Contrôle d'accès mandataire pour Windows 7

Damien Gros[1], Jérémy Briffaut[2], and Christian Toinard[2]
damien.gros(@)cea.fr
jeremy.briffaut(@)ensi-bourges.fr
christian.toinard(@)ensi-bourges.fr

[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

Résumé Cet article présente une implémentation novatrice pour le renforcement des fonctions de contrôle d'accès de Windows 7. Basée sur le *Type Enforcement*, cette approche propose de mettre en place un contrôle d'accès de type mandataire supportant l'application de propriétés de sécurité avancées. De façon similaire à *LSM* pour le noyau Linux, notre implémentation offre un contrôle d'accès à grain fin reposant sur le détournement des appels système. L'ajout d'une labellisation du système de fichiers offre une correspondance précise entre la ressource et son contexte de sécurité. De plus, dans le but de faciliter l'écriture d'une politique de sécurité, nous avons mis en place un mécanisme d'apprentissage qui se base sur les événements observés par notre solution pour les transformer en règles de contrôle d'accès. Ainsi, cette implémentation assure le contrôle des flux d'information directs, que ce soit entre un sujet et un objet ou entre deux sujets, offrant ainsi protection ou confinement contre les attaques de type *0-day*. Cet article propose également une preuve de concept développée pour Windows 7, portable sur les autres systèmes d'exploitation de la famille Windows. Nous avons pu tester l'impact de notre implémentation sur les performances du système.

Résumé This paper presents a novel solution of Mandatory Access Control (MAC) providing *Type Enforcement* for the Windows 7 kernel. Our approach enforces a fine-grained MAC policy to control the system call permissions. In contrast with the Windows Mandatory Integrity Control, access control is supported. Labeling of the system resources provides security contexts with consistent types. An automated policy generation process facilitates the definition of the required rules for the various applications. The various system calls are controlled through an extensible kernel hooking framework. Thus, an efficient method is available for controlling direct information flows and preventing against 0-day vulnerabilities. Despite the purpose is to provide a proof of concept, the performance of the proposed solution is encouraging us to follow this approach. The current overhead can be further reduced using a compiled policy method. Moreover, that solution implements a control mechanism that can guarantee advanced security properties associated with transitive information flows.

1 Introduction

La sécurité des systèmes d'exploitation repose en général sur le modèle de contrôle d'accès dit discrétionnaire. Celui-ci délègue à l'utilisateur final l'attribution des droits sur les fichiers qu'il possède. Toutefois ce modèle a prouvé son manque de robustesse et n'autorise pas la définition de propriétés de sécurité [9]. Le contrôle d'accès mandataire offre la possibilité de définir des propriétés de sécurité avancées. Pour le noyau Linux, des mécanismes existent tels que grsecurity [1] ou SELinux [12]. Ce dernier est basé sur le modèle nommé *Type Enforcement* [3]. Ainsi, il est possible de protéger le système contre les attaques de type *0-days* cherchant à compromettre le système. Pour bloquer les attaques dites complexes ou mettant en œuvre les flux d'information indirects, l'utilisation de solutions telles que PIGA-IPS [7] doit être envisagée. Elles effectuent le pré-calcul des possibles violations de politique mais offrent surtout la possibilité de définir des propriétés de sécurité. Cependant ces solutions existent exclusivement pour les systèmes à base Linux mais pas encore pour la famille Windows (en particulier la version 7).

Cet article propose une implémentation efficace du modèle *Type Enforcement* pour Windows 7. Tout d'abord, nous présentons une solution pour labelliser dynamiquement le système de fichiers. Ensuite, un procédé d'automatisation du calcul de la politique MAC est introduit pour simplifier le travail d'écriture de celle-ci. Puis nous détaillons le mécanisme d'interception des appels système, pensé de façon générique pour la portabilité vers d'autres systèmes de la famille Windows. Enfin, une implémentation réelle est détaillée, et deux types de résultats sont présentés. Le premier porte sur la partie protection système et montre comment des attaques peuvent être confinées grâce à notre solution. Le deuxième porte sur l'impact de notre solution sur les performances du système. Nous proposons des solutions possibles dans le but de réduire l'impact observé. En conclusion nous montrons que l'un des atouts de notre prototype est la possibilité de le coupler avec PIGA-HIPS [6] pour garantir des propriétés de sécurité avancées mettant en jeu des flux d'information indirects.

2 État de l'art

Comme le montre [9], il n'est pas possible de définir des propriétés de sécurité grâce au modèle de protection discrétionnaire. En effet, de part sa conception, la politique sur laquelle ce modèle repose, n'est pas consistante. Cela signifie qu'à partir d'un état sûr de la machine, il est

difficile de démontrer qu'il n'existe pas au moins un chemin pour atteindre un état non sûr de la machine.

Pour contrôler les actions faites sur le système d'exploitation, le noyau se doit d'intercepter les flux d'information entre les processus et les ressources. Par exemple, lorsqu'un processus, que l'on appelle sujet, lit un fichier, appelé objet, il y a création d'un flux d'information direct allant de l'objet au sujet. Un mécanisme de contrôle d'accès mandataire ne peut contrôler que les flux d'information directs.

De plus, on trouve de nombreux travaux sur le contrôle d'accès mandataire dans la littérature. La majorité d'entre eux décrivent des modèles de protection théoriques comme par exemple Bell et La Padula [4] et Biba [5]. Ces modèles garantissent soit la confidentialité soit l'intégrité du système mais sont complexes à mettre en œuvre dans des systèmes d'exploitation modernes. Cependant certains travaux traitent plus particulièrement de l'intégration de contrôle d'accès mandataire au sein du noyau. Pour Linux, le plus mature est SELinux [12] initié par la NSA et actuellement maintenu par Red Hat. Une architecture spécifique nommée *Flask* [15] a été développée pour SELinux et repose sur la séparation de la prise de décision et d'application de la politique de sécurité. Pour appliquer cette politique, *Flask* représente les ressources du système sous la forme de **contextes de sécurité**. C'est un mécanisme fournissant une couche d'abstraction pour la politique de sécurité vis à vis du mécanisme de sécurité sous-jacent. Un contexte de sécurité, aussi appelé label de sécurité, est composé sous SELinux de trois éléments principaux. Le premier est l'identité qui est liée à l'UID standard de Linux mais qui diffère puisqu'elle est spécifique à SELinux. Un rôle, lié au modèle Role-Based Access Control [8], offre à l'utilisateur l'accès à différents types. Enfin, le type est la partie *Type Enforcement* de SELinux et c'est sur celui-ci que va se baser la politique de sécurité. Cette solution offre la possibilité d'avoir une politique plus facile à écrire puisque basé sur une notion de rôle, fournit le principe de *Type Enforcement* qui permet de 1) confiner les applications et 2) réduire leurs privilèges. Ainsi, 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 entièrement pour chaque programme les actions autorisées, il est parfois très long et coûteux de devoir écrire une politique fonctionnelle pour tout un système.

