JAVA Security (J2SE 5)

Christophe Borelly

IUT Béziers, Dépt R&T

http://www.borelly.net/

Christophe@Borelly.net

Généralités (J2SE 5)

- Il existe plusieurs API sur la securité :
- Java Cryptography Extension (JCE)
- Java Secure Socket Extension (JSSE)
- Java Authentication and Authorization Service (JAAS)
- Java Generic Security Services (Java GSS-API)
- Simple Authentication and Security Layer (Java SASL)
- ...

JCE : Java Cryptography Extension

- La version 1.1 de JAVA proposait des fonctionnalités cryptographiques (signatures et message digest) avec JCA (Java Cryptography Architecture). Le support des certificats X.509v3 a été ensuite ajouté dans le Java 2 SDK.
- Dans les versions 1.2.x and 1.3.x, le paquetage JCE (Java Cryptography Extension) était optionnel. Il est, depuis la version 1.4, intégré à la distribution standard.

Améliorations depuis Java 1.4

- Support des suites de chiffrement de Kerberos dans JSSE.
- Le TrustManager par défaut de JSSE est du type CertPath.
- Ajout de SSLEngine pour le support des sockets SSL/TLS non bloquantes.
- Support de providers SSL/TLS externes, usine à sockets SSL/TLS pour RMI.
- Support de ECC, du chiffrement RSA et du provider PKCS#11.
- Support de Time-Stamp Protocol (TSP - RFC 3161), On-Line Certificate Status Protocol (OCSP - RFC 2560).
- Amélioration de l'implémentation de PKCS#12, de PKIX (RFC 3280) et de Kerberos (TGT renewal, encryption DES3).
- Support de SASL.

JCE : Java Cryptography Extension

- Cette API permet de réaliser :
  - Chiffrement symétrique par bloc (DES, etc…)
  - Chiffrement symétrique par flot (RC4, etc…)
  - Chiffrement asymétrique (RSA, etc…)
  - Chiffrement PBE (Password-Based Encryption)
  - Échange de clés (Key Agreement)
  - Message Authentication Codes (MAC)
- Un **provider** est une librairie proposant l'implémentation de certaines fonctionnalités.

Affichage des providers

```
import java.security.Security;
import java.security.Provider;
...
Provider[] providers=Security.getProviders();
for (Provider p : providers) {
  System.out.println(p);
  System.out.println("Name    :"+p.getName());
  System.out.println("Info    :"+p.getInfo());
  System.out.println("Version :"+p.getVersion());
  System.out.println("-----------------------------");
}
```

Exemple de providers :

```
SUN version 1.42
Name    : SUN
Info    : SUN (DSA key/parameter generation; DSA signing;
SHA-1, MD5 digests; SecureRandom; X.509 certificates;
JKS keystore; PKIX CertPathValidator; PKIX CertPathBuilder;
LDAP, Collection CertStores)
-------------------------------------
SunJSSE version 1.42
Name    : SunJSSE
Info    : Sun JSSE provider(implements RSA Signatures, PKCS12,
SunX509 key/trust factories, SSLv3, TLSv1)
-------------------------------------
SunRsaSign version 1.42
Name    : SunRsaSign
Info    : SUN's provider for RSA signatures
```

Ajouter un provider (1)

- Ajout dans le code :

```
Security.addProvider(
 new org.bouncycastle.jce.provider.BouncyCastleProvider()
 );
```

- Ajouter le paquetage **bcprov-jdkxx-yyy.jar** dans un répertoire du CLASSPATH ou le copier dans <JAVA_HOME>/jre/lib/ext pour la compilation.

Ajouter un provider (2)

- Ajout dans le fichier de configuration :
  - <JAVA_HOME>/jre/lib/security/java.security
  - <JRE_HOME>/lib/security/java.security

```
...
security.provider.1=sun.security.provider.Sun
security.provider.2=com.sun.net.ssl.internal.ssl.Provider
security.provider.3=com.sun.rsajca.Provider
security.provider.4=com.sun.crypto.provider.SunJCE
security.provider.5=sun.security.jgss.SunProvider
security.provider.6=
    org.bouncycastle.jce.provider.BouncyCastleProvider
...
```

MessageDigest

```
import java.security.MessageDigest;
...
String msg="Hello";
byte[] data=msg.getBytes();
MessageDigest md=MessageDigest.getInstance("MD5");
// MessageDigest.getInstance("MD5","SUN");
byte[] digest=md.digest(data);

MD5 : 16 (octets)
8B:1A:99:53:C4:61:12:96:A8:27:AB:F8:C4:78:04:D7
root@pccb ~/$ echo -n Hello|md5sum
8b1a9953c4611296a827abf8c47804d7 *-
```

Algorithmes de MessageDigest

- /docs/guide/security/CryptoSpec.html#AppA
- Le nom de l'algorithme n'est pas sensible à la casse (MD5 identique à md5).
- MD2 : Message Digest algorithm RFC 1319
- MD5 : RFC 1321
- SHA-1 : Secure Hash Algorithm NIST FIPS 180-1
- SHA-256, SHA-384 et SHA-512 : FIPS 180-2
- ...

