UNIDAD 5
Técnicas de Programación Segura

1 INTRODUCCIÓN 2

2 PRÁCTICAS DE PROGRAMACIÓN SEGURA 2

3 TÉCNICAS DE SEGURIDAD. 5

3.1 CRIPTOGRAFÍA 5
3.2 FIRMA DIGITAL 7
3.3 CERTIFICADOS DIGITALES 9
3.4 CONTROL DE ACCESO 10

4 SEGURIDAD EN EL ENTORNO JAVA 11

4.1 FICHEROS DE POLÍTICAS DE SEGURIDAD 13

5 CRIPTOGRAFÍA CON JAVA 16

5.1 HERRAMIENTAS DE SEGURIDAD DE LA PLATAFORMA JAVA 17
5.2 FUNCIONES HASH 20
5.3 CLASES E INTERFACES PARA REPRESENTACIÓN DE CLAVES CRIPTOGRÁFICAS 22
5.3.1 Generación de claves para cifrado asimétrico 22
5.3.2 Almacenamiento de claves públicas y privadas en ficheros 23
5.4 GENERACIÓN Y VERIFICACIÓN DE FIRMAS DIGITALES 24
5.5 CIFRADO 26
5.5.1 Cifrado simétrico 27
5.5.2 Almacenamiento de claves simétricas en ficheros 28
5.5.3 Cifrado asimétrico 29
5.5.4 Cifrar y descifrar flujos de datos 31

6 COMUNICACIONES SEGURAS CON JAVA. JSSE 32

6.1 SSLSOCKET Y SSLSERVERSOCKET 32

7 CONTROL DE ACCESO CON JAVA. JASS 34

7.1 AUTENTICACIÓN 35
7.1.1 Código de EjemploJaasAutenticacion.java: 36
7.1.2 Código de MyCallbackHandler.java 37
7.1.3 Código de EjemploLoginModulejava 38
7.2 AUTORIZACIÓN 39
7.2.1 Código de EjemploAutenticacionJAAS 41
7.2.2 Código de EjemploLoginModule.java: 41
7.2.3 Código de EjemploPrincipal.java 42
7.2.4 Código de EjemploAcción.java: 43

1 Introducción

Empezaremos esta unidad estableciendo las pautas que llevan al desarrollo de programas seguros. A continuación, se presentarán algunas de las técnicas más importantes para garantizar la seguridad de sistemas y aplicaciones: criptografía, certificados digitales y control de acceso.

Después se tratarán los recursos que proporciona la plataforma Java para el desarrollo de aplicaciones seguras, comenzando con el estudio de los ficheros de políticas de seguridad, seguido del estudio de diferentes librerías estándar para la implementación de la seguridad: librerías criptográficas (JCA y JCE), la extensión de sockets seguros (JSSE) y el servicio de autenticación y autorización de Java (JAAS).

2 Prácticas de programación segura

Es habitual que los consumidores de productos software reciban por parte de los vendedores avisos de parches de seguridad de sus productos para corregir agujeros de seguridad. Estos son especialmente frecuentes en aplicaciones que interactúan con Internet, un ejemplo son los parches de seguridad que publica Microsoft para corregir vulnerabilidades graves detectadas en su navegador web.

En este apartado se expone una lista de buenas prácticas que nos servirán de ayuda a la hora de desarrollar aplicaciones seguras:

1. **Informarse**:
    - Una forma de evitar fallos es estudiar y comprender los errores que otros hayan cometido a la hora de desarrollar software.
    - Internet es el hogar de una gran variedad de foros públicos donde se debaten con frecuencia problemas de vulnerabilidad de software.
    - Leer libros y artículos sobre prácticas de codificación segura y análisis de los defectos del software.
    - Explorar el software de código abierto, donde podemos encontrar cantidad de ejemplos de buenas prácticas, pero también numerosos ejemplos de cómo no se deben hacer las cosas.
2. **Precaución en el manejo de datos:** La mayoría de los programas aceptan diferentes tipos de datos de entrada que pueden provenir de múltiples fuentes, y el programador debe establecer los mecanismos que permitan verificar esos datos realizando tareas como:
    - Examinar los datos de entrada para evitar que un atacante pueda alterarlos con fines maliciosos.
    - Realizar la comprobación de límites. Un problema muy típico es el desbordamiento de búfer. Hemos de aseguramos de verificar que los datos proporcionados al programa pueden caber en el espacio que se asigna para ello. En los arrays hemos de revisar los índices para garantizar que permanecen dentro de sus límites.
    - Revisar los ficheros de configuración. Es necesario validar los datos, como si se tratase de una entrada de datos por teclado, ya que pueden ser manipulados por un atacante.
    - Comprobar los parámetros de línea de comandos.
    - No fiarse de las URLs web. Muchos diseñadores de aplicaciones web utilizan URLs para insertar variables y sus valores. El usuario puede alterar la URL directamente dentro de su navegador por variables de ajuste y/o de sus valores a cualquier configuración que elija. Si la aplicación web no realiza una comprobación de los datos puede ser atacada con éxito.

- Cuidado con los contenidos web. Muchas aplicaciones web insertan variables en campos HTML ocultos. Tales campos también pueden ser modificados por el usuario en una sesión de navegador.
- Comprobar las cookies web. Los valores pueden ser modificados por el usuario final y no se debe confiar en ellas.
- Comprobar las variables de entorno. Un uso común de las variables de entorno es pasar configuración de preferencias a los programas. Los atacantes pueden proporcionar variables de entorno no previstas por el programador.
- Establecer valores iniciales válidos para los datos. Es un buen hábito inicializar correctamente las variables.
- Comprender las referencias de nombre de fichero (rutas de acceso de ficheros y directorios) y utilizarlas correctamente dentro de los programas.
- Especial atención al almacenamiento de información sensible. Es de vital importancia proteger la confidencialidad e integridad de información considerada como confidencial, como contraseñas, números de cuenta de tarjetas de crédito, etc.

3. **Reutilización de código seguro siempre que sea posible**, es decir, aquel que ha sido completamente revisado y probado, y ha resistido las pruebas del tiempo y de los usuarios.
4. **Insistir en la revisión de los procesos.** Siempre es aconsejable seguir una práctica de revisión de los fallos de seguridad en el código fuente. Si un programa es confiado a varias personas, todas deben participar en la revisión. Algunas prácticas comúnmente utilizadas son:
    - Realizar una revisión por pares (dos o más revisores). Para entornos de desarrollo relativamente informales, un proceso de revisión de código por pares puede ser suficiente. Es recomendable el desarrollo de una lista de cosas que buscar, esta tiene que ser mantenida y actualizada con los nuevos fallos de programación encontrados.
    - Realizar una validación y verificación independiente. Algunos proyectos de programación necesitan una revisión más formal que implica revisar el código fuente de un programa, línea por línea, para garantizar que se ajusta a su diseño, así como a otros criterios (por ejemplo, las condiciones de seguridad).
    - Identificar y utilizar las herramientas de seguridad disponibles. Hay herramientas de software disponibles para ayudar en la revisión de fallos en el código fuente. Son útiles para la captura de errores comunes, pero no tan útiles para detectar cualquier otro error.
5. **Utilizar listas de control de seguridad.** Estas listas pueden ser muy útiles para asegurarse de que se han cubierto todas las fases durante la ejecución. Esto sería un ejemplo de una lista de control:
    - Este sistema de aplicación requiere una contraseña para que los usuarios puedan acceder.
    - Todos los inicios de sesión de usuario son únicos.
    - Esta aplicación utiliza el sistema de control de acceso basado en roles.
    - Las contraseñas no se transmiten a través de la red en texto plano.
    - El cifrado se utiliza para proteger los datos que se transfieren entre servidores y clientes.

6. **Ser amable con los mantenedores.** El mantenimiento del código puede ser de vital importancia para la seguridad del software en el transcurso de su vida útil. Seguiremos estas prácticas de mantenimiento del código:
    - Utilizar normas. Se pueden tener normas con respecto a cosas como la documentación en línea del código fuente, la selección de los nombres de las variables, etc; para que hagan la vida más fácil para aquellos que posteriormente mantendrán el código. El código modular, que está bien documentado y es fácil de seguir es más fácil de mantener.
    - Retirar código obsoleto. Si el código no es necesario, se recomienda quitarlo.
    - Analizar todos los cambios en el código. Hemos de aseguramos de probar a fondo los cambios en el código antes de entrar en producción.

Es muy importante hacer una lista de las cosas que se deben hacer a la hora de aplicar código seguro en las aplicaciones que hagamos, igualmente importante es hacer otra lista con las **cosas que NO se deben hacer**:

- Escribir código que utiliza nombres de fichero relativos. La referencia a un nombre de fichero debe ser completa.
- Referirse dos veces en el mismo programa a un fichero por su nombre. Se recomienda abrir el fichero una vez por su nombre y utilizar el identificador a partir de entonces.
- Invocar programas no confiables dentro de los programas.
- Asumir que los usuarios no son maliciosos.
- Dar por sentado el éxito. Cada vez que se realiza una llamada al sistema (por ejemplo, abrir un fichero, leer un fichero, recuperar una variable de entorno), comprobar el valor de retomo por si la llamada falló.
- Invocar un shell o una línea de comandos.
- Autenticarse en criterios que no sean de confianza.
- Utilizar áreas de almacenamiento con permisos de escritura. Si es absolutamente necesario, hay que suponer que la información pueda ser manipulada, alterada o destruida por cualquier persona o proceso que así lo desee.
- Guardar datos confidenciales en una base de datos sin protección de contraseña.
- Hacer eco de las contraseñas o mostrarlas en la pantalla del usuario.
- Emitir contraseñas vía e-mail.
- Distribuir mediante programación información confidencial a través de correo electrónico.
- Guardar las contraseñas sin cifrar (o cualquier otra información confidencial) en disco en un formato fácil de leer (texto sin cifrar). Se debe utilizar en su lugar certificados, encriptación fuerte, o transmisión segura entre hosts de confianza.
- Transmitir entre los sistemas contraseñas sin encriptar (o cualquier otra información confidencial). Se debe utilizar como antes certificados, encriptación fuerte, o transmisión segura entre hosts de confianza.
- Tomar decisiones de acceso basadas en variables de entorno o parámetros de línea de comandos que se pasan en tiempo de ejecución.
- Evitar, en la medida que se pueda, confiar en el software o los servicios de terceros para operaciones críticas.

3 Técnicas de seguridad.

A continuación, se presentan algunas de las técnicas y mecanismos más importantes para garantizar la seguridad de sistemas y aplicaciones: la criptografía, la generación y uso de certificados digitales y el control de acceso.

3.1 Criptografía

El término criptografía es un derivado de la palabra griega kryptos, que significa «oculto». El objetivo de la criptografía es ocultar el significado de un mensaje mediante un proceso que recibe el nombre de cifrado.

El proceso general de cifrado y descifrado de mensajes se muestra en la Figura 1:

- Si a un texto legible se le aplica un algoritmo de cifrado, que en general depende de una clave, esto arroja como resultado un texto cifrado que es el que se envía o guarda. A este proceso se le llama cifrado.
- Si a ese texto cifrado se le aplica el mismo algoritmo, dependiente de la misma clave o de otra clave (esto depende del algoritmo), se obtiene el texto legible original. A este segundo proceso se le llama descifrado.

Figura 1. Proceso general de cifrado y descifrado de mensajes.

Existen 3 clases de procesos criptográficos:

1. **Funciones de una sola vía (funciones hash):** Permiten mantener la integridad de los datos, tanto en almacenamiento como en su envío a través de una red. También se utilizan como parte de los mecanismos de firma digital. Su utilidad se basa en que dado un mensaje x, es muy fácil calcular el *hash de x*, también conocido como *resumen* o *message digest*, siendo prácticamente imposible calcular x a partir de su resumen, Figura 2. Las funciones de una sola vía tienen un amplio abanico de usos en la seguridad informática. Prácticamente cualquier protocolo las usa para procesar claves, encadenar una secuencia de eventos, o incluso autenticar eventos y son esenciales en la autenticación por firmas digitales. Entre las funciones hash más utilizadas en la actualidad se encuentran las variantes de SHA-2 o las funciones KDF (*Key Derivation Function*).

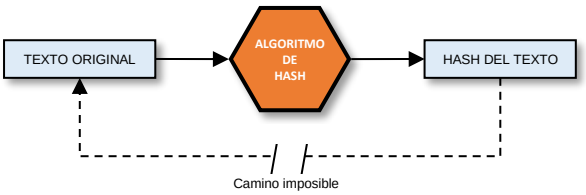


Figura 2. Función de una sola vía.

2. **Criptografía de clave simétrica:** Se basa en algoritmos criptográficos que utilizan una misma clave para cifrar y descifrar conocida como *clave simétrica* o *clave secreta*. Estos algoritmos son seguros y altamente eficientes cuando se usan de manera adecuada, de modo que pueden usarse para cifrar grandes cantidades de datos sin tener un efecto negativo en el rendimiento. Entre los algoritmos de cifrado simétrico que se usan en la actualidad cabe mencionar AES (*Advanced Encryption Standard*) que opera con tamaños de clave de 128, 192 y 256 bits y ChaCha20 que opera con tamaños de clave de 128 y 256 bits.

Figura 3. Criptografía simétrica o de clave privada.

3. **Criptografía de clave pública o asimétrica**: Se basa en la utilización de un par de claves relacionadas entre sí: una clave publica que estará a disposición de todo el mundo y una clave privada asociada que sólo estará a disposición de su propietario. Los algoritmos de cifrado asimétrico funcionan bajo la premisa de que lo que se cifra con la clave pública, solo puede descifrarse con su clave privada asociada y viceversa. Veamos su funcionamiento con un ejemplo: supongamos la situación representada en la Figura 4, en la que un sujeto **A** tiene que enviar por red un mensaje cifrado a un sujeto **B**, de forma que sólo este último pueda descifrarlo. Previamente a la transmisión del mensaje habrán ocurrido dos cosas:

    1. **B** habrá generado su par de claves y habrá difundido la clave pública.
    2. **A** habrá obtenido la clave pública que **B** ha difundido.

    Ahora **A** usará la clave pública de **B** para cifrar el mensaje y se lo enviará a **B** con la seguridad de que sólo este último puede descifrarlo con su clave privada.

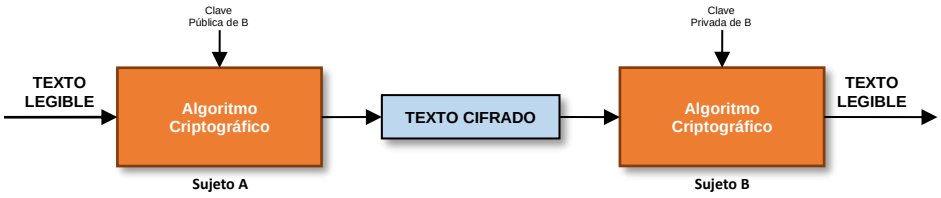


Figura 4. Criptografía asimétrica o de clave pública.

Existen múltiples formas de generar y almacenar los pares de claves para cifrado. El algoritmo asimétrico más popular es el RSA que debe su nombre a sus inventores Rivest, Shamir y Adleman. Su uso es prácticamente universal como método de autenticación y firma digital y es componente de protocolos y sistemas como IPSec (Internet Protocol Security), SSL (Secure Sockets Layer), PGP (Pretty Good Privacy), etc.

