PIGA-Windows : contrôle des flux d'information avancés sur les systèmes d'exploitation Windows 7

Mathieu Blanc, Damien Gros, Jérémy Briffaut, Christian Toinard

MajecSTIC 2012
Lille, France, du 29 au 31 octobre 2012

PIGA-Windows : contrôle des flux d'information avancés sur les systèmes Windows 7

Mathieu Blanc[1] et Damien Gros[1 2] ; Jérémy Briffaut[2] et Christian Toinard[2]

1 : CEA, DAM, DIF, F-91297 Arpajon, France
2 : Laboratoire d'Informatique Fondamentale d'Orléans
ENSI de Bourges – LIFO, 88 bd Lahitolle, 18020 Bourges cedex, France

Contact : damien.gros@cea.fr

Résumé
Cet article décrit les avancements des travaux réalisés sur le renforcement du contrôle d'accès de Windows 7. Le noyau du système d'exploitation contrôle les interactions entre processus et ressources, néanmoins, ceci ne concerne pas les flux d'information complexes. Pour répondre à ce besoin, nous avons implanté un mécanisme appelé PIGA-Windows avec deux niveaux de contrôle d'accès mandataire. Le premier est inspiré du modèle *Type Enforcement* et concerne les interactions directes. Le second est PIGA-Monitor, un moniteur de référence appliquant des propriétés de sécurité avancées. Nous détaillons l'implantation de ces deux niveaux de contrôle. Notre solution contrôle l'activité système et tous les types de flux d'information, qu'ils soient directs ou indirects. De plus, dans le but de faciliter l'écriture d'une politique de contrôle d'accès obligatoire, nous avons inclus un mécanisme d'apprentissage qui transforme les événements observés par notre solution en règles de contrôle d'accès. Pour protéger contre des attaques plus complexes, des propriétés de sécurité complémentaires sont appliquées avec le moniteur de référence PIGA. Des exemples de menaces bloquées par PIGA-Windows sont détaillés dans cet article ainsi que l'étude des performances de PIGA-Monitor.

Mots-clés : système d'exploitation, sécurité, contrôle d'accès, flux d'information

1. Contexte du sujet

La vérification de propriétés de sécurité sur un système d'exploitation est un problème vaste et complexe. Le contrôle des actions des utilisateurs est nécessaire pour s'assurer qu'aucune fuite d'information n'est possible. Deux principaux modèles théoriques sont définis par Bell – La Padula [1] et Biba [2] et établissent les propriétés de confidentialité et d'intégrité des données. Toutefois ces modèles se sont révélés difficilement applicables aux systèmes d'exploitation modernes [13].
Le contrôle d'accès est le mécanisme vérifiant qu'un processus à l'origine d'une action possède bien toutes les autorisations nécessaires pour la faire. Sur les systèmes d'exploitation classiques tels que Windows, Linux ou MacOS, ce contrôle d'accès est discrétionnaire DAC. C'est donc l'utilisateur propriétaire d'une ressource (le propriétaire) qui décide qui est autorisé ou non à y accéder. Ce modèle n'est toutefois pas fiable. En effet, il est possible, par simple négligence ou par une action malveillante, qu'un utilisateur laisse filer de l'information. De plus, dans le cadre d'une entreprise, il peut être nécessaire de définir des niveaux de confidentialité pour l'accès à certains documents (habilitation nécessaire par exemple).
Plusieurs articles comme [9] montrent que le modèle DAC n'est pas pérenne. En effet, ce modèle n'assure pas une politique consistante vis à vis d'objectifs de sécurité, c'est-à-dire qu'on ne peut prouver qu'il n'existe pas au moins un moyen d'atteindre un état non sûr de la machine à partir d'un état sûr. Il existe des modèles de contrôle d'accès, nommés contrôle d'accès obligatoire ou mandataire MAC, qui résolvent ces problèmes. Ces modèles sont basés sur l'autorisation explicite de chaque accès, et reposent sur l'écriture d'une politique de sécurité décrivant tous les accès autorisés. Dans le cas de Linux, cette politique est appliquée par le noyau.

