情報通信実験（課題⑥　共通鍵暗号方式による暗号化と復号を用いた通信）

実験レポート

出席番号：　１６　名前：　小形　孔明　　実験日：2025 年　11 月　13 日

１．課題目的

UTF-8 文字列を AES/CBC/PKCS5Padding と Base64 により安全に送受信する一連の流れ（文字列↔バイト、暗号↔復号、エンコード↔デコード）を正しく実装し、鍵 16byte・IV 16byte 要件と不一致時の挙動を理解する。

２．課題内容

encode/decode を実装して全角混在文字の往復一致を確認し、既存の ServerSocket／Socket／BufferedReader／PrintWriter／Thread に組み込んで暗号化通信を実現する。また、鍵・IV 不一致では復号失敗や無意味データとなり、例えば "bye" 判定は発火しないことを示す。なお、確認する文字列は

・a
・hello
・あ
・制御
・Σ

３．作成プログラムの重要箇所の解説

1. サーバ通信本体（受信→復号→bye 判定→暗号化返信）

String msg = MyCrypt.decode(in.readLine(), AES_KEY, AES_IV);

if ("bye".equals(msg)) break; else out.println(MyCrypt.encode("ECHO: " + msg, AES_KEY, AES_IV));//復号直後に`"bye"`を判定し、通常は暗号化して返信する。

2. クライアント送信（入力→暗号化→送出）

String s = MyCrypt.encode(std_in.readLine(), AES_KEY, AES_IV);

out.println(s);//送信前に必ず AES+Base64 でエンコードする。

3. クライアント受信（取得→復号→表示）

String resp = MyCrypt.decode(in.readLine(), AES_KEY, AES_IV);

System.out.println("Client> " + resp);//受信直後に復号し、UTF-8 文字列として表示する。

4. サーバ I/O 生成（UTF-8 固定）

in　= new BufferedReader(new InputStreamReader(client_socket.getInputStream(),　StandardCharsets.UTF_8));

out = new PrintWriter(new OutputStreamWriter(client_socket.getOutputStream(), StandardCharsets.UTF_8), true);//文字化け防止の要所で、行送信を自動フラッシュにする。

5. 鍵・IV の定義（サーバ／クライアント共通で一致が必須）

static final String AES_KEY = "0123012301230123", AES_IV = "abcdefghijklmnop";//鍵と IV は UTF-8 で 16byte（AES-128）かつ送受で同一にするのが前提。

４．結果

変換前	変換後
a	309EB05798E08DAD6C27B26299624E28

hello	0F03E4BB2F7E0CDF105E3ECCCF050B50
あ	3B1B793887B4E618C4320E4BFD1D1B22
制御	1679D6DA05812C49F82E55E1650DB2ED
Σ	AD28C0B2FDD67A52A4BE66E46391AE6F

なお、UTF-8,CBC,PKCS#5,鍵＝0123012301230123,初期化ベクトル＝abcdefghijklmnop

また、https://tool.hiofd.com/ja/aes-encrypt-online/

のようなサイトを使用すれば、容易に確認することができる。

５．感想・考察

本実験では、テキスト送受信に暗号処理を組み込む基本手順を、Eclipse のコンソール入出力とデバッガのみで確認した。また、いわゆる失敗実験は行っておらず、例外系として確かめたのは「鍵や初期化ベクトルがサーバと異なる場合」と「鍵や初期化ベクトルに全角文字を含めた場合」の 2 点に限定している。送信側の処理は「UTF-8 文字列をバイト列に変換し、AES(CBC/PKCS5Padding)で暗号化し、最後に Base64 で文字列化する」、受信側は「Base64 デコード、AES 復号、UTF-8 で文字列化」という直列の流れである。AES のブロック長が常に 128bit で一定であること、Eclipse のコンソール表示で把握できた。

実装では、ServerSocket/Socket で接続を確立し、BufferedReader/PrintWriter を UTF-8 固定で生成したうえで、行単位 I/O の直前直後に MyCrypt.encode と MyCrypt.decode を挿入した。重要な注意点は 2 つで、暗号化直後のバイト配列を直接 String にしないこと、getBytes と newString で常に UTF-8 を明示することである。これらを守るだけで、「あ」「や」「Σ」といった多バイト文字を含む文字列でも送受の往復で一致した。今回の範囲で確認した例外系の 1 つめは、鍵や初期化ベクトル(IV)がサーバと異なる場合であり、復号結果が無意味な文字列となったり、CBC では BadPaddingException が出たりして、アプリ層の「bye」判定には到達しないことが分かった。2 つめは、鍵や IV に全角文字を含めた場合で、UTF-8 では 1 文字が複数バイトになるため、鍵は 16/24/32byte、IV はモードに応じた所定長(CBC なら 16byte)を満たすよう文字列全体の合計バイト数を厳密に管理する必要がある。今回は Eclipse のデバッガで文字列から得た byte[]の長さを確認し、条件を満たしていることを個別に確かめた。つまり、英数字に限定しなくても理屈上は利用可能だが、最終的に「所定のバイト長」に合致していなければ動作しないという基本を確認できた。

観測は Eclipse 内での入出力のみだが、この範囲でも要点は十分に把握できた。具体的には、暗号化後のデータが Base64 により英数字と記号(+/=)のみの見た目になること、1 行が 1 メッセージとして区切られるため行境界が扱いやすいこと、平文の長さと比べて出力文字数が増える傾向があること、などである。

運用上の理解として、Base64 は秘匿化ではなく運搬のための可逆エンコードであること、AES は文字ではなくバイト列を対象とする変換であることを整理できた。今回の実験では固定の鍵と IV を使って挙動確認を行ったが、実運用であれば、鍵はパスフレーズをそのまま使うのではなく KDF で 16/24/32byte を導出し、IV は毎回 SecureRandom で生成して暗号文の先頭に付与するのが望ましい。また、完全性まで必要とする場面では、CBC ではなく AES-GCM のような AEAD を採用し、タグ検証に失敗したメッセージは即座に破棄する設計が適切である。今回の確認範囲は Eclipse 内だけではあるが、これらの原則を頭に置けば、より安全な構成への移行はスムーズだと感じた。

総括すると、今回の実験を通じて、文字列処理、暗号処理、ネットワーク処理の 3 層を一貫した流れとして捉え、正しい順序で処理を並べることの重要性を理解できた。具体的には、(1)文字列からバイト列へ、暗号化から Base64 へ、そして受信側でその逆順に戻す、という直列手順を守れば、多バイト文字を含む日本語でも問題なく往復できること、(2)鍵と IV は送受で一致が必須で、違えば復号できずアプリ層のロジック(「bye」など)も働かないこと、(3)鍵や IV に全角文字を用いること自体は可能だが、UTF-8 での合計バイト長という制約を満たさなければならないこと、の 3 点である。