PIGA-Cloud : une protection obligatoire des environnements d'informatique en nuage

Zaina Afoulki*, Aline Bousquet*, Jérémy Briffaut*, Laurent Clévy**,
Jonathan Rouzaud-Cornabas***, Christian Toinard*, Benjamin Venelle**

*ENSI de Bourges – LIFO, 88 bd Lahitolle, 18020 Bourges cedex, France
{zaina.afoulki, aline.bousquet, jeremy.briffaut, christian.toinard}@ensi-bourges.fr
**Alcatel-Lucent Bell Labs, 7 route de Villejust, 91620 Nozay, France
{laurent.clevy, benjamin.venelle}@alcatel-lucent.fr
***LIP ENS Lyon 46 allée d'Italie, F-69364 Lyon cedex 07, France
jonathan.rouzaud-cornabas@inria.fr

Résumé. La garantie de propriétés de sécurité nécessite la mise en place d'un contrôle d'accès obligatoire (Mandatory Access Control). Les Clouds supportent mal ces mécanismes sans offrir une protection pour tous les niveaux (hôte, invité, application, réseau). PIGA-Cloud garantit des propriétés avancées pour des flux indirects et des combinaisons variées de flux et protège en profondeur en contrôlant les flux entre les systèmes d'exploitation invités et l'hôte, les flux internes à un invité mais aussi les flux entre objets Java et les flux réseau. L'article montre comment PIGAiser des environnements aussi divers que des machines Unix, des applications Java et des Clouds. Il étend les politiques d'accès directs SELinux et sVirt à une machine virtuelle Java pour au final protéger de façon avancée des Clouds de type IaaS, PaaS ou SaaS. L'approche simplifie l'administration des politiques directes en empêchant les millions de vulnérabilités résiduelles. Ce travail est partiellement supporté par le projet européen Seed4C.

1 Introduction

Pour garantir des propriétés de confidentialité et d'intégrité, il est nécessaire d'avoir d'une part des mécanismes cryptographiques pour chiffrer et signer les informations et d'autre part des protections qui contrôlent les accès à l'information. Les deux sont indispensables. Ici nous ne traitons que du contrôle d'accès et de la seule méthode permettant des garanties à savoir une protection du type MAC qui met les politiques de protection hors de portée des usagers finaux et même de l'administrateur système (root) de la machine. L'approche développée dans PIGA-OS (J. Briffaut, 2011), durant le défi sécurité Sec&Si de l'Agence Nationale de la Recherche, a montré qu'il est nécessaire d'avoir une protection en profondeur c'est-à-dire couvrant tous les niveaux d'un système (interface graphique, processus, application et réseau) afin de contrôler efficacement les flux d'informations et de s'adapter dynamiquement aux différents domaines d'usages. En effet, la sécurité d'un système repose sur la sécurité de chacune de ses couches : une fragilité à un seul niveau compromet la garantie visée. Cependant, offrir une protection en

profondeur pour des systèmes virtuels est plus complexe car il y a un plus grand nombre de niveaux à contrôler, à savoir tous les niveaux du système hôte mais aussi tous ceux des invités. Même si nous utilisons l'approche développée dans PIGA-OS, cet article va beaucoup plus loin puisque nous montrons la généricité de l'approche PIGA en protégeant non seulement des Clouds du type IaaS reposant sur des systèmes d'exploitation virtuels, mais aussi des Clouds du type PaaS et SaaS puisque nous sommes capables de contrôler les flux à l'intérieur d'une application ce qui n'était pas possible dans PIGA-OS. De plus, mettre en place des protections obligatoires à tous les niveaux introduit une complexité de configuration importante. Nous montrons que PIGA simplifie l'effort d'administration puisque quelques règles PIGA suffisent à protéger contre les millions de vulnérabilités potentielles que présentent les multiples et complexes politiques obligatoires directes. Aussi loin que nous ayons investigué, nous n'avons pas trouvé d'approches aussi flexibles et efficaces s'adaptant à de nombreux environnements tout en simplifiant et renforçant la sécurité. Notre approche propose ainsi une vraie solution de protection obligatoire en profondeur et de bout en bout puisque nous pouvons protéger ainsi tous les noeuds d'un Cloud ainsi que les accès réseau entre ces noeuds.

Bien que nous ayons intégré l'approche PIGA à l'environnement de Cloud libre OpenNebula, l'approche est adaptable à d'autres infrastructures de Cloud et indépendante de la technique de virtualisation utilisée. Tout d'abord, nous montrons que nous pouvons déployer des images virtuelles et les faire migrer en leur associant des politiques directes cohérentes. Nous rappelons l'approche PIGA utilisée pour améliorer la sécurité des systèmes SELinux. Nous proposons aussi une nouvelle méthode appelée Security Enhanced Java (SEJava), en référence à Security Enhanced Linux (SELinux), grâce à laquelle nous pouvons contrôler les flux entre objets Java. Grâce au moniteur de référence partagé PIGA-Shared, nous pouvons non seulement autoriser certains flux entre deux invités tout en interdisant d'autres flux mais aussi garantir des propriétés avancées pour les objets d'une application Java.

L'approche PIGA-Cloud est extensible et simple d'usage. Tout d'abord, elle supporte la définition de patrons de protection permettant de couvrir un très large ensemble de besoins de protection. Ensuite, l'administrateur instancie des politiques PIGA grâce à ces patrons en réutilisant les contextes disponibles dans les politiques d'accès directs existantes à savoir SELinux, sVirt ou SEJava . Enfin, il suffit de quelques règles PIGA pour contrôler des millions de vulnérabilités potentielles garantissant ainsi une réelle sécurité en profondeur.

2 Principe de PIGA-Cloud

2.1 Fonctionnement général

L'objectif de PIGA-Cloud est de fournir une protection obligatoire en profondeur d'une infrastructure de type Cloud via trois mécanismes de protection :

- Contrôle d'un système d'exploitation : protection des ressources au sein d'un système d'exploitation qu'il s'agisse de celui de l'hôte ou de celui d'une machine virtuelle (Virtual Machine)
- Contrôle des VMs : protection du système vis-à-vis des VMs, et des VMs entre elles
- Contrôle d'une application Java : protection au sein d'une machine virtuelle java (JVM)

L'infrastructure PIGA-Cloud présentée dans la figure 1 utilise la notion de *contexte de sécurité* pour identifier les différentes entités/ressources (processus, fichier, machine virtuelle,

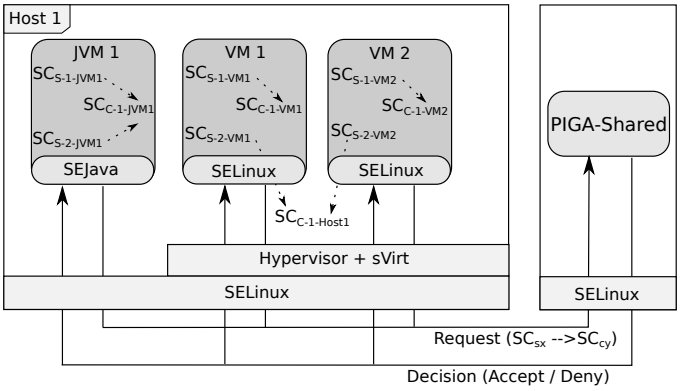

FIG. 1 – *PIGA-Cloud.*

objet Java, ...) et le principe de moniteur de référence partagé (PIGA-Shared) afin de contrôler les opérations entre ces contextes de sécurité (SELinux, sVirt et SEJava). Le principe de SELinux, sVirt et SEJava est de contrôler un flux direct pour un contexte de sécurité source qui accède à un contexte de sécurité cible $sc_s \rightarrow sc_c$. Un accès en écriture correspond à un flux direct $sc_s > sc_c$ et un accès en lecture à un flux direct $sc_s < sc_c$. PIGA permet non seulement de contrôler les flux indirects $sc_{s1} >> sc_{s2}$, correspondant à une fermeture causale du type $sc_{s1} > sc_c > sc_{s2}$, mais aussi d'écrire des propriétés contrôlant plusieurs flux tant directs qu'indirects. C'est bien la capacité d'écrire de telles propriétés avancées qui facilite l'administration et permet de renforcer considérablement la sécurité. Pour appliquer PIGA, on doit disposer de la notion de contexte de sécurité et de politique de contrôle des accès directs entre ces contextes.

