Unidad 5 – Anexo 2

Hash de contraseñas

INTRODUCCIÓN **2**

FUNCIONES KDF **2**

CIFRADO Y VERIFICACIÓN DE CONTRASEÑAS CON JAVA **3**

ACTIVIDADES **5**

Introducción

Para que los procesos de autenticación sean seguros es necesario almacenar las contraseñas cifradas. El proceso de cifrado se realiza mediante una función matemática conocida como *función hash criptográfica*. Si bien existe una gran variedad de funciones hash, las que se adaptan a las necesidades del cifrado de contraseñas deben cumplir cuatro propiedades fundamentales para ser seguras:

1. Determinismo: el mismo mensaje procesado por la misma función hash siempre debe producir el mismo hash.
2. Irreversibilidad: ha de ser impracticable la posibilidad de generar la contraseña a partir de su hash.
3. Alta entropía: cualquier pequeña variación en la contraseña debería producir un hash muy diferente.
4. Resistencia a colisiones: dos contraseñas diferentes no deberían producir el mismo hash.

Una función hash que tenga estas cuatro propiedades será un candidato fuerte para el cifrado de contraseñas, ya que las cuatro propiedades juntas incrementan enormemente la dificultad de recuperar la contraseña a partir de su hash usando ingeniería inversa.

Sin embargo, para el cifrado de contraseñas es deseable una quinta propiedad: que las funciones hash sean deliberadamente lentas para dificultar el crackeo de la contraseña a través de ataques de diccionario o ataques de *rainbow table*. Lo contrario facilitaría este tipo de ataques al posibilitar el cifrado y comparación de miles de millones de contraseñas potenciales en intervalos de tiempo cortos.

Funciones KDF

En criptografía, una función de derivación de claves o ***KDF*** (***K**ey **D**erivation **F**unction*) es un algoritmo criptográfico que deriva una clave secreta (clave de cifrado simétrico) a partir de un dato secreto, como una clave maestra, una contraseña o una frase de contraseña, usando una función pseudoaleatoria.

De entre los múltiples usos que pueden tener las funciones *KDF*, el más común es el hash de contraseñas, pese a que originalmente no fueron concebidas para este propósito. Esto es debido a que tienen todas las características mencionadas anteriormente que las hacen aptas para ser usadas como *funciones hash criptográficas*.

Este tipo de funciones reciben dos parámetros básicos:

- Un conjunto de bits aleatorios denominados sal criptográfica (en inglés, *salt*) que se unen a la contraseña para formar un conjunto de bits final del cual se obtendrá el hash.
- El número de iteraciones que ha de ejecutar la función pseudoaleatoria hasta completar el cálculo del hash.

Para determinar el número de bits de sal criptográfica y el número de iteraciones se pueden seguir las recomendaciones de guías como *OWASP Password Storage Cheat Sheet* o *NIST Recommendation for Password-Based Key Derivation*.

Un ejemplo de función *KDF* es ***PBKDF2***, que es una especificación de un algoritmo genérico de alto nivel que no exige el uso de ninguna función pseudoaleatoria en particular. En cualquier caso, la elección más común para la función pseudoaleatoria es ***HMAC***, una especificación de alto nivel que utiliza internamente una función hash criptográfica. De nuevo, esta especificación no exige ninguna función hash en particular, siendo habitual usar alguna de las funciones de la familia **SHA2**.

Cifrado y verificación de contraseñas con Java

Creamos la clase *Passwords* en la que se definen métodos de clase siguientes para cifrar y verificar contraseñas usando el algoritmo "PBKDF2WithHmacSHA512":

- Método *cifrar*: sobrecargado con tres versiones, dos de ellas públicas y una privada.
    - Las versiones públicas retornan la contraseña cifrada.
    - La versión privada es para uso interno de los métodos públicos.
- Método *verificar*: verifica que una contraseña cifrada corresponde una determinada contraseña en claro.

```
1
2   public class Passwords {
3
4       public static String cifrar(String password) throws NoSuchAlgorithmException,
5               InvalidKeySpecException, IOException {
6           return cifrar(password, 16, 310000, 512);
7       }
8
9       public static String cifrar(String password, Integer longitudSal, Integer iteraciones, int longitudClave)
10              throws NoSuchAlgorithmException, InvalidKeySpecException, IOException {
11          byte[] sal = new byte[longitudSal];
12          new SecureRandom().nextBytes(sal);
13          try (ByteArrayOutputStream as = new ByteArrayOutputStream();
14                  DataOutputStream out = new DataOutputStream(as)) {
15              byte[] key = cifrar(password.toCharArray(), sal, iteraciones, longitudClave);
16              out.writeInt(iteraciones);
17              out.writeInt(sal.length);
18              out.writeInt(key.length);
19              out.write(sal);
20              out.write(key);
21              return Base64.getEncoder().encodeToString(as.toByteArray());
22          }
23      }
24
25      private static byte[] cifrar(char [] password, byte[] salt, int iteraciones, int longitudClave)
26              throws NoSuchAlgorithmException, InvalidKeySpecException {
27          KeySpec keySpec = new PBEKeySpec(password, salt, iteraciones, longitudClave);
28          return SecretKeyFactory.getInstance("PBKDF2WithHmacSHA512").generateSecret(keySpec).getEncoded();
29      }
30
31      static boolean verificar(String password, String cifrada)
32              throws NoSuchAlgorithmException, InvalidKeySpecException, IOException {
33          try (DataInputStream in =
34                  new DataInputStream(new ByteArrayInputStream(Base64.getDecoder().decode(cifrada)))) {
35              int iteraciones = in.readInt();
36              byte [] sal = new byte[in.readInt()];
37              byte [] clave = new byte[in.readInt()];
38              in.read(sal);
39              in.read(clave);
40              return Arrays.equals(clave, cifrar(password.toCharArray(), sal, iteraciones, clave.length * 8));
41          }
42      }
43
44  }
45
```

