Nome:**Dhulkifli Amisse Malique**

1. Analisamdo o código **`PKCS1SignatureExample.java`**

a) O código demonstra como fazer uma assinatura digital usando o algoritmo RSA com uma função de síntese **SHA-1**. A função de síntese subjacente à assinatura é SHA-1.Aqui **SHA1withRSA** indica que o algoritmo de assinatura utiliza **SHA-1** como a função de hash subjacente.
Se usarmos **SHA-256** em vez de SHA-1 aumenta a segurança da assinatura, pois SHA-256 é considerado mais seguro e resistente a colisões.

b) O tamanho das assinaturas geradas é determinado pelo tamanho da chave RSA usada. No código, a chave é gerada com um tamanho de 512 bits.

Isso significa que as assinaturas terão 512 bits de tamanho. O tamanho da assinatura é diretamente proporcional ao tamanho da chave utilizada. Para aumentar a segurança, é recomendável usar tamanhos de chave maiores, como 2048 bits ou 4096 bits.

c) O valor da assinatura pode ser diferente em diferentes corridas do programa devido à aleatoriedade introduzida na geração da chave e na função de hash. A função **createFixedRandom()** usada no código fornece um **SecureRandom** fixo, o que torna a geração de chaves determinística. No entanto, a função de hash SHA-1 ainda é influenciada pela mensagem a ser assinada. Portanto, Se assinar mensagens diferentes, as assinaturas serão diferentes.

Para garantir que as assinaturas sejam idênticas em diferentes corridas do programa, você precisaria usar a mesma mensagem e a mesma chave de assinatura.

d) Para modificar o código a fim de produzir e verificar uma assinatura digital usando o algoritmo DSA e gerar chaves de diferentes tamanhos,podemos alterar:

- Altere a instância da classe `KeyPairGenerator` para usar o algoritmo DSA:

  KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "BC");

- Alterar a função de hash da assinatura para corresponder ao DSA:

  Signature signature = Signature.getInstance("SHA1withDSA", "BC");

- Podemos ajustar o tamanho da chave DSA alterando o parâmetro `keyGen.initialize`. Por exemplo, para gerar uma chave DSA de 1024 bits:

  keyGen.initialize(1024, new SecureRandom());

2.1)O código **AESWrapRSAExample** demonstra como proteger uma chave privada RSA com uma chave AES. Essa técnica de proteção de chaves privadas é conhecida como **key wrapping**.

Proteção de Chaves Privadas DSA Para aptar o código para proteger chaves privadas DSA, Alterando a geração da chave para uma chave DSA e Alterar a chave usada para o envolvimento (wrapping) para uma chave AES,

KeyPairGenerator fact = KeyPairGenerator.getInstance("DSA", "BC");

Key wrapKey = Utils3.createKeyForAES(256, random);

Vantagens da Técnica:

Proteção Adicional: A técnica de wrapping permite proteger as chaves privadas, tornando-as menos vulneráveis a acessos não autorizados. A chave privada é criptografada e só pode ser descriptografada com a chave de descriptografia correta.

Melhor Desempenho: As operações de criptografia simétrica, como o AES, tendem a ser mais rápidas do que as operações de criptografia assimétrica, como o RSA ou DSA. Isso significa que o acesso à chave privada protegida é mais rápido do que se a chave fosse mantida em texto simples.

Flexibilidade:Essa técnica pode ser usada para proteger várias chaves privadas e é independente do algoritmo de chave pública (RSA, DSA, etc.) sendo usado. Você pode proteger qualquer chave privada que desejar.

2.2.O código **RSAKeyExchangeExample** mostra como criar envelopes de chave pública para distribuição e estabelecimento de chaves de sessão para garantir a confidencialidade em um canal de comunicação.

1. O código gera uma chave simétrica AES sKey e um vetor de inicialização (IV) AES sIvSpec para serem usados para criptografar os dados de sessão.

2. O envelope da chave simétrica é criado da seguinte maneira:

- O envelope é composto pelo IV AES seguido dos bytes da chave simétrica AES.

