

# Cryptographie avec Bouncy Castle



par [nyal](#)

Date de publication : 07/07/2004

Dernière mise à jour :

Ce tutoriel constitue une première approche pour l'utilisation de la bibliothèque Bouncy Castle.

- 1 - Introduction
- 2 - Installation
  - 2.1 - Bibliothèque Bouncy Castle
  - 2.2 - JCE: Unlimited Strength Jurisdiction Policy Files
  - 2.3 - Certificats
  - 2.4 - Cryptonit
- 3 - Gestion des certificats
  - 3.1 - Les certificats PKCS#12 (.p12, .pfx)
  - 3.2 - Les certificats X.509 (.cer)
- 4 - Signature d'un document
  - 4.1 - Signer au format CMS/PKCS#7
  - 4.2 - Vérifier la signature
- 5 - Chiffrement d'un document
  - 5.1 - Chiffrement par clé symétrique
  - 5.2 - Chiffrement par clé asymétrique (format CMS/PKCS#7)
- 6 - L'empreinte d'un document
- 7 - Conclusion
- Remerciements

## 1 - Introduction

Bouncy Castle est une bibliothèque de cryptographie libre et open-source. Elle s'apparente à la librairie C [openssl](#) qui est conforme aux différents standards en vigueur.

Bouncy Castle n'est pas installé de base sur les plateformes java. Suivez donc bien les consignes d'installation.

Ce tutoriel va vous expliquer comment réaliser les opérations les plus courantes de cryptographie (signature, chiffrement,...) avec des exemples clairs.

Il est nécessaire d'avoir des notions en cryptographie. Car même si des explications seront apportées sur les méthodes, elles ne seront peut être pas suffisantes pour une compréhension totale.

Bonne lecture.

## 2 - Installation

### 2.1 - Bibliothèque Bouncy Castle

Téléchargez les deux bibliothèques sur le site [Bouncy Castle](#):

- `bcprov.jar`
- `bcmail.jar`

Choisissez les fichiers correspondant à votre JDK (normalement, ils sont présents pour toutes les versions). Maintenant déplacez les deux `.jar` dans le répertoire `lib/ext` de votre JDK.

### 2.2 - JCE: Unlimited Strength Jurisdiction Policy Files

Cette partie concerne les versions 1.4 et supérieures du JDK. Ces JDK possèdent des limitations au niveau de la cryptographie. Cela est dû aux législations de certains pays qui interdisent de chiffrer avec des clés de taille trop importante selon les algorithmes.

Téléchargez donc les **Unlimited Strength Jurisdiction Policy Files** pour votre JDK. L'archive se trouve sur le site de sun à l'endroit où vous avez téléchargé votre JDK (se situe au bas de la page).

Une fois l'archive téléchargée, décompressez-la et copiez les fichiers dans le répertoire `lib/security/` de votre JDK (vous pouvez écraser les anciens).

Si cette manipulation a mal été réalisée, vous aurez un message d'erreur avec les exemples de ce tutoriel :

```
java.lang.SecurityException: Unsupported keysize or algorithm parameters
at javax.crypto.Cipher.init(DashoA6275)
```

### 2.3 - Certificats

Les différents exemples de ce tutoriel ont besoin de certificats. Je vous fournis donc des certificats pour que vous puissiez faire des tests :

- [personnal\\_nyal.p12](#) avec pour password `2[$0wUOS`
- [personnal\\_nyal.cer](#)

Ces certificats ne sont que des certificats de test. Mais je vous prie de les utiliser pour tester les exemples de cet article seulement. Sinon je devrais les révoquer.

Vous pouvez vous en procurer gratuitement chez les AC (authority certification) suivantes (par exemple):

- [thawte](#)
- [globalsign](#)
- ...

### 2.4 - Cryptonit

Cryptonit est un logiciel développé en C++ par la société [idealx](#).

Ce logiciel est libre et téléchargeable [ici](#) pour toutes les plateformes (windows, linux, mac,...).

Cryptonit, qui est un logiciel de cryptographie, va nous permettre de vérifier si les fichiers créés seront bien utilisables par les autres applications du marché.

