SEGURIDAD EN REDES Y SERVICIOS

PRÁCTICA 1

Estudio y manejo de APIs criptográficas. Criptografía simétrica y asimétrica

Curso 2018-2019

1 Objetivos

Con el desarrollo de esta práctica se pretende que el alumno consolide los conocimientos sobre los algoritmos criptográficos de cifrado simétrico y asimétrico. Asimismo, se pretende que se familiarice con el manejo de APIs criptográficas para el desarrollo de aplicaciones de seguridad.

El objetivo a alcanzar con esta práctica, cuya consecución es objeto de evaluación, consiste en programar una aplicación que permita, a través de un conjunto de menús y operando sobre ficheros, crear claves simétricas, claves asimétricas y cifrar/descifrar/firmar utilizando las claves generadas. Para ello deberá utilizarse, tal como se indica más adelante, la API criptográfica proporcionada por *Bouncy Castle* (http://www.bouncycastle.org/specifications.html).

2 Definición de la práctica

2.1 Duración

La práctica tiene una duración de **tres semanas** de laboratorio.

2.2 Entorno

La práctica debe ser desarrollada en el lenguaje de programación Java y utilizando la API criptográfica *Bouncy Castle versión 1.60*[1]. No se exige la utilización de un entorno de desarrollo concreto, si bien se recomienda utilizar el IDE Eclipse para facilitar las tareas de programación. Asimismo, se recomienda utilizar una versión del JDK de Java igual o superior a la 1.6.

Como paso previo al comienzo de la práctica se debe configurar la instalación de Java para ser capaz de interactuar con las librerías criptográficas proporcionadas por *Bouncy Castle*. En el Anexo 1 se encuentran las instrucciones para llevar a cabo dicha configuración.

2.3 Estructura

La práctica se estructura en dos partes, una dedicada a la criptografía simétrica y otra a la criptografía asimétrica.

El programa a desarrollar debe mostrar al usuario un menú en el que se le permita elegir el tipo de criptografía a utilizar, tal como se muestra en la Figura 1.

```
¿Qué tipo de criptografía desea utilizar?
1. Simétrica.
2. Asimétrica.
3. Salir.
```

Figura 1 – Menú de primer nivel

[1] No se admitirán prácticas que empleen versiones anteriores de esta librería o que usen clases obsoletas.

2.3.1 Criptografía simétrica

En el caso de la criptografía simétrica se utilizará:

- Algoritmo **Rijndael**.
- Tamaño de clave: 256 bits.
- Tamaño de bloque: 192 bits.
- Encadenamiento de bloques: CBC.
- Relleno (*padding*): según recomendación ANSI X.923.

Una vez que el usuario de la aplicación selecciona la opción de criptografía simétrica en el menú principal, se le debe mostrar un segundo menú (ver Figura 2) con las opciones correspondientes a generar una clave, cifrar y descifrar.

```
Elija una opción para CRIPTOGRAFIA SIMÉTRICA:
0. Volver al menú anterior.
1. Generar clave.
2. Cifrado.
3. Descifrado.
```

Figura 2 – Menú de segundo nivel: criptografía simétrica

Todas las operaciones están referidas al algoritmo Rijndael y su comportamiento es el siguiente:

- **Generar una clave**: Permite generar una clave de 256 bits. La clave binaria generada será una secuencia de unos y ceros, por lo que, si intentamos visualizarla, el resultado será una ristra de caracteres ininteligible. Para facilitar su visualización, la clave deberá almacenarse como una representación legible en formato hexadecimal. A continuación, se muestra un ejemplo del resultado con un único octeto de la clave generada:

  Clave generada (1 octeto): 1000 0001
  Representación en hexadecimal como caracteres (2 octetos): 81

  De esta manera una clave binaria de 256 bits (32 octetos) al ser representada en formato hexadecimal tendrá una longitud de 64 octetos.