La tabla siguiente permite comparar criptografía simétrica y asimétrica exponiendo los puntos fuertes y débiles de cada una de ellas:

CRIPTOGRAFÍA SIMÉTRICA O DE CLAVE SECRETA

Puntos Fuertes	Puntos Débiles
• Cifran más rápido que los algoritmos de clave pública. • Sirven habitualmente como base para los sistemas criptográficos basados en hardware.	• Requieren un sistema de distribución de claves muy seguro que obliga a realizar una administración compleja. • La seguridad desaparece cuando la clave cae en manos no autorizadas. • Si se asume que es necesaria una clave por cada pareja de usuarios de una red, el número total de claves crece rápidamente con el aumento del número de usuarios.

CRIPTOGRAFIA ASIMETRICA O DE CLAVE PUBLICA

Puntos Fuertes	Puntos Débiles
• Permiten conseguir autenticación y no repudio para muchos protocolos criptográficos. • Suelen emplearse en colaboración con cualquiera de los otros métodos criptográficos. • Permiten tener una administración sencilla de claves al no necesitar que haya intercambio de claves seguro.	• Son algoritmos más lentos que los de clave secreta, con lo que no suelen utilizarse para cifrar gran cantidad de datos. • Sus implementaciones son comúnmente hechas en sistemas software. • Para una gran red de usuarios y/o máquinas se requiere un sistema de certificación de la autenticidad de las claves públicas.

Podemos decir que la criptografía juega 3 papeles principales en la implementación de sistemas seguros:

- Se emplea para mantener el secreto y la integridad de la información donde quiera que pueda estar expuesta a ataques.
- Como base de los mecanismos para autenticar la comunicación entre pares de principales (un principal puede ser un usuario o un proceso).
- Para implementar el mecanismo de firma digital.

3.2 Firma digital

Una firma digital es un conjunto de datos asociados a un documento electrónico que identifican al firmante dotando de validez legal al documento. Las funciones básicas de una firma digital son:

- Identificar al autor verificando que es quien dice ser.
- Verificar la integridad del documento verificando que no ha sido modificado después de ser firmado.
- Garantizar el no repudio en el origen.

Una firma digital reconocida debe cumplir además los requisitos siguientes:

- Identificar al firmante de forma inequívoca.
- Garantizar el no repudio en el origen.
- Contar con la participación de un tercero de confianza.
- Estar basada en un certificado electrónico reconocido.

La base legal de la Firma Electrónica está recogida en la Ley 59/2003 de Firma Electrónica y se desarrolla en más profundidad en la sección Base legal de las Firmas. La sección también explora, bajo qué circunstancias la ley equipara la firma electrónica a la firma manuscrita, añade notas respecto a la normativa europea y hace distintas referencias legales a firmas con sellos de tiempo y avanzadas.

La creación de una firma digital está basada en la criptografía de clave pública, también conocida como criptografía asimétrica, que usa pares de claves pública/privada para llevar a cabo procesos de cifrado y autenticación.

El método de firma digital más extendido es el RSA. En este modelo, el procedimiento de firma de un mensaje por parte del emisor se muestra en la Figura 5:

1. El emisor genera un resumen del mensaje mediante una función acordada. A este resumen le llamamos H1.
2. H1 es cifrado con la clave privada del emisor. El resultado es lo que se conoce como firma digital, que se envía adjunta al mensaje.
3. El emisor envía el mensaje y su firma al receptor.

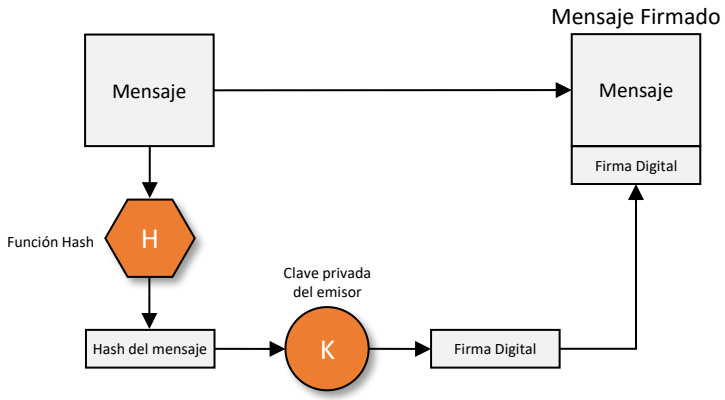


Figura 5. Firma digital de un mensaje.

El receptor realiza las siguientes operaciones:

- Separa el mensaje de la firma.
- Genera el resumen H2 del mensaje recibido usando la misma función que el emisor.
- Descifra la firma FD mediante la clave pública del emisor obteniendo el hash original H1.
- Si los dos resúmenes coinciden se puede afirmar que el mensaje ha sido enviado por el titular de la clave pública y que no fue modificado en el transcurso de la comunicación.

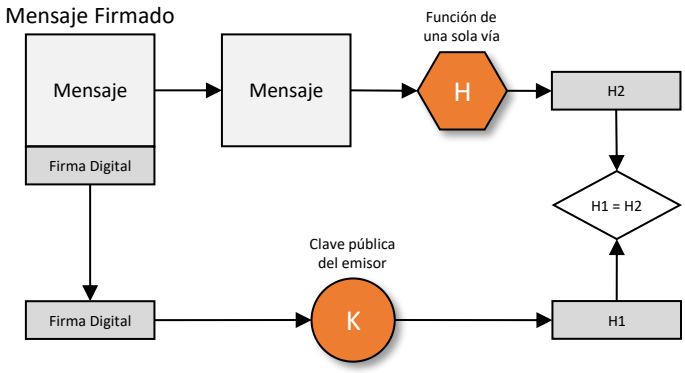


Figura 6. Comprobación en el receptor del mensaje firmado.

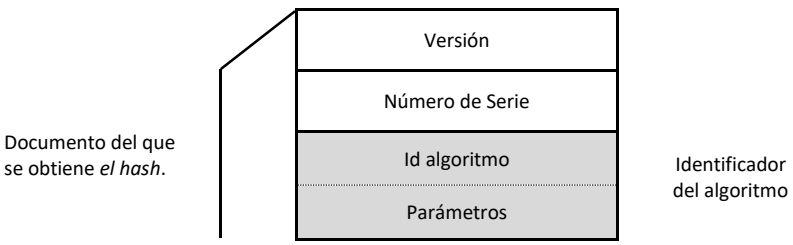
Todo sistema criptográfico de firma digital descansa sobre un pilar fundamental: la autenticidad de la clave pública de cada participante. Para asegurar la autenticidad de la clave pública de cada participante se recurre a las Autoridades de Certificación (AC), que garantizan que una firma es de quien dice ser.

3.3 Certificados digitales

Un certificado de clave pública es un tipo de documento digital que vincula los datos de su titular con una clave pública. Para que ese vínculo quede legalmente establecido, el certificado tiene que estar firmado digitalmente por una Autoridad de Certificación o CA (*Certification Authority*). En criptografía, una CA es una entidad de confianza, responsable de emitir y revocar certificados digitales usando criptografía de clave pública. Jurídicamente es un caso particular de Prestador de Servicios de Certificación, con autoridad para garantizar que cualquier uso de la clave privada se puede vincular de forma inequívoca al titular del certificado asociado.

Un certificado digital tiene un formato estándar universalmente aceptado, conocido como X.509. Tiene los siguientes campos:

- Versión: indica la versión del formato del certificado, normalmente X.509v3.
- Número de serie: identificador numérico único dentro del dominio de la AC.
- Algoritmo de firma y parámetros: identifican el algoritmo asimétrico y la función de una sola vía que se usa para firmar el certificado.
- Emisor del certificado: el nombre X.500 de la AC.
- Fechas de inicio y final de validez: determinan el periodo de validez del certificado.
- Nombre del propietario de la clave pública que se está firmando.
- Identificador del algoritmo que se está utilizando, la clave pública del usuario y otros parámetros si son necesarios.
- La firma digital de la AC, es decir, el resultado de cifrar mediante el algoritmo asimétrico y la clave privada de la AC, el hash obtenido del documento X.509.



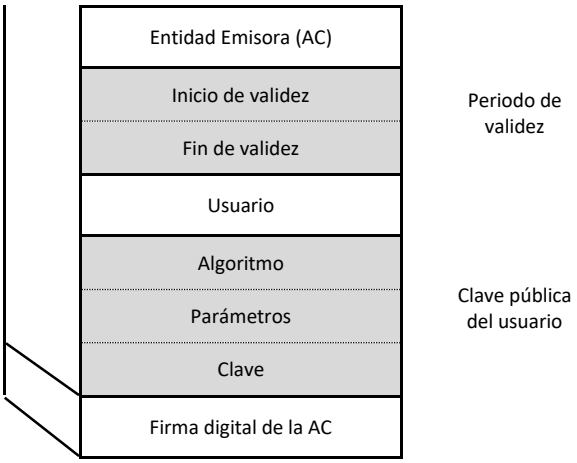


Figura 7. Formato X509.

Las principales aplicaciones de los certificados digitales son:

- Autentificar la identidad del usuario, de forma electrónica, ante terceros.
- Trámites electrónicos ante la Agencia Tributaria, la Seguridad Social, las Cámaras y otros organismos públicos.
- Trabajar con facturas electrónicas.
- Firmar digitalmente e-mail y todo tipo de documentos.
- Cifrar datos para que solo el destinatario del documento pueda acceder a su contenido.

Algunas AC españolas que emiten certificados electrónicos de empresa son: Fábrica Nacional de Moneda y Timbre (FNMT), Agencia Catalana de Certificació (CATCert), Agencia Notarial de Certificación (ANCERT), ANF Autoridad de Certificación (ANF AC) o la Autoritat de Certificació de la Comunitat Valenciana (ACCV).

Los certificados digitales se pueden solicitar a través de la aplicación web de la AC. Por ejemplo, una persona física (no persona jurídica) para solicitar un certificado a la Fábrica Nacional de Moneda y Timbre (FNMT), accede a la web www.ceres.fnmt.es y sigue una serie de pasos: solicitud del certificado (a través de la web), acreditación de la identidad mediante personación física en una oficina de registro y descarga del certificado desde Internet. Las claves criptográficas se generan en el momento de la solicitud del certificado y quedan unidas inequívocamente al titular de las mismas.

3.4 Control de acceso

En general, cualquier empresa u organización tiene diversos tipos de recursos de carácter privado a los que solo tienen acceso las personas autorizadas. Es necesario, por tanto, llevar a cabo un control de acceso a los recursos en el que normalmente intervienen varios procesos:

- Identificación: proceso mediante el cual el sujeto suministra información diciendo quién es.
- Autenticación: es cualquier proceso por el cual se verifica que alguien es quien dice ser. Esto implica generalmente un nombre de usuario y una contraseña, pero puede incluir cualquier otro método para demostrar la identidad, como una tarjeta inteligente, exploración de la retina, reconocimiento de voz o las huellas dactilares.
- Autorización: es el proceso de determinar, una vez autenticado el sujeto, a qué recursos tiene acceso.

El sistema de control de acceso debe permitir el acceso al usuario correctamente autenticado y debe impedir el acceso a los demás. Debería también guardar un buen registro de auditoría de todos los accesos autorizados, así como de los intentos de acceso fallidos.

Las medidas de identificación y autenticación se centran en una de estas tres formas:

- Algo que se sabe, algo que se conoce: habitualmente se trata de **contraseñas**.
- Algo que se es: medidas que utilizan la **biometría** (identificación por medio de la voz, la retina, la huella dactilar, geometría de la mano, etc).
- Algo que se tiene: los ***acces tokens*** (sistemas de tarjetas).

4 Seguridad en el entorno java

Antes de que la máquina virtual Java comience el proceso de ejecución de una aplicación, debe realizar una serie de tareas para preparar el entorno en el que el programa se ejecutará. Este es el punto en el que se implementa la seguridad interna de Java. Hay tres componentes en el proceso:

- **El cargador de clases**. Es el responsable de encontrar y cargar los bytecodes almacenados en los archivos *.class*. Cada programa Java tiene como mínimo tres cargadores:
  - El cargador de clases bootstrap que carga las clases del sistema, desde el fichero *$JAVA_HOME/lib/rt.jar* antes de la versión 9 de java, y a partir de esta, desde el fichero *$JAVA_HOME/lib/modules*.
  - El cargador de clases de extensión que carga una extensión estándar desde el directorio *$JAVA_HOME/lib/ext*
  - El cargador de clases de la aplicación que localiza las clases y los ficheros *JAR/ZIP* a partir de las rutas de acceso establecidas en la variable de entorno *CLASSPATH* o en la opción *-classpath* de la línea de comandos.
- **El verificador de ficheros de clases**. Se encarga de validar los bytecodes. Algunas de las comprobaciones que lleva a cabo son: que las variables estén inicializadas antes de ser utilizadas, que las llamadas a un método coinciden con los tipos de referencias de objeto, que no se han infringido las reglas para el acceso a los métodos y clases privados, etc.
- **El gestor de seguridad**. Permite o deniega la realización de operaciones como la carga de subprocesos desde el hilo actual, el acceso a paquetes específicos, el acceso o modificación de propiedades del sistema, la lectura, escritura o eliminación de ficheros específicos, la conexión mediante sockets con hosts y números de puerto específicos, etc.

Cuando se ejecuta una aplicación Java, no se instala ningún gestor de seguridad por defecto. Para hacerlo tendremos que especificar el parámetro `-Djava.security.manager` al invocar a la JVM o invocar al método `System.setSecurityManager()` desde el código Java de la aplicación.

A continuación, veremos un ejemplo de un programa que accede a ciertas propiedades de sistema invocando al método `System.getProperty(String property)` para mostrar por pantalla el valor que tienen asignado:

```
public class Ejemplo1 {
    public static void main(String[] args) {
        String t[] = {"java.class.path", "java.home", "java.vendor", "java.version",
            "os.name", "os.version", "user.dir", "user.home", "user.name"};
        System.out.println("PROPIEDAD                VALOR");
        for (int i = 0; i < t.length; i++)
            System.out.format("%-20s ==>  %s\n", t[i], System.getProperty(t[i]));
    }
}
```

La ejecución del código anterior muestra la salida siguiente:

```
PROPIEDAD                 VALOR
java.class.path      ==>  C:\Users\usuario\git\ejemplospsp\target\classes
java.home            ==>  C:\Program Files\Java\jre1.8.0_321
java.vendor          ==>  Oracle Corporation
java.version         ==>  1.8.0_321
os.name              ==>  Windows 10
os.version           ==>  10.0
user.dir             ==>  C:\Users\usuario\ejemplospsp
user.home            ==>  C:\Users\usuario
user.name            ==>  juliolb
```

A continuación, se muestra el ejemplo anterior modificado para instalar un gestor de seguridad:

```
public class Ejemplo2 {
    public static void main(String[] args) {
        String t[] = {
            "java.class.path", "java.home", "java.vendor", "java.version",
            "os.name", "os.version", "user.dir", "user.home", "user.name"
        };
        System.setSecurityManager(new SecurityManager());
        System.out.println("PROPIEDAD                 VALOR");
        for (int i = 0; i < t.length; i++)
            try {
                System.out.format("%-20s ==>  %s\n", t[i], System.getProperty(t[i]));
            } catch (AccessControlException e) {
                System.out.println(e.getLocalizedMessage());
            }
    }
}
```

En la salida que genera el programa se puede observar cómo se deniega el acceso a ciertas propiedades del sistema:

```
PROPIEDAD                 VALOR
access denied ("java.util.PropertyPermission" "java.class.path" "read")
access denied ("java.util.PropertyPermission" "java.home" "read")
java.vendor          ==>  Oracle Corporation
java.version         ==>  1.8.0_321
os.name              ==>  Windows 10
os.version           ==>  10.0
access denied ("java.util.PropertyPermission" "user.dir" "read")
access denied ("java.util.PropertyPermission" "user.home" "read")
access denied ("java.util.PropertyPermission" "user.name" "read")
```

Al ejecutar un programa Java se carga por defecto un fichero de políticas predeterminado que otorga permisos al código para acceder solamente a aquellas propiedades para las que dicho acceso no representa un riesgo para la seguridad. En cambio, el fichero de políticas deniega el acceso a otras propiedades como `user.home` y `java.home`, por lo que cualquier intento de acceder a ellas provoca el lanzamiento de la excepción `AccessControlException`.

