

Bac à sable

Philippe PRADOS

site@philippe.prados.name



*Préservez l'environnement,
n'imprimez pas ce document*

TABLE DES MATIERES

1.	Privilèges du code java	3
2.	Ecrire des privilèges	3
3.	Les privilèges dans les serveurs d'application.....	4
4.	Les droits minimums	5
5.	Bac à sable	6
6.	Gardien	9

Avant-propos

Java propose un mécanisme de sécurité très fin, permettant de contrôler l'accès à toutes les ressources sensibles. Nous allons regarder comment l'utiliser et comment l'enrichir pour faire exécuter du code avec moins de privilèges que prévu, ce qui est parfois nécessaire pour supprimer des failles de sécurité.

Java propose plusieurs outils permettant de renforcer la sécurité des applications. Il existe des API de cryptographie avec la possibilité d'écrire de nouveau driver, des API ouvertes d'authentications, etc. Pour protéger les API sensibles d'une utilisation malveillante, java 2 propose un mécanisme de politique de sécurité, permettant à certaines parties du code de bénéficier de privilèges.

1. PRIVILEGES DU CODE JAVA

Les API sensibles peuvent alors exiger que l'appelant possède des privilèges pour pouvoir s'exécuter. Comment cela fonctionne et comment est-ce architecturé ?

Les classes sont chargés à partir d'une ressource. Cela peut être un fichier `class`, une archive `jar`, ou une ressource récupérée sur le WEB. À chaque classe est associée la ressource initiale d'où elle est extraite et éventuellement des signataires. La classe `CodeSource` permet d'associer la source du chargement de la classe et les signatures.

Un domaine de protection permet d'associer une source de code et un ensemble de privilèges. La classe `ProtectionDomain` se charge de cette association. Lors du chargement d'une classe, à partir d'un code source donnée, les privilèges sont récupérés dans le fichier `java.policy` de la JVM et un domaine de protection est créé. Celui-ci est associé à la classe. La méthode `Class.getProtectionDomain()` permet de consulter ces informations. Il est alors possible de retrouver la source d'une classe et de consulter les privilèges associés.

Le mécanisme de sécurité de java s'appuie sur ces informations. Un code s'exécute avec certains privilèges. Il peut invoquer des classes venant d'une autre archive ou faisant partie de la JVM. Toutes les méthodes invoquées obtiennent, par héritage d'appel, les privilèges de l'appelant. Un code critique qui désire vérifier que l'appelant possède des droits particuliers doit utiliser la classe `AccessController`.

```
if (System.getSecurityManager() != null)
    java.security.AccessController.checkPermission(
        new FilePermission("log.txt", "write"));
```

Ce code génère une exception si l'appelant ne possède pas le privilège d'écrire sur le fichier `log.txt`. Pour obtenir ce privilège, il faut demander à la JVM d'exécuter un code avec les droits associés à la classe lors du chargement. Cela s'effectue toujours avec la classe `AccessController`. Les méthodes `doPrivileged()` permettent d'exécuter un code avec les droits associés à la classe. Cela permet d'ouvrir les privilèges pour pouvoir invoquer des API plus sensibles. Deux interfaces sont proposées. La première, `PrivilegedAction`, permet d'exécuter un traitement sans pouvoir propager d'exception, autres que les exceptions non vérifiées. La deuxième, `PrivilegedExceptionAction` propose un service similaire, mais toutes les exceptions peuvent être propagées. Celles-ci peuvent être capturées par l'appelant, encapsulées dans une exception `PrivilegedActionException`. Généralement, l'application utilise une classe anonyme pour implémenter ces interfaces.

```
// Ici, privilèges hérités de l'appelant
FileOutputStream out=(FileOutputStream)AccessController.doPrivileged(
    new PrivilegedAction()
    {
        public Object run()
        {
            // Ici, privilèges propres à la classe
            return new FileOutputStream("log.txt"); // Accès possible
        }
    });
// Ici, à nouveau les privilèges hérités de l'appelant
```

Comment associer des privilèges à une classe ? Il faut modifier le fichier `java.policy` de la JVM ou placer votre code dans un répertoire considéré comme sûr par la JVM. C'est le cas du répertoire `/lib/ext`. Toutes les classes venant de ce répertoire bénéficient de tous les privilèges. La règle suivante déclare ceci :

```
grant codeBase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};
```

