Universidade Vigo

Escola **S**uperior de **E**nxeñaría **I**nformática

Memoria de la práctica de

Uso del API de cifrado de Java

para la asignatura de **S**eguridade en **S**istemas **I**nformáticos

Luis Garbayo Fernández

Uxío Raluy Martínez

Tabla de Contenidos

DESCRIPCIÓN BREVE

La práctica consiste en diseñar y programar, con la *Java Cryptography Architecture* (JCA), un sistema simplificado de intercambio de facturas electrónicas entre tres actores: la **Empresa** que genera la factura, la **Autoridad de Sellado** que certifica su fecha, y **Hacienda** que la recibe y valida.

El alumno debe implementar el empaquetado de la factura (confidencialidad y autenticidad), su sellado de tiempo (prueba de existencia en una fecha), y el desempaquetado/verificación en destino, cumpliendo requisitos de confidencialidad, integridad, autenticidad, no repudio y detección de manipulaciones, priorizando además un coste computacional bajo (uso mínimo de criptografía asimétrica). Para ello se proporcionan y/o emplean los módulos: GenerarClaves (dado), EmpaquetarFactura, SellarFactura y DesempaquetarFactura, trabajando sobre un formato de Paquete que encapsula la factura y los metadatos/firmas necesarios.

En conjunto, el objetivo es aplicar las primitivas criptográficas estándar de Java para asegurar el ciclo completo de creación, sellado, envío y verificación de una Factura Empaquetada de forma segura.

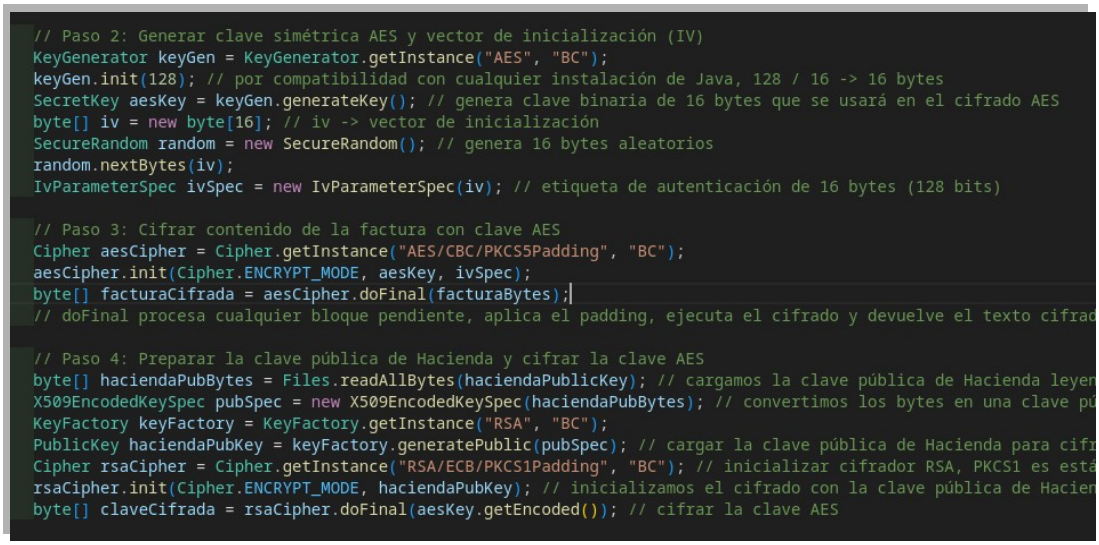
DESCRIPCIÓN Y JUSTIFICACIÓN DE LAS ESTRATEGIAS

A continuación se relaciona cada requisito R1–R8 con la estrategia criptográfica aplicada y con los pasos concretos del empaquetado, sellado y desempaquetado implementados en el código entregado.

R1. Confidencialidad de la Factura Empaquetada (Empresa → Hacienda)

Estrategia: criptografía híbrida.