- **Cifrar**: Permite cifrar el contenido de un fichero de entrada, utilizando para ello la clave leída de otro fichero y almacenar el resultado del cifrado en un fichero de salida. Para ello se le solicita al usuario el nombre del fichero que contiene la clave a utilizar, el nombre del fichero de entrada con los datos "en claro" y el nombre del fichero de salida donde se almacenarán los datos cifrados. Los datos se almacenarán en binario (tal cual resulten de la operación de cifrado, sin realizar ninguna conversión ni añadir ningún carácter extra como retornos de carro o saltos de línea) (ver el apartado de 4 *Cuestiones generales*). El fichero a cifrar puede tener, a priori, cualquier contenido, aunque se recomienda hacer las pruebas al principio con unos pocos caracteres, luego con varias líneas de texto y posteriormente con una imagen.

- **Descifrar**: Permite descifrar el contenido de un fichero de entrada, utilizando la clave leída de otro fichero y almacenar el resultado del descifrado en un fichero de salida. Para ello se le solicitará al usuario el nombre del fichero que contiene la clave a utilizar, el nombre del fichero de entrada con los datos cifrados, y el nombre del fichero de salida donde se almacenará el resultado del descifrado.

2.3.2 Parte 2: Criptografía asimétrica

En el caso de la criptografía asimétrica se utilizará el algoritmo RSA, con un tamaño de clave de 1024 bits. Como función resumen se utilizará el algoritmo SHA-256.

Una vez el usuario de la aplicación selecciona la opción de criptografía asimétrica en el menú principal, se le debe mostrar un segundo menú (ver Figura 3) con las opciones correspondientes a generar una pareja de claves, cifrar, descifrar, firmar y verificar la firma.

```
Elija una opción para CRIPTOGRAFIA ASIMÉTRICA:
0. Volver al menú anterior.
1. Generar pareja de claves.
2. Cifrado.
3. Descifrado.
4. Firmar digitalmente.
5. Verificar firma digital.
```

Figura 3 – Menú de segundo nivel: criptografía asimétrica

Todas las operaciones están referidas al algoritmo RSA y su comportamiento es el siguiente:

- **Generar una pareja de claves**: Permite generar una pareja de claves RSA de 1024 bits en formato módulo/exponente. Cada clave de la pareja generada se almacenará en un fichero independiente, cuyo nombre se le solicita al usuario en el momento de la generación. Al igual que en el caso anterior, guardaremos los resultados en una representación legible en hexadecimal. El formato de cada uno de estos ficheros será: **módulo** en hexadecimal seguido de un carácter CR y otro LF y luego el **exponente** también en hexadecimal, sin ningún carácter adicional.

  Las claves pública y privada también deberán almacenarse en formato PEM (ver Anexo 2), utilizando para ello la clase *GuardarFormatoPEM* que se proporciona. Esta clase almacenará las claves pública y privada en los ficheros PublicaPEM.txt y PrivadaPEM.txt, respectivamente. Estas claves almacenadas en formato PEM no serán usadas en el resto de la práctica. El único propósito de esta operación es que el alumno se familiarice con el formato de representación de estructuras X.509 que se empleará en la práctica 2.

- **Cifrar**: Permite cifrar el contenido de un fichero de entrada, utilizando la clave leída de otro fichero (formato hexadecimal) y almacenar el resultado del cifrado en un fichero de salida. Para ello se le solicita al usuario el nombre del fichero de entrada con los datos "en claro", el nombre del fichero que contiene la clave pública o privada[2] a utilizar y el nombre del fichero de salida donde se almacenarán los datos cifrados. Los datos se almacenarán en binario (tal cual resulten de la operación de cifrado, sin realizar ninguna conversión ni añadir ningún carácter extra, tales como retornos de carro o saltos de línea). El fichero a cifrar puede tener, a priori, cualquier contenido, aunque se recomienda hacer las pruebas con ficheros de texto que permitan comprobar fácilmente si el cifrado/descifrado se ha realizado correctamente (al principio con unos pocos caracteres, luego con varias líneas de texto y posteriormente con una imagen).

- **Descifrar**: Permite descifrar el contenido de un fichero de entrada utilizando la clave leída de otro fichero y almacenar el resultado del descifrado en un fichero de salida. Para ello se le solicita al usuario el nombre del fichero de entrada con los datos cifrados, el nombre del fichero que contiene la clave privada o pública a utilizar y el nombre del fichero de salida donde se almacenará el resultado del descifrado (los datos "en claro").