## 3 - Gestion des certificats

### 3.1 - Les certificats PKCS#12 (.p12, .pfx)

Le format PKCS#12 permet de rassembler:

- Paire de clé privée/publique
- Certificat X.509 de la clé publique de la paire
- Certificat X.509 signataire
- Chaîne des certificats jusqu'au certificat ROOT (peut ne pas être présent)

Ce format contient toutes les informations nécessaires pour installer et vérifier la provenance d'un certificat. Ainsi, lorsque vous ferez une demande pour un certificat, l'AC vous fournira un fichier PKCS#12 que vous pourrez utiliser dans vos logiciels : navigateur web, logiciel de messagerie, ... Les fichiers PKCS#12 auront pour extension **.p12** voir **.pfx**.



*Evitez d'utiliser l'extension .pfx qui correspond à "l'ancêtre" du format .p12. Cependant, Microsoft utilise toujours cette extension et contribue à la confusion entre les formats.*

Lors de la création de votre fichier PKCS#12, l'AC va hacher puis signer une partie de votre certificat X.509 nommée TBS (to be signed) qui contient diverses informations comme la clé publique, les dates de validité,... Une fois la signature réalisée, elle est placée dans votre certificat public avec la fonction hachage et de chiffrement asymétrique utilisée (sha1WithRSAEncryption par exemple). Vous devez comprendre que cette signature permettra de vérifier la validité de votre certificat. Pour plus d'informations sur les signatures, vous pouvez vous référer à la section sur [la signature d'un document](#).

Nous allons voir comment récupérer la paire de clés privée/publique et votre certificat public. Ces éléments serviront dans les chapitres suivants pour réaliser les opérations de chiffrement.

```
import java.io.*;
import java.util.*;
import java.security.*;

import org.bouncycastle.jce.provider.BouncyCastleProvider;
import java.security.cert.X509Certificate;

// CHARGEMENT DU FICHER PKCS#12

Keystore ks = null;
char[] password = null;

Security.addProvider(new BouncyCastleProvider());
try {
    ks = KeyStore.getInstance("PKCS12");
    // Password pour le fichier personnel_nyal.p12
    password = "2!$0wUOS".toCharArray();
    ks.load(new FileInputStream("personnel_nyal.p12"), password);
} catch (Exception e) {
    System.out.println("Erreur: fichier " +
        "personnel_nyal.p12" +
        " n'est pas un fichier pkcs#12 valide ou passphrase incorrect");
}

return ;

// RECUPERATION DU COUPLE CLE PRIVEE/PUBLIQUE ET DU CERTIFICAT PUBLIQUE

X509Certificate cert = null;
PrivateKey privatekey = null;
PublicKey publickey = null;
```

```

try {
    Enumeration en = ks.aliases();
    String ALIAS = "";
    Vector vectaliases = new Vector();

    while (en.hasMoreElements())
        vectaliases.add(en.nextElement());
    String[] aliases = (String []) (vectaliases.toArray(new String[0]));
    for (int i = 0; i < aliases.length; i++)
        if (ks.isKeyEntry(aliases[i]))
            {
                ALIAS = aliases[i];
                break;
            }
    privatekey = (PrivateKey)ks.getKey(ALIAS, password);
    cert = (X509Certificate)ks.getCertificate(ALIAS);
    publickey = ks.getCertificate(ALIAS).getPublicKey();
} catch (Exception e) {
    e.printStackTrace();
    return ;
}

```

### 3.2 - Les certificats X.509 (.cer)

Les certificats X.509 contiennent les informations nécessaires pour identifier un utilisateur ou une entité (un serveur web par exemple). Ce fichier n'est absolument pas chiffré et peut être diffusé librement. Ainsi, un utilisateur pourra vérifier la chaîne de certification et donc la validité du certificat. Il pourra alors décider d'utiliser la clé publique du certificat pour déchiffrer des messages.

Voici comment charger un certificat X.509 :

```

import java.io.*;
import java.security.cert.X509Certificate;
import java.security.cert.CertificateFactory;

try {
    // Chargement du certificat à partir du fichier
    InputStream inStream = new FileInputStream("personnal_nyal.cer");
    CertificateFactory cf = CertificateFactory.getInstance("X.509");
    X509Certificate cert = (X509Certificate)cf.generateCertificate(inStream);
    inStream.close();

    // Affiche le contenu du certificat
    System.out.println(cert.toString());
} catch (Exception e) {
    e.printStackTrace();
    return ;
}

```

L'exemple ne fait qu'afficher le contenu du certificat. Mais, il se peut que vous vouliez récupérer certaines parties comme les dates de validité. Voici un exemple :

```

import java.security.cert.X509Certificate;

// La variable cert est de type X509Certificate

String[] infos_emetteur = cert.getIssuerDN().getName().split("(=|,)", -1);
String[] infos_titulaire = cert.getSubjectDN().getName().split("(=|,)", -1);

System.out.println("CommonName : " + infos_titulaire[3]);
System.out.println("EmailAdresse : " + infos_titulaire[1] + "\n");

for (int i = 0; i < infos_emetteur.length; i += 2)

