**/**

**Assignment 1**

**PRN: 21510041**

**Name : Paras Wane**

**1. Perform encryption, decryption using the following substitution techniques:**

**a. Ceaser cipher**

**Ans:**

The Caesar Cipher is a simple encryption technique where each letter in a message is shifted by a fixed number of positions in the alphabet. For example, with a shift of 3, "A" becomes "D," "B" becomes "E," and so on. It's one of the oldest known ciphers and is easy to implement but also easy to break.

**Code:**

#include <iostream>

#include <string>

using namespace std;

// Function to encrypt the plaintext using Caesar Cipher

string encryptCaesarCipher(string plaintext, int shift) {

    string ciphertext = "";

    for (char& ch : plaintext) {

        if (isalpha(ch)) {

            char base = isupper(ch) ? 'A' : 'a';

            ch = ((ch - base) + shift) % 26 + base;

        }

        ciphertext += ch;

    }

    return ciphertext;

}

// Function to decrypt the ciphertext using Caesar Cipher

string decryptCaesarCipher(string ciphertext, int shift) {

    return encryptCaesarCipher(ciphertext, 26 - shift);

}

int main() {

    string plaintext, ciphertext;

    int shift;

    // Input plaintext and shift value

    cout << "Enter plaintext: ";

    getline(cin, plaintext);

    cout << "Enter shift: ";

    cin >> shift; // or key

    ciphertext = encryptCaesarCipher(plaintext, shift);

    cout << "Encrypted Text: " << ciphertext << endl;

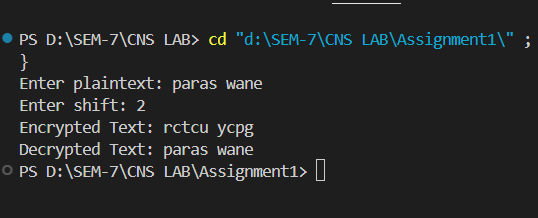
    string decryptedText = decryptCaesarCipher(ciphertext, shift);

    cout << "Decrypted Text: " << decryptedText << endl;

    return 0;

}

**Output:**



**Advantages:**

* **Simplicity**: Easy to understand and implement.
* **Efficiency**: Fast encryption and decryption.

**Disadvantages:**

* **Weak Security**: Vulnerable to frequency analysis and brute-force attacks (only 25 possible shifts).
* **Predictability**: Does not change much between different texts.

**b. Playfair cipher**

**Ans:**

The Playfair Cipher is a digraph substitution cipher that encrypts pairs of letters. It uses a 5x5 matrix of letters created from a keyword. To encrypt, locate each letter pair in the matrix and swap or substitute based on their positions. It’s more secure than simple substitution ciphers because it encodes pairs of letters rather than individual letters.

**code:**

#include <iostream>

#include <vector>

#include <algorithm>

#include <cctype>

using namespace std;

class PlayfairCipher {

private:

    char matrix[5][5];

    string key;

    // Function to remove duplicates from the key

    string removeDuplicates(string s) {

        string result = "";

        vector<bool> seen(26, false);

        for (char c : s) {

            if (!seen[c - 'A']) {

                seen[c - 'A'] = true;

                result += c;

            }

        }

        return result;

    }

    // Function to preprocess the text (remove non-letters, convert to uppercase, replace J with I)

    string preprocessText(string text) {

        text.erase(remove\_if(text.begin(), text.end(), [](char c) { return !isalpha(c); }), text.end());

        transform(text.begin(), text.end(), text.begin(), ::toupper);

        replace(text.begin(), text.end(), 'J', 'I');

        return text;

    }

    // Function to create the 5x5 matrix using the key

    void generateMatrix() {

        string matrixKey = removeDuplicates(key + "ABCDEFGHIKLMNOPQRSTUVWXYZ");

        vector<bool> seen(26, false);

        seen['J' - 'A'] = true; // J is omitted

        int k = 0;

        for (char c : matrixKey) {

            if (!seen[c - 'A']) {

                seen[c - 'A'] = true;

                matrix[k / 5][k % 5] = c;

                k++;

            }

        }

    }

    // Function to find the position of a character in the matrix

    void findPosition(char c, int &row, int &col) {

        for (int i = 0; i < 5; i++) {

            for (int j = 0; j < 5; j++) {

                if (matrix[i][j] == c) {

                    row = i;

                    col = j;

                    return;

                }

            }

        }

    }

    // Function to encrypt a pair of characters

    string encryptPair(char a, char b) {

        int row1, col1, row2, col2;

        findPosition(a, row1, col1);

        findPosition(b, row2, col2);

        if (row1 == row2) {

            // Same row: move to the right

            return string(1, matrix[row1][(col1 + 1) % 5]) + string(1, matrix[row2][(col2 + 1) % 5]);

        } else if (col1 == col2) {

            // Same column: move down

            return string(1, matrix[(row1 + 1) % 5][col1]) + string(1, matrix[(row2 + 1) % 5][col2]);

        } else {

            // Rectangle: swap columns

            return string(1, matrix[row1][col2]) + string(1, matrix[row2][col1]);

        }

    }

    // Function to decrypt a pair of characters

    string decryptPair(char a, char b) {

        int row1, col1, row2, col2;

        findPosition(a, row1, col1);

        findPosition(b, row2, col2);

        if (row1 == row2) {

            // Same row: move to the left

            return string(1, matrix[row1][(col1 + 4) % 5]) + string(1, matrix[row2][(col2 + 4) % 5]);

        } else if (col1 == col2) {

            // Same column: move up

            return string(1, matrix[(row1 + 4) % 5][col1]) + string(1, matrix[(row2 + 4) % 5][col2]);

        } else {

            // Rectangle: swap columns

            return string(1, matrix[row1][col2]) + string(1, matrix[row2][col1]);

        }

    }

public:

    PlayfairCipher(string key) {

        this->key = preprocessText(key);

        generateMatrix();

    }

    // Function to encrypt the plaintext

    string encrypt(string plaintext) {

        plaintext = preprocessText(plaintext);

        // Adjust plaintext to form pairs (inserting 'X' between identical letters and at the end if needed)

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

            if (i + 1 == plaintext.length() || plaintext[i] == plaintext[i + 1]) {

                plaintext.insert(i + 1, "X");

            }

        }

        string ciphertext = "";

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

            ciphertext += encryptPair(plaintext[i], plaintext[i + 1]);

        }

        return ciphertext;

    }

    // Function to decrypt the ciphertext

    string decrypt(string ciphertext) {

        ciphertext = preprocessText(ciphertext);

        string plaintext = "";

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

            plaintext += decryptPair(ciphertext[i], ciphertext[i + 1]);

        }

        // Optionally remove padding character 'X' (if inserted during encryption)

        for (int i = 0; i < plaintext.length(); i++) {

            if (i + 1 < plaintext.length() && plaintext[i] == 'X' && plaintext[i - 1] == plaintext[i + 1]) {

                plaintext.erase(i, 1);

            }

        }

        return plaintext;

    }

    // Function to display the matrix (for debugging)

    void displayMatrix() {

        cout << "Playfair Cipher Matrix:" << endl;

        for (int i = 0; i < 5; i++) {

            for (int j = 0; j < 5; j++) {

                cout << matrix[i][j] << ' ';

            }

            cout << endl;

        }

    }

};

int main() {

    string key, plaintext, ciphertext;

    cout << "Enter the key: ";

    getline(cin, key);

    PlayfairCipher playfair(key);

    cout << "Enter the plaintext: ";

    getline(cin, plaintext);

    ciphertext = playfair.encrypt(plaintext);

    cout << "Encrypted Text: " << ciphertext << endl;

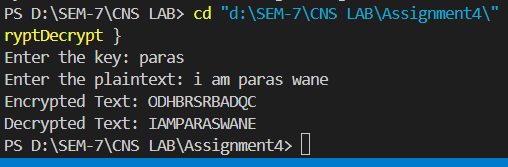
    string decryptedText = playfair.decrypt(ciphertext);

    cout << "Decrypted Text: " << decryptedText << endl;

    return 0;

}

**Output:**



**Advantages:**

* **Improved Security**: More secure than Caesar Cipher as it encrypts digraphs (pairs of letters).
* **Simplicity**: Slightly more complex but still relatively easy to implement.

**Disadvantages:**

* **Key Management**: Requires a good keyword and matrix setup.
* **Vulnerability**: Can still be broken with modern techniques like frequency analysis of digraphs.

**c. Vigenere cipher**

**Ans:**

The Vigenère Cipher is a method of encrypting text using a keyword. It works by shifting each letter in the plaintext by an amount determined by the corresponding letter in the keyword. The key repeats itself if it's shorter than the plaintext.

**How It Works:**

1. **Keyword**: Choose a keyword (e.g., "KEY").
2. **Encryption**:
   * Write the keyword repeatedly above the plaintext.
   * Shift each letter in the plaintext by the position of the corresponding letter in the keyword (A=0, B=1, ..., Z=25).
3. **Decryption**:
   * Use the same keyword to reverse the shifts and recover the plaintext.

**Python Code:**

def vigenere\_encrypt(plain\_text, key):

    """

    Encrypt the plain text using the Vigenere cipher.

    Parameters:

    plain\_text (str): The input text to be encrypted.

    key (str): The key for the Vigenere cipher.

    Returns:

    str: The encrypted text.

    """

    plain\_text = plain\_text.upper().replace(" ", "")

    key = key.upper().replace(" ", "")

    key\_length = len(key)

    encrypted\_text = ""

    for i, char in enumerate(plain\_text):

        if char.isalpha():

            shift = ord(key[i % key\_length]) - ord('A')

            encrypted\_char = chr((ord(char) - ord('A') + shift) % 26 + ord('A'))

            encrypted\_text += encrypted\_char

        else:

            encrypted\_text += char

    return encrypted\_text

def vigenere\_decrypt(cipher\_text, key):

    """

    Decrypt the cipher text using the Vigenere cipher.

    Parameters:

    cipher\_text (str): The input text to be decrypted.

    key (str): The key for the Vigenere cipher.

    Returns:

    str: The decrypted text.

    """

    cipher\_text = cipher\_text.upper().replace(" ", "")

    key = key.upper().replace(" ", "")

    key\_length = len(key)

    decrypted\_text = ""

    for i, char in enumerate(cipher\_text):

        if char.isalpha():

            shift = ord(key[i % key\_length]) - ord('A')

            decrypted\_char = chr((ord(char) - ord('A') - shift + 26) % 26 + ord('A'))

            decrypted\_text += decrypted\_char

        else:

            decrypted\_text += char

    return decrypted\_text

def main():

    """

    The main function to run the menu-driven program.

    """

    while True:

        print("\nVigenere Cipher Program")

        print("1. Encrypt")

        print("2. Decrypt")

        print("3. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            plain\_text = input("\nEnter the plain text: ")

            key = input("Enter the key: ")

            encrypted\_text = vigenere\_encrypt(plain\_text, key)

            print(f"\nEncrypted Text: {encrypted\_text}")

        elif choice == '2':

            encrypted\_text = input("\nEnter the encrypted text: ")

            key = input("Enter the key: ")

            decrypted\_text = vigenere\_decrypt(encrypted\_text, key)

            print(f"\nDecrypted Text: {decrypted\_text}")

        elif choice == '3':

            print("Exiting the program.")

            break

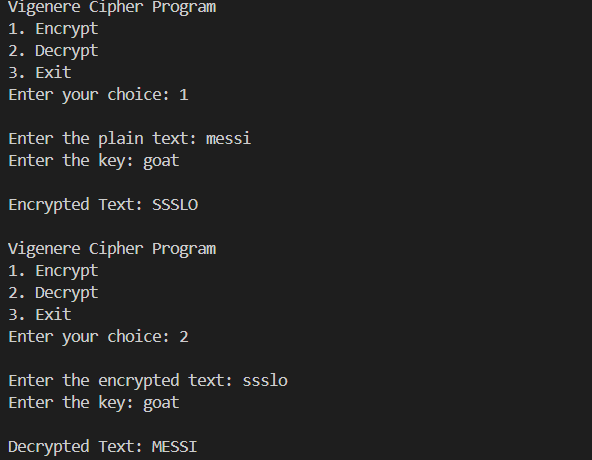
        else:

            print("Invalid choice. Please try again.")

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

    main()

**Output:**

****

**Advantages:**

* **Polyalphabetic**: Uses a keyword to shift letters, making it more secure than Caesar Cipher.
* **Improved Security**: Harder to crack with frequency analysis if the keyword is long and complex.

**Disadvantages:**

* **Keyword Management**: Security depends on the keyword length and complexity.
* **Vulnerabilities**: Can be broken with techniques like the Kasiski examination or frequency analysis if the keyword is short.

**ASSIGNMENT 2**

**PRN:21510041**

**NAME : Paras Wane**

**1. Perform encryption and decryption using following transposition techniques**

**a. Rail fence**

**Ans:**

The Rail Fence Cipher is a type of transposition cipher where the plain text is written in a zigzag pattern across multiple "rails" (rows) and then read row by row to create the cipher text. Decryption involves reconstructing the zigzag pattern to retrieve the original message.

**code:**

#include <iostream>

#include <string>

#include <vector>

using namespace std;

// Function to encrypt the plaintext using Rail Fence Cipher

string encryptRailFenceCipher(string plaintext, int key) {

    if (key == 1) {

        return plaintext;  // No change if key is 1

    }

    vector<string> rail(key);

    int row = 0;

    bool directionDown = true;

    // Place characters in a zigzag pattern across the rails

    for (char& ch : plaintext) {

        rail[row] += ch;

        if (row == 0) {

            directionDown = true;

        } else if (row == key - 1) {

            directionDown = false;

        }

        row += directionDown ? 1 : -1;

    }

    // Read the rails line by line to create the ciphertext

    string ciphertext = "";

    for (const string& line : rail) {

        ciphertext += line;

    }

    return ciphertext;

}

// Function to decrypt the ciphertext using Rail Fence Cipher

string decryptRailFenceCipher(string ciphertext, int key) {

    if (key == 1) {

        return ciphertext;  // No change if key is 1

    }

    vector<string> rail(key);

    vector<int> index(key, 0);

    int length = ciphertext.length();

    int row = 0;

    bool directionDown = true;

    // Determine the length of each rail

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

        index[row]++;

        if (row == 0) {

            directionDown = true;

        } else if (row == key - 1) {

            directionDown = false;

        }

        row += directionDown ? 1 : -1;

    }

    // Fill the rails with the ciphertext

    int pos = 0;

    for (int r = 0; r < key; r++) {

        rail[r] = ciphertext.substr(pos, index[r]);

        pos += index[r];

    }

    // Read the rails in a zigzag pattern to decrypt the text

    string plaintext = "";

    row = 0;

    directionDown = true;

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

        plaintext += rail[row][0];

        rail[row].erase(rail[row].begin());

        if (row == 0) {

            directionDown = true;

        } else if (row == key - 1) {

            directionDown = false;

        }

        row += directionDown ? 1 : -1;

    }

    return plaintext;

}

int main() {

    string plaintext, ciphertext;

    int key;

    // Input plaintext and key

    cout << "Enter plaintext: ";

    getline(cin, plaintext);

    cout << "Enter key (number of rails): ";

    cin >> key;

    // Encrypt the plaintext

    ciphertext = encryptRailFenceCipher(plaintext, key);

    cout << "Encrypted Text: " << ciphertext << endl;

    // Decrypt the ciphertext

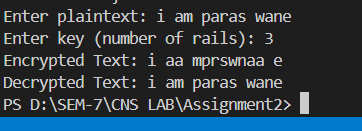
    string decryptedText = decryptRailFenceCipher(ciphertext, key);

    cout << "Decrypted Text: " << decryptedText << endl;

    return 0;

}

**Output:**



**Advantages:**

* **Simplicity**: Easy to understand and implement.
* **Low Computation**: Requires minimal computational resources for encryption and decryption.

**Disadvantages:**

* **Weak Security**: Very easy to break with simple analysis or known-plaintext attacks.
* **Pattern Recognition**: The regular zigzag pattern makes it susceptible to pattern recognition, which can be exploited to decode the message.

**b. row and Column Transformation**

**Ans:**

Row and column transformation is a type of transposition cipher where the message is written in a grid (matrix) and the order of rows and columns is changed according to a key.

**Row Transposition**: Encrypts text by writing it into rows of a grid, then permuting the columns according to a specific key.

**Column Transposition**: Encrypts text by writing it into columns of a grid, then permuting the rows according to a specific key.

**How It Works**:

1. **Write** the plaintext into a grid according to the number of rows or columns.
2. **Permute** the rows or columns based on the key.
3. **Read** off the text in the new order to get the ciphertext.