L'utilisation de SELinux permet de protéger un système d'exploitation. Il peut s'agir du système d'exploitation de l'hôte ou du système d'exploitation d'une machine virtuelle invitée. SELinux utilise des politiques obligatoires directes minimisant les privilèges de processus vis à vis des ressources du système. L'apport de PIGA-Shared est de renforcer les politiques SELinux en garantissant des propriétés avancées.

L'utilisation de SELinux/sVirt permet de protéger les VMs et ce quelque soit l'hyperviseur utilisé (KVM/Qemu, Xen, Vmware, ...) en disposant de politique d'accès directs. Afin de supporter la migration des images virtuelles, PIGA-Cloud associe un contexte de sécurité unique à chaque machine virtuelle et maintient la cohérence des contextes dans l'infrastructure. Cependant, sVirt contrôle uniquement les flux d'informations directs entre l'image et l'hôte grâce à l'utilisation de SELinux. Ainsi, PIGA-Shared étend ce contrôle en permettant de définir des propriétés avancées qui pallient aux problèmes de SELinux et sVirt. Ainsi, nous montrerons qu'une propriété PIGA peut généralement empêcher des millions d'activités illégitimes qui sont permises par SELinux/sVirt.

SEJava permet de protéger les applications Java s'exécutant dans le Cloud. SEJava permet d'ajouter la notion de contexte à chacun des objets Java et ainsi de contrôler les flux au sein de la JVM. Tout comme SELinux, SEJava ne contrôle que les flux directs entre les objets Java. Il est cependant possible d'écrire des règles PIGA s'appliquant aux contextes SEJava et ainsi

proposer une protection renforcée de la JVM.

PIGA-Shared est le moniteur de référence garantissant des propriétés de sécurité avancées pour tout type de contextes de sécurité (SELinux, sVirt, SEJava). PIGA-Shared reçoit des requêtes pour les différents accès $sc_s \rightarrow sc_c$ en provenance des différents moniteurs de référence directs (SELinux, sVirt, SEJava) et renvoie une décision (autorisé/interdit) qui dépend du respect des propriétés PIGA requises. PIGA ne remplace donc pas les moniteurs de référence directs mais vient les renforcer.

2.2 Modélisation de l'architecture

2.2.1 Contexte de sécurité

Un contexte de sécurité, noté sc, permet d'identifier une ressource/entité dans PIGA-Cloud en lui associant un label. Suivant le type de protection, un contexte peut représenter :

- une machine virtuelle : un processus KVM et les fichiers correspondants aux disques
- une ressource d'un système : un processus ou un fichier
- une ressource dans la JVM : un objet Java

PIGA-Cloud utilise la syntaxe SELinux pour représenter les contextes de sécurité. Un contexte est un ensemble d'attributs formant un label associé à la ressource/entité. Cette notation permet de s'abstraire du nom/placement/chemin des ressources/entités.

2.2.2 Activité directe

Une activité directe, notée $sc_s \rightarrow sc_c$, représente une opération réalisée par le contexte de sécurité source sc_s sur le contexte cible sc_c. Une opération peut correspondre à un appel système (lecture, écriture, exécution, ...) ou à une action au sein de la JVM (appel de méthode, accès à un attribut). Une activité directe correspond à un flux $sc_s > sc_c$ ou $sc_s < sc_c$ selon que l'activité est de type écriture ou lecture.

2.2.3 Activité avancée

Une activité avancée est une combinaison d'activités directes. Elle peut correspondre à une séquence, notée $sc_1 \Rightarrow sc_n$, qui représente une séquence transitive d'interactions $sc_1 \rightarrow sc_2 ... sc_{n-1} \rightarrow sc_n$. Une telle séquence peut correspondre à un flux d'information indirect $sc_1 >> sc_n$ correspondant à une fermeture transitive d'actions causalement liées (Briffaut, 2007). En pratique, il s'agit d'une séquence d'activités lecture/écriture, aboutissant à l'échange ou la fuite d'information. Par exemple, un processus d'une machine virtuelle sc_{s-vm1} peut échanger de l'information avec un processus d'une autre machine sc_{s-vm2} en utilisant une ressource partagée $sc_{c-host1}$.

3 Protection obligatoire directe

Cette section décrit les différents mécanismes mis en place pour contrôler les activités directes dans les trois types de protection définis. Pour chaque type, nous décrivons la notion de contexte de sécurité et le modèle de politique de protection directe utilisée. Le modèle de protection obligatoire avancée PIGA est décrit ensuite dans la section 4.

3.1 Protection à l'intérieur d'un système d'exploitation

La protection des ressources à l'intérieur d'un système d'exploitation permet de contrôler les activités directes à l'intérieur de l'hôte et aussi à l'intérieur d'une machine virtuelle invitée. Elle utilise un mécanisme de contrôle obligatoire direct, SELinux dans le cas de PIGA-Cloud. Dans SELinux, chaque processus/fichier est identifié par un contexte de sécurité. Une politique SELinux définit l'ensemble des interactions directes autorisées, une interaction correspondant à une opération (un appel système) entre deux contextes. Le noyau Linux contrôle les interactions et garantit le respect de la politique SELinux. Toute tentative de réalisation d'une interaction non définie dans la politique est interdite.

3.1.1 Contextes de sécurité

Un contexte de sécurité est un label associé à chaque ressource du système (processus/fichier) constitué de trois attributs : une identité, un rôle et un type. Dans SELinux, un utilisateur est associé à une identité qui lui permet d'accéder à des rôles. Chaque rôle autorise l'accès à un ensemble de types. Les règles de contrôle d'accès définissent l'ensemble des opérations (appels système) autorisées entre les types. Ainsi la notion de contexte permet de s'abstraire du nom et de l'emplacement des ressources. Le listing suivant représente l'association entre fichiers/processus et contextes de sécurité :

```
/etc/shadow : system_u:object_r:shadow_t
/usr/bin/apache : system_u:object_r:apache_exec_t
processus apache : system_u:system_r:httpd_t
```

Dans cet exemple, l'identité `system_u` correspond aux services ou ressources du système (non liés à un utilisateur réel). Le rôle `object_r` est associé aux ressources passives qui ne peuvent pas incarner un rôle. Le rôle `system_r` est associé aux processus de type service système. Le type `shadow_t` caractérise le fichier contenant les mots de passe hachés, `apache_exec_t` correspond aux binaires apache et `httpd_t` aux processus de type serveur Web.

3.1.2 Protection obligatoire directe

Une politique de contrôle d'accès SELinux permet de contrôler les activités directes à l'intérieur d'un système d'exploitation (ex : le système d'exploitation d'une VM). Cette politique définit l'ensemble des opérations autorisées entre tous les contextes de sécurité, ce qui représente plusieurs millions de règles. L'objectif de SELinux est d'appliquer le *principe du moindre privilège* pour chaque processus. Ainsi, un type, appelé aussi domaine, est associé à chaque catégorie de processus (serveur Web, Mail, ...). La politique SELinux ne définit que l'ensemble des opérations directes nécessaires à l'exécution de ce processus.