- **Firmar**: Permite leer el contenido de un fichero de entrada y realizar una firma digital (basada en la función resumen SHA-256) utilizando la clave privada leída de otro fichero. El resultado de la firma se almacenará en un fichero de salida. Para ello se le solicita al usuario el nombre del fichero

[2] El programa debe permitir que la opción de cifrar pueda hacerse bien con la clave pública, en cuyo caso se deberá utilizar la clave privada para descifrar o bien con la clave privada, en cuyo caso se deberá utilizar la clave pública para descifrar.

de entrada con los datos a firmar, el nombre del fichero que contiene la clave a utilizar (clave privada) y el nombre del fichero de salida donde se almacenará la firma.

- **Verificar firma**: Comprueba la firma digital asociada a un fichero indicando si es correcta o no. Para ello solicita al usuario el fichero original, el fichero con la firma digital y el fichero con la clave pública utilizada. Aplicando la clave pública a la firma digital se obtendrá el resumen original que se comparará con el resumen SHA-256 obtenido a partir del fichero original.

3 Librería criptográfica Bouncy Castle

Cuando los programadores afrontan la integración de técnicas criptográficas en una aplicación es necesario tener en cuenta dos cuestiones: los algoritmos y protocolos criptográficos que se utilizarán para cumplir con los requisitos de seguridad y las herramientas disponibles para desarrollarlos. De esta forma el programador debe enfrentarse a dos abstracciones distintas: las relacionadas con la criptografía y las relacionadas con las acciones computacionales necesarias para llevarlas a cabo. Dicho de una forma más sencilla, una cosa es saber cómo funciona un algoritmo criptográfico y otra cómo implementarlo en un entorno computacional.

Existen algoritmos en los que las dos abstracciones son muy parecidas ya que la propia definición del algoritmo se realiza en términos computacionales. Sin embargo, hay algoritmos donde la diferencia es significativa. Un ejemplo es el algoritmo RSA. La base de funcionamiento de este algoritmo es la matemática modular. Eso hace que los mensajes sean tratados como números. Sin embargo, a la hora de implementarlo se requiere un criterio para poder dividir un mensaje en números que puedan ser tratados por el algoritmo. Además, se requiere definir un formato y codificación estándar para esos números y tratar todos los casos particulares existentes relacionados con ese paso del mundo de las matemáticas (abstracción matemática) al mundo computacional (abstracción computacional).

Además, según las herramientas de desarrollo de las que se disponga, la abstracción computacional puede ser diferente. Las herramientas de desarrollo pueden ser muy variadas: implementación de los algoritmos en el lenguaje de programación elegido, uso de librerías externas, uso de recursos propios del sistema operativo, acceso a hardware especializado, etc. Cada una de estas herramientas provee primitivas distintas para llevar a cabo la transformación del algoritmo matemático a acciones computacionales. En esta práctica se utilizará el lenguaje de programación *Java* y la librería *Bouncy Castle*.

Java, además de un lenguaje de programación, define un conjunto de recursos disponibles para los programadores de forma independiente al sistema operativo nativo sobre el que se ejecute la aplicación. Dentro de sus recursos estándar ofrece un marco para el desarrollo de servicios criptográficos denominado *JCA* (*Java Cryptography Architecture*). El *JCA* permite no solo el acceso a servicios criptográficos implementados en la plataforma *Java,* sino que cualquier proveedor de servicios criptográficos pueda integrar sus soluciones dentro de este marco. De esta forma el programador, en vez de usar el proveedor criptográfico estándar que trae la plataforma Java, puede asociar otro proveedor criptográfico que, por ejemplo, tenga más rendimiento en la ejecución de algoritmos. El esquema de proveedor de servicios criptográficos tiene una ventaja en cuanto a la interoperabilidad de soluciones: todos los servicios de criptografía de todos los proveedores compatibles con *JCA* tienen la misma forma de acceso. Eso significa que si se realiza la aplicación sobre *JCA* se podría cambiar el proveedor de servicios criptográficos sin modificar el código fuente de la aplicación.