National Institute of Standards and Technology (NIST)

Federal Information Processing Standard (FIPS)

Les clés

- Les clés sont obtenues avec des générateurs (**KeyGenerator**, **KeyPairGenerator**), des paramètres (**KeyFactory**) ou des fichiers (**KeyStore**).
- Quelques classes implémentant java.security.Key :
  - DHPrivateKey, DHPublicKey
  - DSAPrivateKey, DSAPublicKey
  - PBEKey
  - PrivateKey, PublicKey
  - RSAMultiPrimePrivateCrtKey, RSAPrivateCrtKey
  - RSAPrivateKey, RSAPublicKey
  - SecretKey

Interface java.security.Key

- `String` **`getAlgorithm`**`()`
  - RSA, DSA, DES, etc...
- `byte[]` **`getEncoded`**`()`
  - Représentation ASN.1 de la clé
- `String` **`getFormat`**`()`
  - Format d'encodage de la clé (e.g. X.509)

Clé secrète DES (KeyGenerator)

- Les algorithmes de clés secrètes possibles sont : AES, ARCFOUR/RC4, Blowfish, DES, DESede, HmacMD5, HmacSHA1, HmacSHA256, HmacSHA384, HmacSHA512, RC2, ...

```
import java.security.KeyGenerator;
import java.security.SecureRandom;
import javax.crypto.SecretKey;
...
KeyGenerator keyGen=KeyGenerator.getInstance("DES");
SecureRandom random=new SecureRandom();
keyGen.initialize(1024,random);
SecretKey secKey=keyGen.generateKey();
```

Clé secrète DES (SecretKeyFactory)

```
import javax.crypto.spec.DESKeySpec;
import javax.crypto.SecretKeyFactory;
import javax.crypto.SecretKey;
...
SecretKeyFactory keyFactory=
  SecretKeyFactory.getInstance("DES");
byte[] desKeyData={
   (byte)0x01,(byte)0x02,(byte)0x03,(byte)0x04,
   (byte)0x05,(byte)0x06,(byte)0x07,(byte)0x08};
DESKeySpec desKeySpec=new DESKeySpec(desKeyData);
SecretKey secretKey=
  keyFactory.generateSecret(desKeySpec);
```

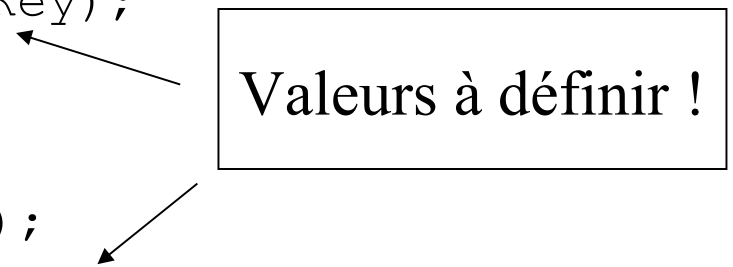
Clé secrète DES (SecretKeySpec)

```
import javax.crypto.spec.SecretKeySpec;
...
byte[] desKeyData= {
    (byte)0x01,(byte)0x02,(byte)0x03,(byte)0x04,
    (byte)0x05,(byte)0x06,(byte)0x07,(byte)0x08};
SecretKeySpec secretKey=new
  SecretKeySpec(desKeyData,"DES");
```

Cipher

```
import javax.crypto.Cipher;
...
String msg="Hello";
byte[] data=msg.getBytes();
Cipher c=Cipher.getInstance("DES");
c.init(Cipher.ENCRYPT_MODE,secretKey);
byte[] enc=c.dofinal(data);

Cipher d=Cipher.getInstance("DES");
d.init(Cipher.DECRYPT_MODE,secretKey);
byte[] dec=d.dofinal(enc);
```



Algorithmes de Cipher

- /docs/guide/security/CryptoSpec.html#AppA
- Deux écritures : *algorithm* ou *algorithm/mode/padding*
- **AES** : Advanced Encryption Standard
- **DES** : Digital Encryption Standard - FIPS PUB 46-2.
- **DESede** : Triple DES
- **ECIES** : Elliptic Curve Integrated Encryption Scheme
- **RC2**, **RC4** et **RC5** :
- **RSA** : Rivest Shamir Adleman PKCS #1.
- **PBEWith**<digest>And<encryption>, **PBEWith**<prf>And<encryption> PKCS #5: Password-Based Encryption Standard. (prf : pseudo-random function)
  - PBEWithMD5AndDES, PBEWithHmacSHA1AndDESed
- ...

Modes de Cipher

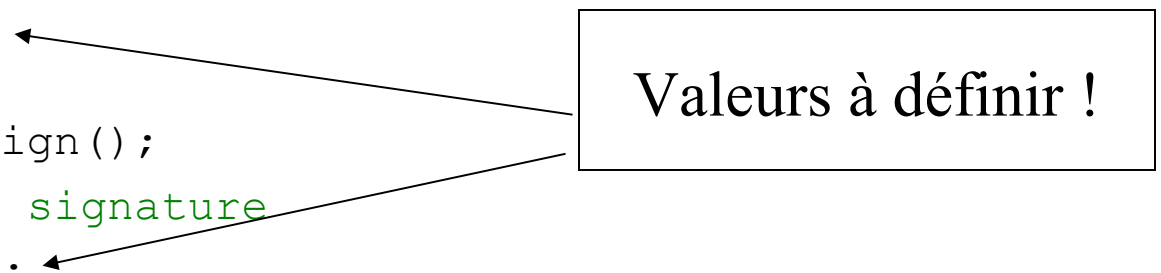
- **NONE** : Aucun mode.
- **CBC** : Cipher Block Chaining - FIPS PUB 81.
- **CFB** : Cipher Feedback - FIPS PUB 81.
- **ECB** : Electronic CodeBook - FIPS PUB 81.
- **OFB** : Output Feedback - FIPS PUB 81.
- **PCBC** : Propagating Cipher Block Chaining – KerberosV4
- ...

Paddings de Cipher

- **NoPadding** : Aucun padding.
- **ISO10126Padding** : padding pour block ciphers - W3C XML Encryption Syntax and Processing.
- **OAEPWith**<digest>And<mgf>Padding : Optimal Asymmetric Encryption Padding - PKCS #1. (mgf : mask generation function)
  - OAEPWithMD5AndMGF1Padding.
- **PKCS5Padding** : PKCS #5 : Password-Based Encryption Standard.
- **SSL3Padding** : SSL Protocol Version 3.0.
- ...