- Cifrado simétrico: la factura JSON se cifra con AES/CBC/PKCS5Padding usando una clave simétrica AES (16 bytes) y un vector de inicialización aleatorio (IV de16 bytes).
- Encapsulado de clave: la clave AES se cifra con RSA/PKCS1 usando la clave pública de Hacienda (solo Hacienda, con su privada, podrá recuperarla).



```
// Paso 2: Generar clave simétrica AES y vector de inicialización (IV)
KeyGenerator keyGen = KeyGenerator.getInstance("AES", "BC");
keyGen.init(128); // por compatibilidad con cualquier instalación de Java, 128 / 16 -> 16 bytes
SecretKey aesKey = keyGen.generateKey(); // genera clave binaria de 16 bytes que se usará en el cifrado AES
byte[] iv = new byte[16]; // iv -> vector de inicialización
SecureRandom random = new SecureRandom(); // genera 16 bytes aleatorios
random.nextBytes(iv);
IvParameterSpec ivSpec = new IvParameterSpec(iv); // etiqueta de autenticación de 16 bytes (128 bits)

// Paso 3: Cifrar contenido de la factura con clave AES
Cipher aesCipher = Cipher.getInstance("AES/CBC/PKCS5Padding", "BC");
aesCipher.init(Cipher.ENCRYPT_MODE, aesKey, ivSpec);
byte[] facturaCifrada = aesCipher.doFinal(facturaBytes);
// doFinal procesa cualquier bloque pendiente, aplica el padding, ejecuta el cifrado y devuelve el texto cifrad

// Paso 4: Preparar la clave pública de Hacienda y cifrar la clave AES
byte[] haciendaPubBytes = Files.readAllBytes(haciendaPublicKey); // cargamos la clave pública de Hacienda leyen
X509EncodedKeySpec pubSpec = new X509EncodedKeySpec(haciendaPubBytes); // convertimos los bytes en una clave pú
KeyFactory keyFactory = KeyFactory.getInstance("RSA", "BC");
PublicKey haciendaPubKey = keyFactory.generatePublic(pubSpec); // cargar la clave pública de Hacienda para cifr
Cipher rsaCipher = Cipher.getInstance("RSA/ECB/PKCS1Padding", "BC"); // inicializar cifrador RSA, PKCS1 es está
rsaCipher.init(Cipher.ENCRYPT_MODE, haciendaPubKey); // inicializamos el cifrado con la clave pública de Hacien
byte[] claveCifrada = rsaCipher.doFinal(aesKey.getEncoded()); // cifrar la clave AES
```

Primero se genera la clave AES con el vector de inicialización para pasar al posterior cifrado con AES/CBC. Se cifra la clave AES con RSA (*public key* de Hacienda), y se guarda la factura cifrada, la clave cifrada y ese vector de inicialización.

Para el posterior desempaquetado solo Hacienda, con su *private key* (RSA), podrá descifrar la clave AES, la factura con AES/CBC y recuperar el IV.

R2. Autenticidad/Integridad de lo aportado por la Empresa (verificable por Autoridad y Hacienda)

Estrategia: firma digital de Empresa (SHA256withRSA). Se firma el conjunto crítico FACTURA_CIFRADA || CLAVE_CIFRADA con la clave privada de Empresa. Esto une criptográficamente la factura cifrada con la clave de sesión cifrada. Primero se firma con la clave privada de Empresa en el módulo de empaquetado para posteriormente realizar respectivas verificaciones en los módulos de sellado y desempaquetado.

```
// Paso 5: Firmar paquete con la clave privada de la Empresa
byte[] empresaPrivBytes = Files.readAllBytes(empresaPrivateKey); // cargamos la clave privada de la Empresa leyen
PKCS8EncodedKeySpec privSpec = new PKCS8EncodedKeySpec(empresaPrivBytes); // convertimos los bytes en una clave p
PrivateKey empresaPrivKey = keyFactory.generatePrivate(privSpec); // cargar la clave privada de la Empresa para f
Signature signature = Signature.getInstance("SHA256withRSA", "BC");
signature.initSign(empresaPrivKey); // inicializamos el firmador con la clave privada de la Empresa
signature.update(facturaCifrada); // actualizamos el firmador con el contenido crítico
signature.update(claveCifrada); // actualizamos el firmador con la clave cifrada
byte[] firmaEmpresa = signature.sign(); // firmamos el contenido crítico
```

```
// Paso 3: Verificar la firma de la Empresa
byte[] mensajeFirmadoEmpresa = concatenarBytes(facturaCifrada, claveCifrada); // concatenar los datos que la Empre
Signature verificadorEmpresa = Signature.getInstance("SHA256withRSA", "BC"); // inicializar verificador con SHA-51
verificadorEmpresa.initVerify(clavePublicaEmpresa); // inicializar la verificación con la clave pública de la Empr
verificadorEmpresa.update(mensajeFirmadoEmpresa); // actualizar el verificador con el mensaje original
if (!verificadorEmpresa.verify(firmaEmpresa)) {
    System.err.println("Firma de la Empresa: ¡FALLIDA! El contenido de la Empresa fue alterado o la clave pública
    System.exit(1);
}
```

R3. Garantizar que lo que llega a Autoridad de sellado no está modificado

Estrategia: verificación previa de la firma de Empresa en el módulo de sellado antes de sellar. La Autoridad no añade sello si la firma de Empresa no verifica sobre FACTURA_CIFRADA || CLAVE_CIFRADA.

```
// Verificar firma de la Empresa
byte[] mensajeFirmadoEmpresa = concatenarBytes(facturaCifrada, claveCifrada); // concatenar factura cifrada y clav
Signature verificadorEmpresa = Signature.getInstance("SHA256withRSA", "BC");
verificadorEmpresa.initVerify(clavePublicaEmpresa);
verificadorEmpresa.update(mensajeFirmadoEmpresa);
if (!verificadorEmpresa.verify(firmaEmpresa)) {
    System.err.println("La verificación de la firma de la Empresa falló.");
    System.exit(1);
}
```