Pour n'ouvrir que certains privilèges, il faut enrichir le fichier `java.policy` afin de proposer des privilèges différents suivant la source des classes. Vous pouvez également demander l'utilisation d'un fichier de politique de sécurité différente à l'aide du paramètre `-Djava.security.policy` lors du lancement de la JVM.

```
java -Djava.security.manager -Djava.security.policy=<url> Main
```

La sécurité est vérifiée si, et seulement si, une instance de `SecurityManager` est installée dans la JVM.

```
System.setSecurityManager(new SecurityManager());
```

2. ECRIRE DES PRIVILEGES

De nombreux privilèges sont proposés par java. Un privilège est généralement décomposé en deux parties : une clef identifiant une ressource et une liste d'actions associées à la ressource. Suivant les privilèges, des syntaxes spécifiques permettent de regrouper des ressources ou des actions. Par exemple, la classe `FilePermission` accepte en premier paramètre le nom d'un fichier, le nom d'un répertoire suivi d'une étoile pour représenter tous les fichiers présents dans le répertoire, ou suivit d'un moins pour représenter tous les fichiers présents dans le

répertoire et dans ses sous-répertoires. Le nom particulier <<ALL FILES>> représente tous les fichiers. La classe propose quatre actions pouvant être combinée par des virgules : `read`, `write`, `execute` et `delete`. Voici quelques exemples valides.

```
new FilePermission("/batch/-", "execute");
new FilePermission(System.getProperty("java.io.tmpdir")+
    File.separatorChar+'*', "read,write");
```

Pour offrir ses propres droits, il faut déclarer une classe héritant de `java.security.Permission` dont le constructeur possède un ou deux paramètres. Celui-ci sera invoqué lors de l'analyse du fichier `java.policy`. Une instance de permission ayant une sémantique par valeur, il est nécessaire de coder correctement les méthodes `equals()` et `hashCode()`. La méthode `getActions()` doit retourner une chaîne de caractère avec toutes les actions déclarées, séparées par une virgule. La méthode `implies()` permet de vérifier si un privilège implique un autre privilège. C'est la technique utilisée par implémenter la classe `AllPermission`, donnant tous les droits. La méthode `implies()` de `AllPermission` retourne `true`. Le Source 1 est un exemple simple de privilège acceptant les actions `read` et `write` sur une ressource.

```
public class MySecurity extends Permission
{
    private boolean read;
    private boolean write;

    public MySecurity(String resource, String action)
    {
        super(resource);
        read=(action.indexOf("read")!=-1);
        write=(action.indexOf("write")!=-1);
    }
    public boolean implies(Permission permission)
    {
        if (permission instanceof MySecurity)
        {
            MySecurity perm=(MySecurity)permission;
            return (getName().equals(perm.getName()) &&
                (read | write) &&
                (!read || perm.read==read) &&
                (!write || perm.write==write));
        }
        return false;
    }
    public boolean equals(Object obj)
    {
        MySecurity perm=(MySecurity)obj;
        return (getName().equals(perm.getName()) &&
            (perm.read==read) && (perm.write==write));
    }
    public int hashCode()
    {
        return getName().hashCode();
    }
    public String getActions()
    {
        StringBuffer buf=new StringBuffer();
        if (read) buf.append("read,");
        if (write) buf.append("write,");
        if (buf.length()>0)
            buf.setLength(buf.length() - 1);
        return buf.toString();
    }
}
```

Source 1

Il est alors possible de vérifier un privilège à l'aide d'un code comme celui-ci.

```
if (System.getSecurityManager() != null)
    java.security.AccessController.checkPermission(
        new MySecurity("abc", "write"));
```

3. LES PRIVILEGES DANS LES SERVEURS D'APPLICATION

L'intégration de la sécurité fine dans les serveurs d'application n'est pas encore normalisée. Chaque implémentation utilise une approche différente.

Pour Tomcat, il faut modifier le fichier `catalina.policy`. (<http://jakarta.apache.org/tomcat/tomcat-5.5-doc/security-manager-howto.html>). Le `ClassLoader` des applications Web se charge d'associer les privilèges correspondant à chaque application. Ainsi, dans un hébergement mutualisé, vous pouvez limiter les droits de chaque application. Pour lancer Tomcat avec la sécurité Java 2, ajoutez le paramètre `-security` lors du lancement du serveur. Ceci est fortement recommandé.

