**Walchand College Of Engineering**

**CNS LAB – LA2**

**Name:** Yash Rajesh Nawale

**PRN:** 21510074

**Batch:** B4

1. **Implementation of RSA**

RSA (Rivest–Shamir–Adleman) is one of the most widely used public-key cryptosystems, primarily for secure data transmission. It relies on the mathematical properties of large prime numbers and modular arithmetic.

**Key Components of RSA:**

* **Prime Numbers**: Two large prime numbers p and q are selected.
* **Modulus (n)**: The product of p and q is computed to obtain the modulus n = p \* q. This value is used in both the encryption and decryption processes.
* **Euler’s Totient Function (φ(n))**: Euler's Totient function is calculated as φ(n) = (p-1) \* (q-1). This function is crucial for generating keys.
* **Public and Private Keys**:
  + The **public key** is a pair (e, n) where e is the public exponent. e is chosen such that 1 < e < φ(n) and the greatest common divisor (gcd) of e and φ(n) is 1.
  + The **private key** is d, calculated as the modular inverse of e with respect to φ(n). The relationship between e and d ensures that e \* d ≡ 1 (mod φ(n)).
* **Encryption:**

To encrypt a plaintext message M (where M is less than n), the following formula is used:

C=M^e mod n

Here, C is the ciphertext that can be safely transmitted.

* **Decryption:**

To decrypt the ciphertext C, the following formula is used:

M=C^dmod n

This returns the original message M

Code:

#include <bits/stdc++.h>

using namespace std;

set<int> prime;  // a set of prime numbers

int public\_key;  // stores the public key (e)

int private\_key; // stores the private key (d)

int n;           // n = p \* q (part of the key pair)

void primefiller() {

    cout << "Generating prime numbers up to 250 using Sieve of Eratosthenes...\n";

    vector<bool> seive(250, true);

    seive[0] = false;

    seive[1] = false;

    for (int i = 2; i < 250; i++) {

        for (int j = i \* 2; j < 250; j += i) {

            seive[j] = false;

        }

    }

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

        if (seive[i]) prime.insert(i);

    }

    cout << "Prime numbers generated and stored.\n";

}

int pickrandomprime() {

    int k = rand() % prime.size();

    auto it = prime.begin();

    while (k--) it++;

    int ret = \*it;

    prime.erase(it);

    cout << "Random prime selected: " << ret << "\n";

    return ret;

}

void setkeys() {

    cout << "Setting up keys...\n";

    int prime1 = pickrandomprime();

    int prime2 = pickrandomprime();

    cout << "Selected primes p = " << prime1 << " and q = " << prime2 << endl;

    n = prime1 \* prime2;

    cout << "Calculated n (p \* q) = " << n << endl;

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

    cout << "Calculated φ(n) = " << fi << endl;

    int e = 2;

    while (1) {

        if (\_\_gcd(e, fi) == 1) break;

        e++;

    }

    public\_key = e;

    cout << "Public key (e) selected as: " << public\_key << endl;

    int d = 2;

    while (1) {

        if ((d \* e) % fi == 1) break;

        d++;

    }

    private\_key = d;

    cout << "Private key (d) calculated as: " << private\_key << endl;

}

long long int encrypt(double message) {

    int e = public\_key;

    long long int encrypted\_text = 1;

    // cout << "Encrypting message character (ASCII): " << message << endl;

    while (e--) {

        encrypted\_text \*= message;

        encrypted\_text %= n;

    }

    // cout << "Encrypted value: " << encrypted\_text << endl;

    return encrypted\_text;

}

long long int decrypt(int encrypted\_text) {

    int d = private\_key;

    long long int decrypted = 1;

    // cout << "Decrypting message code: " << encrypted\_text << endl;

    while (d--) {

        decrypted \*= encrypted\_text;

        decrypted %= n;

    }

    // cout << "Decrypted ASCII value: " << decrypted << endl;

    return decrypted;

}

vector<int> encoder(string message) {

    // cout << "\nEncoding message...\n";

    vector<int> form;

    for (auto& letter : message) {

        // cout << "Character: " << letter << " ASCII: " << (int)letter << endl;

        form.push\_back(encrypt((int)letter));

    }

    return form;

}

string decoder(vector<int> encoded) {

    // cout << "\nDecoding message...\n";

    string s;

    for (auto& num : encoded) {

        s += decrypt(num);

    }

    return s;

}

int main() {

    primefiller();

    setkeys();

    string message;

    cout << "Input the text you want to encrypt: ";

    getline(cin, message);

    vector<int> coded = encoder(message);

    cout << "\nInitial message:\n" << message << endl;

    cout << "\nThe encoded message (encrypted by public key):\n";

    for (auto& p : coded) cout << p << " ";

    cout << endl;

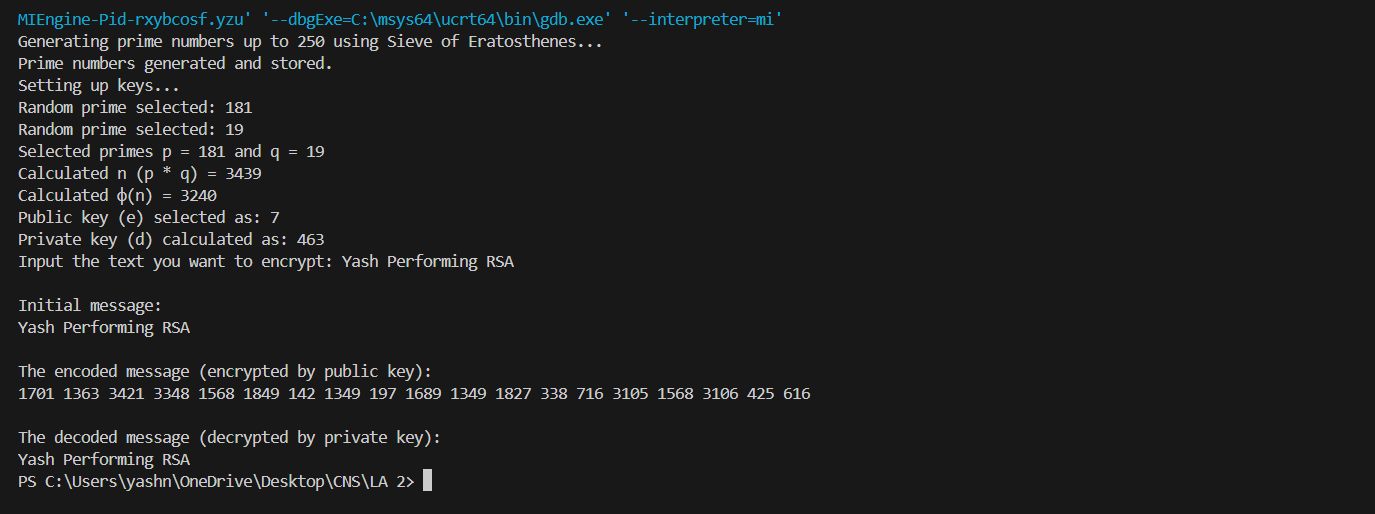
    cout << "\nThe decoded message (decrypted by private key):\n";

    cout << decoder(coded) << endl;

    return 0;

}

Ouput:



**Diffie-Hellman Key Exchange**

Explanation:

The Diffie-Hellman Key Exchange is a method that allows two parties to securely establish a shared secret key over a public channel. It works as follows:

1. **Prime Number (p) and Primitive Root (g)**: Both parties agree on a large prime number p and a primitive root g modulo p. These values are public.
2. **Private Keys**: Each party selects a private key (privateKeyA for User A and privateKeyB for User B) that remains secret.
3. **Public Keys**: Using their private keys and g, they compute public keys and exchange them:
   * PublicKeyA = (g^privateKeyA) mod p
   * PublicKeyB = (g^privateKeyB) mod p
4. **Shared Secret**: Both parties then compute the same shared secret key using their private key and the other party's public key:
   * SharedSecretA = (PublicKeyB^privateKeyA) mod p
   * SharedSecretB = (PublicKeyA^privateKeyB) mod p

The shared secret is identical for both parties and can be used for secure communication. The security relies on the difficulty of solving the discrete logarithm problem.

Code:

/\* This program calculates the Key for two persons

using the Diffie-Hellman Key exchange algorithm using C++ \*/

#include <cmath>

#include <iostream>

using namespace std;

// Power function to return value of a ^ b mod P

long long int power(long long int a, long long int b, long long int P) {

    if (b == 1)

        return a % P;  // Ensure it returns modulo P

    return (((long long int)pow(a, b)) % P);

}

// Driver program

int main() {

    long long int P, G, x, a, y, b, ka, kb;

    // Taking user input for public keys

    cout << "Enter a prime number P: ";

    cin >> P;

    cout << "Enter a primitive root G for P: ";

    cin >> G;

    // Taking user input for Alice's private key

    cout << "Enter the private key a for Alice: ";

    cin >> a;

    x = power(G, a, P); // Gets the generated public key for Alice

    cout << "Public key generated by Alice (x) = G^a % P = " << x << endl;

    // Taking user input for Bob's private key

    cout << "Enter the private key b for Bob: ";

    cin >> b;

    y = power(G, b, P); // Gets the generated public key for Bob

    cout << "Public key generated by Bob (y) = G^b % P = " << y << endl;

    // Generating the secret key after the exchange of keys

    ka = power(y, a, P); // Secret key for Alice

    kb = power(x, b, P); // Secret key for Bob

    cout << "Secret key for Alice is : " << ka << endl;

    cout << "Secret key for Bob is : " << kb << endl;

    // Verify that both secret keys match

    if (ka == kb) {

        cout << "Shared secret key successfully generated: " << ka << endl;

    } else {

        cout << "Error: Shared secret keys do not match!" << endl;

    }

    return 0;

}

Output:

A computer screen with blue text

Description automatically generated

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

Explanation:

The following code implements the SHA-1 (Secure Hash Algorithm 1) to calculate the message digest of a given input string. SHA-1 is a cryptographic hash function that takes an input and produces a 160-bit (20-byte) hash value known as a message digest.