Bouncy Castle es una librería criptográfica para *Java* y *C#*. En su versión *Java* puede utilizarse de dos formas distintas: como proveedor de servicios criptográficos compatible con *JCA* o mediante acceso directo a su

API (llamada *lightweight API* en la documentación). Si se utiliza como proveedor de servicios criptográficos hay que entender el esquema de acceso propuesto por *JCA* para su uso. Si se accede a través de la *lightweight API* será necesario entender el esquema de acceso que propone la librería *Bouncy Castle*. Para esta práctica se utilizará el esquema basado en la *lightweight API*, es decir, que no se utilizará el marco propuesto por *JCA*.

3.1 Visión general de Bouncy Castle lightweight API

Al comenzar con una nueva librería es necesario saber qué servicios ofrece y a través de qué interfaz. Como *Java* es lenguaje de objetos la librería estará estructurada en clases. Las citadas clases, así como su jerarquía, describen los servicios de la librería y su forma de acceso. Por lo tanto, hay que comenzar a identificar las clases existentes y su utilidad para llevar a cabo las técnicas criptográficas deseadas.

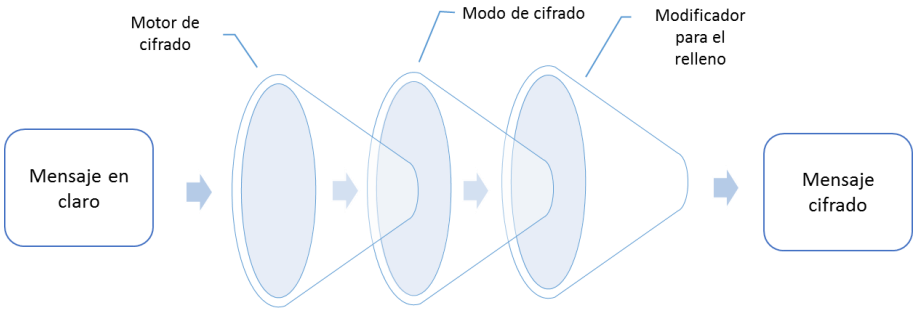
En lo referente a los algoritmos de cifrado, varias librerías criptográficas (como es el caso de *Bouncy Castle*) diferencian entre cifrado simétrico y asimétrico. Por lo tanto, es bueno seguir estrategias distintas para la detección de clases.

3.1.1 Cifrado simétrico basado en bloques

Como estamos usando Java es común el uso de *wrappers* (envoltorios) para ir añadiendo funcionalidad a una clase básica. Un ejemplo de este patrón es la entrada/salida en Java. Si se requiere crear un flujo de salida con formato, primero se genera un flujo de salida básico, enlazado con el dispositivo de salida al cual se mandará la información. Posteriormente se crea otra instancia de flujo de salida, que contiene al primero, y que da el formato o el comportamiento deseado a esa salida. A continuación, se muestra un ejemplo de cómo se asocia un fichero de salida a varias clases distintas para obtener finalmente un objeto que permita formatear de forma sencilla los datos:

```
fsalida = new BufferedOutputStream(new FileOutputStream(NombreFicheroSalida));
```

Para el cifrado simétrico la librería sigue el mismo patrón. Es necesario crear un motor de cifrado y después un modo de cifrado. Estas cuestiones son dependientes del algoritmo matemático a emplear. Una cuestión importante, que ya tiene que ver con la implementación en un sistema computacional, es el relleno (*padding*) aplicado. En la *Bouncy Castle Lighweight API* el relleno se trata de nuevo como una clase más que puede envolver a las otras creadas. Finalmente queda un esquema de clases como el siguiente:



La figura anterior muestra el hecho de que existirá un patrón parecido al mostrado con los flujos de entrada/salida. Será trabajo del estudiante buscar e identificar las clases concretas que realizan esa labor.