Dans le contexte des systèmes d'exploitation Windows, quelques travaux traitent de l'implémentation du MAC. Le premier [11], développé

pour Windows XP, fournit une approche multi-niveau pour l'implémentation d'un MAC. Depuis l'arrivée de Windows Vista et Windows 7, qui reposent tous les deux sur le noyau NT 6.x, Microsoft a mis en place un système similaire basé sur des niveaux d'intégrité. Connu sous le nom de Mandatory Integrity Control, il offre la possibilité de définir le niveau d'intégrité des ressources présentes sur le système. Ce contrôle d'intégrité n'est pas une application fidèle du modèle théorique de Biba [5], mais plutôt une simplification qui ne protège que les ressources système. En particulier, il n'est pas possible d'écrire une politique de contrôle d'accès qui repose sur la définition d'un sujet (processus) à qui on associe une action (appel système) sur un objet (ressource). Il faut utiliser un mécanisme de *broker* pour démarrer des processus avec un niveau d'intégrité différent de celui du processus parent. Il est ainsi possible de lancer, par exemple, *Internet Explorer* avec un niveau d'intégrité bas, puis de télécharger un fichier en niveau moyen.

Cependant, il reste possible d'effectuer des actions demandant un accès administrateur. Donc, par une augmentation légitime de ses privilèges, un utilisateur peut atteindre le niveau nécessaire pour administrer la machine. Dans ce cas là, il n'est plus soumis aux différents contrôles d'accès, qu'il soit discrétionnaire ou d'intégrité. Ce défaut de sécurité n'est pas présent que sous Windows puisque la commande *sudo* autorise une élévation de privilège similaire sous Linux. Toutefois, par rapport à SELinux, le MIC de Windows ne contrôle pas les appels système fait par les processus privilégiés et ayant pour objectif de violer une propriété de sécurité. C'est dans ce but que l'implémentation du *Type Enforcement*, capable de contrôler tous les processus du système, doit être développée pour Windows.

Enfin, la plupart des articles sur la sécurité du contrôle d'accès des systèmes Windows ne traitent que de vérification de politique discrétionnaire [14]. Un premier papier traitant du contrôle d'accès sous Windows 2000 et proposant une granularité fine pour le contrôle d'accès a été fait par Microsoft [16]. Mais cet article ne fait qu'une extension du modèle discrétionnaire. [13] étudie l'augmentation de privilèges dans les fonctions RPC de Windows. Cependant il n'existe aucune implémentation d'un contrôle d'accès mandataire sous Windows similaire à ce qui existe sous Linux. Premièrement, Windows ne supporte pas actuellement l'approche MAC. Deuxièmement, il n'y a pas de composant de type *Linux Security Module* présent dans les appels système offrant la possibilité de contrôler les actions. Il existe néanmoins des approches non officielles [10]

pour détourner les appels système et par conséquent ajouter des contrôles supplémentaires au sein du noyau Windows.

3 Motivations

Le but de cet article est de fournir un mécanisme de *Type Enforcement* pour Windows 7. Comme dit dans l'état de l'art, peu de travaux existent traitant du contrôle des processus, même privilégiés. Plusieurs difficultés existent quant à l'intégration des composants de contrôle d'accès mandataire dans le noyau de Windows 7. Tout d'abord, il faut fournir une solution générique capable de détourner les appels système. Comme notre objectif est de fournir une preuve de concept, nous avons fait le choix de rester sur les systèmes 32 bits. Ensuite, dans le but de garantir que la politique MAC soit correctement appliquée, il est nécessaire de concevoir un moniteur de référence [2] pour le noyau de Windows 7. Troisièmement, le principe de *Type Enforcement* doit être appliqué au système entier. Enfin, des mécanismes d'automatisation doivent être proposés pour aider à l'écriture de la politique MAC.

Les réponses à ces différents points sont données dans cet article. D'abord, nous décrirons un mécanisme générique capable de détourner les appels système. Ensuite, une description de notre moniteur de référence implémenté sous la forme d'un driver noyau sera faite. Il sera chargé de donner l'autorisation ou le refus pour une requête vis à vis de la politique MAC. Une méthode générique est proposée pour labelliser dynamiquement toutes les ressources du système d'exploitation. Enfin, un module d'apprentissage a été développé pour l'écriture d'une politique MAC prenant en compte toutes les applications du système.

Même si notre approche est inspirée de SELinux, par exemple pour le format des contextes de sécurité ou les règles de la politique de contrôle d'accès, des différences majeures existent. En effet, le détournement des appels système sur un système Windows est plus difficile que sur Linux puisqu'il n'existe pas de composants tels que les *LSM* qui implémentent nativement des *hooks* dans les appels système. Il nous faut donc proposer une méthode générique pour le détournement des appels système. De plus, il n'est pas possible d'intégrer le moniteur de référence directement dans le noyau de Windows, il doit donc être conçu en tant que driver. Enfin, l'implémentation du *Type Enforcement* sur un système Windows est plus complexe que sur Linux. En effet, il existe plusieurs façons de nommer une même ressource. Par exemple `C:32.exe` et `232.exe` sont deux noms distincts désignant la même ressource `cmd.exe`. Nous proposons donc une

labellisation dynamique pour les systèmes Windows dans le but de définir lors de l'exécution des noms factorisés associés à des contextes de sécurité pour tout le système. Enfin, un processus d'automatisation capable d'analyser les *logs* facilite la création de règles pour la politique. Ainsi il est possible de générer une politique complète par apprentissage.

Les principaux composants de notre solution sont les éléments suivants : un processus de labellisation générique, un driver chargé d'appliquer la politique MAC et un processus d'automatisation qui offre la possibilité de créer par apprentissage une politique MAC. Le processus de labellisation associe chaque ressource avec un contexte de sécurité précis. Le driver détourne les appels système et autorise ou non l'action au regard de la politique. L'automatisation du calcul de la politique fournit les règles d'accès pour chaque sujet avec un jeu de permissions sur les différents objets.

4 Labellisation dynamique

4.1 Objectif

La principe de la définition des contextes de sécurité sous Windows est d'associer chaque ressource (par exemple `C:32.exe`) avec un nom unique (`%systemroot%32.exe`) et un contexte de sécurité (`system_u:object_r:cmd_exec_t`). Le nom unique résout les problèmes de désignation des ressources. En effet, sous Windows, une ressource peut avoir plusieurs noms différents. Pour chaque ressource, le contexte de sécurité précise le type qui sera utilisé pour définir les règles de contrôle d'accès dans la politique.

Le problème est donc d'avoir une labellisation la plus précise possible :

- chaque ressource possède un nom unique ;
- chaque ressource a un type qui lui est propre.

