IIT BHUBANESWAR

SECURITY AND FORENSICS LAB 1

ASSIGNMENT-8

NAME - CHANDAN KESHARI
ROLL NUMBER - 24CS06022
BRANCH - CSE(MTECH)

TEAM MEMBERS

NAME - SUBHAM SANKET ROUT
ROLL NUMBER - 24CS06017
BRANCH - CSE(MTECH)

NAME - NAVEEN KUMAR
ROLL NUMBER - 24CS06011
BRANCH - CSE(MTECH)

Questions

AES and Elliptic Curve Digital Signature Algorithm (ECDSA)

Implement a secure file transfer protocol using AES for encryption and Elliptic Curve Digital Signature Algorithm (ECDSA) for signing. Demonstrate tampering detection using ECDSA signatures and a brute force attack on a weak AES key to show/justify the importance of strong key policies. Find the details of the tasks below:

1. Implement a secure file transfer system that uses AES in CBC mode with a 128-bit key for file encryption, ensuring secure transmission between clients and the server.
2. Generate an ECDSA key pair (private and public) and sign the encrypted file with the sender’s private key. Send the signed file along with the public key for signature verification.
3. At the receiver's end, verify the authenticity and integrity of the received file using the sender's public key to validate the ECDSA signature.
4. Simulate a tampering attack by modifying the file during transmission. Implement mechanisms to detect tampering through ECDSA signature verification.
5. Conduct a brute-force attack on files encrypted with a weak AES key (e.g., a reduced 16-bit key) to demonstrate the ease of cracking weak encryption. Report the results of this attack.
6. Recommend stronger key policies to prevent such attacks. Discuss the importance of using strong AES keys (128-bit, 192-bit, or 256-bit). Explain how strong key policies enhance security.

Goal of Assignment:

The objective of this assignment is to implement and demonstrate a **secure file transfer system** using cryptographic techniques, focusing on **AES encryption, ECDSA signatures, tampering detection,** and a **brute-force attack simulation** on weak keys. The purpose is to highlight the importance of security policies, particularly around encryption keys and digital signatures, and to provide hands-on experience in:

1. **File Encryption and Secure Transmission:**
    - Using the **Advanced Encryption Standard (AES)** in **CBC** mode to ensure data confidentiality during file transfers between clients and servers.
    - CBC mode ensures that even if the same plaintext data is transmitted multiple times, the ciphertext remains unique by incorporating an **Initialization Vector (IV)**.
2. **Digital Signatures for Integrity and Authenticity:**
    - Implementing **Elliptic Curve Digital Signature Algorithm (ECDSA)** to **sign encrypted files** using the sender's private key.
    - Using **public key cryptography** to verify the integrity and authenticity of files at the receiver's end. This ensures that the data has not been tampered with during transmission.
3. **Detection of Tampering through ECDSA:**
    - Simulating a **tampering attack** by intentionally modifying the encrypted file.
    - Using **ECDSA signature verification** to detect unauthorized changes, ensuring that tampered files are identified and rejected.
4. **Brute-Force Attack on Weak AES Keys:**
    - Simulating a **brute-force attack** on AES encryption with a **weakened key (e.g., 16-bit)** to demonstrate how easily such keys can be broken.
    - This highlights the **importance of using strong encryption keys** (128-bit, 192-bit, or 256-bit AES keys) in real-world scenarios.
5. **Recommendations for Secure Key Management:**
    - Analyzing the brute-force attack results and recommending **stronger key policies** to prevent such attacks.
    - Exploring how **longer keys and secure policies** enhance cryptographic strength, protect against brute-force attacks, and ensure secure communication.

Understanding AES Encryption (128-bit) in CBC Mode

AES (Advanced Encryption Standard) is a symmetric encryption algorithm widely used for secure data encryption. It operates on blocks of 128 bits (16 bytes) and is considered highly secure when implemented correctly with appropriate key sizes (128, 192, or 256 bits).

When used in CBC (Cipher Block Chaining) mode, it enhances security by ensuring that the same plaintext encrypted multiple times produces different ciphertexts. Below is a detailed breakdown of how AES encryption with a 128-bit key in CBC mode works.

1. Key Components of AES in CBC Mode

- Key: A 128-bit (16-byte) key for both encryption and decryption.
- Block size: AES always encrypts data in 128-bit (16-byte) blocks.
- Initialization Vector (IV): A random 128-bit value used to ensure that encryption results are unique even if the same data is encrypted multiple times.
- Padding: If the plaintext is not a multiple of the block size (16 bytes), padding is added to make it align with the 16-byte boundary.

2. How CBC Mode Works

Encryption Process:

1. Split the plaintext into 128-bit (16-byte) blocks.
    a. Example: "Hello World!" (12 bytes) will need 4 extra bytes of padding to make it a full block.
2. Generate a random Initialization Vector (IV) for the first block.
    a. IV ensures that encryption will produce different ciphertexts for identical plaintexts.
3. Process each block iteratively:
    a. First block: XOR the plaintext block with the IV, then encrypt it using AES.
    b. Subsequent blocks: XOR each plaintext block with the previous ciphertext block before encrypting.
4. Combine all encrypted blocks to produce the final ciphertext.

Decryption Process:

- Extract the IV from the ciphertext.
- Decrypt each block using the AES decryption algorithm.