Le listing suivant représente un exemple simplifié de politique SELinux :

```
#définition d'une identité et association des rôles
user root roles { sysadm_r staff_r user_r }
user system_u roles { system_r }
#définition des types
type passwd_t;
type httpd_t;
[...]
#association des rôles aux types
```

```
role sysadm_r types passwd_t;
role system_r types httpd_t;
[...]
#définition des règles de contrôle entre types
allow passwd_t shadow_t:file { open getattr read }
allow httpd_t apache_exec_t:file { open getattr read execute }
[...]
```

Cette politique autorise donc deux actions directes :

```
root:sysadm_r:passwd_t -- READ --> system_u:object_r:shadow_t
system_u:system_r:httpd_t  -- EXECUTE --> system_u:object_r:apache_exec_t
```

La première action permet à un processus de l'identité `root` correspondant à l'exécution du binaire `passwd` de lire le fichier `/etc/shadow`. La seconde action permet au processus exécutant `apache` d'exécuter une nouvelle instance d'`apache`.

En définissant ainsi l'ensemble des opérations autorisées pour chaque processus du système de manière fine (de l'ordre de l'appel système), SELinux permet de confiner chaque processus dans un domaine d'exécution avec un ensemble de droits minimal. En cas de prise de contrôle d'un processus, l'attaquant ne pourra pas obtenir de droits supplémentaires réduisant ainsi l'impact de l'attaque. Ainsi, même en cas de prise de contrôle d'un serveur Web s'exécutant avec l'identité `root`, l'attaquant ne pourra jamais lire dans le fichier `/etc/shadow`.

3.2 Protection des machines virtuelles

Une machine virtuelle s'exécutant sur un hyperviseur correspond en fait à un processus (une instance de Qemu/KVM) et un ensemble de fichiers (fichiers de configuration et les disques virtuels). La seconde couche de protection de PIGA-Cloud vise à isoler les machines virtuelles du système en utilisant sVirt. sVirt est une extension de SELinux utilisant la notion de catégorie pour contrôler les interactions directes entre images virtuelles, afin :

- d'isoler la machine virtuelle du système : en cas d'attaque sur une VM, l'attaquant obtient un accès restreint au système hôte
- d'isoler les machines entre elles : une machine virtuelle ne peut pas interagir avec les processus/fichiers d'une autre machine virtuelle

Le problème actuel de sVirt est qu'il n'apporte pas de support de la migration des catégories entre plusieurs hyperviseurs. Deux machines virtuelles sur deux hôtes différents peuvent avoir la même catégorie et donc interagir entre elles via des ressources partagées.

Un des avantages de PIGA-Cloud est d'offrir un service de désignation des images qui fournit des contextes de sécurité uniques et cohérents pour des VMs pouvant migrer sur différents hyperviseurs. Ainsi, une VM peut migrer d'un hôte source H1 vers un hôte destination H2, en conservant le contexte de protection requis.

3.2.1 Contextes de sécurité

PIGA-Cloud utilise le service de labelisation statique disponible avec sVirt afin d'assigner un contexte unique non seulement au processus qui exécute l'image mais aussi aux fichiers qui contiennent le système de fichiers de l'image.

Dans l'implémentation actuelle de sVirt, un contexte est composé d'un label SELinux (`identité:rôle:type`) et d'une catégorie (c_x : c_y). La partie SELinux permet de pro-

téger le système en cas de corruption de la machine virtuelle à l'aide d'une politique SELinux. Le listing suivant présente quelques labels SELinux proposés par sVirt :

```
# Types sujets
system_u:system_r:svirt_t : processus KVM
system_u:system_r:virtd_t : daemon libvirtd
# Types objets
system_u:object_r:svirt_image_t : fichier image en lecture/écriture
system_u:object_r:virt_image_t : fichier image en lecture seule (VM arrêtée)
system_u:object_r:virt_content_t : fichier image en lecture seule (pour les
    disques partagés)
system_u:object_r:svirt_etc_t : fichiers XML de configuration des images
```

Les catégories permettent quant à elles d'isoler les machines virtuelles entre elles : un processus source peut interagir avec une entité cible si et seulement si l'ensemble des catégories de la cible est un sous-ensemble des catégories de le source. Par exemple, un processus labelisé par c_0, c_5 n'aura accès qu'à des entités labelisées avec les catégories c_0 ou c_5, avec le couple c_0, c_5 ou sans catégorie. On obtient alors des contextes de la forme suivante :

```
# Types sujets
system_u:system_r:svirt_t:s0:c1,c5 : processus KVM VM1
system_u:system_r:svirt_t:s0:c2,c10 : processus KVM VM2
# Types objets
system_u:object_r:svirt_image_t:s0:c1,c5 : fichier image VM1
system_u:object_r:svirt_image_t:s0:c2,c10 : fichier image VM2
system_u:object_r:svirt_image_t:s0 : fichier image partagé
```

Un mécanisme de désignation a ainsi été intégré aux pilotes de OpenNebula via la librairie libvirt. Dans l'intégration proposée, les pilotes d'OpenNebula sont responsables du choix des contextes et de leur écriture dans les fichiers de libvirt. Ainsi, lorsqu'une image est suspendue ou migrée, le contexte choisi sur le premier noeud est préservé sur les noeuds ultérieurs. Le service de désignation fournit un fichier de configuration XML pour sVirt dont l'extrait du listing 1 montre l'association du contexte de sécurité `system_u:system_r:svirt_t:s0:c1,c8` et `system_u:object_r:svirt_image_t:s0:c1,c8` pour le processus et les fichiers de la machine virtuelle.

```
<seclabel type='static' model='selinux'>
  <label>system_u:system_r:svirt_t:s0:c1,c8 </label>
  <imagelabel>system_u:object_r:svirt_image_t:s0:c1,c8 </imagelabel>
</seclabel>
```

Listing 1 – *Extrait de la configuration du déploiement d'une machine virtuelle.*

Pour assurer une protection efficace, l'attribution des contextes de sécurité aux images ne doit pas dépendre de l'utilisateur ou de l'administrateur du Cloud. Pour cette raison, le système de désignation gère les contextes de manière totalement transparente pour l'utilisateur.

Afin de pouvoir proposer des contextes cohérents au niveau de l'ensemble de l'infrastructure, c'est au gestionnaire du Cloud, ici OpenNebula, de déterminer quels sont les contextes à attribuer aux images.

Pour cela, le système de désignation utilise une base de données qui peut soit être une base dédiée, soit être intégrée à OpenNebula (ou à un autre système). Cette base de données permet au service de désignation de savoir quels sont les contextes déjà attribués et à quelles VMs ils sont associés. De cette manière, lors du lancement d'une nouvelle VM, le service de désignation peut interroger la base de données et attribuer un contexte non-utilisé à la VM. Une

fois ce contexte choisi, les pilotes modifiés d'OpenNebula chargés de déployer la VM peuvent ajouter le contexte aux fichiers images et au processus KVM lors du déploiement de cette VM.

Le listing suivant montre le résultat des commandes Unix ps et ls après le déploiement d'une machine virtuelle, et permet de vérifier que les contextes sont correctement attribués aux processus et aux images :

```
[root@node1 ~]# ps auxZ | grep one-186
system_u:system_r:svirt_t:s0:c1,c8 oneadmin 7901 17.3  [...]
[root@node1 ~]# ls --scontext /one/var/186/images/disk.0
system_u:object_r:virt_image_t:s0:c1,c8 /one/var/186/images/disk.0
```

Afin que la protection soit totale, elle doit être valable tout au long de la vie de la VM, et notamment lors d'une migration sur un nouvel hôte. Il est important qu'une migration prenant en compte le contexte de sécurité de la VM puisse s'effectuer sans intervention de l'utilisateur. Le maintien de la cohérence des contextes est donc pris en charge par le service de désignation. En effet, lors d'une migration de VM, le service de désignation applique les contextes (qu'il peut retrouver dans sa base de données) sur le nouvel hôte.

On obtient alors le résultat suivant pour les commandes ps et ls :

```
[root@node2 ~]# ps auxZ | grep one-186
system_u:system_r:svirt_t:s0:c1,c8 oneadmin 6110 17.3 [...]
[root@node2 ~]# ls --scontext /one/var/186/images/disk.0
system_u:object_r:virt_image_t:s0:c1,c8 /one/var/186/images/disk.0
```

3.2.2 Protection obligatoire directe

L'avantage d'un contrôle d'accès comme celui fournit par SELinux est de protéger les accès directs des processus aux ressources.