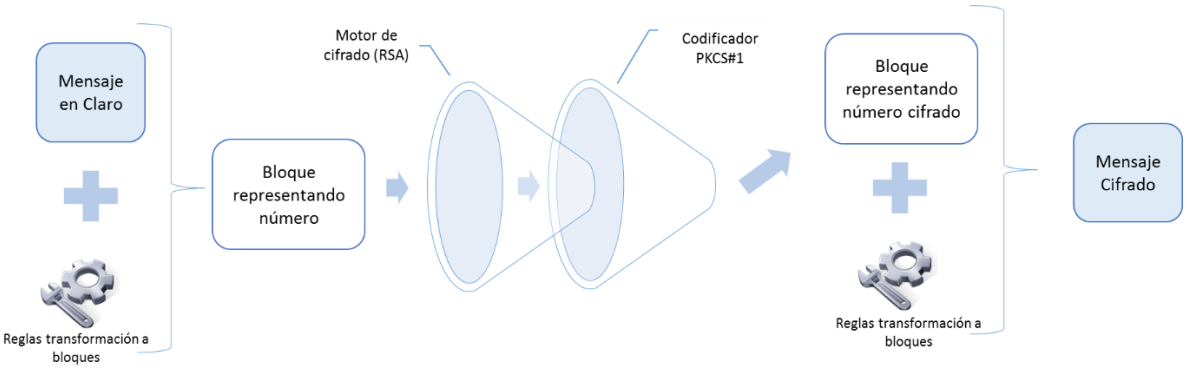
Otro aspecto fundamental es la división del texto a cifrar en bloques. Todo texto debe ser dividido en bloques ya que el cifrador elegido es un cifrador de bloques. Hay que generar el código necesario para ir introduciendo los bloques del tamaño adecuado en todo momento. En el proceso de cifrado simplemente hay que consultar el tamaño de bloque permitido en el motor (típicamente hay un método para ello). Si el tamaño del bloque de salida no es igual que el tamaño del bloque de entrada, además hay que conocer el tamaño del citado bloque de salida para poder realizar el proceso de descifrado. Además es típico que el último bloque se trate de forma independiente al resto mediante la invocación de algún método especial[3].

3.1.2 Cifrado asimétrico

El cifrado asimétrico de nuevo sigue el patrón de envoltorio visto en el cifrado simétrico por bloques, aunque aplicado a este tipo de algoritmos. Por ejemplo, no es necesario realizar tantos envoltorios si bien la generación de claves es algo más sofisticada.

Un aspecto importante en este tipo de algoritmos es la obtención del tamaño del bloque. Una cuestión importante es decidir cuál es el tamaño del bloque ya que teóricamente puede ser un valor arbitrario siempre que sea compatible con el algoritmo (por ejemplo, en RSA el tamaño del bloque no puede permitir que el número a cifrar sea mayor que el módulo de la operación). Para aligerar este proceso de cálculo la *Bouncy Castle Lighweight API* ofrece en sus clases de cifrado asimétrico dos métodos que calculan automáticamente los tamaños de bloque tanto de entrada como de salida (similar al del cifrado simétrico en bloque).

El tamaño de bloque debe seguir algún estándar ya que de lo contrario distintas implementaciones podrían tener distintos tamaños de bloque que las harían incompatibles. Además del tamaño del bloque hay más aspectos que deben ser estandarizados al aplicar el algoritmo RSA a soluciones computacionales. Todos estos aspectos están recogidos en las normas *PKCS*. A la hora de cifrar y descifrar debe tenerse en cuenta básicamente el estándar PKCS#1. Para ello de nuevo la *Bouncy Castle Lighweight API* ofrece una clase que permite que el proceso de cifrado y descifrado se realice conforme al estándar PKCS#1.



[3] En nuestro caso será necesario emplear el método **doFinal()** para procesar el último bloque cifrado/descifrado del buffer interno. La implementación del algoritmo en modo CBC mantiene un buffer interno que debe ser también grabado en el fichero de salida.

3.1.3 Firma Digital

El proceso de firma digital, tal y como se ha visto en clase de teoría, involucra varios algoritmos para poder proveer los servicios de seguridad que propone. Desde el punto de vista de su implementación y su integración en soluciones reales el proceso es bastante más complejo ya que debe garantizarse la interoperabilidad entre soluciones. Esto hace que existan protocolos de generación y representación de firmas digitales relativamente sofisticados y sujetos, en la mayoría de los casos, a estándares.

Para esta práctica no se utilizará ningún estándar de representación para firma digital. La firma consistirá únicamente en el cifrado RSA con clave privada de un resumen realizado con el algoritmo SHA-256. El resultado del proceso se almacenará en un fichero en formato binario (no es necesario aplicar ninguna codificación especial).

