CSE 433S HW 1 - Comm with Stream Cipher

Zihan Chen

September 2024

Contents

1 Implementation

I have desgined a Diffie-Hellman function to exchange AES key and IV to make sure our key is secure.

However, some unrecognized problem happened, and after 4 hours of debugging, I still cannot find the problem. So I failed to implement the Diffie-Hellman function.

I don't know why there is a block error in AES decryption. I don't want to waste too much time on this problem since I have already spent 4 hours on it and I have other homework to do.

I think this problem will happended only when my AES is too long and more than my buffer but this is impossible.

```
Msg from client: p
20E084347EEF0000:error:1C80006B:Provider routines:
    ossl_cipher_generic_block_final:wrong final block
    length:../providers/implementations/ciphers/
    ciphercommon.c:429:
Aborted
```

I find AES is a block cipher, so I change to use stream cipher, such as RC4 and chacha20. rc4 is not secure, so I choose chacha20.

1.1 OTP

I have desgined a OTP encryption function to encrypt the message. I use rand() function to generate the OTP key. And the OTP has the same length with the message. And OTP has the same encryption and decryption function as follows.

```
    // Init OTP as length of the client_message
    char otp_key[strlen(client_message)];
    for (int i = 0; i < strlen(client_message); i++) {
        otp_key[i] = rand() % 256;
    }

void otp_encrypt(char *plaintext, char *key, char *
    ciphertext) {
    for (int i = 0; i < strlen(plaintext); i++) {
        ciphertext[i] = plaintext[i] ^ key[i];
    }
}
```

1.2 AES

I have designed a AES stream cipher encryption and decryption function as follows.

Figure 1: Caption

I choose AES-256-CBC as my encryption method. This is because AES-256 is a secure encryption method and CBC is the most secure mode of operation in my understanding. I use openssl library rand() function to generate the key and iv.

```
// Declare key and iv
unsigned char key[AES_KEY_LENGTH];
unsigned char iv[AES_BLOCK_SIZE];

// Initialize key and iv
if(RAND_bytes(key, AES_KEY_LENGTH) != 1) {
    printf("Error in generating key\n");
    return;
}

if(RAND_bytes(iv, AES_BLOCK_SIZE) != 1) {
    printf("Error in generating iv\n");
    return;
}
```

1.3 RC4

I have designed a RC4 stream cipher encryption and decryption function. But it can't run successfully because openssl library thinks RC4 is not secure.

```
// Declare key
unsigned char key[RC4_KEY_LENGTH];

// Generate key
if (!RAND_bytes(key, RC4_KEY_LENGTH)) {
    handleErrors();
}
```

1.4 chacha20

I have designed a chacha20 stream cipher encryption and decryption function. But the decryption function is not working correctly. I don't know why.

Figure 2: Caption

Figure 3: Caption

Figure 4: Caption

Figure 5: Caption

After using base64 to encode the message, the chacha20 decryption function is still not working correctly. I don't know why.

2 Result_old

The server and client are able to communicate with each other.

The client send a message within two types of encryption, AES and OTP.

The server will decrypt AES message and OTP message, and send a "Test Over" message to the client.

Here it is very interesting that the AES encrypted message is not the same in the client and server. Why this happended? It is very strange. I think this may related to the padding of the message.

Also, the OTP encrypted message is not the same in the client and server. Interesting.

