Documentación sobre Programación Segura y Criptografía en Java

1. CONCEPTOS BÁSICOS

La criptografía es la ciencia que estudia técnicas para garantizar la seguridad de la información. Su objetivo es permitir que los datos sean transmitidos o almacenados sin ser accesibles por terceros no autorizados. Aunque tradicionalmente la criptografía se ha definido como el arte de escribir en clave secreta, en la actualidad es una disciplina matemática que se aplica a la seguridad informática y a las comunicaciones seguras.

El proceso de **encriptar** consiste en transformar información legible en un formato ilegible sin la clave de desencriptación. En informática, se utilizan términos como **codificar/descodificar** o **cifrar/descifrar** como sinónimos de encriptar/desencriptar.

Ejemplo 1: Escenario de comunicación segura

Pedro quiere enviar un mensaje privado a Ana a través de Internet. Para evitar que terceros accedan al contenido, Pedro cifra el mensaje con una clave secreta antes de enviarlo. Ana, al recibir el mensaje, usa la misma clave para descifrarlo y leerlo en su formato original.

2. APLICACIONES DE LA CRIPTOGRAFÍA

La criptografía tiene múltiples aplicaciones en la seguridad informática:

- **Identificación y autenticación**: Se utiliza para validar la identidad de usuarios y sistemas.
- **Certificación**: Permite establecer la identidad de un usuario mediante agentes confiables.
- **Seguridad en comunicaciones**: Se emplea en protocolos como HTTPS para garantizar la confidencialidad de los datos transmitidos.
- **Comercio electrónico**: Asegura que las transacciones en línea sean seguras y que los datos financieros no sean vulnerables a ataques.

Ejemplo 2: Seguridad en Internet

Los protocolos **SSL y TLS** son utilizados en las conexiones HTTPS para proporcionar seguridad a las transacciones web, evitando que la información pueda ser interceptada y modificada.

3. HISTORIA DE LA CRIPTOGRAFÍA

El uso de la criptografía se remonta a la antigüedad, cuando se empleaban métodos simples como la sustitución de letras para ocultar mensajes.

- **Escítalo espartano (siglo V a.C.)**: Un cilindro de madera en el que se enrollaba una tira de cuero con el mensaje escrito. Solo con un cilindro del mismo diámetro se podía leer correctamente.
- **Método César**: Consistía en sustituir cada letra del mensaje por otra desplazada un número fijo de posiciones en el alfabeto.
- **Cifrado de Felipe II**: Utilizaba más de 500 símbolos distintos para ocultar información, con combinaciones de letras, números y trazos especiales.
- **Máquina Enigma (Segunda Guerra Mundial)**: Un dispositivo mecánico-electrónico de cifrado utilizado por el ejército alemán para enviar mensajes seguros. Su descifrado por Alan Turing y su equipo fue clave en la victoria aliada.

Ejemplo 3: Cifrado César

Si desplazamos 3 posiciones en el alfabeto:

- Mensaje original: HOLA
- Mensaje cifrado: LROD

4. CARACTERÍSTICAS DE LOS SERVICIOS DE SEGURIDAD

Para garantizar la seguridad de la información, un sistema criptográfico debe cumplir con ciertas características:

- **Confidencialidad**: Garantiza que solo los usuarios autorizados puedan acceder a la información.
- **Integridad**: Asegura que los datos no han sido alterados sin autorización.
- **Autenticación**: Verifica la identidad del emisor y el receptor.
- **No repudio**: Evita que un usuario niegue haber realizado una acción específica.

Ejemplo 4: Autenticación y firma digital

Una empresa usa firmas digitales para garantizar que los documentos enviados por correo electrónico sean auténticos y no hayan sido alterados.

5. ESTRUCTURA DE UN SISTEMA SECRETO

Un sistema criptográfico consta de dos funciones principales:

- **Función de cifrado**: Convierte un mensaje en texto claro a un mensaje cifrado.
- **Función de descifrado**: Transforma el mensaje cifrado nuevamente en texto claro.

Para mejorar la seguridad, un sistema criptográfico debe ser resistente al análisis de frecuencia y ataques de fuerza bruta.