Signatures

```
import java.security.Signature;
...
String msg="Hello";
byte[] data=msg.getBytes();
Signature s=Signature.getInstance("MD5withRSA");
// Pour signer des données
s.initSign(privKey);
s.update(data);
byte[] signature=s.sign();
// Pour vérifier une signature
s.initVerify(pubKey);
s.update(data);
if (s.verify(signature)) ...
```



Algorithmes de Signature

- /docs/guide/security/CryptoSpec.html#AppA
- **ECDSA** : Elliptic Curve Digital Signature Algorithm
- **MD2withRSA**, MD5withRSA : PKCS #1
- **SHA1withRSA** : PKCS #1
- **NONEwithDSA** : FIPS PUB 186
- **SHA1withDSA** : FIPS PUB 186
- ...

Message Authentication Code (MAC)

```
import java.security.Mac;
...
String msg="Hello";
byte[] data=msg.getBytes();
Mac mac=Mac.getInstance("HmacMD5");
mac.init(secKey);
byte[] result=mac.doFinal(data);
// Autre façon de calculer
mac.reset();
mac.update(data);
byte[] result=mac.doFinal();
```

Valeur à définir

Algorithmes de Mac

- **HmacMD5**, **HmacSHA1**, **HmacSHA256**, **HmacSHA384**, **HmacSHA512** : Hashing for Message Authentication - RFC2104
- **PBEWith**<mac> : PKCS#5
  - PBEWithHmacSHA1
  - ...

Échange de clés DH (1)

```
AlgorithmParameterGenerator paramGen=
   AlgorithmParameterGenerator.getInstance("DH");
paramGen.init(512);
AlgorithmParameters params=
   paramGen.generateParameters();
DHParameterSpec dhParamSpec=(DHParameterSpec)
  params.getParameterSpec(DHParameterSpec.class);
KeyPairGenerator aliceKpairGen=
  KeyPairGenerator.getInstance("DH");
aliceKpairGen.initialize(dhParamSpec);
KeyPair aliceKpair=aliceKpairGen.generateKeyPair();
```

Échange de clés DH (2)

```
KeyAgreement aliceKeyAgree=
   KeyAgreement.getInstance("DH");
aliceKeyAgree.init(aliceKpair.getPrivate());
byte[] alicePubKeyEnc=
   aliceKpair.getPublic().getEncoded();
// Alice envoie sa clé publique à Bob.
...
// Alice reçoit la clé publique de Bob.
byte[] bobPubKeyEnc=...
```

Échange de clés DH (3)

```
KeyFactory aliceKeyFac=KeyFactory.getInstance("DH");
X509EncodedKeySpec x509KeySpec=
      new X509EncodedKeySpec(bobPubKeyEnc);
PublicKey bobPubKey=
      aliceKeyFac.generatePublic(x509KeySpec);
aliceKeyAgree.doPhase(bobPubKey,true);

byte[] sharedSecret=aliceKeyAgree.generateSecret();
SecretKey desKey=aliceKeyAgree.generateSecret("DES");
```

Les paires de clés

- Les algorithmes de clés possibles sont : DSA, RSA

```
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.KeyPair;
import java.security.PrivateKey;
import java.security.PublicKey;
...
KeyPairGenerator keyGen=KeyPairGenerator.getInstance("RSA");
SecureRandom random=new SecureRandom();
keyGen.initialize(1024,random);
KeyPair pair=keyGen.generateKeyPair();
PrivateKey privKey=pair.getPrivate();
PublicKey pubKey=pair.getPublic();
```

KeyFactory

- Fabrication à partir des paramètres de la clé :

```
import java.security.KeyFactory;
import java.math.BigInteger;
import java.security.spec.DSAPrivateKeySpec;
import java.security.PrivateKey;
...
KeyFactory keyFactory=KeyFactory.getInstance("DSA");
BigInteger x=new BigInteger("123...",16); // Private key
BigInteger p=new BigInteger("345...",16); // Prime
BigInteger q=new BigInteger("678...",16); // Sub-prime
BigInteger g=new BigInteger("9AB...",16); // Base
DSAPrivateKeySpec dsaPrivKeySpec=new DSAPrivateKeySpec(x,p,q,g);
PrivateKey privKey=keyFactory.generatePrivate(dsaPrivKeySpec);
```

CertificateFactory

- Récupération d'une clé publique à partir d'un certificat :

```
import java.io.FileInputStream;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.security.PublicKey;
...
FileInputStream fis=new FileInputStream("srv.crt");
CertificateFactory cf=
   CertificateFactory.getInstance("X.509");
X509Certificate cert=
   (X509Certificate)cf.generateCertificate(fis);
PublicKey pubKey=cert.getPublicKey();
fis.close();
```

KeyStore

```
import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.cert.Certificate;
...
String storePass="passwd";
String keyPass="cbpass";
FileInputStream fis=new FileInputStream("cbstore");
KeyStore ks=KeyStore.getInstance("JKS");
ks.load(fis,storePass.toCharArray());
PrivateKey privKey=
    (PrivateKey)ks.getKey("cb",keyPass.toCharArray());
Certificate cert=ks.getCertificate("cb");
PublicKey pubKey=cert.getPublicKey();
fis.close();
```

Keytool (1)

- Génération d'une clé et d'un certificat :

```
keytool -genkey -alias cb
-dname "cn=cb,ou=IUT,o=RT,l=Beziers,st=Herault,c=FR"
-validity 180 -keyalg RSA -keysize 1024
-keypass cbpass -keystore cbstore -storepass passwd
```