> En https://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html se publica la referencia de los tipos de permisos, sus destinos y las acciones que se pueden permitir para cada uno de ellos.

La plataforma Java incluye un conjunto de APIs que abarca las principales áreas de seguridad, incluyendo la criptografía, la infraestructura de clave pública, la autenticación, la comunicación segura y el control de acceso, permitiendo a los desarrolladores integrar fácilmente la seguridad en sus aplicaciones. Las que trataremos en esta unidad son:

- **JCA** (*Java Cryptography Architecture*). Es un framework que proporciona la infraestructura necesaria para el acceso a los principales servicios criptográficos, incluyendo firmas digitales, resumen de mensajes, creación y validación de certificados, cifrado simétrico y asimétrico, gestión de claves y generación segura de números aleatorios.
- **JSSE** (*Java Secure Socket Extensión*). Suministra un framework para comunicaciones seguras en Internet implementando la versión Java de los protocolos SSL y TLS e incluye funcionalidad para cifrado de datos, autenticación del servidor, integridad de mensajes y autenticación del cliente.
- **JAAS** (*Java Authentication and Authorization Service*). Es una extensión estándar del Java 2 Software Development Kit que permite a las aplicaciones Java acceder a servicios de control de acceso en los que se llevan a cabo procesos de autenticación y autorización.

4.1 Ficheros de políticas de seguridad

Los ficheros de políticas se usan para conceder permisos del sistema al código de diferentes fuentes. En el fichero `<java.home>/lib/security/java.security` se especifican las localizaciones de los ficheros de políticas. sustituir `<java.home>` por el valor que tenga esta propiedad

Por defecto, en este fichero se especifican los ficheros de políticas siguientes:

```
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
```

Estas URLs hacen referencia a ficheros locales, pero existe la posibilidad de incluir URLs que hagan referencia a ficheros de políticas que residan en servidores remotos, evitando así que puedan ser modificados por los usuarios.

El fichero `${java.home}/lib/security/java.policy` está destinado a otorgar permisos de código a nivel de sistema. La instalación del JDK crea este fichero para otorgar todos los permisos a las extensiones estándar, permitir que cualquiera escuche en puertos sin privilegios y permitir que cualquier código lea propiedades que no son sensibles a la seguridad, como las propiedades `os.name` o `file.separator`.

El fichero `${user.home}/.java.policy` no se crea por defecto. Cada usuario tendrá que crear el suyo propio si desea establecer políticas de seguridad específicas para su perfil.

Una práctica habitual consiste en especificar un fichero de políticas para una aplicación en el momento de invocar a la JVM para su ejecución:

```
java -Djava.security.manager -Djava.security.policy=URL aplicación_java
```

El parámetro `-Djava.security.manager` no es obligatorio si la aplicación instala el gestor de seguridad. Las políticas definidas en el fichero especificado en *URL* se añaden a las definidas en los ficheros de políticas especificados en el fichero `<java.home>/lib/security/java.security`. Si se desea que sólo se apliquen las definidas en fichero especificado en la línea de comando, se ha de invocar a la JVM de la forma siguiente (nótese el doble =):

```
java -Djava.security.manager -Djava.security.policy==URL aplicación_java
```

Los ficheros de políticas contienen una secuencia de entradas `grant` en las que se especifican uno o más permisos con el formato siguiente:

```
grant CodeBase "URL" {
  permission nombre_clase "Nombre_destino", "Acción";
  permission nombre_clase "Nombre_destino", "Acción";
  ...
}
```

A la derecha de `CodeBase` se indica la ubicación del código base sobre el que se van a definir los permisos, y si se omite, los permisos se aplican a cualquier código Java. Su valor es una URL y siempre se debe utilizar la barra diagonal (/) como separador de directorio incluso para las URL de tipo file en Windows. Por ejemplo:

```
grant CodeBase "file:/C:/somepath/miapp/" {
  ...
}
```

En cada permiso, *`nombre_clase`* contiene el nombre de la clase de permisos. Por ejemplo:

- `java.io.FilePermission` representa el permiso de acceso a ficheros o directorios.
- `java.net.SocketPermission` representa el permiso de acceso a la red vía sockets.
- `java.util.PropertyPermission` representa el permiso de acceso a propiedades del sistema.

El parámetro *`nombre_destino`* identifica al destino del permiso en función de la clase de permiso mediante:

- La ruta a un fichero o directorio si la clase es `java.io.FilePermission`.
- La dirección de un servidor y un número de puerto si la clase es `java.net.SocketPermission`.
- Una propiedad del sistema para la clase `java.util.PropertyPermission`.

En el parámetro Acción se indica una lista de acciones separadas por comas, por ejemplo:

- `read`, `write`, `delete` o `execute` para una clase de permiso `java.io.FilePermission`
- `accept`, `listen`, `connect`, `resolve` para una clase de permiso `java.net.SocketPermission`
- `read`, `write` para una clase de permiso `java.util.PropertyPermission`

El siguiente ejemplo muestra el contenido de un fichero llamado *ejemplo2.policy* que da permiso de lectura a propiedades del sistema para las que dicho permiso se deniega por defecto:

```
grant {
  permission java.util.PropertyPermission "java.class.path", "read";
  permission java.util.PropertyPermission "java.home", "read";
  permission java.util.PropertyPermission "user.home", "read";
  permission java.util.PropertyPermission "user.name", "read";
  permission java.util.PropertyPermission."user.dir", "read";
};
```

Para probar el efecto de este fichero de políticas y comprobar que se muestra el valor de estas propiedades, ejecutamos la clase `Ejemplo2` desde la línea de comando de la forma siguiente:

```
java -Djava.security.policy=ejemplo2.policy Ejemplo2
```

A continuación, se muestra un ejemplo de un programa Java que crea un fichero, escribe una línea y finalmente lee el contenido.

```
public class Ejemplo3 {
    public static void main(String[] args) throws IOException, AccessControlException {
        File file = new File(System.getProperty("user.home") + "/Documentos/fichero.txt");
        System.out.println("escribiendo ...");
        try (PrintWriter out = new PrintWriter(new FileWriter(file))) {
            out.println("Línea 1");
            System.out.println("leyendo ...");
            try (BufferedReader in = new BufferedReader(new FileReader(file))){
                String linea;
                while ((linea = in.readLine()) != null)
                    System.out.println(linea);
            }
        }
}
```

El comportamiento de este programa será diferente cuando se use gestor de seguridad de cuando no se use. Cuando no se usa, la salida confirma que no se aplica ninguna restricción para crear o leer el fichero:

```
escribiendo ...
leyendo ...
Línea 1
```

Si añadimos al principio del método `main` del código anterior la sentencia siguiente:

```
System.setSecurityManager(new SecurityManager());
```

El intento de escribir en el fichero provoca el lanzamiento de la excepción `AccessControlException` y, por tanto, que muestre la salida siguiente:

```
escribiendo ...
access denied ("java.io.FilePermission" "C:\Users\usuario\Documents\fichero.txt" "write")
```

Obviamente, tampoco se podría realizar la lectura de este, pero si creamos un fichero de políticas como el siguiente:

<user.home>/.java.security

```
grant codeBase "file:${user.home}/-" {
    permission java.util.PropertyPermission "user.home", "read";
    permission java.io.FilePermission "${user.home}/Documents/*", "write";
}
```

El gestor de seguridad otorgará, a las clases Java almacenadas en cualquier subdirectorio que se encuentre dentro del directorio `user.home`, el permiso para escribir en cualquier fichero almacenado en el directorio `Documents` del usuario, tal y como se puede apreciar observando la salida que ahora se obtiene al ejecutar el programa:

```
escribiendo ...
leyendo ...
access denied ("java.io.FilePermission" "C:\Users\juliolb\Documents\fichero.txt" "read")
```

Para permitir tanto la lectura como la escritura modificamos el permiso de la forma siguiente:

```
    permission java.io.FilePermission "${user.home}/Documents/*", "read, write";
```

La omisión del parámetro `CodeBase` en una entrada `grant` del fichero de políticas hace que sus permisos se otorguen a cualquier clase Java, independientemente de donde esté localizada.

Las rutas especificadas en el fichero de políticas pueden finalizar con uno de los caracteres siguientes:

- `*` Indica que se incluyen todos los ficheros contenidos en el directorio.
- `-` Indica un directorio y de forma recursiva, todos los ficheros y subdirectorios contenidos en ese directorio.

Para la clase de permiso `java.net.SocketPermission` se especifica un nombre de host o dirección IP, un puerto o rango de puertos y un conjunto de acciones: `accept`, `listen`, `connect`, y `resolve`. La sintaxis es la siguiente:

```
permission java.net.SocketPermission "host", "acciones"
```

donde

```
host = (hostname | IPaddress)[:portrange]
portrange = portnumber | -portnumber | portnumber-[portnumber]
```

A continuación, se muestran algunos ejemplos:

`permission java.net.SocketPermission "localhost:6000", "connect";`	puerto 6000
`permission java.net.SocketPermission "localhost:-6000", "connect";`	puerto 6000 y anteriores
`permission java.net.SocketPermission "localhost:6000-", "connect";`	puerto 6000 y posteriores
`permission java.net.SocketPermission "localhost:6000-7000", "connect";`	del puerto 6000 al 7000

En https://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html se describe la sintaxis que se ha de seguir para especificar cada uno de los tipos de permisos, sus destinos y las acciones correspondientes dentro de los ficheros de políticas.

El JDK proporciona la herramienta `policytool` que podemos invocar desde la línea de comando para editar los ficheros de políticas. Se recomienda usar la herramienta para editar cualquier fichero de políticas y verificar la sintaxis de su contenido. Los errores de sintaxis del fichero de políticas provocan una excepción `AccessControlException` cuando se ejecuta la aplicación.

ACTIVIDAD 1

Modifica un ejercicio cliente/servidor de la unidad 3 para que tanto cliente como servidor utilicen un gestor de seguridad y un fichero de políticas que les conceda, sólo a ellos, los permisos siguientes:

- Al cliente, permiso de lectura para conectarse con el servidor en el puerto que corresponda.
- Al servidor, permiso para escuchar en el puerto que corresponda.

5 Criptografía con java

JCA (*Java Cryptography Architecture*), basada en una arquitectura de proveedor de seguridad, está formada por un conjunto de APIs para firma digital, resumen de mensajes, gestión y validación de certificados, cifrado simétrico y asimétrico, generación y mantenimiento de claves y generación de números aleatorios, entre otros.

Las clases en interfaces principales de JCA, definidas en el paquete `java.security`, son:

- `Provider`: clase abstracta que representa el concepto de proveedor. Define métodos para acceder al nombre del proveedor, su número de versión y otros datos sobre la implementación de los algoritmos de generación conversión y gestión de claves y de generación de firmas y resúmenes.
- `Security`: clase que centraliza la gestión de todas las propiedades de seguridad y proveedores. Su método `getProviders` retorna un array de objetos `Provider` que contiene todos los proveedores registrados.
- `MessageDigest`: clase para crear generadores de resúmenes de mensajes.
- `Signature`: clase para la generación de firmas digitales.

- `AlgorithmParameter`: clase para el manejo de representaciones opacas de los parámetros criptográficos.
- `AlgorithmParameterGenerator`: clase que facilita la generación del conjunto de parámetros apropiados para diferentes algoritmos criptográficos.
- `Key`: interface para el manejo de representaciones opacas de claves criptográficas.
- `KeySpec`: interface para el manejo de representaciones transparentes de claves criptográficas.
- `KeyFactory`: clase para conversión de claves opacas de tipo `Key` en especificaciones de clave (representaciones transparentes de los parámetros criptográficos de una clave) y viceversa.
- `CertificateFactory`: clase para generar certificados de clave pública y listas de revocación.
- `KeyPair`: clase para crear objetos contenedores de pares de claves pública, privada.
- `KeyPairGenerator`: clase para crear generadores de pares de claves pública, privada.
- `KeyStore`: clase para la gestión de almacenes de claves.
- `SecureRandom`: clase para crear generadores seguros de números aleatorios.

Por su parte, la extensión JCE añade capacidad de encriptación y desencriptación de datos mediante clases e interfaces definidas en el paquete `javax.crypto`. Las más importantes son:

- `Cipher`: clase para crear objetos de cifrado por bloques.
- `CipherInputStream/CipherOutputStream`: clases para crear objetos de cifrado de datos que se leen de o se escriben en streams de entrada y de salida.
- `KeyGenerator`: clase para crear generadores de claves secretas para cifrado simétrico.
- `SecretKeyFactory`: clase para conversión de claves simétricas opacas de tipo `SecretKey` en especificaciones de clave (representaciones transparentes de los parámetros criptográficos de una clave) y viceversa.
- `KeyAgreement`: clase que proporciona la funcionalidad para el protocolo del mismo nombre para intercambio de mensajes de forma segura sin intercambiar una clave secreta.
- `MAC`: clase que proporciona la funcionalidad del algoritmo MAC, *Message Authentication Code*.

5.1 Herramientas de seguridad de la plataforma Java

Java proporciona dos herramientas de línea de comando relacionadas con la seguridad:

- `keytool`: permite administrar claves y certificados en un almacén de claves.
- `jarsigner`: permite firmar y verificar ficheros JAR usando claves y certificados de un almacén de claves.

Como alternativa a keytool, existe una aplicación open source denominada KeyStore Explorer, que facilita el trabajo con almacenes de claves de Java a través de su interfaz gráfica de usuario.

Un almacén de claves es una colección de entradas de claves y certificados con alias, que normalmente se guardan en un fichero protegido mediante una contraseña. La instalación de Java incluye un almacén de claves por defecto en el fichero `<java.home>/jre/lib/security/cacerts` con contraseña `chageit`.

La herramienta `keytool` se invoca desde la línea de comando con el formato siguiente:

```
> keytool -comando resto_de_parámetros …
```

Donde `-comando` representa cualquiera de los comandos que se relacionan a continuación agrupados en las categorías siguientes:

- Crear en o añadir datos a un almacén de claves:
  `-gencert`, `-genkeypair`, `-genseckey`, `-importcert`, `-importpass`
- Importar a un almacén de claves el contenido de otro almacén: `-importkeystore`
- Generar una petición de firma de certificado: `-certreq`
- Exportar datos: `-exportcert`
- Mostrar datos: `-list`, `-printcert`, `-printcertreq`, `-printcrl`
- Gestionar un almacén de claves: `-storepasswd`, `-keypasswd`, `-delete`, `-changealias`
- Obtener ayuda: `-help`

El resto de parámetros determinarán como se llevará a cabo la tarea encomendada. Por ejemplo, es habitual usar uno de los dos siguientes con la mayoría de los comandos:

- `-keystore`, para especificar la ruta de acceso a un fichero que contenga un almacén de claves.
- `-cacerts`, para indicar que se use el almacén de claves por defecto de Java.