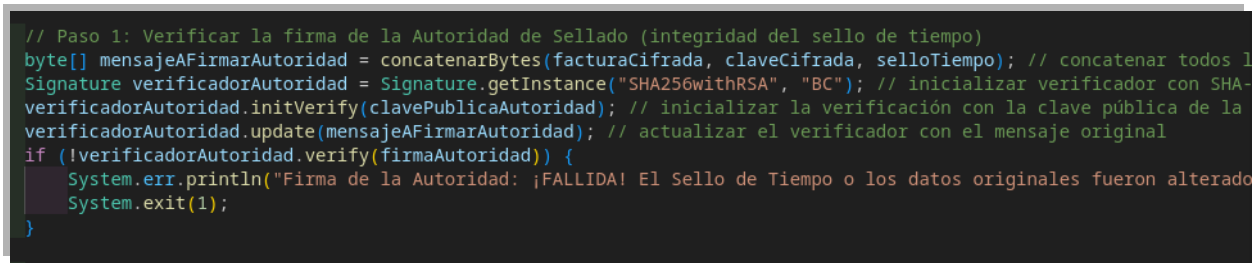
R4. No repudio por parte de la Empresa ante lo enviado

Estrategia: firma digital de Empresa y custodia de su clave privada. La firma SHA256withRSA sobre los datos críticos impide que la Empresa niegue autoría de la Factura Empaquetada.

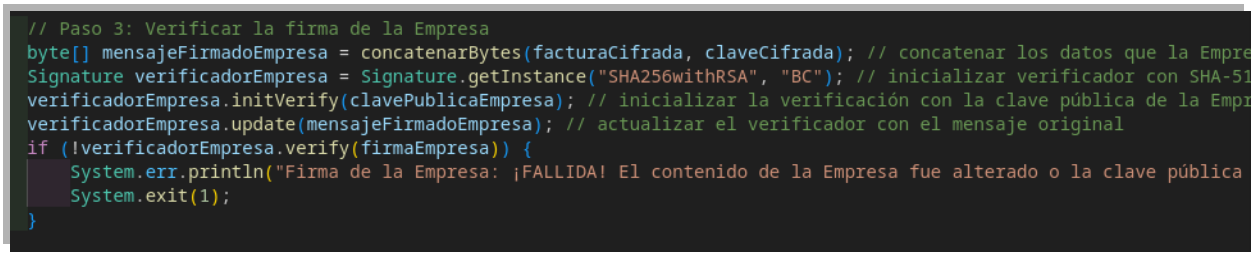
R5. Garantizar a Hacienda que no hay cambios en la Factura Empaquetada

Estrategia: doble verificación en destino. Al desempaquetar, Hacienda primero verifica la Autoridad (integridad del sello y datos sellados) y, después, verifica la Empresa (integridad del contenido aportado por esta misma). Si cualquiera falla, aborta. La doble verificación se produce gracias a:

- Firma de Autoridad.
- Firma de Empresa.



```
// Paso 1: Verificar la firma de la Autoridad de Sellado (integridad del sello de tiempo)
byte[] mensajeAFirmarAutoridad = concatenarBytes(facturaCifrada, claveCifrada, selloTiempo); // concatenar todos l
Signature verificadorAutoridad = Signature.getInstance("SHA256withRSA", "BC"); // inicializar verificador con SHA-
verificadorAutoridad.initVerify(clavePublicaAutoridad); // inicializar la verificación con la clave pública de la
verificadorAutoridad.update(mensajeAFirmarAutoridad); // actualizar el verificador con el mensaje original
if (!verificadorAutoridad.verify(firmaAutoridad)) {
    System.err.println("Firma de la Autoridad: ¡FALLIDA! El Sello de Tiempo o los datos originales fueron alterado
    System.exit(1);
}
```



```
// Paso 3: Verificar la firma de la Empresa
byte[] mensajeFirmadoEmpresa = concatenarBytes(facturaCifrada, claveCifrada); // concatenar los datos que la Empre
Signature verificadorEmpresa = Signature.getInstance("SHA256withRSA", "BC"); // inicializar verificador con SHA-51
verificadorEmpresa.initVerify(clavePublicaEmpresa); // inicializar la verificación con la clave pública de la Empr
verificadorEmpresa.update(mensajeFirmadoEmpresa); // actualizar el verificador con el mensaje original
if (!verificadorEmpresa.verify(firmaEmpresa)) {
    System.err.println("Firma de la Empresa: ¡FALLIDA! El contenido de la Empresa fue alterado o la clave pública
    System.exit(1);
}
```

R6. Garantizar la fecha (evidencia temporal del envío)