- XOR the decrypted block with the previous ciphertext block (or IV for the first block) to recover the original plaintext.
- Remove padding if added during encryption.

Elliptic Curve Digital Signature Algorithm (ECDSA)

The **Elliptic Curve Digital Signature Algorithm (ECDSA)** is a **public key cryptographic algorithm** used for generating and verifying digital signatures. It ensures both **data authenticity** and **integrity** by allowing the receiver to confirm that a message was indeed signed by the legitimate sender and hasn't been tampered with.

ECDSA is a variant of the **Digital Signature Algorithm (DSA)** that uses the mathematical structure of **elliptic curves** over finite fields to provide the same level of security with **smaller key sizes**, making it computationally more efficient.

Key Components of ECDSA

- **Private Key (d):**
  - A randomly generated large integer.
  - This is kept secret by the signer and used to generate signatures.
- **Public Key (Q):**
  - Derived from the private key and the elliptic curve's generator point GGG.
  - This is shared publicly and used for verifying signatures.
- **Elliptic Curve Parameters:**
  - Prime *p*: A large prime number that defines the field size.
  - Curve Equation: Typically of the form y2=x3+ax+by^2 = x^3 + ax + by2=x3+ax+b.
  - Generator Point *G*: A predefined point on the curve used for public key generation.
  - Order *n*: A large integer representing the number of points on the curve (used to generate random values).
- **Hash Function:**
  - A cryptographic hash function (e.g., SHA-256) is used to convert the input message to a fixed-length hash value for signing.

How ECDSA Works – Step by Step

1. Key Generation

The signer generates:

1. Private Key (d): A random integer in the range [1,n−1], where n is the order of the curve.
2. Public Key (Q): $Q = d \cdot G$

   (where G is the generator point on the curve, and · represents scalar multiplication).

The private key is used to sign messages, and the public key is shared with others to verify signatures.

2. Signing a Message

To generate a digital signature for a message:

1. Hash the message:
2. Compute the hash of the message H(m) using a cryptographic hash function like SHA-256.
3. Generate a random integer k in the range [1,n−1]. This value must be unique for each signature.
4. Calculate the point $(x1, y1) = k \cdot G$ on the elliptic curve.
5. Compute the signature components:
   - r:

     $r = x1 \bmod n$

     If r=0, generate a new random k.
   - s:

     $s = k^{(-1)} \cdot (H(m) + d \cdot r) \bmod n$

     (where k^(-1) is the modular inverse of k).

1. Output the signature: The final signature is the pair (r, s).

3. Verifying the Signature

The receiver verifies the authenticity of the message by using the sender's public key:

1. Hash the message:

   Compute the hash H(m)H(m)H(m) of the received message.
2. Verify the signature components (r, s):

   Ensure r and s are within the range [1, n−1]. If not, the signature is invalid.
3. Calculate intermediate values:

   Compute $w = s^{(-1)} \bmod n$ (the modular inverse of s).

   Compute: $u1 = H(m) \cdot w \bmod n$ and $u2 = r.w \bmod n$
4. Calculate the point $(x1, y1) = u1 \cdot G + u2 \cdot Q$ on the curve.
5. Verify:

   If $r = x1 \bmod n$, the signature is valid.

   If not, the signature is invalid (indicating tampering or forgery).

Explanation of the Code for Questions 1, 2 and 3:

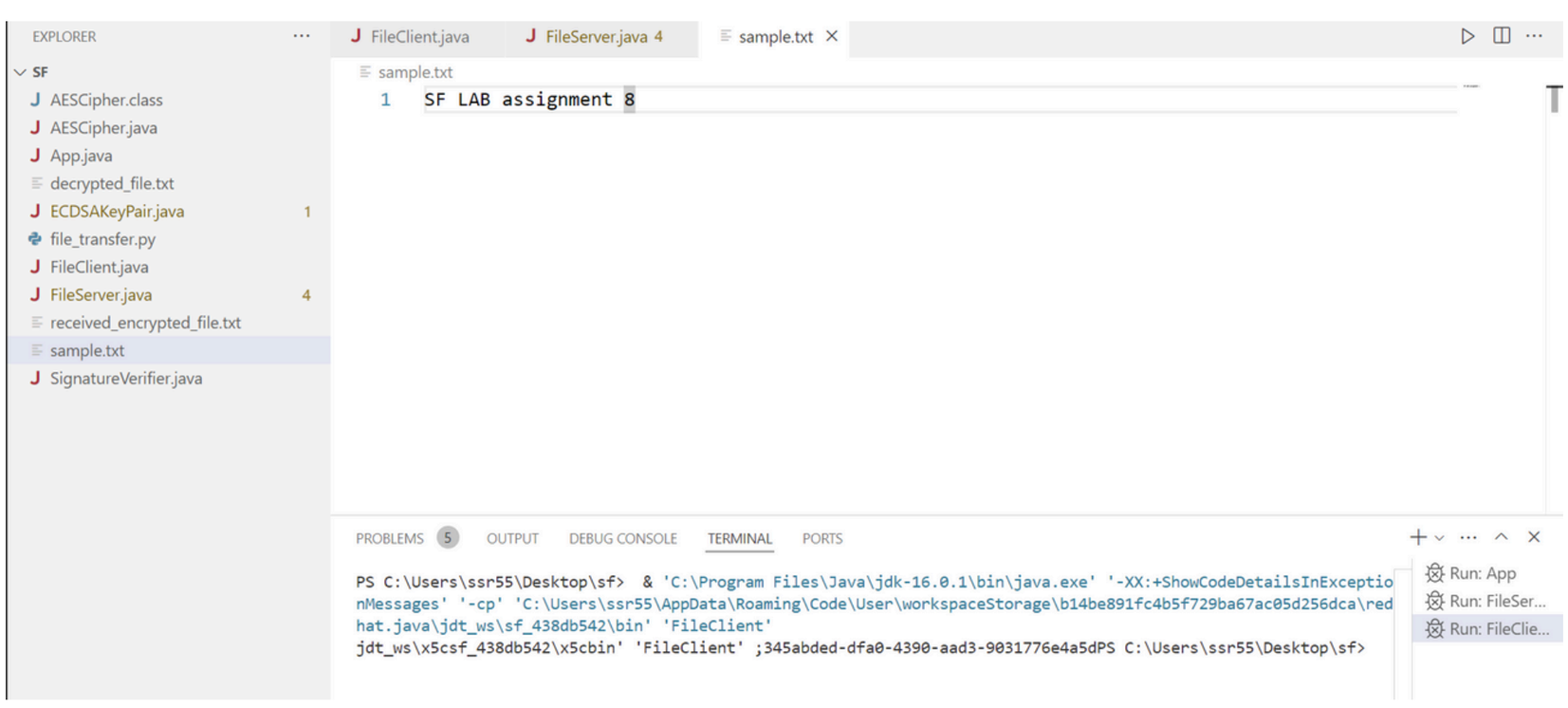
FileClient.java code explanation

- First step is to generate the AES key and IV, and then encrypt the file to be sent to the other host (Server).
- Then generate the ECDSA key pair and sign the encrypted file.
- Start the socket connection.
- In the next step, send the encrypted file data to the other host (Server).
- Then send the ECDSA signature to the other host (Server).
- After that send public key to the other side (Server) for verification.
- At last, send AES key and IV (converted to byte arrays) to the other host (Server).

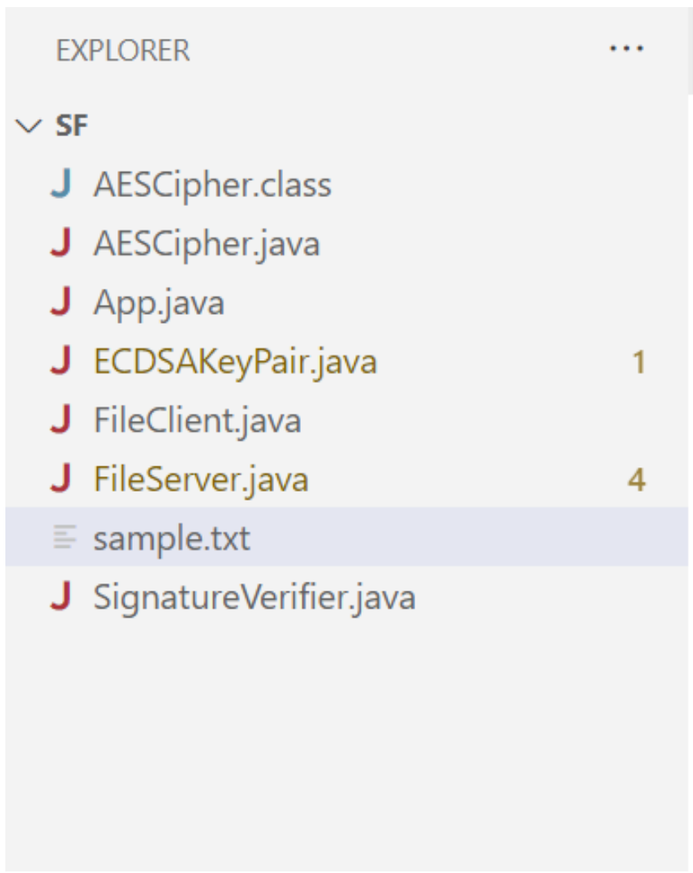
FileServer.java code explanation

- Create a socket and start the connection.
- Listen for any active connections from client side and accept it to get connected with it.
- Receive the encrypted file data from the other host (Client).
- Receive the signature from the other host (Client).
- Receive the public key for verification too.
- Receive the AES key from the other host (Client).
- Receive the IV as well.
- Verify the signature to ensure authenticity
- Convert AES key and IV from byte arrays to usable objects (They will be used to decrypt the file data).
- Then, decrypt the encrypted file data.
- Lastly, save decrypted data to a file.

Results of the experiment:



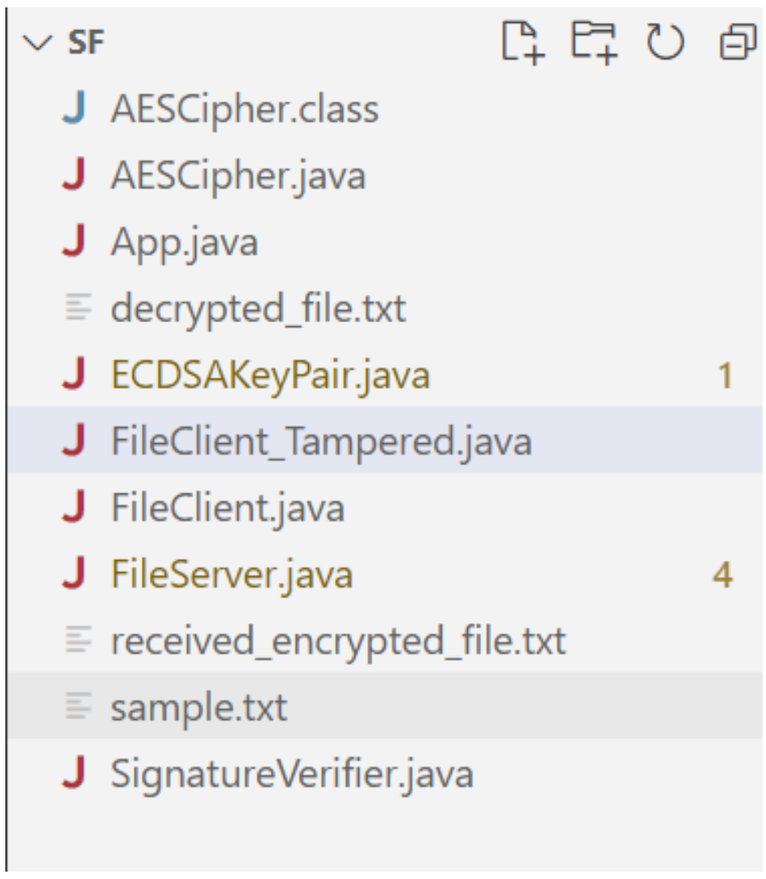
Create a text file and insert some characters into it and name it as “Sample.txt”.



Initially, we will run the file ‘App.java’, ‘AESCipher.java’ and ‘ECDSAKeyPair.java’ files. (they can be seen in the attached file snapshot of the file explorer)

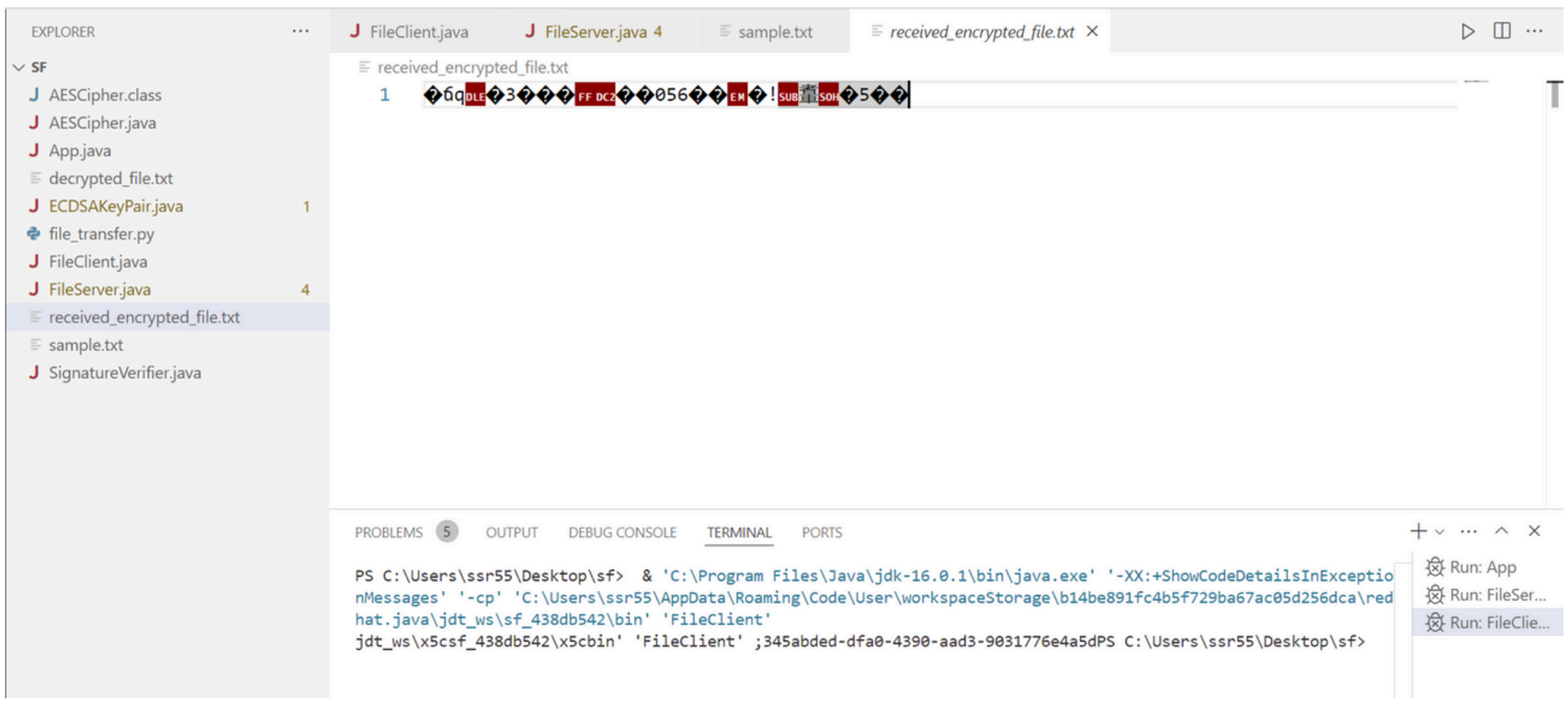
After that we’ll run the following files in the given order.