WebSphere™ utilise une autre approche. Un fichier `was.policy` peut être présent dans le descripteur de déploiement de l'EAR. Cela permet de donner des privilèges à l'application WEB. D'autres fichiers similaires existent pour les droits des EJB.

Les droits des systèmes d'exploitations permettent de limiter les accès aux applications. Les serveurs d'applications J2EE sont lancés par un seul programme, la machine virtuelle java. Ainsi, toutes les applications WEB bénéficient des droits accordés par le système d'exploitation à la JVM, pour l'utilisateur utilisé lors du lancement. Pour améliorer cela, et proposer des droits limités suivant les différentes applications hébergées, il faut utiliser la sécurité java 2. Celle-ci se met en place par un paramétrage du serveur ou en ajoutant le paramètre `-Djava.security.manager` lors du lancement de la JVM. La sécurité fine de Java 2 est plus riche que ce que propose le système d'exploitation.

4. LES DROITS MINIMUMS

Quels sont les droits minimums nécessaires ? Pour qu'une application fonctionne correctement, il faut au moins avoir le droit de lire des ressources à partir du [CodeSource](#) de la classe, l'URL d'où est issu la classe. En effet, lors du chargement d'une classe, celle-ci peut avoir besoins d'autres classes. Un chargeur de classe est alors invoqué. Celui-ci s'exécute avec les droits de l'appelant, la classe initiale. Il doit pouvoir charger les autres fichiers `.class`. Lors de l'installation d'une classe, il faut ajouter un accès au code base. Cela peut être un accès répertoire, un accès à une archive seule, un accès à une machine du réseau,... Le fichier source 2 permet d'ajouter les privilèges nécessaires.

```
public static PermissionCollection addResourcesPermission(CodeSource cs,
    PermissionCollection perms)
{
    try
    {
        Permission perm=null;
        URL base=cs.getLocation();
        if ("file".equals(base.getProtocol()))
        {
            File f=new File(new URI(base.getProtocol(), base.getPath(), base.getRef()));
            boolean isDir=true;
            try
            {
                isDir=f.isDirectory();
            }
            catch (AccessControlException x)
            {
                // Ignore, because the acces is open inside the directory but not for the directory.
            }
            if (isDir)
                perm=new FilePermission(f.getAbsolutePath()+File.separatorChar+"-", "read");
            else
                perm=new FilePermission(f.getAbsolutePath(), "read");
        }
        else if ("jar".equals(base.getProtocol()))
        {
            String path=base.getPath();
            path=path.substring(0,path.indexOf('!'));
            base=new URL(path);
            File f=new File(new URI(base.getProtocol(), base.getPath(), base.getRef()));
            perm=new FilePermission(f.getAbsolutePath(), "read");
        }
        else if ("http".equals(base.getProtocol()) || "https".equals(base.getProtocol()))
        {
            String host=base.getHost();
            int port=base.getPort();
            if (port==-1) port=base.getDefaultPort();
            perm=new SocketPermission(host+": "+port, "connect");
        }
        if (trace) System.out.println("-> add "+perm);
        if (perm!=null) perms.add(perm);
    }
    catch (URISyntaxException e)
    {
        // Ignore, impossible to set the privilege
    }
    catch (MalformedURLException e)
    {
        // Ignore, impossible to set the privilege
    }
}
return perms;
}
```

Source 2

Un code qui bénéficie d'une augmentation de privilège doit être le plus petit possible, et ne doit pas invoquer de code non sûr. Par exemple, si vous implémentez un mécanisme de notification, les événements ne doivent pas être invoqué dans les sections critiques. Sinon, les gestionnaires d'évènement enregistrés peuvent bénéficier des privilèges hérités. Il ne faut pas retourner des informations critiques dans un code privilégié. Par exemple, si une méthode publique utilise ses privilèges pour obtenir le nom de l'utilisateur, il ne faut pas le retourner à l'appelant. Sinon, la protection sur cette information confidentielle tombe. Pour contourner cette limitation, il faut utiliser les [GuardedObject](#) décrits plus loin.

