CSE 433S HW 2 - Comm with Block Cipher

Zihan Chen

September 2024

Contents

1 Desgin

I implemented a simple client-server communication using AES block cipher.

The whole process is as follows:

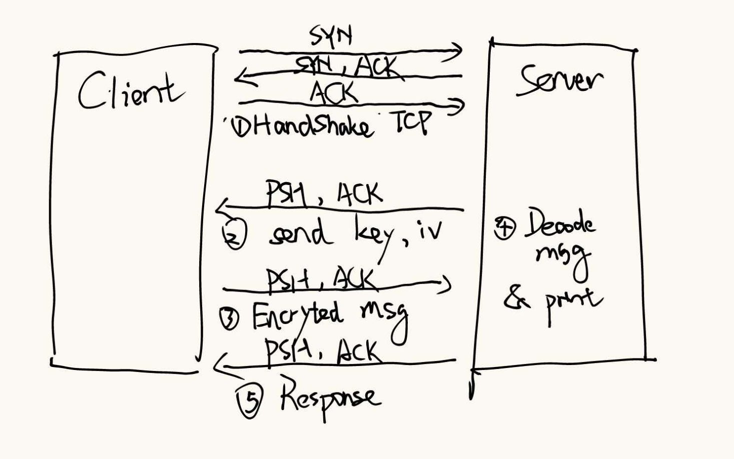
1. Sever and Client establish a connection through TCP handshake.

2. The server generates a random key and iv, sends the key and iv to the client, and then decrypts the message sent by the client.

3. The client encrypts the message using the key and iv received from the server and sends the encrypted message to the server.

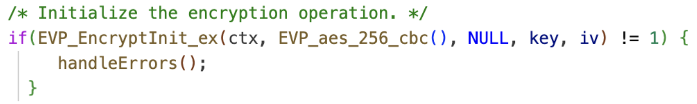
4. The server then decrypts the message and print on the screen.

5. The server sends a response to the client.



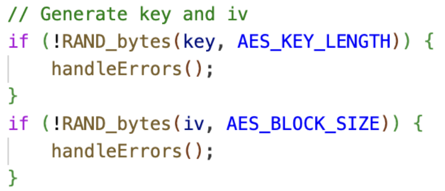
1.1 AES

I choose to use the AES-256-CBC encryption algorithm. AES is a well-known symmetric encryption algorithm. AES-256-CBC is famous for its security and efficiency.



```
/* Initialize the encryption operation. */
if(EVP_EncryptInit_ex(ctx, EVP_aes_256_cbc(), NULL, key, iv) != 1) {
    handleErrors();
  }
```

The key and iv are generated randomly using the RAND_bytes function provided by the OpenSSL library.



```
// Generate key and iv
if (!RAND_bytes(key, AES_KEY_LENGTH)) {
    handleErrors();
}
if (!RAND_bytes(iv, AES_BLOCK_SIZE)) {
    handleErrors();
}
```

1.2 base64

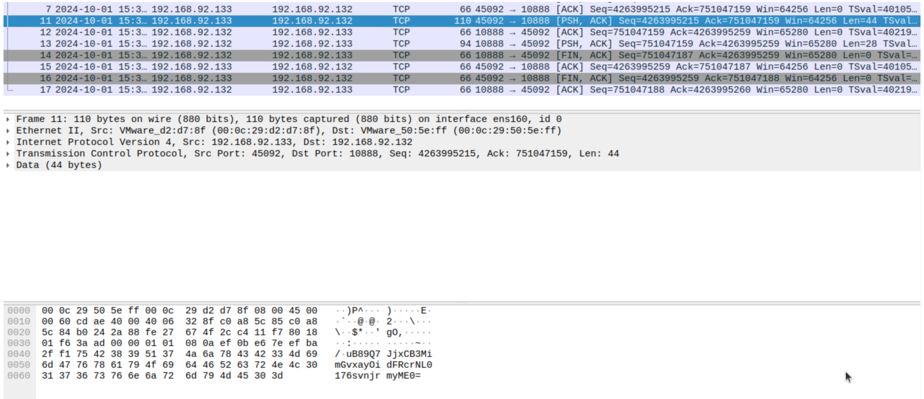
I use the base64 encoding and decoding functions provided by the OpenSSL library to encode and decode the message.

