Introducción

En general, cuando se envía algo a través de sockets se envía como «texto plano», es decir, no sabemos si hay alguien usando un sniffer en la red y por tanto no sabemos si alguien está capturando los datos.

Cualquier sistema que pretenda ser seguro necesitará usar cifrado.

Prácticas de programación segura.

Para enviar mensajes cifrados se necesita algún mecanismo o algoritmo para convertir un texto normal en uno más difícil de comprender.

Criptografía de clave pública y clave privada.

Los principales sistemas modernos de seguridad utilizan dos claves, una para cifrar y otra para descifrar. Esto se puede usar de diversas formas.

Principales aplicaciones de la criptografía.

- Mensajería segura: todo el mundo da su clave de cifrado, pero conserva la de descifrado. Si queremos enviar un mensaje a alguien cogemos su clave de cifrado y ciframos el mensaje que le enviamos. Solo él podrá descifrarlo.
- Firma digital: pilar del comercio electrónico. Permite verificar que un archivo no ha sido modificado.
- Autenticación: los sistemas de autenticación intentan resolver una cuestión clave en la informática: **verificar que una máquina es quien dice ser**

Protocolos criptográficos.

En realidad, protocolos criptográficos hay muchos, y suelen dividirse en sistemas simétricos o asimétricos.

- Los sistemas simétricos son aquellos basados en una función que convierte un mensaje en otro mensaje cifrado. Si se desea descifrar algo se aplica el proceso inverso con la misma clave que se usó.

- Los sistemas asimétricos utilizan una clave de cifrado y otra de descifrado, donde estas claves son complementarias. Aunque se tenga una clave es matemáticamente imposible averiguar la otra clave por lo que se puede dar a todo el mundo una de las claves (llamada habitualmente **clave pública**) y conservar la otra (llamada **clave privada**). Además, podemos usar las claves para lo que queramos y por ejemplo en unos casos cifraremos con la clave pública y en otros tal vez cifremos con la clave privada.

Hoy por hoy, las mayores garantías las ofrecen los asimétricos, de los cuales hay varios sistemas. El inconveniente que pueden tener los asimétricos es que son más lentos computacionalmente.

Encriptación de información.

El siguiente código muestra cómo crear una clase que permita cifrar y descifrar textos, con el cifrado asimétrico RSA.

Clases e Interfaces en Java

Para aplicar encriptación asimétrica usaremos básicamente las siguientes clases de **java.security**:

KeyPair	Encapsula un par de claves. **getPublic()** devuelve la clave pública y **getPrivate()** la privada
PublicKey	Interface para claves públicas. En el package **java.security.interfaces** está la sub-interface **RSAPublicKey** que define las claves para el algoritmo RSA y permite acceder a información de las claves RSA.
PrivateKey	Similar a la anterior, para la clave privada. Hay que considerar las sub-interfaces **RSAPrivateKey** y **RSAPrivateCrtKey** que contienen métodos extra para coger parámetros de estas claves.
KeyPairGenerator	Las claves pública y privada siempre se generan juntas con el método **genKeyPair().**

```
import java.io.UnsupportedEncodingException;
import java.security.InvalidKeyException;
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;

import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.NoSuchPaddingException;

public class GestorCifrado {
	KeyPair claves;//Clave publica y privada
	KeyPairGenerator generadorClaves;
	Cipher cifrador;

	public GestorCifrado() throws NoSuchAlgorithmException,
NoSuchPaddingException {
		//Generador de claves: RSA
		generadorClaves = KeyPairGenerator.getInstance("RSA");
		// Usaremos una longitud de clave de 1024 bits
		generadorClaves.initialize(1024);
		//Generamos la clave publica y privada
		claves = generadorClaves.generateKeyPair();
		cifrador = Cipher.getInstance("RSA");
	}

	public PublicKey getPublica() {
		return claves.getPublic();
	}

	public PrivateKey getPrivada() {
		return claves.getPrivate();
	}

	/**
	 * Encripta los datos con la clave de cifrado aportada
	 * @param paraCifrar
	 * @param claveCifrado
	 * @return byte[]
	 *
	 * @throws InvalidKeyException
	 * @throws IllegalBlockSizeException
	 * @throws BadPaddingException
	 */
	public byte[] cifrar(byte[] paraCifrar, Key claveCifrado)
			throws InvalidKeyException, IllegalBlockSizeException,
BadPaddingException {

		// Se pone el cifrador en modo cifrado, y se le pasa la clave
para encriptar
		cifrador.init(Cipher.ENCRYPT_MODE, claveCifrado);
		//Encripta los datos
		byte[] resultado = cifrador.doFinal(paraCifrar);
		return resultado;
	}
```

```
	/**
	 * Desencripta los datos con la clve de descrifrado aportada
	 * @param paraDescifrar
	 * @param claveDescifrado
	 * @return byte[]
	 *
	 * @throws InvalidKeyException
	 * @throws IllegalBlockSizeException
	 * @throws BadPaddingException
	 */
	public byte[] descifrar(byte[] paraDescifrar, Key claveDescifrado)
			throws InvalidKeyException, IllegalBlockSizeException,
BadPaddingException {

		// Se pone el cifrador en modo descifrado
		cifrador.init(Cipher.DECRYPT_MODE, claveDescifrado);
		// Desencriptamos
		byte[] resultado = cifrador.doFinal(paraDescifrar);
		return resultado;
	}

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

		try {
			GestorCifrado gestorCifrado = new GestorCifrado();
			Key clavePublica = gestorCifrado.getPublica();
			//Los objetos que cifran y descifran en Java utilizan
estrictamente objetos byte[]
			String mensajeOriginal = "Hola mundo";
			byte[] mensajeCifrado =
gestorCifrado.cifrar(mensajeOriginal.getBytes(), clavePublica);
			String cadCifrada = new String(mensajeCifrado, "UTF-8");

			System.out.println("Cadena original:" + mensajeOriginal);
			System.out.println("Cadena cifrada:" + cadCifrada);

			/*
			 * Cogemos la cadCifrada y la desciframos con la otra
clave
			 */
			Key clavePrivada = gestorCifrado.getPrivada();
			byte[] descifrada =
gestorCifrado.descifrar(mensajeCifrado, clavePrivada);
			/* E imprimimos el mensaje */
			String mensajeDescifrado = new String(descifrada, "UTF-
8");
			System.out.println("El mensaje descifrado es:" +
mensajeDescifrado);
		}
		catch(InvalidKeyException | IllegalBlockSizeException |
BadPaddingException | UnsupportedEncodingException e) {
			e.printStackTrace();
			throw e;
		}

	}
}
```

Protocolos seguros de comunicaciones.

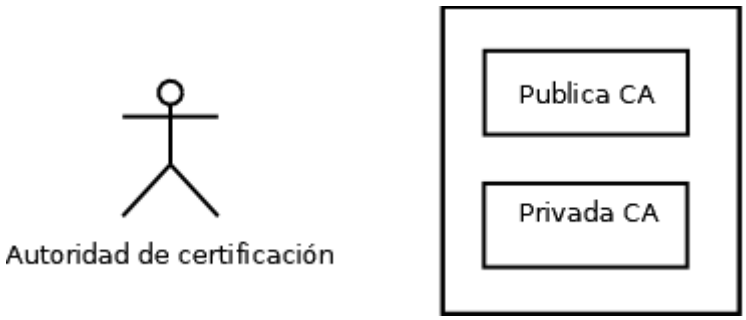
En general, ahora que ya conocemos sockets, el uso de servidores y clientes y el uso de la criptografía de clave asimétrica ya es posible crear aplicaciones que se comuniquen de forma muy segura.

En general, todo protocolo que queramos implementar dará estos pasos.

1. Todo cliente genera su pareja de claves.
2. Todo servidor genera su pareja de claves.
3. Cuando un cliente se conecte a un servidor, le envía su clave de cifrado (normalmente la clave pública) y conserva la de descifrado (normalmente la clave privada).
4. Cuando un servidor recibe la conexión de un cliente recibe la clave de cifrado de dicho cliente.
5. El servidor envía su clave pública al cliente.
6. Ahora cliente y servidor pueden enviar mensajes al otro con la garantía de que solo servidor y cliente respectivamente pueden descifrar. El cliente enviará los mensajes al servidor encriptando los mensajes con la clave pública del servidor, y el servidor enviará los mensajes al cliente usando la clave pública del cliente.

Infraestructura de clave pública (PKI)

Para garantizar la seguridad es necesario que entre un tercer jugador en el intercambio de claves entre clientes y servidores. Este tercer individuo son las ***autoridades de certificación***.



