**Cryptography and Network Security**

**LA-2**

**Name : Sourabh Shankar Patil**

**PRN : 21510045**

**Batch : B2**

**Assignment no 07**

**RSA:**

import java.io.DataInputStream;

import java.io.IOException;

import java.math.BigInteger;

import java.util.Random;

public class RSA {

    private BigInteger P;

    private BigInteger Q;

    private BigInteger N;

    private BigInteger PHI;

    private BigInteger e;

    private BigInteger d;

    private int maxLength = 1024;

    private Random R;

    public RSA() {

        R = new Random();

        P = BigInteger.probablePrime(maxLength, R);

        Q = BigInteger.probablePrime(maxLength, R);

        N = P.multiply(Q);

        PHI = P.subtract(BigInteger.ONE).multiply(Q.subtract(BigInteger.ONE));

        e = BigInteger.probablePrime(maxLength / 2, R);

        while (PHI.gcd(e).compareTo(BigInteger.ONE) > 0 && e.compareTo(PHI) < 0) {

            e.add(BigInteger.ONE);

        }

        d = e.modInverse(PHI);

    }

    public RSA(BigInteger e, BigInteger d, BigInteger N) {

        this.e = e;

        this.d = d;

        this.N = N;

    }

    public static void main(String[] arguments) throws IOException {

        RSA rsa = new RSA();

        DataInputStream input = new DataInputStream(System.in);

        String inputString;

        System.out.println("Enter message you wish to send.");

        inputString = input.readLine();

        System.out.println("Encrypting the message: " + inputString);

        System.out.println("The message in bytes is:: "

                + bToS(inputString.getBytes()));

        // encryption

        byte[] cipher = rsa.encryptMessage(inputString.getBytes());

        // decryption

        byte[] plain = rsa.decryptMessage(cipher);

        System.out.println("Decrypting Bytes: " + bToS(plain));

        System.out.println("Plain message is: " + new String(plain));

    }

    private static String bToS(byte[] cipher) {

        String temp = "";

        for (byte b : cipher) {

            temp += Byte.toString(b);

        }

        return temp;

    }

    // Encrypting the message

    public byte[] encryptMessage(byte[] message) {

        return (new BigInteger(message)).modPow(e, N).toByteArray();

    }

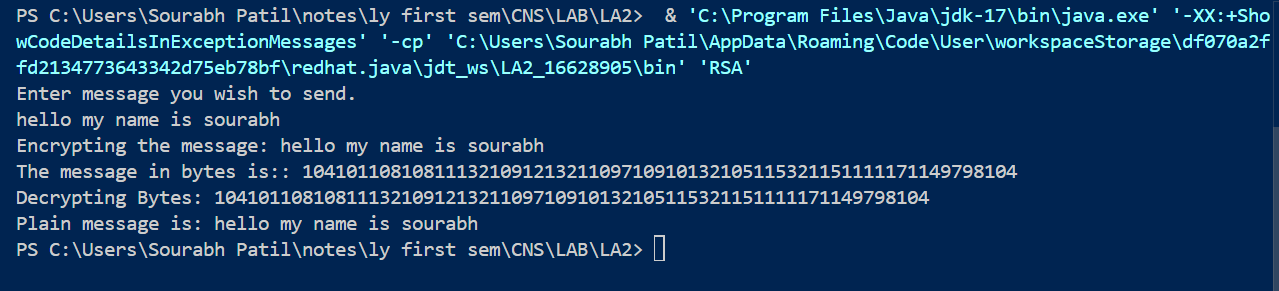
    // Decrypting the message

    public byte[] decryptMessage(byte[] message) {

        return (new BigInteger(message)).modPow(d, N).toByteArray();

    }

}



**Explanation-**

The provided code is an implementation of the **RSA encryption** algorithm in Java. RSA (Rivest-Shamir-Adleman) is a public-key encryption algorithm used to securely transmit messages over the internet. It involves generating two keys: a public key for encryption and a private key for decryption. Below is an explanation of each part of the code:

**1. Class Variables:**

* P and Q: Two large prime numbers generated randomly.
* N: The product of P and Q (N = P \* Q), used as part of the public and private keys.
* PHI: The totient of N, calculated as PHI = (P - 1) \* (Q - 1). It's required to generate the public and private keys.
* e: The public exponent (part of the public key). It is a random number chosen such that gcd(e, PHI) = 1 and e < PHI.
* d: The private exponent (part of the private key). It is the modular inverse of e modulo PHI, such that d \* e ≡ 1 (mod PHI).
* maxLength: The bit-length used to generate the prime numbers P and Q. Here, the value is set to 1024 bits.
* R: A Random object used to generate random prime numbers.

**2. Constructor:**

There are two constructors:

* **Default Constructor RSA()**:
  + Randomly generates two large primes P and Q.
  + Computes N = P \* Q and PHI = (P - 1) \* (Q - 1).
  + Selects a random public exponent e such that gcd(e, PHI) = 1 (i.e., e is coprime to PHI).
  + Calculates the private key exponent d using the formula d = e^(-1) mod PHI, which is the modular inverse of e modulo PHI.
* **Parameterized Constructor RSA(BigInteger e, BigInteger d, BigInteger N)**:
  + Initializes an RSA object with predefined public key components (e and N) and private key (d).

**3. Main Method:**

The main method allows a user to input a message, encrypt it, and then decrypt it back to the original message:

* The message is entered by the user via DataInputStream.
* The message is encrypted using the encryptMessage method.
* The encrypted message is then decrypted back to the original form using the decryptMessage method.
* Both the encrypted and decrypted messages are printed.

**4. Helper Methods:**

* **bToS(byte[] cipher)**: Converts a byte array (ciphertext or plaintext) to a string for easy display by converting each byte to a string and appending it to temp.

**5. Encryption (encryptMessage):**

Encrypts a message using the formula:

ciphertext=messageemod  N\text{ciphertext} = \text{message}^e \mod Nciphertext=messageemodN

Here, the message is treated as a BigInteger, raised to the power of e (public exponent), and taken modulo N.

**6. Decryption (decryptMessage):**

Decrypts a message using the formula:

plaintext=ciphertextdmod  N\text{plaintext} = \text{ciphertext}^d \mod Nplaintext=ciphertextdmodN

Here, the ciphertext is raised to the power of d (private exponent), and taken modulo N to retrieve the original plaintext message.

**7. Important Concepts:**

* **Public Key (e, N)**: Used to encrypt messages.
* **Private Key (d, N)**: Used to decrypt messages.
* **Modular Arithmetic**: Both encryption and decryption use modular exponentiation (modPow) to ensure messages can be encrypted and decrypted efficiently, even for large numbers.

**8. Execution Flow:**

1. RSA object is created using the default constructor (new RSA()).
2. The user inputs a message.
3. The message is converted to a byte array and encrypted.
4. The encrypted byte array (ciphertext) is decrypted back to the original message.

This code is a basic demonstration of how RSA encryption and decryption work. It's not production-ready, as additional security measures (such as padding and key management) are typically required for real-world applications.

**Assignment no 08**

**DiffieHellmanAlgorithm:**

import java.util.Scanner;

public class DiffieHellmanAlgorithmExample {

    public DiffieHellmanAlgorithmExample() {

    }

    public static void main(String[] var0) {

        try (Scanner var17 = new Scanner(System.in)) {

            System.out.println("Both the users should be agreed upon the public keys G and P");

            System.out.println("Enter value for public key G:");

            long var3 = var17.nextLong();

            System.out.println("Enter value for public key P:");

            long var1 = var17.nextLong();

            System.out.println("Enter value for private key a selected by user1:");

            long var7 = var17.nextLong();

            System.out.println("Enter value for private key b selected by user2:");

            long var11 = var17.nextLong();

            long var5 = calculatePower(var3, var7, var1);

            long var9 = calculatePower(var3, var11, var1);

            long var13 = calculatePower(var9, var7, var1);

            long var15 = calculatePower(var5, var11, var1);

            System.out.println("Secret key for User1 is:" + var13);

            System.out.println("Secret key for User2 is:" + var15);

        }

    }

    private static long calculatePower(long var0, long var2, long var4) {

        long var6 = 0L;

        if (var2 == 1L) {

            return var0;

        } else {

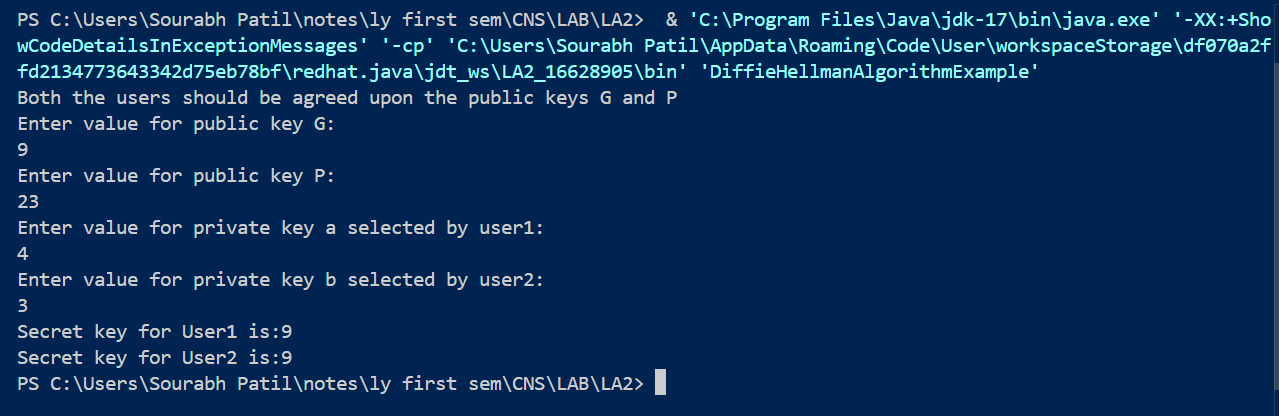
            var6 = (long) Math.pow((double) var0, (double) var2) % var4;

            return var6;

        }

    }

}



T

his Java code demonstrates the **Diffie-Hellman Key Exchange Algorithm**, a method used for securely exchanging cryptographic keys over a public channel. The two parties, User1 and User2, agree on public values and then each selects a private value to compute a shared secret key.