Nos expériences montrent qu'il est ainsi possible d'avoir une protection des processus qui exécutent les machines virtuelles. Ainsi, nous sommes capables d'exécuter différentes VMs sur le même noeud en les isolant les unes des autres, mais également de partager un même fichier d'image entre plusieurs VMs s'exécutant sur ce noeud.

Par exemple, lorsqu'un processus avec un label inadéquat essaie d'accéder à une ressource qui ne lui appartient pas, SElinux l'en empêche comme présenté dans le listing 2. En pratique, dans les traces d'OpenNebula, nous observons que la VM n'a pas pu démarrer (c.f. listing 3). C'est bien le comportement souhaité puisqu'il s'agirait d'un processus malveillant qui tenterait de récupérer les informations présentes dans une image ou de modifier cette image. Le mécanisme permet donc bien à la fois de protéger contre des violations de confidentialité et d'intégrité des fichiers image.

Ainsi, SELinux empêche la VM1 (`system_u:system_r:svirt_t:s0:c1,c3`) d'accéder au fichier image de la VM2 (`system_u:object_r:svirt_image_t:s0:c1,c2`).

```
type=AVC msg=audit(1321711900.859:169931): avc:  denied { read write }
 for  pid=1796 comm="kvm" name="disk.0" dev=dm-5 ino=7471710
 scontext=system_u:system_r:svirt_t:s0:c1,c3
 tcontext=system_u:object_r:virt_image_t:s0:c1,c2 tclass=file
```

Listing 2 – *SELinux empêche les accès directs malveillants.*

```
error: Failed to create domain from
 /one/var/127/images/deployment.0
```

```
error: internal error process exited while connecting to
monitor: kvm: -drive file=/one/var/127/images/disk.0
 ,if=none,id=drive-ide0-0-0,format=raw: could not open
 disk image /one/var/127/images/disk.0:  Permission denied
```

Listing 3 – *OpenNebula ne peut lancer la VM malveillante.*

Ainsi, les différentes expérimentations montrent que notre service de protection permet d'associer des labels uniques aux différents machines virtuelles s'exécutant dans le Cloud. Malgré les évènements de migration ou de suspension, les contextes de sécurité sont préservés.

Grâce à SELinux, PIGA-Cloud contrôle les flux d'information directs entre les différentes VMs. De cette façon, il est possible 1) d'empêcher les flux directs entre une VM et des fichiers images, 2) d'autoriser deux VMs à partager le même fichier d'image et 3) de transmettre la requête SELinux au moniteur de référence partagé PIGA-Shared afin d'avoir une meilleure protection. La section 4 décrit ce dernier point. Elle explique comment le moniteur partagé PIGA-Shared permet de garantir des propriétés de sécurité avancées à l'intérieur d'une VM mais aussi entre deux VMs.

3.3 Protection à l'intérieur d'une JVM

Les deux premiers types de protection de PIGA-Cloud permettent de protéger l'hôte, les machines virtuelles et les processus au sein des machines virtuelles. Cependant, des applications Java s'exécutent soit directement sur un hôte soit dans une VM afin de fournir une solution de type PaaS ou SaaS.

Les solutions de contrôle d'accès obligatoire actuelles ne contrôlent que les opérations entre processus et sont incapables d'observer le comportement interne d'un processus comme une JVM. Cependant, pour pouvoir disposer d'une protection en profondeur du Cloud, il est nécessaire d'être capable de contrôler de manière obligatoire les applications s'exécutant dans le Cloud.

Security Enhanced Java (SEJava) est un modèle de contrôle d'accès obligatoire pour les activités directes à l'intérieur de la JVM. Comme tout modèle MAC, SEJava a besoin d'un moniteur de référence pour intercepter l'ensemble des opérations qui ont lieu. Dans le cas de SEJava, les sources et les cibles à contrôler sont les objets Java. Ils interagissent entre eux grâce aux appels de méthodes et aux accès aux attributs correspondant aux activités directes à contrôler.

SEJava utilise l'interface JVMTI (Java Virtual Machine Tool Interface, l'interface standard d'instrumentation de la JVM) afin d'intercepter les interactions entre objets. L'agent SEJava JVMTI contrôle ainsi les opérations sur les objets. Ce moniteur de référence vérifie que les opérations satisfont la politique directe requise entre les objets.

3.3.1 Contextes de sécurité

Dans le cadre de SEJava, un contexte de sécurité correspond au type d'un objet, i.e. le nom de la classe associée à l'objet. Par exemple, $Ljava/io/File;$ est la signature de la classe $java.io.File$.

3.3.2 Protection obligatoire directe

Afin d'être capable d'exprimer les activités directes autorisées, SEJava a besoin de définir les permissions pouvant se produire entre deux contextes de sécurité. Pour cela, SEJava reprend les types d'opérations définis dans les spécifications Java correspondant à :

- des appels de méthodes : invocations
- des accès aux attributs : lecture ou écriture

```
package se.java;  // Java's package to which the usecase belong
public class UseCase {
    public static void main(String args[]) {
        Guest guest = new Guest();  // An object with restricted privileges
        Admin admin = new Admin();  // An object owning a secret
        Storage f = new Storage();  // An ordinary Java object

        // The Admin's "secret" is copied to a shared data
        f.Write(admin.GetSecret());

        // The content of this shared data is then copied as Guest's secret
        guest.SetSecret(f.Read());

        // From this execution point:
        //  An instance of Guest knows the "secret" of an Admin's instance
        //  The confidentiality of Admin's secret field is thus violated.
        }
}
```

Listing 4 – *Code de l'exemple.*

Le listing 4 propose un exemple simple d'application utilisant trois objets Java :

- *Guest* correspond à un utilisateur non-privilégié. La classe possède un attribut privé et les méthodes get et set correspondantes.
- *Admin* correspond à un utilisateur privilégié. Cette classe hérite de *Guest*
- *Storage* est un objet Java implémentant des méthodes $read$ et $write$

```
# [allow/deny] signature --{ permission }--> signature
# Allows Usecase to create new instances of Guest, Admin and Storage
allow Usecase --{ main invoke <init> }--> Admin
allow Usecase --{ main invoke <init> }--> Guest
allow Usecase --{ main invoke <init> }--> Storage
# Main has to be able to invoke methods from Storage
allow Usecase --{ main invoke Read }--> Storage
allow Usecase --{ main invoke Write }--> Storage
# Main is allowed to invoke GetSecret on Admin
# but not SetSecret on Guest (see commented rule)
allow Usecase --{ main invoke GetSecret }--> Admin
allow Usecase --{ main invoke SetSecret }--> Guest
```

Listing 5 – *Extrait de la politique directe SEJava.*

Le listing 5 contient un extrait de la politique SEJava associée à cette application. Cette politique permet de limiter les opérations pouvant être effectuées par l'application. Ici, la politique autorise trois types d'actions :

- les trois premières règles autorisent la fonction *main()* de la classe *Usecase* à créer des instances des classes *Admin*, *Guest* et *Storage*

– les deux règles suivantes permettent à la fonction *main()* de lire et d'écrire dans les instances de *Storage*
– les deux dernières règles autorisent *main()* à lire le secret d'*Admin* et à écrire celui de *Guest*

```
# decision (signature, id) --{ permission }--> (signature, id)
ALLOW (Usecase, 552) --{ main invoke <init> }--> (Guest, 480)
ALLOW (Guest, 480) --{ <init> invoke <init> }--> (Guest, 480)
ALLOW (Usecase, 552) --{ main invoke <init> }--> (Admin, 056)
ALLOW (Admin, 056) --{ <init> invoke <init> }--> (Admin, 056)
ALLOW (Admin, 056) --{ <init> invoke <init> }--> (Admin, 056)
ALLOW (Usecase, 552) --{ main invoke <init> }--> (Storage, 912)
ALLOW (Storage, 912) --{ <init> invoke <init> }--> (Storage, 912)

ALLOW (Usecase, 552) --{ main invoke GetSecret }--> (Admin, 056)
ALLOW (Usecase, 552) --{ main invoke write }--> (Storage, 912)

ALLOW (Usecase, 552) --{ main invoke read }--> (Storage, 912)
ALLOW (Usecase, 552) --{ main invoke SetSecret }--> (Guest, 480)
```

Listing 6 – *Décision de SEJava.*

Le listing 6 contient les traces de SEJava générées lors de l'exécution de cette application. Les appels aux constructeurs des objets sont visibles dans la première partie des traces. On observe que les trois objets peuvent bien être instanciés, ce qui correspond aux trois premières règles du listing 5. De même, l'écriture du secret d'*Admin* dans *Storage* et la lecture de *Storage* par *Guest* sont également autorisées.

