Source code and detailed setup scripts are available at https://git.trance-0.com/Trance-0/CSE4303H1.

1. Description of experimental setup

   We use the same docker image and settings with environment files as last time.

   For encryption method, I checked some wikis on the internet and found aes seems to be the best practically safe algorithm I can found.

   For the detailed method and distinctions for different variations for aes, I checked the post on stackoverflow for steam cipher, seems that `EVP_aes_256_ctr()` is the optimal choice for us.

   For docker network, I used the same network as last time with the following compose file:

```
services:
server:
  image: debian:bookworm-slim
  container_name: hw1-server
  working_dir: /app
  env_file:
    - .env
  networks:
    net-hw1:
      ipv4_address: ${SERVER_IP}
  volumes:
    - ./bin/server:/usr/local/bin/server:ro
  entrypoint: ["bash", "-lc", "apt-get update && apt-get install
    -y --no-install-recommends libstdc++6 iproute2 libssl-dev;
    exec stdbuf -oL -eL /usr/local/bin/server"]

client:
  image: debian:bookworm-slim
  container_name: hw1-client
  working_dir: /app
  env_file:
    - .env
  networks:
    net-hw1:
      ipv4_address: ${CLIENT_IP}
  depends_on:
    - server
  stdin_open: true
  tty: true
  volumes:
    - ./bin/client:/usr/local/bin/client:ro
  entrypoint: ["bash", "-lc", "apt-get update && apt-get install
    -y --no-install-recommends libstdc++6 iproute2 libssl-dev;
    exec stdbuf -oL -eL /usr/local/bin/client"]
```

```
networks:
  net-hw1:
    name: net-hw1
    driver: bridge
    ipam:
      config:
        - subnet: ${NET_SUBNET}
```

The environment file is as follows:

```
NET_SUBNET=172.30.0.0/24
SERVER_IP=172.30.0.10
CLIENT_IP=172.30.0.11
SERVER_PORT=3030
INITIAL_VECTOR=EA514659DC556DEECFFA5FD061363642
SECRET_KEY=0214
    BA9480AE8D21842B80FF5287418FD0465113070FDFF8263F0F1FCD6CF030
```

2. Server code

   (a) Screenshot or well-formatted copy of code

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

// open ssl
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>

#include <errno.h> // for perror()
#include <unistd.h> // for close()

// load environment variables from .env file
#include <fstream>
#include <cstdlib>

void load_env(const std::string& path) {
    std::ifstream f(path);
    std::string line;
    while (std::getline(f, line)) {
        if (line.empty() || line[0] == '#') continue;
        auto pos = line.find('=');
        if (pos == std::string::npos) continue;

        std::string key = line.substr(0, pos);
        std::string val = line.substr(pos + 1);

        #ifdef _WIN32
                _putenv_s(key.c_str(), val.c_str());
        #else
                setenv(key.c_str(), val.c_str(), 1);
        #endif
    }
}

// hex to byte helper function
int hex_to_bytes_upper(const char* hex, unsigned char* out,
   size_t out_size)
{
    size_t i = 0;
    while (hex[0] && hex[1]) {
        if (i >= out_size) return -1;
```

```
            unsigned char h = hex[0];
            unsigned char l = hex[1];
            int hi = (h <= '9') ? (h - '0') : (h - 'A' + 10);
            int lo = (l <= '9') ? (l - '0') : (l - 'A' + 10);
            // minimal sanity check
            if (hi < 0 || hi > 15 || lo < 0 || lo > 15)
                return -1;
            out[i++] = (unsigned char)((hi << 4) | lo);
            hex += 2;
        }
        return (int)i;
}

// Define the decryption function
int stream_decrypt(unsigned char *ciphertext, int
    ciphertext_len, unsigned char *key, unsigned char *iv,
    unsigned char *plaintext) {
    /* Declare cipher context */
    EVP_CIPHER_CTX *ctx;
    int len, plaintext_len;

    /* Create and initialise the context */
    ctx = EVP_CIPHER_CTX_new();
    if (!ctx) {
        ERR_print_errors_fp(stderr);
    }