Estrategia: sello de tiempo firmado por Autoridad. La Autoridad genera un timestamp ISO-8601 y lo añade al paquete; a continuación firma FACTURA_CIFRADA || CLAVE_CIFRADA || SELLO_TIEMPO.

```java
// Paso 3: Generar Timestamp
String timestampStr = LocalDateTime.now().format(DateTimeFormatter.ISO_LOCAL_DATE_TIME);
byte[] selloTiempo = timestampStr.getBytes("UTF-8");
paquete.anadirBloque(nombre:"SELLO_TIEMPO", selloTiempo);

// Paso 4: Firmar con la Autoridad
byte[] mensajeAFirmarAutoridad = concatenarBytes(facturaCifrada, claveCifrada, selloTiempo);
Signature firmadorAutoridad = Signature.getInstance("SHA256withRSA", "BC");
firmadorAutoridad.initSign(clavePrivadaAutoridad);
firmadorAutoridad.update(mensajeAFirmarAutoridad);
byte[] firmaAutoridad = firmadorAutoridad.sign();
paquete.anadirBloque(nombre:"FIRMA_AUTORIDAD", firmaAutoridad);
```

En el módulo de desempaquetado, Hacienda muestra el *timestamp* y verifica la firma de Autoridad; si el sello fue alterado, la verificación falla.

```java
// Paso 2: Obtener el sello de tiempo
String timestampStr = new String(selloTiempo, "UTF-8"); // convertir el timestamp de bytes a String
System.out.println("Sello de Tiempo (Timestamp): " + timestampStr);

// Paso 3: Verificar la firma de la Empresa
byte[] mensajeFirmadoEmpresa = concatenarBytes(facturaCifrada, claveCifrada); // concatenar los datos que la Empre
Signature verificadorEmpresa = Signature.getInstance("SHA256withRSA", "BC"); // inicializar verificador con SHA-51
verificadorEmpresa.initVerify(clavePublicaEmpresa); // inicializar la verificación con la clave pública de la Empr
verificadorEmpresa.update(mensajeFirmadoEmpresa); // actualizar el verificador con el mensaje original
if (!verificadorEmpresa.verify(firmaEmpresa)) {
    System.err.println("Firma de la Empresa: ¡FALLIDA! El contenido de la Empresa fue alterado o la clave pública
    System.exit(1);
}
```

R7. Autenticidad / no repudio de la Autoridad de sellado

Estrategia: firma digital de Autoridad (SHA256withRSA). La Autoridad firma (con su *private key*) el triple “factura cifrada + clave cifrada + sello”. En el desempaquetado, terceros como Hacienda verifican con su *public key*.

R8. Coste computacional reducido (minimizar criptografía asimétrica)

Estrategia: patrón híbrido. Uso de criptografía simétrica para datos y asimétrica solo para encapsular claves y firmas. El cifrado masivo se hace con AES. La asimetría RSA se limita a encapsular la clave de sesión y a firmar/verificar (operaciones puntuales). En el desempaquetado se utiliza RSA para abrir la clave y verificar firmas y AES para descifrar los datos.

Pasos que se siguen:

EMPAQUETAR FACTURA	Primero se lee la factura en formato JSON para posteriormente generar la clave simétrica (AES) con la que se cifrará el contenido de la factura. Se cifra la clave AES con la "hacienda.publica". Se firma el paquete con la clave privada de la empresa y, finalmente, se añade todo en el paquete, por ejemplo, "factura.paquete".
SELLAR FACTURA	La Autoridad de Sellado recibe el paquete, verifica la firma de la Empresa con su clave pública, genera el sello de tiempo para su posterior firmado con su clave privada. Añade el sello de tiempo y su firma al paquete.
DESEMPAQUETAR FACTURA	Comienza con la verificación de la firma de la Autoridad de Sellado y la obtención del *timestamp*. Verifica la firma de la Empresa, descifra la clave AES con su clave privada para posteriormente descifrar la factura con la clave AES. Finalmente, se muestra el resultado.

DESCRIPCIÓN DE FORMATO Y ESTRUCTURA DE “PAQUETE”

El “**paquete**” es un fichero de texto con formato tipo *ASCII armor* que encapsula datos binarios en bloques claramente delimitados. Su estructura comienza con una cabecera global -----INICIO PAQUETE----- y termina con -----FIN PAQUETE-----. Entre ambas marcas se suceden bloques, cada uno rodeado por sus propias cabeceras -----INICIO BLOQUE NOMBRE----- y -----FIN BLOQUE NOMBRE-----. El contenido binario real de cada bloque (cifrados, firmas, IV, etc.) no se escribe tal cual, sino codificado en Base64 y envuelto a 65 caracteres por línea, de forma que el archivo resultante es legible y portable por canales “solo texto”. Esta lógica está implementada en la clase proporcionada Paquete.java.

Con nuestro código, tras empaquetar, el paquete incluye: FACTURA_CIFRADA, CLAVE_CIFRADA, VECTOR_INICIALIZACION y FIRMA_EMPRESA. La factura se cifra con AES (*Advanced Encryption Standard*) usando el modo CBC (*Cipher Block Chaining*) y *padding* PKCS5Padding. Se prefiere elegir JSON, ya que es un formato ampliamente utilizado que puede ser procesado fácilmente por diferentes aplicaciones y lenguajes de programación. El bloque de clave cifrada contendrá la clave AES utilizada para el cifrado. La firma de la Empresa se genera con el algoritmo SHA256withRSA, que combina una función hash (SHA-256) con firma RSA usando la *private key* de la Empresa.