Le processus de labellisation consiste à obtenir le nom Windows quel que soit son format puis de le transformer en nom canonique. Un nom canonique est un nom capable de factoriser les différents noms Windows d'une même ressource. Si on compare avec SELinux, notre processus de labellisation est dynamique puisqu'il n'y a pas d'élément de configuration décrivant les associations entre fichier et contexte (sous SELinux, cette configuration est stockée dans le fichier `file_context`). De plus, cette méthode rend le processus complètement indépendant vis à vis du système de fichiers cible.

Nous avons fait le choix de calculer dynamiquement les contextes de sécurité au sein de notre driver plutôt que de stocker ces éléments dans un flux NTFS car cela permet d'avoir une compatibilité complète, même avec

les systèmes de fichiers de type FAT32. Sous SELinux, les contextes de sécurité sont stockés dans les attributs étendus nommés *security.selinux*.

4.2 Noms canoniques

Comme expliqué précédemment en 4.1, une même ressource peut être désignée par des noms différents suivant le *namespace* : *Win32 File Namespace*, *NT Namespace* ou *Kernel Namespace*. Par exemple, `C:32.exe` est le nom provenant du *Win32 File Namespace*, `232.exe` du *Nt Namespace* et enfin `\??:32.exe` du *Kernel Namespace*. Ces trois noms sont associés avec une seule et même ressource. Ils doivent donc être factorisés en un seul et même nom canonique.

Dans le but de calculer un nom canonique, notre solution utilise certaines variables d'environnement. Par exemple, la variable `%systemroot%` désigne le nom canonique pour le répertoire `Windows`. Donc le nom canonique pour le fichier `C:32.exe` est `%systemroot%32.exe`.

L'approche précédente est généralisée pour toutes les variables d'environnement décrivant les principaux répertoires. Le listing 1.1 recense les variables actuellement utilisées pour calculer les noms canoniques pour tout le système. La variable `%systemdrive%` est pour la lettre d'installation du système (souvent `C:`) et la variable `%programfiles%` est pour le répertoire `Program Files`. Nous nous sommes contentés de n'utiliser que ces variables, mais il est tout à fait possible d'en utiliser d'autre. De plus, comme le processus est dynamique, il est possible d'en ajouter ainsi que de les modifier.

```
%systemroot%
%systemdrive%
%programfiles%
```

Listing 1.1. Variables d'environnement

4.3 Création des contextes de sécurité

Cette partie décrit la création des contextes de sécurité grâce au nom canonique. Nous considérerons deux cas distincts : le premier traitant les fichiers avec une extension spécifique et le deuxième traitant les répertoires. Il est important de distinguer les deux éléments puisque le contexte de sécurité est un moyen de caractériser précisément une ressource du système.

Premièrement, le nom canonique pour les fichiers est extrait de la façon suivante. Si un fichier possède une extension telle que `exe`, `dll` ou `sys`, alors le type du contexte de sécurité associé aura un nom spécifique.

Par exemple, le fichier `%systemroot%32.exe` qui a l'extension `exe` aura pour type spécifique associé un type finissant par `_exec_t`. Le type complet pour le fichier `cmd.exe` est `cmd_exec_t`. Actuellement, le contexte de sécurité (qui comprend une identité, un rôle et un type) utilise l'identité `system_u` et le rôle `object_r`. Par conséquent, nous obtenons le contexte de sécurité `system_u:object_r:cmd_exec_t` pour le fichier `%systemroot%32.exe`. La même approche est utilisée pour le calcul des types correspondant aux extensions `dll` et `sys`. Nous avons fait le choix de ne traiter que ces extensions dans un premier temps car ce sont les extensions des fichiers binaires exécutables et donc les fichiers potentiellement les plus dangereux pour le système. Cette approche peut être étendue aux extensions de scripts tels que `bat` pour prendre en compte tout type de fichiers exécutables.

Ensuite, les noms canoniques sont calculés différemment pour les répertoires. Le type associé au répertoire a pour fin `_dir_t`. Par rapport aux fichiers, un répertoire possède une chaîne de caractères plus complexe pour établir le type associé. Mis à part les variables d'environnement, le chemin complet du nom canonique permet de définir un type unique. Par exemple, le nom canonique de `%programfiles%` est associé au type `mozilla|plugins_dir_t`. Le contexte de sécurité complet est donc `system_u:object_r:mozilla|plugins_dir_t`.

4.4 Discussion

L'utilisation du caractère *pipe* "|", qui est un caractère interdit dans les noms Windows nous assure qu'il n'y aura pas d'interférence. Mais, si par une méthode d'évasion quelconque, un programme malveillant venait à détourner notre système de nommage des ressources, il devrait modifier tout le flux d'interaction pour qu'il puisse être exécuté. Il devrait de plus modifier directement la politique pour que le type qu'il aura soit autorisé à faire des actions sur le système.

5 Calcul de la politique MAC

Les contextes de sécurité sont à la base de la définition d'une politique de contrôle d'accès. 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 suivante 1.2 autorise le contexte sujet `system_u:system_r:explorer_t` à exécuter l'objet `system_u:system_r:cmd_exec_t`.

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

Listing 1.2. Règle d'accès

L'écriture d'une politique pour tout un système est un travail à la fois long et fastidieux. C'est pourquoi nous avons mis en place un procédé capable d'automatiser la création de la politique par un module d'apprentissage.

Comme le montre la figure 1, notre solution prend une seule entrée. Cette entrée est un message d'audit de contrôle d'accès généré par le driver. La sortie est donc un fichier contenant les règles d'autorisation d'accès telles que le montre le listing 1.3.

Figure 1. politique MAC

```
allow explorer_t cmd_exec_t:file  { read execute write create
    getattr }
```

Listing 1.3. Règle de polique MAC pour explorer

Cette approche est similaire au fonctionnement de la commande `audit2allow` fournie avec les utilitaires pour SELinux. Elle utilise les traces générées par SELinux pour compiler une liste de règles correspondant aux demandes d'accès effectuées. Si on compare avec SELinux, la principale difficulté pour Windows est d'obtenir un message d'audit avec suffisamment d'information. Nous générons ces traces avec le *driver* décrit dans le partie suivante. Un exemple est fournit dans le listing 1.4.

Pour le calcul de la politique, les deux choses importantes sont le `scontext` qui identifie le sujet de l'action et le `tcontext` qui identifie l'objet. Avec ces deux contextes de sécurité et les permissions demandées sur le type d'objet (`file: read execute write create getattr`), notre programme calcule la règle correspondante telle que montrée dans le listing 1.3. Comme notre solution est très proche du modèle utilisé par

SELinux, chaque permission est composée d'une classe (`file`) et d'un ensemble de droits (`read execute write create getattr`).

