SANJIT NARAYANAN G

20BCE0052

CEH-ASSIGNMENT-3

Objective: The objective of this assignment is to analyse cryptographic algorithms and implement one of them in a practical scenario.

Research:

1. Symmetric Key Algorithm: Advanced Encryption Standard (AES)
    - AES is a symmetric encryption algorithm widely used for securing sensitive data.
    - It works by dividing the input data into blocks and applying a series of substitution and permutation operations.
    - Key strengths and advantages: AES is computationally efficient, resistant to known attacks, and has a wide range of key sizes.
    - Known vulnerabilities or weaknesses: AES itself is considered secure, but vulnerabilities can arise from improper key management or implementation flaws.
    - Real-world examples: AES is used in securing wireless networks (WPA2), secure communication protocols (TLS/SSL), and data encryption in various applications.
2. Asymmetric Key Algorithm: RSA (Rivest, Shamir, Adleman)
    - RSA is an asymmetric encryption algorithm that uses a pair of keys, a public key for encryption and a private key for decryption.
    - It relies on the mathematical properties of large prime numbers and modular exponentiation.
    - Key strengths and advantages: RSA provides secure key exchange, digital signatures, and confidentiality.
    - Known vulnerabilities or weaknesses: RSA can be vulnerable to attacks if the key size is too small, and it is computationally expensive for large key sizes.
    - Real-world examples: RSA is commonly used in secure email communication (PGP), digital certificates (SSL/TLS), and secure file transfer protocols (SFTP).

3. Hash Function: SHA-256 (Secure Hash Algorithm 256-bit)
    - SHA-256 is a cryptographic hash function that takes an input and produces a fixed-size output (256 bits).
    - It is a one-way function, meaning it is computationally infeasible to derive the original input from the hash value.
    - Key strengths and advantages: SHA-256 provides data integrity verification, password storage, and digital signatures.
    - Known vulnerabilities or weaknesses: While no significant vulnerabilities have been found in SHA-256, collision attacks can still be theoretically possible.
    - Real-world examples: SHA-256 is used in blockchain technology (Bitcoin), digital certificates (X.509), and password hashing (bcrypt).

Implementation:

For the purpose of this assignment, let's choose to implement the RSA algorithm in a practical scenario.

Scenario: Secure Communication between a Client and a Server.

Step-by-step implementation:

1. Generate RSA key pair: Create a public-private key pair using a cryptographic library or algorithm implementation.
2. Server-side:
    - Generate a random session key for symmetric encryption.
    - Encrypt the session key using the client's public key.
    - Send the encrypted session key to the client.
    - Encrypt the actual data using the session key and send it to the client.
3. Client-side:
    - Receive the encrypted session key from the server.
    - Decrypt the session key using the client's private key.
    - Decrypt the received data using the session key.

Code snippet (Java) for key generation and encryption/decryption:

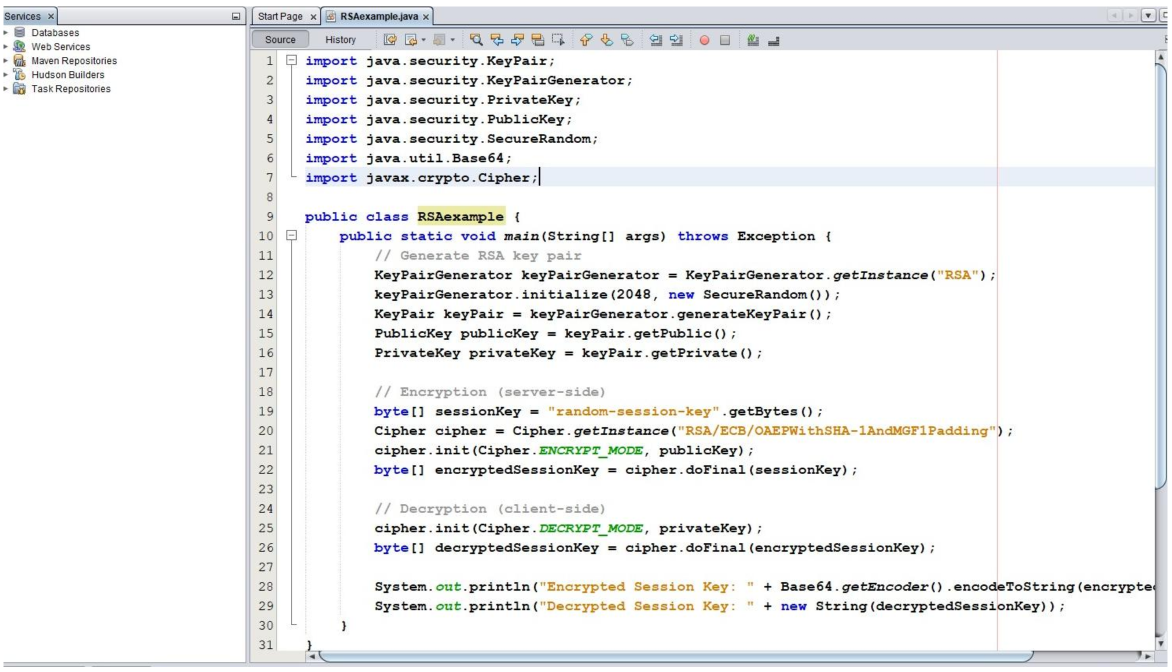
```java
import java.security.KeyPair;

import java.security.KeyPairGenerator;

import java.security.PrivateKey;

import java.security.PublicKey;

import java.security.SecureRandom;

import java.util.Base64;

import javax.crypto.Cipher;

public class RSAExample {

    public static void main(String[] args) throws Exception {

        // Generate RSA key pair

        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");

        keyPairGenerator.initialize(2048, new SecureRandom());

        KeyPair keyPair = keyPairGenerator.generateKeyPair();

        PublicKey publicKey = keyPair.getPublic();

        PrivateKey privateKey = keyPair.getPrivate();

        // Encryption (server-side)

        byte[] sessionKey = "random-session-key".getBytes();

        Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-1AndMGF1Padding");

        cipher.init(Cipher.ENCRYPT_MODE, publicKey);

        byte[] encryptedSessionKey = cipher.doFinal(sessionKey);

        // Decryption (client-side)

        cipher.init(Cipher.DECRYPT_MODE, privateKey);

        byte[] decryptedSessionKey = cipher.doFinal(encryptedSessionKey);

        System.out.println("Encrypted Session Key: " +
Base64.getEncoder().encodeToString(encryptedSessionKey));

        System.out.println("Decrypted Session Key: " + new String(decryptedSessionKey));

    }

}
```