Tras sellarse agregan dos bloques más: SELLO_TIEMPO y FIRMA_AUTORIDAD, que proporcionan confirmación de fecha y hora del procesamiento de la factura. Antes de sellar, se verifica la FIRMA_EMPRESA; si no es válida, no se modifica el paquete. Todo esto lo realiza SellarFactura.java. En el desempaquetado se exigen y leen esos bloques; primero se verifica la firma de Autoridad (cubre el sello y los datos sellados), luego la firma de Empresa (cubre lo aportado por la Empresa). Si ambas se verifican exitósamente, se descifra la clave AES con la privada de Hacienda y, con el IV del bloque VECTOR_INICIALIZACION, se descifra FACTURA_CIFRADA para recuperar el JSON original.

En el directorio proporcionado se puede observar el ejemplo de esta estructura en “factura.paquete”.

DESCRIPCIÓN BREVE DE IMPLEMENTACIÓN

La práctica se compone de cuatro ejecutables de negocio —**GenerarClaves, EmpaquetarFactura, SellarFactura, DesempaquetarFactura**— y una utilidad de contenedor —Paquete— que define el formato del fichero “paquete” y su E/S. Se emplea la librería BouncyCastle como *provider* criptográfico y las primitivas estándar de JCA (API de cifrado de Java).

GenerarClaves (proporcionado por docente) crea un par RSA (512 bits) con BouncyCastle, y vuelca las claves en binario. **Paquete** (proporcionado por docente) es el contenedor textual (*ASCII-armor*) donde guardamos los bytes de cada elemento (factura cifrada, clave cifrada, firmas, IV, sello).

En **EmpaquetarFactura**, la Empresa lee el JSON de la factura, genera una clave simétrica de sesión AES y un vector de inicialización. Posteriormente cifra el contenido de la factura con la clave AES generada y dicho vector (AES/CBC/PKCS5Padding) para después encapsular la clave simétrica usando la pública de Hacienda (RSA/PKCS1Padding). A continuación firma digitalmente el par crítico FACTURA_CIFRADA || CLAVE_CIFRADA con SHA256withRSA para garantizar integridad y autenticidad en origen. El resultado se vuelca al paquete contenedor con los bloques FACTURA_CIFRADA, CLAVE_CIFRADA, VECTOR_INICIALIZACION y FIRMA_EMPRESA.

En **SellarFactura**, la Autoridad comienza verificando la firma de la Empresa; si falla, aborta el proceso para impedir sellar contenido alterado. Si verifica correctamente, genera un *timestamp* ISO-8601 y lo añade como SELLO_TIEMPO, y luego firma el triple FACTURA_CIFRADA || CLAVE_CIFRADA || SELLO_TIEMPO con SHA256withRSA usando su clave privada, produciendo FIRMA_AUTORIDAD. De este modo, el paquete queda enriquecido con evidencia temporal y no repudio por parte de la Autoridad antes de ser persistido nuevamente.

En **DesempaquetarFactura**, Hacienda exige que el paquete contenga los bloques críticos de Empresa y de Autoridad, y valida primero la firma de la Autoridad y la firma de la Empresa. Solo tras superar ambas verificaciones procede a descifrar la clave AES con su privada RSA, recuperar el vector de inicialización

desde VECTOR_INICIALIZACION y descifrar la factura con AES/CBC, escribiendo el JSON original a disco.

En **SellarFactura** y **DesempaquetarFactura** se utilizan métodos auxiliares para cargar claves públicas/privadas a partir de ficheros binarios (cargarClavePublica, cargarClavePrivada) y para concatenar bytes antes de firmar o verificar (concatenarBytes), evitando duplicación y errores sutiles en la construcción de los mensajes firmados. Asimismo, en cada uno de los cuatro ejecutables de negocio, se incorpora el método auxiliar mensajeAyuda() con la sintaxis exacta requerida por el enunciado, facilitando la ejecución de casos de prueba y asegurando el cumplimiento del contrato de línea de comandos.