Si se omite el parámetro `-keystore` y el parámetro `-cacerts`, se usa el almacén de claves por defecto del usuario, *<user.home>*`/.keystore`, que se creará automáticamente si no existe.

A continuación, se muestran varios ejemplos de uso de `keytool` y `jarsigner`.

Ejemplo 1: generar un certificado auto firmado:

```
>keytool -genkeypair -keyalg EC -validity 3650 -alias elsuper
```

`-keyalg`	Algoritmo utilizado para generar las claves (se pueden consultar los nombres válidos en https://docs.oracle.com/en/java/javase/17/docs/specs/security/standard-names.html#keypairgenerator-algorithms)
`-validity`	Opcional, especifica el periodo de validez del certificado expresado en número de días a partir de la fecha actual.
`-alias`	Nombre con el que se almacenará el certificado.

Se pide la contraseña del almacén de claves (si no existe el almacén, se crea y se pide una contraseña nueva y su confirmación) y los datos del titular del certificado, como se puede ver en la salida que se muestra a continuación:

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  VICENTE RUINEZ
What is the name of your organizational unit?
  [Unknown]:  SECCIÓN DE AGENTES
What is the name of your organization?
  [Unknown]:  T.I.A.
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:  es
Is CN=VICENTE RUINEZ, OU=SECCIÓN DE AGENTES, O=T.I.A., L=Unknown, ST=Unknown, C=es correct?
  [no]:  yes

Generating 256 bit EC (secp256r1) key pair and self-signed certificate (SHA256withECDSA) with a
validity of 3.650 days
        for: CN=VICENTE RUINEZ, OU=DIRECCION, O=T.I.A., L=Unknown, ST=Unknown, C=es
```

Obsérvese que en todas las preguntas en las que se piden datos del titular, se acepta `Unknown` como respuesta por defecto al pulsar directamente la tecla *INTRO* sin escribir una respuesta. Finalmente se validan los datos introducidos respondiendo `yes` cuando se muestran y se pregunta si son correctos.

También es posible suministrar la contraseña del almacén de claves y los datos del titular mediante parámetros en la línea de comando.

Ejemplo 2: firmar una aplicación Java almacenada en un fichero JAR:

Suponiendo que en el directorio de trabajo se encuentra el archivo "*holamundo.jar*", se firma con la utilidad `jarsigner`, usando la clave privada generada en el ejemplo anterior, de la forma siguiente:

```
>jarsigner -signedjar holamundofirmado.jar holamundo.jar elsuper
Enter Passphrase for keystore:
jar signed.

Warning:

The signer's certificate is self-signed.
```

En la línea de comando se indica: el nombre del archivo donde se guardará la aplicación firmada (parámetro `-signedjar`), el fichero JAR que se va a firmar y el alias de la clave privada con la que se va a firmar (se lee del almacén de claves por defecto al no especificar uno). Después de introducir la contraseña del almacén de claves se puede ver una advertencia como consecuencia de la utilización de una clave privada asociada a un certificado auto firmado.

En https://docs.oracle.com/en/java/javase/17/docs/specs/man/jarsigner.html se describen los parámetros de funcionamiento de esta herramienta.

Ejemplo 3: Autenticar la firma de un fichero JAR firmado:

Para este ejemplo se necesita la clave pública asociada a la clave privada con la que se realizó la firma. Por tanto, el primer paso consistirá en exportar el certificado autofirmado que se generó en el primer ejemplo:

```
>keytool -exportcert -alias elsuper -file elsuper.crt
Enter keystore password:
Certificate stored in file <elsuper.crt>
```

El certificado exportado se difundirá de alguna forma para que cualquiera pueda importarlo a su almacén de claves y verificar las firmas. Para importarlo podemos escribir en la línea de comando:

```
>keytool -importcert -file elsuper.crt
Enter keystore password:
Re-enter new password:
Owner: CN=VICENTE RUINEZ, OU= SECCIÓN DE AGENTES, O=T.I.A., L=Unknown, ST=Unknown, C=es
Issuer: CN=VICENTE RUINEZ, OU= SECCIÓN DE AGENTES, O=T.I.A., L=Unknown, ST=Unknown, C=es
Serial number: 4756ec5b377f8054
Valid from: Fri Feb 18 10:11:34 CET 2022 until: Mon Feb 16 10:11:34 CET 2032
Certificate fingerprints:
         SHA1: E5:FF:8C:F9:25:9A:00:1F:4B:E0:CC:7E:44:56:BD:69:93:41:72:79
         SHA256:
FB:22:20:4B:54:2B:61:08:68:F3:98:6D:DC:16:58:09:01:31:B0:FA:D6:DF:1F:7E:BF:7C:05:F9:85:CE:1D:D8
Signature algorithm name: SHA256withECDSA
Subject Public Key Algorithm: 256-bit EC (secp256r1) key
Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 2A 8C 42 82 4E 4B 4A 10   41 57 CF 33 53 9A 99 4F  *.B.NKJ.AW.3S..O
0010: 4B A3 4E 0D                                        K.N.
]
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
```

Antes de importar el certificado, `keytool` muestra sus datos y le pregunta al usuario si confía en él. Si la respuesta es afirmativa, el certificado se guarda en el almacén.

Una vez que se ha importado el certificado, para verificar la autenticidad de la firma del fichero JAR, escribimos lo siguiente desde la línea de comando:

```
>jarsigner -verify -verbose holamundofirmado.jar

s        140 Fri Feb 18 10:35:16 CET 2022 META-INF/MANIFEST.MF
         310 Fri Feb 18 10:35:16 CET 2022 META-INF/ELSUPER.SF
         953 Fri Feb 18 10:35:16 CET 2022 META-INF/ELSUPER.EC
sm       517 Fri Feb 18 10:32:32 CET 2022 Main.class

  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore

- Signed by "CN=VICENTE RUINEZ, OU= SECCIÓN DE AGENTES, O=T.I.A., L=Unknown, ST=Unknown, C=es"
    Digest algorithm: SHA-256
    Signature algorithm: SHA256withECDSA, 256-bit key

jar verified.

Warning:
This jar contains entries whose certificate chain is invalid. Reason: PKIX path building failed:
sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid certification path
to requested target
This jar contains entries whose signer certificate is self-signed.
This jar contains signatures that do not include a timestamp. Without a timestamp, users may not be
able to validate this jar after any of the signer certificates expire (as early as 2032-02-16).

Re-run with the -verbose and -certs options for more details.

The signer certificate will expire on 2032-02-16.
```

La opción `-verbose` muestra información suficiente para identificar al que ha realizado la firma, pero también muestra algunas advertencias derivadas del uso de un certificado auto firmado para llevar a cabo la verificación, indicando que no es de confianza.

5.2 Funciones Hash

En la actualidad, las más usadas pertenecen a un conjunto de funciones conocido como SHA-2, consideradas las más seguras en el ámbito criptográfico: SHA-224, SHA-256, SHA-384 y SHA-512.

En el año 2005 se libera SHA3 como sucesor de SHA-2, que utiliza un enfoque diferente para generar los resúmenes. Está disponible en la plataforma Java a partir de su versión 9.

Java suministra la clase `MessageDigest` para crear resúmenes de mensajes. Solo se puede instanciar mediante el método `getInstance` suministrándole el nombre del algoritmo de resumen.

> En https://docs.oracle.com/en/java/javase/17/docs/specs/security/standard-names.html#messagedigest-algorithms se pueden consultar los nombres válidos para especificar el algoritmo de resumen.

Métodos de la clase MessageDigest

`static MessageDigest` **`getInstance`**`(String algorithm)`
`static MessageDigest` **`getInstance`**`(String algorithm, Provider provider)`
`static MessageDigest` **`getInstance`**`(String algorithm, String provider)`
	Devuelve un objeto `MessageDigest` que implementa el algoritmo de resumen especificado.
	En la primera versión, los proveedores de seguridad se buscan según el orden establecido en el fichero `java.security`. En la segunda y tercera se usa el proveedor especificado.

`void` **`update`**`(byte input)`
`void` **`update`**`(byte[] input)`
`void` **`update`**`(byte[] input)`
	Actualizan el mensaje con nuevos bloques de datos.

`byte[] digest()`
`byte[] digest(byte[] input)`
`byte[] digest(byte[] buff, int offset, int len)`
	Completan el cálculo del resumen y lo retornan en un array de bytes. La dos últimas completan en mensaje con nuevos bloques de datos antes de completar el cálculo del resumen.

`void reset()`	Reinicializa el objeto `MessageDigest` para un nuevo uso.
`int getDigestLength()`	Retorna la longitud en bytes del resumen.
`String getAlgorithm()`	Retorna el nombre del algoritmo.
`Provider getProvider()`	Retorna el proveedor.
`static Boolean isEqual(byte[] digesta, byte[] digestb)`	Comprueba si dos mensajes resumen son iguales. Devuelve `true` si son iguales y `false` en caso contrario

El ejemplo siguiente genera el resumen de un texto codificado en UTF-8 usando el algoritmo SHA-256. Una vez generado, muestra el resumen en notación hexadecimal y codificado en Base64, junto con algunos datos más:

```
public class Ejemplo4 {

    public static void main(String[] args) throws NoSuchAlgorithmException {
        MessageDigest md;
        md = MessageDigest.getInstance("SHA-256");
        md.update("un mensaje de prueba".getBytes(StandardCharsets.UTF_8));
        byte [] resumen = md.digest();
        System.out.println("Algoritmo: " + md.getAlgorithm());
        System.out.println("Proveedor: " + md.getProvider().toString());
        System.out.println("Longitud del resumen: " + md.getDigestLength());
        System.out.println("Resumen en Hexadecimal: " + hexadecimal(resumen));
        System.out.println("Resumen en Base64: " +  Base64.getEncoder().encodeToString(resumen));
    }

    static String hexadecimal(byte[] resumen) {
        StringBuilder hex = new StringBuilder();
        for (int i = 0; i < resumen.length; i++)
            hex.append(String.format("%02X", resumen[i]));
        return hex.toString();
    }

}
```

La ejecución muestra la siguiente salida:

```
Algoritmo: SHA-256
Proveedor: SUN version 1.8
Longitud del resumen: 32
Resumen en Hexadecimal: 37077C16234F1A0A8A2E8DA102E58E23D0DBF213CBCE030A22149094719F8B68
Resumen en Base64: Nwd8FiNPGgqKLo2hAuWOI9Db8hPLzgMKIhSQlHGfi2g=
```

ACTIVIDAD 2

Crea un programa Java que reciba a través de un parámetro de línea de comando la ruta a un fichero de texto. El programa tiene que generar un resumen del contenido del fichero usando el algoritmo SHA-256 y almacenarlo en otro fichero con el mismo nombre, pero con extensión *.sha*.

Crea otro programa Java que reciba a través de parámetros de línea de comando la ruta al fichero de texto y la ruta al fichero que contiene su resumen y compruebe si el fichero de texto ha sido modificado, mostrando el mensaje correspondiente.

5.3 Clases e interfaces para representación de claves criptográficas

El API JCA, *Java Cryptography Architecture*, define dos interfaces que son la base de una variedad de clases e interfaces relacionadas con la manipulación de claves criptográficas:

- La interface `Key` para el manejo de representaciones opacas de los parámetros criptográficos de las claves. Declara los métodos siguientes:

`String` **`getAlgorithm`**`()`	retorna el nombre del algoritmo de cifrado para la clave.
`byte[]` **`getEncoded`**`()`	retorna la clave en su formato codificado o `null` si no soporta un formato codificado.
`String` **`getFormat`**`()`	retorna el nombre del formato de codificación para la clave o `null` si no soporta un formato codificado.

  Las subinterfaces `PrivateKey`, `PublicKey`, `DHPrivateKey`, `DHPublicKey`, `DSAPrivateKey`, `DSAPublicKey`, `ECPrivateKey`, `ECPublicKey`, `RSAMultiPrimePrivateCrtKey`, `RSAPrivateCrtKey`, `RSAPrivateKey` y `RSAPublicKey` representan claves asimétricas, y las subinterfaces `PBEKey` y `SecretKey` representa claves secretas para cifrado simétrico.

- La interface `KeySpec` se define para el manejo de especificaciones de clave, que son representaciones transparentes de los parámetros criptográficos de las claves. Esta interface no declara ninguna funcionalidad y se define con el único propósito de agrupar todas las posibles especificaciones de clave. Las clases `DHPrivateKeySpec`, `DHPublicKeySpec`, `DSAPrivateKeySpec`, `DSAPublicKeySpec`, `ECPrivateKeySpec`, `ECPublicKeySpec`, `RSAMultiPrimePrivateCrtKeySpec`, `RSAPrivateCrtKeySpec`, `RSAPrivateKeySpec`, `RSAPublicKeySpec`, `EncodedKeySpec`, `PKCS8EncodedKeySpec` y `X509EncodedKeySpec` la implementan para el manejo de especificaciones de claves asimétricas, y las clases `DESedeKeySpec`, `DESKeySpec`, `PBEKeySpec` y `SecretKeySpec` la implementan para el manejo de especificaciones de claves secretas para cifrado simétrico.

5.3.1 Generación de claves para cifrado asimétrico

Para generar un par de claves pública/privada en Java se usa la clase `KeyPairGenerator`. No dispone de constructores públicos, por lo que se ha de instanciar invocando al método `getInstance`.

Métodos de la clase KeyPairGenerator	
`static KeyPairGenerator` **`getInstance`**`(String algoritmo)` `static KeyPairGenerator` **`getInstance`**`(String algoritmo, String provider)`	Devuelve un objeto `KeyPairGenerator` que genera un par de claves púbica/privada para el algoritmo especificado. Puede lanzar la excepción `NoSuchAlgorithmException`. También acepta un proveedor, pero lanzará `NoSuchProviderException` si no se encuentra.
`void` **`initialice`**`(int keysize, SecureRandom random)`	Inicializa el generador de par de claves aceptando un tamaño de clave y un generador de números aleatorios.
`KeyPair` **`generateKeyPair`**`()` `KeyPair` **`genKeyPair`**`()`	Genera un par de claves y retorna una referencia al objeto `KeyPair` que las contiene.

La clase `KeyPair` es una clase soporte para almacenar las claves pública y privada. Dispone de dos métodos:

Métodos de la clase KeyPair	
`PrivateKey` **`getPrivate`**`()`	Retorna una referencia a la clave privada del par de claves.
`PublicKey` **`getPublic`**`()`	Devuelve una referencia a la clave pública del par de claves.

`PrivateKey` y `PublicKey` son las interfaces base de todas las subinterfaces que representan los diferentes formatos de clave privada y pública respectivamente.

El primer paso para generar el par de claves es instanciar el generador invocando al método `getInstance` de la clase `KeyPairGenerator`, pasándole el nombre del algoritmo que deberá usar.

El ejemplo siguiente muestra como instanciar un generador de claves DSA:

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
```

En https://docs.oracle.com/en/java/javase/17/docs/specs/security/standard-names.html#keypairgenerator-algorithms se pueden consultar los nombres de los algoritmos que se pueden usar para instanciar la clase `KeyPairGenerator`.

A continuación, se inicializa el generador de claves invocando a su método `initialize`, sobrecargado para especificar el tamaño de clave y el generador de números aleatorios de diferentes formas.

El tamaño de clave que se ha de especificar para el algoritmo DSA tiene que ser un múltiplo de 64 dentro del intervalo 512 a 1024. En caso contrario este método lanzará la excepción `InvalidParameterException`.