    /* Initialise the decryption operation. */
    if (EVP_DecryptInit_ex(ctx, EVP_aes_256_ctr(), NULL, key,
        iv) != 1){
        ERR_print_errors_fp(stderr);
    }

    /* Provide the message to be decrypted, and obtain the
        plaintext output. EVP_DecryptUpdate can be called
        multiple times if necessary */
    if (EVP_DecryptUpdate(ctx, plaintext, &len, ciphertext,
        ciphertext_len) != 1) {
        ERR_print_errors_fp(stderr);
    }

    /* Finalize the decryption. Further plaintext bytes may be
        written at this stage. */
    if (EVP_DecryptFinal_ex(ctx, plaintext + len, &
        plaintext_len) != 1) {
```

```
        ERR_print_errors_fp(stderr);
    }

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    return len + plaintext_len;
}

int main(void){
    printf("Server starting...\n");
    load_env(".env");

    // Declare variables
    const char *server_ip = std::getenv("SERVER_IP");
    const int server_port = std::atoi(std::getenv("SERVER_PORT"
        ));
    char client_message[2048];
    char server_message[2048];
    const char * custom_message="Server: Hello from server, 
        message recieved!\n";

    // decryption parameters
    const char *inital_vector = std::getenv("INITIAL_VECTOR");
    const char *secret_key = std::getenv("SECRET_KEY");
    unsigned char key_bytes[32], iv_bytes[16];

    int key_len = hex_to_bytes_upper(secret_key, key_bytes, 
        sizeof(key_bytes));
    int iv_len  = hex_to_bytes_upper(inital_vector, iv_bytes, 
        sizeof(iv_bytes));

    if (key_len != 32 || iv_len != 16) {
        fprintf(stderr, "Invalid key/IV size for AES-256\n");
        return 1;
    }

    // debug
    printf("Server starting at IP: %s, Port: %d\n", server_ip, 
        server_port);

    // Create socket
    const int server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (server_socket == -1) {
        perror("Failed to create socket");
```

```
    return 1;
}

// Bind to the set port and IP
struct sockaddr_in server_addr;
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(server_port);
inet_pton(AF_INET, server_ip, &server_addr.sin_addr);

if (bind(server_socket, (struct sockaddr*)&server_addr,
    sizeof(server_addr)) == -1) {
    perror("Failed to bind socket");
    close(server_socket);
    return 1;
}
printf("Done with binding with IP: %s, Port: %d\n",
    server_ip, server_port);

// Listen for clients:
const char *client_ip = std::getenv("CLIENT_IP");
if (listen(server_socket, 1) == -1) {
    perror("Failed to listen on socket");
    close(server_socket);
    return 1;
}
printf("Listening for incoming connections...\n");

// Accept an incoming connection
struct sockaddr_in client_addr;
socklen_t client_addr_len = sizeof(client_addr);
int client_socket = accept(server_socket, (struct sockaddr
    *)&client_addr, &client_addr_len);
if (client_socket == -1) {
    perror("Failed to accept connection");
    close(server_socket);
    return 1;
}
printf("Client connected at IP: %s \n", client_ip);

// clean exising buffer
memset(client_message, 0, sizeof(client_message));

// Receive client's message
while (1) {
    // clean exising buffer
```

```
        memset(client_message, 0, sizeof(client_message));

        ssize_t n = recv(client_socket, client_message, sizeof(
            client_message), 0);
        if (n == 0) break;
        if (n < 0) { perror("recv"); break; }

        // decrypt client message
        unsigned char plaintext[2048];
        int plaintext_len = stream_decrypt((unsigned char*)
            client_message, (int)n, key_bytes, iv_bytes,
            plaintext);
        if (plaintext_len < 0) {
            fprintf(stderr, "decrypt failed\n");
            break;
        }

        if (plaintext_len >= (int)sizeof(plaintext))
            plaintext_len = (int)sizeof(plaintext) - 1;
        plaintext[plaintext_len] = '\0';

        if (strcmp((char*)plaintext, "\\exit\n") == 0) break;

        printf("Msg from client: %s", plaintext);