```

```
{
    if (infos_emetteur[i].equals("C"))
        System.out.println("CountryName : " + infos_emetteur[i + 1]);
    if (infos_emetteur[i].equals("O"))
        System.out.println("OrganizationName : " + infos_emetteur[i + 1]);
    if (infos_emetteur[i].equals("CN"))
        System.out.println("CommonName : " + infos_emetteur[i + 1]);
}

System.out.println("");
System.out.println("du : " + cert.getNotBefore());
System.out.println("au : " + cert.getNotAfter());
```



## 4 - Signature d'un document

### 4.1 - Signer au format CMS/PKCS#7

Le format CMS/PKCS#7 (.p7) permet de rassembler:

- Le certificat de l'individu qui a signé le document
- La chaîne des certificats (facultatif)
- Le document qui a été signé (facultatif)

Vous vous demandez peut être pourquoi créer une signature. Une signature permet de vérifier l'identité du signataire et l'intégrité du fichier (c'est à dire si le fichier n'a pas été modifié). Voici les étapes nécessaires pour la création d'une signature :

- Le signataire va hacher le document qu'il veut signer à l'aide d'un algorithme comme SHA-1, MD5,... Il en résulte une empreinte du document qui est considérée comme unique. Bien entendu, il peut y avoir deux fichiers différents pour une même empreinte. Mais la chance de trouver la même empreinte dans un temps raisonnable est infime. Considérez donc que cette empreinte est quasi-unique et non réversible. C'est à dire qu'on ne peut retrouver le fichier à partir de l'empreinte. Si c'était le cas, ce serait le meilleur algorithme de compression. ;)
- Le signataire va chiffrer cette empreinte avec une clé privée et un algorithme à clé asymétrique (RSA, DSA).
- La clé publique associée à la clé privée est mise à disposition. Cela signifie que le certificat public est associé directement avec la signature (ce qui est le cas avec le format CMS) où ils sont séparés.

La librairie Bouncy Castle va permettre d'effectuer ces étapes de manière très simple en utilisant l'objet **CMSSignedDataGenerator** :

```
import java.io.*;
import java.util.*;
import java.security.*;
import java.security.cert.X509Certificate;

import org.bouncycastle.cms.CMSSignedData;
import org.bouncycastle.cms.CMSProcessable;
import org.bouncycastle.cms.CMSProcessableByteArray;
import org.bouncycastle.cms.CMSSignedDataGenerator;

import java.security.cert.CertStore;
import java.security.cert.CollectionCertStoreParameters;

try {
    // Chargement du fichier qui va être signé

    File file_to_sign = new File("fichier_a_signer.txt");
    byte[] buffer = new byte[(int)file_to_sign.length()];
    DataInputStream in = new DataInputStream(new FileInputStream(file_to_sign));
    in.readFully(buffer);
    in.close();

    // Chargement des certificats qui seront stockés dans le fichier .p7
    // Ici, seulement le certificat personnel_nyal.cer sera associé.
    // Par contre, la chaîne des certificats non.

    ArrayList certList = new ArrayList();
    certList.add(cert);
    CertStore certs = CertStore.getInstance("Collection",
                                         new CollectionCertStoreParameters(certList), "BC");

    CMSSignedDataGenerator signGen = new CMSSignedDataGenerator();

    // privatekey correspond à notre clé privée récupérée du fichier PKCS#12
    // cert correspond au certificat publique personnel_nyal.cer
```

```

// Le dernier argument est l'algorithme de hachage qui sera utilisé
signGen.addSigner(privatekey, cert, CMSSignedDataGenerator.DIGEST_SHA1);
signGen.addCertificatesAndCRLs(certs);
CMSProcessable content = new CMSProcessableByteArray(buffer);

// Generation du fichier CMS/PKCS#7
// L'argument deux permet de signifier si le document doit être attaché avec la signature
// Valeur true: le fichier est attaché (c'est le cas ici)
// Valeur false: le fichier est détaché

CMSSignedData signedData = signGen.generate(content, true, "BC");
byte[] signeddata = signedData.getEncoded();

// Ecriture du buffer dans un fichier.

FileOutputStream envfos = new FileOutputStream(file_to_sign.getName() + ".pk7");
envfos.write(signeddata);
envfos.close();
} catch (Exception e) {
    e.printStackTrace();
    return ;
}

```

## 4.2 - Vérifier la signature

Voici la méthode utilisée pour vérifier la signature :

- Hachage du document qui a été signé. Utilisation du bon algorithme de hachage (le même que le signataire)
- Récupération de la signature et du certificat du signataire
- Utilisation de la clé publique pour déchiffrer la signature
- Vérification des deux empreintes. Si elles correspondent alors le fichier n'a pas été modifié et a été signé par l'utilisateur du certificat.

