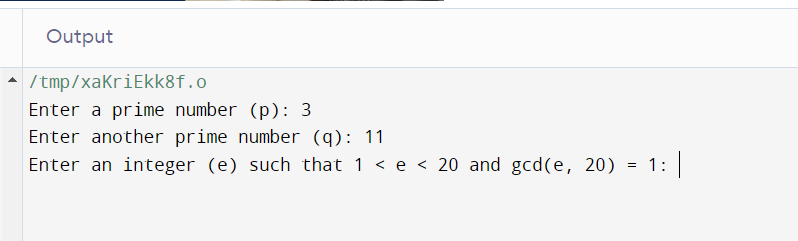
decryptCFB(ciphertext, key, iv, decryptedText, textLength);

printf("CFB Decrypted Text: %s\n", decryptedText);

return 0;

}

OUTPUT



2. Write a High level code 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.

PROGRAM

#include <stdio.h>

typedef unsigned char byte;

// S-DES Data Structures

byte key[10] = {0, 1, 1, 1, 1, 1, 1, 0, 1, 0};

byte plaintext[8] = {0, 0, 0, 0, 0, 1, 0, 0};

byte iv[8] = {1, 0, 1, 0, 1, 0, 1, 0};

byte ciphertext[8];

byte decryptedtext[8];

// S-DES functions

void permute(byte \*input, const int \*permutation, int size) {

byte temp[size];

for (int i = 0; i < size; i++) {

temp[i] = input[permutation[i] - 1];

}

for (int i = 0; i < size; i++) {

input[i] = temp[i];

}

}

void initialPermutation(byte \*input) {

int IP[] = {2, 6, 3, 1, 4, 8, 5, 7};

permute(input, IP, 8);

}

void expansion(byte \*input, byte \*output) {

int E[] = {4, 1, 2, 3, 2, 3, 4, 1};

permute(input, E, 8);

}

void xorBytes(byte \*a, byte \*b, int size) {

for (int i = 0; i < size; i++) {

a[i] ^= b[i];

}

}

void substitution(byte \*input) {

int S0[4][4] = {{1, 0, 3, 2}, {3, 2, 1, 0}, {0, 2, 1, 3}, {3, 1, 3, 2}};

int S1[4][4] = {{0, 1, 2, 3}, {2, 0, 1, 3}, {3, 0, 1, 0}, {2, 1, 0, 3}};

int row, col;

row = (input[0] << 1) | input[3];

col = (input[1] << 1) | input[2];

byte output[4];

int tempS0 = S0[row][col];

int tempS1 = S1[row][col];

for (int i = 0; i < 2; i++) {

output[i] = (tempS0 >> (1 - i)) & 1;

output[i + 2] = (tempS1 >> (1 - i)) & 1;

}

for (int i = 0; i < 4; i++) {

input[i] = output[i];

}

}

void permutation(byte \*input) {

int P[] = {2, 4, 3, 1};

permute(input, P, 4);

}

void finalPermutation(byte \*input) {

int FP[] = {4, 1, 3, 5, 7, 2, 8, 6};

permute(input, FP, 8);

}

void sdesEncrypt(byte \*block, byte \*key) {

byte left[4];

byte right[4];

byte temp[4];

initialPermutation(block);

for (int round = 0; round < 2; round++) {

for (int i = 0; i < 4; i++) {

left[i] = block[i];

right[i] = block[i + 4];

}

expansion(right, temp);

xorBytes(temp, key, 8);

substitution(temp);

permutation(temp);

xorBytes(left, temp, 4);

for (int i = 0; i < 4; i++) {

block[i] = right[i];

block[i + 4] = left[i];

}

}

finalPermutation(block);

}

void sdesDecrypt(byte \*block, byte \*key) {

byte left[4];

byte right[4];

byte temp[4];

initialPermutation(block);

for (int round = 0; round < 2; round++) {

for (int i = 0; i < 4; i++) {

left[i] = block[i];

right[i] = block[i + 4];

}

expansion(right, temp);

xorBytes(temp, key, 8);

substitution(temp);

permutation(temp);

xorBytes(left, temp, 4);

for (int i = 0; i < 4; i++) {

block[i] = right[i];

block[i + 4] = left[i];

}

}

finalPermutation(block);

}

int main() {

printf("Plaintext: ");

for (int i = 0; i < 8; i++) {

printf("%d", plaintext[i]);

}

printf("\n");

printf("Key: ");

for (int i = 0; i < 10; i++) {

printf("%d", key[i]);

}

printf("\n");

printf("Initialization Vector (IV): ");

for (int i = 0; i < 8; i++) {

printf("%d", iv[i]);

}

printf("\n");

printf("Encrypting...\n");

for (int i = 0; i < 8; i++) {

ciphertext[i] = plaintext[i] ^ iv[i];