Base64 is important because it can aviod the problem of sending 0x00 in the message, which may cause the message to be truncated.

```c
// Base64 decode
int base64_decode(const unsigned char *input, int length, unsigned char *output) {
    int len = EVP_DecodeBlock(output, input, length);
    //remove '\0' from the output
    while(output[len-1] == '\0') len--;
    return len;
}


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

We can see the message is transported by base64 code.



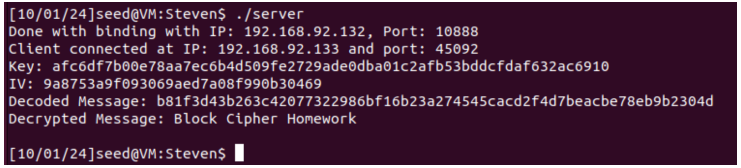
```
 7 2024-10-01 15:3… 192.168.92.133   192.168.92.132   TCP    66 45092 → 10888 [ACK] Seq=4263995215 Ack=751047159 Win=64256 Len=0 TSval=40105…
11 2024-10-01 15:3… 192.168.92.133   192.168.92.132   TCP   110 45092 → 10888 [PSH, ACK] Seq=4263995215 Ack=751047159 Win=64256 Len=44 TSval…
12 2024-10-01 15:3… 192.168.92.132   192.168.92.133   TCP    66 10888 → 45092 [ACK] Seq=751047159 Ack=4263995259 Win=65280 Len=0 TSval=40219…
13 2024-10-01 15:3… 192.168.92.132   192.168.92.133   TCP    94 10888 → 45092 [PSH, ACK] Seq=751047159 Ack=4263995259 Win=65280 Len=28 TSval…
14 2024-10-01 15:3… 192.168.92.132   192.168.92.133   TCP    66 10888 → 45092 [FIN, ACK] Seq=751047187 Ack=4263995259 Win=65280 Len=0 TSval=…
15 2024-10-01 15:3… 192.168.92.133   192.168.92.132   TCP    66 45092 → 10888 [ACK] Seq=4263995259 Ack=751047187 Win=64256 Len=0 TSval=40105…
16 2024-10-01 15:3… 192.168.92.133   192.168.92.132   TCP    66 45092 → 10888 [FIN, ACK] Seq=4263995259 Ack=751047188 Win=64256 Len=0 TSval=…
17 2024-10-01 15:3… 192.168.92.132   192.168.92.133   TCP    66 10888 → 45092 [ACK] Seq=751047188 Ack=4263995260 Win=65280 Len=0 TSval=40219…

Frame 11: 110 bytes on wire (880 bits), 110 bytes captured (880 bits) on interface ens160, id 0
Ethernet II, Src: VMware_d2:d7:8f (00:0c:29:d2:d7:8f), Dst: VMware_50:5e:ff (00:0c:29:50:5e:ff)
Internet Protocol Version 4, Src: 192.168.92.133, Dst: 192.168.92.132
Transmission Control Protocol, Src Port: 45092, Dst Port: 10888, Seq: 4263995215, Ack: 751047159, Len: 44
Data (44 bytes)

