NETWORK SECURITY LAB RECORD

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQURIEMENTS

FOR THE AWARD OF DEGREE

OF

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE

SUBMITTED BY:

KALIVARAPU SAI VIGNESH

22WU0101044

CSE A

UNDER SUPERVISION OF :

Prof.Vaishali Thakur

DEPARTMENT OF SCHOOL OF TECHNOLOGY

WOXSEN UNVERSITY

KAMKOLE,SADASIVPET ,HYDERABAD,TELANGANA 502345

LAB 1: To study and implement various **substitution cipher techniques** used in cryptography to replace plaintext characters with ciphertext characters, ensuring secure communication.

PLAY FAIR CIPHER

Algorithm:-

Step 1:-

- Take 5*5 Matrix
- Check for duplicates & remove them
- Append the key in matrix
- After appending write the remaining alphabets which are not in key
- Always take I & J as a pair as there are only 25 blots in matrix

Step 2:-

- Divide the plain text in a pair of 2 letters
- Make sure that there are not same pair of alphabets in a pair
- In that case add X
- If there is a single letter left at last add Z and make a pair

Step3:-

- Locate the words in the table
- Same row:- Replace with next letter to its immediate right.
- Same Column:- Replace with next letter to its immediate below.
- Different row & column:- Replace with one in its row but in the column of the other letter.

Code:-

```
def create_matrix(key):
    key = ''.join(sorted(set(key), key=key.index)).replace(" ", "").upper()
    key = key.replace('J', 'I')
    matrix = []

    for char in key:
        if char not in matrix and char.isalpha():
            matrix.append(char)

    for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":
        if char not in matrix:
            matrix.append(char)

    return matrix
```

```
def format_plaintext(plaintext):
    plaintext = plaintext.replace(" ", "").upper()
    plaintext = plaintext.replace('J', 'I')
    digraphs = []
    i = 0

    while i < len(plaintext):
        if i + 1 < len(plaintext):
            if plaintext[i] == plaintext[i + 1]:
                digraphs.append(plaintext[i] + 'X')
                i += 1
            else:
                digraphs.append(plaintext[i] + plaintext[i + 1])
                i += 2
        else:
            digraphs.append(plaintext[i] + 'Z')
            i += 1

    return digraphs

def encrypt(plaintext, key):
    matrix = create_matrix(key)
    digraphs = format_plaintext(plaintext)
    encrypted_text = ""

    for digraph in digraphs:
        row1, col1 = divmod(matrix.index(digraph[0]), 5)
        row2, col2 = divmod(matrix.index(digraph[1]), 5)

        if row1 == row2:
            encrypted_text += matrix[row1 * 5 + (col1 + 1) % 5]
            encrypted_text += matrix[row2 * 5 + (col2 + 1) % 5]
        elif col1 == col2:
            encrypted_text += matrix[((row1 + 1) % 5) * 5 + col1]
            encrypted_text += matrix[((row2 + 1) % 5) * 5 + col2]
        else:
            encrypted_text += matrix[row1 * 5 + col2]
            encrypted_text += matrix[row2 * 5 + col1]

    return encrypted_text

def decrypt(ciphertext, key):
    matrix = create_matrix(key)
    decrypted_text = ""

    for i in range(0, len(ciphertext), 2):
        row1, col1 = divmod(matrix.index(ciphertext[i]), 5)
        row2, col2 = divmod(matrix.index(ciphertext[i + 1]), 5)

        if row1 == row2:
            decrypted_text += matrix[row1 * 5 + (col1 - 1) % 5]
```

```
            decrypted_text += matrix[row2 * 5 + (col2 - 1) % 5]
        elif col1 == col2:
            decrypted_text += matrix[((row1 - 1) % 5) * 5 + col1]
            decrypted_text += matrix[((row2 - 1) % 5) * 5 + col2]
        else:
            decrypted_text += matrix[row1 * 5 + col2]
            decrypted_text += matrix[row2 * 5 + col1]

    return decrypted_text

def main():
    key = input("Enter the key: ")
    plaintext = input("Enter the plaintext: ")

    encrypted_text = encrypt(plaintext, key)
    print("Encrypted text:", encrypted_text)

    decrypted_text = decrypt(encrypted_text, key)
    print("Decrypted text:", decrypted_text)

if __name__ == "__main__":
    main()
```

Output:-

```
Enter the key:  MONARCHY
Enter the plaintext:  INSTRUMENTS
Encrypted text: GATLMZCLRQTX
Decrypted text: INSTRUMENTSZ
```

CAESR CIPHER

Algorithm:-

1. Input:
   - A plaintext .
   - A shift value (key) that determines how many places each letter is shifted in the alphabet.