Les autres informations données par le driver sont aussi importantes dans un but de traçabilité des actions. On peut en effet vérifier que l'action est bien légitime vis à vis d'une politique que l'on aura précédemment mise en place. Nous montrerons en 7.3 comment on peut détecter facilement une violation de propriété.

```
audit(1285243100:4599)
avc:denied { read execute write create getattr } for pid=1576
comm="%systemroot%\explorer.exe" ppid=1528 path="%systemroot%\
    system32\cmd.exe"
scontext=system_u:system_r:explorer_t tcontext=system_u:system_r:
    cmd_exec_t
tclass=file
```

Listing 1.4. Journal d'accès

6 Driver

Le but de notre driver est premièrement de détourner les appels système, deuxièmement d'appliquer rigoureusement la politique de protection MAC et enfin de fournir des *logs* pour la création simplifiée de politique MAC et le suivi des actions qui se sont déroulées sur le système. Le schéma 2 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 que nous détaillerons un peu 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. 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.

6.1 Détournement des appels système

Lors du chargement du driver, des hooks sont placés sur un ensemble d'appels système dans le but de les contrôler. Comme présenté dans la figure 2, 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 réelle en mémoire. Cette partie décrit comment le

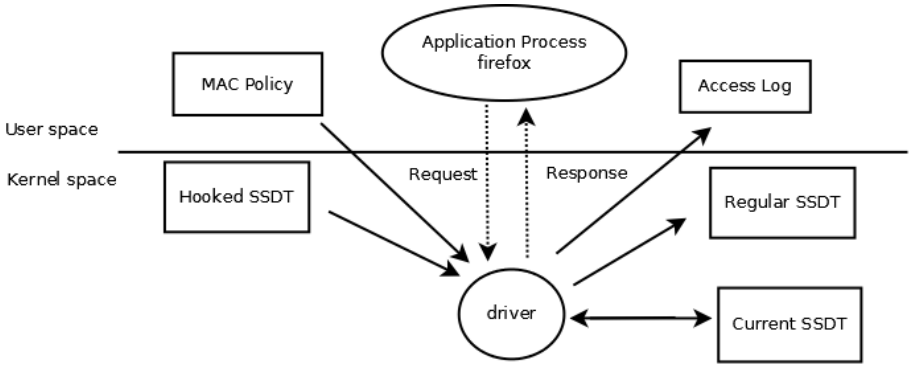

Figure 2. Driver

driver réalise ces deux opérations. Cette méthode est fortement inspirée du chapitre 4 de [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 permettant de 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 du driver. Puis, la SSDT modifiée contenant les différents hooks remplace la SSDT d'origine.

Comme nous sommes dans une optique de protection système et non d'audit, lorsque la fonction hookée est appelée, elle doit vérifier que la requête est valide. Une requête n'est valide que si et seulement si, il existe une entrée correspondante dans la politique de contrôle d'accès. 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 exécuter le vrai appel système pour que l'action puisse avoir lieu.

Le listing 1.5 décrit comment la SSDT est remplacée sur l'exemple de l'appel système `NtCreateFile`.

1. La première ligne définit comment obtenir une référence `g_pmdlSystemCall` sur la SSDT ; (ligne 1)
2. La seconde ligne permet de mapper la référence `g_pmdlSystemCall` dans un espace en mémoire non paginée ; (ligne 2)
3. La troisième ligne retourne un pointeur sur la SSDT ; (ligne 3)
4. La fonction `InterlockedExchange` remplace de manière atomique l'adresse courante de la fonction `currentSSDT[NtCreateFile_SystemCallNumber]` avec notre hook `hookedSSDT[NtCreateFile_SystemCallNumber]`. Cette

opération renvoie l'adresse originelle de l'appel système `regularSSDT[NtCreateFile_SystemCallNumber]`. (lignes 6-9)

```
1  g_pmdlSystemCall=IoAllocateMdl(KeServiceDescriptorTable.
       ServiceTableBase,
2          KeServiceDescriptorTable.NumberOfServices*4, 0, 0, NULL);
3  MmBuildMdlForNonPagedPool(g_pmdlSystemCall);
4  currentSSDT=MmMapLockedPages(g_pmdlSystemCall, KernelMode);
5  __try{
6      regularSSDT[NtCreateFile_SystemCallNumber] = (PVOID)
7      InterlockedExchange( (
8      PLONG) &currentSSDT[NtCreateFile_SystemCallNumber], (
9      LONG) hookedSSDT[NtCreateFile_SystemCallNumber]);
10 }
11 __except(1){
12     DbgPrint("IPS: Hook_Function : Hook failed");
13 }
```

Listing 1.5. Code détournant une fonction de la SSDT

Notre solution gère les principaux appels système, mais il est tout à fait possible de tous les détourner. Par exemple, notre driver hooke efficacement la création de nouveaux processus. En effet, l'appel système `NtCreateSection` est détourné par notre driver. Ce hook nous permet de gérer à la fois la création des nouveaux processus, mais aussi le chargement des drivers et des bibliothèques. Notre prototype gère aussi les opérations dites basiques comme les opérations sur les fichiers ou les opérations réseaux.

6.2 Application de la politique MAC

L'application de la politique de contrôle d'accès se fait par l'intermédiaire des hooks posés sur les appels système spécifiques. Nous allons par exemple décrire le processus de prise de décision lorsque la fonction `NtCreateFile` est appelée. Dans une première partie, nous décrirons les opérations réalisées par la fonction remplaçant la fonction hookée puis dans une seconde partie comment se déroule la prise de décision.

Hooking Un hook sur la SSDT se doit de respecter le prototype originel de la fonction hookée. Les prototypes des appels système courants sont définis dans les headers du Windows Driver Kit. Par exemple, le prototype de la fonction `NtCreateFile` est définit dans le fichier `ntifs.h`. La fonction `HookedNtCreateFile` a donc le même prototype que l'appel système originel. Le listing 1.6 donne le code pour la fonction `HookedNtCreateFile`.

