Introduction aux techniques de chiffrement

Pas de panique, cette introduction n'est présente que pour comprendre les chiffrements qui sont mis en œuvre couramment dans la sécurisation des applications. Il s'agit juste de présenter les grands principes, sans entrer dans les détails mathématiques et algorithmiques.

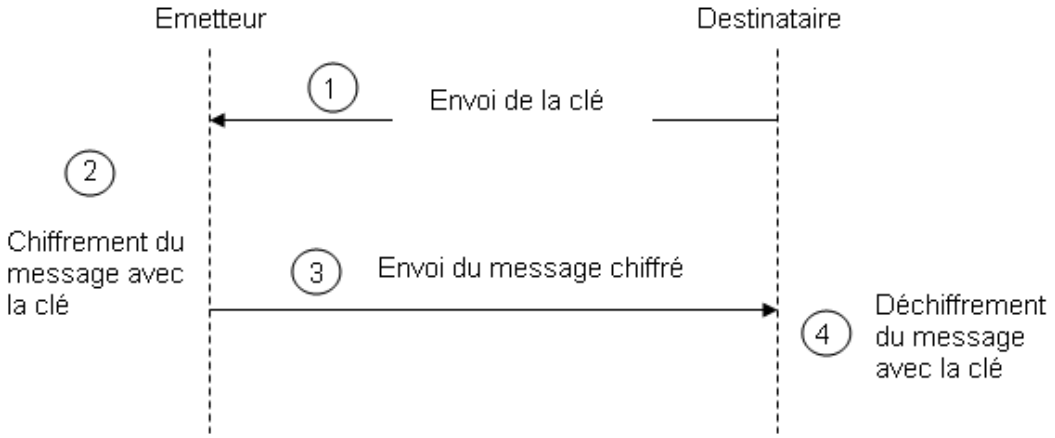
Une des premières étapes de la sécurisation des applications distribuées, est de sécuriser les communications entre les clients et les serveurs. Pour ce faire, les messages échangés vont être codés. Le fait de coder les messages s'appelle le chiffrement, l'opération inverse est le déchiffrement.

L'opération de chiffrement est effectuée à l'aide d'une clé de chiffrement, le décodage l'est par une clé de déchiffrement.

Lorsque les clés de chiffrement et de déchiffrement sont identiques, les clés sont dites symétriques. Lorsque les clés de chiffrement et de déchiffrement sont différentes, les clés sont dites asymétriques. Le chiffrement est dit aussi, par clé publique.

1. Chiffrement symétrique

La même clé est utilisée pour les opérations de chiffrement et déchiffrement. La clé est connue par les deux parties prenantes de la communication. Ce chiffrement repose sur le principe que les deux utilisateurs (au sens large : humain ou système) connaissent la clé de chiffrement. Il y a donc échange d'un secret : la clé.



Le principal avantage de ce type de chiffrement est sa simplicité, et donc, sa rapidité, en terme de temps de calcul, pour l'algorithme de codage et décodage.

Les désavantages de ce chiffrement symétrique sont les suivants :

- Le système repose sur l'échange d'un secret (la clé de chiffrement). L'émetteur du message a besoin de la clé du destinataire. Il faut donc sécuriser le canal d'échange.
- Si un utilisateur doit communiquer avec plusieurs autres utilisateurs, il faut une clé par interlocuteur, ce qui, pour un groupe de N utilisateurs désirant utiliser entre eux ce système, il faut distribuer, et gérer N*(N-1)/2 clés.
- De plus, il a été démontré par Claude SHANNON, que pour que le système soit totalement sûr, la longueur de la clé doit être égale à la longueur du message.

2. Chiffrement asymétrique

Dans ce type de chiffrement, les opérations de chiffrement et de déchiffrement sont réalisées au moyen de clés différentes.

Le chiffrement est réalisé à l'aide d'une clé dite publique, car connue par tous les émetteurs de message.