**Key Steps in SHA-1 Implementation:**

1. **Padding the Message**:
   * The message is padded to ensure its length is congruent to 448 modulo 512. A 1 bit is added followed by 0 bits. The message length (in bits) is then appended as a 64-bit value.
2. **Process in 512-bit Chunks**:
   * The message is processed in chunks of 512 bits (64 bytes). Each chunk is divided into sixteen 32-bit words and extended to eighty 32-bit words using a specific function.
3. **Main Loop**:
   * SHA-1 performs 80 rounds of bitwise operations, using a combination of logical functions (AND, OR, XOR) and circular left shifts. The constants used in these rounds are based on the fractional parts of the square roots of small prime numbers.
4. **Final Hash Value**:
   * The final 160-bit hash is represented as a 40-character hexadecimal string, which is the SHA-1 message digest.

Code:

#include <iostream>

#include <string>

#include <sstream>

#include <iomanip>

#include <bitset>

#include <cstring>

using namespace std;

#define BLOCK\_SIZE 64  // 512-bit block size (64 bytes)

// Circular left shift (rotate left)

unsigned int leftRotate(unsigned int value, unsigned int bits) {

    return (value << bits) | (value >> (32 - bits));

}

// SHA-1 padding

string padMessage(const string& input) {

    unsigned long long originalLength = input.size() \* 8; // in bits

    string paddedMessage = input;

    // Append a '1' bit followed by '0' bits until the length is congruent to 448 mod 512

    cout << "\n=== Padding the Message ===\n";

    cout << "Original message length (in bits): " << originalLength << endl;

    paddedMessage += static\_cast<char>(0x80);  // 0x80 is binary: 10000000

    while (paddedMessage.size() % 64 != 56) {

        paddedMessage += static\_cast<char>(0x00);

    }

    // Append the original message length as a 64-bit big-endian integer

    for (int i = 7; i >= 0; --i) {

        paddedMessage += static\_cast<char>((originalLength >> (i \* 8)) & 0xFF);

    }

    cout << "Padded message length (in bytes): " << paddedMessage.size() << endl;

    cout << "Message after padding (in hex): ";

    for (unsigned char c : paddedMessage) {

        cout << hex << setw(2) << setfill('0') << (int)c;

    }

    cout << "\n\n";

    return paddedMessage;

}

// Process message in 512-bit chunks (64 bytes)

void processChunk(const string& chunk, unsigned int\* H) {

    unsigned int W[80];

    // Break chunk into sixteen 32-bit big-endian words W[0..15]

    cout << "\nProcessing 512-bit chunk:\n";

    for (int i = 0; i < 16; ++i) {

        W[i] = (chunk[i \* 4 + 0] & 0xFF) << 24 |

               (chunk[i \* 4 + 1] & 0xFF) << 16 |

               (chunk[i \* 4 + 2] & 0xFF) << 8 |

               (chunk[i \* 4 + 3] & 0xFF);

        cout << "W[" << i << "]: " << hex << setw(8) << setfill('0') << W[i] << endl;

    }

    // Extend the sixteen 32-bit words into eighty 32-bit words

    for (int i = 16; i < 80; ++i) {

        W[i] = leftRotate(W[i - 3] ^ W[i - 8] ^ W[i - 14] ^ W[i - 16], 1);

    }

    // Initialize hash value for this chunk

    unsigned int a = H[0];

    unsigned int b = H[1];

    unsigned int c = H[2];

    unsigned int d = H[3];

    unsigned int e = H[4];

    // Main loop

    for (int i = 0; i < 80; ++i) {

        unsigned int f, k;

        if (i < 20) {

            f = (b & c) | ((~b) & d);

            k = 0x5A827999;

        } else if (i < 40) {

            f = b ^ c ^ d;

            k = 0x6ED9EBA1;

        } else if (i < 60) {

            f = (b & c) | (b & d) | (c & d);

            k = 0x8F1BBCDC;

        } else {

            f = b ^ c ^ d;

            k = 0xCA62C1D6;

        }

        unsigned int temp = leftRotate(a, 5) + f + e + k + W[i];

        e = d;

        d = c;

        c = leftRotate(b, 30);

        b = a;

        a = temp;

    }

    // Add this chunk's hash to result so far

    H[0] += a;

    H[1] += b;

    H[2] += c;

    H[3] += d;

    H[4] += e;

    cout << "\nUpdated hash values after processing chunk:\n";

    for (int i = 0; i < 5; ++i) {

        cout << "H[" << i << "]: " << hex << setw(8) << setfill('0') << H[i] << endl;

    }

}

// Convert hash values to hexadecimal string

string hashToHexString(const unsigned int\* H) {

    stringstream ss;

    for (int i = 0; i < 5; ++i) {

        ss << hex << setw(8) << setfill('0') << H[i];

    }

    return ss.str();

}

// Main SHA-1 function

string sha1(const string& input) {

    // Initial hash values (first 32 bits of the fractional parts of the square roots of the first 5 primes)

    unsigned int H[5] = {

        0x67452301,  // H0

        0xEFCDAB89,  // H1

        0x98BADCFE,  // H2

        0x10325476,  // H3

        0xC3D2E1F0   // H4

    };

    // Step 1: Pad the message

    string paddedMessage = padMessage(input);

    // Step 2: Process the message in 512-bit chunks

    for (size\_t i = 0; i < paddedMessage.size(); i += 64) {

        processChunk(paddedMessage.substr(i, 64), H);

    }

    // Step 3: Produce the final hash value (big-endian)

    return hashToHexString(H);

}

int main() {

    string message;

    cout << "=== SHA-1 Hashing CLI ===\n";

    cout << "Enter the Message: ";

    getline(cin, message);

    cout << "\n=== Computing SHA-1 Hash ===\n";

    string sha1Hash = sha1(message);

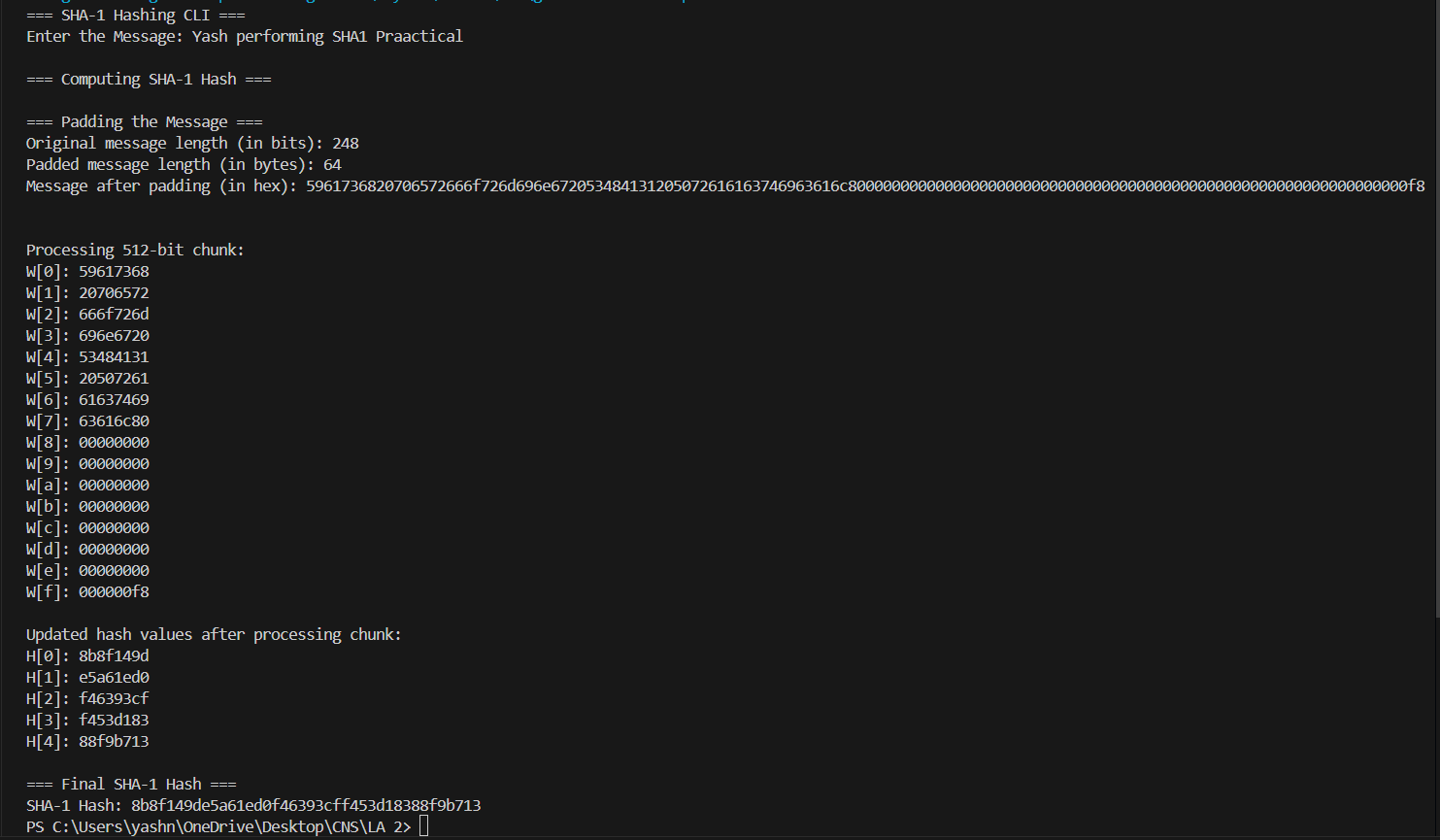
    cout << "\n=== Final SHA-1 Hash ===\n";

    cout << "SHA-1 Hash: " << sha1Hash << endl;

    return 0;

}

Ouput:



**4] Implement the SIGNATURE SCHEME – Digital Signature Standard**

Explanation:

The code demonstrates the **Digital Signature Standard (DSS)** using **SHA-256 with RSA** for signing and verifying a message.