Dans cet article, nous allons tout d'abord faire un état de l'art des modèles de contrôle d'accès existants. Puis, nous décrirons notre solution pour les systèmes d'exploitation Windows. Ensuite, nous expliquerons comment créer une politique de contrôle d'accès obligatoire pour gérer les flux d'information directs. Enfin, sur différents exemples, nous montrerons comment gérer et contrer les attaques avancées.

2. État de l'art

La majorité des travaux du domaine définissent des modèles de protection répondant aux problèmes introduits par le contrôle d'accès discrétionnaire. Par exemple, Bell et La Padula [1] et Biba [2] présentent des modèles de protection orientés militaire qui sont généralement très difficiles à mettre en place sur un système d'exploitation. Plus globalement, les modèles décrits dans les articles de recherche sont souvent très complexes à implémenter. Une convergence naturelle des domaines de classification rend le système soit inutilisable, soit sans niveau de classification. [13] Cependant, peu de travaux traitent des problèmes d'intégration de ces modèles dans le noyau des systèmes d'exploitation.
Les articles de recherche tels que Efstathopoulos [6] et Zeldovich [17] sont orientés vers le contrôle des flux d'information. Le but principal de ces travaux est de protéger les utilisateurs les uns des autres. Toutefois ces travaux demandent que les politiques de sécurité soient écrites par les développeurs des applications. Enfin il manque un langage de haut niveau pour définir des propriétés de sécurité à respecter.
Pour le noyau Linux, plusieurs approches sont possibles. Une des plus matures, initiée par la NSA, est SELinux [12]. Cette solution fournit une implémentation du modèle `Type Enforcement` [4]. Ce mécanisme se base sur un label donné aux ressources pour autoriser ou non l'action. Il contrôle ainsi tous les processus, mêmes les plus privilégiés.
SELinux utilise une architecture nommée *Flask* [16] qui sépare la partie prise de décision et d'application de la politique de sécurité. Cette couche va abstraire les ressources en leur associant une chaîne de caractères les décrivant, appelée **contextes de sécurité**. Un contexte de sécurité, aussi appelé label, est composé sous SELinux de trois éléments principaux. Le premier est une identité spécifique à SELinux qui est liée à l'UID standard de Linux. Ensuite un rôle, lié au modèle Role-Based Access Control [7], offre à l'utilisateur l'accès à différents types. Enfin, le type, qui est une chaîne de caractère caractérisant la ressource, est la partie *Type Enforcement* de SELinux et c'est à partir de ces types qu'est définie la politique de sécurité. Cette solution offre la possibilité d'avoir une politique plus simple à écrire qu'une politique basée sur les identités puisque les rôles fournissent un niveau d'indirection par rapport aux types. Ainsi le modèle du *Type Enforcement* va pouvoir 1) confiner les applications et 2) réduire leurs privilèges. Par conséquent, SELinux ne pourra pas empêcher une vulnérabilité présente dans l'application d'être exploitée mais réduira l'impact de l'attaque sur le système. SELinux se base sur une politique textuelle, qui définit explicitement ce qui est autorisé. Comme il est nécessaire de définir les actions autorisées pour chaque programme, il est parfois très long et coûteux de devoir écrire une politique fonctionnelle pour tout un système.
En ce qui concerne les systèmes d'exploitation Windows, plusieurs travaux traitent de l'intégration d'un mécanisme de contrôle d'accès mandataire. Le premier est Core Force [11] qui fournit une première implémentation pour le noyau de Windows XP. De plus, Microsoft a développé depuis Windows Vista et surtout Windows 7 une implémentation restreinte de Biba basée sur les niveaux d'intégrité Mandatory Integrity Control ou MIC. Cependant ce mécanisme ne confine pas les applications. Le contrôle d'intégrité mandataire de Windows concerne uniquement la protection des données et fichiers système. Par une augmentation légitime des privilèges, un utilisateur peut se retrouver avec les droits d'administration sans que ces actions ne soient contrôlées. Mais ce problème n'est pas présent que sous Windows, puisque grâce à la commande *sudo*, un utilisateur peut aussi augmenter ses privilèges sous Linux.
Dans le contexte des systèmes Windows, les solutions sont assez limitées. Les travaux abordant la sécurité des systèmes Windows sont généralement orientés vers l'étude des politiques DAC [15]. Miller [14] considère lui les escalades de privilèges grâce à des failles dans les programmes RPC (Remote Procedure Call).