5. BAC A SABLE

Parfois, certaines parties du code doivent bénéficier de moins de privilèges que le reste. Java ne propose pas d'API pour cela. Il est possible de bénéficier de plus de privilèges, mais pas de moins. Pourquoi est-ce nécessaire ? Parfois, l'application doit intégrer dynamiquement du code ou des traitements. Ceux-ci ne doivent pas forcément bénéficier des mêmes privilèges que l'application. Par exemple, imaginez une application permettant à certains utilisateurs de télécharger sur le serveur des feuilles XSL. Cela permet, par exemple, de produire des morceaux de pages pour un portail. Mais, le moteur XSL utilisé offre des fonctionnalités cachées, permettant l'écriture de fichiers et l'exécution d'un code arbitraire java. Un pirate ayant obtenu l'identification d'un utilisateur ayant le droit d'envoyer un fichier XSL, ou un utilisateur malveillant, peuvent prendre la main sur l'ensemble du serveur par cette fonctionnalité, apparemment sans risque. Par exemple, le filtre XSL, source 3, exploite les extensions de Xalan pour générer un fichier et l'exécuter.

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:java="http://xml.apache.org/xslt/java"
                xmlns:lxslt="http://xml.apache.org/xslt"
                xmlns:redirect="org.apache.xalan.xslt.extensions.Redirect"
                extension-element-prefixes="redirect"
                version="1.0">
  <xsl:output method="text"/>
  <xsl:template match="/">
    <redirect:open file="run.bat"/>
    <redirect:write file="run.bat">
notepad.exe
    </redirect:write>
    <redirect:close file="run.bat"/>
    <xsl:variable name="runtime" select="java:java.lang.Runtime.getRuntime()"/>
    <xsl:value-of select="java:exec($runtime, 'run.bat')"/>
  </xsl:template>
</xsl:stylesheet>
```

Source 3

Ce filtre XSL, sans protection particulière, permet de lancer un traitement sur le serveur. Ce type d'attaque produit généralement des pages JSP dans le serveur d'application. Celles-ci possèdent du code java bénéficiant des droits de l'application. Elles peuvent lire les fichiers de paramétrage avec les mots de passes, décompiler toutes les classes, etc.

Une autre attaque consiste à récupérer le contenu de fichiers présent sur le serveur. Un simple fichier XML permet cela.

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE hack [
<!ENTITY include SYSTEM "/etc/passwd">
]>
<hack>
&include;
</hack>
```

Lors de l'analyse du fichier XML, le contenu du fichier `/etc/passwd` est affiché.

Pour interdire cela, le moteur XSL ou tout autres composant dont vous n'avez pas la garantie d'innocuité, doit être placé dans un bac à sable, réduisant les privilèges du traitement au minimum. Ainsi, quelles que soit le contenu de la feuille XSL, il ne pourra pas porter préjudice au serveur. L'approche du bac à sable est utilisée pour le chargement des applets. Elles ne bénéficient de pratiquement aucun privilège.

Comment obtenir cela ? Comme nous l'avons vue, les droits sont associés à une classe lors de son chargement. Il n'est donc pas possible de perdre des privilèges. Le seul moyen à notre disposition est d'installer une nouvelle version d'une classe, en associant un `ProtectionDomain` sans privilège ou avec des privilèges réduits. Cette nouvelle classe va avoir besoin d'autres classes. Elles devront également être associées à un `ProtectionDomain` sans privilège. Ainsi, elles ne peuvent demander de privilèges supplémentaires. Le privilège d'accès en lecture au code source est le seul à ajouter.

Parfois, il est nécessaire de permettre l'invocation d'API de l'application. Cela risque de charger une nouvelle version des classes. Pour éviter cela, certains packages ne doivent pas être rechargés. Il y a en fait trois zones : Une zone verte avec les classes sans privilèges. Ce sont les classes dans le bac à sable. Une zone orange avec des classes pouvant obtenir des privilèges, mais dont le code est de confiance. Une zone rouge ne pouvant être invoquée par une classe de la zone verte.

