Network Security Lab 4: Symmetric Encryption Lab Solutions

Student Name: Assem Chebly

December 5, 2025

Question 1: Basic File Encryption with Multiple Algorithms

Step-by-Step Solution

1. **Create a test file:**

```
echo "This is top secret data for TP4. Top Secret." > secret.txt
```

2. **Encrypt using AES-256-CBC:**

```
openssl enc -aes-256-cbc -pbkdf2 -in secret.txt -out file.enc
```

password:TP4-NETSEC.

3. **Verify the encrypted file is unreadable:**

```
cat file.enc
```

The output will show unintelligible binary data.$Salted_{i,dhwrBUSM'ej}$

4. **Share with colleague and decrypt:**

```
openssl enc -aes-256-cbc -pbkdf2 -d -in file.enc -out decrypted.txt
```

Enter the same password used for encryption.password:TP4-NETSEC

5. **Verify restoration:**

```
diff plaintext.txt decrypted.txt
```

No output means files are identical.

6. **Encrypt with Base64 encoding:**

```
openssl enc -aes-256-cbc -a -in plaintext.txt -out file_base64.enc
```

The `-a` *or* `-base64` *option creates a text-safe encoded output.*

7. **Try different algorithms:**

```
# Using AES-192
openssl enc -aes-192-cbc -a -in plaintext.txt -out file_aes192.enc
```

```
# Using DES3 (Triple DES)
openssl enc -des3 -a -in plaintext.txt -out file_des3.enc

# Using Blowfish
openssl enc -bf-cbc -a -in plaintext.txt -out file_blowfish.enc
```

Key Points

Base64 encoding makes encrypted data text-safe for transmission through email or text channels, but does not provide additional security. The `-a` flag is equivalent to `-base64`.

Question 2: Symmetric Encryption with Mode of Operation

Step-by-Step Solution

We tested Cipher Feedback Mode (CFB) with a new secret file.

1. **Encrypt using AES-256 in CFB mode:**

```
openssl enc -aes-256-cfb -in plaintext.txt -out ciphertext.enc
```

Enter a password when prompted: TP4-NETSEC

2. **Decrypt the ciphertext:**

```
openssl enc -aes-256-cfb -d -in ciphertext.enc -out recovered.txt
```

Enter the same password.

3. **Verify successful recovery:**

```
cat recovered.txt
```

Explanation: CFB Mode

CFB (Cipher Feedback) is a mode that turns block ciphers into stream ciphers, allowing encryption of data without padding requirements.

Question 3: File Size Analysis

Step-by-Step Solution

We analyzed the size difference between plaintext and ciphertext to understand padding and salting.

1. **Create a test file with known size:**

```
echo -n "1234567890" > test.txt
ls -l test.txt
```

2. **Encrypt without Base64:**

```
openssl enc -aes-256-cbc -in test.txt -out test.enc
```

3. **Compare sizes:**

```
ls -l test.txt test.enc
```

Detailed Explanation: File Size Increase

The encrypted file is larger than the plaintext due to several factors:

- **Salt:** OpenSSL adds an 8-byte random salt by default (16 bytes including the "Salted__" header).
- **Padding:** Block ciphers like AES require input to be a multiple of the block size (16 bytes for AES). PKCS#7 padding adds 1-16 bytes to align data.

Example Calculation: A 10-byte file encrypted with AES-CBC will be:

$$16 \text{ bytes (header + salt)} + 16 \text{ bytes (padded data)} = 32 \text{ bytes}$$

If Base64 encoding is used, the size increases by approximately **33%** due to the encoding overhead.

Question 4: Decryption with Incorrect Password

Step-by-Step Solution

1. **Encrypt a file:**

```
echo "Secret data" > secret.txt
openssl enc -aes-256-cbc -in secret.txt -out secret.enc
# Use password: "12341234"
```

2. **Attempt decryption with wrong password:**

```
openssl enc -aes-256-cbc -d -in secret.enc -out failed.txt
# Use password: "wrong456"
```

Results Observed

- OpenSSL will display an error: `"bad decrypt"` or `"digital envelope routines::bad decrypt"`.
- The output file may contain garbage data or be corrupted.
- This demonstrates the importance of password management and key derivation.

Resulting Error

```
bad decrypt
40A7C8FB697C0000:error:1C800064:Provider routines:
ossl_cipher_unpadblock:bad decrypt
```

Technical Explanation

The password is used to derive the encryption key through a Key Derivation Function (KDF). An incorrect password produces a different key, resulting in decryption failure.