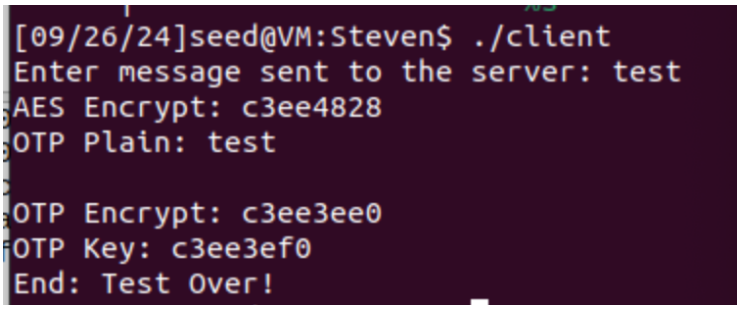

Figure 6: Client

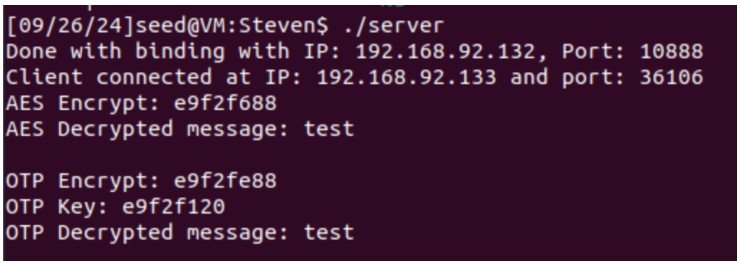

Figure 7: Server

3 Result

4 appendix

4.1 Server.c

```
    #include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <errno.h>
#include <netinet/in.h>
#include <unistd.h>

#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <openssl/rand.h>
#include <openssl/aes.h>
#include <openssl/dh.h>
#include <openssl/bio.h>
```

```
#include <openssl/engine.h>

#define AES_KEY_LENGTH 32
#define AES_BLOCK_SIZE 16

#define RC4_KEY_LENGTH 16
#define CHACHA_KEY_LENGTH 32
#define CHACHA_IV_LENGTH 12

// Handle errors
void handleErrors(void)
{
    ERR_print_errors_fp(stderr);
    abort();
}

// Base64 decode
int base64_decode(const unsigned char *input, int length,
    unsigned char *output) {
    return EVP_DecodeBlock(output, input, length);
}

// Base64 encode
int base64_encode(const unsigned char *input, int length,
    unsigned char *output) {
    return EVP_EncodeBlock(output, input, length);
}

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 initialize the context */
   ctx = EVP_CIPHER_CTX_new();
   if(!ctx) {
        handleErrors();
    }

   /* Initialize the decryption operation. */
   if(EVP_DecryptInit_ex(ctx, EVP_chacha20(), NULL, key,
      iv) != 1) {
```

```
        handleErrors();
    }

    /* 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, &plaintext_len,
       ciphertext, ciphertext_len) != 1) {
        handleErrors();
    }


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

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);

    return plaintext_len;
}

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 = 0;

    /* Create and initialize the context */
    ctx = EVP_CIPHER_CTX_new();
     if(!ctx) {
        handleErrors();
     }

    /* Initialize the encryption operation. */
    if(EVP_EncryptInit_ex(ctx, EVP_chacha20(), NULL, key,
       iv) != 1) {
        handleErrors();
      }
```

```
    /* 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, &ciphertext_len
        , plaintext, plaintext_len) != 1) {
         handleErrors();
      }


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

    /* Clean up */
    EVP_CIPHER_CTX_free(ctx);


    return ciphertext_len;
}


// Define the decryption function

int main() {
    // Declare variables
        ssize_t varread;
    char server_message[1024];
    char client_message[1024];

    struct sockaddr_in server_addr;
    char server_ip[16]= "192.168.92.132";
    int server_port = 10888;

    struct sockaddr_in client_addr;
    socklen_t client_addr_len = sizeof(client_addr);

    // Create socket
    int client_sock;
        int sever_sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sever_sock < 0) {
```

```
        perror("Socket creation failed with error!");
        return 1;
    }
    if(setsockopt(sever_sock, SOL_SOCKET, SO_REUSEADDR,
        &(int){1}, sizeof(int)) < 0) {
        perror("Socket option failed with error!");
        return 1;
    }
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(server_port);
    server_addr.sin_addr.s_addr = inet_addr(server_ip);


    // Bind to the set port and IP
    if(bind(sever_sock, (struct sockaddr*)&server_addr,
        sizeof(server_addr)) < 0) {
        perror("Binding failed with error!");
        return 1;
    }
    printf("Done with binding with IP: %s, Port: %d\n",
        server_ip, server_port);

    // Listen for clients:
    if(listen(sever_sock, 3) < 0) {
        perror("Listening failed with error!");
        return 1;
    }

    // Accept an incoming connection
    if((client_sock = accept(sever_sock, (struct sockaddr
        *)&client_addr, &client_addr_len)) < 0) {
        perror("Accepting failed with error!");
        return 1;
    }
    char * client_ip = inet_ntoa(client_addr.sin_addr);
    int client_port = ntohs(client_addr.sin_port);
    printf("Client connected at IP: %s and port: %i\n",
        client_ip, client_port);

    // Declare key
    //unsigned char key[RC4_KEY_LENGTH];

    // Generate key
    // if (!RAND_bytes(key, RC4_KEY_LENGTH)) {
    //     handleErrors();
    // }
```

```
// Send the key to the client
//send(client_sock, key, RC4_KEY_LENGTH, 0);

// Declare key and iv
unsigned char key[CHACHA_KEY_LENGTH];
unsigned char iv[CHACHA_IV_LENGTH];

// Generate key and iv
if (!RAND_bytes(key, CHACHA_KEY_LENGTH)) {
    handleErrors();
}
if (!RAND_bytes(iv, CHACHA_IV_LENGTH)) {
    handleErrors();
}

// print key and iv
printf("Key:");
for (int i = 0; i < CHACHA_KEY_LENGTH; i++) {
    printf("%02x", key[i]);
}
printf("\n");

printf("IV:");
for (int i = 0; i < CHACHA_IV_LENGTH; i++) {
    printf("%02x", iv[i]);
}
printf("\n");

// correct the key
send(client_sock, key, CHACHA_KEY_LENGTH, 0);

// Send the iv to the client
send(client_sock, iv, CHACHA_IV_LENGTH, 0);


// Receive client's message
varread = recv(client_sock, client_message, 1024, 0);
printf("CHACHA20-Encrypt:%s\n", client_message);
printf("CHACHA20-Encrypt-Length:%d\n", strlen(
    client_message));
// print cipher as byte
printf("CHACHA20-Encrypt-Byte:");
for (int i = 0; i < strlen(client_message); i++) {
    printf("%02x",client_message[i]);
}
```

```
printf("\n");

// Base64 decode
unsigned char decoded_message[1024];
int decoded_message_len = base64_decode(
    client_message, strlen(client_message),
    decoded_message);

// Decrypt the message
unsigned char decrypted_message[1024];
int decrypted_message_len = stream_decrypt(
    decoded_message, decoded_message_len, key, iv,
    decrypted_message);

// Print the decrypted message
printf("CHACHA20 Decrypted message: %s\n",
    decrypted_message);


// Receive OTP encrypted message
char otp_encrypted_message[1024];
char otp_key[varread];
varread = recv(client_sock, otp_encrypted_message,
    1024, 0);
varread = recv(client_sock, otp_key, 1024, 0);
printf("OTP Encrypt: %x\n", otp_encrypted_message);
printf("OTP Key: %x\n", otp_key);

// Decrypt the OTP message
char otp_decrypted_message[varread];
for (int i = 0; i < varread; i++) {
    otp_decrypted_message[i] = otp_encrypted_message[
        i] ^ otp_key[i];
}

// Print the OTP decrypted message
printf("OTP Decrypted message: %s\n",
    otp_decrypted_message);

// Respond to client
strcpy(server_message, "Test Over!");

// Send the message to the client
send(client_sock, server_message, strlen(
    server_message), 0);
```

```
    // Close the socket
    close(client_sock);
    close(sever_sock);


    return 0;
}
```