1. Tout d'abord, la fonction `ExGetPreviousMode` vérifie si l'appel système provient de l'espace utilisateur ou de l'espace noyau. Seuls les appels système venant de l'espace utilisateur sont traités dans notre prototype et notre politique MAC ; (ligne 20)
2. La fonction `GetProcessInfo` récupère des informations importantes pour la prise de décision en lisant la structure `EPROCESS`. Les informations récupérées sont les suivantes : nom du processus, son chemin complet, son identifiant et l'identifiant de son père. Toutes ses informations permettent de décrire complètement le contexte sujet ; (lignes 25-26-27)
3. Le nom de l'objet est récupéré sous forme de nom Windows ; (ligne 44)
4. Ensuite, la classe de l'objet est récupérée : *file, directory, driver, device* ... . Cette classe est essentielle pour le calcul des permissions demandées ; (lignes 31-41)
5. Puis, les droits demandés sur l'objet sont récupérés. Par exemple : `read`, `write`, ... La classe ainsi que les droits fournissent les permissions demandées sur l'objet ; (lignes 44-57)
6. Enfin, la fonction qui hooke l'appel système appelle la fonction `Authorization` pour autoriser ou bloquer l'appel système ; (ligne 59)
7. Si la requête est autorisée, alors le vrai appel système est exécuté (ligne 64). Sinon, le programme retourne un statut d'erreur `STATUS_ACCESS_DENIED`. (ligne 79)

```
1
2  NTSTATUS HookedNtCreateFile(
3                       PHANDLE FileHandle,
4                       ACCESS_MASK DesiredAccess,
5                       POBJECT_ATTRIBUTES ObjectAttributes,
6                       PIO_STATUS_BLOCK IoStatusBlock,
7                       PLARGE_INTEGER AllocationSize OPTIONAL,
8                       ULONG FileAttributes,
9                       ULONG ShareAccess,
10                      ULONG CreateDisposition,
11                      ULONG CreateOptions,
12                      PVOID EaBuffer OPTIONAL,
13                      ULONG EaLength
14                      )
15
16 SUBJECT subject;
17 OBJECT object;
18 PERMISSION permission;
19
20 if(ExGetPreviousMode() == UserMode)
21 {
```

```
22
23      process = GetProcessInfo();
24
25      subject.pid = process.pid;
26      subject.ppid = process.ppid;
27      strncpy(subject.pathname, process.pathname, strlen(process.
            pathname));
28
29      type =  GetObjectType(ObjectAttributes->ObjectName);
30
31      switch(type)
32      {
33
34      case 1:
35          strncpy(permission.class, "file", strlen("file"));
36          break;
37      case 2:
38          strncpy(permission.class, "driver", strlen("driver"));
39          break;
40      [...]
41      }
42
43      RtlUnicodeStringToAnsiString(&ansi_string,
44              ObjectAttributes->ObjectName, TRUE);
45      strncpy(object.name,ansi_string.Buffer,
46              strlen(ansi_string.Buffer);
47
48      if( (DesiredAccess & FILE_READ_DATA) ||
49          (DesiredAccess &STANDARD_RIGHTS_READ) )
50      {
51          sprintf(permission.droit,"read ");
52      }
53      if( (DesiredAccess & FILE_EXECUTE) ||
54          (DesiredAccess & FILE_GENERIC_EXECUTE ) )
55      {
56          sprintf(permission.droit,"%sexecute ", permission.droit);
57      }
58
59      autorisation =  Authorization(subject, object, permission);
60
61      if(autorisation == 1)
62      {
63          retour = ((NTCREATEFILE)(
64          regularSSDT[NtCreateFile_SystemCallNumber])) (
65                                  FileHandle,
66                                  DesiredAccess,
67                                  ObjectAttributes,
68                                  IoStatusBlock,
69                                  AllocationSize,
70                                  FileAttributes,
71                                  ShareAccess,
72                                  CreateDisposition,
73                                  CreateOptions,
74                                  EaBuffer,
75                                  EaLength);
76          return retour;
77      }
```

```
78      else
79          return STATUS_ACCESS_DENIED;
80  }
```

Listing 1.6. HookedNtCreateFile

La fonction Authorization Nous allons détailler la fonction `Authorization` qui autorise ou refuse l'appel système dont le code est donné dans le listing 1.7. Cette fonction prend en paramètres : 1) le contexte sujet (une structure contenant le PID de l'appelant, le PPID et le nom Windows du processus), 2) l'objet (le nom Windows) et 3) les permissions (la classe de l'objet et les droits d'accès demandés) dans le but de prendre une décision.

1. Le nom du sujet est transformé en nom canonique ; (ligne 3)
2. Ce nom canonique est dérivé en contexte de sécurité du sujet ; (ligne 4)
3. De la même manière, le nom de l'objet est transformé en nom canonique et son contexte de sécurité est lui aussi calculé ; (lignes 5-6)
4. La fonction `AutorisationPol` cherche dans la politique une règle correspondant à ce contexte sujet, ce contexte objet et le jeu de permissions. La fonction recherche donc une règle satisfaisant cette requête. Si une règle est trouvée pour la requête alors la fonction renvoie *true*. Mais, comme notre approche est similaire à celle de SELinux, ce traitement n'est fait que sur les types. Enfin, si l'action n'est pas autorisée, la fonction va écrire un log d'accès permettant, *a posteriori*, soit d'ajouter une règle manquante en cas d'apprentissage, soit d'avoir une traçabilité des accès bloqués (tentatives d'attaque...) ;
5. Chaque refus est enregistré dans un fichier de log.

```
1   int Authorization( SUBJECT subject, OBJECT object, PERMISSION
        permission)
2   {
3       TransformToVariable(sujet.pathname, sujet_ok);
4       RetrieveSubjectContext(sujet_ok, context_sujet);
5       TransformToVariable(objet.name, objet_ok);
6       RetrieveObjectContext(objet_ok, context_objet);
7
8
9       decision = AutorisationPol(type_sujet, type_objet, permission.
            class,
10                  permission.droit);
11      if(decision == FALSE)
12      {
13          WriteInLog(subject, object, permission, scontext, tcontext,
                "deny");
```

```
14      }
15
16  return decision;
```

Listing 1.7. Authorization function

6.3 Journal d'accès

Comme expliqué précédemment, le journal d'accès est créé par la fonction gérant les autorisations. Il est le point de départ de la création de politique de contrôle d'accès par apprentissage. Le format est inspiré du format de SELinux. Le listing 1.8 montre le code de la fonction `WriteInLog`.

Nous allons décrire le format de notre fichier de log.

1. La première partie permet de récupérer le *timestamp* pour l'appel système ainsi que le numéro de la trace. Cela facilite la reconstruction des actions lors d'une analyse des traces ;
2. Ensuite, plusieurs informations nécessaires sur le sujet sont écrites :
   – Les droits demandés ;
   – Le sujet, défini grâce à son PID, son nom canonique et son PPID.
3. Le nom canonique de l'objet est aussi enregistré ;
4. Le contexte de sécurité de chaque élément ;
5. La classe de l'objet.

```
1  WriteInLog(SUBJECT, sujet, OBJECT objet, PERMISSION permission,
2          char * scontext, char * tcontext, char * decision)
3  {
4      trace++;
5      KeQuerySystemTime(&time);
6
7      TransformToVariable(sujet.pathname, sujet_ok);
8      TransformToVariable(objet.name, objet_ok);
9
10     sprintf(buffer, "audit(%I64d:%i) avc:%s { %s} for pid=%i
11     comm=\"%s\" ppid=%i path=\"%s\" scontext=%s tcontext=%s
12     tclass=%s\n", time.QuadPart,trace,decision, permission.droit,
13     subject.pid,sujet_ok , subject.ppid ,objet_ok, scontext,
14     tcontext, permission.class);
15     ntStatus = ZwWriteFile(handlelog,
16                  NULL,
17                  NULL,
18                  NULL,
19                  &iostatus,
20                  buffer,
21                  strlen(buffer),
22                  0,
23                  NULL
24                  );
```

```
25 }
```

Listing 1.8. FonctionWriteInLog

6.4 Appels système

Dans cette partie, nous allons lister les appels système que nous avons détournés. La plupart des appels système ont un nom assez explicite pour ne pas nécessiter d'expliciter leur utilité. Il est à noter que nous pouvons détourner tous les appels système au besoin. Nous avons considéré que pour une première approche, le détournement des appels système listés ci-dessous était suffisant.