1. FileServer.java
2. FileClient.java

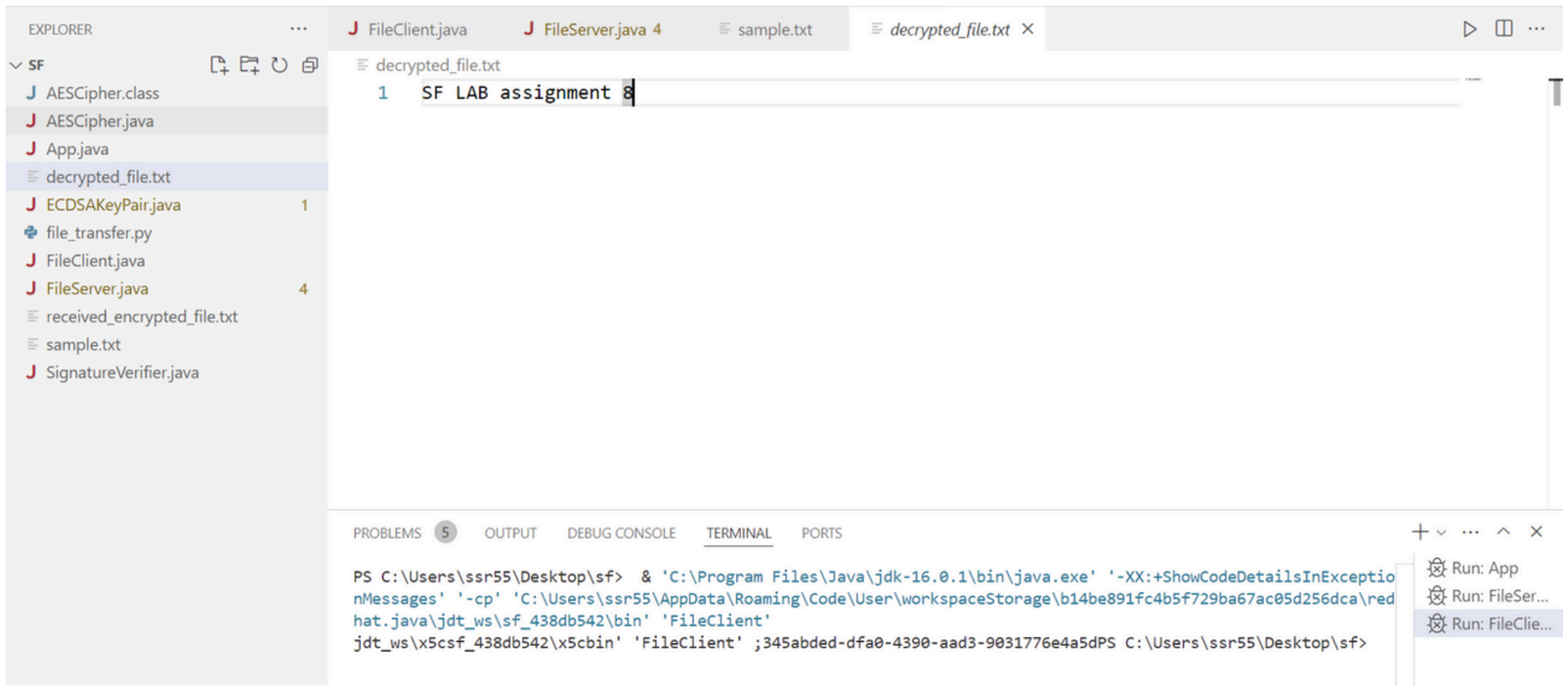


We see that two new files got created named as “received_encrypted_file.txt” and “decrypted_file.txt”.

When we open the “received_encrypted_file.txt” file, we can see the encrypted text.



When we open the “decrypted_file.txt” file, we can see the original plaintext that we had inserted in the file “Sample.txt”.



Log File for Questions 1,2, & 3

Here is the log of the file transfers showing the encryption, decryption and signature verification.

log.txt

```
[2024-10-27 20:14:57] Server : Server started and waiting for client connections...
[2024-10-27 20:15:11] Client : AES key generated and used for encryption: DC57D23268A2E5205236DBCBBD21D62B
[2024-10-27 20:15:12] Client : ECDSA signature generated for the encrypted file: 3044022006A47B9C63CD4B255B6BD96F0608
[2024-10-27 20:15:12] Server : Client connected.
[2024-10-27 20:15:12] Client : Encrypted file data sent to server.
[2024-10-27 20:15:12] Client : ECDSA signature sent to server.
[2024-10-27 20:15:12] Server : Encrypted file data received from client.
[2024-10-27 20:15:12] Client : ECDSA public key sent to server: 3059301306072A8648CE3D020106082A8648CE3D0301070342000
[2024-10-27 20:15:12] Client : AES key and IV sent to server.
[2024-10-27 20:15:12] Client : Connection closed.
[2024-10-27 20:15:12] Server : ECDSA signature received from client: 3044022006A47B9C63CD4B255B6BD96F060836550A0667C4
[2024-10-27 20:15:12] Server : ECDSA public key received from client: 3059301306072A8648CE3D020106082A8648CE3D0301070
[2024-10-27 20:15:12] Server : AES key received from client: DC57D23268A2E5205236DBCBBD21D62B
[2024-10-27 20:15:12] Server : AES IV received from client: EED6C0675303BFD896ECAC33AF304470
[2024-10-27 20:15:12] Server : File verification successful. The file is authentic.
[2024-10-27 20:15:12] Server : Decryption successful.
[2024-10-27 20:15:12] Server : Decrypted file saved as 'decrypted_file.txt'.
[2024-10-27 20:15:12] Server : Client connection closed.
```