2. Process:
   - For each character in the plaintext:
     - Determine its position in the alphabet (A-Z or a-z).
     - Apply the shift value to this position, wrapping around if necessary (
     - Replace the original letter with the shifted letter.

3. Output:
   - The resulting text after applying the shift to each letter (encrypted text).

4. Decryption:

- To decrypt an encrypted message, apply the same process but use the negative of the shift value to reverse the encryption.

Code:-

```
def caesar_cipher(plaintext, shift):
    encrypted_text = ""

    for char in plaintext:
        if char.isalpha():  # Check if the character is a letter
            # Determine if the character is uppercase or lowercase
            start = ord('A') if char.isupper() else ord('a')
            # Shift the character and wrap around using modulo
            encrypted_char = chr((ord(char) - start + shift) % 26 + start)
            encrypted_text += encrypted_char
        else:
            # Non-alphabetic characters are added unchanged
            encrypted_text += char

    return encrypted_text

def caesar_cipher_decrypt(ciphertext, shift):
    decrypted_text = ""

    for char in ciphertext:
        if char.isalpha():  # Check if the character is a letter
            # Determine if the character is uppercase or lowercase
            start = ord('A') if char.isupper() else ord('a')
            # Shift the character back and wrap around using modulo
            decrypted_char = chr((ord(char) - start - shift) % 26 + start)
            decrypted_text += decrypted_char
        else:
            # Non-alphabetic characters are added unchanged
            decrypted_text += char

    return decrypted_text

def main():
    # Take input for key shift and plaintext
    shift = int(input("Enter the key shift (0-25): "))
    plaintext = input("Enter the plaintext: ")

    # Encrypt the plaintext using the Caesar cipher
    encrypted_text = caesar_cipher(plaintext, shift)
    print("Encrypted text:", encrypted_text)

    # Decrypt the ciphertext back to plaintext
    decrypted_text = caesar_cipher_decrypt(encrypted_text, shift)
    print("Decrypted text:", decrypted_text)
```

```
if __name__ == "__main__":
    main()
```

Output:-

```
Enter the key shift (0-25):  4
Enter the plaintext:  VIGNESH
Encrypted text: ZMKRIWL
Decrypted text: VIGNESH
```

LAB 2: To implement and analyze the transposition-based Rail Fence cipher for encrypting text by rearranging characters in a zigzag pattern.

Algorithm:- Transposition technique

Encryption:

1. Choose the number of rails (rows).
2. Write the plaintext in a zigzag pattern across the rails.
3. Read the message row by row to get the ciphertext.

Decryption:

1. Calculate the zigzag pattern to determine the placement of characters.
2. Place the ciphertext characters back into the zigzag structure.
3. Read row by row to reconstruct the original message.

Code:-

```
4. def encrypt_rail_fence(text, key):
5.     # Create the matrix for rail fence
6.     rail = [['\n' for _ in range(len(text))] for _ in range(key)]
7.     direction_down = False
8.     row, col = 0, 0
9.
10.    # Place the characters in zigzag, including spaces
11.    for char in text:
12.        if row == 0 or row == key - 1:
13.            direction_down = not direction_down
14.        rail[row][col] = char
15.        col += 1
16.        row += 1 if direction_down else -1
17.
18.    # Read the matrix row by row
19.    encrypted_text = []
20.    for r in rail:
21.        encrypted_text.extend([char for char in r if char != '\n'])
22.
23.    return ''.join(encrypted_text)
24.
25.def decrypt_rail_fence(cipher, key):
26.    # Create the matrix for rail fence
27.    rail = [['\n' for _ in range(len(cipher))] for _ in range(key)]
28.    direction_down = None
29.    row, col = 0, 0
30.
31.    # Mark positions for letters, including spaces
```

```
    for _ in cipher:
        if row == 0:
            direction_down = True
        if row == key - 1:
            direction_down = False
        rail[row][col] = '*'
        col += 1
        row += 1 if direction_down else -1

    # Fill the rail matrix with cipher text
    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

    # Read the matrix in zigzag
    result = []
    row, col = 0, 0
    for _ in range(len(cipher)):
        if row == 0:
            direction_down = True
        if row == key - 1:
            direction_down = False
        result.append(rail[row][col])
        col += 1
        row += 1 if direction_down else -1

    return ''.join(result)

# Examples
plaintext1 = "NETWORK SECURITY"
key1 = 2
encrypted1 = encrypt_rail_fence(plaintext1, key1)
print(f"Encrypted (KEY-{key1}):", encrypted1)

plaintext2 = "CRYPTHOGRAPHY"
key2 = 3
encrypted2 = encrypt_rail_fence(plaintext2, key2)
print(f"Encrypted (KEY-{key2}):", encrypted2)

plaintext3 = "KALIVARAPU SAI VIGNESH"
key3 = 3
encrypted3 = encrypt_rail_fence(plaintext3, key3)
print(f"Encrypted (KEY-{key3}):", encrypted3)

# Optional decryption examples
decrypted1 = decrypt_rail_fence(encrypted1, key1)
print(f"Decrypted (KEY-{key1}):", decrypted1)