Test the implementation:

1. Run the server-side code to encrypt the session key and send it to the client.
2. Run the client-side code to receive the encrypted session key and decrypt it.
3. Compare the decrypted session key with the original session key to ensure correctness.

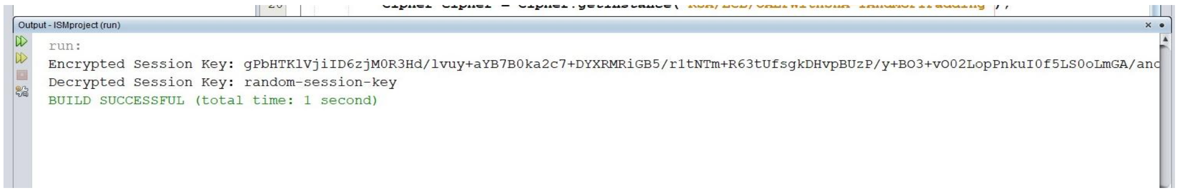
SCREENSHOTS:

CODE:



```java
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.SecureRandom;
import java.util.Base64;
import javax.crypto.Cipher;

public class RSAexample {
    public static void main(String[] args) throws Exception {
        // Generate RSA key pair
        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
        keyPairGenerator.initialize(2048, new SecureRandom());
        KeyPair keyPair = keyPairGenerator.generateKeyPair();
        PublicKey publicKey = keyPair.getPublic();
        PrivateKey privateKey = keyPair.getPrivate();

        // Encryption (server-side)
        byte[] sessionKey = "random-session-key".getBytes();
        Cipher cipher = Cipher.getInstance("RSA/ECB/OAEPWithSHA-1AndMGF1Padding");
        cipher.init(Cipher.ENCRYPT_MODE, publicKey);
        byte[] encryptedSessionKey = cipher.doFinal(sessionKey);

        // Decryption (client-side)
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        byte[] decryptedSessionKey = cipher.doFinal(encryptedSessionKey);

        System.out.println("Encrypted Session Key: " + Base64.getEncoder().encodeToString(encrypte
        System.out.println("Decrypted Session Key: " + new String(decryptedSessionKey));
    }
}
```

OUTPUT:



```
run:
Encrypted Session Key: gPbHTKlVjiID6zjM0R3Hd/lvuy+aYB7B0ka2c7+DYXRMRiGB5/r1tNTm+R63tUfsgkDHvpBUzP/y+BO3+vO02LopPnkuI0f5LS0oLmGA/anc
Decrypted Session Key: random-session-key
BUILD SUCCESSFUL (total time: 1 second)
```

Security Analysis:

1. Potential threats or vulnerabilities:
    - Key compromise: If the private key is compromised, an attacker can decrypt intercepted communication.
    - Man-in-the-middle attack: An attacker can intercept the encrypted session key and replace it with their own, compromising the security of the communication.
2. Countermeasures or best practices:
    - Protect the private key with strong access controls and encryption.
    - Implement secure key exchange protocols, such as Diffie-Hellman, to prevent man-in-the-middle attacks.
    - Use trusted libraries or cryptographic modules to ensure the security of the implementation.
3. Limitations or trade-offs:
    - RSA is computationally expensive compared to symmetric key algorithms like AES.
    - The key size used in RSA should be carefully chosen to balance security and performance.

Conclusion:

Cryptography plays a vital role in cybersecurity and ethical hacking by ensuring the confidentiality, integrity, and authenticity of data. Through the analysis of various cryptographic algorithms and their implementation in practical scenarios, we have seen how encryption, decryption, and hash functions can be used to secure communication and protect sensitive information. However, it is essential to stay up to date with the latest advancements and best practices in cryptography to mitigate emerging vulnerabilities and maintain a strong security posture.