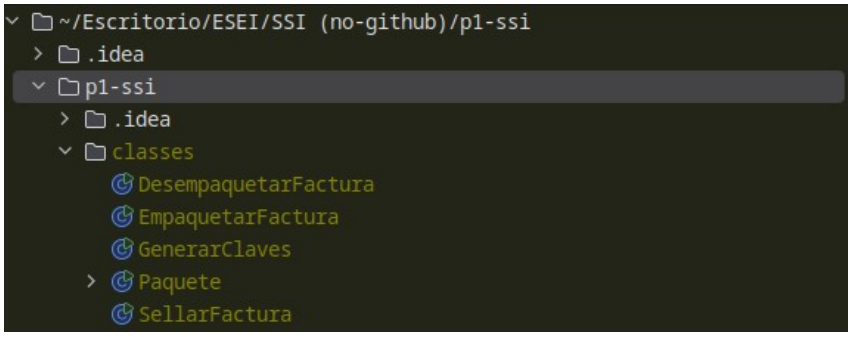
INSTRUCCIONES DE COMPILACIÓN Y EJEMPLOS DE USO

Requisitos: JDK 8+ (Java), librería *BouncyCastle* (bcprov) en el classpath en tiempo de compilación y ejecución.

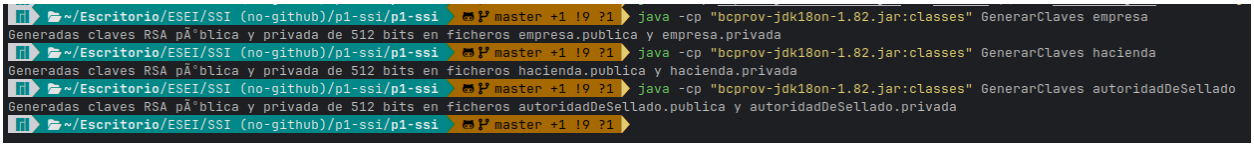
Paso 1: descargar bcprov (por ejemplo bcprov-jdk18on-1.82.jar).

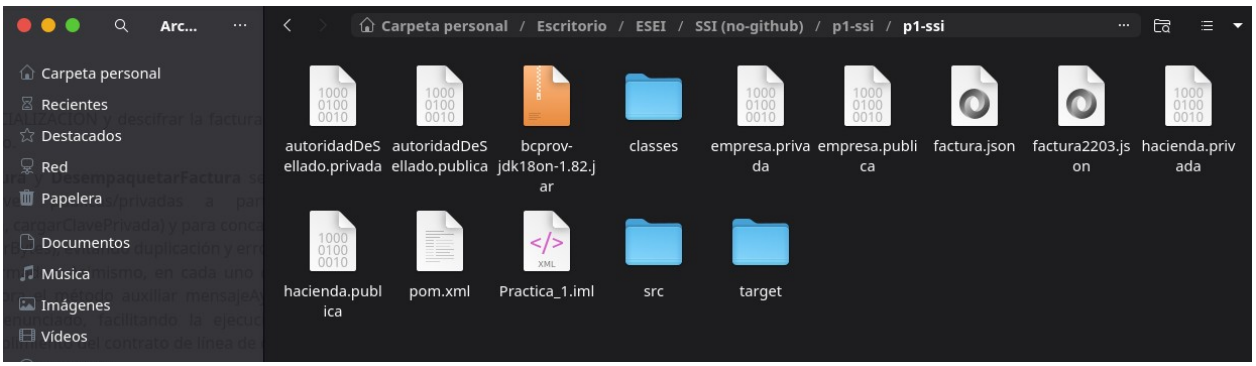
Paso 2: Compilar todo.

```
mkdir -p classes
javac -cp bcprov-jdk18on-1.82.jar -d classes $(find src/main/java -name '*.java')
```



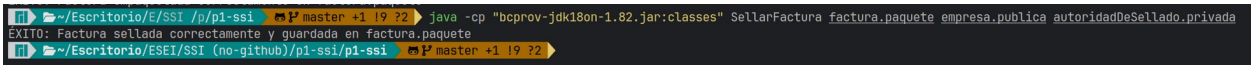
Paso 3: Generar las claves para cada actor.



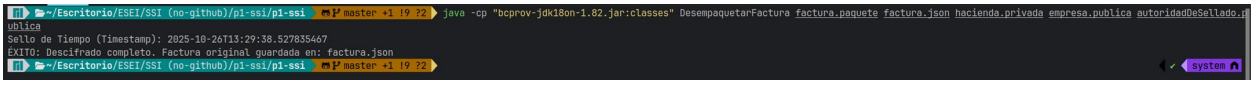


Paso 4: Empaquetado (Empresa).

Paso 5: Sellado (Autoridad).



Paso 6: Desempaquetado (Hacienda).

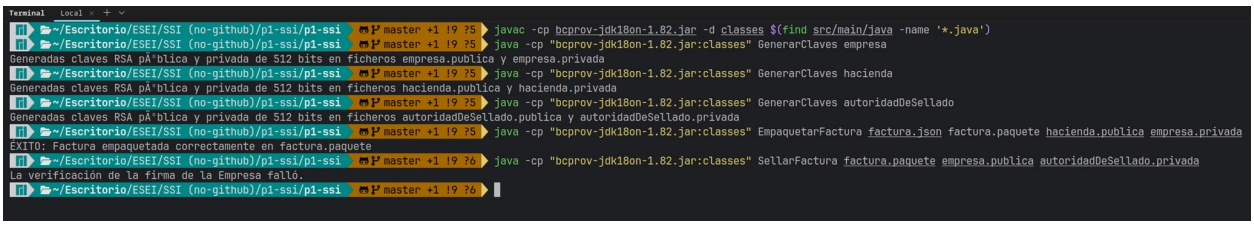


Ejemplos de ejecución de casos de prueba

Caso 1: Recepción de una *Factura Empaquetada* correcta. Se genera un paquete íntegro, firmas válidas y claves correctas; se imprime el sello de tiempo y mensaje de éxito. La captura de pantalla del paso 6, además del archivo adjunto, verifica este caso de prueba.