- Export au format DER (binaire) :

```
keytool -export -alias cb -file cb.cer
-keystore cbstore -storepass passwd
```

- Export au format PEM (ASCII) :

```
keytool -export -alias cb -file cb.crt -rfc
-keystore cbstore -storepass passwd
```

Keytool (2)

- Affichage du contenu :

```
keytool -list -keystore cbstore -storepass passwd -v
```

- Effacement d'une clé :

```
keytool -delete -alias cb
-keystore cbstore -storepass passwd
```

- Importation d'un certificat de confiance :

```
keytool -import -alias ca -file ca.crt -trustcacerts
-keystore cbstore -storepass passwd
```

javax.crypto.SealedObject

```
SealedObject(Serializable object,Cipher c)
```

- Valeur de l'algorithme qui a été utilisé :

```
String getAlgorithm()
```

- Les paramètres de décryptage (e.g. IV) sont sauvegardés dans l'objet scellé. On peut donc ne spécifier que la clé de décryptage pour récupérer l'objet :

```
Object getObject(Cipher c)
Object getObject(Key key)
Object getObject(Key key,String provider)
```

SealedObject (1)

```
Object secretObject=new ... // implements Serializable
SecretKey secKey=...
// Get the outputStream on a socket connection
ObjectOutputStream out=
  new ObjectOutputStream(socket.getOutputStream());
Cipher cipher=
  Cipher.getInstance("DES/ECB/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE,secKey);
SealedObject so=
  new SealedObject(secretObject,cipher);
out.writeObject(so);
```

SealedObject (2)

```
...
SecretKey secKey=...
// Get the inputStream on a socket connection
ObjectInputStream in=
   new ObjectInputStream(socket.getInputStream());
SealedObject so=(SealedObject)in.readObject();
Object secretObject=so.getObject(secKey);
```

JSSE : Java Secure Socket Extension

- Implémentation de Secure Sockets Layer (SSL) et Transport Layer Security (TLS - RFC 2246)
- Paquetages : javax.net, javax.net.ssl et javax.security.cert

Client SSL simple (HTTPS)

```
String host="www.borelly.net";
int port=443; // port HTTPS
SSLSocketFactory factory=
   (SSLSocketFactory)SSLSocketFactory.getDefault();
SSLSocket s=
   (SSLSocket)factory.createSocket(host,port);
s.startHandshake();
PrintStream ps=new PrintStream(s.getOutputStream());
...
```

Exécution du client HTTPS

- On peut afficher des informations de debug en précisant par exemple (la valeur help permet d'afficher les différentes options) :

```
-Djavax.net.debug=ssl,handshake
```

- Le serveur https envoi un certificat pour s'identifier. Il faut ajouter le certificat de CA au truststore (<JRE_HOME>/lib/security/cacerts) de la machine virtuelle pour que cela fonctionne :

```
keytool -import -trustcacerts -alias ca
-file ca.crt -storepass changeit
-keystore $JRE_HOME/lib/security/cacerts
```

- On peut aussi spécifier un truststore particulier :

```
-Djavax.net.ssl.trustStore=cbTrustStore
-Djavax.net.ssl.trustStorePassword=passwd
```

Fichier de configuration /lib/security/java.security

```
# Determines whether this properties file can be
# appended to or overridden on the command line
# via -Djava.security.properties
security.overridePropertiesFile=true
# Determines the default key and trust manager factory
# algorithms for the javax.net.ssl package.
ssl.KeyManagerFactory.algorithm=SunX509
ssl.TrustManagerFactory.algorithm=SunX509
# Determines the default
#ssl.SocketFactory.provider=
#ssl.ServerSocketFactory.provider=
```

Configuration de la connexion

- On peut configurer les propriétés de la connexion SSL avant de lancer la méthode startHandcheck() :

```
// guide/security/jsse/JSSERefGuide.html#AppA
String[] proto={"SSLv3","TLSv1"};
s.setEnabledProtocols(proto);
String[] cipherSuites={"SSL_RSA_WITH_NULL_MD5",
   "TLS_RSA_WITH_AES_128_CBC_SHA",
   "SSL_RSA_WITH_RC4_128_SHA"};
s.setEnabledCipherSuites(cipherSuites);
```

Configuration de la connexion (2)

```
String[] protoSuites=s.getEnabledProtocols();
for (String proto : protoSuites) {
  System.out.println("Protocol "+i+" = "+proto);
}
String[] cipherSuites=s.getEnabledCipherSuites();
for (String cipher : cipherSuites) {
  System.out.println("Cipher Suite "+j
                     +" = "+cipher);
}
```

SSLSession

- Par l'intermédiaire de l'objet SSLSession, on peut afficher les propriétés courantes de la connexion SSL :

```
SSLSession sess=s.getSession();
System.out.println("Protocole : "+sess.getProtocol());
System.out.println("Cipher    : "+sess.getCipherSuite());
Certificate[] peerCerts=sess.getPeerCertificates();
for (Certificate cert : peerCerts) {
   System.out.println("Certificat : "
                      +((X509Certificate)cert).getSubjectDN());
}
```

Authentification du client

- Quand le serveur demande l'authentification du client, celui-ci doit envoyer un certificat.
- Il peut être créé avec l'outil keytool :

```
keytool -genkey -alias id
-dname "cn=CB,ou=IUT,o=RT,l=Beziers,st=Herault,c=FR"
-validity 180 -keyalg RSA -keysize 1024 -keypass idpass
-keystore idstore -storepass idpass
```

- On peut ensuite spécifier le keystore avec :

```
-Djavax.net.ssl.keyStore=idstore
-Djavax.net.ssl.keyStorePassword=idpass
// Même mot de passe pour la clé et le store !!!
```

Configuration à l'exécution

- On peut modifier les propriétés avec :

```
System.setProperty("javax.net.ssl.keyStore","idstore");
System.setProperty("javax.net.ssl.keyStorePassword",
                   "idpass");
System.setProperty("javax.net.ssl.trustStore","castore");
System.setProperty("javax.net.ssl.trustStorePassword",
                   "capass");
```

- Dans le cas où un mot de passe différent est utilisé pour la clé et le keystore, il faut utiliser un contexte SSL.