Voici un exemple pour vérifier la signature d'un document qui est attaché avec la signature (fichier au format CMS/PKCS#7) :

```

import java.io.*;
import java.util.*;

import java.security.cert.CertStore;
import java.security.cert.X509Certificate;

import org.bouncycastle.cms.CMSSignedData;
import org.bouncycastle.cms.CMSProcessable;
import org.bouncycastle.cms.SignerInformation;

try {
    // Chargement du fichier signé
    File f = new File("fichier_signer.txt.pk7");
    byte[] buffer = new byte[(int)f.length()];
    DataInputStream in = new DataInputStream(new FileInputStream(f));
    in.readFully(buffer);
    in.close();

    CMSSignedData signature = new CMSSignedData(buffer);
    SignerInformation signer = (SignerInformation)signature
        .getSignerInfos().getSigners().iterator().next();
    CertStore cs = signature
        .getCertificatesAndCRLs("Collection", "BC");
    Iterator iter = cs.getCertificates(signer.getSID()).iterator();
    X509Certificate certificate = (X509Certificate) iter.next();
    CMSProcessable sc = signature.getSignedContent();
    byte[] data = (byte[]) sc.getContent();

    // Verifie la signature
    System.out.println(signer.verify(certificate, "BC"));

    FileOutputStream envfos = new FileOutputStream("document_non_signer.txt");
    envfos.write(data);
}

```

```
        envfos.close();
    } catch (Exception e) {
        e.printStackTrace();
        return ;
    }
```

Cet exemple fonctionne pour les fichiers au format PKCS#7 avec la signature et le document signé. Il vérifie seulement la signature et non la validité du certificat (le certificat peut être révoqué, date de validité dépassée,...).

## 5 - Chiffrement d'un document

### 5.1 - Chiffrement par clé symétrique

Pour cette technique, l'émetteur et le destinataire du message dispose de la même clé secrète. L'émetteur chiffre le message avec la clé et le destinataire déchiffre le message avec la même clé.

Pour réaliser ce chiffrement, vous avez le choix avec un grand nombre d'algorithmes comme AES, Triple DES, Blowfish,...

 **Attention, selon le mode de chiffrement (CBC, CFB, OFB), vous devrez fournir une IV (initial value) qui devra être la même lors des opérations de chiffrement/déchiffrement.**

```
try {
    // Fichier à chiffrer
    File f = new File("fichier_a_chiffrer");
    byte[] buffer = new byte[(int)f.length()];
    DataInputStream in = new DataInputStream(new FileInputStream(f));
    in.readFully(buffer);
    in.close();

    // Choix de l'iv
    byte[] iv = { (byte) 0xc9, (byte) 0x36, (byte) 0x78, (byte) 0x99,
                 (byte) 0x52, (byte) 0x3e, (byte) 0xea, (byte) 0xf2 };


    IvParameterSpec salt = new IvParameterSpec(iv);
    // Clé secrète choisie
    byte[] raw = "ma cle secreta".getBytes();
    SecretKey sKeySpec = new SecretKeySpec(raw, "Blowfish");

    // Chiffrement du fichier
    Cipher c = Cipher.getInstance("Blowfish/CBC/PKCS5Padding", "BC");
    c.init(Cipher.ENCRYPT_MODE, sKeySpec, salt);
    byte[] buf_crypt = c.doFinal(buffer);

    FileOutputStream envfos = new FileOutputStream("fichier_chiffre");
    envfos.write(buf_crypt);
    envfos.close();

    // Déchiffrement du fichier
    c = Cipher.getInstance("Blowfish/CBC/PKCS5Padding", "BC");
    c.init(Cipher.DECRYPT_MODE, sKeySpec, salt);
    byte[] buf_decrypt = c.doFinal(buf_crypt);

    envfos = new FileOutputStream("fichier_dechiffre");
    envfos.write(buf_decrypt);
    envfos.close();
} catch (Exception e) {
    e.printStackTrace();
}
```

 La bibliothèque standard `sunJCE` fournit comme algorithmes symétriques: `DES`, `TripleDES`, `Blowfish`, `PBEWithMD5AndDES`, et `PBEWithMD5AndTripleDES`.