El **cifrado simétrico** es un método de encriptación en el que se utiliza la misma clave secreta tanto para cifrar como para descifrar la información. Esto significa que el emisor y el receptor deben compartir previamente la

clave para poder comunicarse de manera segura. Es un método rápido y eficiente, pero presenta el desafío de distribuir la clave de manera segura para evitar accesos no autorizados.

El **Advanced Encryption Standard (AES)** es un algoritmo de cifrado simétrico que protege la información mediante bloques de **128 bits**, usando claves de **128, 192 o 256 bits**. Funciona mediante una serie de transformaciones matemáticas en varias rondas, que incluyen sustitución de bytes, permutación de filas, mezcla de columnas y combinación con una clave secreta. Es rápido, seguro y ampliamente utilizado en comunicaciones seguras, almacenamiento cifrado y protocolos como SSL/TLS.

6. HERRAMIENTAS DE PROGRAMACIÓN PARA CIFRADO

Java proporciona diversas clases para manejar claves y cifrado:

- **Key**: Representa una clave de cifrado.
- **KeySpec**: Define especificaciones de claves en formatos estándar.
- **Cipher**: Implementa algoritmos de cifrado y descifrado.

Ejemplo 5: Uso de Java para cifrado

```
// Importamos las clases necesarias para el cifrado simétrico en Java
import javax.crypto.Cipher;// Clase que permite cifrar y descifrar datos
import javax.crypto.KeyGenerator;  // Clase para generar claves de cifrado simétrico
import javax.crypto.SecretKey;     // Representa la clave secreta usada en el cifrado

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

        // 1. Generación de una clave secreta AES
        // Creamos un generador de claves para AES
        KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
      // Establecemos la longitud de la clave en 128 bits (puede ser 192 o 256 también)
        keyGenerator.init(128);
        SecretKey secretKey = keyGenerator.generateKey(); // Generamos la clave AES

        // 2. Creación del objeto Cipher en modo de cifrado
        // Creamos un objeto Cipher configurado para AES
        Cipher cipher = Cipher.getInstance("AES");
        // Inicializamos el cifrador en modo de ENCRIPTACIÓN con la clave generada
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);

        // 3. Definimos el mensaje que queremos cifrar
        String message = "Mensaje seguro"; // Texto original que será cifrado

        // 4. Ciframos el mensaje
        // Convertimos el mensaje en bytes y lo ciframos

        byte[] encryptedMessage = cipher.doFinal(message.getBytes());

        // 5. Inicializamos el Cipher en modo de descifrado
```

```
        // Ahora el cifrador se usa para descifrar, utilizando la misma clave
        cipher.init(Cipher.DECRYPT_MODE, secretKey);

        // 6. Desciframos el mensaje
       // Desciframos el mensaje cifrado
        byte[] decryptedMessage = cipher.doFinal(encryptedMessage);

        // 7. Mostramos los resultados en consola
        // Mostramos el mensaje original
        System.out.println("Mensaje original: " + message);
       // Mostramos el mensaje cifrado (puede no ser legible)
        System.out.println("Mensaje cifrado: " + new String(encryptedMessage));
      // Mostramos el mensaje descifrado correctamente
        System.out.println("Mensaje descifrado: " + new String(decryptedMessage));
    }
}
```

Documentación sobre Programación Segura y Criptografía en Java (Parte 2)

7. MODELO DE CLAVE PRIVADA

El modelo de clave privada, también conocido como **criptografía simétrica**, utiliza una única clave secreta para cifrar y descifrar la información. Ambos participantes en la comunicación deben poseer esta clave antes de intercambiar mensajes cifrados.

7.1 Algoritmos de Cifrado Simétrico

- **DES (Data Encryption Standard)**: Algoritmo de cifrado por bloques de 56 bits que ha sido reemplazado por alternativas más seguras debido a su vulnerabilidad a ataques de fuerza bruta.
- **3DES (Triple DES)**: Aplica DES tres veces consecutivas para aumentar la seguridad.
- **AES (Advanced Encryption Standard)**: Ofrece claves de 128, 192 y 256 bits y es ampliamente utilizado en la actualidad.