**Explanation of the Code:**

1. **Public Keys:**
   * P: A prime number chosen publicly.
   * G: A base (primitive root modulo P), another public value.
2. **Private Keys:**
   * a: Private key selected by **User1**.
   * b: Private key selected by **User2**.
3. **Intermediate Values:**
   * x: Calculated by User1 as Gamod  PG^a \mod PGamodP, then sent to User2.
   * y: Calculated by User2 as Gbmod  PG^b \mod PGbmodP, then sent to User1.
4. **Shared Secret Keys:**
   * After exchanging the intermediate values (x and y):
     + User1 calculates the shared key ka as yamod  Py^a \mod PyamodP (using the value y received from User2).
     + User2 calculates the shared key kb as xbmod  Px^b \mod PxbmodP (using the value x received from User1).

**Code Walkthrough:**

1. **User Input for Public and Private Keys**:
   * P (a prime number) and G (a generator) are public keys, agreed upon by both users.
   * a (User1's private key) and b (User2's private key) are input by the users.
2. **Calculation of Intermediate Keys**:
   * x = G^a mod P: This value is sent from User1 to User2.
   * y = G^b mod P: This value is sent from User2 to User1.
3. **Calculation of Secret Keys**:
   * User1 calculates their shared secret ka = y^a mod P (using User2's intermediate value y).
   * User2 calculates their shared secret kb = x^b mod P (using User1's intermediate value x).
4. **Output**:
   * Both users will have the same secret key (ka == kb), as the Diffie-Hellman algorithm ensures both parties derive the same shared secret.

**Key Methods:**

* **calculatePower(long x, long y, long P)**: This method computes xymod  Px^y \mod PxymodP. If y equals 1, it directly returns x. Otherwise, it computes the power using Math.pow() and then applies modulo P.

**Assignment no 09**

**SHA:**

import java.security.MessageDigest;

import java.util.Scanner;

public class SHA {

    public static void main(String[] a) {

        try (Scanner sc = new Scanner(System.in)) {

            try {

                MessageDigest md = MessageDigest.getInstance("SHA1");

                System.out.println("Enter the Message: ");

                String input = sc.nextLine();

                md.update(input.getBytes());

                byte[] output = md.digest();

                System.out.println();

                System.out.println("SHA of \"" + input + "\" = " + bytesToHex(output));

                System.out.println(" ");

                System.out.println("Enter the Message: ");

                String input1 = sc.nextLine();

                md.update(input1.getBytes());

                output = md.digest();

                System.out.println();

                System.out.println("SHA of \"" + input1 + "\" = " + bytesToHex(output));

                System.out.println(" ");

                System.out.println("Enter the Message: ");

                String input2 = sc.nextLine();

                md.update(input2.getBytes());

                output = md.digest();

                System.out.println();

                System.out.println("SHA of \"" + input2 + "\" = " + bytesToHex(output));

                System.out.println("");

            } catch (Exception e) {

                System.out.println("Exception: " + e);

            }

        }

    }

    public static String bytesToHex(byte[] b) {

        char hexDigit[] = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F' };

        StringBuffer buf = new StringBuffer();

        for (int j = 0; j < b.length; j++) {

            buf.append(hexDigit[(b[j] >> 4) & 0x0f]);

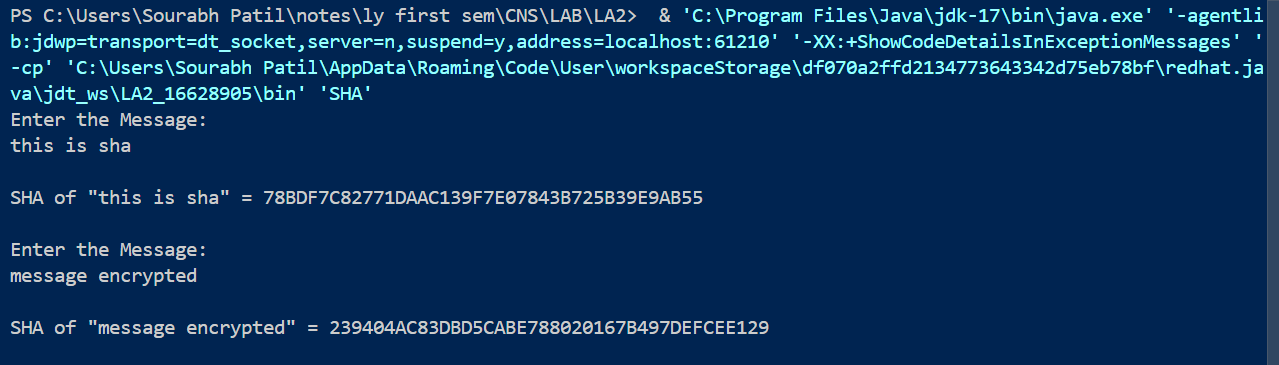
            buf.append(hexDigit[b[j] & 0x0f]);

        }

        return buf.toString();

    }

}

****

Explanation-

This Java program implements **SHA-1 hashing** using the MessageDigest class from the java.security package. It allows the user to input multiple messages, generates their SHA-1 hash values, and outputs them in hexadecimal format.

**Explanation of the Code:**

1. **Importing Libraries**:
   * java.security.\*: Provides classes such as MessageDigest for cryptographic hash functions.
   * java.util.Scanner: To take user input from the console.
2. **Main Class (SHA)**:
   * The main method prompts the user to input a message, computes its SHA-1 hash, and prints the result.
3. **Key Concepts**:
   * **SHA-1 (Secure Hash Algorithm 1)**: A cryptographic hash function that takes an input and produces a 160-bit (20-byte) hash value. It is no longer considered secure for cryptographic purposes but is still used in legacy systems.
   * **MessageDigest**: Used to compute the SHA-1 hash of input messages.

**Code Walkthrough:**

1. **User Input**:
   * A Scanner object is created to read input messages from the user.
2. **Message Digest Initialization**:
   * The MessageDigest.getInstance("SHA1") method creates a MessageDigest object initialized to use the SHA-1 hashing algorithm.
3. **Hashing Process**:
   * The md.update(input.getBytes()) method feeds the user's input (converted to bytes) into the digest algorithm.
   * md.digest() computes the final hash value for the input message, returning it as a byte array.
4. **Hexadecimal Conversion**:
   * The bytesToHex(byte[] b) method converts the resulting byte array into a hexadecimal string for easy readability.
5. **Multiple Inputs**:
   * The program allows the user to input three separate messages, calculates the SHA-1 hash for each, and prints the results.

**Example Output:**

mathematica

Copy code

Enter the Message:

Hello World

SHA of "Hello World" = 2EF7BDE608CE5404E97D5F042F95F89F1C232871

Enter the Message:

Test

SHA of "Test" = A94A8FE5CCB19BA61C4C0873D391E987982FBBD3

Enter the Message:

Java Programming

SHA of "Java Programming" = 12E1215D4C9A3E8B3FFBE2E9F9939B642AA4290D

**Key Method:**

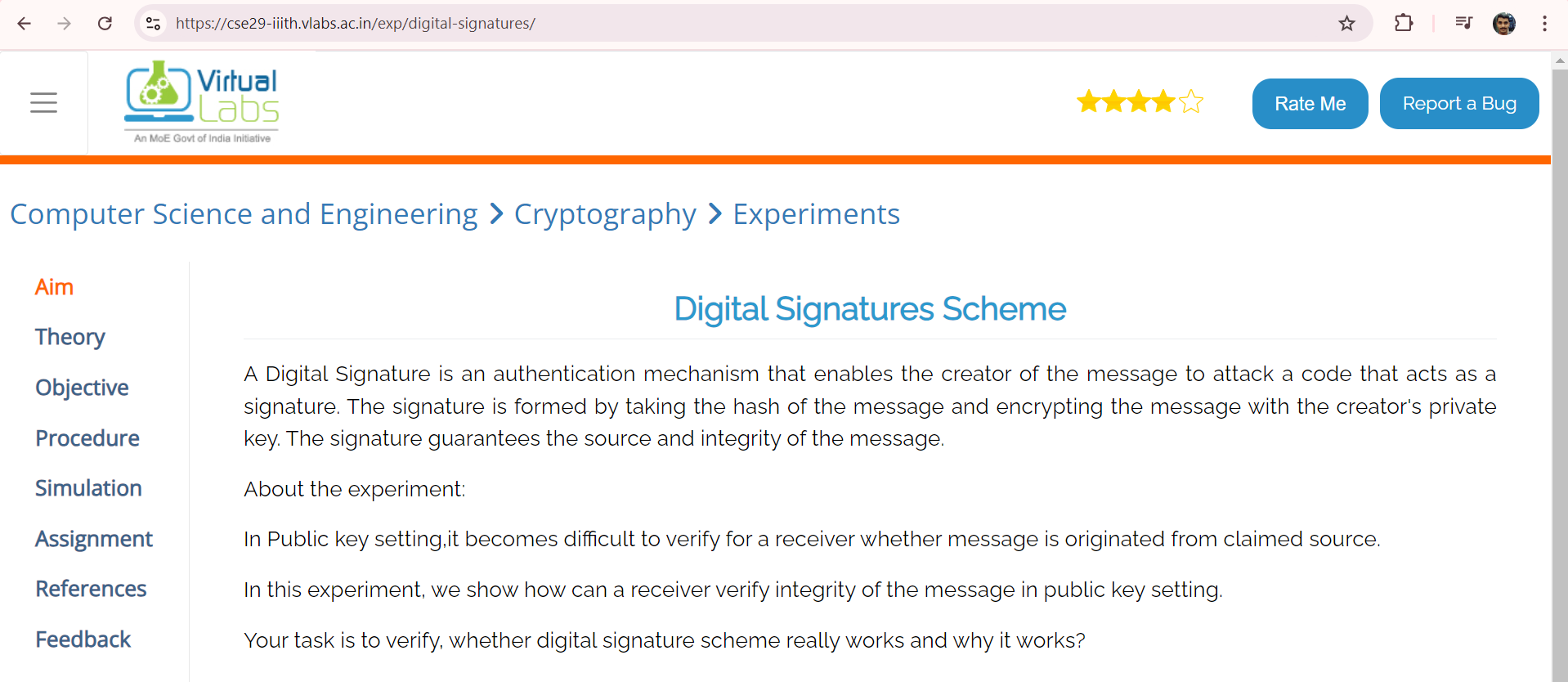
* **bytesToHex(byte[] b)**:
  + This method converts a byte array into a hexadecimal string.
  + The byte values are processed in two steps:
    - (b[j] >> 4) & 0x0f extracts the higher 4 bits (hex digit).
    - b[j] & 0x0f extracts the lower 4 bits (hex digit).
  + The hexadecimal characters are appended to a StringBuffer for efficient string concatenation.

**Improvements:**

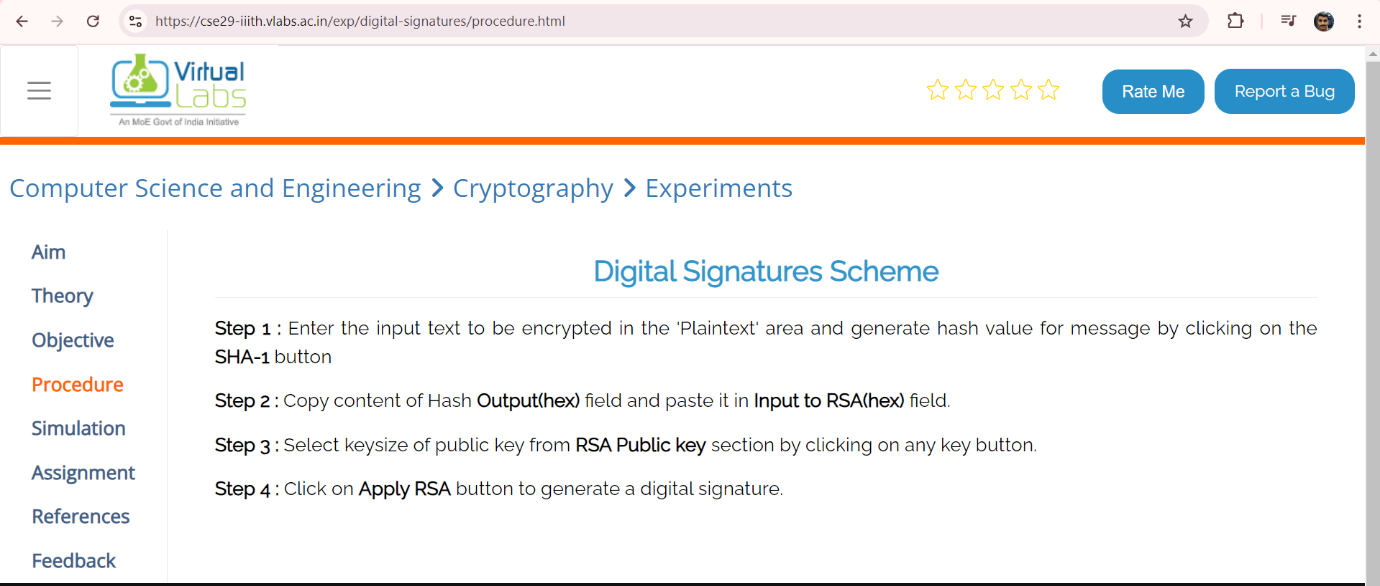
1. **Use StringBuilder**:
   * Replace StringBuffer with StringBuilder in bytesToHex for better performance since synchronization is not needed in this case.
2. **SHA-256 Update**:
   * Since SHA-1 is deprecated in security-critical contexts, you could update the program to use SHA-256 by replacing "SHA1" with "SHA-256" in MessageDigest.getInstance().

**Assignment no 10**

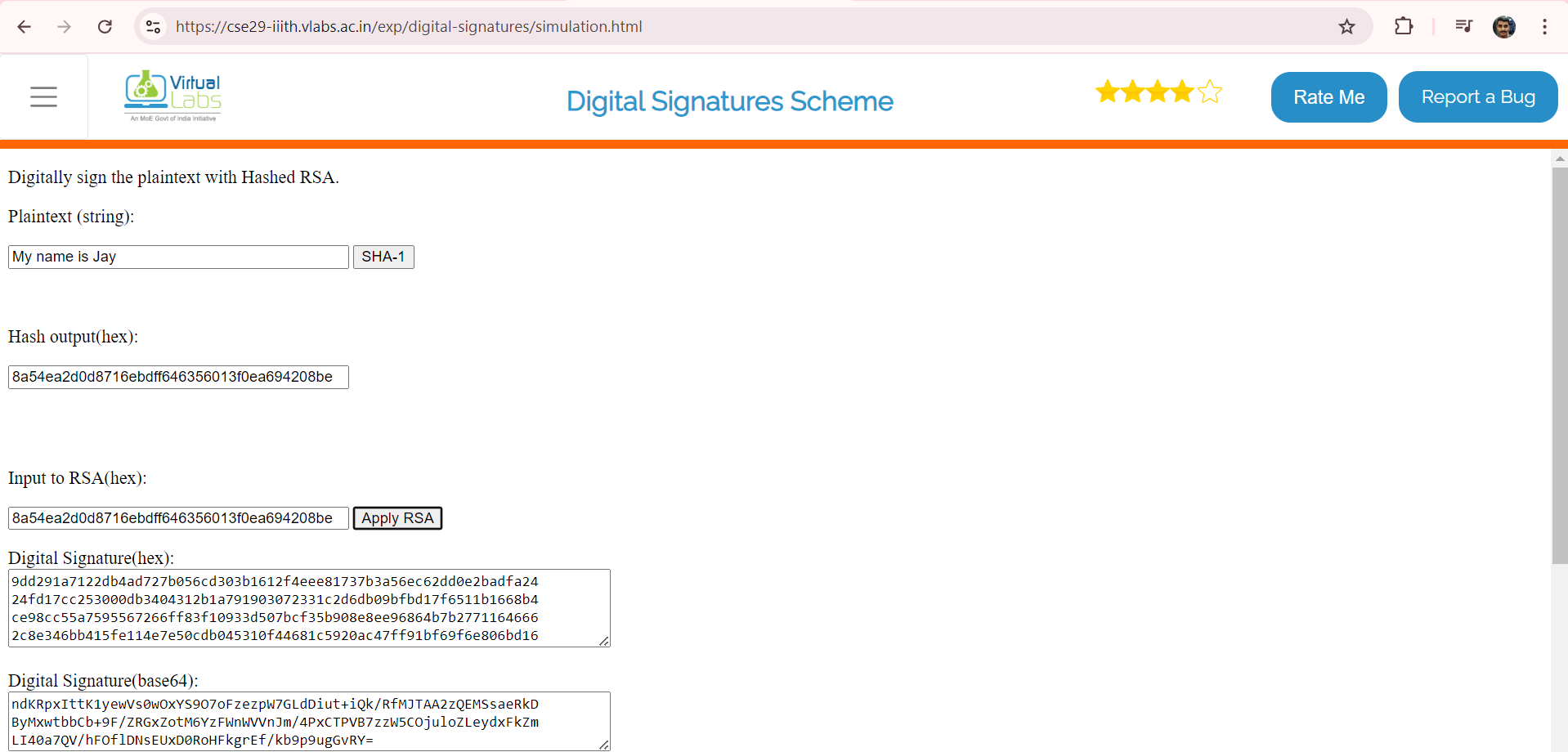
1]Aim:



2]Objective: To understand "How and Why Digital signature schemes?"