3. PIGA-Windows

3.1. Architecture

Cette section décrit l'architecture que nous proposons pour notre modèle de contrôle d'accès. Le contrôle d'accès obligatoire est renforcé par l'utilisation de deux moniteurs de référence. Le premier est un *driver* noyau qui capture les appels système et contrôle les flux d'information directs. Un processus en espace utilisateur, nommé PIGA-Monitor, fournit un second moniteur de référence garantissant des propriétés de sécurité avancées.

Notre implémentation est inspirée de l'approche utilisée par SELinux. Chaque ressource du système, est associée à un contexte de sécurité. Ce contexte de sécurité décrit la ressource grâce à trois identifiants : une identité, un rôle et un type. Dans SELinux, un fichier nommé `file_contexts` définit la correspondance entre contextes et ressources, une fois appliqués les contextes sont conservés dans les attributs étendus du système de fichiers. Nous avons choisi une implémentation différente pour Windows. Notre *driver* calcule dynamiquement les contextes de sécurité lorsqu'il capture un appel système.

3.2. Problématique de la labellisation

La labellisation dynamique a été choisie pour pouvoir s'affranchir du système de fichiers utilisé. Elle assure une complète portabilité entre les différents systèmes Windows. Toutefois quelques difficultés apparaissent car il faut tenir compte des points de montage ainsi que des différentes lettres d'installation du système. Il est donc nécessaire d'uniformiser la façon de nommer les ressources du système. Sous Windows, un même fichier peut avoir plusieurs noms suivant différentes conventions de nom. Dans le but d'avoir une abstraction totale vis à vis de toute convention, nous utilisons les variables d'environnement présentes nativement sous Windows.

Par exemple, la variable `%systemroot%` est la représentation canonique pour le répertoire de `Windows`. Notre *driver* calcule les noms canoniques pour toutes les ressources présentes dans le répertoire `Windows` en utilisant cette variable d'environnement. Le nom canonique pour le fichier `C:\Windows\System32\cmd.exe` est `%systemroot%\system32\cmd.exe`. Grâce à cette méthode, il sera aisé de gérer le cas d'un répertoire d'installation différent.

3.3. *Driver*

Les objectifs de notre *driver* sont 1) de détourner les appels système, 2) d'appliquer rigoureusement la politique de protection MAC et 3) de fournir des *logs* pour la création simplifiée de politique MAC et le suivi des actions qui se sont déroulées sur le système. La figure 1 montre comment le *driver* implémente les trois fonctions. La première fonction doit gérer la *System Service Dispatch Table* (SSDT), qui est la table contenant les adresses des appels système. Elle gère au final deux SSDT comme nous l'expliquons plus loin. La seconde fonction doit analyser les requêtes provenant des appels système et donner une réponse (autoriser ou non l'action) en fonction de la politique de contrôle d'accès obligatoire. La troisième fonction fournit des logs d'accès lorsque l'appel système est refusé. Ce sont ces logs d'accès qui seront utilisés par notre programme pour écrire la politique MAC.

Lors du chargement du *driver*, des détournements (*hooks*) sont placés sur un ensemble d'appels système dans le but de les contrôler. Comme présenté dans la figure 1, le *driver* récupère la SSDT courante en y appliquant les modifications voulues dans le but de détourner les appels système. Afin de pouvoir exécuter le code d'origine des appels système, le *driver* doit sauvegarder une version de la SSDT précédente en mémoire. Cette partie décrit comment le *driver* réalise ces deux opérations, suivant une méthode inspirée du chapitre 4 de Hoglund [10].

Pour créer la nouvelle SSDT, qui contiendra donc les détournements voulus, le *driver* doit tout d'abord récupérer la SSDT courante. Ensuite, il crée la SSDT qui va contenir les *hooks* pour détourner les appels système vers des fonctions maîtrisées par le *driver*. Pour pouvoir exécuter le code d'origine des appels système, la SSDT courante est conservée en mémoire. Puis, la SSDT modifiée contenant les différents *hooks* remplace la SSDT d'origine.