Autoridades de certificación.

Programación de aplicaciones con comunicaciones seguras.

Por fortuna Java dispone de clases ya prefabricadas que facilitan enormemente el que dos aplicaciones intercambios datos de forma segura a través de una red. Se deben considerar los siguientes puntos:

- El servidor debe tener su propio certificado. Si no lo tenemos, se puede generar primero una pareja de claves con la herramienta de Gestión de Certificados y Claves que viene con el JDK, llamada `keytool`. La herramienta guardará la pareja de claves en un almacén llamado **keystore**.

En Java un **keystore** o almacén de claves es una colección de certificados y claves. El keystore contiene dos tipos de entradas: certificados y claves.

Los certificados son en efecto certificados activos válidos. Las claves son privadas y deben estar asociadas a certificados que contienen los sujetos que poseen dichas claves. El almacén usa passwords para proteger el acceso a las claves privadas.

La herramienta que maneja almacenes de claves y permite crear certificados es la aplicación **keytool**. Algunas de las opciones de esta aplicación son:

-certreq	Crea una petición de certificado por ejemplo para obtener un certificado de una CA, como Verisign, que esté contemplada en esta aplicación.
-delete	Borra una entrada del *keystore*.
-genkey	Genera un par de claves para un certificado auto-firmado. Se puede especificar el algoritmo con *-keyalg*. Por ejemplo *-keyalg RSA*
-keyclone	Copia una entrada en el almacén.
-keystore	Especifica un fichero como almacén.
-printcert	Muestra un certificado digital.
-selfcert	Genera un certificado digital auto-firmado.
-storepasswd	Cambia el password del almacén.
-export	Exporta un certificado de un almacén codifidado en DER. Si queremos codificación BASE64 hay que añadir la opción *-rfc*.

Por ejemplo, con **keytool -v -list** listaremos las entradas del fichero keystore.

Si queremos añadir una entrada, identificada por un alias, haremos lo siguiente: **keytool -genkey -alias test** donde test es el alias. La aplicación nos irá pidiendo los datos para el certificado.

Si queremos exportarlo a codificación DER haremos lo siguiente: **keytool -export -alias test -file micertificado.cer.**

- El código del servidor necesitará indicar el fichero donde se almacenan las claves y la clave para acceder a ese almacén.
- El cliente necesita indicar que confía en el certificado del servidor. Dicho certificado del servidor puede estar guardado (por ejemplo) en el almacén de claves del cliente.
- Aunque no suele hacerse también podría hacerse a la inversa y obligar al cliente a tener un certificado que el servidor pudiera importar, lo que aumentaría la seguridad.

Los pasos desglosados implican ejecutar estos comandos en el servidor:

- Generar un par de claves en el servidor, que se almacenará en un fichero llamado 'clavesservidor':

```
keytool -genkeypair -keyalg RSA -alias servidor -keystore clavesservidor
```

- Generar un certificado con:

```
keytool -export -file certificadoservidor.cer -keystore clavesservidor.
```

En el cliente daremos estos pasos:

- Generar una pareja de claves (en realidad no nos hace falta solo queremos tener un almacén de claves.

```
keytool -genkeypair -keyalg RSA -alias cliente -keystore clavescliente
```

- Importar el certificado del servidor indicando que pertenece a la lista de certificados confiables.

```
keytool -importcert -trustcacerts -alias servidor -file servidor.cer -
keystore clavescliente
```

Una vez creados los ficheros iniciales se deben dar los siguientes pasos en Java (servidor y cliente van por separado):

1. El servidor debe cargar su almacén de claves (el fichero `clavesservidor`)
2. Ese almacén (cargado en un objeto Java llamado `KeyStore`), se usará para crear un gestor de claves (objeto `KeyManager`), el cual se obtiene a partir de una «fábrica» llamada `KeyManagerFactory`.
3. Se creará un contexto SSL (objeto `SSLContext`) a partir de la fábrica comentada.
4. El objeto `SSLContext` permitirá crear una fábrica de sockets que será la que finalmente nos permita tener un `SSLServerSocket`, es decir un socket de servidor que usará cifrado.