SSLContext

```
import java.security.*;
...
String
   keyStorePass="idpass",keyPasswd="idpasswd",caStorePass="capasswd";
KeyStore ks=KeyStore.getInstance("JKS");
ks.load(new FileInputStream("idstore"),keyStorePass.toCharArray());
KeyManagerFactory kmf=KeyManagerFactory.getInstance("SunX509");
kmf.init(ks,keyPasswd.toCharArray());

KeyStore ts=KeyStore.getInstance("JKS");
ts.load(new FileInputStream("castore"),caStorePass.toCharArray());
TrustManagerFactory tmf=TrustManagerFactory.getInstance("SunX509");
tmf.init(ts);

SSLContext ctx=SSLContext.getInstance("SSLv3");
ctx.init(kmf.getKeyManagers(),tmf.getTrustManagers(),null);
SSLSocketFactory sslsf=ctx.getSocketFactory();
```

Serveur SSL

Sans contexte SSL :

```
SSLServerSocketFactory sslssf=
   (SSLServerSocketFactory)SSLServerSocketFactory.getDefault();
```

Avec contexte SSL :

```
...
SSLServerSocketFactory sslssf=ctx.getServerSocketFactory();

SSLServerSocket ss=
   (SSLServerSocket)factory.createServerSocket(port);
// Demande d'authentification du client
ss.setNeedClientAuth(true);
SSLSocket c=(SSLSocket)ss.accept();
...
```

Généralités JAAS

- API depuis v. 1.3, intégrée depuis v. 1.4
- Java Authentication and Authorization Service (JAAS)
  - Permet de déterminer qui exécute le programme JAVA (Authentification).
  - Permet de vérifier les actions et les permissions (Autorisation)

Authentification

- Pour que les applications JAVA soient indépendantes du système d'authentification, JAAS utilise le système PAM (Pluggable Authentication Module).
- L'authentification se fait par l'intermédiaire d'un objet javax.security.auth.login.LoginContext.
- La configuration se fait dans un fichier texte qui peut être passé en paramètre à l'exécution :

```
-Djava.security.auth.login.config=sample.conf
```

Fichier de configuration par défaut

- <JAVA_HOME>/jre/lib/security/java.security
- <JRE_HOME>/lib/security/java.security

```
# For javax.security.auth.login.Configuration provider
login.configuration.provider=
   com.sun.security.auth.login.ConfigFile
# Default login configuration file
login.config.url.1=file:${user.home}/.java-login.conf
```

Identité de l'utilisateur

- L'objet javax.security.auth.Subject retourné par un LoginModule représente les différentes identités de l'utilisateur (interface java.security.Principal).
- Par la suite, on peut exécuter une action par l'intermédiaire de :
  - `static Object doAs(Subject subject, PrivilegedAction action)`
  - `static Object doAsPrivileged(Subject subject, PrivilegedAction action,AccessControlContext acc)`

javax.security.auth .login.Configuration

- Chaque entrée est identifiée par un nom, une liste ordonnée de modules de login, un mode et des options facultatives.
- Il y a 4 modes :
  - Required : Le succès est obligatoire. On passe dans tout les cas au module suivant si il y en a un.
  - Requisite : Le succès est obligatoire. Si l'authentification fonctionne, on passe au module suivant, sinon on retourne à l'application.
  - Sufficient : Le succès est facultatif. Si l'authentification fonctionne, on retourne à l'application, sinon on passe au module suivant.
  - Optional : Le succès est facultatif. On passe dans tout les cas au module suivant si il y en a un.

Exemple de configuration

```
JaasSample {
  com.sun.security.auth.module.Krb5LoginModule required;
};
Sample {
  sample.module.SampleLoginModule required debug=true;
};
other {
  com.sun.security.auth.module.UnixLoginModule required;
  com.sun.security.auth.module.Krb5LoginModule optional
   useTicketCache="true"
   ticketCache="${user.home}${/}tickets"; };
```

Classes disponibles (J2SE 5)

- **com.sun.security.auth.module :**
  - Krb5LoginModule
  - KeyStoreLoginModule
  - UnixLoginModule
- **com.sun.security.auth :**
  - UnixPrincipal, UnixNumericUserPrincipal, UnixNumericGroupPrincipal
- **javax.security.auth.x500.X500Principal**
- **javax.security.auth.kerberos.KerberosPrincipal**

Exemple d'authentification

```
import javax.security.auth.login.LoginContext;
import com.sun.security.auth.callback.TextCallbackHandler;
...
LoginContext lc=null;
try {
   lc=new LoginContext("JaasSample",
                       new TextCallbackHandler());
   lc.login();
}
catch (Exception e) {
   System.err.println("LoginContext : "+e);
   System.exit(-1);
}
System.out.println("Subject : "+lc.getSubject());
```

Fichier de configuration Kerberos

```
[libdefaults]
default_realm = IUTBEZIERS.FR
[realms]
IUTBEZIERS.FR = { kdc = 10.0.0.1:88 }
```