Lorsque la fonction détournée est appelée, elle doit vérifier que la requête est valide. Une requête est valide si et seulement si il existe une entrée correspondante dans la politique de contrôle d'accès obligatoire. L'entrée est définie par le sujet, l'objet, la classe de l'objet et l'action demandée par le sujet sur l'objet. Si une telle entrée existe, il faut alors demander à PIGA-Monitor si l'action est

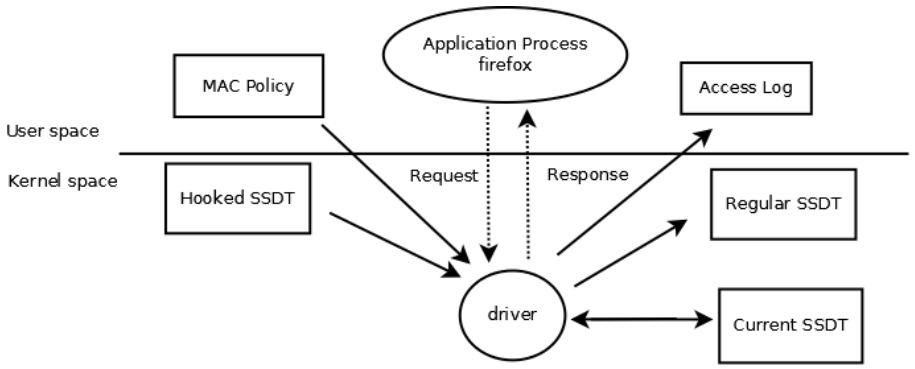

FIGURE 1 – Driver

conforme.

3.4. PIGA-Monitor

PIGA-Monitor est un processus qui s'exécute en espace utilisateur. Il obtient les requêtes auprès du *driver*, donc les actions *a priori* autorisées, et vérifie pour chaque requête si elle est conforme aux propriétés de sécurité avancées définies par l'administrateur. Après vérification, il informe le *driver* s'il doit autoriser ou non l'appel système.

Pour effectuer cette vérification, PIGA-Monitor construit une base de données de toutes les séquences d'actions illégales laissées possibles par la politique de contrôle d'accès obligatoire appliquée au niveau du *driver*. Cette base est créée par un pré-calcul sur la politique de contrôle d'accès obligatoire et les propriétés de sécurité avancées. Quand il reçoit une requête venant du *driver*, il va vérifier que l'action ne fait pas partie d'un ensemble d'actions pouvant conduire à une violation de propriété. PIGA-Monitor conserve donc un historique des séquences d'actions. Si la requête demandée fait partie d'un ensemble d'activités illégales, PIGA-Monitor va refuser l'appel système.

3.5. Avantages

Cette architecture offre plusieurs avantages. Le premier est qu'elle fournit deux niveaux de contrôle d'accès obligatoire assurant une protection en profondeur du système d'exploitation. Ces deux couches contrôlent à la fois les flux d'information directs grâce au *driver*, mais aussi les flux d'information indirects ou complexes grâce à PIGA-Monitor. Le second est le mode apprentissage de notre *driver* qui crée facilement une politique de contrôle d'accès obligatoire. Comme cette politique ne prend pas en compte les flux d'information indirects, PIGA-Monitor est utilisé pour garantir des propriétés de sécurité avancées. Ces propriétés sont écrites dans un langage de haut niveau SPL qui décrit les activités mettant en jeux une multitude de flux. Ce langage facilite la création de propriétés avancées associées à de multiples flux. En pré-calculant les possibles violations de politique, ce mécanisme est suffisamment rapide et robuste pour être utilisé en temps réel sans pour autant bloquer ou ralentir le système d'exploitation. Enfin, en pratique, les propriétés de sécurité avancées déjà définies peuvent être réutilisées, donc la charge de travail pour l'administrateur est faible.

4. Contrôle des flux d'information directs

Cette section décrit le contrôle des flux d'information directs appliqué par le *driver* noyau. Nous allons illustrer ce mécanisme sur plusieurs exemples.