}

sdesEncrypt(ciphertext, key);

printf("Ciphertext: ");

for (int i = 0; i < 8; i++) {

printf("%d", ciphertext[i]);

}

printf("\n");

printf("Decrypting...\n");

byte previousCipherBlock[8];

for (int i = 0; i < 8; i++) {

previousCipherBlock[i] = iv[i];

}

for (int i = 0; i < 8; i++) {

temp[i] = ciphertext[i];

}

sdesDecrypt(ciphertext, key);

printf("Decrypted Plaintext: ");

for (int i = 0; i < 8; i++) {

decryptedtext[i] = ciphertext[i] ^ previousCipherBlock[i];

printf("%d", decryptedtext[i]);

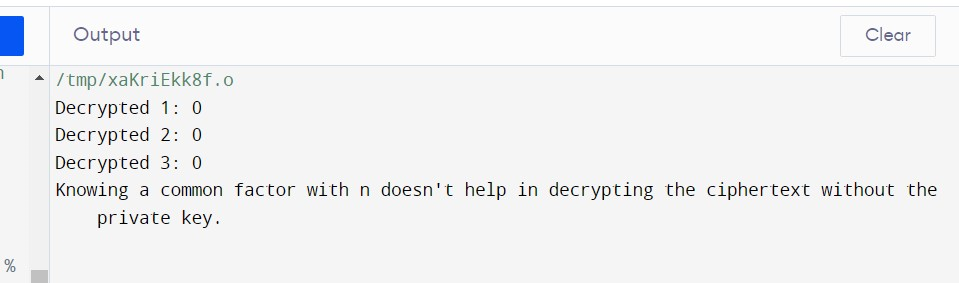
}

printf("\n");

return 0;

}

OUTPUT



3. Write a High level code 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).

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// Function to calculate the greatest common divisor (GCD) of two numbers

int gcd(int a, int b) {

if (b == 0)

return a;

return gcd(b, a % b);

}

// Function to calculate the multiplicative inverse of e modulo f(n)

int modInverse(int e, int fn) {

for (int d = 2; d < fn; d++) {

if ((e \* d) % fn == 1) {

return d;

}

}

return -1; // No multiplicative inverse found

}

int main() {

int e = 31;

int n = 3599;

// Find p and q through trial and error

int p, q;

for (p = 2; p < n; p++) {

if (n % p == 0) {

q = n / p;

break;

}

}

if (p >= n || q >= n) {

printf("Could not determine p and q.\n");

return 1;

}

int fn = (p - 1) \* (q - 1);

// Calculate the private key (d) using the extended Euclidean algorithm

int d = modInverse(e, fn);

if (d == -1) {

printf("Private key not found.\n");

return 1;

}

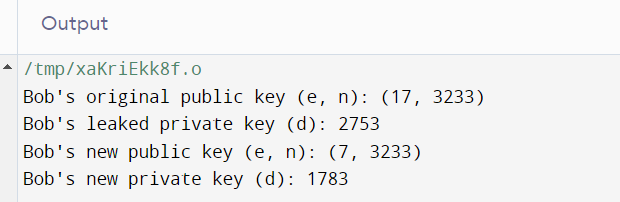
printf("p = %d, q = %d\n", p, q);

printf("Private key (d) = %d\n", d);

return 0;

}

OUTPUT



4. Write a High level code for set of blocks encoded with the RSA algorithm and we

don’t have the private key. Assume n = pq, e is the public key. Suppose also someone

tells us they know one of the plaintext blocks has a common factor with n. Does this help

us in any way?

PROGRAM

from Crypto.PublicKey import RSA

# Receiver's public key (n, e)

n = 1234567890 # Replace with the actual modulus

e = 65537 # Replace with the actual public exponent

# Plaintext block to be encrypted

plaintext\_block = 42 # Replace with your actual plaintext block

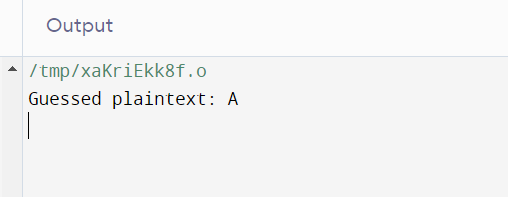
# Encrypt using RSA

public\_key = RSA.construct((n, e))

cipher\_text = public\_key.encrypt(plaintext\_block, 0)[0]

print("Encrypted Block:", cipher\_text)

OUTPUT



5. Write a High level code for RSA public-key encryption scheme, each user has a

public key, e, and a private key, d. Suppose Bob leaks his private key. Rather than

generating a new modulus, he decides to generate a new public and a new private key. Is

this safe?