**code:**

#include <iostream>

#include <algorithm>

#include <string>

#include <vector>

using namespace std;

// Function to generate the key sequence based on the keyword

vector<int> generateKeySequence(string keyword) {

    int len = keyword.length();

    vector<pair<char, int>> keyMap(len);

    // Store characters of the keyword with their positions

    for (int i = 0; i < len; i++) {

        keyMap[i] = { keyword[i], i };

    }

    // Sort the characters alphabetically along with their positions

    sort(keyMap.begin(), keyMap.end());

    // Extract the positions in the sorted order

    vector<int> keySequence(len);

    for (int i = 0; i < len; i++) {

        keySequence[i] = keyMap[i].second;

    }

    return keySequence;

}

// Function to encrypt the plaintext using Columnar Transposition Cipher

string encryptColumnarCipher(string plaintext, string keyword) {

    int numRows = (plaintext.length() + keyword.length() - 1) / keyword.length();

    int numCols = keyword.length();

    vector<int> keySequence = generateKeySequence(keyword);

    // Fill the matrix row-wise

    vector<vector<char>> grid(numRows, vector<char>(numCols, ' '));

    for (int i = 0; i < plaintext.length(); i++) {

        grid[i / numCols][i % numCols] = plaintext[i];

    }

    // Read the grid column-wise based on the key sequence

    string ciphertext = "";

    for (int col : keySequence) {

        for (int row = 0; row < numRows; row++) {

            ciphertext += grid[row][col];

        }

    }

    return ciphertext;

}

// Function to decrypt the ciphertext using Columnar Transposition Cipher

string decryptColumnarCipher(string ciphertext, string keyword) {

    int numRows = (ciphertext.length() + keyword.length() - 1) / keyword.length();

    int numCols = keyword.length();

    vector<int> keySequence = generateKeySequence(keyword);

    // Fill the grid column-wise based on the key sequence

    vector<vector<char>> grid(numRows, vector<char>(numCols, ' '));

    int index = 0;

    for (int col : keySequence) {

        for (int row = 0; row < numRows; row++) {

            if (index < ciphertext.length()) {

                grid[row][col] = ciphertext[index++];

            }

        }

    }

    // Read the grid row-wise to get the plaintext

    string plaintext = "";

    for (int row = 0; row < numRows; row++) {

        for (int col = 0; col < numCols; col++) {

            plaintext += grid[row][col];

        }

    }

    return plaintext;

}

int main() {

    string plaintext, keyword, ciphertext;

    // Input plaintext and keyword

    cout << "Enter plaintext: ";

    getline(cin, plaintext);

    cout << "Enter keyword: ";

    cin >> keyword;

    // Encrypt the plaintext

    ciphertext = encryptColumnarCipher(plaintext, keyword);

    cout << "Encrypted Text: " << ciphertext << endl;

    // Decrypt the ciphertext

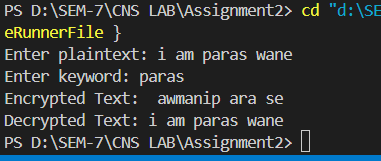
    string decryptedText = decryptColumnarCipher(ciphertext, keyword);

    cout << "Decrypted Text: " << decryptedText << endl;

    return 0;

}

**Output:**



**Advantages**:

* **Increased Security**: More complex than simple transpositions.
* **Flexibility**: Key-based rearrangement can add security.

**Disadvantages**:

* **Complexity**: Can be more complex to implement and manage compared to simple ciphers.
* **Pattern Recognition**: Still susceptible to pattern analysis if not combined with other encryption methods.

**Assignment 3**

**PRN:** 21510041

**Name: Paras Wane**

1. **Implementation of Euclidean and Extended Euclidean Algorithm**

**Ans:**

The Euclidean and Extended Euclidean algorithms are essential for finding the greatest common divisor (GCD) of two integers. The Extended Euclidean algorithm also finds the coefficients of Bézout's identity, which are useful in solving linear Diophantine equations and in modular arithmetic.

**Euclidean Algorithm**

The Euclidean algorithm finds the GCD of two numbers by repeatedly applying the following rule: gcd(a, b) = gcd(b, a % b) until b becomes zero. The GCD is then the non-zero remainder.

**Extended Euclidean Algorithm**

The Extended Euclidean algorithm not only computes the GCD of two integers a and b, but also finds integers x and y such that ax + by = gcd(a, b).

**Code:**#include <iostream>

using namespace std;

// Function to compute GCD using the Euclidean algorithm

int euclidean\_algorithm(int a, int b) {

    while (b != 0) {

        int temp = b;

        b = a % b;

        a = temp;

    }

    return a;

}

// Function to compute GCD and coefficients x and y using Extended Euclidean algorithm

int extended\_euclidean\_algorithm(int a, int b, int &x, int &y) {

    if (b == 0) {

        x = 1;

        y = 0;

        return a;

    }

    int x1, y1;

    int gcd = extended\_euclidean\_algorithm(b, a % b, x1, y1);

    x = y1;

    y = x1 - (a / b) \* y1;

    return gcd;

}

int main() {

    while (true) {

        cout << "\nEuclidean and Extended Euclidean Algorithm" << endl;

        cout << "1. Compute GCD using Euclidean Algorithm" << endl;

        cout << "2. Compute GCD and coefficients using Extended Euclidean Algorithm" << endl;

        cout << "3. Exit" << endl;

        int choice;

        cout << "Enter your choice: ";

        cin >> choice;

        if (choice == 1) {

            int a, b;

            cout << "\nEnter the first integer (a): ";

            cin >> a;

            cout << "Enter the second integer (b): ";

            cin >> b;

            int gcd = euclidean\_algorithm(a, b);

            cout << "\nGCD of " << a << " and " << b << " is: " << gcd << endl;

        } else if (choice == 2) {

            int a, b;

            cout << "\nEnter the first integer (a): ";

            cin >> a;

            cout << "Enter the second integer (b): ";

            cin >> b;

            int x, y;

            int gcd = extended\_euclidean\_algorithm(a, b, x, y);

            cout << "\nGCD of " << a << " and " << b << " is: " << gcd << endl;

            cout << "Coefficients x and y are: x = " << x << ", y = " << y << endl;

            cout << "\nBézout's identity: " << a << "\*(" << x << ") + " << b << "\*(" << y << ") = " << gcd << endl;

        } else if (choice == 3) {

            cout << "Exiting the program." << endl;

            break;

        } else {

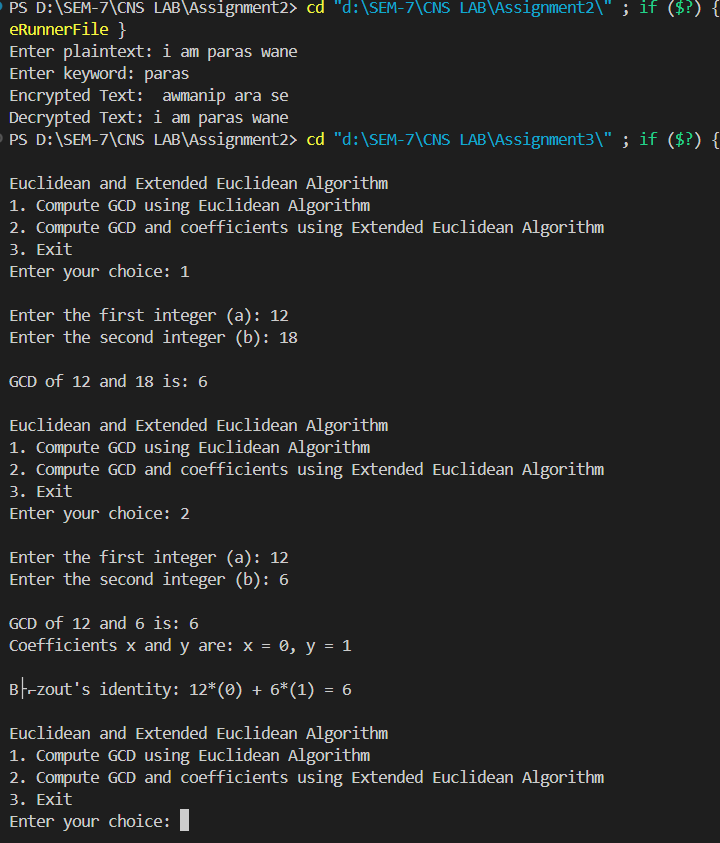
            cout << "Invalid choice. Please try again." << endl;

        }

    }

    return 0;

}

**Output:**

This implementation of the Euclidean and Extended Euclidean algorithms is fundamental in cryptography, number theory, and algorithms related to modular arithmetic.

1. Impliment brutforce attack

#include <iostream>

#include <string>

using namespace std;

// Function to decrypt a single shift in Caesar Cipher

string decryptWithKey(const string &cipherText, int key) {

    string result = "";

    for (char c : cipherText) {

        if (isalpha(c)) {

            char base = isupper(c) ? 'A' : 'a';

            result += char(int(base + (c - base - key + 26) % 26));

        } else {

            result += c; // Non-alphabet characters remain unchanged

        }

    }

    return result;

}

// Function to brute-force all possible keys in Caesar Cipher

void bruteForceCaesarCipher(const string &cipherText) {

    for (int key = 0; key < 26; key++) {

        string decryptedText = decryptWithKey(cipherText, key);

        cout << "Key " << key << ": " << decryptedText << endl;

    }

}

int main() {

    string cipherText;

    cout << "Enter the cipher text: ";

    getline(cin, cipherText);

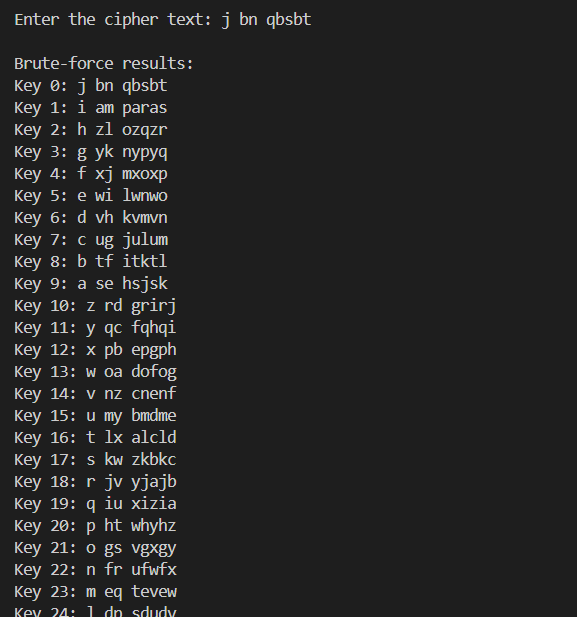
    cout << "Brute-force results:" << endl;

    bruteForceCaesarCipher(cipherText);

    return 0;

}

Output:-



**Assignment 4**

**PRN:** 21510041  **Name: Paras Wane**

1. **Implementation of Chinese Remainder Theorem (CRT)**

**Ans:**

The Chinese Remainder Theorem (CRT) is a powerful tool in number theory that provides a solution to a system of simultaneous congruences with pairwise coprime moduli. Given a system of congruences, the CRT allows us to find a unique solution modulo the product of the moduli.

**Problem Description**

Given n congruences: x≡a1 (mod m1), x≡a2 (mod m2) ⋮x ≡ an (mod mn)

Where the moduli m1, m2, …, mn are pairwise coprime, the CRT provides a unique solution modulo M=m1×m2×⋯× mn.

For each congruence x ≡ ai (mod mi), it calculates the partial solution using the formula: x ≡ ai × Mi × inverse(Mi, mi) (mod M) where Mi=M/mi

The final solution is obtained by summing all partial solutions modulo M.

**Python code:**

def extended\_euclidean\_algorithm(a, b):

    """

    Compute the GCD of a and b, as well as the coefficients x and y

    such that ax + by = gcd(a, b) using the Extended Euclidean algorithm.

    Parameters:

    a (int): First integer.

    b (int): Second integer.

    Returns:

    tuple: (gcd, x, y) where gcd is the GCD of a and b, and x, y are

    the coefficients of Bézout's identity.

    """

    if b == 0:

        return a, 1, 0

    else:

        gcd, x1, y1 = extended\_euclidean\_algorithm(b, a % b)

        x = y1

        y = x1 - (a // b) \* y1

        return gcd, x, y

def chinese\_remainder\_theorem(a, m):

    """

    Solve the system of congruences using the Chinese Remainder Theorem.

    Parameters:

    a (list): List of remainders.

    m (list): List of moduli (must be pairwise coprime).

    Returns:

    int: The smallest non-negative solution to the system of congruences.

    """

    assert len(a) == len(m), "The number of remainders and moduli must be the same"

    # Calculate the product of all moduli

    M = 1

    for mi in m:

        M \*= mi

    # Initialize the solution

    x = 0

    # Apply the CRT

    for ai, mi in zip(a, m):

        Mi = M // mi  # M\_i = M / m\_i

        gcd, inverse, \_ = extended\_euclidean\_algorithm(Mi, mi)

        if gcd != 1:

            raise ValueError("Moduli are not pairwise coprime")

        x += ai \* inverse \* Mi

    return x % M

def main():

    """

    The main function to run the program.

    """

    while True:

        print("\nChinese Remainder Theorem (CRT)")

        print("1. Solve System of Congruences")

        print("2. Exit")

        choice = input("Enter your choice: ")

        if choice == '1':

            n = int(input("\nEnter the number of congruences: "))

            a = []

            m = []

            for i in range(n):

                ai = int(input(f"\nEnter remainder a[{i+1}]: "))

                mi = int(input(f"Enter modulus m[{i+1}]: "))

                a.append(ai)

                m.append(mi)

            solution = chinese\_remainder\_theorem(a, m)

            print(f"\nThe solution to the system of congruences is: {solution}")

        elif choice == '2':

            print("Exiting the program.")

            break

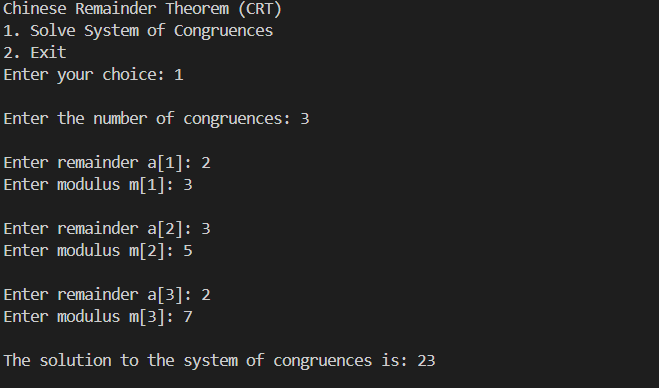
        else:

            print("Invalid choice. Please try again.")

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

    main()

**Output:**



**Assignment 5**

**PRN:** 21510041  **Name: Paras Wane**

1. **Apply DES algorithm for practical applications**

**Ans:**The Data Encryption Standard (DES) is a symmetric-key algorithm for the encryption of digital data. Although DES is now considered insecure for many applications due to its small key size, it is still an important algorithm for understanding the basics of cryptography.

**Practical Application of DES Algorithm**

To apply the DES algorithm in a practical application, we can use the **pycryptodome** library in Python, which provides an implementation of DES. Below is an example that demonstrates how to use DES to encrypt and decrypt a message.

**Python Code:**

from Crypto.Cipher import DES

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get\_random\_bytes

def des\_encrypt(plain\_text, key):

    """

    Encrypt the plain text using DES algorithm.

    Parameters:

    plain\_text (str): The text to be encrypted.

    key (bytes): The encryption key (must be 8 bytes long).

    Returns:

    bytes: The encrypted cipher text.

    """

    cipher = DES.new(key, DES.MODE\_ECB)

    padded\_text = pad(plain\_text.encode(), DES.block\_size)

    encrypted\_text = cipher.encrypt(padded\_text)

    return encrypted\_text

def des\_decrypt(cipher\_text, key):

    """

    Decrypt the cipher text using DES algorithm.

    Parameters:

    cipher\_text (bytes): The encrypted text to be decrypted.

    key (bytes): The decryption key (must be 8 bytes long).

    Returns:

    str: The decrypted plain text.

    """

    cipher = DES.new(key, DES.MODE\_ECB)

    decrypted\_text = unpad(cipher.decrypt(cipher\_text), DES.block\_size)

    return decrypted\_text.decode()

def main():

    """

    The main function to run the program.

    """

    print("\nDES Encryption and Decryption")

    # Generate a random 8-byte key for DES

    key = get\_random\_bytes(8)

    print(f"\nGenerated Key (in hexadecimal): {key.hex()}")

    # Input plaintext

    plain\_text = input("Enter the plain text to encrypt: ")

    # Encrypt the plaintext

    encrypted\_text = des\_encrypt(plain\_text, key)

    print(f"\nEncrypted Text (in hexadecimal): {encrypted\_text.hex()}")

    # Decrypt the ciphertext

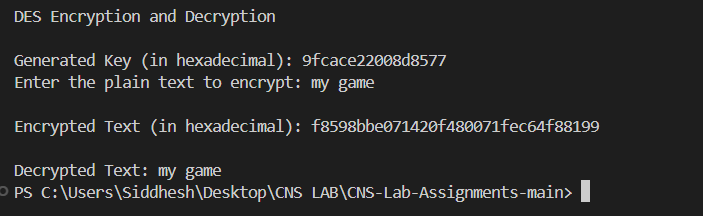
    decrypted\_text = des\_decrypt(encrypted\_text, key)

    print(f"\nDecrypted Text: {decrypted\_text}")

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

    main()

**Output:**

****

**Practical Applications:**

* **File Encryption:** DES can be used to encrypt sensitive files before storing them in insecure locations.
* **Secure Communication:** DES ensures that messages sent over a network are unreadable to unauthorized parties.
* **Password Storage:** Encrypting passwords before storing them in databases (though modern standards recommend stronger algorithms like AES).

While DES itself is outdated and not recommended for secure applications, understanding how it works is crucial for grasping more advanced encryption algorithms like AES.