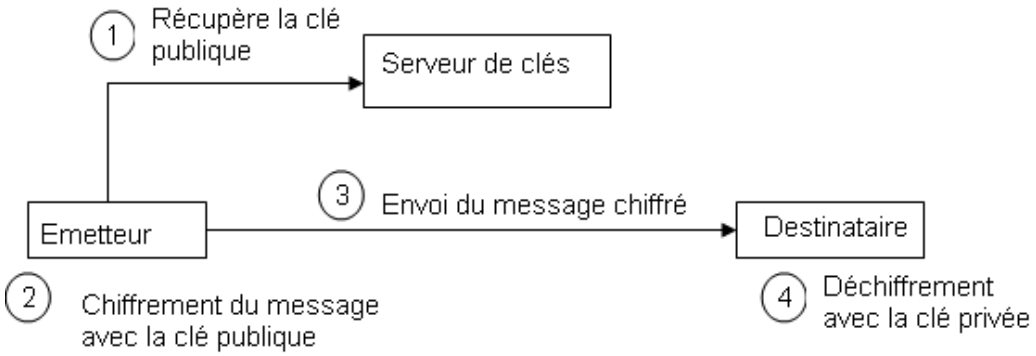
Le déchiffrement est réalisé à l'aide d'une clé privée, connue seulement du destinataire des messages.

Contrairement au chiffrement symétrique, les algorithmes de chiffrement et déchiffrement sont totalement différents.

Ce système utilise des fonctions faciles à calculer dans un sens, le chiffrement, et très difficiles à inverser algorithmiquement dans l'autre sens, le déchiffrement. Schématiquement, ces fonctions se comportent comme le carré

et son inverse. L'algorithme de la mise au carré d'un nombre est aisé à coder, c'est ce nombre multiplié par lui-même. L'opération inverse, la racine carrée, possède un algorithme déjà nettement plus complexe.

Le destinataire du message crée, aléatoirement, la clé privée. La clé publique est déduite de la clé privée. La clé publique est diffusée par un canal qui n'a pas besoin d'être sécurisé.



Les principaux avantages du chiffrement asymétrique sont les suivants :

- il n'y a pas partage d'un secret entre l'émetteur et le destinataire du message, donc pas de nécessité d'utiliser un canal de communication sécurisé ;
- il n'y a plus de multiples clés à gérer avec les différents utilisateurs qui veulent échanger des messages. Chaque utilisateur possède une clé privée, et diffuse sa clé publique.

Le principal inconvénient est que le temps de calcul de décodage est moins rapide que pour le chiffrement symétrique.

Il faut aussi considérer que l'émetteur du message doit être sûr que la clé publique qu'il reçoit est bien celle du destinataire. Pour permettre d'assurer l'origine de la clé publique, cette clé va être associée à une entité (organisme, personne, machine…) au sein d'un certificat, qui sera délivré par une autorité de certification (CA pour *Certification Authority*).

3. Le certificat

L'objectif du certificat est de garantir que la clé publique est bien celle de celui à qui nous voulons envoyer le message. Les certificats sont structurés en deux parties, selon la norme X 509.

Une partie contient les informations sur le certificat lui-même :

- Version de la norme X 509 ;
- Numéro de série du certificat ;
- Algorithme de chiffrement employé ;
- Le nom de l'autorité de certification ;
- La date de début de validité du certificat ;
- La date de fin de validité du certificat ;
- L'objet de l'utilisation de la clé publique ;
- La clé publique du propriétaire du certificat.

L'autre partie contient la signature de l'organisme de certification. Une empreinte des informations précédentes est créée par une fonction de hashage. Cette empreinte est chiffrée par la clé privée de l'autorité de certification pour générer la signature.