Como generador de números aleatorios se usa una instancia de la clase `SecureRandom`. El ejemplo siguiente crea un generador de números aleatorios que usa el algoritmo *SHA1PRNG*, para inicializar el generador de claves. Finalmente genera el par de claves y obtiene una referencia a cada una de ellas:

```
public class Ejemplo5 {
    public static void main(String[] args) throws NoSuchAlgorithmException {
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
        SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
        keyGen.initialize(1024, random);
        KeyPair par = keyGen.generateKeyPair();
        PrivateKey privada = par.getPrivate();
        PublicKey publica = par.getPublic();
    }
}
```

5.3.2 Almacenamiento de claves públicas y privadas en ficheros

Para almacenar una clave en un fichero, podemos usar un `OutputStream` y guardar los bytes que se obtienen como resultado de su codificación. El fragmento de código siguiente es un ejemplo de cómo hacerlo creando el fichero dentro de la carpeta personal del usuario:

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
keyGen.initialize(1024, random);
KeyPair par = keyGen.generateKeyPair();
File file = new File(System.getProperty("user.home") + "/miclave.priv");
try (FileOutputStream out = new FileOutputStream("miclave.priv")) {
    out.write(par.getPrivate().getEncoded());
}
File file = new File(System.getProperty("user.home") + "/miclave.pub");
try (FileOutputStream out = new FileOutputStream("miclave.pub")) {
    out.write(par.getPublic().getEncoded());
}
```

Para recuperar las claves de los ficheros necesitamos la clase `KeyFactory` que proporciona métodos para convertir claves de formato criptográfico (PKCS8 para claves privadas y X.509 para claves públicas) a especificaciones de claves y viceversa.

El fragmento de código siguiente realiza el proceso de recuperación de claves almacenadas en ficheros, creadas con el algoritmo DSA:

```
// recuperar una clave privada almacenada en un fichero
KeyFactory factory = KeyFactory.getInstance("DSA");
File filePrivada = new File(System.getProperty("user.home") + "/miclave.priv");
```

```java
try (FileInputStream in = new FileInputStream(filePrivada)) {
    byte[] bufferPriv = new byte[in.available()];
    in.read(bufferPriv);
    PKCS8EncodedKeySpec clavePrivadaSpec = new PKCS8EncodedKeySpec(bufferPriv);
    PrivateKey privada = factory.generatePrivate(clavePrivadaSpec);
    System.out.println("Clave Privada: " + privada.toString());
}

// recuperar una clave pública almacenada en un fichero
File filePublica = new File(System.getProperty("user.home") + "/miclave.pub");
try (FileInputStream in = new FileInputStream(filePublica)) {
    byte[] bufferPub = new byte[in.available()];
    in.read(bufferPub);
    X509EncodedKeySpec clavePrivadaSpec = new X509EncodedKeySpec(bufferPub);
    PublicKey publica = factory.generatePublic(clavePrivadaSpec);
    System.out.println("Clave Publica: " + publica.toString());
}
```

5.4 Generación y verificación de firmas digitales

Con un par de claves de cifrado asimétrico, se pueden generar y verificar firmas digitales usando instancias de la clase `Signature`. Esta clase no proporciona constructores públicos, por lo que se ha de instanciar invocando a su método de clase `getInstance`.

Métodos de la clase Signature	
`static Signature getInstance(String algorithm)` `static Signature getInstance(String algorithm, String provider)`	Devuelve un objeto `Signature` que implementa el algoritmo especificado. Puede lanzar la excepción `NoSuchAlgorithmException`. En el segundo método se especifica el proveedor. Si el nombre de proveedor no se encuentra se produce `NoSuchProviderException`.
`void initSign(PrivateKey privateKey, SecureRandom random)`	Inicializa el objeto para la firma. Se especifica la clave privada de la identidad cuya firma se va a generar y la fuente de aleatoriedad. Si la clave no es válida puede lanzar la excepción `InvalidKeyException`.
`void update(byte b)`	Actualiza los datos a firmar o verificar añadiendo el byte especificado.
`void update(byte[] data)`	Actualiza los datos a firmar o verificar añadiendo los almacenados en el array de bytes especificado.
`void update(ByteBuffer data)`	Actualiza los datos a firmar o verificar añadiendo los almacenados en el `ByteBuffer` especificado.
`byte[] sign()`	Devuelve en un array de bytes la firma de los datos
`void initVerify(PublicKey publicKey)`	Inicializa el objeto para la verificación de la firma aceptando la clave pública como parámetro. Si la clave no es válida puede lanzar la excepción `InvalidKeyException`.
`boolean verify(byte[] signature)`	Verifica la firma que se pasa como parámetro.

Al especificar el nombre del algoritmo de firma, también se debe incluir el nombre de un algoritmo de resumen que se usará.

> En https://docs.oracle.com/javase/9/docs/specs/security/standard-names.html#signature-algorithms se pueden consultar los nombres de los algoritmos que se pueden especificar para instanciar la clase `Signature`.

La firma de datos, así como su verificación usando objetos `Signature`, se llevan a cabo en tres fases:

- Inicialización para verificar invocando al método `initVerify` con clave pública, o para firmar invocando al método `initSign` con clave privada.
- Actualización de los datos que se van a firmar o verificar, invocando a métodos `update`.
- Firma invocando al método `sign`, o verificación invocando al método `verify`.

El siguiente ejemplo amplia el ejemplo anterior usando el algoritmo `"SHA256withDSA"` para generar una firma para una cadena de texto y posteriormente llevar a cabo su verificación:

```
public class Ejemplo5 {

    public static void main(String[] args) throws Exception {
        KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA");
        SecureRandom random = SecureRandom.getInstance("SHA1PRNG");
        keyGen.initialize(1024, random);
        KeyPair par = keyGen.generateKeyPair();
        PrivateKey privada = par.getPrivate();
        PublicKey publica = par.getPublic();
        Signature signature = Signature.getInstance("SHA256withDSA");

        // FIRMA

        // fase 1: inicialización
        signature.initSign(privada, random);

        // fase 2: actualización
        signature.update("texto de prueba".getBytes());

        // fase 3: firma
        byte [] firma = signature.sign();

        // VERIFICACIÓN

        // fase 1: inicialización
        signature.initVerify(publica);

        // fase 2: actualización
        signature.update("texto de prueba".getBytes());

        // fase 3: verificación
        System.out.println(signature.verify(firma) ? "VERIFICADO" : "NO VERIFICADO");
    }

}
```

ACTIVIDAD 3

Crea un programa Java con interfaz gráfica de usuario que pueda realizar las tareas que se describen a continuación:

- Generar la firma de cualquier fichero con los datos almacenados en él, usando una clave privada almacenada en otro fichero o en un almacén de claves.
- Verificar la firma usando la clave pública correspondiente almacenada en un fichero o en un almacén de claves.

5.5 Cifrado

Java incluye, dentro de la extensión criptográfica JCE, la clase `javax.crypto.Cipher` para cifrado y descifrado de datos.

Para crear un objeto `Cipher` se utiliza el método `getInstance`, pasando como argumento una cadena con el nombre del algoritmo y, opcionalmente, el nombre de un proveedor.

> En https://docs.oracle.com/en/java/javase/17/docs/specs/security/standard-names.html#cipher-algorithm-names se pueden consultar los nombres de los algoritmos, modos y rellenos que se pueden especificar para instanciar la clase `Signature`.

Métodos de la clase Cypher	
`static Cipher getInstance(String algoritmo)` `static Cipher getInstance(String algoritmo, String proveedor)`	Retorna un objeto `Cipher` que implementa el algoritmo especificado. En el segundo caso se especifica el proveedor. El parámetro `algoritmo` tiene la forma: algoritmo/modo/relleno. Por ejemplo: `"AES/CBC/NoPadding"`, `"AES/CBC/PKCS5Padding"`, etc. Puede lanzar las excepciónes `NoSuchAlgorithmException`, `NoSuchPaddingException` o `NoSuchProviderException`.
`int getBlockSize()`	Retorna el tamaño del bloque en bytes.
`int getOutputSize(int inputLen)`	Retorna el tamaño en bytes de un buffer de salida que es necesario si la siguiente entrada tiene el número de bytes indicado.
`void init(int modo, Key clave)`	Inicializa el objeto del algoritmo codificador, modo puede ser `Cypher.NCRYPT_MODE` (cifrar datos), `Cypher.DECRYPT_MODE` (descifrar datos), `Cypher.WRAP_MODE` o `Cypher.UNWRAP_MODE`. Los modos *wrap* y *unwrap* se utilizan para cifrar una clave con otra. Para inicializar el objeto hay que proporcionar una clave.
`byte[] update(byte[] entrada)` `byte[] update(byte[] entrada, int desplazamiento, int longitud)`	Transforma (cifra o descifra) el bloque de datos suministrado en el parámetro `entrada` y retorna el resultado.
`byte[] doFinal()` `byte[] doFinal(byte[] entrada)`	Termina la operación de cifrado o descifrado y limpia el buffer de ese objeto algoritmo. En el segundo método se indican los bytes de entrada que se procesarán. Retorna el resultado final.
`byte[] wrap(Key key)`	Envuelve una clave con la clave usada para inicializar el objeto Cypher.
`Key unwrap(byte[] wrappedKey, String wrappedKeyAlgorithm, int wrapped KeyType)`	Desenvuelve una clave previamente envuelta. `wrappedKey` es la clave a desenvolver, `wrappedKeyAlgorithm` es el algoritmo asociado con la clave envuelta, `wrappedKeyType` es el tipo de la clave envuelta; debe ser uno de los siguientes: `SECRET_KEY`, `PRIVATE_KEY` o `PUBLIC_KEY`.

Los modos indican la forma de trabajar del algoritmo. Por ejemplo, el modo ECB (*Electronic Cookbook Mode*) los mensajes se dividen en bloques y cada uno de ellos es cifrado por separado utilizando la misma clave K. La desventaja de este método es que a bloques de texto plano o claro idénticos les corresponden bloques idénticos de texto cifrado, de manera que se pueden reconocer estos patrones como guía para descubrir el texto en claro a partir del texto cifrado. De ahí que no sea recomendable para protocolos cifrados.

En el modo CBC (*Cipher Block Chaining*), a cada bloque de texto plano se le aplica la operación XOR con el bloque cifrado anterior antes de ser cifrado. De esta forma, cada bloque de texto cifrado depende de todo

el texto en claro procesado hasta este punto. Para hacer cada mensaje único se utiliza asimismo un vector de inicialización.

El relleno se utiliza cuando el mensaje a cifrar no es múltiplo de la longitud de cifrado del algoritmo, entonces es necesario indicar la forma de rellenar los últimos bloques.

Para suministrar una clave al método `init` de la clase `Cipher` se usa la clase `javax.crypto.KeyGenerator`. Algunos de sus métodos son:

`static KeyGenerator getInstance(String algoritmo)`	Retorna un objeto `KeyGenerator` que genera claves secretas para el algoritmo especificado, por ejemplo `"DES"`. Puede lanzar la excepción `NoSuchAIgorithmException`.

`void init(int keysize, SecureRandom random)`
`void init(int keysize)`
`void init(SecureRandom random)`
	Inicializa el generador de claves para un determinado tamaño de clave y un generador de números aleatorios.
	En el segundo método solo se pasa el tamaño de clave y en el tercero el generador de números aleatorios.

`SecretKey generateKey()`	Genera y retorna una clave secreta.

5.5.1 Cifrado simétrico

Los proveedores de seguridad registrados en la plataforma Java implementan los siguientes algoritmos de cifrado simétrico:

- **DES** (*Data Encryption Standard*): ofrece una mala seguridad, por lo que generalmente no se recomienda. Usa claves de 56 bits de longitud.
- **DESede**: algoritmo conocido como DES-EDE, 3DES o Triple DES que aplica el algoritmo DES tres veces con claves 168 bits de longitud. Tampoco se recomienda su uso.
- **AES**: (*Advanced Encryption Standard)* fue el encargado de sustituir a DES. Usa claves de longitud 128, 192 o 256 bits.
- **PBE**: (*Password Based Encryption*) puede ser usado con algoritmos de clave privada y funciones de resumen.

La Figura 8 muestra el esquema básico para cifrado simétrico.

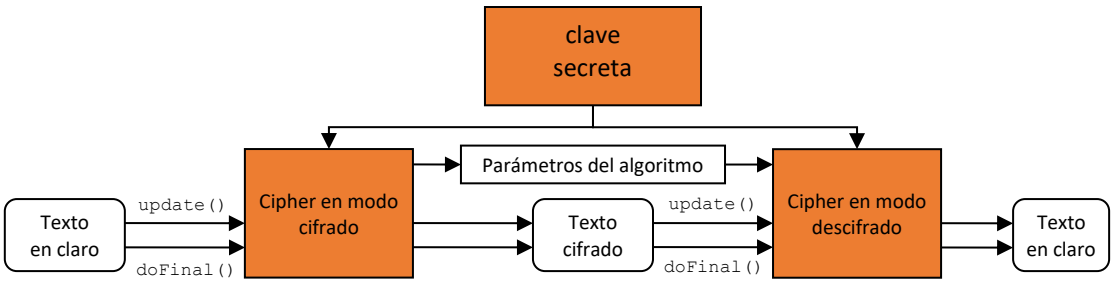


Figura 8. Proceso de cifrado y descifrado con clave secreta.

El ejemplo siguiente muestra cómo llevar cabo un proceso de cifrado simétrico en Java usando el algoritmo AES:

```
public class Ejemplo6 {

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

        /* crear de una clave secreta AES de 256 bits */
        KeyGenerator kg = KeyGenerator.getInstance("AES");
```

```
        kg.init(256);
        SecretKey clave = kg.generateKey();

        /* crear un objeto Cipher que usará el algoritmo de cifrado "AES/ECB/PKCS5Padding" */
        Cipher c = Cipher.getInstance("AES/ECB/PKCS5Padding");

        /* inicializar el objeto Cipher para cifrar con la clave secreta */
        c.init(Cipher.ENCRYPT_MODE, clave);

        /* obtener del texto cifrado a partir del texto original */
        byte [] textoCifrado = c.doFinal("TOP SECRET".getBytes());
        System.out.println("Texto cifrado: " + new String(textoCifrado));

        /* re-inicializar el objeto Cipher para descifrar con la misma clave secreta */
        c.init(Cipher.DECRYPT_MODE, clave);

        /* Obtener el texto original a partir del texto cifrado */
        System.out.println("Texto original: " + new String(c.doFinal(textoCifrado)));
    }

}
```

Salida:

```
Texto cifrado: ¤¡CÊ-{õz' K"¥øT_
Texto original: TOP SECRET
```

Algunos modos de algoritmo, como por ejemplo CBC, necesitan pasar al método `init` un vector de inicialización para el proceso de descifrado. El vector de inicialización se ha de obtener en el proceso de cifrado con la ayuda de la clase `IvParameterSpec`. El ejemplo siguiente es una variación del ejemplo anterior usando el algoritmo `"AES/CBC/PKCS5Padding"` para cifrar y descifrar:

```
public class Ejemplo7 {

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

        /* creación de una clave secreta AES de 256 bits */
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        kg.init(256);
        SecretKey clave = kg.generateKey();

        /* creación de un objeto Cipher que usará el algoritmo de cifrado "AES/ECB/PKCS5Padding" */
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");

        /* inicialización del objeto Cipher para cifrar con la clave secreta */
        cipher.init(Cipher.ENCRYPT_MODE, clave);