3]Procedure:  


4]Simulation:





Explanation-

This program demonstrates how to create and verify a **digital signature** using the RSA algorithm and the SHA-256 hash function in Java. Here's an explanation of each part:

**Explanation:**

1. **Imports**:
   * The program uses various security-related classes (KeyPair, Signature, PrivateKey, PublicKey, etc.) from the java.security package.
   * HexFormat: From java.util (Java 17+), it provides utility methods for converting bytes into hexadecimal strings for easy readability.
2. **Key Concepts**:
   * **Digital Signature**: A way to verify the authenticity and integrity of a message. It involves generating a signature using a private key, which can be verified using the corresponding public key.
   * **RSA Algorithm**: A widely used asymmetric encryption algorithm where a pair of keys (public and private) is used for signing and verifying.
   * **SHA-256 with RSA**: The hashing algorithm (SHA-256) is used to hash the input message, and RSA is used to sign the hash.

**Code Breakdown:**

1. **Constants**:
   * SIGNING\_ALGORITHM: Specifies the cryptographic signing algorithm to be used (SHA256withRSA).
   * RSA: Specifies the RSA algorithm for key pair generation.
2. **Create\_Digital\_Signature Method**:
   * This method takes a byte[] input and a PrivateKey as parameters.
   * It initializes a Signature object with the specified algorithm (SHA256withRSA), signs the data using the private key, and returns the generated signature as a byte array.
3. **Generate\_RSA\_KeyPair Method**:
   * It generates a new RSA key pair (private and public key).
   * A SecureRandom object is used to ensure cryptographically strong randomness.
   * The key size is set to 2048 bits for adequate security.
4. **Verify\_Digital\_Signature Method**:
   * This method takes the input data, the signature to verify, and the public key.
   * It initializes the Signature object for verification using the public key.
   * It then verifies whether the provided signature matches the one generated by the Create\_Digital\_Signature method.
5. **main Method**:
   * The main method generates an RSA key pair.
   * It signs a string message "Hello I am Sujan" using the private key.
   * The generated signature is displayed in hexadecimal format for easy reading.
   * The program then verifies the signature using the corresponding public key and prints the verification result (true or false).

**Detailed Example:**

1. **Key Pair Generation**:
   * A new RSA key pair is generated using KeyPairGenerator with a key size of 2048 bits. The key pair consists of a private key (for signing) and a public key (for verification).
2. **Signing Process**:
   * The input message ("Hello I am Sujan") is signed using the private key. This signature ensures that the message comes from the owner of the private key and that the message has not been tampered with.
   * The output is printed in hexadecimal format using the HexFormat.of().formatHex(signature) function.
3. **Verification Process**:
   * The program verifies whether the message matches the generated signature using the public key. If the verification is successful, it prints true, indicating that the signature is valid.

**Example Output:**

yaml

Copy code

Signature Value:

3046022100e3b2cf5a5f9f54f89ba8e19364fa2fe92420a48c50b15479ab47a6bc9f11ab022022100b08f5b988d9fb6ae9c91f3a8ddf74fe40293d7a95a2162d51b484bdb30d09

Verification: true

**Key Classes Used:**

* **KeyPairGenerator**: Generates a public-private key pair.
* **Signature**: Used for both signing and verifying digital signatures.
* **HexFormat**: Converts byte arrays into a human-readable hexadecimal format.

**Improvements:**

1. **Exception Handling**:
   * The code can be improved by adding more detailed exception handling to catch specific errors related to key generation, signing, or verification.
2. **User Input**:
   * To make the program more dynamic, it can be extended to take user input for the message instead of hardcoding the message in the program.

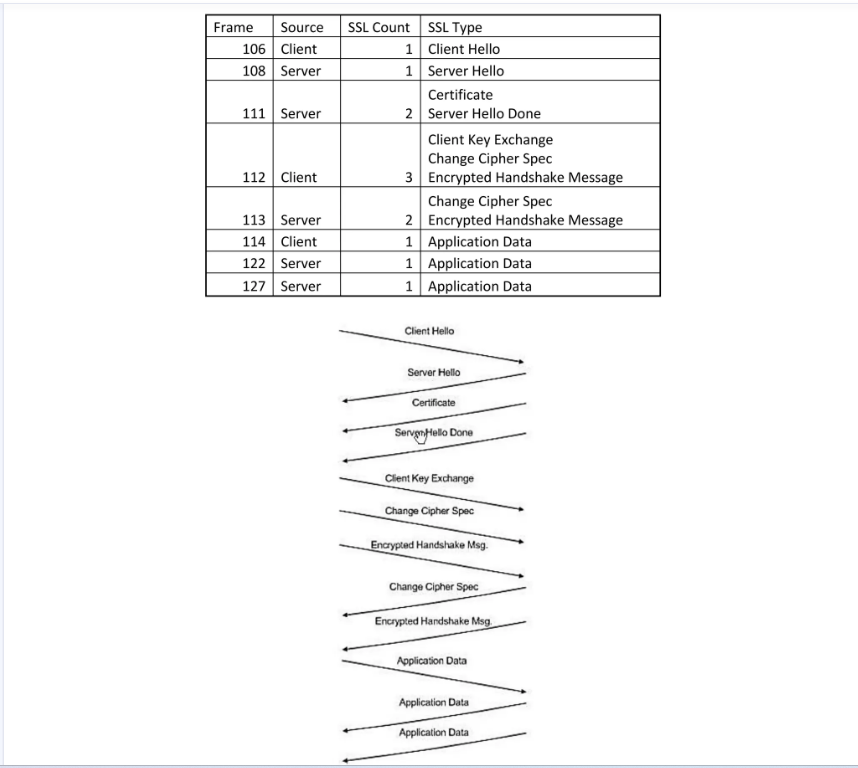
With these key concepts, this program demonstrates how digital signatures can be used to verify the integrity and authenticity of messages.

**Assignment no 11**

**SSL using Wireshark packets**

**Q1] 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.**

Ans-



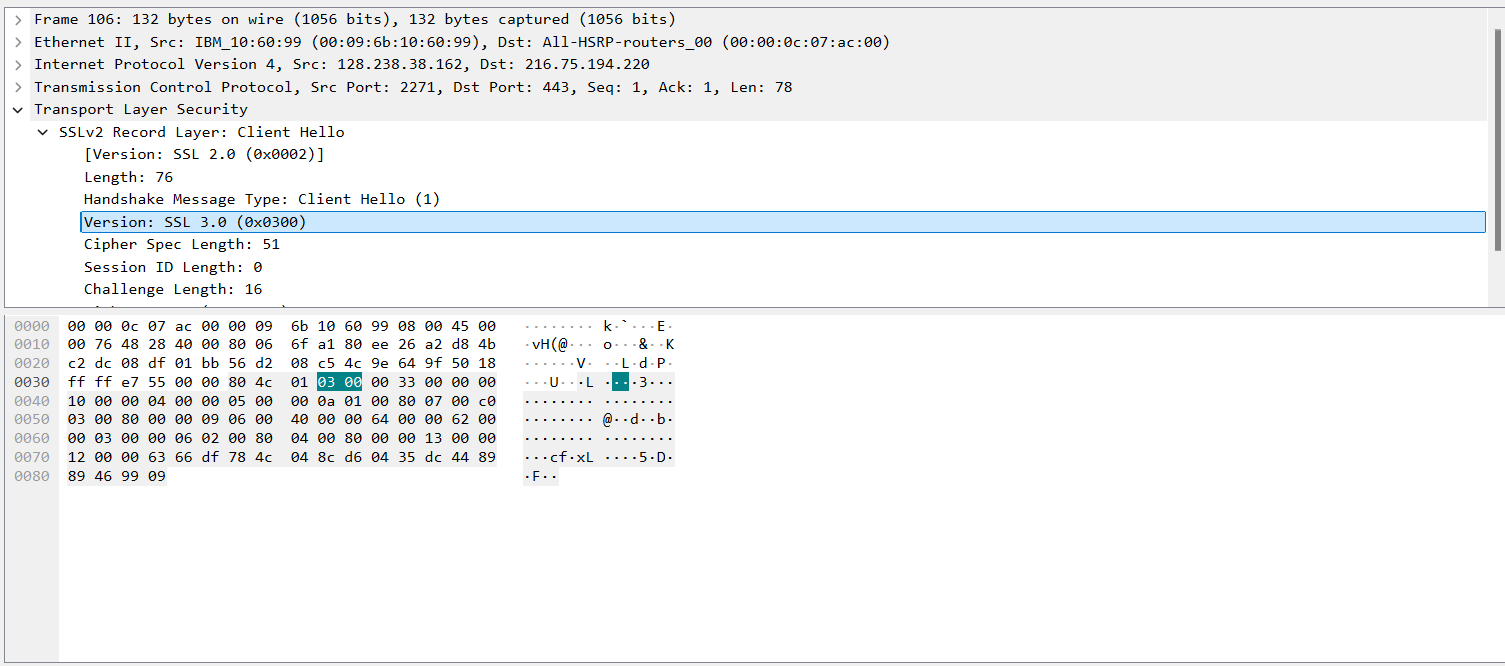
**Q2] Each of the SSL records begins with the same three fields (with possibly different values). One of these fields is “content type” and has length of one byte. List all three fields and their lengths.**