Le certificat est un fichier ressemblant à ceci :

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1218035244 (0x4899be2c)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=FR, ST=Paris, L=Paris, O=organisation, OU=unite, CN=client
        Validity
            Not Before: Aug  6 15:07:24 2008 GMT
            Not After : Nov  4 15:07:24 2008 GMT
        Subject: C=FR, ST=Paris, L=Paris, O=organisation, OU=unite, CN=client
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
            RSA Public Key: (1024 bit)
                Modulus (1024 bit):
                    00:94:cb:b5:8a:51:44:a2:97:bd:0d:51:32:dc:0b:
                    97:7e:15:52:d7:28:41:36:fe:27:cd:83:af:da:3b:
                    6c:0f:74:9d:68:a2:a1:68:49:f8:22:b0:2e:11:aa:
                    d3:c0:53:7e:26:67:b1:55:da:74:11:39:43:12:df:
                    0b:f1:31:07:8b:1b:e9:e5:c3:d4:66:bd:7e:20:85:
                    2b:27:52:48:b7:eb:82:d1:2f:f2:c3:46:2c:c7:5e:
                    e3:34:1d:e9:a2:ba:13:8f:70:6a:d5:ec:f4:2f:a9:
                    a8:2b:38:fb:75:4c:a5:14:a2:d6:f9:45:c9:e8:21:
                    be:b3:d8:42:86:db:c0:b7:1b
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha1WithRSAEncryption
        63:81:15:4e:7f:09:b0:f0:18:65:a2:99:b8:f0:6e:3a:b8:84:
        41:28:98:72:10:93:ce:35:f7:3b:16:fe:61:23:30:b8:b0:67:
        fd:23:9a:8d:b3:dd:7d:2c:78:69:4c:db:9c:a8:70:a3:47:5f:
        de:b9:fa:36:5f:ac:4a:34:69:8d:e4:58:a1:a8:30:f9:d6:29:
        48:8b:5a:79:18:77:7a:8b:6a:3f:7e:ef:91:63:0c:1d:c6:6f:
        83:06:0e:71:06:28:72:f3:b2:4a:52:36:70:3c:98:16:3b:af:
        2a:45:30:a4:90:04:1c:87:1c:0f:19:5d:41:33:58:d0:18:b0:
        24:24
-----BEGIN CERTIFICATE-----
MIICQTCCAaqgAwIBAgIESJm+LDANBgkqhkiG9w0BAQUFADBlMQswCQYDVQQGEwJG
UjEOMAwGA1UECBMFUGFyaXMxDjAMBgNVBAcTBVBhcmlzMRUwEwYDVQQKEwxvcmdh
bmlzYXRpb24xDjAMBgNVBAsTBXVuaXRlMQ8wDQYDVQQDEwZjbGllbnQwHhcNMDgw
ODA2MTUwNzI0WhcNMDgxMTA0MTUwNzI0WjBlMQswCQYDVQQGEwJGUjEOMAwGA1UE
CBMFUGFyaXMxDjAMBgNVBAcTBVBhcmlzMRUwEwYDVQQKEwxvcmdhbmlzYXRpb24x
DjAMBgNVBAsTBXVuaXRlMQ8wDQYDVQQDEwZjbGllbnQwgZ8wDQYJKoZIhvcNAQEB
BQADgY0AMIGJAoGBAJTLtYpRRKKXvQ1RMtwLl34VUtcoQTb+J82Dr9o7bA90nWii
oWhJ+CKwLhGq08BTfiZnsVXadBE5QxLfC/ExB4sb6eXD1Ga9fiCFKydSSLfrgtEv
8sNGLMde4zQd6aK6E49watXs9C+pqCs4+3VMpRSi1vlFyeghvrPYQobbwLcbAgMB
AAEwDQYJKoZIhvcNAQEFBQADgYEAY4EVTn8JsPAYZaKZuPBuOriEQSiYchCTzjX3
Oxb+YSMwuLBn/SOajbPdfSx4aUzbnKhwo0df3rn6Nl+sSjRpjeRYoagw+dYpSIta
eRh3eotqP37vkWMMHcZvgwYOcQYocvOySlI2cDyYFjuvKkUwpJAEHIccDxldQTNY
0BiwJCQ=
-----END CERTIFICATE-----
```

Pour améliorer la manipulation des certificats, ceux-ci sont enregistrés dans des magasins (keystore). Chaque éditeur possède son propre format de keystore, par exemple :

- JKS *(Java Key Standard)* pour le monde Java ;
- PCKS12 *(Public Key Cryptography Standard 12)* pour les navigateurs ;
- PEM *(Privacy Enchance Mail)* pour OpenSSL.

Il nous faudra donc plusieurs outils pour créer et convertir les certificats : **keytool**, **OpenSSL**.

4. Notion de clé de session

La clé de session est un compromis entre les chiffrements symétriques et asymétriques. En effet, le chiffrement asymétrique permet de s'affranchir des étapes d'échange de la clé du chiffrement symétrique (le secret partagé), mais ceci au détriment du temps de calcul. La notion de clé de session permet d'utiliser le meilleur des deux techniques.

a. Principe

L'émetteur du message récupère la clé publique du destinataire.

L'émetteur crée une clé aléatoire qui sera utilisée dans un chiffrage symétrique. Cette clé sera la clé de session. Puis, il chiffre la clé symétrique (la clé de session) qu'il vient de générer, avec la clé publique du destinataire.

L'émetteur envoie au destinataire la clé symétrique chiffrée. Celui-ci déchiffre la clé de session avec sa clé privée. Les deux participants ont maintenant la clé de session en commun, ils peuvent chiffrer et déchiffrer leurs messages avec celle-ci.

En résumé, la clé de session est le secret que partagent les deux interlocuteurs, cette clé sera utilisée pour le chiffrement symétrique. L'envoi de la clé de session se fait par le biais d'un canal sécurisé mis en place par un chiffrement asymétrique.