        // Respond to client
        // prepare server message
        memcpy(server_message, custom_message, strlen(
            custom_message));

        size_t reply_len = strlen(server_message);
        if (send(client_socket, server_message, reply_len, 0)
            == -1) {
            perror("Send failed");
            break;
        }

        printf("Response sent to client: %s\n", server_message)
            ;
    }
    // Close the socket
    close(client_socket);
    close(server_socket);

    return 0;
```

}

(b) Quick overview of what the code does in your own words.

Compared with previous assignment, we added a new function to decrypt the message from client, and modify the code and allow the server to process multiple client request.

In addition to environment loading function, we build another hex to byes function to convert hex string to bytes used for initial vector and secret key.

In `stream_decrypt` function, first we initialize the context for aes, then we decrypt the message from client, and finally we return the decrypted message. For each step, we capture possible errors and output the error message using `ERR_print_errors_fp` function.

After decrypting the message from client, we print the message on the server side and told the client we received the encrypted message.

Here is the screenshot for the server output:

Figure 1: Server output

3. Client code

   (a) Screenshot or well-formatted copy of code

```
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>

// open ssl
#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>

#include <errno.h> // error handling
#include <unistd.h> // for close()

// load environment variables from .env file
#include <fstream>
#include <cstdlib>

void load_env(const std::string& path) {
    std::ifstream f(path);
    std::string line;
    while (std::getline(f, line)) {
        if (line.empty() || line[0] == '#') continue;
        auto pos = line.find('=');
        if (pos == std::string::npos) continue;

        std::string key = line.substr(0, pos);
        std::string val = line.substr(pos + 1);

        #ifdef _WIN32
                _putenv_s(key.c_str(), val.c_str());
        #else
                setenv(key.c_str(), val.c_str(), 1);
        #endif
    }
}

// hex to byte helper function
int hex_to_bytes_upper(const char* hex, unsigned char* out,
   size_t out_size)
{
    size_t i = 0;
    while (hex[0] && hex[1]) {
        if (i >= out_size) return -1;
```

```
            unsigned char h = hex[0];
            unsigned char l = hex[1];
            int hi = (h <= '9') ? (h - '0') : (h - 'A' + 10);
            int lo = (l <= '9') ? (l - '0') : (l - 'A' + 10);
            // minimal sanity check
            if (hi < 0 || hi > 15 || lo < 0 || lo > 15)
                return -1;
            out[i++] = (unsigned char)((hi << 4) | lo);
            hex += 2;
        }
        return (int)i;
}

// steam cipher encryption function
int stream_encrypt(unsigned char *plaintext, int plaintext_len,
    unsigned char *key, unsigned char *iv, unsigned char *
   ciphertext) {
    /* Declare cipher context */
    EVP_CIPHER_CTX *ctx;

    int len, ciphertext_len;

    /* Create and initialise the context */
    ctx = EVP_CIPHER_CTX_new();
    if (!ctx) {
        ERR_print_errors_fp(stderr);
    }

    /* Initialise the encryption operation. */
    // choice for aes_256 ref: https://stackoverflow.com/
       questions/1220751/how-to-choose-an-aes-encryption-mode-
       cbc-ecb-ctr-ocb-cfb
    if (EVP_EncryptInit_ex(ctx, EVP_aes_256_ctr(), NULL, key,
       iv) != 1){
        ERR_print_errors_fp(stderr);
    }

    /* Provide the message to be encrypted, and obtain the
       encrypted output. EVP_EncryptUpdate can be called
       multiple times if necessary */
    if (EVP_EncryptUpdate(ctx, ciphertext, &len, plaintext,
       plaintext_len) != 1) {
        ERR_print_errors_fp(stderr);
    }
```

```
    /* Finalize the encryption. Further cipher text bytes may
       be written at this stage. */
    if (EVP_EncryptFinal_ex(ctx, ciphertext + len, &
       ciphertext_len) != 1) {
        ERR_print_errors_fp(stderr);
    }

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    return len + ciphertext_len;
}

int main(void){
    load_env(".env");
    // Declare variables
    const char *server_ip = std::getenv("SERVER_IP");
    const int server_port = std::atoi(std::getenv("SERVER_PORT"
       ));
    printf("Connecting to server %s:%d\n", server_ip,
       server_port);

    char client_message[1024];
    char server_message[1024];
    const char * custom_message="Zheyuan Wu: ";