Tal y como se indicó previamente no se permitirá el uso de la clase *RSADigestSigner* que realiza el proceso automáticamente. En vez de ello los estudiantes deben utilizar las clases previamente creadas para el cifrado/descifrado RSA y utilizar los servicios de obtención de resúmenes ofrecidos por la librería.

4 Cuestiones generales

- La práctica deberá estar estructurada de acuerdo a la orientación a objetos del lenguaje Java, es decir, no se admitirán prácticas con un solo fichero `.java` en el que se recoja todo el código solicitado. Deberán existir, al menos, tres clases, una para el programa principal, una para el cifrado simétrico y otra para el asimétrico.
- Se deberán utilizar las clases proporcionadas por *Bouncy Castle Lighweight API*, no pudiendo utilizarse las funciones definidas en el *JCA* salvo para la generación de números aleatorios (ver Anexo 1).
- El proceso de firma deberá realizarse a partir de primitivas de cifrado y descifrado RSA y la función resumen SHA-256. Aunque la *Bouncy Castle Lighweight API* provee la clase *RSADigestSigner* que lo realiza de forma automática, no podrá ser utilizada.
- Debe tenerse especial cuidado en que durante los procesos de cifrado/descifrado o generación de claves no se introduzcan caracteres adicionales en los ficheros (como caracteres NULL, CR o LF). A fin de comprobar que el resultado de nuestra operación es correcto se recomienda utilizar un editor que muestre los caracteres no imprimibles, como por ejemplo el Notepad++.

5 Pruebas

- El cifrado/descifrado deberá funcionar correctamente tanto con ficheros de texto como binarios (imágenes, por ejemplo) y, a priori, de cualquier tamaño.
- En el cifrado simétrico se exigirá que cualquier otro compañero del aula pueda descifrar nuestro fichero a partir de la clave simétrica que hemos empleado (en formato hexadecimal) y del fichero cifrado.
- En el cifrado asimétrico se exigirá que cualquier otro compañero del aula pueda descifrar nuestro fichero a partir de la clave pública que hemos empleado y del fichero cifrado. Asimismo, debe ser posible que cualquiera pueda verificar la firma que hemos creado, al pasarle el fichero original y nuestra clave pública.

6 Entrega

Los resultados deben ser entregados a través de la plataforma Moodle. La fecha tope de entrega para cada grupo CINCO DÍAS DESPUÉS de la última sesión de laboratorio, esto es:

Grupo **L5**: viernes 1 de marzo de 2019 a las 23:50

Grupo **M3**: sábado 23 de febrero de 2019 a las 23:50

Grupo **X5**: domingo 24 de febrero de 2019 a las 23:50

No se admitirá ninguna entrega por correo electrónico.

Cada alumno de laboratorio deberá entregar los ficheros correspondientes a su código fuente **debidamente comentados** en el espacio habilitado a tal efecto en la plataforma Moodle.