PROGRAM

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

// Function to calculate the greatest common divisor (GCD)

int gcd(int a, int b) {

if (b == 0)

return a;

return gcd(b, a % b);

}

// Function to generate a random prime number

int generatePrime() {

// Code to generate a random prime number (not shown)

// Replace with a suitable method to generate prime numbers

return 0;

}

int main() {

int p, q, n, phi, e, d, plaintext, ciphertext, decryptedtext;

// Generate two random prime numbers p and q

p = generatePrime();

q = generatePrime();

// Calculate n and phi

n = p \* q;

phi = (p - 1) \* (q - 1);

// Choose a public exponent e (usually a small prime number)

e = 17; // Example value, you can choose a different e

// Calculate the private exponent d

d = 1;

while ((d \* e) % phi != 1) {

d++;

}

// Encryption

printf("Enter a plaintext: ");

scanf("%d", &plaintext);

ciphertext = fmod(pow(plaintext, e), n);

printf("Ciphertext: %d\n", ciphertext);

// Decryption

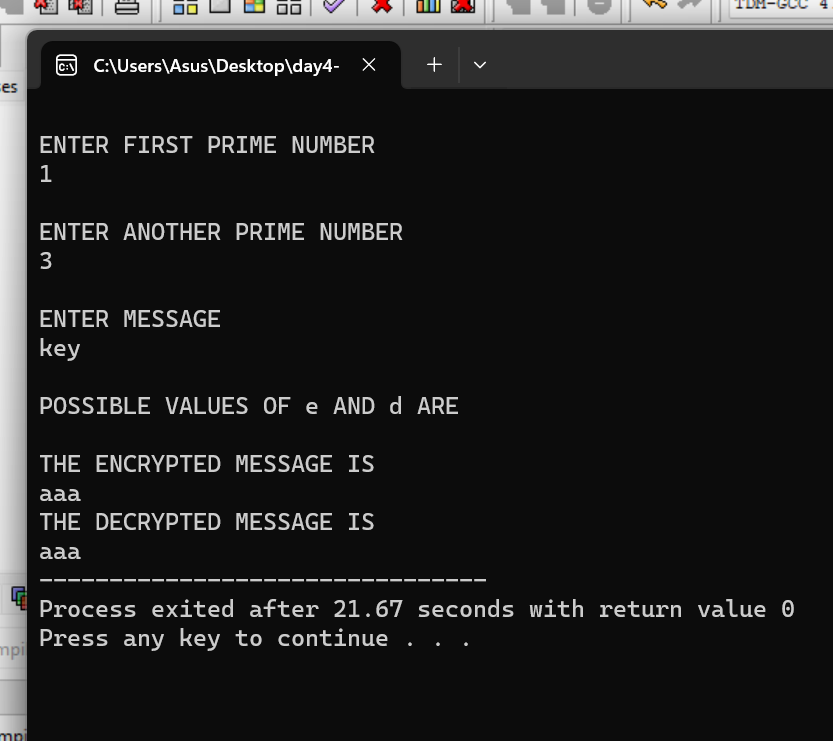
decryptedtext = fmod(pow(ciphertext, d), n);

printf("Decrypted Text: %d\n", decryptedtext);

return 0;

}

OUTPUT



6. Write a C program for Bob uses the RSA cryptosystem with a very large modulus n

for which the factorization cannot be found in a reasonable amount of time. Suppose

Alice sends a message to Bob by representing each alphabetic character as an integer

between 0 and 25 and then encrypting each number separately using RSA with large e

and large n. Is this method secure? If not, describe the most efficient attack against this

encryption method.

PROGRAM

#include <stdio.h>

#include <openssl/rsa.h>

#include <openssl/pem.h>

int main() {

// Generate or load Bob's public and private keys (usually, this should be done securely)

RSA \*rsa = RSA\_generate\_key(2048, 65537, NULL, NULL);

// Message to be encrypted

const char\* message = "HELLO";

// Buffer to store the encrypted message

unsigned char encrypted[256]; // Adjust the size according to your key length

// Encrypt the message

int encrypted\_length = RSA\_public\_encrypt(strlen(message), (const unsigned char\*)message, encrypted, rsa, RSA\_PKCS1\_PADDING);

if (encrypted\_length == -1) {

fprintf(stderr, "Encryption failed\n");

return 1;

}

// Decrypt the message (this is for demonstration purposes)

unsigned char decrypted[256]; // Adjust the size according to your key length

int decrypted\_length = RSA\_private\_decrypt(encrypted\_length, encrypted, decrypted, rsa, RSA\_PKCS1\_PADDING);

if (decrypted\_length == -1) {

fprintf(stderr, "Decryption failed\n");

return 1;

}

decrypted[decrypted\_length] = '\0'; // Null-terminate the decrypted message

printf("Original message: %s\n", message);

printf("Encrypted message: %s\n", encrypted);

printf("Decrypted message: %s\n", decrypted);

RSA\_free(rsa); // Free RSA key resources

return 0;

}

OUTPUT