Pour offrir cela, nous avons besoin de rédiger un chargeur de classe particulier. Celui-ci rechargera des classes à partir des codes sources. Déclarons un `SandBoxClassLoader` qui hérite de `SecureClassLoader`. Deux attributs sont nécessaires : un `ProtectionDomain` et une liste de packages de la zone orange.

```
class SandBoxClassLoader extends SecureClassLoader
{
  private List packages_ = new ArrayList();
  private ProtectionDomain domain_;
}
```

Un chargeur de classes possède généralement une liste de ressource pour y puiser les classes à installer. Dans notre cas, nous désirons nous appuyer sur les chargeurs de classes existant. Nous devons alors interroger les classes pour connaître le code source à utiliser. Le constructeur de notre `SandBoxClassLoader` attend trois paramètres : une classe dont la version privilégiée permet de retrouver le code source de la zone à protéger, une liste de permission à associer aux classes, et une liste de packages pour la zone orange. Dans tous les cas, nous ajoutons les privilèges d'accès en lecture au code source, pour permettre le chargement des classes (Source 4).

```

SandBoxClassLoader(Class clazz,PermissionCollection perms,Package[] packs)
{
    super(Thread.currentThread().getContextClassLoader());
    CodeSource cs=new CodeSource(clazz.getProtectionDomain().getCodeSource().getLocation(),
        null);
    addResourcesPermission(cs,perms);
    domain_=new ProtectionDomain(cs,perms);
    if (packs!=null)
        for (int i=packs.length-1;i>=0;--i)
            packages_.add(packs[i].getName());
}

```

Source 4

Nous pouvons alors ajouter une méthode permettant de savoir si une classe à installer doit faire partie ou non du bac à sable. Toutes les classes des packages javas ne seront pas rechargées. Les classes dont les packages correspondent à ceux enregistrés pour la zone orange ne seront pas réinstallées. Trois classes sont spéciales et ne doivent pas faire partie du bac à sable.

```

private final boolean isForSandBox(String name)
{
    if (name.startsWith("java.") ||
        name.startsWith("javax.") ||
        name.equals(UnprivilegedAction.class.getName()) ||
        name.equals(UnprivilegedActionException.class.getName()) ||
        name.equals(UnprivilegedExceptionAction.class.getName()))
        return false;
    final int idx=name.lastIndexOf('.');
    if (idx==-1) return true;
    return !packages_.contains(name.substring(0,idx));
}

```

Nous pouvons alors proposer une méthode `loadClass()`. Celle-ci cherche à savoir si la classe à installer doit faire partie du bac à sable. Si c'est le cas, la méthode demande à son propre classe loader de charger la ressource dont le nom correspond au nom de la classe avec les points remplacés par des slashes et avec le suffixe « `.class` ». Cette technique permet de récupérer les fichiers classes en ignorant leurs provenances (archives, réseaux, fichiers, base de données, etc.) Invoquer le chargeur de classes du chargeur de classe demande d'avoir des privilèges. En effet, nous entrons dans la zone rouge. C'est pour cela qu'une section critique est nécessaire dans la méthode `loadClass()` (Source 5).

```

public synchronized Class loadClass(final String className, boolean resolve)
    throws ClassNotFoundException
{
    Class c = findLoadedClass(className);
    if (c != null) return c;

    final String slashClassName=className.replace('.', '/').concat(".class");

    if (isForSandBox(className))
    {
        byte[] dotclass;
        try
        {
            dotclass =(byte[])AccessController.doPrivileged(new PrivilegedExceptionAction()
            {
                public Object run() throws ClassNotFoundException, IOException
                {
                    InputStream in=(getParent() == null)
                        ? ClassLoader.getSystemResourceAsStream(slashClassName)
                        : getClass().getClassLoader().getResourceAsStream(slashClassName);
                    if (in == null)
                        throw new ClassNotFoundException(className);

                    int len=in.available();
                    int start=0;
                    byte[] tampon=new byte[len];
                    do
                    {
                        start+=in.read(tampon,start,len);
                    }
                    while (start<len);
                    return tampon;
                }
            });
        }
        catch (PrivilegedActionException e)
        {
            if (e.getCause() instanceof ClassNotFoundException)

```

```

        throw (ClassNotFoundException)e.getCause();
    if (e.getCause() instanceof IOException)
        throw new ClassNotFoundException(className);
    }
}
return super.loadClass(className, resolve);
}

```

Source 5

L'invocation de `in.available()` permet de connaître la taille du fichier class. Un tableau correspondant est créé. Si le flux vient d'un fichier compressé, la lecture du tampon peut retourner moins d'octets que demandés. Il faut alors continuer la lecture jusqu'à récupération complète du fichier. Ensuite, l'invocation de `defineClass()` permet d'installer une nouvelle version de la classe, associé à un domaine moins privilégié.