**Assignment 6**

**PRN:** 21510041  **Name: Paras Wane**

1. **Apply AES algorithm for practical applications**

Ans:

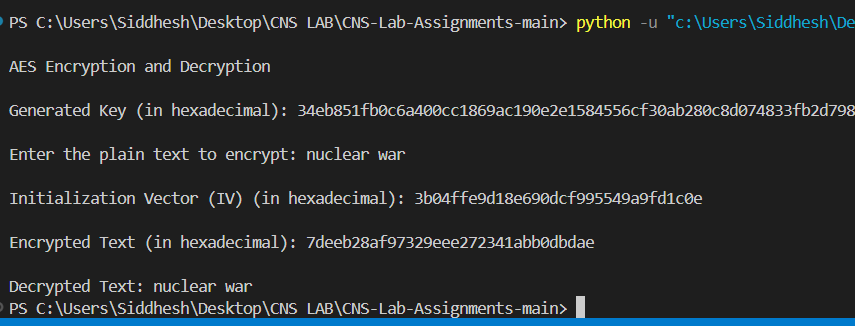
The Advanced Encryption Standard (AES) is a widely used symmetric encryption algorithm that is both fast and secure. It is the standard encryption algorithm used by governments, financial institutions, and many other organizations. Unlike DES, which is now considered insecure, AES is robust and provides a high level of security.

**Practical Application of AES Algorithm**

We can use the **pycryptodome** library in Python to implement AES encryption and decryption. The AES algorithm can work with key sizes of 128, 192, or 256 bits, and it operates on 128-bit blocks. In this example, we'll use AES with a 256-bit key in Cipher Block Chaining (CBC) mode.

**Code:**

**Output:**



**Practical Applications of AES:**

* **File Encryption:** Encrypting sensitive files before storing them on disk.
* **Secure Communication:** Ensuring that data sent over the network remains confidential.
* **Data Protection in Applications:** Encrypting user data, such as passwords, to protect them from unauthorized access.

AES is widely adopted due to its strength and efficiency, and it remains the standard for securing digital data across various industries.

**Assignment 7**

**PRN:** 21510041  **Name: Paras Wane**

**1. Implementation of RSA Algorithm**

**Ans:**

The RSA algorithm is one of the first public-key cryptosystems and is widely used for secure data transmission. It is an asymmetric cryptographic algorithm, meaning it uses a pair of keys: a public key for encryption and a private key for decryption. It relies on the mathematical properties of prime numbers.

**How RSA Works:**

1. **Key Generation:**
   * Choose two large prime numbers p and q.
   * Compute n = p \* q.
   * Compute the totient φ(n) = (p-1) \* (q-1).
   * Choose an encryption key e such that 1 < e < φ(n) and gcd(e, φ(n)) = 1. The integer e is the public key exponent.
   * Calculate the decryption key d such that d \* e ≡ 1 (mod φ(n)). The integer d is the private key exponent.
2. **Encryption:**
   * The public key is (n, e).
   * Given a plaintext message M, the ciphertext C is computed as:

C = M^ e mod n.

1. **Decryption:**
   * The private key is (n, d).
   * Given a ciphertext C, the plaintext M is recovered as:

M = C^d mod n

To implement the RSA algorithm **using large prime numbers with 2048 bits** and converting plaintext into numbers, we'll use the **Crypto library in Python**, which provides the **necessary tools to handle such large prime numbers and perform RSA encryption and decryption.**

**The large primes and the strong key sizes make RSA secure against most attacks when implemented correctly.**

**Python Code:**

import random

from sympy import isprime, mod\_inverse

def generate\_prime\_candidate(length):

    """Generate an odd integer randomly."""

    p = random.getrandbits(length)

    # Ensure p is odd

    p |= (1 << length - 1) | 1

    return p

def generate\_prime\_number(length):

    """Generate a prime number."""

    p = 4

    while not isprime(p):

        p = generate\_prime\_candidate(length)

    return p

def generate\_keypair(keysize):

    """Generate RSA public and private keys."""

    # Generate two large primes p and q

    p = generate\_prime\_number(keysize)

    q = generate\_prime\_number(keysize)

    print("\np: ", p)

    print("\nq: ", q)

    # Compute n = p \* q

    n = p \* q

    # Compute Euler's Totient φ(n) = (p-1)\*(q-1)

    phi = (p - 1) \* (q - 1)

    # Choose an integer e such that 1 < e < phi(n) and gcd(e, phi(n)) = 1

    e = random.randrange(2, phi)

    g = gcd(e, phi)

    while g != 1:

        e = random.randrange(2, phi)

        g = gcd(e, phi)

    # Compute d, the modular inverse of e

    d = mod\_inverse(e, phi)

    # Public key (e, n) and Private key (d, n)

    return ((e, n), (d, n))

def gcd(a, b):

    """Compute the greatest common divisor using Euclid's algorithm."""

    while b != 0:

        a, b = b, a % b

    return a

def encrypt(public\_key, plaintext):

    """Encrypt plaintext using the public key."""

    e, n = public\_key

    cipher = [pow(ord(char), e, n) for char in plaintext]

    return cipher

def decrypt(private\_key, ciphertext):

    """Decrypt ciphertext using the private key."""

    d, n = private\_key

    plain = [chr(pow(char, d, n)) for char in ciphertext]

    return ''.join(plain)

def main():

    """Run RSA algorithm."""

    print("RSA Encryption/Decryption")

    keysize = 2048  # Keysize in bits

    # Generate public and private keys

    public\_key, private\_key = generate\_keypair(keysize)

    print(f"\nPublic key: {public\_key}")

    print(f"Private key: {private\_key}")

    # Input plaintext

    plaintext = input("\nEnter a message to encrypt: ")

    # Encrypt the message

    encrypted\_msg = encrypt(public\_key, plaintext)

    print(f"\nEncrypted message: {encrypted\_msg}")

    # Decrypt the message

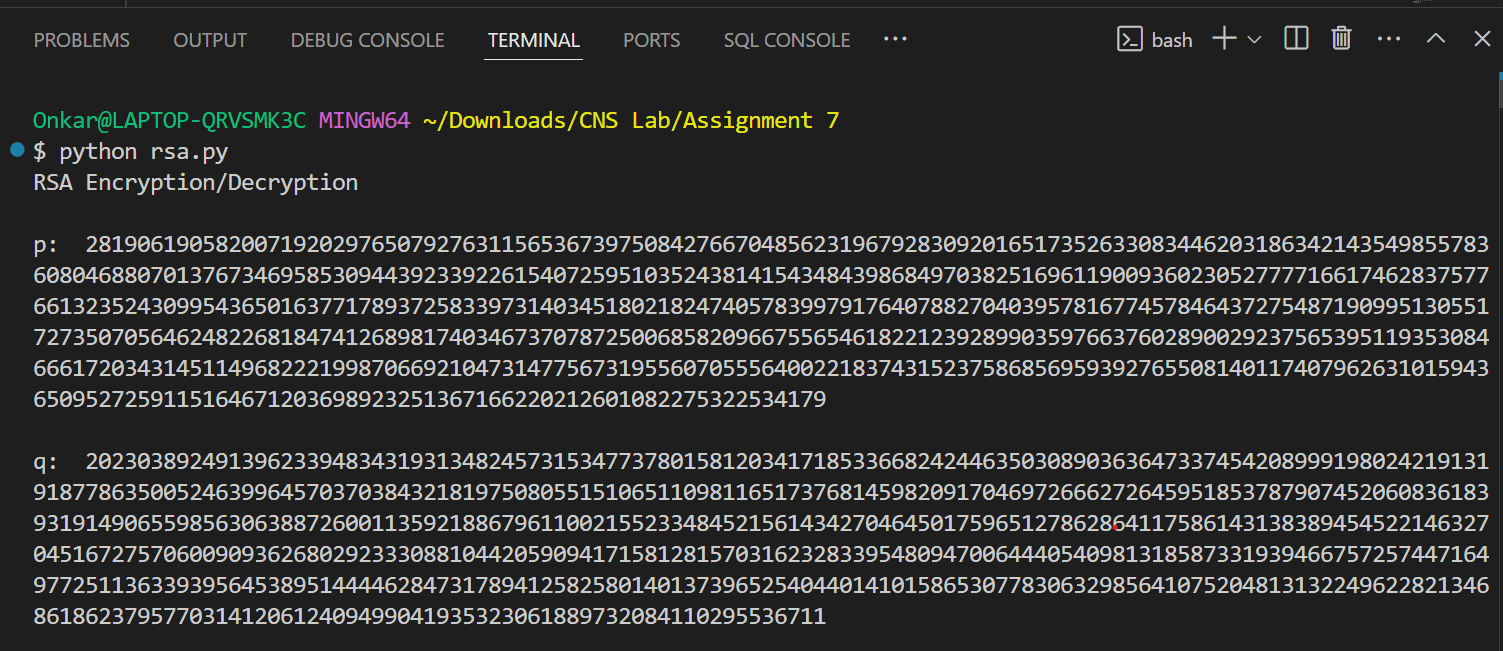
    decrypted\_msg = decrypt(private\_key, encrypted\_msg)

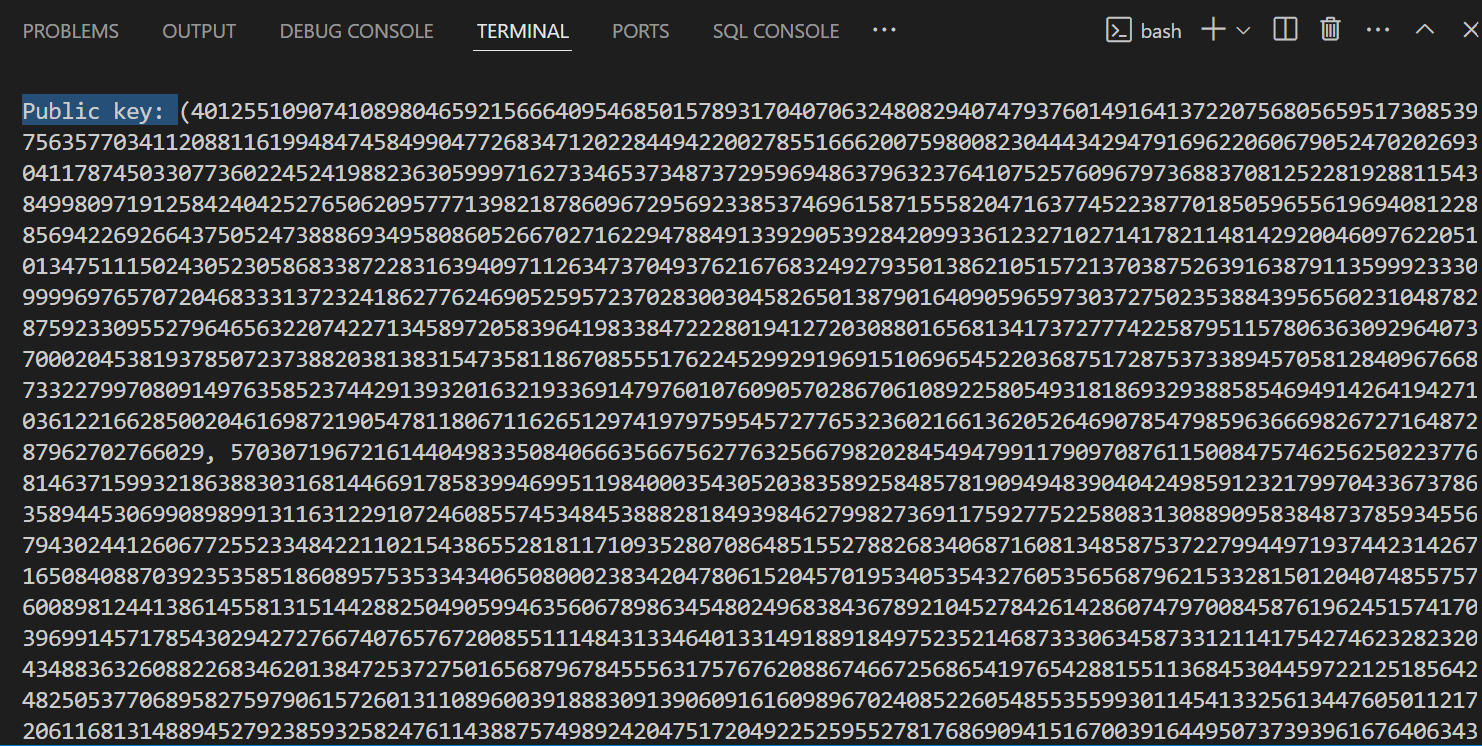
    print(f"\nDecrypted message: {decrypted\_msg}")

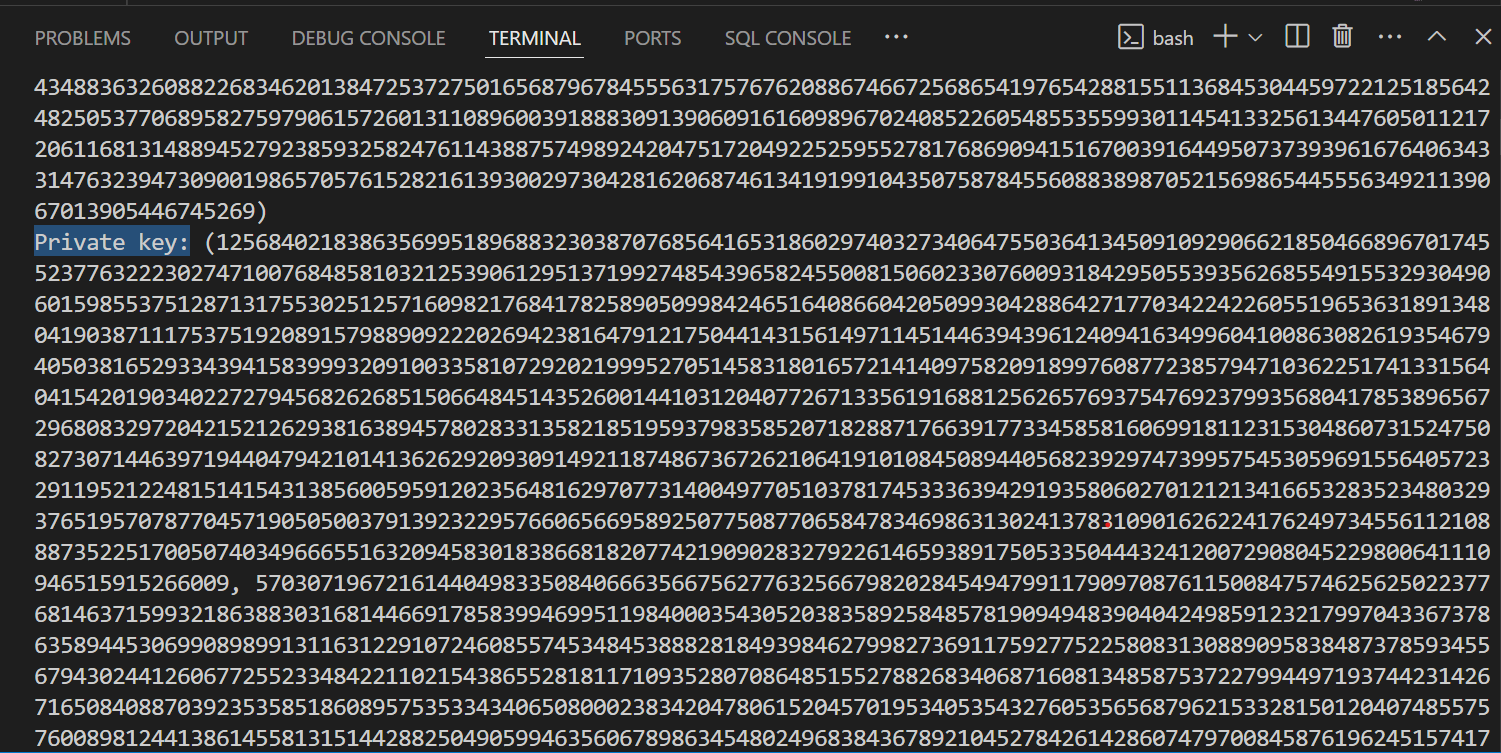
if \_\_name\_\_ == "\_\_main\_\_":