Le protocole HTTPS dont nous verrons la mise en œuvre plus loin, utilise cette notion de clé de session.

Introduction à JAAS

Le framework JAAS permet de gérer l'authentification et les autorisations d'un utilisateur. Il crée un objet, partagé par l'authentification et l'autorisation, et étend le modèle de sécurité standard pour gérer cet objet.

Lors du développement d'une application JEE, nous n'avons pas à prendre en compte les profils des utilisateurs et leurs droits vis-à-vis de l'application.

Le modèle déclaratif de sécurité délègue au serveur d'application le soin d'appliquer les contrôles nécessaires. Le développeur peut, aussi, gérer lui-même la sécurité, ceci ne sera pas traité ici.

Chaque utilisateur possède un identifiant et un mot de passe. L'utilisateur est ici défini au sens large, cela peut être une personne, un groupe, un service…

Un utilisateur est relié à un rôle dans l'application. Le rôle représente l'ensemble des droits d'accès à l'application. Le développeur pourra lors du développement, prévoir les rôles qui auront accès à certaines ressources du site Web, ou à certaines méthodes des EJB.

Il faut ensuite définir les mécanismes par lesquels le serveur retrouvera les informations d'identification de l'utilisateur.

1. Les concepts clés de JAAS

Un utilisateur est représenté par **sujet**, qui est un objet de type `Subject`. Le sujet est composé d'un ensemble **d'identités**, de type `Principal`.

a. Les classes de base de JAAS

- `javax.security.auth.`**`Subject`** : représente un demandeur (individu, organisation, groupe) qui peut posséder plusieurs identités. La phase d'authentification va permettre d'authentifier le sujet.

- `java.security.`**`Principal`** : représente une identité associée à un objet `Subject`. Un sujet peut avoir plusieurs identités `Principal` : son nom (John Doe), son numéro de sécurité sociale (1730844....), son surnom (Joe).

b. Les classes d'authentification des JAAS

- `javax.security.auth.login.`**`LoginContext`** : permet aux objets `Subject` de se connecter et déconnecter du système. Cette API n'est pas liée à la technologie d'identification : écran d'authentification, carte à puce, empreinte…

- `javax.security.auth.spi.`**`LoginModule`** : interface qui doit être implémentée par les fournisseurs de service d'authentification. La classe `org.jboss.security.auth.spi.UsersRolesLoginModule` implémente cette interface. Elle possède 5 méthodes qui permettent l'authentification :

    - `initialize` qui initialise l'instance de LoginModule ;

    - `login` qui effectue l'authentification ;

    - `commit` qui valide l'authentification et ajoute les Principal au sujet ;

    - `abort` qui abandonne le processus d'authentification ;

    - `logout` qui supprime les Principal.

- `javax.security.auth.login.`**`Configuration`** : permet de spécifier quels sont les LoginModule qui seront utilisés, leur ordre d'utilisation et leur comportement.

- `javax.security.auth.callback.`**`CallbackHandler`** : interface à implémenter par les applications qui doivent utiliser les informations du service d'authentification, par exemple pour coder la manière dont l'utilisateur saisit son identifiant et mot de passe.

- `javax.security.auth.callback.`**`Callback`** : interface marqueur implémentée par les objets utilisés par une implémentation `CallbackHandler`. Des implémentations standard existent déjà : NameCallback, PaswordCallback par exemple.

c. Exemple d'une implémentation d'authentification personnalisée

Vous trouverez le code complet dans le projet "Chap 8 - AAS".

Le code présenté ici est volontairement très simple pour se focaliser sur les mécanismes mis en œuvre. L'objectif est de faire une saisie d'un identifiant et un mot de passe, en mode console, et de valider l'identifiant "toto"' avec le mot de passe "toto".

La classe `MainAuthentification` contient le point d'entrée de l'application qui commence par instancier un `LoginContext`, en lui passant le nom de la configuration et notre classe de type `CallbackHandler`.