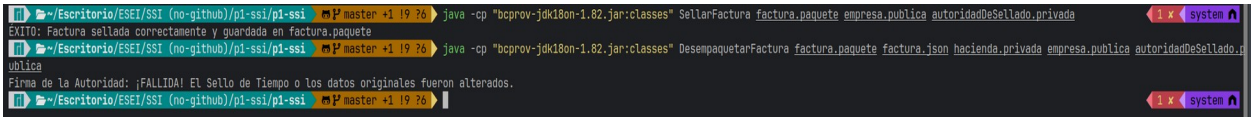
Caso 2: Recepción de una *Factura Empaquetada* con los contenidos aportados por la *Empresa* alterados. Si los bloques que firmó la Autoridad (FACTURA_CIFRADA || CLAVE_CIFRADA || SELLO_TIEMPO) fueron modificados después del sellado, la verificación de la Autoridad falla (se verifica primero).

Para probar este caso, se realizará el procedimiento principal hasta el empaquetado, y luego se alterará a mano un caracter del bloque de FACTURA_CIFRADA (o CLAVE_CIFRADA) de “factura.paquete”. Como consecuencia, obtenemos esta salida:



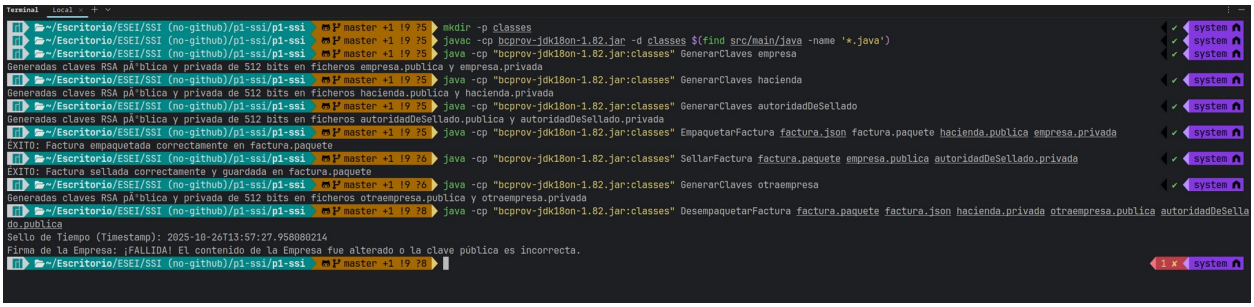
Caso 3: Recepción de una *Factura Empaquetada* con un sello de tiempo alterado. El sello de tiempo forma parte de lo que firmó la Autoridad; cualquier alteración provoca fallo en la verificación de la Autoridad.

Para probar este caso, se alterará un caracter a mano del bloque de SELLO_TIEMPO de “factura.paquete” tras el proceso de sellado, y antes del desempaquetado. Como consecuencia, obtenemos esta salida:



Caso 4: Uso por parte de *Hacienda* de una clave pública de *Empresa* incorrecta. La Autoridad puede haber validado todo, pero Hacienda usa una clave pública de Empresa incorrecta al verificar la firma de la Empresa; entonces la verificación de Empresa falla.

Para probar dicho caso se va a generar otra clave pública de Empresa y con ello vamos a probar cual sería la salida de Desempaquetado si usamos esa *public key*. Como consecuencia, obtenemos esta salida:



SIMPLIFICACIONES REALIZADAS Y CONSECUENCIAS

A continuación se recogen las simplificaciones exigidas por el enunciado junto con su impacto y cómo deberían resolverse en un sistema real.

Claves generadas y guardadas como ficheros simples por cada actor. En la práctica, cada actor (Empresa, Autoridad, Hacienda) crea su propio par de claves y lo almacena en ficheros binarios sin protección adicional. Esto facilita las pruebas, pero en producción la clave privada RSA debe protegerse, para ello, se debería cifrar con una contraseña fuerte, limitar el acceso físico y digital, y almacenar la clave en un dispositivo seguro como un token USB o una tarjeta inteligente; y auditar su uso para evitar su exfiltración o uso indebido.

Sin distribución confiable de claves públicas (sin PKI). Suponemos que cada actor "ya posee" la clave pública correcta del otro. En la realidad esto no es aceptable, se exige infraestructura de clave pública (PKI), certificados X.509, cadenas de confianza y revocación (CRL/OCSP) para impedir ataques de suplantación por intercambio malicioso de claves. En un despliegue real todo esto se resolvería con certificados firmados por una CA común y políticas de gestión del ciclo de vida.