```

```
decrypted2 = decrypt_rail_fence(encrypted2, key2)
print(f"Decrypted (KEY-{key2}):", decrypted2)

decrypted3 = decrypt_rail_fence(encrypted3, key3)
print(f"Decrypted (KEY-{key3}):", decrypted3)
```

OUTPUT:-

```
Encrypted (KEY-2): NTOKSCRTEWR EUIY
Encrypted (KEY-3): CTRYRPHGAHYOP
Encrypted (KEY-3): KVPAISAIAAUSIVGEHLR  N
Decrypted (KEY-2): NETWORK SECURITY
Decrypted (KEY-3): CRYPTHOGRAPHY
Decrypted (KEY-3): KALIVARAPU SAI VIGNESH
```

LAB 3: To study and apply single and double columnar transposition techniques to encrypt messages by rearranging text in a columnar grid format.

SINGLE COLUMAR (Transposition technique)

Algorithm:-

Encryption:

1. Choose a keyword (e.g., `"KEY"`) and assign a numerical order to its letters.
2. Write the plaintext row-wise into a table with columns equal to the length of the key.
3. Rearrange the columns based on the alphabetical order of the keyword.
4. Read the columns top to bottom to form the ciphertext.

Decryption:

1. Recreate the column structure using the key.
2. Fill in the ciphertext column-wise using the key order.
3. Read row-wise to reconstruct the original plaintext.

Code:-

```
def encrypt_columnar_with_keyword(plaintext, keyword):
    plaintext = plaintext.replace(" ", "_")
    column_order = sorted([(char, index) for index, char in enumerate(keyword)])
    column_order = [index for _, index in column_order]
    num_columns = len(keyword)
    num_rows = -(-len(plaintext) // num_columns)
    grid = [['' for _ in range(num_columns)] for _ in range(num_rows)]
    idx = 0
    for row in range(num_rows):
        for col in range(num_columns):
            if idx < len(plaintext):
                grid[row][col] = plaintext[idx]
                idx += 1
            else:
                grid[row][col] = '_'
    ciphertext = ''
    for col in column_order:
        for row in range(num_rows):
            ciphertext += grid[row][col]
    return ciphertext
```

```python
def decrypt_columnar_with_keyword(ciphertext, keyword):
    num_columns = len(keyword)
    num_rows = -(-len(ciphertext) // num_columns)
    column_order = sorted([(char, index) for index, char in enumerate(keyword)])
    column_order = [index for _, index in column_order]
    grid = [['' for _ in range(num_columns)] for _ in range(num_rows)]
    idx = 0
    for col in column_order:
        for row in range(num_rows):
            if idx < len(ciphertext):
                grid[row][col] = ciphertext[idx]
                idx += 1
    plaintext = ''
    for row in range(num_rows):
        for col in range(num_columns):
            plaintext += grid[row][col]
    return plaintext.replace('_', ' ')

def main():
    plaintext = input("Enter the plaintext: ")
    keyword = input("Enter the columnar transposition keyword: ")
    columnar_encrypted = encrypt_columnar_with_keyword(plaintext, keyword)
    print("Columnar Encrypted text:", columnar_encrypted)
    columnar_decrypted = decrypt_columnar_with_keyword(columnar_encrypted, keyword)
    print("Columnar Decrypted text:", columnar_decrypted)

if __name__ == "__main__":
    main()
```

OUTPUT:-

```
Enter the plaintext:  KRISHNA RANJAN
Enter the columnar transposition keyword:  NICK
Columnar Encrypted text: IAN_RNANS_J_KHRA
Columnar Decrypted text: KRISHNA RANJAN
```

DOUBLE COLUMAR (Transposition technique)

Algorithm:-

Encryption:

1. **First Columnar Transposition:**
   - Choose a **first key** (e.g., `"KEY"`).

- Arrange the plaintext in a table with columns equal to the length of the key.
- Rearrange the columns based on the alphabetical order of the key.
- Read column-wise to get the intermediate ciphertext.

2. **Second Columnar Transposition:**
   - Use a **second key** (e.g., `"LOCK"`).
   - Arrange the intermediate ciphertext in a table with columns equal to the second key length.
   - Rearrange the columns based on the alphabetical order of the second key.
   - Read column-wise to get the final ciphertext.