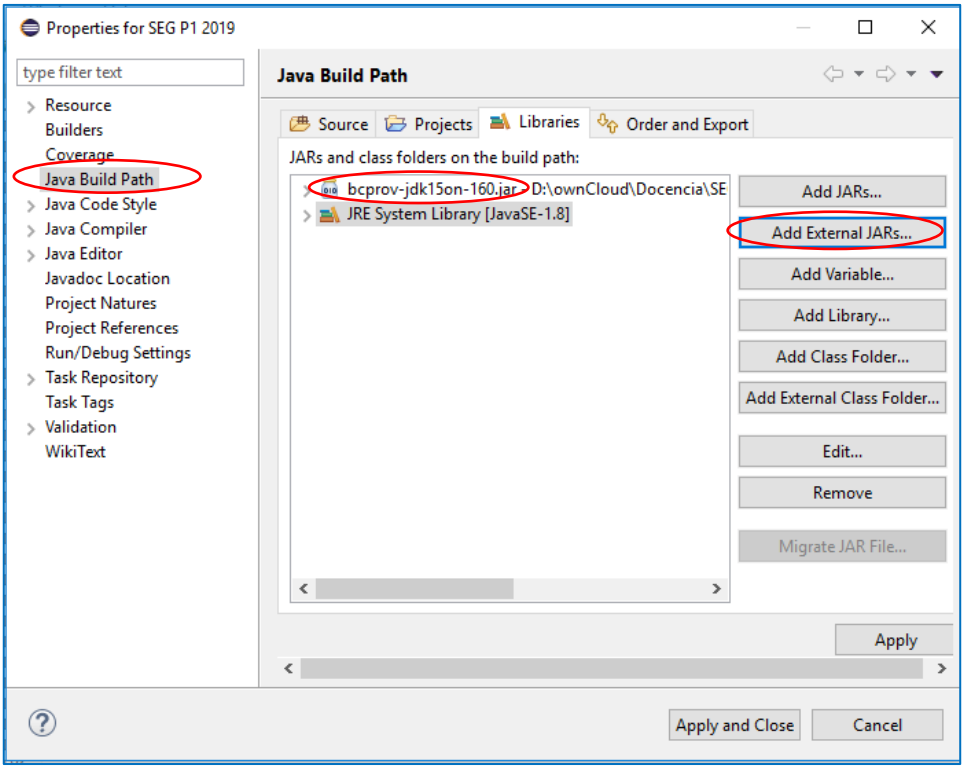
Anexo 1. Uso librerías Bouncycastle

Las librerías criptográficas de Bouncy Castle correspondientes a la versión 1.60 se encuentran instaladas en los módulos del laboratorio (bcprov-jdk15on-160.jar). Si se desea emplear el ordenador propio para hacer la práctica pueden descargarse en el siguiente enlace la última versión disponible, la 1.60:

http://www.bouncycastle.org/download/bcprov-jdk15on-160.jar

Ambas librerías son totalmente válidas para realizar la práctica. Las librerías descargadas pueden ser colocadas en cualquier lugar del sistema. No obstante, para evitar posibles problemas, se recomienda que los ficheros con extensión JAR sean colocados en el directorio `$JAVA_HOME/jre/lib/ext`.

En cualquier caso, habrá que indicar a Java que vamos a usar las librerías de Bouncy Castle. Para ello, cambiaremos las propiedades de nuestro proyecto para añadir las librerías que hemos descargado.



La documentación (Javadoc) correspondiente a la API se encuentra en el siguiente enlace:

http://www.bouncycastle.org/docs/docs1.5on/index.html

Anexo 2: Formatos de almacenamiento de claves

La Recomendación X.509 especifica la sintaxis de un certificado en lenguaje ASN.1 (*Abstract Syntax Notation One*). Este lenguaje formal permite la definición de los tipos de datos y los valores, y cómo se utilizan y combinan esos tipos de datos y valores en las diversas estructuras de datos. El objetivo de este estándar es definir una sintaxis abstracta de la información, sin obligar al autor de una Recomendación a que especifique cómo se codifica la información para su transmisión.

Un ejemplo de un certificado X.509 en formato ASN.1 es:

```
Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }
CertificateSerialNumber  ::=  INTEGER
Validity ::= SEQUENCE {
notBefore      Time,
notAfter       Time }
Time ::= CHOICE {
   utcTime        UTCTime,
   generalTime    GeneralizedTime }
```

Es necesario, por tanto, recurrir a otros estándares para indicar cómo se van a codificar los diversos tipos de datos y sus valores. Estos estándares definen un conjunto de reglas para codificar datos ASN.1 en una secuencia de octetos que se pueden transmitir a través de un enlace. Dos de los formatos más extendidos son BER (***Basic Encoding Rules*)** y DER (***Distinguished Encoding Rules***), en particular se emplea **DER** cuando se habla de firma electrónica y certificación.

De esta manera tenemos los formatos PKCS#8 y SPKI que permiten representar respectivamente la clave privada y la clave pública:

PKCS#8: Codificación en DER de la estructura **PrivateKeyInfo**
SPKI: Codificación en DER de la estructura **SubjectPublicKeyInfo**

Como resultado de esta codificación obtendremos datos binarios que entenderán transmisor y receptor, pero que, sin embargo, pueden ser difíciles de manejar para las aplicaciones pensadas para manejar solo datos de texto. Para facilitar el manejo de información en formato DER o BER se utiliza el sistema de intercambio de información diseñado para PEM (*Privacy-enhanced Electronic Mail*). Un archivo PEM es una versión imprimible del archivo DER, donde la información se ha codificado en BASE64 y está delimitada por una cabecera y un pie (ver ejemplos más adelante).