- O envelope é criptografado usando a chave pública RSA. Para isso, o código utiliza a classe `Cipher` com o modo de operação OAEP (Optimal Asymmetric Encryption Padding) e preenchimento com SHA-1 e MGF1 (Mask Generation Function 1).

3. Os dados de sessão são criptografados usando a chave simétrica AES e o IV.

4. O código demonstra a descriptografia dos dados de sessão e a recuperação da chave simétrica usando a chave privada RSA. Ele segue os seguintes passos:

- A chave privada RSA (`privKey`) é usada para descriptografar o envelope da chave simétrica.

- O envelope descriptografado é dividido em IV e chave simétrica.

- A chave simétrica e o IV são usados para descriptografar os dados de sessão.

3.1 TwoWayDHExample é um exemplo que demonstra um acordo de chave entre duas partes (Two-Way) usando o algoritmo de Diffie-Hellman

1. Definindo os Parâmetros de Diffie-Hellman:

- **g512** e **p512**: São os parâmetros de domínio Diffie-Hellman, onde **g512** é o gerador e **p512** é um grande número primo. Esses parâmetros são públicos e usados por ambas as partes para o acordo de chave.

2. Configurando o Gerador de Par de Chaves Diffie-Hellman:

- **DHParameterSpec dhParams = new DHParameterSpec(p512, g512):** Aqui, os parâmetros **p512** e **g512** são usados para configurar o gerador de par de chaves Diffie-Hellman.

3. Geração de Chaves:

- **KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DH", "BC"):** É criado um gerador de pares de chaves para o algoritmo Diffie-Hellman usando o provedor Bouncy Castle (BC).

- **keyGen.initialize(dhParams, UtilsDH.createFixedRandom()):** Inicializa o gerador de chaves com os parâmetros Diffie-Hellman e um gerador de números aleatórios fixo (isso não é seguro na prática, é apenas para fins de demonstração).

4. Configurando o Acordo de Chave para Ambas as Partes:

- **KeyAgreement aKeyAgree = KeyAgreement.getInstance("DH", "BC")`:** Cria uma instância para A realizar o acordo de chave.

- **KeyAgreement bKeyAgree = KeyAgreement.getInstance("DH", "BC**"): Cria uma instância para B realizar o acordo de chave.

5. Geração de Par de Chaves para A e B:

- K**eyPair aPair = keyGen.generateKeyPair():** Gera um par de chaves (pública e privada) para A.

- **KeyPair bPair = keyGen.generateKeyPair():** Gera um par de chaves (pública e privada) para B.

6. Inicialização do Acordo de Chave:

- **aKeyAgree.init(aPair.getPrivate()):** Inicializa o acordo de chave de A com sua chave privada.

- **bKeyAgree.init(bPair.getPrivate()):** Inicializa o acordo de chave de B com sua chave privada.

7. Realização do Acordo de Chave:

- **aKeyAgree.doPhase(bPair.getPublic(), true):** A realiza o acordo de chave com a chave pública de B e especifica que é a última fase.

- **bKeyAgree.doPhase(aPair.getPublic(), true):** B realiza o acordo de chave com a chave pública de A e especifica que é a última fase..

3.2. **ThreeWayDHExample.java** este exemplo é uma extensão do anterior, envolvendo três partes, A, B e C, em um acordo de chave Diffie-Hellman.

1. Definição dos Parâmetros de Diffie-Hellman:

- Os parâmetros **g512** e **p512** definem os parâmetros de domínio Diffie-Hellman, assim como no exemplo Two-Way. São os mesmos valores públicos que todas as partes utilizam.

2. Configurando o Gerador de Par de Chaves Diffie-Hellman:

- O código utiliza o mesmo gerador de pares de chaves **keyGen** e os mesmos parâmetros Diffie-Hellman **dhParams** do exemplo Two-Way.

3. Geração de Pares de Chaves para as Três Partes:

- Três pares de chaves são gerados, um para cada uma das três partes: A, B e C. Isso é feito com **keyGen.generateKeyPair()**

4. Inicialização dos Acordos de Chave:

- O código cria três instâncias de **KeyAgreement**: **aKeyAgree, bKeyAgree** e **cKeyAgree**, uma para cada parte.