Code:-

```
import math

def columnar_encrypt(msg, key):
    """Encrypt the message using a single columnar transposition cipher."""
    cipher = ""
    key_lst = sorted(list(key))
    col = len(key)
    row = int(math.ceil(len(msg) / col))

    # Pad the message with underscores if needed
    msg_lst = list(msg)
    msg_lst.extend('_' * (row * col - len(msg)))

    # Create the matrix
    matrix = [msg_lst[i:i + col] for i in range(0, len(msg_lst), col)]

    # Read the matrix column-wise using the key order
    for k in key_lst:
        curr_idx = key.index(k)
        cipher += ''.join(row[curr_idx] for row in matrix)

    return cipher


def columnar_decrypt(cipher, key):
    """Decrypt the message encrypted with a single columnar transposition
cipher."""
    msg = ""
    col = len(key)
    row = int(math.ceil(len(cipher) / col))
    key_lst = sorted(list(key))

    # Create an empty matrix
    dec_matrix = [[None] * col for _ in range(row)]

    # Fill the matrix column-wise using the key order
    msg_idx = 0
    for k in key_lst:
        curr_idx = key.index(k)
```

```
        for r in range(row):
            dec_matrix[r][curr_idx] = cipher[msg_idx]
            msg_idx += 1

    # Read the matrix row-wise to reconstruct the original message
    for r in dec_matrix:
        msg += ''.join(r)

    # Remove padding underscores
    return msg.rstrip('_')


def double_columnar_encrypt(msg, key1, key2):
    """Encrypt the message using Double Columnar Transposition Cipher."""
    intermediate = columnar_encrypt(msg, key1)
    cipher = columnar_encrypt(intermediate, key2)
    return cipher


def double_columnar_decrypt(cipher, key1, key2):
    """Decrypt the message encrypted with Double Columnar Transposition Cipher."""
    intermediate = columnar_decrypt(cipher, key2)
    msg = columnar_decrypt(intermediate, key1)
    return msg


# Main Function
if __name__ == "__main__":
    print("Double Columnar Transposition Cipher")
    msg = input("Enter the message: ").strip()
    key1 = input("Enter the first key: ").strip()
    key2 = input("Enter the second key: ").strip()

    encrypted = double_columnar_encrypt(msg, key1, key2)
    print(f"\nEncrypted Message: {encrypted}")

    decrypted = double_columnar_decrypt(encrypted, key1, key2)
    print(f"Decrypted Message: {decrypted}")
```

OUTPUT:-

```
Double Columnar Transposition Cipher
Enter the message:  KRISHNA RANJAN
Enter the first key:  NICK
Enter the second key:  BOAT

Encrypted Message: NAJRIRSKAN H_N_A
Decrypted Message: KRISHNA RANJAN
```

LAB 4: To implement and understand the working of the DES algorithm, a symmetric-key encryption method that operates using a 56-bit key.

Algorithm:-

DES is based on the two fundamental attributes of cryptography: substitution (also called confusion) and transposition (also called diffusion). DES consists of 16 steps, each of which is called a round. Each round performs the steps of substitution and transposition. Let us now discuss the broad-level steps in DES.

- In the first step, the 64-bit plain text block is handed over to an initial Permutation (IP) function.
- The initial permutation is performed on plain text.
- Next, the initial permutation (IP) produces two halves of the permuted block; saying Left Plain Text (LPT) and Right Plain Text (RPT).
- Now each LPT and RPT go through 16 rounds of the encryption process.
- In the end, LPT and RPT are rejoined and a Final Permutation (FP) is performed on the combined block
- The result of this process produces 64-bit ciphertext.

Code:-

```
from Crypto.Cipher import DES
import binascii
def pad(text):
   while len(text) % 8 != 0:
       text += " "  # Padding with spaces
   return text
key = b"8charKey"  # Must be exactly 8 bytes long
cipher = DES.new(key, DES.MODE_ECB)

# Message to be encrypted
plaintext = "HelloDES"

# Pad the plaintext to fit 8-byte blocks
padded_text = pad(plaintext)

# Encrypt the message
encrypted_text = cipher.encrypt(padded_text.encode())

# Convert encrypted data to hexadecimal for readability
hex_encrypted = binascii.hexlify(encrypted_text).decode()
```

```
# Decrypt the message
decrypted_text = cipher.decrypt(encrypted_text).decode().strip()

# Print the results
print(f"Original Message: {plaintext}")
print(f"Encrypted (Hex): {hex_encrypted}")
print(f"Decrypted Message: {decrypted_text}")
```

Output:-

```
Original Message: HelloDES
Encrypted (Hex): c4ff165fb410d0a2
Decrypted Message: HelloDES
```

LAB 5:-

To study and implement the RSA algorithm for secure asymmetric encryption using large prime numbers and modular arithmetic.