Pour faciliter l'invocation de ce code, et singer la classe `AccesControler`, nous allons proposer deux interfaces permettant de lancer un code dans un bac à sable.

```

public interface UnprivilegedAction
{
    public Object run(Map params);
}
public interface UnprivilegedExceptionAction
{
    public Object run(Map params) throws Exception;
}

```

Une méthode `preparedUnprivileged()` permet d'obtenir une instance présent dans le bac à sable, implémentant l'une des deux interfaces.

```

public static UnprivilegedExceptionAction preparedUnprivileged(
    UnprivilegedExceptionAction action,PermissionCollection perms,Package[] packs)
    throws UnprivilegedActionException,InstantiationException
{
    try
    {
        ClassLoader loader=new SandBoxClassLoader(action.getClass(),perms,packs);
        return (UnprivilegedExceptionAction) loader.loadClass(
            action.getClass().getName()).newInstance();
    }
    catch (InstantiationException e)
    {
        throw new IllegalArgumentException(e.getMessage());
    }
    catch (IllegalAccessException e)
    {
        throw new IllegalArgumentException(e.getMessage());
    }
    catch (ClassNotFoundException e)
    {
        throw new IllegalArgumentException(e.getMessage());
    }
}

```

Le paramètre `action` permet de présenter un modèle d'instance à créer dans le bac à sable. L'instance retournée par la méthode peut être réutilisée afin d'éviter le rechargement des classes dans un nouveau bac à sable lors de chaque invocation. Une méthode `doUnprivileged()` permet de lancer un traitement en rechargeant les classes pour y supprimer les privilèges excessifs.

```

public static Object doUnprivileged(UnprivilegedAction action,
    Map params,PermissionCollection perms,Package[] packs)
{
    return preparedUnprivileged(action,perms,packs).run(params);
}

```

Les paramètres permettent une communication entre le bac à sable et l'extérieur. Dans le source complet, des déclinaisons des fonctions `doUnprivileged()` permettent d'imposer plus où moins de paramètres.

Comment utiliser ce code ? Il faut déclarer une classe implémentant une des deux interfaces proposées (Source 6). Attention, il n'est pas possible d'utiliser une classe anonyme car une classe anonyme ne possède pas de constructeur sans paramètre.

```

public static class XslSandBox implements UnprivilegedExceptionAction
{
    public Object run(Map params)
        throws TransformerConfigurationException, TransformerException
    {
        TransformerFactory transformer=TransformerFactory.newInstance();
        Templates template=transformer.newTemplates(
            new StreamSource(getClass().getResourceAsStream("test.xml")));
        template.newTransformer()
    }
}

```

```

        .transform(
            new StreamSource(getClass().getResourceAsStream("test.xml")),
            new StreamResult(System.out));
    return null;
}
}

```

Source 6

Puis utiliser les API.

```
Sandbox.doUnprivileged(new XslSandbox(),null);
```

Imaginons que la classe `org.orange.Orange`, désire être invoquée par une classe du bac à sable, et exige des privilèges pour s'exécuter. Elle peut utiliser la méthode `doPrivileged()` de `AccessController` pour les obtenir.

```

public class Orange
{
    public void privileges()
    {
        AccessController.doPrivileged(new PrivilegedAction()
        {
            public Object run()
            {
                // Code avec privilège de la zone orange
                return null;
            }
        });
    }
}

```

Mais, il ne faut pas qu'une version de cette classe soit rechargée par le chargeur de classe du bac à sable. Sinon, les privilèges associés sont inexistantes. Pour régler cela, il faut déclarer le package `org.orange` comme exclu du bac à sable.

```
Sandbox.doUnprivileged(new MySandbox(),null,
    new Package[]{Orange.class.getPackage()});
```

L'instance retournée par le traitement du bac à sable peut être d'une classe du bac à sable. Les conversions de types peuvent alors ne pas fonctionner. En effet, avec java, les classes ne sont pas identifiées uniquement par leurs noms, mais également par le chargeur de classe les ayant installées.