Question 5: AES-128 Encryption with Explicit Key and IV

Objective

To encrypt and decrypt a file by manually specifying the Hexadecimal Key (`-K`) and Initialization Vector (`-iv`), bypassing the automatic password-to-key derivation process.

Execution Log

1. **Key and IV Generation:** We generated a random 128-bit key (16 bytes) and IV using OpenSSL.

```
openssl rand -hex 16
# Generated Key: b0f3c2193c4b95aad4456c546cc86610

openssl rand -hex 16
# Generated IV:  05dc23528d8c42824ca7e3bbe589ff83
```

2. **File Creation:**

```
echo "Direct Key Encryption Test" > plain_q5.txt
```

3. **Encryption (Explicit Parameters):** We encrypted the file using AES-128-CBC. Note that when `-K` and `-iv` are used, OpenSSL does not ask for a password and does not write a "Salted" header to the file.

```
openssl enc -aes-128-cbc \
    -K b0f3c2193c4b95aad4456c546cc86610 \
    -iv 05dc23528d8c42824ca7e3bbe589ff83 \
    -in plain_q5.txt -out file_q5.enc
```

4. **Decryption:** We decrypted using the exact same hex values.

```
openssl enc -aes-128-cbc -d \
    -K b0f3c2193c4b95aad4456c546cc86610 \
    -iv 05dc23528d8c42824ca7e3bbe589ff83 \
    -in file_q5.enc -out decrypted_q5.txt
```

5. **Verification:**

```
cat decrypted_q5.txt
# Output: Direct Key Encryption Test

diff decrypted_q5.txt plain_q5.txt
# (No output indicates files are identical)
```

Technical Insight: Direct Key vs. Password

In previous exercises, we provided a password. OpenSSL automatically:

1. Generated a random Salt.
2. Used a Key Derivation Function (PBKDF2) to mix the Password + Salt to create the Key and IV.

In **Question 5**, we skipped the derivation. We provided the raw binary key directly. Because no derivation or salt is involved, the encrypted file does not contain the "Salted__" header and is slightly smaller in size.

Question 6: DES-EDE Encryption (Legacy Algorithm)

Objective

To repeat the manual key/IV encryption process using the `-des-ede` (Triple DES) algorithm and observe the differences in block size and mode behavior.

Execution Log

1. **Key and IV Generation:** DES uses a 64-bit block size (8 bytes), unlike AES (16 bytes). Therefore, we generated an 8-byte IV.

```
openssl rand -hex 16
# Key: 54b4bbb16d677d47ba1d4afbd873d7c7

openssl rand -hex 8
# IV:  7d561fe333c284d8
```

2. **Encryption (Warning Observed):** We executed the encryption command using the generated parameters.

```
openssl enc -des-ede \
    -K 54b4bbb16d677d47ba1d4afbd873d7c7 \
    -iv 7d561fe333c284d8 \
    -in plain_q6.txt -out file_q6.enc
```

System Warning

```
warning: iv not used by this cipher
```

3. **Decryption:**

```
openssl enc -des-ede -d \
    -K 54b4bbb16d677d47ba1d4afbd873d7c7 \
    -iv 7d561fe333c284d8 \
    -in file_q6.enc -out decrepted_q6.txt
```

Note: The same warning regarding the unused IV appeared.

4. **Verification:**

```
cat decrepted_q6.txt
# Output: Test DES-EDE

diff plain_q6.txt decrepted_q6.txt
# (No output indicates successful restoration)
```

Technical Explanation: Why was the IV ignored?

The OpenSSL command `-des-ede` defaults to **ECB (Electronic Codebook)** mode.

- **ECB Mode:** Encrypts each block of data independently. It does not chain blocks together, so it does **not** require an Initialization Vector (IV).
- **Result:** OpenSSL accepted the arguments but correctly warned that the provided IV was ignored during the process. To use an IV with this algorithm, we would have needed to specify `-des-ede-cbc`.

Question 7: DES-CBC with Base64 Encoded Key

Objective

To decrypt a file using the legacy DES-CBC algorithm, where the key is provided in Base64 format and the system runs OpenSSL 3.0+.

