Suhas EC21B1113 | Thribhuvan EC21B1093

Embedded Systems Lab Report

Encrypted Serial Communication

Aim:

To exchange encrypted data between two microcontrollers via a serial port (Ex: Bluetooth, …)

Theory:

The process of transmitting data between two devices or systems over a serial connection using encryption techniques to ensure the confidentiality and integrity of the transmitted data is referred to as encrypted serial communication. Serial communication entails bit-by-bit data transmission over a single communication channel or wire, which is typically a physical wire or a wireless connection.

Encryption is the process of converting data into an unreadable format that can only be deciphered with the help of a secret key or password. Encrypted communication is a critical tool for ensuring secure data transmission over public networks such as the internet, where data can be intercepted or manipulated by malicious actors.

For serial communication, several encryption techniques are available, including symmetric key encryption, asymmetric key encryption, and hashing. Asymmetric key encryption uses different keys for encryption and decryption, whereas symmetric key encryption uses the same secret key for both encryption and decryption. Hashing is a process that converts data of any size into a fixed-size output that can be used to validate the data's integrity.

The data in encrypted serial communication is encrypted by the sender using an encryption algorithm before being transmitted over the serial connection. The data is then decrypted by the receiver using the same encryption algorithm and decryption key. This procedure ensures that only the intended recipient has access to and reads the information.

Encrypted serial communication, in summary, is a secure method of transmitting data over a serial connection that involves encrypting the data using encryption techniques to ensure the confidentiality and integrity of the transmitted data. This procedure assists in preventing malicious actors from intercepting and tampering with sensitive data.

This library uses Elliptic Curve Cryptography (ECC) with Diffie-Hellman algorithm to create a shared secret, which is then used to encrypt messages. An external library uECC (https://github.com/kmackay/micro-ecc) was used.

Materials Required:

Any 2 Arduino boards, 2 Bluetooth modules (for testing, operating at 115200 baud rate)

Library overview:

The library uses the following macros to improve code readability and reduce writing faulty/repetitive code.

0. **Constants**
   CMDLEN = 5
   CMDMSG = "MESS\n"
   CMDHELLO1 = "HEL1\n"
   CMDHELLO2 = "HEL2\n"
   CMDRECP = "PUBL\n"

CMDRECA = "PUBA\n"
key_size = 20
DEBUG

1. **LOG_START_S(X) LOG_END_S(X) LOG_START(X) LOG_END(X)**
   The above macros are used for efficient debugging.
   If the *DEBUG* macro is set, the macros print the status of the code in the function (start, stop).

```
#ifdef DEBUG
#define LOG_START_S(X) Serial.print("[DEBUG BEGIN] FN:"); Serial.println(X);
#define LOG_END_S(X) Serial.print("[DEBUG END] FN:"); Serial.println(X);
#else
#define LOG_START_S(X)
#define LOG_END_S(X)
#endif

#define LOG_START LOG_START_S(__func__);
#define LOG_END LOG_END_S(__func__);
```

2. **DELAYUS**
   This macro provides the least delay in microseconds, useful to delay for *pinMode(…)* changes.

```
#define DELAYUS delayMicroseconds(100);
```

3. **COOLDOWN**
   This macro delays the processor by 0.5s. This is useful when we are trying to filter out bad data from the useful data.

```
#define COOLDOWN delay(500);
```

4. **RECV**
   This macro reads `*CMDLEN*` bytes into the `*cmd*` buffer.
5. **COMP(X)**
   This macro invokes *strncmp(…)* on the `*cmd*` buffer.
6. **WAIT(X)**
   This macro waits until the `*COMP(X)*` returns 0 and invokes `*COOLDOWN*`

```
#define RECV recv_line(cmd, CMDLEN);
#define COMP(X) strncmp(cmd, X, CMDLEN)
#define WAIT(X) while (COMP(X)) {RECV; COOLDOWN;}
```

The library has the following usable functions:

1. **`secure_serial::begin`**
   Creates a `SoftwareSerial` object and a key pair using a secure random number generator.