- Si la valeur java.security.krb5.conf existe dans le fichier <JAVA_HOME>/jre/lib/security/java.security, elle spécifie le répertoire de recherche du fichier de configuration.
- Sinon, on recherche dans l'ordre dans les répertoires :
  - `<JAVA_HOME>/jre/lib/security`
  - `c:\winnt\krb5.ini` [Windows]
  - `/etc/krb5.conf` [Linux]
  - `/etc/krb5/krb5.conf` [Solaris]

Alternative au fichier krb5.conf

- Valeurs spécifiées dans java.security :

```
java.security.krb5.realm=...
java.security.krb5.kdc=...
```

- Options spécifiées à l'exécution :

```
-Djava.security.krb5.realm=IUTBEZIERS.FR
-Djava.security.krb5.kdc=10.0.0.1
```

Les outils kerberos JAVA

- kinit, klist et ktab
- kinit cb@IUTBEZIERS.FR

```
Password for cb@IUTBEZIERS.FR:******
New ticket is stored in cache file C:\Documents and
   Settings\Administrateur\krb5cc_Administrateur
```

- klist

```
Credentials cache: C:\Documents and
   Settings\Administrateur\krb5cc_Administrateur
Default principal: cb@IUTBEZIERS.FR, 1 entry found.
[1]  Service Principal:  krbtgt/IUTBEZIERS.FR@IUTBEZIERS.FR

   Valid starting:   Apr 13,  2005 16:28
   Expires:          Apr 14,  2005 00:28
```

Différentes configurations

- En utilisant auparavant kinit cb, l'application de demande plus le mot de passe !!!

```
com.sun.security.auth.module.Krb5LoginModule required
   useTicketCache=true;
```

- Avec l'identité spécifiée dans la configuration

```
com.sun.security.auth.module.Krb5LoginModule required
   useTicketCache=true principal="cb@IUTBEZIERS.FR";
```

- Avec un fichier de cache spécial :
  - kinit -c krb5.cb cb

```
com.sun.security.auth.module.Krb5LoginModule required
   useTicketCache=true ticketCache=krb5.cb
   principal="cb@IUTBEZIERS.FR" debug=true;
```

Utilisation de ktab

- ktab permet de gérer les identités kerberos dans un fichier (Attention à la sécurité !!!)
- Création du fichier : **ktab -K krb5.cb -a cb**
- Contenu du fichier : **ktab -l -k krb5.cb**
- La configuration peut s'écrire :

```
com.sun.security.auth.module.Krb5LoginModule
   required principal="cb@IUTBEZIERS.FR"
   useKeyTab=true keyTab=krb5.cb;
```

Identification d'un service

- Pour un service, il n'y a pas de mot de passe. Sur le KDC, on créé la clé avec :

```
kadmin.local:  addprinc -randkey host/pccb
```

- On peut l'exporter dans un fichier avec :

```
kadmin.local:  ktadd -k krb5.pccb host/pccb
```

- La configuration peut s'écrire :

```
com.sun.security.auth.module.Krb5LoginModule
   required principal="host/pccb@IUTBEZIERS.FR"
   storeKey=true
   useKeyTab=true keyTab=krb5.pccb;
```

Autorisation

- L'utilisateur doit être authentifié avant d'exécuter une action.
- Les autorisations sont fixées dans les fichiers policy.
- Une action doit implémenter java.security.PrivilegedAction avec la méthode public Object **run**()

Exemple d'action

```
import java.security.PrivilegedAction;
// Il faut définir la permission :
// permission java.util.PropertyPermission "java.home","read";
public class SampleAction implements
  PrivilegedAction {
   public Object run() {
    System.out.println("java.home : "+
            System.getProperty("java.home"));
    return null;
  }
}
```

Exemple d'exécution

```
import javax.security.auth.*;
import javax.security.auth.login.*;
...
LoginContext lc=new LoginContext("Sample");
lc.login();
Subject id=lc.getSubject();
PrivilegedAction action=new SampleAction();
Subject.doAsPrivileged(id,action,null);
```

Les fichiers policy

- Les fichiers policy permettent de fixer des permissions d'accès, d'exécution, etc...
  - Une seule entrée keystore.
  - Zéro ou plusieurs entrées grant.
- On peut utiliser plusieurs fichiers policy.
- Editeur graphique : policytool
- Classe abstraite java.security.Permission

Utilisation du fichier policy

- On peut spécifier un fichier de policy dans la ligne de commandes :

```
java -Djava.security.manager
     -Djava.security.policy=pURL SomeApp
appletviewer -J-Djava.security.policy=pURL SomeApplet.html
```

- L'option -Djava.security.manager installe le manager de sécurité par défaut. Elle est inutile si le programme installe lui même un manager de sécurité (SecurityManager security=new SecurityManager();).

Fichier de configuration /lib/security/java.security

```
policy.provider=sun.security.provider.PolicyFile
# whether or not we expand properties in the policy file
# if this is set to false, properties (${...})
# will not be expanded in policy files.
policy.expandProperties=true
# The default is to have a single system-wide policy file,
# and a policy file in the user's home directory.
policy.url.1=file:${java.home}/lib/security/java.policy
policy.url.2=file:${user.home}/.java.policy
# whether or not we allow an extra policy to be passed on
# the command line with -Djava.security.policy=somefile.
policy.allowSystemProperty=true
```

Format de l'entrée grant

```
grant [SignedBy "signer_names"]
 [, CodeBase "URL"]
 [, Principal [principal_class_name] "principal_name"]
 [, Principal [principal_class_name] "principal_name"] ...
{
    permission permission_class_name [ "target_name" ]
                [, "action"] [, SignedBy "signer_names"];
    permission ...
};
```