Ejemplo 6: Cifrado con AES en Java

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;

public class AESExample {
    public static void main(String[] args) throws Exception {
        KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
        keyGenerator.init(128);
        SecretKey secretKey = keyGenerator.generateKey();

        Cipher cipher = Cipher.getInstance("AES");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);

        String mensaje = "Texto secreto";
        byte[] mensajeCifrado = cipher.doFinal(mensaje.getBytes());

        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        byte[] mensajeDescifrado = cipher.doFinal(mensajeCifrado);

        System.out.println("Mensaje original: " + mensaje);
        System.out.println("Mensaje cifrado: " + new
String(mensajeCifrado));
        System.out.println("Mensaje descifrado: " + new
String(mensajeDescifrado));
    }
}
```

8. MODELO DE CLAVE PÚBLICA

A diferencia de la criptografía simétrica, el modelo de clave pública o **criptografía asimétrica** emplea un par de claves: una pública y otra privada. La clave pública se usa para cifrar los mensajes, mientras que la clave privada se emplea para descifrarlos.

8.1 Algoritmos de Cifrado Asimétrico

- **RSA**: Se basa en la dificultad de factorizar números grandes en sus factores primos.
- **Diffie-Hellman**: Permite el intercambio seguro de claves sobre un canal inseguro.

Ejemplo 7: Cifrado con RSA en Java

```
// Importamos las clases necesarias para la criptografía en Java
import javax.crypto.Cipher; // Clase utilizada para realizar cifrado y descifrado
import java.security.*;     // Proporciona clases para la generación y manejo de claves
de cifrado

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

        // 1. Generación de un par de claves RSA
        // Creamos un generador de claves para RSA
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
        // Configuramos la longitud de la clave en 2048 bits (mayor seguridad)
        keyGen.initialize(2048);
        // Generamos el par de claves (pública y privada)
        KeyPair keyPair = keyGen.generateKeyPair();

        // 2. Creación del objeto Cipher en modo de cifrado
        // Creamos un objeto Cipher configurado para RSA
        Cipher cipher = Cipher.getInstance("RSA");
        // Inicializamos el cifrador en modo cifrado usando la clave pública
        cipher.init(Cipher.ENCRYPT_MODE, keyPair.getPublic());

        // 3. Definimos el mensaje que queremos cifrar
        // Texto original que será cifrado
        String mensaje = "Mensaje seguro con RSA";

        // 4. Ciframos el mensaje con la clave pública
        // Convertimos el mensaje en bytes y lo ciframos
        byte[] mensajeCifrado = cipher.doFinal(mensaje.getBytes());

        // 5. Configuración del Cipher en modo de descifrado
        // Ahora usamos la clave privada para descifrar
        cipher.init(Cipher.DECRYPT_MODE, keyPair.getPrivate());

        // 6. Desciframos el mensaje previamente cifrado
        // Desciframos el mensaje con la clave privada
        byte[] mensajeDescifrado = cipher.doFinal(mensajeCifrado);

        // 7. Mostramos los resultados en consola
        // Imprimimos el mensaje original
        System.out.println("Mensaje original: " + mensaje);
        // Mostramos el mensaje cifrado (puede no ser legible)
        System.out.println("Mensaje cifrado: " + new String(mensajeCifrado));
        // Imprimimos el mensaje descifrado
        System.out.println("Mensaje descifrado: " + new String(mensajeDescifrado));
    }
}
```

9. FIRMA DIGITAL

La **firma digital** permite garantizar la autenticidad y la integridad de los mensajes, asegurando que no han sido alterados en tránsito. Se basa en la combinación de un algoritmo de cifrado asimétrico y una función hash.

Una **función hash** es un algoritmo matemático que convierte un mensaje de cualquier longitud en un resumen de tamaño fijo, generalmente en **64, 128 o 256 bits**. Se utiliza en múltiples aplicaciones, como la identificación de archivos, la seguridad de contraseñas y la verificación de integridad de datos. Para ser útil, una función hash debe ser **rápida de calcular**, generar valores únicos para entradas distintas (evitando **colisiones**) y ser una **función de un solo sentido**, lo que significa que no se puede reconstruir el mensaje original a partir del hash. Además, debe ser resistente a ataques, de modo que, aunque se conozcan múltiples pares de datos y sus hashes, no sea posible deducir el proceso para generar un hash específico.

9.1 Algoritmos de Firma Digital

- **DSA (Digital Signature Algorithm)**: Estándar de firma digital basado en criptografía asimétrica.
- **ECDSA (Elliptic Curve Digital Signature Algorithm)**: Variante basada en curvas elípticas, utilizada en sistemas de alto rendimiento.

Ejemplo 8: Firma Digital con DSA en Java