Execution Log

1. **Key Preparation:** The key was provided as Base64: `QUJBQkFCQUJBQkFCQUJBQg==`. We decoded it to Hex to obtain the required 8-byte key for DES.

```
echo -n "QUJBQkFCQUJBQkFCQUJBQg==" | base64 -d | xxd -p
# Output: 4142414241424142414241424142
```

Note: We used the first 8 bytes: 4142414241424142.

2. **Initial Attempt (Failure):** We attempted to run the command using standard syntax.

```
openssl enc -des-cbc -K 4142414241424142 -iv ABABABABABABABAB -in
    secret_origin.txt -out secret.enc
```

Error: Unsupported Algorithm

```
Error setting cipher DES-CBC
40D71B3A527D0000:error:0308010C:digital envelope routines:
inner_evp_generic_fetch:unsupported:Algorithm (DES-CBC : 8)
```

Diagnosis: OpenSSL 3.0+ defaults to a secure provider that disables legacy algorithms like Single DES.

3. **Corrected Execution (Legacy Provider):** We explicitly loaded the legacy and default providers to enable DES-CBC.

```
# Encrypt (Creating the source file)
openssl enc -des-cbc -provider legacy -provider default \
    -K 4142414241424142 -iv ABABABABABABABAB \
    -in secret_origin.txt -out secret.enc

# Decrypt
openssl enc -des-cbc -d -provider legacy -provider default \
    -K 4142414241424142 -iv ABABABABABABABAB \
    -in secret.enc -out decrypted_q7.txt
```

4. **Verification:**

```
cat decrypted_q7.txt
# Output: Question 7
```

Implications of Base64 Keys

- **Usage:** Base64 is used to transmit binary keys over text-only protocols (like Email or HTTP) without data corruption.
- **Conversion:** Encryption tools like OpenSSL require raw Hex or Binary input. Administrators must decode the Base64 string (as done in Step 1) before the key can be used for cryptographic operations.

Question 8: Performance Analysis of Encryption Algorithms

Objective

To evaluate the computational efficiency of different symmetric encryption algorithms by measuring the time required to process a 100MB file.

Methodology

1. **Test Data:** A 100MB file generated from `/dev/urandom`.
2. **Tool:** The Linux `time` command was used to measure the "real" (wall-clock) time.
3. **Conditions:**
   - For AES-256, we measured writing to disk vs. discarding output.
   - For the algorithm comparison, we directed output to `/dev/null` to measure pure CPU cryptographic speed without disk I/O bottlenecks.

Execution Log & Results

1. File Creation:

```
head -c 100M /dev/urandom > bigfile.bin
ls -lh bigfile.bin
# Output: 100M
```

2. Performance Measurements:

```
# AES-256-CBC Encryption (Writing to Disk)
time openssl enc -aes-256-cbc -pbkdf2 -pass pass:12345 -in bigfile.bin
    -out bigfile.enc
# Real Time: 15.915s

# AES-256-CBC Decryption (Discarding Output)
time openssl enc -aes-256-cbc -d -pbkdf2 -pass pass:12345 -in bigfile.
    enc -out /dev/null
# Real Time: 6.330s

# AES-128-CBC (Discarding Output)
time openssl enc -aes-128-cbc -pbkdf2 -pass pass:12345 -in bigfile.bin
    -out /dev/null
# Real Time: 2.588s

# DES-EDE3 (Triple DES)
time openssl enc -des-ede3 -pbkdf2 -pass pass:12345 -in bigfile.bin -
    out /dev/null
# Real Time: 7.645s

# ChaCha20 (Stream Cipher)
time openssl enc -chacha20 -pbkdf2 -pass pass:12345 -in bigfile.bin -
    out /dev/null
# Real Time: 2.360s
```

Algorithm	Output Destination	Time (Real)
AES-256-CBC (Encrypt)	Disk File	15.915s
AES-256-CBC (Decrypt)	/dev/null	6.330s
AES-128-CBC	/dev/null	2.588s
ChaCha20	/dev/null	**2.360s**
DES-EDE3 (3DES)	/dev/null	7.645s

Table 1: Performance Benchmark (100MB Input)

Comparative Analysis

Key Observations

1. **Disk I/O vs. CPU:** The initial AES-256 encryption took **15.9s** because it was writing 100MB to the disk. The decryption took only **6.3s** because the output was discarded (`/dev/null`). This highlights that disk speed is often the bottleneck, not the CPU.
2. **The Fastest (ChaCha20):** At **2.36s**, ChaCha20 was the fastest. It is a stream cipher designed for high performance in software without requiring special hardware acceleration.
3. **The Slowest (3DES):** At **7.64s**, Triple DES was significantly slower. This is expected as it applies the DES algorithm three times per block and has a smaller block size (64-bit), making it inefficient for large data.
4. **AES-128 vs AES-256:** AES-128 (2.58s) is faster than AES-256 (6.33s) because it performs fewer rounds of processing (10 rounds vs. 14 rounds).