Ans-

Content Type : 1 byte

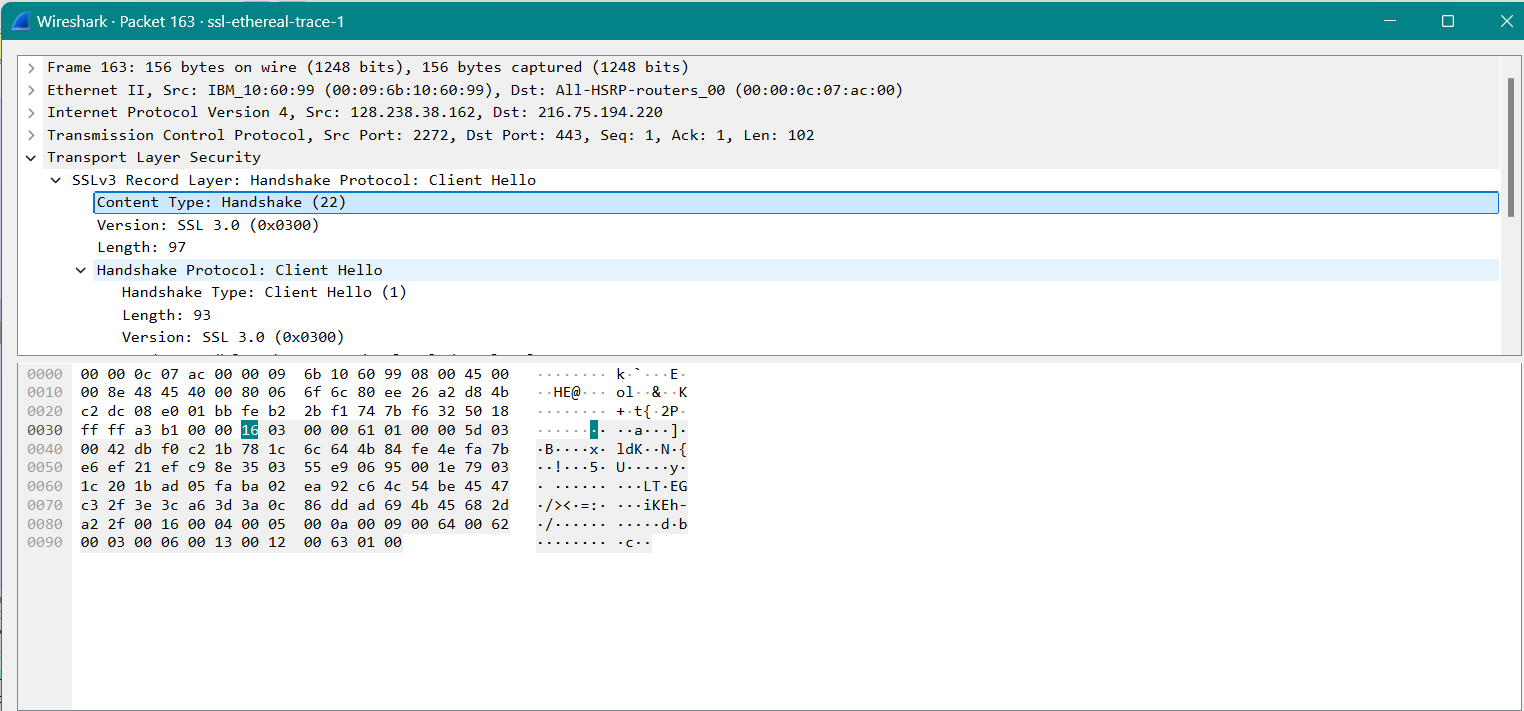
Version : 2bytes

Length : 2bytes



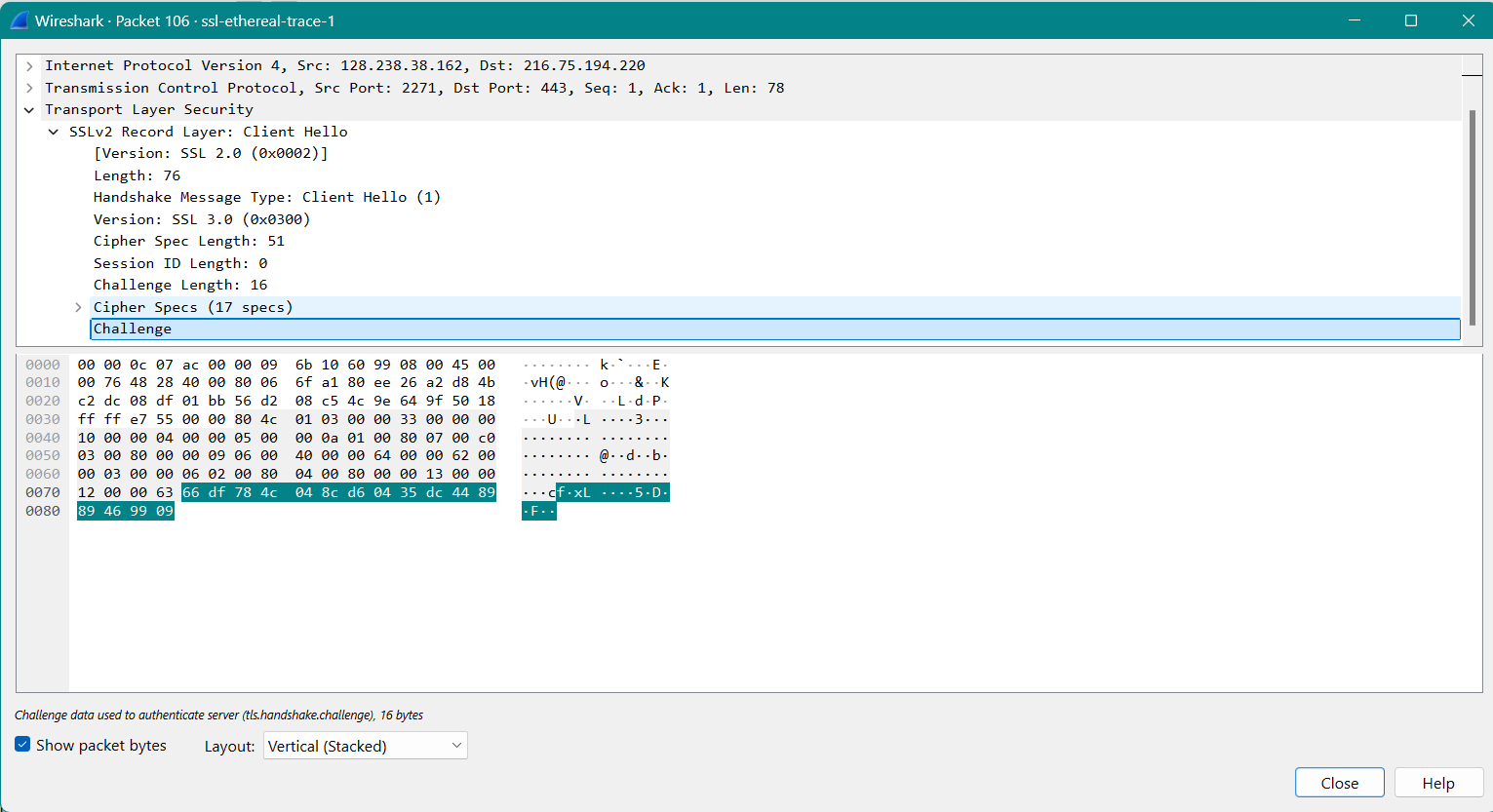
**Q3] Expand the ClientHello record. (If your trace contains multiple ClientHello records, expand the frame that contains the first one.) What is the value of the content type?**

Ans- The content type is 22,for Handshake message ,with a handshake type of 01,Client Hello.