        /* obtención del vector de inicialización */
        IvParameterSpec iv = new IvParameterSpec(cipher.getIV());

        /* obtención del texto cifrado a partir del texto original */
        byte[] textoCifrado = cipher.doFinal("TOP SECRET".getBytes());
        System.out.println("Texto cifrado: " + new String(textoCifrado));

        /* re-inicialización del objeto Cipher para cifrar con la misma clave secreta y
           el mismo vector de inicialización */
        cipher.init(Cipher.DECRYPT_MODE, clave, iv);

        /* Obtención del texto original a partir del texto cifrado */
        System.out.println("Texto original: " + new String(cipher.doFinal(textoCifrado)));
    }

}
```

Salida:

```
Texto cifrado: Aþ=3u_#•''›?H$*^
Texto original: TOP SECRET
```

5.5.2 Almacenamiento de claves simétricas en ficheros

El siguiente ejemplo genera una clave secreta AES y la almacena en el fichero `<user.home>/miclave.key` serializando el objeto SecretKey:

```
public class Ejemplo8 {
```

```
    public static void main(String[] args) throws NoSuchAlgorithmException, IOException {
        KeyGenerator kg = KeyGenerator.getInstance("AES");
        kg.init(256);
        SecretKey clave = kg.generateKey();
        File file = new File(System.getProperty("user.home") + "/miclave.key");
        try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(file))) {
            out.writeObject(clave);
        }
    }
}
```

El ejemplo siguiente muestra el proceso contrario:

```
public class Ejemplo9 {
    public static void main(String[] args) throws IOException, ClassNotFoundException {
        File file = new File(System.getProperty("user.home") + "/miclave.key");
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(file))){
            SecretKey clave = (SecretKey) in.readObject();
            System.out.println(clave.getAlgorithm());
        }
    }
}
```

5.5.3 Cifrado asimétrico

El cifrado simétrico presenta el problema de la distribución de la clave, donde corre el riesgo de ser interceptada poniendo en peligro la seguridad del sistema.

En el cifrado asimétrico o de clave pública se resuelve el problema ya que la clave para cifrar se puede compartir sin problemas y la clave para descifrar solo la tiene que poseer el receptor del mensaje. Imaginemos, por ejemplo, que dos sujetos **A** y **B** desean mantener una conversación privada, usando un medio de comunicación público. Para asegurarse de que la conversación será privada, harán lo siguiente:

- **A** crea un par de claves, pública y privada, y difunde su clave pública para que cualquiera la pueda obtener, en este caso **B**.
- **B** crea su par de claves y hace lo mismo con su clave pública.
- **A** crea un mensaje, lo cifra con la clave pública de **B** y le envía el mensaje. **B** recibe el mensaje y lo descifra con su clave privada.
- **B** crea un mensaje, lo cifra lo con la clave pública de **A** y le envía el mensaje. **A** recibe el mensaje y lo descifra con su clave privada.

El fragmento de código siguiente muestra cómo crear el par de claves:

```
KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
keyGen.initialize (1024);
KeyPair par = keyGen.generateKeyPair();
PrivateKey clavepriv = par.getPrivate ();
PublicKey clavepub = par.getPublic();
```

Se crea un objeto `Cipher` con el algoritmo RSA, `"RSA/ECB/PKCS1Padding"`, y se inicializa en modo cifrado con la clave pública:

```
Cipher c = Cipher.getInstance("RSA/ECB/PKCS1Padding");
c.init(Cipher.ENCRYPT_MODE, clavepub);
```

Realizamos el cifrado de la información con el método `doFinal`:

```
byte textoCifrado[] = c.doFinal("Esto es un Texto Plano".getBytes());
```

Configuramos el objeto `Cipher` en modo descifrado con la clave privada para descifrar el texto usando el método `doFinal`:

```
c.init(Cipher.DECRYPT_MODE, clavepriv);
byte desencriptado[] = c.doFinal(textoCifrado);
```

El cifrado asimétrico es más lento que el cifrado simétrico, por lo que no resulta práctico utilizar este cifrado para cifrar grandes cantidades de información. Este problema se puede solucionar combinando cifrado de clave pública con cifrado simétrico. Los sujetos **A** y **B** de supuesto anterior llevaría a cabo el proceso de la forma siguiente:

- **A** crea una clave secreta de cifrado simétrico para cifrar los mensajes que le envía a **B**.
- **A** cifra la clave secreta con la clave pública del **B**.
- **A** envía a **B** la clave secreta cifrada junto con el mensaje cifrado.
- **B** utiliza su clave privada para descifrar la clave secreta que recibe de **A** y descifra el mensaje.

Nadie excepto el sujeto **B** puede descifrar la clave simétrica ya que solo él tiene la clave privada para descifrarla. El cifrado de clave pública sólo se aplica a una pequeña porción de datos.

En el ejemplo siguiente, se generan un par de claves pública y privada con el algoritmo RSA y una clave secreta con el algoritmo AES. Con la clave secreta se cifra un texto, esta clave se usará para cifrar el texto. La clave secreta es cifrada mediante la clave pública utilizando el método `wrap` de la clase `Cipher` . Para descifrar el texto primero necesitamos descifrar la clave secreta con la clave privada y a continuación descifrar el texto con esa clave; usaremos el método `unwrap`:

```
public class Ejemplo10 {

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

            /* Creación del par de claves RSA */
            KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
            keyGen.initialize(2048);
            KeyPair par = keyGen.generateKeyPair();
            PrivateKey clavePrivada = par.getPrivate();
            PublicKey clavePublica = par.getPublic();

            /* Creación de la clave secreta AES para cifrado simétrico */
            KeyGenerator kg = KeyGenerator.getInstance("AES");
            kg.init(256);
            SecretKey claveSecreta = kg.generateKey();

            /* Cifrado de la clave secreta AES con la clave pública RSA */
            Cipher c = Cipher.getInstance("RSA/ECB/PKCS1Padding");
            c.init(Cipher.WRAP_MODE, clavePublica);
            byte claveCifrada[] = c.wrap(claveSecreta);

            /* Cifrado del texto con la clave secreta */
            c = Cipher.getInstance("AES/ECB/PKCS5Padding");
            c.init(Cipher.ENCRYPT_MODE, claveSecreta);
            byte textoCifrado[] = c.doFinal("Mensaje".getBytes());
            System.out.println("Cifrado: " + new String(textoCifrado));

            /* Descifrado de la clave secreta AES con la clave RSA privada */
            Cipher c2 = Cipher.getInstance("RSA/ECB/PKCS1Padding");
            c2.init(Cipher.UNWRAP_MODE, clavePrivada);
            Key claveDescifrada = c2.unwrap(claveCifrada, "AES", Cipher.SECRET_KEY);

            /* Descifrado del texto con la clave secreta */
            c2 = Cipher.getInstance("AES/ECB/PKCS5Padding");
            c2.init(Cipher.DECRYPT_MODE, claveDescifrada);
            byte texto[] = c2.doFinal(textoCifrado);
            System.out.println("Descifrado: " + new String(texto));
    }

}
```

Salida:

```
Cifrado: ©7§Uz3B›UZ]öv¬ÖÂ
Descifrado: mensaje
```

El concepto de clave de sesión es un término medio entre el cifrado simétrico y asimétrico que permite combinar las dos técnicas. Consiste en generar una clave de sesión K y cifrarla usando la clave pública del receptor. El receptor descifra la clave de sesión usando su clave privada. El emisor y el receptor comparten una clave que solo ellos conocen y pueden cifrar sus comunicaciones usando la misma clave de sesión.

5.5.4 Cifrar y descifrar flujos de datos

La biblioteca JCE proporciona las clases `CipherOutputStream` para cifrar datos hacia un stream de salida y `CipherInputStream` para descifra datos provenientes de un stream de entrada. Estas clases manipulan de forma transparente las llamadas a `update` y `doFinal`.

El ejemplo siguiente utiliza una clave secreta almacenada en el fichero `<user.home>/miclave.key` para cifrar el documento PDF almacenado en el fichero `<user.home>/unidad5.pdf`:

```
public class Ejemplo11 {
  public static void main(String[] args) throws Exception {
          /* Recuperamos la clave secreta del fichero */
          File file = new File(System.getProperty("user.home") + "/miclave.key");
          try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(file))) {
                  SecretKey clave = (SecretKey) in.readObject();
                  System.out.println(clave.getAlgorithm());

                  /* Creamos un objeto Cipher para cifrar con el algoritmo AES */
                  Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
                  cipher.init(Cipher.ENCRYPT_MODE, clave);

                  /*
                   * Se crean los streams de entrada salida para el fichero pdf y
                   * el fichero cifrado y se lleva a cabo el proceso de cifrado
                   */
                  File original = new File(System.getProperty("user.home") + "/unidad5.pdf");
                  File cifrado = new File(System.getProperty("user.home") + "/cifrado.pdf");
                  try (FileInputStream in = new FileInputStream(original);
                       CipherOutputStream out = new CipherOutputStream(
                       new FileOutputStream(cifrado), cipher)) {
                           /* Creamos un buffer con el tamaño de bloque del objeto Cipher*/
                           byte[] bloque = new byte[cipher.getBlockSize()];
                           /* Leemos bloque a bloque y lo guardamos cifrado */
                           int n;
                           while ((n = in.read(bloque)) != -1)
                                   out.write(bloque, 0, n);
                  }
          }
  }

}
```

El ejemplo siguiente utiliza una clave secreta almacenada en el fichero `<user.home>/miclave.key` para descifrar el documento PDF almacenado en el fichero `<user.home>/cifrado.pdf`:

```
public class Ejemplo12 {
    public static void main(String[] args) throws Exception {
        /* Recuperamos la clave secreta del fichero */
        File file = new File(System.getProperty("user.home") + "/miclave.key");
        try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(file))) {
            SecretKey clave = (SecretKey) in.readObject();
            System.out.println(clave.getAlgorithm());

            /* Creamos un objeto Cipher para descifrar con el algoritmo AES */
            Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
            cipher.init(Cipher.DECRYPT_MODE, clave);

            /*
             * Se crean los streams de entrada salida para el fichero cifrado y
             * el fichero pdf y se lleva a cabo el proceso de descifrado
             */
            File cifrado = new File(System.getProperty("user.home") + "/cifrado.pdf");
            File descifrado = new File(System.getProperty("user.home") + "/descifrado.pdf");
```

```
            try (FileOutputStream out = new FileOutputStream(descifrado);
                    CipherInputStream inc = new CipherInputStream(
                        new FileInputStream(cifrado), cipher)) {
                /* Obtenemos el tamaño de bloque del objeto Cipher */
                int tamBloque = cipher.getBlockSize();
                /* Creamos un buffer para leer bloque a bloque */
                byte[] bloque = new byte[tamBloque];
                /* Leemos bloques descifrados y los guardamos en el fichero pdf */
                int n;
                while ((n = inc.read(bloque)) != -1)
                    out.write(bloque, 0, n);
            }
        }
    }
}
```

ACTIVIDAD 4

El modo de cifrado ECB, usado por el algoritmo de cifrado en los dos ejemplos anteriores, presenta una vulnerabilidad, que lo hace inadecuado para cifrar más de un bloque de datos. Modifica ambos ejemplos para que el algoritmo de cifrado use el modo CBC.

6 Comunicaciones seguras con java. JSSE

Los datos que viajan a través de la red pueden ser interceptados por personas que no son los destinatarios de los mismos. Cuando incluyen información privada, como contraseñas y números de tarjetas de crédito, se deben tomar medidas para que sean incomprensibles a las partes no autorizadas. También es importante asegurarse de que los datos no se modifiquen durante el transporte ya sea intencionadamente o no. Los protocolos SSL (*Secure Sockers Layer*) y TLS (*Transport Layer Security*) se han diseñado para ayudar a proteger la privacidad y la integridad de los datos mientras se transfieren a través de una red.

JSSE (*Java Secure Socket Extension*) es un conjunto de paquetes que permiten el desarrollo de aplicaciones seguras en Internet. Proporciona un marco y una implementación para la versión Java de los protocolos SSL y TLS e incluye funcionalidad de encriptación de datos, autenticación de servidores, integridad de mensajes y autenticación de clientes. Con JSSE, los desarrolladores pueden ofrecer intercambio seguro de datos entre un cliente y un servidor que ejecuta un protocolo de aplicación, tales como HTTP, Telnet o FTP, a través de TCP/IP. Las clases de JSSE se encuentran en los paquetes `javax.net` y `javax.net.ssl`.

6.1 SSLSocket y SSLServerSocket

Vamos a tratar el uso de SSL basado en las clases `SSLSocket` y `SSLServerSocket` que representan sockets seguros y son derivadas de las ya familiares `Socket` y `ServerSocket` respectivamente.

JSSE tiene dos clases `SSLServerSocketFactory` y `SSLSocketFactory` que proporcionan el método estático `getDefault` que servirá para la creación de sockets seguros. Entonces para obtener un `SSLServerSocket` escribimos:

```
SSLServerSocket servidorSSL = (SSLServerSocket)
    SSLServerSocketFactory.getDefault().createServerSocket(puerto);
```

El método `createServerSocket(int puerto)` devuelve un socket de servidor enlazado al puerto especificado. Para crear un `SSLSocket` escribimos:

```
SSLSocket Cliente = (SSLSocket) SSLSocketFactory.getDefault().createSocket(host, puerto);
```

El método `createSocket (String host, int puerto)` crea un socket y lo conecta con el `host` en el `puerto` especificado.

A continuación, se muestra el programa servidor que crea una conexión sobre un socket servidor seguro y que atenderá la conexión de un cliente que se identificará con un certificado válido. El servidor espera la conexión del cliente, una vez aceptada la conexión recibe un mensaje del cliente y a continuación le envía un saludo:

```
public class ServidorSSL {
    public static void main(String[] arg) throws IOException {
        SSLServerSocket serverSocket = null;
        try {
            serverSocket = (SSLServerSocket)
                    SSLServerSocketFactory.getDefault().createServerSocket(6000);
            while (true)
                atenderPeticion((SSLSocket) serverSocket.accept());
        } finally {
            if (serverSocket != null)
                    serverSocket.close();
        }
    }

    static void atenderPeticion(SSLSocket socket) throws IOException {
        try {
            DataInputStream in = new DataInputStream(socket.getInputStream());
            DataOutputStream out = new DataOutputStream(socket.getOutputStream());
            System.out.println("Recibido: " + in.readUTF());
            out.writeUTF("Saludos del servidor");
        } finally {
            socket.close();
        }
    }

}
```

El programa cliente envía un mensaje al servidor y visualiza el que el servidor le devuelve:

```
public class ClienteSSL {
    public static void main(String[] args) throws IOException {
        SSLSocket socket = null;
        try {
            socket = (SSLSocket) SSLSocketFactory.getDefault().createSocket("localhost", 6000);
            DataOutputStream out = new DataOutputStream(socket.getOutputStream());
            out.writeUTF("Saludos del cliente");
            DataInputStream in = new DataInputStream(socket.getInputStream());
            System.out.println("Recibido: " + in.readUTF());
        } finally {
            if (socket != null)
                socket.close();
        }
    }
}
```

El servidor necesita disponer de un certificado que mostrar a los clientes que se conecten a él. Usaremos la herramienta `keytool` para crear uno auto firmado en el almacén de claves por defecto:

```
> keytool -genkey -alias miServidor -keyalg RSA -storepass practicas
```

El cliente necesitará el certificado del servidor en su almacén de claves. Por tanto, lo exportamos a un fichero desde el almacén de claves del servidor y lo importamos al almacén de claves del cliente.

Para ejecutar tanto cliente como servidor, es necesario suministrarles los certificados correspondientes. Tenemos dos opciones:

- Hacerlo desde la línea de comando. Por ejemplo, en el servidor:

  ```
  > java -Djavax.net.ssl.keyStore=<almacén> -Dj avax.net.ssl.keyStorePassword=practicas
  ServidorSSL
  ```

  sustituyendo *<almacén>* por la ruta de acceso al almacén de claves que corresponda. Si se omite este parámetro, se usará el almacén de claves por defecto.

- Desde el código Java estableciendo el valor de las propiedades JSSE que corresponda con el método `System.setProperty`. En la siguiente tabla se muestran las propiedades JSSE:

`javax.net.ssl.keyStore`	Ruta de acceso al almacén de claves de Java. En Windows, se usará el carácter `/`, como separador.
`javax.net.ssl.keyStorePassword`	Contraseña para acceder al almacén de claves especificado por `javax.net.ssl.keyStore`.
`javax.net.ssl.trustStore`	Ruta de acceso al almacén de claves que contiene la colección de certificados de confianza. En Windows, se usará el carácter `/`, como separador.

Por ejemplo:

```
System.setProperty("javax.net.ssl.key Store", "ruta_al_almacén");
System.setProperty("javax.net.ssl.keyStorePassword", "practicas");
SSLSocket
```

Opcionalmente podemos registrar un proveedor SSL en los programas. Por ejemplo:

```
java.security.Security.addProvider(new com.sun.net.ssl.internal.ssl.Provider());
```

El método `getSession` del objeto `SSLSocket` devuelve un objeto `SSLSession` con la sesión SSL utilizada por la conexión, a partir de ella podemos obtener información como el identificador de la sesión:

```
SSLSession session = socket.getSession();
System.out.println("Host: " + session.getPeerHost());
System.out.println("Cifrado: " + session.getCipherSuite());
System.out.println("Protocolo: " + session.getProtocol());
System.out.println("IDentificador:" + new BigInteger(session.getId()));
System.out.println("Creación de la sesión: " + session.getCreationTime());
X509Certificate certificate = (X509Certificate) session.getPeerCertificates()[0];
System.out.println("Propietario: " + certificate.getSubjectDN());
System.out.println("Algoritmo: " + certificate.getSigAlgName());
System.out.println("Tipo: " + certificate.getType());
System.out.println("Emisor: " + certificate.getIssuerDN());
System.out.println("Número Serie: " + certificate.getSerialNumber());
```

En el programa servidor para obtener la información del certificado tendríamos que utilizar el método `getLocalCertificates` que devuelve el certificado enviado al cliente durante la negociación, en lugar de `getPeerCertificates`.

ACTIVIDAD 4

Modifica el servidor de contactos desarrollado en la unidad 3 y el cliente para utilicen sockets seguros.

7 Control de acceso con java. JASS

JAAS (*Java Authentication and Authorization Service*) es una interfaz que permite a las aplicaciones Java acceder a servicios de control de autenticación y acceso. Se puede utilizar para dos propósitos:

- Para la autenticación de usuarios: para determinar de forma fiable y segura quién está ejecutando nuestro código Java.
- Para la autorización de los usuarios: para asegurarse de que quien lo ejecuta tiene los permisos necesarios para realizar las acciones.

En el proceso de autenticación y autorización mediante JAAS están involucradas las siguientes clases e interfaces:

- `LoginContext`, contexto de inicio de sesión: inicia y gestiona el proceso de autenticación mediante la creación de un `Subject`. La autenticación se hace llamando al método `login`.
- `LoginModule`, módulo de conexión: es la interfaz que debe implementarse para definir los mecanismos de autenticación en la aplicación. Se deben implementar los siguientes métodos: `initialize`, `login`, `commit`, `abort` y `logout`. Se encarga de validar los datos en un proceso de autenticación.
- `Subject`: clase que representa a un ente autenticable dentro de la aplicación (entidad, usuario, sistema).
- `Principal`: clase que representa los atributos que posee cada `Subject` recuperado una vez que se efectúa el ingreso a la aplicación. Un `Subject` puede contener varios principales.
- `CallBackHandler`: interfaz que se debe implementar cuando se necesita recibir del usuario la información para la autenticación, se encarga de la interacción con el usuario para obtener los datos de autenticación. Al implementarla se debe desarrollar el método `handle`.

Los paquetes en los que están disponibles las clases e interfaces principales de JAAS son:

- `javax.security.auth`: contiene las clases de base e interfaces para los mecanismos de autenticación y autorización.
- `javax.security.auth.callback`: contiene las clases e interfaces para definir las credenciales de autenticación de la aplicación.
- `javax.secur¡ty.auth.login`: contiene las clases para entrar y salir de un dominio de aplicación.
- `javax.security.auth.spi`: contiene interfaces para un proveedor de JAAS para implementar módulos JAAS.

En los siguientes apartados veremos ejemplos básicos para llevar a cabo la autenticación y autorización con JASS.

7.1 Autenticación

El proceso básico de autenticación con JAAS consta de los siguientes pasos:

1. Creación de una instancia de `LoginContext`, uno o más `LoginModule` son cargados basándose en el archivo de configuración de JAAS.
2. La instanciación de cada `LoginModule` es opcionalmente provista con un `CalIbackHandler` que gestionará el proceso de comunicación con el usuario para obtener los datos con los que este tratará de autenticarse.
3. Invocación del método `login` del `LoginContext` el cual invocará el método `login` del `LoginModule`.
4. Los datos del usuario se obtienen por medio del `CallbackHandler`.
5. El `LoginModule` comprueba los datos introducidos por el usuario y los valida. Si la validación tiene éxito el usuario queda autenticado.

La Figura 9 muestra el esquema básico de las clases y ficheros que se van a utilizar en el siguiente ejemplo para probar la autenticación básica con JAAS. Los ficheros son los siguientes:

- Fichero `EjemploJaasAutenticación.java`, es la aplicación que será autenticada y autorizada mediante JAAS.
- Fichero `EjemploLoginModule.java`, implementa el módulo `LoginModule` que autentifica a los usuarios mediante su nombre y contraseña.
- Fichero `MyCallbackHandler.java` que implementa `CallbackHandler`. Transfiere la información requerida al módulo `LoginModule`.
- Fichero de autenticación de JAAS, `jaas.config`, donde se configura el módulo de login. En el ejemplo se vincula el nombre `EjemploLogin` al módulo `LoginModule` de nombre `EjemploLoginModule`, la palabra `required` al lado indica que el módulo de conexión asociado debe ser capaz de autenticar al sujeto en todo el proceso de autenticación para poder tener éxito. El contenido es:

```
EjemploLogin {
    EjemploLoginModule required;
};
```

- Fichero de autorización JAAS, `policy.config`, en el que destacamos los permisos de lectura sobre las propiedades usuario y clave que se introducirán desde la línea de comando y el permiso para crear un contexto de inicio de sesión, `createLoginContext`, al nombre `EjemploLogin` vinculado con la clase `EjemploLoginModule`. El contenido del fichero es el siguiente:

```
grant {
    permission java.util.PropertyPermission "usuario", "read";
    permission java.útil.PropertyPermission "clave", "read";
    permission javax.security.auth.AuthPermission "createLoginContext.EjemploLogin";
};
```

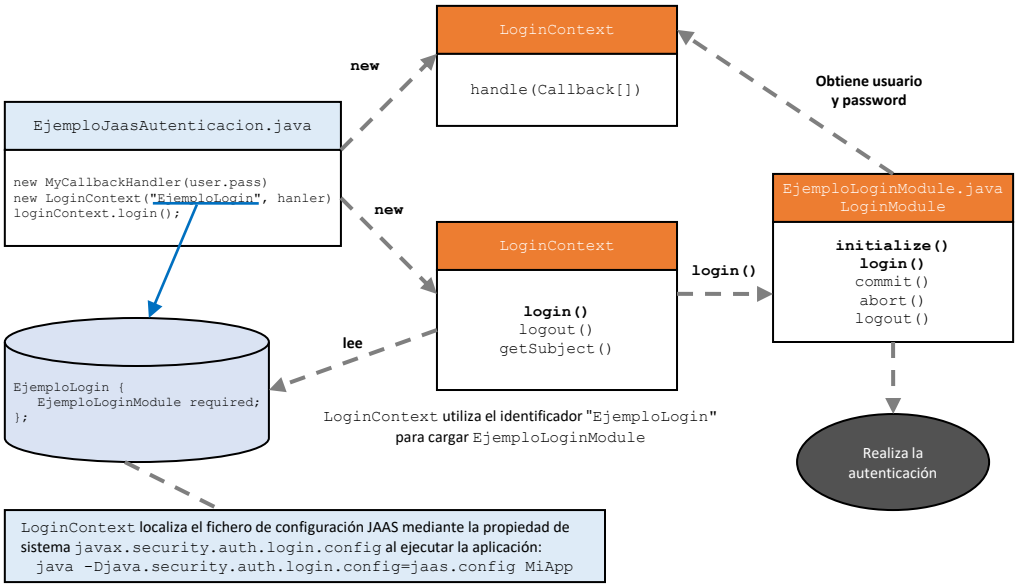


Figura 9. Clases para el ejemplo de autenticación.

El contenido de los ficheros Java se muestra a continuación.

7.1.1 Código de EjemploJaasAutenticacion.java:

El proceso de autenticación se inicia con la creación de una instancia de `LoginContext`. La clase tiene varios constructores, en el ejemplo se utiliza `LoginContext(String, CallbackHandler)`, el primer parámetro es el nombre que hace referencia a una entrada con el mismo nombre en el fichero de configuración JAAS (`jaas.config`) y el segundo es un manejador de devolución de llamada que se utiliza

para pasar información de inicio de sesión al módulo `LoginModule` (en el ejemplo se pasa el nombre de usuario y la clave); después se realiza una llamada al método `login`:

El método `login` establece una conexión o lanza `LoginException` si no se puede establecer. Invoca el método `login` del fichero `EjemploLoginModule` que implementa `LoginModule`.

En el ejemplo el nombre de usuario y la clave a autenticar se pasan desde la línea de comandos como propiedades de sistema, para ejecutarlo escribimos lo siguiente:

```
java -Dusuario=alumno -Dclave=practicas -Djava.security.manager -Djava.security.policy=policy.config
-Djava.security.auth.login.config=jaas.config EjemploJaasAutenticación
```

El código completo es el siguiente:

```
public class EjemploAutenticacionJAAS {
    public static void main(String[] args) {
        /* datos proporcionados desde la linea de comando a la JVM */
        String usuario = System.getProperty("usuario");
        String clave = System.getProperty("clave");
        /* Se crea el CallbackHandler pasándole en el constructor el nombre
         * de usuario y la clave para que el LoginModule acceda a ellos */
        CallbackHandler handler = new MyCallbackHandler(usuario, clave);
        try {
            LoginContext loginContext = new LoginContext("EjemploLogin", handler);
            /* invocamos al método login para realizar la autenticación */
            loginContext.login();
            System.out.println("Usuario autenticado    ");
        } catch (LoginException e) {
            /* la autenticación no tiene éxito */
            System.err.println("No se puede autenticar el usuario");
        }
    }
}
```

7.1.2 Código de MyCallbackHandler.java

En algunos casos, el `LoginModule` debe comunicarse con el usuario para obtener la información de autenticación. Para este propósito se utilizan los `CallbackHandler`. En el ejemplo se pasa un `CallbackHandler` como argumento para la creación de instancias `LoginContext`, al `CallbackHandler` se le pasan el nombre y la clave del usuario. El `LoginContext` reenvía el `CallbackHandler` directamente a los `LoginModules` subyacentes.

`CallbackHandler` tiene el método `handle` que será invocado por el `LoginModule` al que le pasará un array de objetos `Callbacks`, que contiene los campos de datos que se necesitan. Cada `Callback` representa uno de los datos comunicados por el usuario en el proceso de autenticación (nombre, password, etc), es necesario recorrer este array para recuperar los datos. Se dispone de varios `Callbacks`:

`NameCallback`	El `LoginModule` quiere que el usuario introduzca un nombre.
`PasswordCallback`	El `LoginModule` quiere que el usuario introduzca un password.
`ChoiceCallback`	El `LoginModule` quiere que el usuario elija un valor de un conjunto de valores.
`ConfirmationCallback`	El `LoginModule` quiere que el usuario responda `sí/no/cancelar` a una pregunta en particular.
`TextOutputCallback`	El `LoginModule` envía algún tipo de información al usuario.

En el ejemplo desde el constructor se da valor a las variables que contienen el nombre de usuario y la clave. En el método `handle`, que será invocado por el `LoginModule`, se le asigna al callback `NameCallback` el nombre de usuario y a `PasswordCallback` la contraseña del usuario:

```
public class MyCallbackHandler implements CallbackHandler {

    private String usuario;
    private String clave;
```

```
    public MyCallbackHandler(String usuario, String clave) {
        this.usuario = usuario;
        this.clave = clave;
    }

    @Override
    public void handle(Callback[] callbacks) throws IOException,
            UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            Callback callback = callbacks[i];
            if (callback instanceof NameCallback) {
                NameCallback nameCB = (NameCallback) callback;
                /* se asigna al NameCallback el nombre de usuario */
                nameCB.setName(usuario);
            } else if (callback instanceof PasswordCallback) {
                PasswordCallback passwordCB = (PasswordCallback) callback;
                /* se asigna al PasswordCallback la clave */
                passwordCB.setPassword(clave.toCharArray());
            }
        }
    }
}
```

7.1.3 Código de EjemploLoginModulejava

En esta la clase se implementa el `LoginModule` que autenticará a los usuarios en su método `login`. Debe implementar los siguientes métodos:

- `initialize`: el propósito de este método es inicializar el `LoginModule` con la información relevante. El `Subject` pasado a este método se usa para almacenar los principales (`Principal`) y credenciales si la conexión tiene éxito. Recibe un `CallbackHandler` que puede ser usado para introducir información de la autenticación.
- `login`: el propósito de este método es autenticar al `Subject`.
- `commit`: se llama a este método si tiene éxito la autenticación total del `LoginContext`.
- `abort`: informa al `LoginModule` de que algún proveedor o módulo ha fallado al autenticar al `Subject`. Se llama a este método si la autenticación global de `LoginContext` ha fallado.
- `logout`: desconecta al `Subject` borrando los principales y credenciales del `Subject`. Finalizará la sesión del usuario.

En el ejemplo se implementan los métodos `initialize` y `login`. En el método `login` es donde se realizará la autenticación, se invocará al método `handle` del `CallbackHandler` para obtener el nombre del usuario y la clave y se comprobarán sus valores. `login` devuelve `true` si la autenticación tiene éxito, en caso contrario devuelve `false`. El código es el siguiente:

```
public class EjemploLoginModule implements LoginModule {

    private Subject subject;
    private CallbackHandler callbackHandler;

    public boolean commit() throws LoginException {
        return true;
    }

    public boolean logout() throws LoginException {
        return true;
    }

    public boolean abort() throws LoginException {
        return true;
    }

    public void initialize(Subject subject, CallbackHandler handler, Map State, Map options) {
        this.subject = subject;
        this.callbackHandler = handler;
    }

    public boolean login() throws LoginException {
        boolean autenticado = false;
```

```
        if (callbackHandler == null) {
            throw new LoginException("Se necesita CallbackHandler");
        }