Les contextes de sécurité sont à la base de la définition d'une politique de contrôle d'accès obligatoire. D'une manière générale, une politique MAC établit un jeu de règles $\{S - (P) \rightarrow O\}$ où S est un contexte sujet, P une permission et O un contexte objet. Par exemple, la règle présentée en listing 1 autorise le contexte sujet `system_u:system_r:explorer_t` à exécuter l'objet `system_u:system_r:cmd_exec_t`.

Listing 1 – Règle d'accès

```
system_u:system_r:explorer_t (execute) system_u:system_r:cmd_exec_t
```

L'écriture d'une politique pour tout un système est un travail à la fois long et fastidieux. Dans un premier temps, l'administrateur doit créer une politique de contrôle d'accès obligatoire. Dans le but de faciliter l'écriture d'une telle politique, un programme nommé *policyProducer* parcourt les fichiers de *log* et les transforme en règles de contrôle d'accès obligatoire. Tout le mécanisme de création de la politique est décrit par Gros *et al.* [8].
Par exemple, le listing 2 montre comment autoriser un programme qui possède le type `opera_t` à écrire dans un fichier possédant le type `file_t`.

Listing 2 – Règle autorisant Opera à écrire dans un fihcier

```
allow opera_t file_t:file { write }
```

Une fois que l'administrateur a fini d'écrire la politique de contrôle d'accès obligatoire, il met le *driver* en mode *enforcing* pour protéger le système. Pour avoir plusieurs niveaux de contrôle d'accès obligatoire, le *driver* écrit toutes les requêtes qu'il reçoit dans le fichier de log. Le listing 3 montre que la règle décrite par le listing 2 est bien autorisée par notre *driver*. C'est ce fichier de log qui sera lu par PIGA-Monitor et qui appliquera des propriétés de sécurité avancées.

Listing 3 – Log d'accès pour Opera

```
type=AVC msg=audit(129648732400631895,904)
avc : granted { write } for pid=3884 com="%programfiles%\opera\opera.exe"
ppid=1960 path="%systemdrive%\shared_firefox_opera\web_files.txt"
scontext=system_u:system_r:opera_t tcontext=system_u:object_r:file_t
tclass=file
```

Ce modèle fournit plusieurs avantages pour l'administrateur. La possibilité d'avoir un outil qui automatise la création de la politique de contrôle d'accès obligatoire facilite son travail. Cette politique, basée sur les types des sujets et des objets, offre une bonne portabilité pour notre politique. Cela signifie que le format de la politique est complètement indépendant du système de fichiers mais aussi de Windows. De plus, notre solution ne se substitue pas au contrôle classique des droits sous Windows.

5. Contrôle de propriétés de sécurité avancées

Dans le but de garantir des propriétés de sécurité avancées mettant en jeu des flux d'information directs et indirects, nous utilisons un langage SPL dédié. Ce langage définit des propriétés concernant la description des différentes actions possibles entre les nombreux contextes de sécurité présents sur le système ainsi que les activités associées à ces actions. Comme nous le montrons dans [3], à partir de propriétés écrites dans ce langage, on peut calculer toutes les activités illégales vis à vis d'une politique SELinux mais aussi garantir des propriétés de sécurité au travers d'une approche efficace appelée PIGA-HIPS. C'est ce mécanisme que nous proposons d'adapter pour les systèmes d'exploitation Windows.

5.1. Propriétés de sécurité avancées

Dans cette section, nous allons montrer comment utiliser le SPL pour définir des propriétés de sécurité avancées [3]. Nous allons décrire ces propriétés de sécurité avancées sur deux exemples.
Le listing 4 montre comment écrire une propriété simple en SPL. Cette propriété empêche les flux d'information directs $>$ et les flux d'information indirects $>>$ entre les deux contextes de sécurité sc_1 et sc_2. Dans la pratique, cela signifie qu'il ne peut y avoir d'interaction provenant de sc_2 et allant vers sc_1 (i.e. un appel système de sc_2 ayant pour cible sc_1) ou une combinaison d'interactions mettant en jeu des contextes de sécurité différents. Cette séquence se caractérise par l'association de plusieurs flux d'information indirects allant de sc_2 à sc_1 et composée par de multiple flux d'information directs tous autorisés [5].