Tandis que `Bouncy Castle` gère un plus grand nombre d'algorithmes avec des clés de tailles différentes (`Twofish`, `Serpent`, `Skipjack`, `AES`,...).

### 5.2 - Chiffrement par clé asymétrique (format CMS/PKCS#7)

Différentes méthodes peuvent être utilisées pour le chiffrement par clé asymétrique. Nous allons en voir deux.

La première méthode consiste à chiffrer le fichier complètement avec la clé publique du certificat du destinataire. Ainsi le destinataire pourra déchiffrer le message avec sa clé privée. Cette méthode n'est pas conseillée car les algorithmes de chiffrement par clé asymétrique sont très lents (il vaut mieux préférer un chiffrement par clé symétrique comme dans la seconde méthode).

Voici donc un exemple pour chiffrer/déchiffrer entièrement un document avec l'algorithme RSA:

```
import java.io.*;
import java.util.*;
import java.security.*;

import javax.crypto.*;
import java.security.cert.X509Certificate;

try {
    // Chargement du fichier à chiffrer
    File f = new File("fichier_a_chiffrer");
    byte[] buffer = new byte[(int)f.length()];
    DataInputStream in = new DataInputStream(new FileInputStream(f));
    in.readFully(buffer);
    in.close();

    // Chiffrement du document
    // Seul le mode ECB est possible avec RSA
    Cipher cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding", "BC");
    // publickey est la cle publique du destinataire
    cipher.init(Cipher.ENCRYPT_MODE, publickey,
        new SecureRandom("nyal rand".getBytes()));
    int blockSize = cipher.getBlockSize();
    int outputSize = cipher.getOutputSize(buffer.length);
    int leavedSize = buffer.length % blockSize;
    int blocksSize = leavedSize != 0 ?
        buffer.length / blockSize + 1 : buffer.length / blockSize;
    byte[] raw = new byte[outputSize * blocksSize];
    int i = 0;
    while (buffer.length - i * blockSize > 0)
    {
        if (buffer.length - i * blockSize > blockSize)
            cipher.doFinal(buffer, i * blockSize, blockSize,
                raw, i * outputSize);
        else
            cipher.doFinal(buffer, i * blockSize,
                buffer.length - i * blockSize,
                raw, i * outputSize);
        i++;
    }

    // Ecriture du fichier chiffré sur le disque dur
    FileOutputStream envfos = new FileOutputStream(f.getName() + ".pl");
    envfos.write(raw);
    envfos.close();

    // Déchiffrement du fichier
    cipher = Cipher.getInstance("RSA/ECB/PKCS1Padding", "BC");
    //
    // La variable privatekey correspond à la clé privée associée à la clé
    // publique précédente. (Voir la section sur le fichier PKCS#12 pour
    // récupérer la clé privée)
    //
    cipher.init(cipher.DECRYPT_MODE, privatekey);
    blockSize = cipher.getBlockSize();
    ByteArrayOutputStream bout = new ByteArrayOutputStream(64);
    int j = 0;

    while (raw.length - j * blockSize > 0)
    {
        bout.write(cipher.doFinal(raw, j * blockSize, blockSize));
        j++;
    }

    envfos = new FileOutputStream("fichier_dechiffre");
    envfos.write(bout.toByteArray());
    envfos.close();
} catch (Exception e) {
```

```

    e.printStackTrace();
}

```

La seconde méthode est bien plus performante et doit être préférée. La procédure est la suivante:

- Création d'une clé qui permettra de chiffrer le fichier entier avec un algorithme à clé symétrique (DES, AES, blowfish,...)
- Chiffrement du fichier avec la clé symétrique créée
- Chiffrement de la clé symétrique avec la clé publique du destinataire

Enfin, un fichier au format CMS/PKCS#7 est créé.