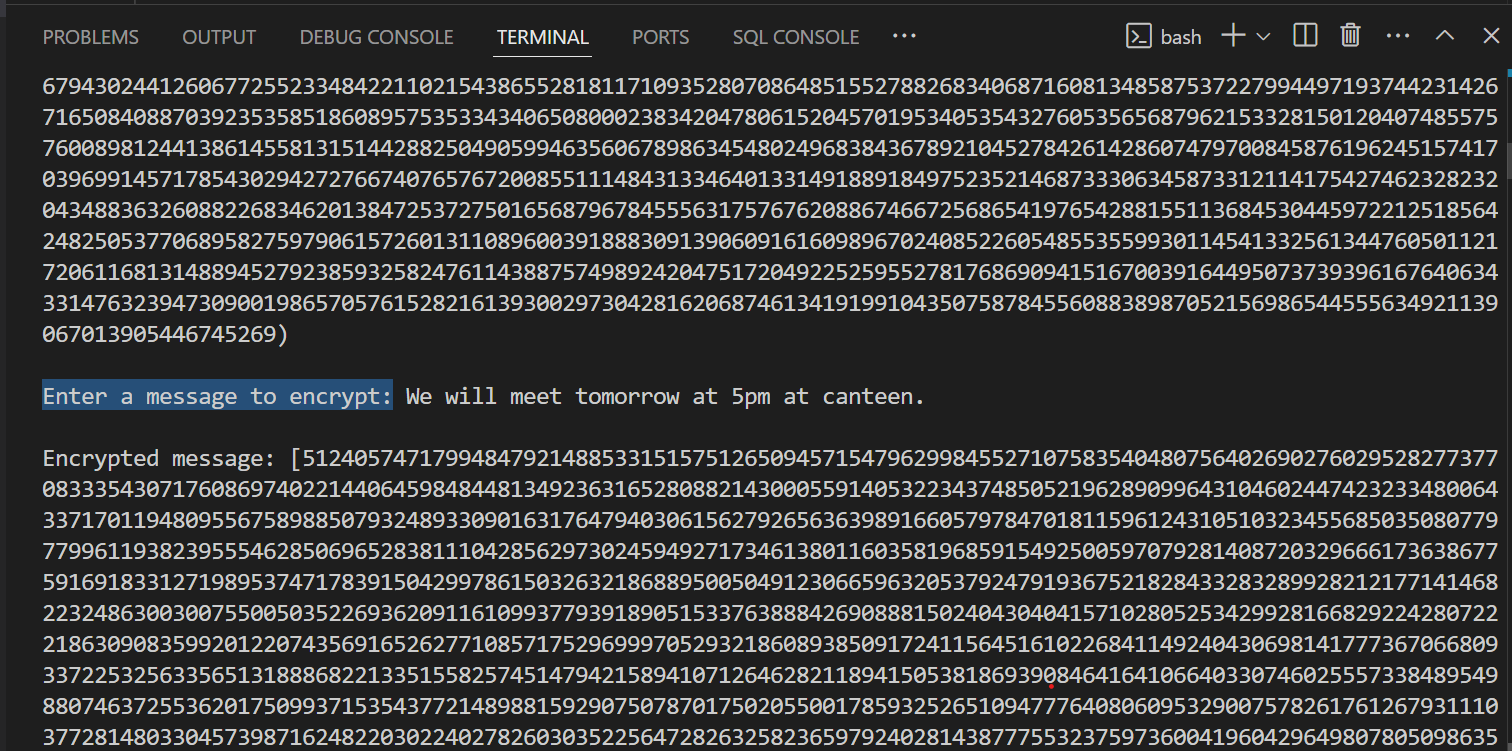
    main()