Exemple de fichier policy

```
keystore "http://foo.bar.com/blah/.keystore";


grant codeBase "http://java.sun.com/*", signedBy "Li" {
    permission java.io.FilePermission "/tmp/*","read";
    permission java.io.SocketPermission "*","connect";
};

grant codeBase "file:./Login.jar", principal "Alice" {
    permission java.io.FilePermission "/tmp/toto","write";
    permission java.io.SocketPermission
   "www.alice.org:1234","listen,accept";
};
```

java.io.FilePermission

- Tous les fichiers du répertoire utilisateur :
  - `permission java.io.FilePermission "${user.home}${/}*", "read";`
- Tous les fichiers du répertoire et des sous répertoires :
  - `permission java.io.FilePermission "/toto/-", "read,execute";`
- Actions : read, write, delete et execute

java.io.SocketPermission

- Permission java.io.SocketPermission "*","accept"
  - "204.160.241.99","accept"
  - "*.com","connect"
  - *.sun.com:80","accept"
  - "*.sun.com:-1023","accept"
  - "*.sun.com:1024-","connect"
  - "java.sun.com:8000-9000", "connect,accept"
  - "localhost:1024-", "accept,connect,listen"
- Actions : accept, connect, listen et resolve

Grant : codeBase

- On peut utiliser les caractères spéciaux : / * -
- Tous les fichiers .class du répertoire :
  - `codeBase "file:/tmp/"`
- Tous les fichiers .class et .jar du répertoire :
  - `codeBase "file:/tmp/*"`
- Tous les fichiers .class et .jar du répertoire et des sous répertoires :
  - `codeBase "file:/tmp/-"`

Grant : principal

```
grant principal
 javax.security.auth.x500.X500Principal "cn=Alice" {
permission java.io.FilePermission
   "/tmp/games","read,write"; };
```

- Le code exécuté par Alice peut écrire et lire dans le répertoire /tmp/games
- On peut utiliser ${{self}} dans les permissions pour représenter l'identité du principal de l'entrée grant.

Grant : keystore

```
keystore "http://foo.bar.com/blah/.keystore";
grant principal "Alice" {
permission java.io.FilePermission
   "/tmp/games","read,write"; };
```

- Si on défini l'identité avec une seule chaîne, on recherche cet alias dans le keystore et on remplace l'identité par :

```
javax.security.auth.x500.X500Principal "cn=Alice"
```

- On peut utiliser ${{alias:Alice}} dans les permissions pour représenter l'entrée DN (distinguished name) du certificat de l'alias Alice dans le keystore.

SecurityManager

```
import java.io.FilePermission;
...
// Pour ne pas avoir a utiliser -Djava.security.manager
// SecurityManager security=new SecurityManager();
SecurityManager security=System.getSecurityManager();
if (security!=null)
{
 System.out.println("checkPermission...");
 FilePermission perm=new FilePermission("/tmp","read");
 // Vérification du droit de lecture
 // Génère une exception suivant le fichier policy
 security.checkPermission(perm);
}
```

JAVA GSS-API

- Generic Security Services Application Program Interface (GSS-API RFC-2853)
- GSSAPI sert à échanger des messages sécurisés (tokens) entre applications. Équivalent à JSSE (SSL/TLS).
- Modèle client/serveur avec un contexte de sécurité (GSSContext).

Contexte client

```
import org.ietf.jgss.*;
...
Oid krb5Oid=new Oid("1.2.840.113554.1.2.2");
GSSManager manager=GSSManager.getInstance();
String server="host/pcacer@NT.BEZIERS.FR";
GSSName serverName=manager.createName(server,null);
GSSContext context=manager.createContext(serverName,
   krb5Oid,null,GSSContext.DEFAULT_LIFETIME);
context.requestMutualAuth(true);
context.requestConf(true);
context.requestInteg(true);
```

Contexte serveur

```
import org.ietf.jgss.*;
...
// Pour utiliser l'identité par défaut
GSSContext context=manager.createContext((GSSCredential)null);
// Pour fixer l'identité du serveur
Oid krb5Oid=new Oid("1.2.840.113554.1.2.2");
Oid krb5PrincipalNameType=new Oid("1.2.840.113554.1.2.2.1");
String server="host/pcacer@NT.BEZIERS.FR";
GSSName serverName=
    manager.createName(server,krb5PrincipalNameType);
GSSCredential serverCreds=manager.createCredential(serverName,
   GSSCredential.DEFAULT_LIFETIME,krb5Oid,
   GSSCredential.ACCEPT_ONLY);
GSSContext context=manager.createContext(serverCreds);
```

Établissement du contexte client (1)

```
byte[] token=new byte[0];
while (!context.isEstablished())
{
   token=context.initSecContext(token,0,token.length);
   if (token!=null)
   {
     System.out.println("=> Token : "
                        +token.length+" octets.");
      outStream.writeInt(token.length);
      outStream.write(token);
     outStream.flush();
   }
```

Établissement du contexte client (2)

```
   if (!context.isEstablished())
   {
       token=new byte[inStream.readInt()];
       System.out.println("<= Token : "
                          +token.length+" octets.");
       inStream.readFully(token);
   }
} // Fin du while
System.out.println("Cli :"+context.getSrcName());
System.out.println("Srv :"+context.getTargName());
```

Établissement du contexte serveur (1)

```
byte[] token=null;
while (!context.isEstablished())
{
  token=new byte[inStream.readInt()];
  System.out.println("<= Token : "
                     +token.length+" octets.");
  inStream.readFully(token);
  token=context.acceptSecContext(token,0,
  token.length);
```

Établissement du contexte serveur (2)

```
   if (!context.isEstablished())
   {
       System.out.println("=> Token : "
                          +token.length+" octets.");
       outStream.writeInt(token.length);
       outStream.write(token);
       outStream.flush();
   }
} // Fin du while
System.out.println("Cli :"+context.getTargName());
System.out.println("Srv :"+context.getSrcName());
```