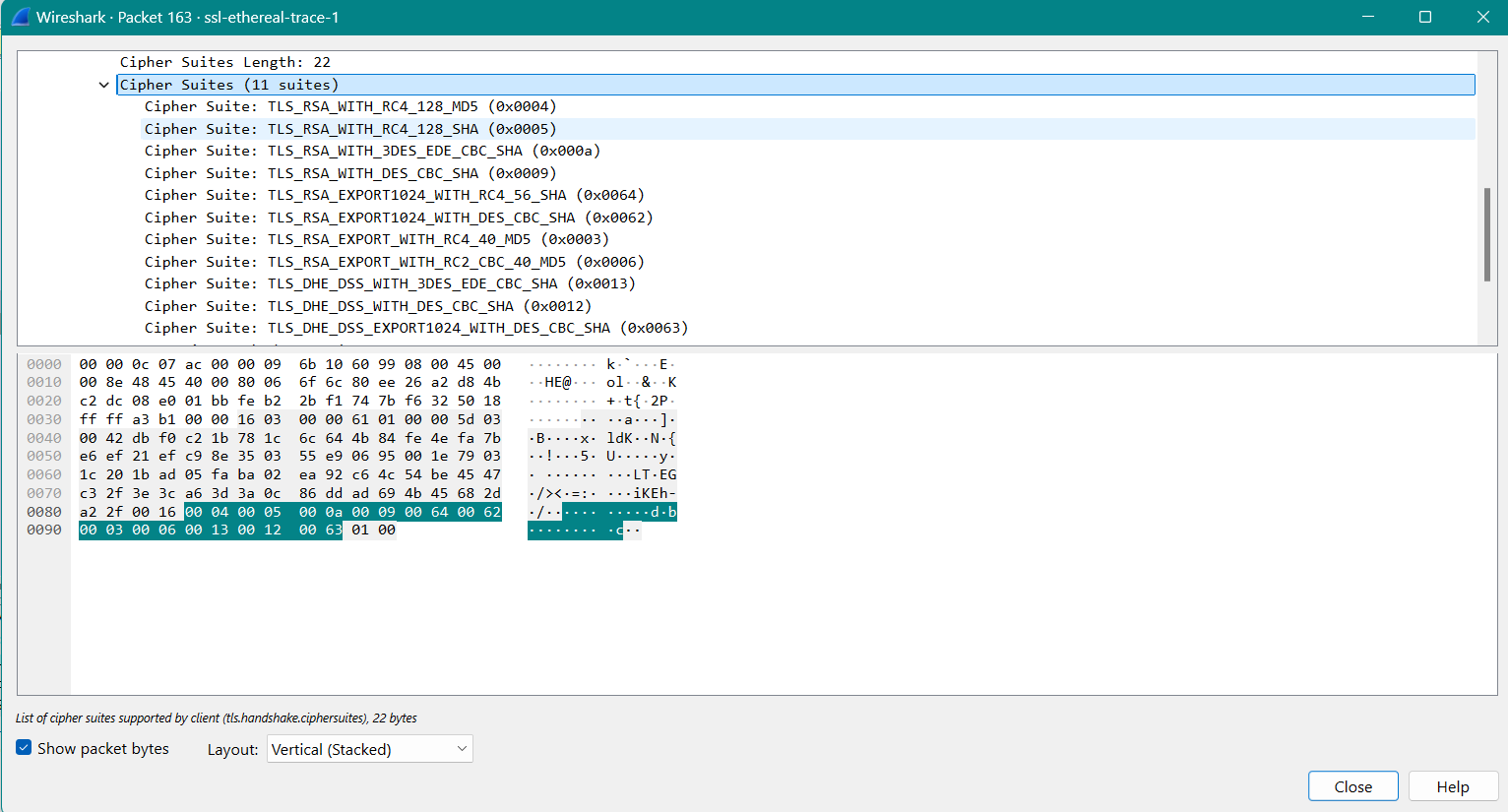
**Q4] Does the ClientHello record contain a nonce (also known as a “challenge”)? If so, what is the value of the challenge in hexadecimal notation?**

Ans-The client hello challenge is –



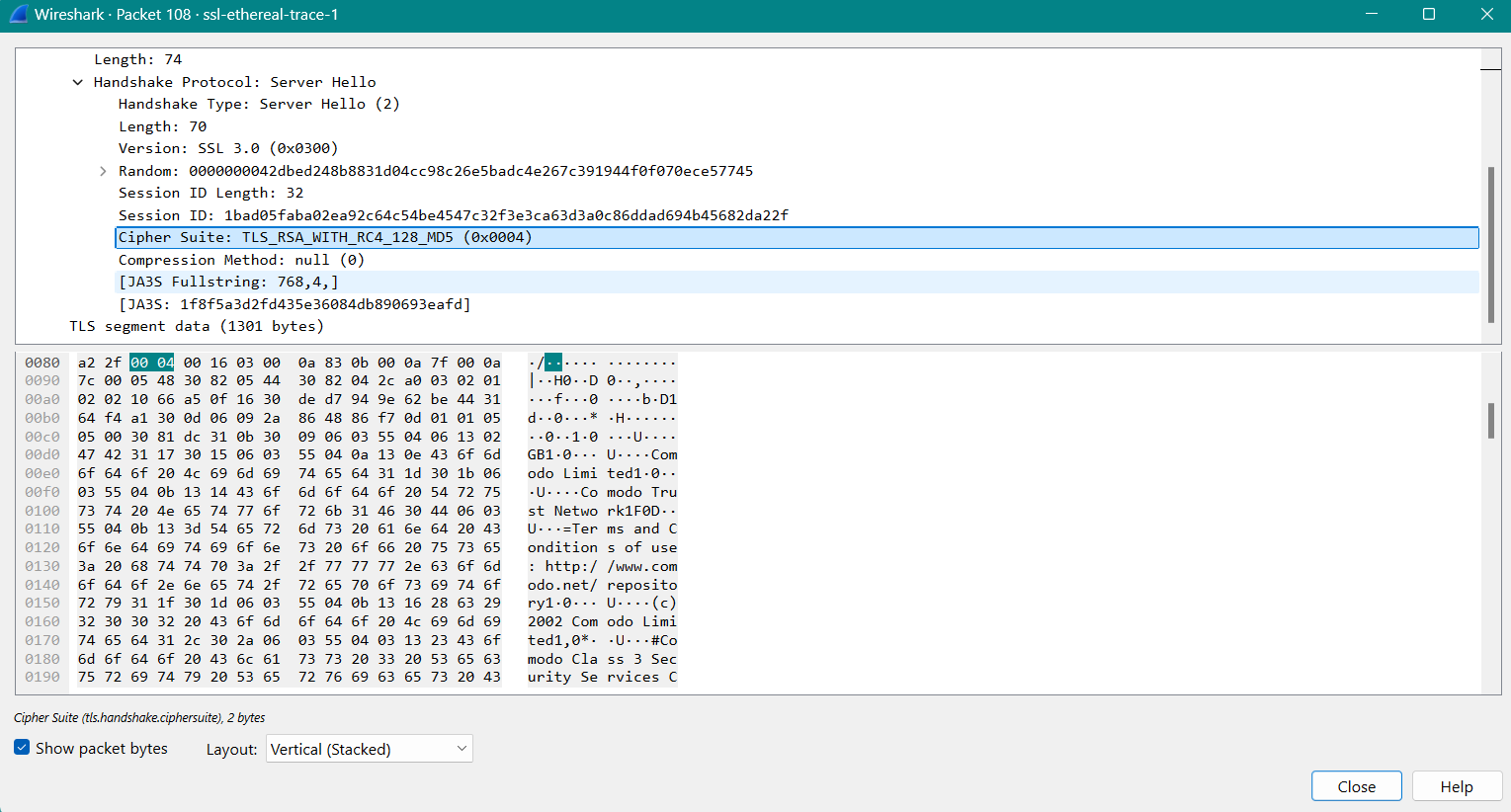
**Q5] Does the ClientHello record advertise the cyber suites it supports? If so, in the first listed suite, what are the public-key algorithm, the symmetric-key algorithm, and the hash algorithm?**

Ans-The first suite uses RSA for public key cryptograpgy,RC4 for the symmetric-key cipher and uses MD5 hash algorithm.



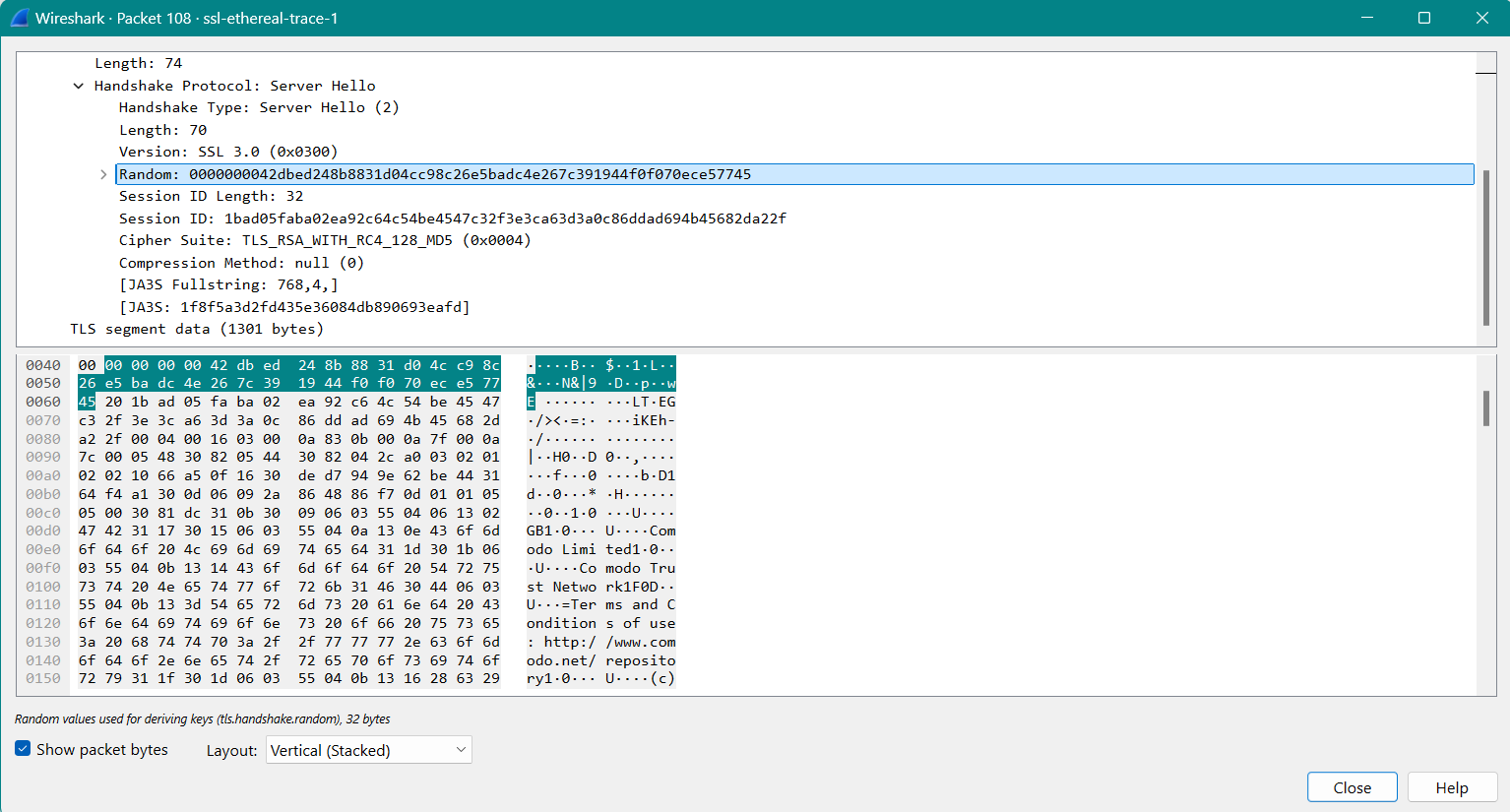
**Q6] Locate the ServerHello SSL record. Does this record specify a chosen cipher suite? What are the algorithms in the chosen cipher suite?**

Ans- The first suite uses RSA for public key cryptograpgy,RC4 for the symmetric-key cipher and uses MD5 hash algorithm.