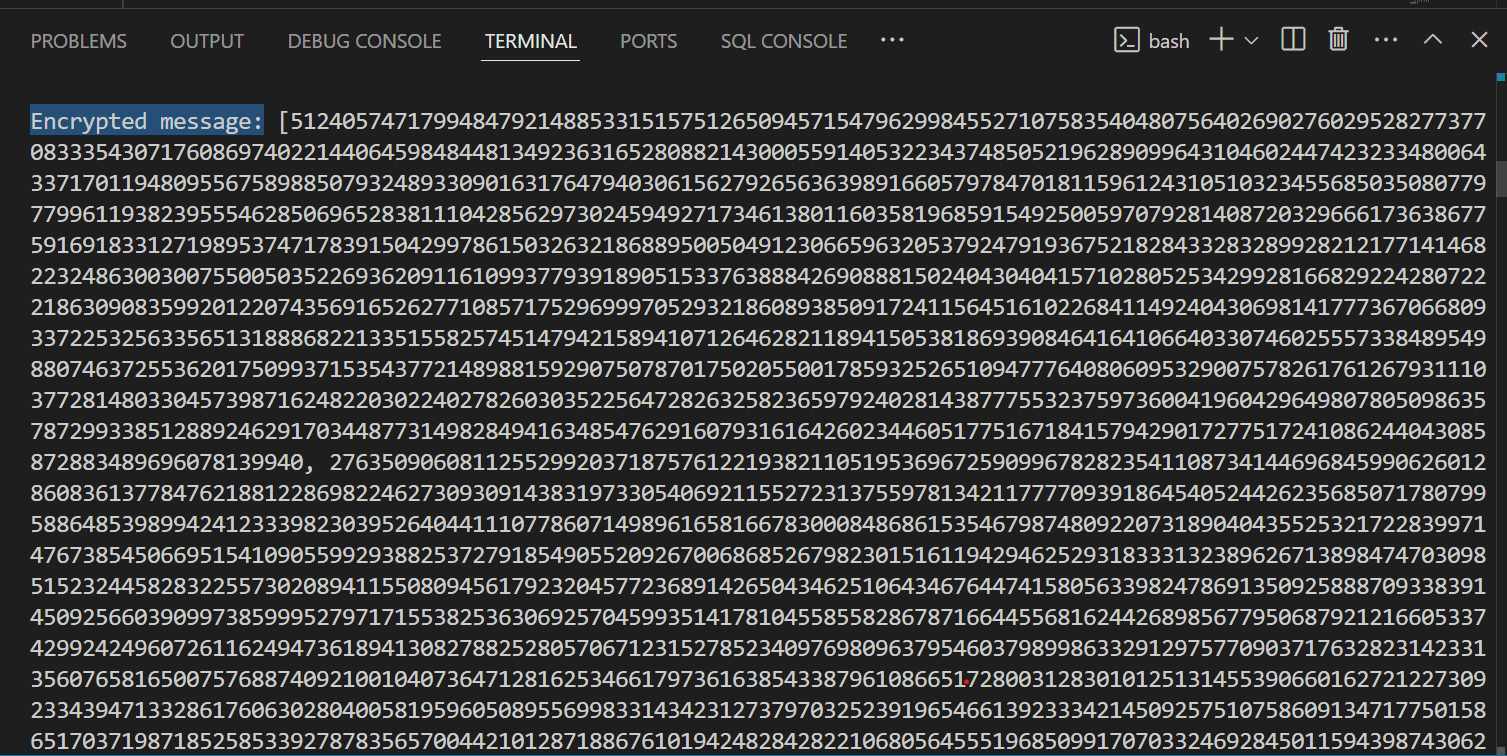
**Output:**

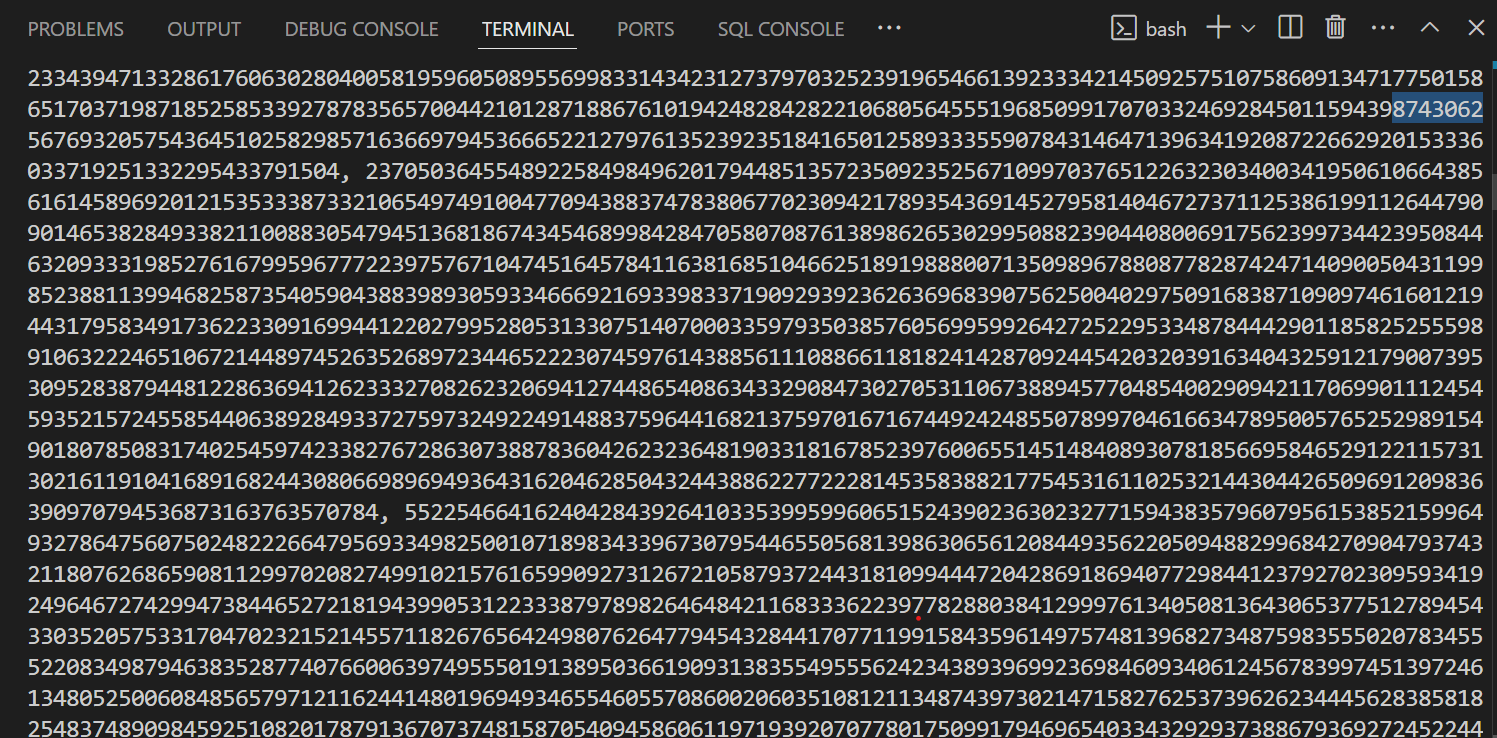


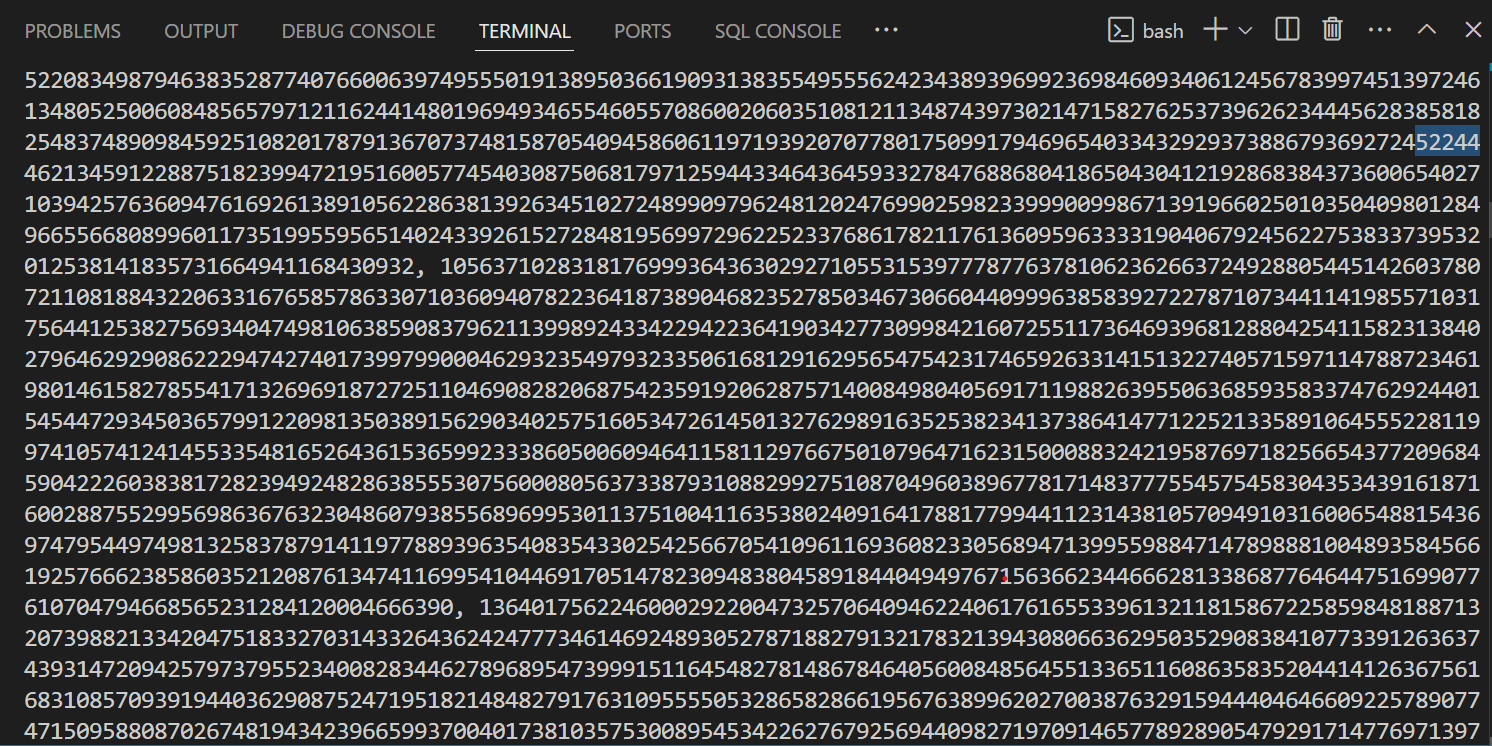


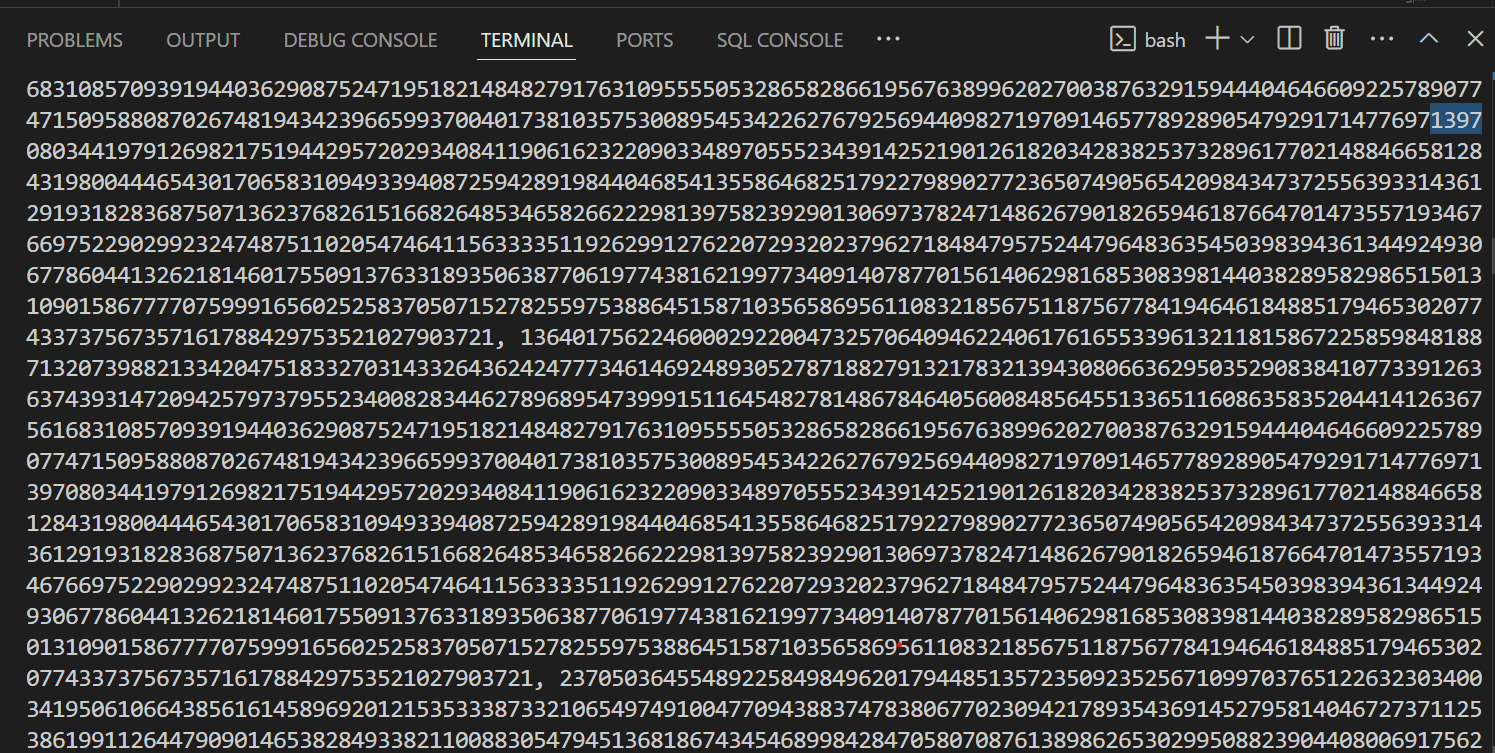


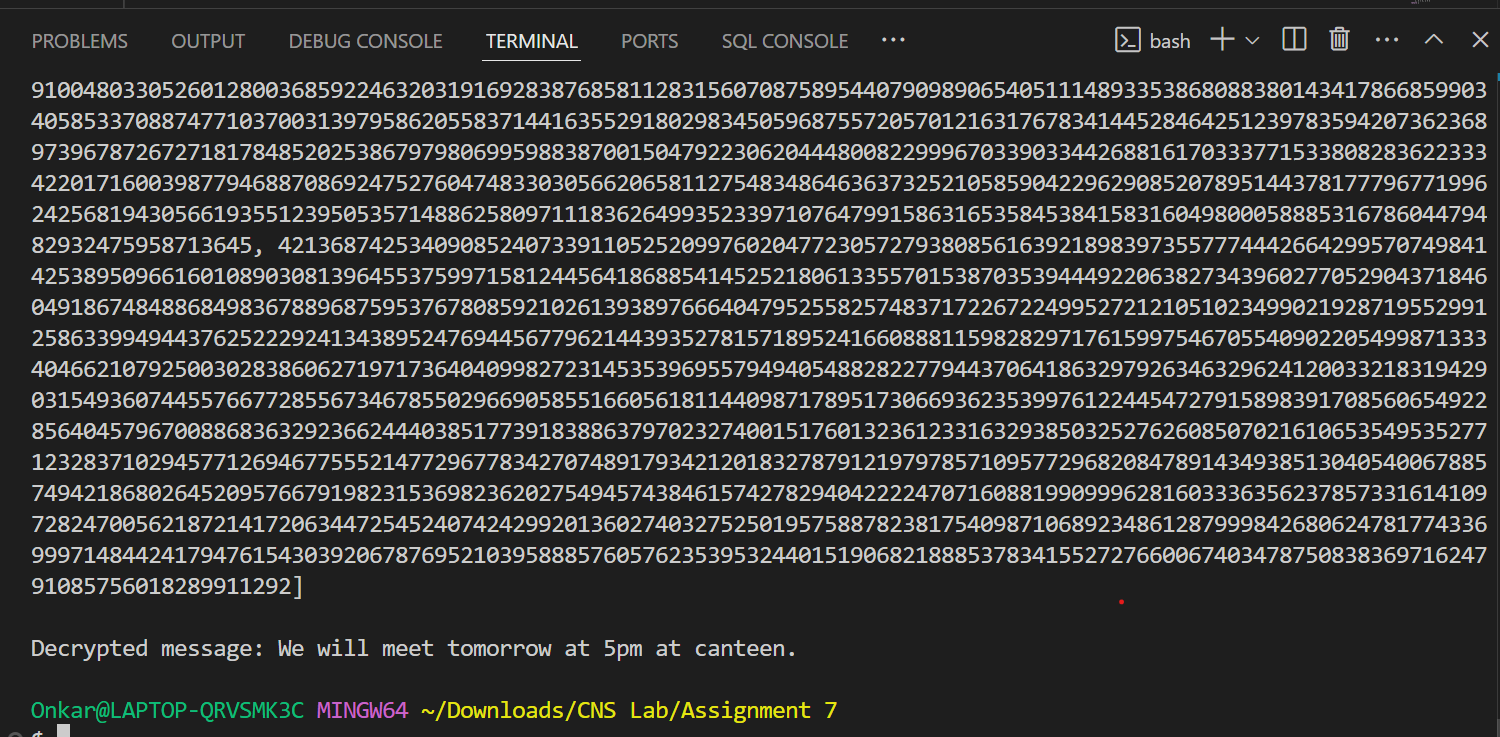












**Practical Applications of RSA**

* **Secure Communication:** Encrypting emails and messages.
* **Digital Signatures:** Verifying the authenticity of a message or document.
* **Key Exchange:** Securely exchanging keys for symmetric encryption algorithms.

RSA is widely used in various security protocols, including SSL/TLS for secure internet communications.

RSA ensures security through the difficulty of factoring large numbers. It is commonly used for securing sensitive data, digital signatures, and in SSL/TLS protocols.

**LA 2 WRITEUP**

**Name: Paras Wane**

**PRN : 21510041**

**Batch: B 2**

ASSIGNMENT 7

#### **1. Key Generation:**

* **Step 1:** Choose two large prime numbers, ppp and qqq.
* **Step 2:** Compute n=p×qn = p \times qn=p×q and ϕ(n)=(p−1)(q−1)\phi(n) = (p-1)(q-1)ϕ(n)=(p−1)(q−1).
* **Step 3:** Select the public exponent eee such that 1<e<ϕ(n)1 < e < \phi(n)1<e<ϕ(n), and eee is co-prime to ϕ(n)\phi(n)ϕ(n).
* **Step 4:** Compute the private exponent ddd, which is the modular multiplicative inverse of eee modulo ϕ(n)\phi(n)ϕ(n), i.e., d×e≡1 (mod ϕ(n))d \times e \equiv 1 \ (\text{mod} \ \phi(n))d×e≡1 (mod ϕ(n)).

#### **2. Encryption:**

* **Formula:** c≡me (mod n)c \equiv m^e \ (\text{mod} \ n)c≡me (mod n)
  + Where:
    - mmm is the plaintext message.
    - ccc is the ciphertext.
    - eee is the public key exponent.
    - nnn is the product of the two primes ppp and qqq.

#### **3. Decryption:**

* **Formula:** m≡cd (mod n)m \equiv c^d \ (\text{mod} \ n)m≡cd (mod n)
  + Where:
    - mmm is the decrypted message (plaintext).
    - ccc is the ciphertext.
    - ddd is the private key exponent.

#### **4. Digital Signatures:**

* **Signing:**
  + **Formula:** s≡md (mod n)s \equiv m^d \ (\text{mod} \ n)s≡md (mod n)
  + Where:
    - sss is the digital signature.
    - mmm is the message.
    - ddd is the private key.
* **Verification:**
  + **Formula:** m≡se (mod n)m \equiv s^e \ (\text{mod} \ n)m≡se (mod n)
  + Where:
    - mmm is the original message.
    - sss is the signature.
    - eee is the public key.

### **Applications:**

* **Secure Communications:** Used in protocols such as SSL/TLS.
* **Digital Signatures:** Ensures the authenticity and integrity of messages.

#### **Conclusion:**

RSA is fundamental for ensuring data security in various digital applications, particularly for secure communications and digital signatures.

CODE:

#include <iostream>

#include <cmath>

#include <cstdlib>

#include <ctime>

using namespace std;

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

int gcd(int a, int b) {

    while (b != 0) {

        int temp = b;

        b = a % b;

        a = temp;

    }

    return a;

}

// Function to perform modular exponentiation (x^y % p)

long long mod\_exp(long long x, long long y, long long p) {

    long long result = 1;

    x = x % p;

    while (y > 0) {

        if (y % 2 == 1)

            result = (result \* x) % p;

        y = y / 2;

        x = (x \* x) % p;

    }

    return result;

}

// Function to find the multiplicative inverse of e under mod phi

int mod\_inverse(int e, int phi) {

    for (int d = 1; d < phi; d++) {

        if ((e \* d) % phi == 1)

            return d;

    }

    return -1;

}

// Function to check if a number is prime

bool is\_prime(int num) {

    if (num < 2) return false;

    for (int i = 2; i <= sqrt(num); i++) {

        if (num % i == 0) return false;

    }

    return true;

}

int main() {

    srand(time(0));

    // Step 1: Choose two large prime numbers p and q

    int p, q;

    do {

        p = rand() % 100 + 100; // Random number between 100 and 200

    } while (!is\_prime(p));

    do {

        q = rand() % 100 + 100; // Random number between 100 and 200

    } while (!is\_prime(q));

    // Step 2: Compute n = p \* q

    int n = p \* q;

    // Step 3: Compute the totient phi = (p - 1) \* (q - 1)

    int phi = (p - 1) \* (q - 1);

    // Step 4: Choose an integer e such that 1 < e < phi and gcd(e, phi) = 1

    int e;

    do {

        e = rand() % phi;

    } while (gcd(e, phi) != 1);

    // Step 5: Compute d such that (d \* e) % phi = 1 (d is the modular inverse of e)

    int d = mod\_inverse(e, phi);

    // Display public and private keys

    cout << "Public Key: {" << e << ", " << n << "}\n";

    cout << "Private Key: {" << d << ", " << n << "}\n";

    // Step 6: Encryption

    int message;

    cout << "Enter the message (an integer): ";

    cin >> message;

    long long encrypted\_message = mod\_exp(message, e, n);

    cout << "Encrypted Message: " << encrypted\_message << "\n";

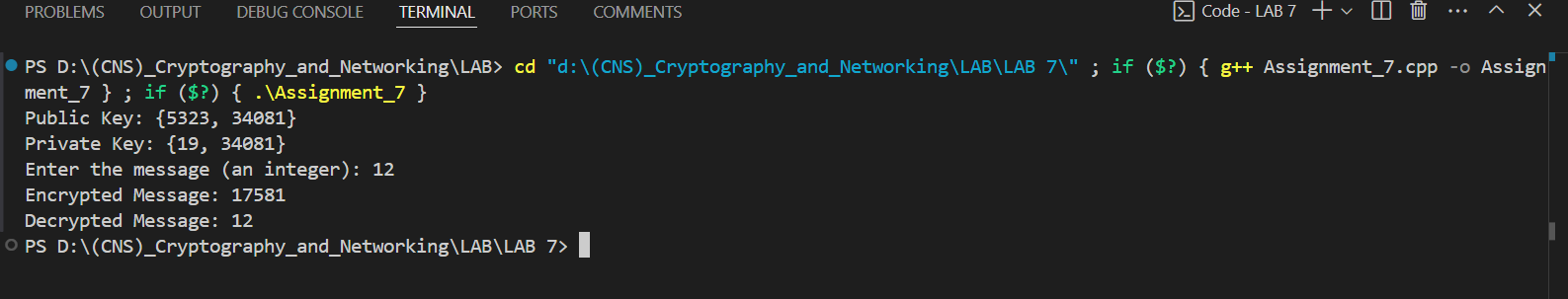
    // Step 7: Decryption

    long long decrypted\_message = mod\_exp(encrypted\_message, d, n);

    cout << "Decrypted Message: " << decrypted\_message << "\n";

    return 0;

}



**ASSIGNMENT 8**

**Implement the Diffie-Hellman Key Exchange algorithm for a given problem.**

**Introduction:** The Diffie-Hellman Key Exchange (DHKE) is a method that allows two parties to securely share a secret key over a public channel without prior secret sharing.

**Key Concepts:**

* **Public Parameters:** A large prime number ppp and a primitive root ggg (also known as a generator).
* **Private Keys:** Each party selects a private key (secret) that is kept confidential.

**Algorithm Steps:**

1. **Initialization:**
   * Two parties, Alice and Bob, agree on a prime number ppp and a base ggg.
2. **Key Generation:**
   * Alice selects a private key aaa and computes A=gamod  pA = g^a \mod pA=gamodp.
   * Bob selects a private key bbb and computes B=gbmod  pB = g^b \mod pB=gbmodp.
3. **Exchange:**
   * Alice sends AAA to Bob, and Bob sends BBB to Alice.
4. **Shared Secret Computation:**
   * Alice computes the shared secret: SA=Bamod  pS\_A = B^a \mod pSA​=Bamodp.
   * Bob computes the shared secret: SB=Abmod  pS\_B = A^b \mod pSB​=Abmodp.
   * Both will arrive at the same shared secret SSS.

**Security Basis:** The security of DHKE lies in the difficulty of solving the discrete logarithm problem, making it challenging for an eavesdropper to determine the shared secret from the exchanged values.

**Applications:**

* Used in secure communications (e.g., SSL/TLS) and to establish secure keys for symmetric encryption.

**Conclusion:** The Diffie-Hellman Key Exchange is a foundational technique for secure key exchange, enabling secure communication over untrusted networks.

**CODE:**

#include <iostream>

#include <cmath>

using namespace std;

// Function to perform modular exponentiation (a^b % mod)

long long mod\_exp(long long base, long long exp, long long mod) {

    long long result = 1;

    base = base % mod;

    while (exp > 0) {

        if (exp % 2 == 1)

            result = (result \* base) % mod;

        exp = exp >> 1;

        base = (base \* base) % mod;

    }

    return result;

}

int main() {

    // Step 1: Publicly agree on a large prime number (p) and a primitive root (g)

    long long p = 23;  // Example prime number

    long long g = 5;   // Example primitive root

    cout << "Publicly Shared Prime (p): " << p << endl;

    cout << "Publicly Shared Primitive Root (g): " << g << endl;

    // Step 2: Each user chooses a private key

    long long a;  // Private key for User A

    long long b;  // Private key for User B

    cout << "Enter the private key for User A: ";

    cin >> a;

    cout << "Enter the private key for User B: ";

    cin >> b;

    // Step 3: Each user calculates their public value

    long long A = mod\_exp(g, a, p);  // User A's public value

    long long B = mod\_exp(g, b, p);  // User B's public value

    cout << "User A's Public Value (A): " << A << endl;

    cout << "User B's Public Value (B): " << B << endl;

    // Step 4: Each user computes the shared secret

    long long shared\_secret\_A = mod\_exp(B, a, p);  // User A's computed shared secret

    long long shared\_secret\_B = mod\_exp(A, b, p);  // User B's computed shared secret

    cout << "User A's Computed Shared Secret: " << shared\_secret\_A << endl;

    cout << "User B's Computed Shared Secret: " << shared\_secret\_B << endl;

    // If both computed shared secrets match, the exchange is successful

    if (shared\_secret\_A == shared\_secret\_B) {

        cout << "Shared secret successfully established: " << shared\_secret\_A << endl;

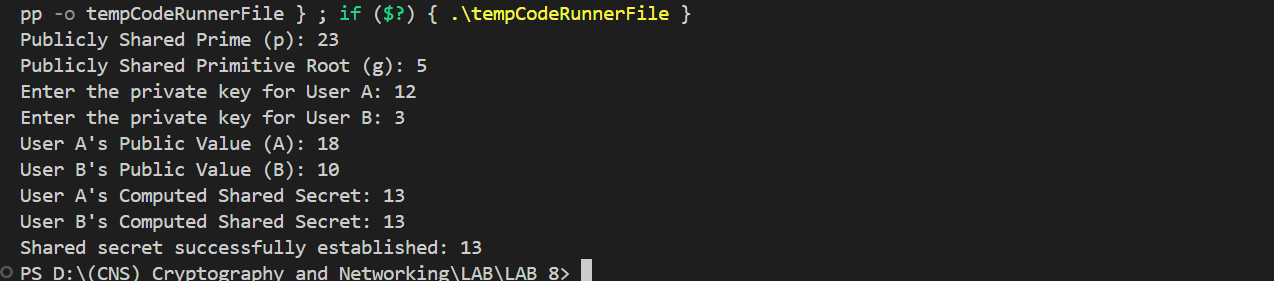
    } else {

        cout << "Error in establishing the shared secret." << endl;

    }

    return 0;

}

****

**ASSIGNMENT 9**

**Calculate the message digest of a text using the SHA-1 algorithm**.

**SHA-1 Message Digest: Overview**

**Introduction:** SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function designed to produce a fixed-size (160-bit) message digest from an input of arbitrary size. It is commonly used in various security applications and protocols, including TLS and SSL, PGP, and SSH.

**Key Characteristics:**

* **Deterministic:** The same input will always produce the same output.
* **Fixed Output Size:** Regardless of input size, the output is always 160 bits.
* **Fast Computation:** SHA-1 is designed for efficient processing.
* **Pre-image Resistance:** Given a hash output, it should be computationally infeasible to find any input that hashes to that output.
* **Collision Resistance:** It should be challenging to find two different inputs that produce the same hash output.

**SHA-1 Algorithm Steps:**

1. **Input Preparation:**
   * Convert the input text into a binary format and append padding bits to ensure the total length is congruent to 448 modulo 512.
   * Append the length of the original message as a 64-bit integer.
2. **Initialization:**
   * Start with predefined hash values (H0, H1, H2, H3, H4) which are 32-bit integers.
3. **Processing Message in Blocks:**
   * The message is divided into 512-bit blocks, and each block is processed through a series of operations that involve bitwise operations, modular additions, and logical functions.
4. **Finalization:**
   * After processing all blocks, the final hash value is obtained by concatenating the hash values, resulting in a 160-bit message digest.

**Applications:**

* **Data Integrity:** Ensures that data has not been altered.
* **Digital Signatures:** Used to verify the authenticity of messages.
* **Password Hashing:** Commonly used to store hashed passwords.

**Limitations:** Despite its popularity, SHA-1 is considered weak against collision attacks, and its use is being phased out in favor of stronger hash functions like SHA-256 and SHA-3.

**Conclusion:** Calculating the SHA-1 message digest is a crucial process in cryptography, providing data integrity and authentication. However, due to its vulnerabilities, it is essential to consider more secure alternatives for sensitive applications.

import hashlib

def main():

    # Input text

    message = input("Enter the message: ")

    # Step 1: Create a SHA-1 hash object

    sha1 = hashlib.sha1()

    # Step 2: Update the hash object with the bytes of the message

    sha1.update(message.encode('utf-8'))

    # Step 3: Calculate the hash digest in hexadecimal format

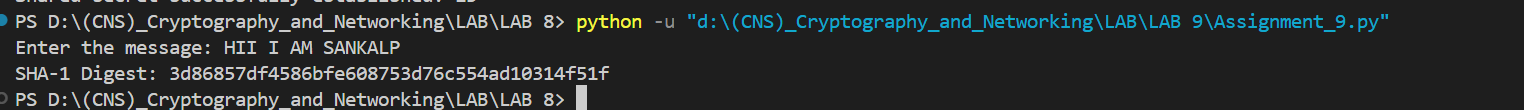
    digest = sha1.hexdigest()

    # Step 4: Output the message digest (in hexadecimal)

    print("SHA-1 Digest:", digest)

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

    main()

****

**ASSIGNMENT 10**

**Implement the SIGNATURE SCHEME – Digital Signature Standard.**

**Introduction:** The Digital Signature Standard (DSS) is a federal standard for digital signatures that provides a method for ensuring the authenticity and integrity of digital messages or documents. It is widely used in various applications, including secure communications, software distribution, and financial transactions.