Par exemple, l'application propose une classe `Utilisateur`. Un traitement est lancé dans le bac à sable retournant une instance `Utilisateur`. Le bac à sable va recharger la classe `Utilisateur`, version bac à sable. Une instance de cette version de la classe sera renvoyer à l'application lors de la sortie du bac à sable. Si l'application désire convertir l'objet retourné en `Utilisateur`, une exception est générée.

```
Utilisateur u=(Utilisateur)doUnprivileged(new ReturnSandboxUtilisateur(),null);
```

En effet, la classe `Utilisateur` en dehors du bac à sable n'est pas la même que la classe `Utilisateur` du bac à sable. S'il est nécessaire de retourner une instance `Utilisateur`, le package correspondant doit être déclaré dans la zone orange, afin de ne pas être rechargé par le chargeur de classe.

```
Sandbox.doUnprivileged(new MySandbox(),null,new Package[]{Utilisateur.class.getPackage()});
```

Vous pouvez télécharger cet outil sur mon site www.philippe.prados.name.

6. GARDIEN

Parfois, il est nécessaire de retourner une instance privilégiée, qui sera utilisée plus tard par l'application. Comment être certains que le code qui utilisera l'instance obtenue à l'aide de privilèges ait le droit de le faire ? Pour résoudre cela, java propose la classe `GuardedObject` et l'interface `Guard`. Une instance `GuardedObject` mémorise un objet et un privilège.

Imaginez la classe suivante qui propose un service retournant un flux vers le fichier `/readme.txt` :

```

public class SampleGuarded
{
    public static GuardedObject getGuarded()
    {
        return (GuardedObject)AccessController.doPrivileged(new PrivilegedAction()
        {
            public Object run()
            {
                FileInputStream f;
                try
                {
                    f = new FileInputStream("/readme.txt");
                    FilePermission p = new FilePermission("/readme.txt", "read");
                    return new GuardedObject(f, p);
                }
            }
        });
    }
}

```

```

        catch (FileNotFoundException e)
        {
            e.printStackTrace();
            return null;
        }
    });
}
}

```

La méthode `getGuarded()` demande des privilèges pour pouvoir accéder au fichier `/readme.txt`. Cet objet est placé dans une instance `GuardedObject`, et le privilège nécessaire à la manipulation de l'instance y est associé. La classe `SampleGuarded` doit être placée dans la zone orange du bac à sable. C'est-à-dire que la classe peut être manipulée par une classe du bac à sable, mais qu'elle ne sera pas rechargée avec moins de privilèges. Elle peut alors bénéficier des privilèges standard de l'application, et lire par exemple, le fichier `readme.txt`.

L'instance retournée est manipulable par une classe du bac à sable. Elle peut être mémorisée dans un conteneur, transférée à d'autres classes, envoyée à une classe orange si nécessaire. Mais, il n'est pas possible à une classe du bac à sable d'obtenir un pointeur vers l'instance stream protégée.

```

public class GuardedSandBox implements UnprivilegedAction
{
    public Object run(Map params)
    {
        GuardedObject obj=SampleGuarded.getGuarded();
        obj.getObject(); // Exception !
        return null;
    }
}

```

Cela permet de faire transiter des objets sensibles à travers le bac à sable, sans sacrifier la sécurité. Il faut faire très attentions aux objets récupérés dans les zones critiques. Ils ne doivent pas pouvoir être manipulés sans privilège. Parfois, les méthodes sensibles ne sont accessibles que par d'autres classes du même package. Ainsi, il n'est pas nécessaire d'utiliser un `GuardedObject`. Mais, cela suppose qu'il est impossible d'ajouter une classe dans le même package, afin de bénéficier artificiellement des accès. Ce n'est pas garanti ! Pour interdire la création de classe dans un package, il faut que toutes les classes du package soient dans une archive, et que celle-ci soit déclarée scellé. <http://java.sun.com/j2se/1.4.2/docs/guide/extensions/spec.html#sealing>.

En faisant attentions aux classes et méthodes disponibles dans les packages proposés pour la zone orange, il est facile de limiter les possibilités de nuisance d'un code non sûr.

Les dernières spécifications de Java permettent de vérifier les privilèges associés à un utilisateur particulier. Pour cela, il doit s'être identifié à l'aide d'un driver JAAS. Lorsqu'une normalisation sera proposée pour les serveurs d'applications, il sera possible de proposer des API différentes suivant les utilisateurs, et de le vérifier à l'exécution.

Philippe PRADOS