```
public static void main(String[] args) throws Exception
{
    LoginContext  lc = new LoginContext("Exemple",
                               new AuthCallbackHandler());
    lc.login();
}
```

Le fichier de configuration *exemple_jaas.config* contient la configuration requise. Ce fichier est précisé par le passage au compilateur de la propriété :

```
-Djava.security.auth.login.config==exemple_jaas.config
```

Si vous utilisez Eclipse, vous devez ajouter cette propriété aux arguments passés à la machine virtuelle, dans la configuration du "run" de votre projet.

```
Exemple
{
    fr.editions.eni.jboss.jaas.ExempleLoginModule required debug=true;
};
```

On y retrouve :

- le nom de la configuration : `Exemple` ;
- la classe qui implémente notre méthode d'authentification et dont les méthodes seront appelées par l'API JAAS, ici : `fr.editions.eni.jboss.jaas.ExempleLoginModule` ;
- flag utilisé si plusieurs `LoginModule` sont spécifiés. Cela permet à JAAS de définir la stratégie à suivre si une authentification échoue. Les modules `LoginModule` sont appelés dans l'ordre de leur déclaration ;
- `required` : module nécessaire pour l'authentification globale. Quel que soit le résultat de l'authentification par ce module, l'authentification continue sur les modules suivants ;
- `requisite` : module nécessaire pour l'authentification globale. Si l'authentification est effectuée, les modules suivants sont appelés, sinon le contrôle repasse à l'application ;
- `sufficient` : module non nécessaire pour l'authentification globale. Si l'authentification réussit les autres modules ne sont pas appelés, si elle échoue, les autres modules sont appelés ;
- `optional` : module non nécessaire pour l'authentification globale. Quel que soit le résultat de l'authentification, les modules suivants sont appelés ;
- une option qui est liée au module et qui nous permet ici d'afficher des informations si le mode debug est choisi.

L'exemple de configuration suivant, tiré de la documentation de Sun permet de mieux appréhender le rôle du flag

dans une liste de modules.

```
Login2 {
       sample.SampleLoginModule required;
       com.sun.security.auth.module.NTLoginModule sufficient;
       com.foo.SmartCard requisite debug=true;
       com.foo.Kerberos optional debug=true;
    };
```

États d'authentification de Login2									
SampleLoginModule	required	V	V	V	V	F	F	F	F
NTLoginModule	sufficient	V	F	F	F	V	F	F	F
SmartCard	requisite	*	V	V	F	*	V	V	F
Kerrebos	optional	*	V	F	*	*	V	F	*
Authentification globale		V	V	V	F	F	F	F	F

V indique que l'authentification passe.

F indique que l'authentification échoue.

* indique que le module n'est pas appelé.

La classe `ExempleLoginModule` implémente l'interface `LoginModule`. Elle est chargée de l'authentification et délègue au `CallbackHandler` le moyen de récupérer les informations (identifiant et mot de passe) auprès de l'utilisateur.