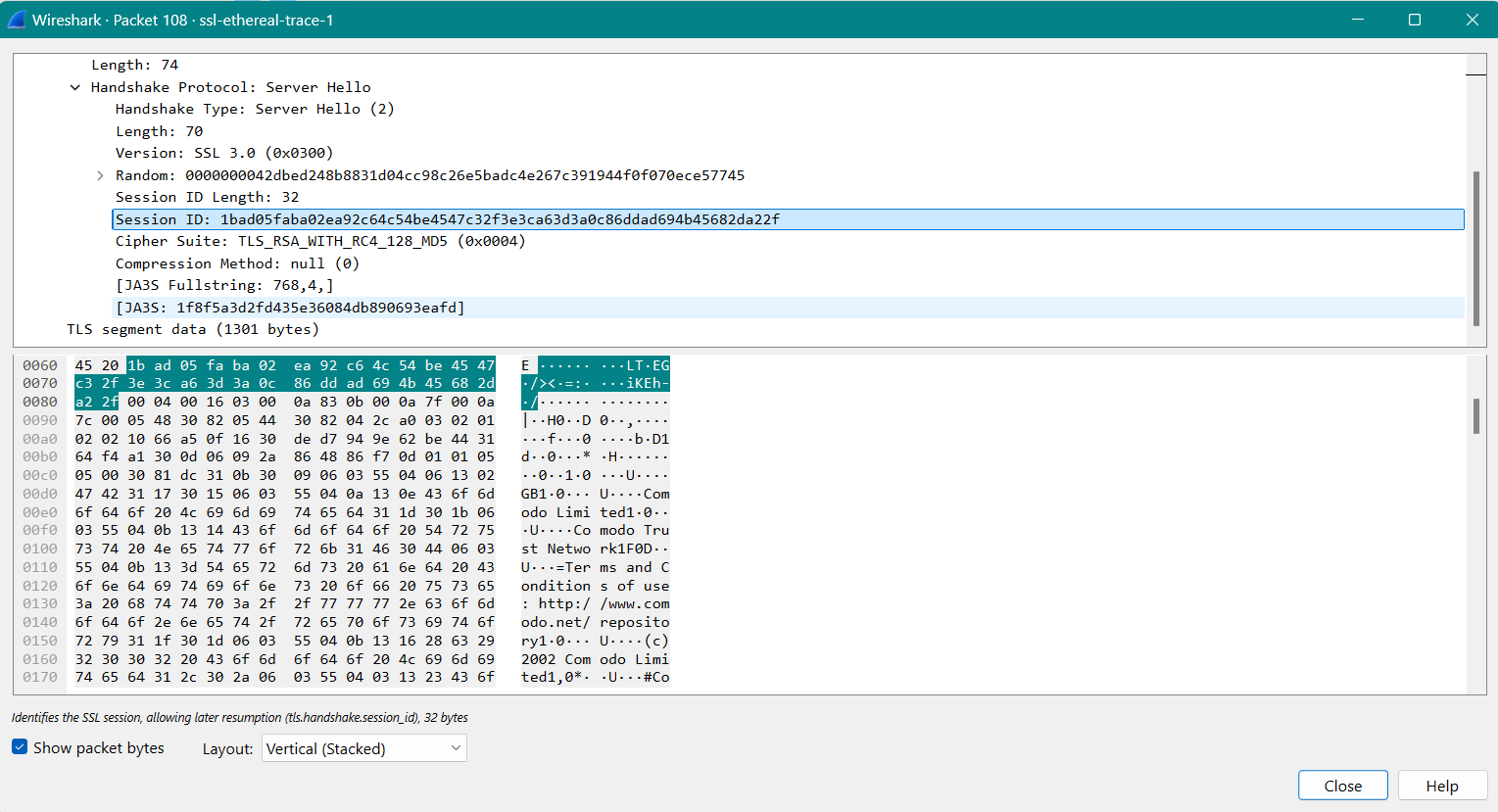
**Q7] Does this record include a nonce? If so, how long is it? What is the purpose of the client and server nonces in SSL?**

Ans-Yes,the record includes nonce listed under random.The nonce is 32 bits long,28 for the data and 4 for the time.The purpose is to prevent replay attack.



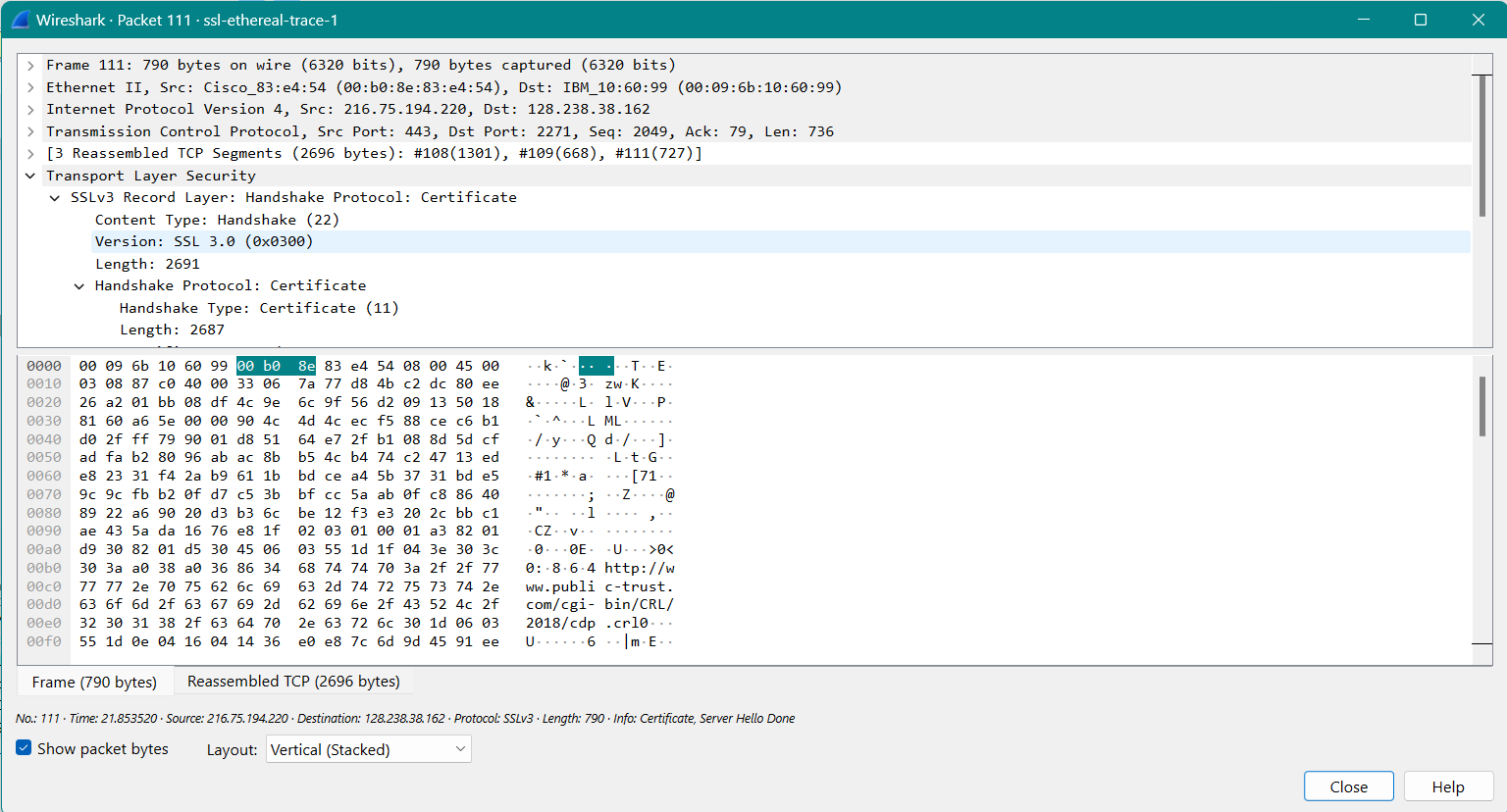
**Q8] Does this record include a session ID? What is the purpose of the session ID?**

Ans- Yes,it provides a unique persistent identifier for the SSL session which is sent.The client may resume the same session later by using the server provided session id when it sends the ClientHello.