On peut cependant remarquer l'existence d'un flux indirect entre *Admin* et *Guest*, passant par l'objet *Storage*. Ce flux représente une violation de la confidentialité du secret d'*Admin* et devrait donc être interdit. Cependant, un tel flux indirect ne peut pas être bloqué par SEJava qui ne traite que les interactions directes. Ce cas peut alors être traité par PIGA-Shared, comme décrit dans la section suivante.

SEJava permet en revanche de bloquer une exécution anormale du programme. En effet, tout comportement non prévu par la politique (par exemple, l'écriture du secret d'*Admin)* sera stoppé. Ce mécanisme permet de bloquer des failles existantes, comme par exemple, la faille CVE-2012-1723 [1]

4 Protection obligatoire avancée

Les solutions de contrôle d'accès obligatoire permettant le contrôle des activités directes sont complexes à configurer. De plus, ces solutions ne permettent pas de contrôler des activités avancées telles que les flux indirects ou les combinaisons d'activités. La configuration d'un système complet peut nécessiter des millions de règles. Il est alors très difficile de pouvoir garantir qu'une telle politique satisfait bien aux objectifs de sécurité requis tels que l'intégrité ou la confidentialité de certaines ressources. Ce problème est encore plus complexe dans le cadre d'un Cloud où il est nécessaire de prendre en compte la possible migration des machines/ressources/services et les ressources partagées.

1. http ://www.symantec.com/connect/blogs/examination-java-vulnerability-cve-2012-1723

L'approche PIGA, développée dans Briffaut (2007) et Rouzaud-Cornabas (2010), simplifie l'approche de protection obligatoire tout en couvrant un plus large ensemble de propriétés de sécurité.

PIGA réutilise les politiques de contrôle d'accès direct existantes, comme celles de SELinux, sVirt ou SEjava. Les politiques SELinux n'ont alors plus besoin d'être aussi précises. En effet, PIGA complète le contrôle en garantissant des propriétés avancées. De plus, PIGA permet non seulement de contrôler les flux indirects empruntant des contextes intermédiaires mais plus encore de contrôler tout type d'activités malveillantes incluant une combinaison de différentes activités directes et indirectes. En pratique, PIGA permet alors d'empêcher des millions de vulnérabilités potentielles que ne peut contrôler une politique obligatoire directe. Enfin, une politique PIGA nécessite en pratique un faible nombre d'instanciation de propriétés de sécurité. Par exemple, PIGA-OS (J. Briffaut, 2011) requiert seulement une dizaine de règles qui sont alors suffisantes pour renforcer considérablement la sécurité d'un système d'exploitation.

PIGA ne fait qu'ajouter des contrôles obligatoires à ceux existants via les différents moniteurs de référence qui traitent les flux directs. Ainsi, PIGA ne se substitue ni à SELinux, ni à SEJava mais renforce et simplifie la protection offerte par les protections obligatoires classiques. Ainsi grâce à l'approche PIGA, nous pouvons atteindre les objectifs visés, à savoir simplifier l'écriture de politiques obligatoires tout en garantissant un large ensemble de propriétés de sécurité.

4.1 Propriétés PIGA

Afin de faciliter la définition des propriétés de sécurité avancées, PIGA utilise des patrons (*templates*) de propriétés exprimés au moyen d'un langage dédié (SPL : *Security Property Language*). Ce langage permet de définir facilement de nouveaux patrons pour couvrir des objectifs de sécurité spécifiques. Cependant dans la pratique, les administrateurs de la sécurité se contentent d'instancier ces patrons en fournissant en paramètre les contextes de sécurité nécessaires. Les patrons peuvent donc être vus comme des propriétés de sécurité paramétrables tandis que les règles PIGA sont des instances de ces propriétés, définies via des valeurs particulières des paramètres.

Le premier patron présenté a pour but de garantir l'intégrité d'un ensemble sc_1 de contextes de sécurité cibles (fichiers) vis à vis d'un ensemble sc_2 de contextes de sécurité sources (processus). Cette propriété empêche toute activité de type écriture de sc_1 vers sc_2, que ces activités correspondent à des flux directs $>$ ou indirects $>>$:

```
define integrity( sc1 in SC_S, sc2 in SC_C )
[       ¬(sc1 > sc2)
        AND
        ¬(sc1 >> sc2) ];
```

Le second patron présenté a pour but de garantir la confidentialité d'un ensemble sc_2 de contextes de sécurité cibles vis à vis d'un ensemble sc_1 de contextes de sécurité sources. Cette propriété empêche tout flux d'information en lecture, allant des contextes sc_2 vers les contextes sc_1, que les flux soit directs $>$ ou indirects $>>$:

```
define confidentiality( sc1 in SC_S, sc2 in SC_C )
[       ¬(sc2 > sc1)
        AND
        ¬(sc2 >> sc1) ];
```

Le troisième patron empêche un processus de créer un fichier puis d'exécuter un interpréteur (ex. bash) qui enfin tenterait de lire le fichier créé. Ce patron empêche les exécutions interprétées depuis un ensemble de contextes sources $sc1$:

define dutiesseparationbash(sc_1 in SC)
[Foreach sc_2 in SC_C, Foreach sc_3 in SC,
¬ (($sc_1 \rightarrow_{write} sc_2$) -then-> ($sc_1 \rightarrow_{execute} sc_3$) -then-> ($sc_3 \rightarrow_{read} sc_2$))];

Une fois ces patrons définis, il suffit de les instancier avec les contextes sc_i nécessaires. En pratique, l'administrateur se contente donc de définir un jeu réduit de règles de sécurité PIGA en choisissant pour les paramètres des patrons les contextes de sécurité qu'il veut protéger.

4.2 Fonctionnement de PIGA

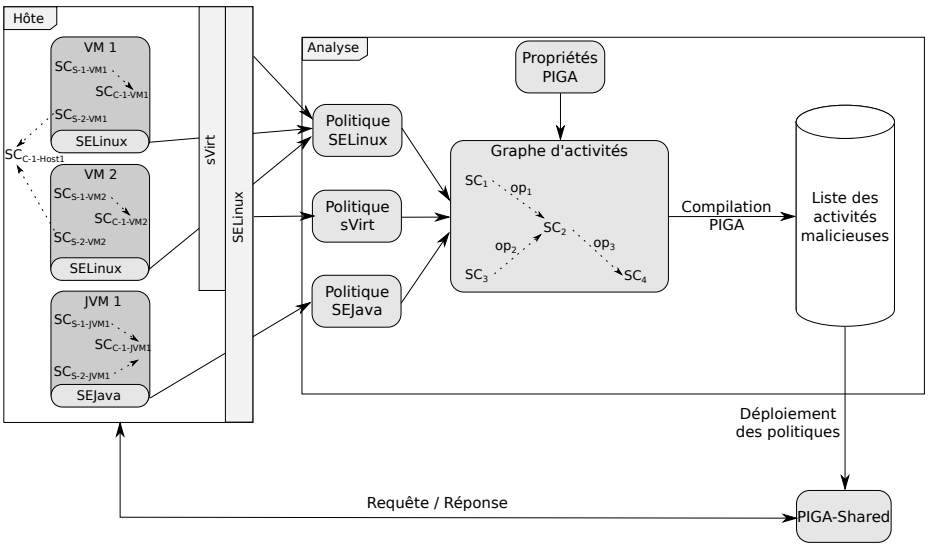

FIG. 2 – *PIGA-Cloud : analyse et déploiement des propriétés.*

Afin de contrôler des activités complexes, PIGA-Cloud utilise les politiques obligatoires pour les accès directs de différents types nécessaires pour protéger les différents composants du Cloud. La figure 2 représente le processus global d'analyse des propriétés de sécurité et de déploiement des politiques PIGA nécessaires au contrôle des activités avancées.