Explanation of the Code for Question 4:

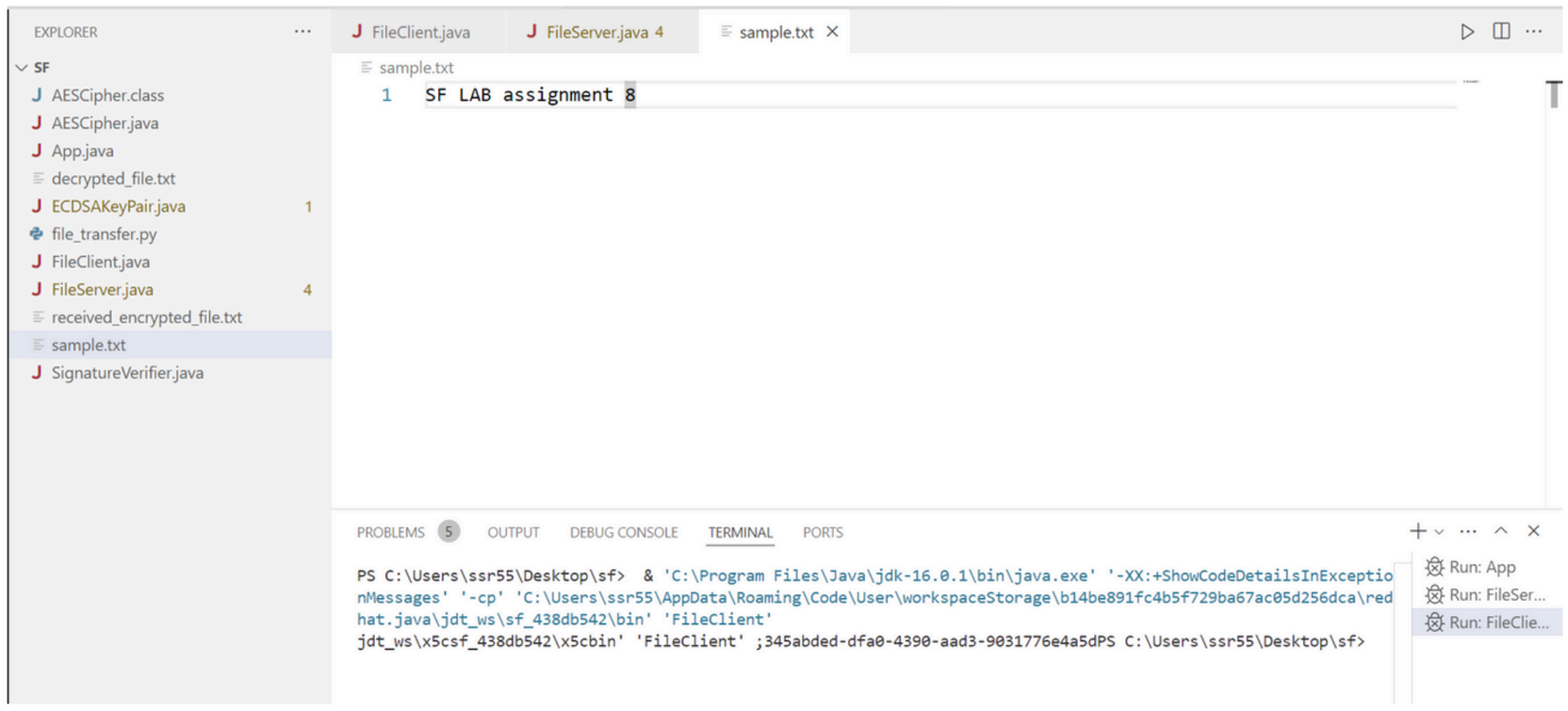
FileClient.java code explanation

- First step is to generate the AES key and IV, and then encrypt the file to be sent to the other host (Server).
- Then generate the ECDSA key pair and sign the encrypted file.
- Start the socket connection.
- In the next step, send the encrypted file data to the other host (Server). (Here, we tampered the encrypted file by flipping a bit)
- Then send the ECDSA signature to the other host (Server).
- After that send public key to the other side (Server) for verification.
- At last, send AES key and IV (converted to byte arrays) to the other host (Server).