Fichiers et dossiers Pour contrôler les actions sur les dossiers ou sur les fichiers, nous avons détourné les appels système suivants :

- `NtCreateFile`
- `NtOpenFile`
- `NtWriteFile`
- `NtReadFile`
- `NtDeleteFile`

Ce sont les actions basiques qu'il est possible de faire sur un système de fichiers : création, ouverture, écriture, lecture et suppression de fichiers.

Processus Le contrôle sur les processus, comme la création, le chargement d'une bibliothèque ou la récupération de `handle` sur autre processus, se fait grâce aux fonctions suivantes :

- `NtOpenProcess`
- `NtTerminateProcess`
- `NtCreateSection`

De plus, le contrôle de `NtOpenProcess` et `NtTerminateProcess` permet un premier pas vers la protection d'un processus utilisateur spécifique.

Le registre Pour contrôler le registre, nous avons employé une technique préconisée par Microsoft qui repose sur l'utilisation de `Kernel Callback`. Le principe est assez simple : lors du chargement du driver, celui-ci s'enregistre auprès du gestionnaire d'entrée/sortie en lui précisant les actions qu'il veut contrôler. On peut ainsi faire des traitements pré et post actions.

Le réseau Pour la surveillance du réseau, nous avons détourné l'appel système `NtDeviceIoControl` en filtrant sur le périphérique `AFS`, une couche d'abstraction de Windows chargée de gérer différents protocoles de communication réseau. Nous nous sommes concentrés sur le filtrage de TCP/UDP.

De plus, comme nous détournons `NtWriteFile` ainsi que `NtReadFile`, il est possible de lire le contenu des paquets tant qu'ils ne sont pas chiffrés.

6.5 Limites

Plusieurs choix ont été fait quant à l'implémentation de notre moniteur de référence sous forme de driver. Le premier fût le choix de ne pas stocker les contextes de sécurité dans un flux NTFS dans le but d'assurer une compatibilité maximale avec les systèmes Windows précédent Windows Vista.

Systèmes 64 bits Comme nous nous sommes concentrés sur les systèmes Windows 32 bits, nous avons pu mettre en place le détournement des appels système grâce à des hooks posés sur la SSDT. Nous obtenons ainsi le contrôle de tout le fonctionnement du système, que ce soit pour les fichiers et les dossiers, mais aussi pour la gestion des processus et du registre. Cependant cette méthode n'est plus applicable sur les systèmes Windows 64 bits. En effet, Microsoft a mis en place une protection au niveau du noyau qui empêche toute modification de la SSDT. Cette protection se nomme *Kernel Patch Protection* ou *Patch Guard*. En plus de la surveillance des tables système, il est chargé de vérifier les signatures numériques des drivers qui se chargent. Ceux-ci doivent être cross-signés par Microsoft pour être autorisés à se charger. Donc si nous voulons porter notre contrôle d'accès sur les systèmes Windows 64 bits, il nous faut repenser le détournement des appels système.

Une solution est d'implémenter un driver en couche, qui se placera dans la pile des drivers. Ainsi, nous pourrons contrôler tout ce qui transite en direction du système de fichiers ou qui en provient. Cette méthode est d'ailleurs celle préconisée par Microsoft pour ce qui est de la surveillance du système.

Pour le contrôle des processus, comme pour le chargement ou le déchargement de drivers, il nous faudra utiliser un *Kernel Callback*. C'est ce que nous faisons pour la gestion du registre. Pour ce qui est du réseau, il sera nécessaire de développer une driver spécifique, se greffant soit sur `AFD` soit sur `NDIS`.

Autres logiciels modifiant la SSDT Une limite de notre implémentation sous forme de hooks est la gestion des logiciels de sécurité se servant aussi de la SSDT pour contrôler l'activité du système. Il s'agit par exemple des antivirus ou des pare-feu pour Windows. Nous avons deux possibilités pour prendre en compte leurs hooks :

- La première est d'interdire toute modification de la SSDT quel que soit le logiciel faisant l'opération car notre contrôle d'accès mandataire risquerait d'être désactivé. En effet, si nous autorisons un logiciel à faire une modification sur la SSDT au même endroit que nous, et si ce logiciel n'utilise pas la précédente adresse présente dans la SSDT mais une adresse directement extraite du noyau, notre driver ne pourrait plus interdire les actions illégitimes vis à vis de la politique que nous aurions mis en place.
- La seconde solution est d'autoriser les logiciels de sécurité à poser leur hooks sur la SSDT, mais que notre driver soit le dernier à modifier la SSDT. Ainsi, il serait le premier à être consulté pour savoir si l'action est légitime ou non et comme nous utilisons l'adresse présente dans la SSDT pour exécuter l'appel, si cette adresse est celle d'un logiciel de sécurité, il pourra lui aussi prendre une décision pour l'action courante.

7 Expérimentations

Cette section décrit les expérimentations que nous avons fait avec notre prototype. Dans une première partie, nos détaillerons combien de contextes de sécurité notre programme doit gérer, puis nous expliquerons sur quels programmes nous avons choisi de faire nos expérimentations, enfin nous appliquerons la politique écrite à un cas réel.

7.1 File context

Comme détaillé en section 4.3, les contextes de sécurité sont calculés dynamiquement par notre driver lorsqu'une action provenant de l'espace utilisateur est détournée. Ces contextes sont utilisés pour déterminer si une action est valide ou non vis à vis de la politique de sécurité mis en place par l'administrateur. En plus des contextes de sécurité, il est nécessaire de récupérer les droits d'accès demandés par le processus appelant.

Pour cet article, nous avons voulu déterminer combien de contextes de sécurité notre driver doit calculer. Nous avons donc développé un programme générant un équivalent au `file_context` de SELinux pour un

système d'exploitation de type Windows 7, en version 32 bits. Nous avons donc un système avec 39243 contextes de sécurité pour les objets et 1772 contextes de sécurité pour les sujets.

Pour une extension future de notre prototype, notamment l'interface avec PIGA-HIPS dans le but de bloquer aussi des attaques complexes, il est nécessaire de générer ce fichier contenant les contextes de sécurité pour mettre en place les propriétés de sécurité avancés voulues.