Une fois le document chiffré reçu, le destinataire devra déchiffrer la clé symétrique avec sa clé privée et ensuite déchiffrer le document avec cette clé symétrique. Bouncy Castle permet ce chiffrement à l'aide de l'objet **CMSEnvelopedDataGenerator**:

```

import java.io.*;
import java.util.*;
import java.security.*;
import java.security.cert.X509Certificate;

import org.bouncycastle.cms.CMSEnvelopedData;
import org.bouncycastle.cms.CMSProcessableByteArray;
import org.bouncycastle.cms.CMSEnvelopedDataGenerator;
import org.bouncycastle.cms.KeyTransRecipientInformation;

try {
    // Chargement du fichier à chiffrer
    File f = new File("fichier_a_chiffrer");
    byte[] buffer = new byte[(int)f.length()];
    DataInputStream in = new DataInputStream(new FileInputStream(f));
    in.readFully(buffer);
    in.close();

    // Chiffrement du document

    CMSEnvelopedDataGenerator gen = new CMSEnvelopedDataGenerator();
    // La variable cert correspond au certificat du destinataire
    // La clé publique de ce certificat servira à chiffrer la clé symétrique
    gen.addKeyTransRecipient((java.security.cert.X509Certificate)cert);

    // Choix de l'algorithme à clé symétrique pour chiffrer le document.
    // AES est un standard. Vous pouvez donc l'utiliser sans crainte.
    // Il faut savoir qu'en france la taille maximum autorisée est de 128
    // bits pour les clés symétriques (ou clés secrètes)
    String algorithm = CMSEnvelopedDataGenerator.AES128_CBC;
    CMSEnvelopedData envData = gen.generate(
        new CMSProcessableByteArray(buffer),
        algorithm, "BC");

    byte[] pkcs7envelopedData = envData.getEncoded();

    // Ecriture du document chiffré
    FileOutputStream envfos = new FileOutputStream(f.getName() + ".pk7");
    envfos.write(pkcs7envelopedData);
    envfos.close();

    // Déchiffrement du fichier

    CMSEnvelopedData ced = new CMSEnvelopedData(pkcs7envelopedData);
    Collection recip = ced.getRecipientInfos().getRecipients();

    KeyTransRecipientInformation rinfo = (KeyTransRecipientInformation)
        recip.iterator().next();
    // privatekey est la clé privée permettant de déchiffrer la clé secrète
    // (symétrique)
    byte[] contents = rinfo.getContent(privatekey, "BC");

    envfos = new FileOutputStream("fichier_non_chiffrer");

```

```
        envfos.write(contents);
        envfos.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
```

## 6 - L'empreinte d'un document

L'empreinte d'un document est appelé **empreinte numérique**. Cette empreinte est créée après être passé par un algorithme de hachage (SHA-1, MD5, ...) à sens unique qui a la particularité de générer la même empreinte pour la même suite de bits. (deux suites de bits peut donner la même empreinte mais considérez que l'empreinte est unique)

Vous comprenez donc que les empreintes numériques vont permettre de vérifier l'intégrité des fichiers. Prenons un exemple courant. Sur un serveur se trouve des fichiers sensibles avec leur empreinte. Une fois un fichier téléchargé et son empreinte générée, comparez les deux empreintes: celle se situant sur le serveur et celle générée. Si elles sont identiques alors le fichier n'a pas été modifié pendant le transfert.

Il est conseillé de vérifier l'intégrité des fichiers. Cependant, cette vérification est rare car elle est fastidieuse comme vous le constaterez.

Voici un exemple pour la création d'une empreinte:

```
import java.io.*;
import java.security.*;
import org.bouncycastle.jce.provider.BouncyCastleProvider;

try {
    // Chargement du document
    File f = new File("fichier");
    byte[] buffer = new byte[(int)f.length()];
    DataInputStream in = new DataInputStream(new FileInputStream(f));
    in.readFully(buffer);
    in.close();

    // Utilisation de l'algorithme SHA-1
    MessageDigest md = MessageDigest.getInstance("SHA-1", "BC");
    byte[] empreinte = Hex.encode(md.digest(buffer));
    // Affichage de l'empreinte en hexa
    System.out.println(new String(empreinte));
} catch (Exception e) {
    e.printStackTrace();
}
```



## 7 - Conclusion

Très simple d'utilisation, la bibliothèque **Bouncy Castle** vous fera gagner un temps précieux dans le développement de vos solutions sécurisés.

Ce tutoriel sera mis à jour constamment pour suivre les changements de la bibliothèque et ajouter des points non traités (validité de la chaîne de certification, les certificats révoqués,...).

Je vous remercie d'avoir lu ce tutoriel et de me contacter en cas d'erreurs. (nyal at voila dot fr)

## Remerciements

Merci à Vedaer, Cyberzoide et Anomaly pour la correction des erreurs qui s'étaient glissées dans le tutoriel.