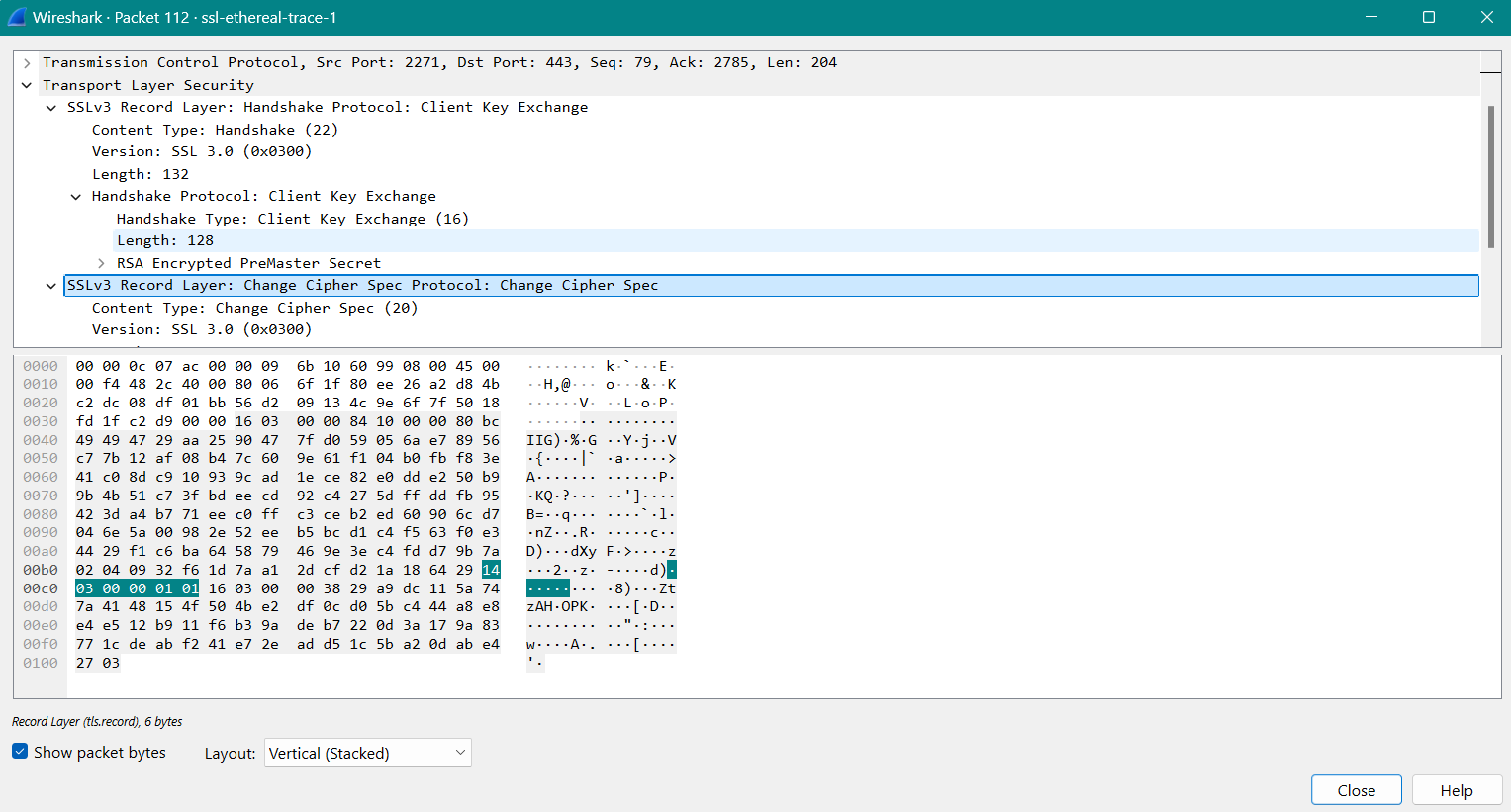
**Q9] Does this record contain a certificate, or is the certificate included in a separate record. Does the certificate fit into a single Ethernet frame?**

Ans-There is no certificate it is another record.It fits into a single ethernet frame.



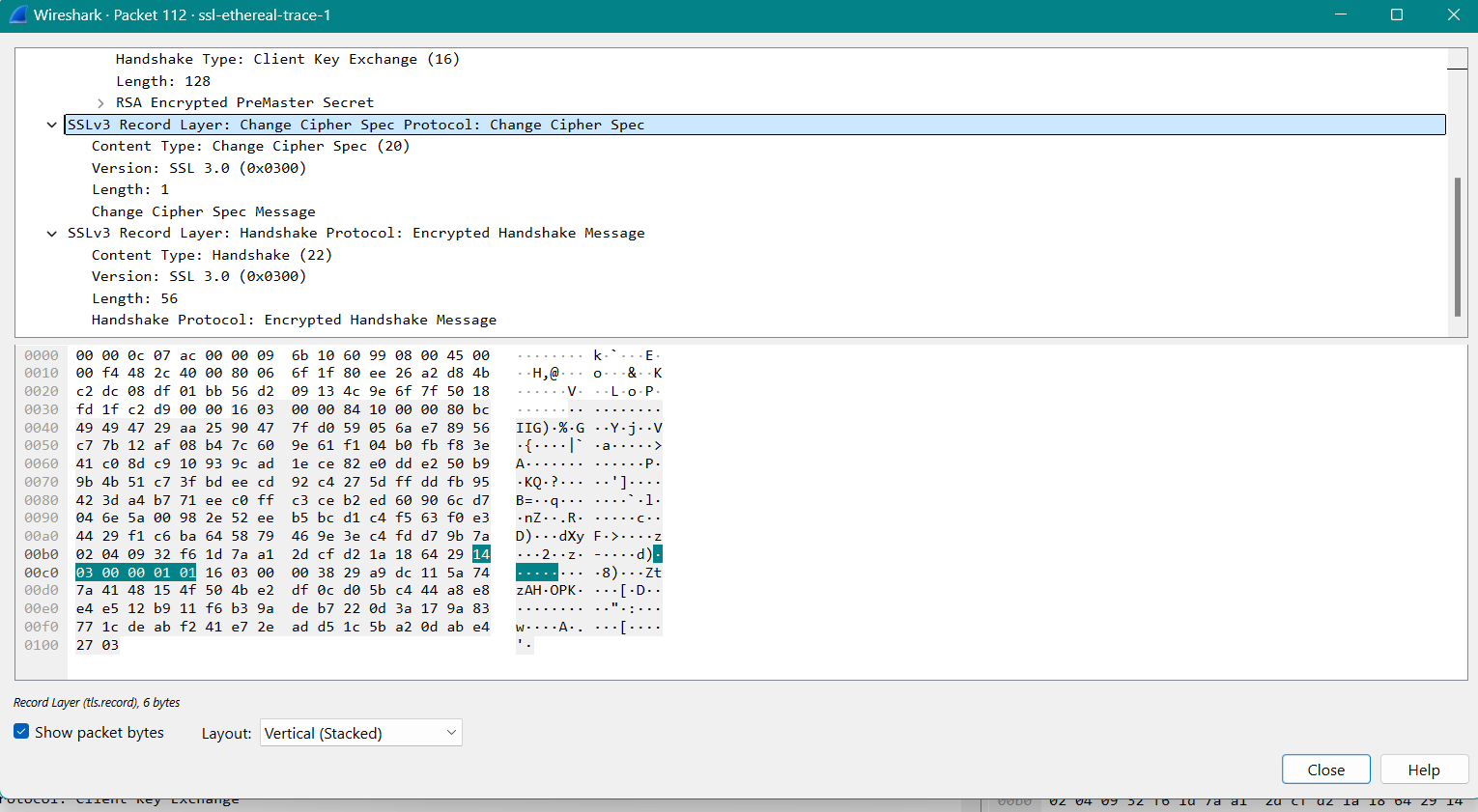
**Q10] Locate the client key exchange record. Does this record contain a pre-master secret? What is this secret used for? Is the secret encrypted? If so, how? How long is the encrypted secret?**

Ans-Yes it contains a premaster secret.Used on both sides client and server to make a secret which is used to generate keys for mac and encryption.The server is encrypted with the public key of server,which cleient extracts from the certificate sent.



**Q11] What is the purpose of the Change Cipher Spec record? How many bytes is the record in your trace?**

Ans-The purpose of the Change Cipher Spec record is to indicate that the contents of the following SSL records sent by the client (data, not header) will be encrypted. This record is 6 bytes long: 5 for the header and 1 for the message segment.



**Q12] In the encrypted handshake record, what is being encrypted? How?**

Ans-In the encrypted handshake record, a MAC of the concatenation of all the previous handshake messages sent from this client is generated and sent to the server.

**Q13] 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?**

Ans-Yes, the server will also send a Change Cipher Spec record and encrypted handshake to the client. The server's encrypted handshake record is different from that sent by the client because it contains the concatenation of all the handshake messages sent from the server rather than from the client. Otherwise, the records would end up being the same.

**Q14] 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?**

Ans-Application data is encrypted using symmetric key encryption algorithm chosen in the handshake phase (RC4) using the keys generated using the pre-master key and nonces from both client and server. The client encryption key is used to encrypt the data being sent from client to server and the server encryption key is used to encrypt the data being sent from the server to the client.

Q15] Comment on and explain anything else that you found interesting in the trace.

Ans-NIL.