Random generator code:

```
static int random_uECC(uint8_t *dest, unsigned size) {
  while (size) {
    uint8_t val = 0;
    for (unsigned i = 0; i < 8; ++i) {
        int init = analogRead(0);
        int count = 0;
        while (analogRead(0) == init) ++count;
        if (!count) val = (val << 1) | (init & 0x01);
```

```
            else val = (val << 1) | (count & 0x01);
        }

        *dest = val;
        ++dest; --size;
      }

      return 1;
    }


    void secure_serial::begin(uint8_t rx, uint8_t tx, bool first_device) {
        LOG_START
        serial = new SoftwareSerial(rx, tx);
        curve = uECC_secp160r1();
        first = first_device;

        pinMode(rx, INPUT);
        pinMode(tx, OUTPUT);

        serial->begin(115200);
        while (!(*serial));

        uECC_set_rng(&random_uECC);
        uECC_make_key(public_key, private_key, curve);
        LOG_END
    }
```

2. secure_serial::initial_sequence

Starts the exchange of key pairs among devices by following the process:

The *first device* performs the following:

1. Send HELLO1
2. Receive HELLO2
3. Send RECP
4. Wait for RECA
5. Receive public key (40 bytes)
6. Wait for RECP
7. Send RECA
8. Send public key (40 bytes)

The *second device* performs the following:

1. Receive HELLO1
2. Send HELLO2
3. Wait for RECP
4. Send RECA
5. Send public key (40 bytes)
6. Send RECP
7. Receive public key (40 bytes)

```
void secure_serial::initial_sequence() {
    LOG_START
    if (first) {
        send_hello();
        WAIT(CMDHELLO2);
        recv_pub_key();
        WAIT(CMDRECP);
        send_pub_key();
```

```
        } else {
            WAIT(CMDHELLO1);
            send_hello2();
            WAIT(CMDRECP);
            send_pub_key();
            recv_pub_key();
        }

        LOG_END
    }
```

3. **secure_serial::send_from_serial**
   This function sends unencrypted raw data from the serial port to the target device.

```
void secure_serial::send_from_serial() {
    LOG_START
    while (Serial.available())
        serial->print((char) Serial.read());
    recv_to_serial();
    LOG_END
}
```

4. **secure_serial::recv_to_serial**
   Receives unencrypted data to the serial port

```
void secure_serial::recv_to_serial() {
    LOG_START
    while (serial->available()) {
        Serial.write((char) serial->read());
    }
    LOG_END
}
```

5. **secure_serial::available**
   Returns the number of available characters in the serial port to read

```
int secure_serial::available() {
    return serial->available();
}
```

6. **secure_serial::send_msg**
   Sends encrypted message to the recipient

```
void secure_serial::send_msg(uint8_t* message, int len) {
    LOG_START
    send(CMDMSG);
    serial->print(len);
    for (int i = 0; i < len; i++) {
        serial->print((char)(message[i] ^ shared_secret[i%key_size]));
    }
    LOG_END
}
```

7. **secure_serial::recv_msg**
   Receives encrypted message and decrypts into a target buffer

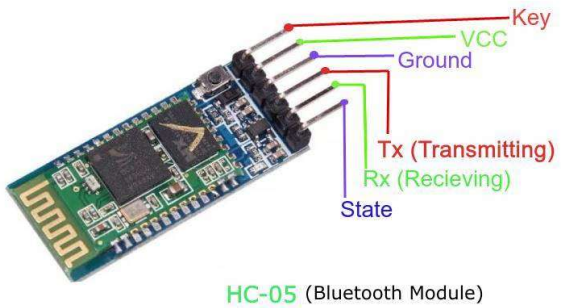
```
void secure_serial::recv_msg(uint8_t* message, int len) {
    LOG_START
    WAIT(CMDMSG);
    int r = serial->read();
    serial->readBytes(message, r);

    for (int i=0; i<r; ++i) {
        message[i] ^= shared_secret[i % key_size];
    }
    LOG_END
}
```

Procedure

[For testing communication]

1. Connect the Bluetooth modules following the pin diagram:



HC-05 (Bluetooth Module)

2. Configure the Bluetooth modules to 115200 baud rates with different names and pair them
3. In the code, include the header: secure_serial.h
4. Create a new secure_serial object with rx pin, tx pin and the first_device parameters.
5. The first_device parameter should be true for only one device.
6. Call the initial_sequence function to share public keys and create a shared key.
7. Use the send_msg(…) and recv_msg(…) functions to send and receive encrypted data

Sample code

```
#include "secure_serial.h"
secure_serial bt;


void setup() {
    Serial.begin(115200);
    LOG_START
    bt.begin(2, 3, true);  // 2 -> TXD 3 -> RXD, false for the second device
    bt.initial_sequence();
    LOG_END
}

uint8_t* message[64];

void loop() {
    bt.send_msg("This is an encrypted message!", 29);
 // bt.recv_msg(message, 64); (Second device)
    delay(1000);
}
```

Result

The Bluetooth modules shared keys and were exchanging encrypted data using the secure_serial library. This library is useful for all types of serial_data and provides very little overhead as it uses XOR encryption with sufficiently large key.