Algorithm:-

1. Key Generation

- Choose two large prime numbers, say p and q. These prime numbers should be kept secret.
- Calculate the product of primes, n = p * q. This product is part of the public as well as the private key.
- Calculate Euler Totient FunctionΦ(n) as Φ(n) = Φ(p * q) = Φ(p) * Φ(q) = (p – 1) * (q – 1).
- Choose encryption exponent e, such that

1. 1 < e < Φ(n)
2. gcd(e, Φ(n)) = 1, that is e should be co-prime with Φ(n).

- Calculate decryption exponent d, such that

❖ (d * e) ≡ 1 mod Φ(n), that is d is modular multiplicative inverse of e mod Φ(n). Some common methods to calculate multiplicative inverse are: Extended Euclidean Algorithm, Fermat's Little Theorem, etc.

❖ We can have multiple values of d satisfying (d * e) ≡ 1 mod Φ(n) but it does not matter which value we choose as all of them are valid keys and will result into same message on decryption.

- Finally, the Public Key = (n, e) and the Private Key = (n, d).

2. Encryption

To encrypt a message M, it is first converted to numerical representation using ASCII and other encoding schemes. Now, use the public key (n, e) to encrypt the message and get the cipher text using the formula:

C = Me mod n, where C is the Cipher text and e and n are parts of public key.

3. Decryption

To decrypt the cipher text C, use the private key (n, d) and get the original data using the formula:

M = Cd mod n, where M is the message and d and n are parts of private key.

Code:-

```
import math
p, q = 3, 11
n = p * q
phi_n = (p - 1) * (q - 1)
e = 7
def mod_inverse(e, phi):
    for d in range(1, phi):
        if (e * d) % phi == 1:
            return d
    return None
d = mod_inverse(e, phi_n)
def encrypt(M, e, n):
    return pow(M, e, n)
def decrypt(C, d, n):
    return pow(C, d, n)
M = 10
C = encrypt(M, e, n)
M_decrypted = decrypt(C, d, n)
print("Solution:-")
print(f"Public Key: (e={e}, n={n})")
print(f"Private Key: (d={d}, n={n})")
print(f"Encrypted: {C}")
print(f"Decrypted: {M_decrypted}")
```

Output:-

```
Solution:-
Public Key: (e=7, n=33)
Private Key: (d=3, n=33)
Encrypted: 10
Decrypted: 10
```

LAB 6:- Diffie-Hellman Key Exchange Algorithm

Algorithm:-

Method of Operation in Diffie-Hellman Key Exchange

Diffie-Hellman key exchange algorithm is based on the principles of modular exponentiation and discrete logarithms to allow two parties to securely establish a shared secret key over an insecure communication channel. Here is an operational overview of the process in context to Alice and Bob :

1. Parameters Setup

Alice and Bob must agree upon two number:

- A large prime number p,
- A generator g of p, which is the primitive root of p

These two number are shared and are not kept secret.

2. Key Generation

- Alice and Bob randomly chose a private key, say xa and xb, where xa is the private key of Alice and xb is the private key of Bob.
- These private keys are kept secret and not being shared.

3. Public Key Exchange

- Both Alice and Bob perform a calculation to generate their corresponding public keys.

$ya = ga \pmod p$

$yb = gb \pmod p$,

where ya is the public key of Alice and yb is the public key of Bob

- The public key are then shared with each other, ya is shared with Bob and yb is shared with Alice.

4. Shared Secret Key Calculation

- Alice then calculates the shared secret using the yb received from Bob and her private key as:

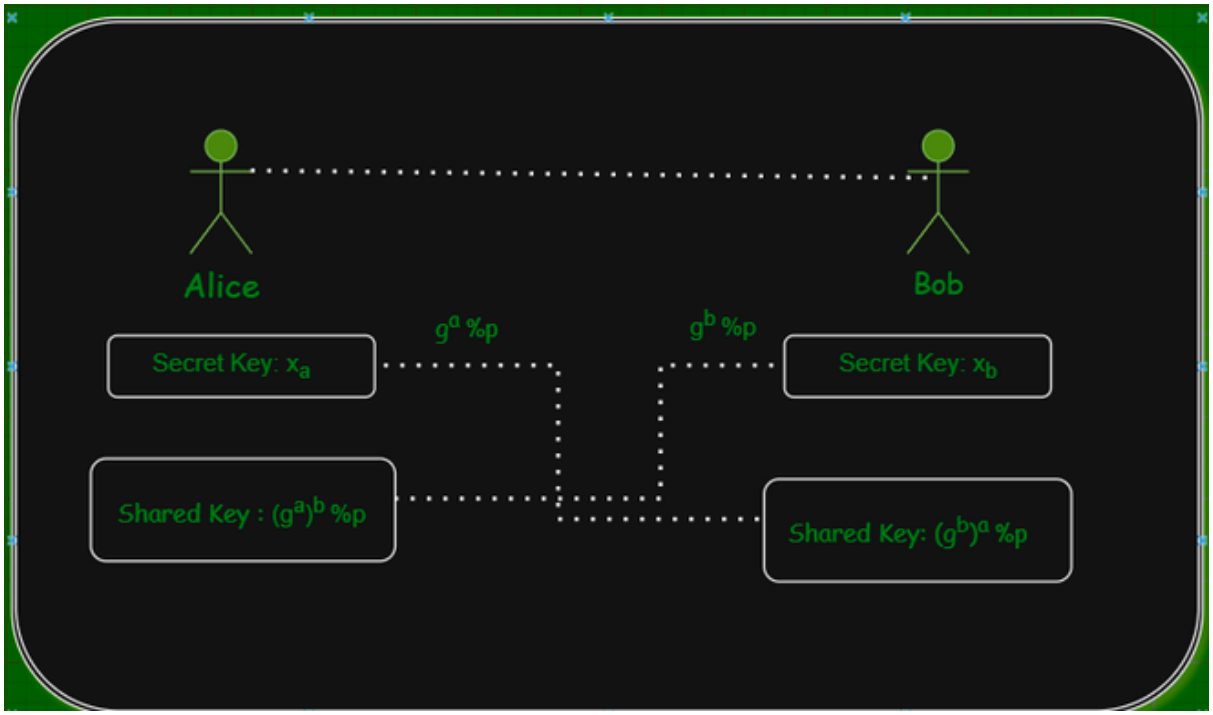
$k = (yb)xa \pmod p$

- Bob also calculates the shared secret using the ya received from Alice and his private key xb as:

$k = (ya)xb \pmod p$

5. Resulting Secret

Alice and Bob will end upon the same shared secret key, which can be used for encryption and decryption of information using symmetric key algorithms.



Code:-

```
import random
p = 17
g = 3
a = 10
A = pow(g, a, p)
b = 14
B = pow(g, b, p)
shared_secret_Alice = pow(B, a, p)
shared_secret_Bob = pow(A, b, p)
assert shared_secret_Alice == shared_secret_Bob
print("Solution:-")
print(f"Publicly Known (p={p}, g={g})")
print(f"Alice: Private Key={a}, Public Key={A}")
print(f"Bob: Private Key={b}, Public Key={B}")
print(f"Shared Secret: {shared_secret_Alice}")
```

Output:-

```
Solution:-
Publicly Known (p=17, g=3)
Alice: Private Key=10, Public Key=8
Bob: Private Key=14, Public Key=2
Shared Secret: 4
```

LAB 7:- SHA1

Algorithm:
1. Initialize a SHA-1 hash object using hashlib.
2. Encode the input string into bytes using UTF-8 encoding.
3. Update the hash object with the encoded data.
4. Compute the hexadecimal digest of the hash.
5. Return the computed hash as a string.

Explanation:
- SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function that produces a 160-bit (20-byte) hash value.
- It is commonly used for integrity checks, though it is now considered weak against certain attacks.
- This function takes a string input, encodes it in UTF-8, and processes it through the SHA-1 algorithm.
- The output is a hexadecimal representation of the 160-bit hash value.

Code:-

```
#SHA1
```

```
import hashlib

def sha1_encode(data: str) -> str:
    sha1_hash = hashlib.sha1()
    sha1_hash.update(data.encode('utf-8'))
    return sha1_hash.hexdigest()

# Example usage
text = "Hello, World!"
encoded_text = sha1_encode(text)
print(f"SHA-1 Hash: {encoded_text}")
```

```
SHA-1 Hash: 0a0a9f2a6772942557ab5355d76af442f8f65e01
```

LAB 8:- To explore common website vulnerabilities and exploits.

Introduction:

Web security is crucial in today's digital age, where cyber threats constantly evolve. Websites are often targeted by attackers to exploit vulnerabilities, steal data, or disrupt services. Understanding common vulnerabilities and exploits can help developers and administrators implement stronger security measures.