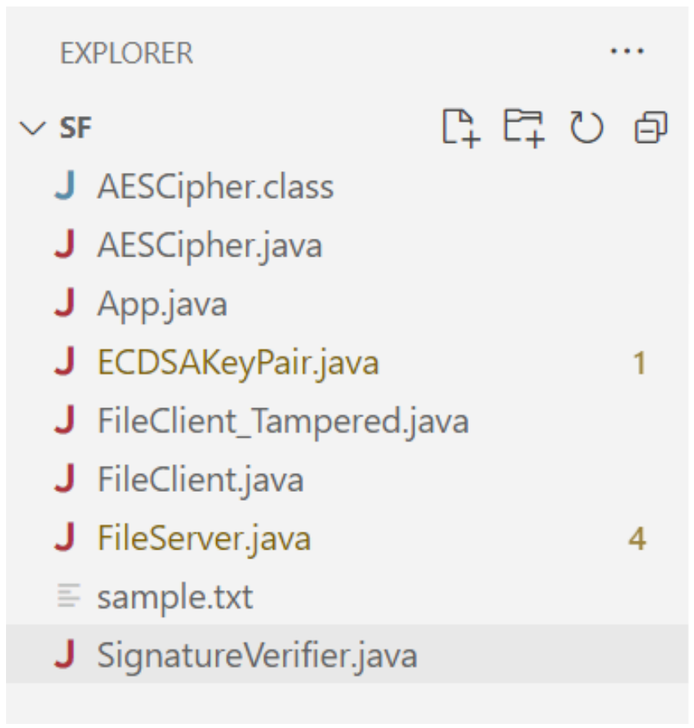
FileServer.java code explanation (Same as above question)

- Create a socket and start the connection.
- Listen for any active connections from client side and accept it to get connected with it.
- Receive the encrypted file data from the other host (Client).
- Receive the signature from the other host (Client).
- Receive the public key for verification too.
- Receive the AES key from the other host (Client).
- Receive the IV as well.
- Verify the signature to ensure authenticity
- Convert AES key and IV from byte arrays to usable objects (They will be used to decrypt the file data).
- Then, decrypt the encrypted file data.
- Lastly, save decrypted data to a file.

Results of the experiment:



Create a text file and insert some characters into it and name it as “Sample.txt”.

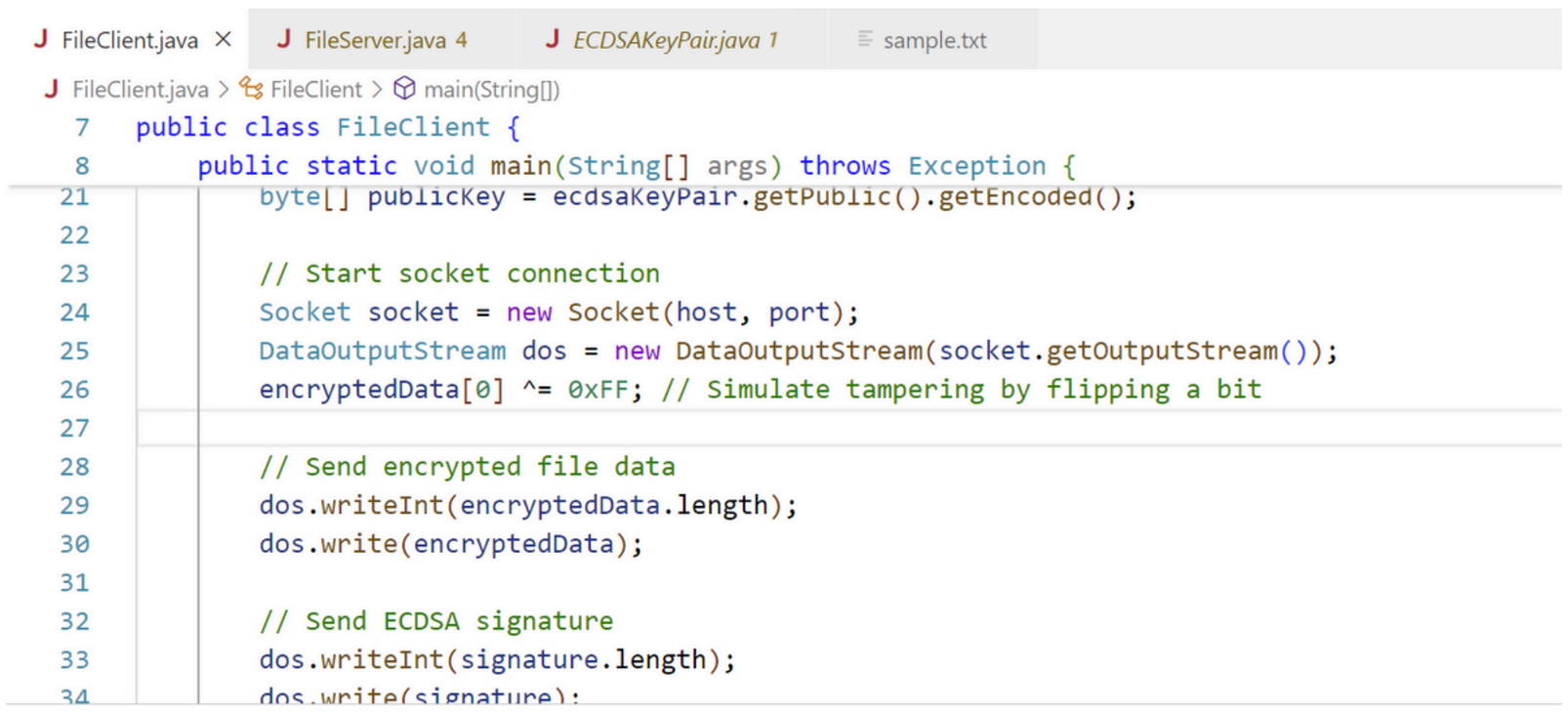


Initially, we will run the file ‘App.java’, ‘AESCipher.java’ and ‘ECDSAKeyPair.java’ files. (they can be seen in the attached file snapshot of the file explorer)

After that we’ll run the following files in the given order.