Listing 4 – Propriété de confidentialité

```
1  define confidentiality( $sc1 IN SCS, $sc2 IN SCO )
2  [ ST { $sc2 > $sc1 }, { not(exist()) };
3    ST { $sc2 >> $sc1 }, { not(exist()) }; ];
```

Le principal avantage de cette approche est qu'elle protège contre les limites de la politique de contrôle d'accès obligatoire de premier niveau.
Pour montrer l'efficacité de notre langage, nous allons donner un deuxième exemple fondé sur une attaque concrète. Celle-ci se décompose en plusieurs points : un processus ayant pour label de sécurité sc_1 écrit un fichier possédant le contexte sc_2 et exécute un shell sc_3 qui souhaite lire le fichier préalablement écrit dans le but de l'exécution. Cette propriété de sécurité peut être utilisée pour empêcher l'exécution accidentelle de scripts téléchargés.

Listing 5 – Règle PIGA bloquant un flux avancé

```
1  define dutiesseparationinterpreter( sc1 IN SC ) [
2          Foreach sc2 IN SCO, Foreach sc3 IN SC,
3                  ¬ ( ( sc1 >write sc2) −then−> (sc1 >execute sc3 ) −then−> (sc3 <read sc2))
4   ];
```

5.2. Propriétés de sécurité spécifiques

Nous allons maintenant à partir sur deux exemples comment les propriétés peuvent être utilisées sur un système Windows.

Listing 6 – Application de la propriété de confidentialité

```
1  confidentiality( $sc1:=".*:.*:opera_t",$sc2:=".*:.*:firefox_t" );
2  confidentiality( $sc1:=".*:.*:firefox_t",$sc2:=".*:.*:opera_t" );
```

Le listing 6 montre comment écrire en pratique les règles de confidentialité. Ici, on s'assure qu'il n'y aura pas d'échange d'informations entre les applications Firefox et Opera.
Dans l'exemple suivant, le navigateur Firefox est utilisé pour aller sur des sites de banques et Opera pour accéder aux réseaux sociaux. Comme on le sait, les sites de réseaux sociaux tracent leurs utilisateurs et c'est pour cela que l'administrateur ne veut pas que des documents puissent être lus d'un navigateur à l'autre. Donc la première propriété du listing 6 empêche tous les flux venant de Firefox et allant à Opera. La seconde propriété empêche les flux d'Opera vers Firefox.
A partir d'une politique de contrôle d'accès obligatoire que nous avons écrite à partir du mode apprentissage de notre *driver*, nous avons appliqué ces deux propriétés sur cette politique grâce à PIGA-Monitor. Il a ainsi pré-calculé respectivement 111 et 99 activités illégales pour les premières et deuxièmes propriétés.
Voici un exemple 7 de flux possible provenant de Firefox et allant vers Opera.

Listing 7 – Trace d'un flux indirect de Firefox vers Opera

```
1\$54 : system_u:system_r:firefox_t −( file { create write } )−>system_u:object_r:systemroot_dir_t ;
system_u:system_r:adobearm_t −( file { execute read } dir { execute } )−>
system_u:object_r:systemroot_dir_t;
system_u:system_r:adobearm_t −( file { create setattr unlink } )−> user_u:object_r:user_home_opera_t ;
system_u:system_r:opera_t −( file { execute getattr read } )−> system_u:object_r:user_home_opera_t ;
```

Firefox écrit des données dans un répertoire. Ces données sont lues par le processus *Adobearm* qui les copie dans le répertoire ayant pour type `user_home_opera_t`. Enfin, Opera lit les données contenues dans le répertoire typé `user_home_opera_t`.
La politique de contrôle d'accès obligatoire, qui ne gère que les flux directs, autorise chaque action prise séparément. Ainsi, le *driver* ne peut pas empêcher les flux d'information indirects mettant en jeux une combinaison de flux autorisés.