L'administrateur définit, dans le langage SPL, l'ensemble des propriétés PIGA à satisfaire pour les trois couches de protection. Des exemples de propriétés sont donnés dans les sous sections suivantes. Afin d'analyser le respect de ces propriétés de sécurité, le compilateur PIGA prend en entrée l'ensemble des politiques obligatoires directes des différents hôtes/machines virtuelles/application Java. Le compilateur calcule les différents graphes d'activités où un noeud du graphe correspond à un contexte de sécurité et un arc à l'ensemble des opérations autorisées entre deux contextes. Un arc dans ce graphe correspond à une activité directe, un chemin à une activité indirecte (un cas particulier d'activité avancée) et une combinaison d'arcs et de chemins à une activité avancée.

A l'aide de ces graphes, le compilateur énumère l'ensemble des activités violant les propriétés souhaitées afin de construire une base d'activités malicieuses. Cette base est ensuite déployée sur le moniteur PIGA-Shared afin de contrôler les activités et d'empêcher les activités malicieuses pour toutes les couches du Cloud. Ce moniteur de référence s'exécute sur une machine dédiée. Il s'agit en fait d'un processus qui exécute du code Java qui peut lui même être protégé avec SELinux comme présenté dans la figure 1.

Durant le fonctionnement du système, PIGA-Shared reçoit des requêtes pour les activités directes provenant :

- des machines SELinux hôtes
- des VMs invitées (c-à-d que chaque VM peut inclure une protection obligatoire du type SELinux pour des machines virtuelles Linux ou SEWindows (D. Gros, 2012) pour des machines virtuelles Windows)
- des machines virtuelles Java

PIGA-Shared autorise une opération (un appel système ou une interaction entre objets Java) lorsque celle-ci ne permet la terminaison d'aucune des activités illicites pré-calculées. Dans le cas contraire, l'opération est refusée. Ainsi, PIGA-Shared garantit que les différents systèmes contrôlés ne violent aucune des propriétés requises.

L'un des principaux problèmes est de réaliser une communication efficace et sûre entre les différents moniteurs de référence, contrôlant uniquement les flux directs, et le moniteur de référence partagé PIGA-Shared. Pour cela, les moniteurs pour les flux directs (notamment SELinux, SEWindows ou SEJava) communiquent avec PIGA-Shared en utilisant des sockets TCP. Briffaut et al. (2011) détaille le mode TCP qui permet de communiquer avec le moniteur partagé PIGA-Shared présent sur une machine distante.

4.3 Protection avancée à l'intérieur d'un système d'exploitation

Le premier objectif de PIGA-Shared est de protéger les flux à l'intérieur d'un système d'exploitation qu'il s'agisse du système d'exploitation de l'hôte ou du système d'exploitation s'exécutant dans une VM. En instanciant des patrons de propriétés avec des contextes SELinux, il est possible de contrôler des activités complexes et ainsi d'assurer une protection avancée du système.

Le listing suivant contient un exemple d'instanciation de propriétés utilisées sur les systèmes hôte et les machines virtuelles :

```
integrity( $sc1:="user_u:user_r:user.*_t", $sc2:=".*:.*:.*_exec_t" );
integrity( $sc1:="user_u:user_r:user.*_t", $sc2:=".*:.*:.*etc_t" );

confidentiality( $sc1:="user_u:user_r:user.*_t", $sc2:=system_u:object_r:
    shadow_t );
confidentiality( $sc1:="user_u:user_r:user.*_t", $sc2:=system_u:object_r:
    memory_device_t );

dutiesseparationbash( "user_u:user_r:user.*_t" );
```

Les deux premières règles permettent de garantir l'intégrité des exécutables et des fichiers de configuration système vis-à-vis de l'utilisateur. Les deux règles suivantes garantissent la confidentialité du fichier `/etc/shadow` et du périphérique `/dev/mem` (qui permet un accès direct à la mémoire physique). Finalement, la dernière règle empêche un processus utilisateur de télécharger un script puis de lancer un interpréteur qui lise le script afin de l'exécuter.

Cette dernière propriété est très intéressante car elle n'empêche ni le téléchargement de fichier, ni le lancement d'interpréteur. Cette règle empêche seulement l'interprétation des scripts qui ont été téléchargés. Bien évidemment, il est possible de définir une propriété moins contraignante. Par exemple en fournissant comme paramètre uniquement le contexte du navigateur Web Firefox, ce qui limitera l'exécution des scripts téléchargés uniquement pour Firefox.

4.4 Protection avancée des machines virtuelles

Le deuxième objectif de PIGA-Cloud est de pouvoir contrôler les flux entre une VM et l'hôte, et entre VMs.

Dans le premier cas, il est nécessaire d'écrire des propriétés utilisant les contextes de sécurité liés aux machines virtuelles afin de contrôler les activités avancées entre le processus KVM exécutant la VM et l'hôte. La règle suivante permet d'empêcher un processus QEMU/KVM d'exécuter un binaire autre que lui-même, les librairies système ou ses librairies :

```
trustedpathexecution( $sc1:=system_u:system_r:svirt_t , $TPE:= { ".*:.*:.*
    qemu_exec_t",  ".*:.*:.*lib_t",  ".*:.*:.*lib_qemu_t" } )
```

Cette simple règle empêche un attaquant, en cas de prise de contrôle d'une VM, d'avoir accès aux binaires du système et de lancer, par exemple, un interpréteur de commande. D'autres règles, protégeant la confidentialité et l'intégrité de l'hôte, ont été définies.

Dans certains cas, il est nécessaire de partager des disques ou des ressources entre deux machines virtuelles. Pour contrôler les activités entre machines virtuelles, un nouvel élément a été rajouté aux contextes SELinux des images afin d'être capable d'identifier les différentes machines. On obtient alors des contextes de la forme `user_u:role_r:type_t:vm_i`, où `vm_i` est l'identifiant de la machine virtuelle. Ainsi, la règle suivante garantit la confidentialité des fichiers de $/etc$ présents dans une première image VM1 vis à vis des utilisateurs d'une seconde image VM2 :

```
confidentiality(user_u:user_r:user.*_t:vm_2, system_u:object_r:etc_t:vm_1);
```

Le listing suivant montre un exemple de séquence qui violerait la propriété précédente au travers d'un partage NFS entre les VMs :

```
system_u:object_r:etc_t:vm_1 -( { write } )-> system_u:object_r:nfs_t ;
    system_u:object_r:nfs_t -( { read } )-> system_u:object_r:user_t:vm_2
```

La deuxième opération de cette séquence sera donc interdite par PIGA.

De cette manière, PIGA-Shared est capable de différencier les contextes de sécurité des machines virtuelles et de l'hôte et donc de proposer des propriétés de sécurité avancées adaptées.

4.5 Protection avancée à l'intérieur d'une JVM

PIGA-Shared permet également de protéger plus efficacement la JVM en se basant sur les contextes générés par SEJava.