1. **Key Pair Generation**:  
   The code generates a pair of cryptographic keys (private and public) using the **RSA** algorithm with a key size of 2048 bits. The Generate\_RSA\_KeyPair function utilizes SecureRandom to ensure the randomness of the keys, which is essential for security.
2. **Signature Creation**:  
   The Create\_Digital\_Signature function creates the digital signature by:
   * Using the sender's **private key**.
   * Applying the **SHA-256** hashing algorithm to the message, which converts the input data into a fixed-size hash.
   * Encrypting the hash with the private key to generate the **digital signature**.

This signature is then printed in **hexadecimal format** for easy readability.

1. **Signature Verification**:  
   The Verify\_Digital\_Signature function allows the recipient to verify the authenticity of the message. It:
   * Uses the sender's **public key**.
   * Hashes the message again with SHA-256.
   * Decrypts the provided signature with the public key and compares it to the newly computed hash.

If the hashes match, the message's integrity and authenticity are confirmed.

This example shows how **RSA** ensures the integrity of a message, allowing recipients to verify that the message has not been tampered with and that it comes from the legitimate sender.

A **message digest** is the result of applying a cryptographic hash function (such as SHA-256) to a message. The output is a fixed-length value, regardless of the length of the input message. It acts as a "fingerprint" or unique identifier for the original message.

Code:

import java.security.KeyPair;

import java.security.KeyPairGenerator;

import java.security.PrivateKey;

import java.security.PublicKey;

import java.security.SecureRandom;

import java.security.Signature;

import java.util.HexFormat;

import java.util.Scanner;

public class DigitalSignatureCLI {

    private static final String SIGNING\_ALGORITHM = "SHA256withRSA";

    private static final String RSA = "RSA";

    private static KeyPair keyPair;

    private static byte[] lastSignature;

    private static String lastMessage;

    public static byte[] createDigitalSignature(byte[] input, PrivateKey key) throws Exception {

        Signature signature = Signature.getInstance(SIGNING\_ALGORITHM);

        signature.initSign(key);

        signature.update(input);

        return signature.sign();

    }

    public static KeyPair generateRSAKeyPair() throws Exception {

        SecureRandom secureRandom = new SecureRandom();

        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(RSA);

        keyPairGenerator.initialize(2048, secureRandom);

        return keyPairGenerator.generateKeyPair();

    }

    public static boolean verifyDigitalSignature(byte[] input, byte[] signatureToVerify, PublicKey key) throws Exception {

        Signature signature = Signature.getInstance(SIGNING\_ALGORITHM);

        signature.initVerify(key);

        signature.update(input);

        return signature.verify(signatureToVerify);

    }

    public static void main(String[] args) throws Exception {

        Scanner scanner = new Scanner(System.in);

        boolean exit = false;

        System.out.println("---- Digital Signature CLI Application ----\n");

        while (!exit) {

            System.out.println("Choose an option:");

            System.out.println("1. Generate RSA Key Pair");

            System.out.println("2. Sign a Message");

            System.out.println("3. Verify a Signature");

            System.out.println("4. Exit");

            System.out.print("Enter your choice: ");

            int choice = scanner.nextInt();

            scanner.nextLine(); // consume newline

            switch (choice) {

                case 1 -> {

                    System.out.println("\nGenerating RSA key pair (2048-bit)...");

                    keyPair = generateRSAKeyPair();

                    System.out.println("Key pair generated successfully!\n");

                }

                case 2 -> {

                    if (keyPair == null) {

                        System.out.println("Please generate a key pair first.\n");

                        continue;

                    }

                    System.out.print("Enter the message to sign: ");

                    lastMessage = scanner.nextLine();

                    lastSignature = createDigitalSignature(lastMessage.getBytes(), keyPair.getPrivate());

                    System.out.println("\nMessage signed successfully!");

                    System.out.println("Signature (Hex):\n" + HexFormat.of().formatHex(lastSignature) + "\n");

                }

                case 3 -> {

                    if (keyPair == null || lastSignature == null || lastMessage == null) {

                        System.out.println("Please generate a key pair and sign a message first.\n");

                        continue;

                    }

                    System.out.println("Verifying the last signed message...");

                    boolean isVerified = verifyDigitalSignature(lastMessage.getBytes(), lastSignature, keyPair.getPublic());

                    System.out.println("Verification Result: " + (isVerified ? "Signature is valid!" : "Signature is invalid!") + "\n");

                }

                case 4 -> {

                    exit = true;

                    System.out.println("Exiting the application. Goodbye!");

                }

                default -> System.out.println("Invalid choice. Please select a valid option.\n");

            }

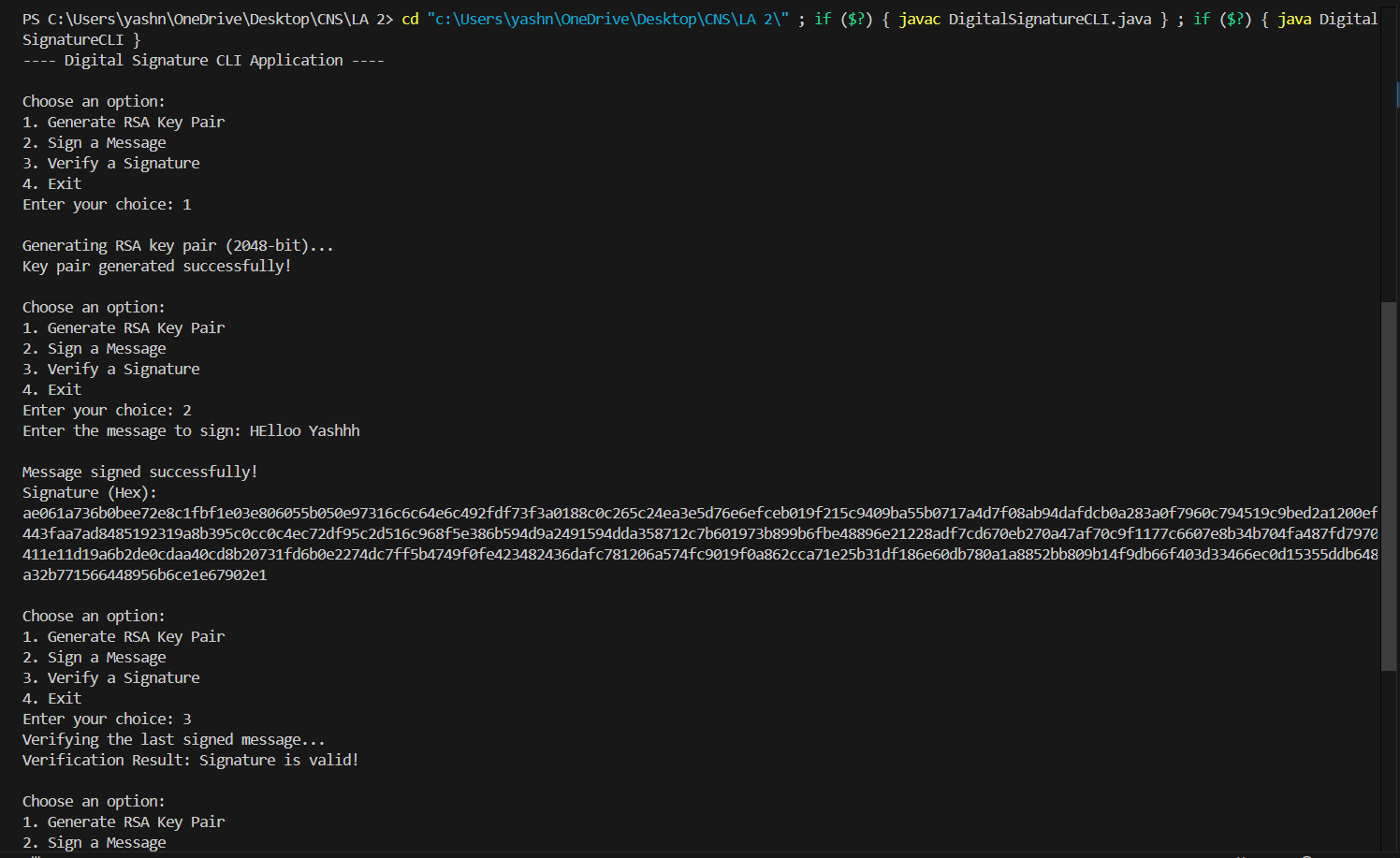
        }

        scanner.close();