**Key Components:**

1. **Digital Signature:** A digital signature is a mathematical scheme for verifying the authenticity and integrity of a message. It serves as a counterpart to handwritten signatures or stamped seals but offers far more inherent security.
2. **Key Pair:** DSS uses asymmetric cryptography, meaning it employs a pair of keys—a private key (known only to the signer) and a public key (shared with recipients).
3. **Hash Function:** DSS utilizes a cryptographic hash function (like SHA-256) to create a fixed-size hash of the message, which is then signed with the private key.

**DSS Process:** The process of creating and verifying a digital signature under DSS involves the following steps:

1. **Key Generation:**
   * A pair of keys (public and private) is generated using a specified algorithm (like DSA - Digital Signature Algorithm).
   * The private key is kept secret, while the public key is distributed to anyone wishing to verify the signature.
2. **Signing a Message:**
   * The sender creates a hash of the original message using a cryptographic hash function.
   * The hash is then signed with the sender's private key to produce the digital signature.
   * The signed hash and the original message are sent to the recipient.
3. **Verifying a Signature:**
   * The recipient receives the message and the digital signature.
   * The recipient generates a hash of the received message using the same hash function.
   * The recipient uses the sender's public key to verify the signature against the generated hash.
   * If the hashes match, it confirms that the message is authentic and has not been altered.

**Applications:**

* **Secure Transactions:** DSS is crucial in e-commerce and financial applications, ensuring the authenticity of transactions.
* **Software Distribution:** Used to verify the integrity and authenticity of software updates or packages.
* **Email Security:** Ensures that emails are not tampered with and verifies the sender’s identity.

**Advantages:**

* **Integrity:** Ensures that the message has not been altered in transit.
* **Authenticity:** Confirms the identity of the sender.
* **Non-repudiation:** The sender cannot deny having sent the message, as only they possess the private key used to sign it.

**Limitations:** DSS relies on the security of the underlying cryptographic algorithms. If vulnerabilities are discovered in the algorithms used for key generation or hashing, the security of the digital signatures can be compromised.

**Conclusion:** The Digital Signature Standard (DSS) is a vital component of modern cryptography, providing a robust mechanism for securing digital communications. By employing a combination of hashing and asymmetric cryptography, DSS ensures the integrity, authenticity, and non-repudiation of digital messages and documents, making it an essential tool in various security-sensitive applications.

from cryptography.hazmat.backends import default\_backend

from cryptography.hazmat.primitives.asymmetric import dsa

from cryptography.hazmat.primitives import serialization

from cryptography.hazmat.primitives import hashes

def generate\_keys():

    # Generate a DSA private key

    private\_key = dsa.generate\_private\_key(key\_size=2048, backend=default\_backend())

    public\_key = private\_key.public\_key()

    # Serialize keys to save them for later use

    private\_pem = private\_key.private\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PrivateFormat.TraditionalOpenSSL,

        encryption\_algorithm=serialization.NoEncryption()  # No encryption

    )

    public\_pem = public\_key.public\_bytes(

        encoding=serialization.Encoding.PEM,

        format=serialization.PublicFormat.SubjectPublicKeyInfo

    )

    return private\_pem, public\_pem

def sign\_message(private\_key\_pem, message):

    # Load the private key from PEM format (add password=None)

    private\_key = serialization.load\_pem\_private\_key(

        private\_key\_pem,

        password=None,  # Specify None for unencrypted key

        backend=default\_backend()

    )

    # Create a signature for the message

    signature = private\_key.sign(

        message.encode(),

        hashes.SHA256()  # Hashing algorithm

    )

    return signature

def verify\_signature(public\_key\_pem, message, signature):

    # Load the public key from PEM format

    public\_key = serialization.load\_pem\_public\_key(public\_key\_pem, backend=default\_backend())

    # Verify the signature

    try:

        public\_key.verify(

            signature,

            message.encode(),

            hashes.SHA256()  # Hashing algorithm

        )

        return True  # Signature is valid

    except Exception:

        return False  # Signature is invalid

def main():

    # Generate keys

    private\_key, public\_key = generate\_keys()

    # Input message to sign

    message = input("Enter the message to sign: ")

    # Sign the message

    signature = sign\_message(private\_key, message)

    print("Signature:", signature.hex())

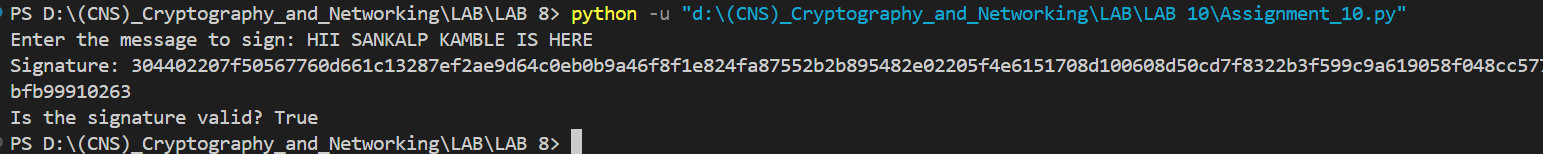
    # Verify the signature

    is\_valid = verify\_signature(public\_key, message, signature)

    print("Is the signature valid?", is\_valid)

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

    main()



Walchand College Of Engineering

CNS LAB – LA2

Name: Viraj Patil

PRN: 21510097

1] Demonstration of SSL using Wireshark

* Demonstration of SSL using Wireshark

Description:

A TLS connection is initiated using a sequence known as the [TLS handshake](https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/). When a user navigates to a website that uses TLS, the TLS handshake begins between the user's device (also known as the *client* device) and the web server.

During the TLS handshake, the user's device and the web server:

* Specify which version of TLS (TLS 1.0, 1.2, 1.3, etc.) they will use
* Decide on which cipher suites (see below) they will use
* Authenticate the identity of the server using the server's TLS certificate
* Generate session keys for encrypting messages between them after the handshake is complete

The TLS handshake establishes a cipher suite for each communication session. The cipher suite is a set of algorithms that specifies details such as which shared [encryption keys](https://www.cloudflare.com/learning/ssl/what-is-a-cryptographic-key/), or [session keys](https://www.cloudflare.com/learning/ssl/what-is-a-session-key/), will be used for that particular session. TLS is able to set the matching session keys over an unencrypted channel thanks to a technology known as [public key cryptography](https://www.cloudflare.com/learning/ssl/how-does-public-key-encryption-work/).

The handshake also handles authentication, which usually consists of the server proving its identity to the client. This is done using public keys. Public keys are encryption keys that use one-way encryption, meaning that anyone with the public key can unscramble the data encrypted with the server's private key to ensure its authenticity, but only the original sender can encrypt data with the private key. The server's public key is part of its TLS certificate.

Once data is encrypted and authenticated, it is then signed with a message authentication code (MAC). The recipient can then verify the MAC to ensure the integrity of the data. This is kind of like the tamper-proof foil found on a bottle of aspirin; the consumer knows no one has tampered with their medicine because the foil is intact when they purchase it.

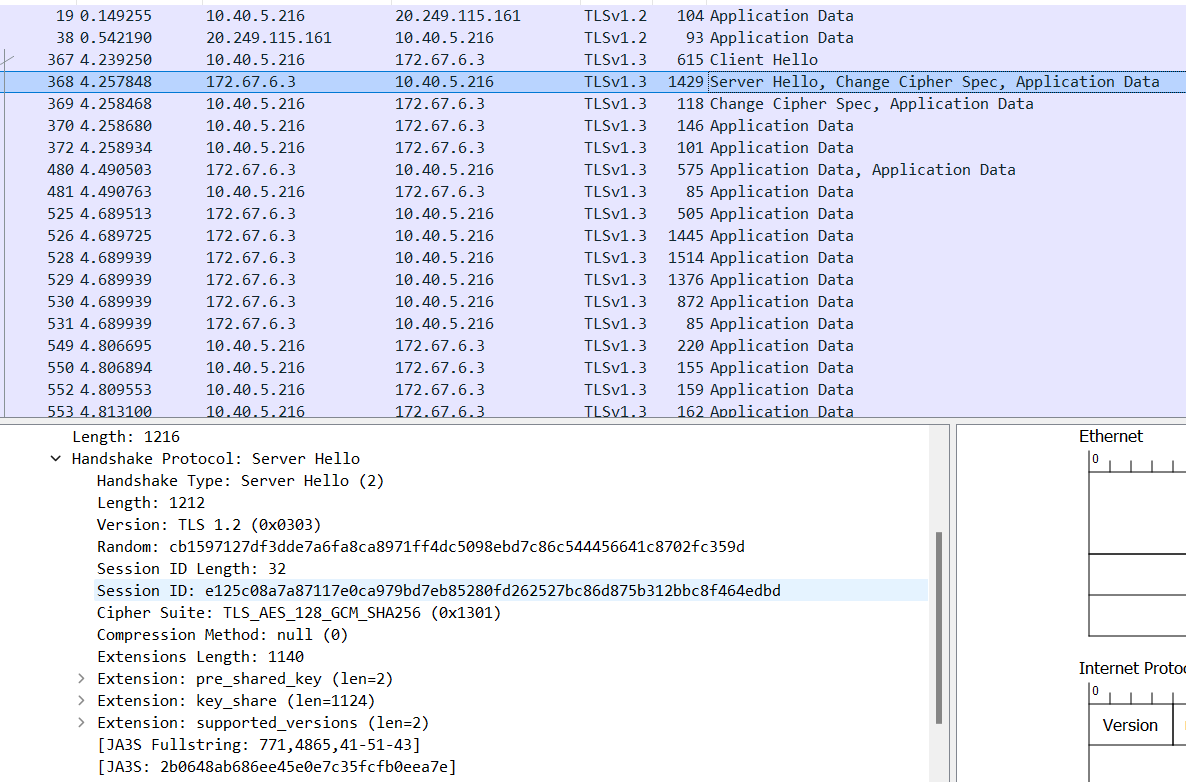
TLS Handshake:

**Step 1: Client Hello**

The client begins the communication. The first step is called **client hello**. The client lists**the versions of SSL/TLS** and **cipher suites** it’s able to use.

**Step 2: Server Hello**

The server will see the list of SSL/TLS versions and cipher suites and pick the newest the server is able to use. Then the server sends a message to the client containing the SSL/TLS version and cipher suite it chose.



**Step 3: Server Key Exchange**

After the server and client agree on the SSL/TLS version and cipher suite, the server sends two things.

* SSL/TLS certificate
* public key and signature

The first is its**SSL/TLS certificate** to the client. The client (web browser) validates the server’s certificate. Web browsers store a list of Root CA(Certificate Authority) in themselves. These root CAs are third parties that are trusted by web browsers. The server’s certificate is issued by root CA or intermediate CA. Intermediate CA is a CA that is trusted by root CA.

Web browsers trust Root CA. Root CA trusts immediate CA. If the server’s certificate is issued by a trusted root CA or immediate CA, then the browser trust the server’s certificate.

The second thing the server sends is its **public key and signature**. The public key is actually included in the certificate. The client and the server use the public key to encrypt messages, which can only be decrypted with the server’s private key. The server never shares its private key with anyone.

At the end of the server key exchange, the server sends a **server hello done** message.

Until now, all the information sent between the client and server is unencrypted. The client has the server’s public key, what will the client do now?

1. It generates a random **session key** (aka pre-master key).
2. Encrypt the session key with the server’s public key.
3. Sends the encrypted session key to the server.

The encrypted session key can only be decrypted with the server’s private key. Only the server has the private key, so only the client and server can know the session key.

**Step 5. Change Cipher Spec**

The change cipher spec message is sent by both the client and server to notify the receiving party that subsequent records will be protected under the just-negotiated CipherSpec and keys

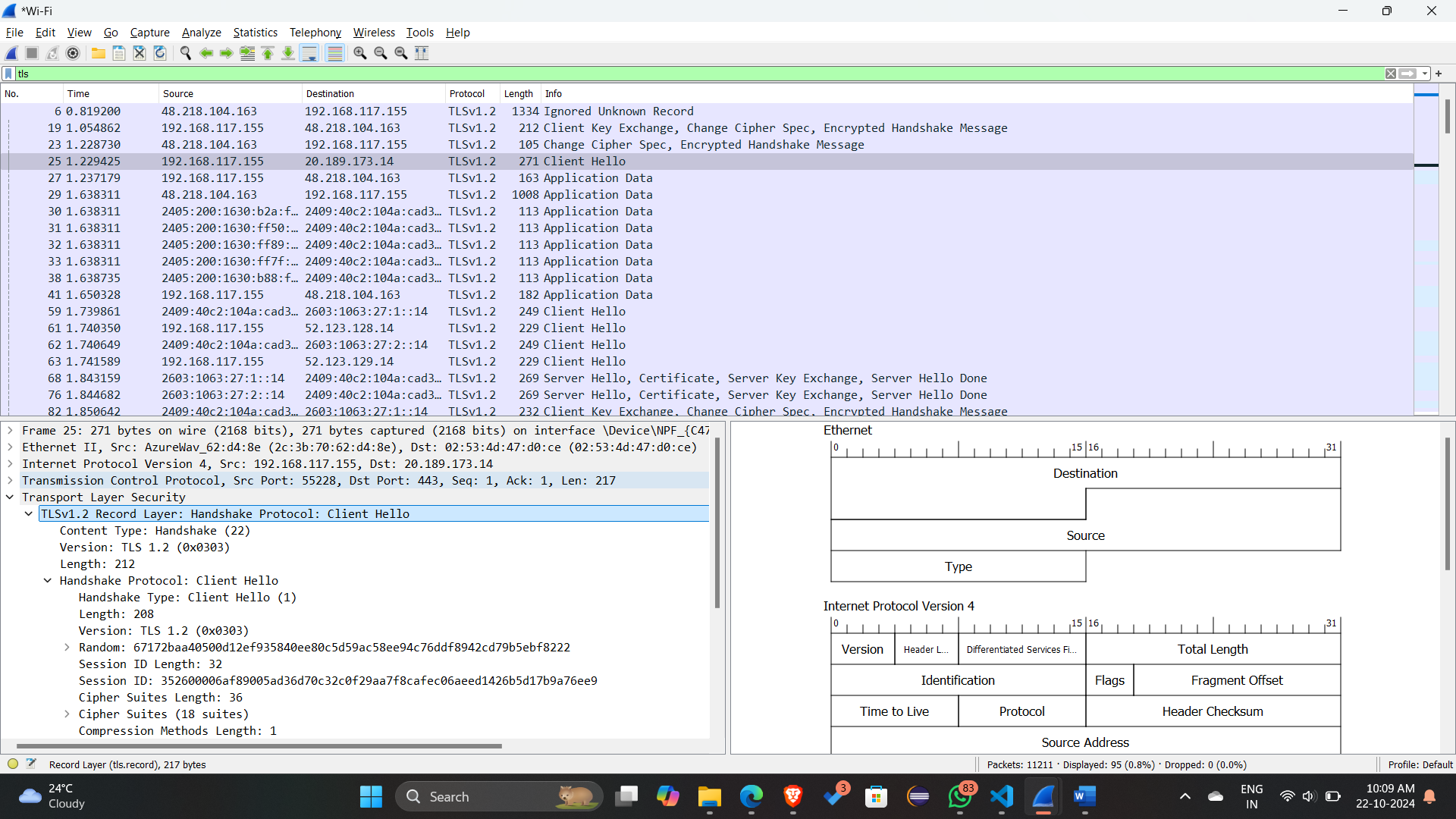
**Step 6. Encrypted Handshake**

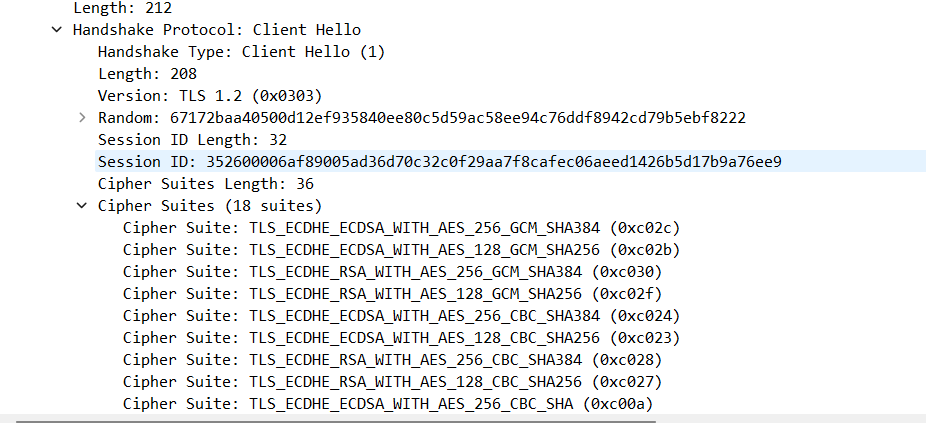
The client and the server send each other an encrypted message saying the key information is correct.