7.2 Calcul de la politique

Notre solution fournit un mode d'apprentissage pour faciliter l'écriture de la politique de contrôle d'accès. Dans ce mode, notre driver enregistre donc toutes les actions refusées, sans pour autant les bloquer, afin de générer un log d'accès suffisamment complet pour écrire une politique MAC. Lors de l'utilisation de ce mode, il est évident qu'un environnement sûr doit être utilisé pour ne pas enregistrer d'actions illégales. Il faut générer à la main des scénarios légitimes et, par conséquent, lancer des applications légitimes dans le but de les inclure dans la politique. Par exemple, il est nécessaire de lancer des applications telles que *Firefox* et *Windows Media Player* dans l'optique de générer les jeux de permissions assurant leur bon fonctionnement. Bien entendu, plusieurs exécutions sont nécessaires pour prendre en compte tous les contextes d'usage possibles de ces logiciels.

Une telle expérimentation nous a fourni une politique avec 1766 règles pour les applications telles que *Firefox* et *Windows Media Player*. Ces règles correspondent au lancement des applications sans aucun accès à internet. Naturellement, d'autres usages fournissent davantage de règles pour remplir la politique. Par exemple, pour accéder à deux sites web, *Firefox* demande l'ajout de 105 nouvelles règles pour faire correctement le rendu de ces sites.

Nous pouvons aussi comparer avec la politique SELinux pour le même logiciel. Par exemple pour *Firefox*, on compte environ 500 règles pour le contexte `mozilla_t`. Sous Windows, nous avons 350 règles sans pour autant avoir réalisé toutes les actions possibles et autorisées pour ce logiciel. Ce décompte n'inclus pas les règles pour le registre. Comme nous nous basons sur les chemins Windows pour labelliser les répertoires, la politique générée pour `Firefox` n'est pas réutilisable pour SELinux.

7.3 Protection

Une fois la politique complète générée, il est possible de fournir une protection en autorisant uniquement les actions légitimes et surtout en bloquant les actions non présentes dans la politique. Par exemple, le listing 1.10 montre une action autorisée par notre driver puisque la règle est explicitement dans la politique 1.9. L'appel système autorisé correspond à un accès réseau de la part de *Firefox*.

Le listing 1.11 montre que notre solution va effectivement bloquer une action qui n'est pas dans la politique. L'action refusée est une tentative de la part de *Firefox* de lire le contenu du répertoire d'*Opera*. Par conséquent, un flux illégal provenant d'*Opera* et allant vers *Firefox* est empêché. Ce flux correspond à une tentative de violation de confidentialité de *Firefox* sur le domaine d'*Opera*.

```
allow firefox_t socket_t:socket  { execute getattr gen_read
    gen_write  }
```

Listing 1.9. Politique

```
audit(129320376678356310:901) avc:granted { execute setattr gen_read
     gen_write }
for pid=1756 comm="%programfiles%\mozilla firefox\firefox.exe" ppid
    =1456
path="\Device\Afd\Endpoint" scontext=system_u:system_r:firefox_t
tcontext=system_u:object_r:socket_t tclass=socket
```

Listing 1.10. Activité légale de *Firefox*

```
audit(129320375967434054:683) avc:denied { execute read} for pid
    =1756
comm="%programfiles%\mozilla firefox\firefox.exe" ppid=1456
path="%systemdrive%\users\ensib\appdata\roaming\opera\opera"
scontext=system_u:system_r:firefox_t
tcontext=system_u:object_r:users|ensib|appdata|roaming|opera|
    opera_dir_t tclass=dir
```

Listing 1.11. Blocage d'un flux d'information vers *Firefox*

Une deuxième expérimentation a été réalisée sur l'exécution d'un *malware* réel. Nous l'avons laissé se dérouler entièrement pour montrer les différents endroits stratégiques où nous pouvons le bloquer. Le *malware* est un *FakeAV* dont l'analyse par VirusTotal se trouve ici [3].

3. https://www.virustotal.com/file/d0d0d53f66b400cb43b3019be9e5e49a9097-458119eba8f18d27d9e7b4ac8d9b/analysis/

Pour réaliser cette étude, nous partons du principe que le binaire a pu être écrit sur le Bureau de l'utilisateur. Un premier log d'accès nous montre que le *malware* a été lancé (listing 1.12). C'est le premier endroit où nous pouvons le bloquer. En effet, il faudrait que cette exécution soit autorisée dans la politique de contrôle d'accès.

```
type=AVC msg=audit(129774391393331470,214) avc : denied { execute }
    for pid=2344
com="%systemroot%\explorer.exe" ppid=2304
path="%systemdrive%\users\bob\desktop\8
    ce6d0d9f6906f3bf0233a3090226f11.exe"
scontext=system_u:system_r:explorer_t
tcontext=system_u:object_r:8CE6D0D9F6906F3BF0233A3090226F11_exec_t
    tclass=load
```

Listing 1.12. Exemple de *malware* n°1

Une fois exécuté par `explorer.exe`, le *malware* va charger des bibliothèques. Puis une fois complètement exécuté, il va écrire un fichier qui sera la charge active de l'infection (listing 1.13).

```
type=AVC msg=audit(129774392137591499,7148) avc : denied { write }
    for pid=3888
com="%systemdrive%\users\bob\desktop\8
    ce6d0d9f6906f3bf0233a3090226f11.exe"
ppid=2344 path="%systemdrive%\programdata\vwqgjwsurthvme.exe"
scontext=system_u:system_r:8CE6D0D9F6906_t tcontext=system_u:
    object_r:file_t
tclass=file
```

Listing 1.13. Exemple de *malware* n°2

Lorsque le *malware* a fini d'écrire le deuxième fichier, il va l'exécuter (listing 1.14).

```
type=AVC msg=audit(129774392138104454,7158) avc : denied { execute }
     for
pid=3888
com="%systemdrive%\users\bob\desktop\8
    ce6d0d9f6906f3bf0233a3090226f11.exe"
ppid=2344 path="%systemdrive%\programdata\vwqgjwsurthvme.exe"
scontext=system_u:system_r:8CE6D0D9F6906_t
tcontext=system_u:object_r:VwQGJwSURThVmE_exec_t tclass=load
```

Listing 1.14. Exemple de *malware* n°3

Comme nous venons de le voir, nous sommes capables de suivre l'évolution de l'infection sur le système. Il est donc possible de le bloquer à plusieurs endroits stratégiques dans son exécution pour l'empêcher de se propager correctement. Il est aussi possible de le bloquer lorsqu'il tente de faire un accès réseau dans le but de récupérer une charge virale.

7.4 Performances

Pour évaluer l'impact de notre système sur les performances, nous avons fait fonctionner une machine virtuelle Windows 7 32 bits avec 2.67 Go de RAM sur une machine hôte Windows 64 bits avec un processeur AMD Phenom^{TM} II x4 965 à 3,51 GHz avec 8 Go de RAM.