Método *cifrar* (línea 4)

Declara el parámetro formal *password* que representa una contraseña en texto claro.

Se apoya en el método *cifrar* definido a partir de la línea 9 para realizar el cifrado usando los siguientes parámetros de derivación por defecto:

- Longitud de la sal criptográfica: 16 bytes (128 bits).
- Número de iteraciones: 310.000.
- Longitud de la clave derivada: 512 bits (64 bytes).

Retorna la contraseña cifrada codificada en Base64 (ver el siguiente método).

Método *cifrar* (línea 9)

Declara los parámetros formales siguientes:

- *password*: contraseña en texto claro.
- *longitudSalt*: longitud en bytes de la sal criptográfica que se va a generar.
- *iteraciones*: número de iteraciones que usara el algoritmo PBKDF2.
- *longitudClave*: longitud en bits de la clave derivada.

Realiza las acciones siguientes:

- Genera de forma aleatoria la sal criptográfica.
- Invoca a la versión privada del método *cifrar*, definida a partir de la línea 25, para obtener la clave derivada en un array de bytes.
- Agrupa en un segundo array de bytes lo que finalmente será la **contraseña cifrada**:
    - El número de iteraciones
    - La longitud de la sal criptográfica en bytes.
    - La longitud de la clave derivada en bytes
    - La sal criptográfica.
    - La clave derivada.

  y lo codifica en Base64.

Retorna la contraseña cifrada codificada en Base64.

Método *cifrar* (línea 25)

Declara los parámetros formales siguientes:

- *password*: array de bytes que representa la contraseña en texto claro.
- *longitudSalt*: longitud en bytes de la sal criptográfica que se va a generar.
- *iteraciones*: número de iteraciones que usara el algoritmo PBKDF2.
- *longitudClave*: longitud en bits de la clave derivada.

Crea una especificación transparente de clave de tipo *PBEKeySpec* con los parámetros que recibe y usa un objeto *SecretKeyFactory* para generar una clave derivada a partir de dicha especificación usando el algoritmo PBKDF2WithHmacSHA512.

Retorna la clave derivada codificada en un array de bytes.

Método *verificar* (línea 31)

Declara los parámetros formales siguientes:

- *password*: cadena que representa una contraseña en texto claro.
- *cifrada*: cadena que contiene una contraseña cifrada tal y como la retornan los métodos *cifrar* públicos.

Realiza las acciones siguientes:

- Decodifica la contraseña cifrada para obtener el array de bytes que contiene los parámetros de derivación y una clave derivada.
- Extrae del array los parámetros de derivación y la clave derivada.
- Invoca al método *cifrar* privado para derivar una clave pasándole la contraseña en claro y los parámetros de derivación extraídos.
- Compara la clave derivada que acaba de obtener con la clave derivada almacenada en el array y si coinciden, la verificación es positiva. En caso contrario la verificación será negativa.

Retorna *true* en caso de verificación positiva y *false* en caso contrario.

Actividades

1. Analiza que ocurrirá si se especifica una longitud de clave que no sea múltiplo de 8.
2. Modifica la clase *Passwords* para que se pueda especificar la función hash que se usará para cifrar las contraseñas, haciendo que la primera versión del método *cifrar* use SHA512 por defecto.
3. Modifica la segunda versión del método cifrar para que mezcle la clave derivada con la sal de forma que resulte complicado separarlas si se desconoce el procedimiento de mezcla.
4. Modifica la segunda versión del método cifrar para que en lugar de codificar la contraseña cifrada en Base 64, la codifique en hexadecimal.
5. Investiga la posibilidad de usar otras funciones KDF para cifrar contraseñas, p. ej. bcrypt, script, Lyra2 o Argon2.
6. Crea un programa de prueba que permita almacenar nombres de usuario y contraseñas cifradas en un fichero de texto. Cada par usuario/contraseña se almacenará en una línea con el formato siguiente:

   nombre_de_usuario:*contraseña_cifrada*

7. Crea un programa de prueba que permita almacenar nombres de usuario y contraseñas cifradas en una tabla de una base de datos relacional, p. ej. MySQL.