Question 9: The Effect of Double Encryption

Objective

To investigate the result of applying the encryption command twice consecutively to a file, and to explain why this does not restore the original plaintext.

Execution Log

1. **Preparation:** We created a plaintext file.

```
echo "This is the original text." > q9_plain.txt
```

2. **First Encryption (Level 1):** We encrypted the plaintext to create the first ciphertext.

```
openssl enc -aes-256-cbc -pbkdf2 -in q9_plain.txt -out level1.enc
```

3. **Second Encryption (Level 2):** We ran the **same encryption command** on the resulting file.

```
openssl enc -aes-256-cbc -pbkdf2 -in level1.enc -out level2.enc
```

4. **Comparison:** We compared the doubly encrypted file with the original plaintext.

```
diff q9_plain.txt level2.enc
# Output: Binary files q9_plain.txt and level2.enc differ
```

5. **Visual Inspection:**

```
cat level2.enc
# Output: Salted__{... (Binary Garbage) ...
```

Theoretical Explanation

The `openssl enc` command performs a mathematical transformation, not a simple toggle (like a light switch).

- **Encryption Function (E):** transforms Plaintext (P) into Ciphertext (C).
- **Decryption Function (D):** transforms Ciphertext (C) back into Plaintext (P).

When we run the command twice, we perform:

$$C_2 = E_{key}(E_{key}(P))$$

This results in "Double Encryption" (a box inside a box). To retrieve the text, we would need to perform the inverse operation twice:

$$P = D_{key}(D_{key}(C_2))$$

Question 10: Algorithm Mode Mismatch

Objective

To observe the consequences of decrypting a file using the correct key and algorithm (AES-128) but the incorrect Mode of Operation (CBC vs. CFB).

Execution Log

1. **Preparation:** We created a plaintext file for the test.

```
echo "This file is for testing Mode mismatch." > plain.txt
```

2. **Encryption (CBC Mode):** We encrypted the file using **AES-128-CBC**.

```
openssl enc -aes-128-cbc -pbkdf2 -in plain.txt -out file_cbc.enc
```

3. **Decryption Attempt (CFB Mode):** We attempted to decrypt using **AES-128-CFB** with the **same password** (and thus the same derived key).

```
openssl enc -aes-128-cfb -d -pbkdf2 -in file_cbc.enc -out
   decrypted_wrong_mode.txt
```

Observation: The command finished without returning an error message.

4. **Verification:** We inspected the output.

```
cat decrypted_wrong_mode.txt
# Output: (Binary Garbage) gQe7pz ywhz...

diff plain.txt decrypted_wrong_mode.txt
# Output: Files differ significantly.
```

Analysis: Why did it produce garbage?

Even though the **Key** was correct, the **Mode of Operation** defines the mathematical path the data takes.

- **CBC (Cipher Block Chaining):** XORs the plaintext block with the previous ciphertext block *before* encryption.
- **CFB (Cipher Feedback):** Encrypts the previous ciphertext block and *then* XORs it with the plaintext.

Because the mathematical logic was different, the bits were scrambled differently. The lack of an error message indicates that standard symmetric modes (without authenticated encryption) do not inherently verify data integrity—they simply process bits.

Question 11: Automation with Shell Scripting

Objective

To create a Bourne Shell script named `myciphercde` that automates the encryption process. The script must:

1. Detect if the input is a **File** or a **Directory**.
2. If it is a Directory: Archive it using `tar` before encryption.
3. If it is a File: Encrypt it directly.