- Cada uma das três partes inicializa seu acordo de chave com sua respectiva chave privada, ou seja, **aKeyAgree.init(aPair.getPrivate()), bKeyAgree.init(bPair.getPrivate())** e **cKeyAgree.init(cPair.getPrivate()).**

5. Realização do Acordo de Chave entre as Três Partes:

- Cada parte realiza uma série de **doPhase** para trocar informações do acordo de chave com as outras partes. O objetivo é que cada parte termine com uma chave compartilhada com as outras duas.

- O cálculo das chaves compartilhadas ocorre por meio das chamadas do**Phase.**

6. Derivação das Chaves Compartilhadas:

- Após a realização dos acordos de chave, cada parte gera sua chave compartilhada chamando **generateSecret()** no objeto **KeyAgreement.**

- As chaves compartilhadas resultantes para A, B e C são **aShared**, **bShared** e **cShared**, respectivamente.

3.3

```
import java.math.BigInteger;
import java.security.*;
import java.security.spec.X509EncodedKeySpec;
import java.security.interfaces ECPublicKey;
import java.security.interfaces ECPrivateKey;
import javax.crypto.KeyAgreement;
import javax.crypto.spec.DHParameterSpec;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.spec.PKCS8EncodedKeySpec;
import java.security.interfaces.DHPublicKey;
import javax.crypto.spec.SecretKeySpec;
public class AuthenticatedDHExample {
   private static BigInteger g512 = new BigInteger(
         "153d5d6172adb43045b68ae8e1de1070b6137005686d29d3d73a7"
       + "749199681ee5b212c9b96bfdcfa5b20cd5e3fd2044895d609cf9b"
       + "410b7a0f12ca1cb9a428cc", 16);
   private static BigInteger p512 = new BigInteger(
         "9494fec095f3b85ee286542b3836fc81a5dd0a0349b4c239dd387"
       + "44d488cf8e31db8bcb7d33b41abb9e5a33cca9144b1cef332c94b"
       + "f0573bf047a3aca98cdf3b", 16)
   public static void main(String[] args) throws Exception {
      DHParameterSpec dhParams = new DHParameterSpec(p512, g512);
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DH");
      keyGen.initialize(dhParams);
      KeyPair aliceKeyPair = keyGen.generateKeyPair();
      DHPublicKey alicePublicKey = (DHPublicKey) aliceKeyPair.getPublic();
      DHPrivateKey alicePrivateKey = (DHPrivateKey) aliceKeyPair.getPrivate()
      KeyPair bobKeyPair = keyGen.generateKeyPair();
```

```
DHPublicKey bobPublicKey = (DHPublicKey) bobKeyPair.getPublic();
DHPrivateKey bobPrivateKey = (DHPrivateKey) bobKeyPair.getPrivate();
KeyAgreement aliceKeyAgree = KeyAgreement.getInstance("DH");
aliceKeyAgree.init(alicePrivateKey);
aliceKeyAgree.doPhase(bobPublicKey, true);
KeyAgreement bobKeyAgree = KeyAgreement.getInstance("DH");
bobKeyAgree.init(bobPrivateKey);
bobKeyAgree.doPhase(alicePublicKey, true);
byte[] aliceSharedSecret = aliceKeyAgree.generateSecret();
byte[] bobSharedSecret = bobKeyAgree.generateSecret();
if (MessageDigest.isEqual(aliceSharedSecret, bobSharedSecret)) {
   System.out.println("SEGRDOS compartilhadOs coincidem!");
} else {
   System.out.println("Segredos compartilhados não coincidem");
}
byte[] message = "Hello, Bob!".getBytes();
Signature aliceSign = Signature.getInstance("SHA256withRSA");
aliceSign.initSign(alicePrivateKey);
aliceSign.update(message);
byte[] aliceSignature = aliceSign.sign();
Signature bobVerify = Signature.getInstance("SHA256withRSA");
bobVerify.initVerify(alicePublicKey);
bobVerify.update(message);
if (bobVerify.verify(aliceSignature)) {
   System.out.println("A mensagem da Alice verificada!");
} else {
   System.out.println("Messagem da Alice nao verificada");
}}}
```