```
public boolean login() throws LoginException
{

    // demande l'identifiant et le mot de passe
    if (callbackHandler == null)
    throw new LoginException("ERREUR : pas de CallbackHandler");

    Callback[] callbacks = new Callback[2];
    callbacks[0] = new NameCallback("identifiant: ");
    callbacks[1] = new PasswordCallback("mot de passe: ", false);

    try
    {
        callbackHandler.handle(callbacks);
        username = ((NameCallback) callbacks[0]).getName();
        pswd = new String(((PasswordCallback)
                           callbacks[1]).getPassword());
        ((PasswordCallback) callbacks[1]).clearPassword();
    }
…
```

Lors de l'appel de la méthode `initialize(..)` un `Subject` et le `AuthCallbackHandler` ont été passés à l'instance de `ExempleLoginModule`. La méthode `login()` crée un tableau de `Callback`, remplit ce tableau avec un `NameCallback` et un `PasswordCallback`, puis la méthode `handle(…)` de `AuthCallbackHandler` est invoquée.

La méthode `handle(..)` de `AuthCallbackhandler` est chargée de récupérer auprès de l'utilisateur, son identifiant et son mot de passe. Un parcours du tableau des `Callback` est effectué, et en fonction du type de `Callback`, le comportement adéquat est mis en place.

```
…
else if (callbacks[i] instanceof NameCallback)
{

    // demande de l'identifiant
    NameCallback nc = (NameCallback) callbacks[i];

    System.err.print(nc.getPrompt());
```

```
    System.err.flush();
    nc.setName((new BufferedReader(new
                    InputStreamReader(System.in))).readLine());

}
else if (callbacks[i] instanceof PasswordCallback)
{

    // demande du mot de passe
    PasswordCallback pc = (PasswordCallback) callbacks[i];
    System.err.print(pc.getPrompt());
    System.err.flush();
    String pswd = new BufferedReader(new
                      InputStreamReader(System.in)).readLine();
    pc.setPassword(pswd.toCharArray());

}
…
```

Les informations sont récupérées sur la ligne de commande et passées aux méthodes des Callback. À l'issue de ce code, NameCallback et PassordCallBack possèdent un identifiant et un mot de passe.

Maintenant, la méthode `login()` de `ExempleLoginModule` peut effectuer l'authentification proprement dite.

```
…
if (username.equals("toto"))
{
    usernameCorrect = true;
    if (pswd.equals("toto"))
    {
        passwordCorrect = true;
    }
…
```

Cette méthode renverra `true` ou `false` en fonction du résultat.

Le gestionnaire de sécurité JBossSX

JBossSX, ou JBoss Security Extension, est le composant chargé de la sécurité dans JBoss. JBossSX fournit un modèle de sécurité déclaratif basé sur les rôles.

1. LoginModule de JBoss

Comme nous l'avons vu précédemment, il est toujours possible d'écrire ses propres `LoginModule`. JBoss nous en fournit déjà quelques-uns.

- `org.jboss.security.auth.spi.`**`IdentityLoginModule`** est un module permettant une simple authentification, avec seulement un identifiant et sans mot de passe.

  Les options pour ce module sont :

  - `principal` : indique que l'identifiant principal sera toujours utilisé ;
  - `roles` : précise la liste des rôles qui sont associés au login principal.

- `org.jboss.security.auth.spi.`**`UsersRolessLoginModule`** permet une authentification avec identifiant et mot de passe.

  Les options disponibles pour ce module sont :

  - `usersProperties` : nom du fichier des identifiants et mots de passe ;
  - `rolesProperties` : nom du fichier des associations entre identifiants et rôles ;
  - `hashAlgorithm` : précise, si nécessaire, le type d'algorithme de chiffrement des mots de passe.

- `org.jboss.security.auth.spi.`**`LdapLoginModule`** permet d'utiliser LDAP comme espace de stockage des identifiants et mots de passe.

- `org.jboss.security.auth.spi.`**`DatabaseServerLoginModule`** permet d'utiliser une base de données comme espace de stockage des identifiants et mots de passe.

2. Configuration du domaine de sécurité

La configuration des services de sécurité se trouve dans le fichier *conf/jboss-service.xml*, dont voici un extrait.