0000  00 0c 29 50 5e ff 00 0c  29 d2 d7 8f 08 00 45 00   ··)P^··· )·····E·
0010  00 60 cd ae 40 00 40 06  32 8f c0 a8 5c 85 c0 a8   ·`··@·@· 2···\···
0020  5c 84 b0 24 2a 88 fe 27  67 4f 2c c4 11 f7 80 18   \··$*··' gO,·····
0030  01 f6 3a ad 00 00 01 01  08 0a ef 0b e6 7e ef ba   ··:····· ·····~··
0040  2f f1 75 42 38 39 51 37  4a 6a 78 43 42 33 4d 69   /·uB89Q7 JjxCB3Mi
0050  6d 47 76 78 61 79 4f 69  64 46 52 63 72 4e 4c 30   mGvxayOi dFRcrNL0
0060  31 37 36 73 76 6e 6a 72  6d 79 4d 45 30 3d         176svnjr myME0=
```

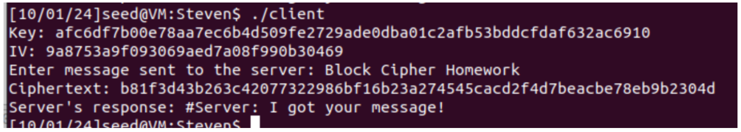
2 Results

When we run the server and client, we can see the following output:



```
[10/01/24]seed@VM:Steven$ ./server
Done with binding with IP: 192.168.92.132, Port: 10888
Client connected at IP: 192.168.92.133 and port: 45092
Key: afc6df7b00e78aa7ec6b4d509fe2729ade0dba01c2afb53bddcfdaf632ac6910
IV: 9a8753a9f093069aed7a08f990b30469
Decoded Message: b81f3d43b263c42077322986bf16b23a274545cacd2f4d7beacbe78eb9b2304d
Decrypted Message: Block Cipher Homework

[10/01/24]seed@VM:Steven$
```



```
[10/01/24]seed@VM:Steven$ ./client
Key: afc6df7b00e78aa7ec6b4d509fe2729ade0dba01c2afb53bddcfdaf632ac6910
IV: 9a8753a9f093069aed7a08f990b30469
Enter message sent to the server: Block Cipher Homework
Ciphertext: b81f3d43b263c42077322986bf16b23a274545cacd2f4d7beacbe78eb9b2304d
Server's response: #Server: I got your message!
[10/01/24]seed@VM:Steven$
```

As the output shows, the server successfully decrypts the message sent by the client and sends a response to the client.

3 Failure Attempt

3.1 Diffie-Hellman Key Exchange

I tried to use the Diffie-Hellman key exchange algorithm to exchange the key between the server and the client. However, I failed to implement it.

Some strange error occurred and I am sure that's not my code problem. It must my environment problem.

3.2 Base64 modern version

I tried to use the modern version of the base64 encoding and decoding functions provided by the OpenSSL library. However, the modern base64 can't be transmitted correctly. It will always cause my send() function to block and the server can't receive the message.

But the code is worth to show.

```
int base64_decode(const unsigned char *input, int
    length, unsigned char *output) {
    BIO *bio, *b64;
    int decoded_len;

    b64 = BIO_new(BIO_f_base64());
    bio = BIO_new_mem_buf(input, length);
    bio = BIO_push(b64, bio);

    // Disable newlines
    BIO_set_flags(bio, BIO_FLAGS_BASE64_NO_NL);

    // Decode the input data
    decoded_len = BIO_read(bio, output, length);
    if (decoded_len < 0) {
        BIO_free_all(bio);
        return -1; // Decoding failed
    }

    // Clean up
    BIO_free_all(bio);

    return decoded_len;
}

// Base64 encode
```

```
int base64_encode(const unsigned char *input, int
    length, unsigned char *output) {
    BIO *bio, *b64;
    BUF_MEM *buffer_ptr;

    b64 = BIO_new(BIO_f_base64());
    bio = BIO_new(BIO_s_mem());
    bio = BIO_push(b64, bio);

    // Write the input data to the BIO
    BIO_write(bio, input, length);
    BIO_flush(bio);
    BIO_get_mem_ptr(bio, &buffer_ptr);

    // Copy the encoded data to the output buffer
    memcpy(output, buffer_ptr->data, buffer_ptr->
        length);
    output[buffer_ptr->length] = '\0'; // Null-
        terminate the output