    // encryption parameters
    const char *inital_vector = std::getenv("INITIAL_VECTOR");
    const char *secret_key = std::getenv("SECRET_KEY");
    unsigned char key_bytes[32], iv_bytes[16];

    int key_len = hex_to_bytes_upper(secret_key, key_bytes,
       sizeof(key_bytes));
    int iv_len  = hex_to_bytes_upper(inital_vector, iv_bytes,
       sizeof(iv_bytes));

    if (key_len != 32 || iv_len != 16) {
        fprintf(stderr, "Invalid key/IV size for AES-256\n");
        return 1;
    }

    // Create socket:
    int client_socket = socket(AF_INET, SOCK_STREAM, 0);
    if (client_socket == -1) {
        perror("Failed to create socket");
```

```
        return 1;
    }else{
        printf("Socket created successfully\n");
    }

    // Send connection request to server, be sure to set por
       tand IP the same as server-side
    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(server_port);
    inet_pton(AF_INET, server_ip, &server_addr.sin_addr);

    if (connect(client_socket, (struct sockaddr*)&server_addr,
       sizeof(server_addr)) == -1) {
        perror("Failed to connect to server");
        close(client_socket);
        return 1;
    }else{
        printf("Connected to server successfully\n");
    }

    // Get input from the user:
    printf("Enter message sent to the server (type \\quit to
       exit): ");
    // clean the buffer
    memset(client_message, 0, sizeof(client_message));
    if (fgets(client_message, sizeof(client_message), stdin) ==
        NULL) {
        // EOF or error reading from stdin, exit the loop
        perror("Error reading from stdin");
        return 1;
    }

    while (strcmp(client_message, "\\quit\n") != 0) {

        // Send the message to server:
        // add my name in the front
        char buffer[2048];
        std::snprintf(buffer, sizeof(buffer), "%s%s",
            custom_message, client_message);

        unsigned char ciphertext[2048];

        int plaintext_len = (int)strlen(buffer);
        int ciphertext_len = stream_encrypt((unsigned char*)
```

```
            buffer, plaintext_len, key_bytes, iv_bytes,
                ciphertext);
            if (ciphertext_len <= 0) {
                printf("encrypt failed or produced empty
                    ciphertext_len=%d\n", ciphertext_len);
                break;
            }

            ssize_t sent = send(client_socket, ciphertext, (size_t)
                ciphertext_len, 0);
            if (sent <= 0) { perror("No message sent to server");
                break; }

            // Receive the server's response:
            // add terminator for string
            ssize_t recvd = recv(client_socket, server_message,
                sizeof(server_message) - 1, 0);
            if (recvd <= 0) { perror("No message received from
                server"); break; }
            server_message[recvd] = '\0';

            printf("Server's response: %s\n", server_message);

            printf("Enter message sent to the server (type \\quit
                to exit): ");

            // clean the buffer
            memset(client_message, 0, sizeof(client_message));
            memset(server_message, 0, sizeof(server_message));

            if (fgets(client_message, sizeof(client_message), stdin
                ) == NULL) {
                // EOF or error reading from stdin, exit the loop
                perror("Error reading from stdin");
                break;
            }
        }

        // Close the socket
        close(client_socket);