1. FileServer.java
2. FileClient_Tampered.java

Unlikely, as seen in the above case, those two new files didn’t get created i.e. “received_encrypted_file.txt” and “decrypted_file.txt” because this time, the encrypted file that we had sent to the server was tampered as shown below…



And this was detected successfully by our server as shown below.

Since the encrypted file was tampered, it couldn't be verified and decrypted back to its plaintext format.

Log File for Question 4

Here is the log of tampering detection using ECDSA signatures

```
[2024-10-27 20:43:52] Server : Server started and waiting for client connections...
[2024-10-27 20:44:38] Client : AES key generated and used for encryption: 2D45FF6A9D5B177DA691E8D87FA8B327
[2024-10-27 20:44:38] Client : ECDSA signature generated for the encrypted file: 3045022100AA2EBA07D260D23024EFF7A95E
[2024-10-27 20:44:38] Server : Client connected.
[2024-10-27 20:44:38] Client : Encrypted file data sent to server.
[2024-10-27 20:44:38] Client : ECDSA signature sent to server.
[2024-10-27 20:44:38] Server : Encrypted file data received from client.
[2024-10-27 20:44:38] Client : ECDSA public key sent to server: 3059301306072A8648CE3D020106082A8648CE3D0301070342000
[2024-10-27 20:44:38] Client : AES key and IV sent to server.
[2024-10-27 20:44:38] Client : Connection closed.
[2024-10-27 20:44:38] Server : ECDSA signature received from client: 3045022100AA2EBA07D260D23024EFF7A95E391594EEB9B5
[2024-10-27 20:44:38] Server : ECDSA public key received from client: 3059301306072A8648CE3D020106082A8648CE3D0301070
[2024-10-27 20:44:38] Server : AES key received from client: 2D45FF6A9D5B177DA691E8D87FA8B327
[2024-10-27 20:44:38] Server : AES IV received from client: D135A77873F9B38118C575C6D0155A81
[2024-10-27 20:44:38] Server : File tampering detected. Signature verification failed.
[2024-10-27 20:44:38] Server : Client connection closed.

```

Step-by-Step Brute-Force Attack on AES (Question 5)

- Encrypt a file with a 16-bit key (padded to 128 bits as required by AES).
- Perform the brute-force attack by trying all possible 16-bit keys (total: 65,536 keys).
- Verify success when the correct key decrypts the file without error.

- **Step 1: Prepare an AES-CBC Encrypted File with a Weak Key**
    - This encryption code simulates the weak-key encryption. It uses a 16-bit key (2 bytes), padded to 128 bits.
    - Save this file as “AESWeakEncryption.java”.
    - Create a text file named example.txt with some text i.e. "Hello, AES Encryption!".
    - Compile and run the code to generate the encrypted file, “encrypted_file.aes”.
- **Step 2: Perform the Brute-Force Attack on the Encrypted File**
    - The brute-force code attempts all 65,536 possible keys (2^16 keys). When the correct key is found, the decrypted output will match the known content of the original file.
    - Save this file as “BruteForceAES.java”.
    - Compile and run the code to perform the brute-force attack.
- **Step 3: Run the Brute-Force Attack**
    - Compile both Java files:

      javac AESWeakEncryption.java BruteForceAES.java
    - Run the encryption:

      java AESWeakEncryption

  This will generate encrypted_file.aes.
    - Run the brute-force attack:

      java BruteForceAES

  This will try all 65,536 keys and print the decrypted text when the correct key is found.
- **Step 4: Results of the Brute-Force Attack**
    - Output:

      Decrypted Text: Hello, AES Encryption!
    - Time Taken:

  On modern hardware, trying all 65,536 keys took only a few seconds.
    - Analysis:

This demonstrates the ease of breaking weak keys like 16-bit keys. The attack completed in seconds, showing that weak keys offer no real security.

Recommendations for Strong Key Policies (Question 6)

- **Use at least 128-bit keys:**
  - AES with 128-bit, 192-bit, or 256-bit keys ensures sufficient protection against brute-force attacks. Even with the fastest supercomputers, breaking a 128-bit key would take billions of years.
- **Rotate keys periodically:**
  - Regular key rotation limits the exposure of keys in case of compromise.
- **Use Secure Key Management Systems:**
  - Store encryption keys in dedicated hardware security modules (HSMs) or key management services (like AWS KMS, Azure Key Vault) to prevent unauthorized access.
- **Avoid hardcoding keys:**
  - Never store keys directly in code. Instead, retrieve them securely from an external source when needed.
- **Use a strong random number generator:**
  - IVs and keys should be generated using a cryptographically secure random number generator to prevent predictability.

Importance of Strong AES Keys

- **Increased Key Space:**
  - A 128-bit key provides 2^(128) possible keys, making brute force infeasible. In comparison, 16-bit keys are trivially small.
- **Future-proof Security:**
  - Stronger keys (192-bit or 256-bit) offer additional security in case quantum computers become practical.
- **Regulatory Compliance:**
  - Many regulations (e.g., GDPR, PCI-DSS) mandate strong encryption policies to protect sensitive data.

Conclusion:

This experiment demonstrates the vulnerability of weak encryption, highlighting the importance of strong keys and sound encryption policies. Using AES with at least 128-bit keys ensures that brute-force attacks are computationally impractical, securing data effectively against malicious actors.