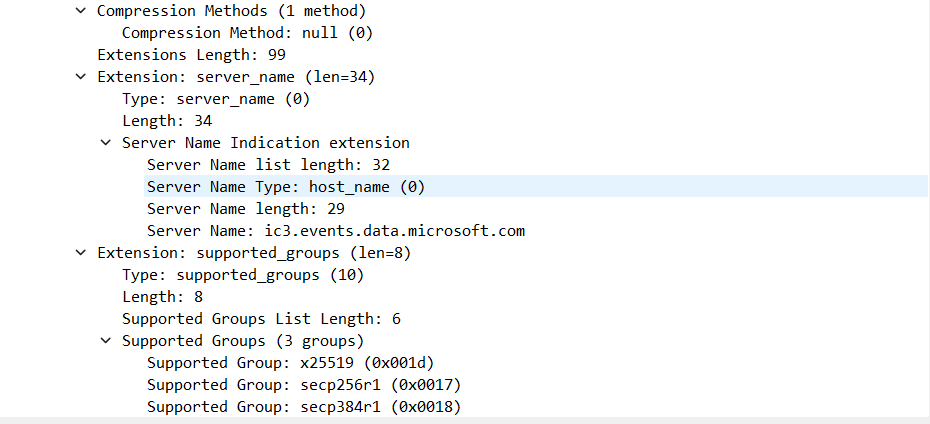
Now the client (web browser) will display a green lock in the address bar. The client and server encrypt HTTP traffic with the session key.

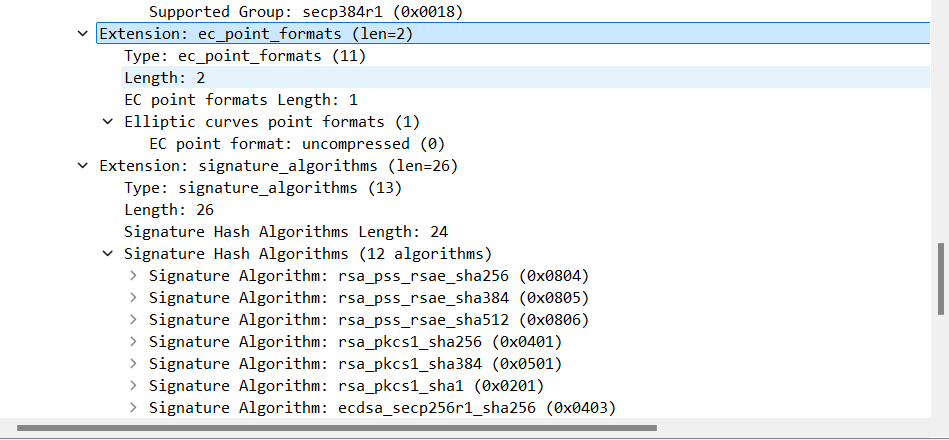
Session key is only valid in one session. If the user closes the client and visits the same server the next day, a new session key will be generated by the client.

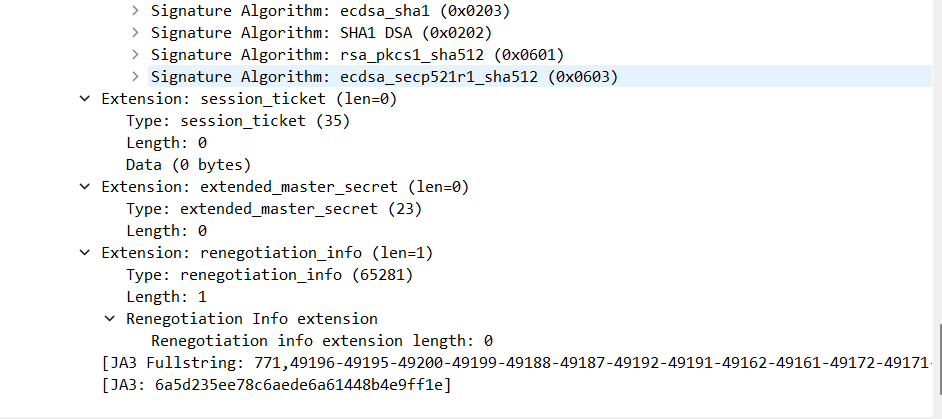
Client Hello:



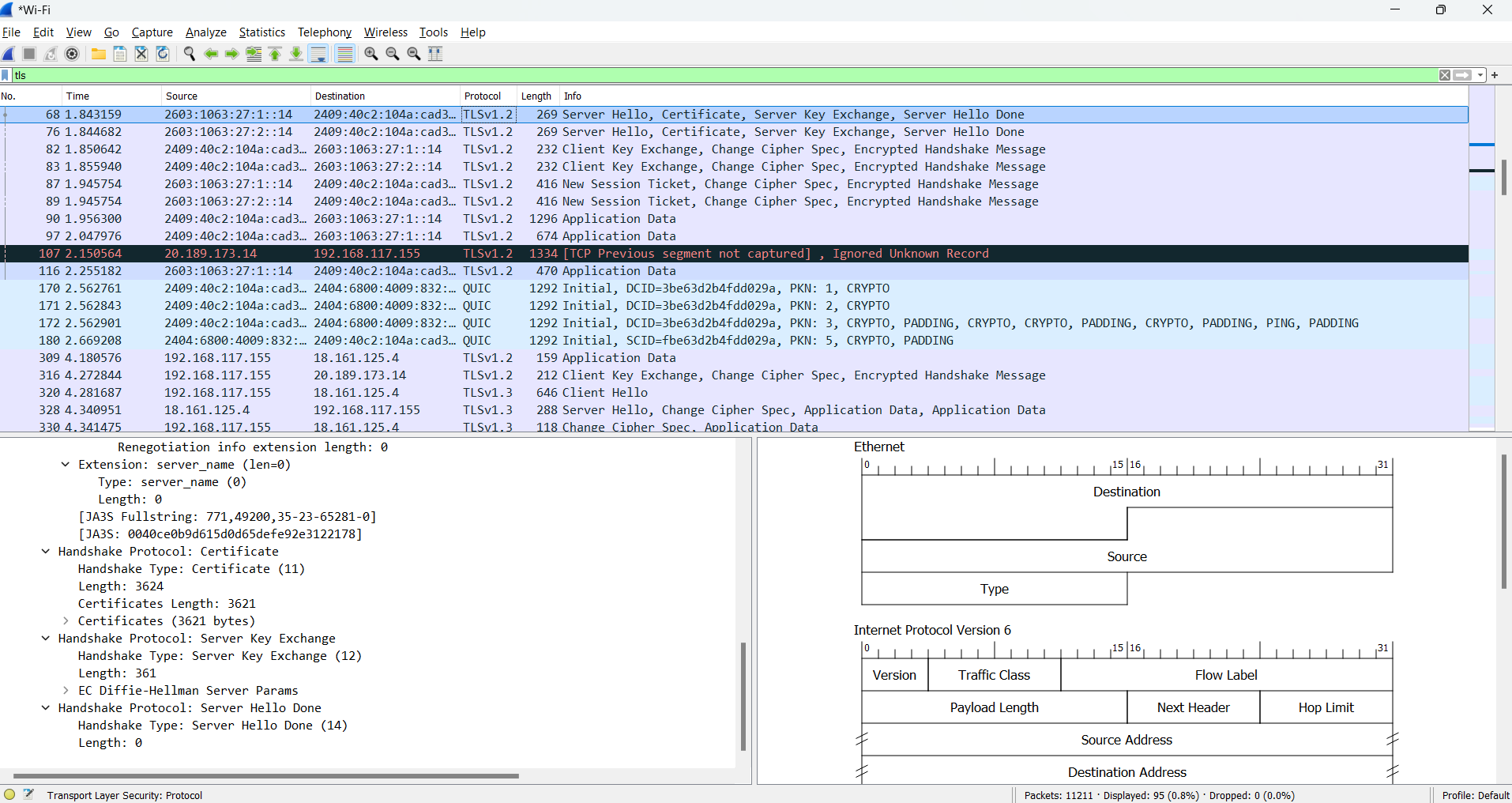


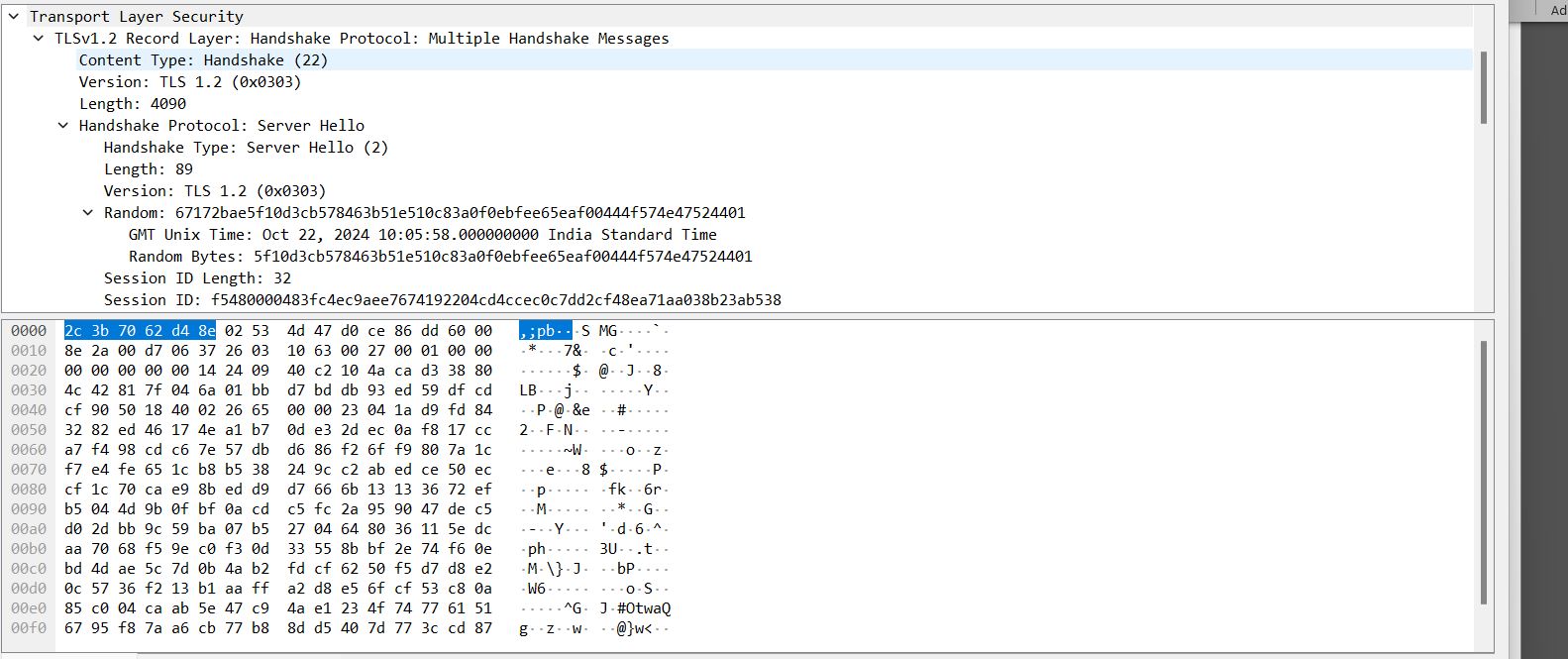




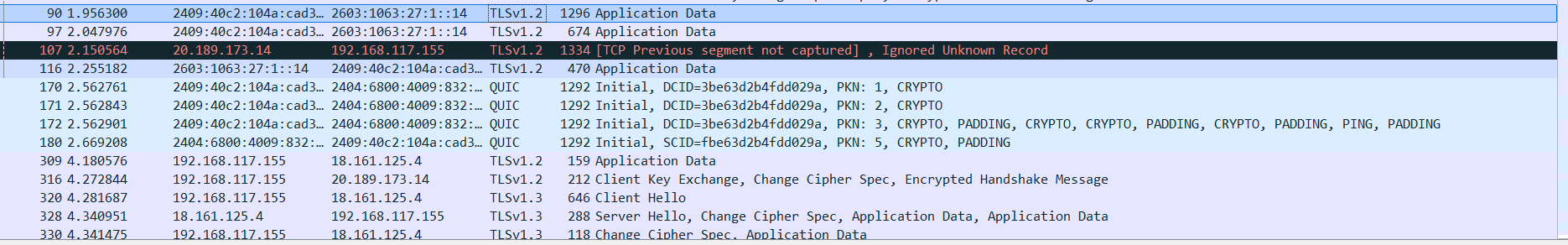


Server Hello:

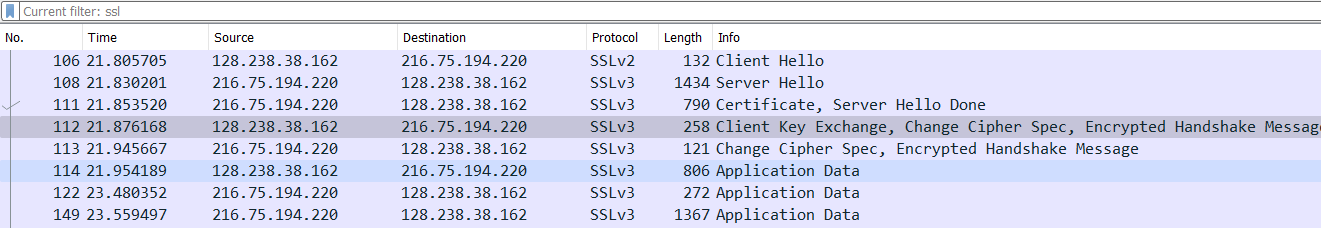


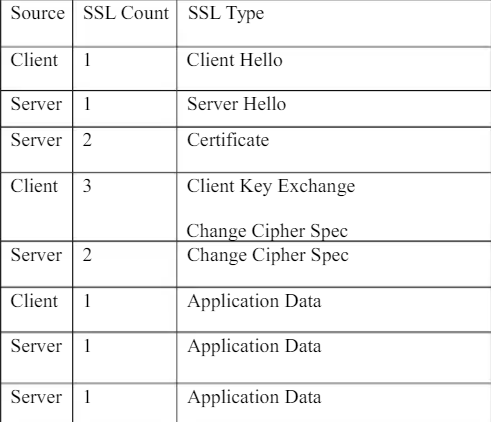


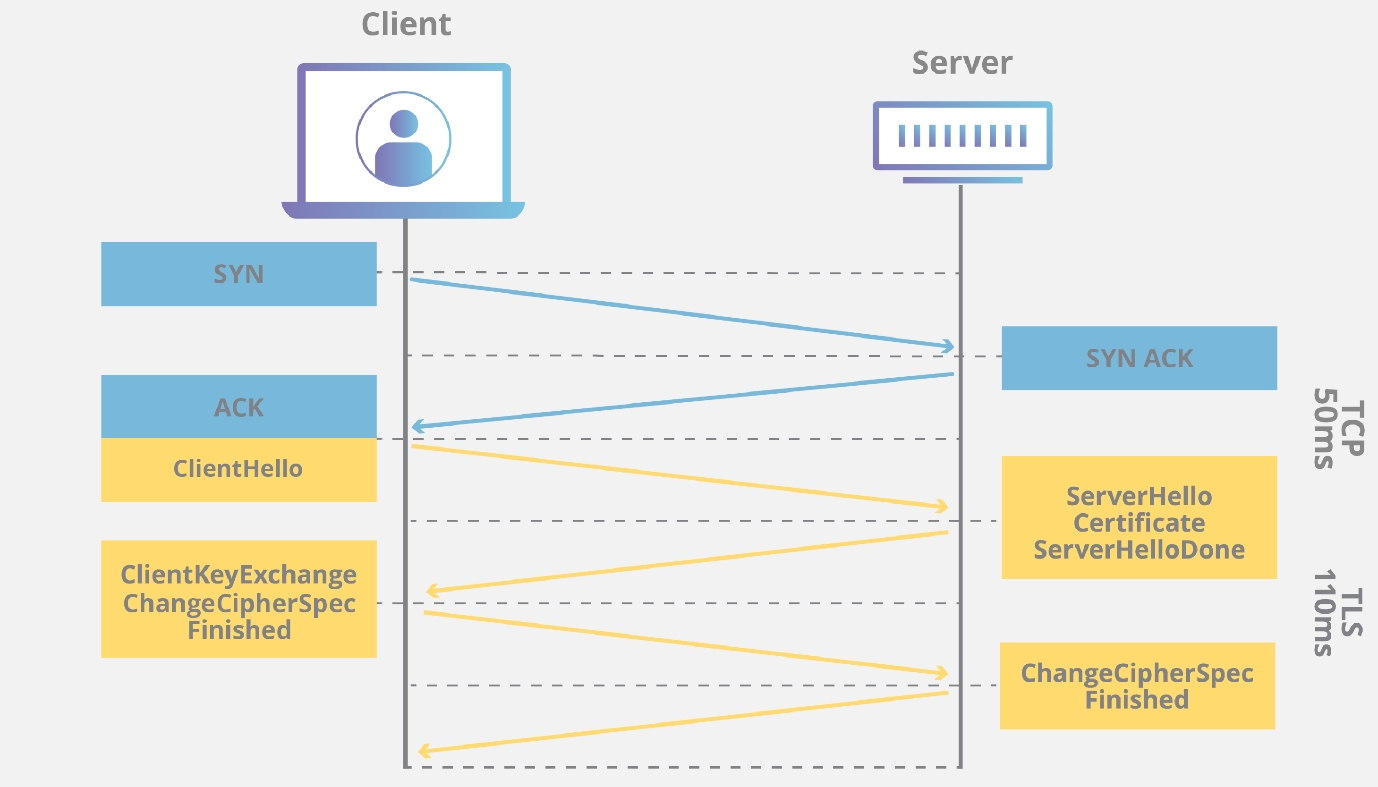
Application Data:



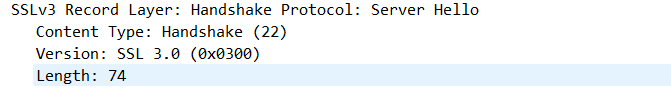
1. For each of the first 8 Ethernet frames, specify the source of the frame (client or server), determine the number of SSL records that are included in the frame, and list the SSL record types that are included in the frame. Draw a timing diagram between client and server, with one arrow for each SSL record.