Common Website Vulnerabilities and Exploits:

1. **SQL Injection (SQLi)**
   - Attackers manipulate SQL queries to access or modify database information.
   - **Example:** Bypassing login authentication by injecting `' OR 1=1 --` in input fields.
   - **Prevention:** Use parameterized queries and ORM frameworks.
2. **Cross-Site Scripting (XSS)**
   - Injecting malicious scripts into web pages, which execute in users' browsers.
   - **Example:** `<script>alert('Hacked!')</script>` in a comment box.
   - **Prevention:** Input validation, encoding user input, and using Content Security Policy (CSP).
3. **Cross-Site Request Forgery (CSRF)**
   - Forcing users to execute unwanted actions on a web application in which they are authenticated.
   - **Example:** Attacker tricks a user into clicking a malicious link that performs an unwanted action (e.g., transferring money).
   - **Prevention:** Use CSRF tokens and implement same-site cookies.
4. **Broken Authentication**
   - Weak password policies, exposed session tokens, or improper session management.
   - **Example:** Session hijacking or credential stuffing attacks.
   - **Prevention:** Use multi-factor authentication (MFA), secure session handling, and strong password policies.
5. **Security Misconfiguration**
   - Default credentials, unnecessary services, or exposed debug pages.
   - **Example:** Exposing `.git` or `/admin` directories.
   - **Prevention:** Secure configurations, disabling unnecessary features, and regular audits.
6. **Sensitive Data Exposure**
   - Unencrypted sensitive information like passwords, credit card details, or API keys.
   - **Example:** Storing passwords in plaintext.
   - **Prevention:** Use encryption (AES, TLS), secure storage, and avoid exposing sensitive data in URLs.
7. **Server-Side Request Forgery (SSRF)**
   - Exploiting a web application to make unauthorized requests to internal resources.

    - **Example:** Requesting `http://localhost/admin` to access private resources.
    - **Prevention:** Restrict outbound requests and validate user input.
8. **Insecure Deserialization**
    - Executing malicious code by tampering with serialized objects.
    - **Example:** Injecting malicious payloads in serialized data.
    - **Prevention:** Avoid deserializing untrusted data and implement strict validation.
9. **Man-in-the-Middle (MITM) Attacks**
    - Intercepting communication between a user and a website.
    - **Example:** Capturing login credentials on an unsecured Wi-Fi network.
    - **Prevention:** Use HTTPS (TLS/SSL), VPNs, and certificate pinning.
10. **Denial of Service (DoS) & Distributed Denial of Service (DDoS)**

- Overloading a website with excessive traffic to disrupt service.
- **Example:** Botnet attacks flooding a website with requests.
- **Prevention:** Implement rate limiting, WAF (Web Application Firewall), and DDoS protection services.

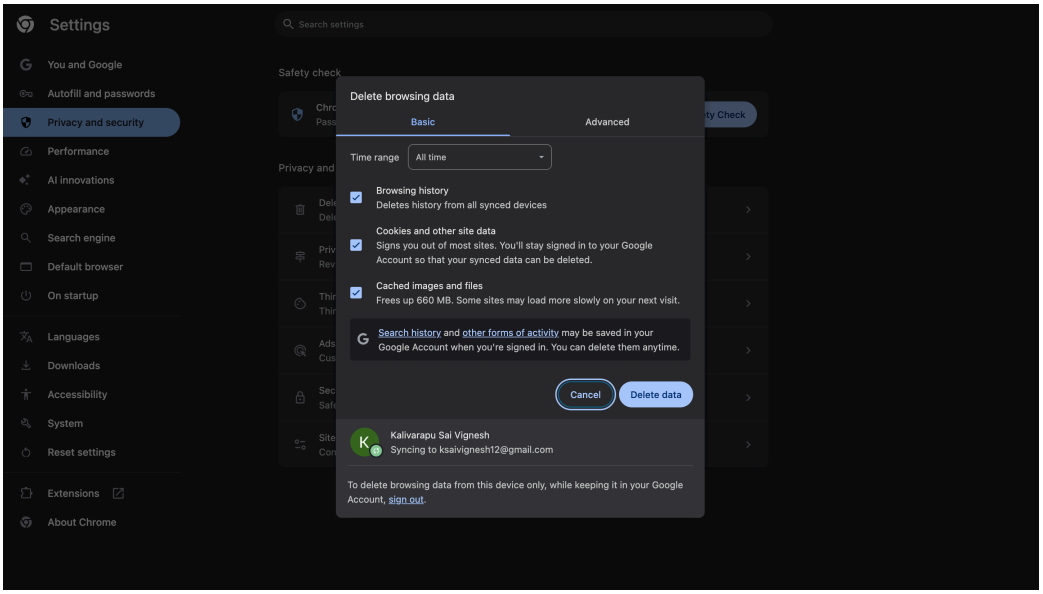
Conclusion:

Understanding and mitigating these vulnerabilities is essential for securing web applications. Developers and security teams should adopt best practices, conduct regular security testing, and stay updated on emerging threats.

LAB 9:- To study and implement various security measures to ensure the safe and secure usage of a web browser, protecting against cyber threats, data breaches, and malicious attacks.