    }

}

Ouput:



**5] 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/).

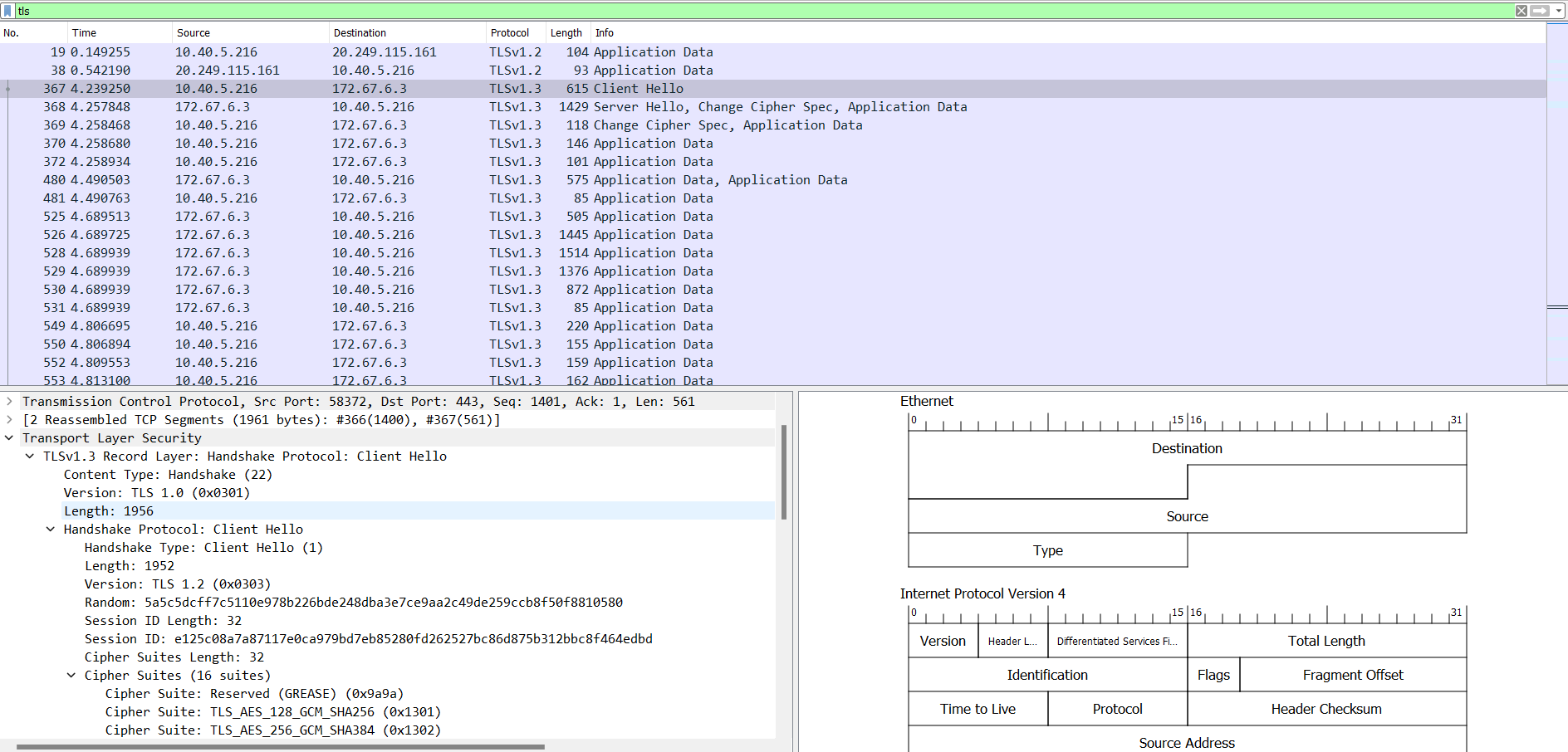
The handshake also handles authentication, which usually consists of the server proving its identity to the client. This is done using public keys. Public keys are encryption keys that use one-way encryption, meaning that anyone with the public key can unscramble the data encrypted with the server's 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.

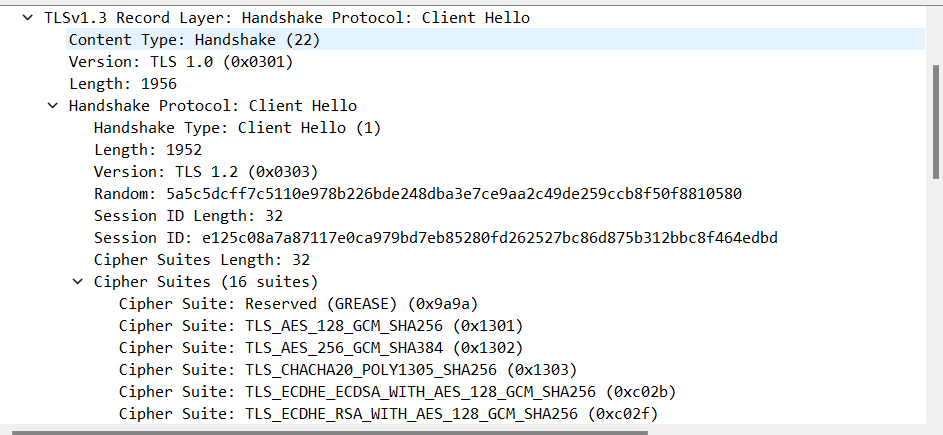
Once data is encrypted and authenticated, it is then signed with a message authentication code (MAC). The recipient can then verify the MAC to ensure the integrity of the data. This is kind of like the tamper-proof foil found on a bottle of aspirin; the consumer knows no one has tampered with their medicine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

TLS Handshake:

**Step 1: Client Hello**

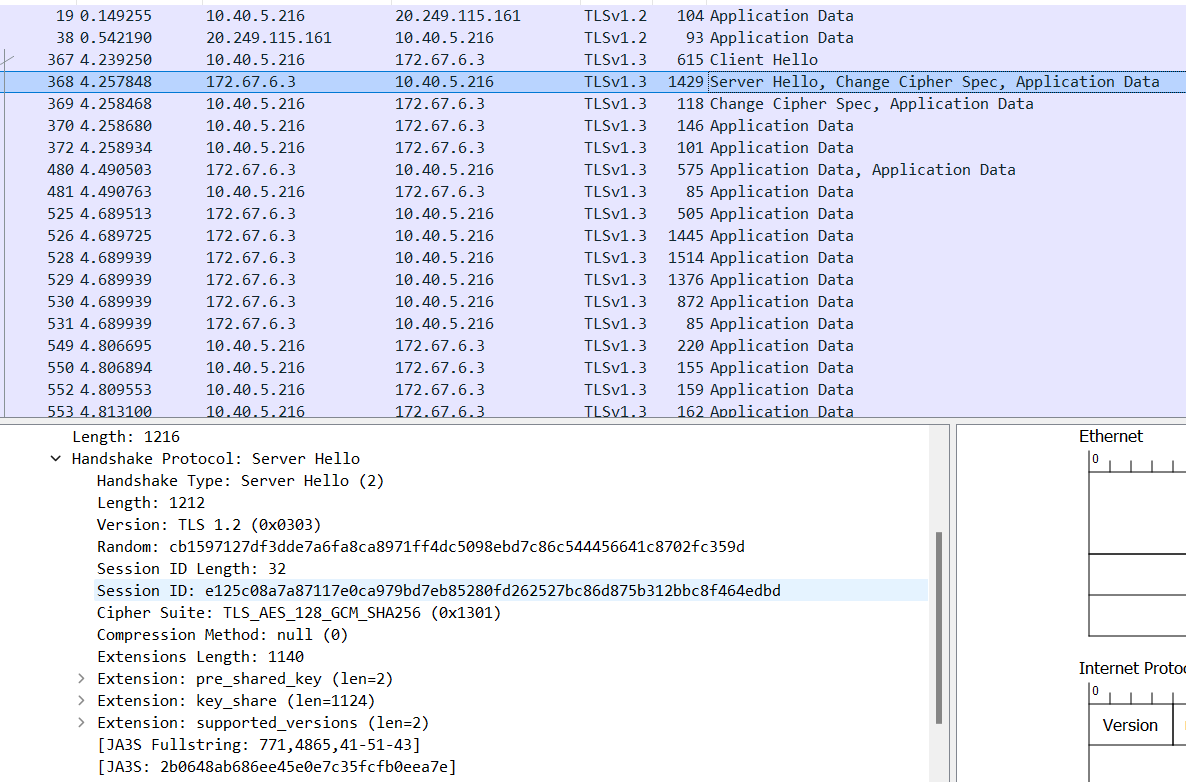
The client begins the communication. The first step is called **client hello**. The client lists**the versions of SSL/TLS** and **cipher suites** it’s able to use.