        Callback[] callbacks = new Callback[2];
        callbacks[0] = new NameCallback("Nombre de usuario: ");
        callbacks[1] = new PasswordCallback("Clave: ", false);
        try {
            /* se invoca al método handle del CallbackHandler
             * para solicitar el usuario y la contraseña */
            callbackHandler.handle(callbacks);
            String usuario = ((NameCallback) callbacks[0]).getName();
            char[] passw = ((PasswordCallback) callbacks[1]).getPassword();
            String clave = new String(passw);
            /* La autenticación se realiza aqui */
            autenticado = ("alumno".equalsIgnoreCase(usuario) &
                    "practicas".equals(clave));
        } catch (Exception e) {
            e.printStackTrace();
        }
        return autenticado;
    }
}
```

Para ejecutar el ejemplo todos los ficheros tienen que estar en la misma carpeta. A continuación, se compilan los ficheros y se ejecuta `EjemploJaasAutenticacion` con el gestor de seguridad habilitado indicando el fichero de políticas y el de configuración JAAS:

```
> javac EjemploLoginModule.java
> javac MyCallbackHandler.java
> javac EjemploJaasAutenticacion.java
> java -Dusuario=alumno -Dclave=practicas -Djava.security.manager
-Djava.security.policy=policy.config -Djava.security.auth.login.config=jaas.config
EjemploJaasAutenticacion
Usuario autenticado
```

Ahora lo ejecutamos escribiendo una clave incorrecta, aparece mensaje de error:

```
> java -Dusuario=alumno -Dclave=practicas -Djava.security.manager
-Djava.security.policy=policy.config -Djava.security.auth.login.config=jaas.config
EjemploJaasAutenticacion
No se puede autenticar el usuario
```

ACTIVIDAD 6

Modificar el ejemplo anterior para que nos pida el nombre de usuario y la clave por la entrada estándar.

7.2 Autorización

La autorización de JAAS extiende la arquitectura de seguridad de Java centrada en el código y se basa en el uso de políticas de seguridad para especificar cuáles son los permisos de control de acceso que se concederán para ejecutar un código. Los permisos se otorgarán no solo en función de qué código se está ejecutando, sino también en quién lo está ejecutando.

Cuando una aplicación utiliza la autenticación de JAAS para autenticar al usuario (u otra entidad, como un servicio), se crea un `Subject` como resultado que representará al usuario autenticado. Un `Subject` se compone de un conjunto de principales (clase `Principal`), donde cada uno representa un atributo para ese usuario. Por ejemplo, un `Subject` puede tener dos principales, uno es el nombre y el otro el DNI.

- Para que la autorización JAAS tenga lugar, se requiere lo siguiente:
- El usuario debe autenticarse, ya hemos visto como se hace.
- En el fichero de políticas se deben configurar entradas para los principales.

- Se debe asociar al `Subject` el contexto de control de acceso actual usando los métodos `doAs` o `doAsPriviIeged` de esta clase.

Para llevar a cabo la autorización partimos de las clases creadas anteriormente. Pero añadiremos algunas cosas, por ejemplo, vamos a permitir acceder a la aplicación a dos usuarios “maria” y “juan”, modificaremos la clase `EjemploLoginModule` para que autentique a los usuarios. En esta clase también modificamos el método `commit` para añadir el principal al sujeto autenticado.

La clase `MyCallbackHandler` no sufre cambio. En la clase que contiene el método `main`, `EjemploAutenticacionJAAS`, se debe obtener el `Subject` (que incluirá un principal que representa al usuario) y hacer que realice las acciones llamando al método `doAsPrivileged`, a este método se le pasa el `Subject` y la acción a realizar (clase que implementa `PrivilegedAction`):

```
Subject subject = loginContext.getSubject();
PrivilegedAction action = new EjemploAccion();
Subject.doAsPrivileged(subject, action, null);
```

A los usuarios les daremos dos privilegios distintos: a “maria" le permitiremos la lectura de un fichero y a “juan” le permitiremos que pueda escribir en él, esto se indicará en el fichero de políticas `policy.config`.

Necesitaremos crear dos nuevas clases y definir los privilegios de acceso en el fichero de políticas. Las clases a crear son:

- `EjemploAccion.java`. Esta clase implementa `PrivilegedAction`, que contiene el método `run` que se ejecutará una vez que el usuario ha sido autenticado, teniendo en cuenta las autorizaciones de los principales definidas en el fichero de políticas.
- `EjemploPrincipal.java`. Esta clase implementa la interface `Principal`, se usa en la clase `EjemploLoginModule` una vez que el usuario ha sido autenticado en el método `commit`.

El fichero de políticas `policy.config` lo definimos de la siguiente manera:

```
grant {
    permission javax.security.auth.AuthPermission "createLoginContext.EjemploLogin";
    permission javax.security.auth.AuthPermission "modifyPrincipals";
    permission javax.security.auth.AuthPermission "doAsPrivileged";
};
grant Principal EjemploPrincipal "maria" {
    permission java.io.FilePermission "fichero.txt", "read";
};
grant Principal EjemploPrincipal "juan" {
    permission java.io.FilePermission "fichero.txt", "write";
};
```

En el primer bloque `grant` se definen 3 permisos ya que la aplicación `EjemploAutenticacionJAAS` hace lo siguiente: crea un objeto `LoginContext` (contexto de inicio de sesión), modifica los principales del sujeto autenticado y llama al método `doAsPrivileged` de la clase `Subject`.

A continuación se definen dos bloques grant el primero concede el privilegio de lectura del `fichero.txt` a un `EjemploPrincipal` denominado “maria” y el segundo concede el privilegio de escritura (incluye creación) en el `fichero.txt` a un `EjemploPrincipal` denominado “juan”.

La Figura 10 muestra el esquema básico de las clases y ficheros que se van a utilizar en el siguiente ejemplo para probar la autorización JAAS.

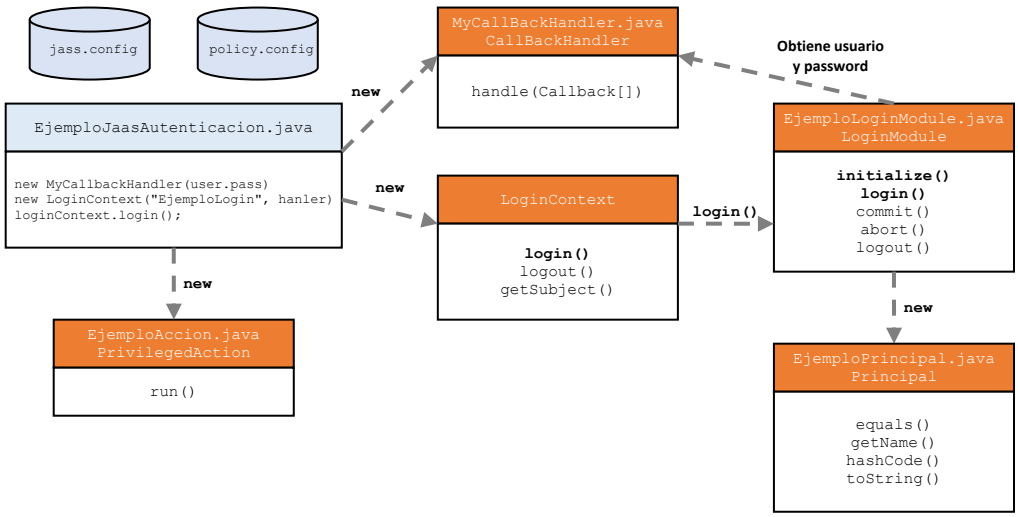


Figura 10. Clases para el ejemplo de autorización.

7.2.1 Código de EjemploAutenticacionJAAS

Sufre algunos cambios con respecto al ejemplo de autenticación. Una vez realizada la autenticación se obtiene el `Subject`, se crea el objeto `PrivilegedAction` y se llama al método `doAsPrivileged` para ejecutar la acción:

```
public class EjemploAutenticacionJAAS {
    public static void main(String[] args) {
        /* datos proporcionados desde la linea de comando a la JVM */
        String usuario = System.getProperty("usuario");
        String clave = System.getProperty("clave");
        LoginContext loginContext = null;

        /* Se crea el CallbackHandler pasándole en el constructor el nombre
         * de usuario y la clave para que el LoginModule acceda a ellos */
        CallbackHandler handler = new MyCallbackHandler(usuario, clave);
        try {
            loginContext = new LoginContext("EjemploLogin", handler);
            /* invocamos al método login para realizar la autenticación */
            loginContext.login();
            System.out.println("Usuario autenticado    ");
        } catch (LoginException e) {
            /* la autenticación no tiene éxito */
            System.err.println("Usuario no autenticado: " + e.getLocalizedMessage());
            System.exit(-1);
        }
        Subject subject = loginContext.getSubject();
        PrivilegedAction action = new EjemploAccion();
        try {
            Subject.doAsPrivileged(subject, action, null);
        } catch (SecurityException e) {
            System.err.println("Acceso denegado: " + e.getLocalizedMessage());
        }
        try {
            loginContext.logout();
        } catch (LoginException e) {
            System.out.println("Logout: " + e.getLocalizedMessage());
        }
    }
}
```

7.2.2 Código de EjemploLoginModule.java:

Los métodos `login`, `commit` y `logout` con respecto al ejemplo de autenticación han cambiado. En el método `login` se ha añadido la comprobación del usuario “juan” con clave “practicas”. En el método

`commit` se crea el principal asociado al usuario autenticado (se le da el nombre del usuario) y se añade al `Subject`. En el método `logout` se elimina el principal que se añadió en `commit`. El código es:

```
import java.util.Map;
import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

public class EjemploLoginModule implements LoginModule {
    private Subject subject;
    private CallbackHandler callbackHandler;
    private String usuario;
    private String clave;
    private EjemploPrincipal usuarioPrincipal;

    public boolean abort() throws LoginException {
        return true;
    }

    public boolean commit() throws LoginException {
        usuarioPrincipal = new EjemploPrincipal(usuario);
        /* se añade el principal (identidad autenticada) al sujeto */
        if (!subject.getPrincipals().contains(usuarioPrincipal))
            subject.getPublicCredentials().add(usuarioPrincipal);
        return true;
    }

    public boolean logout() throws LoginException {
        return true;
    }

    public void initialize(Subject subject, CallbackHandler handler,
            Map State, Map options) {
        this.subject = subject;
        this.callbackHandler = handler;
    }

    public boolean login() throws LoginException {
        boolean autenticado = false;
        if (callbackHandler == null) {
            throw new LoginException("Se necesita CallbackHandler");
        }

        Callback[] callbacks = new Callback[2];
        callbacks[0] = new NameCallback("Nombre de usuario: ");
        callbacks[1] = new PasswordCallback("Clave: ", false);
        try {
            /* se invoca al método handle del CallbackHandler
             * para solicitar el usuario y la contraseña */
            callbackHandler.handle(callbacks);
            String usuario = ((NameCallback) callbacks[0]).getName();
            char[] passw = ((PasswordCallback) callbacks[1]).getPassword();
            String clave = new String(passw);
            /* La autenticación se realiza aqui */
            autenticado = ("maria".equalsIgnoreCase(usuario) & "practicas".equals(clave)) ||
                    ("juan".equalsIgnoreCase(usuario) & "practicas".equals(clave));
        } catch (Exception e) {
            e.printStackTrace();
        }
        return autenticado;
    }
}
```

7.2.3 Código de EjemploPrincipal.java

Esta clase implementa la interfaz `Principal`. El método más importante es `getName`, que devuelve el nombre del principal. En base a los principales se determinan los privilegios. Recordemos las líneas del fichero de políticas donde se daban los privilegios a los usuarios autenticados “maria” y “juan”. A la derecha del principal (`EjemploPrincipal`) ponemos el nombre que le hemos dado (que en este caso es el nombre del usuario autenticado):

```
grant Principal EjemploPrincipal "maria" {
    permission java.io.FilePermission "fichero.txt", "read";
};
grant Principal EjemploPrincipal "juan" {
    permission java.io.FilePermission "fichero.txt", "write";
};
```

A “maria” se le permite leer el `fichero.txt` y “juan” puede escribir en él. El código es:

```
import java.security.Principal;

public class EjemploPrincipal implements Principal, java.io.Serializable {
    private String name;

    public EjemploPrincipal(String name) {
        if (name == null)
            throw new NullPointerException("Entrada nula");
        this.name = name;
    }

    public String getName() {
        return name;
    }

    /* Compara el objeto especificado con el Principal retornando true si son iguales */
    public boolean equals(Object o) {
        if (o == null || !(o instanceof EjemploPrincipal))
            return false;
        return this == o || getName().equals(((EjemploPrincipal) o).getName());
    }

    public int hashCode() {
        return name.hashCode();
    }

    public String toString() {
        return (name);
    }
}
```

7.2.4 Código de EjemploAcción.java:

Este es el código que se ejecutará después de que el usuario haya sido correctamente autenticado. En primer lugar, se comprueba si existe el fichero de nombre fichero.txt. Si existe se muestra su contenido y si no existe se creará y se insertará una cadena de texto:

```
public class EjemploAccion implements PrivilegedAction {
    public Object run() {
        File f = new File("fichero.txt");
        if (f.exists()) {
            FileReader fr;
            try {
                fr = new FileReader(f);
                int i;
                System.out.println("Contenido del fichero: ");
                while ((i = fr.read()) != -1)
                    System.out.print((char) i);
                fr.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else {
            System.out.println("El fichero no existe. Creando fichero ... ");
            try {
                FileWriter fw = new FileWriter(f);
                String cadena = "Esto es una linea de texto";
                fw.append(cadena);
                fw.close();
                System.out.println("Fichero creado con datos...");
            } catch (IOException e) {
                System.err.println("Error: " + e.getLocalizedMessage());
            }
        }
        return null;
    }
}
```

Para probar la aplicación guardamos los ficheros de configuración, de políticas y todas las clases Java en una misma carpeta. La compilación y ejecución muestra la siguiente salida:

```
> javac EjemploAccion.java
> javac EjemploPrincipal.java
> javac EjemploLoginModule.java
> javac MyCallbackHandler.java
> javac EjemploJaasAutenticacion.java
```

Ejecutamos la aplicación como usuario “maria”, la salida muestra un error porque intenta crear el fichero.txt y no tiene permisos:

```
> java -Djava.security.manager -Dj ava.security.policy=policy.config
-Djava.security.auth.login.config=jaas.config EjemploJaasAutenticacion
Nombre de usuario: maria
Clave: practicas
Usuario autenticado
El fichero no existe. Creando fichero ...
Acceso denegado: access denied ("java.io.FilePermission" fichero.txt" "write")
```

Ejecutamos la aplicación como usuario “juan”, ahora el fichero sí se crea ya que el usuario tiene privilegios, y se muestra su contenido:

```
> java -Djava.security.manager -Djava.security.policy=policy.config
-Djava.security.auth.login.config=jaas.config EjemploJaasAutenticacion
Nombre de usuario: juan
Clave: practicas
Usuario autenticado
El fichero no existe. Creando fichero ...
Fichero creado con datos...
```

Por último ejecutamos el programa como usuario “maria”, en este caso como el fichero existe se muestra su contenido:

```
> java -Djava.security.manager -Djava.security.policy=policy.config
-Djava.security.auth.login.config=jaas.config EjemploJaasAutenticacion
Nombre de usuario: maria
Clave: practicas
Usuario autenticado
Contenido del fichero:
Esto es una linea de texto
```