4.2 Client.c

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

#include <openssl/conf.h>
#include <openssl/evp.h>
#include <openssl/err.h>
#include <openssl/dh.h>
#include <openssl/rand.h>

#define AES_KEY_LENGTH 32
#define AES_BLOCK_SIZE 16
#define RC4_KEY_LENGTH 16

#define CHACHA_KEY_LENGTH 32
#define CHACHA_IV_LENGTH 12


void handleErrors(void)
{
    ERR_print_errors_fp(stderr);
    abort();
}

// Base64 decode
int base64_decode(const unsigned char *input, int length,
    unsigned char *output) {
    return EVP_DecodeBlock(output, input, length);
}

// Base64 encode
```

```
int base64_encode(const unsigned char *input, int length,
    unsigned char *output) {
    return EVP_EncodeBlock(output, input, length);
}

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 initialize the context */
   ctx = EVP_CIPHER_CTX_new();
   if(!ctx) {
        handleErrors();
    }

   /* Initialize the decryption operation. */
   if(EVP_DecryptInit_ex(ctx, EVP_chacha20(), NULL, key,
      iv) != 1) {
        handleErrors();
   }

   /* 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, &plaintext_len,
      ciphertext, ciphertext_len) != 1) {
        handleErrors();
   }


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


   /* Clean up */
   EVP_CIPHER_CTX_free(ctx);
```

```
    return plaintext_len;
}

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 = 0;

   /* Create and initialize the context */
   ctx = EVP_CIPHER_CTX_new();
    if(!ctx) {
        handleErrors();
    }

   /* Initialize the encryption operation. */
   if(EVP_EncryptInit_ex(ctx, EVP_chacha20(), NULL, key,
       iv) != 1) {
        handleErrors();
     }


   /* 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, &ciphertext_len
        , plaintext, plaintext_len) != 1) {
        handleErrors();
     }