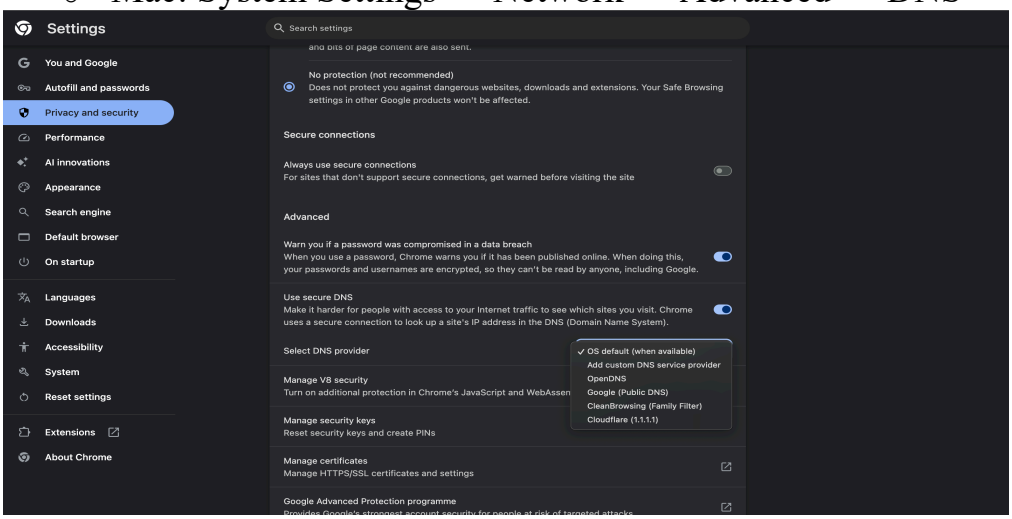
Clear Cookies and Cache Regularly

- Chrome: Settings → Privacy & Security → Clear browsing data
- Set browsers to **auto-delete cookies on exit**.



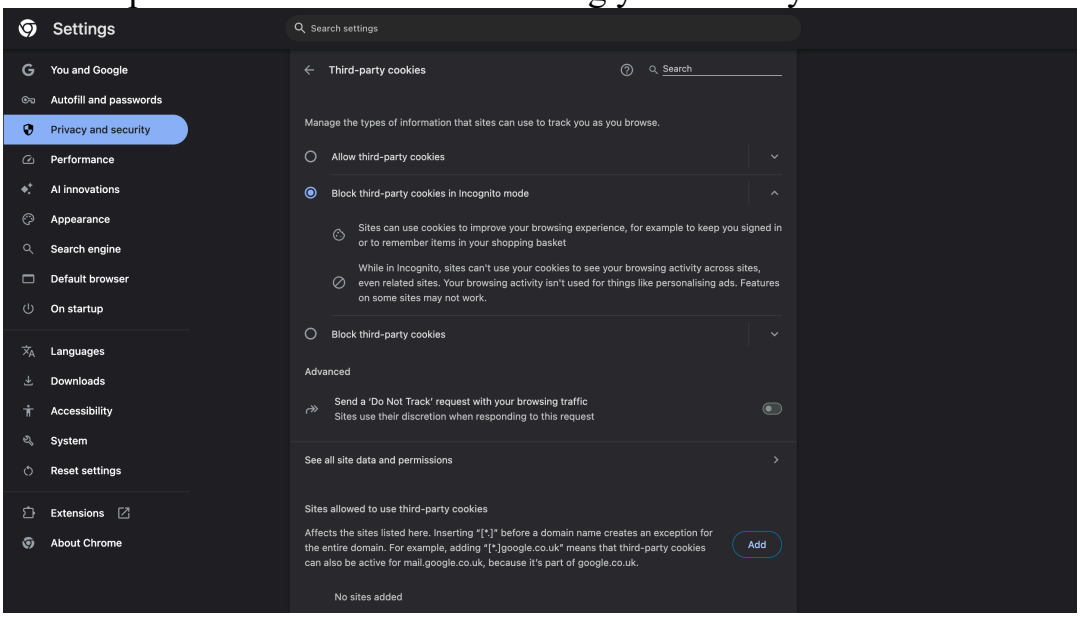
Use a Secure DNS Provider

- Change your DNS to **Cloudflare (1.1.1.1), Google (8.8.8.8), or OpenDNS (208.67.222.222)** for faster and more secure browsing.
- Change DNS settings:
  - Mac: System Settings → Network → Advanced → DNS



Disable Tracking and Third-Party Cookies

- Chrome: Settings → Privacy & Security → Cookies and other site data → Block third-party cookies
- This prevents websites from tracking your activity.



Block Pop-ups and Ads

- Chrome: Settings → Privacy and Security → Site Settings → Pop-ups and redirects (Block)
- Install **uBlock Origin** to block malicious ads.