Script Implementation

File: `myciphercde.sh`

```
#!/bin/bash
TARGET=$1
KEY=$2
ALGO=$3

if [ -d "$TARGET" ]; then
    echo "Directory detected. Archiving..."
    tar -czf - "$TARGET" | openssl enc -"$ALGO" -salt -pbkdf2 -pass
       pass:"$KEY" -out "${TARGET}.tar.enc"
    echo "Created: ${TARGET}.tar.enc"

elif [ -f "$TARGET" ]; then
    echo "File detected."
    openssl enc -"$ALGO" -salt -pbkdf2 -pass pass:"$KEY" -in "$TARGET"
       -out "${TARGET%.*}.enc"
    echo "Created: ${TARGET%.*}.enc"
```

```
fi
```

Execution Log

1. **Test Case 1: Directory Encryption**

```
./myciphercde.sh mydir 12345 aes-256-cbc
# Output: Directory detected... Created: mydir.tar.enc
```

2. **Test Case 2: File Encryption (Extension Handling)**

```
# Input file: myfile.txt
./myciphercde.sh myfile.txt 12345 aes-256-cbc

# Output:
# File detected.
# Created: myfile.enc
```

3. **Verification:**

```
ls -l myfile.enc
```

Question 12: Authentication via Symmetric Encryption

Theoretical Concept

Symmetric encryption is primarily designed for *confidentiality*, but it can also be utilized for **Authentication**. The fundamental principle is based on the possession of the shared secret (the key/password).

- **Principle:** If a user can correctly decrypt a message that was encrypted with a specific key, they effectively prove that they possess that key.
- **Identity Verification:** Since the key is presumably known only to the specific user and the system, successful decryption serves as proof of identity.

The Authentication Workflow

To authenticate a user or process using a password (symmetric key), the following "Challenge-Response" mechanism is typically employed:

1. **The Challenge:** The system generates a known "token" (e.g., a random string or a fixed phrase like "ACCESS_CHECK").
2. **Encryption:** The system encrypts this token using the user's stored password (derived key).
3. **Decryption Attempt:** The user is asked to enter their password. The system attempts to decrypt the challenge using this input.
4. **Verification:**

- If the decrypted output matches the original token, the password is correct → **Authenticated**.
- If the output is garbage or the operation fails (bad decrypt), the password is wrong → **Access Denied**.

Command Line Analogy

In the context of our lab exercises, authentication is implicitly performed every time we run:

```
openssl enc -d -aes-256-cbc -in file.enc ...
```

If OpenSSL returns the original text without error, the user has been "authenticated" as the owner of the correct password.

Question 13: Brute Force Attack Analysis

Parameters

- **Computer Speed:** 3000 MIPS (3×10^9 instructions/sec).
- **Cost per Key:** 1000 instructions.
- **Key Testing Speed (V):**

$$V = \frac{3 \times 10^9}{1000} = 3 \times 10^6 \text{ keys/second}$$

a) Time Required for Brute Force (Single PC)

Assuming a worst-case scenario where we must test all possible keys.

Scenario 1: 40-bit Key ($2^{40} \approx 1.10 \times 10^{12}$ keys)

$$\begin{aligned} T_{40} &= \frac{2^{40}}{3 \times 10^6} \approx 366,504 \text{ seconds} \\ T_{40} &\approx \textbf{4.24 days} \end{aligned}$$

Scenario 2: 128-bit Key ($2^{128} \approx 3.40 \times 10^{38}$ keys)

$$\begin{aligned} T_{128} &= \frac{2^{128}}{3 \times 10^6} \approx 1.134 \times 10^{32} \text{ seconds} \\ T_{128} &\approx \textbf{3.59} \times 10^{24} \textbf{ years} \end{aligned}$$

b) PCs Required to Break in 3 Minutes (180 Seconds)

We need to parallelize the effort to finish in $t = 180$ seconds.

$$\text{PCs} = \frac{\text{Total Keys}}{V \times 180}$$

Scenario 1: 40-bit Key

$$\text{PCs}_{40} = \frac{2^{40}}{3 \times 10^6 \times 180} \approx \textbf{2,036 PCs}$$

Scenario 2: 128-bit Key

$$\text{PCs}_{128} = \frac{2^{128}}{3 \times 10^6 \times 180} \approx \textbf{6.30} \times 10^{29} \textbf{ PCs}$$

c) Moore's Law Prediction (128-bit Key)

Estimate the time until a single PC can break a 128-bit key in 3 minutes.

1. **Required Speedup Factor (F):**

$$F = \frac{1.134 \times 10^{32}}{180} \approx 6.30 \times 10^{29}$$

2. **Number of Doublings (n):**

$$2^n = F \Rightarrow n = \log_2(6.30 \times 10^{29}) \approx 99 \text{ doublings}$$

3. **Years Required (doubling every 1.5 years):**

$$\text{Years} = 99 \times 1.5 = \mathbf{148.5 \text{ years}}$$