Une première série de résultats est disponible dans le tableau 1. Plusieurs applications ont été testées (*Firefox, Internet Explorer, Windows Media Player et Adobe Reader*). Le but de ces différents tests est de montrer l'impact de notre protection sur le démarrage des applications pour différentes utilisations. Le chargement de *Firefox* est impacté de 2 secondes pour un site dit léger pour calculer les permissions nécessaires, mais il y a le même impact pour un site plus complexe. Pour *Windows Media Player*, l'ouverture d'un document multimédia montre qu'une latence est ajoutée, variant entre 1 et 2 secondes. Mais l'impact global sur la lecture du fichier multimédia est très négligeable. Pour *Adobe Reader*, il est plus difficile de déterminer le temps de latence généré par notre protection puisque ce genre de logiciel dépend pour beaucoup de l'interaction avec l'utilisateur. On remarque tout de même qu'une latence est ajoutée au lancement de l'application et que cet impact ne semble pas être influencé par la taille du fichier ouvert.

Comme le montrent donc ces expérimentations, l'impact généré est essentiellement au lancement de l'application, puisque c'est à ce moment la qu'elle fait le plus de requêtes (création de fichiers temporaires, chargement de bibliothèques...) Ensuite, l'impact est négligeable.

Processus et Arguments	Sans IPS	Avec IPS
Firefox : google.fr	2 s	4 s
Firefox : jeuxvideo.com	9 s	11 s
Internet Explorer : google.fr	3 s	5 s
Internet Explorer : msn.com	4 s	6 s
Windows Media Player	2 s	4 s
Windows Media Player : mp3 file 4.22 Mo (222 s)	223 s	225 s
Windows Media Player : mp3 file 15.2 Mo	12 s	13 s
Windows Media Player : avi file 349 Mo	12 s	13 s
Adobe Reader 9.0	1 s	2 s
Adobe Reader 9.0 : file 832 Ko	1 s	2 s
Adobe Reader 9.0 : file 18.5 Mo	2 s	4 s
Adobe Reader 9.0 : file 294 Mo	2 s	4 s

Table 1. Tableau des performances

Même si le principal objectif de notre prototype n'est pas de minimiser le temps de latence, un usage normal du système avec notre protection met en évidence des performances tout à fait satisfaisantes. La plupart des applications accèdent à un grand nombre d'objets à leur lancement, il est donc normal qu'il y ait un impact du fait de l'intégration de notre solution. Mais les performances globales des applications ne sont que peu impactées.

Nous pouvons donner quelques détails sur cet impact ainsi que des pistes pour le réduire. Comme le driver doit faire une recherche dans la politique à chaque requête, il est assez évident que cela entraîne une latence. Pour réduire cet impact, il serait intéressant de développer différentes optimisations. La première serait la mise en place d'un système de cache, comme le fait SELinux avec son *Access Vector Cache*, qui permettrait d'avoir en mémoire les derniers *Access Vector* calculés par notre driver et ainsi éviter la recherche de la même requête deux fois de suite. Ensuite, la politique est pour l'instant stockée sous forme de chaînes de caractères en mémoire. Il est évident que ce système de stockage n'est pas optimisé pour la recherche rapide. Donc une optimisation possible serait la mise en place d'un système de compilation de la politique pour la stocker plutôt sous format binaire. De telles optimisations pourraient réduire la latence générée par notre driver.

Nous pouvons aussi fournir une comparaison avec SELinux, que ce soit en terme de performance mais aussi au niveau d'une politique pour un logiciel multi-plateforme. D'après la littérature, SELinux introduit un surplus d'environ 10% par rapport à une utilisation du logiciel sans contrôle d'accès mandataire.

8 Conclusion

Cet article décrit une implémentation possible du *Type Enforcement* ainsi que de la protection de type MAC associée pour un système de Windows 7. Même si notre approche est similaire au modèle utilisé par SELinux, des solutions dédiées aux systèmes Windows ont du être mises en place pour résoudre quelques problèmes spécifiques. Premièrement, une labellisation dynamique mais aussi générique a du être conçue. Ensuite, une méthode générique pour détourner les appels système a été développée. Enfin, un mécanisme d'automatisation du calcul de la politique de contrôle d'accès a été proposé dans le but de faciliter l'écriture de la politique pour un administrateur. Il est à noter que nous avons ainsi réalisé une implémentation réelle pour un système Windows 7 avec des résultats

très intéressants. Nous avons toutefois identifié plusieurs optimisations potentielles de notre programme comme la compilation de la politique de contrôle d'accès ou le développement d'un système de cache pour sauvegarder les décisions déjà prises.

Le développement de notre solution va se poursuivre avec pour objectif un prototype fonctionnant aussi sur les systèmes 64 bits. En effet, le détournement de la SSDT sur Windows 64 bits n'est plus possible. Nous avons donc commencé à développer un *filter driver* qui peut filtrer les requêtes à destination du système de fichiers dans cet environnement.

Pour aller plus loin, comme dans le cas de SELinux, notre prototype n'est capable que de contrôler les appels système et donc les flux d'information directs entre deux ressources du système (processus ou objets). En le connectant avec un moniteur de référence externe tel que PIGA-HIPS [6], il serait possible 1) de calculer les violations de politique de sécurité vis à vis d'une politique plus globale et 2) de contrôler les flux d'information indirects ou les attaques avancées. Il serait ainsi possible de définir des propriétés de sécurité avancées qui pourraient être garanties par notre prototype et PIGA-HIPS.

Remerciements

Nous tenons à remercier Mathieu Blanc pour sa relecture et ses remarques constructives.

Références

1. Grsecurity. http://grsecurity.net/.
2. J.P. Anderson. Computer security threat monitoring and surveillance. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.
3. Lee Badger, Daniel F. Sterne, David L. Sherman, and Kenneth M. Walker. A domain and type enforcement UNIX prototype. In *Proceedings of the 5th USENIX UNIX Security Symposium*, pages 127–140, Salt Lake City, Utah, USA, June 1995.
4. 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.
5. K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, The MITRE Corporation, June 1975.
6. 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.
7. J. Briffaut, J.-F. Lalande, C. Toinard, and M. Blanc. Enforcement of security properties for dynamic mac policies (best paper award). In *SECURWARE'09 : Proceedings of the Third International Conference on Emerging Security Information, Systems and Technologies*, Athens, Greece, 2009. IARIA.

8. 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.
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. 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.
14. 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.
15. 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.
16. Michael M. Swift, Anne Hopkins, Peter Brundrett, Cliff Van Dyke, Praerit Garg, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. Improving the granularity of access control for windows 2000. *ACM Trans. Inf. Syst. Secur.*, 5(4) :398–437, November 2002.