El código Java del servidor sería algo así:

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.Socket;
import java.security.KeyManagementException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.UnrecoverableKeyException;
import java.security.cert.CertificateException;

import javax.net.ssl.KeyManagerFactory;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLServerSocket;
import javax.net.ssl.SSLServerSocketFactory;

public class ServidorSeguro {

	private String rutaAlmacen;
	private String claveAlmacen;

	public ServidorSeguro(String rutaAlmacen, String claveAlmacen) {
		this.rutaAlmacen = rutaAlmacen;
		this.claveAlmacen = claveAlmacen;
	}

	private SSLServerSocket getServerSocketSeguro(int puerto) throws
KeyStoreException, NoSuchAlgorithmException,
			CertificateException, IOException, KeyManagementException,
UnrecoverableKeyException {

		SSLServerSocket serverSocket = null;
		FileInputStream fichAlmacen=null;
		try {
			/* Paso 1, se carga el almacén de claves */
			fichAlmacen = new FileInputStream(this.rutaAlmacen);
			/*
			 * Paso 1.1, se crea un almacén del tipo por defecto que
es un JKS (Java Key
			 * Store), a día de hoy
			 */
			KeyStore almacen =
KeyStore.getInstance(KeyStore.getDefaultType());
			almacen.load(fichAlmacen, claveAlmacen.toCharArray());
			/*
			 * Paso 2: obtener una fábrica de KeyManagers que ofrezcan
soporte al algoritmo
			 * por defecto
			 */
			KeyManagerFactory fabrica =
KeyManagerFactory.getInstance(KeyManagerFactory.getDefaultAlgorithm());
			fabrica.init(almacen, claveAlmacen.toCharArray());
			/*
```

```
                     * Paso 3:Intentamos obtener un contexto SSL que ofrezca
soporte a TLS (el
                     * sistema más seguro hoy día)
                     */
                    SSLContext contextoSSL = SSLContext.getInstance("TLS");
                    contextoSSL.init(fabrica.getKeyManagers(), null, null);
                    /*
                     * Paso 4: Se obtiene una fábrica de sockets que permita
obtener un
                     * SSLServerSocket
                     */
                    SSLServerSocketFactory fabricaSockets =
contextoSSL.getServerSocketFactory();
                    serverSocket = (SSLServerSocket)
fabricaSockets.createServerSocket(puerto);
              }
              catch(KeyStoreException | NoSuchAlgorithmException |
CertificateException | IOException | KeyManagementException |
UnrecoverableKeyException e) {
                    e.printStackTrace();
                    throw e;
              }
              finally {
                    if (null != fichAlmacen) {
                          fichAlmacen.close();
                    }
              }
              return serverSocket;
       }

       public void escuchar(int puerto)
         throws KeyManagementException, UnrecoverableKeyException,
             KeyStoreException, NoSuchAlgorithmException,
             CertificateException, IOException
     {
          BufferedReader entrada = null;
         PrintWriter salida = null;
         SSLServerSocket socketServidor = null;
         Socket connRecibida = null;
         try {
                    socketServidor=this.getServerSocketSeguro(puerto);
               while (true){
                    try {
                       connRecibida=socketServidor.accept();
                       System.out.println("Conexion segura recibida");
                       entrada=
                           new BufferedReader(
                           new
InputStreamReader(connRecibida.getInputStream()));
                       salida=
                           new PrintWriter(
                               new OutputStreamWriter(
                               connRecibida.getOutputStream()));
                       String linea=entrada.readLine();
                       salida.println(linea.length());
                       salida.flush();
                    }
                    catch(IOException e) {
                          e.printStackTrace();
```

```
                        throw e;
                    }
                    finally {
                        if (null != connRecibida) {
                            connRecibida.close();
                        }
                        if (null != entrada) {
                            entrada.close();
                        }
                        if (null != salida) {
                            salida.close();
                        }
                    }
                }
            }
            catch(KeyStoreException | NoSuchAlgorithmException |
CertificateException | IOException | KeyManagementException |
UnrecoverableKeyException e) {
                    e.printStackTrace();
                    throw e;
                }
                finally {
                    if (null != socketServidor) {
                        socketServidor.close();
                    }
                }
        }