Listing 8 – Schéma du flux indirect de Firefox vers Opera

```
firefox_t > systemroot_dir_t > adobearm_t > user_home_opera_t > opera_t
```

Cette séquence d'actions autorisées peut être empêchée en supprimant une des actions autorisées dans la politique de contrôle d'accès obligatoire. Par exemple en supprimant la règle autorisant *Adobearm* à écrire dans le répertoire utilisateur, il est facile d'interrompre cette séquence.
Cependant, une telle modification de la politique de contrôle d'accès obligatoire pourrait empêcher une application de fonctionner correctement. C'est pourquoi, il est préférable de contrer ce genre d'attaque en considérant plutôt les flux indirects ou la combinaison de flux directs que les interactions directes prises une à une.

5.3. PIGA-monitor

Dans la pratique, PIGA-Monitor récupère les requêtes venant du *driver* sous forme de traces d'appels système dans le but d'autoriser ou non les requêtes. Si une requête correspond à une activité illégale pré-calculée, alors PIGA-Monitor envoie un refus au *driver* pour bloquer l'action en cours. Nous allons montrer un exemple de refus.

Listing 9 – Firefox commençant à écrire dans un fichier temporaire

```
type=AVC msg=audit(begin=[129648732304525985],end=[,]) avc : granted { write } for pid=3884
com="%programfiles%\firefox\firefox.exe" ppid=1960  path="%systemroot%\temp\file.txt"
scontext=system_u:system_r:firefox_t  tcontext=system_u:object_r:file_t tclass=file
```

Listing 10 – Adobearm commençant à lire le contenu du fichier

```
type=AVC msg=audit(begin=[.............41343],end=[]) avc : granted { read } for pid=1880
com="%programfiles%\adobe\adobearm.exe" ppid=160 path="%systemroot%\temp\file.txt"
scontext=system_u:system_r:adobearm_t tcontext=system_u:object_r:file_t tclass=file
```

Listing 11 – Fin de l'écrire de Firefox dans le fichier temporaire

```
type=AVC msg=audit(begin=[.....25985], end=[.....87541]) avc : granted { write  } for pid=3884
com="%programfiles%\firefox\firefox.exe" ppid=1960 path="%systemroot%\temp\file.txt"
scontext=system_u:system_r:firefox_t tcontext=system_u:object_r:file_t tclass=file
```

Listing 12 – Fin de la lecture de Adobearm

```
type=AVC msg=audit(begin=[.......41343], end=[...98312]) avc : granted { read } for pid=1880
com="%programfiles%\adobe\adobearm.exe" ppid=160 path=""%systemroot%\temp\file.txt"
scontext=system_u:system_r:adobearm_t␣tcontext=system_u:object_r:file_t␣tclass=file
```

Les deux conditions citées par [5] de dépendances causales sont remplies : 1) `file_t` est un contexte de sécurité partagé par Firefox et Adobearm et 2) le début de la première interaction (.....25985) est avant la fin de la seconde (.....98312). Donc, PIGA-Monitor détecte une dépendance de cause entre ces deux interactions, correspondant à un flux venant de `firefox_t` > `file_t` > `adobearm_t`. Ensuite, PIGA-Monitor reconstruit l'ensemble des activités illégales du système. Lorsque la dernière interaction intervient (*i.e.* `opera_t` veut lire le contenu de `user_home_opera_t`), PIGA-Monitor demande le refus de cet appel système.

5.4. Performances

Les expériences ont été réalisées sur un système d'exploitation Windows 7 version 32 bits, sur une machine contenant un processeur Core i5 avec 2 Go de mémoire vive. Pour tester les performances de PIGA-Monitor, nous avons appliqué les deux propriétés présentées en listing 4 et 5. Nous avions pour base une politique de contrôle d'accès obligatoire suffisante pour faire fonctionner correctement Firefox et Opera. Celle-ci contient 950 contextes de sécurité et 1509 interactions.
Les performances du *driver* seul sont détaillées dans un précédent article [8]. Nous nous sommes donc intéressés aux performances de PIGA-Monitor. Après avoir appliqué les propriétés de sécurité sur la politique grâce à PIGA-Monitor, nous avons trouvé 111 interactions violant la propriété de confidentialité et 99 violant la propriété d'exécution de fichier.
Dans des conditions similaires, nous avons répété plusieurs fois les mêmes tests dans le but d'avoir une valeur moyenne pour la prise de décision. Il faut en moyenne 0,1 milliseconde pour prendre une décision. C'est le temps nécessaire à PIGA-Monitor pour faire une recherche dans son ensemble d'activités illégales. Nous pouvons donc dire que l'impact sur les performances est assez minime.