```
// Importamos las clases necesarias para la generación de claves y firma
digital
import java.security.KeyPair;          // Clase que representa un par de
claves (privada y pública)
import java.security.KeyPairGenerator; // Clase para generar pares de claves
import java.security.Signature;        // Clase para crear y verificar firmas
digitales

public class SignatureExample {
    public static void main(String[] args) {
        try {
            // 1. Generación del par de claves con el algoritmo DSA (Digital
Signature Algorithm)
            System.out.println("Obteniendo generador de claves con cifrado
DSA");
            // Se obtiene un generador de claves DSA
            KeyPairGenerator keygen = KeyPairGenerator.getInstance("DSA");

            System.out.println("Generando par de claves");
            // Se genera el par de claves (pública y privada)
            KeyPair keypair = keygen.generateKeyPair();

            // 2. Creación del objeto Signature para firmar el mensaje
            System.out.println("Creando objeto signature");
            // Se obtiene una instancia del algoritmo DSA para firma digital
            Signature signature = Signature.getInstance("DSA");

            // 3. Firma del mensaje
            System.out.println("Firmando mensaje");
            // Se inicializa el objeto Signature con la clave privada para
firmar
            signature.initSign(keypair.getPrivate());
            // Mensaje que será firmado
```

```
            String mensaje = "Mensaje para firmar";
            // Se actualiza el objeto Signature con los bytes del mensaje
            signature.update(mensaje.getBytes());
            // Se genera la firma digital del mensaje
            byte[] firma = signature.sign();

            // 4. Verificación de la firma digital
            System.out.println("Comprobando el mensaje firmado");

            // Se inicializa el objeto Signature con la clave pública para
verificar la firma
            signature.initVerify(keypair.getPublic());

            // Se actualiza el objeto Signature con los bytes del mensaje
original
            signature.update(mensaje.getBytes());

            // Se verifica la firma digital comparándola con la clave pública
            if (signature.verify(firma))
                // Si la verificación es exitosa, se muestra este mensaje
                System.out.println("El mensaje es auténtico :-)");

        } catch (Exception e) {
            // Si ocurre algún error, se imprime la traza de la excepción
            e.printStackTrace();
        }
    }
}
```

Documentación sobre Programación Segura y Criptografía en Java (Parte 3)

10. SERVICIOS EN RED SEGUROS: PROTOCOLOS SSL, TLS Y SSH

10.1 SSL (Secure Sockets Layer) y TLS (Transport Layer Security)

SSL y TLS son protocolos de seguridad que permiten el cifrado de datos en la comunicación entre clientes y servidores, garantizando autenticación, integridad y confidencialidad.

- **SSL (versión obsoleta)**: Utilizado en las primeras versiones de seguridad en la web.
- **TLS (versión actualizada de SSL)**: Protocolo más seguro y moderno que reemplazó a SSL.

10.1.1 Funcionamiento de TLS

1. **Handshake**: Cliente y servidor intercambian certificados y establecen una clave de sesión.
2. **Cifrado de datos**: La clave de sesión se usa para cifrar los datos enviados entre ambos.
3. **Autenticación y verificación**: Se usa la firma digital para garantizar la integridad de los datos.

Ejemplo 9: Uso de sockets seguros con TLS en Java