La factura original es un fichero JSON y no se procesa su contenido. Se trata como *blob* opaco: no hay esquema ni validaciones semánticas. En producción suele exigirse un formato normalizado (p.ej., Facturae/UBL/PEPPOL), validaciones y controles antifraude (campos obligatorios, totales, firmas internas, etc.).

Las piezas de información se tratan como String UTF-8 antes/después de cifrado/firma. Esta simplificación es muy práctica, pero en entornos reales existen campos binarios (firmas, sellos, evidencias) y codificaciones explícitas (ASN.1/DER, JSON canonizado, CBOR), para evitar ambigüedades y asegurar reproducibilidad de la firma.

No se implementa almacenamiento/expediente realista. En producción no usaríamos nuestro formato de texto, sino PKCS#7/CMS. El flujo de esta práctica se mapea siempre con el mismo flujo lógico, emitiendo un único *blob* portable, auditable y verificable por terceros.

Validación: mostrar la factura si todas las comprobaciones son correctas. El flujo de validación se limita a verificar firmas y sello y a mostrar el JSON. En producción, además, se registran evidencias, se devuelve motivo detallado de fallo, y se integran políticas de negocio (rechazos formales, reintentos, bitácoras firmadas).

A continuación se recogen las simplificaciones adicionales junto con su impacto y cómo deberían resolverse en un sistema real.

Manejo de errores simple y control por System.exit(1)**.** En entornos reales se requiere de gestión de excepciones, códigos de error normalizados, *logging* firmado, y telemetría para diagnósticos y cumplimiento normativo.

Sin control de reutilización de claves/IV ni políticas de rotación. Para esta práctica cada ejecución genera una clave AES e IV (vector de inicialización) aleatorios. En un entorno real se debería gestionar rotación, no reutilizar IV con la misma clave y registrar material criptográfico temporal.

Cifrado de la clave de sesión con RSA/PKCS1 (no OAEP). El código emplea RSA/ECB/PKCS1Padding. En producción se exige RSA-OAEP por su robustez frente a ataques de *padding*.

Cifrado de datos con AES/CBC sin autenticación. Para ello, primero conviene explicar por qué se ha usado CBC en lugar de ECB. Se ha preferido usar CBC por temas de seguridad, dado que ECB cifra cada bloque de forma independiente y bloques idénticos producen el mismo cifrado. En cambio, CBC encadena los bloques, haciendo que cada bloque dependa de su anterior más un IV aleatorio, por lo que iguala la distribución y oculta patrones aunque haya repeteciones en el texto. Además, CBC con IV aleatorio ofrece confidencialidad semántica frente a ataques de texto en claro elegido (CPA), algo que ECB no proporciona. Aunado a esto, CBC no autentica, por eso se complementa la integridad con firmas. En un entorno real, lo más robusto es usar un modo AEAD como AES-GCM para tener confidencialidad e integridad.

Autoridad de sellado “local” (no TSA estándar). El sellado genera un timestamp ISO-8601 y lo firma con su clave. En un entorno real se usaría una TSA conforme a RFC 3161. Como consecuencia, el sello no es externamente auditable a largo plazo.

RESULTADOS Y CONCLUSIONES

La práctica ha permitido consolidar conocimientos fundamentales sobre criptografía aplicada mediante la implementación exitosa de un sistema de intercambio seguro de facturas electrónicas. Se ha trabajado con el patrón de cifrado híbrido, combinando criptografía simétrica (AES) para datos grandes con asimétrica (RSA) para encapsular claves. La elección de AES/CBC sobre AES/ECB ha evidenciado la importancia de los modos de cifrado para evitar patrones y garantizar confidencialidad semántica, complementándose con un IV aleatorio único por operación.

La implementación ha evidenciado varios principios de diseño seguro fundamentales. La combinación de cifrado, firma y verificación en múltiples capas proporciona defensa en profundidad ante diversos tipos de ataques. Cada actor solo tiene acceso a las claves estrictamente necesarias para su rol, cumpliendo el principio de mínimo privilegio. La validación de firmas antes de procesar contenido previene ataques de manipulación, garantizando que solo se opera sobre datos íntegros.

Si bien la implementación cumple los requisitos de la práctica, se identifican áreas de mejora para un sistema productivo. La incorporación de AES-GCM en lugar de AES-CBC proporcionaría confidencialidad e integridad autenticada en una sola operación AEAD. La implementación de una infraestructura PKI con gestión de certificados X.509, cadenas de confianza y mecanismos de revocación (CRL/OCSP) permitiría validación robusta de identidades.

A modo de consideración, la práctica ha cumplido plenamente su objetivo pedagógico de aplicar la J*ava Cryptography Architecture* para resolver un problema real, aunque algo simple al estar aplicado a alumnos, de intercambio seguro de información.