––––



**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**

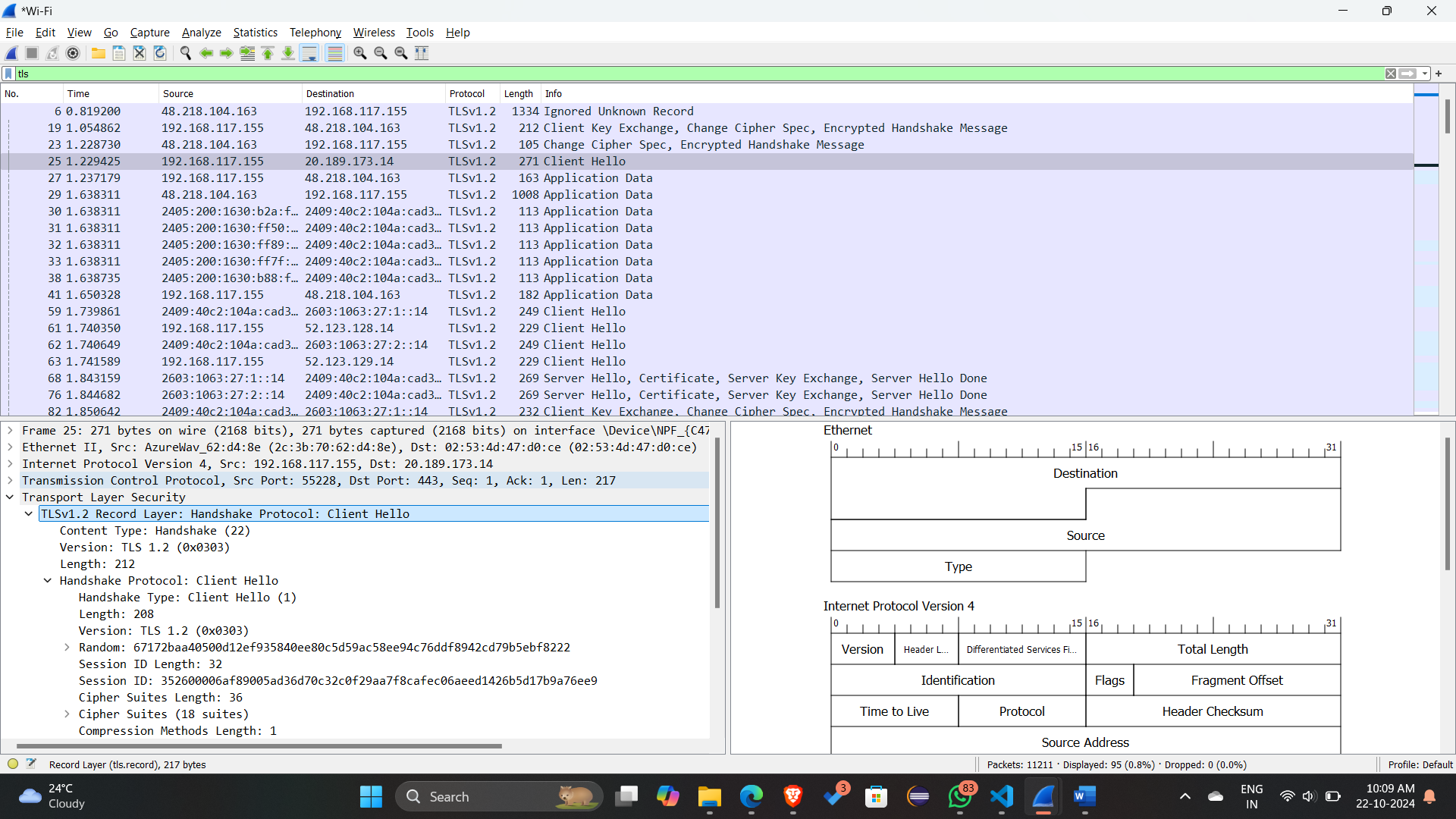
The client and the server send each other an encrypted message saying the key information is correct.

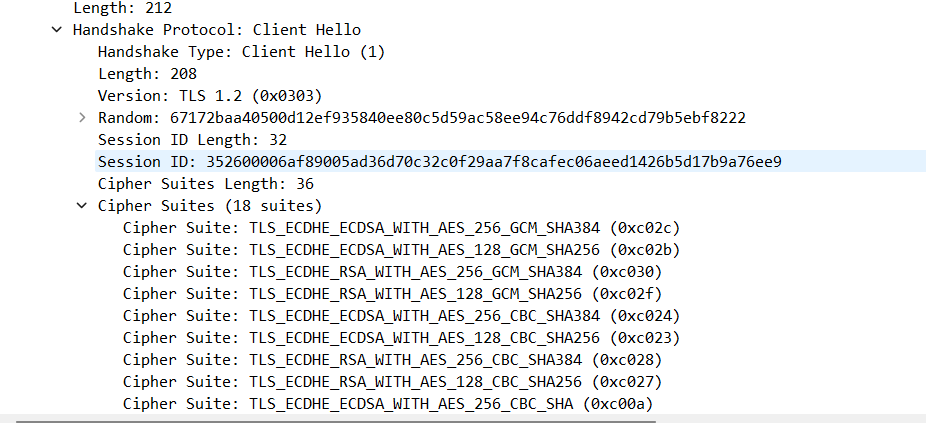
Now the client (web browser) will display a green lock in the address bar. The client and server encrypt HTTP traffic with the session key.

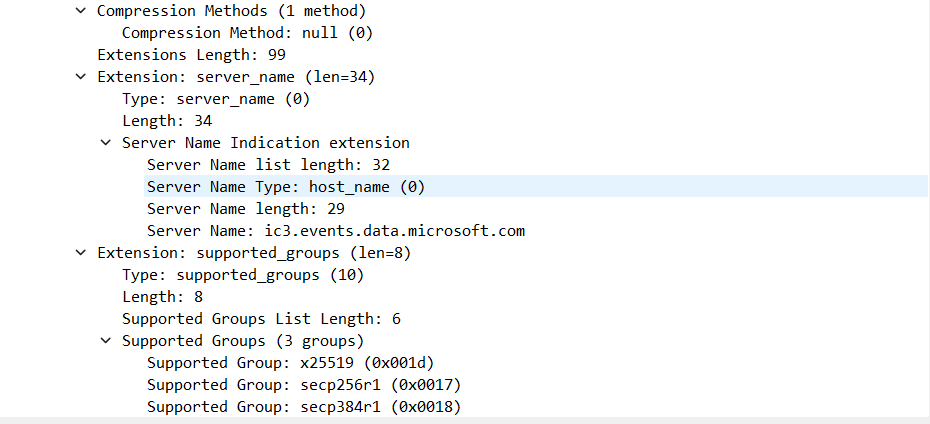
Session key is only valid in one session. If the user closes the client and visits the same server the next day, a new session key will be generated by the client.

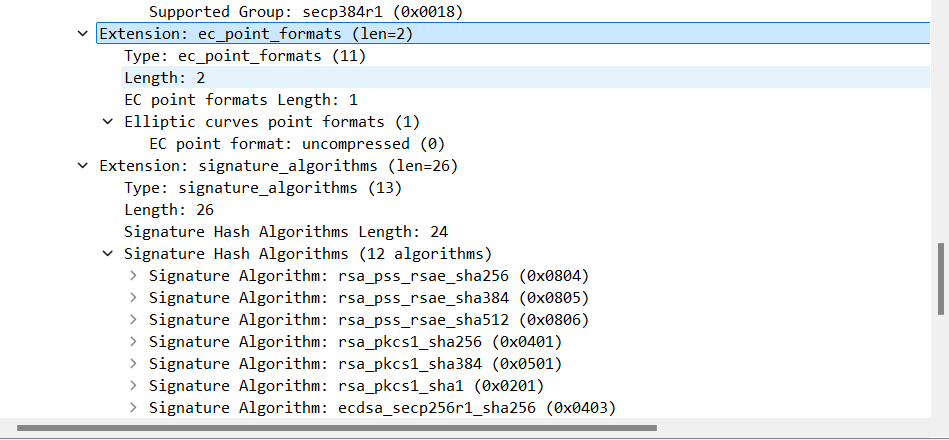
For demonstration I used - <https://youtube.com>

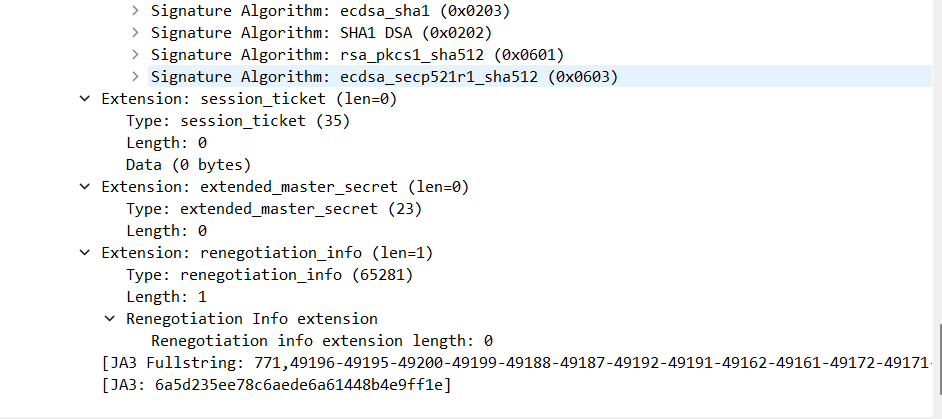
Client Hello:



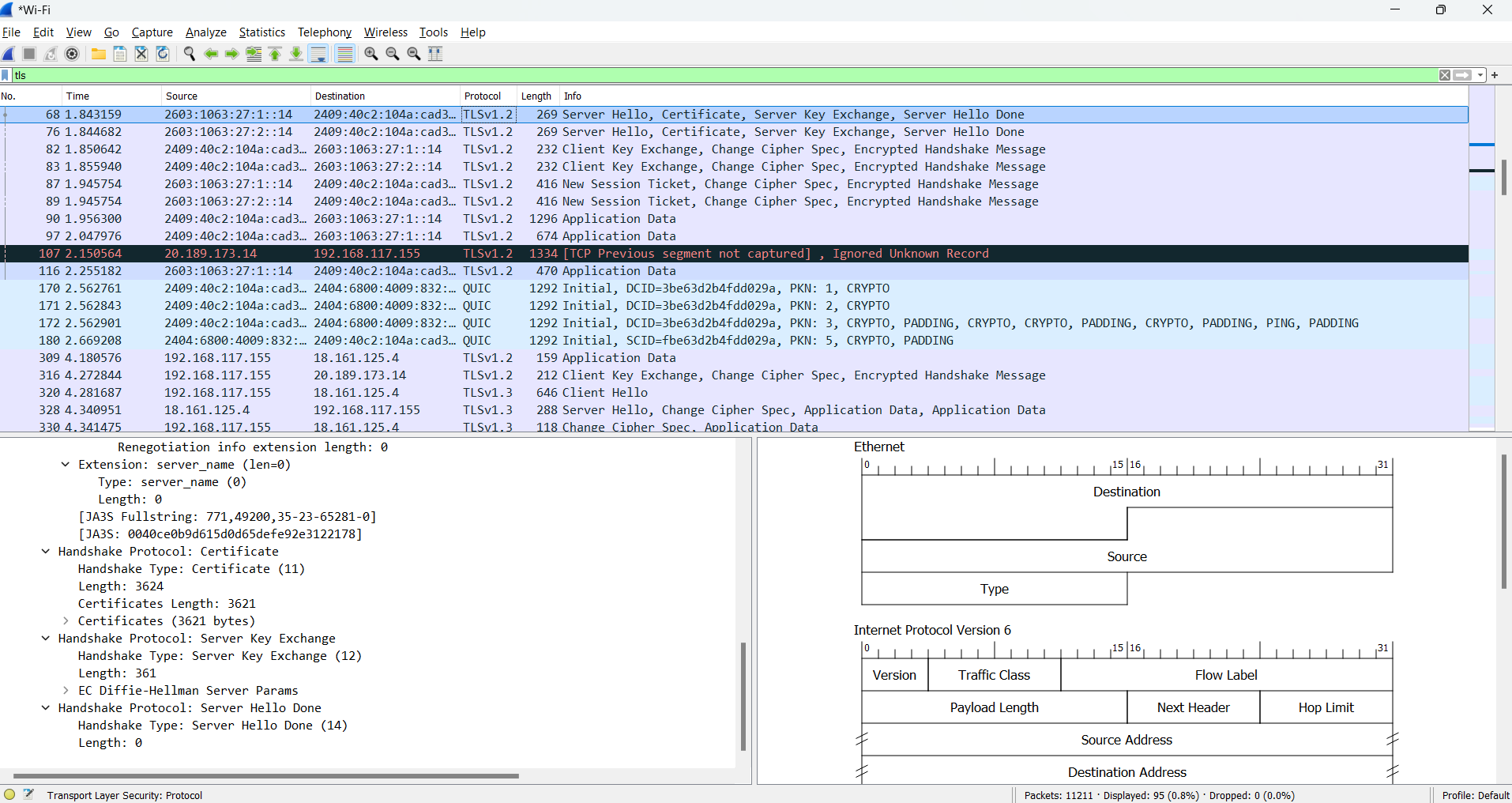


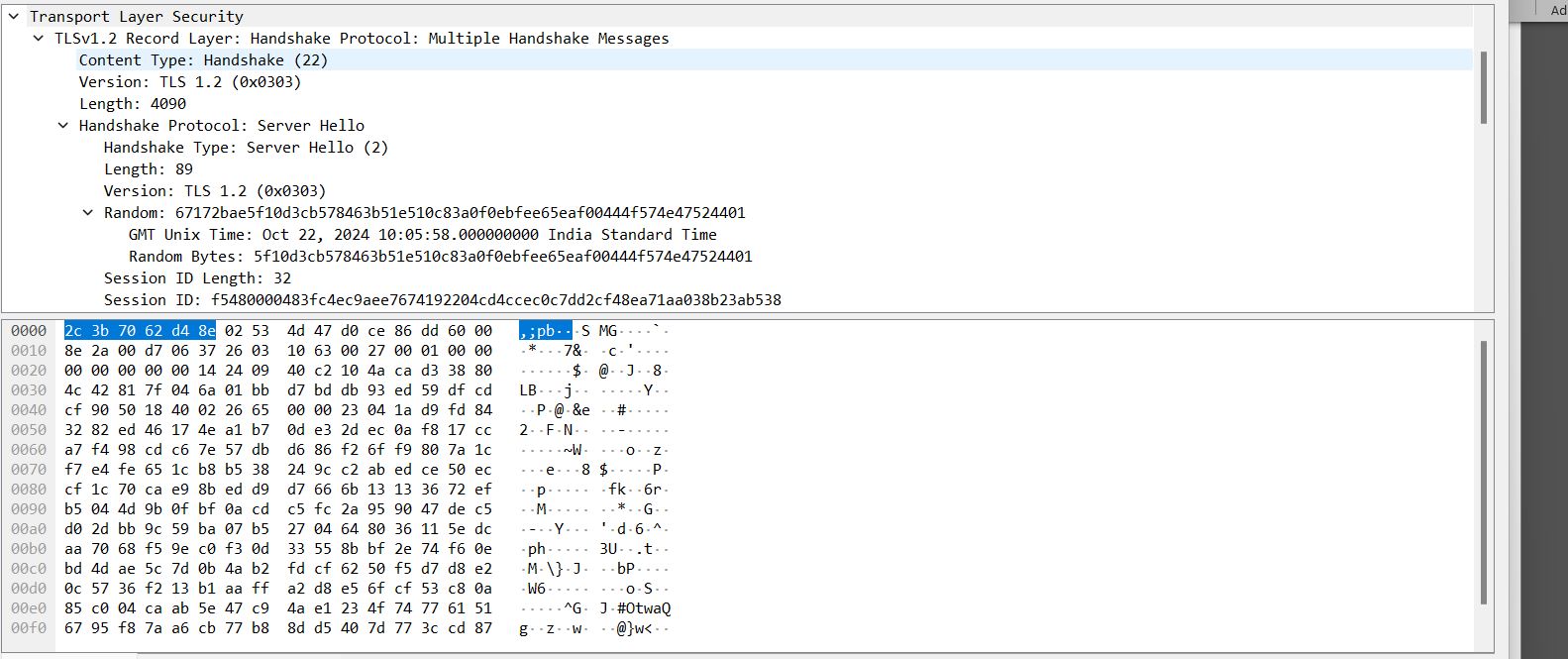




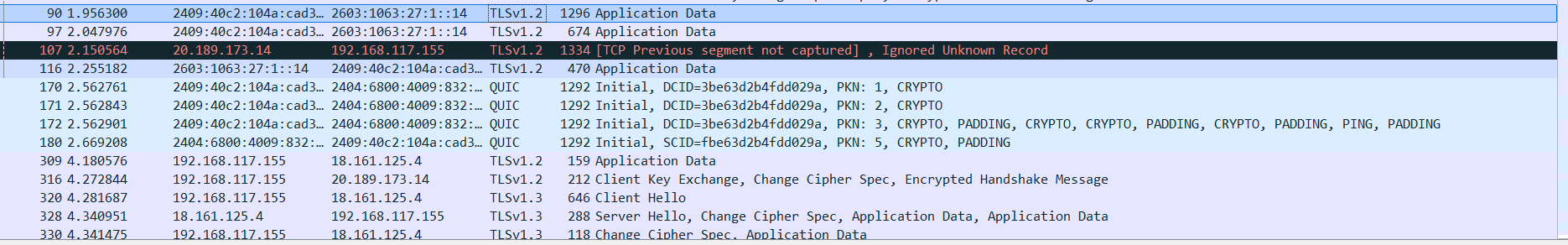


Server Hello:

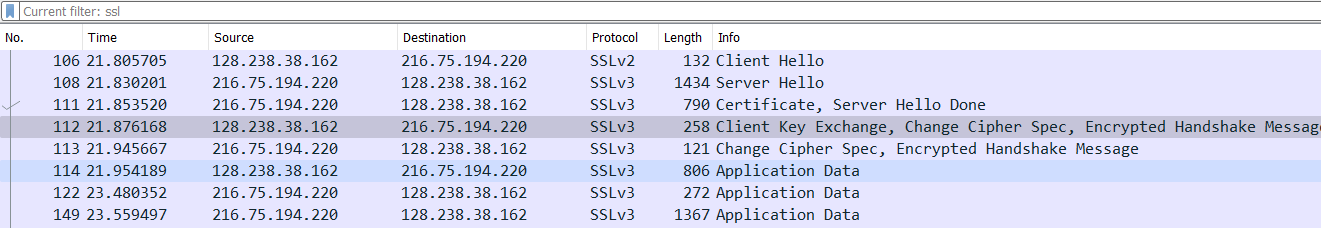


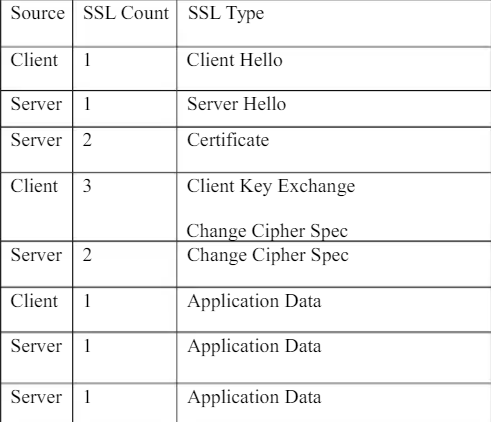


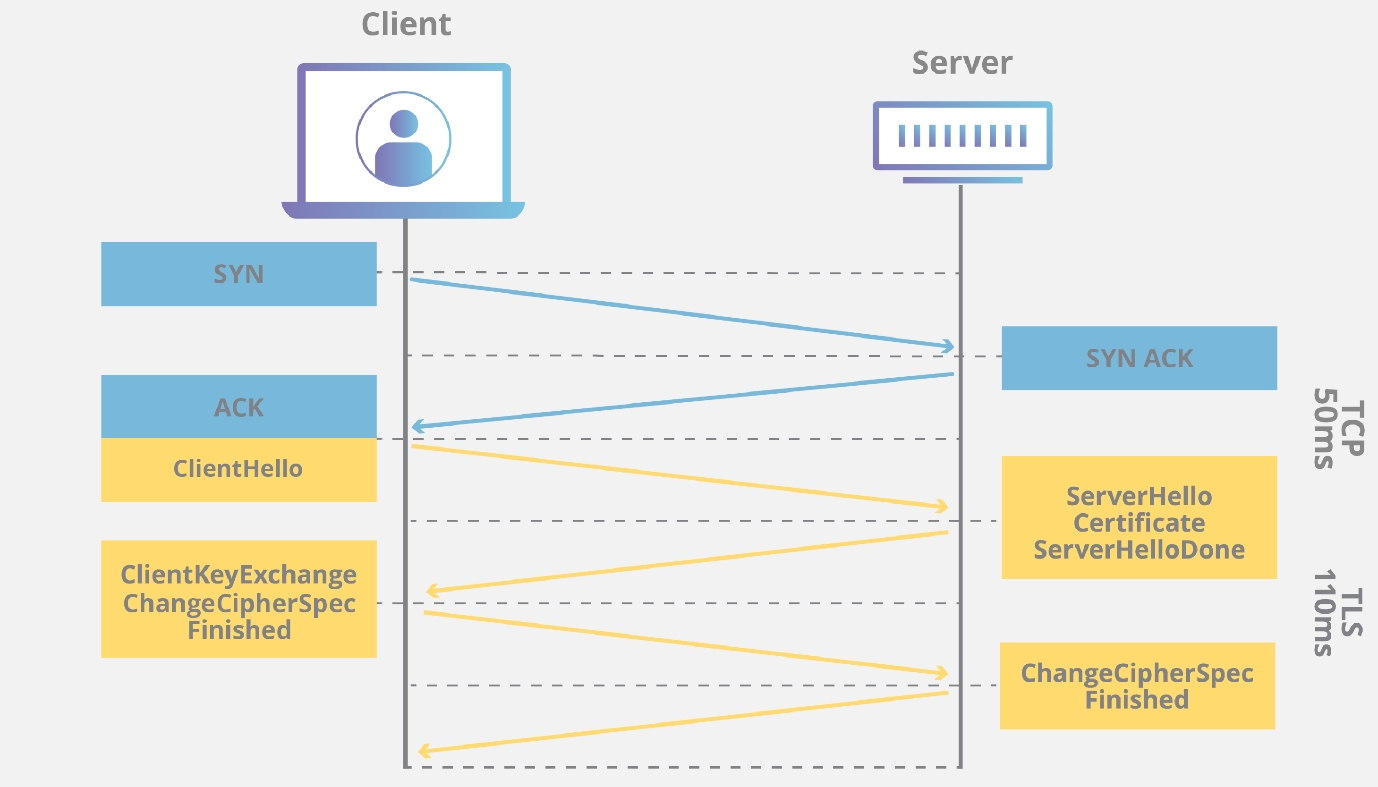
Application Data:



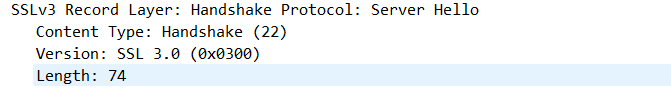
1. For each of the first 8 Ethernet frames, specify the source of the frame (client or server), determine the number of SSL records that are included in the frame, and list the SSL record types that are included in the frame. Draw a timing diagram between client and server, with one arrow for each SSL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



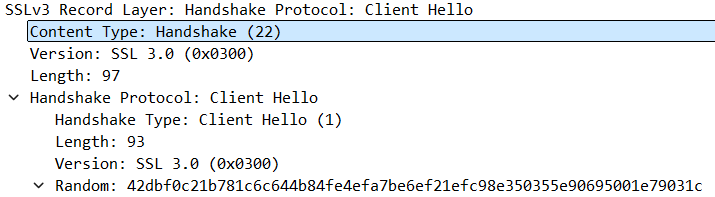




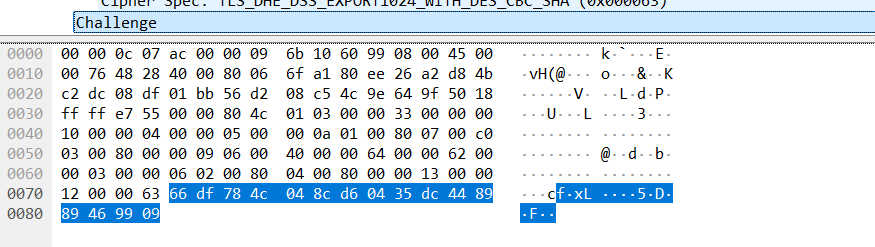
1. Each of the SSL records begins with the same three fields (with possibly different values). One of these fields is “content type” and has length of one byte. List all three fields and their lengths.



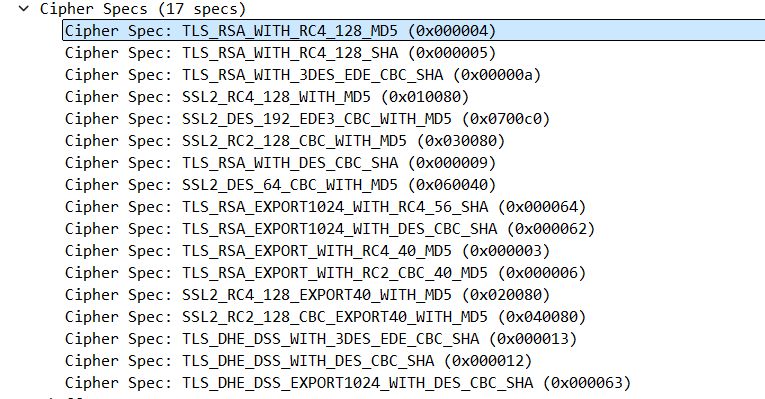
1. Expand the ClientHello record. (If your trace contains multiple ClientHello records, expand the frame that contains the first one.) What is the value of the content type?



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?

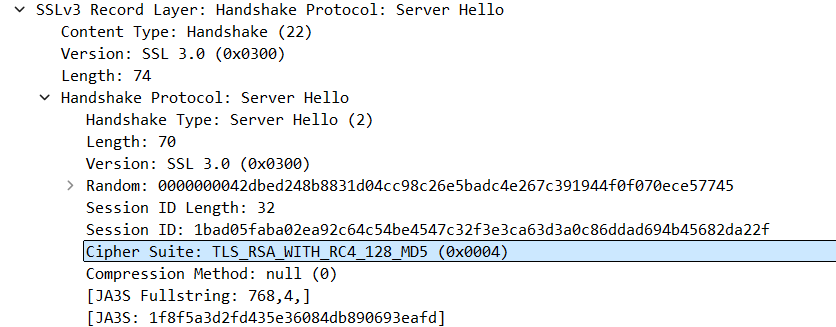


RSA – public key encryption

RC4 – Symmetric key algorithm

MD5 – For hashing

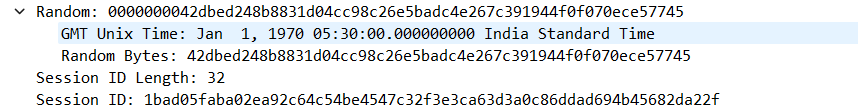
1. Locate the ServerHello SSL record. Does this record specify a chosen cipher suite? What are the algorithms in the chosen cipher suite?



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



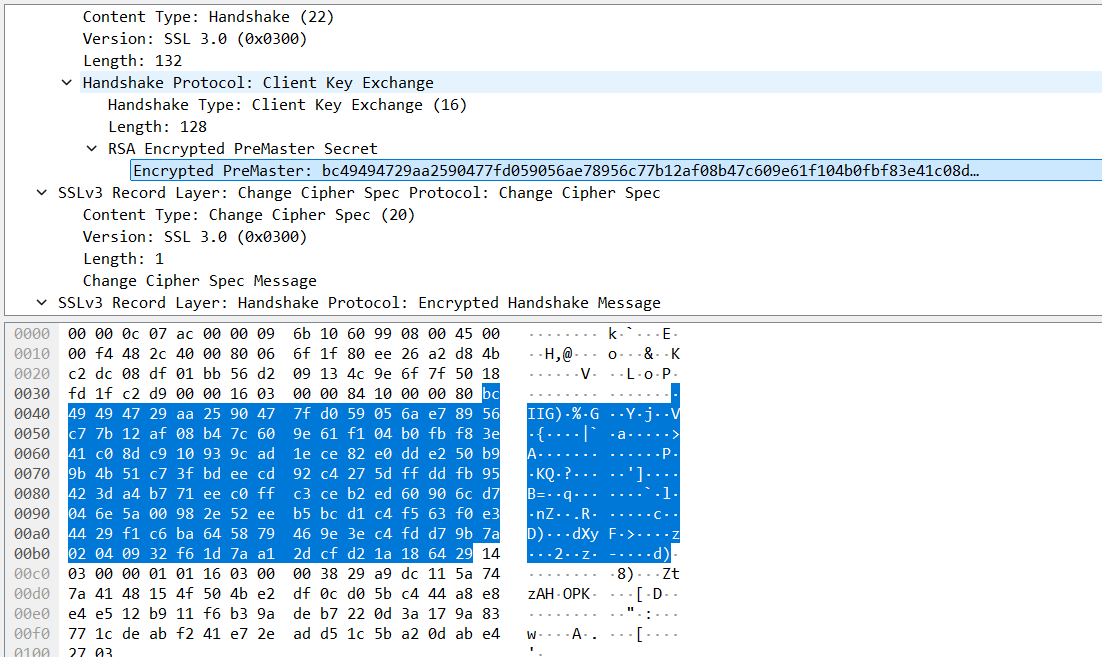
1. Does this record include a session ID? What is the purpose of the session ID?