Dans l'exemple décrit à la section 3.3.2, une activité indirecte menant à une violation de la confidentialité du secret de l'objet `Admin` vis à vis de l'objet `Guest` a pu être calculée par PIGA. Cette activité étant composée de plusieurs interactions, il est impossible de la stopper en utilisant SEJava. Cependant, grâce à PIGA-Shared, on peut instancier une propriété de confidentialité pour bloquer ce flux :

```
confidentiality(system_u:java_r:guest_j, system_u:java_r:admin_j);
```

En utilisant cette propriété, le secret d'`Admin` sera bien écrit dans l'objet partagé `Storage`, mais PIGA-Virt empêchera la copie du secret présent dans `Storage` par `Guest`. La confidentialité des données d'`Admin` sera donc bien préservée.

Une règle similaire garantit la confidentialité de *Guest* :

```
confidentiality(system_u:java_r:admin_j, system_u:java_r:guest_j);
```

Ces deux règles permettent à PIGA de trouver les quatre séquences d'interactions qui violeraient la confidentialité d'*Admin* et de *Guest*. Ces quatre séquences sont présentées dans le listing suivant :

```
#Séquences menant à la violation de la confidentialité d'Admin
system_u:java_r:usecase_j -( getSecret { invoke } )-> system_u:java_r:admin_j
    ; system_u:java_r:usecase_j -( addSecret { invoke } )-> system_u:java_r:
    guest_j
system_u:java_r:admin_j -( addSecret { invoke } )-> system_u:java_r:storage_j
    ; system_u:java_r:guest_j -( getSecret { invoke } )-> system_u:java_r:
    storage_j
#Séquences menant à la violation de la confidentialité de Guest
system_u:java_r:usecase_j -( getSecret { invoke } )-> system_u:java_r:guest_j
    ; system_u:java_r:usecase_j -( addSecret { invoke } )-> system_u:java_r:
    admin_j
system_u:java_r:guest_j -( addSecret { invoke } )-> system_u:java_r:storage_j
    ; system_u:java_r:admin_j -( getSecret { invoke } )-> system_u:java_r:
    storage_j
```

4.6 Analyse de politique

Cette sous-section montre l'effet sur un système de la politique PIGA comportant les règles vues précédemment pour les trois types de protection :

```
#Propriétés appliquées sur les hôtes et les machines virtuelles (niveau 1)
integrity( $sc1:="user_u:user_r:user.*_t", $sc2:=".*:.*:.*_exec_t" );
integrity( $sc1:="user_u:user_r:user.*_t", $sc2:=".*:.*:.*etc_t" );
confidentiality( $sc1:="user_u:user_r:user.*_t", $sc2:=system_u:object_r:
    shadow_t );
confidentiality( $sc1:="user_u:user_r:user.*_t", $sc2:=system_u:object_r:
    memory_device_t );
dutiesseparationbash( "user_u:user_r:user.*_t" );

#Propriété appliquée sur l'hôte (niveau 2)
trustedpathexecution( $sc1:=system_u:system_r:svirt_t , $TPE:= { ".*:.*:.*
    qemu_exec_t",  ".*:.*:.*lib_t",  ".*:.*:.*lib_qemu_t" } )

#Propriété appliquée dans la JVM (niveau 3)
confidentiality(system_u:java_r:guest_j, system_u:java_r:admin_j);
```

Le tableau 1 donne le nombre d'interactions / séquences / compositions de séquences qui violeraient les différentes règles de la politique et seront donc bloquées par PIGA.

On peut donc voir qu'une seule règle PIGA peut détecter et empêcher des millions d'activités illégales.

Dans le cas de la JVM, le nombre d'activités malveillantes détectées est faible : ce résultat s'explique par la simplicité de la politique et de l'exemple utilisé.

Nom		Hôte Centos/SELinux/Svirt	VM Gentoo/Selinux	Java JVM 7
Graphe	NB contextes	2897	577	210
d'interaction	NB opérations	1 879 063	17684	960
	`integrity`	0	0	
	`integrity`	74	30	
Propriétés	`confidentiality`	145 718	86 268	
	`confidentiality`	100 468	65 648	
PIGA	`dutiesseparationbash`	103 747 632	14 629 680	
	`trustedpathexecution`	8 715		
	`confidentiality`			4

TAB. 1 – *Résultats de l'analyse des propriétés.*

5 Etat de l'art

La sécurité est un enjeu majeur pour les Clouds (Pearson et Benameur, 2010). En effet, le périmètre de sécurité devient plus flou entre d'une part l'intranet, et d'autre part l'internet. De plus, la surface d'attaque augmente du fait de la virtualisation comme l'ont montré les résultats du défi sécurité de l'ANR (J. Briffaut, 2011). Plusieurs articles étudient la sécurité des Clouds (Jaeger et Schiffman, 2010; Vaquero et al., 2011; Sandhu et al., 2010). Dans Takabi et al. (2010), les auteurs mettent en avant l'importance du contrôle d'accès dans les Clouds publiques. Ce contrôle doit être obligatoire (Harrison et al., 1976) puisque le contrôle d'accès discrétionnaire sous la responsabilité des utilisateurs finaux est fragile et ne permet pas de garantir des propriétés de confidentialité ou d'intégrité. Le contrôle obligatoire est réalisé par un moniteur de référence (Lampson, 1971) qui place les politiques de protection hors de portée des utilisateurs et des administrateurs des machines. En pratique, les politiques doivent être définies par l'opérateur de l'infrastructure Cloud. De plus, la coopération entre les différents niveaux de contrôle d'accès (ex. hyperviseur, système hôte, VMs, applications, etc.) doit être réalisée dans un langage dédié afin d'uniformiser la politique de protection indépendamment d'un niveau considéré.

Dans Hicks et al. (2010), les auteurs proposent une architecture de contrôle d'accès de bout en bout pour les applications web. Ils utilisent un contrôle obligatoire à l'intérieur et entre les machines virtuelles (Virtual Machines). Cependant, ils sont limités à un noeud et ne peuvent pas exprimer des propriétés de sécurité avancées qui permettraient notamment de traiter les flux d'information indirects empruntant des ressources intermédiaires et correspondant à une fermeture transitive de flux directs entre une ressource sujet (par exemple un processus) et une ressource objet (par exemple un fichier)..

Dans Rueda et al. (2009), une méthode permet d'analyser la politique résultant de la combinaison d'une politique XSM et de multiples politiques SELinux. Mais encore une fois, comme dans Hicks et al. (2010), ils ne disposent que d'un noeud hébergeant de multiples VMs et ne prennent pas en compte le déploiement ni la migration des VMs. Enfin, l'approche est limitée à l'analyse et ne permet pas de protéger le système.

Dans Payne et al. (2007), les auteurs décomposent une politique complexe en plusieurs couches. Ils contrôlent les interactions inter et intra VMs. Mais, l'approche considère un seul

hyperviseur hébergeant les différentes VMs. De plus, ils ne proposent pas de langage pour exprimer des propriétés avancées et ne peuvent contrôler les flux indirects.

L'article Zhang et al. (2011) propose l'ajout d'un mini hyperviseur aux hyperviseurs existants. Ils réalisent ainsi un contrôle d'intégrité à partir d'une base de confiance. Cependant, ils ne peuvent exprimer d'autres propriétés de sécurité et ne savent pas contrôler les flux d'information. De façon similaire, Azab et al. (2011) permet une isolation des machines.

Des solutions existent pour contrôler les flux à l'intérieur de la JVM. JAAS est une API permettant d'assurer un contrôle d'accès basé sur le pile (SBAC), cependant cette API doit être explicitement appelée par le développeur dans l'application, ce qui limite son utilisation et son efficacité (Abadi et Fournet, 2003; Pistoia, 2007). Le contrôle d'accès obligatoire à l'intérieur de la JVM est développé dans Haldar et al. (2005a,b), qui proposent d'intégrer un moniteur de référence dans la JVM. Cependant, les auteurs utilisent de la coloration de données, ce qui limite l'expression des propriétés et les performances. Nair et al. (2008) propose une approche similaire, ayant les mêmes limitations.