    // Clean up
    BIO_free_all(bio);

    return buffer_ptr->length;
}
```

4 appendix

4.1 Server Code

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

#define AES_KEY_LENGTH 32
#define AES_BLOCK_SIZE 16

// 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) {
     int len = EVP_DecodeBlock(output, input, length);
     //remove '\0' from the output
     while(output[len-1] == '\0') len--;
     return len;
}


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

int block_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_aes_256_cbc(), 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 block_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_aes_256_cbc(), 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;
}


int main() {
     // Declare variables
     ssize_t varread;
     char server_message[1024];
     char client_message[4096];

     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);

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

```
// Base64 decoded message
unsigned char decoded_message[4096];
int decoded_message_len;

// plain text
unsigned char plaintext[1024];
int plaintext_len;

// 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);

// Generate key and iv
if (!RAND_bytes(key, AES_KEY_LENGTH)) {
     handleErrors();
}
if (!RAND_bytes(iv, AES_BLOCK_SIZE)) {
     handleErrors();
}

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

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

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

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

// Receive client's message
varread = recv(client_sock, client_message, 4096,
     0);

// Base64 decode
decoded_message_len = base64_decode(
    client_message, varread, decoded_message);

// Print the decoded message as hex
```

```
    printf("Decoded␣Message:␣");
    for (int i = 0; i < decoded_message_len; i++) {
        printf("%02x", decoded_message[i]);
    }
    printf("\n");

    // Decrypt the message
    plaintext_len = block_decrypt(decoded_message,
       decoded_message_len, key, iv, plaintext);

    // Print the decrypted message
    printf("Decrypted␣Message:␣%s\n", plaintext);

    // Respond to client
    strcpy(server_message, "#Server:␣I␣got␣your␣
       message!");
    send(client_sock, server_message, strlen(
       server_message), 0);

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


    return 0;
}
```

4.2 Client Code

```
#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>
#include <openssl/bio.h>
#include <openssl/buffer.h>

#define AES_KEY_LENGTH 32
```

```
#define AES_BLOCK_SIZE 16

// 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) {
    int len = EVP_DecodeBlock(output, input, length);
    //remove '\0' from the output
    while(output[len-1] == '\0') len--;
    return len;
}

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


int block_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_aes_256_cbc(), 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 block_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_aes_256_cbc(), 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;
}

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;

      // Ciphertext
      unsigned char ciphertext[4096];
      int ciphertext_len;

      // Base64 encode ciphertext
      unsigned char base64_encoded_ciphertext[4096];
      int base64_encoded_ciphertext_len;

      // AES key and iv
      unsigned char AES_key[AES_KEY_LENGTH];
      unsigned char AES_iv[AES_BLOCK_SIZE];

      // 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;
      }

      // Receive the key from the server
      varread = read(sock, AES_key, AES_KEY_LENGTH);

      // sleep
      sleep(1);

      // Receive the iv from the server
      varread = read(sock, AES_iv, AES_BLOCK_SIZE);

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

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

      // Get input from the user:
      printf("Enter message sent to the server: ");
      fgets(client_message, sizeof(client_message),
         stdin);

      // Encrypt the message
```

```
    ciphertext_len = block_encrypt(client_message,
        strlen(client_message), AES_key, AES_iv,
        ciphertext);

    // Base64 encode the ciphertext
    base64_encoded_ciphertext_len = base64_encode(
        ciphertext, ciphertext_len,
        base64_encoded_ciphertext);

    // Send the message to server:
    send(sock, base64_encoded_ciphertext,
        base64_encoded_ciphertext_len, 0);

    // print the ciphertext as hex
    printf("Ciphertext:␣");
    for (int i = 0; i < ciphertext_len; i++) {
        printf("%02x", ciphertext[i]);
    }
    printf("\n");


    // Receive the server's response:
    varread = read(sock, server_message, 1024);

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

    // Close the socket:
    close(sock);

    return 0;

}
```