```
<!-- ===================================================== -->
<!-- Security                                              -->
<!-- ===================================================== -->

<mbean code="org.jboss.security.plugins.SecurityConfig"
    name="jboss.security:service=SecurityConfig">
    <attribute name="LoginConfig">
        jboss.security:service=XMLLoginConfig
    </attribute>
</mbean>

<mbean code="org.jboss.security.auth.login.XMLLoginConfig"
    name="jboss.security:service=XMLLoginConfig">
    <attribute name="ConfigResource">
        login-config.xml
    </attribute>
</mbean>
```

Vous remarquez que le service de sécurité `org.jboss.security.plugins.SecurityConfig` utilise un service

`org.jboss.security.auth.login.XMLLoginConfig` qui gère une configuration d'authentification JAAS. Ce fichier *login-config.xml* est situé dans le répertoire *conf*.

Ce fichier XML inclut une racine `<policy>` qui comporte les configurations des domaines de sécurité.

Prenons le domaine de sécurité de la console JMX.

```
…
<application-policy name = "jmx-console">
    <authentication>
    <login-module
        code="org.jboss.security.auth.spi.UsersRolesLoginModule"
    flag = "required">
    <module-option name="usersProperties">
        props/jmx-console-users.properties
    </module-option>
    <module-option name="rolesProperties">
        props/jmx-console-roles.properties
    </module-option>
    </login-module>
    </authentication>
</application-policy>
…
```

L'élément `<login-module>` déclare la classe de type `LoginModule` utilisée, ainsi que le flag de gestion du module.

Puis vient ensuite le paramétrage des options du module dans les éléments `<module-option>`.

Dans le fichier `jmx-console-users.properties` se trouve l'affectation des rôles à un utisateur sous la forme :

```
utilisateur = rôle1, rôle2
```

Le fichier `jmx-console-roles.properties` énumère les utilisateurs et mots de passe sous la forme :

```
utilisateur=mot_de_passe
```

Pour mieux sécuriser l'application et ne pas avoir la liste des mots de passe en clair dans un fichier, d'autres moyens sont évidemment disponibles : utilisation de base de données, de LDAP…

Exemple d'authentification avec interrogation d'une base de données

```
<application-policy name = "jbossmq">
  <authentication>
   <login-module
       code = "org.jboss.security.auth.spi.DatabaseServerLoginModule"
       flag = "required">
    <module-option
       name = "unauthenticatedIdentity">guest</module-option>
    <module-option name = "dsJndiName">java:/DefaultDS</module-option>
    <module-option name = "principalsQuery">
       SELECT PASSWD FROM JMS_USERS WHERE USERID=?
    </module-option>
    <module-option name = "rolesQuery">
       SELECT ROLEID, 'Roles' FROM JMS_ROLES WHERE USERID=?
    </module-option>
   </login-module>
  </authentication>
</application-policy>
```

La classe utilisée ici pour le `LoginModule` est de type `org.jboss.security.auth.spi.DatabaseServerLoginModule`. Le nom des tables (`JMS_ROLES` ici) doit être adapté à votre base de données. Nous retrouvons ensuite les options utilisées par le module :

- l'identité attribuée à un utilisateur non authentifié, ici `guest` ;
- le nom JNDI de la source de données utilisée, ici `java:/DefaultDS` ;
- la requête utilisée sur la base, pour retrouver le mot de passe en fonction de l'identifiant entré par l'utilisateur,

ici `SELECT PASSWD FROM JMS_USERS WHERE USERID=?`. Le point d'interrogation sera remplacé par l'identifiant lors de l'exécution de la requête ;

- la requête utilisée pour récupérer le profil de l'utilisateur en base de données `SELECT ROLEID, 'Roles' FROM JMS_ROLES WHERE USERID=?`. Le point d'interrogation sera remplacé par l'identifiant lors de l'exécution de la requête.

Nous allons voir quelques configurations dans la suite de ce chapitre.