```
// Importamos las clases necesarias para la implementación de un servidor seguro con SSL/TLS
import javax.net.ssl.*;  // Librerías para la gestión de sockets seguros (SSL/TLS)
import java.io.*;        // Para manejar la entrada/salida de datos
import java.net.*;       // Para el manejo de sockets y conexiones de red

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

        // 1. Obtener la fábrica de sockets SSL para crear sockets seguros
        SSLServerSocketFactory factory = (SSLServerSocketFactory)
SSLServerSocketFactory.getDefault();

        // 2. Crear un servidor seguro que escucha en el puerto 8443
        // Este puerto es comúnmente utilizado para conexiones HTTPS con SSL/TLS
        SSLServerSocket serverSocket = (SSLServerSocket) factory.createServerSocket(8443);
        System.out.println("Servidor seguro en espera de conexiones...");

        // 3. Esperar la conexión de un cliente y aceptar la conexión entrante
        SSLSocket socket = (SSLSocket) serverSocket.accept();

        // 4. Crear un BufferedReader para leer datos enviados por el cliente
        BufferedReader reader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));

        // 5. Leer el mensaje enviado por el cliente y mostrarlo en consola
        System.out.println("Mensaje recibido: " + reader.readLine());

        // 6. Cerrar la conexión con el cliente
        socket.close();

        // 7. Cerrar el socket del servidor para liberar el puerto
        serverSocket.close();
    }
}
```

11. PROGRAMACIÓN CON SOCKETS SEGUROS

Los **sockets seguros** permiten la comunicación cifrada entre aplicaciones distribuidas. En Java, la biblioteca `javax.net.ssl` proporciona clases como `SSLSocket` y `SSLServerSocket` para gestionar conexiones seguras.

Ejemplo 10: Cliente seguro con TLS en Java

```
// Importamos las clases necesarias para la comunicación segura con SSL/TLS
import javax.net.ssl.*; // Proporciona clases para gestionar sockets seguros (SSL/TLS)
import java.io.*;       // Para manejar la entrada y salida de datos
import java.net.*;      // Para el manejo de sockets y conexiones de red

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

        // 1. Obtener la fábrica de sockets seguros (SSL/TLS)
        // Esta fábrica se usa para crear sockets que establecen una conexión cifrada
        SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();

        // 2. Crear un socket seguro que se conectará al servidor en el puerto 8443
        SSLSocket socket = (SSLSocket) factory.createSocket("localhost", 8443);

        // 3. Crear un PrintWriter para enviar datos al servidor a través del socket
        PrintWriter writer = new PrintWriter(socket.getOutputStream(), true);

        // 4. Enviar un mensaje al servidor
        writer.println("Mensaje seguro desde el cliente");

        // 5. Cerrar la conexión con el servidor
        socket.close();
    }
}
```

13. CONTROL DE ACCESO EN SISTEMAS SEGUROS

El control de acceso define las reglas que regulan qué usuarios pueden acceder a qué recursos en un sistema.

13.1 Métodos de Control de Acceso

- **Listas de Control de Acceso (ACLs)**: Define permisos a nivel de usuario.
- **Roles y permisos**: Se otorgan permisos según el rol de cada usuario.

13.2 Ejemplo de Control de Acceso en Java

```
// Importamos las clases necesarias para manipular archivos y sus atributos de seguridad
import java.nio.file.*;               // Proporciona clases para trabajar con rutas y archivos
import java.nio.file.attribute.*;     // Permite acceder a los atributos de archivos, como
permisos y ACL

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

        // 1. Definir la ruta del archivo cuyo control de acceso queremos consultar
        Path file = Paths.get("archivo_secreto.txt"); // Ruta relativa al archivo

        // 2. Obtener la vista de atributos ACL (Access Control List) del archivo
        // Esto permite acceder y modificar los permisos de seguridad del archivo en sistemas
compatibles
        AclFileAttributeView aclView = Files.getFileAttributeView(file,
AclFileAttributeView.class);

        // 3. Imprimir la lista de permisos de acceso (ACL) del archivo
        System.out.println("Permisos de acceso: " + aclView.getAcl());
    }
}
```