Envoi d'un token

```
// 1er arg 0 => Quality-of-Protection.
// 2nd arg true => token crypté.
MessageProp prop=new MessageProp(0,true);
byte[] messageBytes="Hello There!\0".getBytes();
token=context.wrap(messageBytes,
      0,messageBytes.length,prop);
System.out.println("Token : "+token.length);
outStream.writeInt(token.length);
outStream.write(token);
outStream.flush();
```

Réception d'un token

```
token=new byte[inStream.readInt()];
System.out.println("Token : "+token.length);
inStream.readFully(token);
// MessageProp default values of 0 and false.
MessageProp prop=new MessageProp(0,false);
byte[] bytes=context.unwrap(token,0,token.length,
   prop);
String str=new String(bytes);
System.out.println("Data : "+str);
System.out.println("Confidentialite : "
                   +prop.getPrivacy());
```

JAVA SASL

- Simple Authentication and Security Layer (RFC 2222)
- Protocole challenge/response.
- Utilisé entre autres par :
  - LDAPv3 (Lightweight Directory Access Protocol)
  - IMAPv4 (Internet Message Access Protocol)
  - SMTP (Simple Mail Transfert Protocol)
- javax.security.sasl.SaslServer
- javax.security.sasl.SaslClient

Exemple

```
SaslServer ss=
   Sasl.createSaslServer(mechanism
     ,protocol,myName,props
     ,callbackHandler);
SaslClient sc=
   Sasl.createSaslClient(mechanisms
     ,authzid,protocol,serverName
     ,props,callbackHandler);
```

Provider SunSASL

- Client Mechanisms
  - PLAIN (RFC 2595).
  - CRAM-MD5 (RFC 2195).
  - DIGEST-MD5 (RFC 2831).
  - GSSAPI (RFC 2222).
  - EXTERNAL (RFC 2222).
- Server Mechanisms
  - CRAM-MD5
  - DIGEST-MD5
  - GSSAPI (Kerberos v5)

Valeurs à définir par le client (1)

- **PLAIN :**
  - authorizationId
  - Sasl.POLICY_NOANONYMOUS
  - NameCallback, PasswordCallback
- **EXTERNAL :**
  - authorizationId, External channel
  - Sasl.POLICY_NOPLAINTEXT, Sasl.POLICY_NOACTIVE, Sasl.POLICY_NODICTIONARY
- **CRAM-MD5 :**
  - authorizationId (option)
  - Sasl.POLICY_NOANONYMOUS, Sasl.POLICY_NOPLAINTEXT
  - NameCallback, PasswordCallback

Valeurs à définir par le client (2)

- **DIGEST-MD5 :**
  - authorizationId, protocolId, serverName
  - Sasl.POLICY_NOANONYMOUS, Sasl.POLICY_NOPLAINTEXT
  - NameCallback, PasswordCallback, RealmCallback, RealmChoiceCallback
  - Sasl.QOP, Sasl.STRENGTH, Sasl.MAX_BUFFER, Sasl.SERVER_AUTH, javax.security.sasl.sendmaxbuffer, com.sun.security.sasl.digest.cipher
- **GSS-API :**
  - JASS Subject, authorizationId, protocolId, serverName
  - Sasl.POLICY_NOACTIVE, Sasl.POLICY_NOANONYMOUS, Sasl.POLICY_NOPLAINTEXT
  - NameCallback, PasswordCallback, RealmCallback, RealmChoiceCallback
  - Sasl.QOP, Sasl.MAX_BUFFER, Sasl.SERVER_AUTH, javax.security.sasl.sendmaxbuffer

Valeurs à définir par le serveur (1)

- **CRAM-MD5 :**
  - serverName
  - Sasl.POLICY_NOANONYMOUS, Sasl.POLICY_NOPLAINTEXT
  - AuthorizeCallback, NameCallback, PasswordCallback
- **DIGEST-MD5 :**
  - protocolId, serverName
  - Sasl.POLICY_NOANONYMOUS, Sasl.POLICY_NOPLAINTEXT
  - AuthorizeCallback, NameCallback, PasswordCallback, RealmCallback
  - Sasl.QOP, Sasl.STRENGTH, Sasl.MAX_BUFFER, javax.security.sasl.sendmaxbuffer, com.sun.security.sasl.digest.realm, com.sun.security.sasl.digest.utf8

Valeurs à définir par le serveur (2)

- **GSS-API :**
  - JASS Subject, protocolId, serverName
  - Sasl.POLICY_NOACTIVE, Sasl.POLICY_NOANONYMOUS, Sasl.POLICY_NOPLAINTEXT
  - AuthorizeCallback
  - Sasl.QOP, Sasl.MAX_BUFFER, javax.security.sasl.sendmaxbuffer

SaslSTRENGTH

- **high :**
  - Triple DES - RC4 128 bits (3des - rc4)
- **medium :**
  - DES - RC4 56 bits (des - rc4-56)
- **low :**
  - RC4 40 bits (rc4-40)

Debug

- CRAM-MD5 : FINE
- DIGEST-MD5 : INFO, FINE, FINER, FINEST
- GSSAPI : FINE, FINER, FINEST

```
javax.security.sasl.level=FINEST
handlers=java.util.logging.ConsoleHandler
java.util.logging.ConsoleHandler.level=FINEST
```

Références

- http://java.sun.com/j2se/1.5.0/docs/guide/security/
- JAVA Cryptography - O'Reilly - 1998
- http://www.nist.gov/itl/div897/pubs/
- http://www.rsa.com/rsalabs/pubs/PKCS/