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

   /* Clean up */
   EVP_CIPHER_CTX_free(ctx);
```

```
    return ciphertext_len;
}

// One Time Pad encryption
void otp_encrypt(char *plaintext, char *key, char *
    ciphertext) {
    for (int i = 0; i < strlen(plaintext); i++) {
        ciphertext[i] = plaintext[i] ^ key[i];
    }
}

int main() {
    // Declare Variables
    struct sockaddr_in server_addr;

    char server_ip[16] = "192.168.92.132";
    int server_port = 10888;
    char server_message[1024];
    char client_message[1024];

    ssize_t varread;
    unsigned char ciphertext[1024];
    unsigned char rc4_key[RC4_KEY_LENGTH];

    // Create socket:
    int sock = socket(AF_INET, SOCK_STREAM, 0);

    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(server_port);
    server_addr.sin_addr.s_addr = inet_addr(server_ip);

    // Send connection request to server, be sure to set
        port and IP the same as server-side
    if(connect(sock, (struct sockaddr*) &server_addr,
        sizeof(server_addr)) < 0) {
        perror("Connection failed with error!");
        return 1;
    }

    // Perform Diffie-Hellman key exchange and decrypt
        AES key and IV

    // Define the chacha20 key and iv
    unsigned char chacha_key[CHACHA_KEY_LENGTH];
    unsigned char chacha_iv[CHACHA_IV_LENGTH];
```

```
// recv key
//varread = recv(sock, rc4_key, RC4_KEY_LENGTH, 0);
// Receive key from server and send it back for
    confirmation
varread = recv(sock, chacha_key, CHACHA_KEY_LENGTH,
    0);

// Receive IV from server and send it back for
    confirmation
varread = recv(sock, chacha_iv, CHACHA_IV_LENGTH, 0);

// Print the key and iv
printf("Key:");
for (int i = 0; i < CHACHA_KEY_LENGTH; i++) {
    printf("%02x", chacha_key[i]);
}
printf("\n");

printf("IV:");
for (int i = 0; i < CHACHA_IV_LENGTH; i++) {
    printf("%02x", chacha_iv[i]);
}
printf("\n");

// Get input from the user:
//printf("Enter message sent to the server: ");
//fgets(client_message, sizeof(client_message), stdin
    );
strcpy(client_message, "Hello, this is a test message
    !");


// Encrypt the message
int ciphertext_len = stream_encrypt(client_message,
    strlen(client_message), chacha_key, chacha_iv,
    ciphertext);

// print the encrypted message
printf("CHACHA20 Encrypt: %s\n",ciphertext);
printf("CHACHA20 Encrypt Length: %d\n",
    ciphertext_len);

// print cipher as byte
printf("CHACHA20 Encrypt Byte:");
for (int i = 0; i < ciphertext_len; i++) {
```

```
        printf("%02x",ciphertext[i]);
    }
    printf("\n");

    // Decrypt the message
    unsigned char decrypted_message[1024];
    int decrypted_message_len = stream_decrypt(ciphertext
        , ciphertext_len, chacha_key, chacha_iv,
        decrypted_message);

    // Print the decrypted message
    printf("CHACHA20 Decrypt: %s\n", decrypted_message);

    // Bse64 encode the message
    unsigned char base64_encoded_message[1024];
    int base64_encoded_message_len = base64_encode(
        decrypted_message, decrypted_message_len,
        base64_encoded_message);

    // Send the message to server:
    send(sock, base64_encoded_message,
        base64_encoded_message_len, 0);

    // Init OTP as length of the client_message
    char otp_key[strlen(client_message)];
    for (int i = 0; i < strlen(client_message); i++) {
        otp_key[i] = rand() % 256;
    }

    // print
    printf("OTP Plain: %s\n", client_message);

    // Encrypt the message with OTP
    char otp_ciphertext[strlen(client_message)];
    otp_encrypt(client_message, otp_key, otp_ciphertext);

    // print the OTP encrypted message
    printf("OTP Encrypt: %x\n",otp_ciphertext);
    printf("OTP Key: %x\n",otp_key);

    // send the OTP ciphertext to server
    send(sock, otp_ciphertext, strlen(otp_ciphertext), 0)
        ;

    // wait
    sleep(1);
```

```
    // send the OTP key to server
    send(sock, otp_key, strlen(otp_key), 0);



    // Receive the server's response:
    varread = read(sock, server_message, 1024);
    printf("End: %s\n", server_message);

    // Close the socket:
    close(sock);

    return 0;

}
```