Para más información consultar:

http://www.cisco.com/c/en/us/support/docs/security/vpn-client/116039-pki-data-formats-00.html

PKCS#8: Información de clave privada (*Private Key Info*)

Clave privada en formato ASN.1:

```
PrivateKeyInfo ::= SEQUENCE {
        version                   Version,
        privateKeyAlgorithm       PrivateKeyAlgorithmIdentifier,
        privateKey                PrivateKey,
        attributes           [0]  IMPLICIT Attributes OPTIONAL }
```

Una vez codificada en formato DER y representada en formato PEM para hacer legible su contenido, la clave privada tendrá el siguiente aspecto:

```
-----BEGIN PRIVATE KEY-----
MIICdgIBADANBgkqhkiG9w0BAQEFAASCAmAwggJcAgEAAoGBALc5rXS3XYptd+h/
gTZicPKljCRRXuiFk2JcrD8VPIZFoHJO36BlTjPCQI3COurEqcWHMjANvf9DmyD9
Uxc0FW6GvzDvfm1HwHQxdlbqBmJhSUxys+hz+sc04HyIHGhDadxVLLiSUbV3MT29
5BtA9XcfEHOl6fmatr13Fgphap2vAgERAoGAO0ddw+D4nboBJZKpzdSN9CaAKd4X
LRwoJ1o3ug5mZ60dUiiTpNV7Ltz2xHOLiDCRR2+BLaoX0pXfXX8iaWOslCYV7GZc
Nmy/KOXG+SIvfVv/R8khfSu9Teg2hL8GD+o4t53OZr5nLDWHjP+J1SRePw34sYqw
tujM3oKITOOy/sECQQDyjitLoO4NycsOQgTXWLxur+DWoL/FP5erjqJXkxXLW7Hc
U+olp+CTaO/WFqrpWGPcewv4pfn+dR4Ivk9xhceRAkEAwWGfUDTXARwquhMhenfz
7YRfLmXUVhT46h20PzDEL7vy7ixLCN9MPlPf6Tqu/nnEuk7l0gqEb5nYZoPkjS/x
PwJBALl7xsFc8kbHfSkFTv8Hnyd3b7MvobT0ZOyaP+idp0EoAHtPOpVEJDR9bBwv
c6Nhtca4cpD3ZNGk2rtkWt5XPkECQQCIgSUpjrXiqniDWMxWcspNTmFN7YbTWhkd
up1ZyBIDk7qKASXoJSbClY7+/D9KN9YpKKI56U5s5RE5Tgq+A7lZAkEAqRZu5vMi
0Lg+IIvADDAoASEgZ519ND1hK9cEUfO2sh3Jb9jURyznYP8W/WV9G2/+e9MlHdg3
I+LLsofvVyzNfA==
-----END PRIVATE KEY-----
```

SPKI: Información de la clave pública

Clave pública en formato ASN.1:

```
SubjectPublicKeyInfo  ::=  SEQUENCE  {
        algorithm            AlgorithmIdentifier,
        subjectPublicKey     BIT STRING  }
```

Una vez codificada en formato DER y representada en formato PEM para hacer legible su contenido, la clave pública tendrá el siguiente aspecto:

```
-----BEGIN PUBLIC KEY-----
MIGdMA0GCSqGSIb3DQEBAQUAA4GLADCBhwKBgQC3Oa10t12KbXfof4E2YnDypYwk
UV7ohZNiXKw/FTyGRaByTt+gZU4zwkCNwjrqxKnFhzIwDb3/Q5sg/VMXNBVuhr8w
735tR8B0MXZW6gZiYUlMcrPoc/rHNOB8iBxoQ2ncVSy4klG1dzE9veQbQPV3HxBz
pen5mra9dxYKYWqdrwIBEQ==
-----END PUBLIC KEY-----
```