        return 0;
    }
```

(b) Quick overview of what the code does in your own words.

Same as the server code, we add hex to byte converting function to decrypt the key and initial vector.

The `stream_encrypt` function initialize the context, the encryption operation, and create cipher text from plain text. After that, we send the cipher text to the server and receive the server's response. Finally, we free up the resources, returning the cipher text length to user.

In main function, we add additional encryption for the client message and send it to the server. We add our name to the front as before.

Here is the screenshot for the client output:

Figure 2: Client output

4. Extra credits

   We use wireshark to capture the packets between the server and client, and we found the following packets:

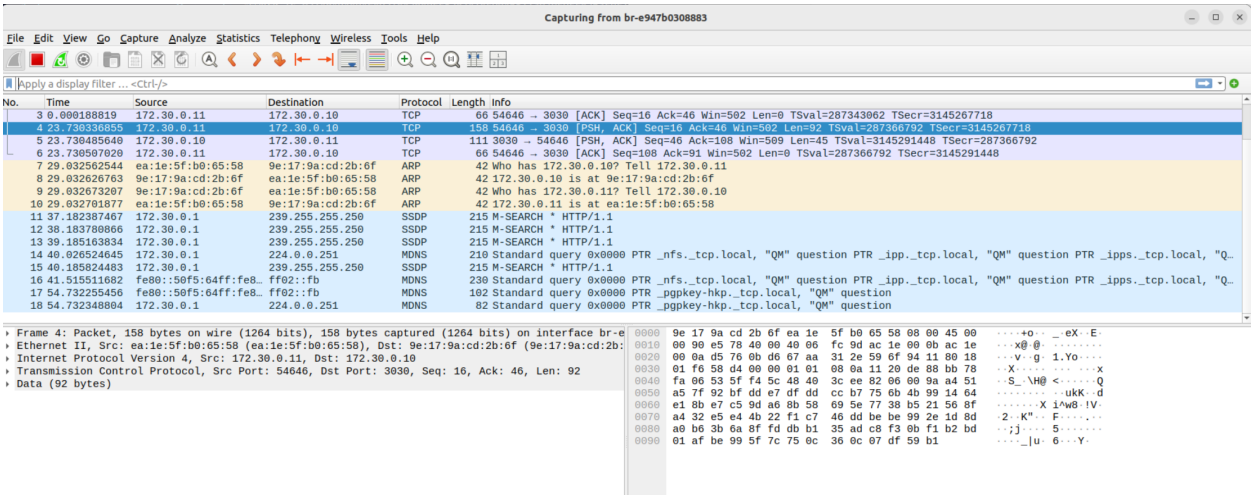

Figure 3: Wireshark capture for client

This packets seems random but immediately after that, server respond with plain text indicating that the decryption succeeded.

The server response is expected as follows shows:

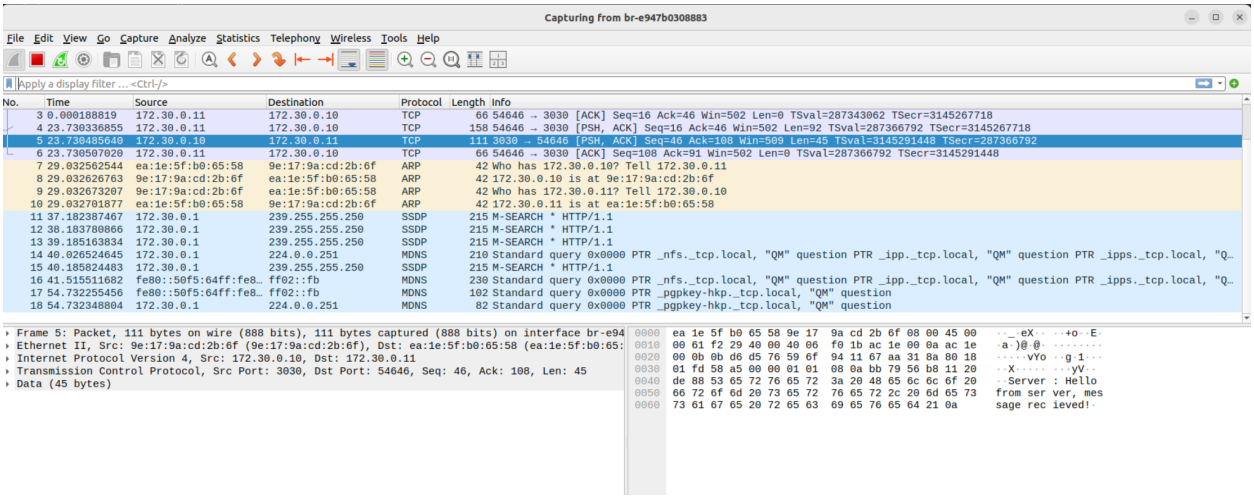

Figure 4: Wireshark capture for server