Ainsi, les solutions manquent pour contrôler les flux transitifs, garantir des propriétés avancées à l'intérieur et entre les machines virtuelles mais aussi à l'intérieur des applications. De plus, aucune solution ne traite le problème de la complexité des politiques obligatoires. Enfin, aucune approche ne permet de gérer de façon cohérente les contextes de sécurité durant tout le cycle de vie des machines virtuelles et des applications alors que c'est un enjeu majeur.

6 Conclusion

Notre article montre comment améliorer la protection des environnements de Cloud via un contrôle d'accès en profondeur. Nous contrôlons ainsi les activités sur le système hôte, à l'intérieur des machines virtuelles invitées nécessaires à un Cloud du type IaaS, mais aussi à l'intérieur des applications Java nécessaires pour un Cloud PaaS ou SaaS. PIGA-Cloud fournit une labelisation cohérente des machines virtuelles tout au long de leur cycle de vie en supportant notamment les migrations et les suspensions. L'approche permet de définir aisément une propriété avancée qui contrôle plusieurs flux d'information directs et indirects. Ainsi, un large ensemble de propriétés de confidentialité et d'intégrité est supporté. PIGA-Cloud réutilise les politiques obligatoires directes existantes, notamment les politiques SELinux et sVirt. De plus, l'article présente l'approche SEJava pour contrôler les flux directs entre les objets d'une application Java. PIGA-Cloud simplifie la protection en profondeur. Grâce à un langage unifié et à la définition de seulement quelques règles, PIGA-Cloud facilite le travail d'un administrateur de la sécurité du Cloud. Bien que l'approche soit indépendante de l'environnement d'administration du Cloud considéré, une intégration et des expérimentations sont proposées pour l'environnement OpenNebula. Ainsi, nous pouvons illustrer l'apport lors du cycle du vie des machines virtuelles et pour le contrôle des différents types de flux et nous présentons aussi la protection offerte pour les applications Java d'un Cloud PaaS ou SaaS.

Les travaux futurs concernent différents aspects. Tout d'abord, une approche complète d'un environnement orienté mission de sécurité va être proposée dans le cadre du projet européen Seed4C. De plus, un portage de SEJava pour la machine virtuelle Dalvik d'Android est disponible, permettant via SEAndroid d'offrir une approche de protection en profondeur PIGA-Android contrôlant le système Linux ainsi que les applications Java. Actuellement, il n'existe aucune approche similaire dans les travaux récents ou en cours sur Android. Ainsi, nous pour-

rons offrir des clients PIGA-Android accédant à des services PIGA-Cloud en garantissant une sécurité de bout en bout partant d'un client et passant par les différents noeuds du Cloud.

Références

Abadi, M. et C. Fournet (2003). Access control based on execution history. In *In Proceedings of the 10th Annual Network and Distributed System Security Symposium*, pp. 107–121.

Azab, A. M., P. Ning, et X. Zhang (2011). Sice : a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, New York, NY, USA, pp. 375–388. ACM.

Briffaut, J. (13 décembre 2007). *Formalisation et garantie de propriétés de sécurité système : application à la détection d'intrusions*. Ph. D. thesis, Thèse de doctorat en informatique, Université d'Orléans.

Briffaut, J., E. Lefebvre, J. Rouzaud-Cornabas, et C. Toinard (2011). Piga-virt : an advanced distributed macprotection of virtual systems. In *VHPC 2011 : 6th Workshop on Virtualization and High-Performance Cloud Computing*, Bordeaux, France.

D. Gros, J. B. e. C. T. (2012). Contrôle d'accès mandataire pour windows 7. In *Symposium sur la sécurité des technologies de l'information et des communications*, pp. 266–291.

Haldar, V., D. Chandra, et M. Franz (2005a). Dynamic taint propagation for java. Technical report, Department of Information and Computer Science - University of California.

Haldar, V., D. Chandra, et M. Franz (2005b). Practical, dynamic information-flow for virtual machines. Technical report, Department of Information and Computer Science - University of California.

Harrison, M. A., W. L. Ruzzo, et J. D. Ullman (1976). Protection in operating systems. *Communications of the ACM 19*(8), 461–471.

Hicks, B., S. Rueda, D. King, T. Moyer, J. Schiffman, Y. Sreenivasan, P. McDaniel, et T. Jaeger (2010). An architecture for enforcing end-to-end access control over web applications. In *Proceedings of the 15th ACM symposium on Access control models and technologies*, SACMAT '10, New York, NY, USA, pp. 163–172. ACM.

J. Briffaut, M. Perès, J. R.-C. J. S. C. T. e. B. V. (2011). Piga-os : Retour sur le système d'exploitation vainqueur du défi sécurité. In *8ième Conférence Francophone sur les Systèmes d'Exploitation*.

Jaeger, T. et J. Schiffman (2010). Outlook : Cloudy with a Chance of Security Challenges and Improvements. *IEEE Security & Privacy Magazine 8*(1), 77–80.

Lampson, B. W. (1971). Protection. In *The 5th Symposium on Information Sciences and Systems*, Princeton University, pp. 437–443.

Nair, S., P. Simpson, B. Crispo, et A. Tanenbaum (2008). Trishul : A policy enforcement architecture for java virtual machines. In *Technical Report IR-CS-045*.

Payne, B. D., R. Sailer, R. Cáceres, R. Perez, et W. Lee (2007). A layered approach to simplified access control in virtualized systems. *SIGOPS Oper. Syst. Rev. 41*, 12–19.

Pearson, S. et A. Benameur (2010). Privacy, security and trust issues arising from cloud computing. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, Washington, DC, USA, pp. 693–702. IEEE Computer Society.

Pistoia, M. (2007). Beyond stack inspection : A unified access-control and information-flow security model. In *In SPï£¡ : Security and Privacy*, pp. 149–163. IEEE.

Rouzaud-Cornabas, J. (2 décembre 2010). *Formalisation de propriétés de sécurité pour la protection des systèmes d'exploitation*. Ph. D. thesis, Thèse de doctorat en informatique, Université d'Orléans.

Rueda, S., H. Vijayakumar, et T. Jaeger (2009). Analysis of virtual machine system policies. In *Proceedings of the 14th ACM symposium on Access control models and technologies*, SACMAT '09, New York, NY, USA, pp. 227–236. ACM.

Sandhu, R., R. Boppana, R. Krishnan, J. Reich, T. Wolff, et J. Zachry (2010). Towards a discipline of mission-aware cloud computing. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, New York, NY, USA, pp. 13–18. ACM.

Takabi, H., J. Joshi, et G. Ahn (2010). Security and privacy challenges in cloud computing environments. *Security Privacy, IEEE 8*(6), 24 –31.

Vaquero, L. M., L. Rodero-Merino, et D. Mor&#x00e1 ;n (2011). Locking the sky : a survey on iaas cloud security. *Computing 91*, 93–118.

Zhang, F., J. Chen, H. Chen, et B. Zang (2011). Cloudvisor : retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, New York, NY, USA, pp. 203–216. ACM.

Summary

To guaranty security properties, a system needs to use Mandatory Access Control (MAC). However, Cloud environments do not currently implement such mechanisms. PIGA-Cloud guaranties advanced properties for controlling indirect flows and combinaisons of flows. It provides an in-depth MAC protection controlling the flows between the guest virtual machines and the host, the internal flows of a guest but also the flows between Java objects and the network flows. This paper shows how to PIGAized various environments such as Unix machines, Java applications and Clouds. PIGA-Cloud extends the direct access mac policies of SELinux and sVirt to the Java Virtual Machine in order to provide an advanced protection of IaaS, PaaS and SaaS Clouds. Our approach simplifies the administration of the direct access policies while preventing against millions of remaining vulnerabilities. This work is partially supported by the Seed4C european project.