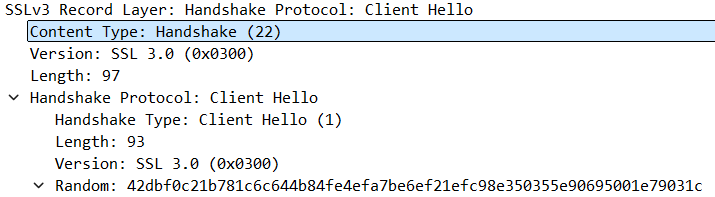




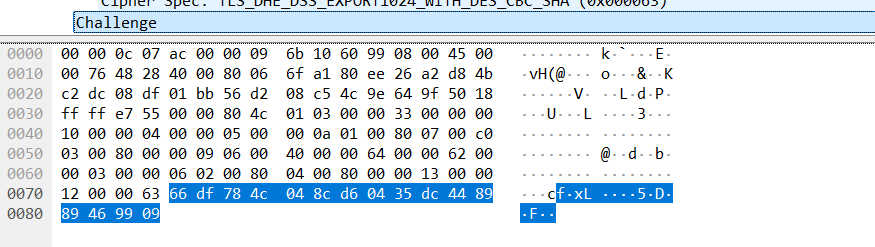
1. Each of the SSL records begins with the same three fields (with possibly different values). One of these fields is “content type” and has length of one byte. List all three fields and their lengths.



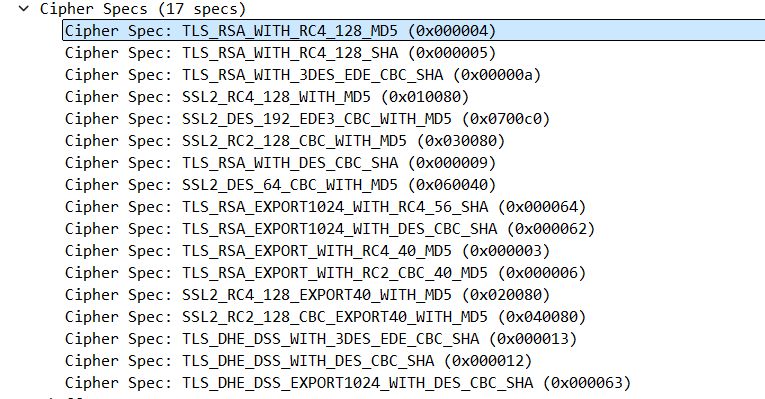
1. Expand the ClientHello record. (If your trace contains multiple ClientHello records, expand the frame that contains the first one.) What is the value of the content type?



1. Does the ClientHello record contain a nonce (also known as a “challenge”)? If so, what is the value of the challenge in hexadecimal notation?



1. Does the ClientHello record advertise the cyber suites it supports? If so, in the first listed suite, what are the public-key algorithm, the symmetric-key algorithm, and the hash algorithm?

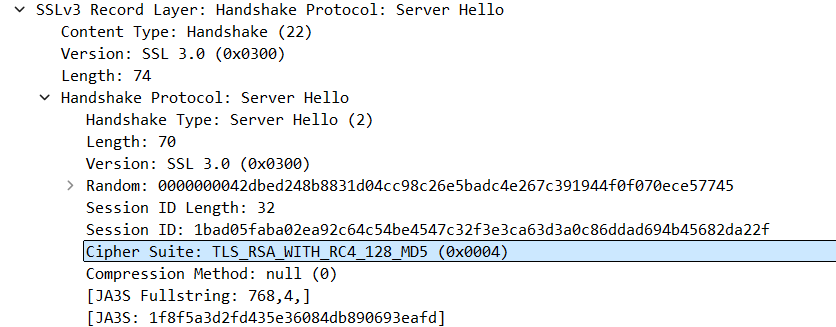


RSA – public key encryption

RC4 – Symmetric key algorithm

MD5 – For hashing

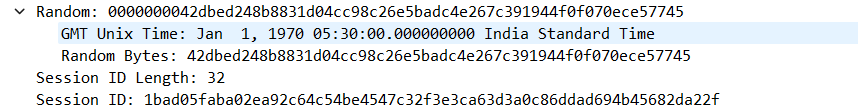
1. Locate the ServerHello SSL record. Does this record specify a chosen cipher suite? What are the algorithms in the chosen cipher suite?



1. Does this record include a nonce? If so, how long is it? What is the purpose of the client and server nonces in SSL?

Yes, record contains the nonce, which is under the Random tab,

It is 32 bits long



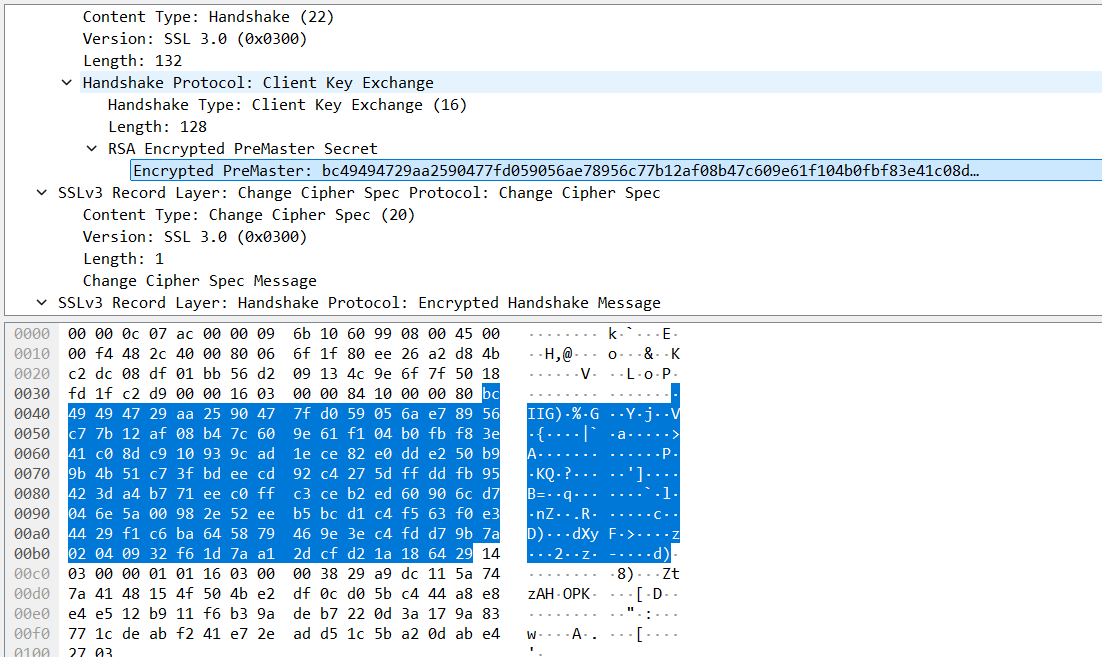
1. Does this record include a session ID? What is the purpose of the session ID?

Yes it includes session Id, It is unique identifier for SSL session

1. Does this record contain a certificate, or is the certificate included in a separate record. Does the certificate fit into a single Ethernet frame?

There is no certificate

1. Locate the client key exchange record. Does this record contain a pre-master secret? What is this secret used for? Is the secret encrypted? If so, how? How long is the encrypted secret?



Yes, it contains premaster secret

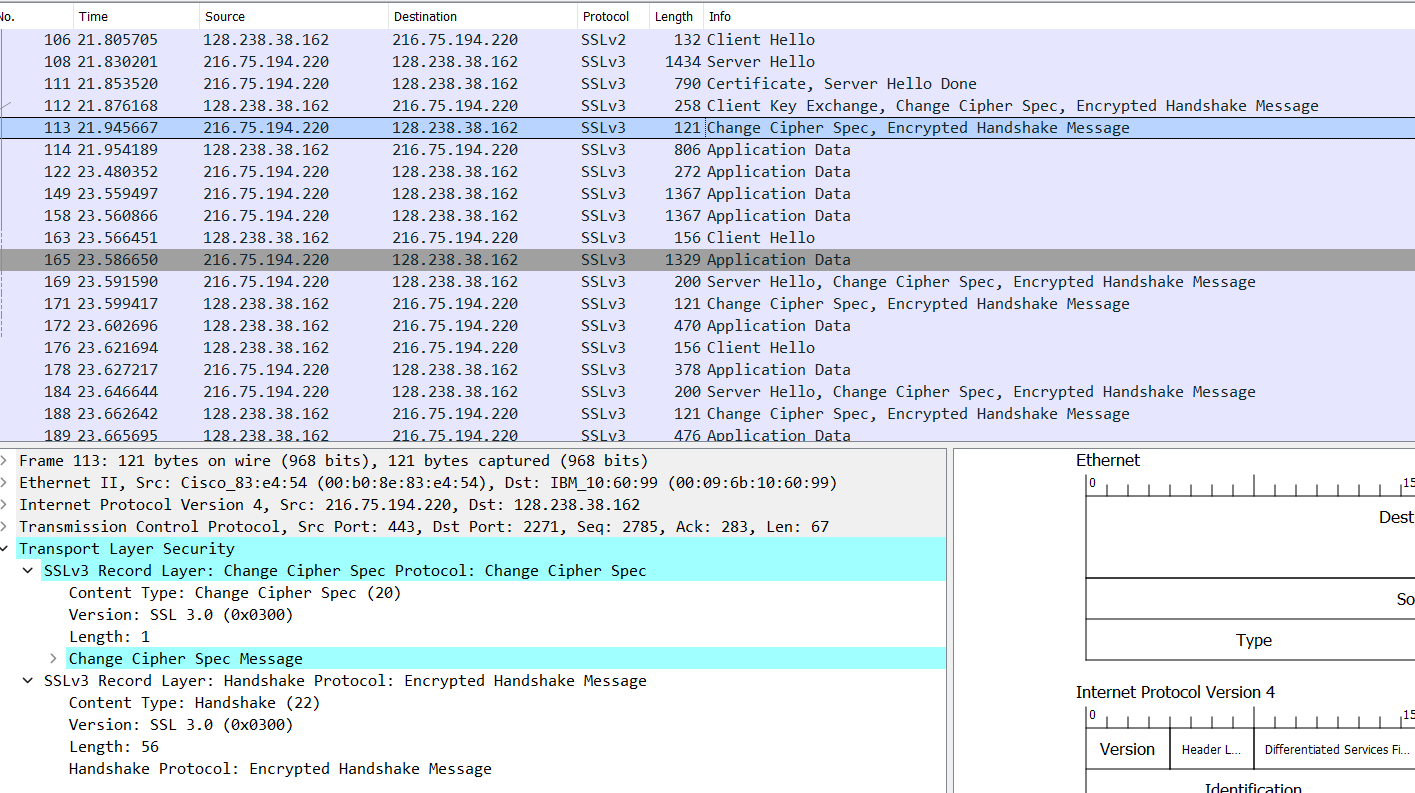
It is used to generate session keys for encryption

Secret is encrypted using public key of server

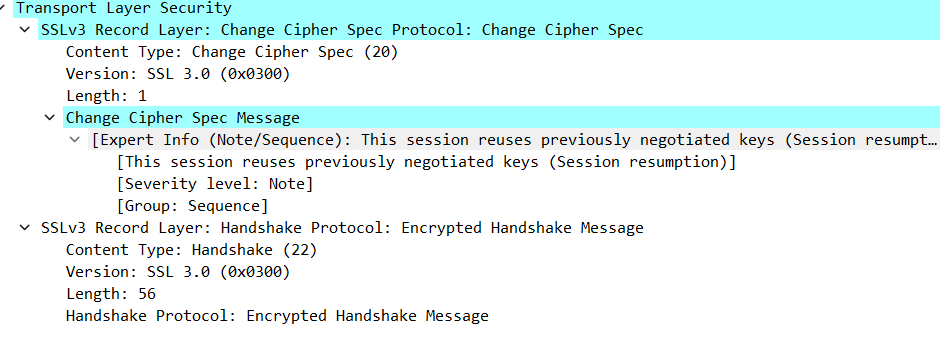
Client gets this key which it extracted from certificate from server

Secret is 128 bytes long

1. What is the purpose of the Change Cipher Spec record? How many bytes is the record in your trace?



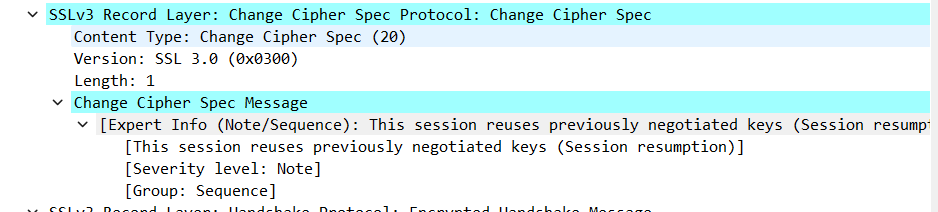
Purpose of change cipher spec record is to indicate that the contents of the following SSL record sent by client will be encrypted



1. In the encrypted handshake record, what is being encrypted? How?

MAC of the concatenation of all previous handshake message sent from this client is generated and sent to server

1. Does the server also send a change cipher record and an encrypted handshake record to the client? How are those records different from those sent by the client?

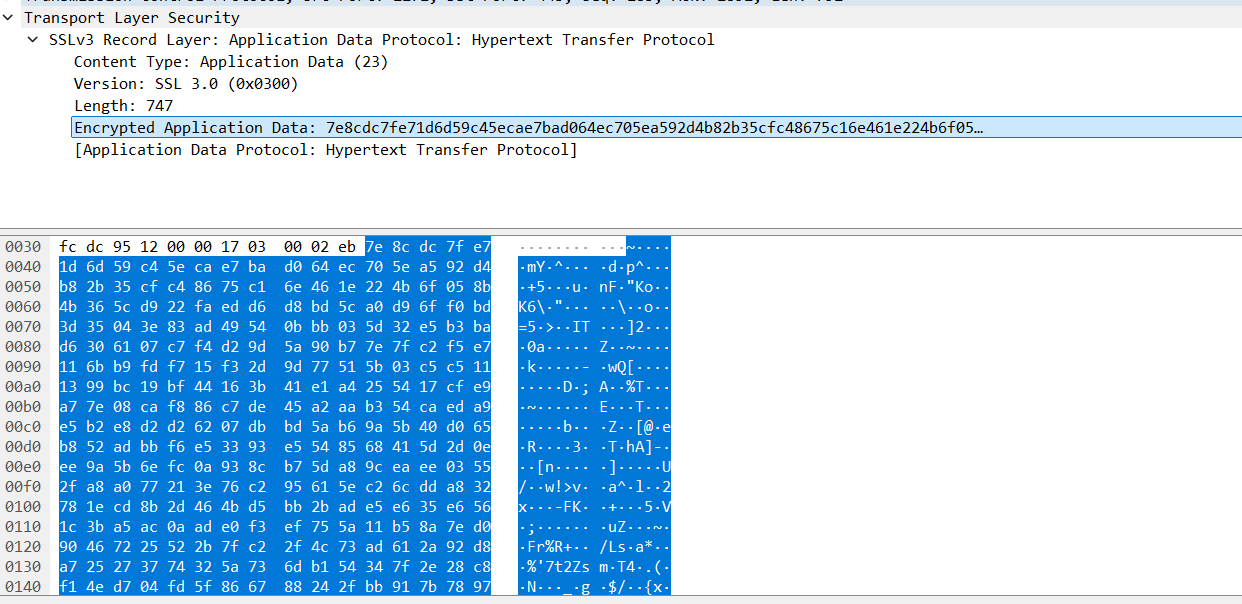


Yes server also send the change cipher record encrypted handshake to the client. It is different from the client’s one as it contains the concatenation of all handshake messages sent from server rather than from client

1. How is the application data being encrypted? Do the records containing application data include a MAC? Does Wireshark distinguish between the encrypted application data and the MAC?

Symmetric key encryption algorithm is used to encrypt the application data in the handshake phase RC4, it uses key generated using pre-master key and nonces from both sides.

Client encryption key is used to encrpt the data being sent from client to server and server encryption key is used to encrypt the data being sent from server to client



1. Comment on and explain anything else that you found interesting in the trace

Only in the first frame i.e frame number 106 version of SSL is 2 (SSLv2) and after that it changes to SSLv3