          public static void main(String[] args) throws Exception {
                String rutaAlmacen = null;
                String claveAlmacen = null;
                ServidorSeguro servidor = new ServidorSeguro(rutaAlmacen,
claveAlmacen);
                servidor.escuchar(9876);
          }
}
```

En el cliente se tienen que dar algunos pasos parecidos:

1. En primer lugar, se carga el almacén de claves del cliente (que contiene el certificado del servidor y que es la clave para poder «autenticar» el servidor)
2. El almacén del cliente se usará para crear un «gestor de confianza» (`TrustManager`) que Java usará para determinar si puede confiar o no en una conexión. Usaremos un `TrustManagerFactory` que usará el almacén del cliente para crear objetos que puedan gestionar la confianza.
3. Se creará un contexto SSL (`SSLContext`) que se basará en los `TrustManager` que pueda crear la fábrica.
4. A partir del contexto SSL el cliente ya puede crear un socket seguro (`SSLSocket`) que puede usar para conectar con el servidor de forma segura.

El código del cliente sería algo así:

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.UnknownHostException;
import java.security.KeyManagementException;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.CertificateException;

import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
import javax.net.ssl.TrustManagerFactory;

public class ClienteServidorSeguro {

	String almacen = "/home/usuario/clavescliente";
	String clave = "abcdabcd";
	SSLSocket conexion;

	public ClienteServidorSeguro(String ip, int puerto)
				throws UnknownHostException, IOException,
				KeyManagementException, NoSuchAlgorithmException,
				KeyStoreException, CertificateException{

		conexion=this.obtenerSocket(ip,puerto);
    }

	/*
	 * Envía un mensaje de prueba para verificar que la conexión SSL es
correcta
	 */
	public void conectar() throws IOException {
		InputStreamReader isr = null;
		BufferedReader entrada = null;
		OutputStreamWriter osw = null;
		PrintWriter salida = null;
		try  {
			System.out.println("Iniciando..");
			isr = new InputStreamReader(conexion.getInputStream());
			entrada = new BufferedReader(isr);
			osw = new OutputStreamWriter(conexion.getOutputStream());
			salida = new PrintWriter(osw);
			/* De esta linea se intenta averiguar la longitud */
			salida.println("1234567890");
			salida.flush();

			/* Si todo va bien, el servidor nos contesta el numero */
			String num = entrada.readLine();
			int longitud = Integer.parseInt(num);

			System.out.println("La longitud devuelta es:" + longitud);
		}
```

```
			catch(IOException e) {
				e.printStackTrace();
				throw e;
			}
			finally {
				if (null != salida) {
					salida.close();
				}
				if (null != osw) {
					osw.close();
				}
				if (null != entrada) {
					entrada.close();
				}
				if (null != isr) {
					isr.close();
				}
			}
		}

		private SSLSocket obtenerSocket(String ip, int puerto) throws
KeyStoreException, NoSuchAlgorithmException,
				CertificateException, IOException, KeyManagementException
{
			System.out.println("Obteniendo socket");
			SSLSocket socket = null;
			FileInputStream ficheroAlmacenClaves = null;
			try {
				/*
				 * Paso 1: se carga al almacén de claves (que recordemos
debe contener el
				 * certificado del servidor)
				 */
				KeyStore almacenCliente =
KeyStore.getInstance(KeyStore.getDefaultType());
				ficheroAlmacenClaves = new FileInputStream(this.almacen);
				almacenCliente.load(ficheroAlmacenClaves,
clave.toCharArray());
				System.out.println("Almacen cargado");
				/*
				 * Paso 2, crearemos una fabrica de gestores de confianza
que use el almacén
				 * cargado antes (que contiene el certificado del
servidor)
				 */
				TrustManagerFactory fabricaGestoresConfianza =
TrustManagerFactory

		.getInstance(TrustManagerFactory.getDefaultAlgorithm());
				fabricaGestoresConfianza.init(almacenCliente);
				System.out.println("Fabrica Trust creada");
				/*
				 * Paso 3: se crea el contexto SSL, que ofrezca soporte al
algoritmo TLS
				 */
				SSLContext contexto = SSLContext.getInstance("TLS");
				contexto.init(null,
fabricaGestoresConfianza.getTrustManagers(), null);
```

```
                    /* Paso 4: Se crea un socket que conecte con el servidor
*/
                    System.out.println("Contexto creado");
                    SSLSocketFactory fabricaSockets =
contexto.getSocketFactory();
                    socket = (SSLSocket) fabricaSockets.createSocket(ip,
puerto);
                    /* Y devolvemos el socket */
                    System.out.println("Socket creado");
             }
             catch(KeyStoreException | NoSuchAlgorithmException |
                    CertificateException | IOException |
KeyManagementException e) {
                    e.printStackTrace();
                    throw e;
             }
             finally {
                    if (null != ficheroAlmacenClaves) {
                           ficheroAlmacenClaves.close();
                    }
             }
             return socket;
      }

      public static void main(String[] args) throws Exception {
             ClienteServidorSeguro cliente = new
ClienteServidorSeguro("localhost", 9876);
             cliente.conectar();

      }

}
```