6. Conclusion

Dans cet article, nous décrivons notre solution pour protéger les systèmes d'exploitation Windows. Notre approche est basée sur SELinux, qui utilise la notion de types pour gérer les interactions sur le système. Cependant, notre approche ne traite pas les flux indirects. L'incorporation de PIGA-Monitor au sein du système d'exploitation est une solution possible pour palier à ce problème. Ainsi, grâce à un ensemble de propriétés de sécurité avancées, notre approche peut contrer un certain nombre d'attaques. De plus, les différents tests que nous avons réalisés montrent que les performances sont peu impactées.

Plusieurs optimisations sont possibles comme la réduction du nombre de règles dans la politique de contrôle d'accès obligatoire. Cette optimisation peut être obtenue en appliquant PIGA sur la politique pour trouver des règles inutiles. D'un point de vue plus technique, la portabilité pour les systèmes d'exploitation Windows 64 bits ne pourra être assurée qu'en créant un *filter-driver*(ou *driver* filtrant) car il n'est plus possible de détourner les fonctions de la SSDT.
Nos futurs travaux s'orienteront vers une meilleure intégration de PIGA-Monitor dans les systèmes d'exploitation Windows dans le but de calculer plus efficacement les violations de propriétés en temps réel. Nous allons aussi utiliser ce mécanisme pour caractériser les *malwares* et ainsi mettre en place de nouvelles propriétés de sécurité spécifiques à Windows. Enfin, toutes ces améliorations pourront être utilisées pour protéger les machines virtuelles, ainsi que les systèmes distribués comme le Cloud.

Bibliographie

1. D. E. Bell and L. J. La Padula. Secure computer systems : Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, Bedford, MA, May 1973.
2. K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation, June 1975.
3. M. Blanc, J. Briffaut, D. Gros, and C. Toinard. Piga-hips : Protection of a shared hpc cluster. *International Journal on Advances in Security*, 4(1) :44–53, 2011.
4. William E. Boebert and Robert Y. Kain. A practical alternative to hierarchical integrity policies. In *The 8th National Computer Security Conference*, Gaithersburg, MD, USA, October 1985.
5. Jeremy Briffaut. Formalization and guaranty of security properties for the operating systems : application to the detection of intrusions. *PhD Thesis of the University of Orleans*, 2007.
6. Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. *SIGOPS Oper. Syst. Rev.*, 39 :17–30, October 2005.
7. D. F. Ferraiolo and D. R. Kuhn. Role-based access controls. In *15th National Computer Security Conference*, pages 554–563, Baltimore, MD, USA, October 1992.
8. D. Gros and J. Briffaut C. Toinard. Contrôle d'accès mandataire pour windows 7. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications 2012*, Rennes, France, June 2012.
9. Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Commun. ACM*, 19(8) :461–471, 1976.
10. Greg Hoglund and Jamie Butler. *Rootkits : Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.
11. Core Labs. Core force user's guide. pages 1–2, October 2005.
12. Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *2001 USENIX Annual Technical Conference (FREENIX '01)*, Boston, Massachusets, United-States, 2001. USENIX Association.
13. John McLean. Reasoning about security models. In *IEEE Symposium on Security and Privacy*, pages 123–133, 1987.
14. Matt Miller. Modeling the trust boundaries created by securable objects. In *WOOT'08 : Proceedings of the 2nd conference on USENIX Workshop on offensive technologies*, pages 1–7, Berkeley, CA, USA, 2008. USENIX Association.
15. Prasad Naldurg, Stefan Schwoon, Sriram Rajamani, and John Lambert. Netra : : seeing through access control. In *FMSE '06 : Proceedings of the fourth ACM workshop on Formal methods in security*, pages 55–66, New York, NY, USA, 2006. ACM.
16. R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The flask security architecture : System support for diverse security policies. In *Proc. of the Eighth USENIX Security Symposium*, pages 123–139, August 1999.
17. Nickolai Zeldovich, Silas Boyd-wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 263–278. USENIX Association, 2006.