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

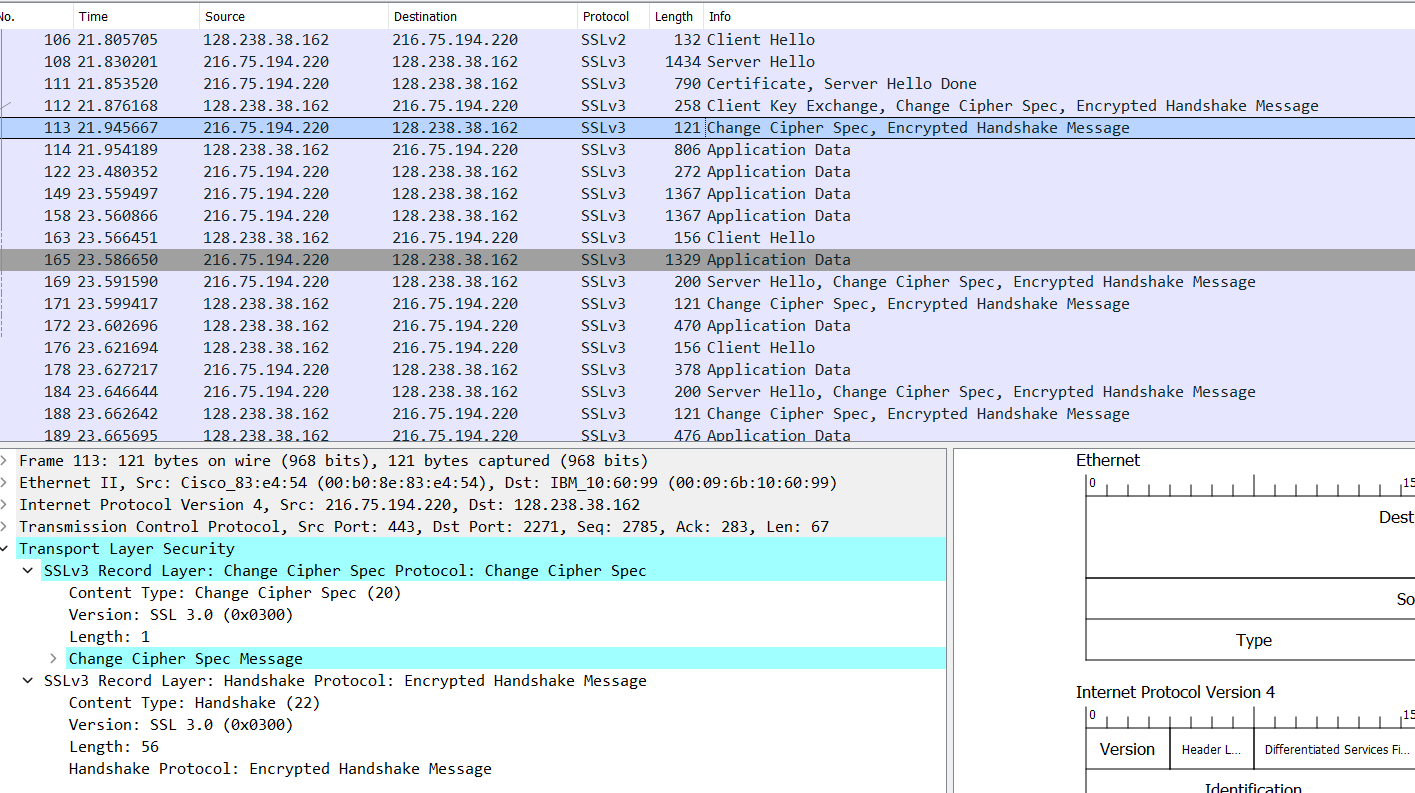
It is used to generate session keys for encryption

Secret is encrypted using public key of server

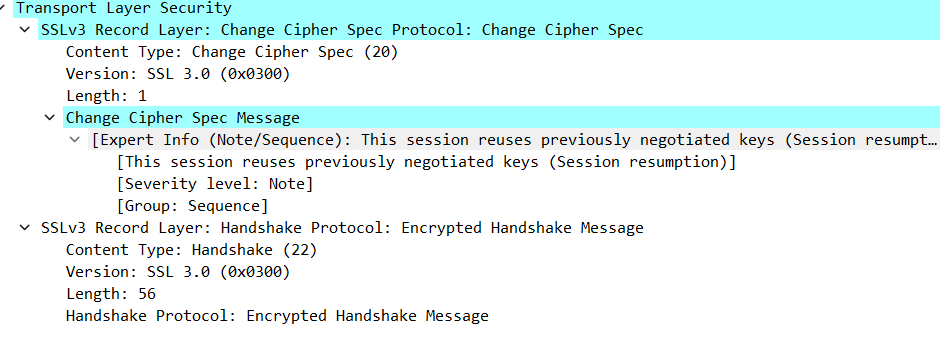
Client gets this key which it extracted from certificate from server

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?



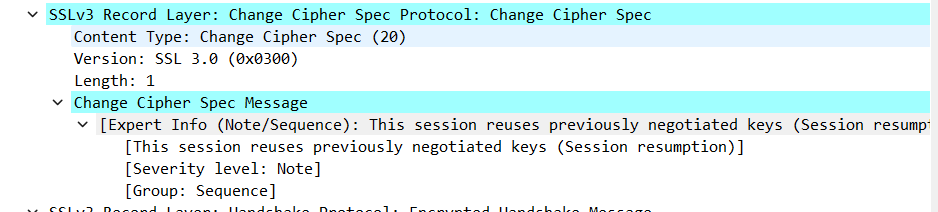
Purpose of change cipher spec record is to indicate that the contents of the following SSL record sent by client will be encrypted



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?

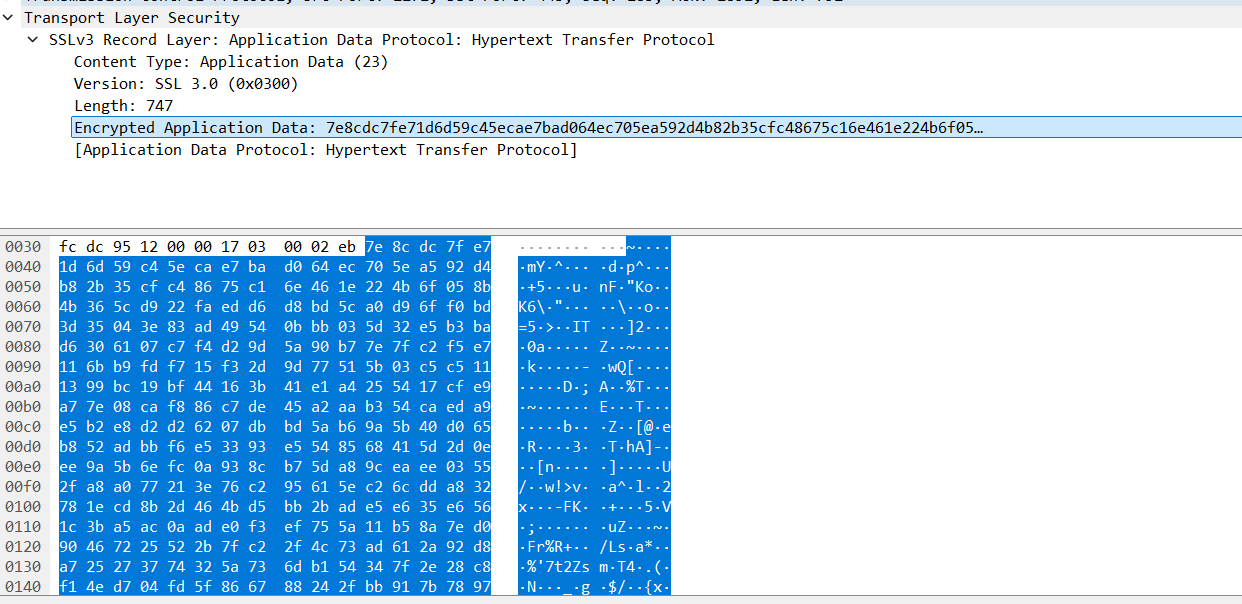


Yes server also send the change cipher record encrypted handshake to the client. It is different from the client’s one as it contains the concatenation of all handshake messages sent from server rather than from client

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