Firmado de aplicaciones

Utilizando la criptografía de clave pública es posible «firmar» aplicaciones. El firmado es un mecanismo que permite al usuario de una aplicación el verificar que la aplicación no ha sido alterada desde que el programador la creó (virus o programas malignos, personal descontento con la empresa, etc...).

Antes de efectuar el firmado se debe disponer de un par de claves generadas con la herramienta `keytool` mencionada anteriormente. Supongamos que el almacén de claves está creado y que en él hay uno o varios *alias* creados. El proceso de firmado es el siguiente:

1. Crear la aplicación, que puede estar formada por un conjunto de clases pero que en última instancia tendrá un `main`.
2. Empaquetar la aplicación con `jar cfe Aplicacion.jar Aplicacion Aplicación.class`. Este comando crea un fichero (`f`) JAR en el cual el punto de entrada (`e`) es la clase `Aplicacion` (que es la que tendrá el `main`).
3. Puede comprobarse que la aplicación dentro del JAR se ejecuta correctamente con `java -jar Aplicacion.jar`.
4. Ahora se puede ejecutar `jarsigner -keystore <ruta-almacen> Aplicacion.jar <alias>`.

Con estos pasos se tiene una aplicación firmada que el usuario puede verificar si así lo desea. De hecho, si se extrae el contenido del JAR con `jar -xf Aplicacion.jar` se extraen los archivos `.class` y un fichero `META-INF/Manifest` que se puede abrir con un editor para ver que realmente está firmado.

Para que otras personas puedan comprobar que nuestra aplicación es correcta los programadores deberemos exportar un certificado que los usuarios puedan importar para hacer el verificado. Recordemos que el comando es:

```
keytool -exportcert -keystore ..\Almacen.store -file
Programador.cer -alias Programador
```

Verificado de aplicaciones

Si ahora otro usuario desea ejecutar nuestra aplicación deberá importar nuestro certificado. El proceso de verificado es simple:

1. El usuario importa el certificado.
2. Ahora que tiene el certificado puede comprobar la aplicación con `jarsigner -verify -keystore <ruta-almacen> Aplicacion.jar <alias_del_programador>`

El comando deberá responder con algo como `jar verified`. Sin embargo, si no tenemos un certificado firmado por alguna autoridad de certificación (CA) la herramienta se quejará de que algunos criterios de seguridad no se cumplen.

Ejercicio

Intenta extraer el archivo JAR y reemplaza el `.class` por alguna otra clase. Vuelve a crear el archivo .JAR y vuelve a intentar verificarlo, ¿qué ocurre?

Recordatorio

Hemos hecho el proceso de firmado y verificado con **certificados autofirmados**, lo cual es útil para practicar, pero **completamente inútil desde el punto de vista de la seguridad**. Para que un certificado sea seguro debemos hacer que previamente alguna autoridad de certificación nos lo firme primero (para lo cual suele ser habitual el tener que pagar).

Política de seguridad.

Java incluye un mecanismo para definir *políticas de seguridad*. La definición oficial de Java para una política es *objeto que especifica qué permisos están disponibles para el código en función de su origen y del usuario con el que se ejecutan*. Este origen puede ser los diversos directorios del sistema operativo o incluso direcciones URL.

Cabe destacar que **todo lo que se menciona aquí no funciona si el usuario tiene acceso al sistema y puede ejecutar el intérprete de Java sin restricciones**. Es decir, se necesita el trabajo de un administrador de sistemas para restringir la manera en la que el usuario ejecuta el código.

Supongamos entonces que estamos en un entorno seguro donde los programas Java se ejecutan utilizando un gestor de seguridad, es decir, se lanzan ejecutando `java -Djava.security.manager Clase`. En principio, los programas no podrán hacer muchas cosas, como, por ejemplo, conectarse a Internet o leer un fichero que no esté en el mismo directorio de la clase.